Distributed File Systems

Distributed File Systems

-Distributing file access across multiple nodes

- single homogeneous large file system

NFS: Network File System

AFS: Andrew File System

Distributed Disk Systems

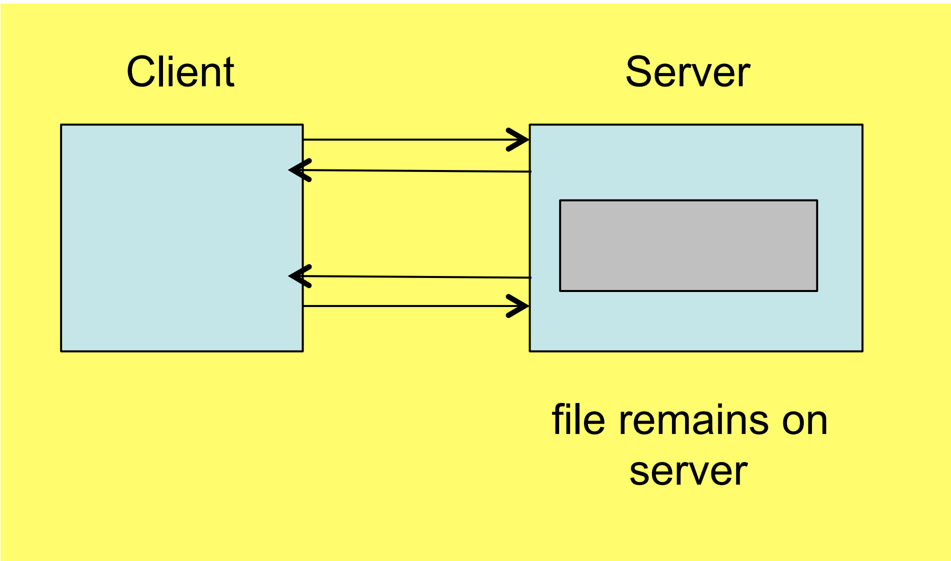
-Distributing data over multiple disks
- higher disk access bandwidth
- higher reliability

RAID: Reliable Array of Inexpensive Disks

Distributed File Systems

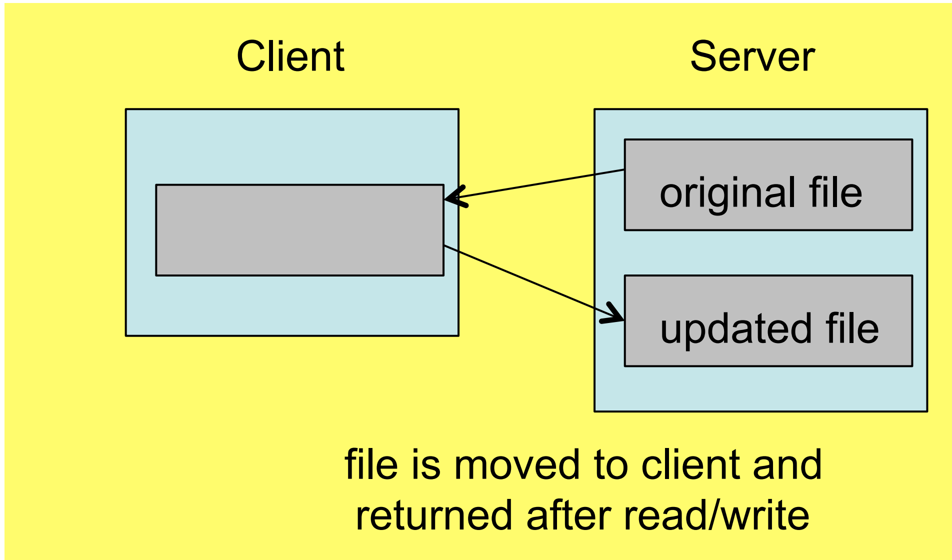
Models of Remote Access

remote acess model



remote execution
of file operations

upload/download model



local execution
of file operations

Distributed File Systems

Naming and Mounting

policy — mechanism

Local Interface to Distributed File System

Issues ?

Synchronization and Caching

Requirements for Distributed File Systems

- Transparencies (access, location, mobilty, performance, scalability)
- Concurrent File Update
- Replication of Files
- Openess (Heterogeneity of OS and Hardware)
- Fault-Tolerance
- Consistency
- Security
- Efficiency

Early milestones in distributed file systems

- D.R. Brownsbridge, L.F. Marshall, B. Randell: "The Newcastle Connection or UNIXes of the World Unite!", Software-Practice and Experience, Vol.12, 1147-1162, 1982
- B. Walker, G. Propek, R. English, C. Kline, and G. Thiel (UCLA)
  The LOCUS Distributed Operating System
  Proceedings of the Ninth ACM Symposium on Operating Systems Principles, October 10-13, 1983, pages. 49-70
- R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh
  The Design and Implementation of the SUN Network File System
  Proceedings Usenix Conference, Portland, Oregon 1985
  } **first commercial system**
- J. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, F.D. Smith
  Andrew: A distributed personal computing environment
  Comm. of the ACM, Vol.29, No. 3, 1986

AFS inspired the development of the "Distributed Computing Environment (DCE)"

First Approaches: The Newcastle Connection

SOFTWARE-PRACTICE AND EXPERIENCE. VOL. 12. 1147-1162 (1982)

The Newcastle Connection

or

UNIXes of the World Unite!

D. R. BROWNBRIDGE, L. F. MARSHALL AND B. RANDELL

Computing Laboratory, The University, Newcastle upon Tyne NE1 7RU, England

SUMMARY

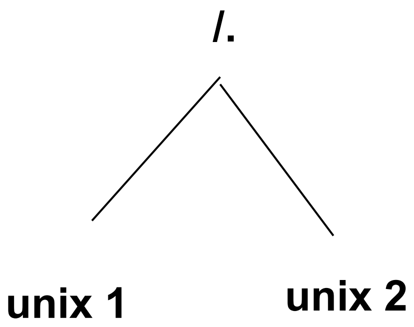
In this paper we describe a software subsystem that can be added to each of a set of physically interconnected UNIX or UNIX look-alike systems, so as to construct a distributed system which is functionally indistinguishable at both the user and the program level from a conventional single-processor UNIX system. The techniques used are applicable to a variety and multiplicity of both local and wide area networks, and enable all issues of inter-processor communication, network protocols, etc., to be hidden. A brief account is given of experience with such a distributed system, which is currently operational on a set of PDP11s connected by a Cambridge Ring. The final sections compare our scheme to various precursor schemes and discuss its potential relevance to other operating systems.

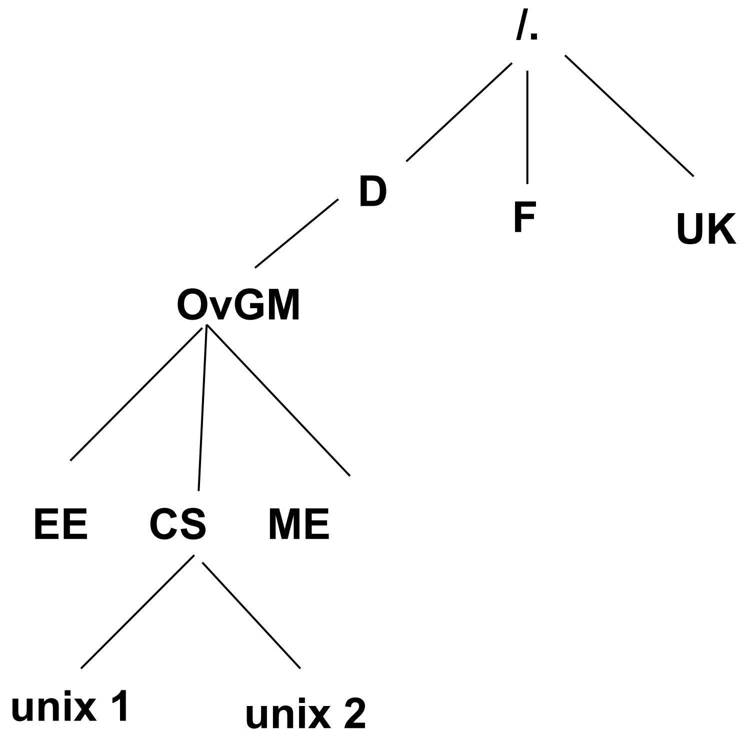
First Approaches: The Newcastle Connection

Principles:

- Extending the hierachical Unix Naming Scheme by a "Super Root",

- Using RPC to perform remote file access





Network File Systems

Newcastle connection provides a single name space for files.

Problems with the Newcastle Connection:

No Location transparency

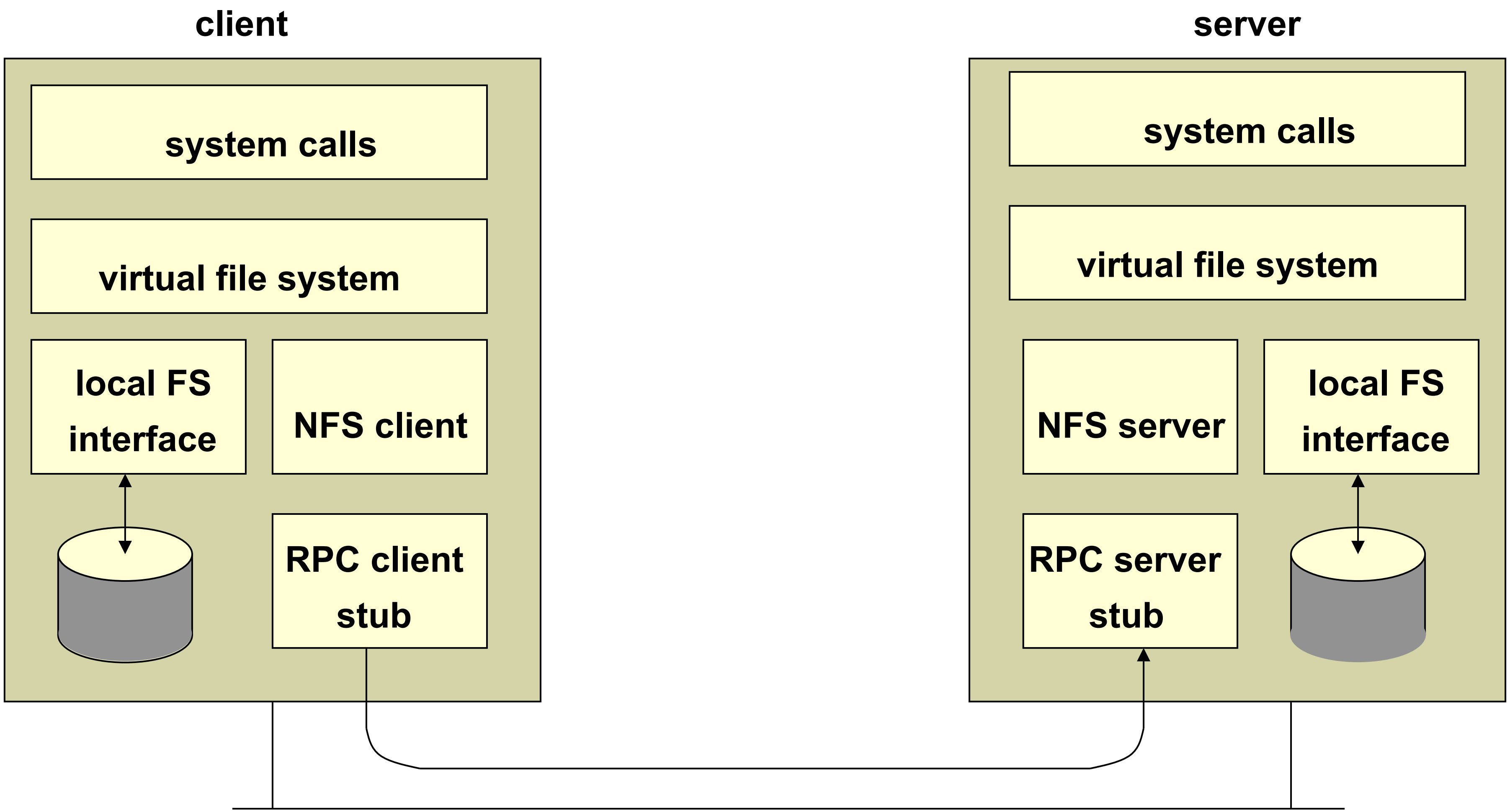
No Replication or Chaching

No Mobility Transparency

Network File Service Architecture

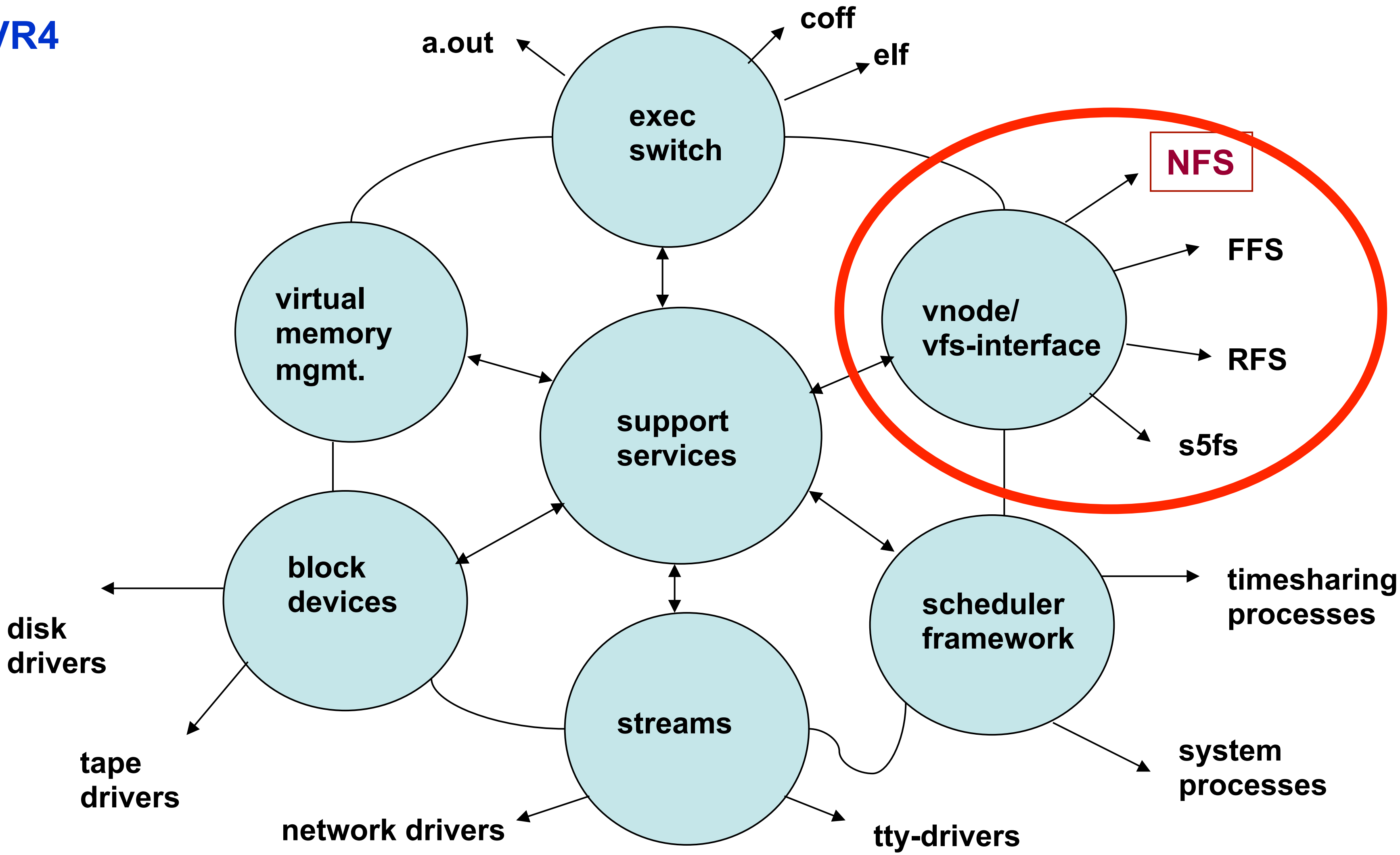
- **location transparency**
- **migration transparency**
- **robustness against client and server faults**

NFS Client-Server Architectures



Recall (BS I): Modern Unix-Kernel (Vahalia 1996)

SVR4



Differences to the Unix File System API

Stateless File Server:

- **no state information about open file**
- **no information about the number and state of clients**

  ➡ **every request must be self-contained.**

Benefit: **A client or a server crash does not require extensive recovery activities.**

- no open or close

- operations are idempotent except "create"

Flat File Service Operations

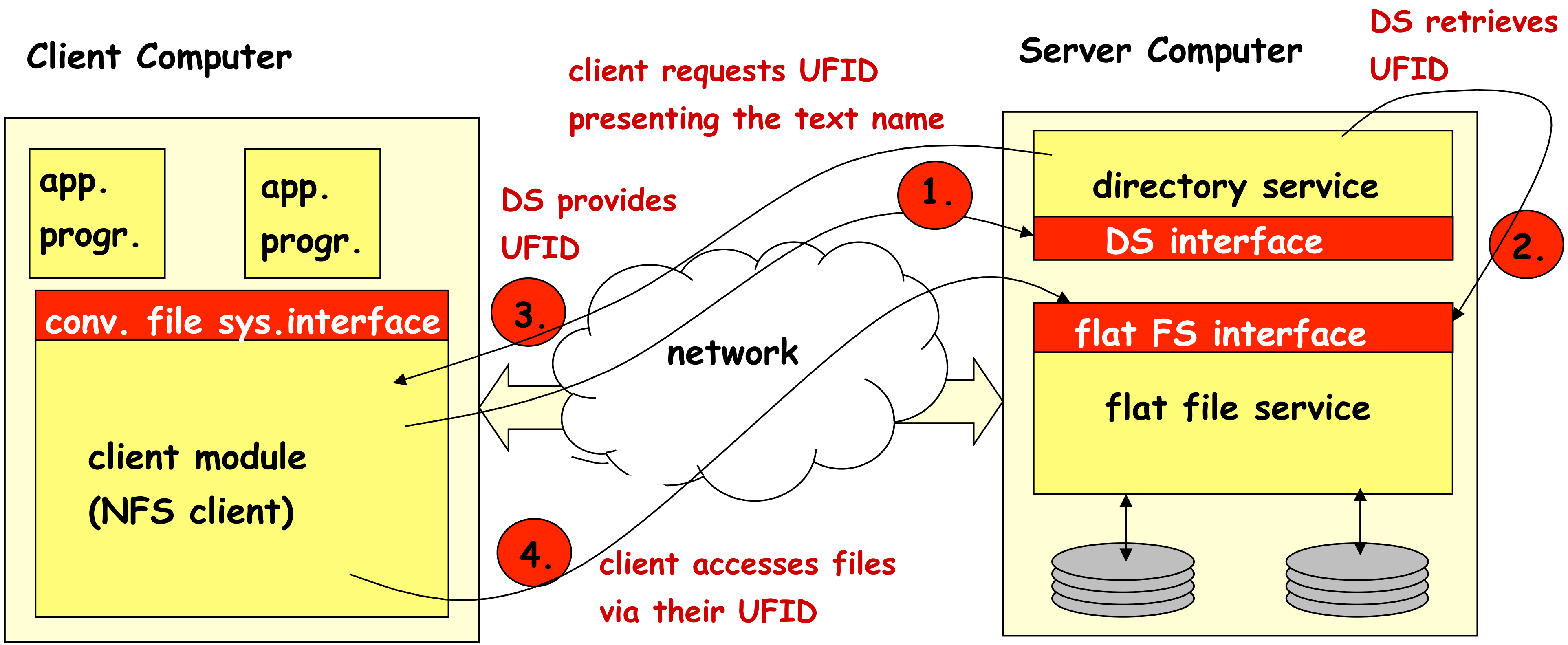
Used by the client module not used by programs at user level file access!

Read (FileId, i,n) → Data - throws *BadPosition*	**If** $1 \leq i \leq Length(File)$**: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data***
Write (FileId, i,n) → Data - throws *BadPosition*	**If** $1 \leq i \leq Length(File)+1$**: Writes a sequence of *Data* to a file starting at item *i*, extending the file if necessary**
Create() → FileId	**Creates a new file of length 0 and delivers a UFID for it.**
Delete(FileId)	**Removes a file from the file store.**
GetAttributes(FileId)→ Attr	**Returns the file attributes for the file.**
SetAttributes(FileId, Attr)	**Sets the file attributes for the file (except owner, type and ACL).**

Directory Service Operations

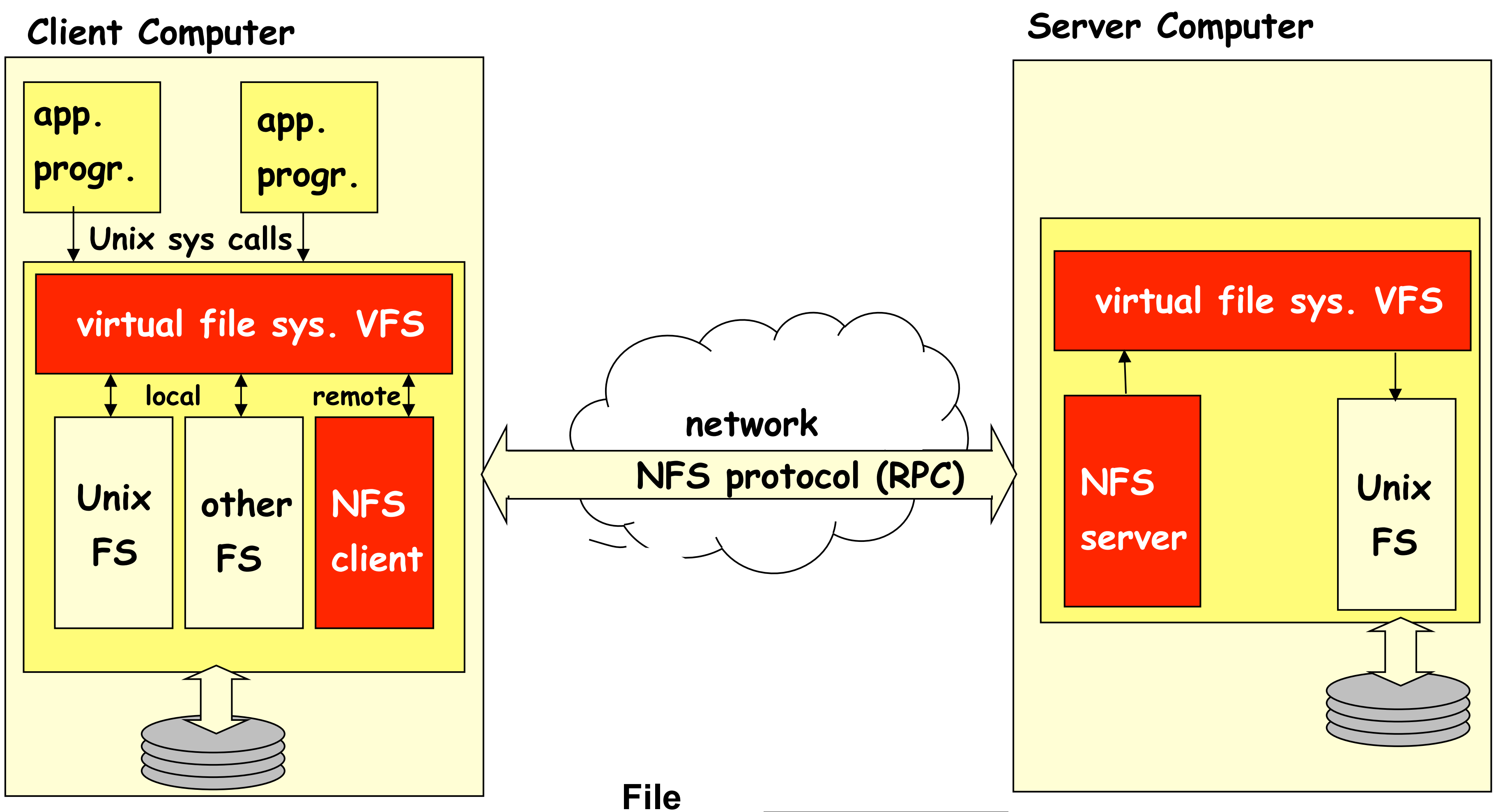
Lookup (Dir, Name) → FileId **- throws *NotFound***	**Locates the text name in the directory and returns the respective UFID. If *Name* is not found, an exception is raised.**
AddName (Dir, Name, File) **- throws *NameDuplicate***	**If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. Throws and exception if *Name* is already in the directory.**
UnName (Dir, Name) **- throws *NotFound***	**If *Name* is in the directory it is removed.** **If *Name* is not in the directory an exception is raised.**
GetNames (Dir, Pattern) → NameSeq	**Return all the text names in the directory that match the regular expresssion *Pattern*.**

Network File System Architecture



- Client-Server architecture using SUN RPC
- Flat FS uses Unique File IDs (UFIDs) instead of hierarchical path names
- DS associates file text names with Unique File IDs (UFID)

SUN NFS Architecture

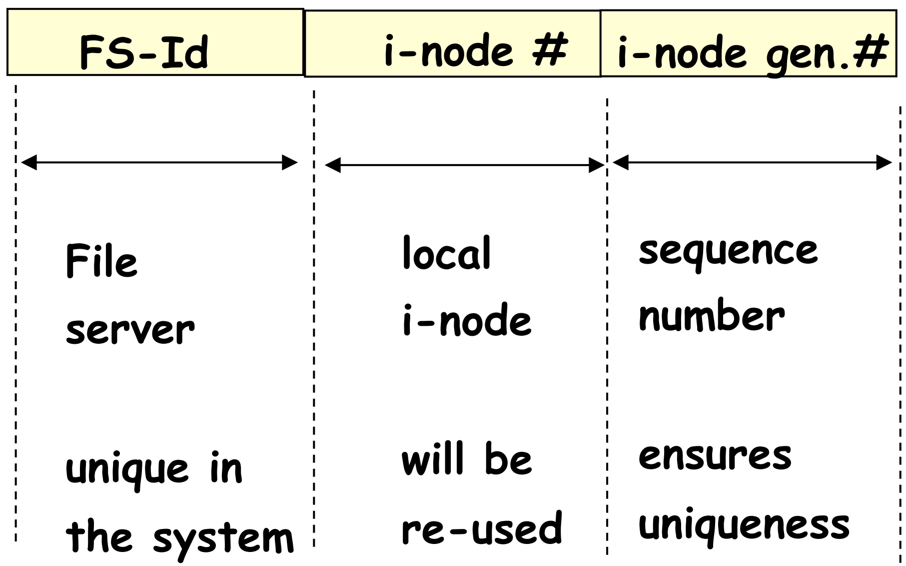


File Handle

FS-Id	i-node #	i-node gen.#

NFS File Handle

File Handle



NFSv2: 32 Byte

NFSv3: 64 Byte

NFSv4: 128 Byte

The File Handle enables file access to any file in the distributed file system without looking it up in the name server.

How to obtain a file handle in a remote file system subtree?

NFS server operations (simplified)

lookup(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) -> newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) status	Removes file name from directory *dirfh*.
getattr(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todirfh*
link(newdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

Continues on next slide ...

NFS server operations (simplified)

symlink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Naming in Network File Systems

Naming distinguishes between:

- User-Level Names e.g. UNIX path names (structured ns)

- Unique File Identifiers (UFID) System-wide unambiguous number (flat ns)

- **Hierarchical naming system is established using (flat) file system UIDs (UFID), and a directory service.**

- UFIDs support location transparency.

NFS mount service

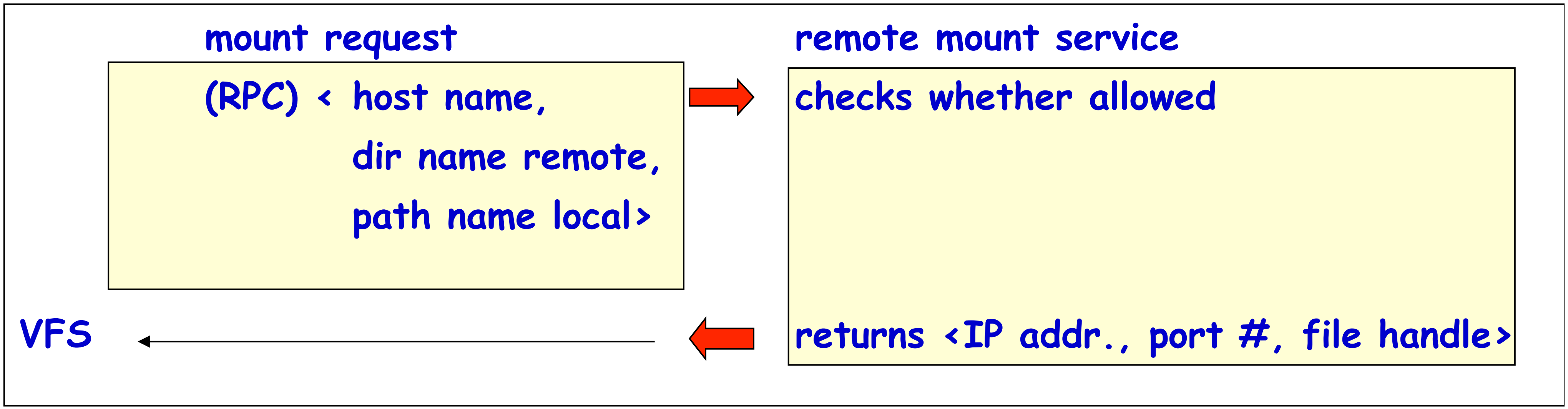
Mount Service Process: executed on every server

Data Structures:

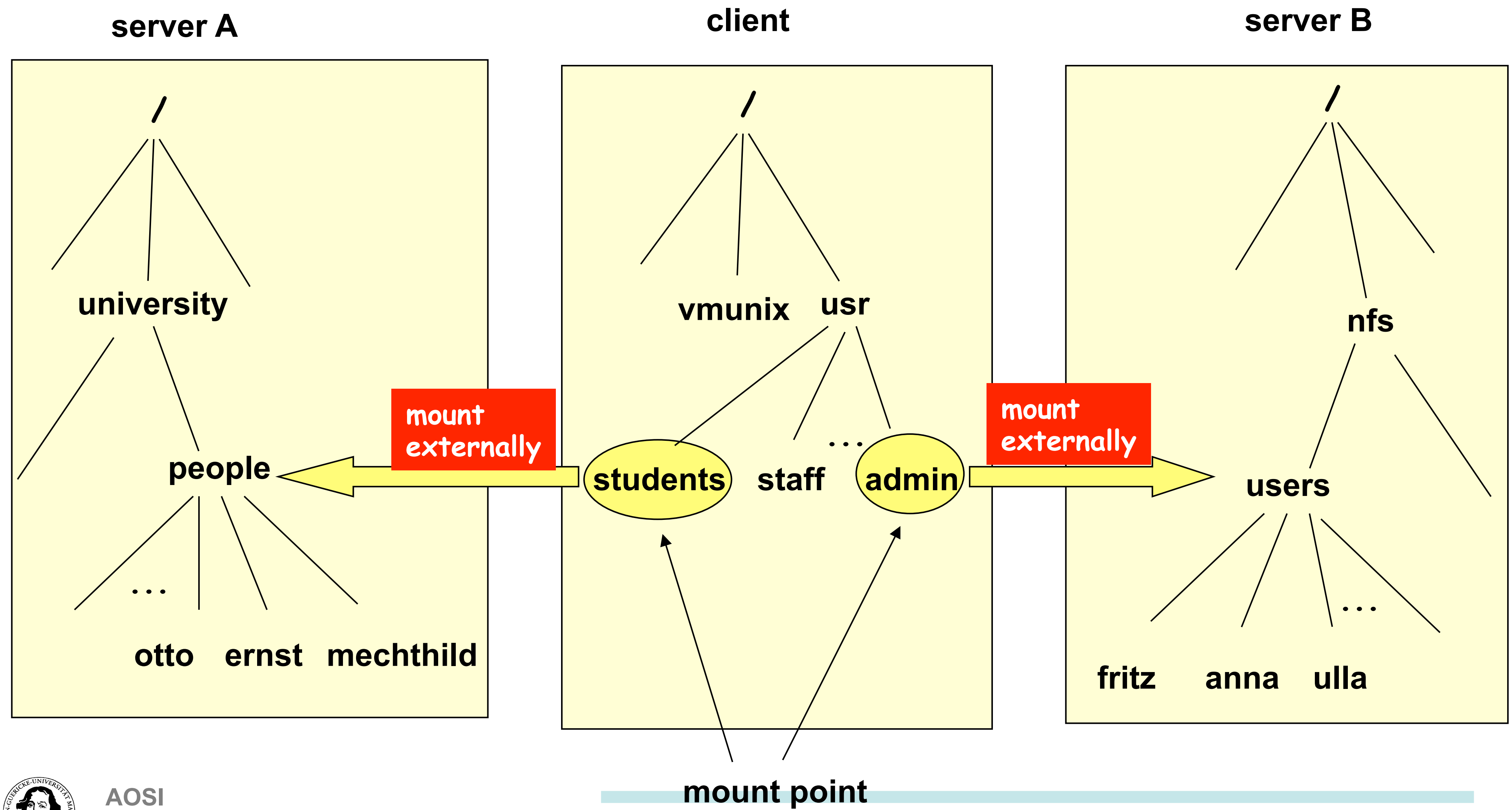
Server: **etc/exports**

contains names of local FS which may be mounted ext.

For every file system a list of names of (client) hosts is associated which are allowed to mount the FS.



NFS mount service



NFS mount service

Hard-Mounted: requesting application-level service blocks until the request is serviced. Server crashes and subsequent recovery is transparent for the application process.

Soft-Mounted: if the request cannot be serviced, the NFS client module signals an error condition to the application.

Soft-Mounting needs a meaningful reaction of the application process. In most cases the transparency of the hard-mounting is preferred.

NFS Server Caching

Standard Unix FS mechanisms

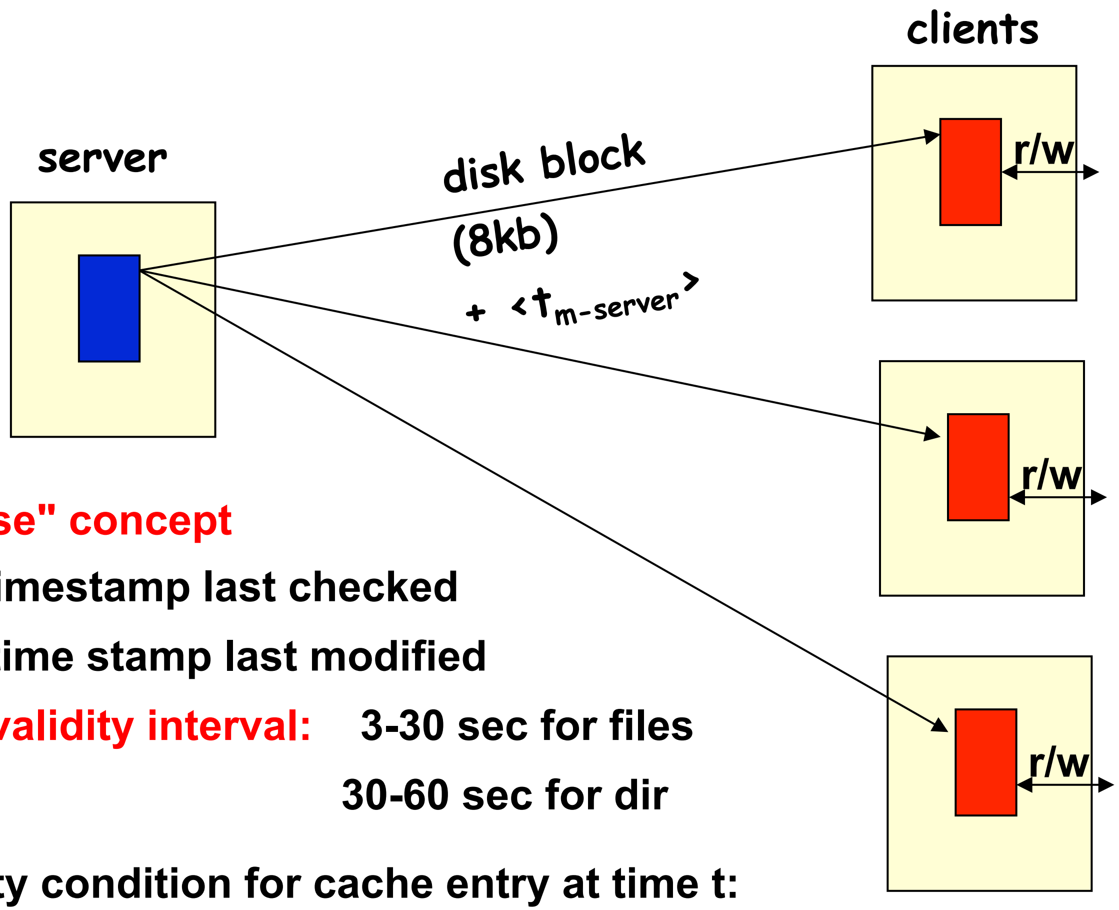
- buffer cache
- read ahead
- delayed write
- sync (periods of 30 sec)

Additionally: Two options for write (NFS version 3)

1.) Data from clients is written to the buffer cache AND the disk (write through). ⇒ Data is persistent when RPC returns.

2.) Data will be held in the cache only. Explicit commit-operation makes data persistent. Default mode for Standard NFS clients. Commit is issued when closing a file.

NFS Client Caching



READ:

all **reads** in an interval of Δt after chaching only go to the cache. **Reads** occuring after that time check the validity of the copy with the server. If still valid they may use it another Δt.

"lease" concept

t_c : timestamp last checked

t_m : time stamp last modified

Δt : validity interval: 3-30 sec for files

30-60 sec for dir

WRITE:

cached locally until a snyc of the client or if file is closed.

Validity condition for cache entry at time t:

$$(t - t_c < \Delta t) \vee (t_{m\text{-client}} = t_{m\text{-server}})$$

Mechanism only approximates 1-Copy-Consistency !

Dealing with shared Files

Unix Semantics: Every operation is instantaneously visible to all processes.

Session Semantics: No changes are visible to other processes until the file is closed.

Immutable files: No updates possible. On update a new file is created.

Transactions: All changes are atomic

Locking Files

Operation	Description
Lock	Create a lock for a range of bytes
Lockt	Test whether a conflicting lock has been created
LockU	Remove a lock from a range of bytes
Renew	Renew the lease on a specified block

"Share reservations“ (NFS 4)

weak form of type-specific access request

requested access	Current file denial state: none	read	write	both
read	succeed	fail	succeed	fail
write	succeed	succeed	fail	fail
both	succeed	fail	fail	fail

a client tries to open a file that has a certain denial status under the a certain access status

current access stat	Requested file denial state: none	read	write	both
read	succeed	fail	succeed	fail
write	succeed	succeed	fail	fail
both	succeed	fail	fail	fail

a client want to change the access status dynamically when other clients already have access under a certain denial state.

NFS Properties

Access Transparency	++	
Location Transparency	++	
Migration Transparency	+-	
Scalability	+	
File Replication	+-	only read replication
Heterogeneity	++	available for many platforms
Fault-Tolerance	+	stateless, restricted fault model
Consistency	+-	"almost" one copy
Security	-	needs additions (e.g. Cerberos)
Efficiency	++	

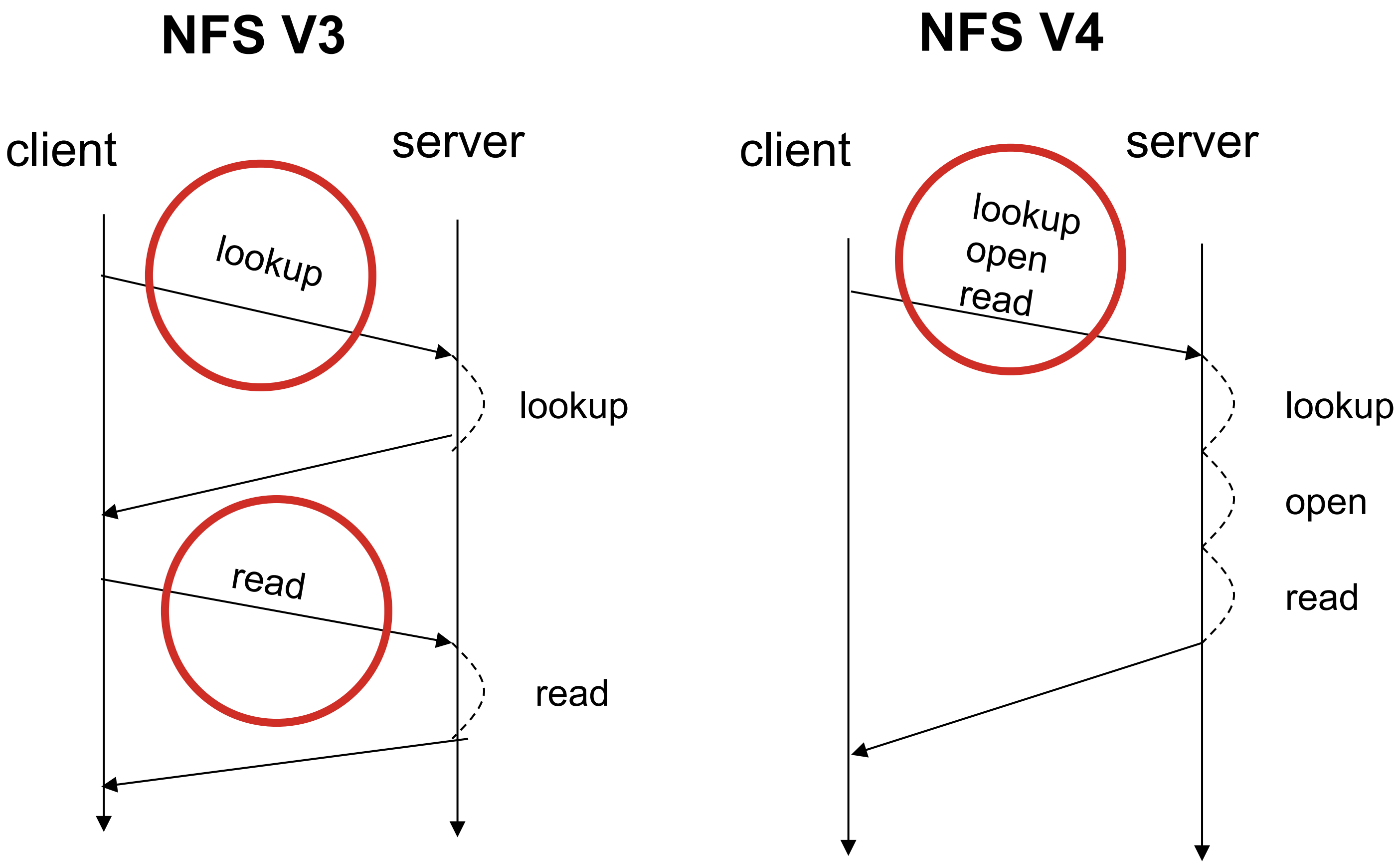
Network File System (NFS) version 4 Protocol

http://www.ietf.org/rfc/rfc3530.txt

New features of NFSv4

- NFSv4 introduces state. NFSv4 is a stateful protocol unlike NFSv2 or NFSv3.
- NFSv4 introduces file delegation. An NFSv4 server can enable an NFSv4 client to access and modify a file in its cache without sending any network requests to the server.
- NFSv4 uses compound remote procedure calls(RPCs) to reduce network traffic. An NFSv4 client can combine several traditional NFS operations (LOOKUP, OPEN, and READ) into a single compound RPC request to carry out a complex operation in one network round trip.
- NFSv4 specifies a number of sophisticated security mechanisms including Kerberos5 and Access Control Lists.
- NFSv4 can seamlessly coexist with NFSv3 and NFSv2 clients and servers.

Compound RPCs in NFS



NFS V4 Compound (mount) Request

nfernand@nf734153:/pdfs026/simple2

```
Network File System
    Program Version: 4
    V4 Procedure: COMPOUND (1)
    Tag: mount
        length: 12
        contents: mount
    minorversion: 0
    Operations (count: 5)
        Opcode: PUTROOTFH (24)
        Opcode: GETFH (10)
        Opcode: LOOKUP (15)
                ...
                ...
        Opcode: GETFH (10)
        Opcode: GETATTR (9)
            attrmask
                mand_attr: FATTR4_SUPPORTED_ATTRS (0)
                mand_attr: FATTR4_TYPE (1)
                ...
                ...
```

mount request

header info

(mount) Reply

nfernand@nf734153:/pdfs026/simple2

```
Network File System
    Program Version: 4
    V4 Procedure: COMPOUND (1)
    Status: NFS4_OK (0)
    Tag: mount
        length: 12
        contents: mount
    Operations (count: 5)
        Opcode: PUTROOTFH (24)
            Status: NFS4_OK (0)
        Opcode: GETFH (10)
            Status: NFS4_OK (0)
                ...
                ...
        Opcode: LOOKUP (15)
            Status: NFS4_OK (0)
        Opcode: GETFH (10)
            Status: NFS4_OK (0)
                ...
                ...
        Opcode: GETATTR (9)
            Status: NFS4_OK (0)
            obj_attributes
                attrmask
                    mand_attr: FATTR4_SUPPORTED_ATTRS (0)
                        attrmask
                            mand_attr: FATTR4_SUPPORTED_ATTRS (0)
                            mand_attr: FATTR4_TYPE (1)
                            ...
                            ...
```

NFS V4 setclientid Request

nfernand@nf734153:/pdfs026/simple2

```
Program Version: 4
V4 Procedure: COMPOUND (1)
Tag: setclientid
    length: 12
    contents: setclientid
Operations (count: 1)
    Opcode: SETCLIENTID (35)
        client
            ...
            ...
        callback
            cb_program: 0x00000000
            cb_location
                ...
        callback_ident: 0x00000000
```

NFS V4 setclientid Reply

nfernand@nf734153:/pdfs026/simple2

```
Network File System
    Program Version: 4
    V4 Procedure: COMPOUND (1)
    Status: NFS4_OK (0)
    Tag: setclientid
        length: 12
        contents: setclientid
    Operations (count: 1)
        Opcode: SETCLIENTID (35)
            Status: NFS4_OK (0)
            clientid: 0x448748b800000066
            ...
```

NFS V4 Open Request

hpdfs026

```
Network File System
    Program Version: 4
    V4 Procedure: COMPOUND (1)
    Tag: open
        length: 12
        contents: open
    minorversion: 0
    Operations (count: 4)
        Opcode: PUTFH (22)
        ...
        ...
        Opcode: OPEN (18)
            seqid: 0x00000001
            share_access: OPEN4_SHARE_ACCESS_BOTH (3)
            share_deny: OPEN4_SHARE_DENY_NONE (0)
            clientid: 0x448748b800000066
            ...
            ...
        Opcode: GETFH (10)
        Opcode: GETATTR (9)
            ...
```

NFS V4 Open Reply

hpdfs026

```
Network File System
    Program Version: 4
    V4 Procedure: COMPOUND (1)
    Status: NFS4_OK (0)
    Tag: open
        length: 12
        contents: open
    Operations (count: 4)
        Opcode: PUTFH (22)
            Status: NFS4_OK (0)
        Opcode: OPEN (18)
            Status: NFS4_OK (0)
            stateid
                seqid: 0x00000001
                other: 44D52AE40000006500000000
            ...
        Opcode: GETFH (10)
            Status: NFS4_OK (0)
            ...
        Opcode: GETATTR (9)
            Status: NFS4_OK (0)
            ...
```

Operation	v3	v4	Beschreibung
Create	Ja	Nein	Erstellen einer regulären Datei
Create	Nein	Ja	Erstellen einer irregulären Datei
Link	Ja	Ja	Erstellen einer direkten Verknüpfung zu einer Datei
Symlink	Ja	Nein	Erstellen einer symbolischen Verknüpfung zu einer Datei
Mkdir	Ja	Nein	Erstellen eines Unterverzeichnisses in einem gegebenen Verzeichnis
Mknod	Ja	Nein	Erstellen einer Spezialdatei
Rename	Ja	Ja	Ändern einer Dateibezeichnung
Remove	Ja	Ja	Entfernen einer Datei aus einem Dateisystem
Rmdir	Ja	Nein	Entfernen eines leeren Unterverzeichnisses aus einem Verzeichnis
Open	Nein	Ja	Öffnen einer Datei
Close	Nein	Ja	Schließen einer Datei
Lookup	Ja	Ja	Suchen einer Datei anhand ihrer Bezeichnung
Readdir	Ja	Ja	Lesen der Einträge eines Verzeichnisses
Readlink	Ja	Ja	Auslesen der in einer symbolischen Verknüpfung gespeicherten Pfadangabe
Getattr	Ja	Ja	Auslesen der Attributwerte einer Datei
Setattr	Ja	Ja	Setzen eines oder mehrerer Attributwerte für eine Datei
Read	Ja	Ja	Auslesen der in einer Datei enthaltenen Daten
Write	Ja	Ja	Schreiben von Daten in eine Datei

AFS Andrew File System

Scalability as primary design goal.

As much as possible local accesses to files.

Any accessed file transferred to the client.

Files stored persistently on local disc cache.

Large files are transfered in large chunks (64 kB).

Active notification mechanisms to approximate one-copy consistency.

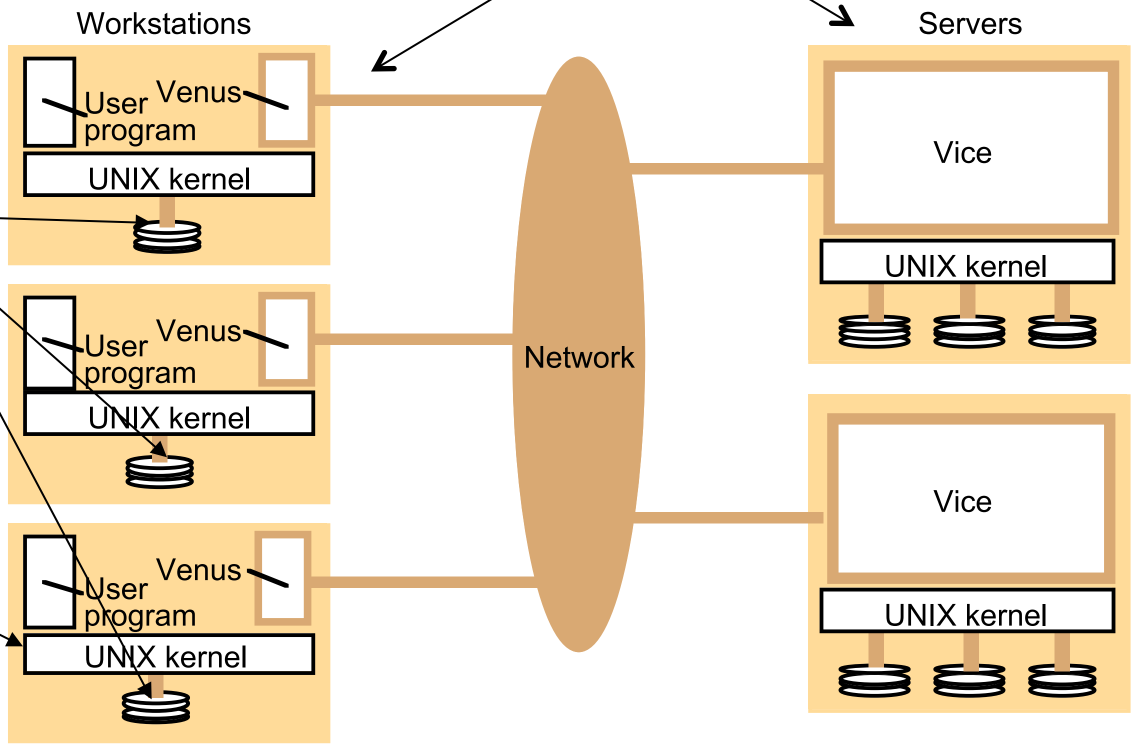
AFS Andrew File System

Questions:

- **How to find the server that holds the copy of the file?**
- **How does AFS obtain control if a client issues an open or close in the shared file space?**
- **Which memory space will be reserved for the cached files?**
- **How to ensure that the cached files constitute the most recent version of the file if more than one client has a copy?**

AFS Architecture

venus and vice



local persistent "file caches", survive crashes of local system.

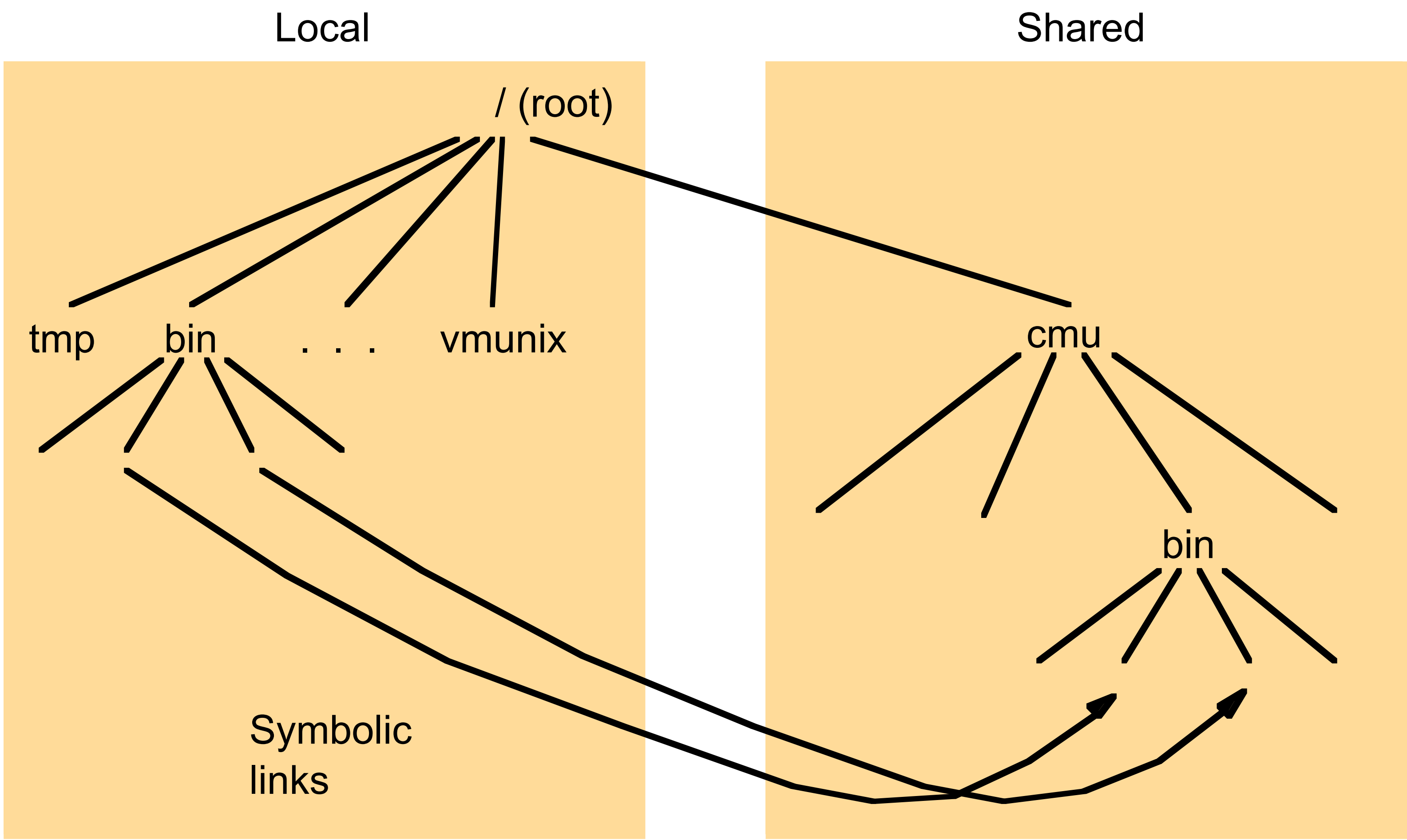
Unix kernel traps FS accesses and redirects requests to remote files to a Venus Process.

Files are organized in migratable "Volumes" (smaller entities compared to file systems in NFS).

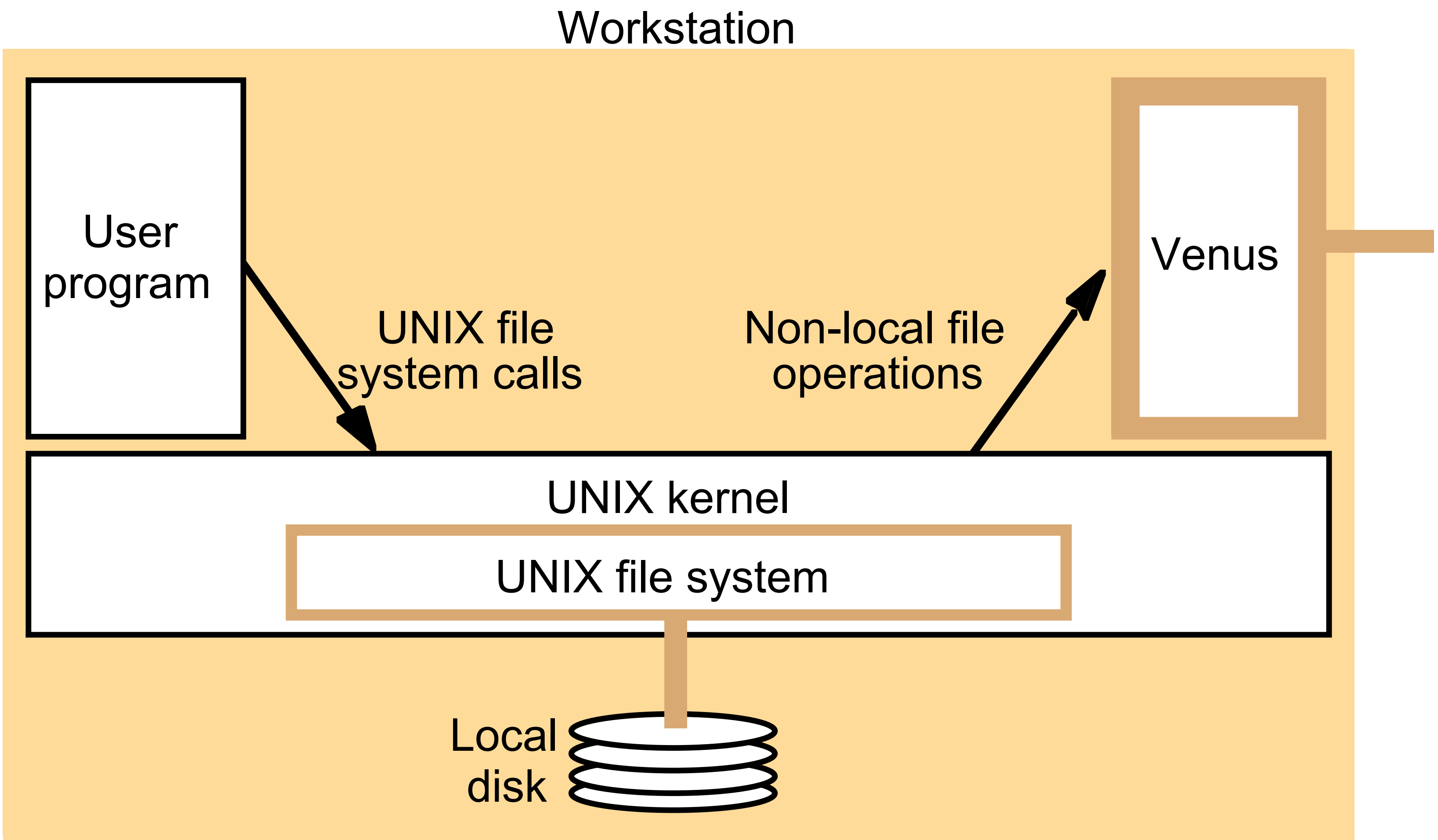
Flat File Service, hierarchical view is established by the Venus Processes

Every File has a unique 96-Bit ID (fid). Path names are translated in FIDs by Venus processes.

File name space seen by clients of AFS



System call interception in AFS



AFS: file system calls

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
		Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
	Open the local file and return the file descriptor to the application.			
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

The main components of the Vice service interface

Fetch(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() -> fid	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

AFS: Basic Consistency Mechanism

AFS consistency relies on a notification concept. The consistency mechanism is based on "Callback Promises“ (similar to write invalidate in chaches).

Callbacks are RPCs issued by the VICE server to the respective remote Venus processes with a Callback Promise Token as parameter. It guarantees that the VENUS process is notified if a client changed a file.

A Callback Promise Token may have the values:

- valid
- cancelled

The Server is responsible to invoke the respective remote Venus process when a file was modified with the value "cancelled".

A subsequent local "read" or "open" on the client must request a new file copy.

AFS properties

Replicated position database for volumes

- each server holds a full local copy of the position database

Write-protected replicas

- read-only volumes like /usr/bin or /man are replicared on multiple servers

Transfer of large chuncks of data (64kb)

Caching of parts of the file

- since version 3 only needed parts of files (64k) are transfered, consistency semantics is maintained.

Performance: Standard Benchmark with 18 server nodes

- Server load AFS: 40%
- Server load NFS: 100%

one reason is notification mechanism in AFS compared to time-out in NFS for consistency maintenance