Distributed Shared Data Storage

Distributed Shared Data Storage

Distributed Shared Memory (DSM)

Structure:

- Orientation, Granularity

Consistency Models:

- From strong to weak
- Protocols

Distributed File Systems (DFS)

- General problems of distribution
- Examples:NFS, AFS

Goal: Keep the well known interface of a single computer system

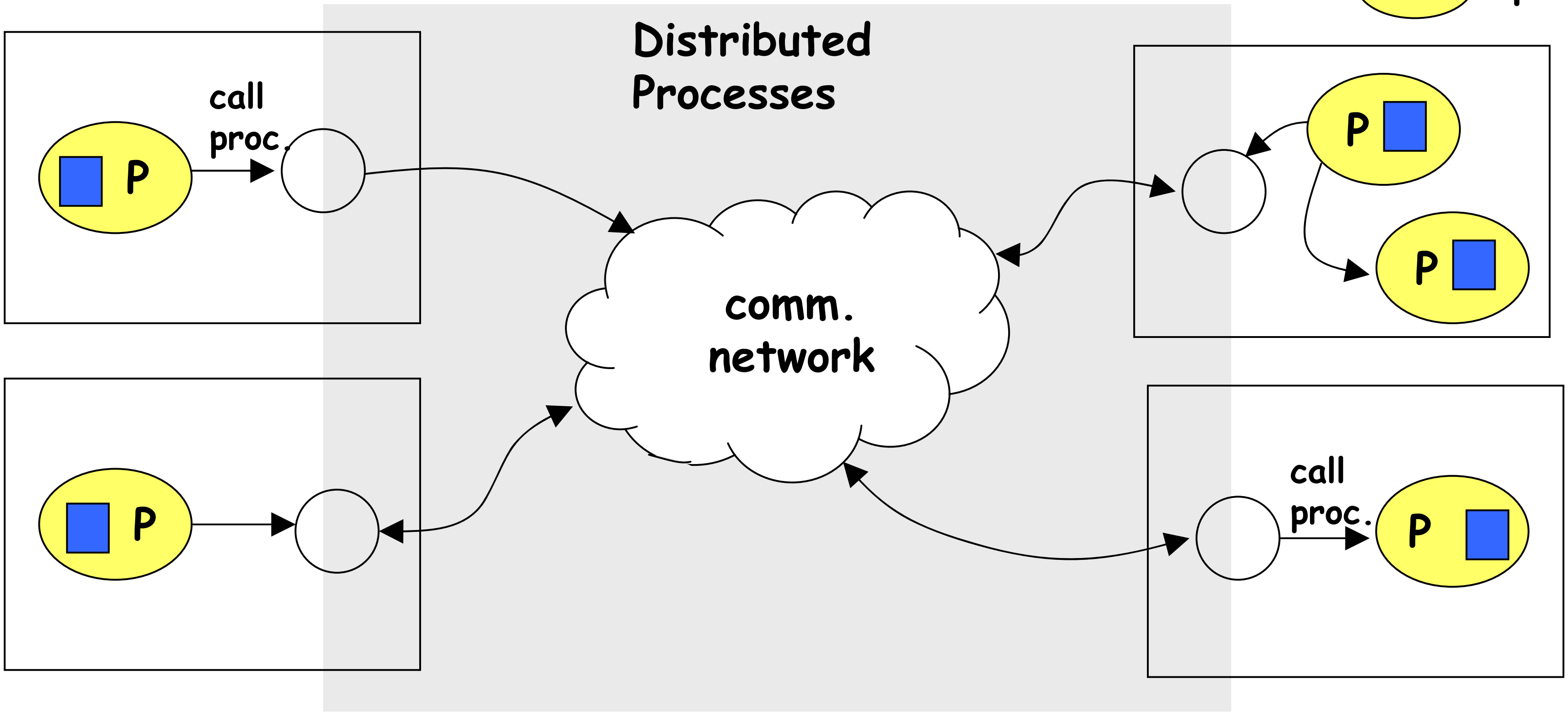
- No explicit communication by messages is needed.
- Programs which run on a single computer will run on a distributed system.
- Multiple computational resources increase the perfomance.

Principles of distributed computations

Function shipping initiates computations in a remote processing entity.
Example: Remote Procedure call.

🟦 memory

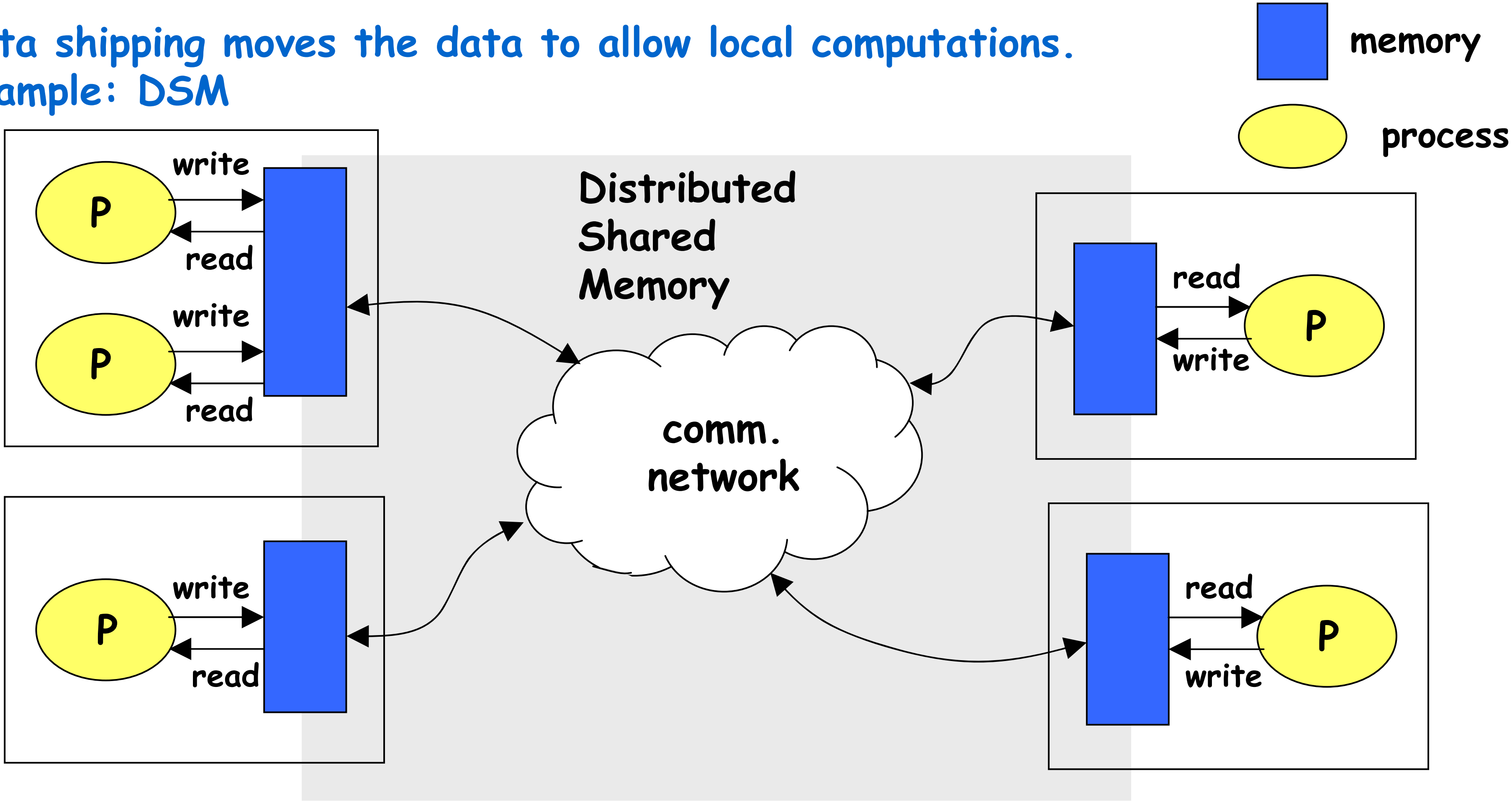
🟡 process



Problem: computation bottlenecks, more complex programming model, references.

Principles of distributed computations

Data shipping moves the data to allow local computations.
Example: DSM



Problem: Performance-Consistency Trade-off
in the presence of concurrency and communcation delays

Structure of a DSM

Byte-oriented DSM:

- ➡ closest to main memory model
  - read and write variables
- ➡ distributed demand paging
  - locking of pages (exclusive /shared)
  - problem: false sharing
- ➡ needs sophisticated consistency models
  - related to mutual exclusion in central storage systems

Structure of a DSM

Object-oriented DSM:

- Operation on DSM have higher semantics than read/write
- Access to state variables only via the Object interface
- Semantics is exploited to define consistency rules
  - Examples: Stacks, Double-ended Queues
- Problem of false sharing is reduced

Structure of a DSM

Immutable Data Storage:

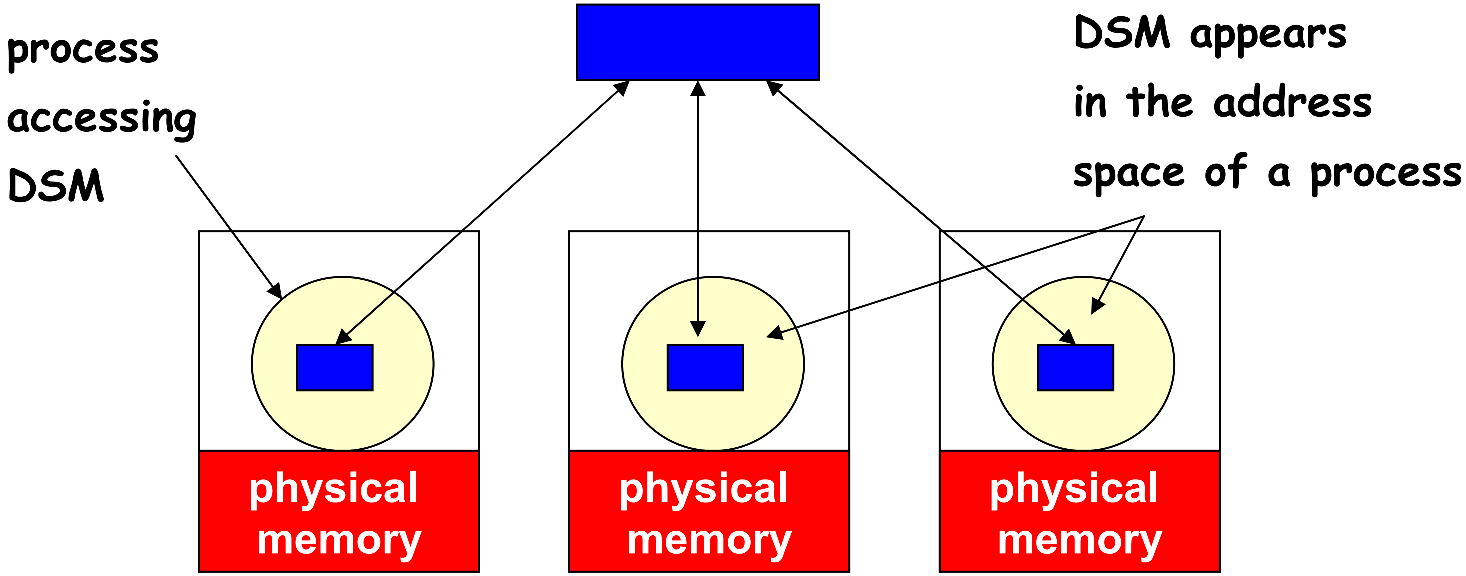
- no write operation
- "out" always adds a data element to the storage
- destructive "in" and non-destructive "read"
- consistency is preserved by ordering accesses
- examples: Linda, JavaSpaces

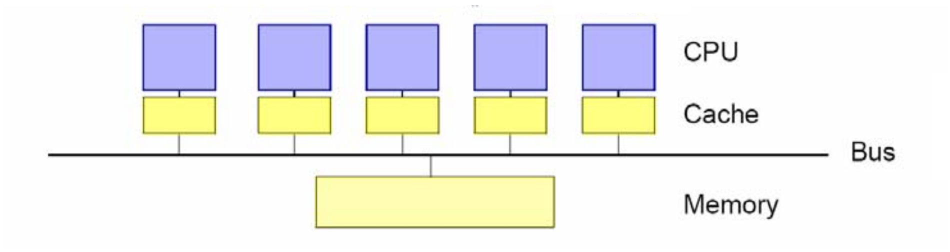
Properties of Storage Systems

	persist- ence	replic. cach.	consist.	example
main memory	no	no	1	RAM
distributed shared memory	no	yes	yes	Munin, Ivy, Midway, ...
file system	yes	no	1	Unix-FS, NTFS
distributed file system	yes	yes	yes	NFS, Andrew, Coda
remote objects	no	no	1	CORBA
persistent object memory	yes	no	1	CORBA Pers.Obj.Service
persistent distr. object mem.	yes	yes	yes	PerDiS, Khanzana, Clouds, Profemo, SpeedOS

Storage abstractions: array of bytes, volatile RAM
persistent file
object (volatile or persistent)

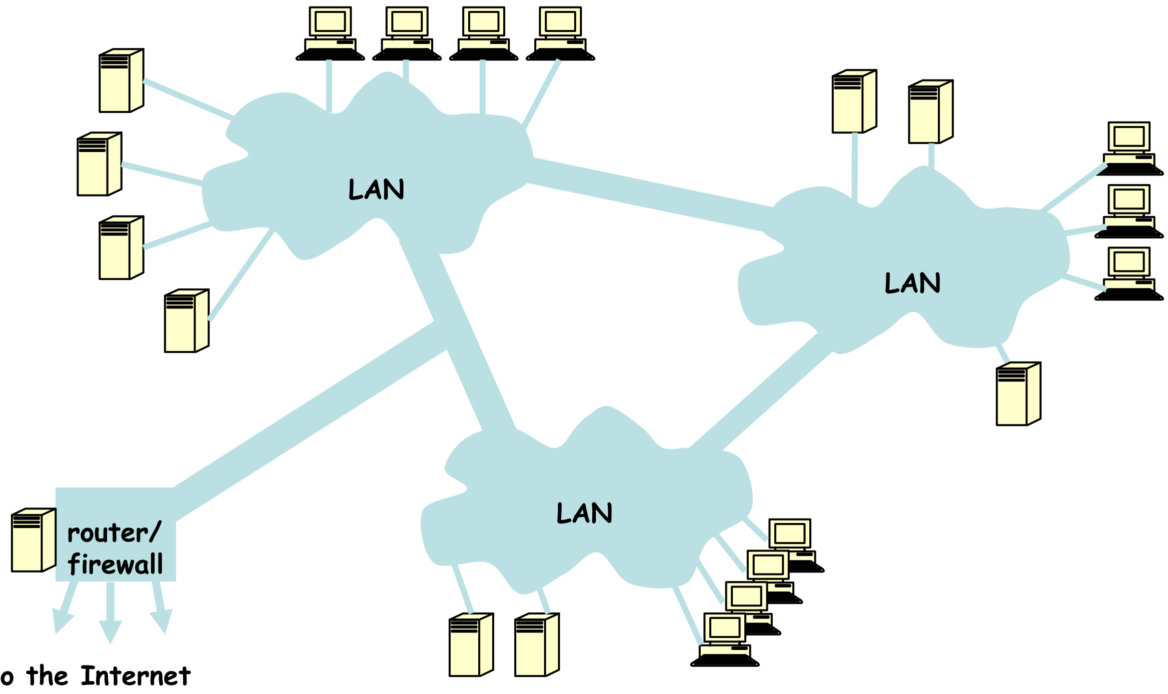
The abstraction of DSM





From a Shared Memory Multiprocessor

to a DSM



?

- can we expect the same transparency?
- what are the trade-offs between ease of use and efficiency?

Accessing shared variables in DSM

process 1

```
br:= b;
ar:= a
if (ar ≥ br) then
        print ("OK");
```

process 2

```
a = a + 1;
b = b + 1;
```

valid value combinations:

ar=0, br=0

ar=1, br=0

ar=1, br=1

due to message delay

it could happen that : ar=0, br=1

Is this considered consistent?

Consistency Models

The characterization of a Consistency Model is the answer of the question:

What result can you expect from a read operation on a DSM with respect to (previous) write operation?

very weak

consistency models

problem-oriented shared memory

....

entry

release

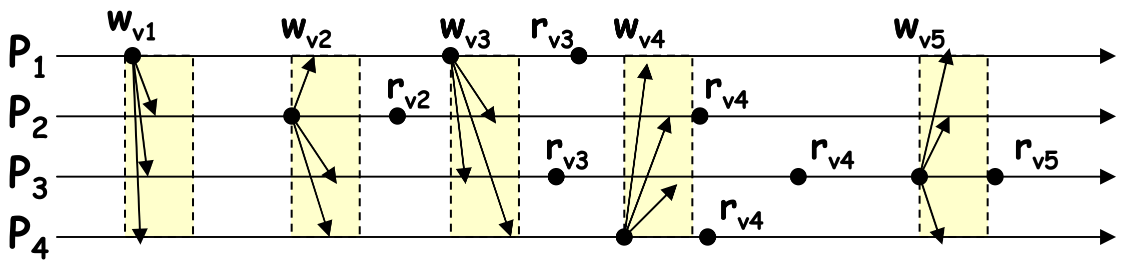
sequential

atomic

very strong

The most actual value which results from the last write operation on the time line.

Consistency Models



Atomically consistent

Strong consistency models:

All write operations are totally ordered and read operations always return the last value written into memory.

Atomic consistency: Write operations in real-time order. All readers see the write operations in the order they were issued on the time-line.

Sequential consistency: Write operations in sequential order i.e. all readers see the write operations (on all memory objects) in the same order.

P. Veríssimo, L. Rodrigues: Distributed Systems for System Architects, Kluwer 2001

Consistency Models

Atomic Consistency is not possible in a distributed system.

Sequential Consistency can be expressed as follows:

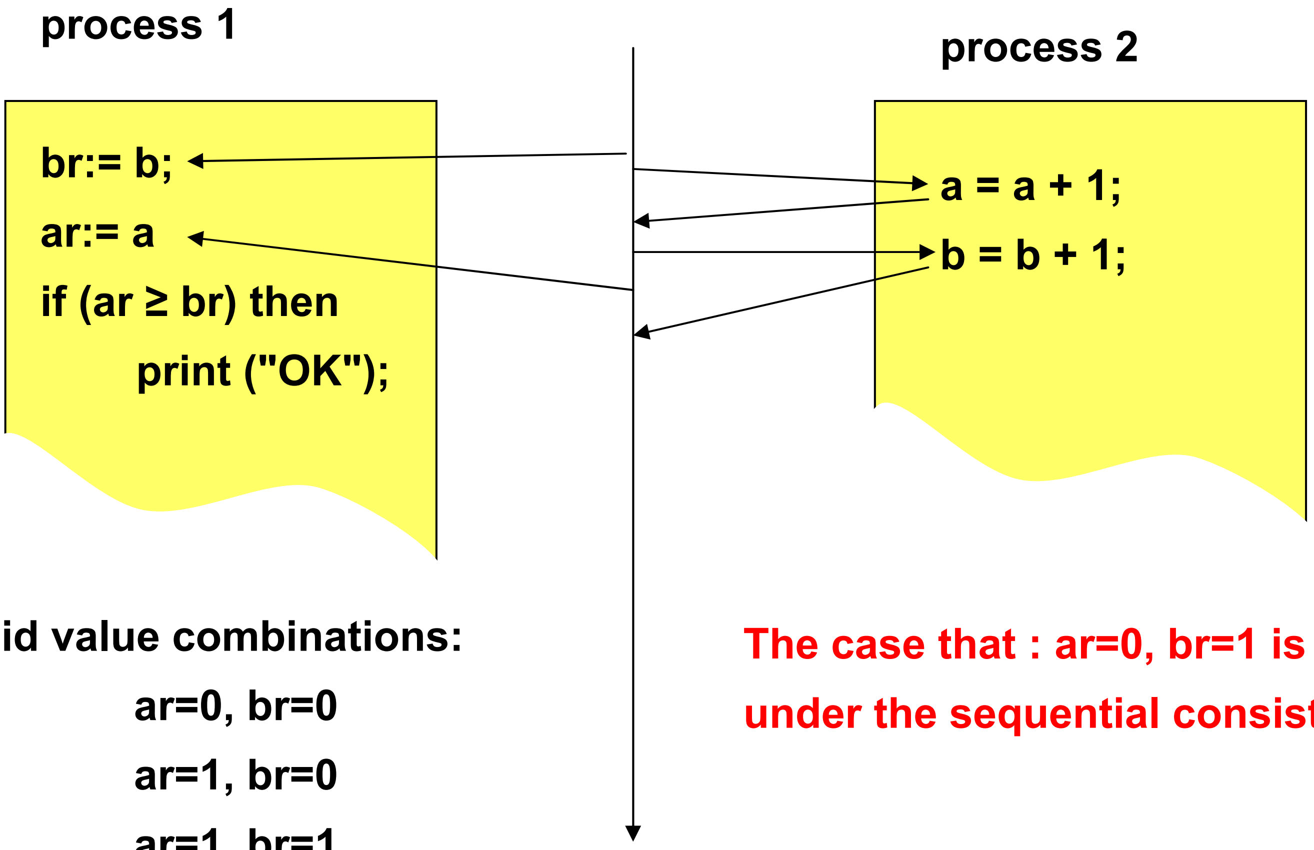
There is a virtual interleaving fo read- and write-operations of all processes on a single virtual memory image. Sequentially consistency is given if:

1.) The program sequence of every individual processor is maintained in the interleaving.

2.) Every process reads the value which was most recently written in the interleaving of operations.

3.) The memory operations for the entire DSM have to be considered - not only the operations on a single memory location.

Interleaving Accesses to shared variables in a DSM



process 1

```
br:= b;
ar:= a
if (ar ≥ br) then
        print ("OK");
```

process 2

```
a = a + 1;
b = b + 1;
```

valid value combinations:

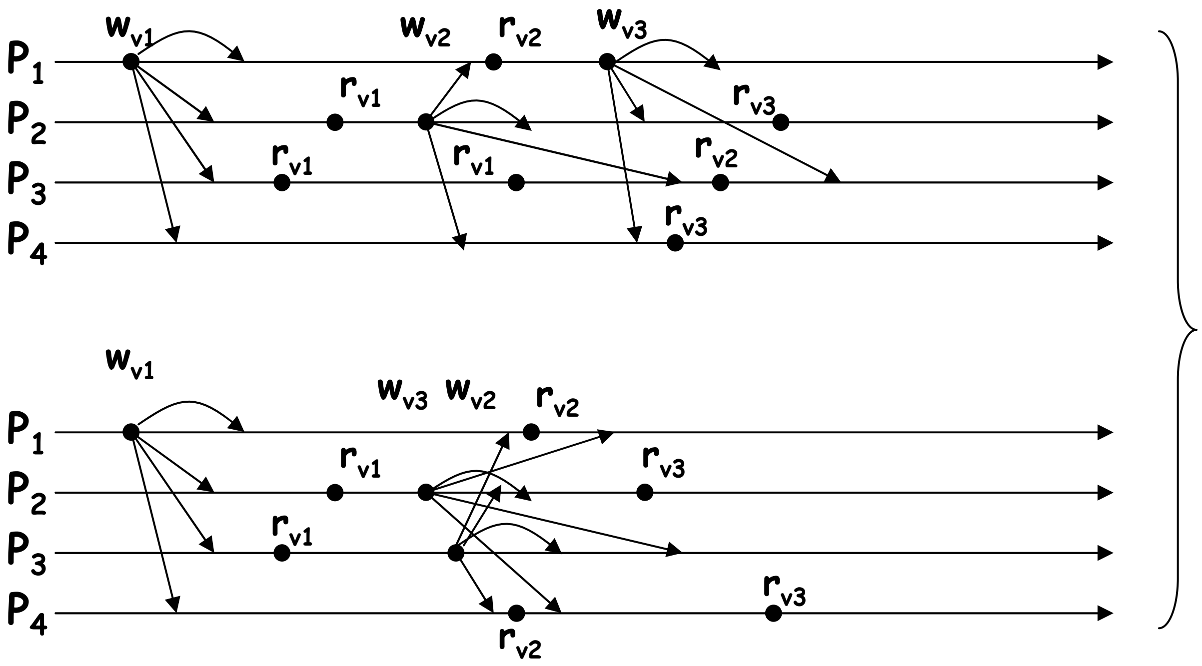
ar=0, br=0

ar=1, br=0

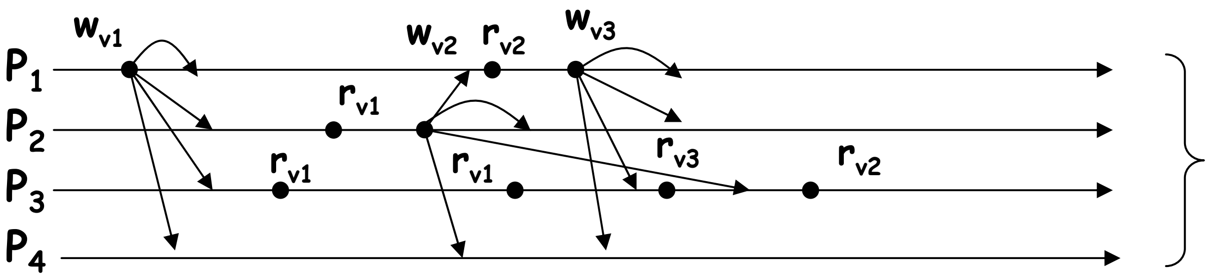
ar=1, br=1

The case that : ar=0, br=1 is excluded under the sequential consistency model.

Consistency Models

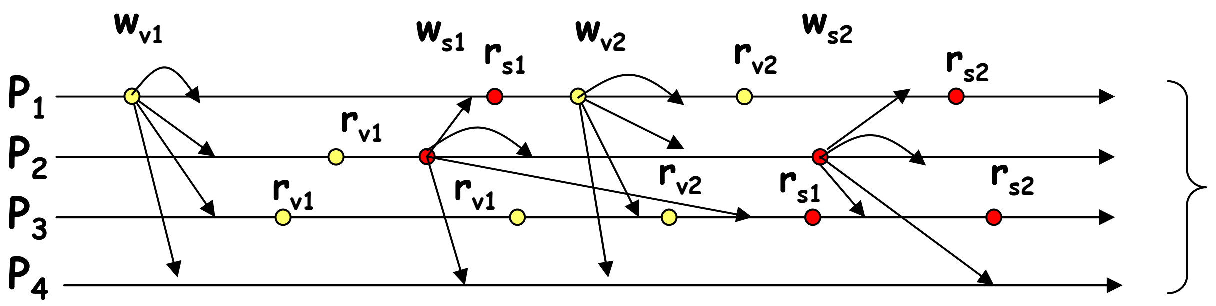


sequentially consistent



not sequentially consistent

Consistency Models

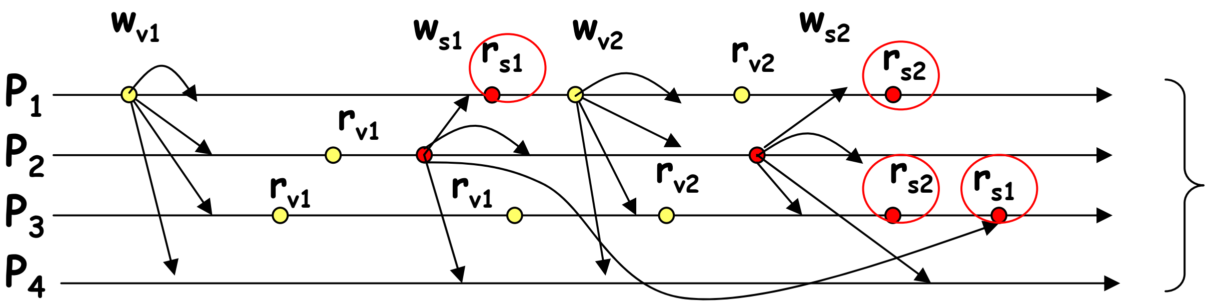


coherent

but not

consistent

Coherency: Sequential consistency for a single memory location.

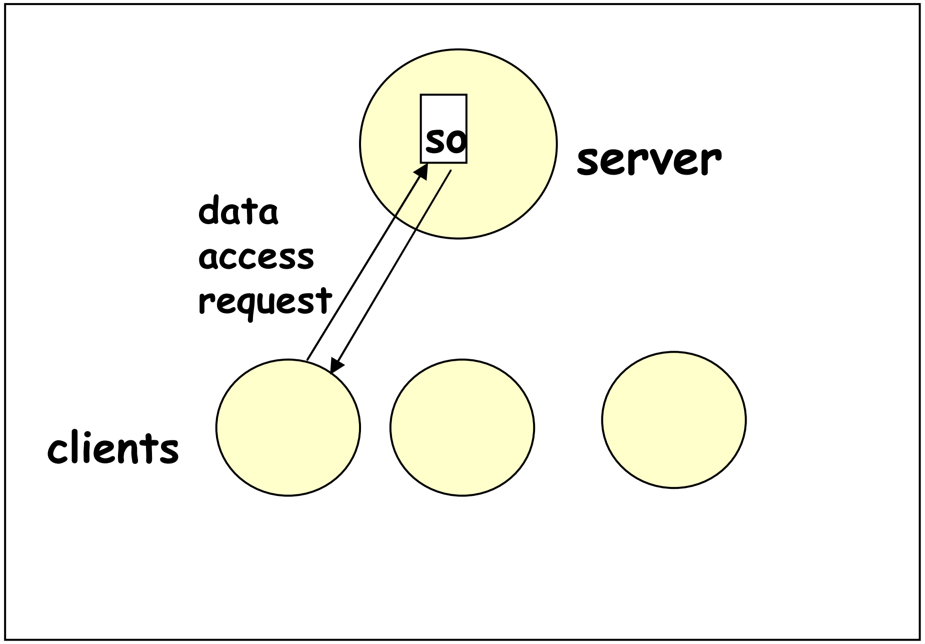


neither coherent

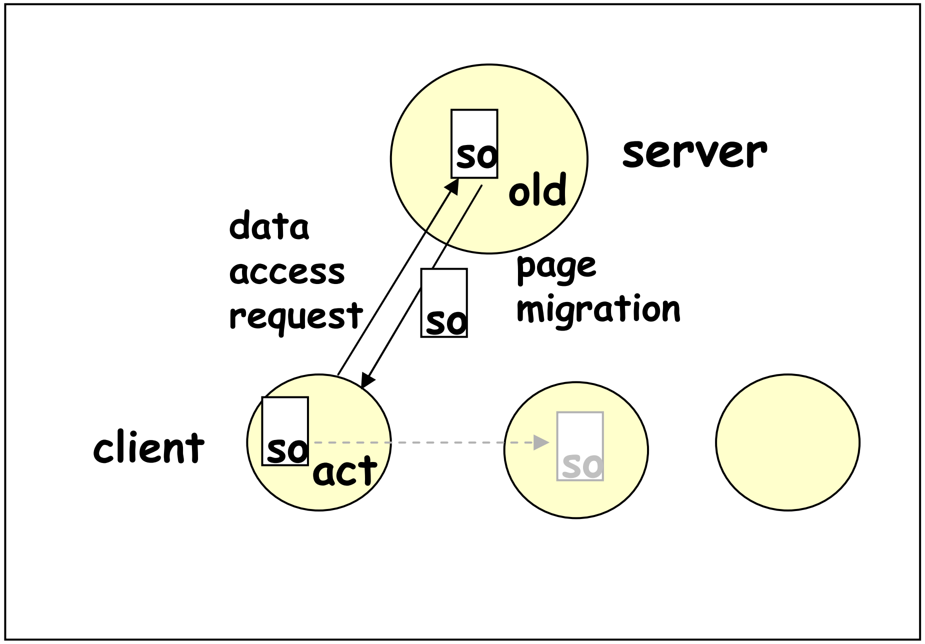
nor consistent

Implementation options

centralized function shipping



centralized data shipping



so: storage object

actual so may be migrated between clients (who provides location information?)

so always is in one place --> no consistency problems for the price of low concurrency.

Update options

Assumption:	Copies of DSM memory images are distributed over multiple process address spaces on multiple nodes.

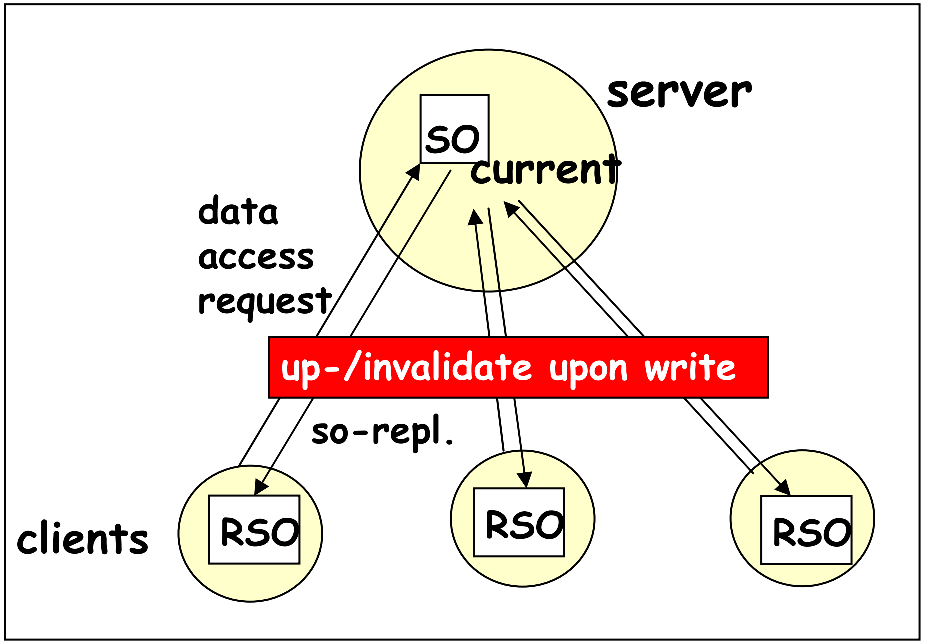
Concurrent reads: no problem

Concurrent writes:

write update:	all copies are updated with the new value
write invalidate:	all copies are invalidated. New reads require to request a new copy of the data items.

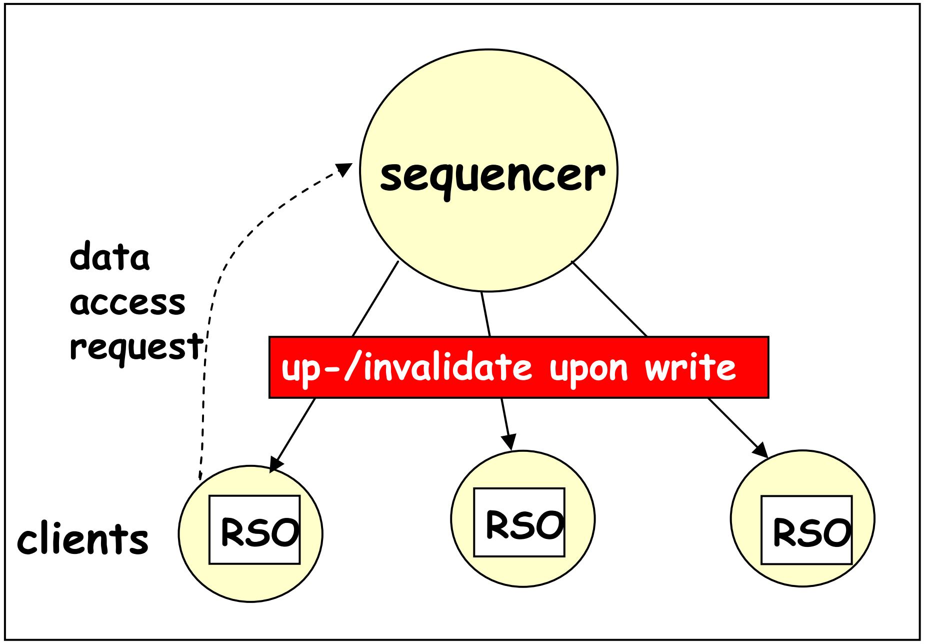
Implementation options

centralized SO replication (read-only)



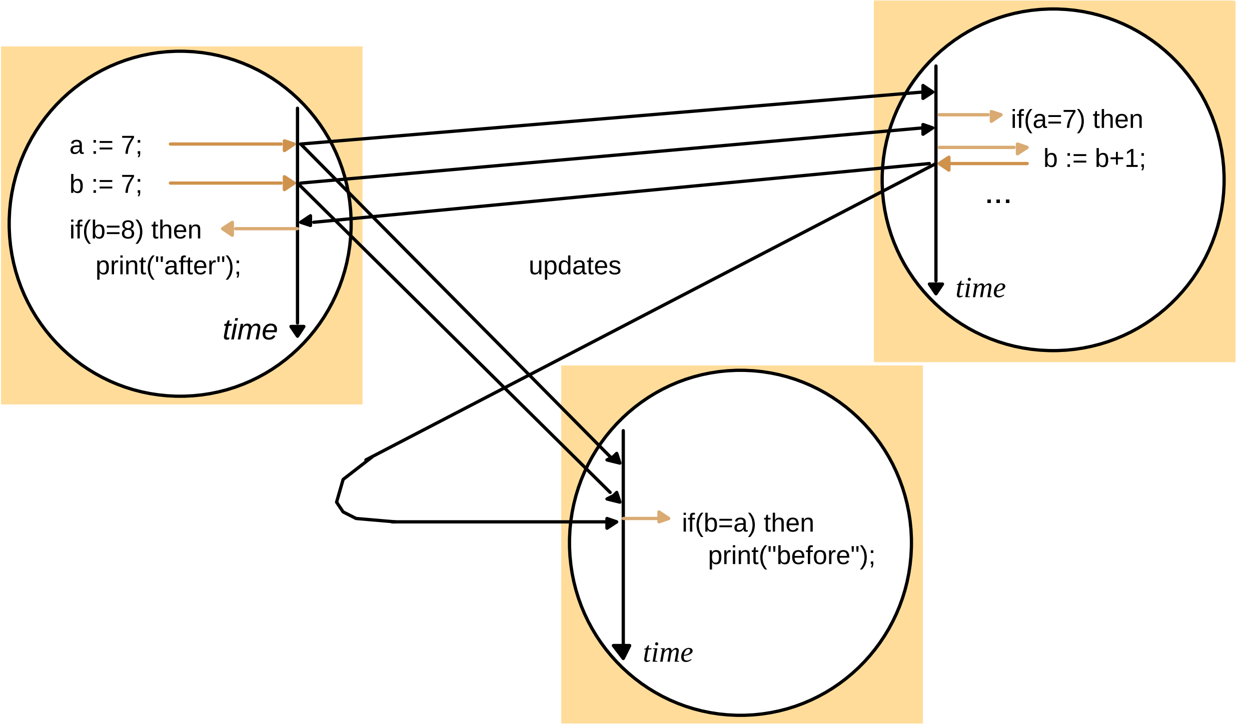
writer only receives a copy of SO iff all RSOs (Replicated Storage objects) are invalidated.

distributed SO replication (read-write)



Update option: Write-update

All changes are multicasted to all nodes which hold the respective memory items.



Problems: Overhead of a totally ordered multicast protocol if sequential consistency is required.

Conclusion: Read operations are cheap, write operations VERY expensive.

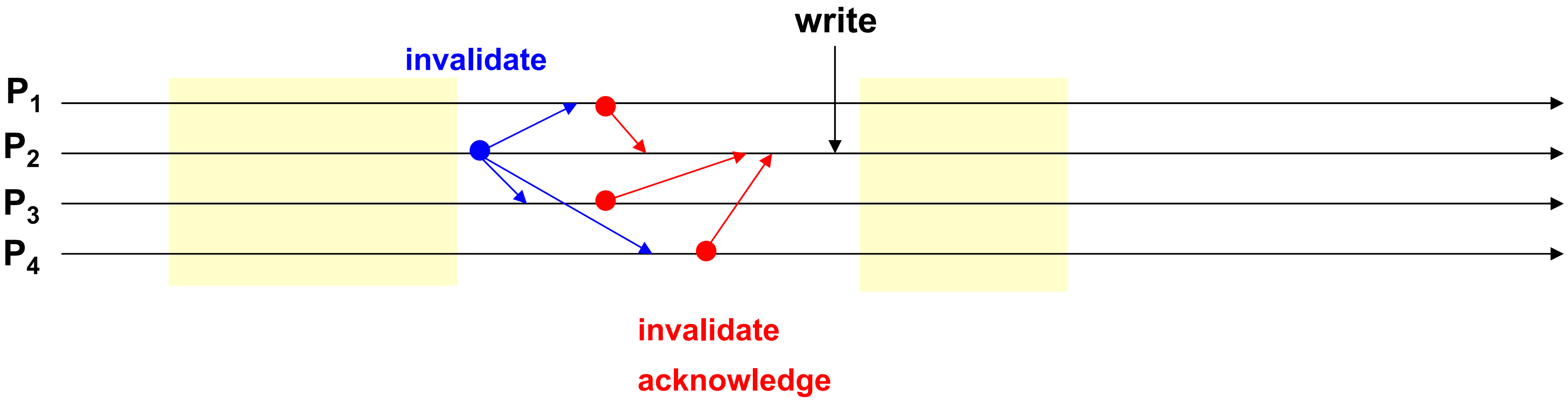
Update option: Write-invalidate

A data item can be either:

- be read by multiple processes
- be written by a single process

Before it can be written, an invalidate is multicasted to all readers.

When having received all invalidation acknowledges, the data is updated.

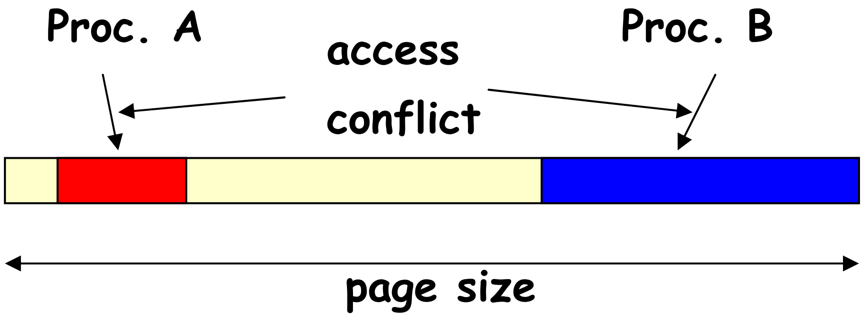


Problems and trade-offs in DSM

Granularity affects:

- amount of data to tranfer
- interference beetween processes
- frequency of requests
- management overhead

➡ **False Sharing**



Problems and trade-offs in DSM

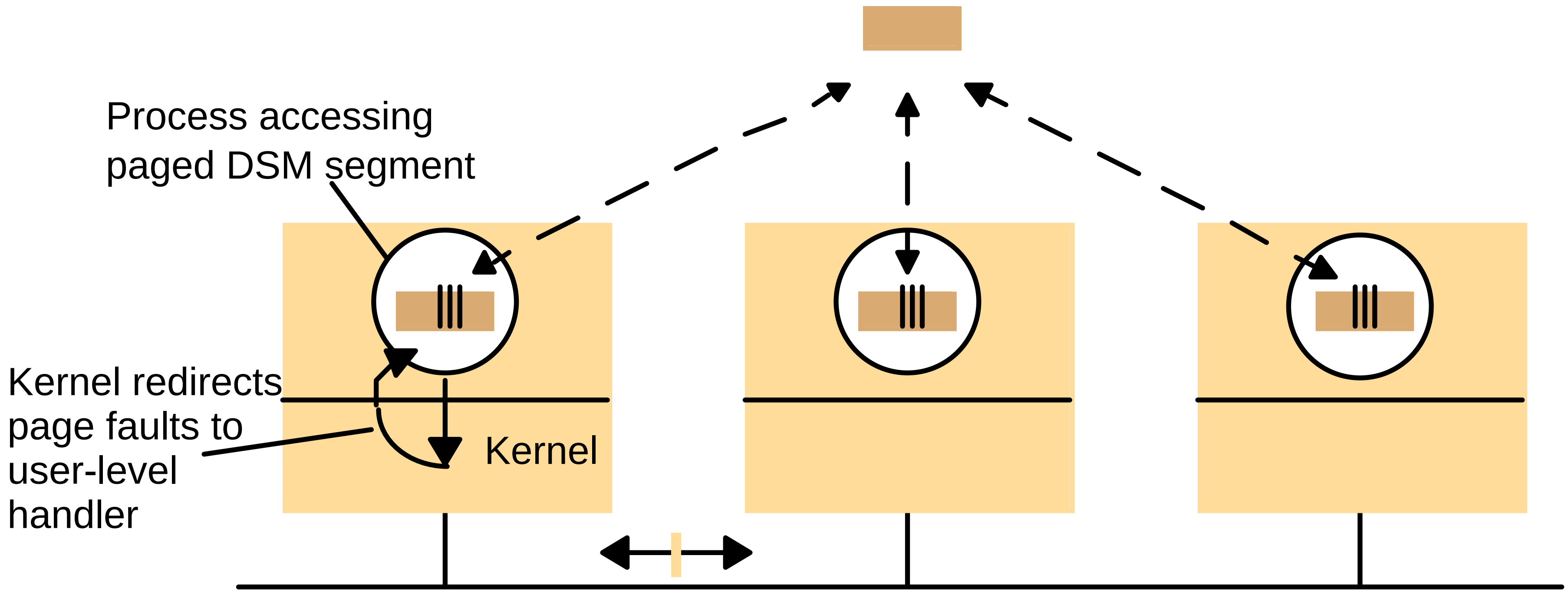
➡ **Trashing:**

- multiple processes access the same data object
- write invalidate
- may be because of real sharing
- may be because of false sharing

define minimum hold time for a data object - Mirage

define usage pattern with appropriate update options - Munin

Implementation Issues: sequential consistency in Ivy



Process accessing paged DSM segment

Kernel redirects page faults to user-level handler

Pages transferred over network

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3

Example: sequential consistency and write update

Problems with write-udate

Assumption: -system exploits hardware page protection,
- rights amy be set to none, read-only or read/write

Algorithm: on write, 1. a page fault is generated, 2. passed to a page-fault handling routine, 3. receives the page and sets appropriate rights, 4. multicasts the update and completes the write operation.

Problem: next write does not generate a page fault! How to detect that a multicast has to be performed?

Solution: put process into trace mode and generate a trace exception. Exception puts page resets the write access rigth. VERY EXPENSIVE !

Optimization: Buffering of write operations and multiple write accesses to a page.

write invalidate

- uses page protection information to enforce consistency:

- possible combinations of read and write rights

  single writer - no other process will have access

  multiple readers - no writer

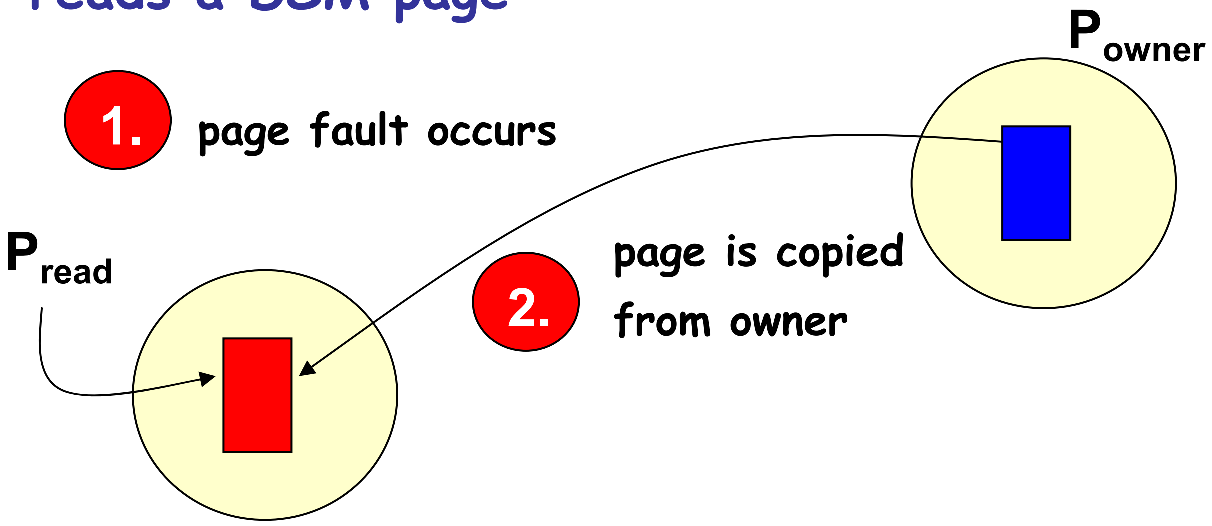
- owner of page (*owner (p)*) holds the most recent version of the page:

  - the (single) writer
  - one of the readers

- the set of processes which hold a copy is called the "copy set" (*copyset (p)*)

copyset and owner transfer during write invalidate

P reads a DSM page

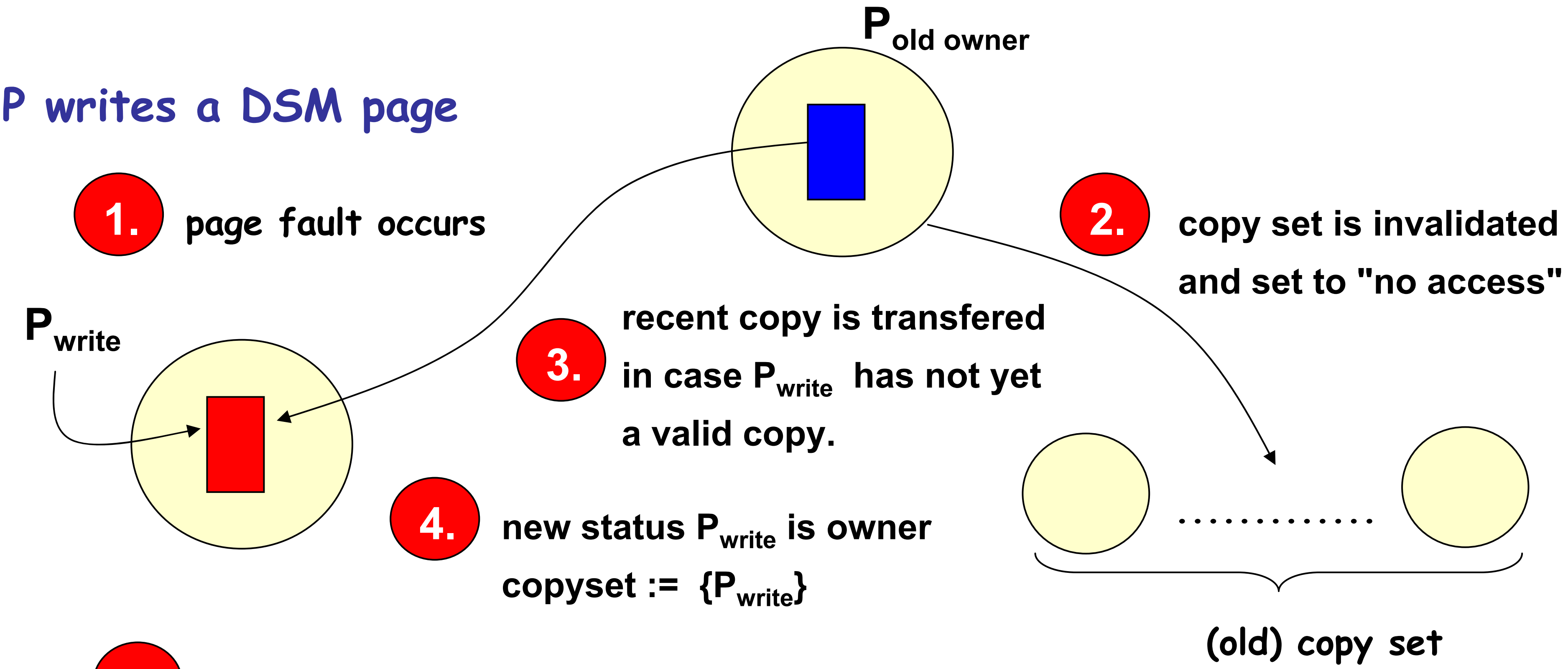


if P_{owner} was writer it retains a read right and remains owner (because this is the most recent copy). It has to handle subsequent requests.

3. copyset := copyset $\cup$ $\{P_{read}\}$

copyset and owner transfer during write invalidate

P writes a DSM page



1. page fault occurs

2. copy set is invalidated and set to "no access"

3. recent copy is transfered in case P_{write} has not yet a valid copy.

4. new status P_{write} is owner
copyset := $\{P_{\text{write}}\}$

5. DSM runtime system maps page in the address space of P_{write} and resumes at the instruction causing the page fault

Issues to solve for implementing DSM

Problems:

1.) Finding the owner of a page

2. Determining the copy set and where it is stored

Solutions:

1.) Central Manager

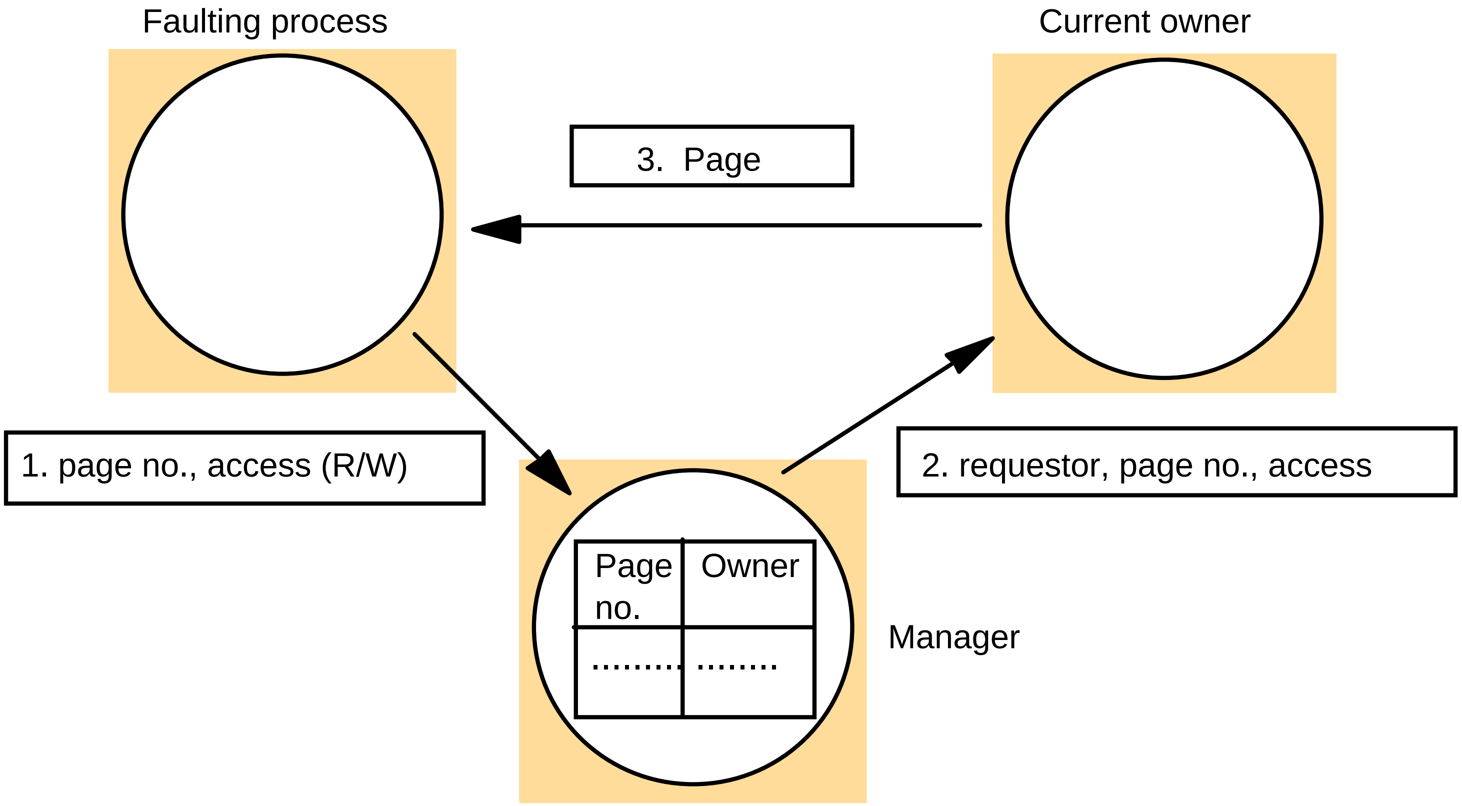
2.) Multicast (totally ordered)

3.) Dynamically Distributed Manager

  - -build a chain of hints
  - -update the hints dynamically

Finding the owner of a page

Central manager approach



Faulting process

Current owner

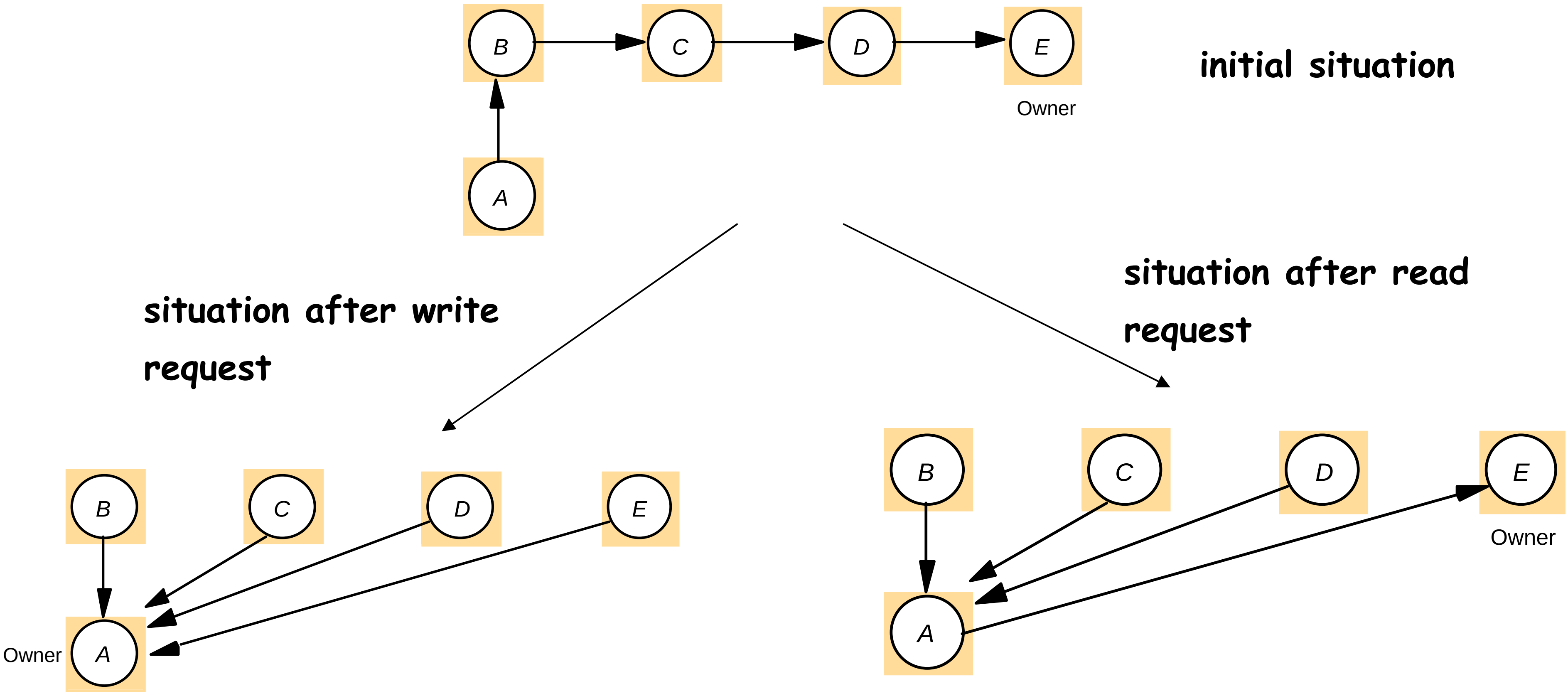
3. Page

1. page no., access (R/W)

2. requestor, page no., access

Finding the owner of a page

Dynamic distributed manager approach



initial situation

situation after write request

situation after read request

Beyond sequential consistency

Approaches to increase efficiency and cost effectiveness of DSM:

- Exploit knowledge of what is shared data and what is not.
  Accesses to shared data have to be synchronized

- Identify a priori known characteristic access pattern.
  Classify data items accordingly and adapt consistency overhead.

Release consistency

Observation:

accesses of two processes compete if

- they occur concurrently
- at least one is a write access

Conclusion:

- multiple read operations do not compete
- multiple synchronized operations do not compete because concurrency is controlled by synchronization mechanisms.

Approach:

- divide competing accesses in synchronizing and non-synchronizing accesses and let the programmer define critical sections.

Release consistency

```
Process 1:
    acquireLock();                    // enter critical section
    a := a + 1;
    b := b + 1;
    releaseLock();                    // leave critical section
Process 2:
    acquireLock();                    // enter critical section
    print ("The values of a and b are: ", a, b);
    releaseLock();                    // leave critical section
```

Release consistency

Definition:

RC1: before a read or write operation can be executed all preceding acquire-operations have to be performed.

RC2: before a release-operation can be performed for another process, alls read and write operations have to be finished.

RC3: acquire and release operations are sequentially consistent to each other.

Release consistency

By knowing the synchronization constraints when accessing shared variables, a better efficiency can be obtained without sacrificing application consistency.

A correctly instrumented program is unable to distinguish between a release consistent and a sequentially consistent DSM.

Munin - a flexible and adaptable DSM

- allows parameterization of protocols
- distinguishes data types according to synchronization constraints

some Data types:

- read-only
- write shared
- producer-consumer
- migratory
- result
- convntional

some protocol options:

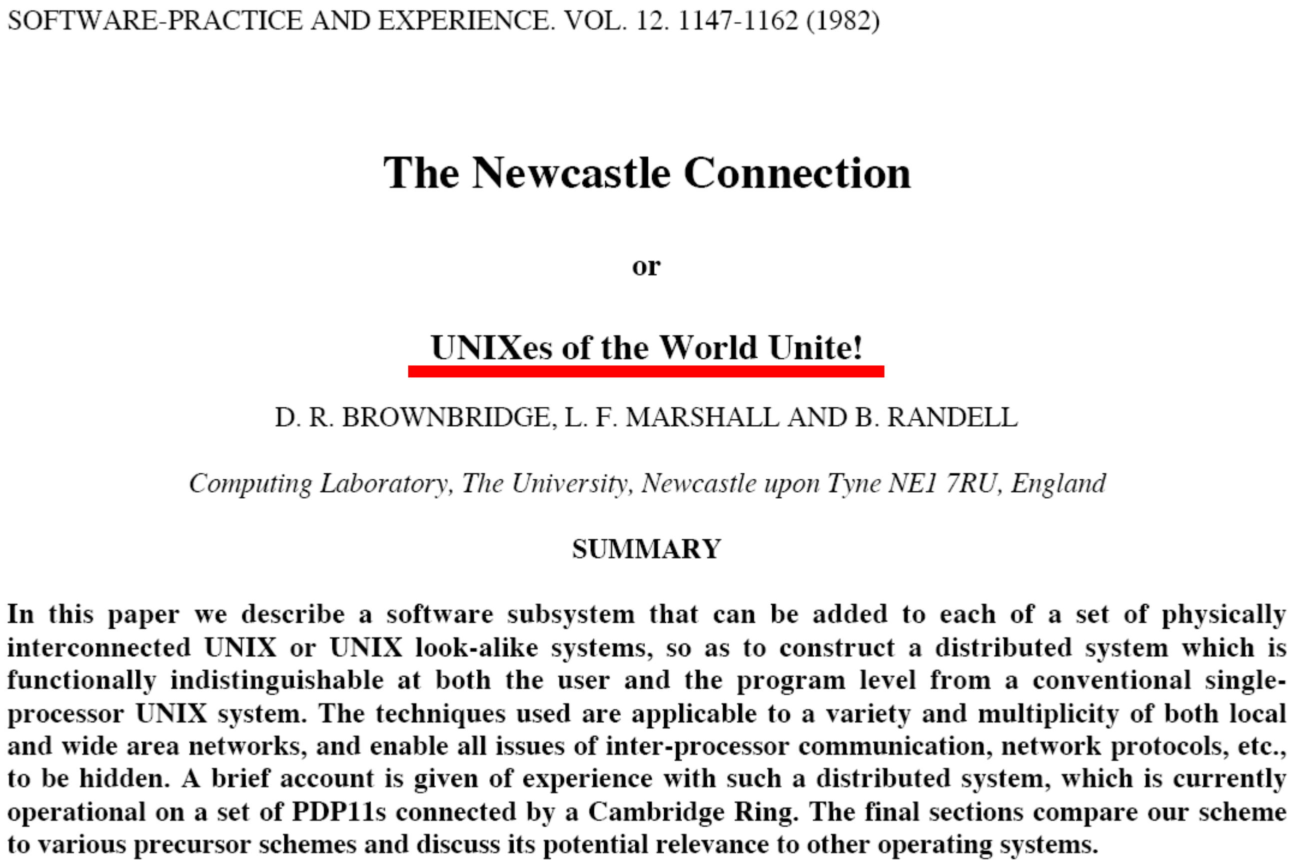
- write update
- write invalidate
- eager or lazy variants
- data element can be modified by multiple writers
  - > needs more semantics
- data item is used by a fixed set of processes

Distributed File Systems

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

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

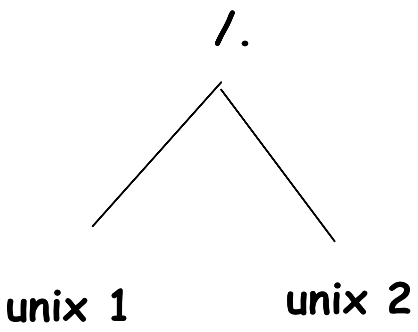
SUMMARY

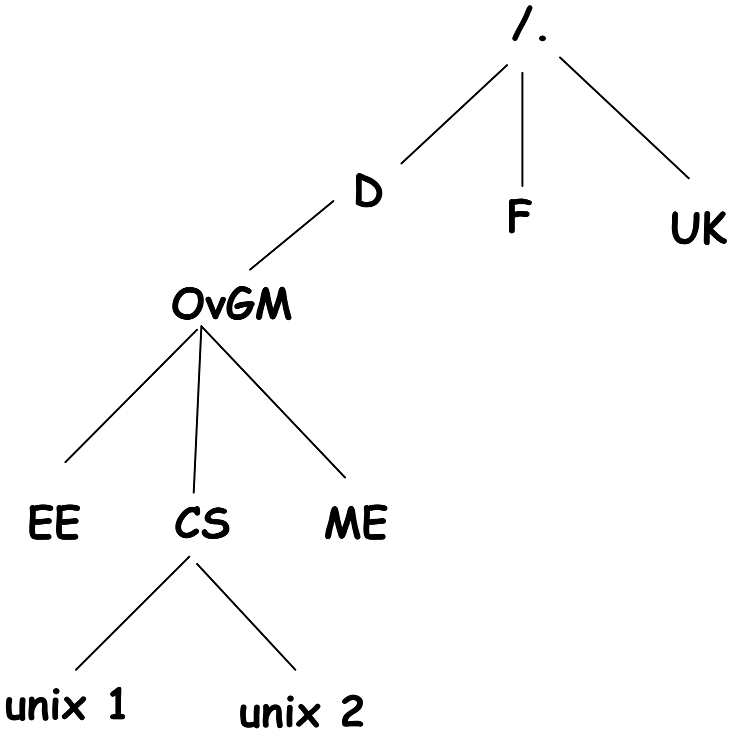
In this paper we describe a software subsystem that can be added to each of a set of physically interconnected UNIX or UNIX look-alike systems, so as to construct a distributed system which is functionally indistinguishable at both the user and the program level from a conventional single-processor UNIX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

Early milestones in distributed file systems

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

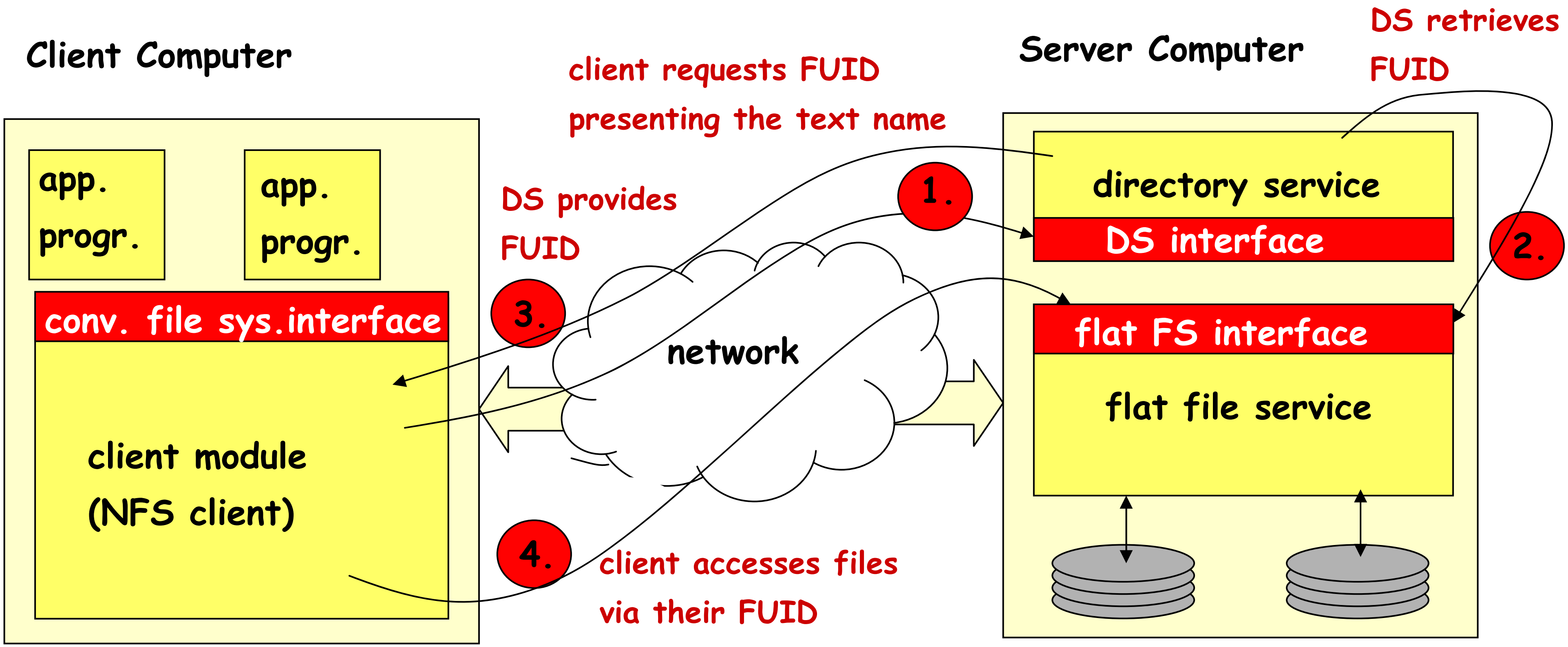
R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh
The Design and Implementation of the SUN Network File System
Proceedings Usenix Conference, Portland, Oregon 1985

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)"

NFS: File Service Architecture



- Client-Server architecture using SUN RPC
- Flat FS uses File UIDs instead of hierarchical path names
- DS associates file text names with file UIDs (FUID)

Flat File Service Operations

Read (FileId, i,n) → *Data* - throws *BadPosition*	**If $l \le i \le Length(File)$: 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 $l \le i \le Length(File)+1$: 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).**

Differences to the Unix File System API

Stateless File Server:

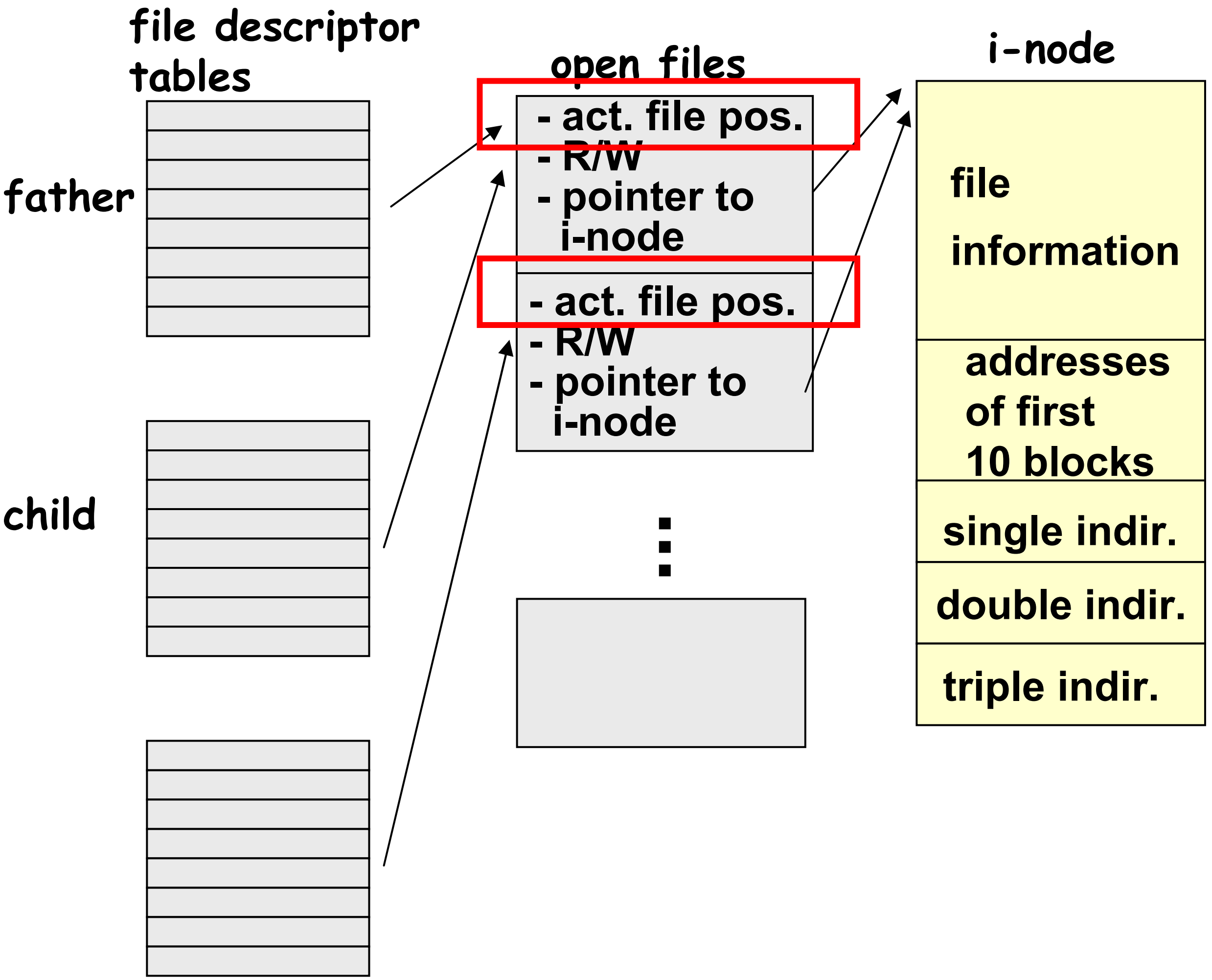
- no state information about open file
- no information about the number and state of clients

➡ 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!

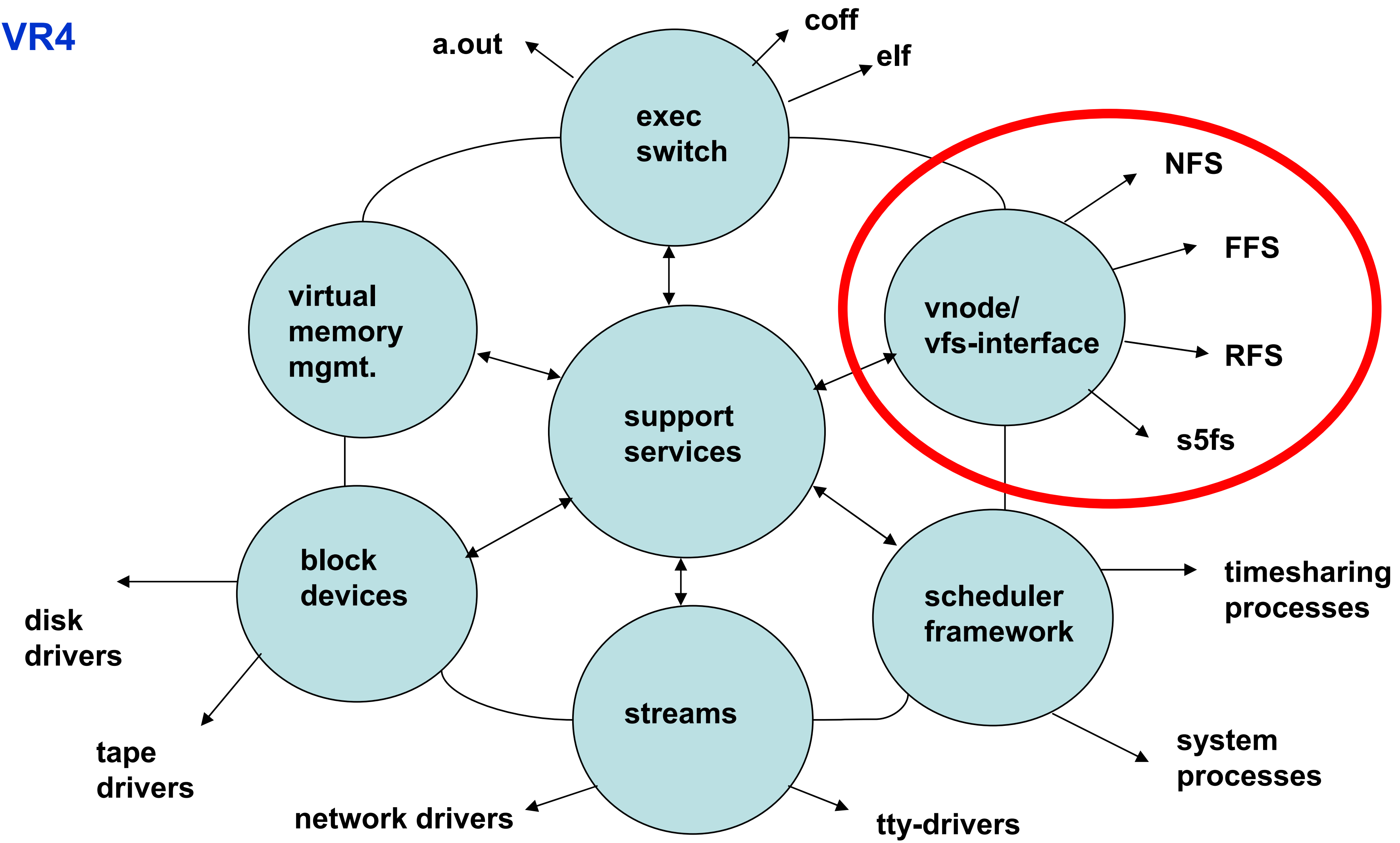
→ read and write NOT idempotent!

Directory Service Operations

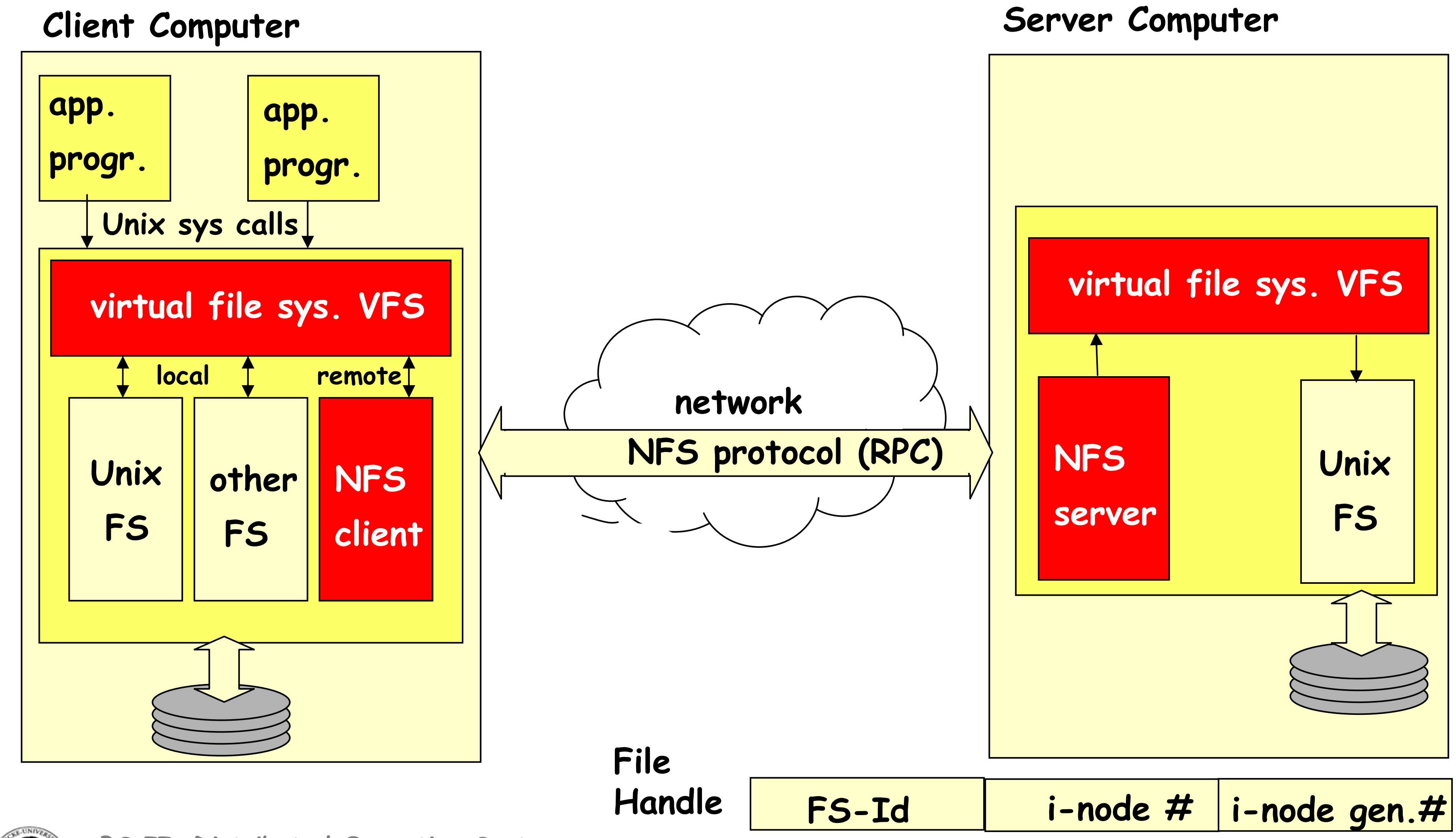
Lookup (Dir, Name) → FileId **- throws** *NotFound*	**Locates the text name in the directory and returns the respective UFID. If** ***Name*** **is not found, an exception is raised.**
AddName (Dir, Name, File) **- throws** *NameDuplicate*	**If** ***Name*** **is not in the directory, adds** ***(Name, 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*****.**

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

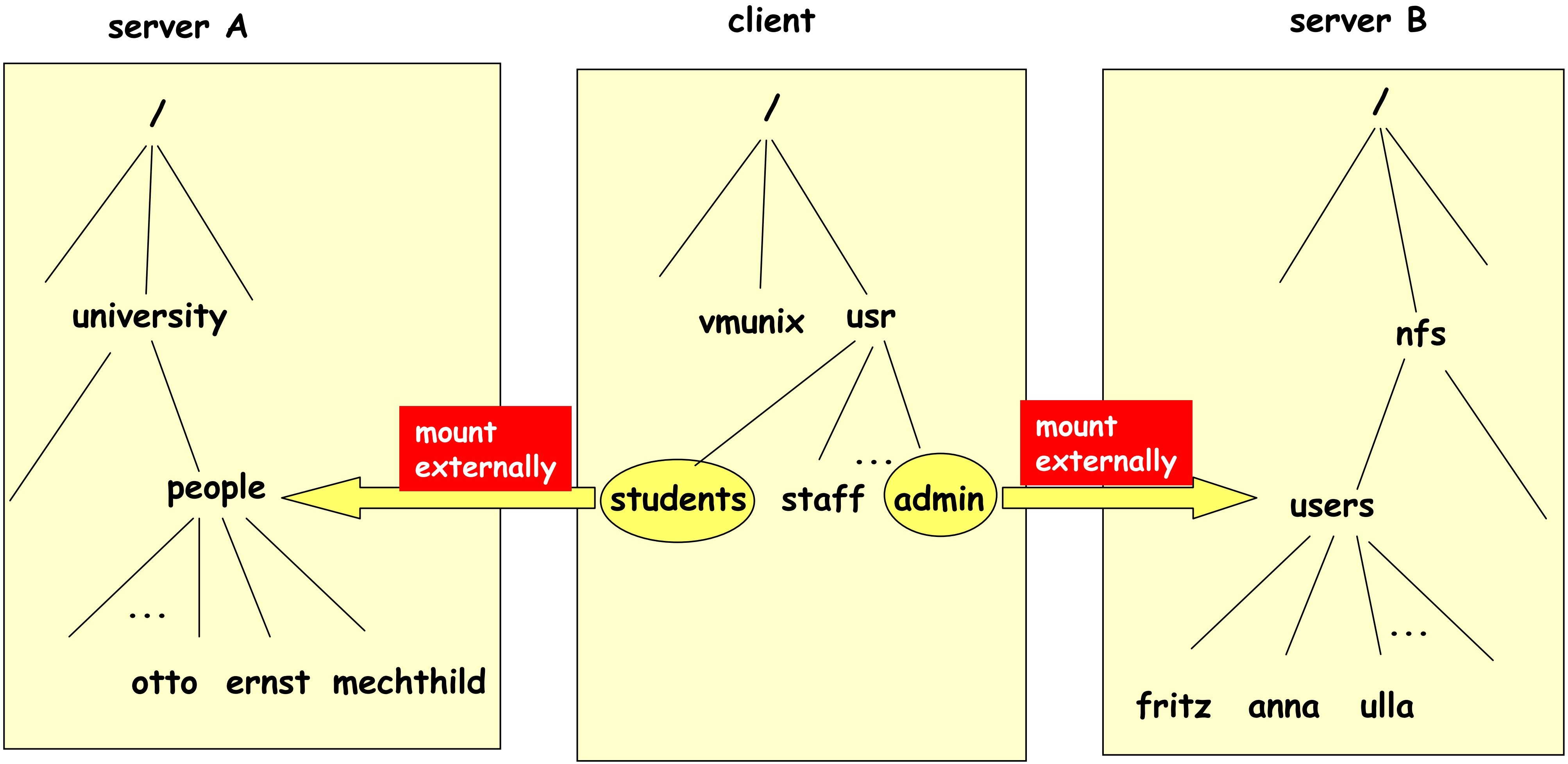
SVR4



SUN NFS Architecture



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.

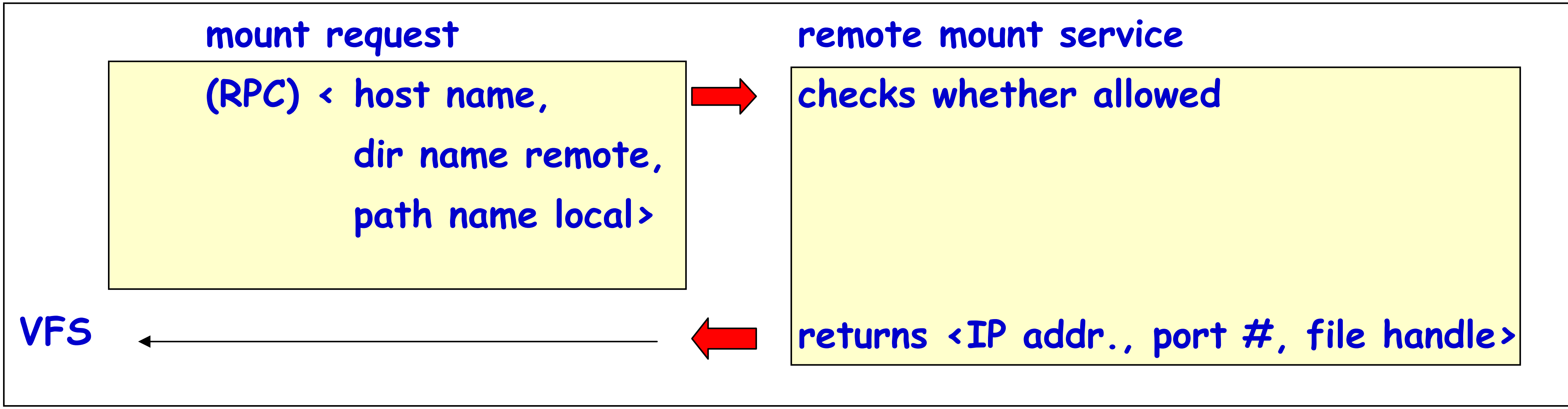
NFS mount service

Mount Service Process: executed on every server

Data Structures:

Server: etc/exports
contains names of local FS which may be mounted ext.

Client: for every file system a list of names of 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

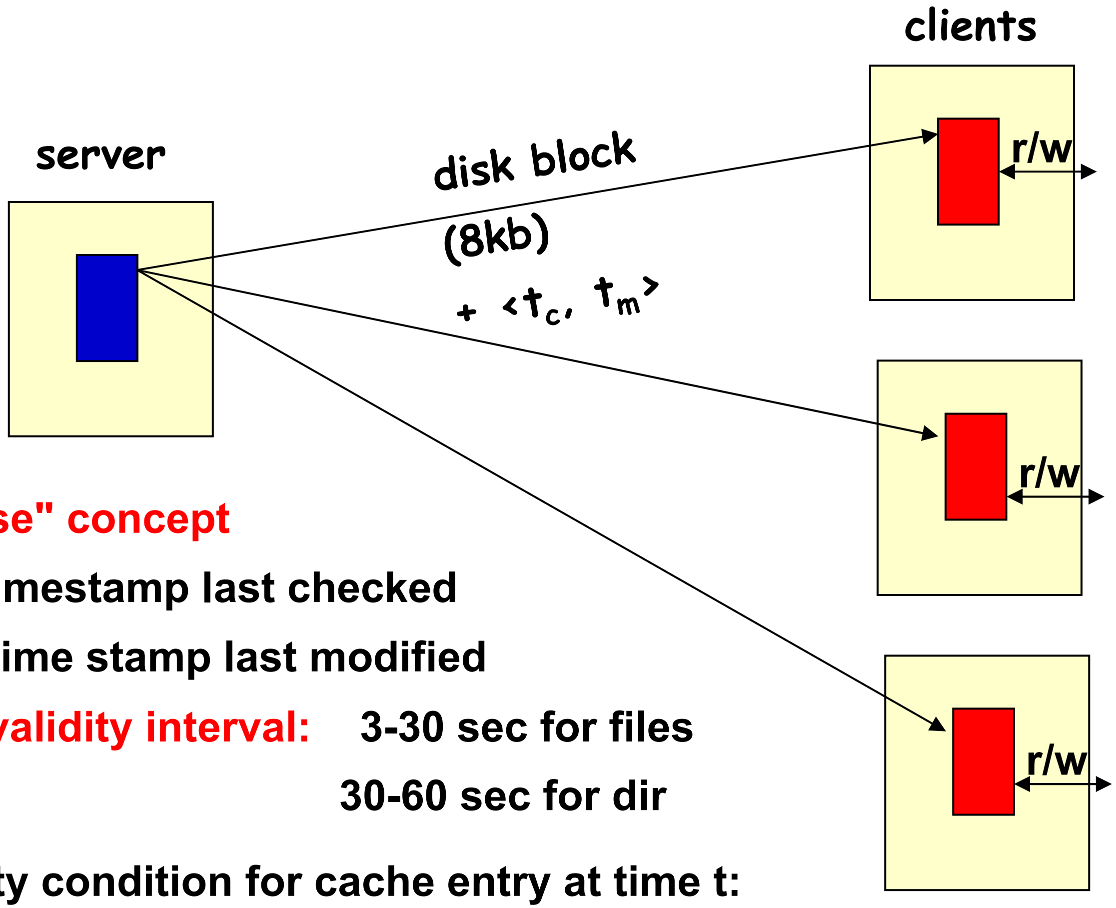
- buffer cache
- read ahead
- delayed write
- sync (periods of 30 sec)

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 only held in the cache. 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:

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

Mechanism only approximates 1-Copy-Consistency !

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	++	

AFS Andrew File System

Scalability as primary design goal.

As much as possible local accesses to files.

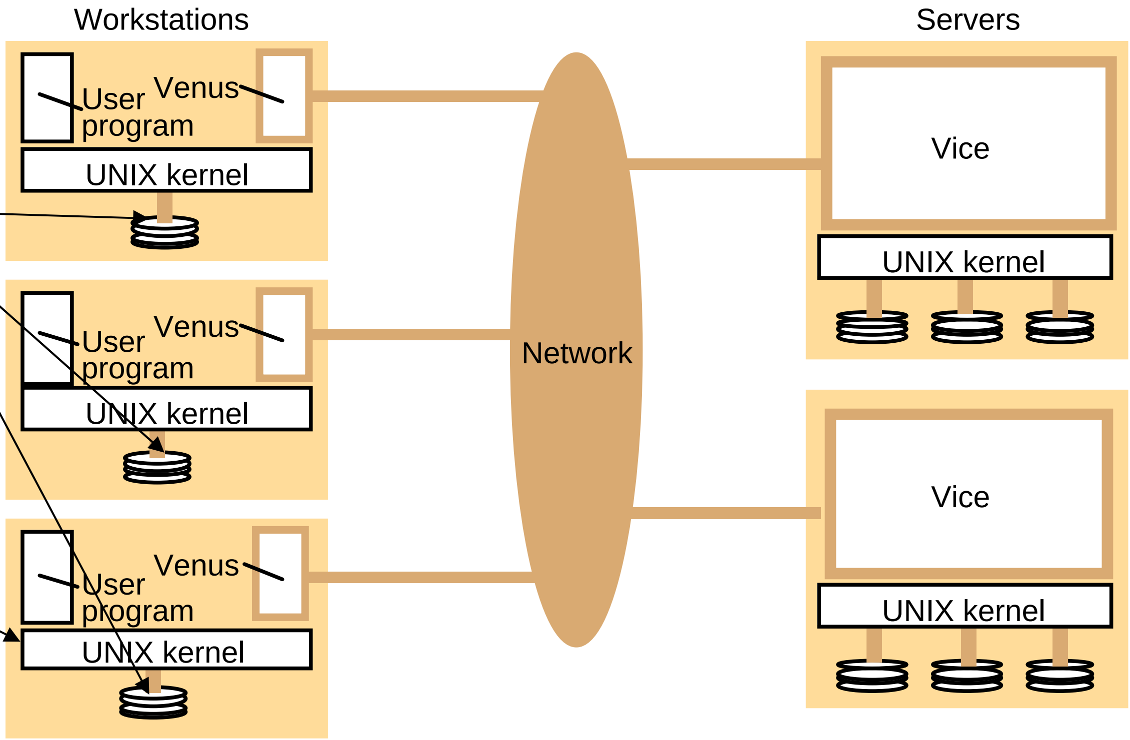
Any accessed file is completely transfered to the client.

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.

The

End