Distributed File Systems

-Distributing data over multiple disks

- higher disk access bandwidth
- higher reliability

RAID: Reliable Array of Inexpensive Disks

-Distributing file access across multiple nodes

- single homogeneous large file system

NFS: Network File System

AFS: Andrew File System

RAID: Reliable Array of Inexpensive Disks

D.A. Patterson, G.A. Gibson, R. Katz: A Case for Redundand Arrays of Inexpensive Disks (RAID), *Proc. ACM SIGMOD Intern. Conference on Management of Data,* 1988

Goals:

Performance Improvement: parallel disks can be accessed concurrently.

Reliability and availability: RAID exploits redundancy of disks.

Transparency: RAID looks like a single large, fast and reliable disk (SLED).

RAID-level 1

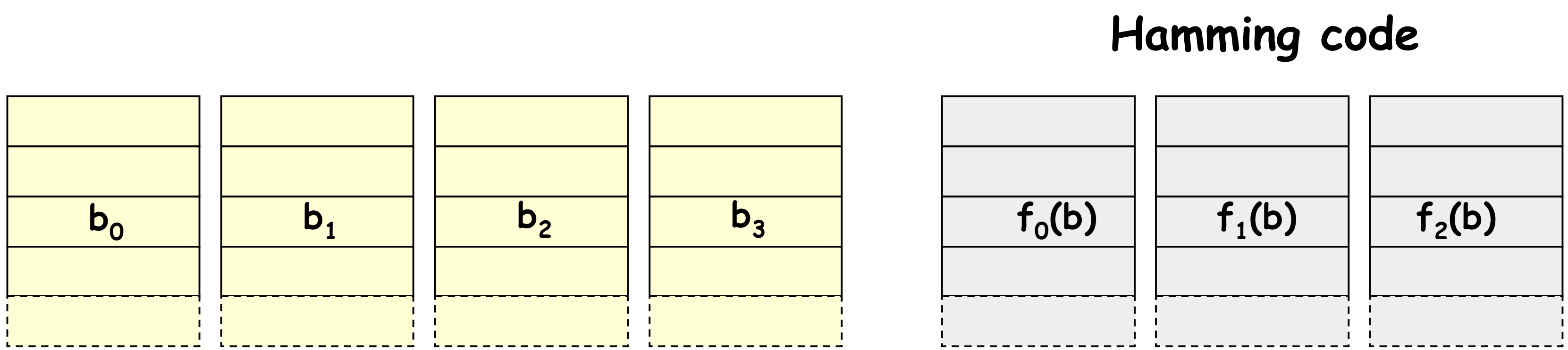
stripe 0	stripe 1	stripe 2	stripe 3
stripe 4	stripe 5	stripe 6	stripe 7
stripe 8	stripe 9	stripe 10	stripe 11
stripe 12	stripe 13	stripe 14	stripe 15

stripe 0	stripe 1	stripe 2	stripe 3
stripe 4	stripe 5	stripe 6	stripe 7
stripe 8	stripe 9	stripe 10	stripe 11
stripe 12	stripe 13	stripe 14	stripe 15

RAID-level 0	RAID-level 1
non-redundant	mirrored disk
high transfer rates	high transfer rates

RAID-level 2

Needs strictly synchronized disks!

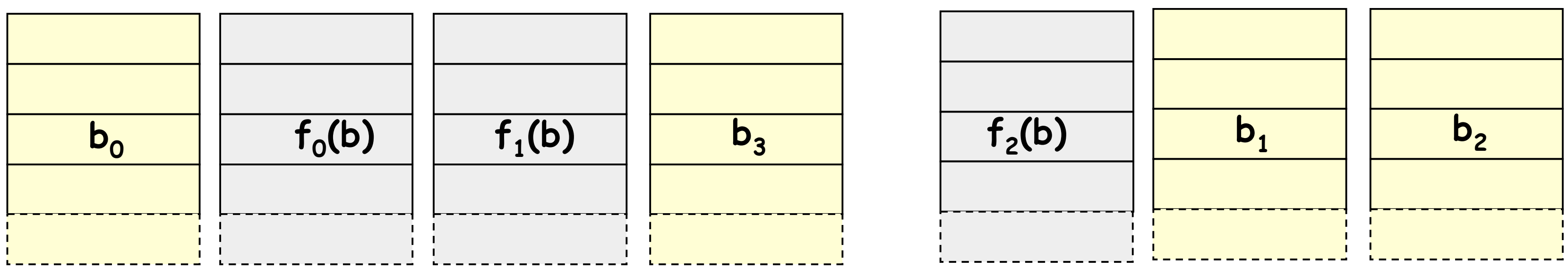


RAID-level 2

word- or byte-oriented

RAID-level 2

Needs strictly synchronized disks!

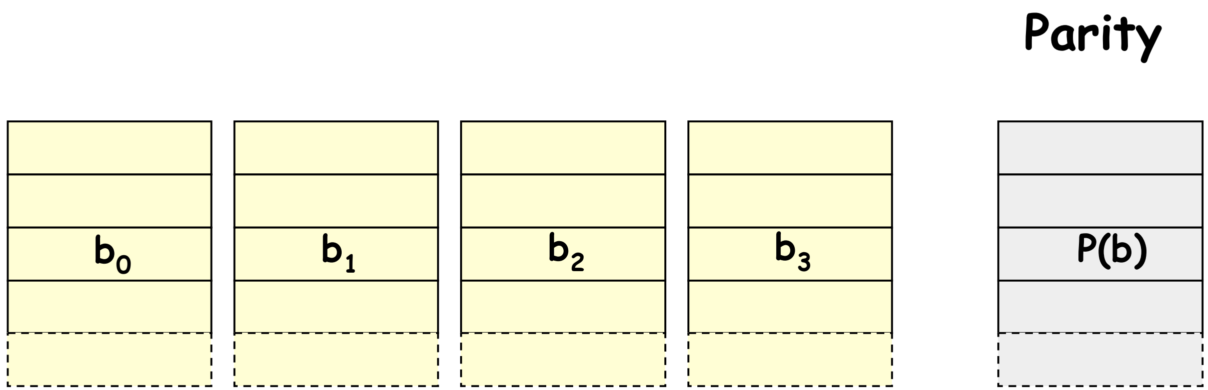


RAID-level 2

word- or byte-oriented

RAID-level 3

Needs strictly synchronized disks!



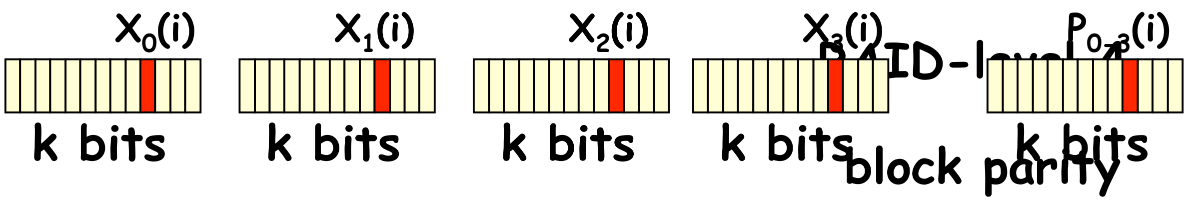
RAID-level 3

word- or byte-oriented

Allows error correction in case of a defective disk because the positon of the defective bit is known !

RAID-level 4

stripe 0	stripe 1	stripe 2	stripe 3	P(0-3)
stripe 4	stripe 5	stripe 6	stripe 7	P(4-7)
stripe 8	stripe 9	stripe 10	stripe 11	P(8-11)
stripe 12	stripe 13	stripe 14	stripe 15	P(12-14)



$P_{0-3}(i) = X_3(i) \oplus X_2(i) \oplus X_1(i) \oplus X_0(i)$ starting point

$P'_{0-3}(i) = X_3(i) \oplus X'_2(i) \oplus X_1(i) \oplus X_0(i)$ changing stripe 2

$P'_{0-3}(i) = X_3(i) \oplus X'_2(i) \oplus X_1(i) \oplus X_0(i) \oplus X_2(i) \oplus X_2(i)$

$P'_{0-3}(i) = P_{0-3}(i) \oplus X'_2(i) \oplus X_2(i)$

A write operation requires 2 reads and 2 writes

RAID-level 5

Problem with RAID-4: Parity disk becomes a bottleneck.

stripe 0	stripe 1	stripe 2	stripe 3	P(0-3)
stripe 4	stripe 5	stripe 6	P(4-7)	stripe 7
stripe 8	stripe 9	P(8-11)	stripe 10	stripe 11
stripe 12	P(12-14)	stripe 13	stripe 14	stripe 15
P(15-19)	stripe 16	stripe 17	stripe 18	stripe 19

RAID-level 5

Block parity

Raid-level 6 tolerates two disk crashes and guarantees a very high availability of data. Needs N+2 disks and has to write 2 Parity blocks on a write operation.

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

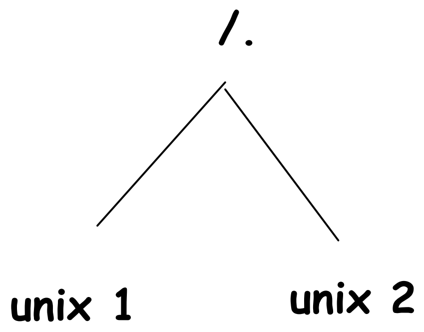
SUMMARY

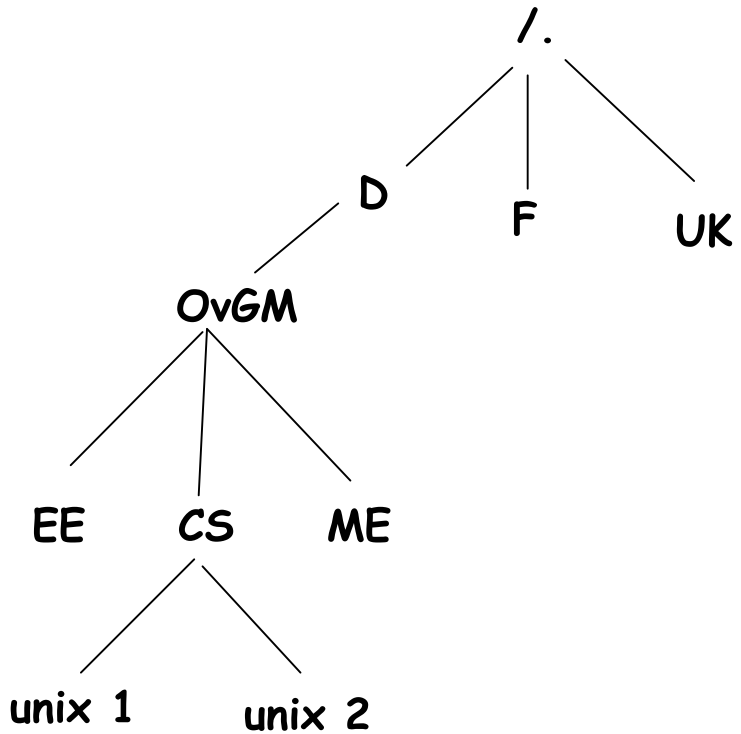
In this paper we describe a software subsystem that can be added to each of a set of physically interconnected UNIX or UNIX look-alike systems, so as to construct a distributed system which is functionally indistinguishable at both the user and the program level from a conventional single-processor UNIX 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





Distributed 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

Distributed 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)

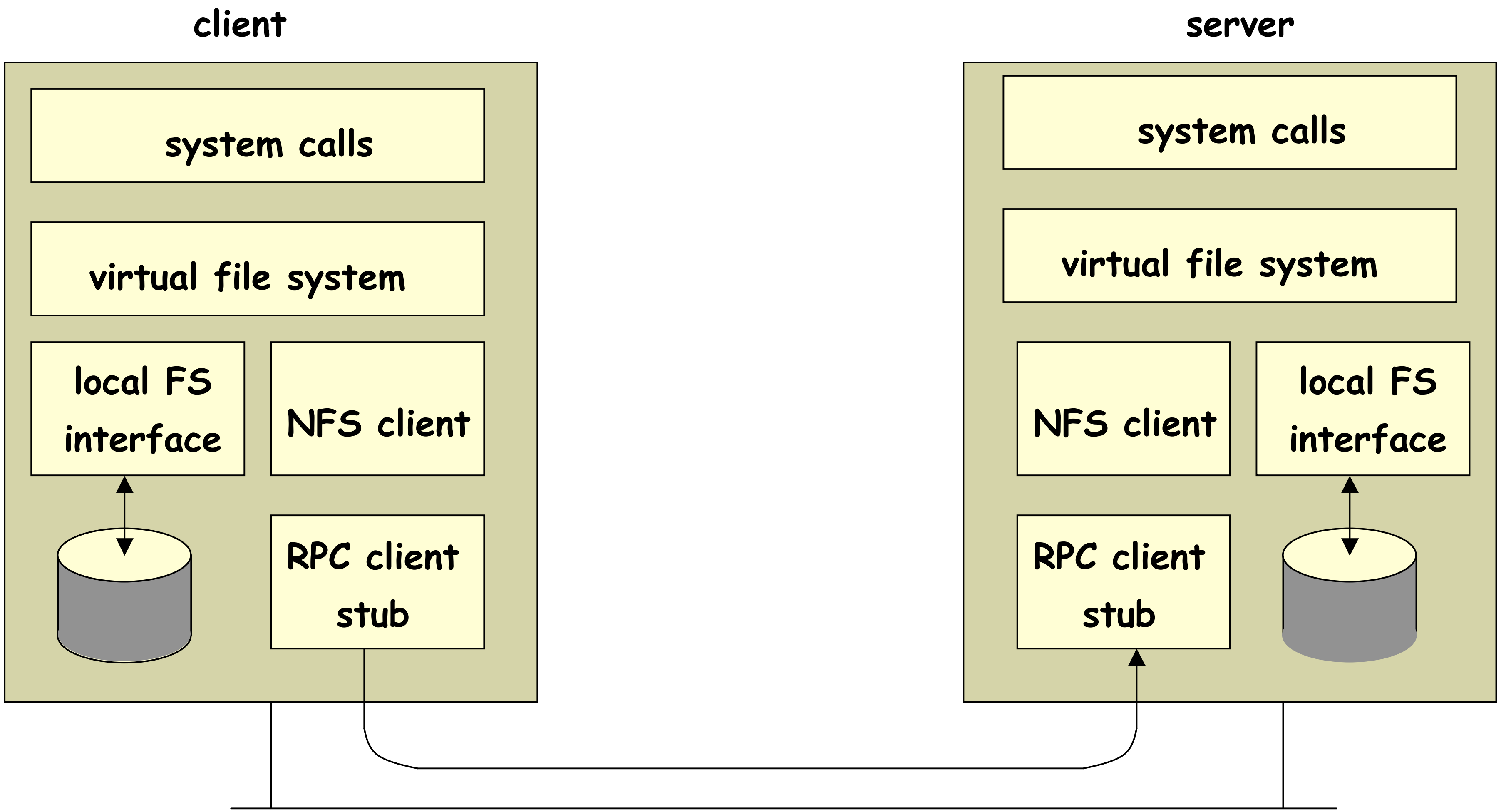
- Hierarchical naming system is established using (flat) file system UIDs (UFID), and a directory service.

- UFIDs support location transparency.

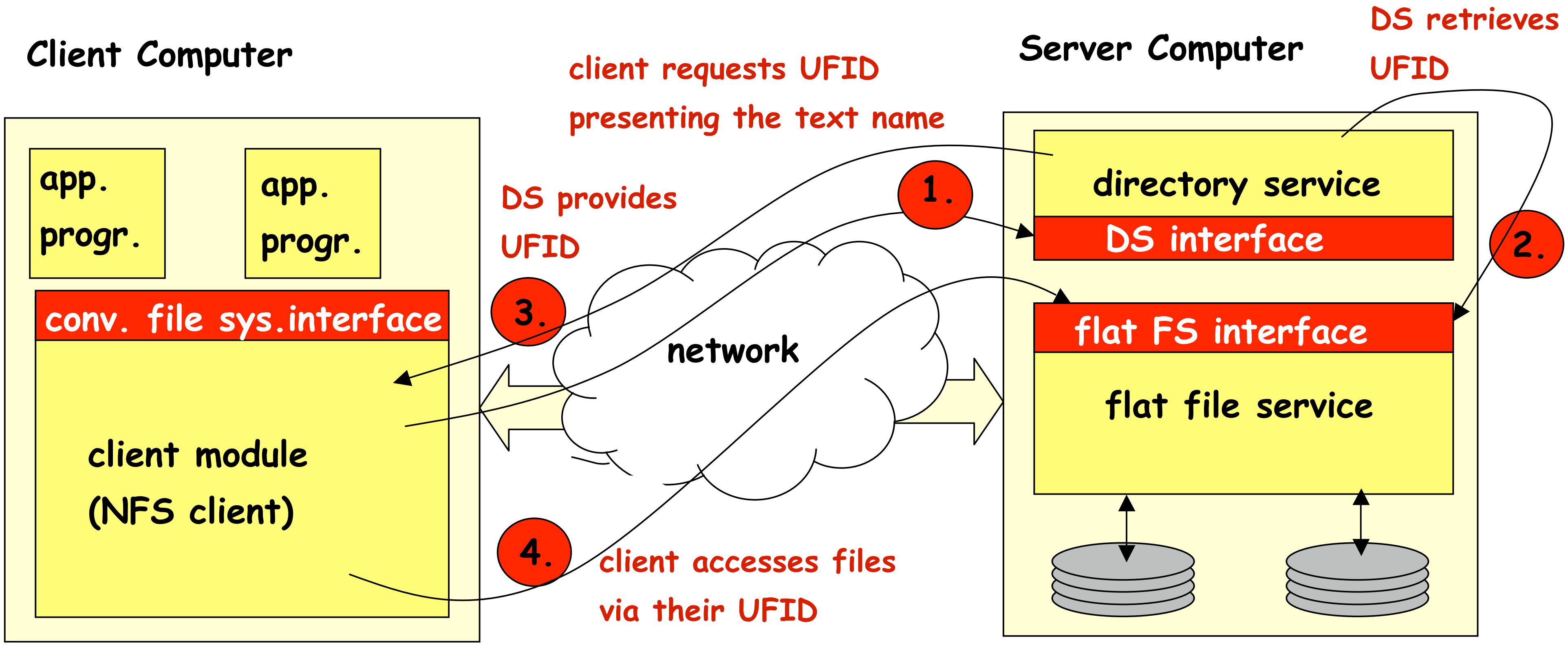
Network File Service (NFS) Architecture

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

Client-Server Architectures



NFS: File Service 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)

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,n) → 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 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*.**

Differences to the Unix File System API

Stateless File Server:

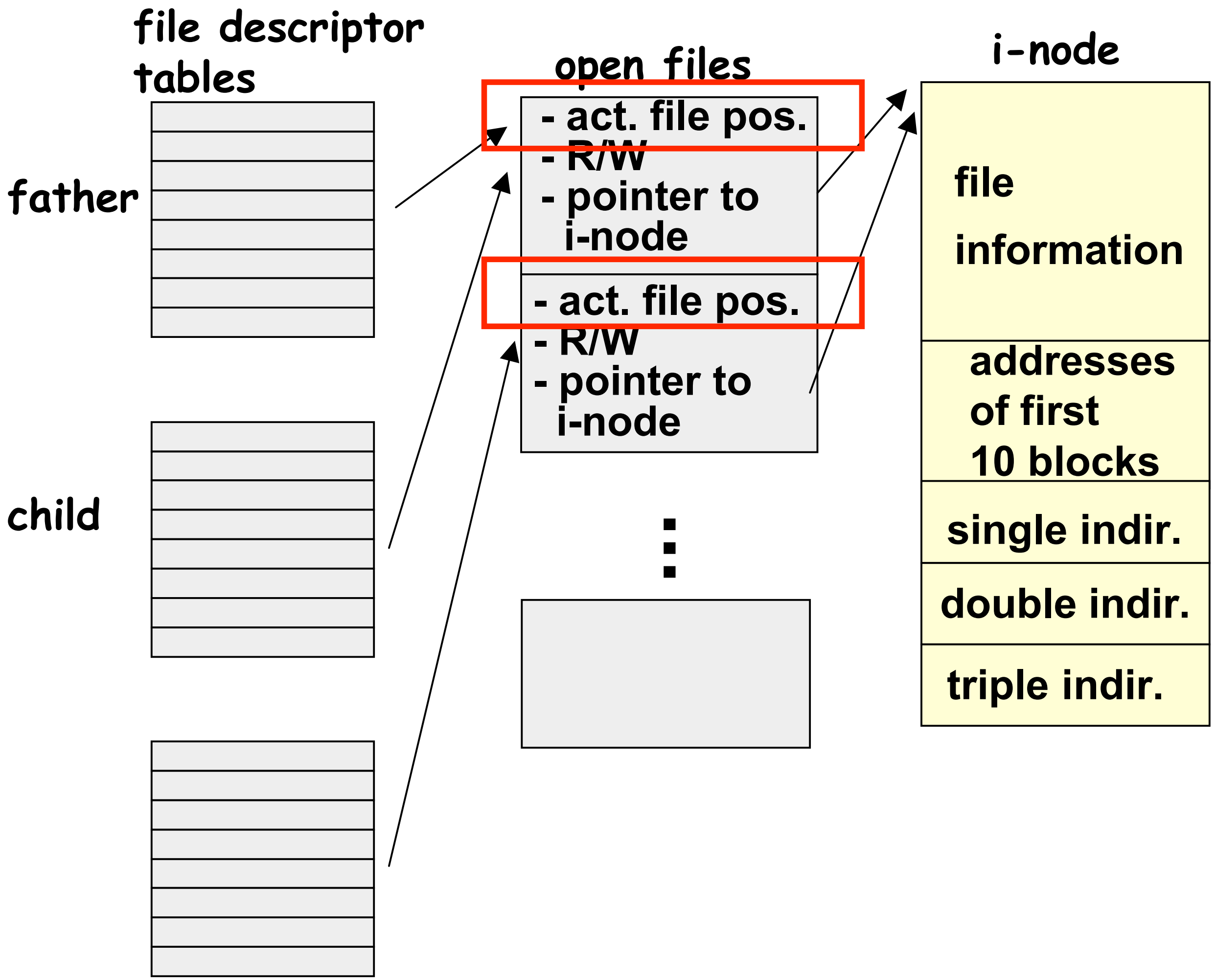
- no state information about open file
- no information about the number and state of clients

  ⇒ 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"

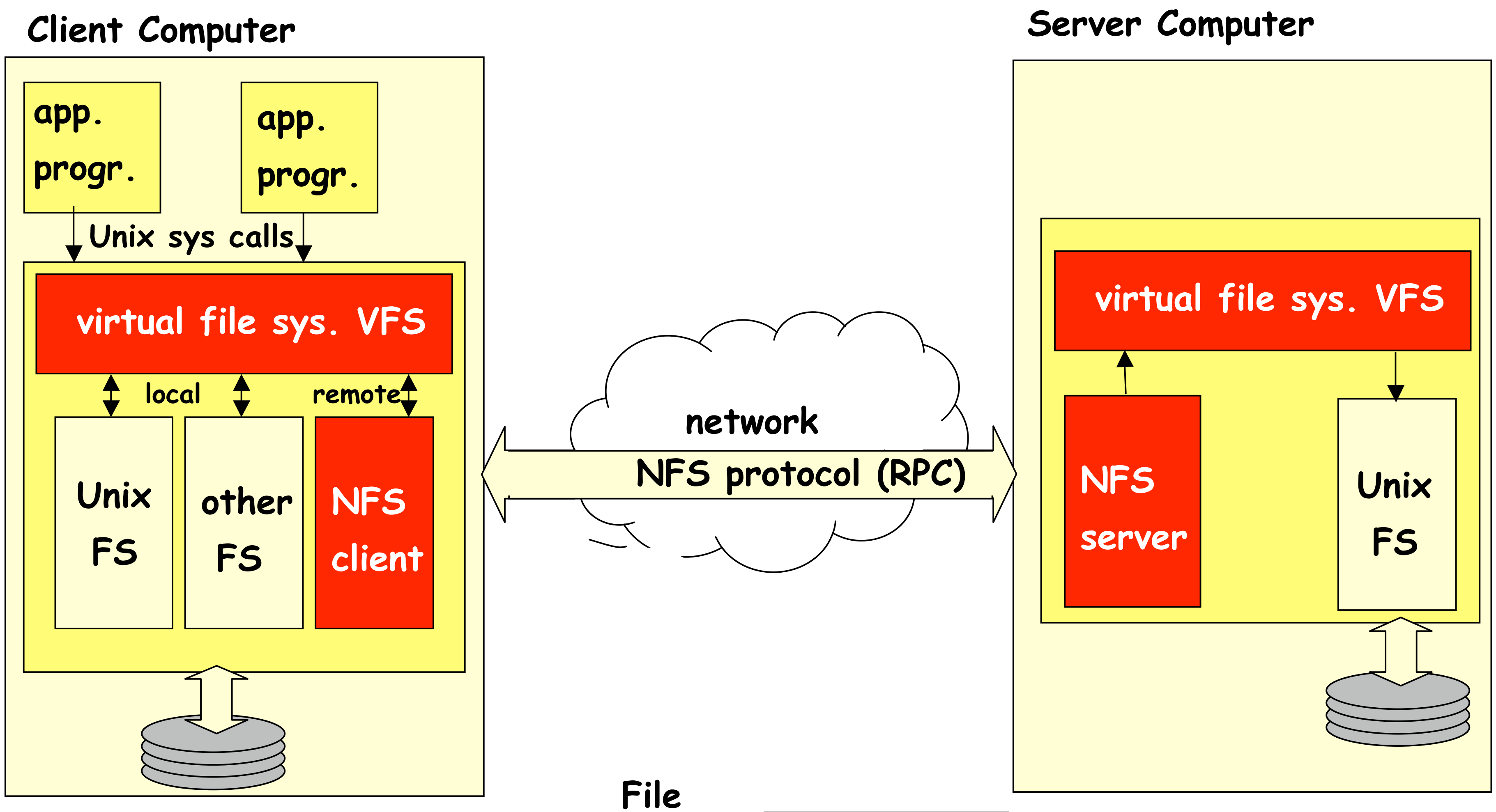
Recall: file allocation in Unix



Unix file system remembers which files are open and the position of the last file access!

→ read and write NOT idempotent!

SUN NFS Architecture



File Handle

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

NFS File Handle

File Handle	FS-Id	i-node #	i-node gen.#
	File server	local i-node	sequence number
	unique in the system	will be re-used	ensures uniqueness

NFSv2: 32 Byte

NFSv3: 64 Byte

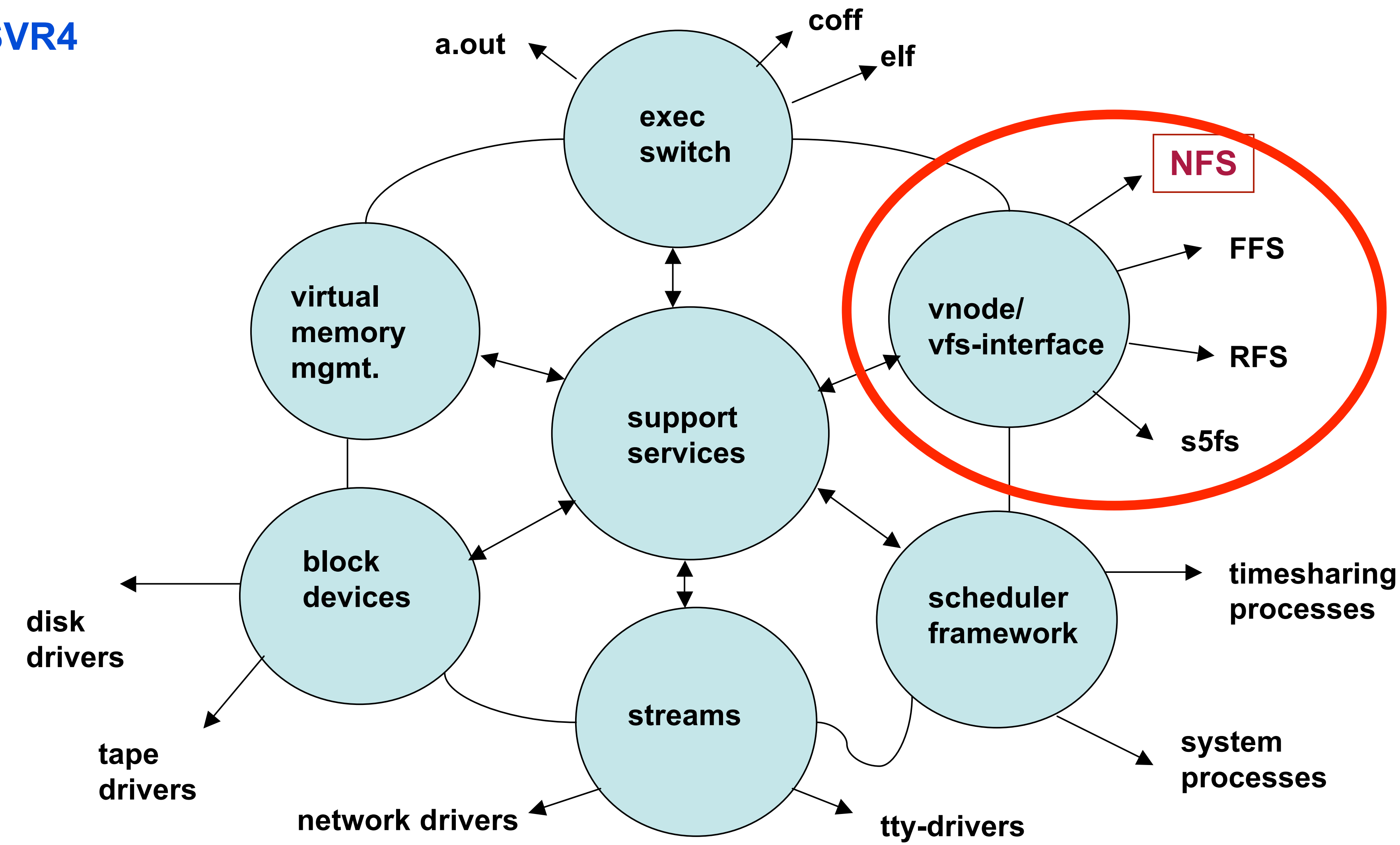
NFSv4: 128 Byte

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

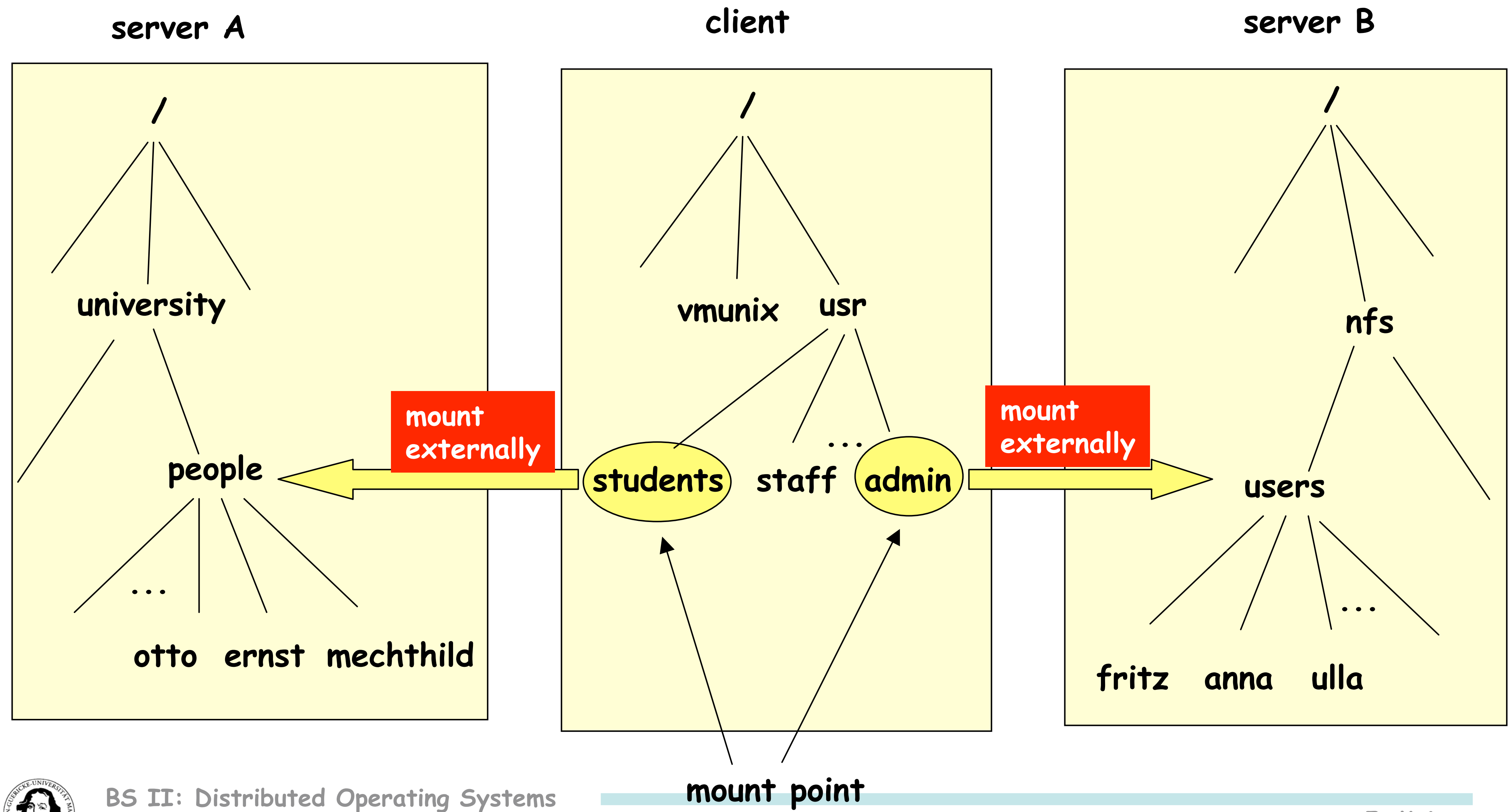
How to obtain a file handle in a remote file system subtree?

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

SVR4



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 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.

mount request

(RPC) < host name,
dir name remote,
path name local>

remote mount service

checks whether allowed

returns <IP addr., port #, file handle>

VFS

NFS Server Caching

Standard Unix FS mechanisms

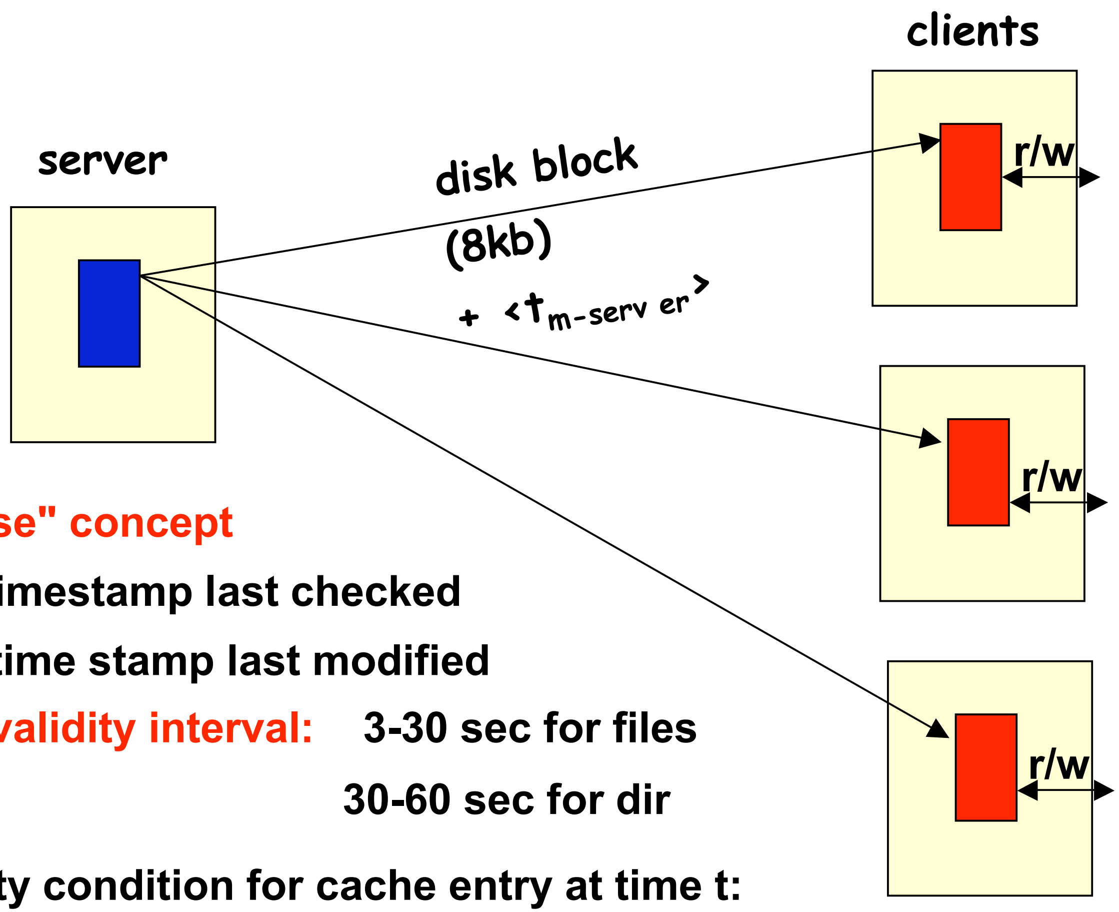
- buffer cache
- read ahead
- delayed write
- sync (periods of 30 sec)

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



"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

Validity condition for cache entry at time t:

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

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.

WRITE:

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

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"

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

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

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	++	

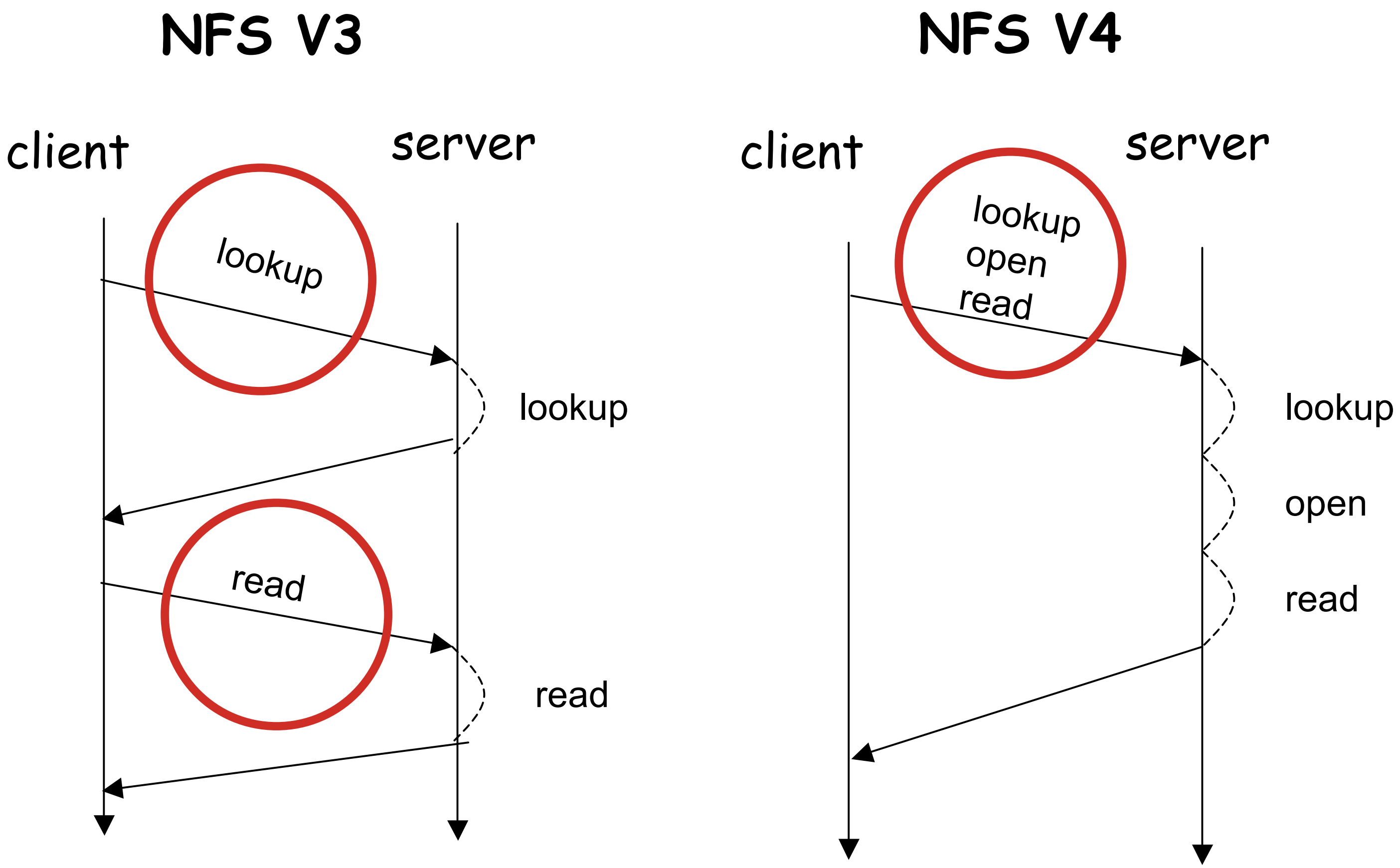
Network File System (NFS) version 4 Protocol

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 is <u>completely</u> transferred to the client.

Files stored persistently on local disc cache.

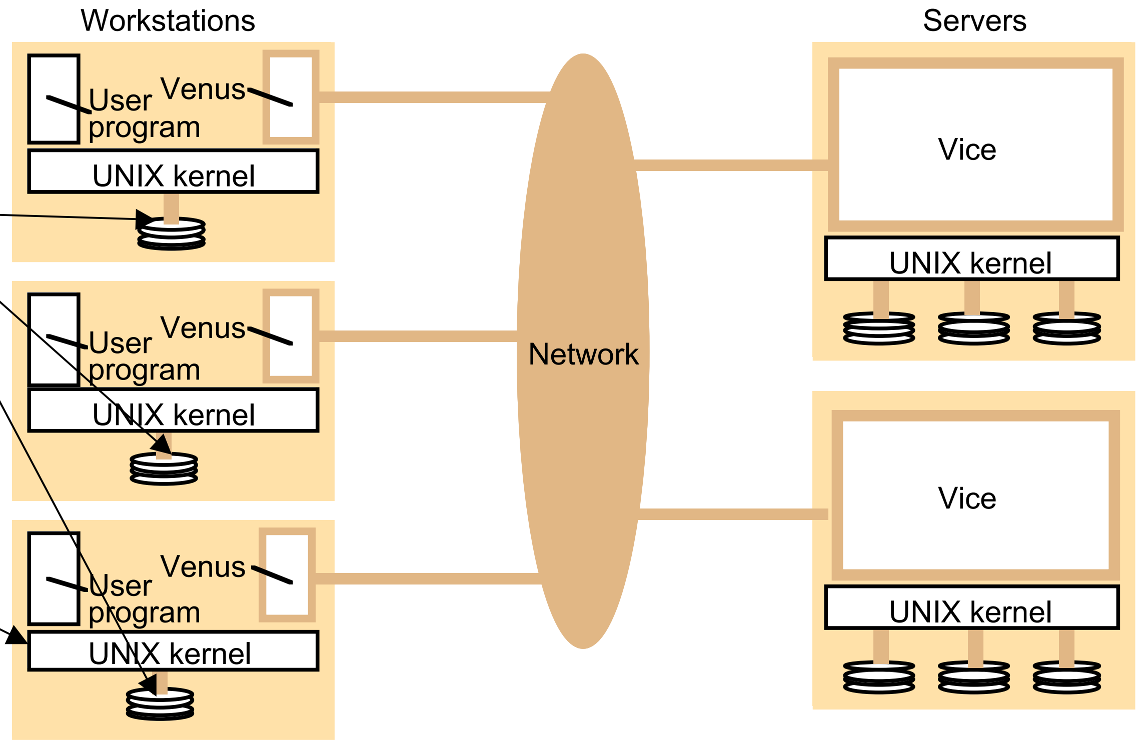
Large files are transfered in large chunks (64 kB).

Active notification mechanisms to approximate one-copy consistency.

AFS Architecture

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.

AFS: Basis Consistency Mechanism

Consistency mechanism is based on "Callback Promises".

AFS relies on a notification concept. Callbacks are RPCs to the respective remote Venus processes with a Callback Promise Token as parameter.

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: 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. Open the local file and return the file descriptor to the application.	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. 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.	→ ←	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
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.