Distributed Systems

14 Consistency

June-29-2009

Gerd Liefländer

System Architecture Group

Outline

- Motivation & Introduction
- Consistency Models
- Continuous Consistency Model
- Data-Centric Consistency Models
  - Strong Consistency
  - Weak Consistency
- (Eventual) Client Centric Consistency Models
  - Monotonic Reads/Writes
  - Read Your Writes/Writes Follow Reads

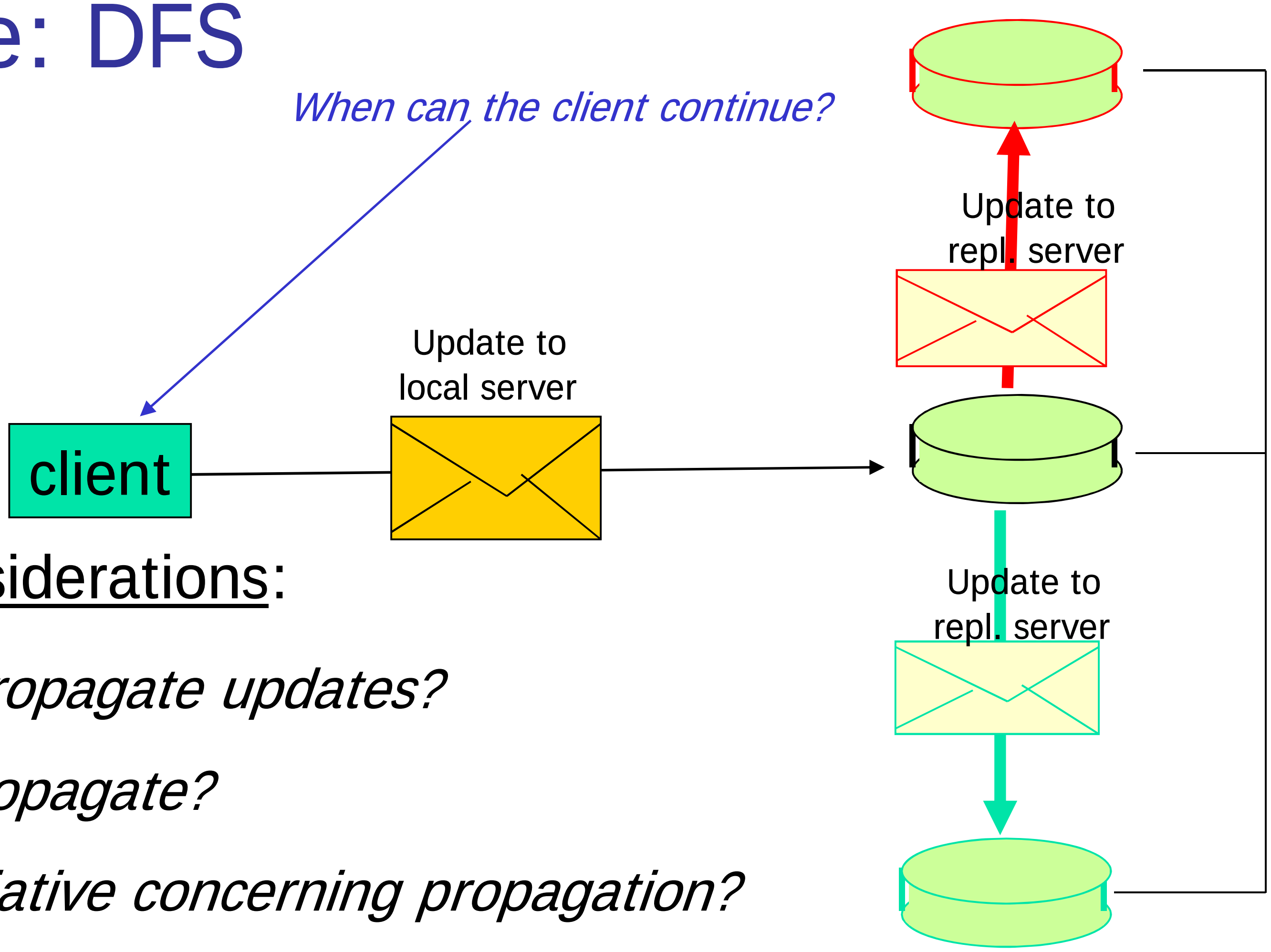
Motivation & Introduction

Why Replication?

Replicated data at multiple sites can improve ...

- *Availability* (by redundancy), e.g.
  - if primary FS crashes, standby FS still works
- *Performance*, e.g.
  - local access is faster with reduced communication delays
  - concurrent requests can be served by n>1 servers
- *Scalability*, e.g.
  - prevents overloading a single server (size scalability)
  - avoids communication latencies (geographic scalability)
- *Fault tolerance*
  - masks node crashes
  - implies data consistency

Example: DFS

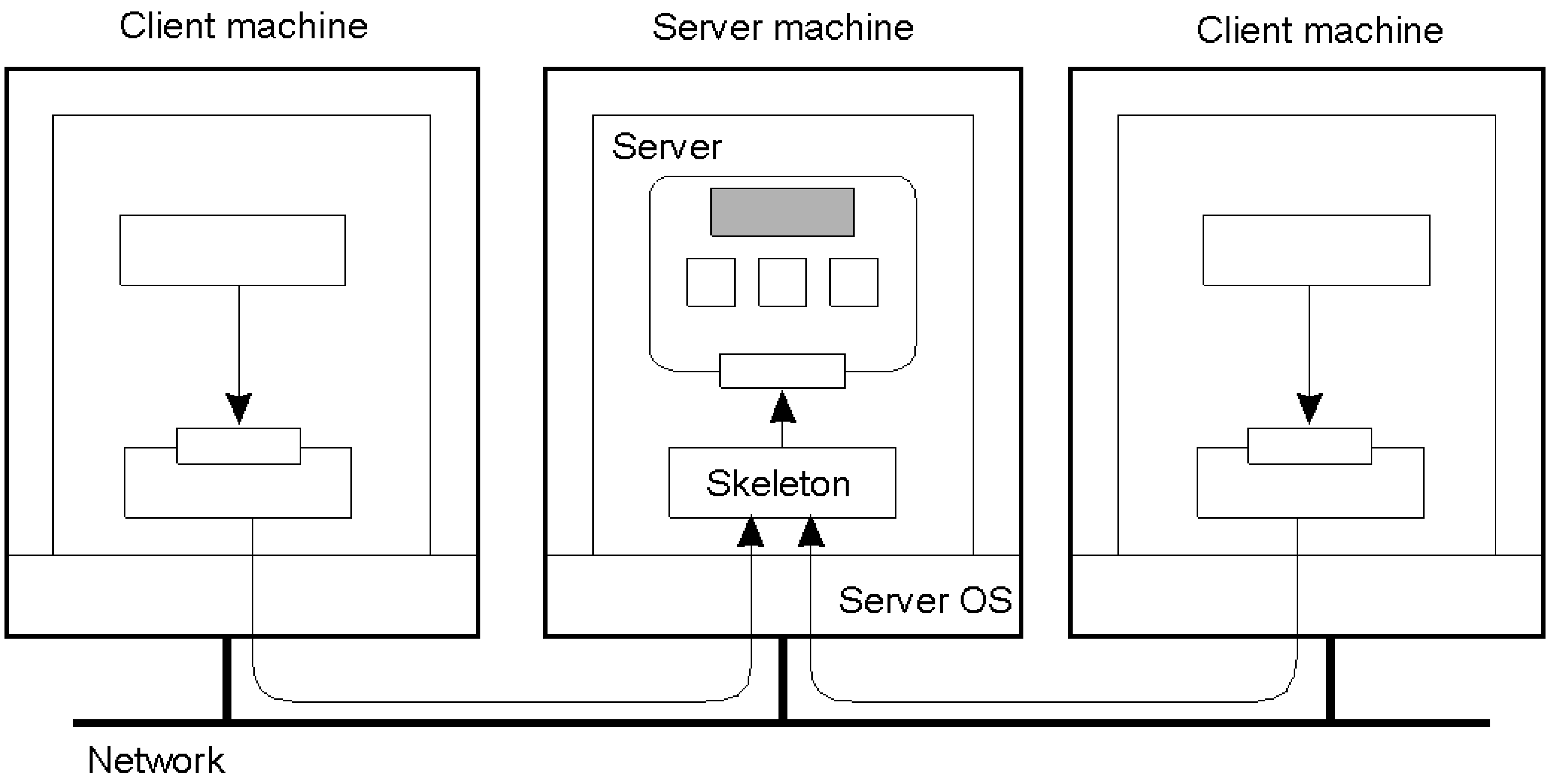


Design considerations:

- *When to propagate updates?*
- *What to propagate?*
- *Who is initiative concerning propagation?*
- *How strict are the consistency requirements?*

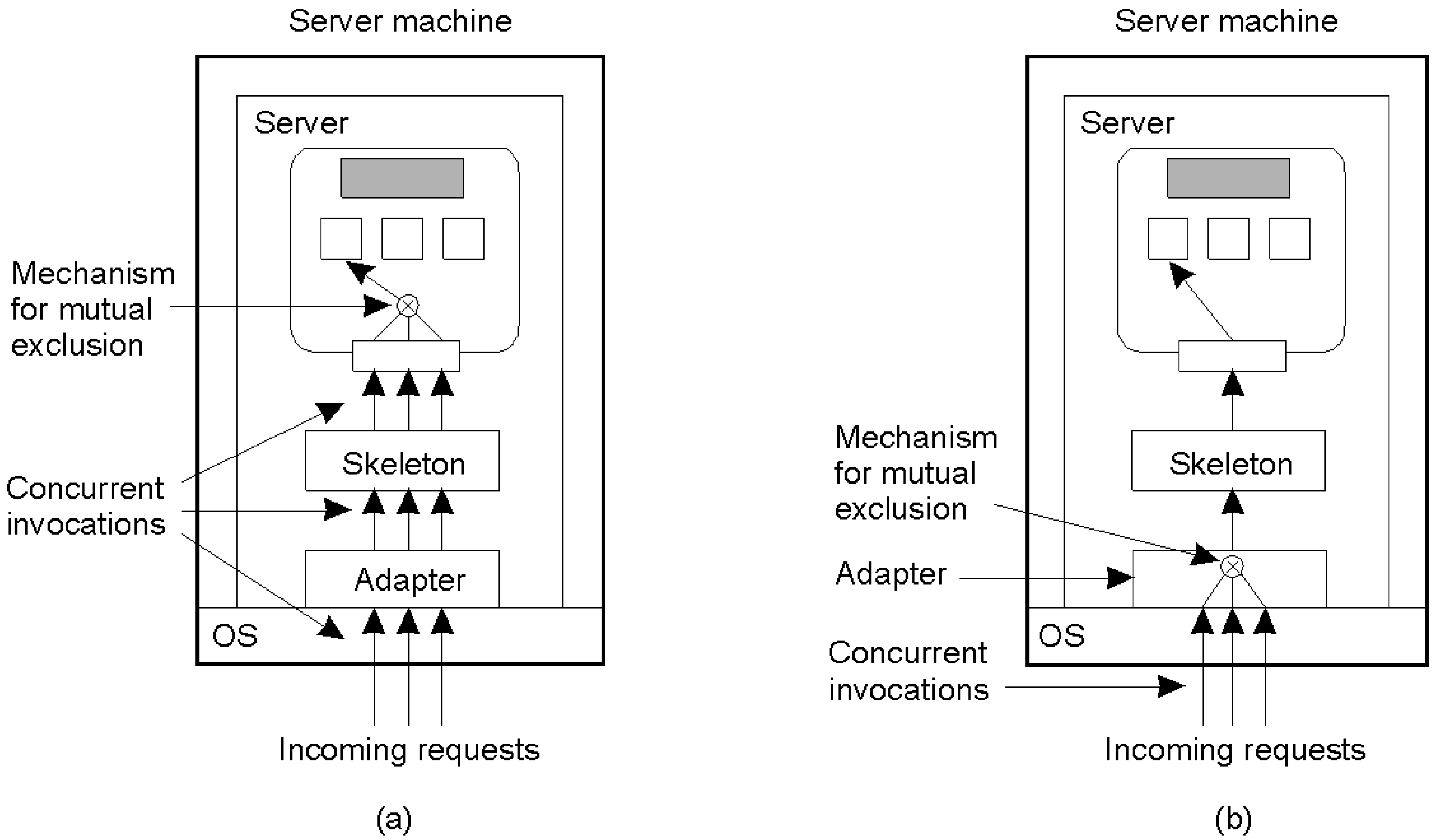
⇒ Consistency Models

Object Replication (1)



- Organization of a distributed remote object shared by two different clients

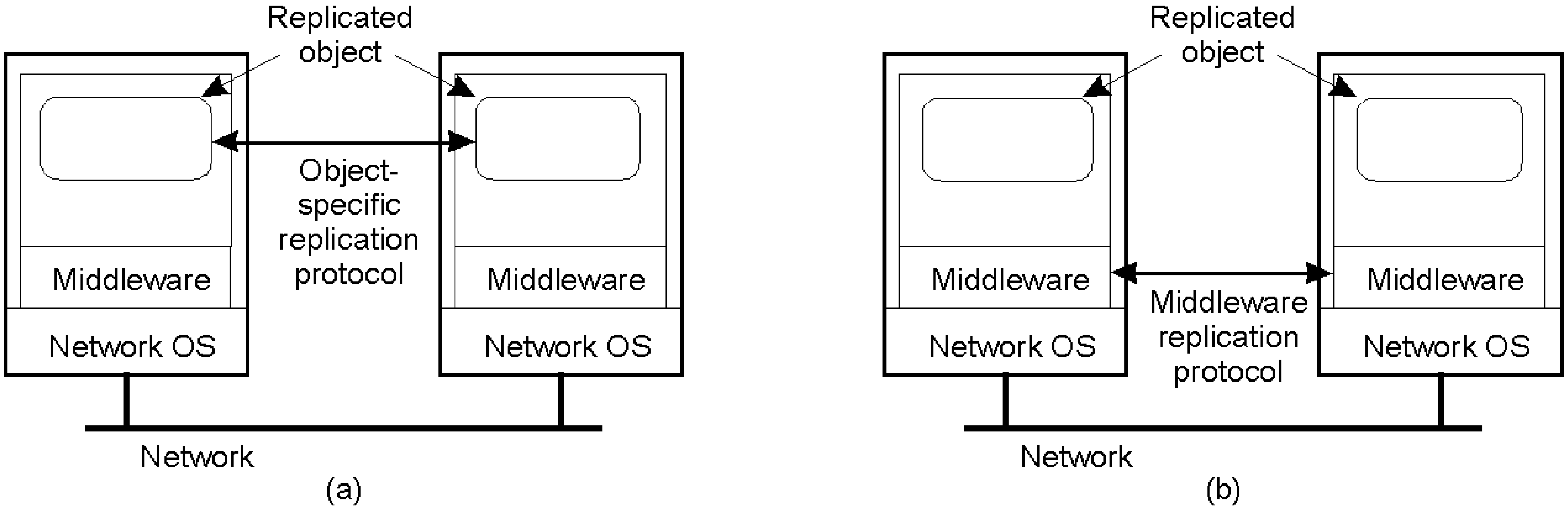
Object Replication (2)



a) A remote object capable of handling concurrent invocations on its own.

b) A remote object for which an object adapter is required to handle concurrent invocations

Object Replication (3)



a) A distributed system for replication-aware distributed objects

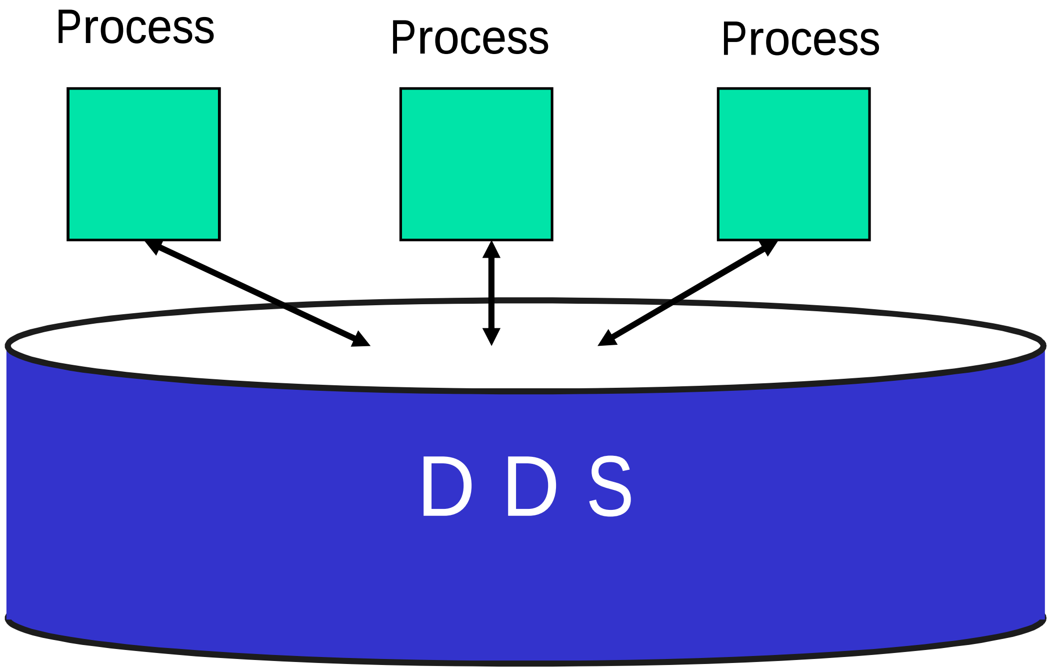
b) A distributed system responsible for replica management

Two Consistency Models

- Data Centric Model
  - Defined consistency is experienced by all clients, i.e. we must provide a system wide consistent view on the data store
    - All clients see same sequence of operations at each replica, hopefully with only minor delay
    - Without additional synchronization hard to achieve in DS
- Client Centric Model
  - If there are no concurrent updates (or only few compared to number of reads) we can weaken consistency requirements
  - Consistency of the data store only reflects one client's view
    - Different clients might see different sequences of operations at their replicas
    - Relatively easy to implement

Data Centric Consistency Models

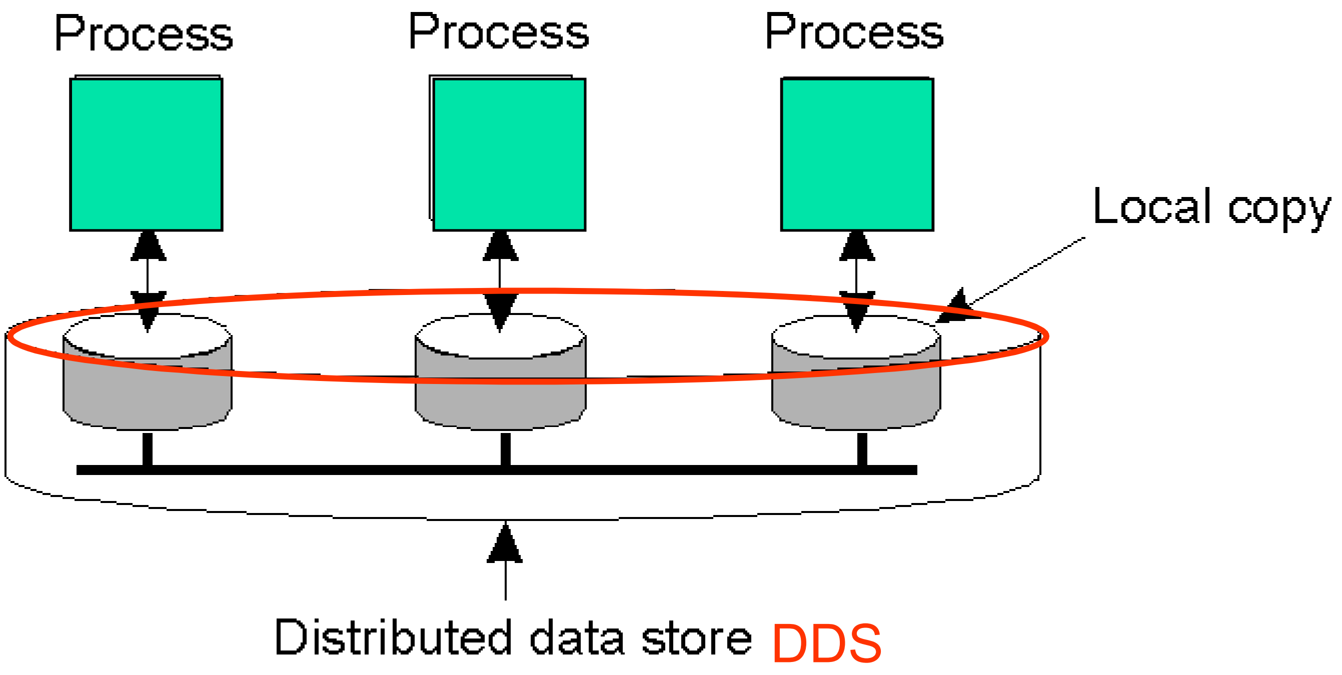
- Distributed Data Store (DDS)



Clients point of view:

- The DDS is capable of storing an amount of data

Distributed Data Store



Data Store’s point of view:

- General organization of a logical data store, physically distributed <u>and/or</u> replicated across multiple nodes
- The more date we replicate the more overhead we introduce in preserving DDS consistency

Data-Centric Consistency Model

- Consistency Model:
  - Contract between processes (clients) and DDS
    - Access rules for the processes
    - Properties of the “read data”, e.g.
      - do they reflect the value of the last update
  - A consistency model restricts potential values of a read
    - If ∃ only few restrictions ⇒
      - model is easy to use, but
      - its implementation is often inefficient
    - If ∃ strong restrictions ⇒
      - model is hard to use, but
      - its implementation is efficient

Continuous Consistency

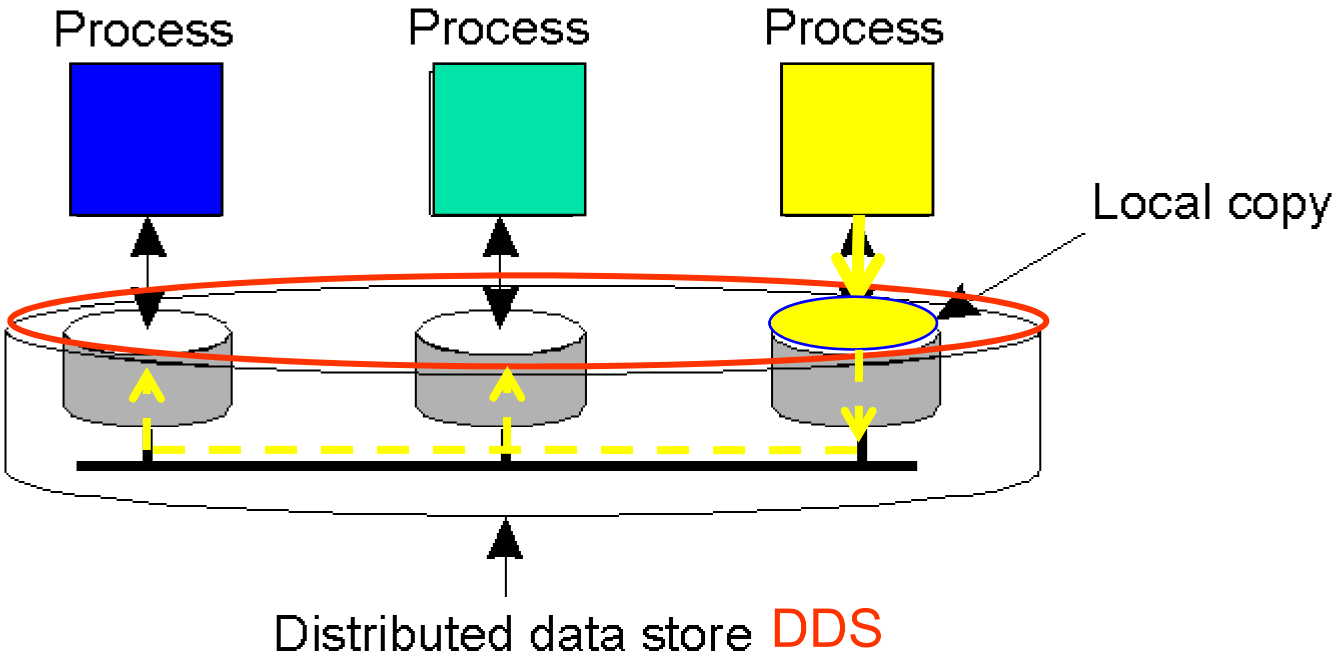
Amin Vahdat, Haifeng Yu: “Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services”

ACM Trans. on Computer Systems, 2002

Read that paper till next Monday

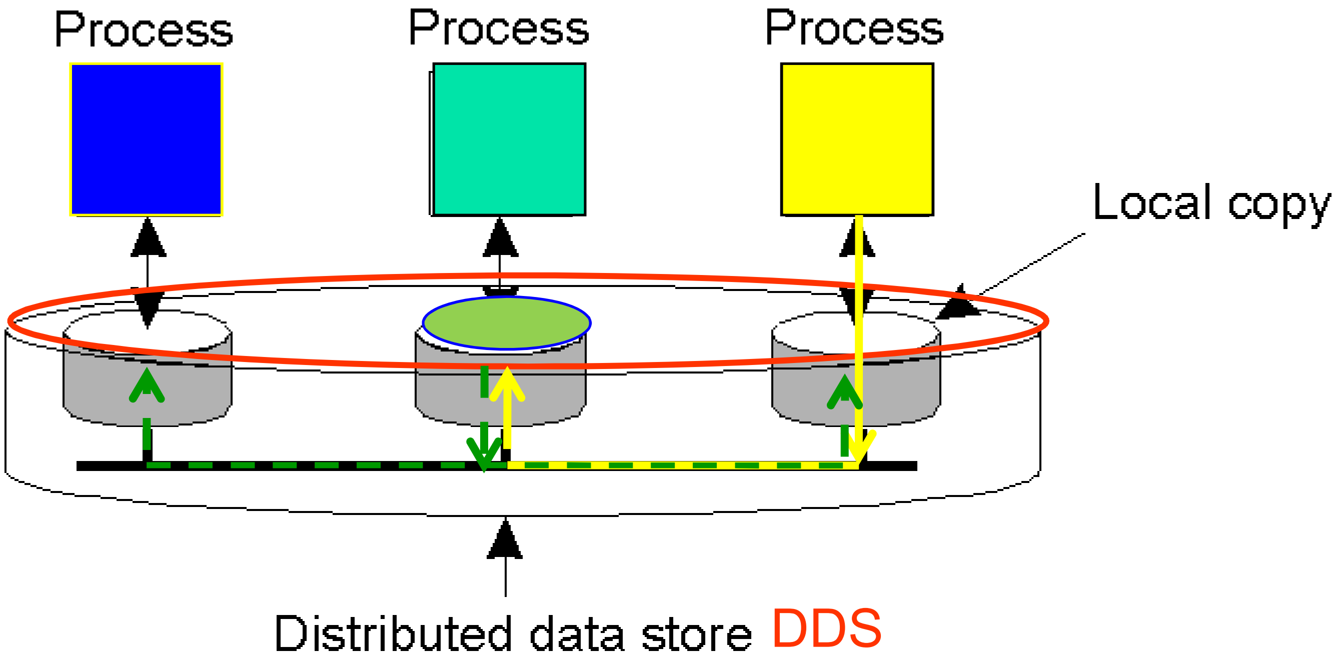
Conit= Consistency Unit

Distributed Data Store



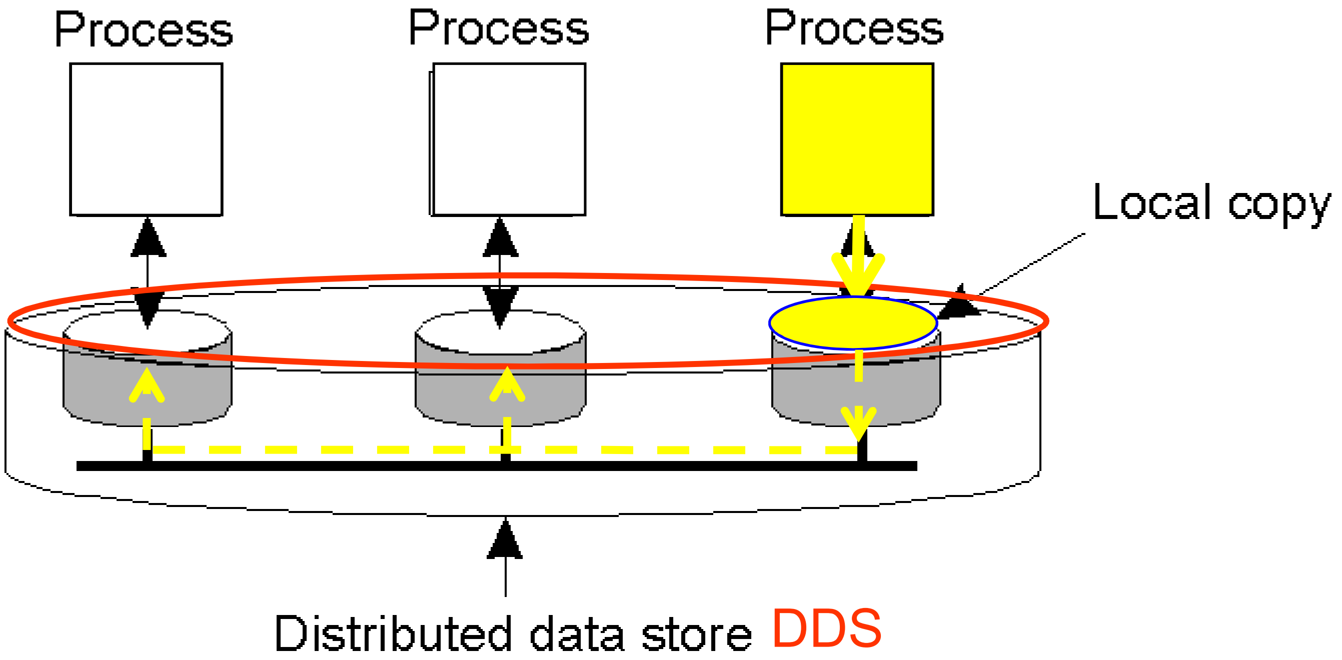
- When every process can write to its local replica, we must propagate these writes ASAP to all other replicas

Distributed Data Store



- When every process can write to its local replica, we have to propagate these writes ASAP to all other replicas
- When one site is the primary replica, all writes are sent to it, primary replica propagates updates from time to time to the other replicas

Distributed Data Store



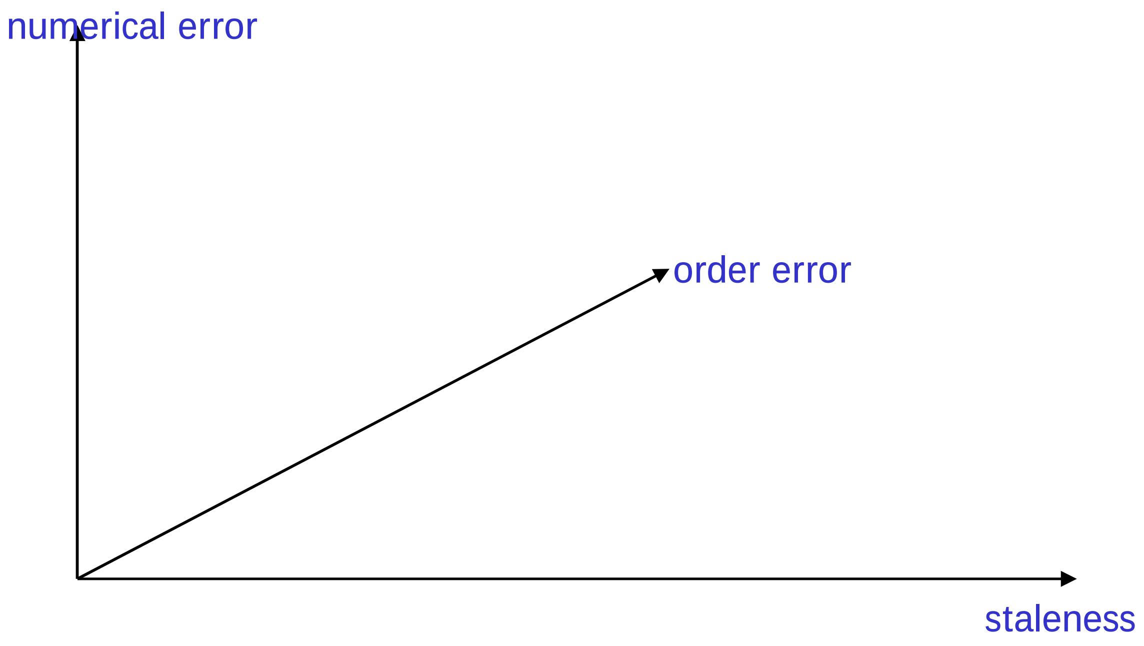
- When every process can write to its local replica, we have to propagate these writes ASAP to all other replicas
- When one site is the primary replica, all writes are sent to it, primary replica propagates updates from time to time to the other replicas
- When there is only one updating process, it is responsible for propagating all writes to all replicas whenever this is necessary

Haifeng & Amin’s Consistency Models

- Strong (pessimistic) Consistency Model
  - One-copy serializability (Bernstein, Goodman, 1984) imposes performance overhead and limits system availability

- Weak (optimistic) Consistency Model
  - Tolerate relaxed consistency (i.e. delayed propagation of updates)
  - Requires less communication, resulting in improved overall performance and availability

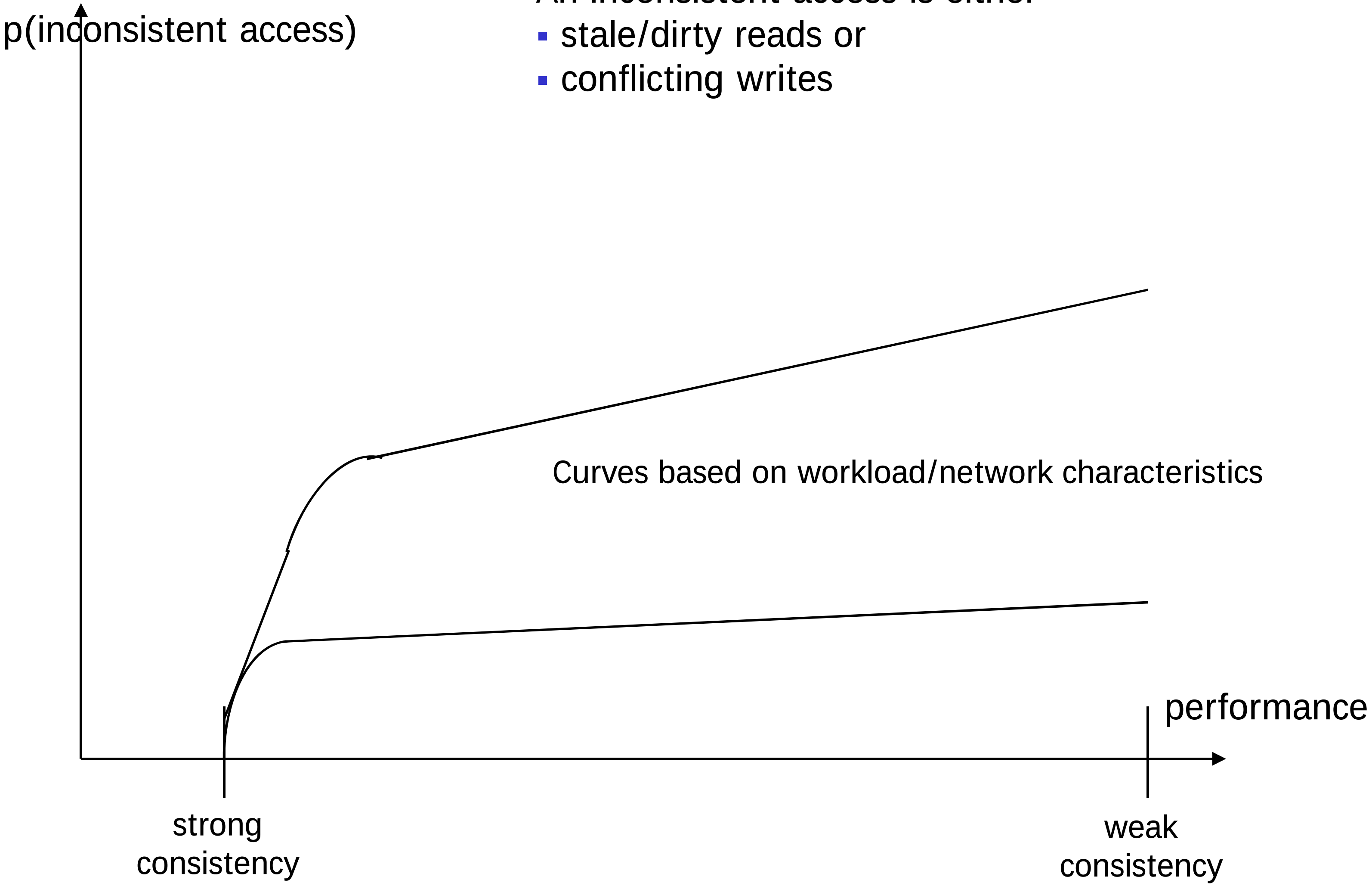
How to Quantify Inconsistency?



- Numerical error limits:
  1. Number of writes applied to a given replica, but not yet seen by others
  2. Absolute or relative numerical deviation of a conit element
- Order error limits the difference concerning the ordering at the different replicas, e.g. not more than 2 out of order
- Staleness places a real-time bound on the delay of write propagation among replicas

Characteristics of Consistency Models

An inconsistent access is either
- stale/dirty reads or
- conflicting writes



TACT Toolkit

- 3 wide area applications
  - *Distributed bulletin board service*
  - *Airline reservation system*
  - *Replicated load distribution front ends to a web server*
- Algorithms to bound each of the previously mentioned consistency metrics
  - Push approach based solely on local info helps to bound numerical errors
  - A write commitment algorithm allows replicas to agree on a global total write order, enforcing bounds on order errors
  - Staleness is maintained by a real-time vector

Example: Numerical Errors

- Absolute maximal deviations of conit values (e.g. stock market prices) of n replicas
  - Deviations of conit values is limited by 0.05 EURO, i.e. we regard a data store still consistent if the following holds, suppose we have n replicas at nodes N_1, N_2, ...N_n: $\forall$ i,j $\in$ [1,n], i $\neq$ j

    $|value_i – value_j|\leq 0.05$ Euro
- Relative numerical deviations
  - A data store is still consistent as long as all conit values do not differ more than 0.5 %
- Numerical error can also be understood in number of updates having been applied to a replica at node N_i, but not yet seen by other (n-1) replicas
  - A web cache might have not seen a batch of operations carried out by a web server

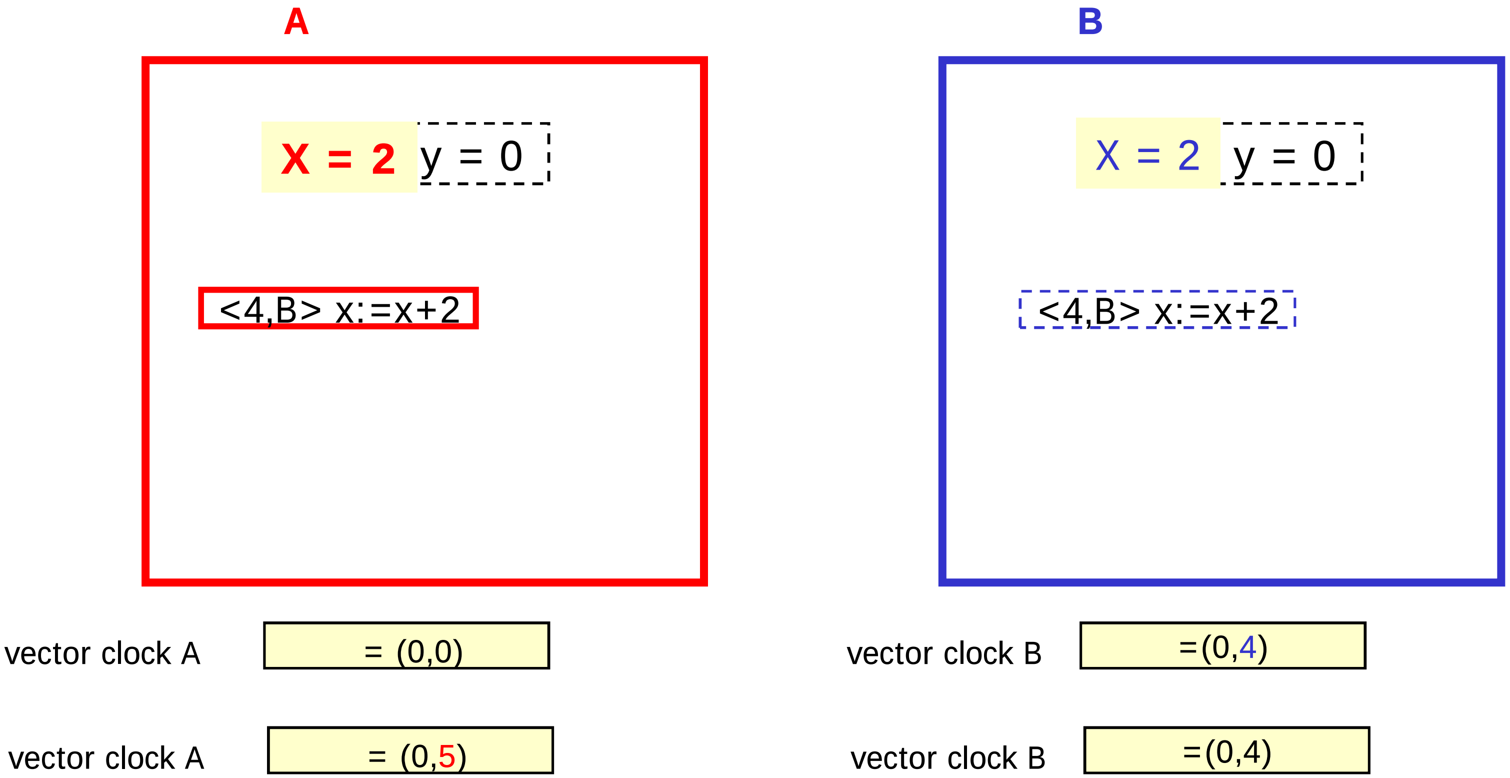
Order Deviations

- Some application allow different orders of updates at the various replicas as long as the number of differences are limited
  - Each update is tentatively done on some local replica awaiting global agreement from all other replicas
  - As a consequence some updates have to be rolled back and applied in a different order before becoming permanent

Staleness Deviations

- Some applications can live with the fact that they use old conit values, as long as the values are not too old
  - Weather reports stay reasonable stable and accurate over some limited time, e.g. hours
  - In this case, the main server might collect a bunch of updates and propagate all of them in one update message to all replicas after some limited time interval (e.g. 2 hours)

Continuous Consistency



A

X = 2 y = 0

<4,B> x:=x+2

B

X = 2 y = 0

<4,B> x:=x+2

vector clock A = (0,0)

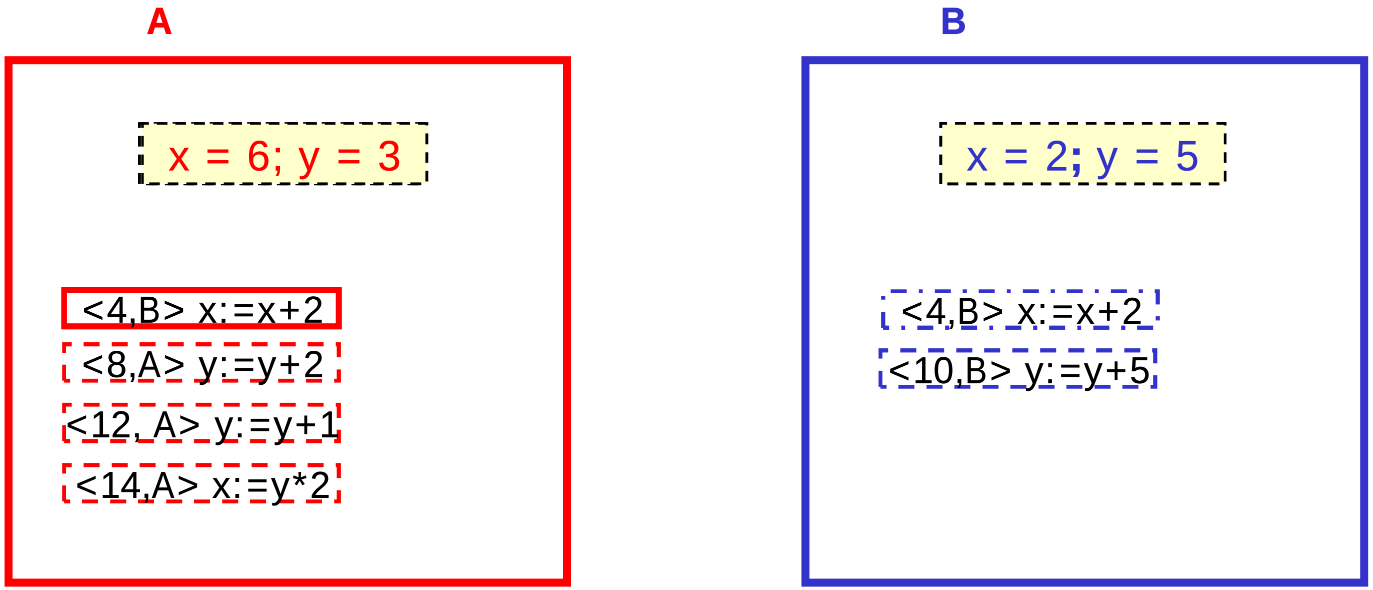
vector clock B =(0,4)

vector clock A = (0,5)

vector clock B =(0,4)

- At vector time (0,4) replica B increments x by 2
- B propagates to A a corresponding update message
- Replica A receives 4,B: x ← x+2 and makes it permanent, adjusting its vector time

Continuous Consistency (2)



A

x = 6; y = 3

<4,B> x:=x+2

<8,A> y:=y+2

<12, A> y:=y+1

<14,A> x:=y*2

B

x = 2; y = 5

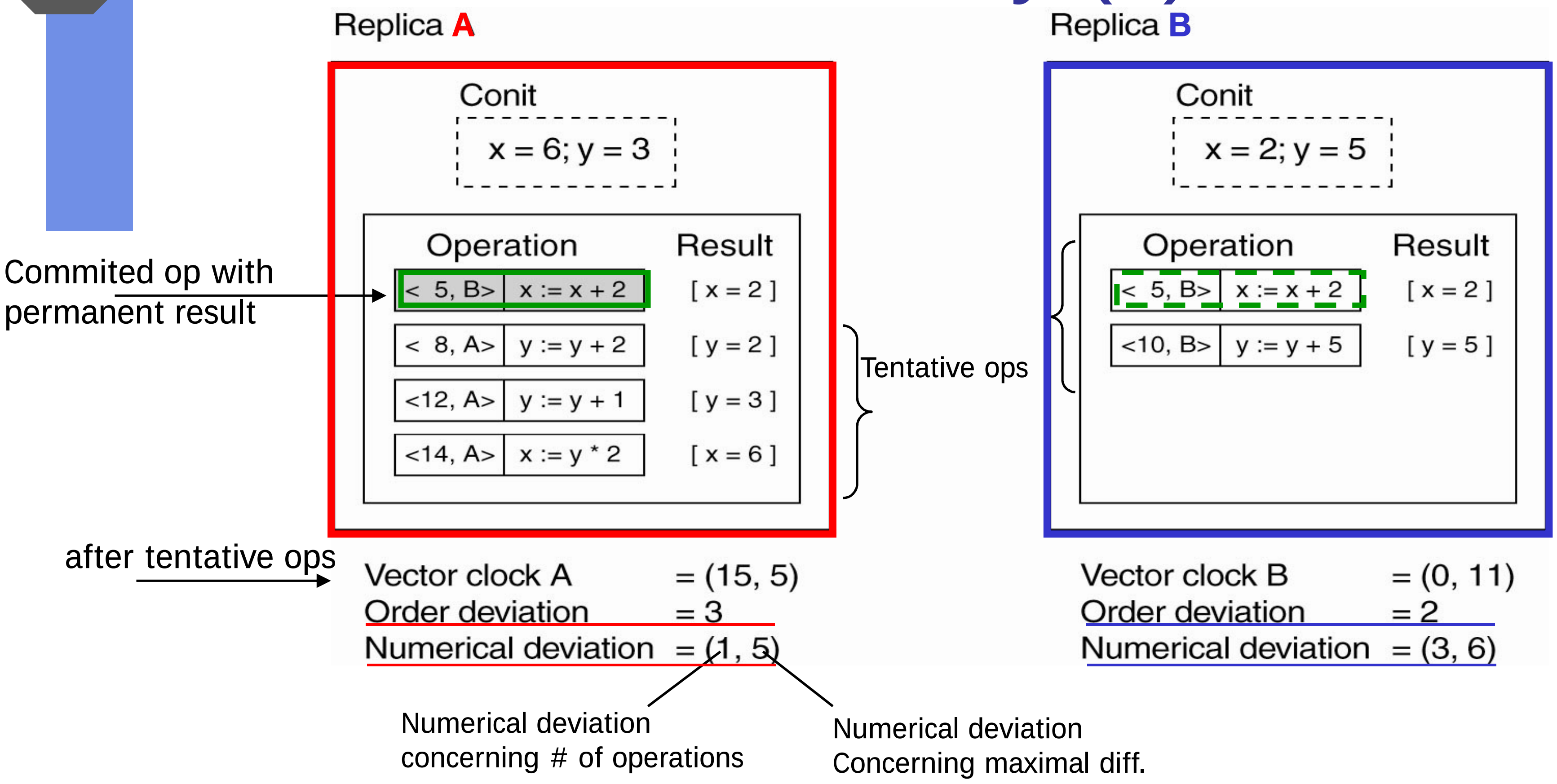
<4,B> x:=x+2

<10,B> y:=y+5

vector clock A	= (0,5)	vector clock B	=(0,5)
vector clock A	= (9,5)	vector clock B	=(0,11)
vector clock A	= (13,5)		
vector clock A	= (15,5)		

- At replica A 3 tentative operations took place, bringing replica A's ordering deviation to 3
- At replica B 2 tentative operations took place, bringing replica B's order deviation to 2

Continuous Consistency (3)



Replica A

Conit

x = 6; y = 3

Operation		Result
< 5, B>	x := x + 2	[x = 2]
< 8, A>	y := y + 2	[y = 2]
<12, A>	y := y + 1	[y = 3]
<14, A>	x := y * 2	[x = 6]

Vector clock A = (15, 5)
Order deviation = 3
Numerical deviation = (1, 5)

Replica B

Conit

x = 2; y = 5

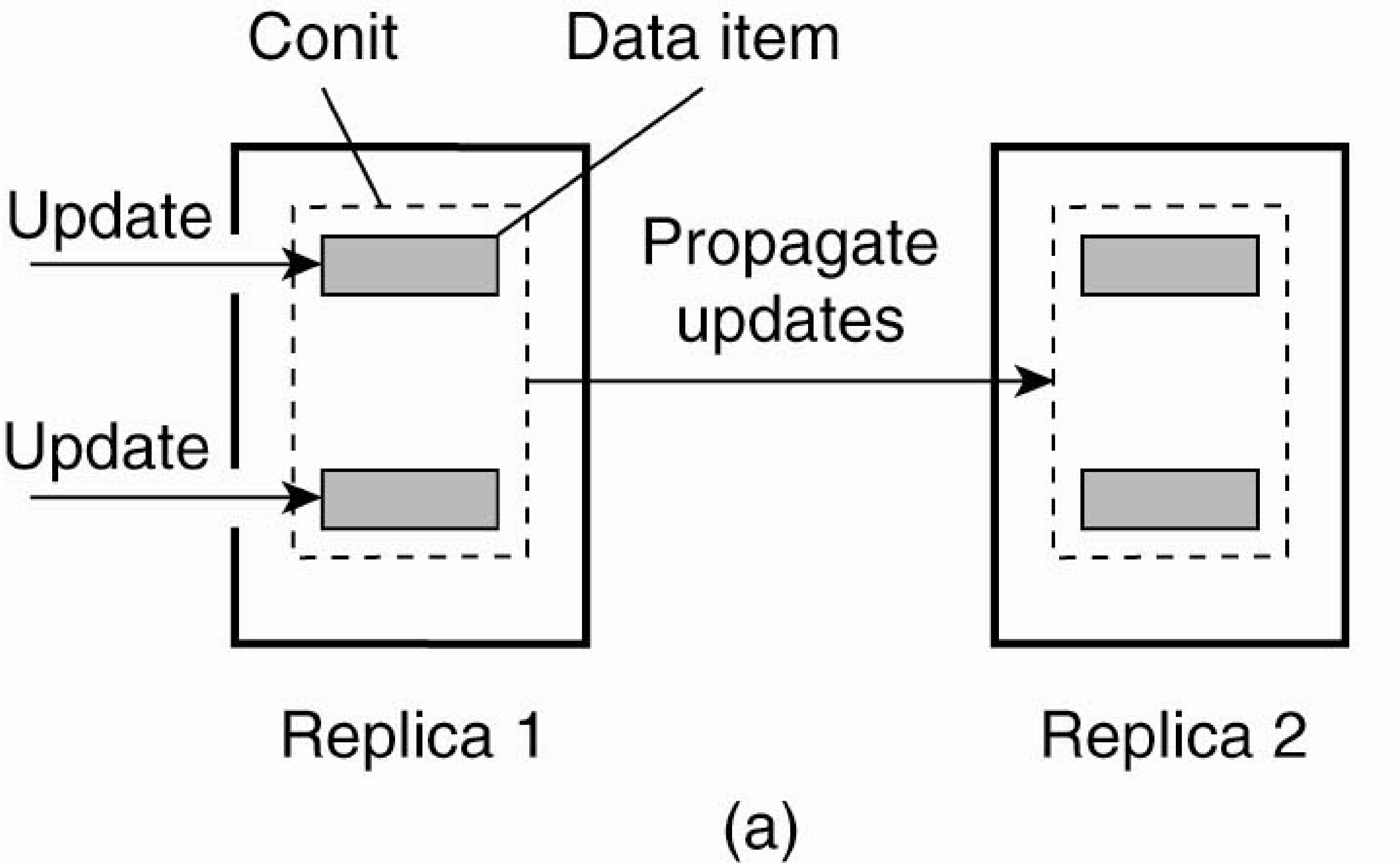
Operation		Result
< 5, B>	x := x + 2	[x = 2]
<10, B>	y := y + 5	[y = 5]

Vector clock B = (0, 11)
Order deviation = 2
Numerical deviation = (3, 6)

Commited op with permanent result
Tentative ops
after tentative ops
Numerical deviation concerning # of operations
Numerical deviation Concerning maximal diff.

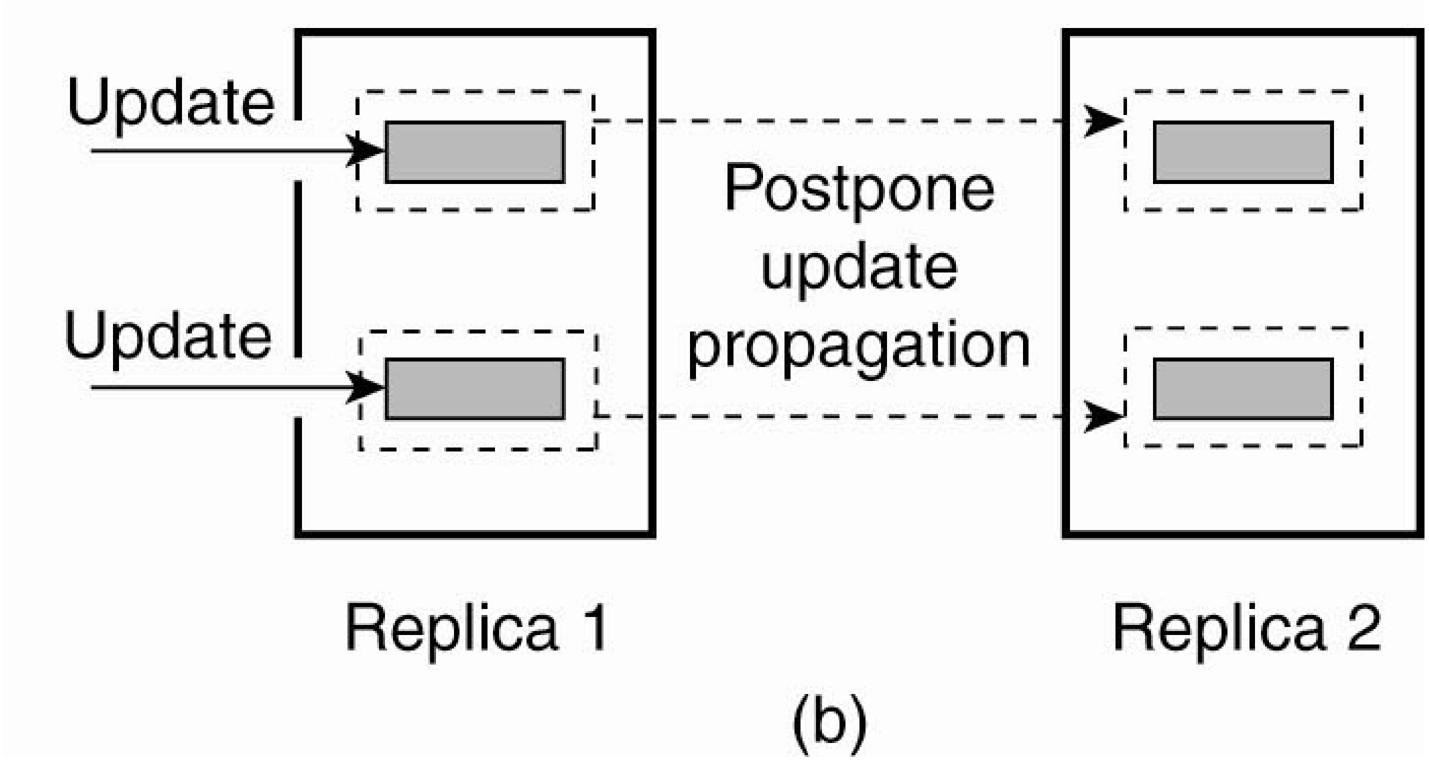
- Replica A has not yet seen 1 operation at B bringing its numerical deviation to 1
- The weight of this deviation can be expressed as the max. difference (5) between the committed values of A (2,0) and the not yet seen values having been updated at B (which results from y)
- Similar considerations with replica B

Continuous Consistency (4)



- Choosing the appropriate granularity for a conit
  (a) Two updates lead to update propagation
- Example “Update propagation rule”: When deviation in the number of update operations per conit is >1, then propagate

Continuous Consistency (5)



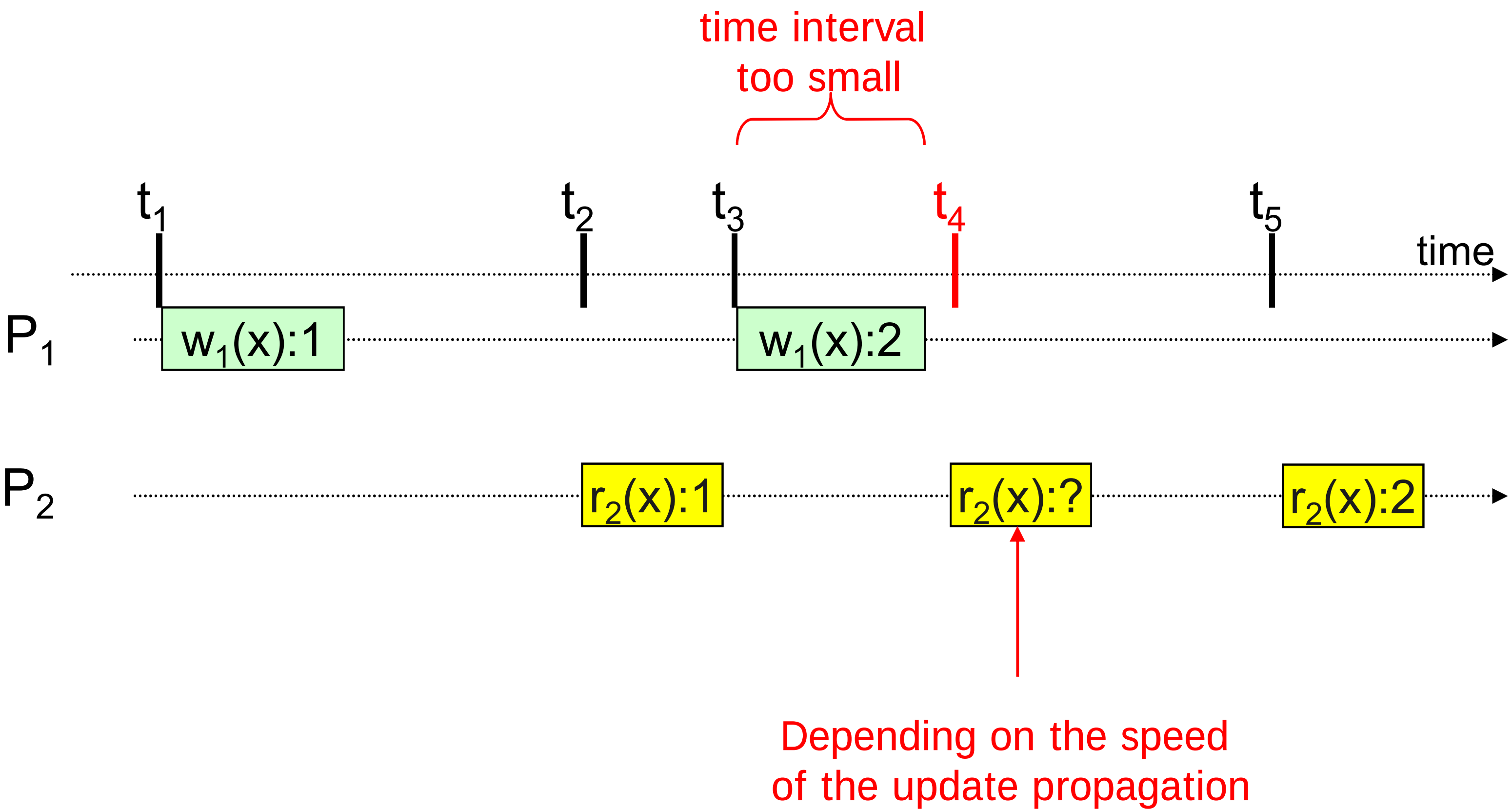
- Choosing the appropriate granularity for a conit (b) No update propagation is needed (yet), although 2 update operations have been done on replica 1 *(but not on the same conit)*

Consistent Order of Operations

Operations on a DDS

- **read:** $r_i(x)$**:b** client P_i reads from data item **x**, read returns value **b**
- **write:** $w_i(x)$**:a** client P_i writes to data item **x** setting it to the new value **a**
- Operations not *instantaneous*
  - Time of issue (when request is sent by client)
  - Time of execution (when request is executed at a replica)
  - Time of completion (when reply is received by client)

Example


time interval
too small
t_1
t_2
t_3
t_4
t_5
time
P_1
$w_1(x){:}1$
$w_1(x){:}2$
P_2
$r_2(x){:}1$
$r_2(x){:}?$
$r_2(x){:}2$
Depending on the speed
of the update propagation

Consistency

- Updates & concurrency result in conflicting operations
- Conflicting operations:
  - Read-write conflict
  - Write-write conflict
- Consistency:
  - Conflicting operations must be executed in same order “everywhere”
  - Partial order: order of a single client’s operations must be maintained
  - Total order: interleaving of all clients’ conflicting operations
  - Program order must be maintained
  - Data coherence must be respected

Example

```
Process P1 (on node 1):
  … x = 0;…
Process P2 (on node 2):
  … x = 1;…
Process P3 (on node 3):
  … print(x); … print(x);…
Process P4 (on node 4):
  … print(x); … print(x);…
```

- Do not accept: 01 and 10 together

Goals of Consistency Models

Expectation:

An application programmer expects a behavior of a DDS being similar to that of a

one copy data store

For sake of simplicity let’s assume, reads and writes are

atomic

Goal:

Each process *accessing shared data* wants to read a value, that is as “*up-to-date*” as possible

Comment:

Ideally each read gets the value of the last write, however, due to propagation delays that is often not possible

Data Centric Consistency Models

Strong Ordering

Weak Ordering

Main Examples:

DDB, DFS, DSM

(in a later lecture)

Data-Centric Consistency Models

- Strong ordering:
  - Strict consistency
  - Linear consistency
  - Sequential consistency
  - Causal consistency
  - FIFO or PRAM consistency

Impetus on individual operations

- Weak ordering:
  - Weak consistency
  - Release consistency
  - Entry consistency

Impetus on group of individual operations

Strict Consistency

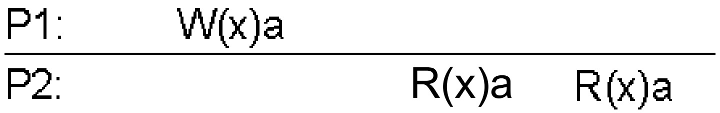
Definition:

A DDS is *strict consistent* if every read on a data item returns the value corresponding to the result of the *most recent write on x*, regardless of the location of the replica *x* or of the processes doing the reads or writes

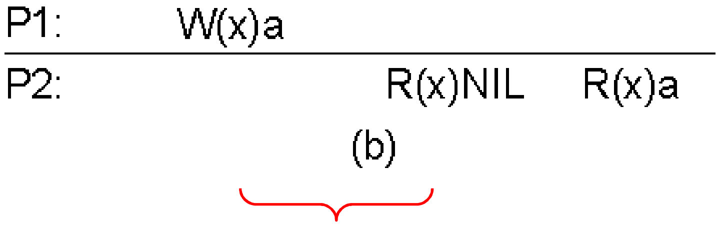
Analysis:

1. In a single processor system strict consistency is for free, it's the behavior of main memory with atomic reads and writes
2. However, in a DSM without the notion of a global time it is hard to determine what is the *most recent write*

Strict Consistency



(a)



(b)

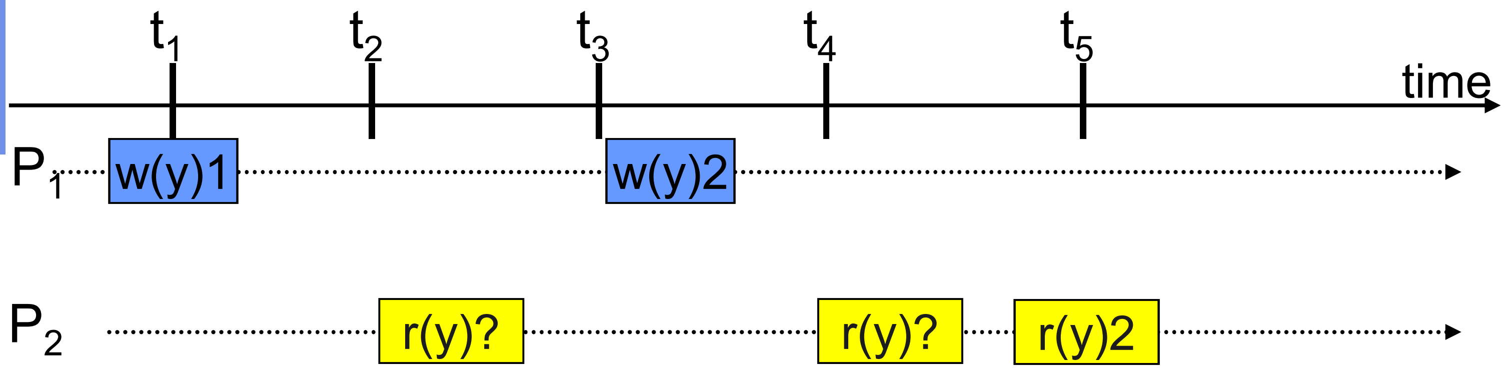
If time interval
is sufficient large

Behavior of two processes, operating on the same data item x of a

a) strictly consistent DDS

b) DDS that is *not strictly consistent*

Strict Consistency Problems

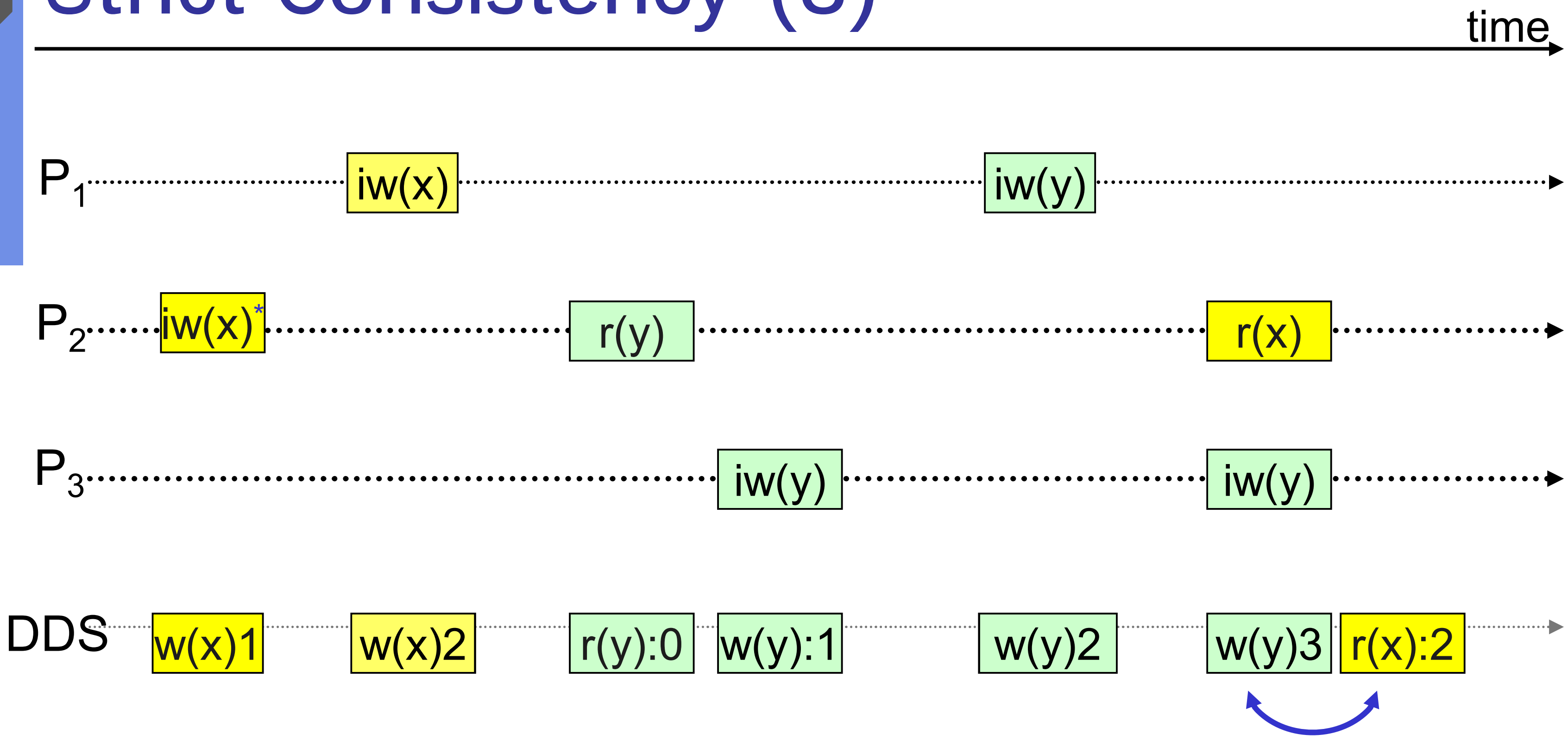


Assumption: y = 0 is stored on node 2,
P_1 and P_2 are processes on node 1 and 2,

Due to message delays r(y) at
$t = t_2$ may result in 0 or 1 and at
$t = t_4$ may result in 0, 1 or 2

Furthermore: If y migrates to node 1 between t_1 and $t_3+\varepsilon$
then r(y) issued at time t_2 may even get value 2
(i.e. “back to the future”).

Strict Consistency (3)



operations are interchangeable

Result:

Global timing order of all distinguishable accesses remains preserved

Contemporary *non conflicting* accesses can be ordered arbitrarily

*Legend:

iw(xyz) is incrementing an integer xyz, having been initialized with 0.

Sequential Consistency[1]

Definition:

A DDS offers *sequential* consistency if *all processes* see the same *order* of accesses to the DDS, whereby

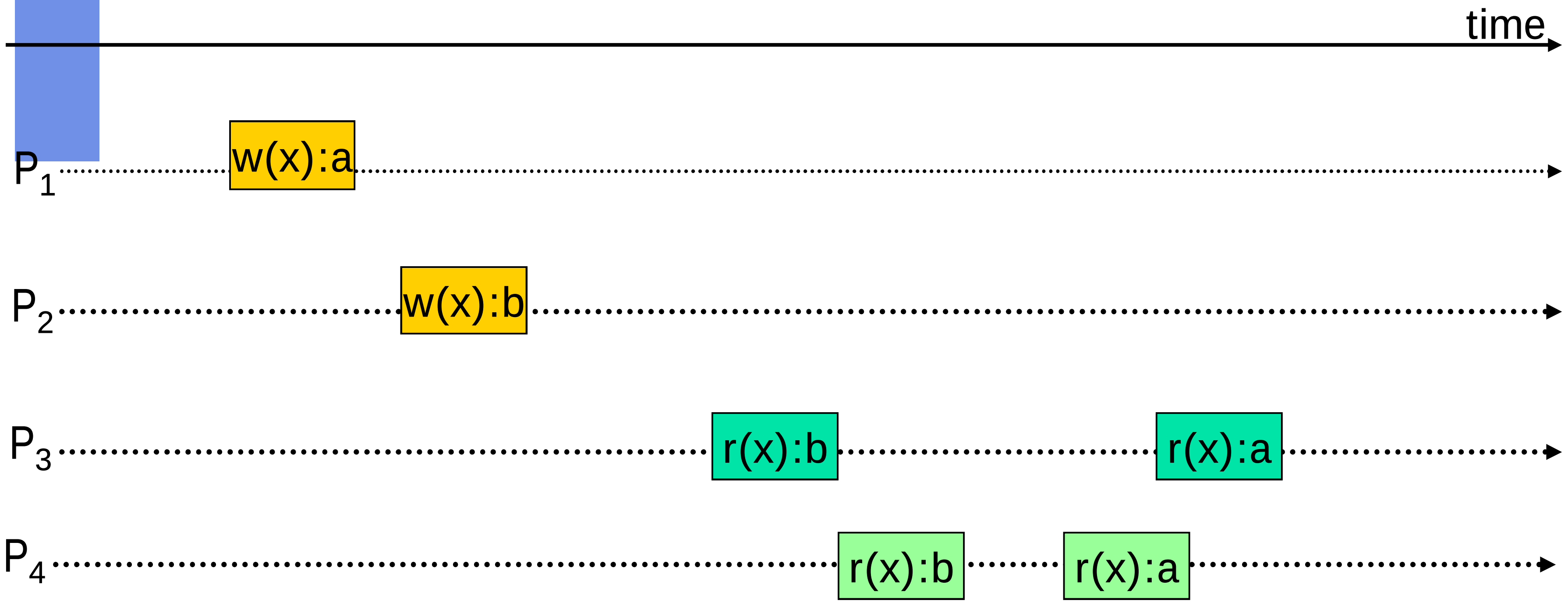
1. reads and writes of an individual process occur in their program order
2. reads and writes of different processes occur in some sequential order as long as interleaving of concurrent accesses is *valid*.

[1] Leslie Lamport 1979 to define the expected behavior of SMPs, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs", IEEE Transactions on Computers.

Sequential Consistency

- Sequential consistency is weaker than strict consistency
- No global timing ordering is required
- Possible Implementation:
  - Writes in same sequential order in all processes (they must agree upon the order)
  - Each process sees all writes from all other processes
  - Time to complete a write does no affect consistency (however, might affect the user)
  - Before we do the *next write* we must *have completed* the *previous one*

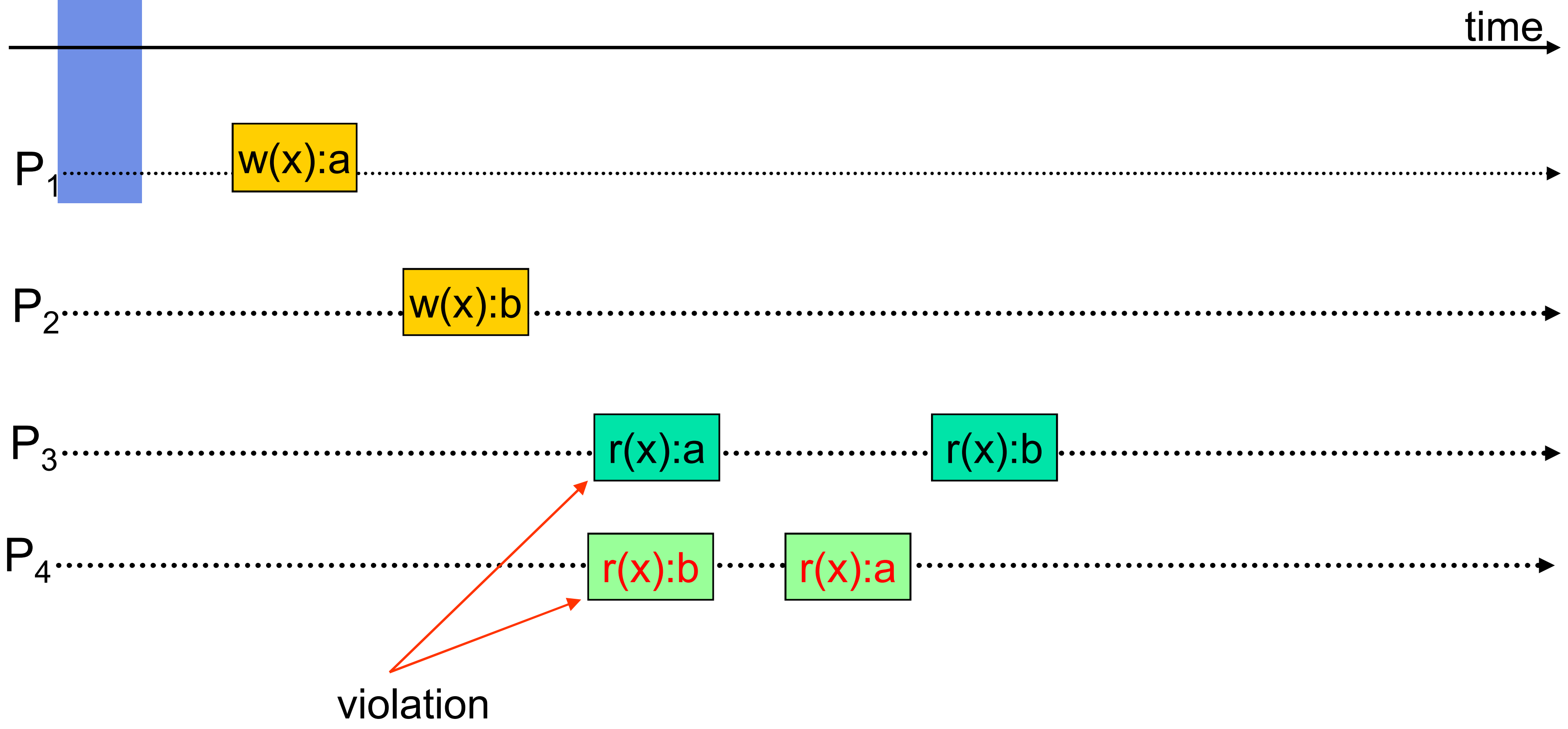
Sequential Consistency (2)



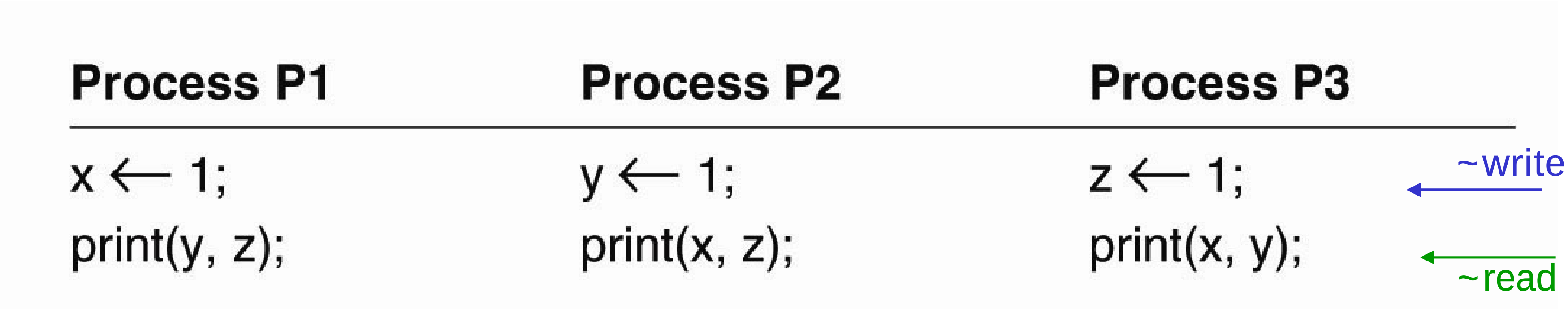
1. Each process sees all accesses in the same order, even though ∃ no strict consistency

2. With rubber band method you can establish a sequential order

Non-Sequential Consistent



Sequential Consistency



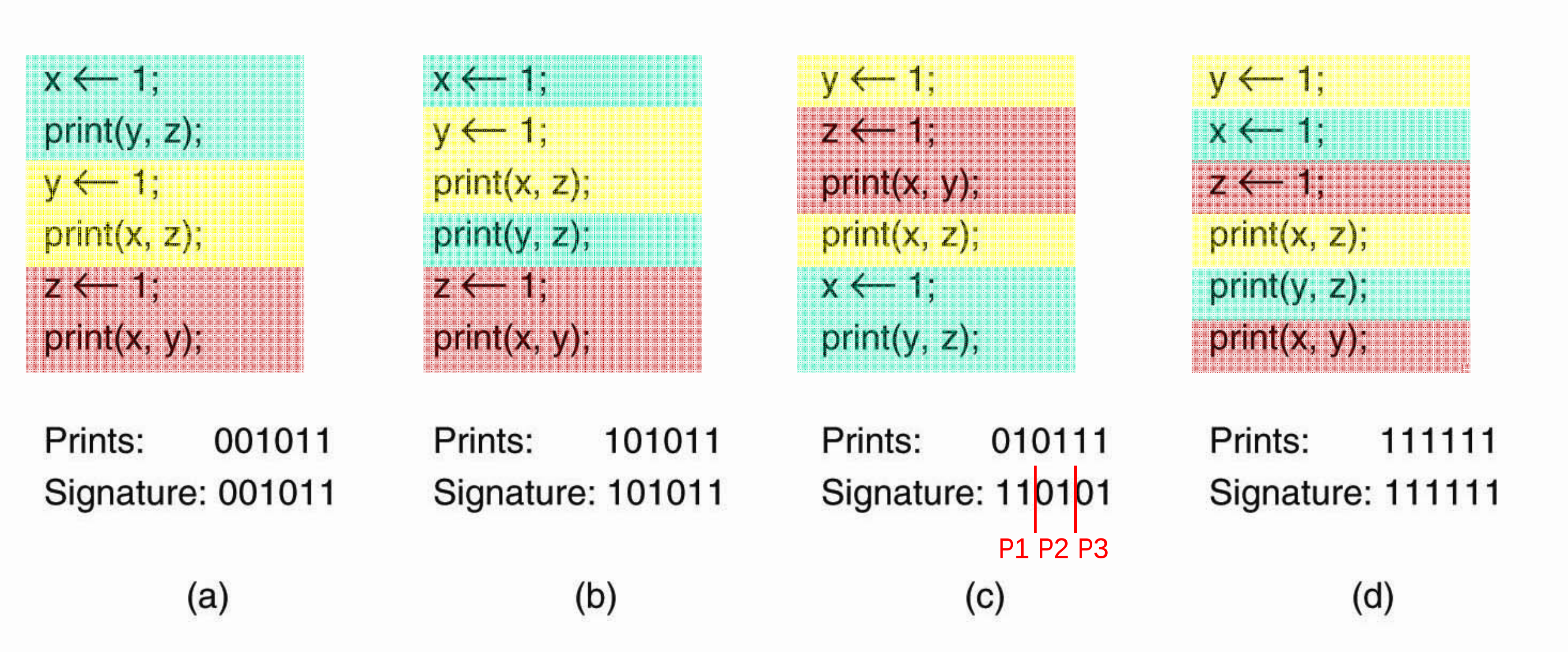
Process P1	Process P2	Process P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

~write

~read

- Three concurrently-executing processes

Sequential Consistency



```
x ← 1;
print(y, z);
y ← 1;
print(x, z);
z ← 1;
print(x, y);
```

Prints: 001011
Signature: 001011

(a)

```
x ← 1;
y ← 1;
print(x, z);
print(y, z);
z ← 1;
print(x, y);
```

Prints: 101011
Signature: 101011

(b)

```
y ← 1;
z ← 1;
print(x, y);
print(x, z);
x ← 1;
print(y, z);
```

Prints: 010111
Signature: 110101

P1 P2 P3

(c)

```
y ← 1;
x ← 1;
z ← 1;
print(x, z);
print(y, z);
print(x, y);
```

Prints: 111111
Signature: 111111

(d)

- 4 (out of 90) valid execution sequences for the processes of previous example
  - (a) First P1, then P2, then P3 ...
  - (b) P1 starts, then P2 starts and completes, then P1 completes, then P3
  - (c) ...
- Signature represents the concatenated printouts in order P1, P2, P3

Summary: Sequential Consistency

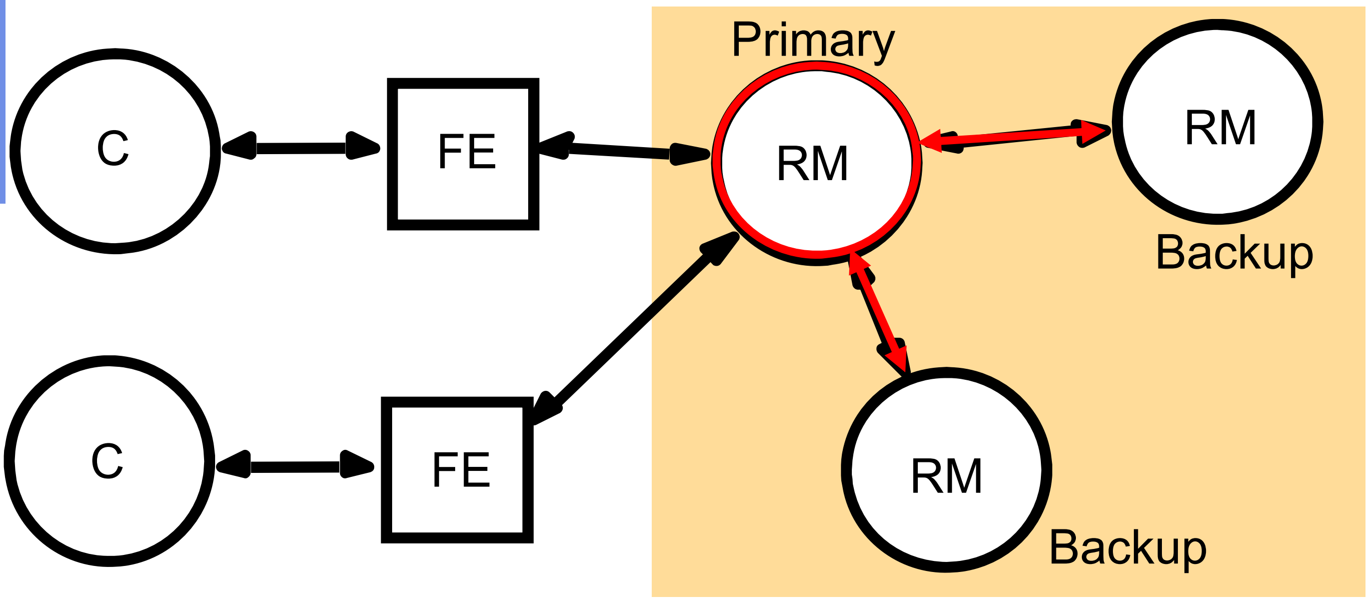
- Any *valid* interleaving of read and write operations of concurrent processes at different sites is acceptable for a sequential consistent DDS, as long as all processes see the same interleaving of operations

- Interesting problem:

  How to achieve sequential consistency?

- Read: L. Lamport: “How to make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs”, IEEE Trans. on Comp., Sept. 1979

Primary Backup Replication for Seq. Cons.



passive replication

1. Frontend sends write request with an unambiguous ID to primary server
2. Primary checks whether it already has performed the request
3. Primary fulfills write locally and stores the answer
4. Primary multicast the update to all other replicas. Backups send their Ack.
5. Having received all Ack. Primary sends back the answer to frontend

“Write Through” Update Propagation

- Before performing a write operation a process first sends a completely sorted “update-message” via multicast to all replicas (incl. to itself)

- All processes agree on a order of update-operations if there are multiple “update-messages”

- All replicas will see the same sequence of updates, ⇒ sequential consistency

Linear Consistency

Definition:

A DDS is said to be linear consistent when each operation has time-stamp TS and the following holds:

1. DDS is sequential consistent
2. In addition, if

   $TS_{OP1}(x) < TS_{OP2}(y)$, then

   $O_{P1}(x)$ should precede $O_{P2}(y)$

(It’s assumed that the time stamps are precise enough)

Linear Consistency

Assumption:

Each operation is assumed to receive a *time stamp* using a globally available clock, but with *only finite precision*, e.g. some loosely coupled synchronized *local* clocks.

Linear consistency is stronger than sequential one, i.e. a linear consistent DDS is also sequentially consistent.

Corollar: With linear consistency no longer each valid interleaving of reads and writes is allowed, but this ordering also must obey the order implied by the inaccurate time-stamps of these operations.

Linear consistency is harder to achieve (see Attiya and Welch:
“Sequential Consistency versus Linearizability”,
ACM Trans. Comp. Syst., May 1994)

Causal Consistency (1)

Definition:

A DDS is said to provide *causal consistency* if, the following condition holds:

> *Writes* that are *potentially causally related** must be seen by all tasks in the *same order*. Concurrent writes may be seen in a different order on different machines.

- If event B is potentially caused by event A, causality requires that everyone else also sees first A, and then B.

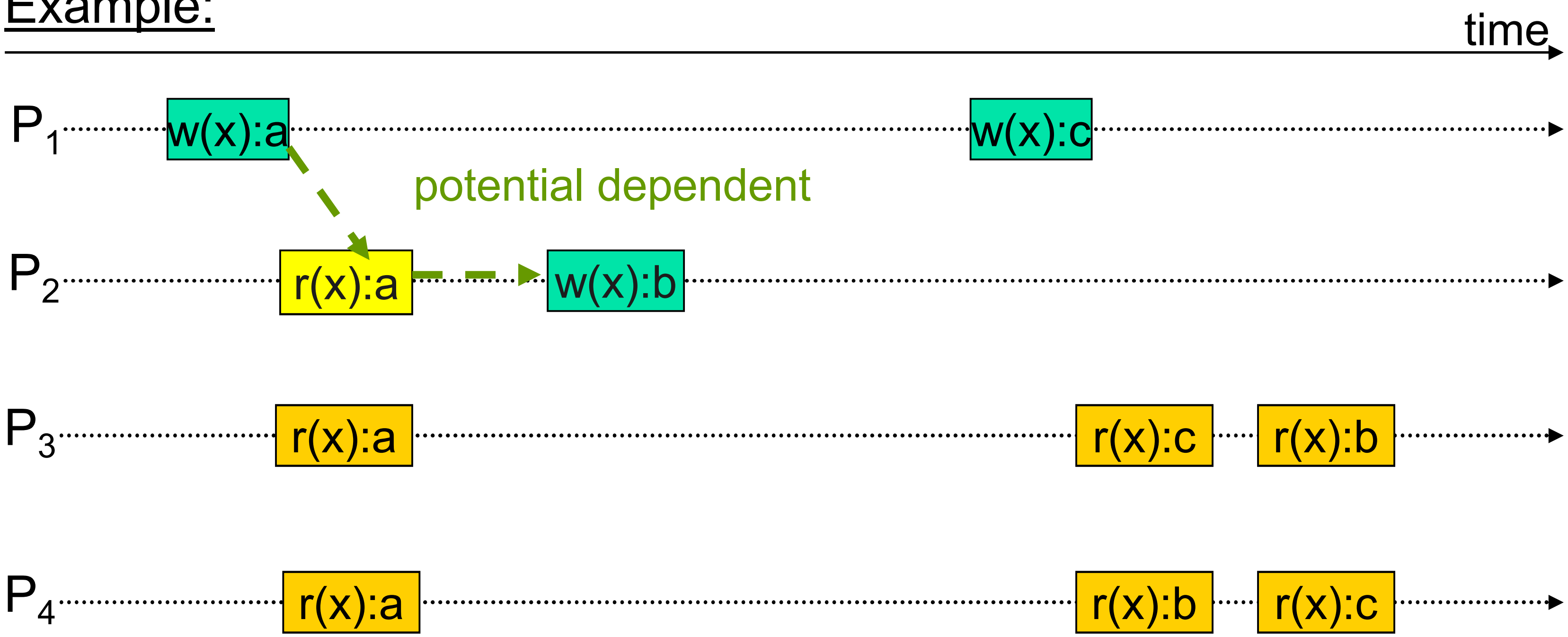
Causal Consistency (2)

Definition:
write2 is potentially dependent on write1, when there is a read between these 2 writes which may have influenced write2

Corollary:
If write2 is potential dependent on write1 $\Rightarrow$ the only valid and consistent sequence is: write1 $\rightarrow$ write2.

Causal Consistency (3)

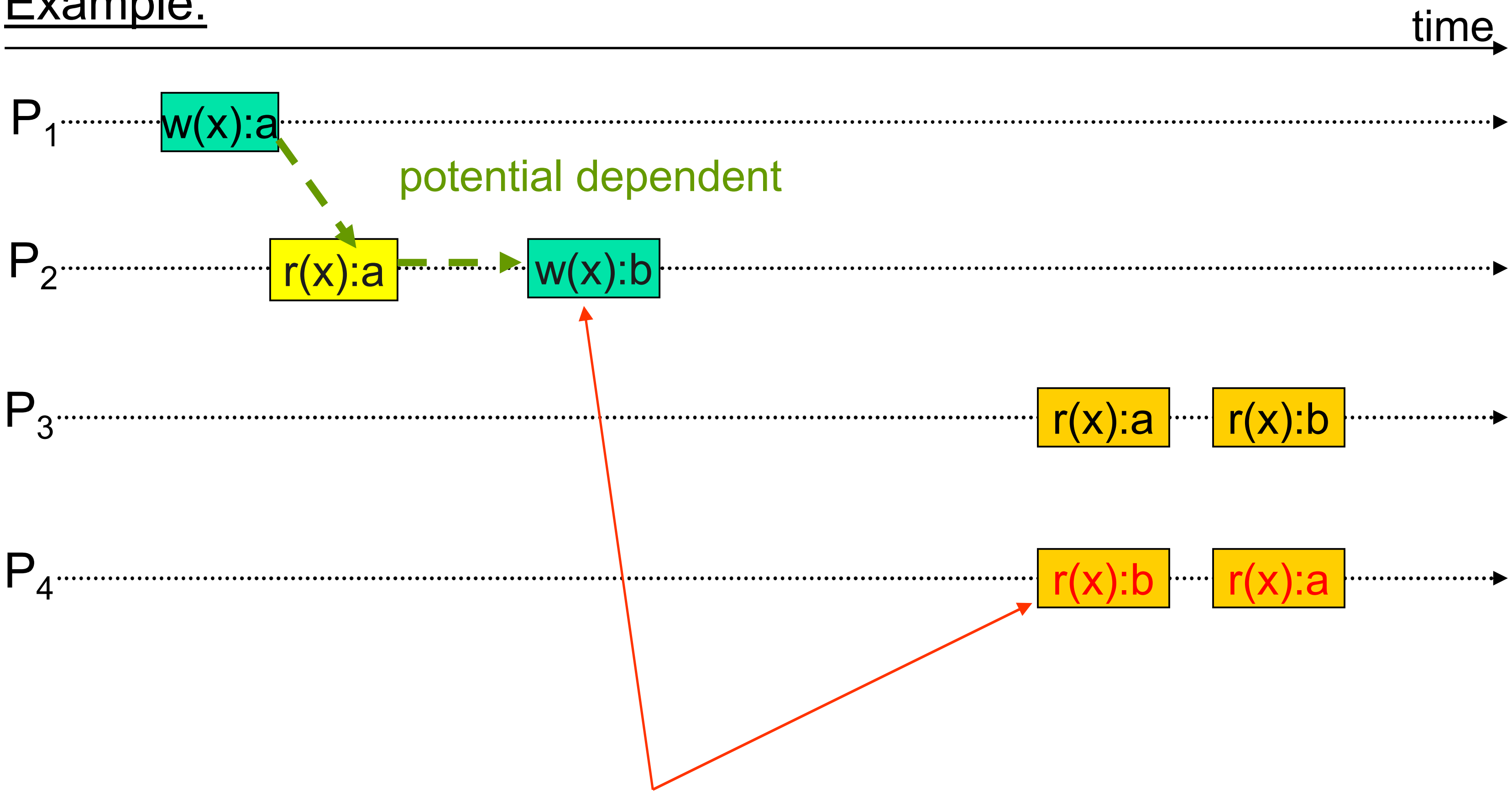
Example:



Result: (causal consistent, but not sequentially consistent)
P3 and P4 can see updates on the shared variable x in a different order, because w(x):b and w(x):c are concurrent

Causal Consistency (4)

Example:



Violation of causal consistency

Causal Consistency (5)

Implementing causal consistency requires keeping track of which processes have seen which writes.

Construction and maintenance of a dependency graph,
Expressing which operations are causally related
(using vector time stamps)

FIFO or PRAM Consistency*

Definition:

DDS implements *FIFO consistency*, when all writes of one process are seen in the same order by all other processes, i.e. they are received by all other processes in the order they were issued.

However, writes from different processes may be seen in a different order by different processes.

Corrolar:

Writes on different processes are concurrent

*Pipelined RAM see Lipton and Sandberg

Implementation of FIFO Consistency

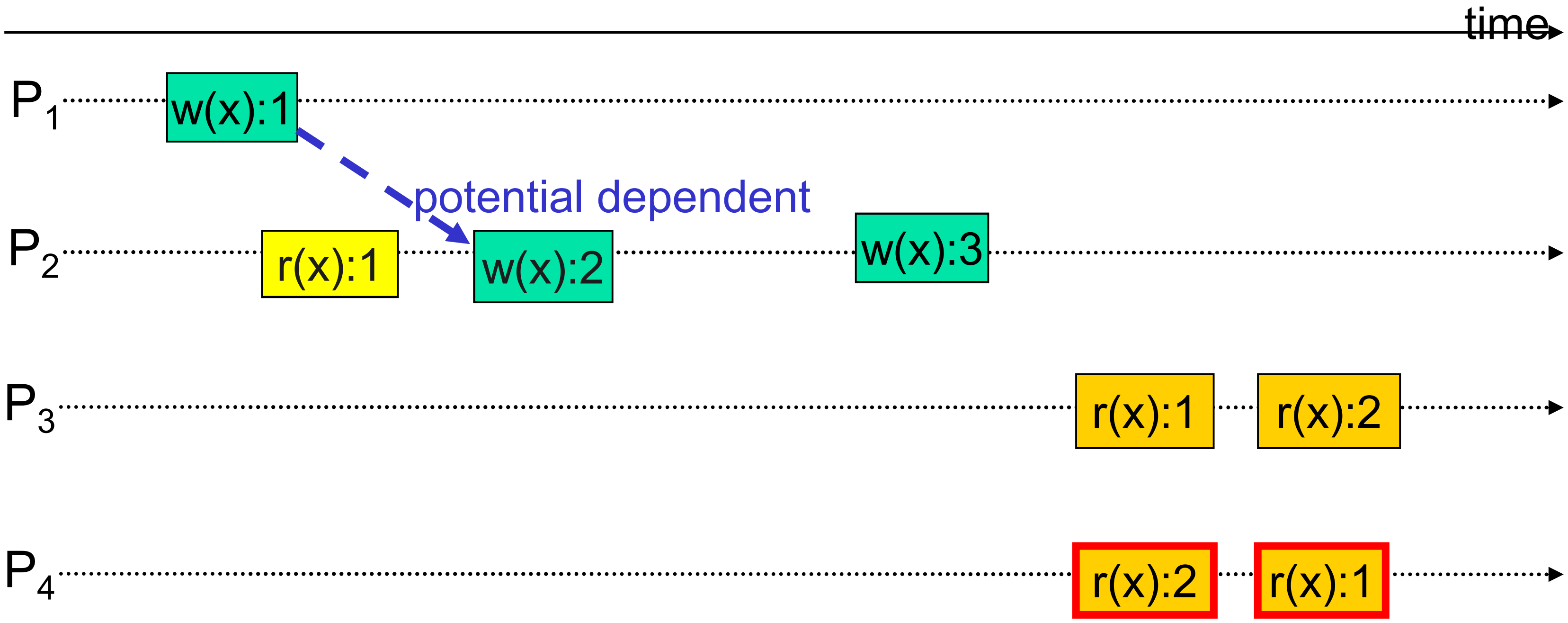
Tag each write-operation of every process with:

(PID, sequence number)

FIFO or PRAM Consistency (2)

Example:

Both writes are seen on processes P3 and P4 in a different order, they still obey FIFO-consistency, but not causal consistency because write 2 is dependent on write1(⇒ r(x):1 can not be read by P_3 or P_4)



FIFO Consistency (4)

Example: Two concurrent processes with variable x,y = 0;

Process P1	Process P2
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);

Result:
With sequential consistency one might conclude:
Either P1 or P2 or no process is aborted.

With FIFO consistency both tasks may be aborted,

Assume:
P1 sees y via read(y):0 before P1 sees P2’s write(y):1 and
P2 sees x via read(x):0 before P2 sees P1’s write(x):1

Weak Consistency

Review: FIFO consistency reflects the correct sequence of all writes of each process

Observation:

In many applications the exact sequence of single operations is not that important, but

programmers want to influence the impetus of a group of operations, e.g. updating a data record or a complete list of data items

⇒

Idea: Group relevant operations together in a "*synchronized section*" and synchronize after the last write respectively before the first read

Weak Consistency

- Every read or write operates *only locally*
- We introduce synchronization variables
  - No relationship with normal variables
  - Call **`synchronize()`** synchronizes the local copies
- Each synchronize call serves as a *synchronization event*, i.e. it
  1. broadcasts its local updates to all other participating processes at different nodes
  2. collects all remote updates from the remote copies before starting a new synchronization section ⇒ successive reads read up-to-date values

Weak Consistency

Definition:

DDS implements weak consistency, if the following holds:

- Accesses to synchronization variables* obey sequential consistency
- No new *synchronize* is allowed to be performed until *all previous writes* have been completed *everywhere due to the current synchronize*
- No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed

*Normal and synchronization variables, Dubois et al: Synchronization, coherence and event ordering in multiprocessors, Distr. Comp., 1993

Interpretation (1)

- A synchronization variable S knows just one operation: **`synchronize(S)`** responsible for all local replicas of the DDS
- Whenever a process calls **`synchronize(S)`** its local updates will be propagated to all replicas of the DDS and all updates of the other processes will be applied to its local replica
- All tasks see all accesses to synchronization-variables in the same order
  - In other words, if task P1 calls synchronize(S1) at the same time that task P2 calls synchronize(S2), the effect will be the same as if either synchronize(S1) preceded synchronize(S2), or vice versa

Interpretation (2)

No data access allowed until all previous accesses to synchronization-variables have been done

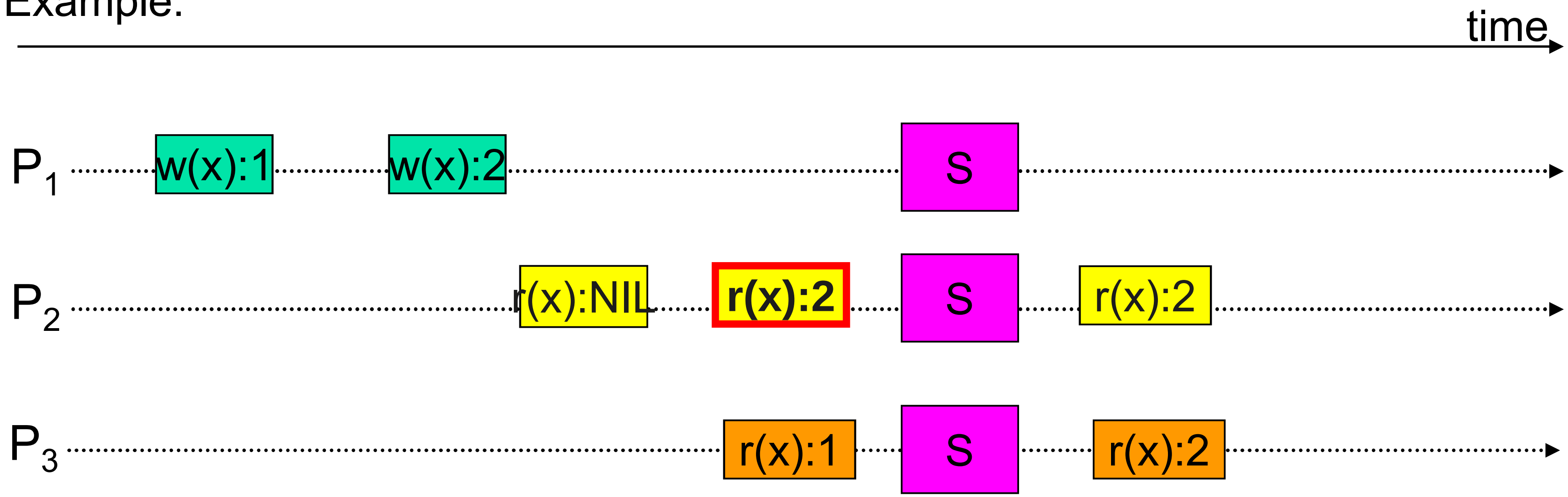
- By doing **`synchronize()`** before reading shared data, a task can be sure of getting “*up to date value*”
- Unlike previous consistency models “weak consistency” forces the programmer to collect critical operations all together

Example1: Weak Consistency

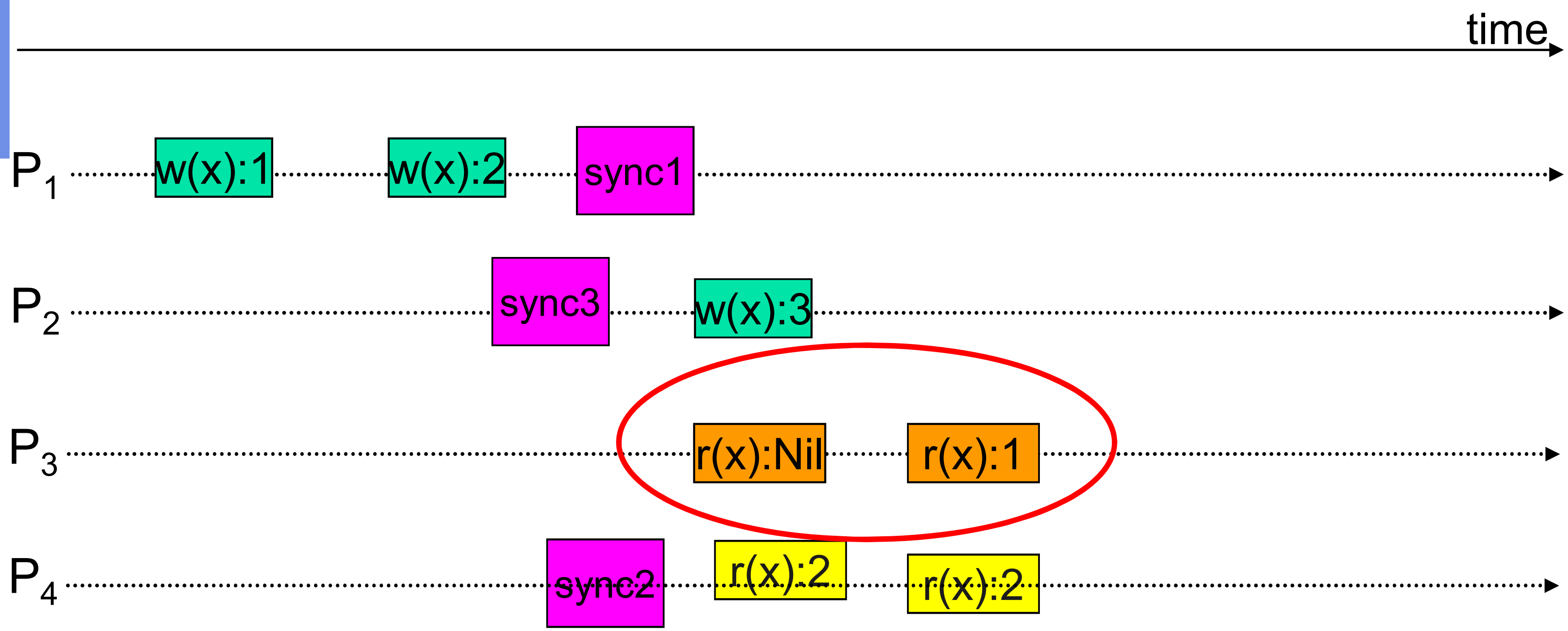
Via synchronization you can enforce that you'll get up-to-date values. Each process must synchronize if its writes should be seen by others.

Process requesting a read without a previous synchronization can get out-of-date values.

Example:



Example2: Weak Consistency?



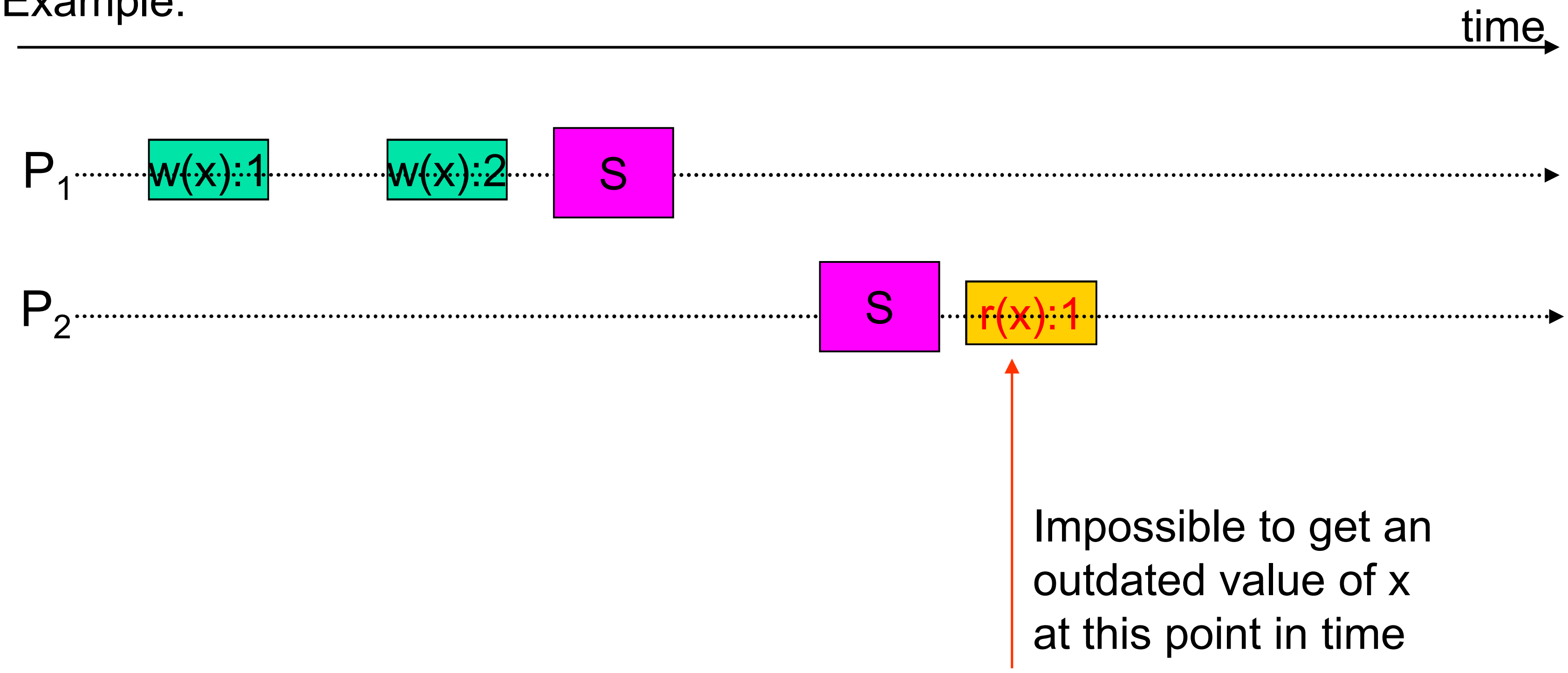
Assume:

sync1 < sync2 < sync3 according to *sequential consistency* ⇒

On all nodes this synchronization sequence can be observed

System with “No Weak Consistency”

Example:



Impossible to get an outdated value of x at this point in time

Possible Implementation

Simple implementation:

- A central synchronization server determines the sequence of synchronization operations
- Per synchronize:
  - Local data copy broadcasts all updated variable values to the centralized sync-server
  - Local data copy gets all update-values from all other data copies in the DDS

- *Try to find a more distributed solution!!*

Analysis: Weak Consistency

Whenever **`synchronize()`** is called, DDS can not distinguish between:

A: begin of a synchronized section SS due to a sequence of successive reads

B: end of a synchronized section SS due to a previous sequence of writes

- If we would distinguish between those two different synchronization events ⇒

we might find *more efficient solutions*

Release Consistency

Idea:

Distinguish between memory accesses in front of a CS **`acquire()`** and behind of a CS **`release()`**

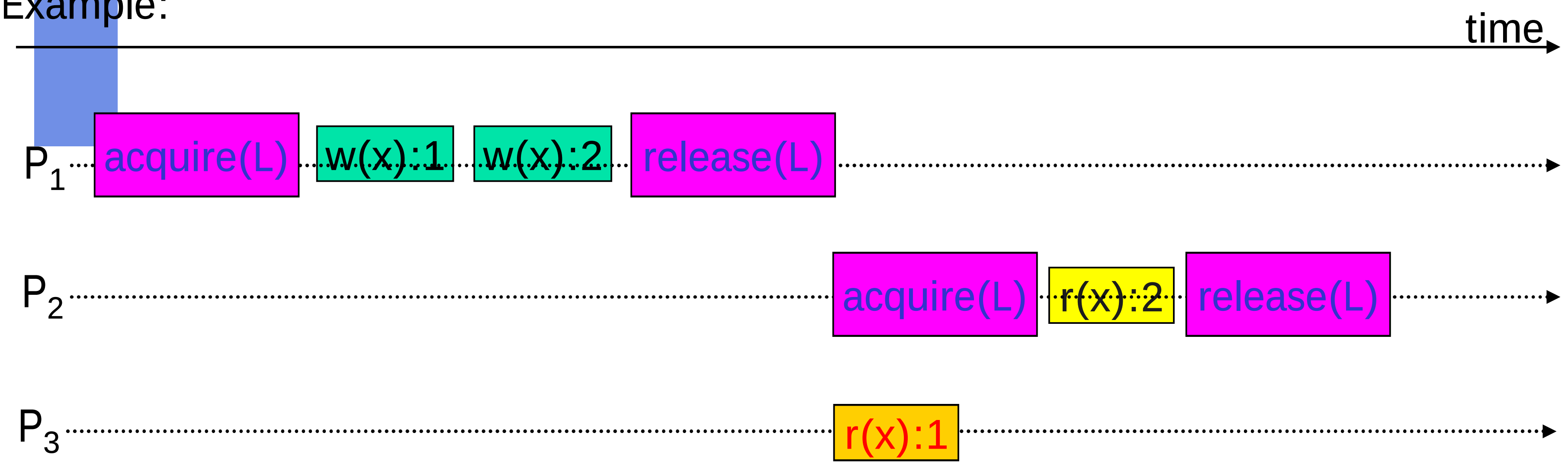
Design:

When an **`acquire()`** is done the calling process will be blocked until all its local data copies have been updated

When a **`release()`** is done, all the protected data that have been updated within the CS have to be propagated to all replicas

Release Consistency

Example:



Valid event sequence for release consistency, even though P_3 missed to use acquire and release

Remark: Acquire is more than a lock (enter_critical_section), it waits until all updates on protected data from other nodes are done to its local store before it enters the critical section

Release Consistency

Definition:

A DDS offers release consistency if the following three conditions hold:

1. Before a read or write operation on shared protected data is performed, all operations needed to do the previous acquire must have completed successfully
   - The local copies of the protected data are up to date, i.e. they are consistent with their remote data
2. Before a release is allowed to be performed, all previous (reads and) writes by the process must have been completed and the updated values are sent to the replicas
3. Accesses to synchronization variables are FIFO consistent.

Release Consistency

Simplified implementation:

- Centralized coordinator:
  - `acquire()` corresponds to lock CS
    - Centralized coordinator knows about all locks
  - `release()` sends all updated values to the coordinator which forwards them to all other participants

Lazy Release Consistency[*]

Problems with “eager” release consistency:
When a release is done, the process doing the release pushes out *all the modified data to all processes* that already have a copy and thus might potentially read them in the future.

There is no way to tell if all the target machines will ever use any of these updated values in the future
⇒
above solution is a bit inefficient, too much overhead.

[*] Keleher, Cox, Zwaenepol: Lazy Release Consistency”, 19th Symp on Computer Architecture, ACM, 1992

Lazy Release Consistency

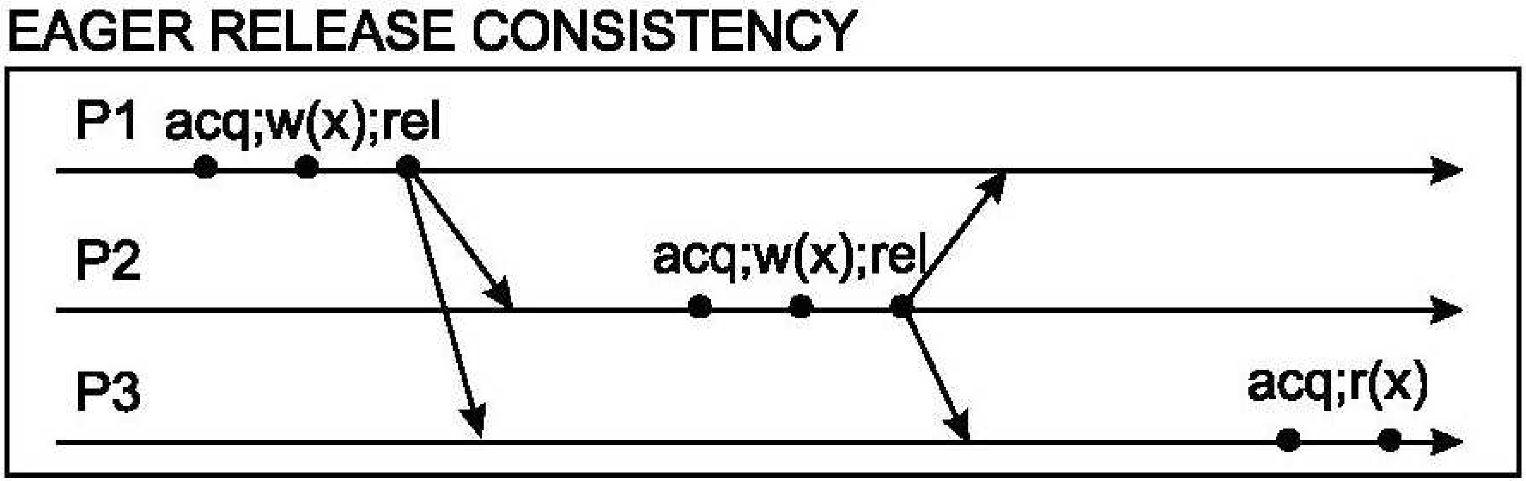
With “lazy” release consistency nothing is done at a **`release().`**

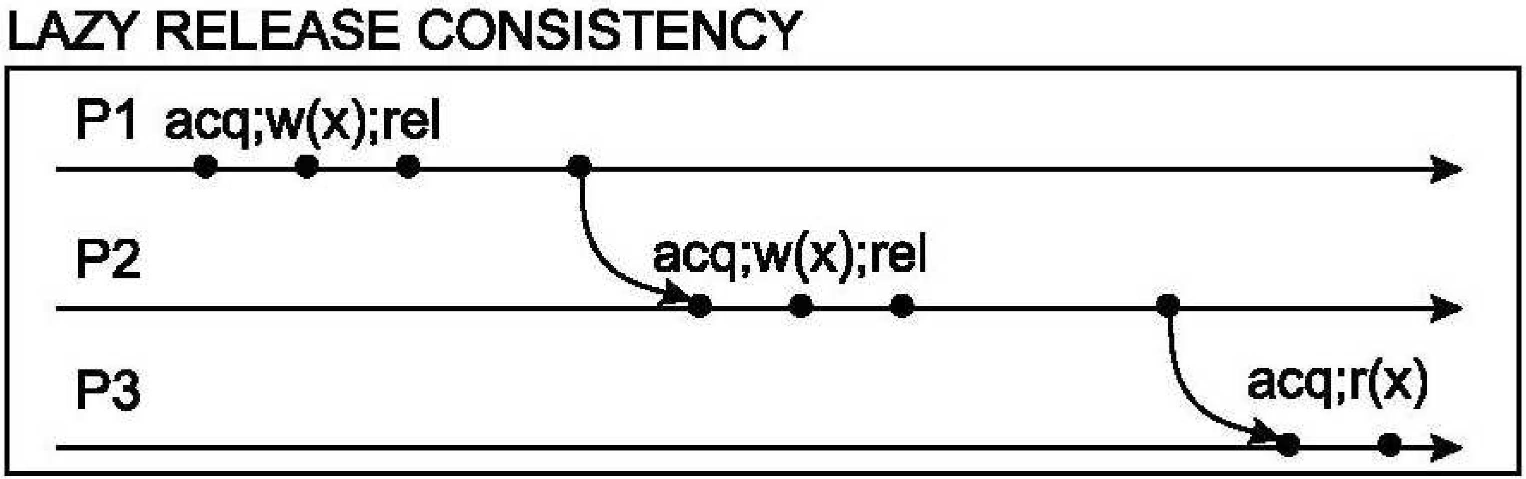
However, at the next **`acquire()`** the processor determines whether it already has all the data it needs.

Only when it needs updated data, it needs to send messages to those places where the data have been changed in the past.

Time-stamps help to decide whether a data is out-dated

Eager versus Lazy Release Consistency





Entry Consistency

Unlike release consistency, entry consistency requires that *every ordinary shared* variable can be protected by a distinct *synchronization variable*

Before a process can enter a critical section it has to acquire all needed synchronization variables

When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent

A list of shared variables may be assigned to a synchronization variable (to reduce overhead)

Entry Consistency*

Definition:

A DSM exhibits *entry consistency* if the following holds:

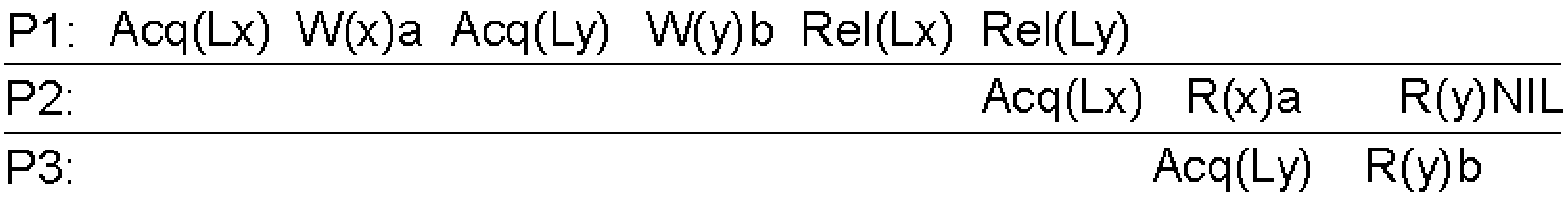
1. An acquire access of a sync-variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process
2. Before an exclusive mode access to a sync-variable by a process is allowed to perform with respect to that process, no other process may hold this sync-variable, not even in non-exclusive mode
3. After an exclusive mode access to a sync-variable has been performed, any other process' next *nonexclusive* mode access to that sync-variable may not be performed until it has performed with respect to that variable's owner

*Bershad, Zerkauskas, Sawdon: "The Midway DSM System",
Proceedings of the IEEE COMPCON Conf., 1993",

How to synchronize in Entry Consistency?

- Every synchronization variable has a current owner
- An owner may enter and leave critical sections protected by this synchronization variable as often as needed without sending any coordination message to the others
- A process wanting to get a synchronization variable has to send a message to the current owner.
- The current owner hands over the synchronization variable all together with all updated values of its previous writes
- Multiple reads in the non-exclusive read modus are possible.

Entry Consistency



```
P1:  Acq(Lx)  W(x)a  Acq(Ly)  W(y)b  Rel(Lx)  Rel(Ly)
P2:                                           Acq(Lx)  R(x)a     R(y)NIL
P3:                                                 Acq(Ly)  R(y)b
```

- A valid event sequence for entry consistency

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	Writes from every single process are seen in their true order by all processes. Writes from different processes may be seen in different order on each process.

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

a) Strong consistency models (no synchronization)

b) Weak consistency models (with explicit synchronization)

Coherence ~ versus Consistency Models

- Consistency models deal with right ordering of operations of different processes on a set of data
- A coherence model deals with the right ordering of concurrent writes to one single data item
  - Sequential consistency is the most popular model for data coherence

Client Centric Consistency

Motivation

Eventual Consistency

Monotonic Reads

Monotonic Writes

Read-your-Writes

Writes-follow-Reads

Client Centric Consistency

Up to now:

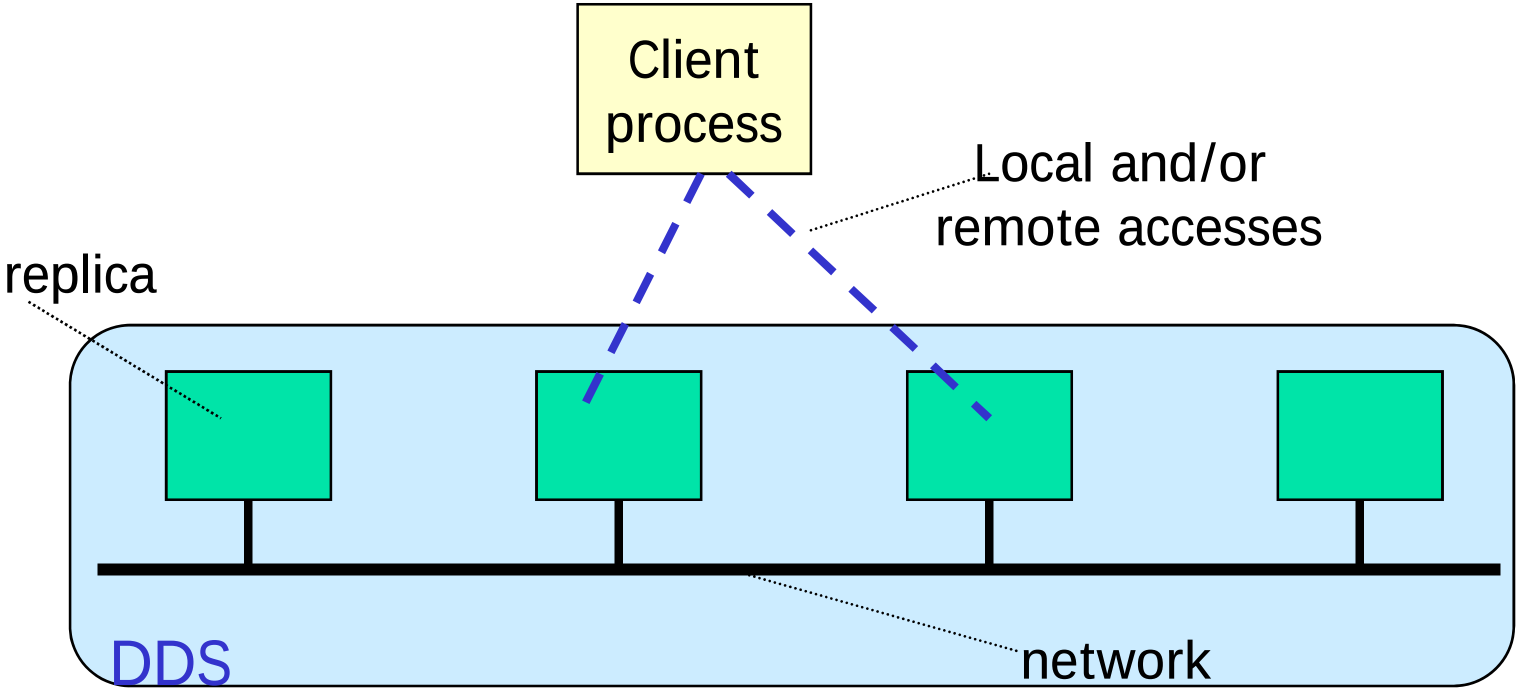
- System wide consistent view on the DDS* independent of number of involved processes and potentially concurrent updates
  - Atomic operations on DDS
  - Processes can access local copies
  - Updates have to be propagated, whenever it is necessary to fulfill the requirements of the consistency model (e.g. release consistency)
- *∃ weaker consistency models?*

*DDS = Distributed Data Store

Client Centric Consistency

- Provides guarantees about ordering of operations only for a single client, i.e.
  - Effects of an operations depend on the client performing it
  - Effects depend on the history of client's operations
  - Applied only when requested by the client
  - No consistency guarantees for interleaved accesses of different clients
- Application:
  - Rare and/or no concurrent updates of the data, e.g.
    - Each domain of DNS has an authority, that is allowed to update its entries, i.e. no write-write conflicts
    - Secondary DNS-server gets new name-bindings after a while
  - A Webpage is only update by its webmaster
    - Stale data can be accepted for a while

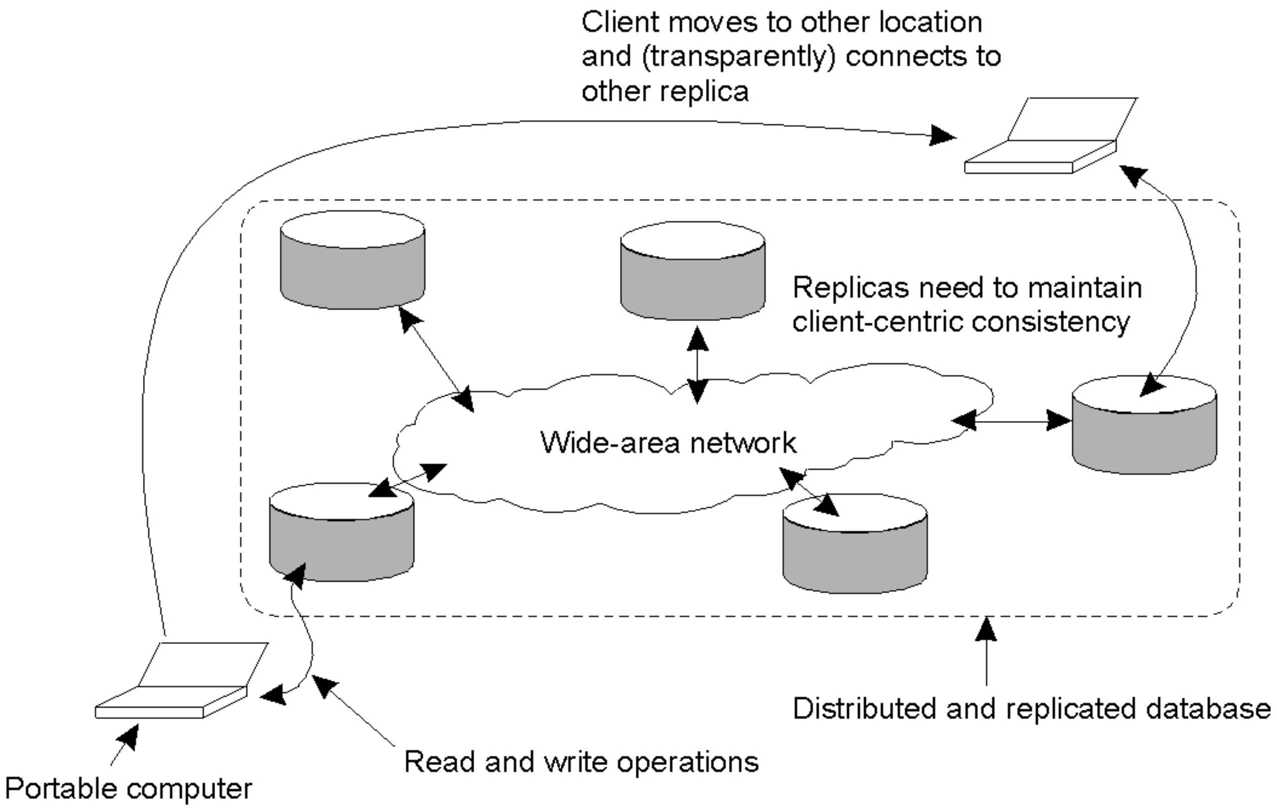
Eventual Consistency



Goal: Weaker, i.e. less restricted consistency models

Idea: Data of DDS will become consistent after “some time”

Example: Eventual Consistency


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

- Mobile user accessing different replicas of a distributed database crossing domain borders
- First ideas in Terry: “The Case for Non-transparent Replication: Examples from Bayou”

Eventual Consistency

If updates do not occur for a long period of time, all replicas will gradually become consistent

- Requirements:
  - Few read/write conflicts
  - No write/write conflicts, because “each data item” has an owner who is the only subject being allowed to update that data item
- Examples:
  - DNS:
    - No write/write conflicts
    - Updates (e.g. new entries) are slowly (1 – 2 days) propagating to all caches
  - WWW:
    - Few write/write conflicts (webmaster is the only writer)
    - Mirrors eventually updated
    - Cached copies (browser or Proxy) replaced

Implement Eventual Consistency

How to propagate updated data?

- Push model
  - After a timeout/periodically send your updates to all other replicas
- Pull-model
  - After timeout/periodically try to get the newest data from all replicas, e.g. web-cache
- Push-Pull model
  - Mutual actualization between the replicas
  - Example: DNS
    - Notification message that zones have changed in DNS
    - Via push updates from primary to secondary

4 Client Centric Consistency Models[1]

- Monotonic Reads
- Monotonic Writes
- Read Your Writes
- Writes Follow Reads

[1] Terry et al: “The Case for Non-transparent Replication: Examples from Bayou”, IEEE Data Engineering, Vol. 21, 1998

Client Centric Consistency Models

Assumptions:

- DDS is physically distributed across different nodes
- A process P -accessing the DDS- always connects to a local, often the nearest replica
- Each data item[1] has one single owner, being the only subject who is allowed to update this data

⇒ no write/write conflicts

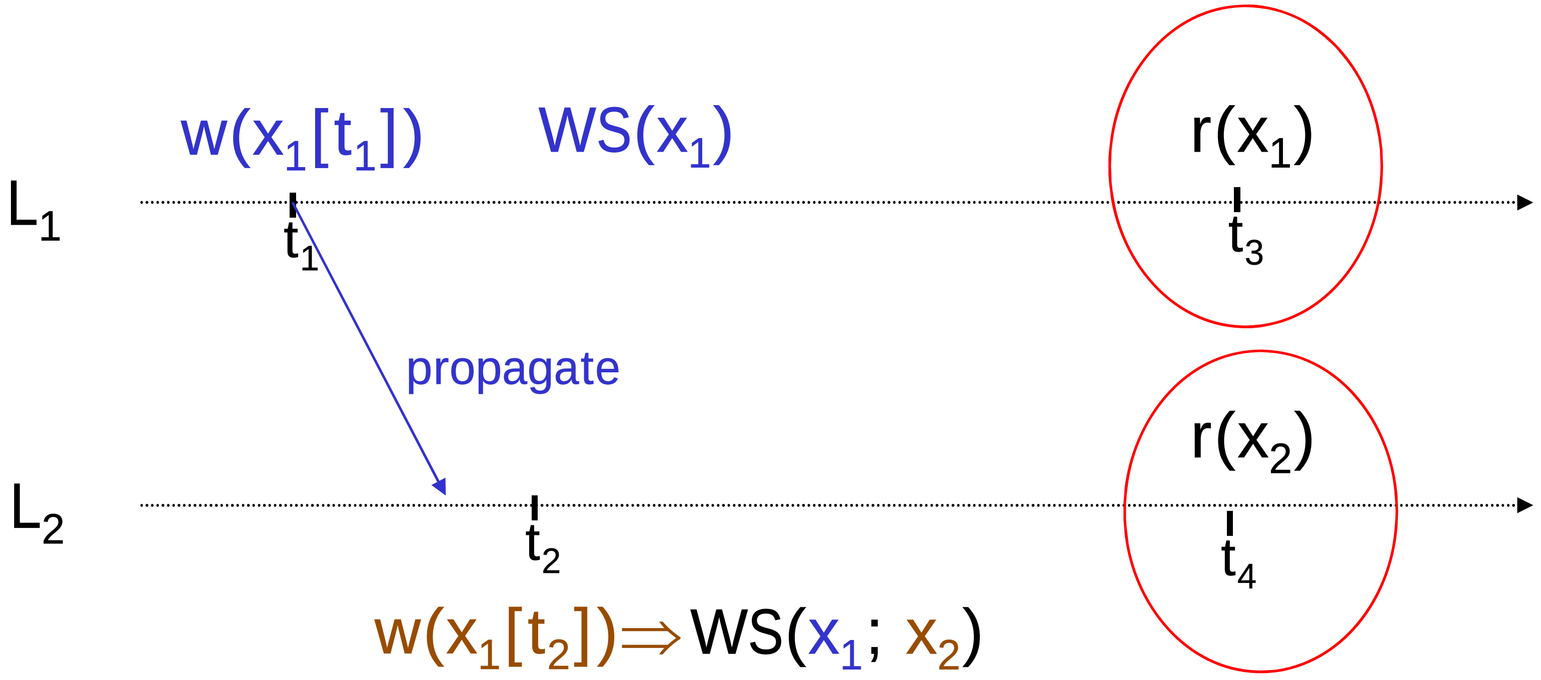
[1] Typical data item: file

Notation

- writes do not completely overwrite data item x
- $x_i[t]$ is the version of data x at time t at replica L_i ($x_i[t]$ = result of a sequence of earlier writes since initialization)
- According write-set is denoted as: $WS(x_i[t])$, i.e. x has taken into account at least all writes at L_i until time t
- $WS(x_i[t_1]; x_j[t_2])$ denotes all writes until t_1 at replica L_i and all writes until t_2 at L_j

A client centric consistent DDS guarantees that at L_j all previous writes at L_i have been updated before the first write can take place at L_j

Timeline Example



Note:

In the above example we only regard one data item x

x_i denotes its replica at note L_i

Implement Client Centric Consistency

Naive Implementation (ignoring performance):

- Each write gets a globally unique identifier
- Identifier is assigned by the server that accepts this write operation for the first time
- For each client two sets of write identifiers are maintained:
  - Read-set(client C) := RS(C)
    {write-IDs relevant for the reads of this client C}
  - Write-set(client C) := WS(C)
    {write-IDs having been performed by client C}

Monotonic Reads

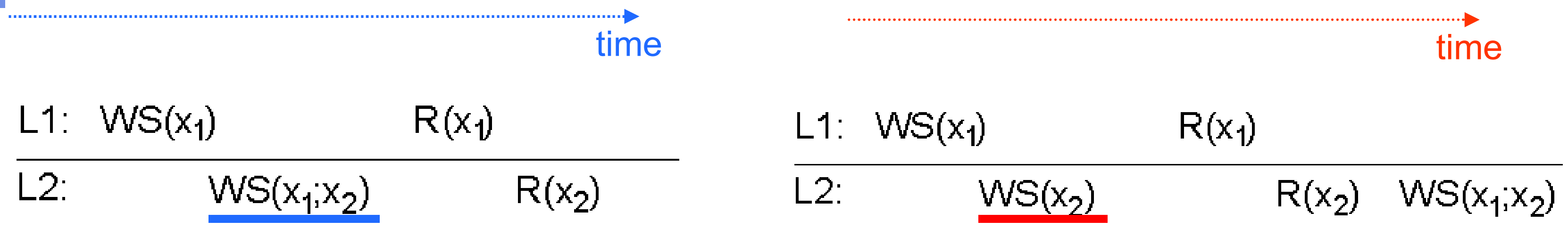
Definition:

A DDS provides “monotonic-read” consistency if the following holds:

> Whenever a process P already has read the value v of a data item x, then any successive read on x by P at any location L will return the same value v or a more recent one v’ (independently of the replica at location L where this new read is done)

Note: At some other location L‘ another process P concurrently might read: v(old) or v or v‘

Monotonic Reading



time

L1: WS(x_1) R(x_1)

L2: WS(x_1;x_2) R(x_2)

(a)

time

L1: WS(x_1) R(x_1)

L2: WS(x_2) R(x_2) WS(x_1;x_2)

(b)

- Read operations performed by process P at two different local copies of data item x

a) A monotonic-read consistent data store

b) A data store that does not provide monotonic reads, suppose x is a vector consisting of 2 entries, i.e.

x = <x[1], x[2]> and

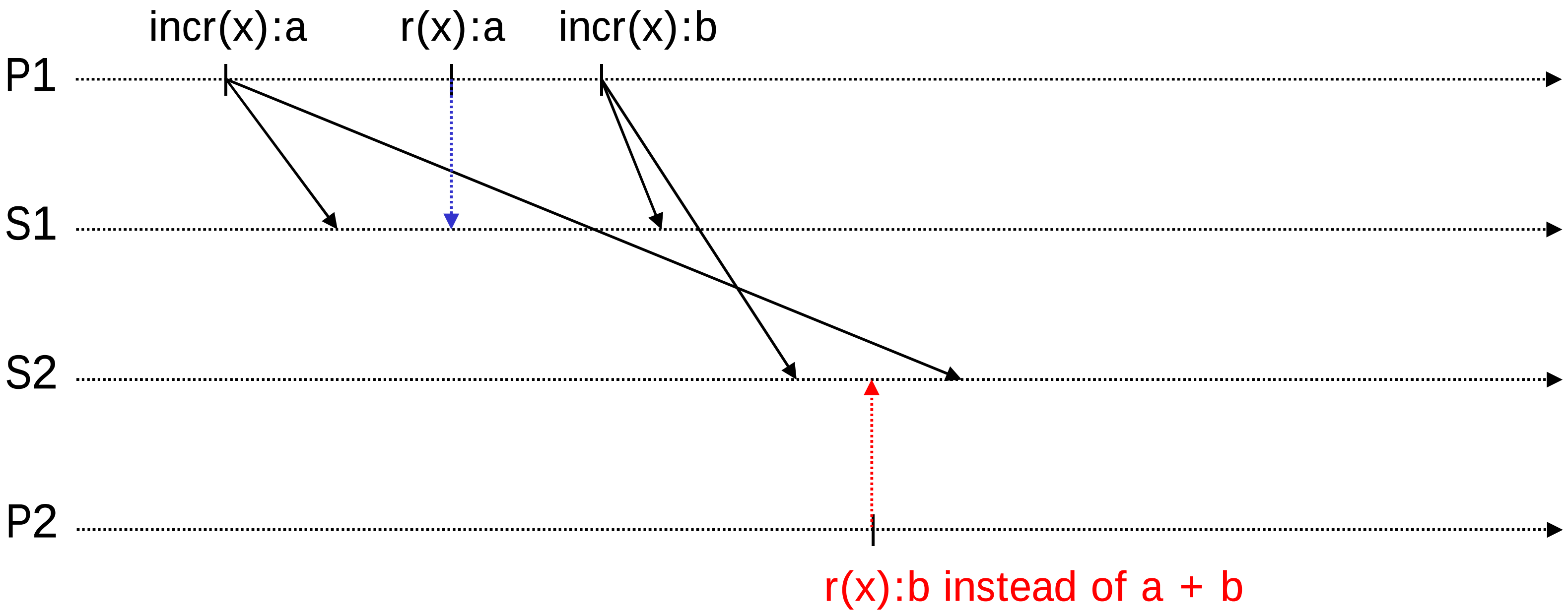
x_1 changed x[1], whereas x_2 has changed x[2]

(the second read only reflects changes made to x[2])

Systems with Monotonic Reads

- Distributed e-mail database with distributed and replicated user-mailboxes
- Emails can be inserted at any location
- However, updates are propagated in a lazy (i.e. on demand) fashion

Example: Non Monotonic Reads



Implement Monotonic Reads

Basic Idea:

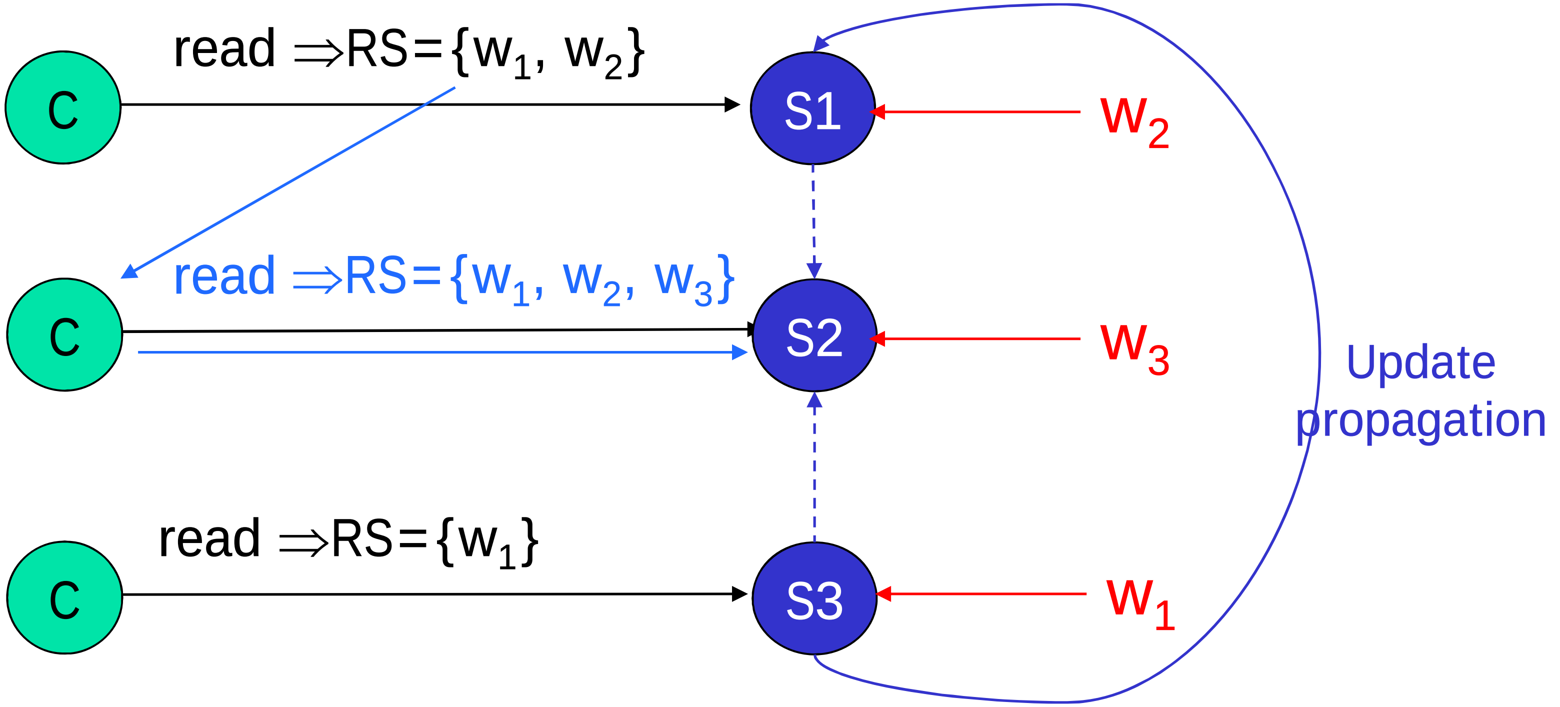
Client keeps a log of writes it has already seen, and makes sure that each replica he wants to read is up to date

- Each write gets a globally unique id (identifying operation and writer)
- Each replica keeps a log of writes that have occurred on that replica
- Each client keeps track of a set of write identifiers
  - Read set:= {writes relevant to reads done by the client}

Implement Monotonic Reads

- Read operation:
  - Client hands its read-set to the server of the local replica, whenever it wants to read from it
  - Server checks the writes in this read set whether all of them have been done on its local replica
  - If not yet done, the server retrieves all missing writes on its replica before it replies to the client's read request

Implement Monotonic Reads



Assume: Client C has already read from S_1 and S_3 implying 2 read sets

When C wants to **read from S_2**, C hands its current RS(C)={w1,w2} to S_2

S_2 controls whether all writes of RS(C) have taken place at S_2

If not, S_2 has to be *updated* before reading can be allowed!

Analysis

- The logs (read sets) can become very large
- To reduce space and time overhead:
  - You can use the usual session semantics to clear logs at the end of a session ⇒
  - before a client leaves the session, its session manager provides that all replicas are updated with all client's writes

Monotonic Writes

Definition:

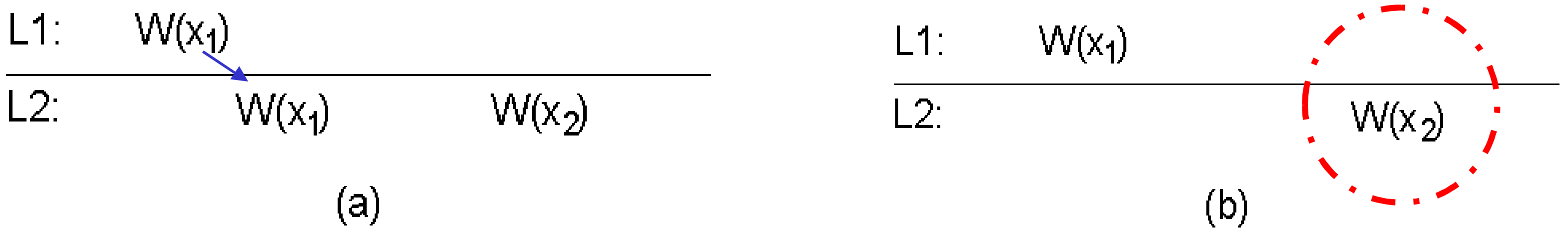
A DDS provides “monotonic-write” consistency if the following holds:

> A write by a process P on a data item x is completed before another write by the same process P can take place on that x.

Remark: Monotonic-writing ~ FIFO consistency

- Only applies to writes from one client process P
- Different clients may see the writes of process P in any order

Monotonic Writes



- The write operations performed by a single process P at two different replicas of the same data store

a) A monotonic-write consistent data store.

b) A data store that does not provide monotonic-write consistency

Implement Monotonic Writes (1)

Each client keeps a write-set = {all writes this client has previously performed}

- Write operation:
  - Whenever a client C wants to perform a write on a local replica it hands its current write-set to the corresponding local server
  - This server checks whether all writes of C's write set have been done yet
  - If not, it contacts the other concerned server(s) to initiate the propagation of the missing writes

Implement Monotonic Writes (2)

- Each write gets an unambiguous sequence number
- Each replica keeps newest sequence number
  - Delay writes with a too high sequence number as long as all missing writes have been updated on the local copy

Read Your Writes

Definition:

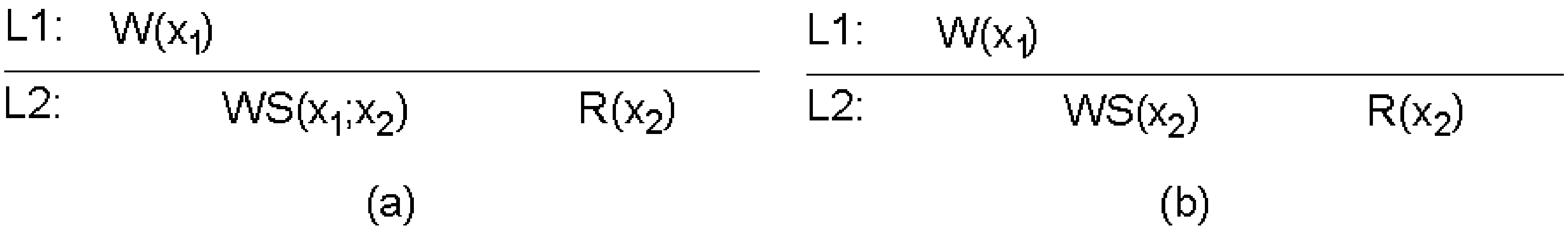
A DDS provides “read your write” consistency if the following holds:

> The effect of a write operation by a process P on a data item x at a location L will always be seen by a successive read operation by the same process wherever this read will take place

Example of a missing read your write consistency:

Updating a website with an editor, if you want to view your updated website, you have to refresh it, otherwise the browser uses the old cached website content

Example: Read Your Writes



a) A DDS providing read-your-writes consistency

b) A DDS that does not

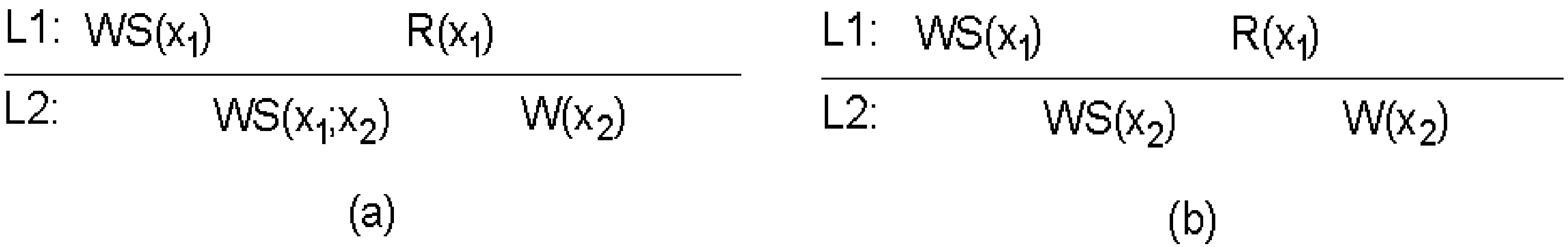
Writes Follow Reads

Definition:

A DDS provides "*writes-follow-reads*" consistency if the following holds:

> A write operation by a process *P* on a data item *x* following a previous read by the same process is guaranteed to take place on the same or even a more recent value of *x*, than the one having been read before.

Writes Follow Reads



L1: WS(x_1) R(x_1)

L2: WS(x_1;x_2) W(x_2)

(a)

L1: WS(x_1) R(x_1)

L2: WS(x_2) W(x_2)

(b)

a) A writes-follow-reads consistent DDS

b) A DDS not providing writes-follow-reads consistency

Summary on Consistency Models

An appropriate consistency model depends on following trade-offs:

- Consistency and redundancy
    - All replicas must be consistent
    - All replicas must contain full state
    - Reduced consistency → reduced reliability
- Consistency and performance
    - Consistency requires extra work
    - Consistency requires extra communication
    - May result in loss of overall performance
- Consistency and scalability
    - Consistency requires extra work
    - Consistency requires extra communication
    - Strong consistency weakens scalability