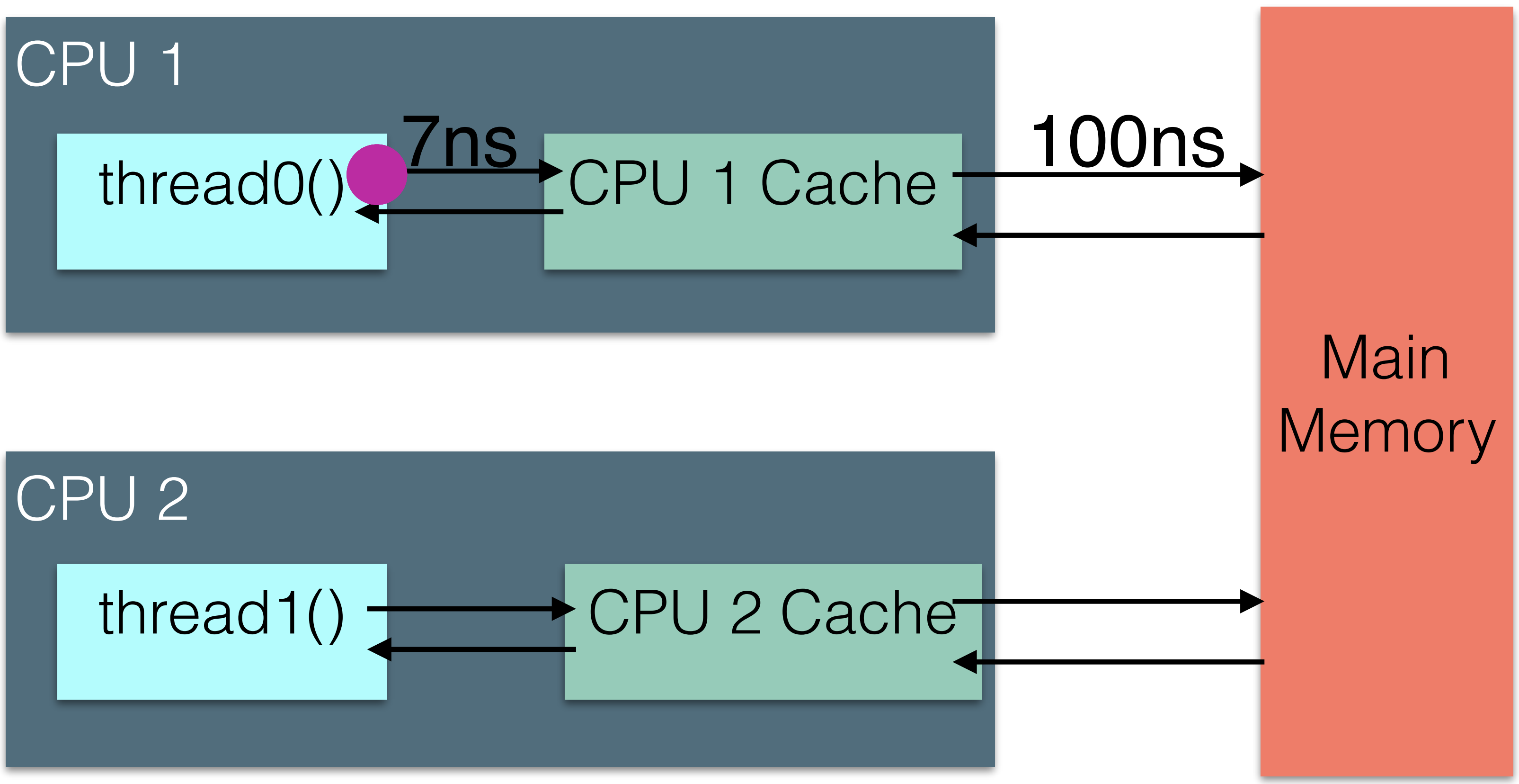
Consistency II

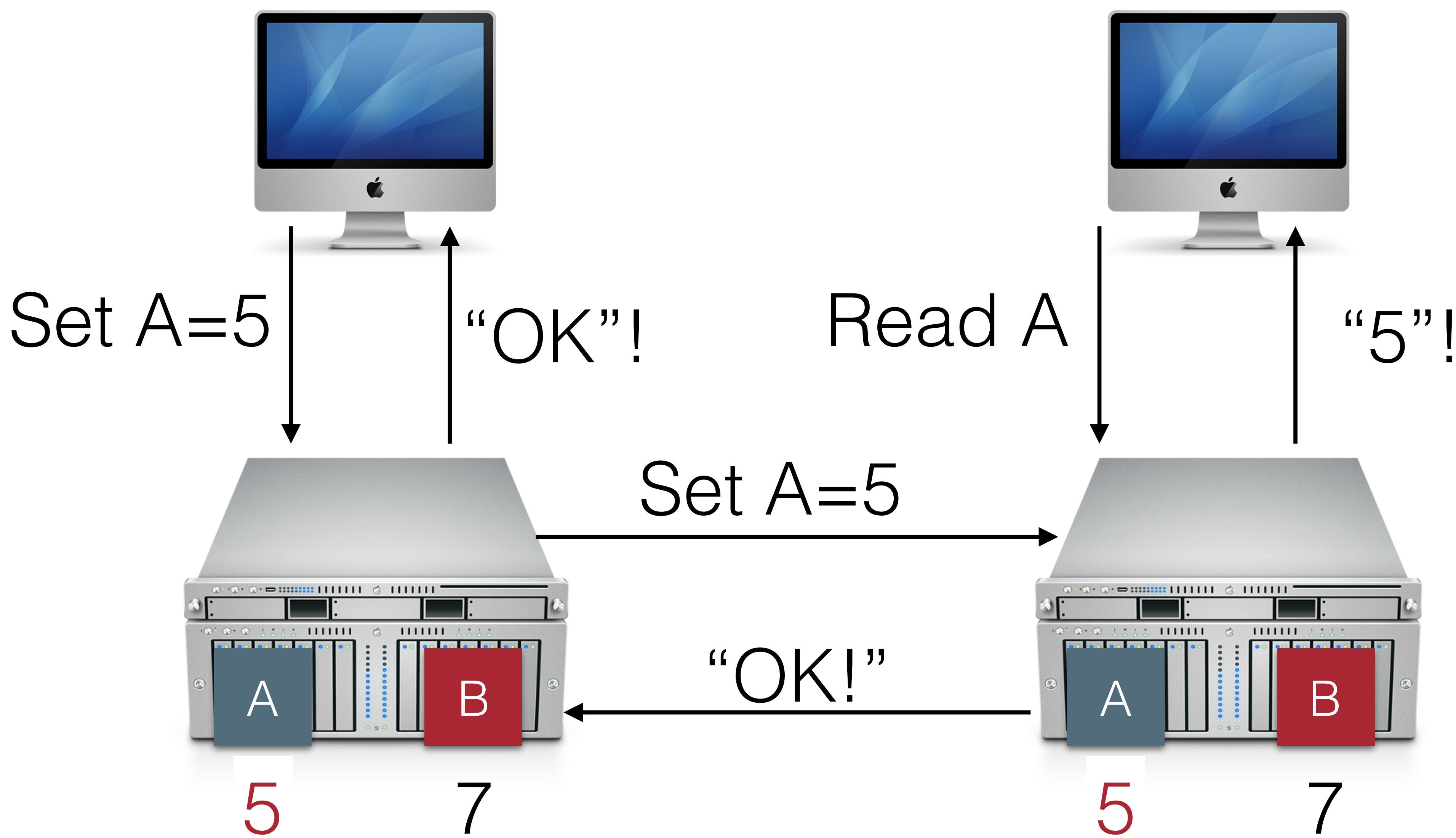
CS 475, Spring 2018
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

Review: Java Memory Model



Review: Consistency



Review: Sequential Consistency

- There is some *total order* of operations so that:
- Each CPUs operations appear in order
- All CPUs see results according to that order (read most recent writes)
- Consider this case, noting that there are **no locks** to enforce the ordering

P1	W(X) a			
P2		W(X) b		
P3			R(X) b	R(X) a
P4			R(X) b	R(X) a

Sequentially consistent. NOT strictly consistent

W(X)b, R(X)b,R(X)b,W(X)a, R(X)a, R(X)a

Review: Sequential Consistency

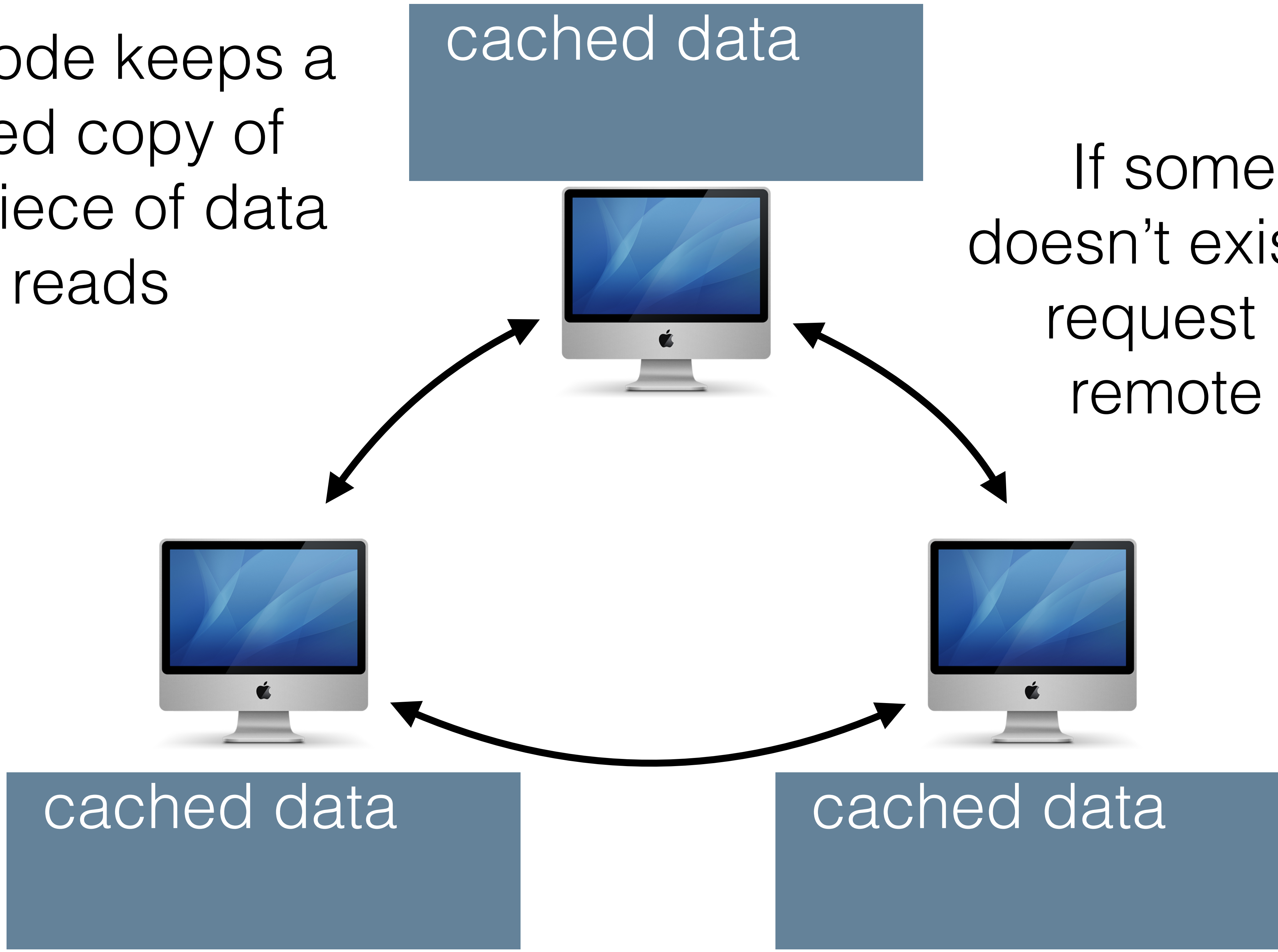
- There is some *total order* of operations so that:
- Each CPUs operations appear in order
- All CPUs see results according to that order (read most recent writes)
- Consider this case, noting that there are **no locks** to enforce the ordering

P1	W(X) a				
P2		W(X) b			
P3			R(X) b		R(X) a
P4				R(X) a	R(X) b

Not sequentially consistent

Review: Ivy Architecture

Each node keeps a cached copy of each piece of data it reads

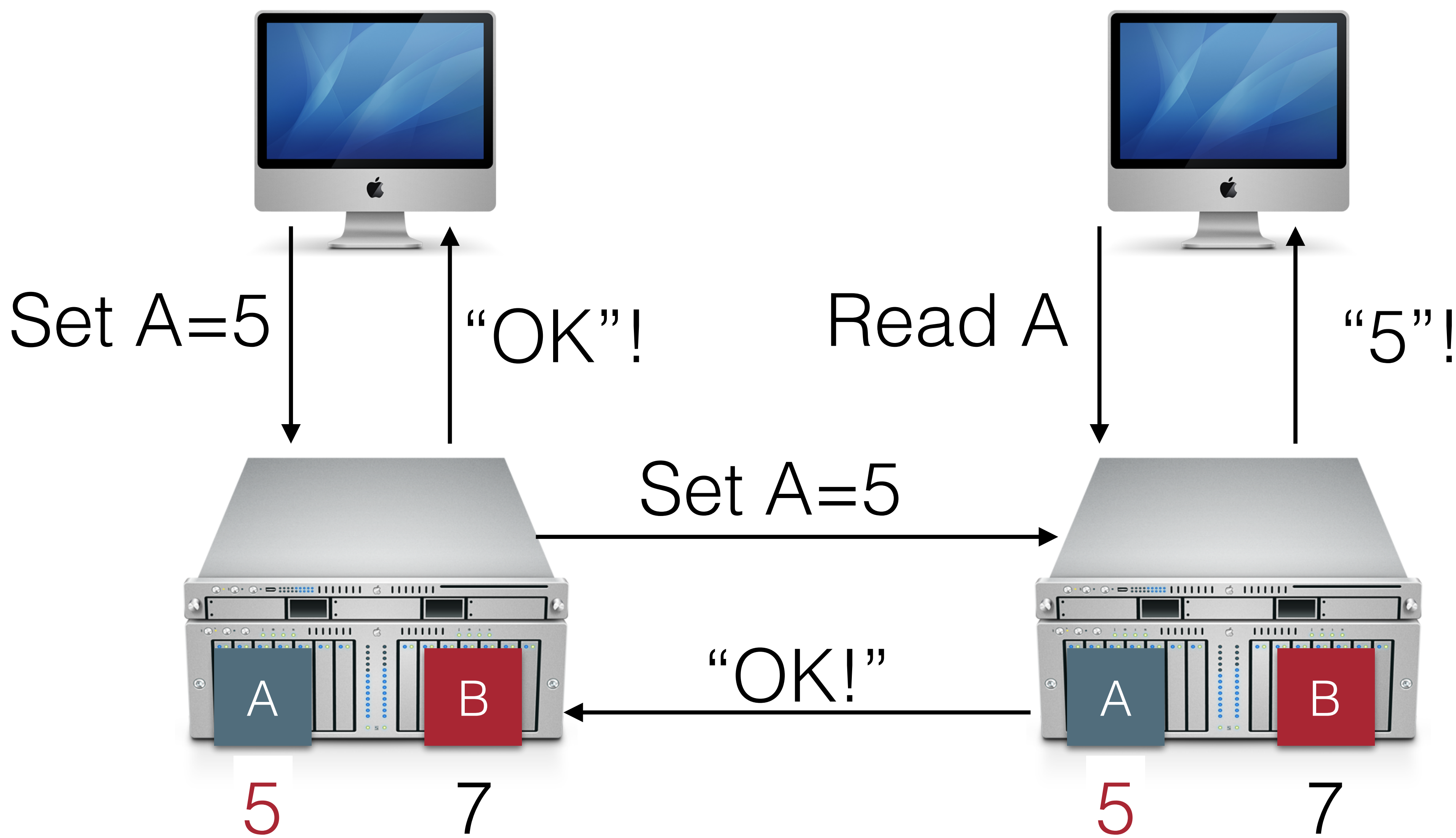


If some data doesn't exist locally, request it from remote node

Announcements

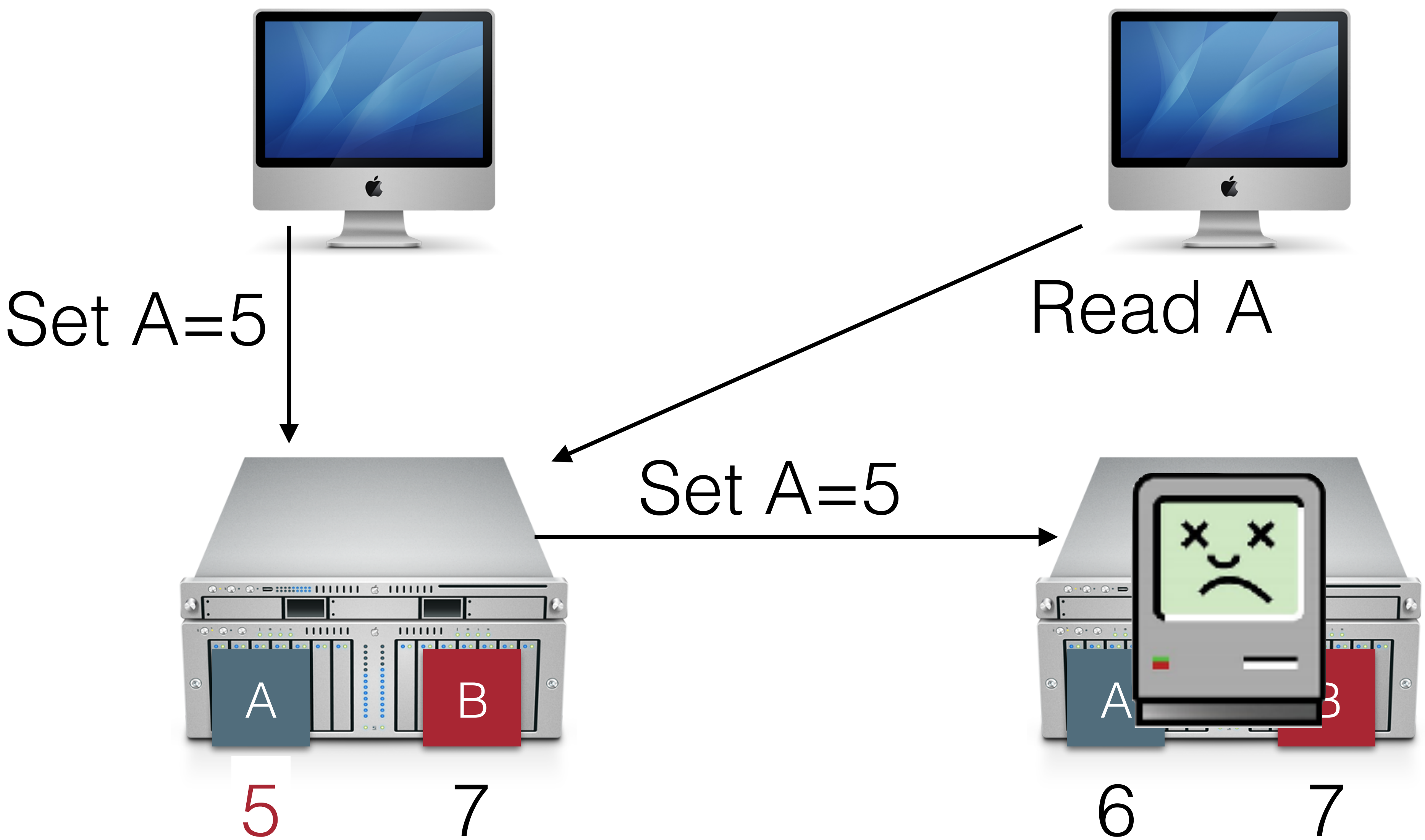
- HW4 is out!
  - http://www.jonbell.net/gmu-cs-475-spring-2018/homework-4/
- Today:
  - Relaxed consistency models
    - Causal consistency
    - Eventual consistency
  - File synchronization
    - Disconnected synchronization
- Road map: Project out on Weds. What's left?
  - Case studies & architectures. P2P. Security & failure modes
- Additional readings:
  - Tannenbaum 7.2-7.3

Sequential Consistency

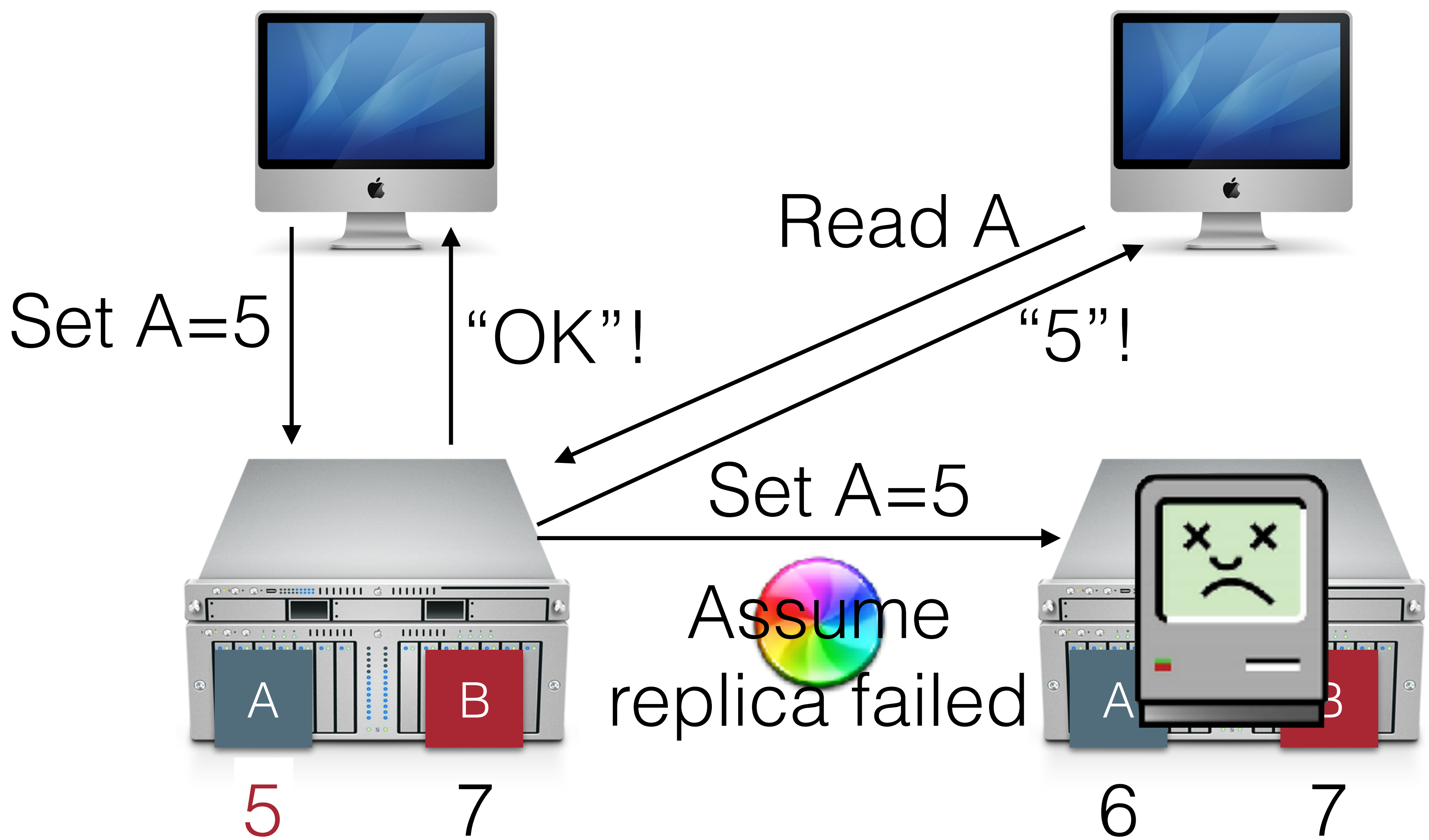


Availability

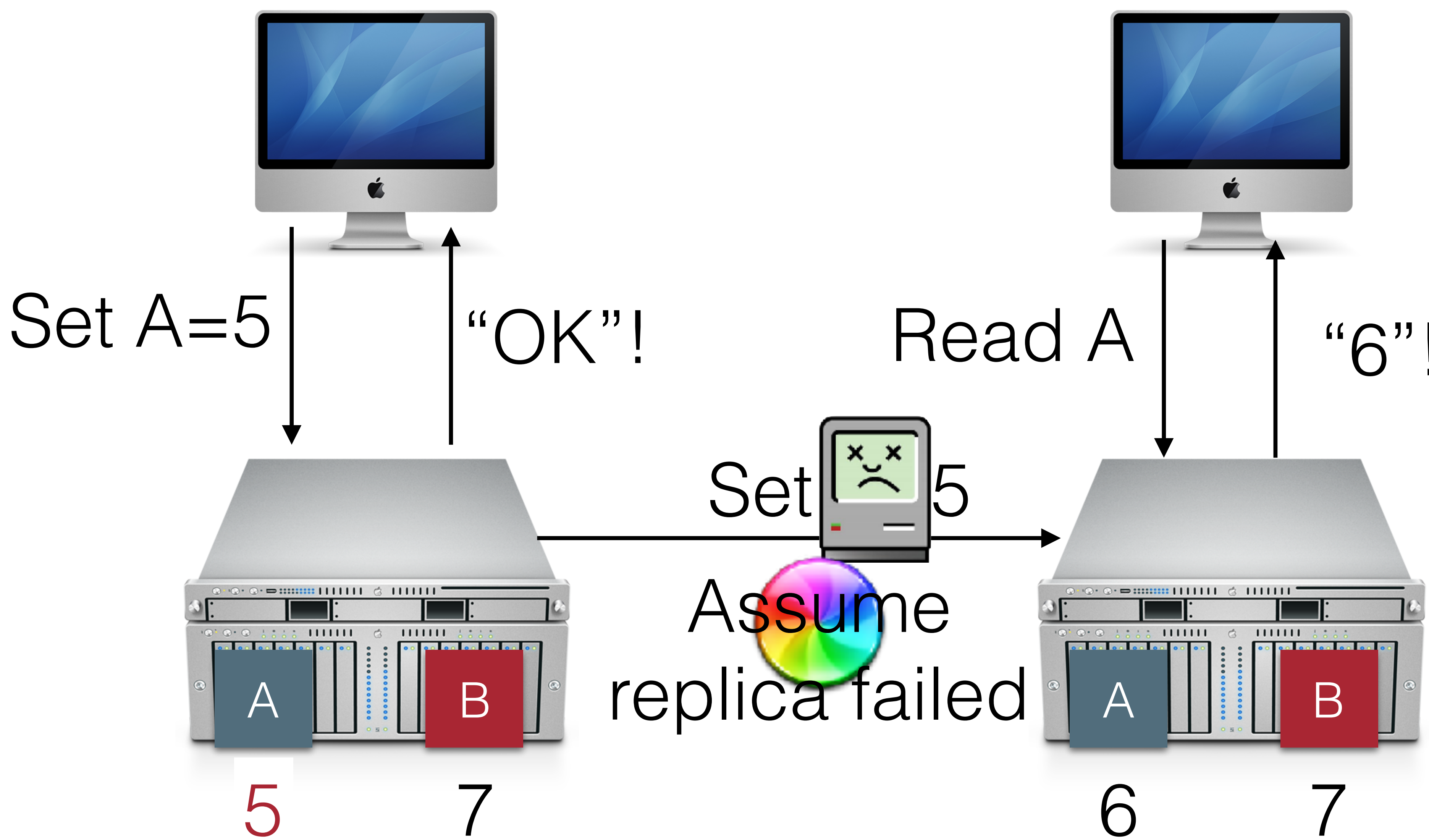
- Our protocol for sequential consistency does NOT guarantee that the system will be available!



Consistent + Available

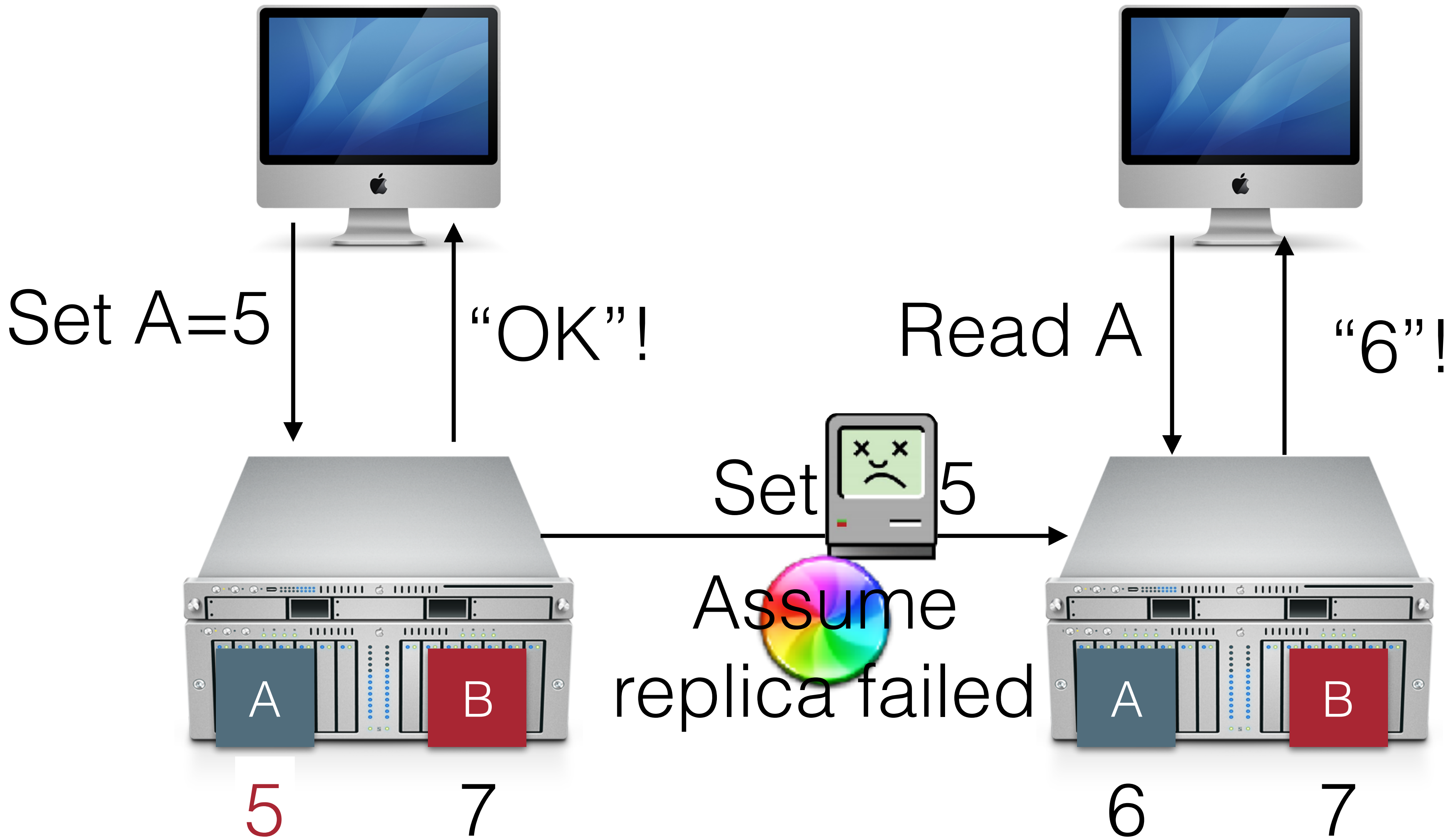


Still broken...



Network Partitions

- The communication links between nodes may fail arbitrarily
- But other nodes might still be able to reach that node



CAP Theorem

- Pick two of three:
    - Consistency: All nodes see the same data at the same time (sequential consistency)
    - Availability: Individual node failures do not prevent survivors from continuing to operate
    - Partition tolerance: The system continues to operate despite message loss (from network and/or node failure)
- You can not have all three, ever

Our goals as system builders

A property of the environment

CAP Theorem vs FLP

- FLP: Can not guarantee both liveness and agreement assuming messages may be delayed but are eventually delivered
- CAP: Can not guarantee consistency, availability, partition-tolerance assuming messages may be dropped
- Nice comparison: http://the-paper-trail.org/blog/flp-and-cap-arent-the-same-thing/

CAP Theorem

- C+A: Provide strong consistency and availability, assuming there are no network partitions
- C+P: Provide strong consistency in the presence of network partitions; minority partition is unavailable
- A+P: Provide availability even in presence of partitions; no sequential consistency guarantee, **maybe can guarantee something else**

Relaxing Consistency

- We can relax two design principles:
  - How stale reads can be
  - The ordering of writes across the replicas

Allowing Stale Reads

P1	W(X) 0		R(X)	R(X)		R(X)
P2		W(X) 1	R(X)		W (X) 0	R(X)
P3			R(X)	R(X)		R(X)

Allowing Stale Reads

```
class MyObj {
    int x = 0;
    int y = 0;

    void thread0()
    {
      x = 1;
      if(y==0)
        System.out.println("OK");
    }
    void thread1()
    {
      y = 1;
      if(x==0)
```

""

"OK"

"OK"
"OK"

Java's memory model is "relaxed" in that you can have stale reads

Relaxing Consistency

- Intuition: less constraints means less coordination overhead, less prone to partition failure

P1	W(X) 0		R(X) [0,1]	R(X) [0,1]		R(X) [0,1]
P2		W(X) 1	R(X) [0,1]		W (X) 0	R(X) [0,1]
P3			R(X) [0,1]	R(X) [0,1]		R(X) [0,1]

Naïve DSM

- Assume each machine has a complete copy of memory
- Reads from local memory
- Writes broadcast update to other machines, then immediately continue

```
class Machine1 {
 DSMInt x = 0;
 DSMInt y = 0;

 static void main(String[] args)
 {
   x = 1;
   if(y==0)
       System.out.println("OK");
 }
}
```

```
class Machine2 {
 DSMInt x = 0;
 DSMInt y = 0;

 static void main(String[] args)
 {
   y = 1;
   if(x==0)
       System.out.println("OK");
 }
}
```

Naïve DSM

- Assume each machine has a complete copy of memory
- Reads from local memory
- Writes broadcast update to other machines, then immediately continue

```
class Machine1 {
  DSMInt x = 1;
  DSMInt y = 0;

  static void main(String[] args)
  {
    x = 1;
    if(y==0)
        System.out.println("OK");
  }
}
```

```
class Machine2 {
  DSMInt x = 0;
  DSMInt y = 1;

  static void main(String[] args)
  {
    y = 1;
    if(x==0)
        System.out.println("OK");
  }
}
```

Naïve DSM

- Assume each machine has a complete copy of memory
- Reads from local memory
- Writes broadcast update to other machines, then immediately continue

```
class Machine1 {
  DSMInt x = 1;
  DSMInt y = 0;

  static void main(String[] args)
  {
   x = 1;
   if(y==0)
       System.out.println("OK");
  }
}
```

Is this correct?

```
ne2 {
= 0;
= 1;

  static void main(String[] args)
  {
   y = 1;
   if(x==0)
       System.out.println("OK");
  }
}
```

Naïve DSM

- It definitely is not sequentially consistent
- Are there any guarantees that it provides though?
    - Reads can be stale
    - Writes can be re-ordered
    - Not really.
- Can we come up with something more clever though with SOME guarantee?
    - (Not as is, but with some modifications maybe it's…)

Causal Consistency

- An execution is **causally-consistent** if all **causally-related** read/write operations are executed in an order that reflects their causality
- Reads are fresh ONLY for writes that they are dependent on
- Causally-related writes appear in order, but not in order to others
- Concurrent writes can be seen in different orders

Causal Consistency

P1	W(X)a		W(X)c		
P2		R(X)a	W(X)b		
P3		R(X)a		R(X)c	R(X)b
P4		R(X)a		R(X)b	R(x)c

Causally Consistent. W(X) b and W(X) c are not related, hence could have happened one either order.
W(X)a and W(X)B ARE causally related and must occur in this order

Causal Consistency

P1	W(X)a				
P2		R(X)a	W(X)b		
P3				R(x)b	R(x)a
P4				R(x)a	R(x)b

NOT Causally Consistent. X couldn’t have been b after it was a

P1	W(X)a			
P2		W(X)b		
P3			R(x)b	R(x)a
P4			R(x)a	R(x)b

Causally Consistent. X can be a or b concurrently

Why Causal Consistency?

- It is clearly **weaker** than sequential consistency
  - (Note that anything that is sequentially consistent is also causally consistent)
- Many more operations for concurrency
  - Parallel (non-dependent) operations can occur in parallel in different places
    - Sequential would enforce a global ordering
  - E.g. if W(X) and W(Y) occur at the same time, and without dependencies, then they can occur without any locking

Eventual Consistency

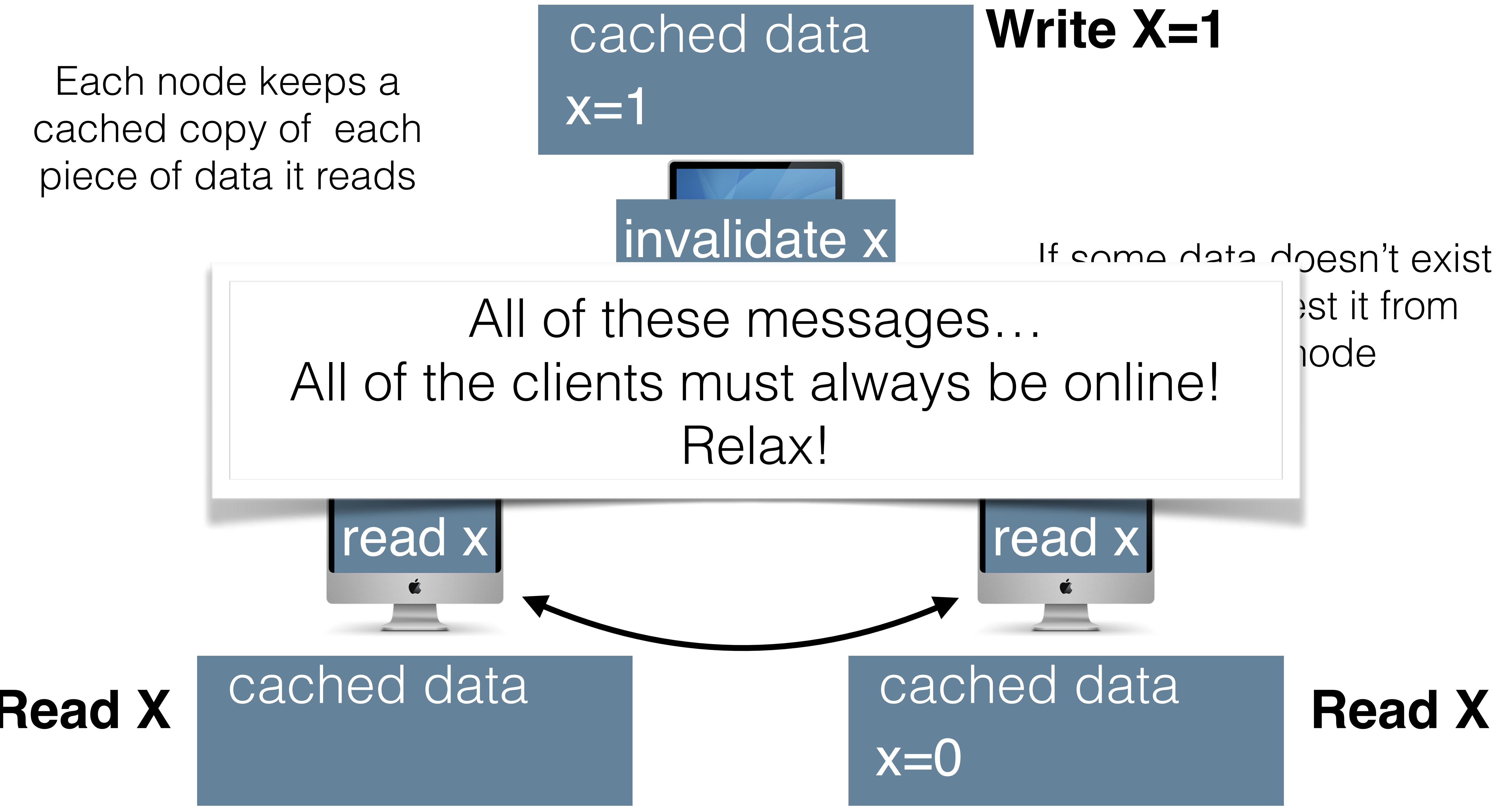
- Allow stale reads, but ensure that reads will **eventually** reflect the previously written values
  - Eventually: milliseconds, seconds, minutes, hours, years…
- Writes are NOT ordered as executed
  - Allows for conflicts. Consider: Dropbox
- Git is eventually consistent

Eventual Consistency

- More concurrency than strict, sequential or causal
  - These require **highly available** connections to send messages, and generate lots of chatter
- Far looser requirements on network connections
  - Partitions: OK!
  - Disconnected clients: OK!
  - Always available!
- Possibility for conflicting writes :(

Review: Ivy Architecture

Each node keeps a cached copy of each piece of data it reads

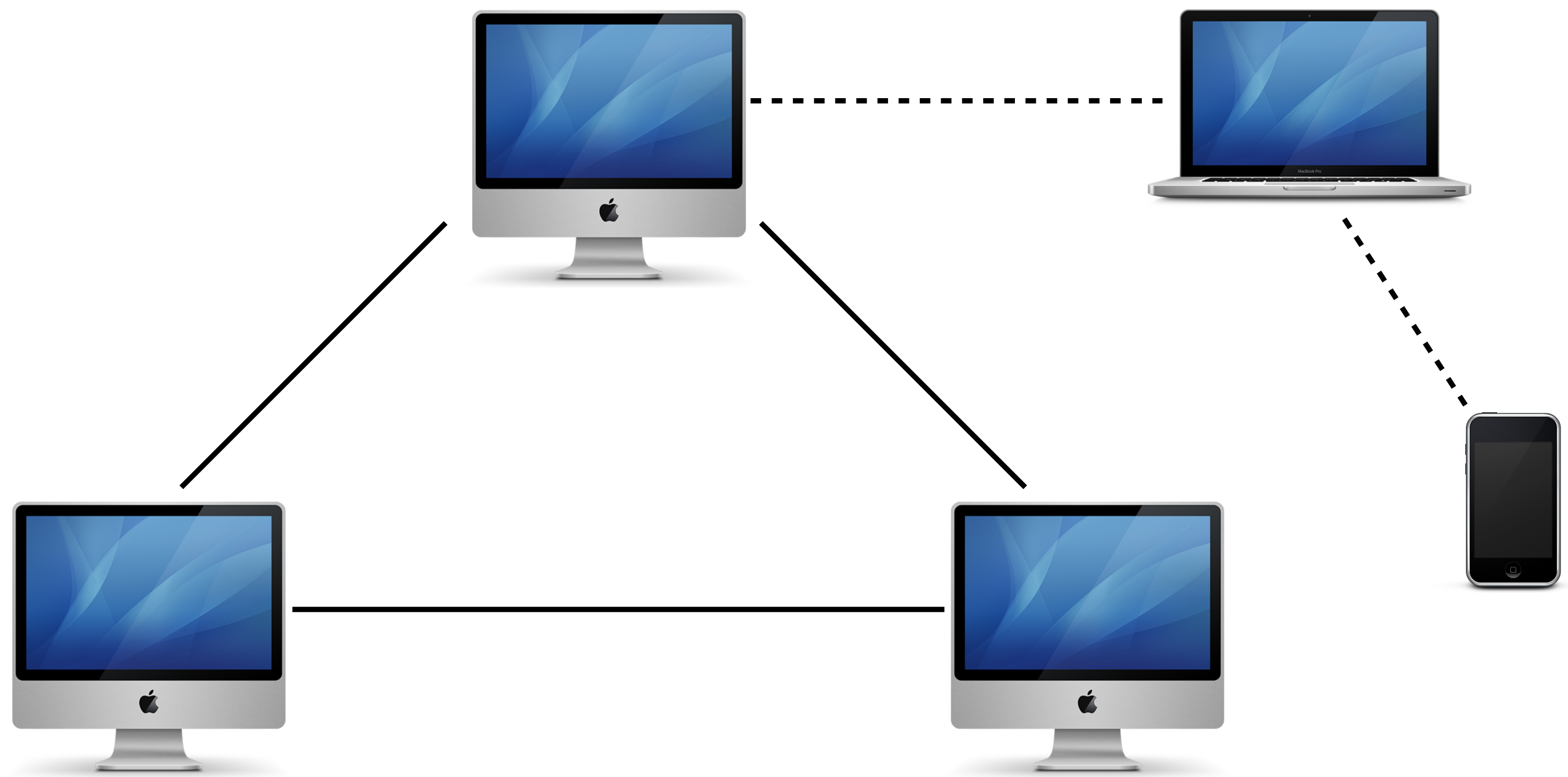


All of these messages…
All of the clients must always be online!
Relax!

Sequential vs Eventual Consistency

- Sequential: “Pessimistic” concurrency control
  - Assume that everything could cause a conflict, decide on an update order as things execute, then enforce it
- Eventual: “Optimistic” concurrency control
  - Just do everything, and if you can’t resolve what something should be, sort it out later
  - Can be tough to resolve in general case

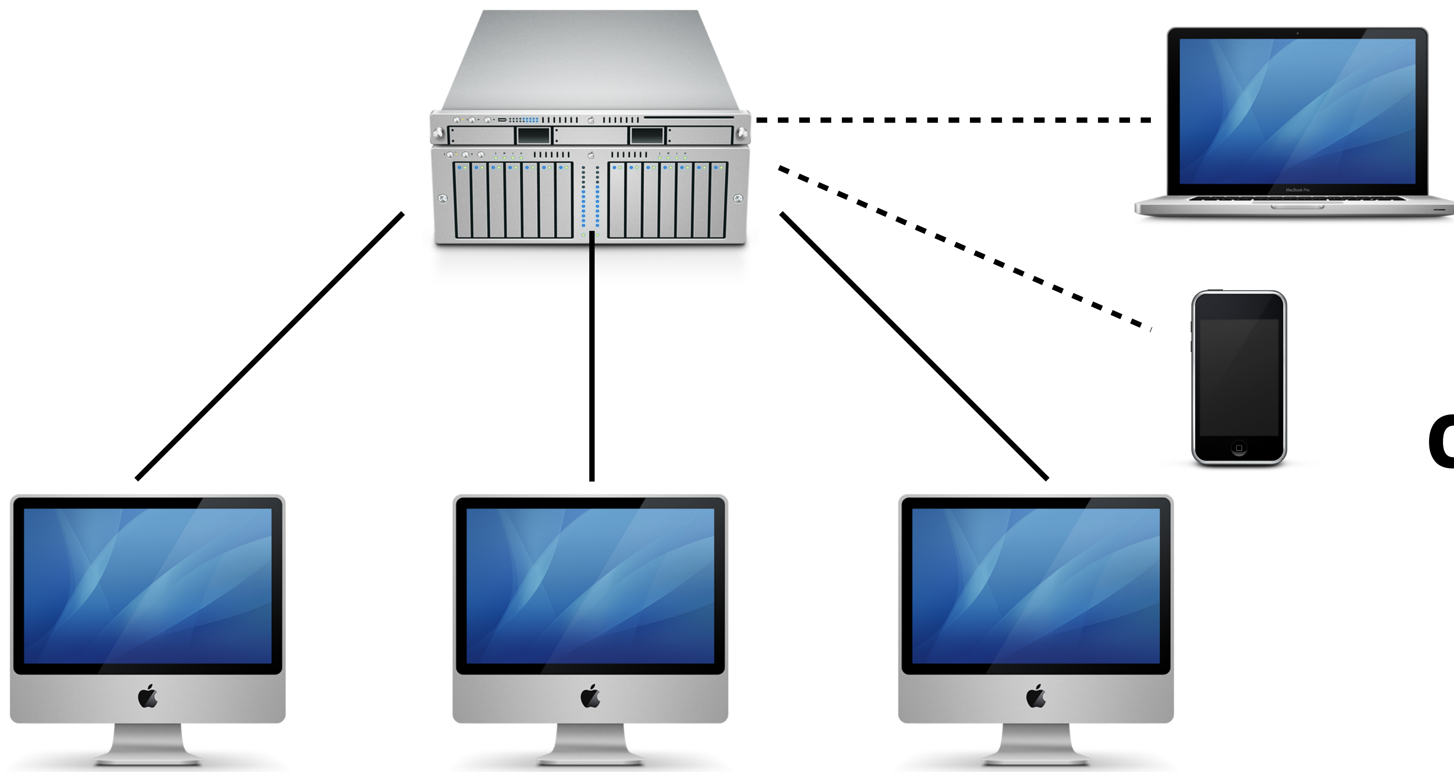
Eventual Consistency: Distributed Filesystem



When everything can talk, it's easy to synchronize, right?

Goal: Everything eventually becomes synchronized.
No lost updates (don't replace new version with old)

Eventual Consistency: Distributed Filesystem



Fix: Add coordinating sync server

When everything can talk, it's easy to synchronize, right?

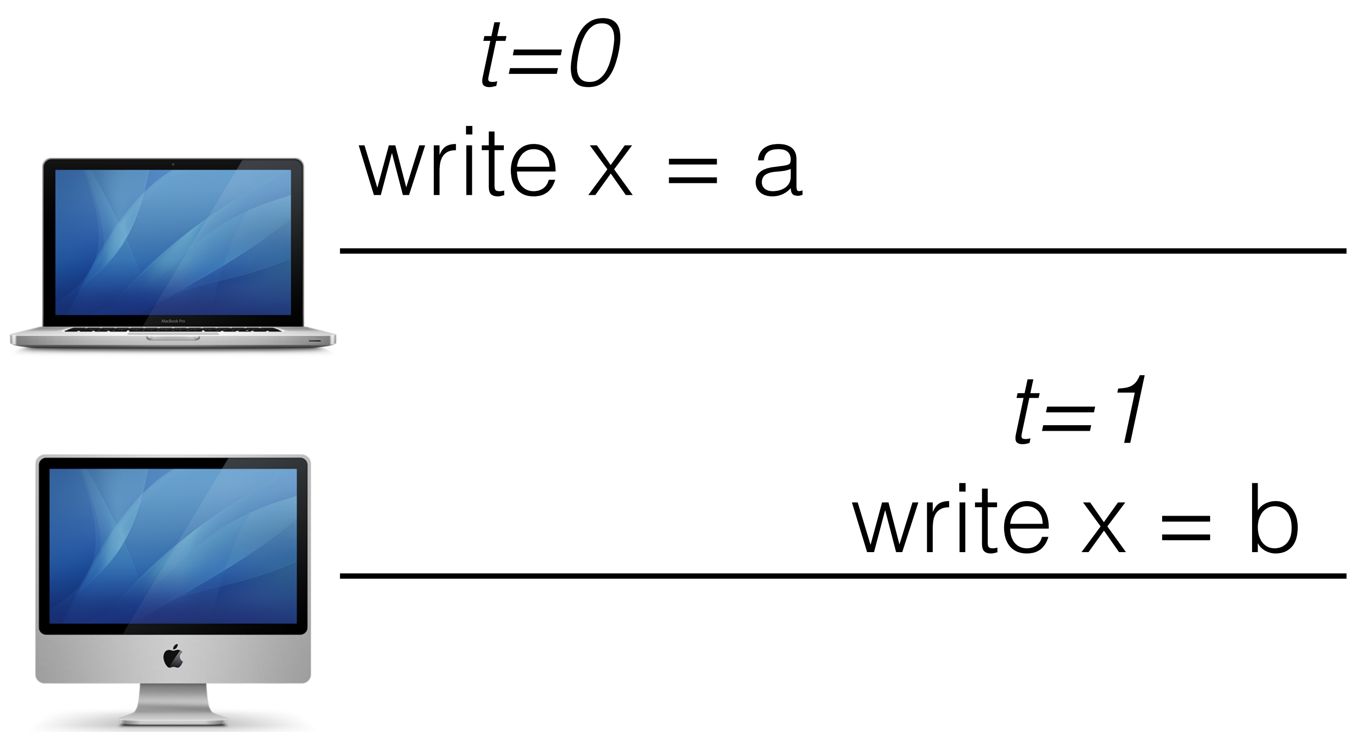
Goal: Everything eventually becomes synchronized.
No lost updates (don't replace new version with old)

Eventual Consistency: Distributed Filesystem

- Role of the sync server:
  - Resolve conflicting changes, report conflicts to user
  - Do not allow sync between clients
  - Detect if updates are sequential
  - Enforce ordering constraints

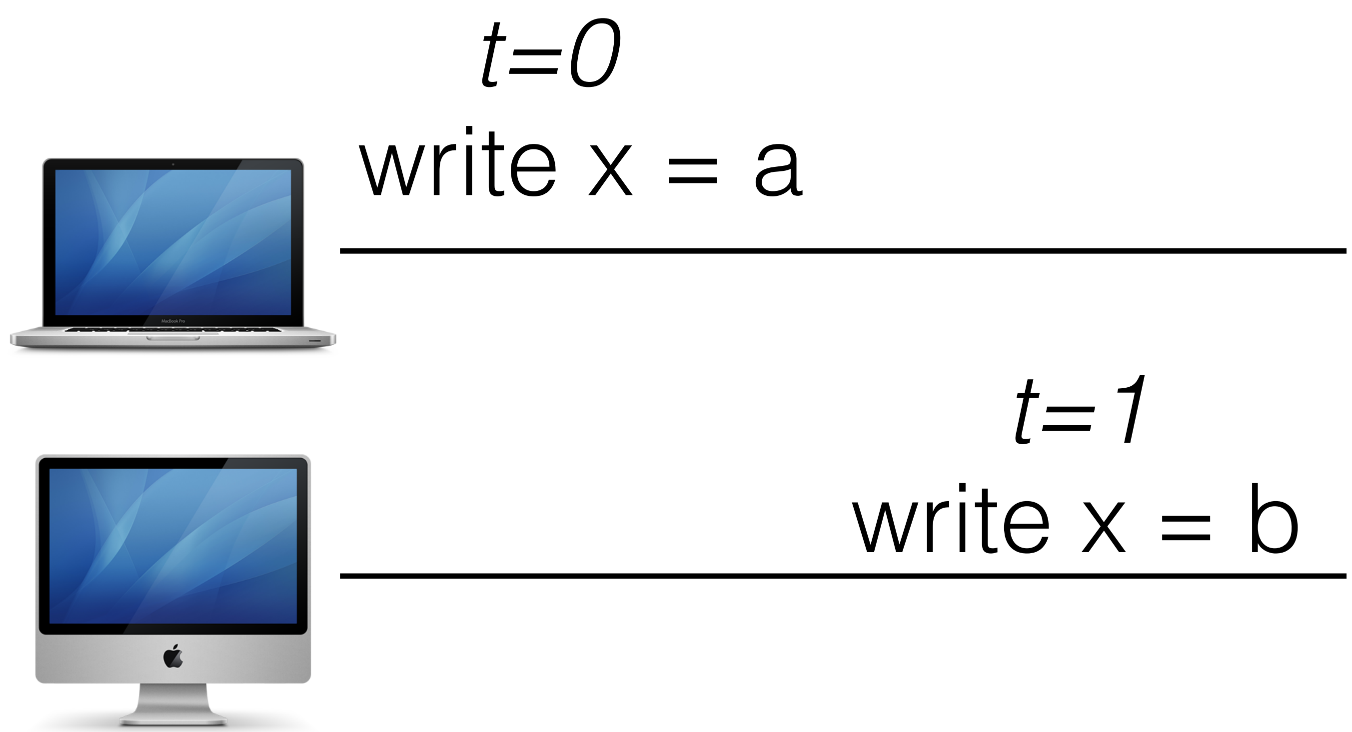
Detecting Conflicts

Do we just use timestamps?



Detecting Conflicts

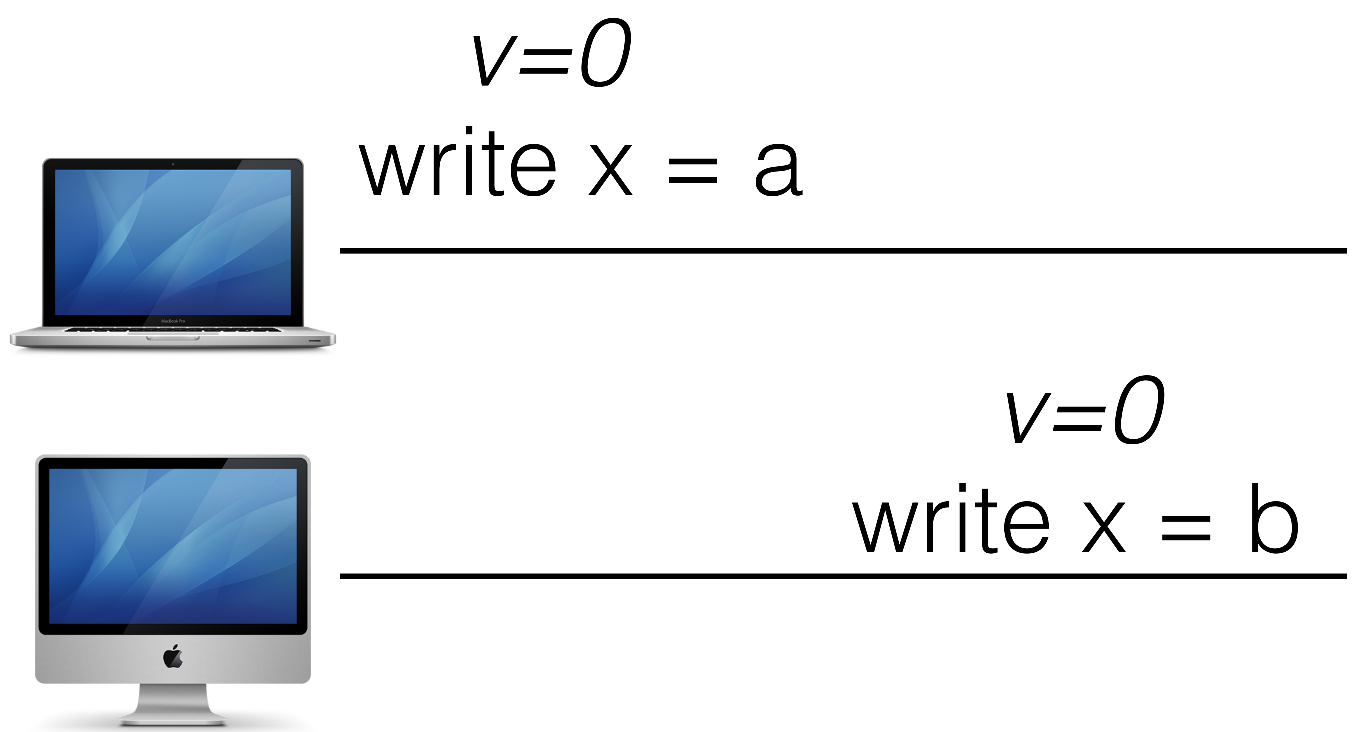
Do we just use timestamps?



NO, what if clocks are out of sync?
NO does not actually detect conflicts

Detecting Conflicts

Solution: Track version history on clients

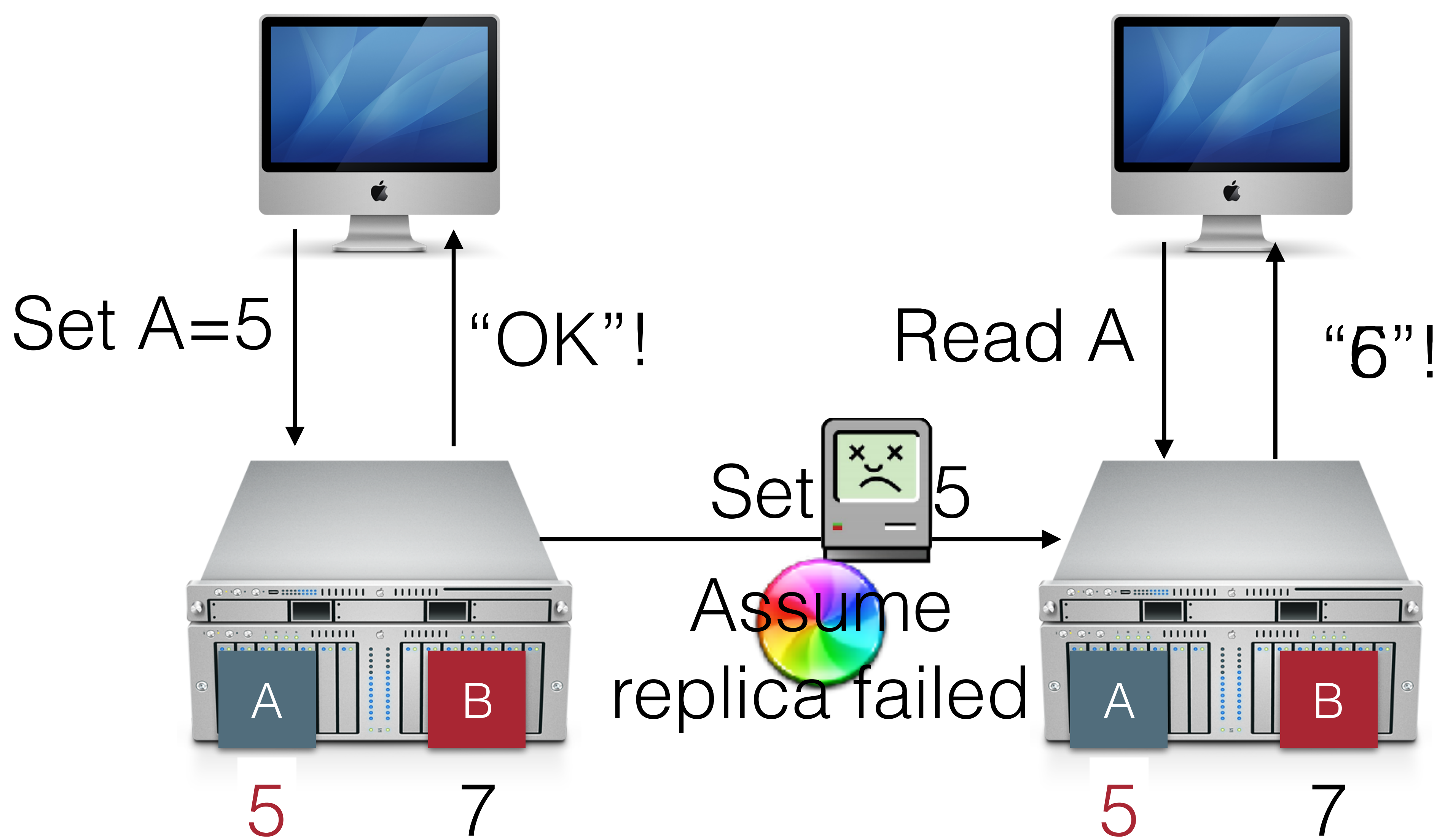


Still doesn't tell us what to do with a conflict

Client-Centric Consistency

- What can we guarantee in disconnected operation?
- Monotinic-reads: any future reads will return the same or newer value (never older)
- Monotonic-writes: A processes’ writes are always processed in order
- Read-you-writes
- Writes follow reads

Eventually Consistent + Available + Partition Tolerant



Choosing a consistency model

- Sequential consistency
  - All over - it's the most intuitive
- Causal consistency
  - Increasingly useful
- Eventual consistency
  - Very popular in industry and academia
  - File synchronizers, Amazon's Bayou and more

Example: Facebook

- Problem: >1 billion active users
- Solutions: Thousands of servers across the world
- What kind of consistency guarantees are reasonable? Need 100% availability!
- If I post a story on my news feed, is it OK if it doesn't immediately show up on yours?
  - Two users might not see the same data at the same time
  - Now this is “solved” anyway because there is no “sort by most recent first” option anyway

Example: Airline Reservations

- Reservations and flight inventory are managed by a GDS (Global Distribution System), who acts as a middle broker between airlines, ticket agencies and consumers [Except for Southwest and Air New Zealand and other oddballs]
- GDS needs to sell as many seats as possible within given constraints
- If I have 100 seats for sale on a flight, does it matter if reservations for flights are reconciled immediately?
- If I have 5 seats for sale on a flight, does it matter if reservations are reconciled immediately?

Example: Airline Reservations

- Result: Reservations can be made using either a strong consistency model or a weak, eventual one
- Most reservations are made under the normal strong model (reservation is confirmed immediately)
- GDS also supports "Long Sell" - issue a reservation without confirmed availability, need to eventually reconcile it
- Long sells require the seller to make clear to the customer that even though there's a confirmation number it's not confirmed!

Filesystem consistency

- What consistency guarantees do a filesystem provide?
- read, write, sync, close
- On sync, guarantee writes are persisted to disk
- Readers see most recent
- What does a network file system do?

Network Filesystem Consistency

- How do you maintain these same semantics?
- (Cheat answer): Very, very expensive
  - EVERY write needs to propagate out
  - EVERY read needs to make sure it sees the most recent write
  - Oof. Just like Ivy.

Consistency Takeaways

- Strong consistency (sequential or strict) comes at a tradeoff: performance, availability
- Weaker consistency also has a tradeoff (weaker consistency)
- But: applications can make these design choices clear to end-users
    - Facebook
    - Dropbox