

Sequential & Release Consistency

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Sequential Consistency

What is Consistency?

Consistency model defines **rules** for the apparent order and visibility of **updates**, and it is a continuum with tradeoffs
– Todd Lipcon

Examples

Local memory:



Single object consistency is also called “coherence”



Database:

Consistency across multiple objects (all or nothing)

Consistency Challenges

No right or wrong consistency models

- Tradeoff between ease of programmability and efficiency
(*e.g.*, what's the consistency model for web pages?)

Consistency is hard in (distributed) systems

- Data replication (Caching)
- Concurrency (no shared clock)
- Failures (machine or network)

Example: Mutual Exclusion

Is this program **correct**?

```
x = 1;  
if (y == 0)  
    critical  
    section  
...
```



```
y = 1;  
if (x == 0)  
    critical  
    section  
...
```

Instruction stream

- CPU-0's: W(x) R(y)
- CPU-1's: W(y) R(x)

All possible inter-leavings

1. W(x) 1 R(y) 0 W(y) 1 R(x) 1
2. W(x) 1 W(y) 1 R(y) 1 R(x) 1
3. W(x) 1 W(y) 1 R(x) 1 R(y) 1
4. ...

None **violates** mutual exclusion

Naive DSM

Is this program **correct**?

```
x = 1;  
if (y == 0)  
    critical  
    section  
...
```

Node-0

Write

Distributed
Shared Memory

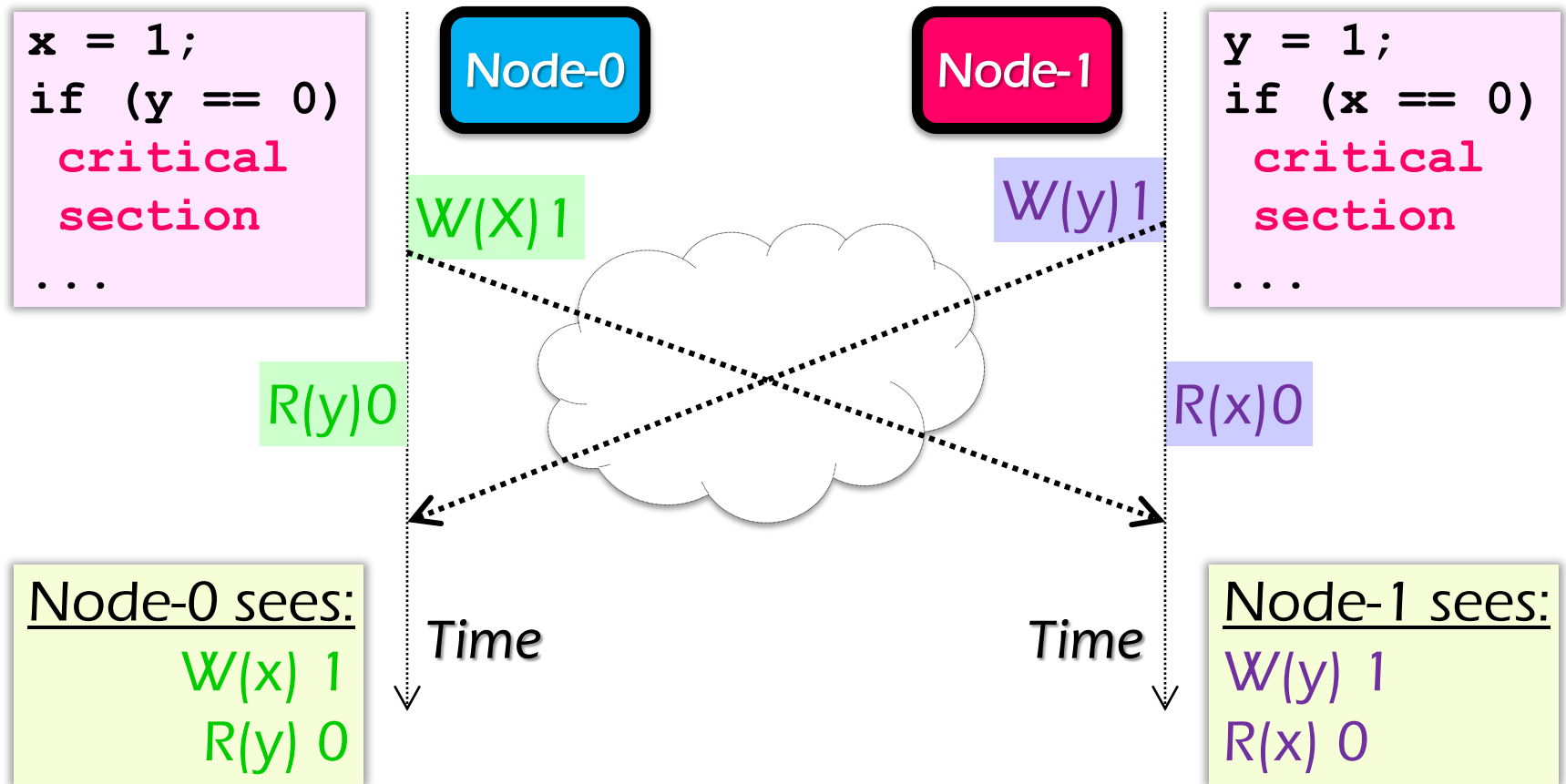
Node-1

Write

```
y = 1;  
if (x == 0)  
    critical  
    section  
...
```

1. Each node has a local copy of state
2. Read from local state
3. Send writes to the other node, but do not wait

Example on Naive DSM



Strict Consistency

Each operation under strict consistency is stamped with a **global wall-lock time**

Consistency rules

1. Each read gets the **latest** write value
2. All operations at one CPU have timestamp in **execution order**

Intuitive Results

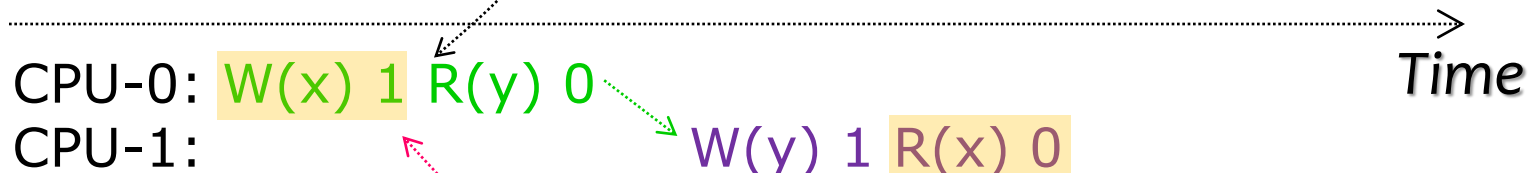
No two Nodes/CPU's in the critical section

Proof: suppose mutual exclusion is violated

Violation results

- CPU-0: $W(x) \ 1 \ R(y) \ 0$
- CPU-1: $W(y) \ 1 \ R(x) \ 0$

Rule 1: read gets **latest** write
→ $W(y)$ must have timestamp later than $R(y)$



But **contradicts** rule 1 (read get latest write)
→ $R(x)$ must see $W(x) \ 1$

Sequential Consistency

Strict consistency is not practical

- No global wall-clock available

Sequential consistency is the closest

Consistency rules

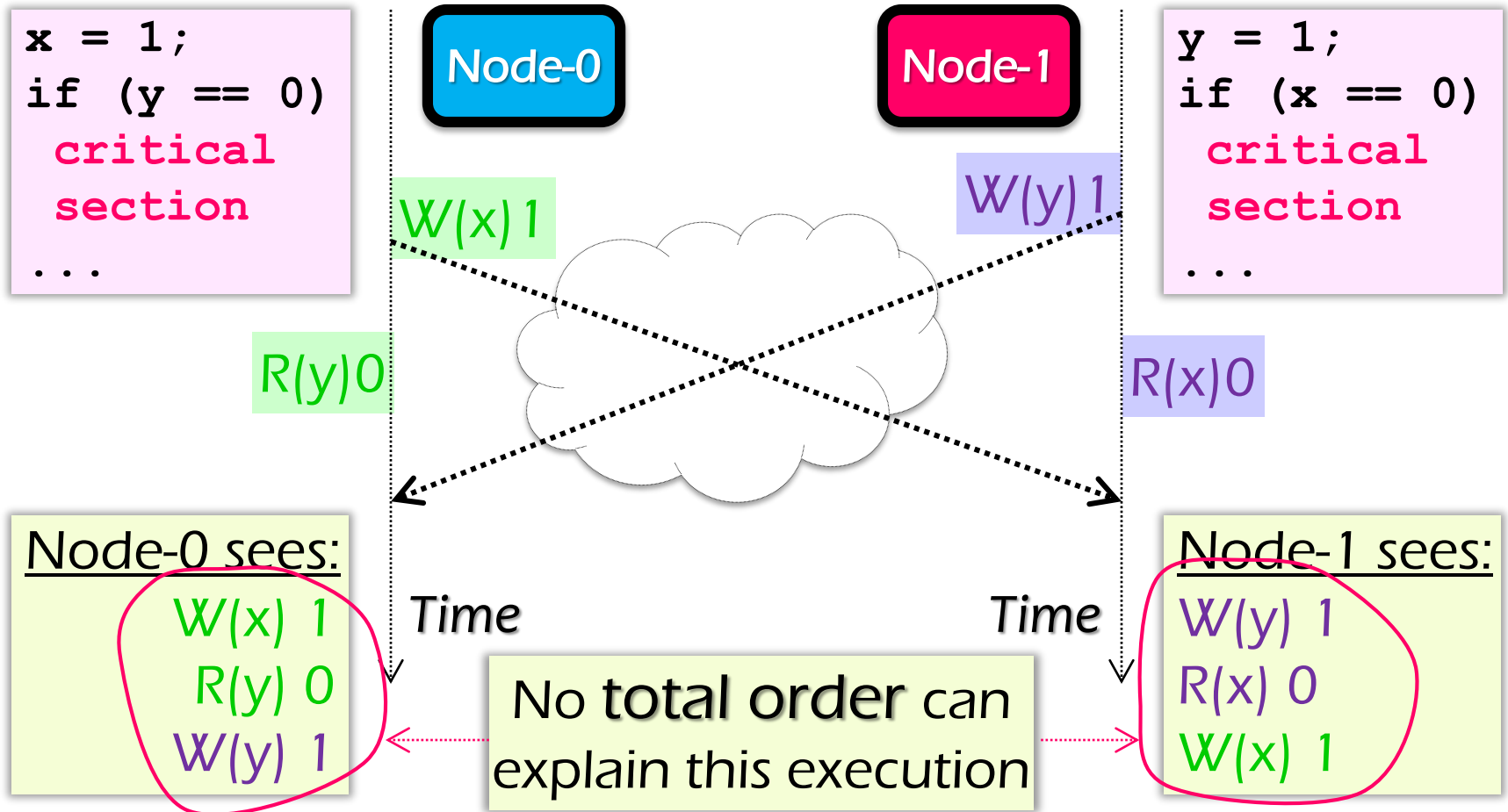
“global wall-clock” → “total order” of ops

1. Each CPUs' ops appear in order
2. All CPUs see results according to total order (*i.e.*, reads see most recent writes)

Sequential consistency is also intuitive results
(just use the **total order** as **timestamp**)

Recall Example

Does the system give **sequential** consistency?



Implementation of SC

Sequential consistency is easier to implement

- There is no notion of real time

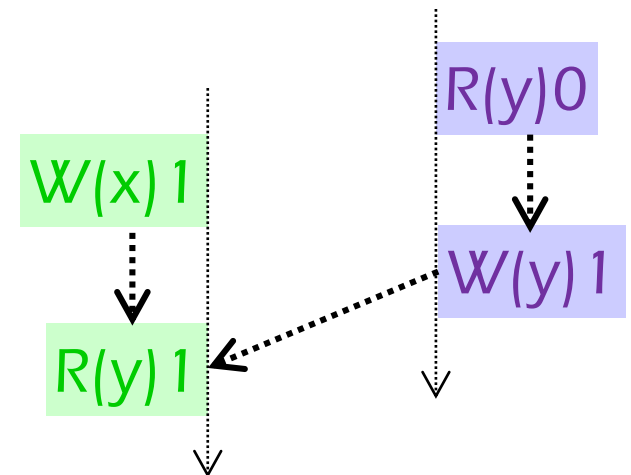
System is free to order concurrent events

However, when the system finishes a write or reveals a read, it commits to certain partial orders

Order: $W(x)1$ $R(y)0$ $W(y)1$ $R(y)1$

or

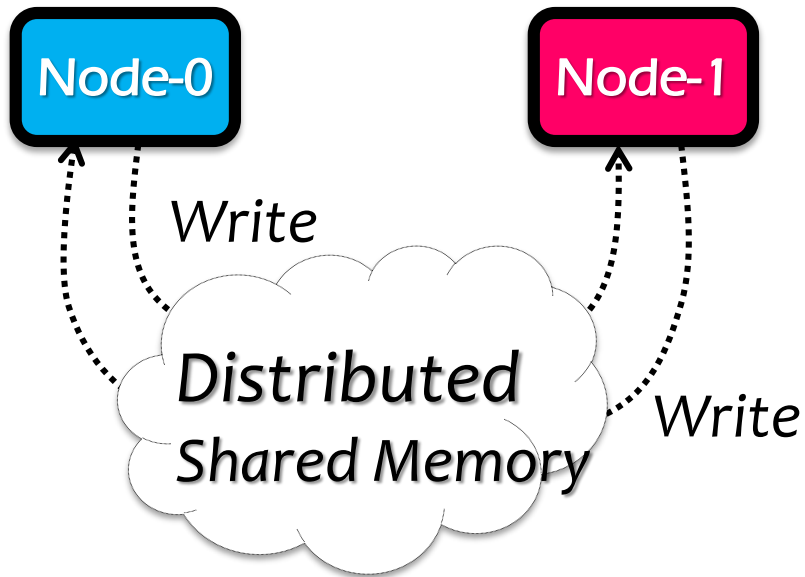
Order: $R(y)0$ $W(x)1$ $W(y)1$ $R(y)1$



Requirement for SC

- R1. each processor issues requests in the order specified by the program
 - Don't issue the next request unless the previous one has finished
- R2. requests to an individual memory location are served from a single FIFO queue
 - Writes occur in a single order
 - Once a read observes the effect of a write → it's ordered behind that write

Issue of Naive DSM



1. Each node has a local copy of state
2. Read from local state
3. Send writes to the other node, but do not wait

R1: issue read before waiting for write to complete
R2: issue writes concurrently, no single order

Ivy: A Shared Virtual Memory System for Parallel Computing

IVY: SVM or DSM

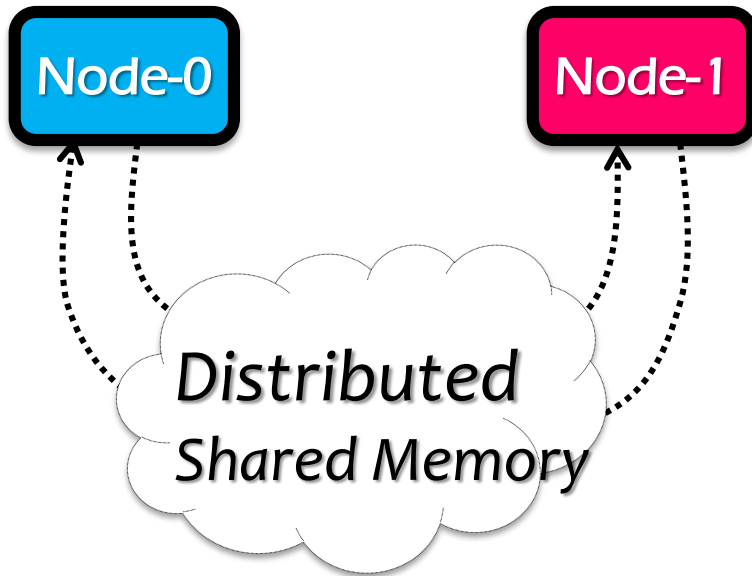
What does IVY do?

- Provides a shared memory system across a group of workstations

Why shared memory?

- Easier to write parallel/distributed programs with than using message passing (*e.g.*, complex data structure)
- We'll come back to this choice of interface in later lectures

Ivy Architecture



1. Each processor's local memory keeps a subset of all pages
2. If page not found in local memory, request from remote node

Why each node cache read pages?

Can a node directly write cache pages?

Questions about IVY

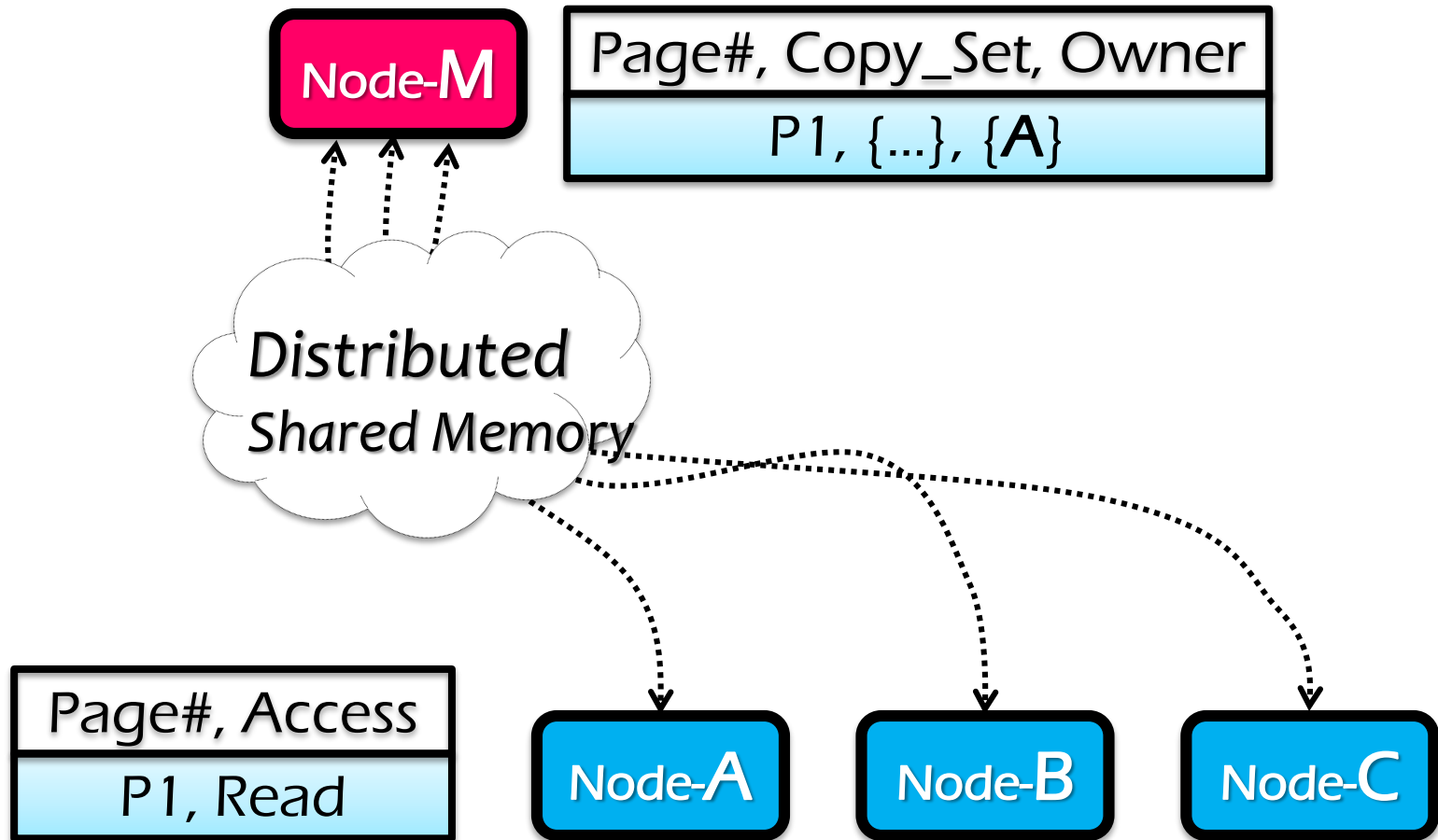
Why the ownership of a page moves across nodes?

- Latest writer becomes the owner

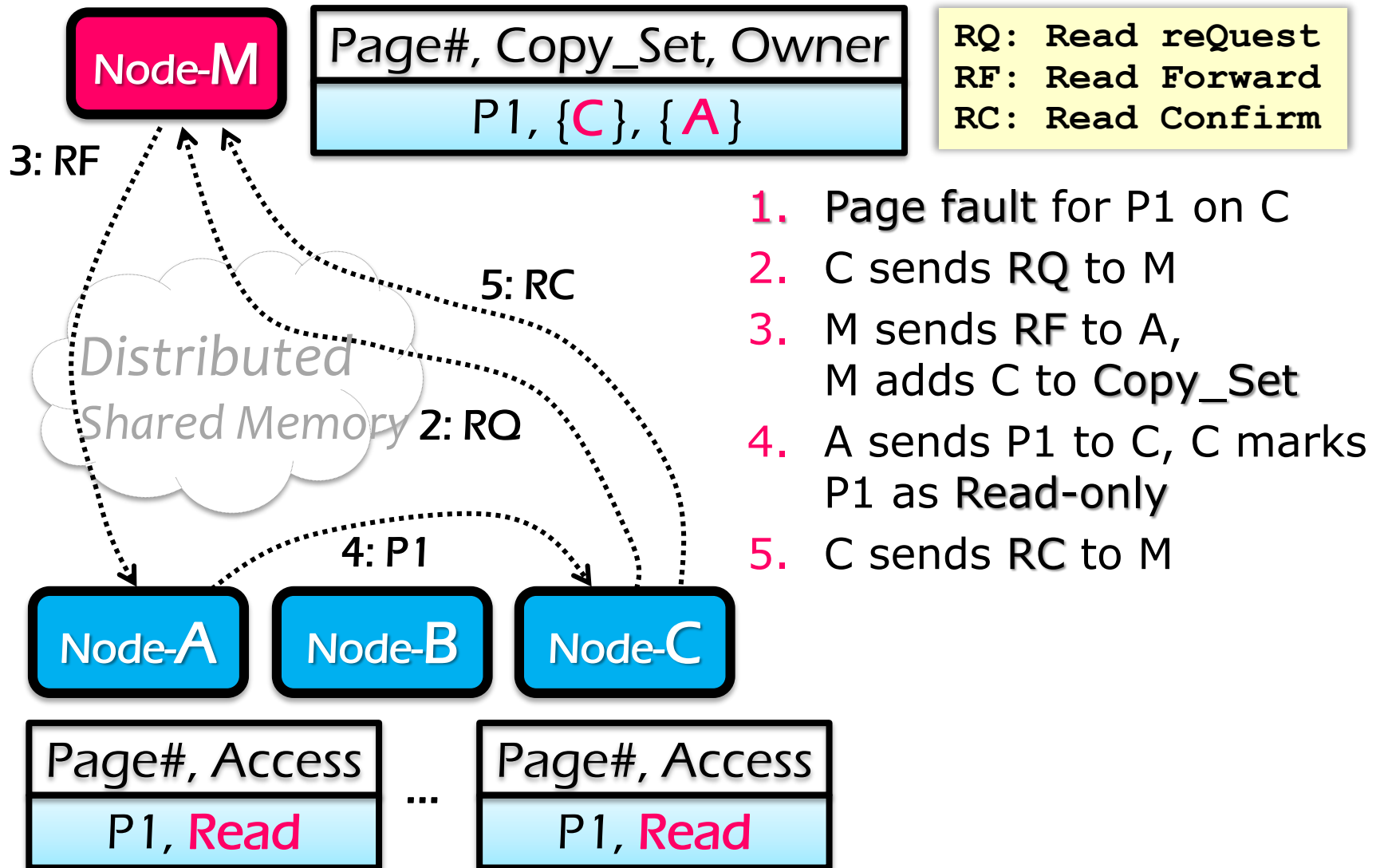
Challenges:

- How to find the own of a page?
- How to ensure one own per page?
- How to ensure all cached pages are invalidated?

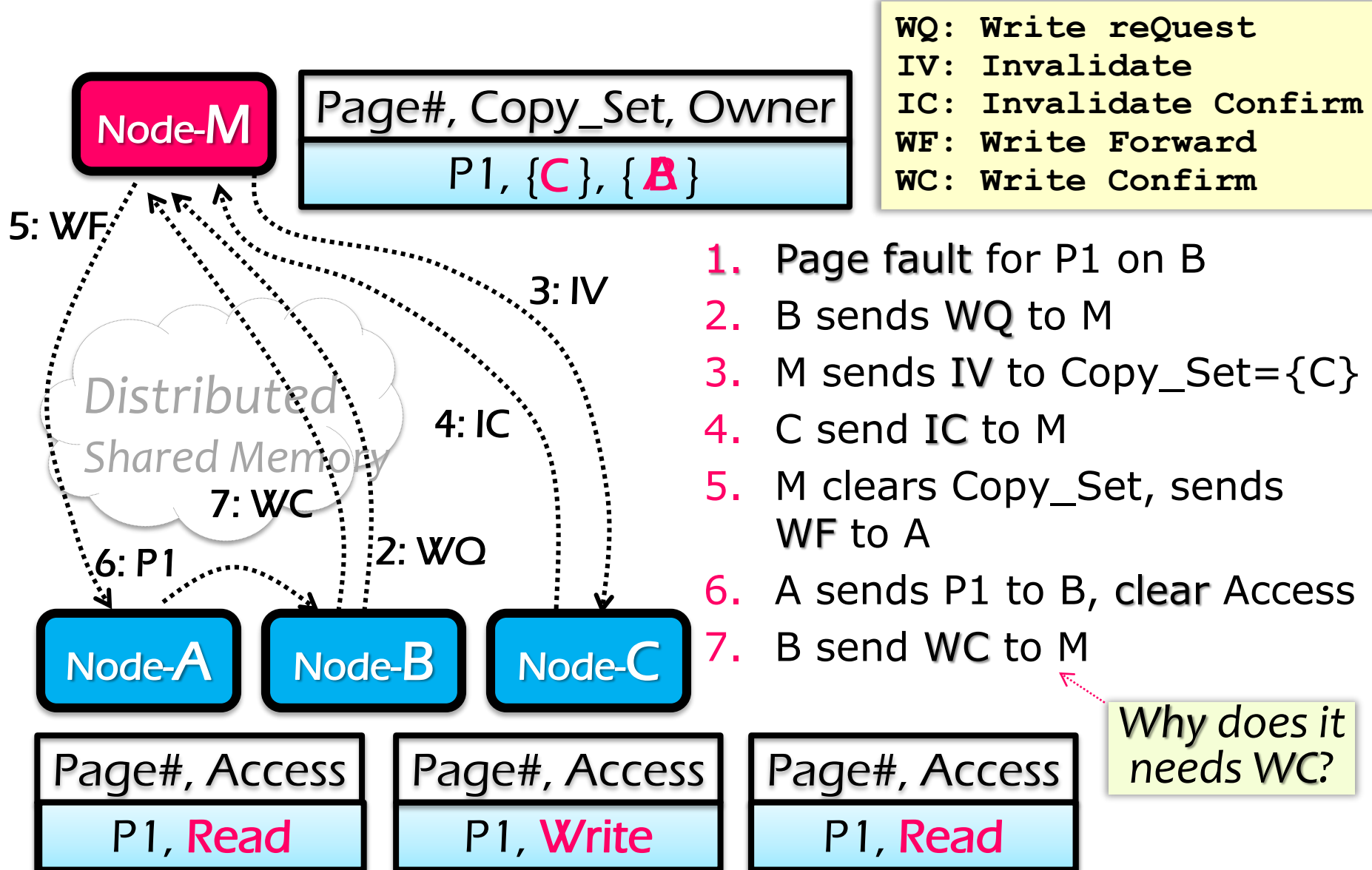
Centralized Manager



Read Operations

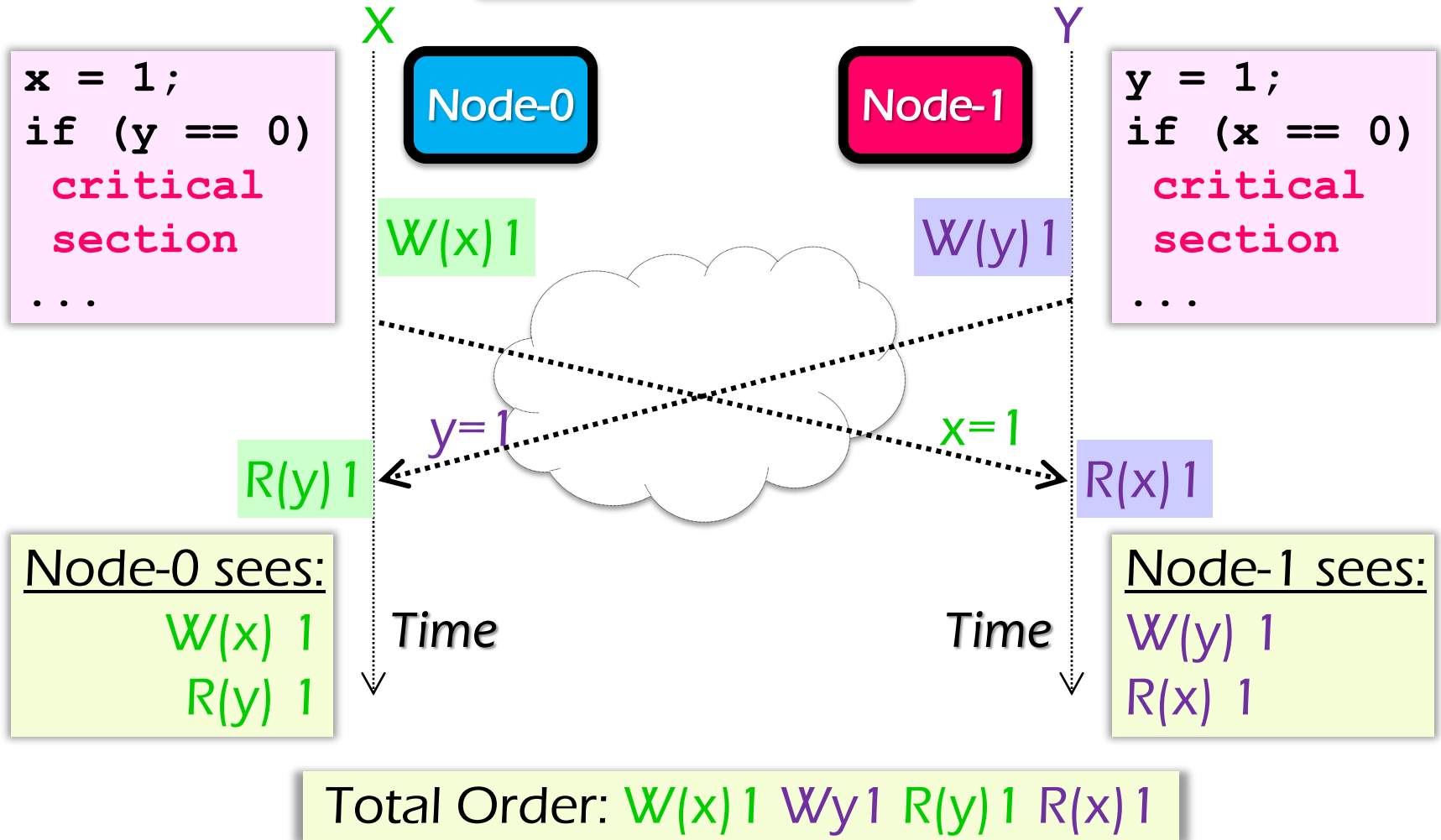


Write Operations



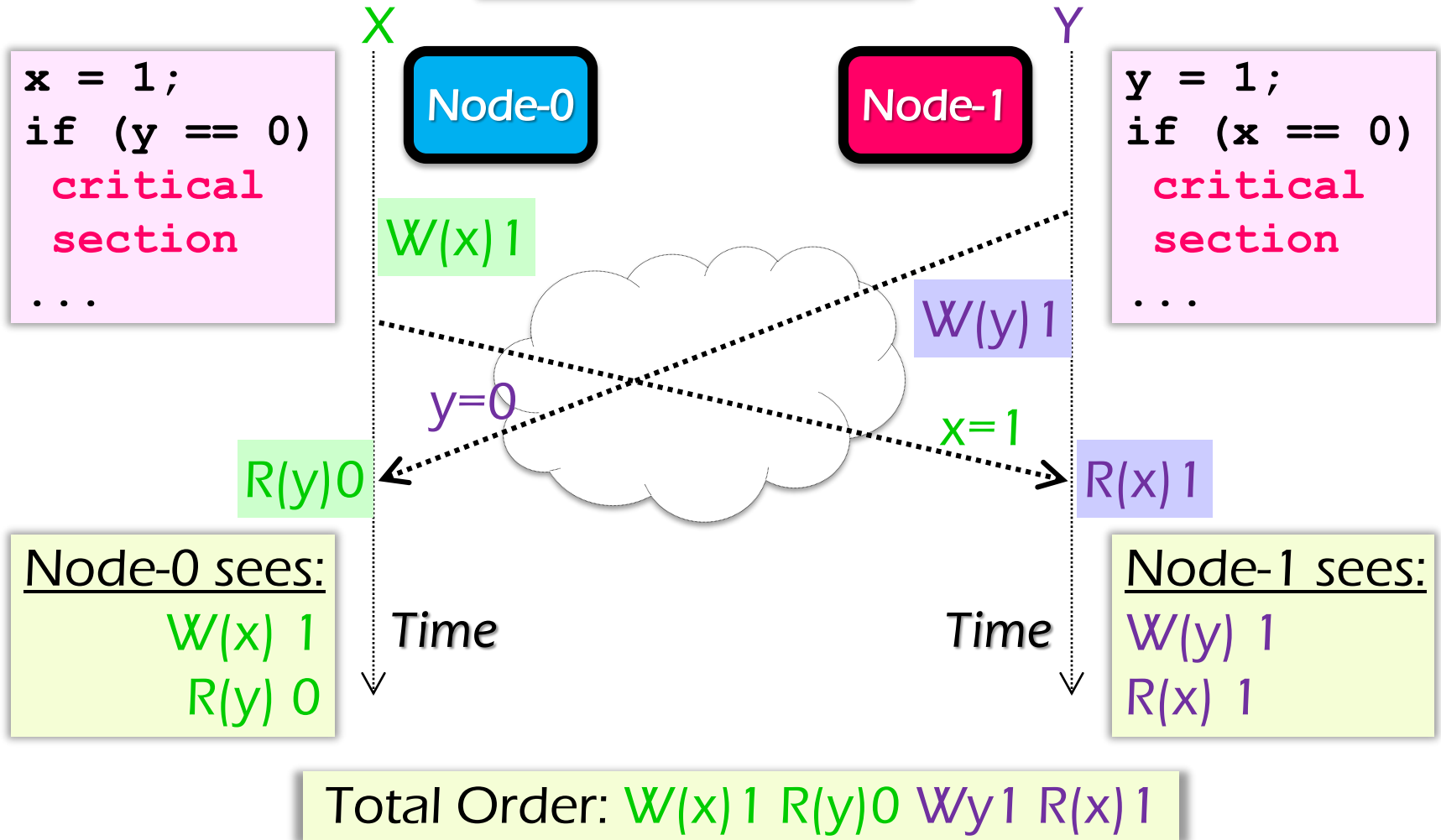
Recall Example

Does **IVY** work?



Recall Example

Does **IVY** work?



IVY Invariants

Every page has **exactly** one current owner

Current owner has a copy of the page

If mapped r/w by owner, **no** other copies

If mapped r/o by owner, identical to other copies

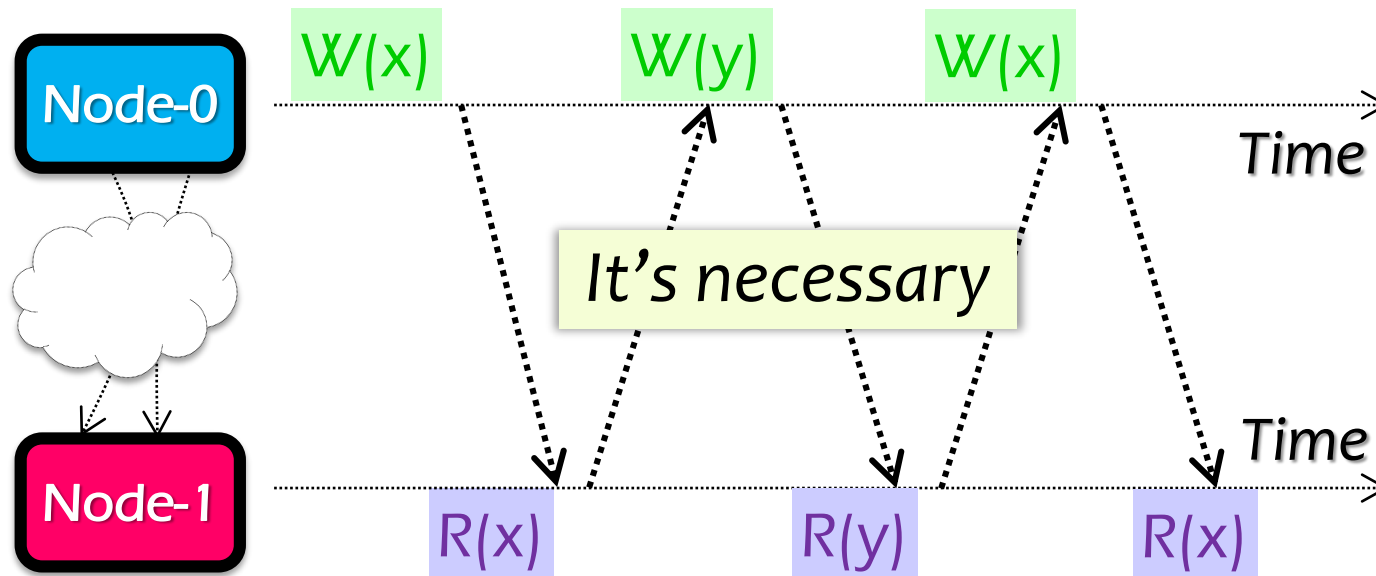
Manager knows about all copies

Release Consistency

Drawbacks of SC

In any implementation of SC, there should be some global control mechanism

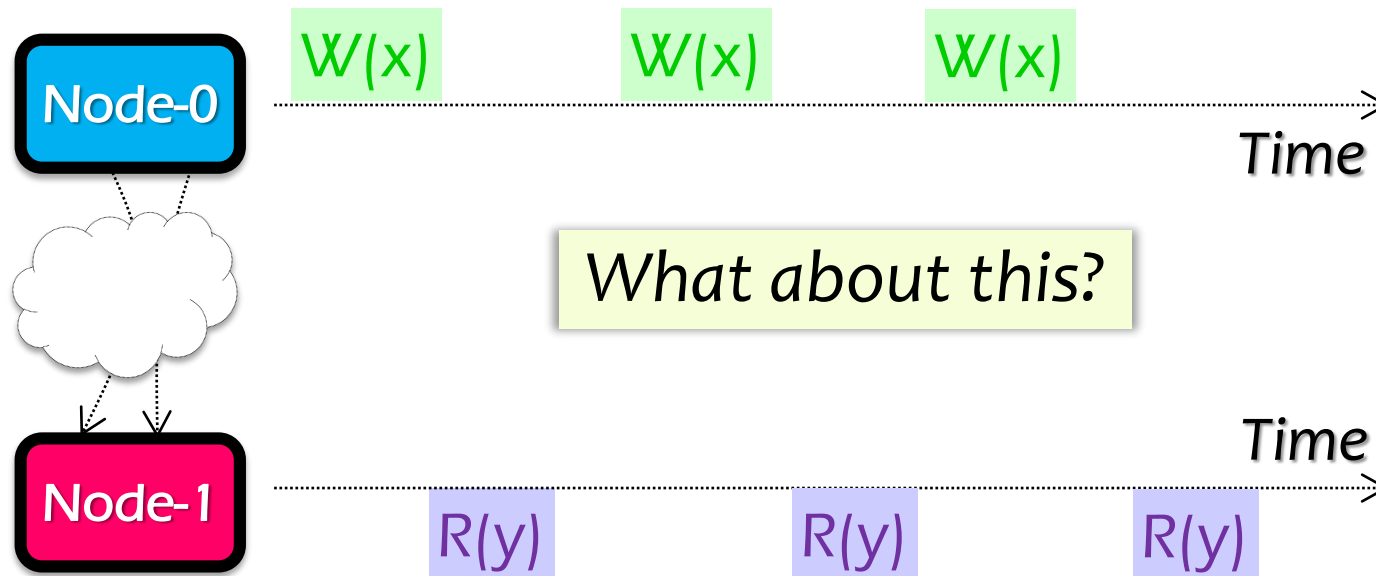
Either of writes or reads require memory synchronization operations



Drawbacks of SC

In any implementation of SC, there should be some global control mechanism

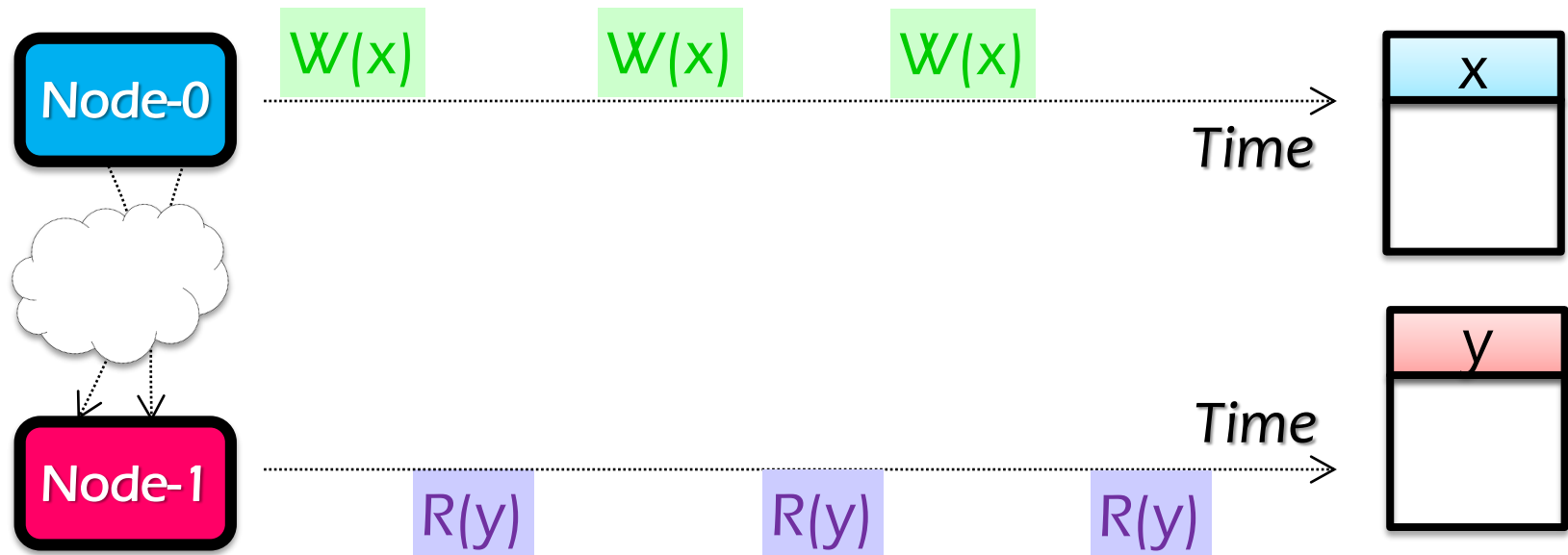
Either of writes or reads require memory synchronization operations



Drawbacks of SC

Mimicking consistency protocols of hardware

- False sharing

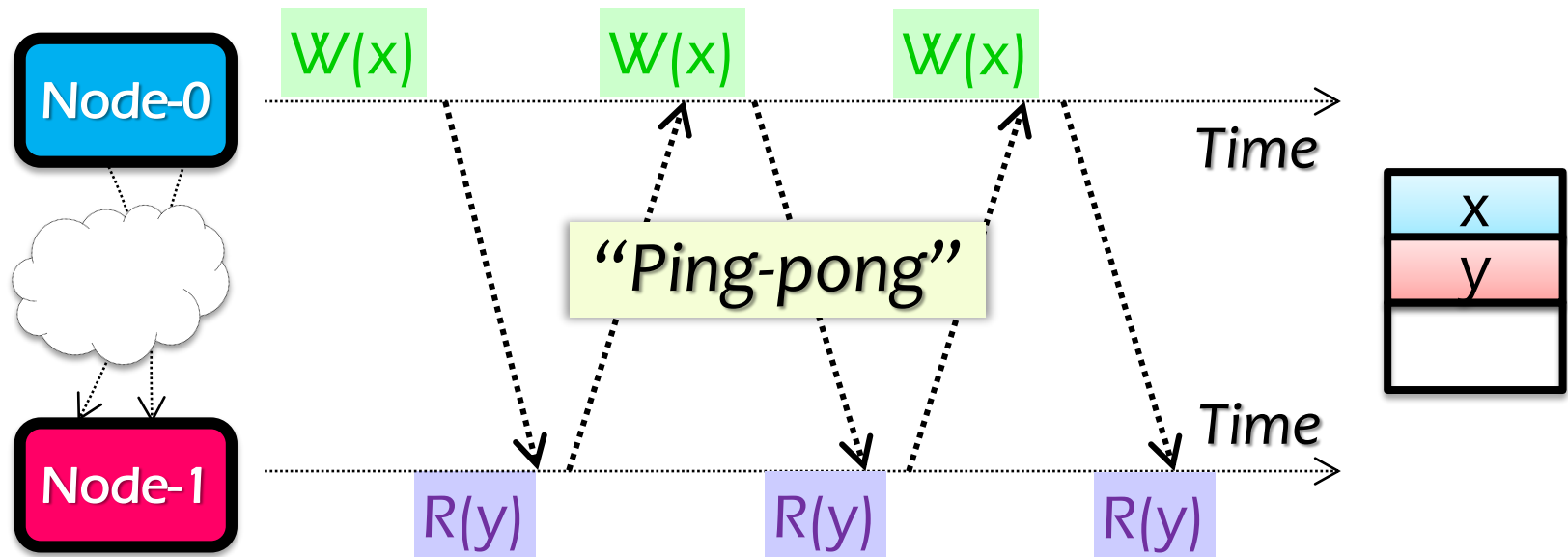


Drawbacks of SC

Mimicking consistency protocols of hardware

- False sharing

Two or more processes access **different** variables **within a page** and at least one of the accesses is a **write**.



Release Consistency

Introduce a special type of variables, called **synchronization variables** or **locks**

Locks can be acquired/released, not read/written

- Denoted by **acquire(L)** and **release(L)**

No more than one process can hold a lock L

- Hold a lock
→ acquired a lock and has not released it

Release Consistency

Sequential Consistency == Release Consistency ?

acquire() and **release()** are sequential consistency

- **release()** is performance after all previous operations have completed

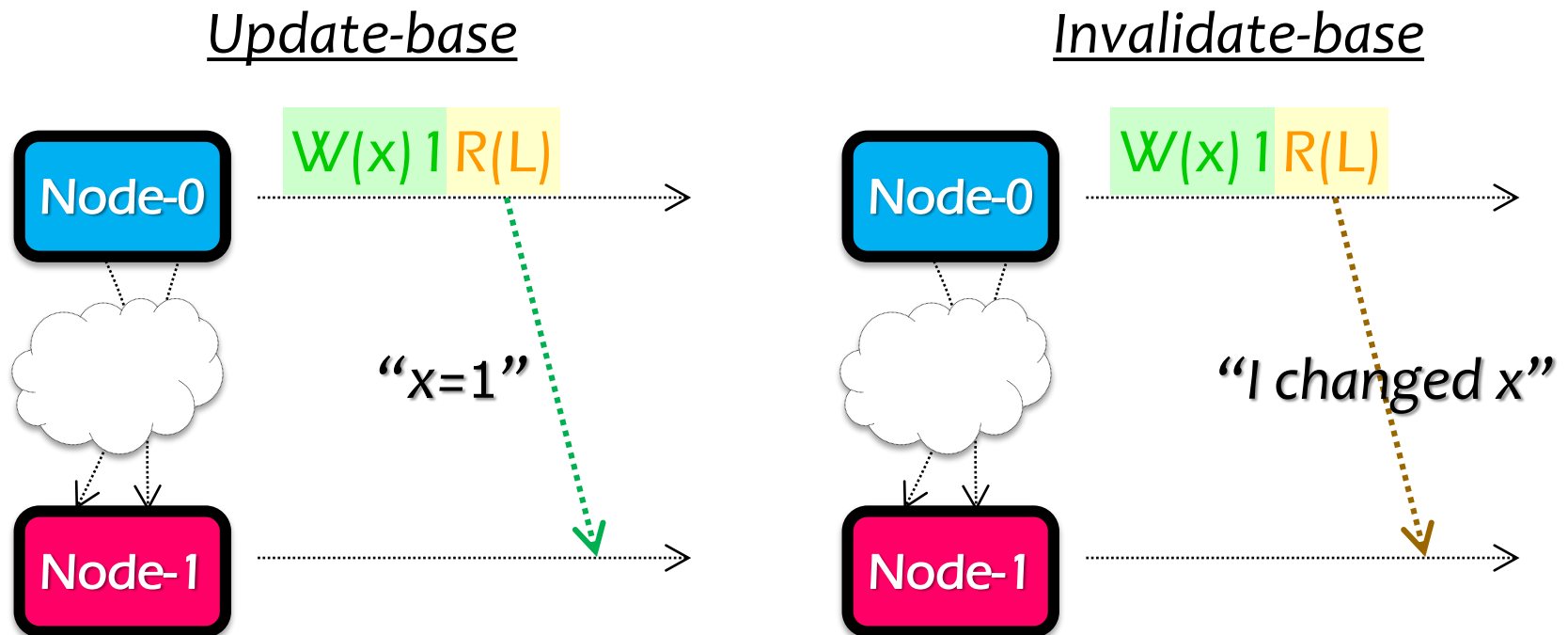
acquire() and **release()** pair between conflicting access

- Release consistency produces the same results of sequential consistency

SynChronization Protocol

Update-base protocol: send modifications

Invalidate-base protocol: send invalidations of modifications

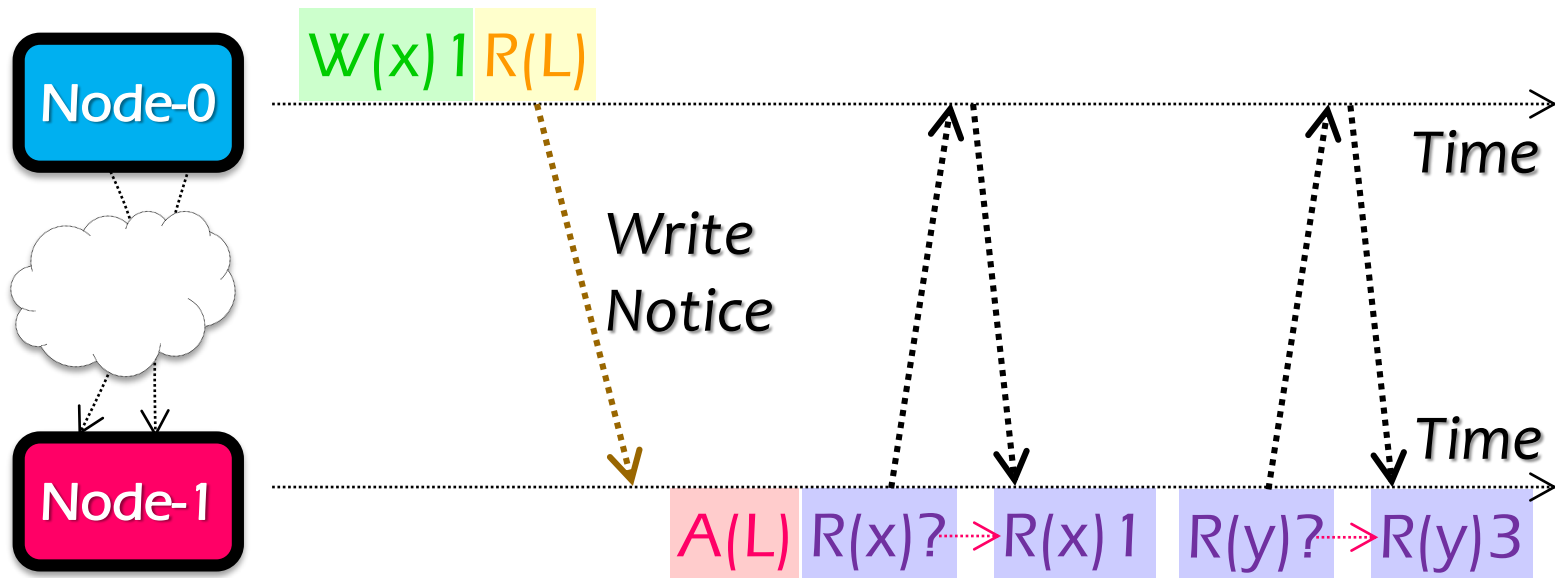


SynChronization Protocol

Invalidations are smaller than the updates

- The bigger “coherency unit”, the bigger “diff”

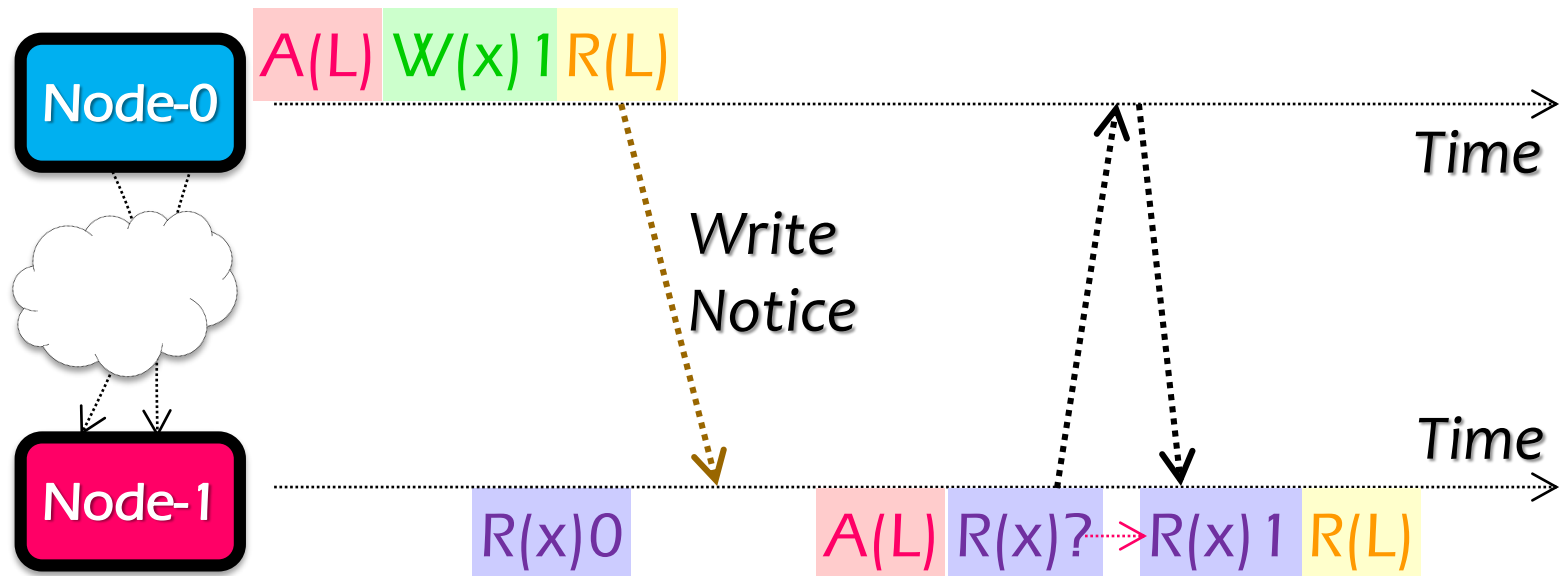
In invalidate-base protocol, there can be significant overhead due to access misses



Eager Release Consistency

Sending all accumulated modification in release to all caching processes

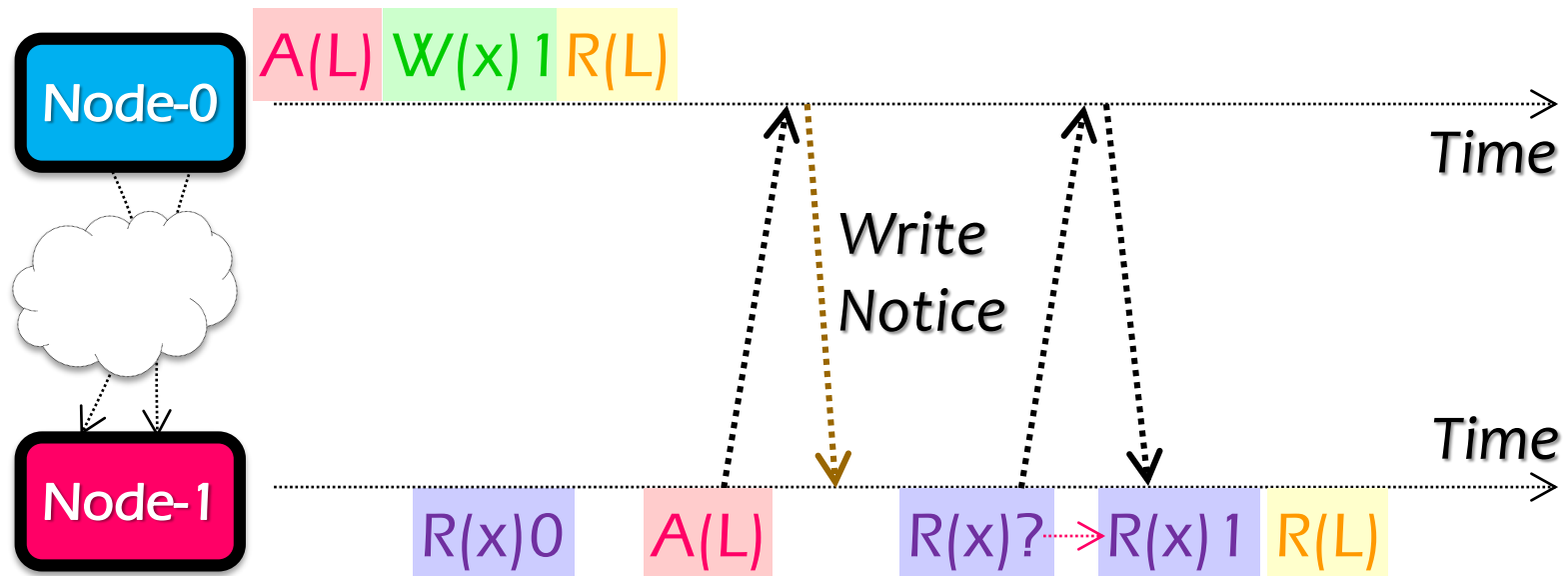
No synchronization operations on acquire



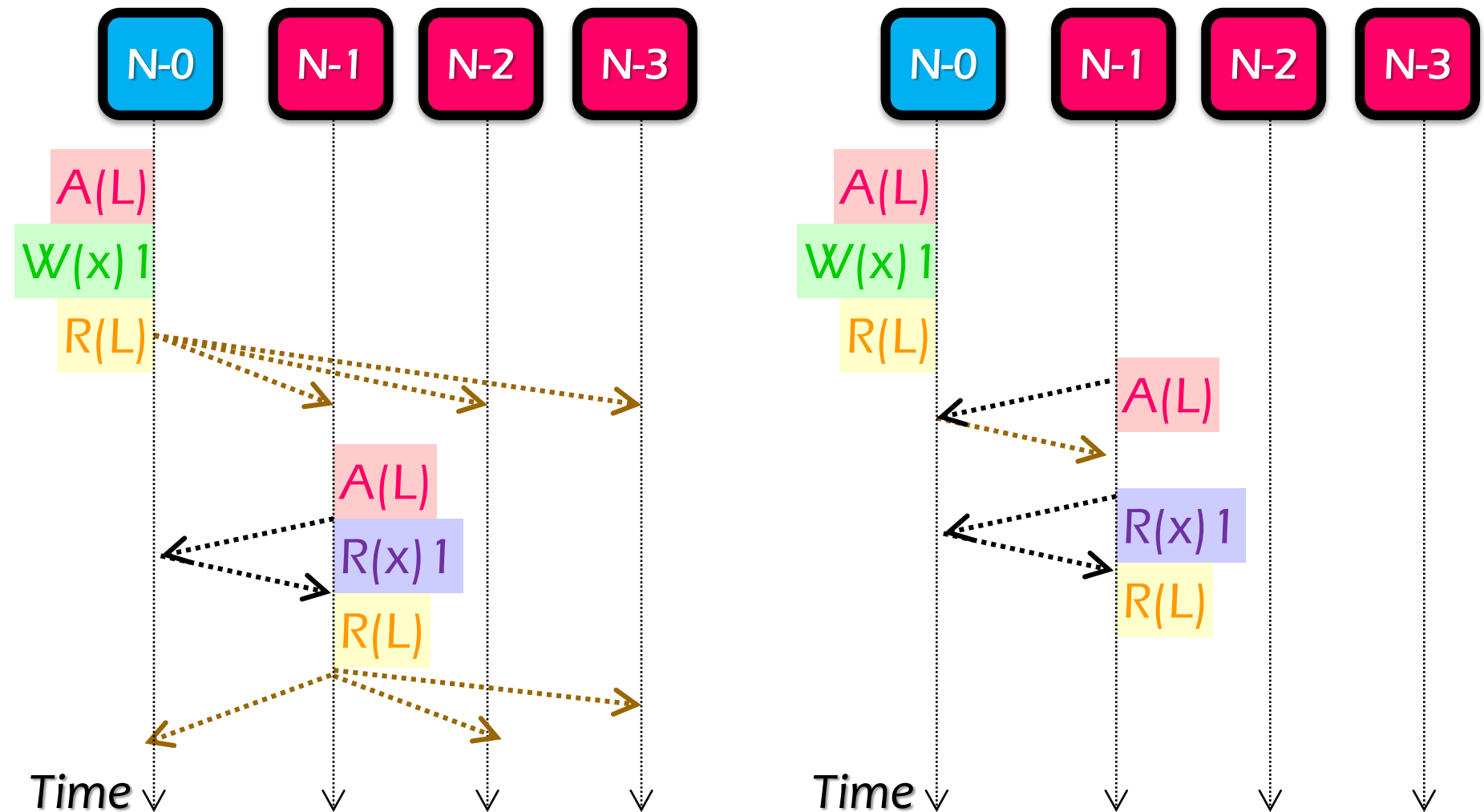
Lazy Release Consistency

Postpone the sending of modifications until a remote process actually need them(acquire)

No need to get irrelevant modifications



Eager RC vs. Lazy RC



Multiple Write Protocol

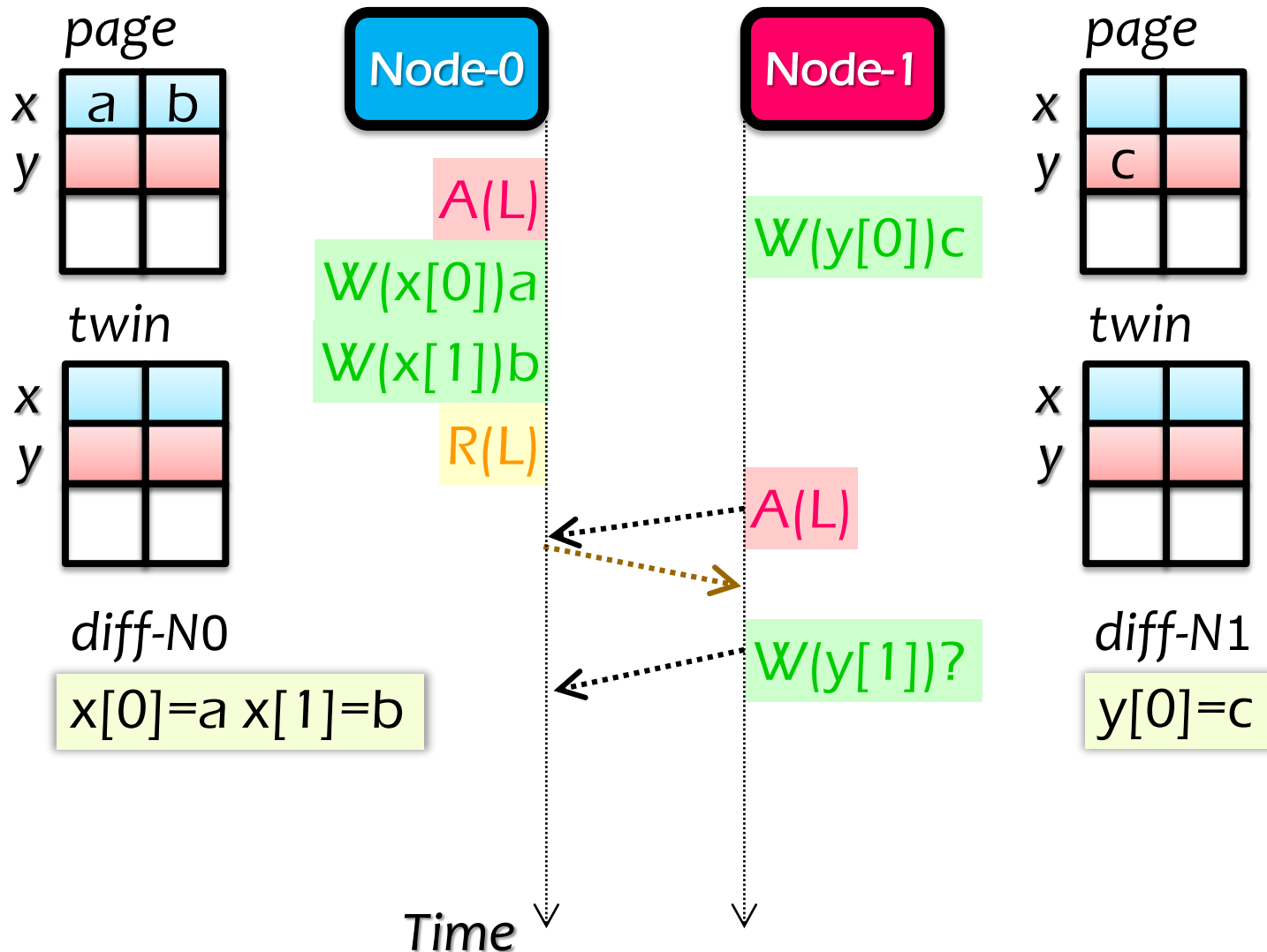
Modifications are detected by **twinning**

- Create a **twin** of page before the first change
- Generate a **diff** by comparing the final page with its twin when releasing

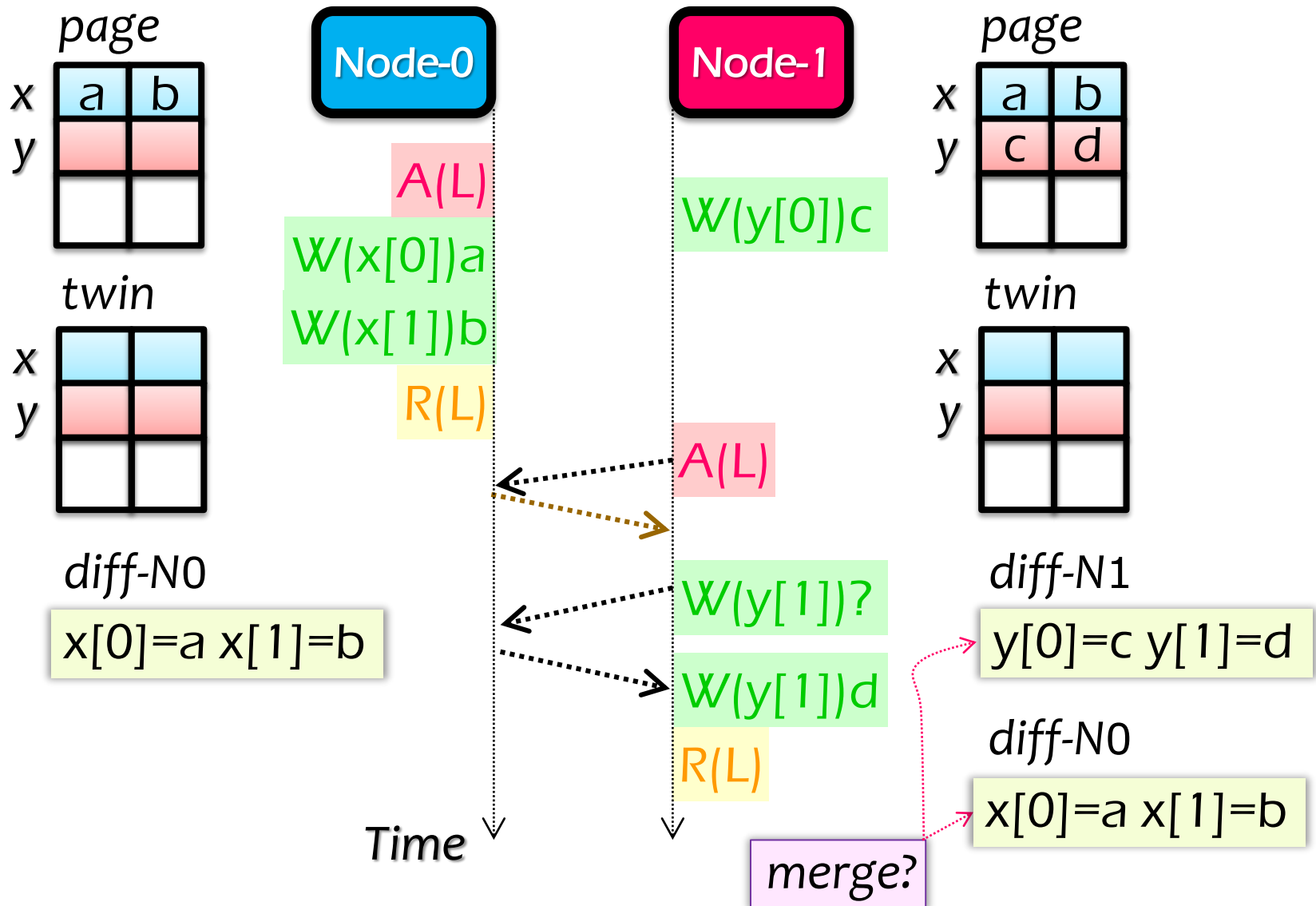
Twinning and **diffing** not only allow multiple writers, but also reduce communication

- Kill the false sharing
- Sending a diff is cheaper than sending an entire page

Multiple Write Protocol



Multiple Write Protocol



Intervals

Satisfying the happened-before relationship between all operations is enough to LRC

- Maintenance and usage of such a detailed ordering is very expensive

For LRC, the ordering is applied to process intervals

- Intervals are segments of time in the execution of a single process
- New interval begins each time a process executes a synchronization ops (**acquire/release**)

Happened-before Intervals

An interval i_1 precedes an interval i_2 according to happened-before of intervals,
← all accesses in i_1 precede accesses in i_2 according to the happened-before of accesses.

A happened-before partial order is define between intervals

Lamport Timestamps

A simple algorithm used to determine the order of events in a distributed computer system

1. A process increments its counter before each event in that process
2. When a process sends a message, it includes its counter value with the message
3. The receiver sets its counter to be greater than the *maximum* of its own value and the received value before it considers the message received

Vector Timestamps

An interval is said to be performed at a process if all interval's access have been performed at that process

Each process p has vector timestamp V_p that tracks which intervals have been performed at that process

- A vector timestamps consists of a set of interval indices, one per process in the system

Vector Timestamps

Vector timestamp is like vector clock

- The lock grant message from releaser to acquirer contains the current timestamp of the releaser
- 1. Just before executing a **release** or **acquire** in P :
$$V_p[q] = V_p[q] + 1$$
- 2. A lock grant message m is time-stamped with
$$t(m) = V_p$$
- 3. Upon **acquire** for every q :
$$V_p[q] = \max\{V_p[q], t(m)[q]\}$$

Review Write Notice

Write Notice is an indication that a given page has been modified

- Write notices are sent in the lock grant message along with the vector timestamp of the releaser

Each process keeps a set of covered intervals

- An interval x of process q is said to be covered:
 V_p if $V_p[q] \geq x$

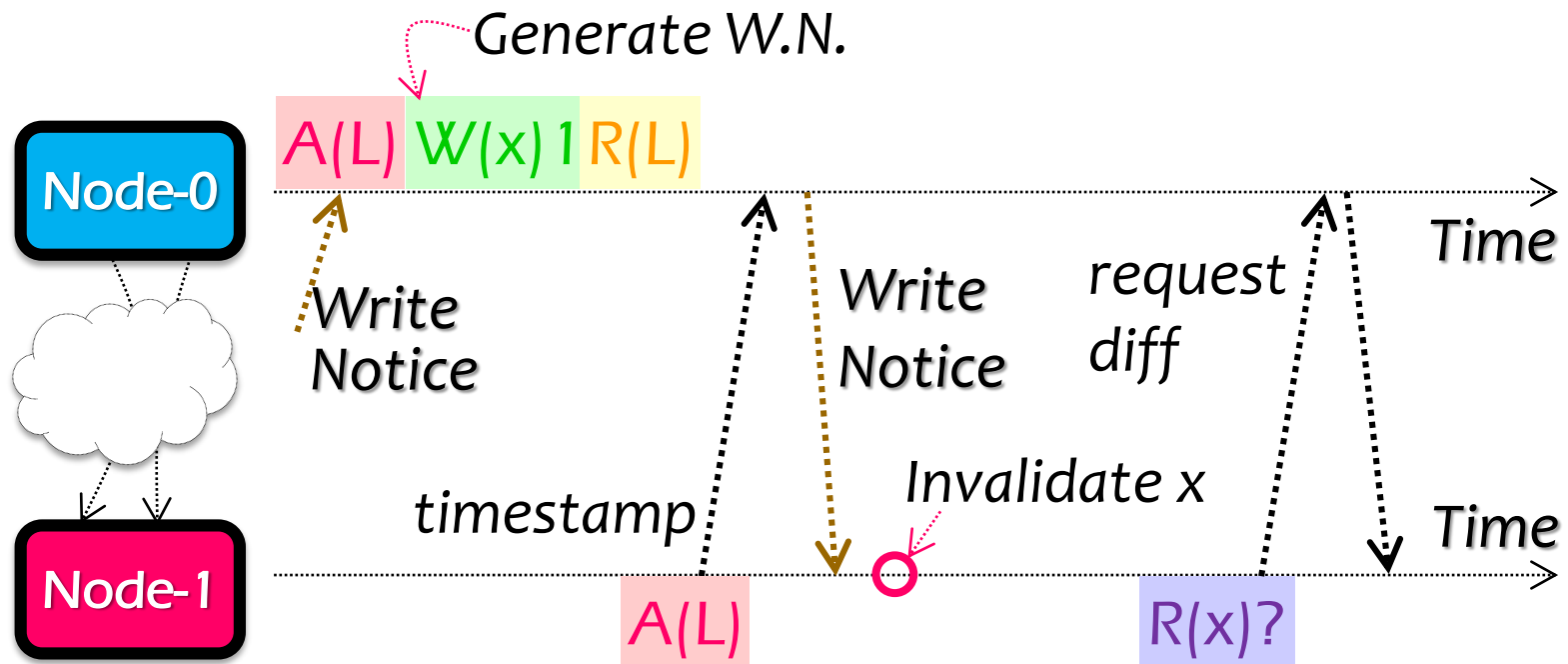
Only send write notices belonging to intervals uncovered by acquirer's vector timestamp

- Send timestamp inside a lock request message

Review Write Notice

Only send write notices belonging to intervals uncovered by acquirer's vector timestamp

- Acquirer sends its timestamp inside a lock request message to releaser



Next Lecture

Topic: Eventual Consistency

Suggested Paper

Wyatt Lloyd et al,

“Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS”. ACM Symposium on Operating Systems Principles (SOSP), **2011**

THANKS