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)

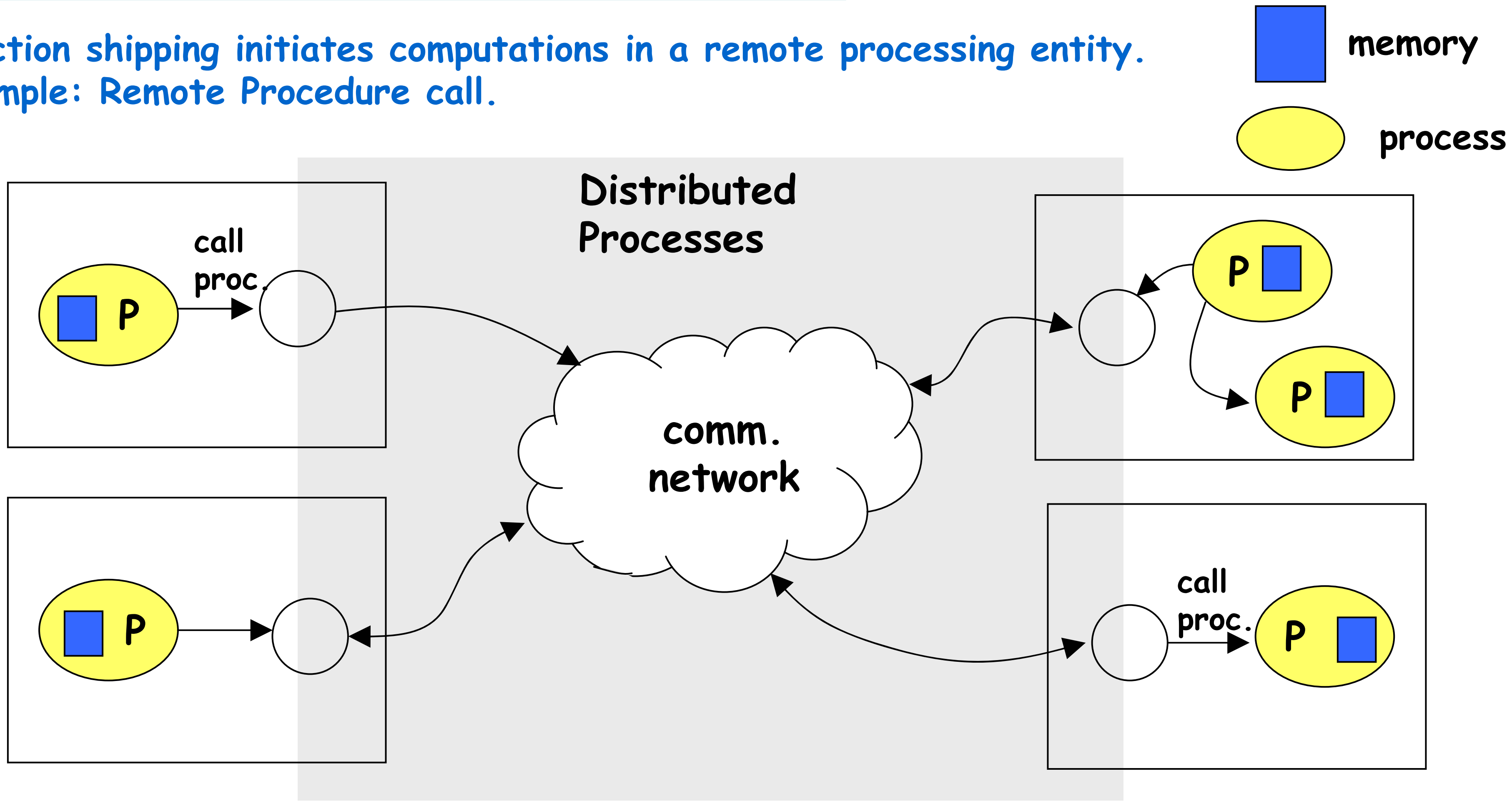
- General problems of distribution
- Examples:NFS, AFS

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

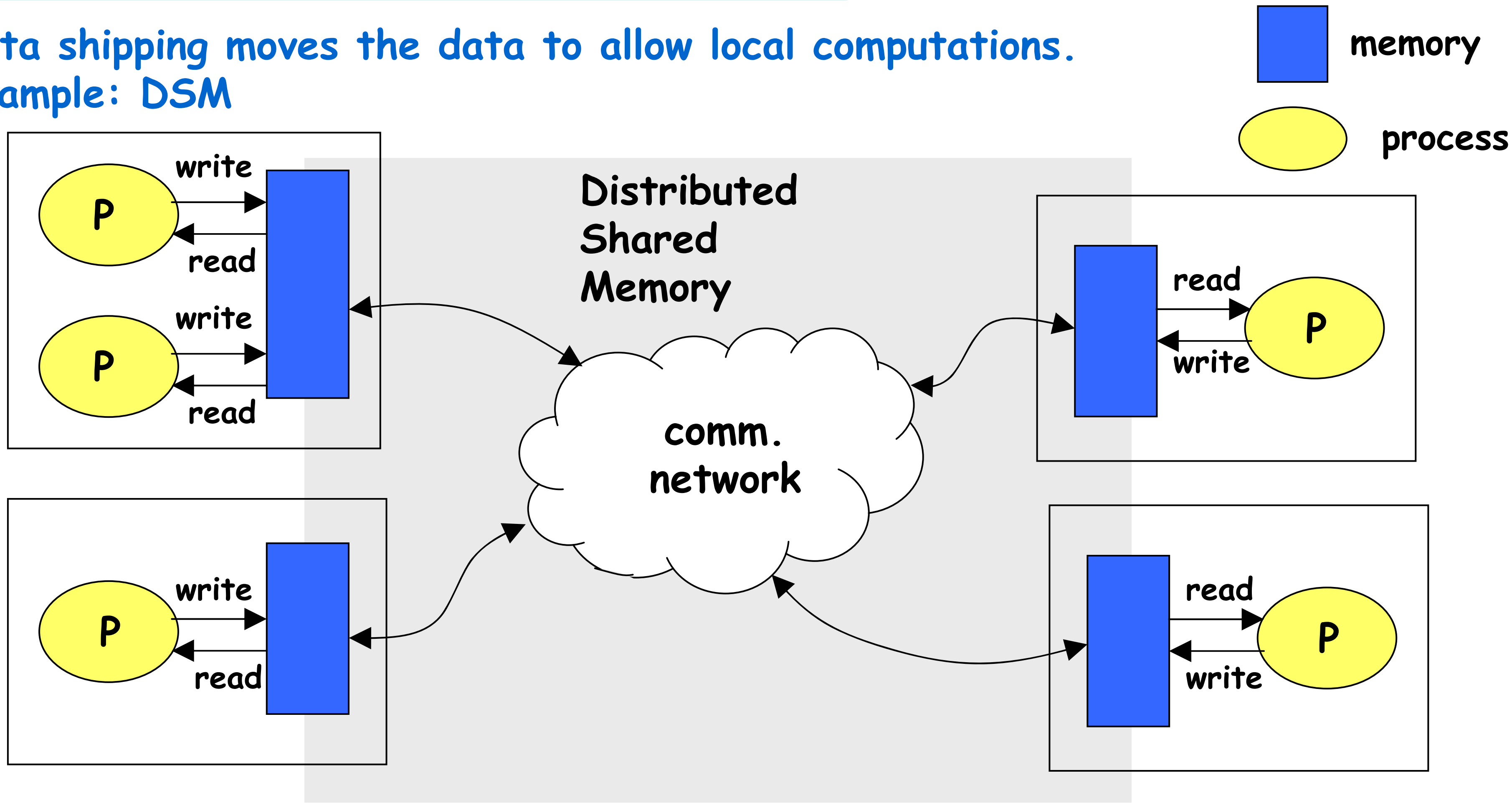
Function shipping initiates computations in a remote processing entity.
Example: Remote Procedure call.



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:

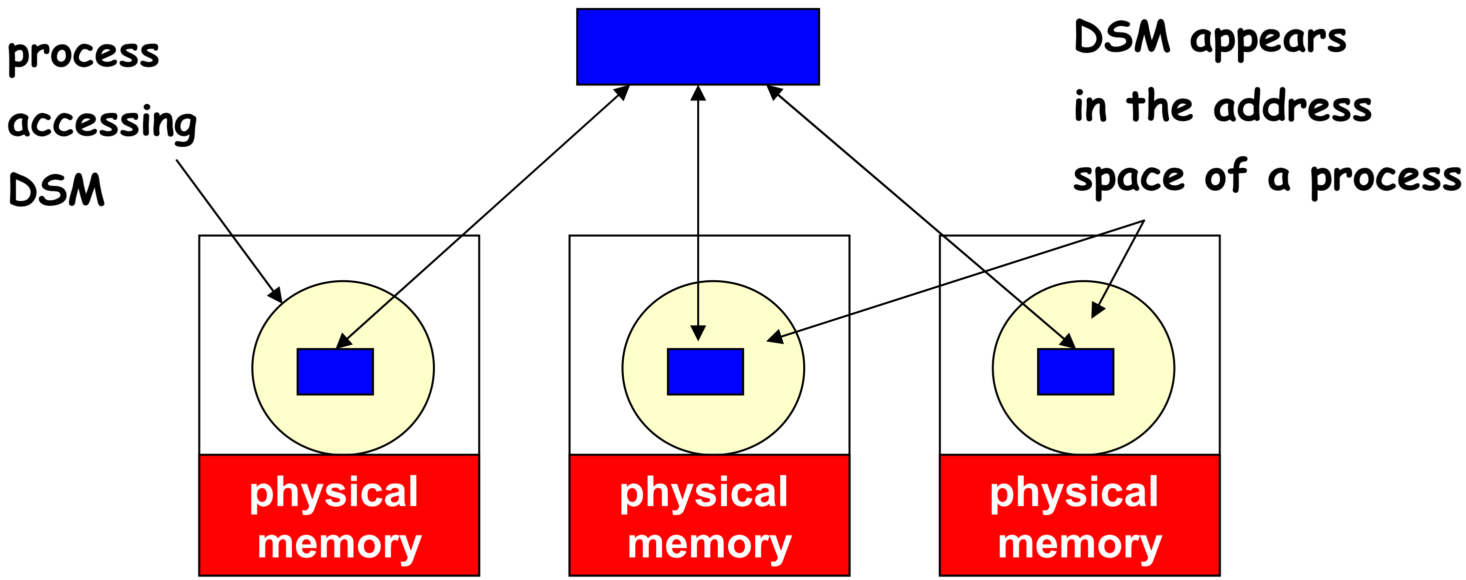
- no write operation
- "out" always adds a data element to the storage
- destructive "in" and non-destructive "read"
- consistency is preserved by ordering accesses
- examples: Linda, JavaSpaces

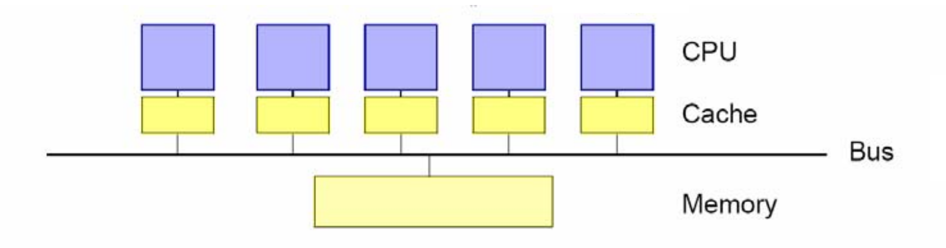
Properties of Storage Systems

	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)

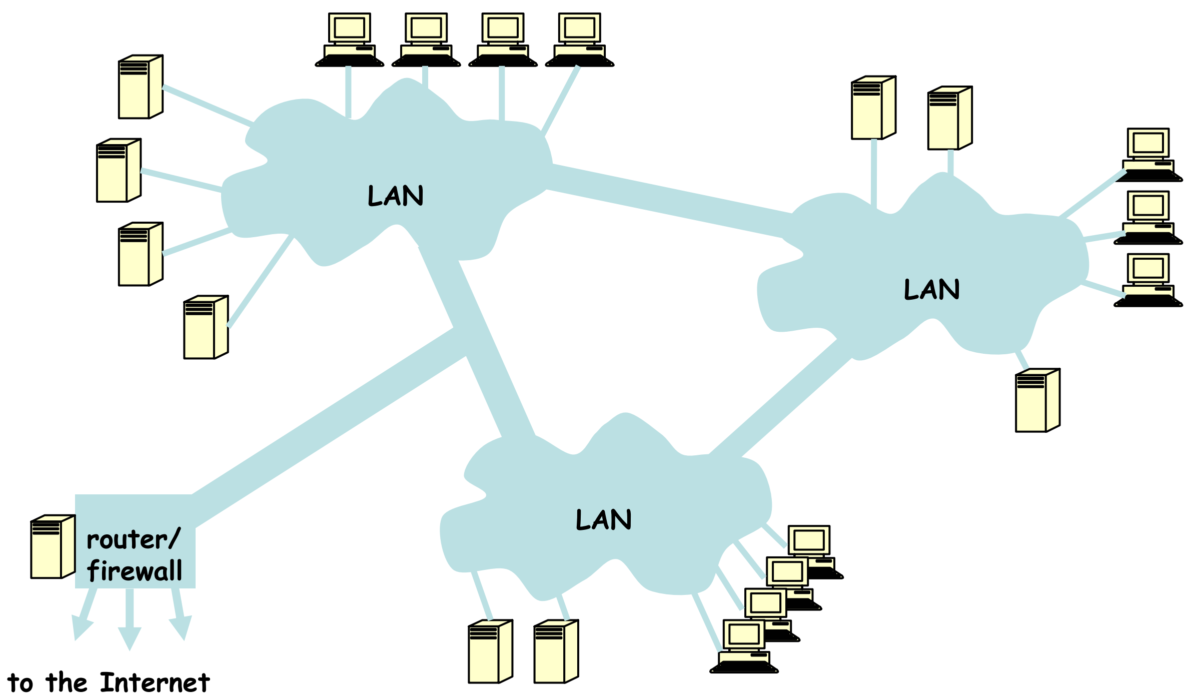
The abstraction of DSM





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 strong

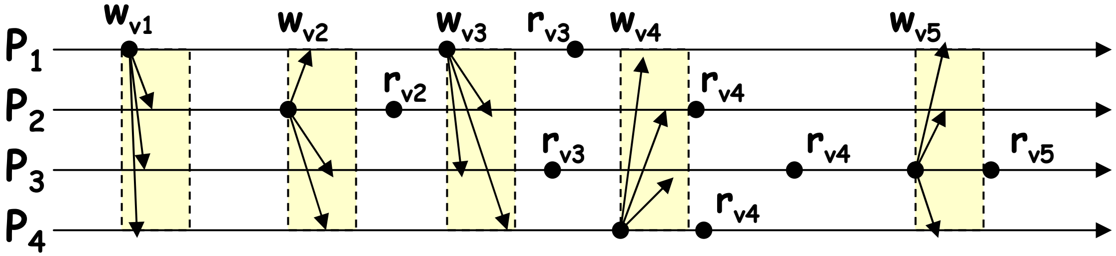
The most actual value which results from the last write operation on the time line.

consistency models

- atomic
- sequential
- release
- entry
- ....
- problem-oriented shared memory (type-specific structure & consistency)

very weak

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 (asynchronous) distributed system.

Sequential Consistency can be expressed as follows:

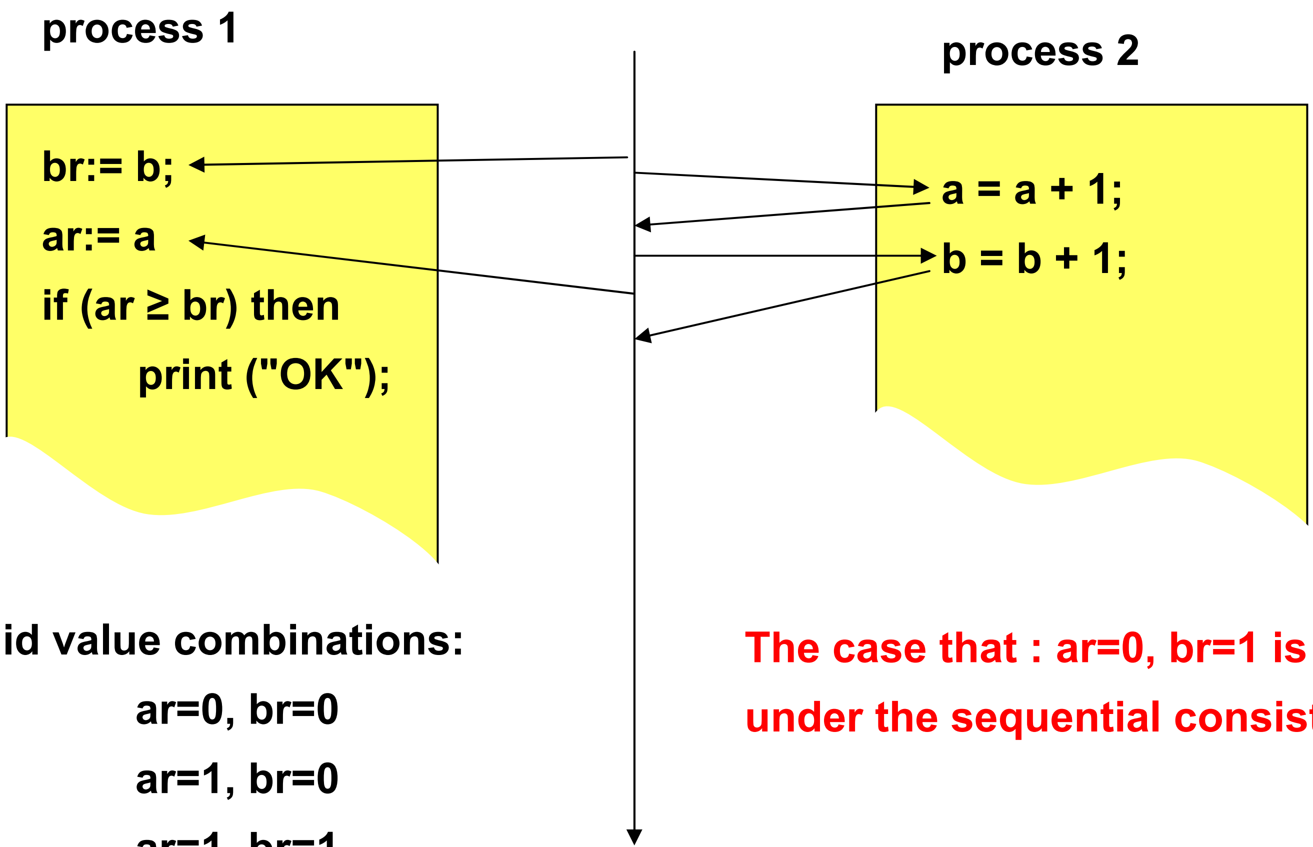
There is a virtual interleaving for read- and write-operations of all processes on a single virtual memory image. Sequentially consistency is given if:

1.) The program sequence of every individual processor is maintained in the interleaving (read and write of the same process appear in the order, in which they have been specified).

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



valid value combinations:

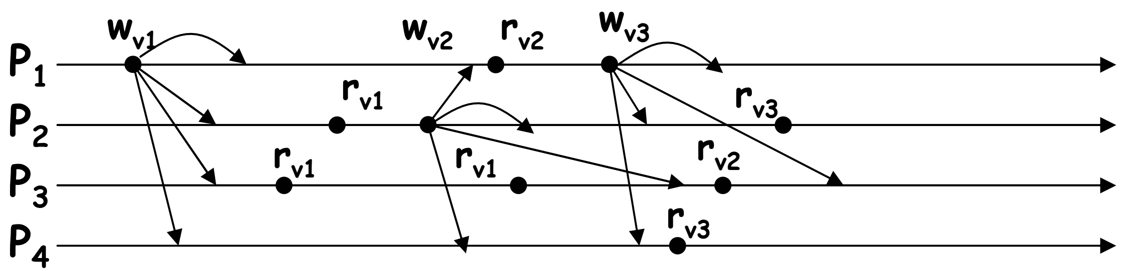
ar=0, br=0

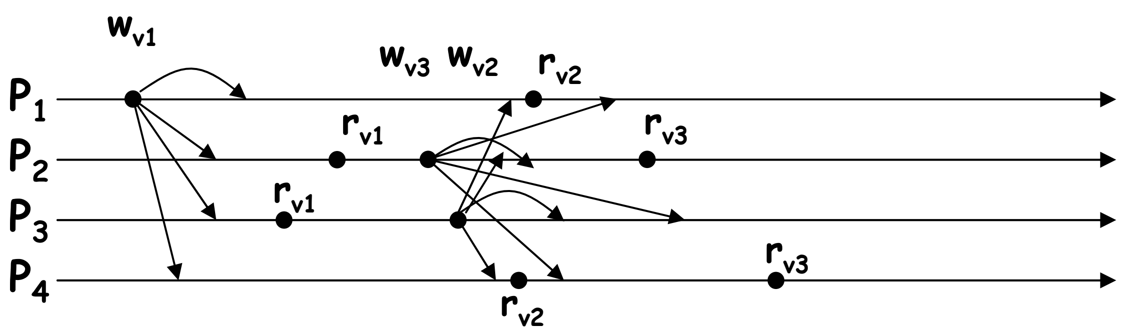
ar=1, br=0

ar=1, br=1

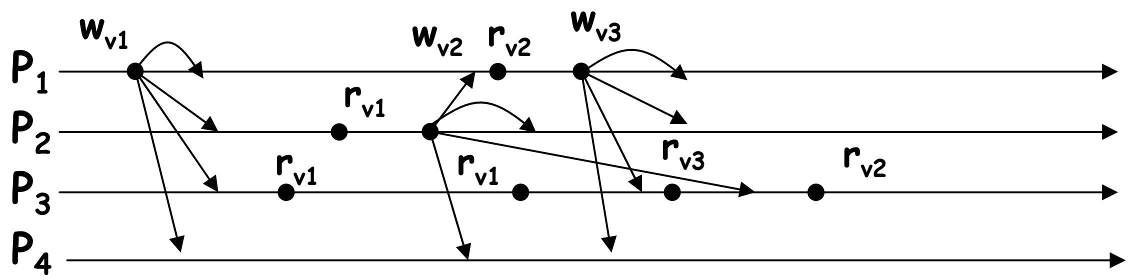
The case that : ar=0, br=1 is excluded under the sequential consistency model.

Consistency Models



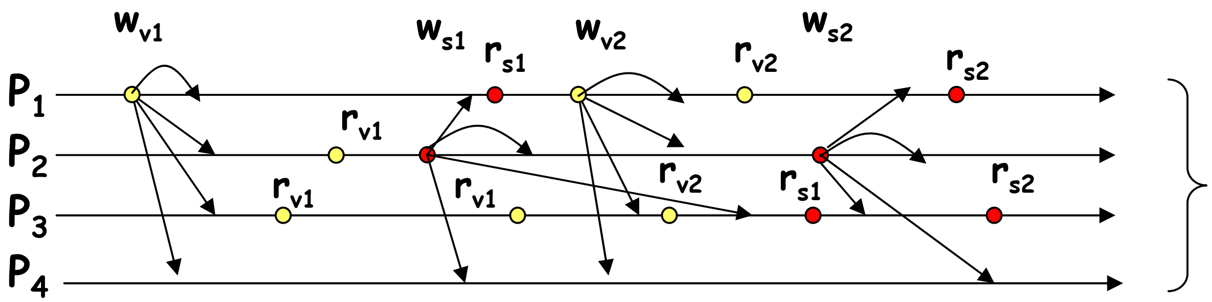


sequentially consistent



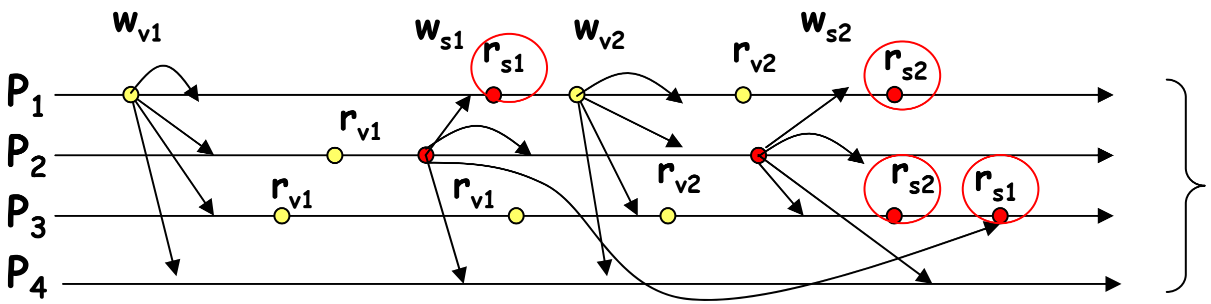
not sequentially consistent

Consistency Models



coherent but not consistent

Coherency: Sequential consistency for a single memory location.



neither coherent nor consistent

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, all 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

<u>some Data types:</u>

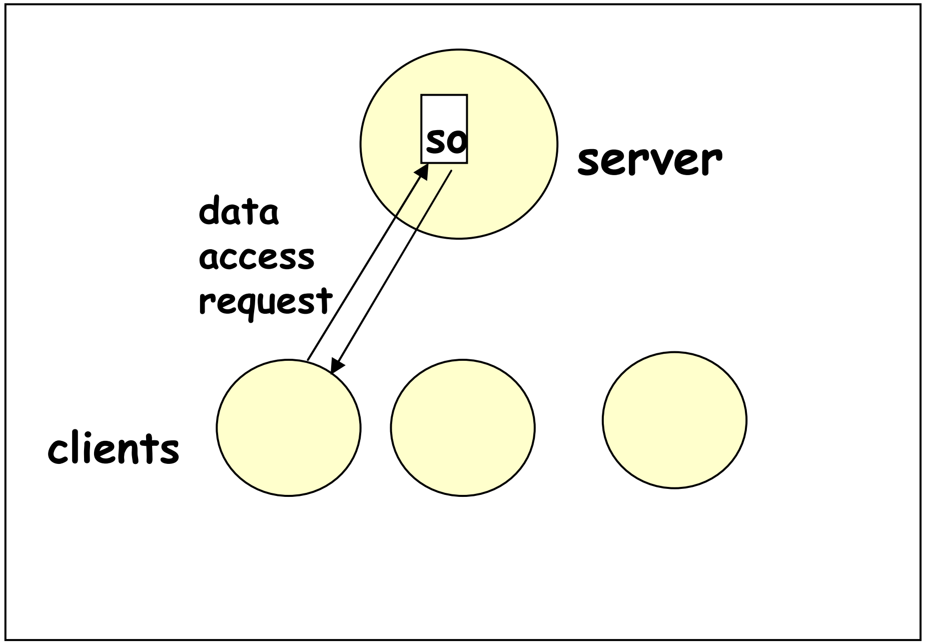
- read-only
- write shared
- producer-consumer
- migratory
- result
- conventional

<u>some protocol options:</u>

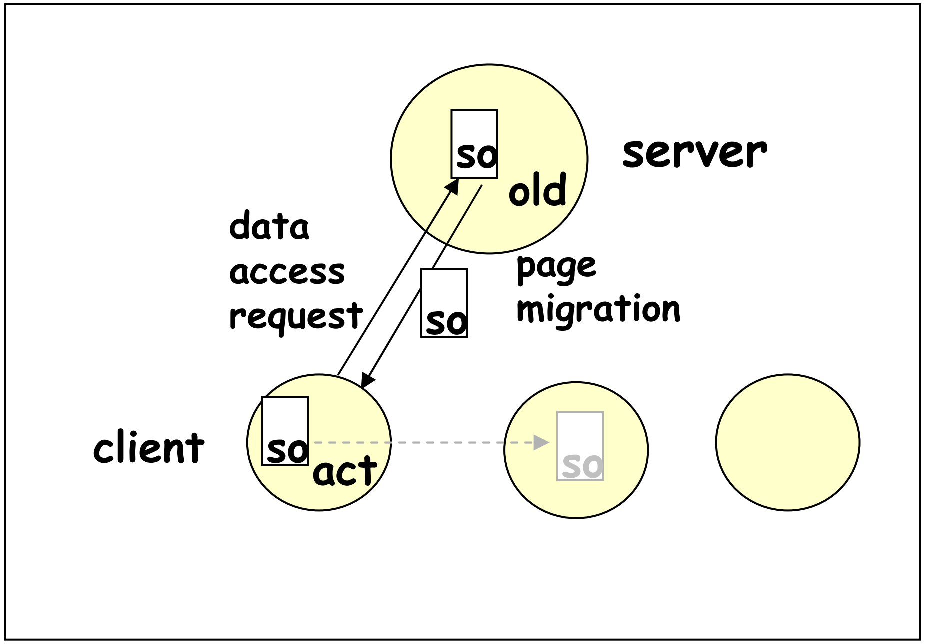
- write update
- write invalidate
- eager or lazy variants
- data element can be modified by multiple writers
  - > needs more semantics (e.g. multiple records on page)
- data item is used by a fixed set of processes

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.

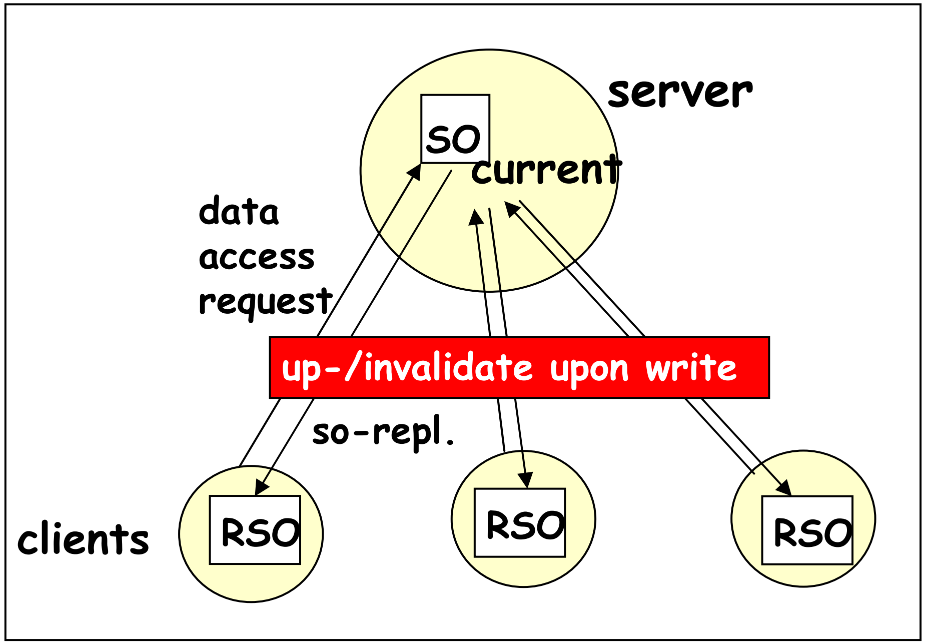
Concurrent reads: no problem

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.

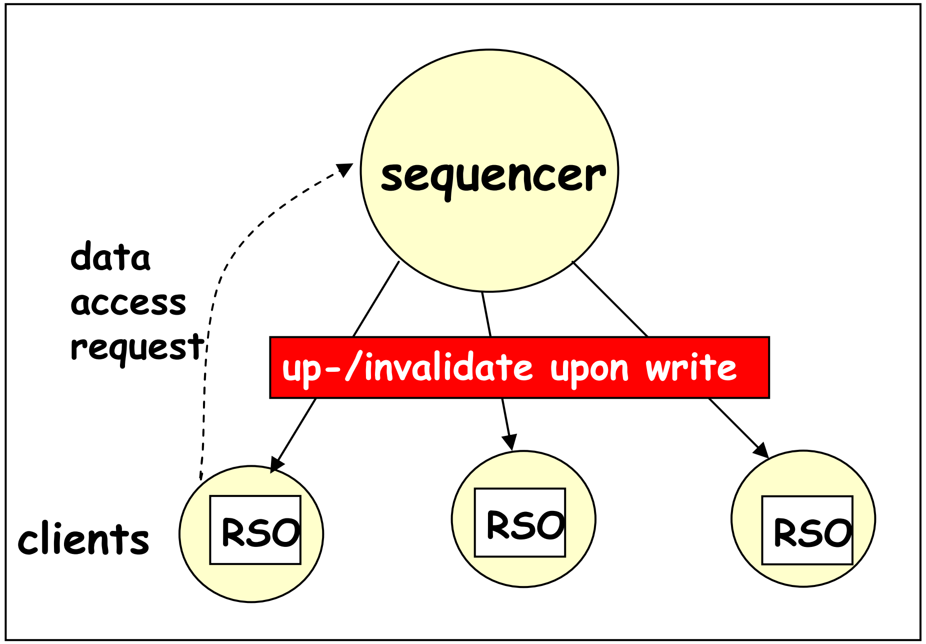
Implementation options

centralized SO replication (read-only)



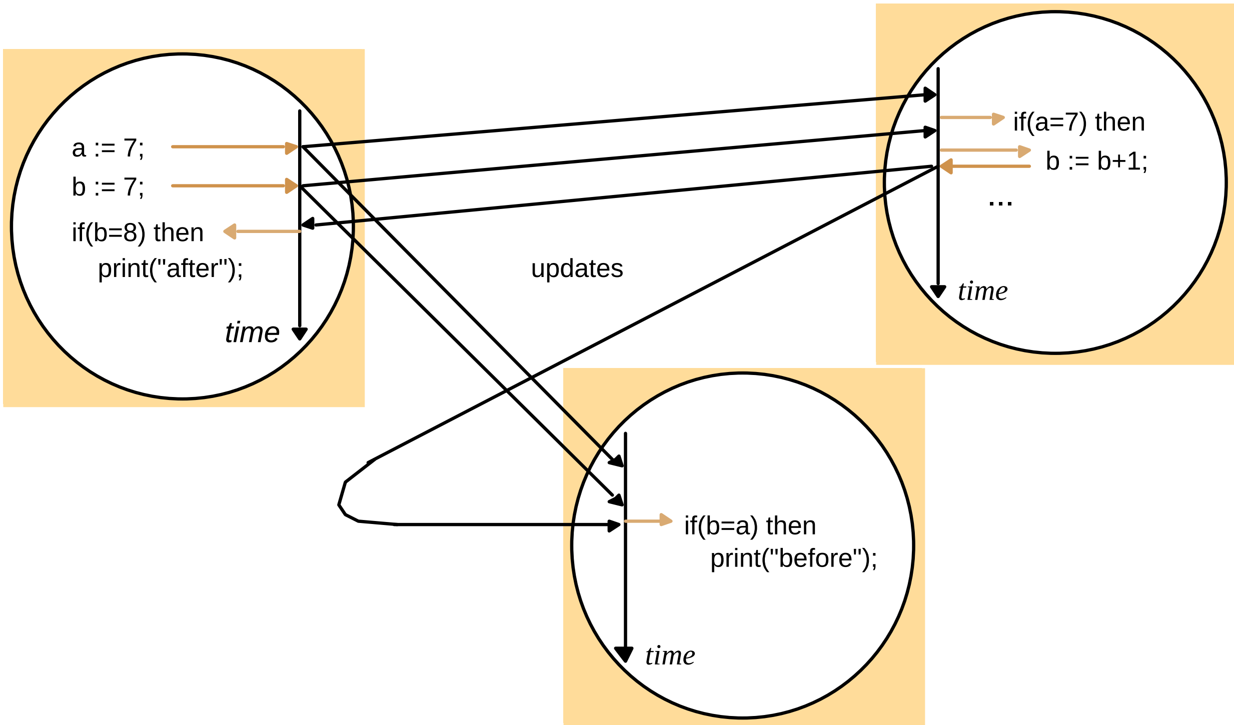
writer only receives a copy of SO iff all RSOs (Replicated Storage objects) are invalidated.

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.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

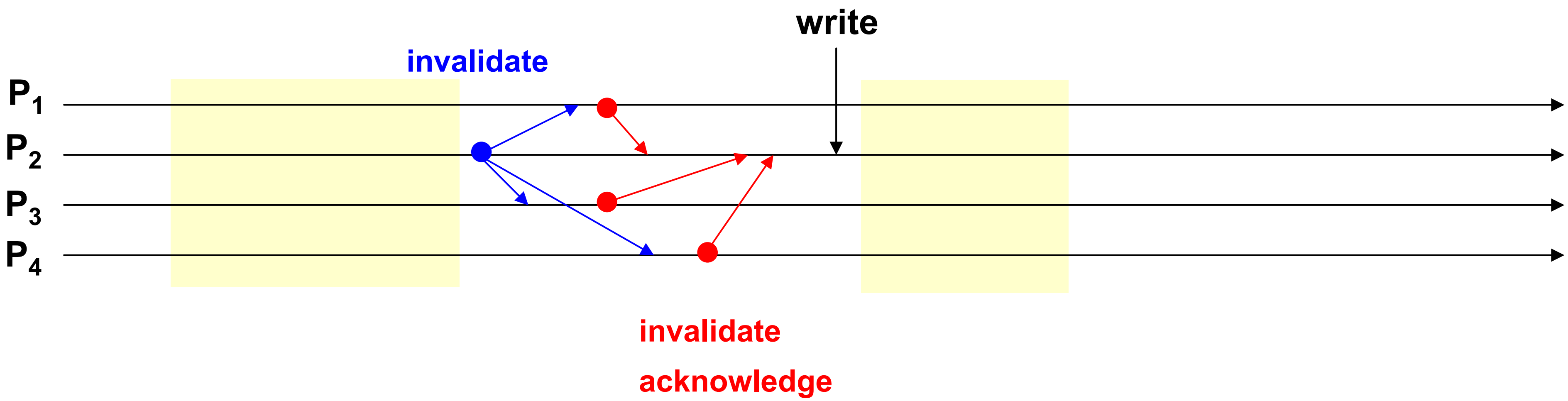
Update option: Write-invalidate

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.

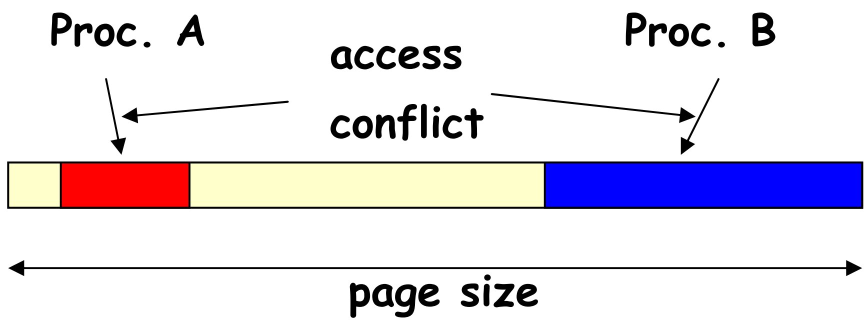


Problems and trade-offs in DSM

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

Example: sequential consistency and write update

Problems with write-udate

Assumption: -system exploits hardware page protection,
- rights may 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

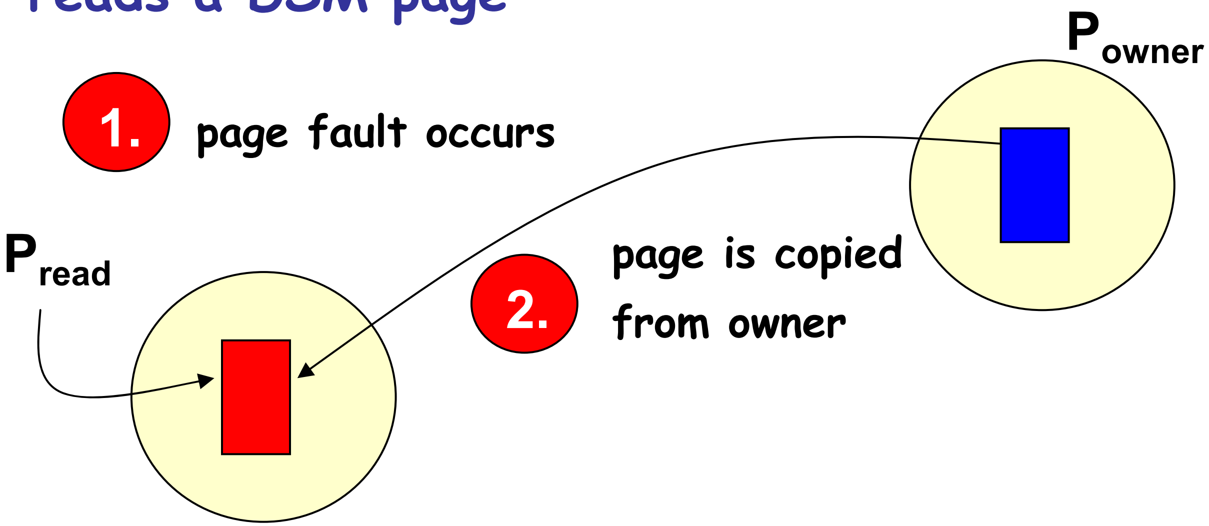
- owner of page (*owner (p)*) holds the most recent version of the page:

  - 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

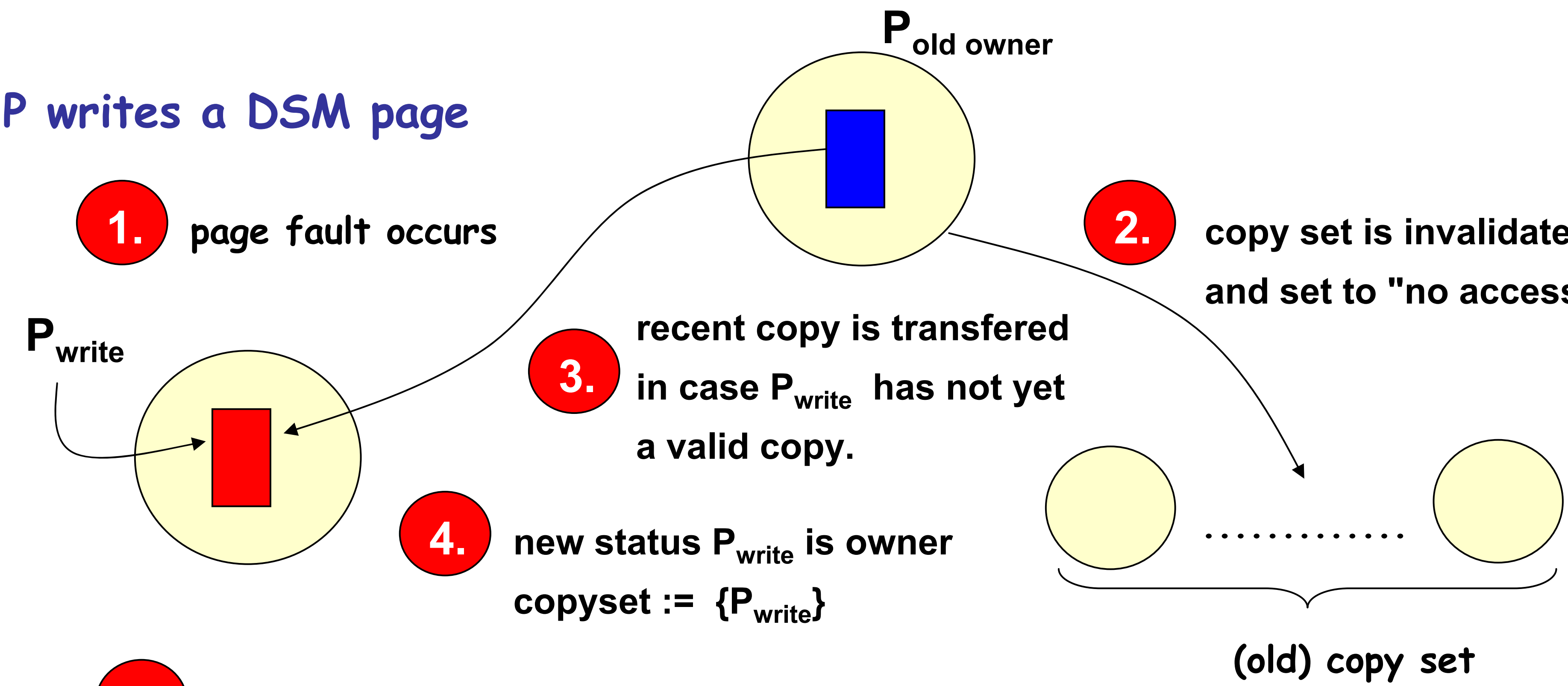


if P_{owner} was writer it retains a read right and remains owner (because this is the most recent copy). It has to handle subsequent requests.

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 := $\{P_{write}\}$
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

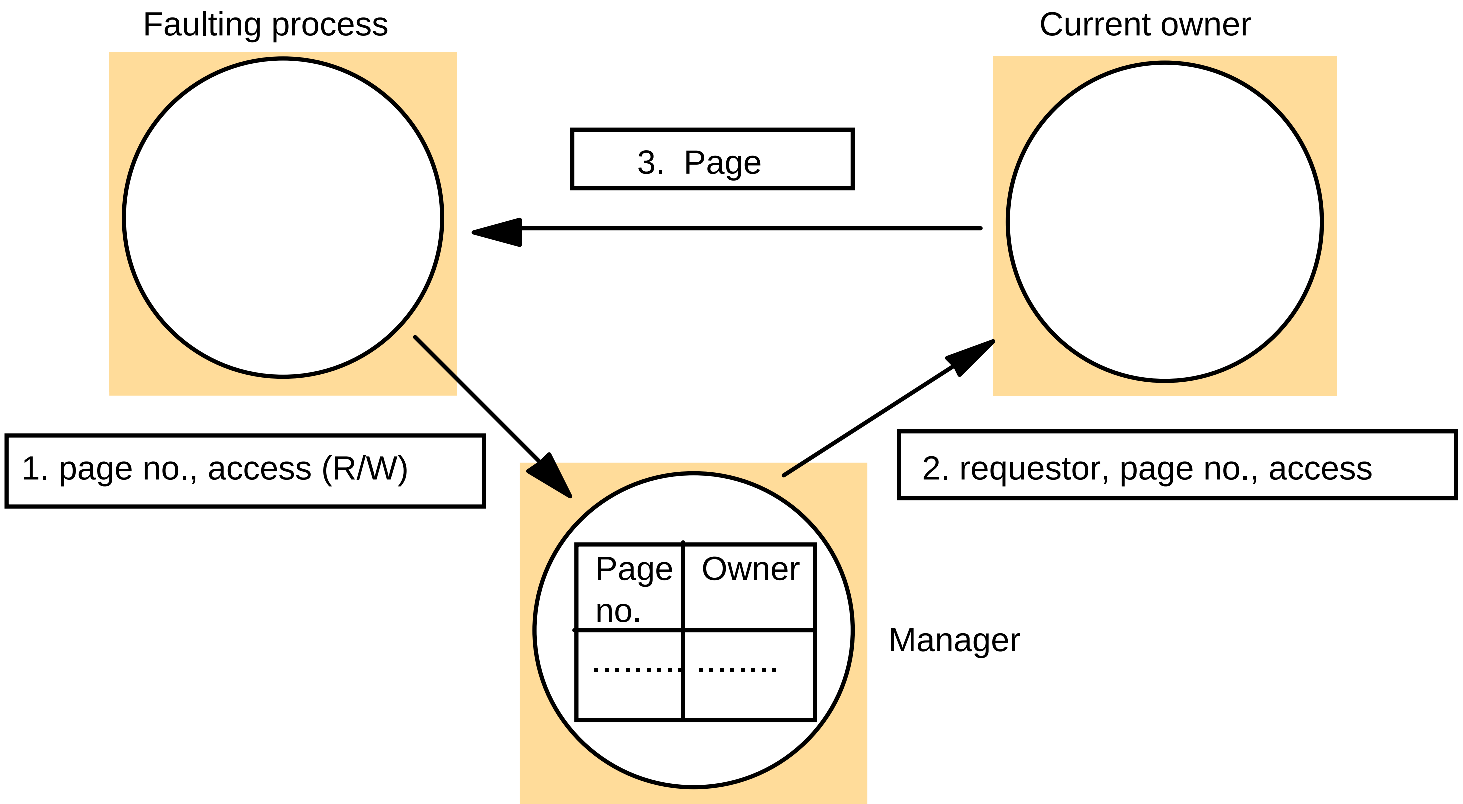
2.) Multicast (totally ordered)

3.) Dynamically Distributed Manager

- build a chain of hints
- update the hints dynamically

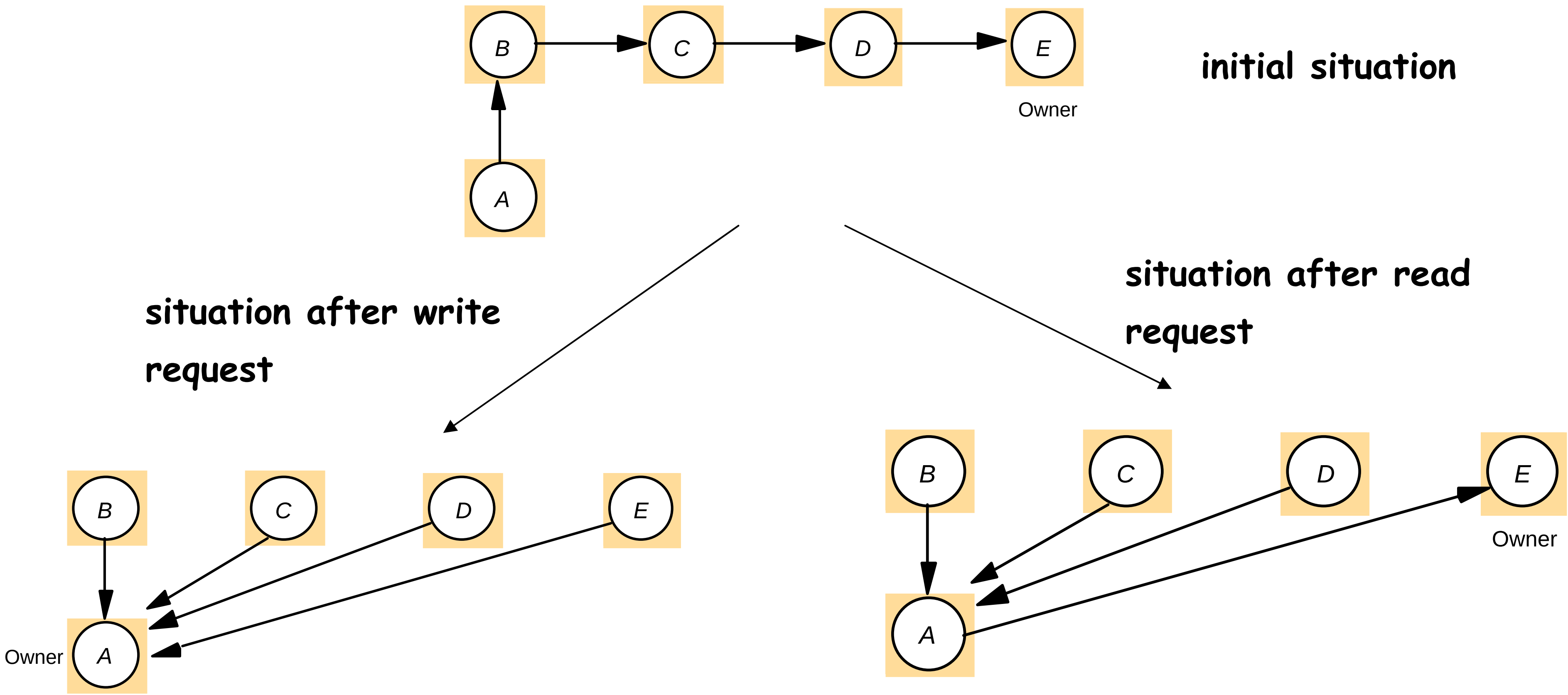
Finding the owner of a page

Central manager approach



Finding the owner of a page

Dynamic distributed manager approach



The

End

roadmap for OS II:

access control and protection

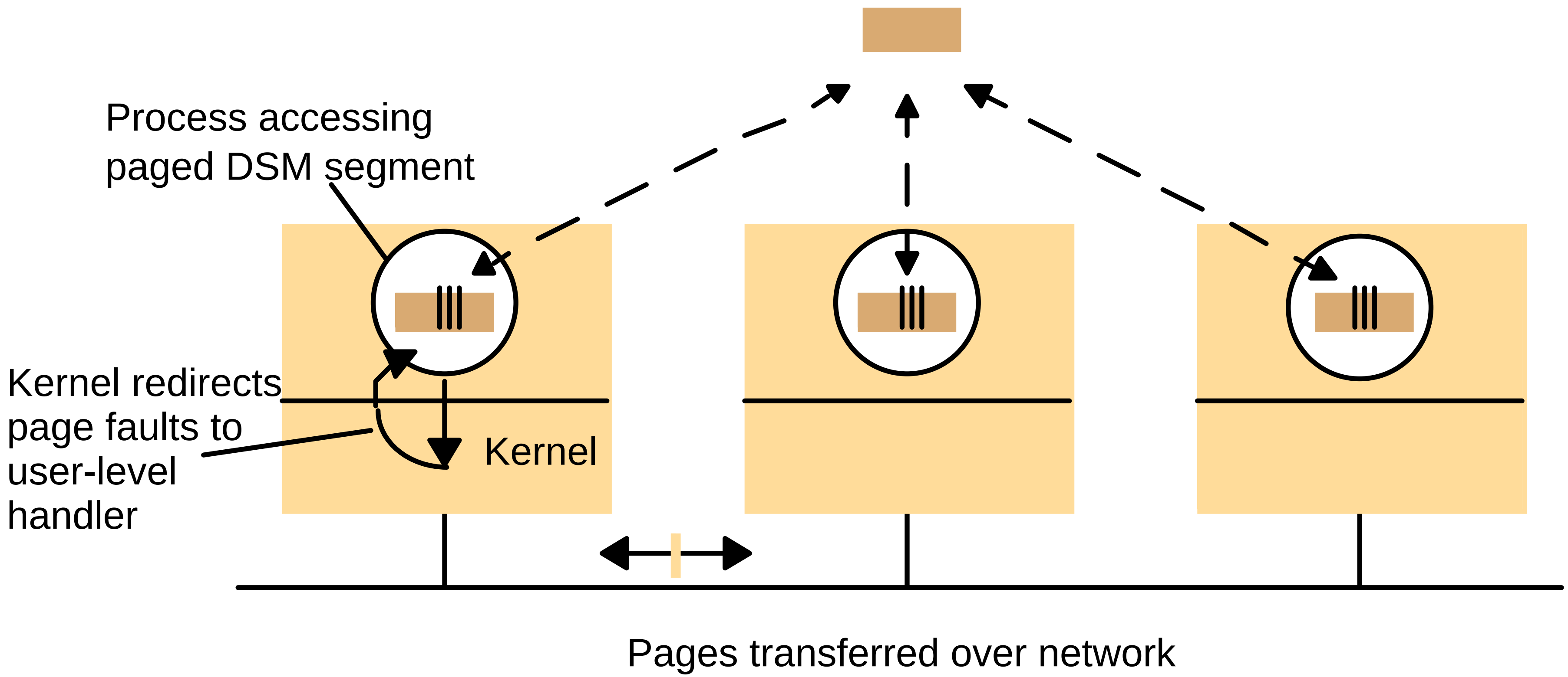
models of distributed systems

communication abstractions and programming models

distributed storage systems

OS for tiny embedded systems

Implementation Issues: sequential consistency in Ivy



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000