

Distributed Systems

[Fall 2012]

Lec 11: Consistency Models

Slide acks: Jinyang Li, Robert Morris

(<http://pdos.csail.mit.edu/6.824/notes/l06.txt>,
<http://www.news.cs.nyu.edu/~jinyang/fa09/notes/ds-consistency.pdf>)

Reminder: Distributed File Systems

- We discussed about three: NFS, AFS, GFS
- Each has its own design, which is oriented toward a particular workload and requirements
- Each trades a form of consistency for performance and scale
 - Remember consistency properties for NFS/AFS/GFS?
 - Remember some mechanisms by which they achieve these?

Reminder: Distributed File Systems

- We discussed about three: NFS, AFS, GFS
- Each has its own design, which is oriented toward a particular workload and requirements
- Each trades a form of consistency for performance and scale
 - AFS: close-to-open semantic
 - NFS: periodic refreshes, close-to-open semantic
 - GFS: atomic-at-least-once record appends

today

But how do these compare? What's "right"?
What do these mean for applications/users?
We need some baselines to judge...

Example: GFS Consistency

- GFS provides:
 - Hardly any guarantees for normal writes
 - **At-least-once atomic appends**
- Record Appends:
 - The client specifies only the data, not the file offset
 - If record fits in chunk, primary chooses the offset and communicates it to all replicas → *offset is arbitrary*
 - If record doesn't fit in chunk, the chunk is padded → *file may have blank spaces*
 - If a record append fails at any replica, the client retries the operation → *file may contain record duplicates*

Implications for Applications

- GFS' consistency is not completely intuitive or generally applicable
- Applications must adapt to its weak semantics – how?
 - Rely on appends rather on overwrites
 - Write self-validating records
 - Checksums to detect and remove *padding*
 - Write self-identifying records
 - Unique Identifiers to identify and discard *duplicates*
- Hence, programmers need be very careful!
 - And applications need be amenable for weak semantics

Today: Consistency Models

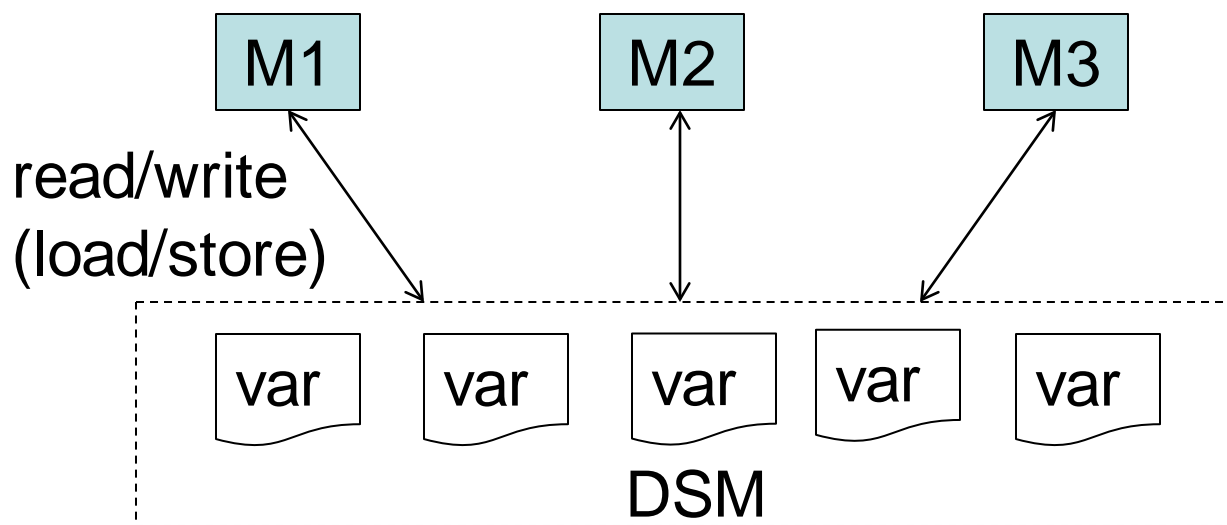
- We'll look at “standard” consistency models
 - Properties we wish we'd have, or that are generally applicable and well understood
- Today: strong consistency models
 - Strict consistency, sequential consistency
- Next time: weaker consistency models
 - Eventual consistency
- We'll relate NFS/AFS/GFS' models to those “baseline” models

What Is Consistency?

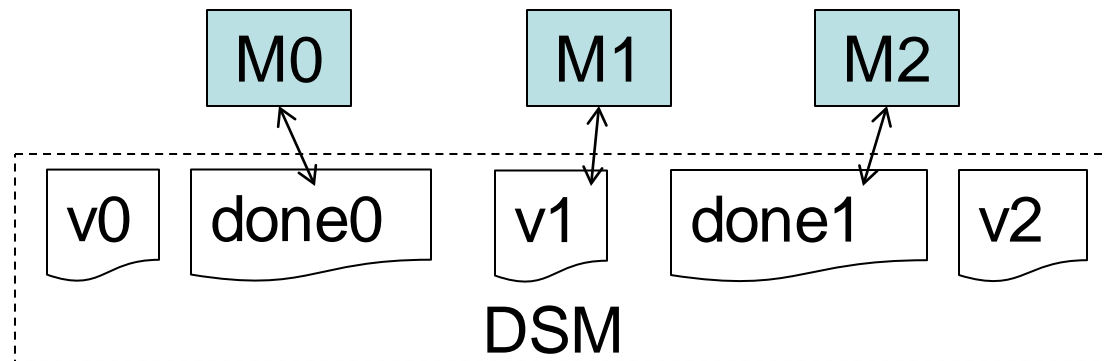
- Consistency = meaning of concurrent reads and writes on shared, possibly replicated, state
- As you've seen, it's a huge factor in many designs
- Choice trades off **performance/scalability** vs. **programmer-friendliness**
- Today we'll look at one case study: distributed shared memory
 - Concepts are similar to those in distributed storage systems, though

Distributed Shared Memory (DSM)

- Two models for communication in distributed systems:
 - message passing
 - shared memory
- Shared memory is often thought more intuitive to write parallel programs than message passing
 - Each machine can access a common address space



Example Application



M0:

```
v0 = f0();  
done0 = 1;
```

M1:

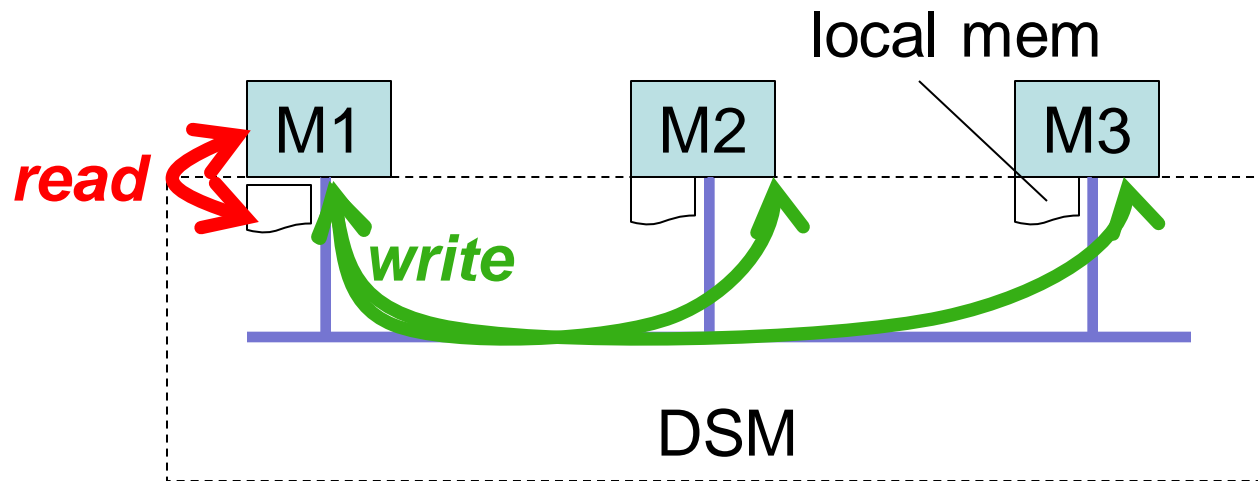
```
while (done0 == 0)  
    ;  
v1 = f1(v0);  
done1 = 1;
```

M2:

```
while (done1 == 0)  
    ;  
v2 = f2(v0, v1);
```

- What's the intuitive intent?
 - M2 should execute f2() with results from M0 and M1
 - waiting for M1 implies waiting for M0

Naïve DSM System

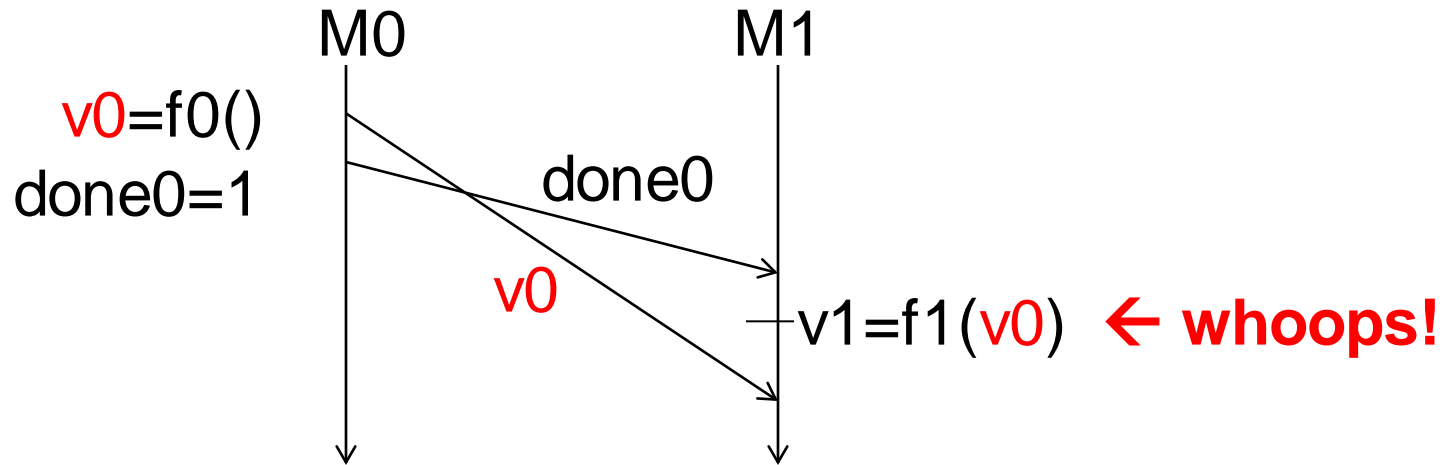


- Each machine has a **local copy of all of memory**
- Operations:
 - **Read**: from local memory
 - **Write**: send update msg to each host (but **don't wait**)
- **Fast**: never waits for communication

Question: Does this DSM work well for our application?

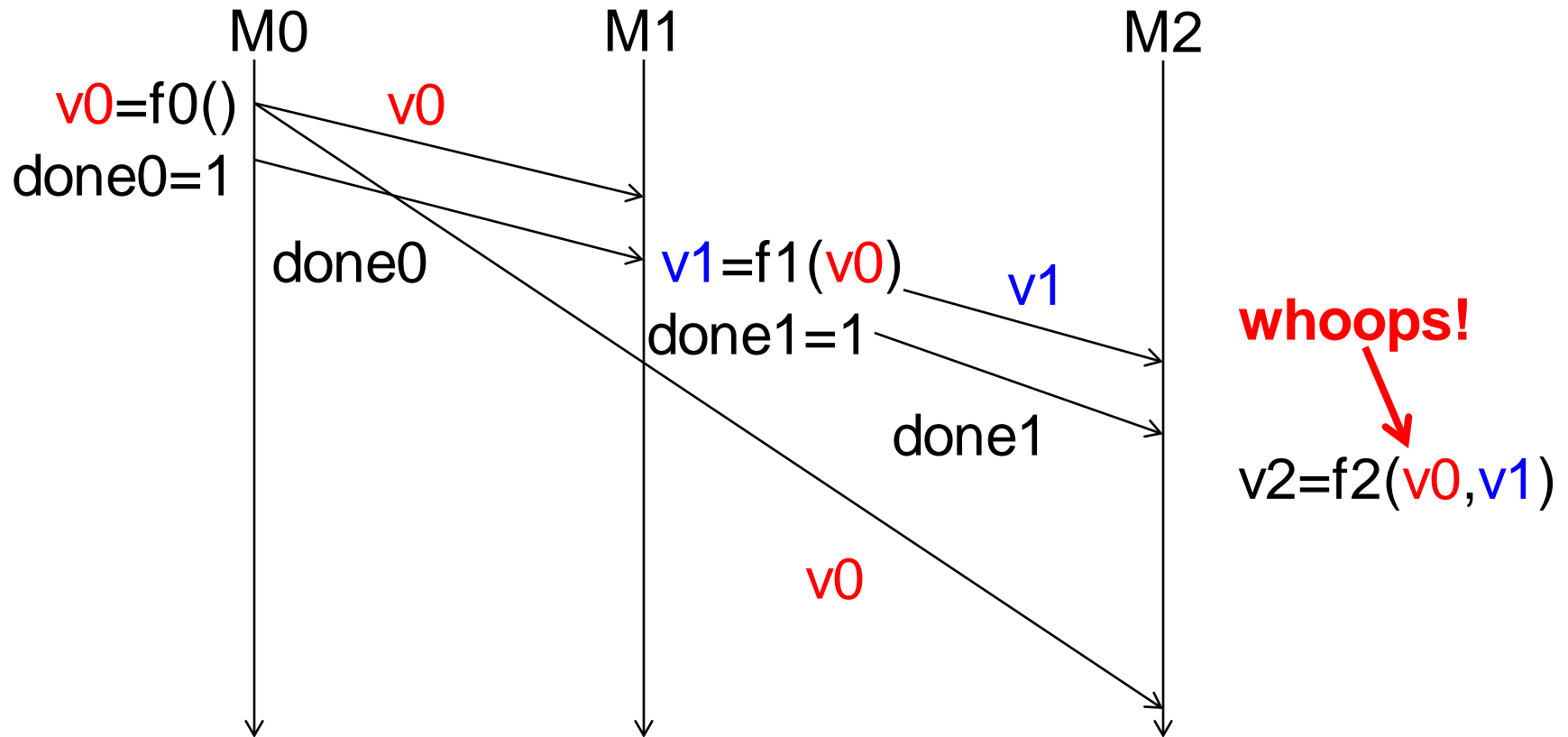
Problem 1 with Naïve DSM

- M0's $v0=...$ and $done0=...$ may be interchanged by network, leaving $v0$ unset but $done0=1$



Problem 2 with Naïve DSM

- M2 sees M1's writes before M0's writes
 - I.e. M2 and M1 disagree on order of M0 and M1 writes



Naïve DSM Properties

- Naive DSM is fast but has unexpected behavior
- Maybe DSM isn't “correct”
- Or maybe we should have never expected the example application to work as we did
 - I.e., maybe we need to fix the app, not the DSM...

Consistency Models

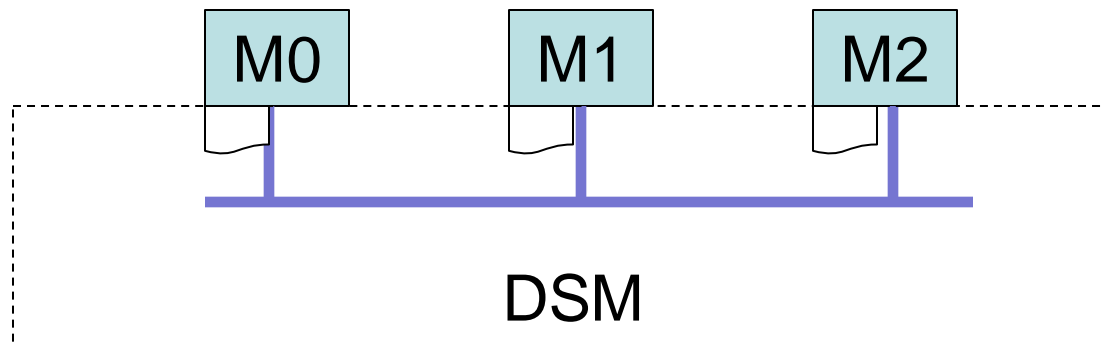
- Memory system promises to behave according to certain **rules**, which constitute the system's "consistency model"
 - We write programs assuming those rules
- The rules are a "contract" between memory system and programmer

Challenges

- No right or wrong consistency models
 - Tradeoff between ease of programmability and efficiency
 - E.g. what's the consistency model for web pages?
 - What should it be for a shared memory system?
- Consistency is hard in (distributed) systems:
 - Data replication (caching)
 - Concurrency
 - Failures

Model 1: Strict Consistency

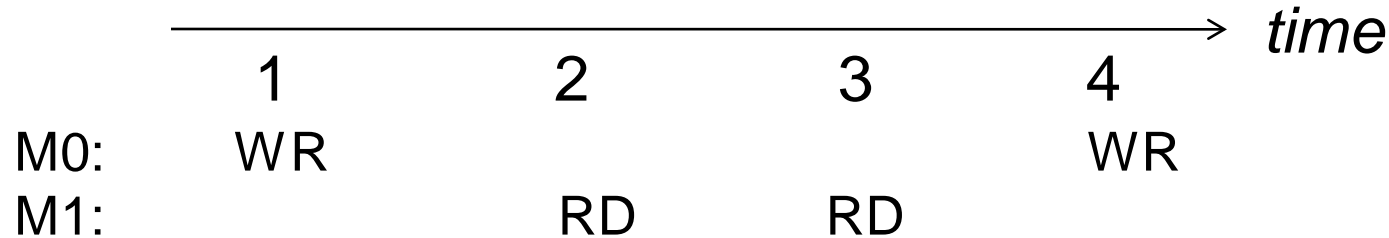
- Each operation is stamped with a **global wall-clock time**
- **Rules:**
 - Rule 1: **Each read gets the latest written value**
 - Rule 2: **All operations at one CPU are executed in order of their timestamps**



Does Strict Consistency Avoid Problems?

- Suppose we implement rules, can we still get problems?
 - Rule 1: Each read gets the latest written value
 - Rule 2: All operations at one CPU are executed in order of their timestamps
- Problem 1: Can M1 ever see v0 unset but done0=1?
- Problem 2: Can M1 and M2 disagree on order of M0 and M1 writes?
- So, strict consistency has very intuitive behavior
 - Essentially, the same semantic as on a uniprocessor!
- But how to implement it efficiently?
 - Without reducing distributed system to a uniprocessor...¹⁷

Implementing Strict Consistency



- To achieve, one would need to ensure:
 - Each read must be aware of, and wait for, each write
 - RD@2 aware of WR@1; WR@4 must know how long to wait...
 - Real-time clocks are strictly synchronized...
- Unfortunately:
 - Time between instructions \ll speed-of-light...
 - Real-clock synchronization is tough (pre-2012 ☺)
- So, strict consistency is tough to implement efficiently

Model 2: Sequential Consistency

- Slightly weaker model than strict consistency
 - Most important difference: doesn't assume real time
- **Rules:** There exists a **total ordering** of ops s.t.
 - Rule 1: Each machine's own ops appear in order
 - Rule 2: All machines see results according to total order (i.e. reads see most recent writes)
- We say that any runtime ordering of operations (also called a *history*) can be “explained” by a **sequential ordering of operations** that follows the above two rules

Does Seq. Consistency Avoid Problems?

- There is a total ordering of events s.t.:
 - Rule 1: Each machine's own ops appear in order
 - Rule 2: All machines see results according to total order
- **Problem 1:** Can M1 ever see v0 unset but done0=1?
 - M0's execution order was v0=... done0=...
 - M1 saw done0=... v0=...
 - Each machine's operations must appear in execution order so cannot happen w/ sequential consistency
- **Problem 2:** Can M1 and M2 disagree on ops' order?
 - M1 saw v0=... done0=... done1=...
 - M2 saw done1=... v0=...
 - This cannot occur given a single total ordering

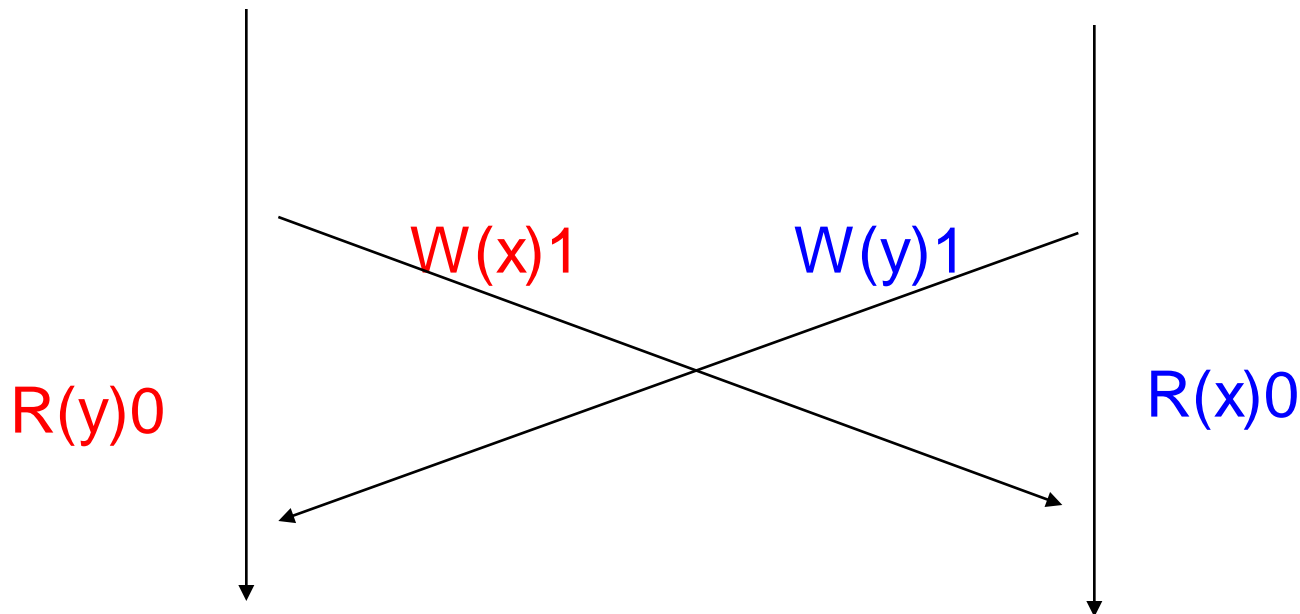
Seq. Consistency Is Easier To Implement Efficiently

- No notion of real time
- System has some leeway in how it interleaves different machines' ops
 - Not forced to order by op start time, as in strict consistency
- Performance is still not great
 - Once a machine's write completes, other machines' reads must see new data
 - Thus communication cannot be omitted or much delayed
 - Thus either reads or writes (or both) will be expensive

Sequential Consistency Requirements

1. Each processor issues requests in the order specified by the program
 - Do not issue the next request unless the previous one has finished
2. Requests to an individual memory location (storage object) 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

Naive DSM violates R1,R2



- Read from local state
- Send writes to the other node, but do not wait

~~R1~~: a processor issues read before waiting for write to complete
~~R2~~: 2 processors issue writes concurrently, no single order

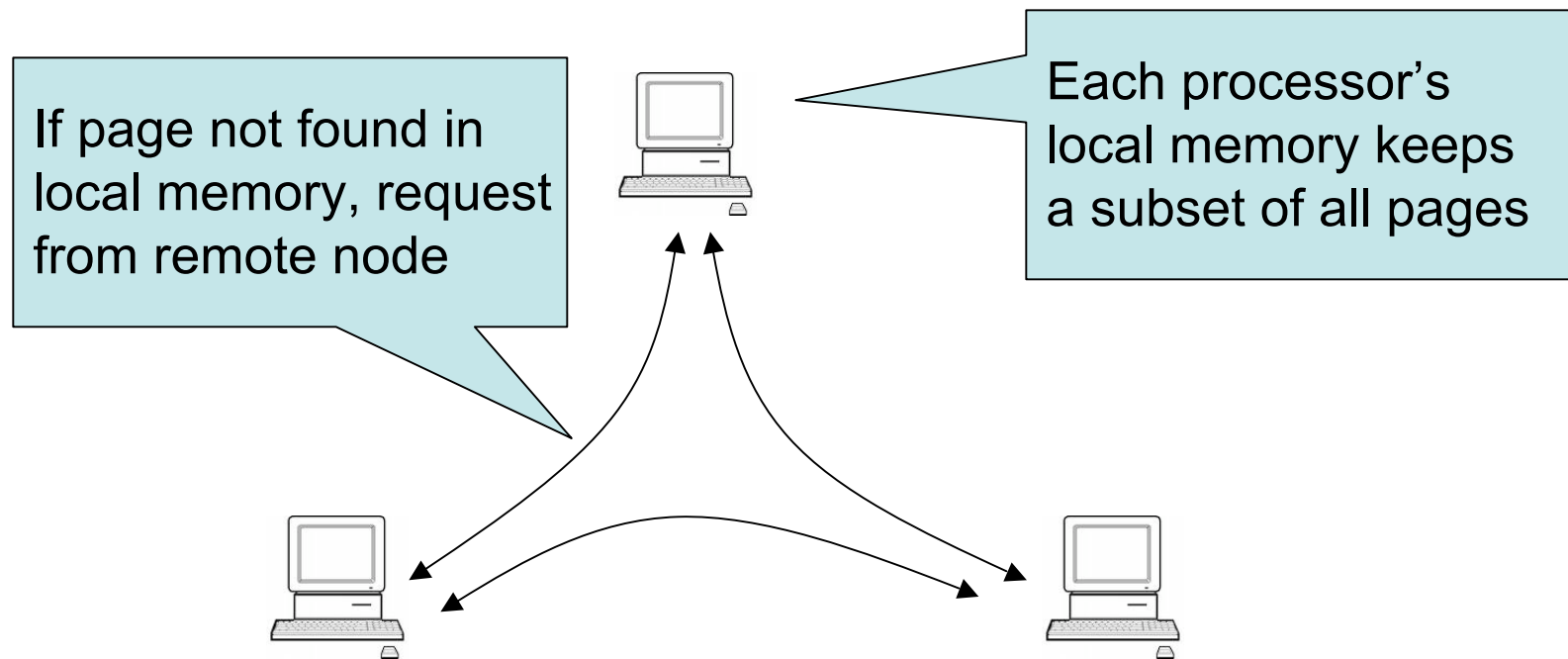
Case Study:

Ivy: Integrated shared Virtual memory at Yale

Ivy distributed shared memory

- What does Ivy do?
 - Provide a shared memory system across a group of workstations
- Why shared memory?
 - Easier to write parallel programs with than using message passing
 - We'll come back to this choice of interface in later lectures

Ivy architecture



- Each node caches read pages
 - Why?
- Can a node directly write cached pages?

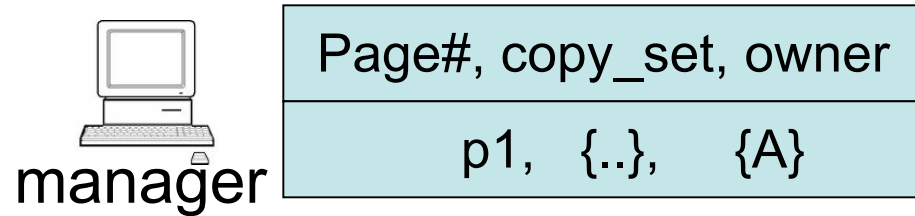
Ivy aims to provide sequential consistency

- How?
 - Always read a fresh copy of data
 - Must invalidate all cached pages before writing a page.
 - This simulates the FIFO queue for a page because once a page is written, all future reads must see the latest value
 - Only one processor (owner) can write a page at a time

Ivy implementation

- The ownership of a page moves across nodes
 - Latest writer becomes the owner
 - Why?
- Challenge:
 - how to find the owner of a page?
 - how to ensure one owner per page?
 - How to ensure all cached pages are invalidated?

Ivy: centralized manager



Page#, access
p1, read



A

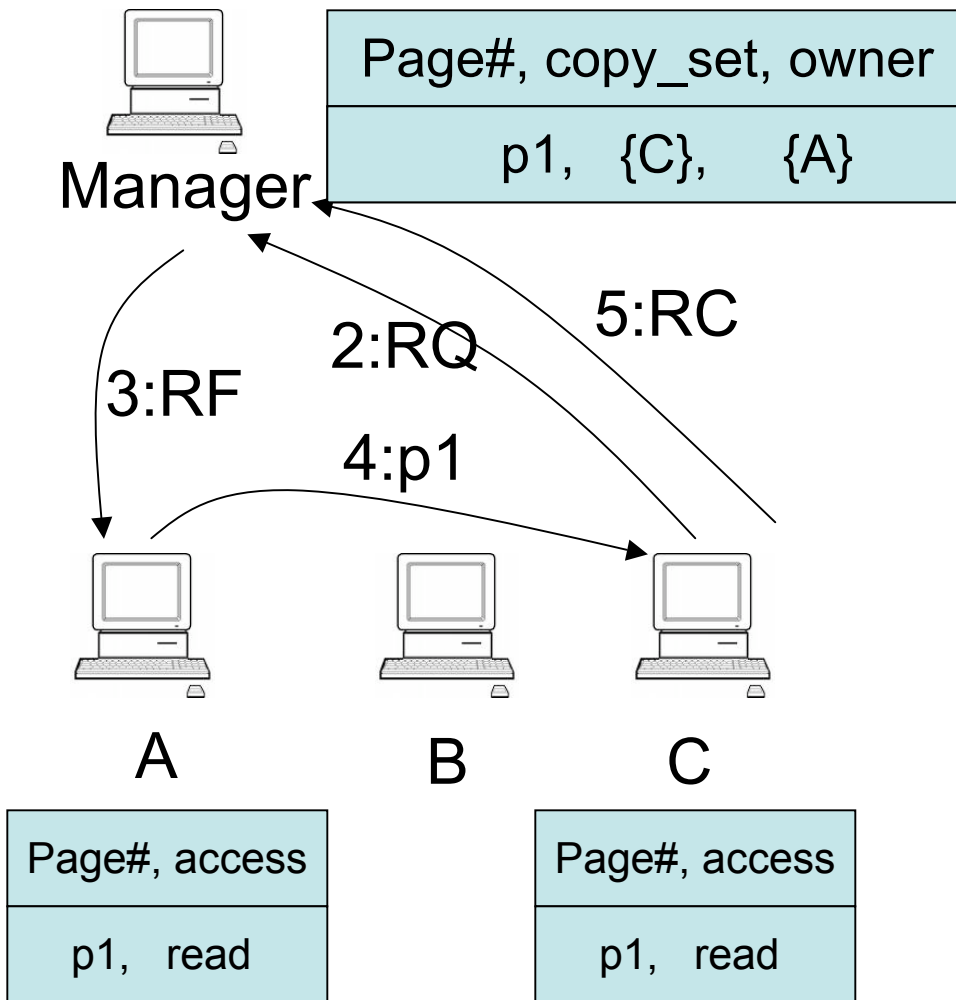


B



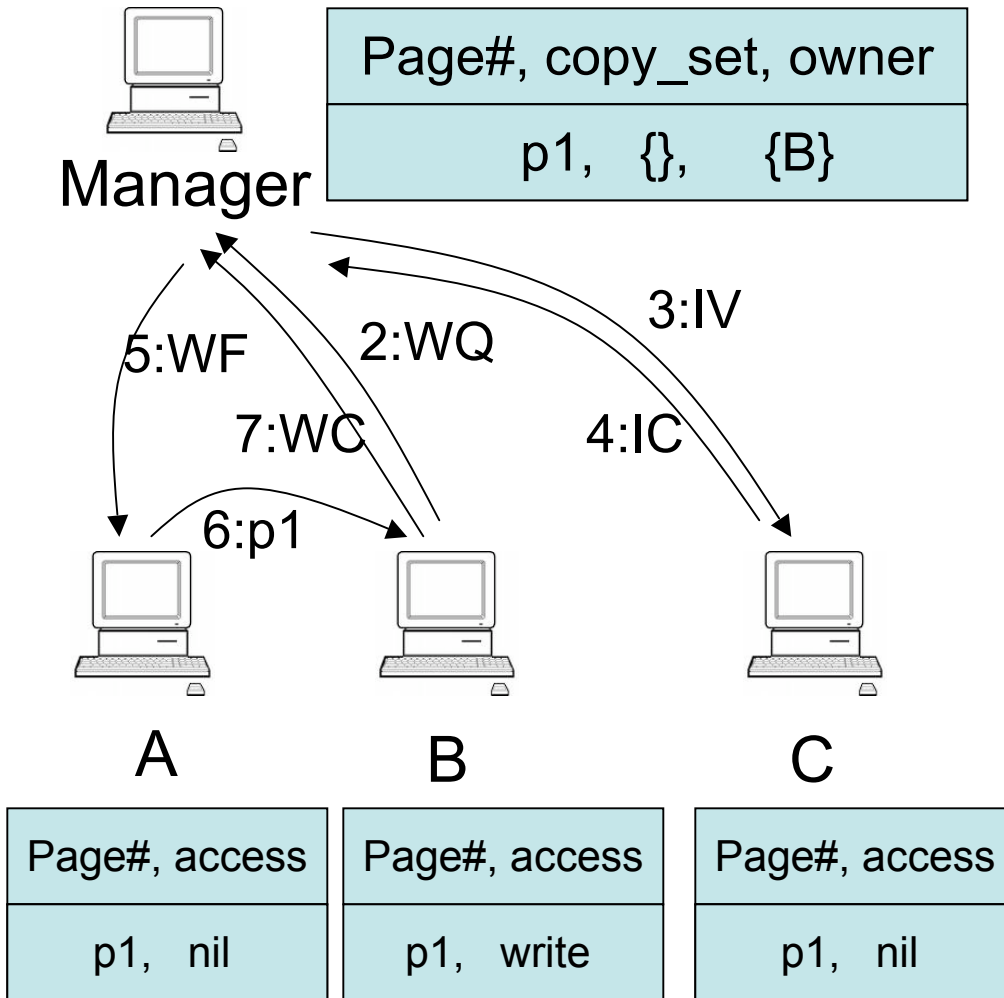
C

Ivy: read



1. Page fault for p1 on C
2. C sends RQ(read request) to M
3. M sends RF(read forward) to A, M adds C to copy_set
4. A sends p1 to C, C marks p1 as read-only
5. C sends RC(read confirmation) to M

Ivy: write



1. Page fault for p1 on B
2. B sends WQ(write request) to M
3. M sends IV(invalidate) to copy_set = {C}
4. C sends IC(invalidate confirm) to M
5. M clears copy_set, sends WF(write forward) to A
6. A sends p1 to B, clears access
7. B sends WC(write confirmation) to M

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

Is Ivy Sequentially Consistent?

- Well, yes, but we're not gonna prove it...
- Proof sketch:
 - Proof by contradiction that there is a schedule that cannot be explained by any sequential ordering that satisfies the two rules
 - This means that there are operations in the schedule that break one of the two rules
 - Reach contradiction on each of the two rules by using definition of reads/writes in Ivy
- For simplicity, let's look instead at why Ivy doesn't have the two problems we've identified for naïve DSM