Data Oriented Consistency

31 May 2026
Lecture 11

Slide Credits: Maarten van Steen

Topics for Today

- Data Oriented Consistency

Sources:

- TvS 7.1-7.3

So Far

- Data Oriented Consistency

Performance and Scalability

Main issue: To keep replicas consistent, we generally need to ensure that all **conflicting** operations are done in the same order everywhere

Conflicting operations: From the world of transactions:

- **Read-write conflict**: a read operation and a write operation act concurrently
- **Write-write conflict**: two concurrent write operations

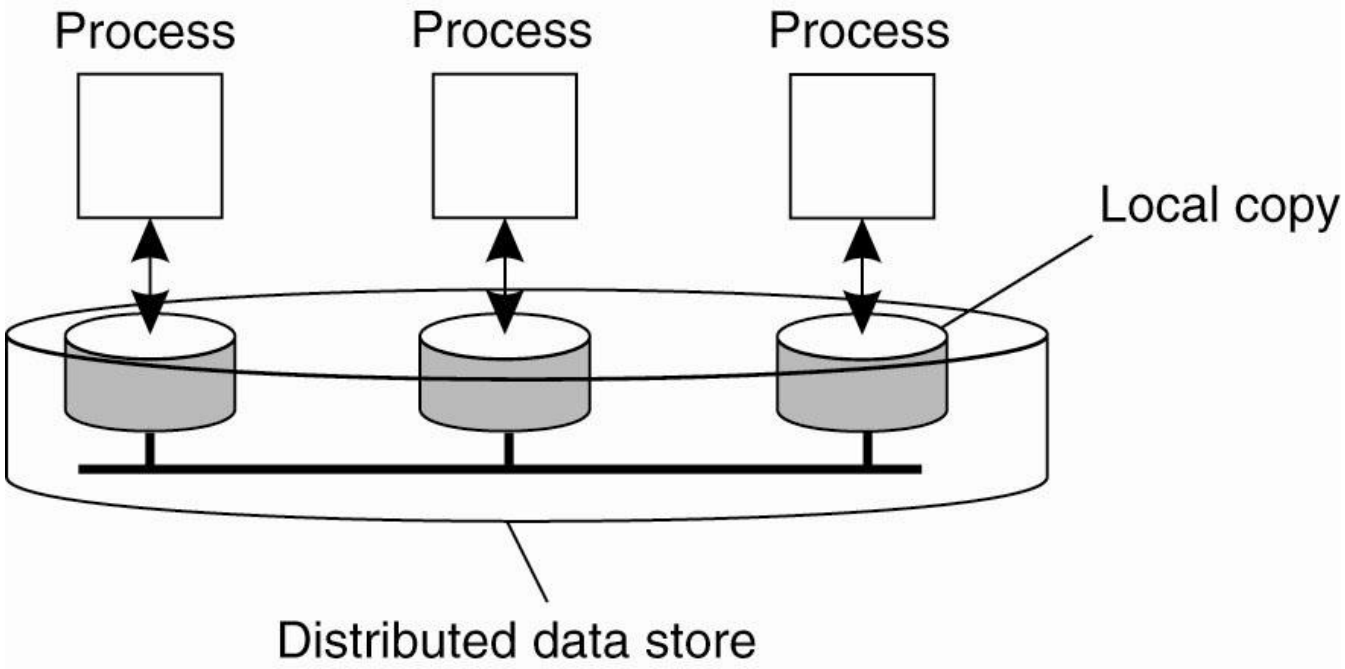
Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability

Solution: weaken consistency requirements so that hopefully global synchronization can be avoided

Data-Centric Consistency Models

Consistency model: a contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

Essence: A data store is a distributed collection of storages accessible to clients:



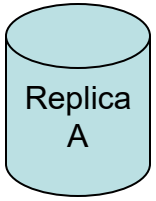
Continuous Consistency

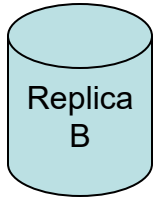
Observation: We can actually talk a about a **degree of consistency**:

- replicas may differ in their **numerical value**
  - Value **may be how many updates are missing**
  - Weight **may be the mathematical distance**
- replicas may differ in their relative **staleness**
- there may be differences with respect to (number and order) of **performed update operations**

Conit: consistency unit → specifies the **data unit** over which consistency is to be measured.

Example: Conit Dimensions





VC [A,B]	Committed		Tentative		Event
	x	y	x	y	
[0,0]	0	0	0	0	Init
[0,5]	0	0	2	0	Receive [0,5] from B
[1,5]	2	0	2	0	Commit [0,5]
[8,5]	2	0	2	2	y := y +2
[12,5]	2	0	2	3	y := y + 1
[14,5]	2	0	6	3	x := y * 2

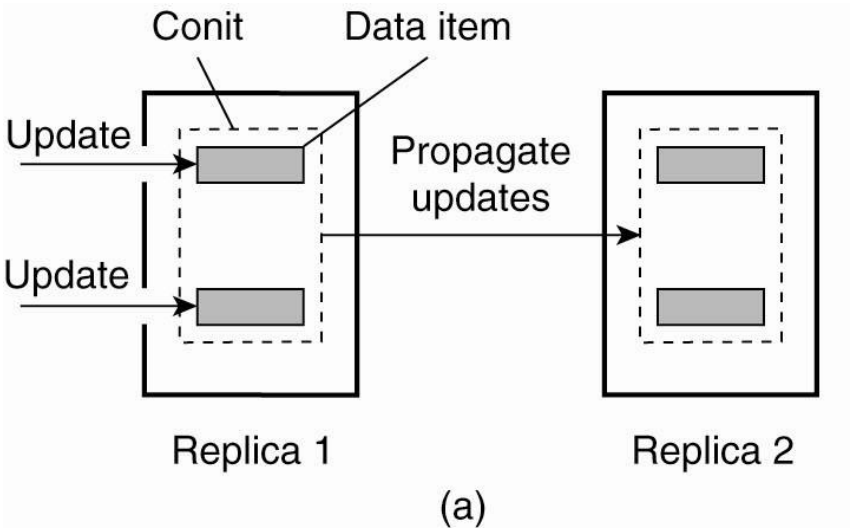
VC [A,B]	Committed		Tentative		Event
	x	y	x	y	
[0,0]	0	0	0	0	Init
[0,5]	0	0	2	0	x := x+2
[0,6]	0	0	2	0	Send [0,5] to A
[0,10]	0	0	2	5	y := y + 5

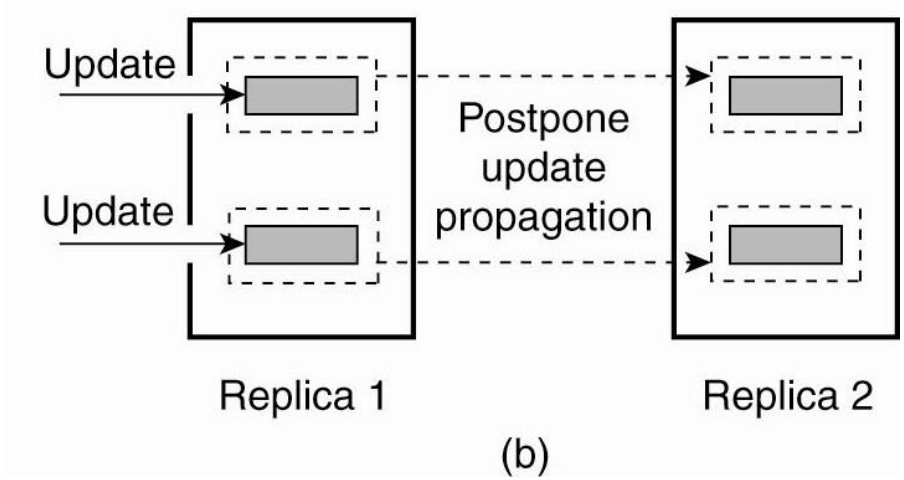
- Dimension 1: **Updates received, not committed**: A = 3, B = 2
- Dimension 2: **Updates a replica didn't see**: A = 1, B = 3
- Dimension 3: **Numerical deviation from missing updates**:
  - A = 5 (x missing 0, y missing 5), B = 6 (x missing 6, y missing 3)

Conits

- The granularity of your conit is important
- With a given consistency policy:
  - Smaller conits mean you may defer pushing updates
  - Bigger conits mans you must push earlier

Example: Do you update a whole table? Each row in the table? Each entry/cell in the row?

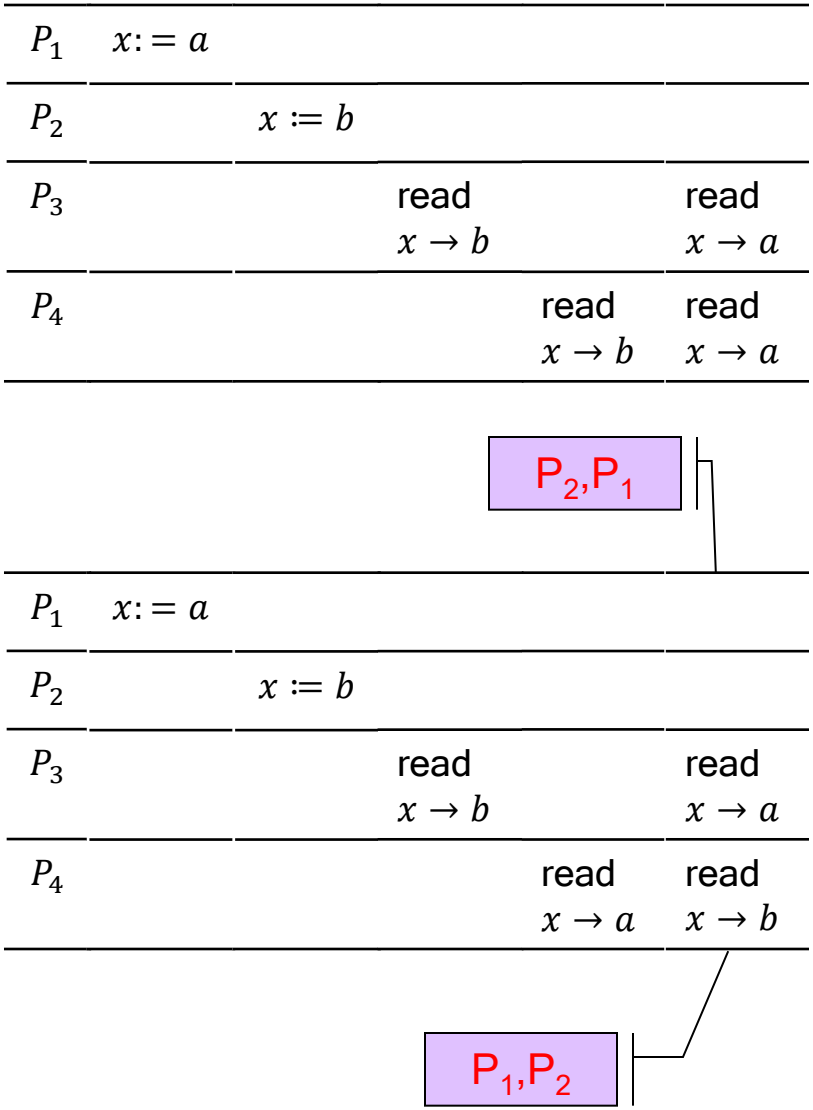




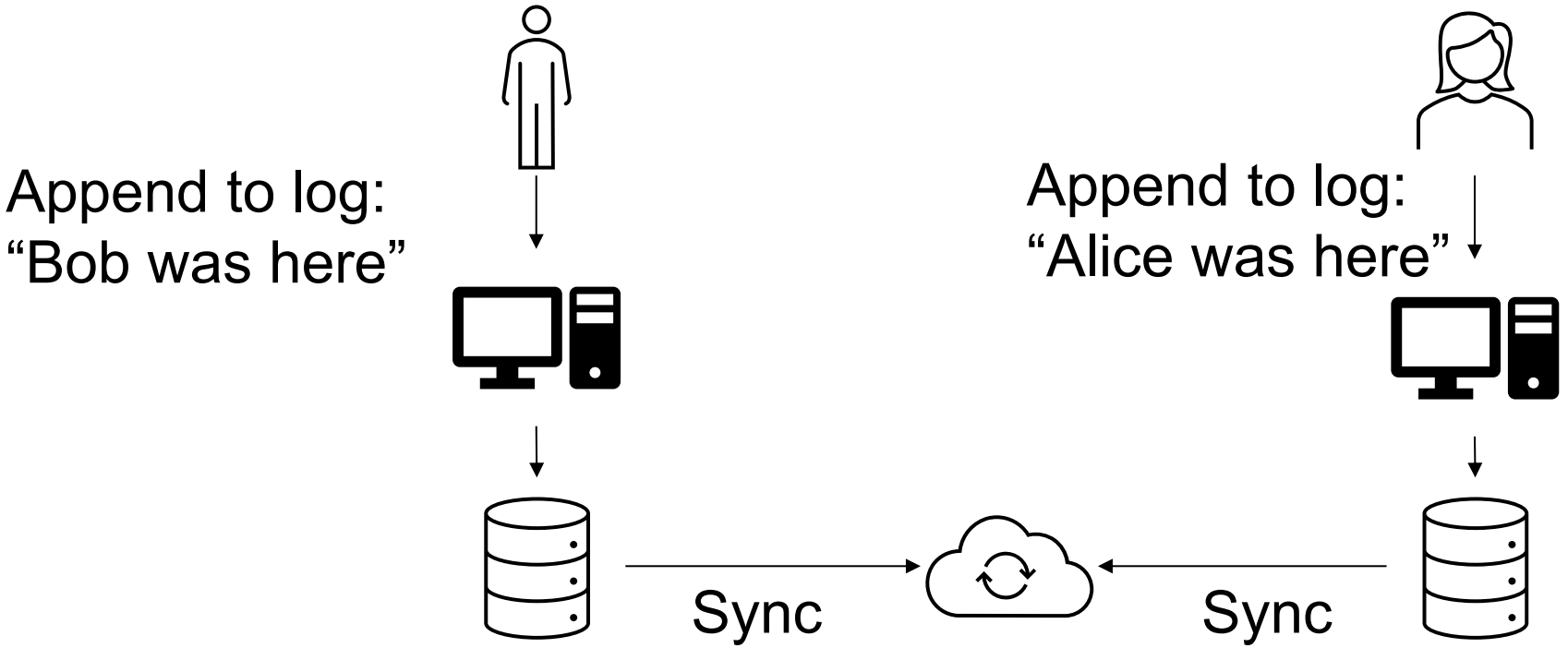
Sequential Consistency

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

Note: We're talking about **interleaved** executions: there is some total ordering for all operations taken together.



Sequential Consistency



Potential log contents:

Alice was here Bob was here	Bob was here Alice was here

Both see the same outcome on their local copies

Causal Consistency

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order by different processes.

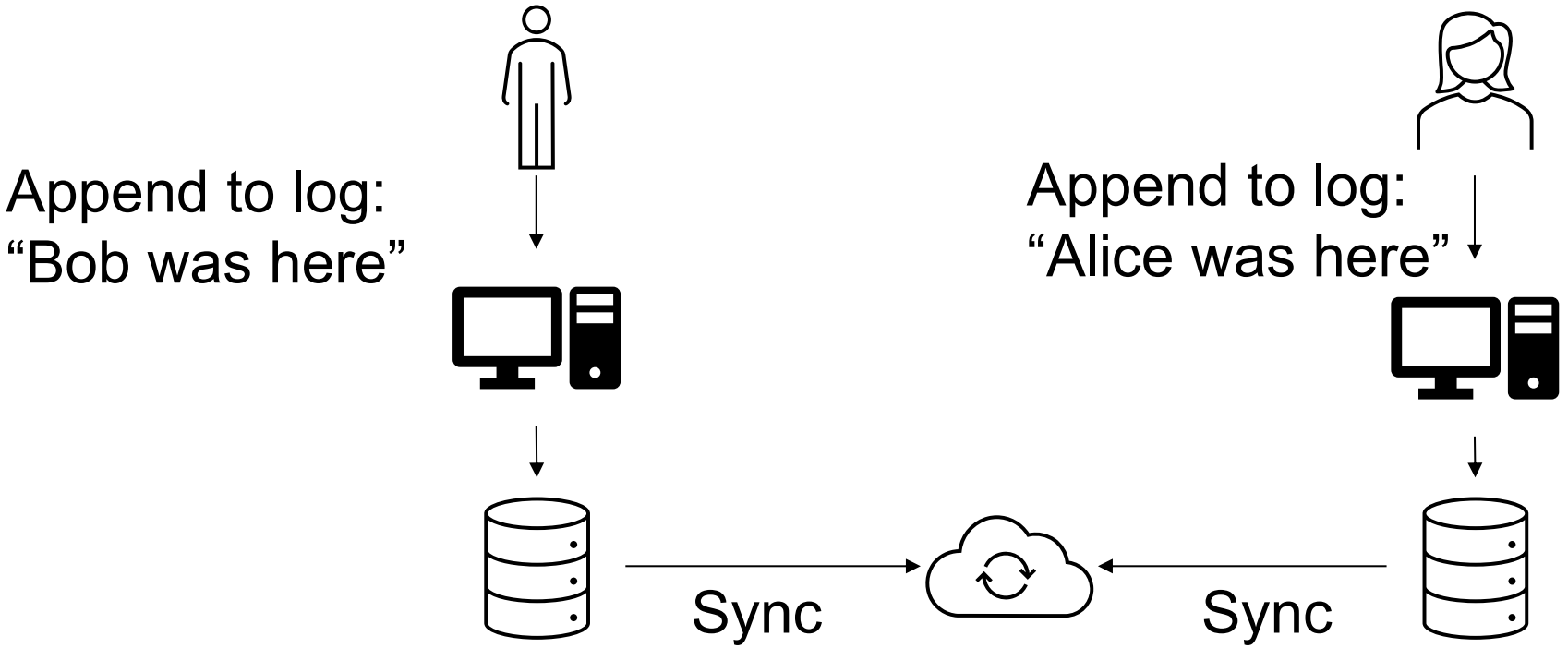
P_1	$x := a$		$x := c$	
P_2		read $x \to a$	$x := b$	
P_3		read $x \to a$	read $x \to c$	read $x \to b$
P_4		read $x \to a$	read $x \to b$	read $x \to c$

OK

P_1	$x := a$			
P_2		read $x \to a$	$x := b$	
P_3			read $x \to b$	read $x \to a$
P_4			read $x \to a$	read $x \to b$

P_1	$x := a$			
P_2			$x := b$	
P_3			read $x \to b$	read $x \to a$
P_4			read $x \to a$	read $x \to b$

Causal Consistency



Potential log contents:

Alice was here Bob was here	Bob was here Alice was here

They might see different versions on their local copies

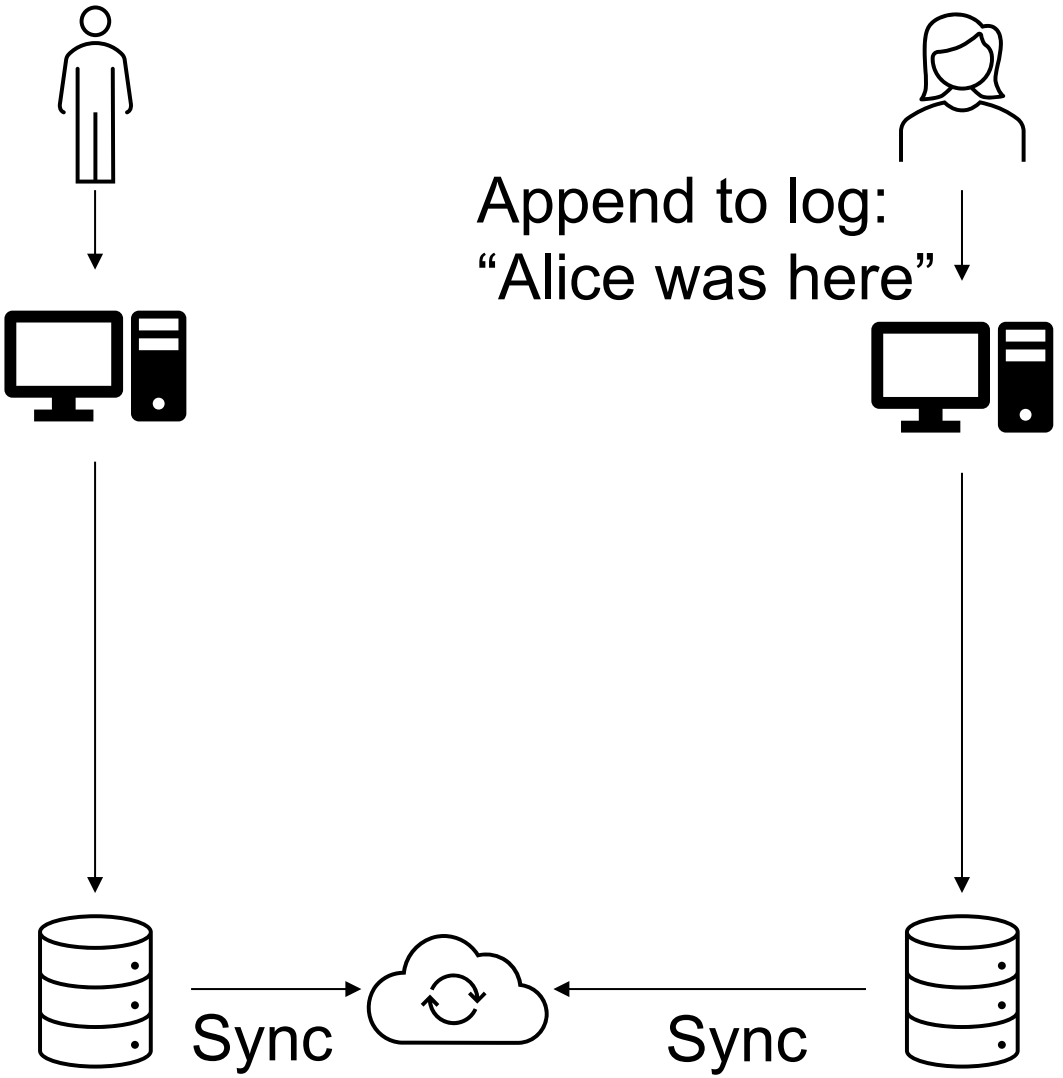
Causal Consistency

1. Read Log and sees "Alice was here"

Then

2. Append to log: "Bob was here"

Append to log: "Alice was here"



Potential log contents:

Alice was here
Bob was here

Grouping Operations (1/2)

- Whenever a process wants to acquire exclusive access to a synchronization variable, it must first wait for the previous owner to send the latest version of the guarded data.
  - That means that on entering an exclusive access critical section, the acquirer must first get all of the outstanding updates on the data
- To modify a shared variable, you first must acquire exclusive access to the relevant synchronization variable.
  - This can only happen when there are no other owners of the *synchronization variable, whether exclusive or non-exclusive*
- Before entering a non-exclusive critical section, the enterer must first get the most recent copy from the owner of the synchronization variable

Basic idea: You don't care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.

Grouping Operations (2/2)

P_1	$Acq(Lock(x))$	$x := a$	$Acq(Lock(y))$	$y := b$	$Rel(Lock(x))$	$Rel(Lock(y))$		
P_2						$Acq(Lock(x))$	read $x \rightarrow a$	read $y \rightarrow n$
P_3							$Acq(Lock(y))$	read $y \rightarrow b$

Observation: Weak consistency implies that we need to lock and unlock data (implicitly or not).

Question: What would be a convenient way of making this consistency more or less transparent to programmers?

Conclusion

- Data Oriented Consistency