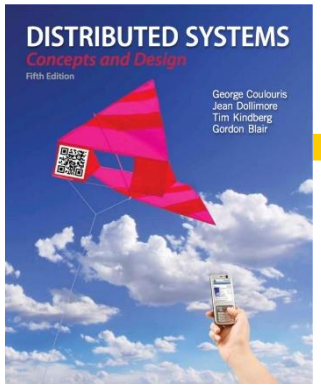
Distributed File Systems (DFS)



Most concepts are drawn from Chapter 12

Updated by Rajkumar Buyya

* **Introduction**
* **File service architecture**
* **Sun Network File System (NFS)**
*→ Andrew File System (personal study)
* **Recent advances**
* **Summary**

Learning objectives

- Understand the requirements that affect the design of distributed services
- NFS: understand how a relatively simple, widely-used service is designed
  - Obtain a knowledge of file systems, both local and networked
  - Caching as an essential design technique
  - Remote interfaces are not the same as APIs
  - Security requires special consideration
- Recent advances: appreciate the ongoing research that often leads to major advances (creation of a widely used storage infrastructures like DropBox).

Introduction

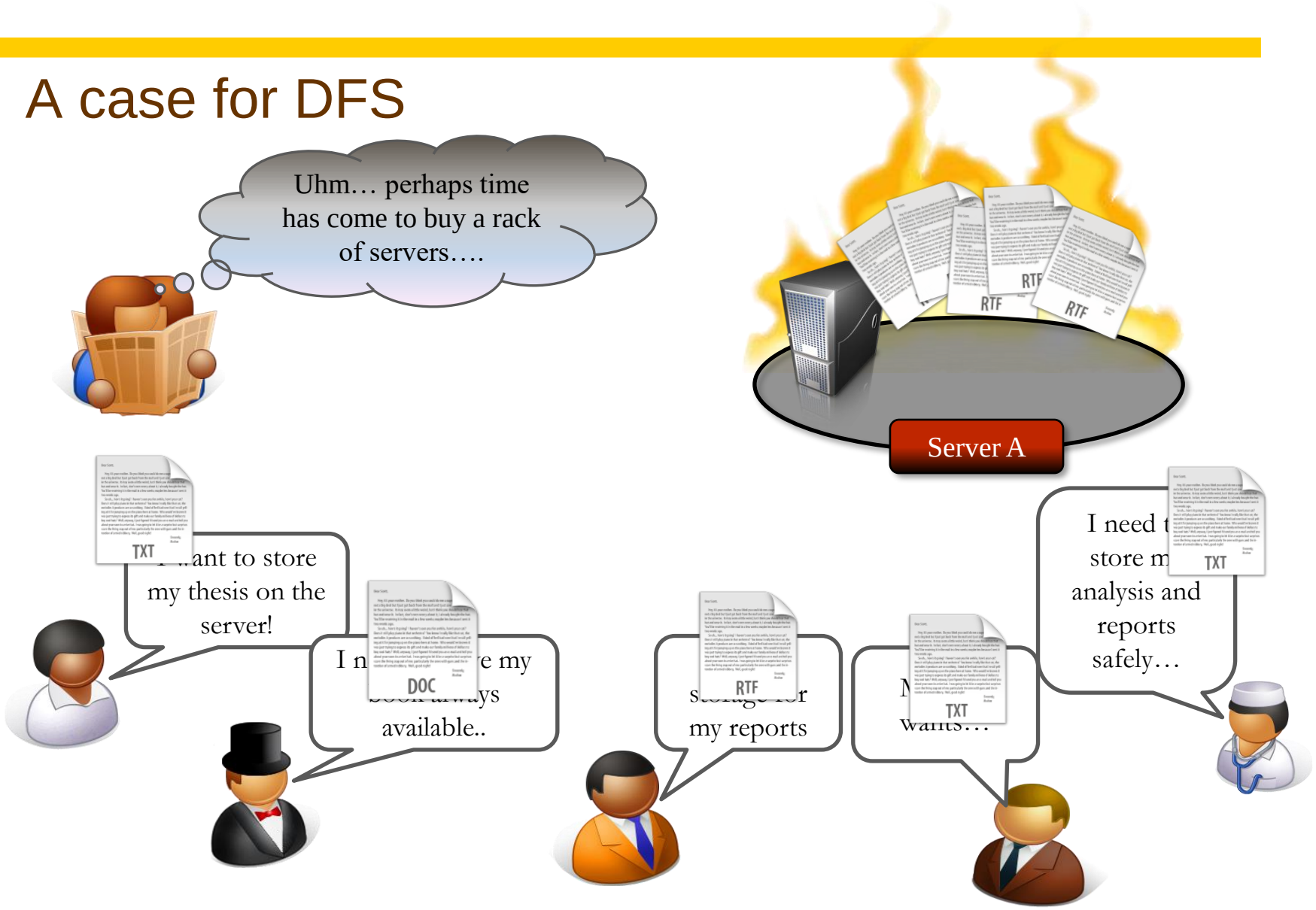
- Why do we need a DFS?
  - Primary purpose of a Distributed System…

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 and applications.*

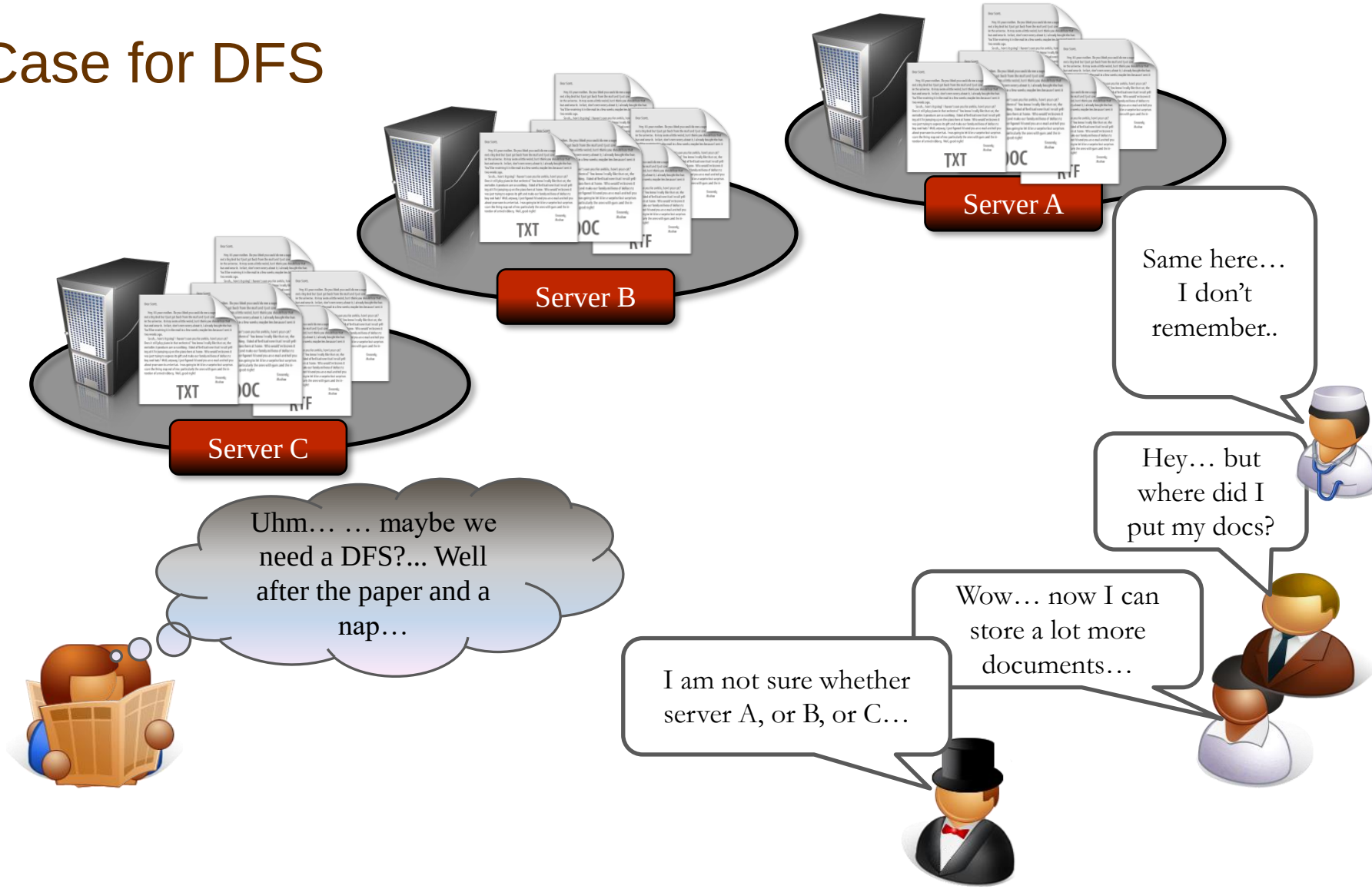
Introduction

- A case for DFS



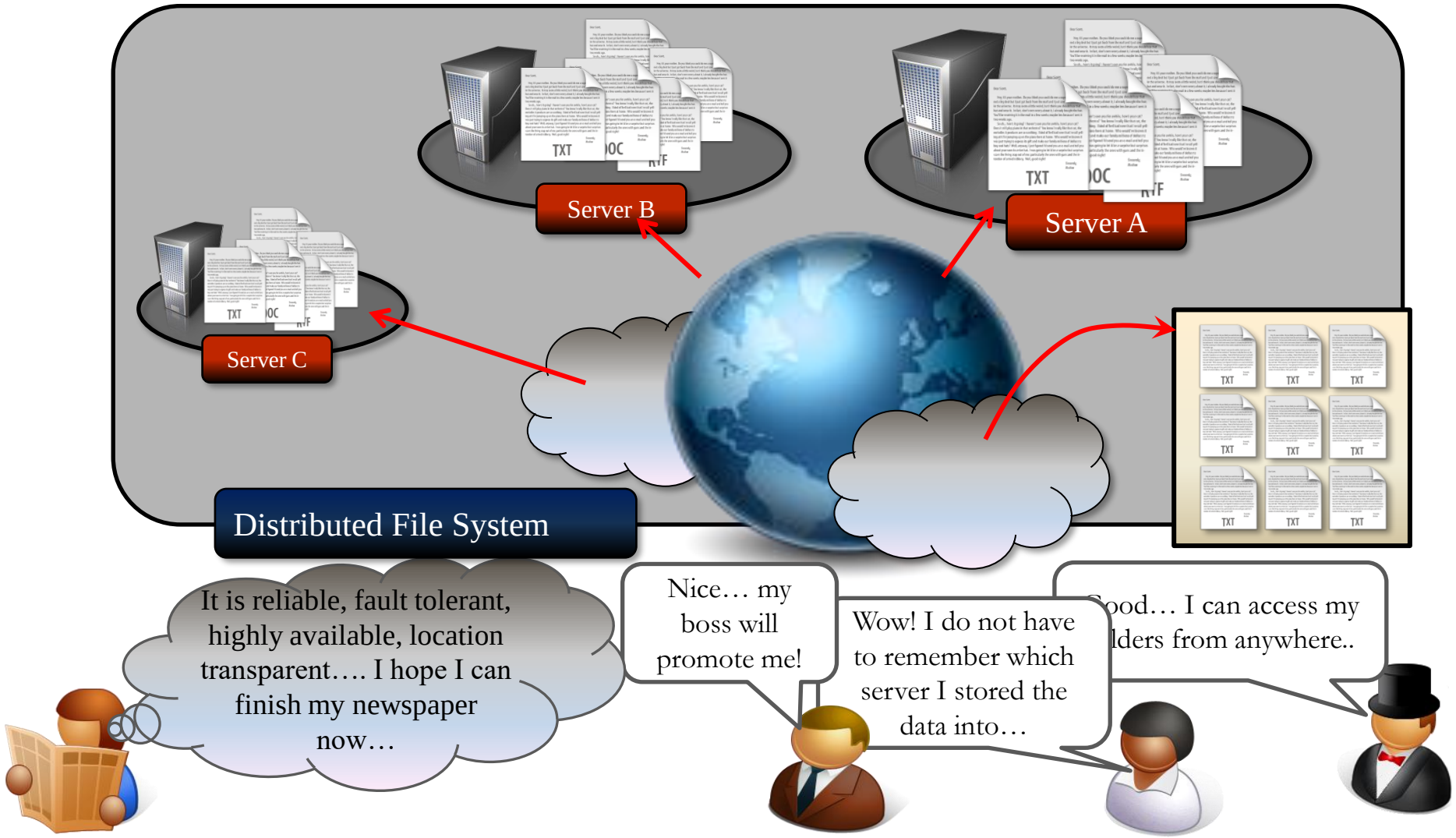
Introduction

- A Case for DFS



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 (GFS)
  - Amazon S3 (**Simple Storage Service)**
  - Cloud Storage (e.g., **DropBox, Google Drive, Microsoft OneDrive**)

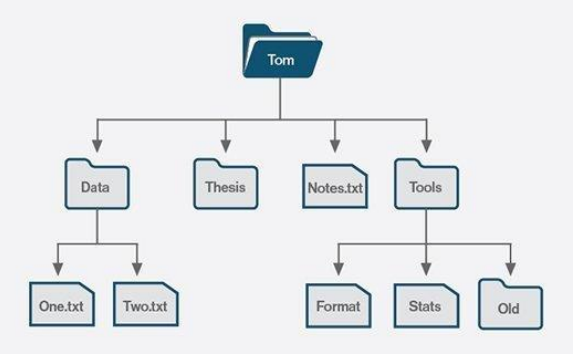
Storage systems and their properties

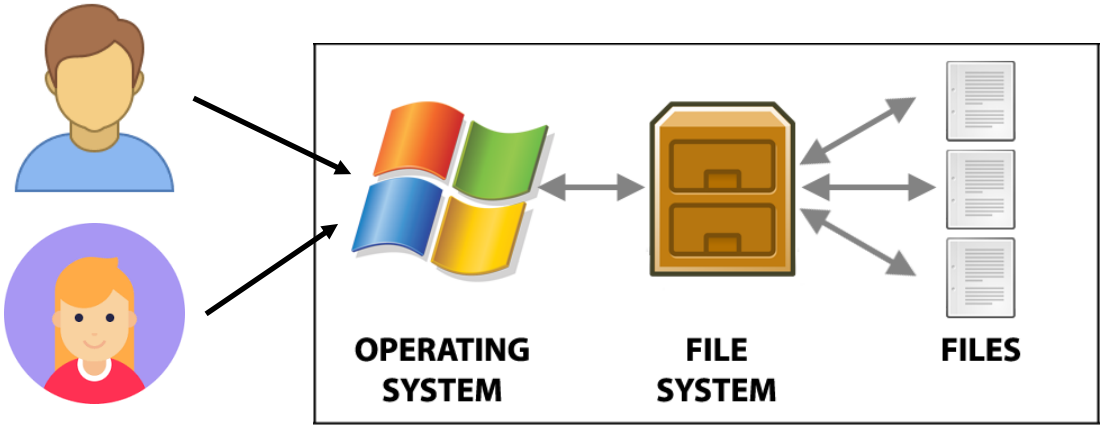
	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (Ch. 16)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage store	✓	✓	✓	2	OcceanStore

Types of consistency between copies: 1 - strict one-copy consistency
√ - approximate/slightly weaker guarantees
X - no automatic consistency
2 – considerably weaker guarantees

What is a file system?

- 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

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)	Gets the file attributes for file *name* into *buffer*.

What is a file system? 2

Class Exercise A

Write a simple C program to copy a file using the UNIX file system operations:

*copyfile(char * oldfile, * newfile)*
{
    <you write this part, using *open(), creat(), read(), write()*>
}

Note: remember that *read()* returns 0 when you attempt to read beyond the end of the file.

A code in C – Copy File program

Write a simple C program to copy a file using the UNIX file system operations.

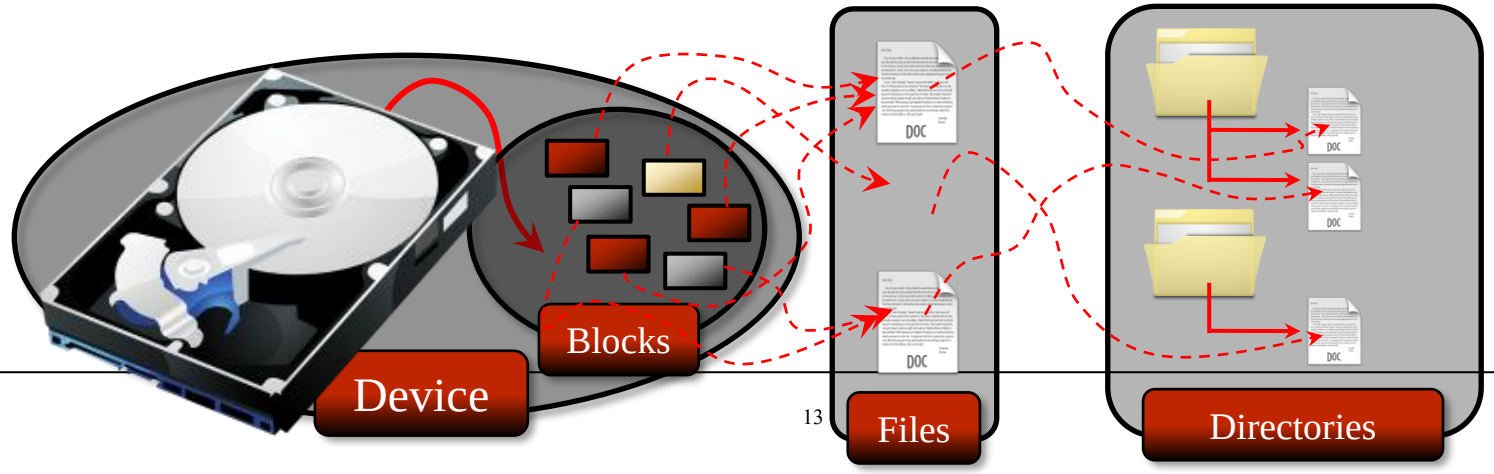
```
#define BUFSIZE 1024
#define READ 0
#define FILEMODE 0644
void copyfile(char* oldfile, char* newfile)
{       char buf[BUFSIZE]; int i,n=1, fdold, fdnew;

        if((fdold = open(oldfile, READ))>=0) {
                fdnew = creat(newfile, FILEMODE);
                while (n>0) {
                        n = read(fdold, buf, BUFSIZE);
                        if(write(fdnew, buf, n) < 0) break;
                }
        close(fdold);    close(fdnew);
        }
        else printf("Copyfile: couldn't open file: %s \n", oldfile);
}
main(int argc, char **argv)  {
        copyfile(argv[1], argv[2]);
}
```

What is a file system?
(a typical module structure for implementation of non-DFS)

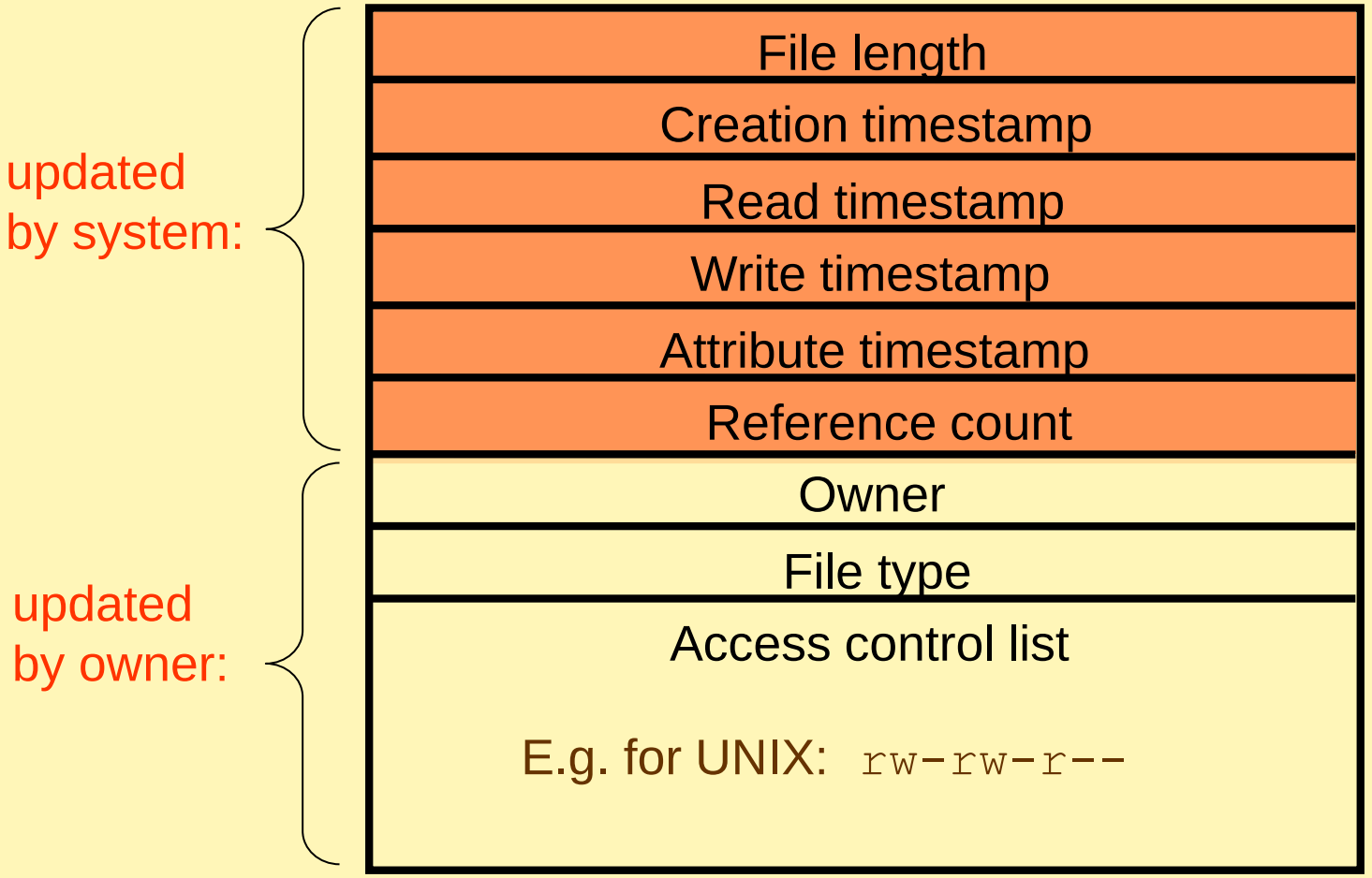
File system modules

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?

File attribute record structure



Distributed File system/service requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency..

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

Efficiency

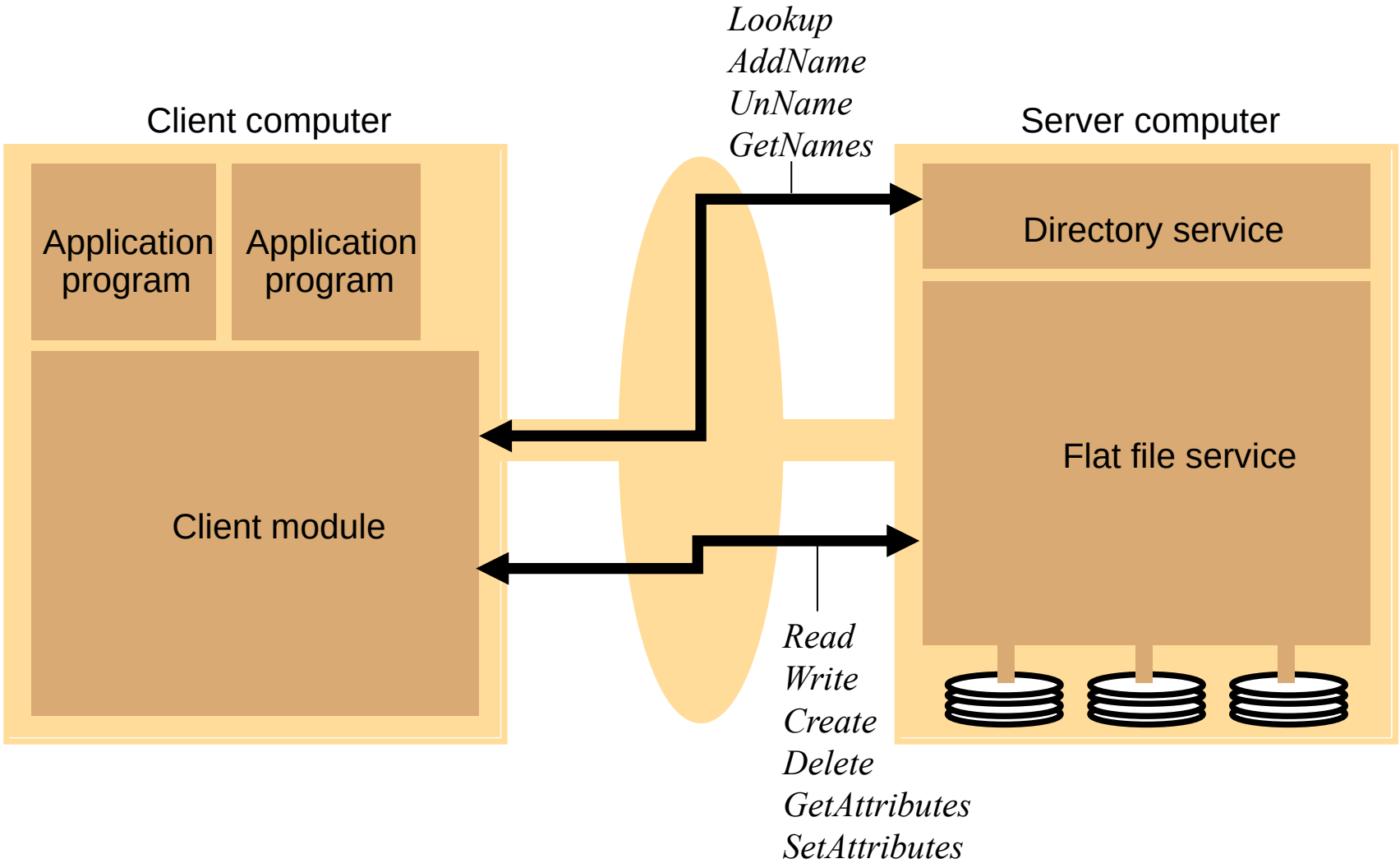
Goal for distributed file systems is usually performance comparable to local file system.

*

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 next).
- The Client module implements exported interfaces by flat file and directory services on server side.

Model file service architecture



Responsibilities of various modules

- 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.
- 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 to generate directories and to add new files to directories.
- 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 operations/interfaces for the model file service

Flat file service

Read(FileId, i, n) -> Data

position of first byte

Write(FileId, i, Data)

position of first byte

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Directory service

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, ~~File~~ FileId)

UnName(Dir, Name)

GetNames(Dir, Pattern) -> NameSeq

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.

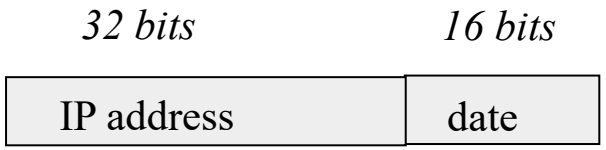
File Group

A collection of files that can be located on any server or moved between servers while maintaining the same names.

- Similar to a UNIX *filesystem*
- Helps with distributing the load of file serving between several servers.
- File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).
  - *Used to refer to file groups and files*

To construct a globally unique ID we use some unique attribute of the machine on which it is created, e.g. IP number, even though the file group may move subsequently.

File Group ID:



DFS: Case Studies

- NFS (Network File System)
  - Developed by Sun Microsystems (in 1985)
  - Most popular, open, and widely used.
  - NFS protocol standardised 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)

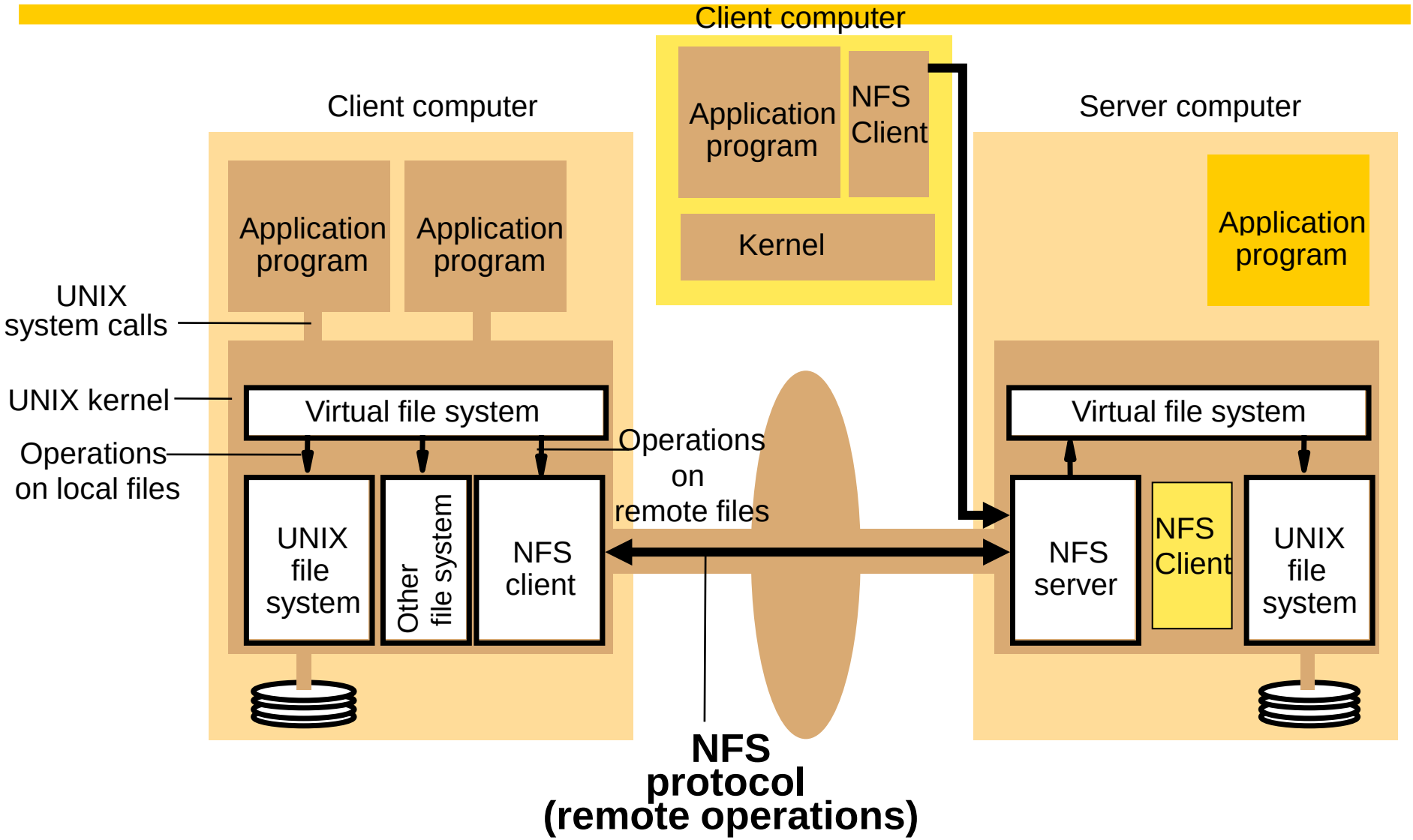
Case Study: Sun NFS

- An industry standard for file sharing on local networks since the 1980s
- An open standard with clear and simple interfaces
- Closely follows the abstract file service model defined above
- Supports many of the design requirements already mentioned:
  - transparency
  - heterogeneity
  - efficiency
  - fault tolerance
- Limited achievement of:
  - concurrency
  - replication
  - consistency
  - security

NFS - History

- 1985: Original Version (in-house use)
- 1989: NFSv2 (RFC 1094)
  - Operated entirely over UDP
  - Stateless protocol (the core)
  - Support for 2GB files
- 1995: NFSv3 (RFC 1813)
  - Support for 64 bit (> 2GB files)
  - Support for asynchronous writes
  - Support for TCP
  - Support for additional attributes
  - Other improvements
- 2000-2003: NFSv4 (RFC 3010, RFC 3530)
  - Collaboration with IETF
  - Sun hands over the development of NFS
- 2010: NFSv4.1
  - Adds Parallel NFS (pNFS) for parallel data access
- 2015
  - RFC 7530 – NFS Version 4 Protocol
  - Unlike earlier versions, it supports traditional file access while integrating support for file locking and the MOUNT protocol. It makes NFS operate well in an Internet environment.

NFS architecture



NFS architecture: does the implementation have to be in the system kernel?

No:

- there are examples of NFS clients and servers that run at application-level as libraries or processes (e.g. early Windows and MacOS implementations, current PocketPC, etc.)

But, for a Unix implementation there are advantages:

- Binary code compatible - no need to recompile applications
  - *Standard system calls that access remote files can be routed through the NFS client module by the kernel*
- Shared cache of recently-used blocks at client
- Kernel-level server can access i-nodes and file blocks directly
  - *but a privileged (root) application program could do almost the same.*
- Security of the encryption key used for authentication.

NFS server operations (simplified)

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> statu*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

fh = fi

File

eration

Model flat file service

Read(FileId, i, n) -> Data

Write(FileId, i, Data)

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Model directory service

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, File)

UnName(Dir, Name)

GetNames(Dir, Pattern) ->NameSeq

NFS access control and authentication

- Stateless server, so the user's identity and access rights must be checked by the server on each request.
  - In the local file system they are checked only on *open()*
- Every client request is accompanied by the userID and groupID
  - which are inserted by the RPC system
- Server is exposed to imposter attacks unless the userID and groupID are protected by encryption
- Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution

Architecture Components (UNIX / Linux)

- Server:
  - nfsd: NFS server daemon that services requests from clients.
  - mountd: NFS mount daemon that carries out the mount request passed on by nfsd.
  - rpcbind: RPC port mapper used to locate the nfsd daemon.
  - /etc/exports: configuration file that defines which portion of the file systems are exported through NFS and how.
- Client:
  - mount: standard file system mount command.
  - /etc/fstab: file system table file.
  - nfsiod: (optional) local asynchronous NFS I/O server.

Mount service

- Mount operation:

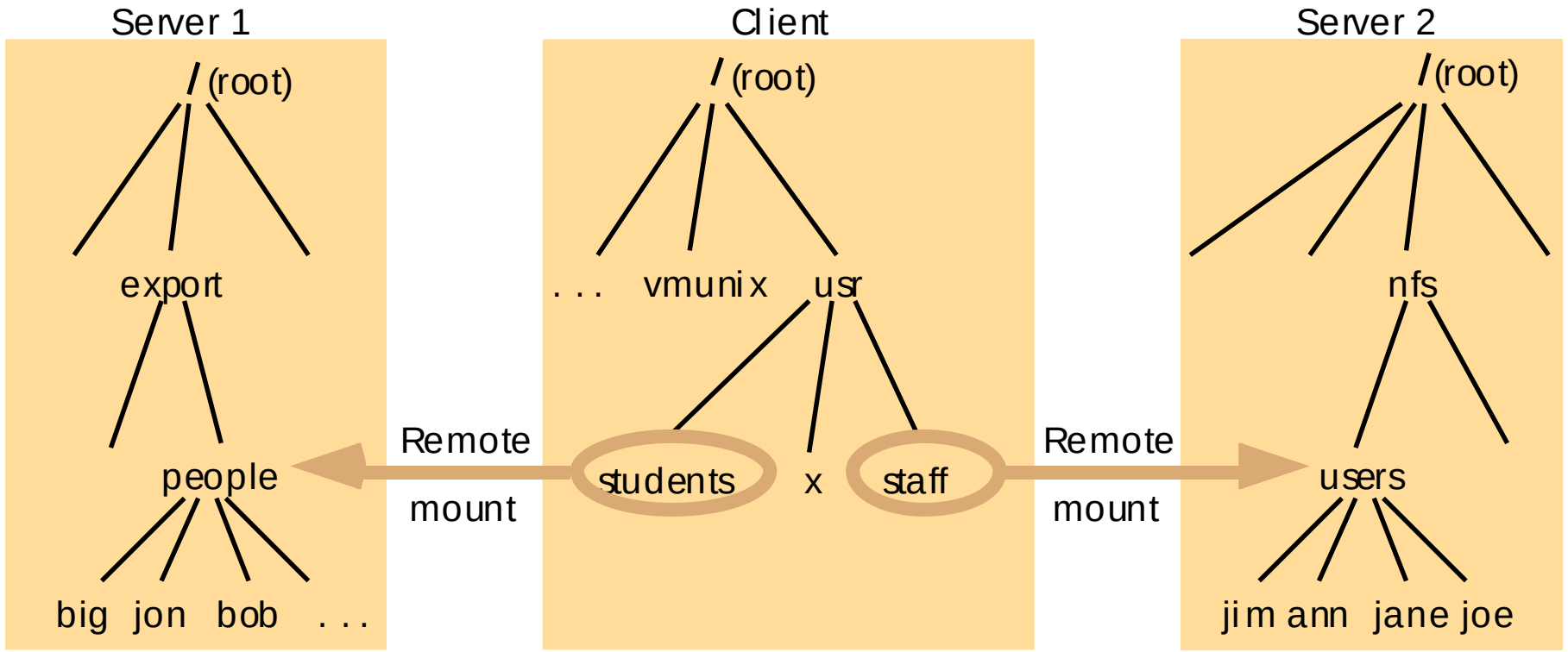
  mount(remotehost, remotedirectory, localdirectory)

- Server maintains a table of clients who have mounted filesystems at that server
- Each client maintains a table of mounted file systems holding:

  < IP address, port number, file handle>

- *Hard* versus *soft* mounts

Local and remote file systems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Automounter

NFS client catches attempts to access 'empty' mount points and routes them to the Automounter

- Automounter has a table of mount points and multiple candidate serves for each
- it sends a probe message to each candidate server and then uses the mount service to mount the filesystem at the first server to respond

- Keeps the mount table small
- Provides a simple form of replication for read-only filesystems
  - E.g. if there are several servers with identical copies of /usr/lib then each server will have a chance of being mounted at some clients.

Kerberized NFS

- Kerberos protocol is too costly to apply on each file access request
- Kerberos is used in the mount service:
  - to authenticate the user's identity
  - User's UserID and GroupID are stored at the server with the client's IP address
- For each file request:
  - The UserID and GroupID sent must match those stored at the server
  - IP addresses must also match
- This approach has some problems
  - can't accommodate multiple users sharing the same client computer
  - all remote filestores must be mounted each time a user logs in

New design approaches

Distribute file data across several servers

- Exploits high-speed networks (InfiniBand, Gigabit Ethernet)
- Layered approach, lowest level is like a 'distributed virtual disk'
- Achieves scalability even for a single heavily-used file

'Serverless' architecture

- Exploits processing and disk resources in all available network nodes
- Service is distributed at the level of individual files

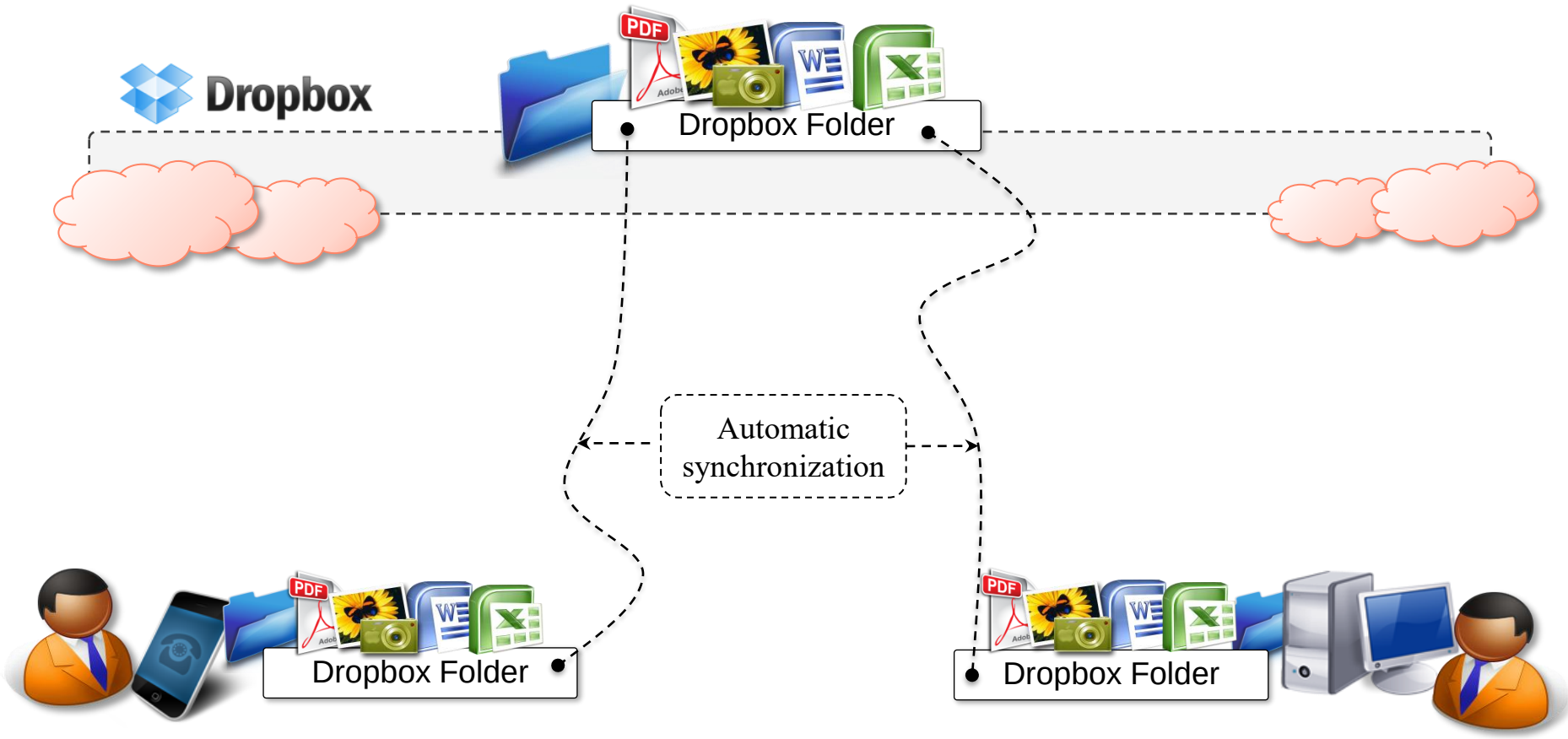
Examples:

xFS : Experimental implementation demonstrated a substantial performance gain over NFS and AFS

Peer-to-peer systems: Napster, OceanStore (UCB), Farsite (MSR), Publius (AT&T research) - see web for documentation on these very recent systems

Cloud-based File Systems: **DropBox**

DropBox Cloud Storage Architecture



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
- Superior scalability can be achieved with whole-file serving (Andrew FS) or the distributed virtual disk approach

Advanced Features:

  - support for mobile users, disconnected operation, automatic re-integration
  - support for data streaming and quality of service (Tiger file system, Content Delivery Networks)

- Modern DFSs are Cloud-based Files Systems (..Dropbox, GoogleDrive, OneDrive,..)