Consistency in Distributed Systems

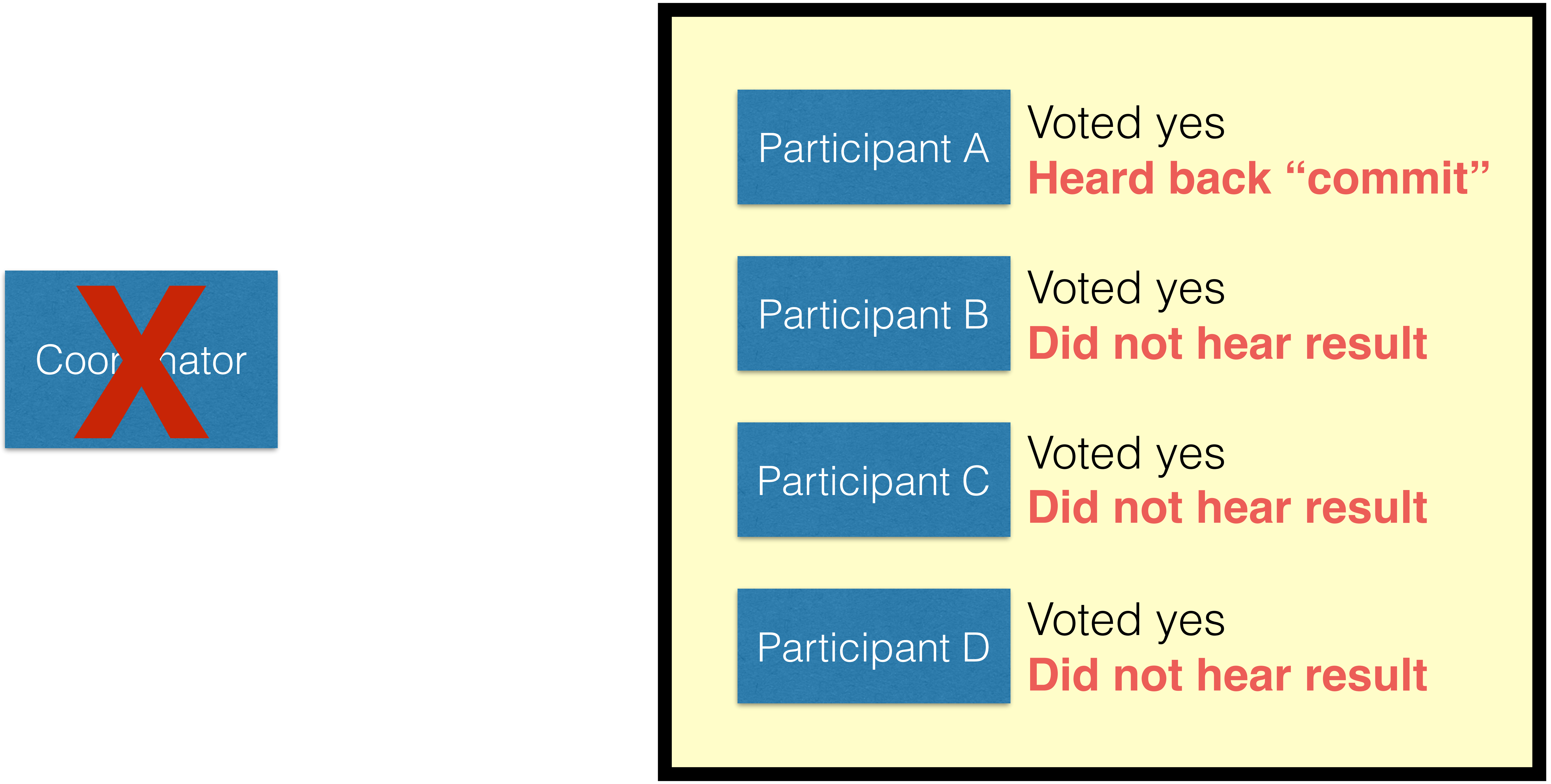
CS 475, Fall 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

Review: Transactions 2PC, 3PC

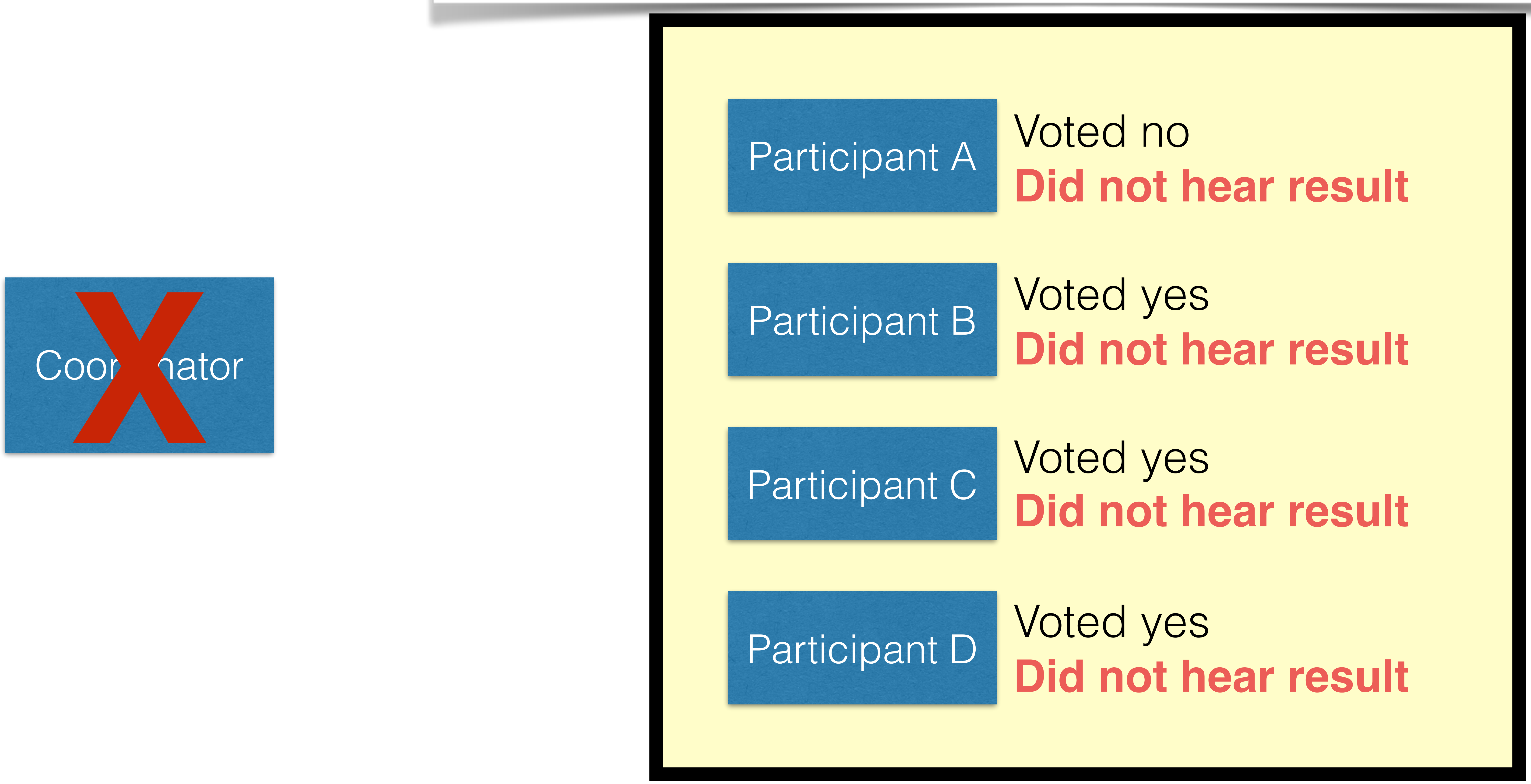
Digging Deeper into 2PC Failures

If they can talk to each other, we know we can commit (good)



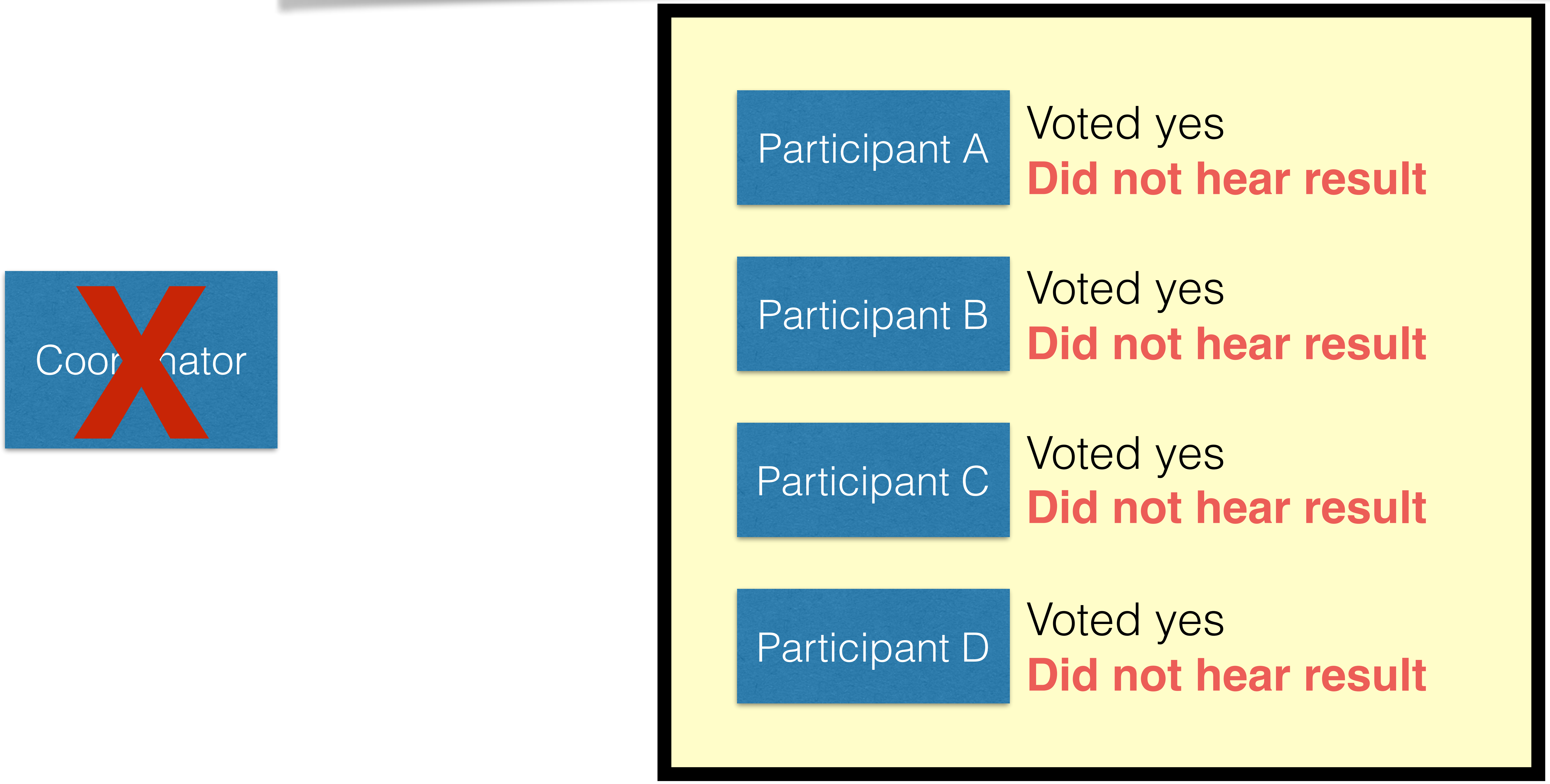
Digging Deeper into 2PC Failures

If they can talk to each other, we know that we can all abort (good)



Digging Deeper into 2PC Failures

If they can talk to each other, we do not know if we can commit/abort (who knows what the coordinator will do?)



3 Phase Commit

- Goal: Avoid blocking on node failure
- How?
  - Think about how 2PC is better than 1PC
    - 1PC means you can never change your mind or have a failure after committing
    - 2PC **still** means that you can't have a failure after committing (committing is irreversible)

Safety in Crashes

Timeout behavior: abort!

Coordinator

Soliciting Votes
Commit Authorized

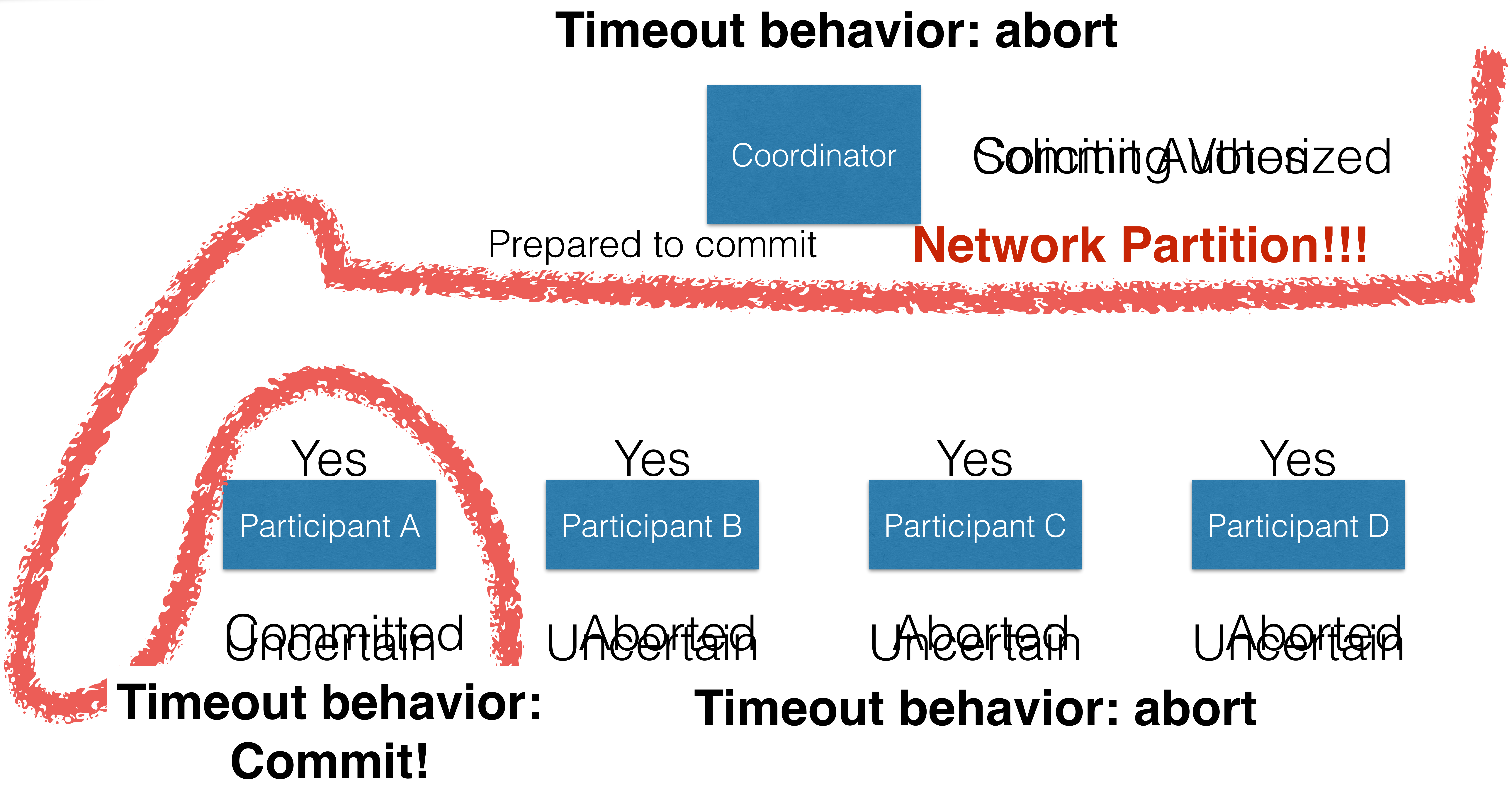
Prepared to commit

Participant A	Participant B	Participant C	Participant D
Yes	Yes	Yes	Yes
Uncertain	Uncertain	Uncertain	Uncertain
Committed	Aborted	Aborted	Aborted

Crashed: do not commit or abort. When recovers, asks coordinator what to do

Partitions

Implication: if networks can delay arbitrarily, 3PC does not guarantee safety!!!!



FLP - Intuition

- Why can’t we make a protocol for consensus/agreement that can tolerate both partitions and node failures?
- To tolerate a partition, you need to assume that **eventually** the partition will heal, and the network will deliver the delayed packages
- But the messages might be delayed **forever**
- Hence, your protocol would not come to a result, until **forever** (it would not have the **liveness** property)

Partitions

Insight: There is a "majority" partition here (B,C,D)
The "minority" know that they are not in the majority (A can only talk to Coordinator, knows B, C, D might exist)

Coordinator

Soliciting Votes
Authorized

Prepared to commit

Network Partition!!!

Yes	Yes	Yes	Yes
Participant A	Participant B	Participant C	Participant D
Committed Uncertain	Aborted Uncertain	Aborted Uncertain	Aborted Uncertain

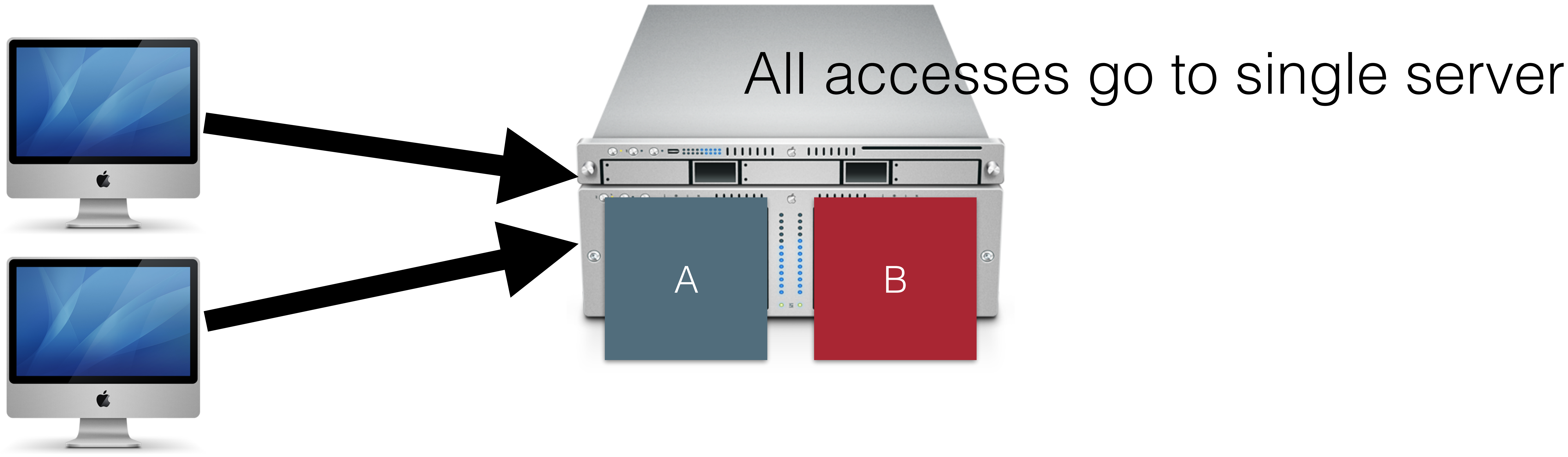
Timeout behavior: Commit!

Timeout behavior: abort

Today

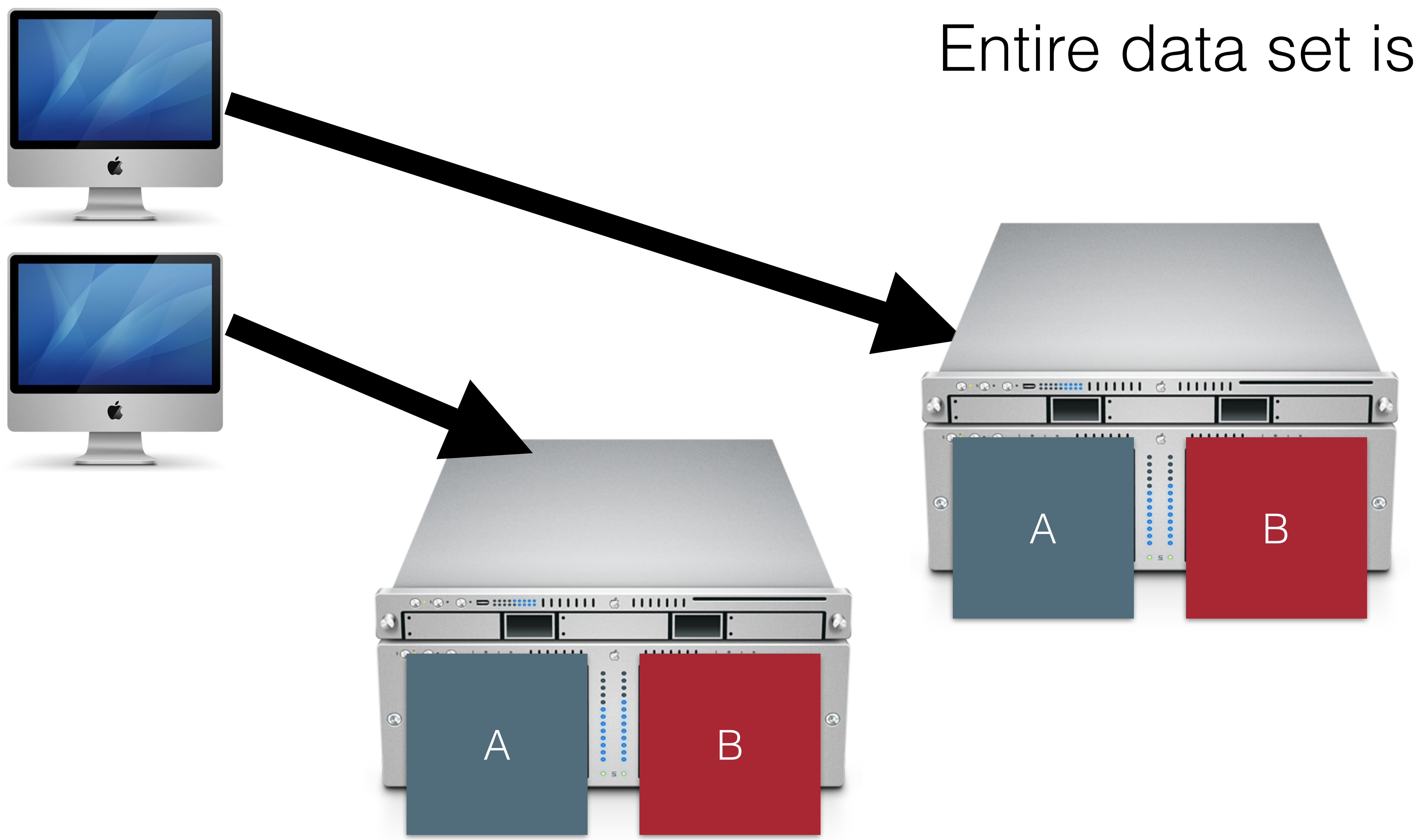
- Consistency in distributed systems
- Ivy - a consistent replicated datastore
- Reminders:
  - HW3 due Weds

Recurring Solution in Distributed Systems: Replication



Recurring Solution in Distributed Systems: Replication

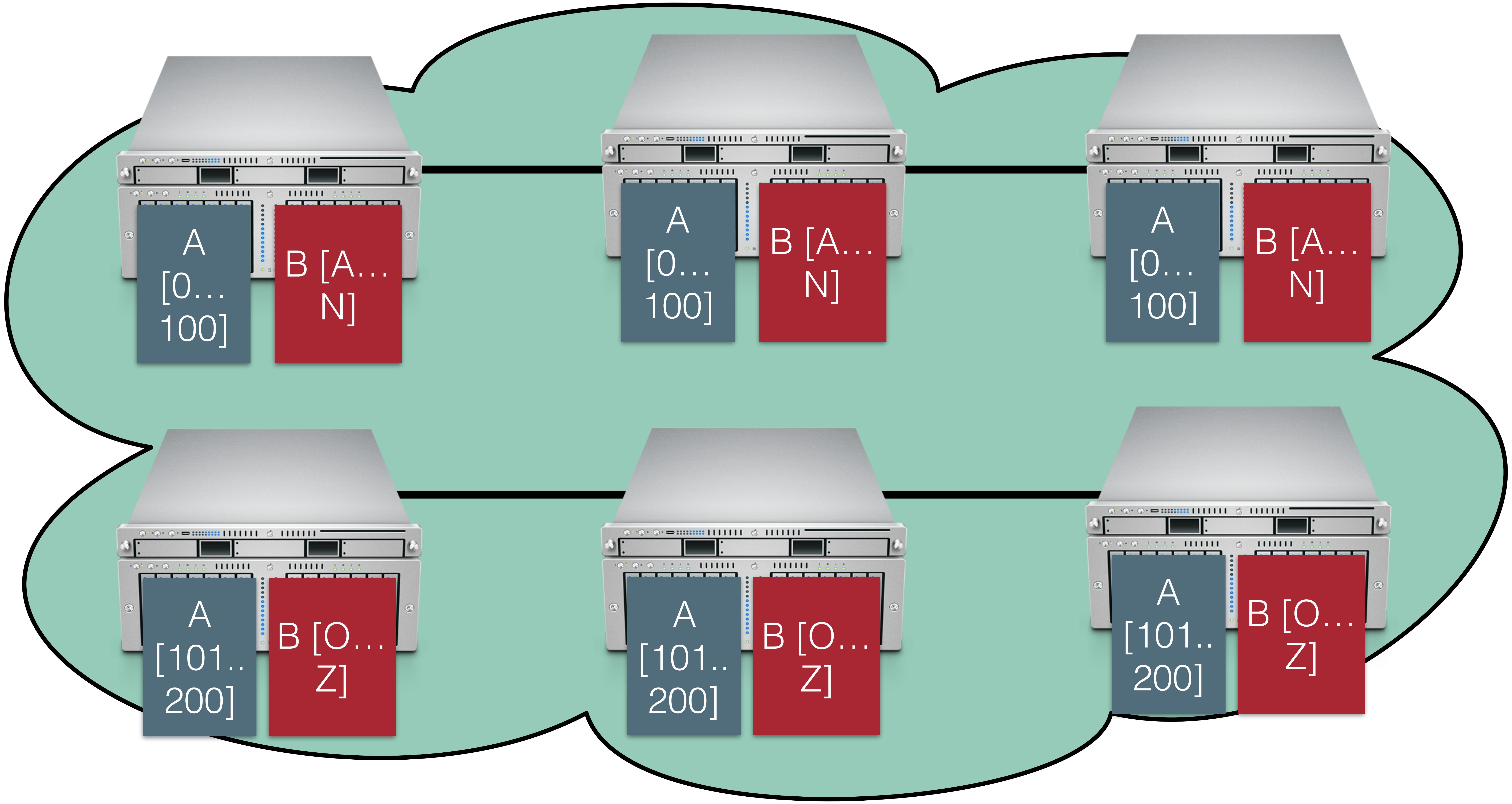
Entire data set is copied



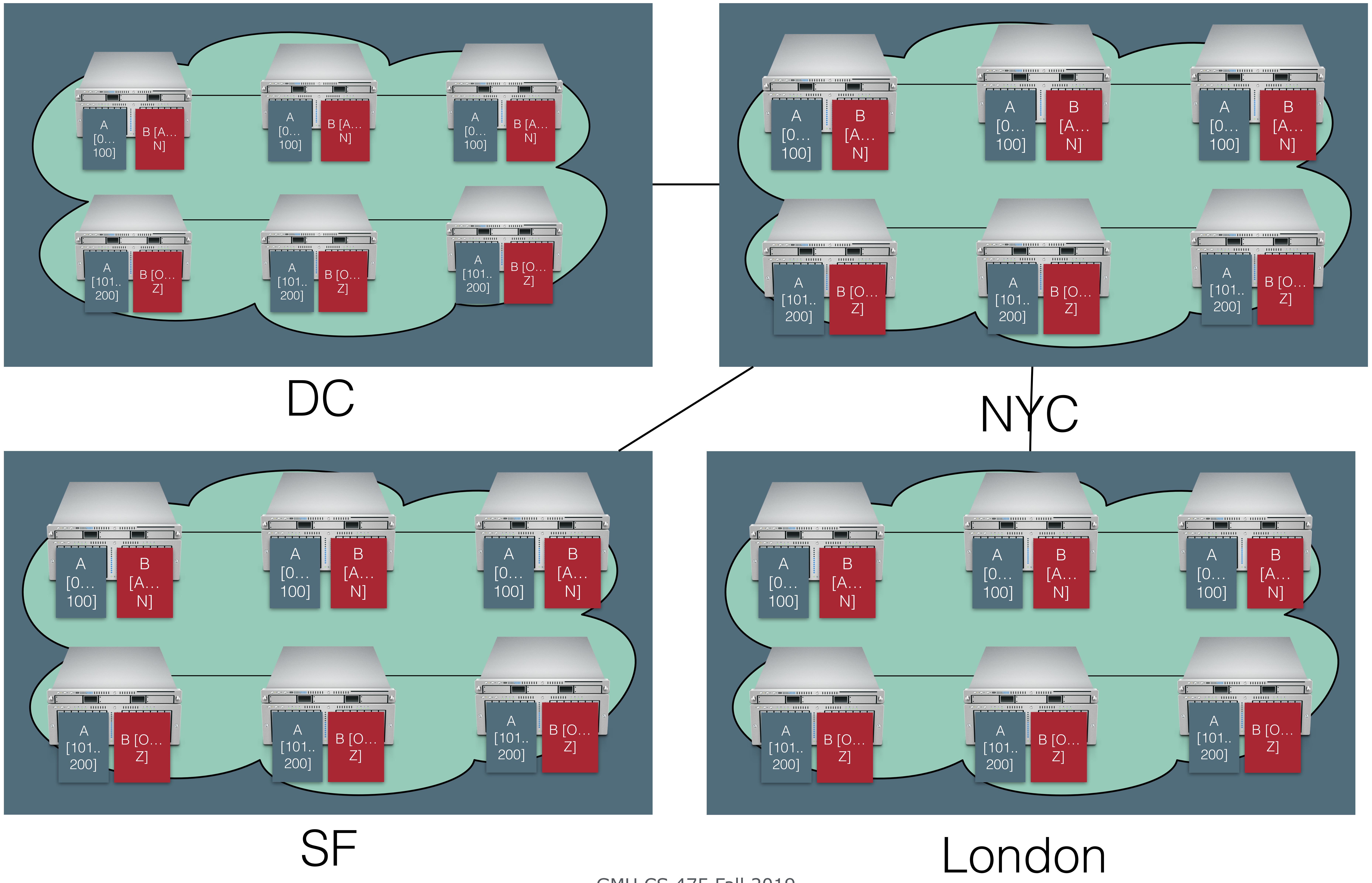
Recurring Solution in Distributed Systems: Replication

- Improves performance:
  - Client load can be evenly shared between servers
  - Reduces latency: can place copies of data nearer to clients
- Improves availability:
  - One replica fails, still can serve all requests from other replicas

Partitioning + Replication

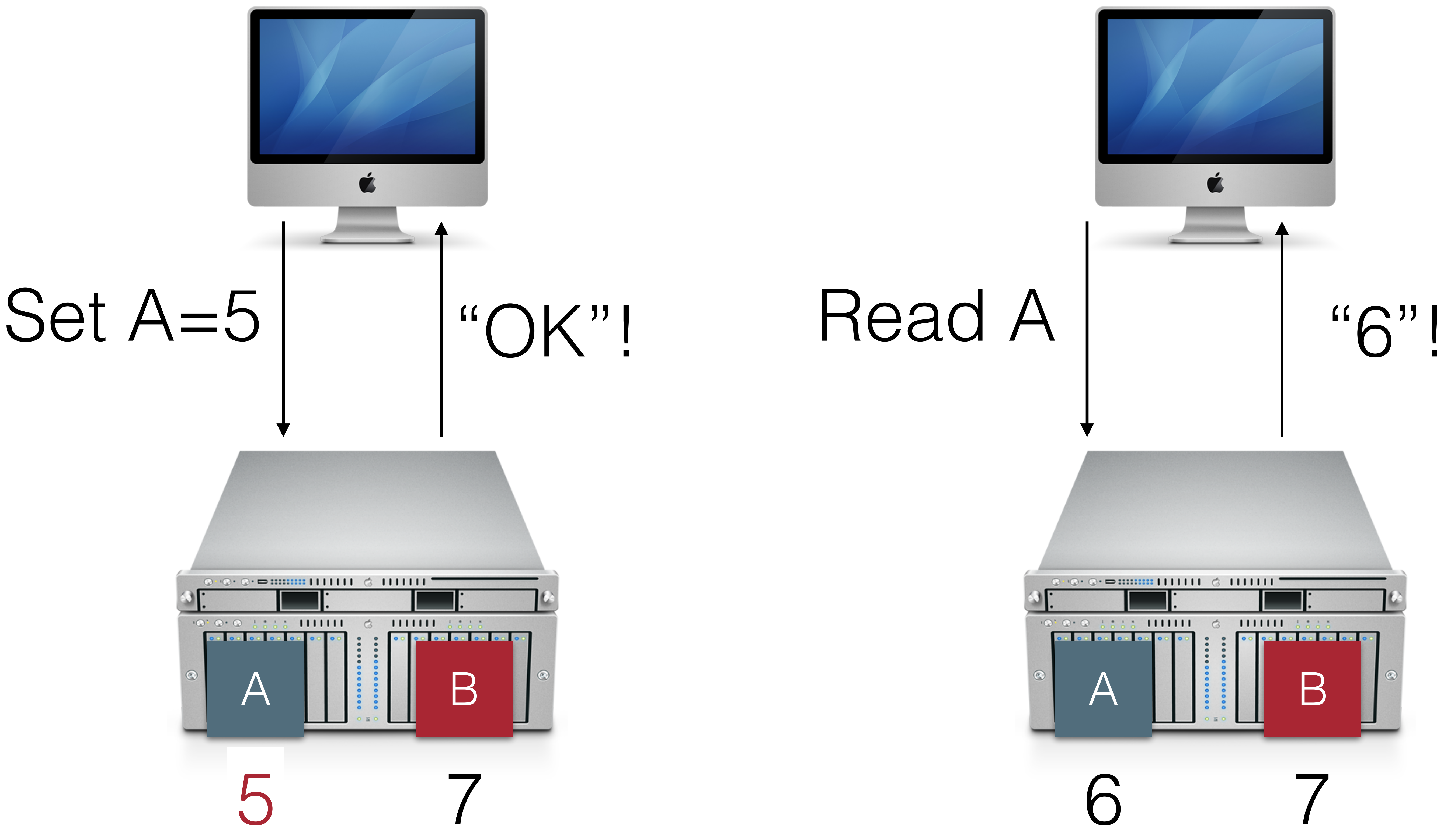


Partitioning + Replication



Recurring Problem: Replication

- Replication solves some problems, but creates a huge new one: consistency

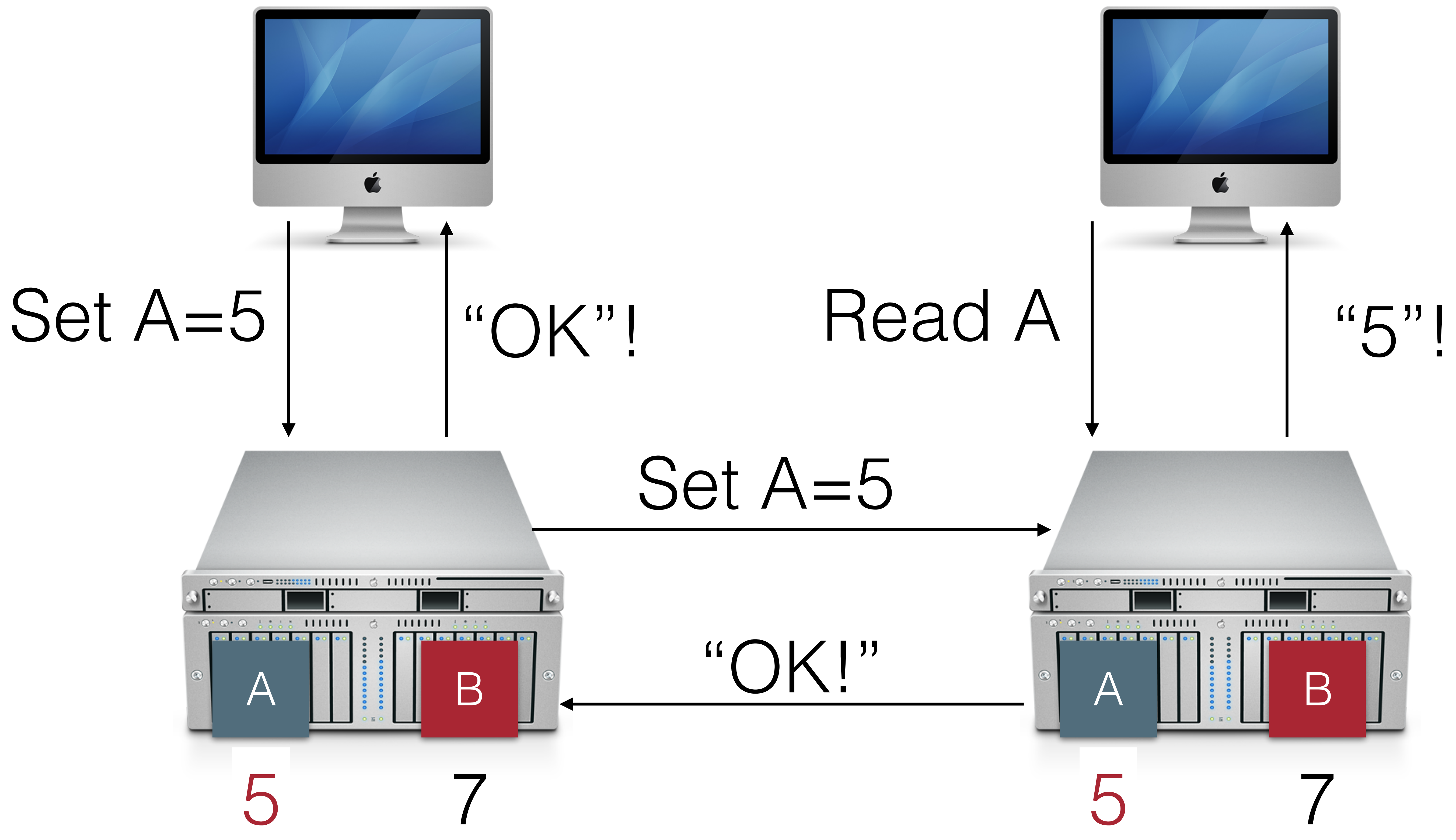


OK, we obviously need to actually do something here to replicate the data… but what?

Consistency

- The problem of consistency arrises whenever some data is replicated
- That data exists in (at least) two places at the same time
- What is a "valid" state?

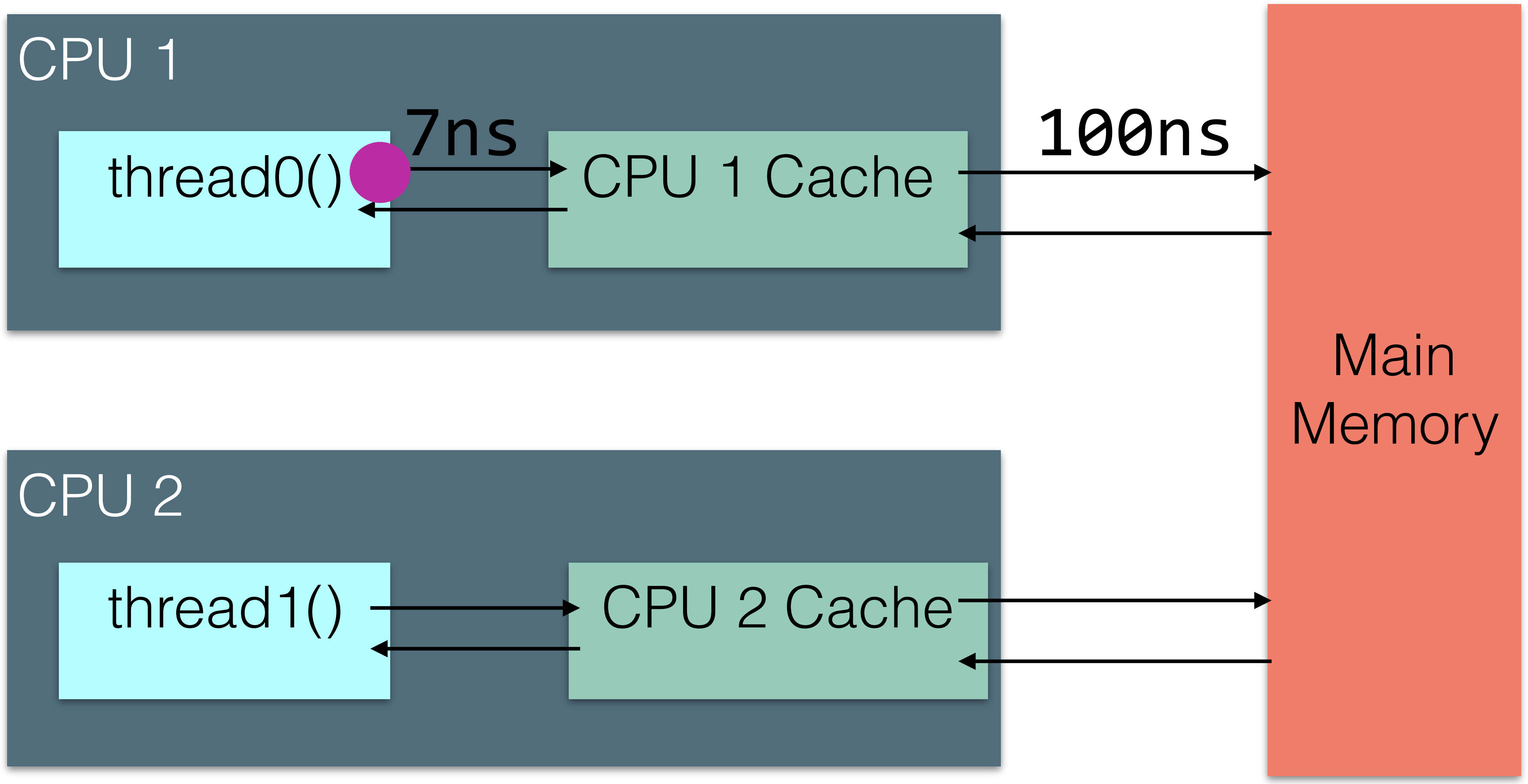
Consistency



Consistency

- Why do we think the prior slide was consistent?
  - Whenever we read, we see the most recent writes
- Even programs running **on a single computer** have to obey some consistency model
  - We talked about: linearizability, sequential consistency
  - Remember that consistency comes at a price

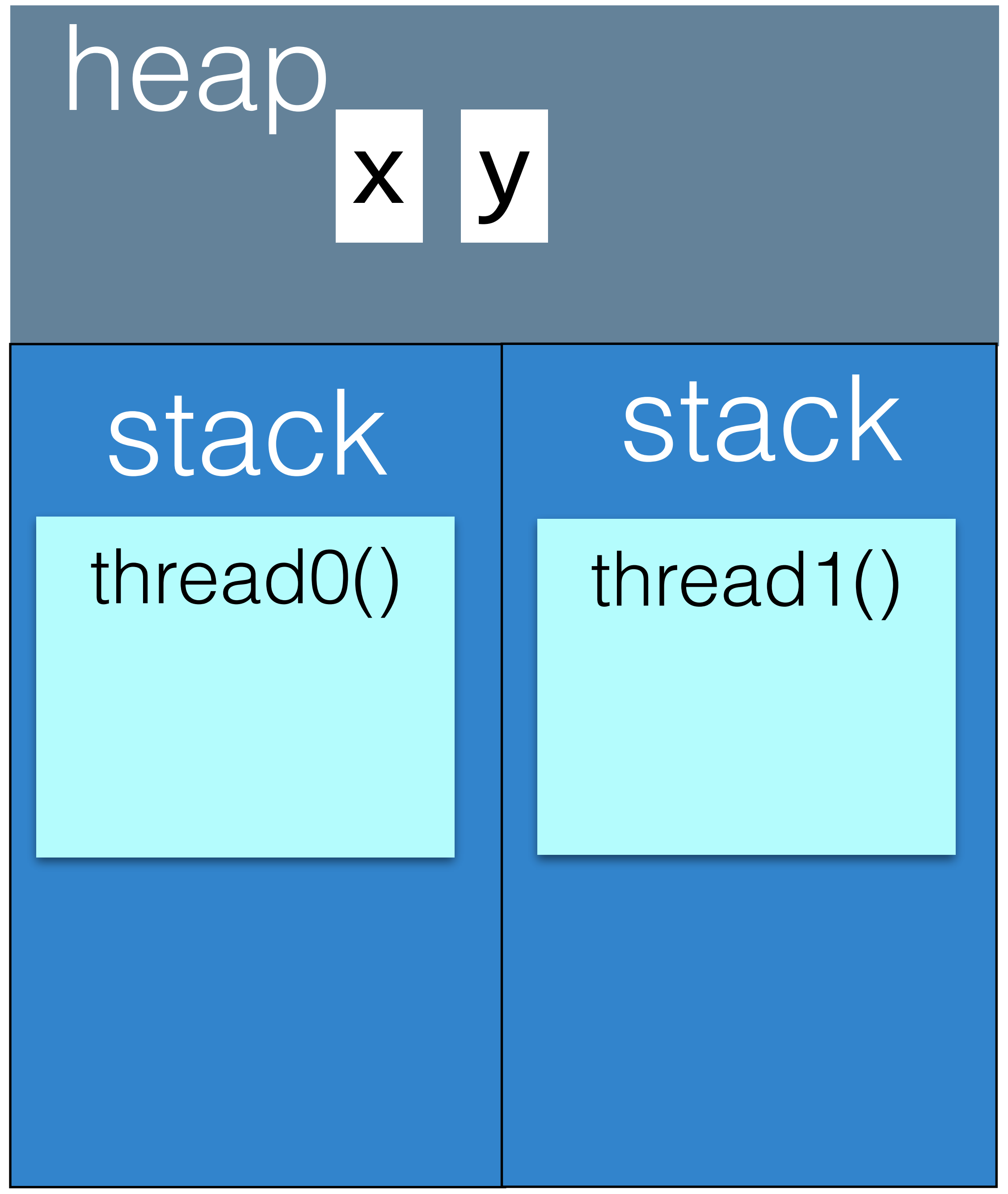
Java Memory Model



Quiz: What’s the output?

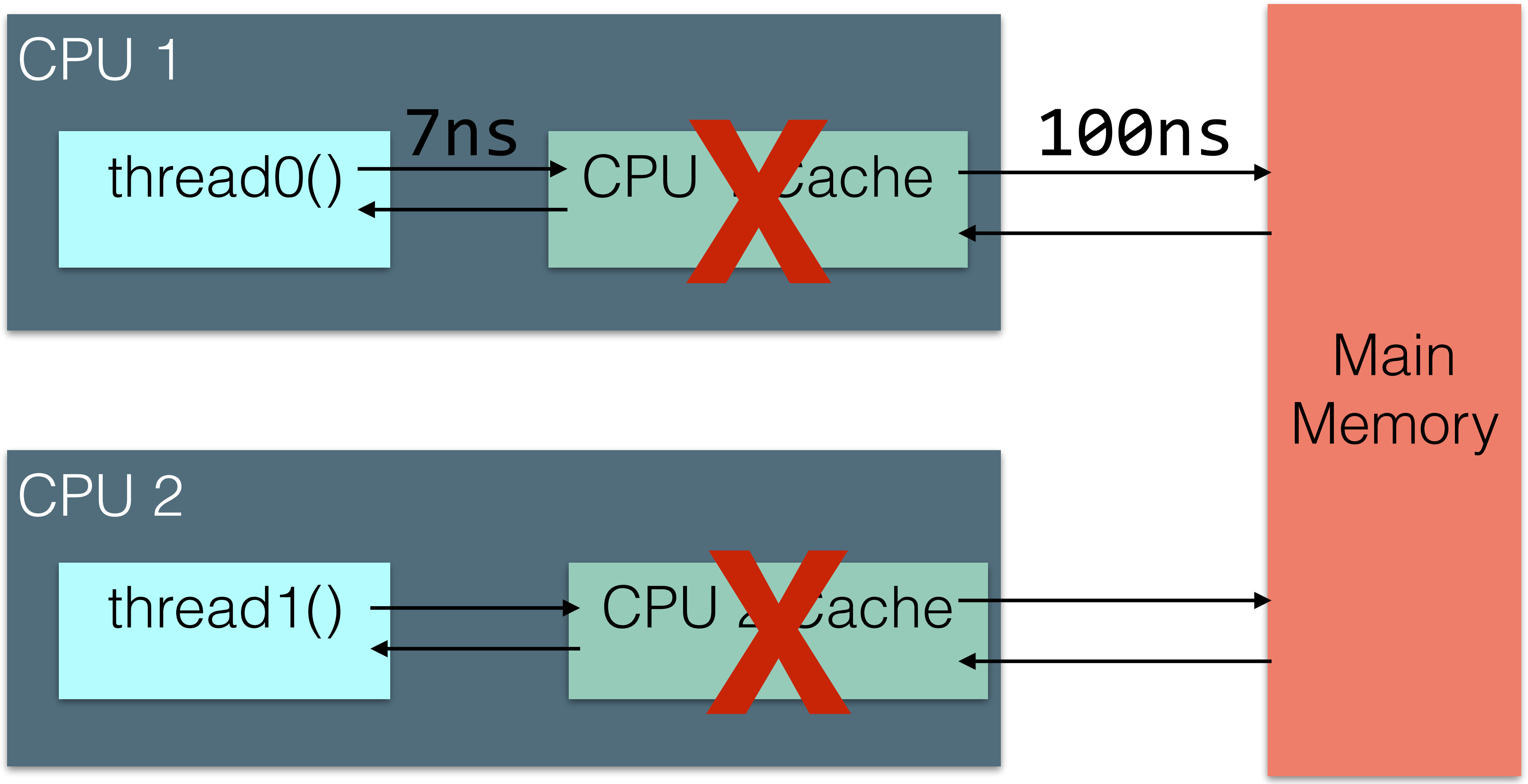
```
class MyObj {
 volatile int x = 0;
 volatile int y = 0;

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



`Volatile` keyword: no per-thread caching of variables

Volatile Keyword



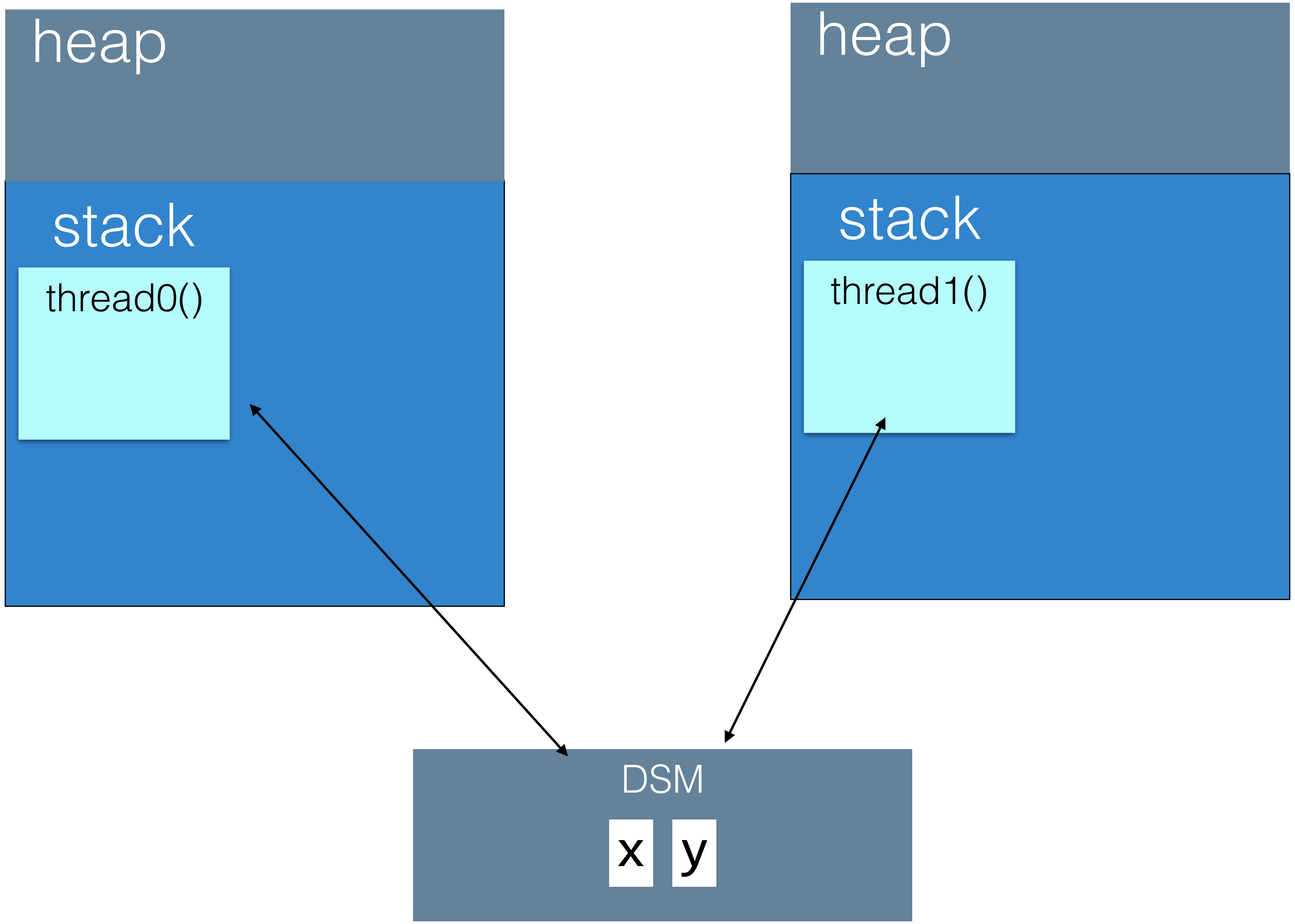
Consistency

- This is a consistency model!
  - Constraints on the system state that are observable by applications
- “When I write y=1, any future reads must say y=1”
  - … except in Java, if it’s a non-volatile variable
- Clearly, this often comes at a cost (see simple example with `volatile`…)

Sequential Consistency

- Strict consistency is often not practical
  - Requires globally synchronizing clocks
- Sequential consistency gets close, in an easier way:
  - 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)

Distributed Shared Memory



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

Is this correct?

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

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

```
ne2 {
 = 0;
 DSMInt y = 1

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

Naïve DSM

- Gets even more funny when we add a third host
  - Many more interleaving possible
- Definitely not sequentially consistent
- Who is at fault?
  - The DSM system?
  - The app?
  - **The developers of the app, if they thought it would be sequentially consistent.**

Sequentially Consistent DSM

- How do we get this system to behave similar to Java's volatile keyword?
- We want to ensure:
  - Each machine's own operations appear in order
  - All machines see results according to some total order (each read sees the most recent writes)
- We can say that some observed runtime ordering of operations can be "explained" by a sequential ordering of operations that follow the above rules

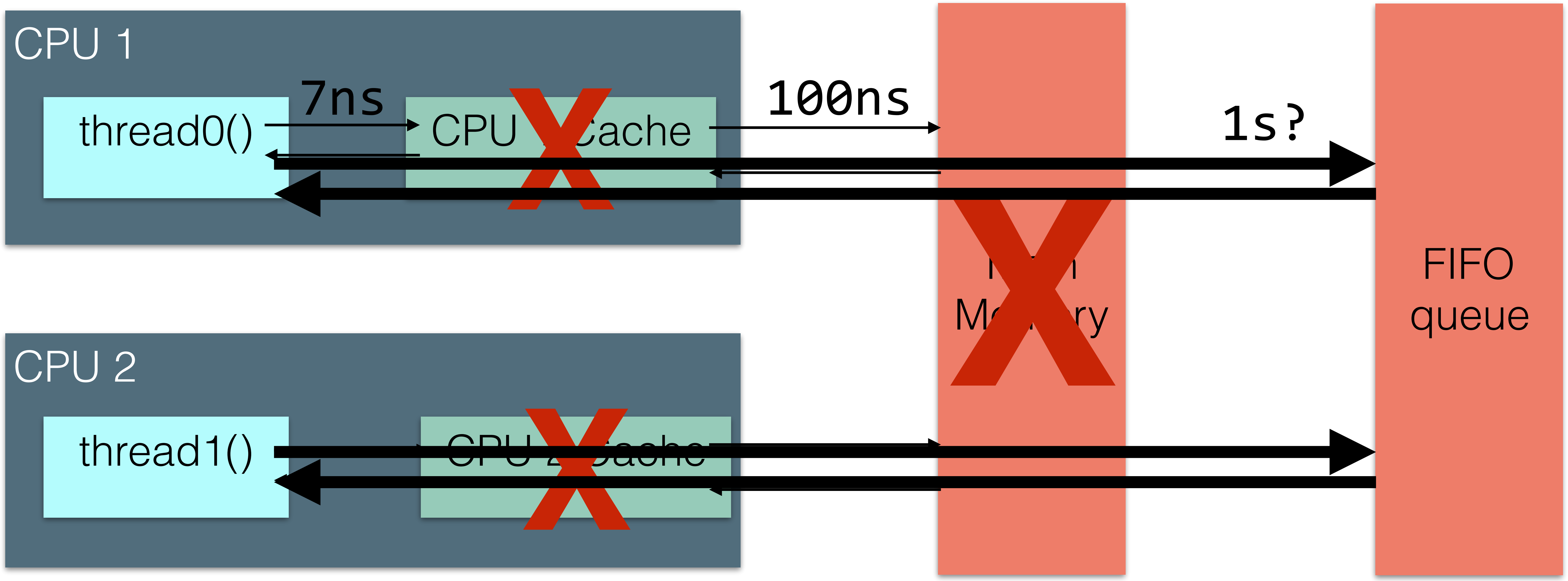
Sequentially Consistent DSM

- Each node must see the most recent writes before it reads that same data
- Performance is not great:
  - Might make writes expensive: need to wait to broadcast and ensure other nodes heard your new value
  - Might make reads expensive: need to wait to make sure that there are no pending writes that you haven't heard about yet

Sequentially Consistent DSM

- Each processor issues requests in the order specified by the program
  - Can’t issue the next request until previous is finished
- Requests to an individual memory location are served from a single FIFO queue
  - Writes occur in single order
  - Once a read observes the effect of a write, it’s ordered behind that write

Sequentially Consistent DSM

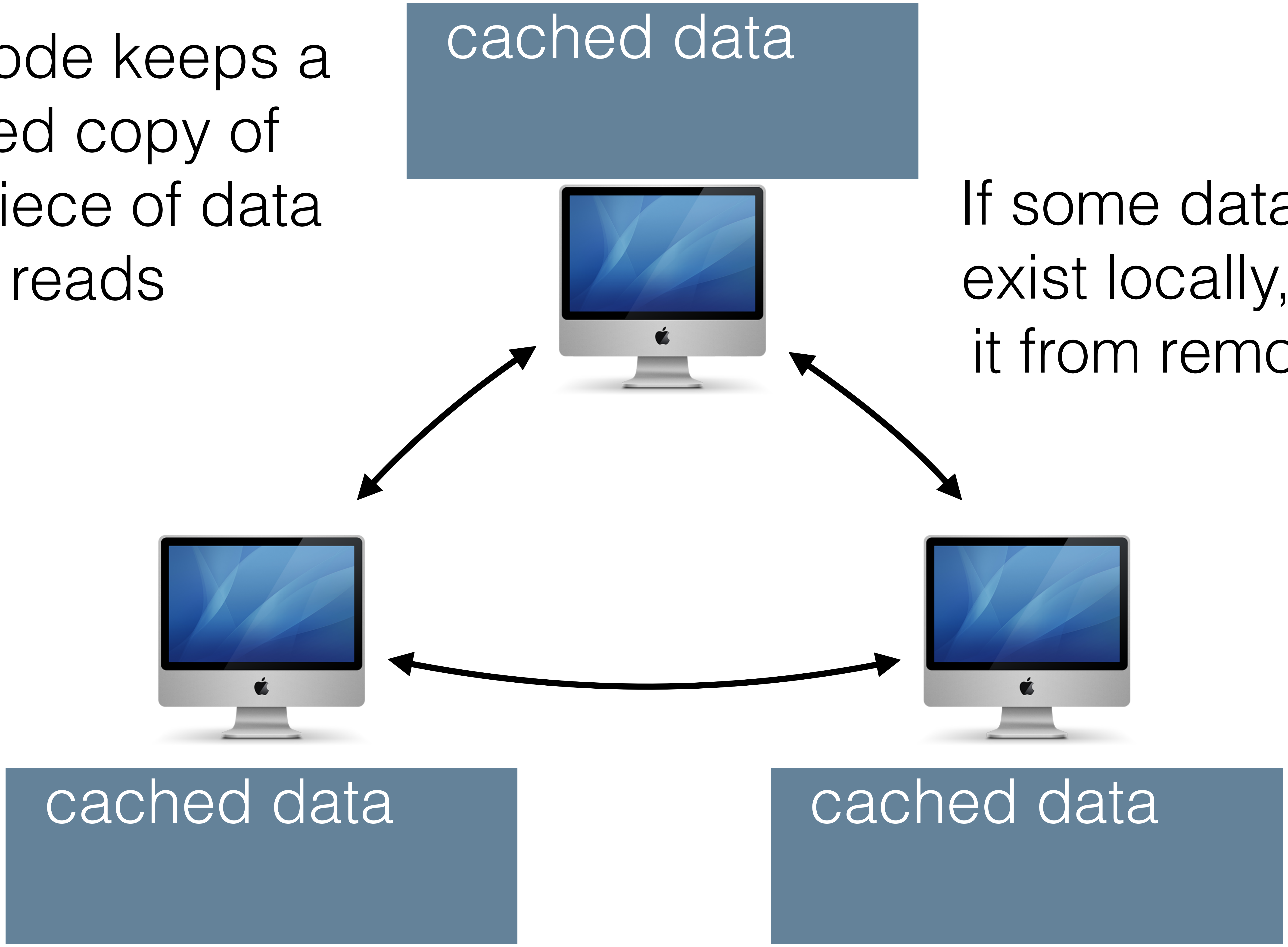


Ivy DSM

- **I**ntegrated shared **V**irtual memory at **Y**ale
- Provides shared memory across a group of workstations
- Might be easier to program with shared memory than with message passing
  - Makes things look a lot more like one huge computer with hundreds of CPUs instead of hundreds of computers with one CPU

Ivy Architecture

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

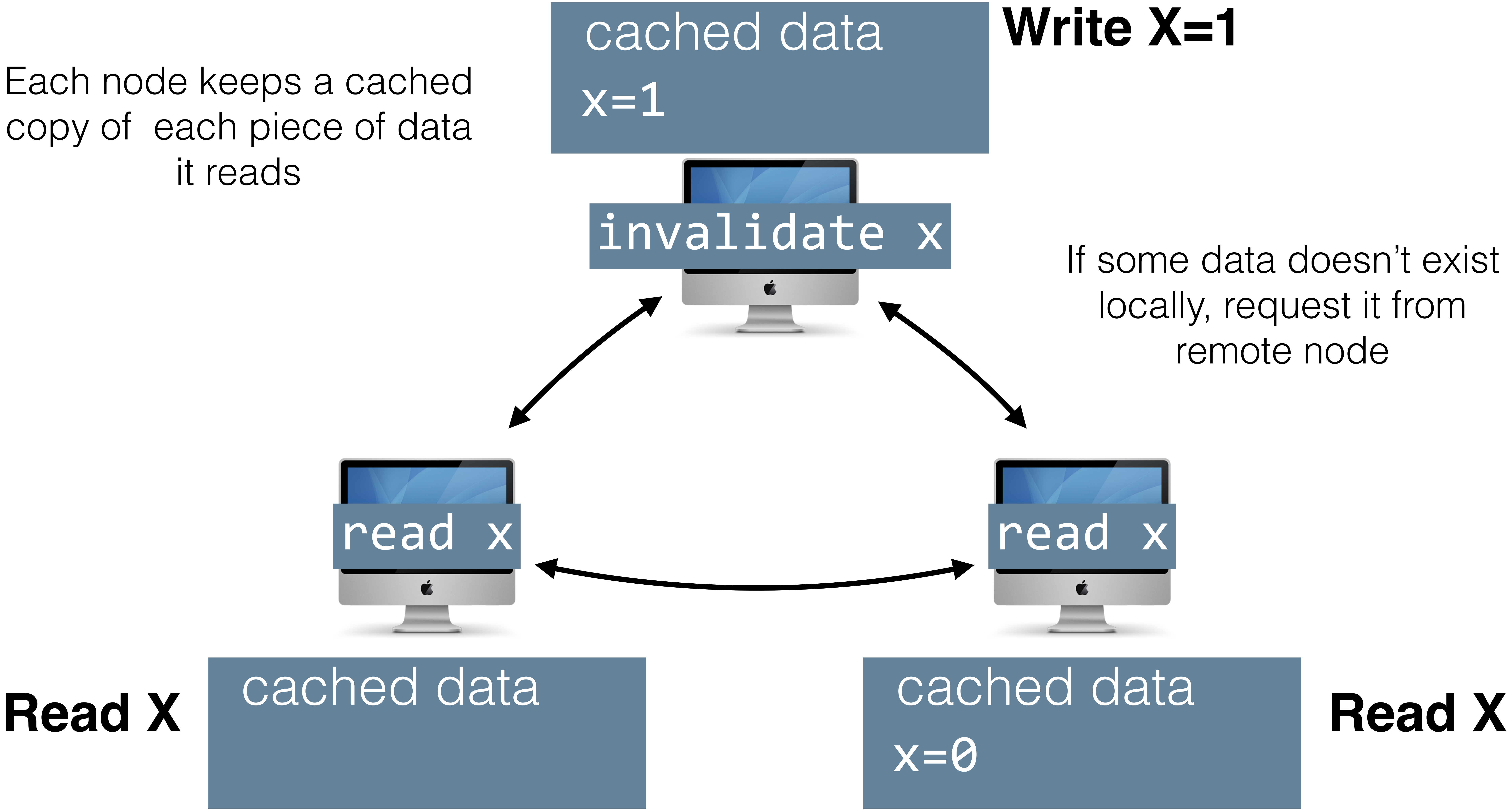


If some data doesn't exist locally, request it from remote node

Ivy provides sequential consistency

- Support multiple readers, single writer semantics
- Write invalidate update protocol
- If I write some data, I must tell everyone who has cached it to get rid of their cache

Ivy Architecture



Ivy Implementation

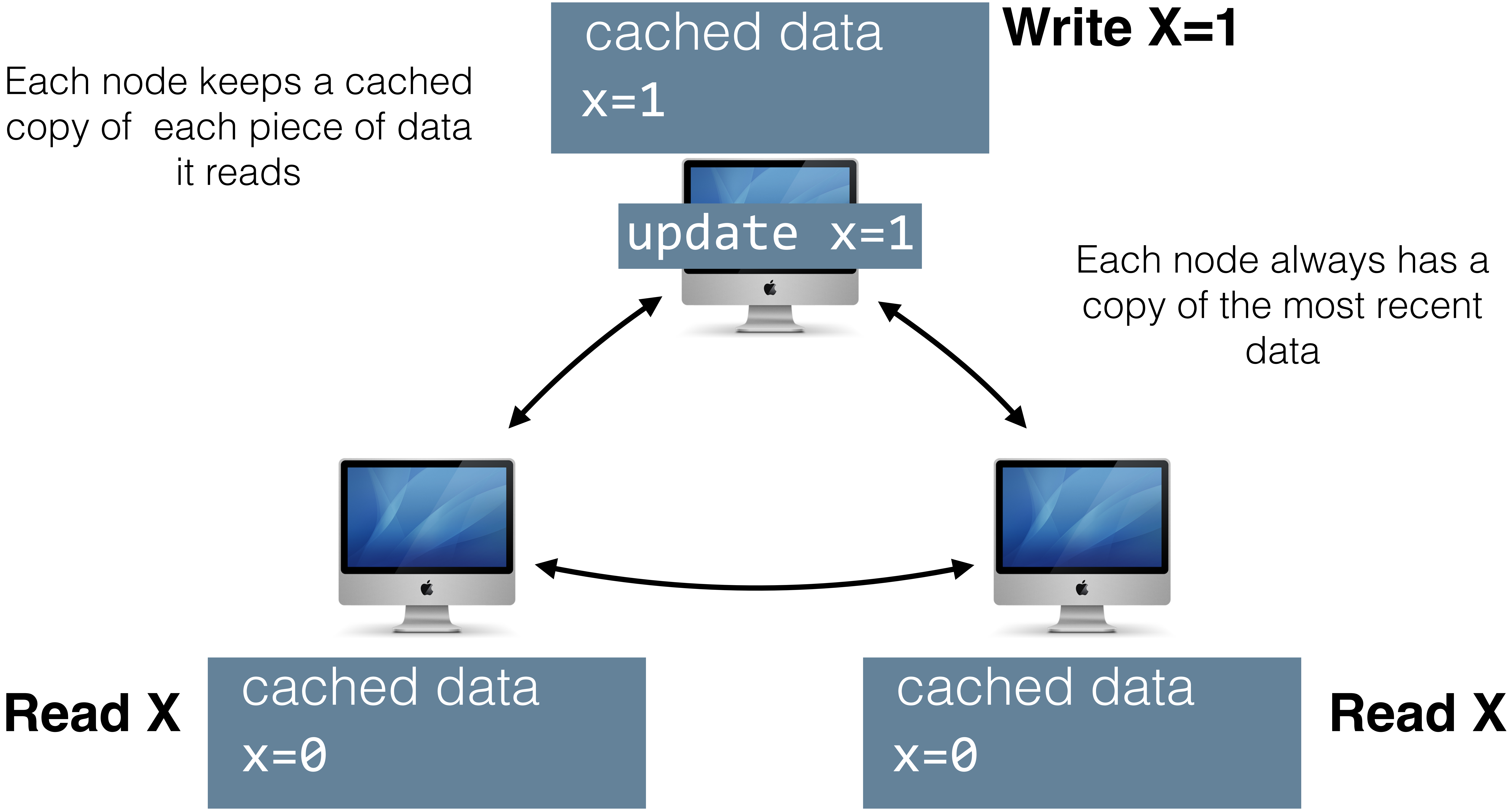
- Ownership of data moves to be whoever last wrote it
- There are still some tricky bits:
  - How do we know who owns some data?
  - How do we ensure only one owner per data?
  - How do we ensure all cached data are invalidated on writes?
- Solution: Central manager node

Ivy invariants

- Every piece of data has exactly one current owner
- Current owner is guaranteed to have a copy of that data
- If the owner has write permission, no other copies can exist
- If owner has read permission, it’s guaranteed to be identical to other copies
- Manager node knows about all of the copies
- Sounds a lot like HW4? :)

HW4 Architecture

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

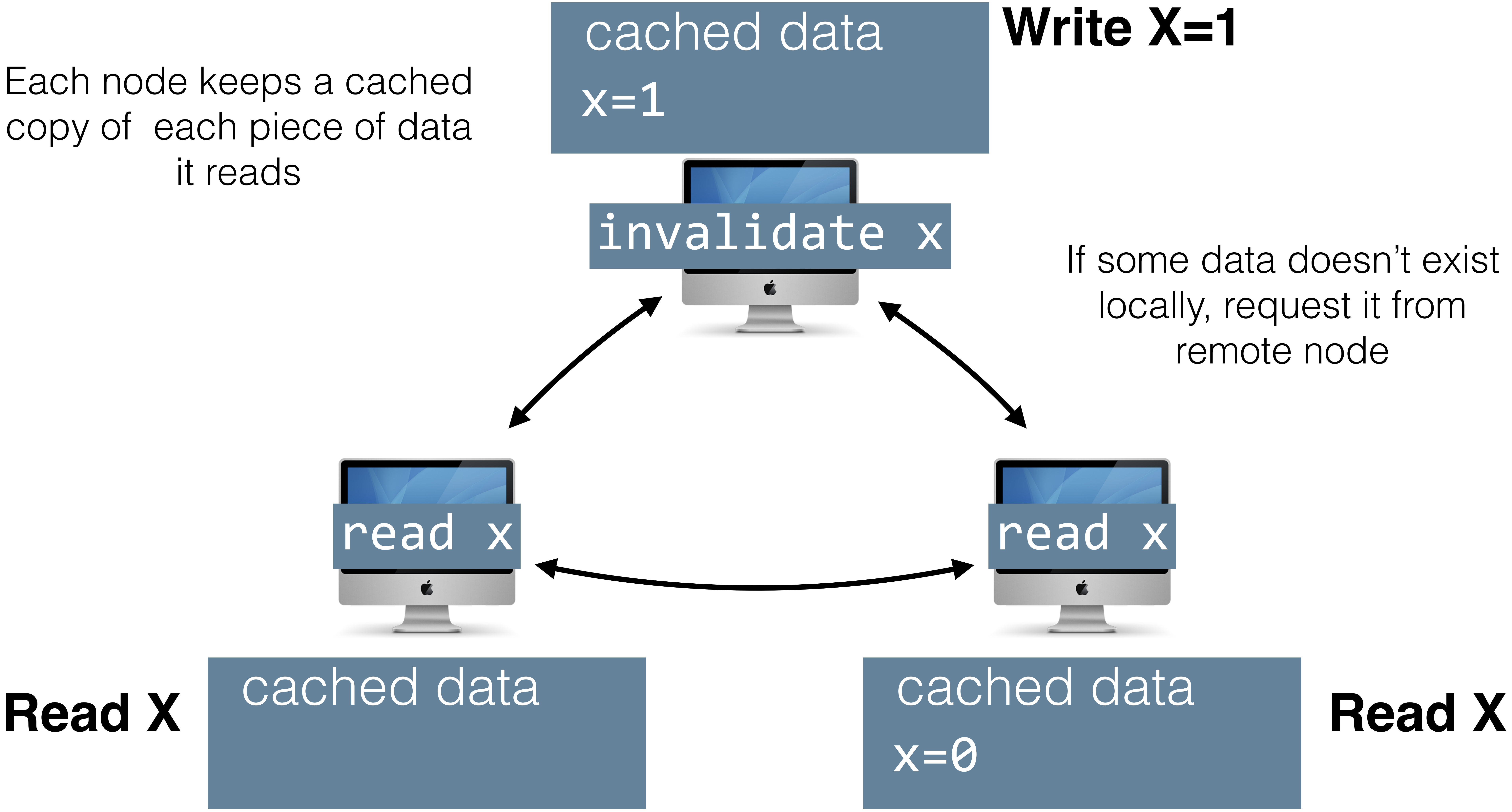


Each node always has a copy of the most recent data

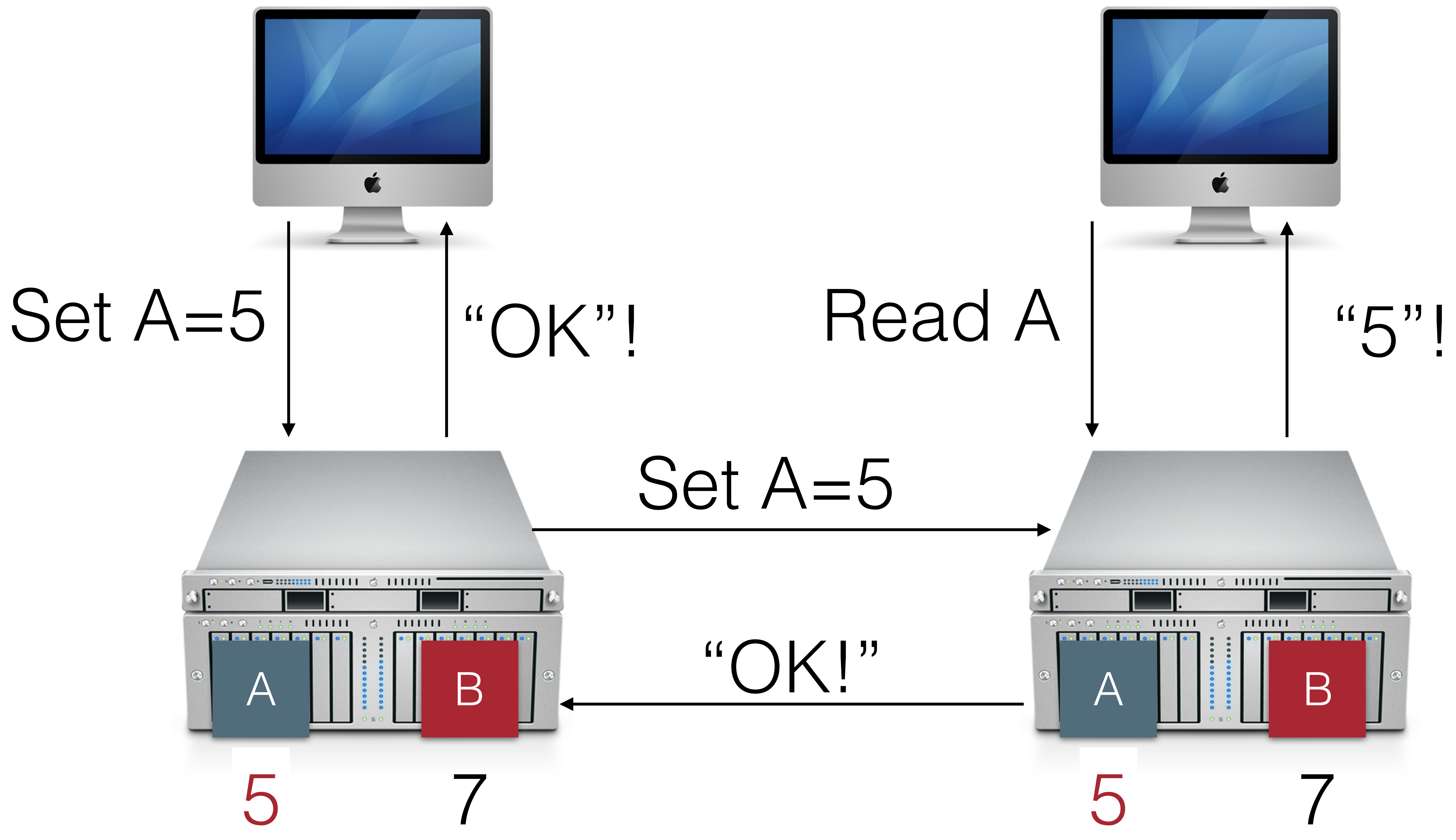
Ivy Architecture

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

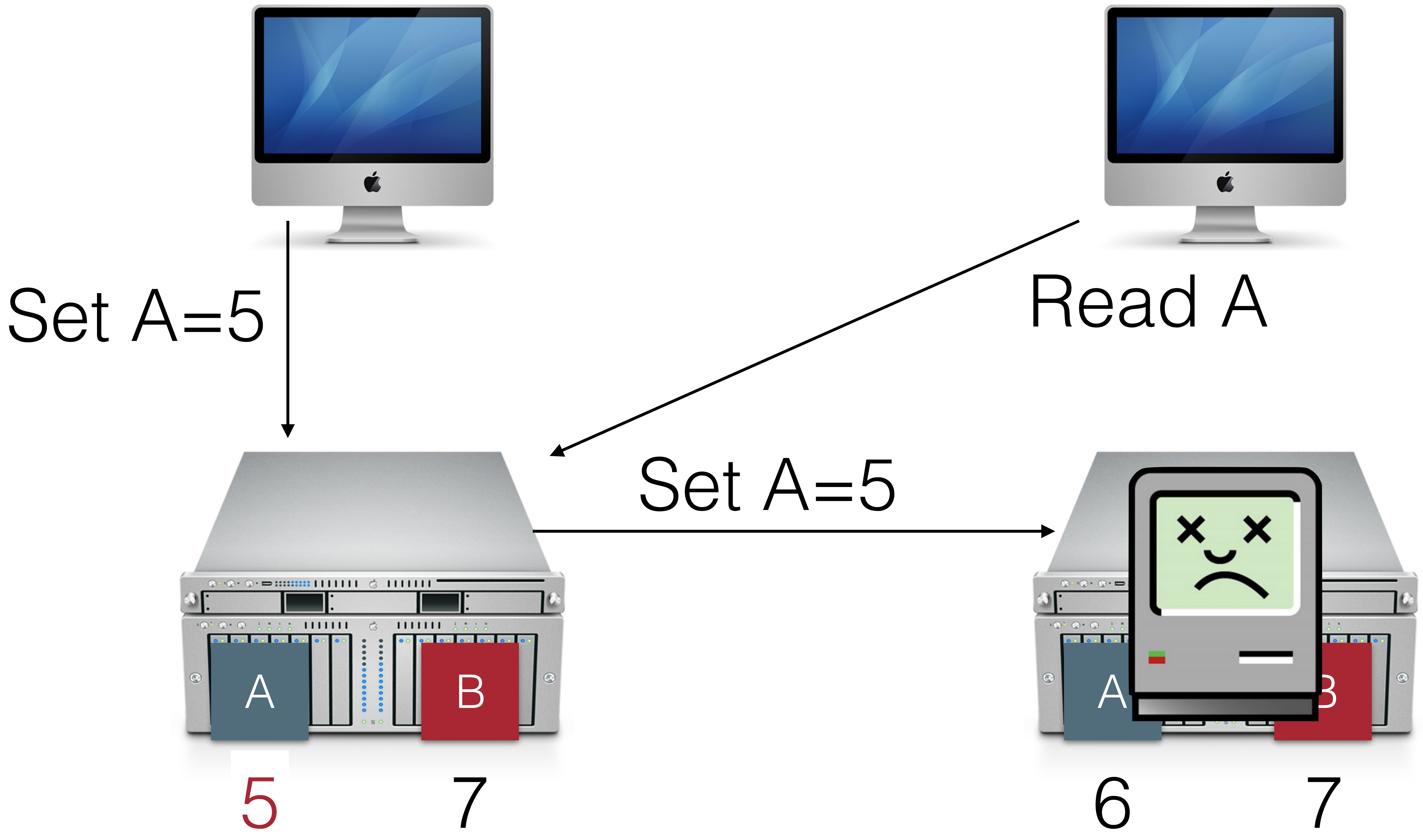


Sequential Consistency

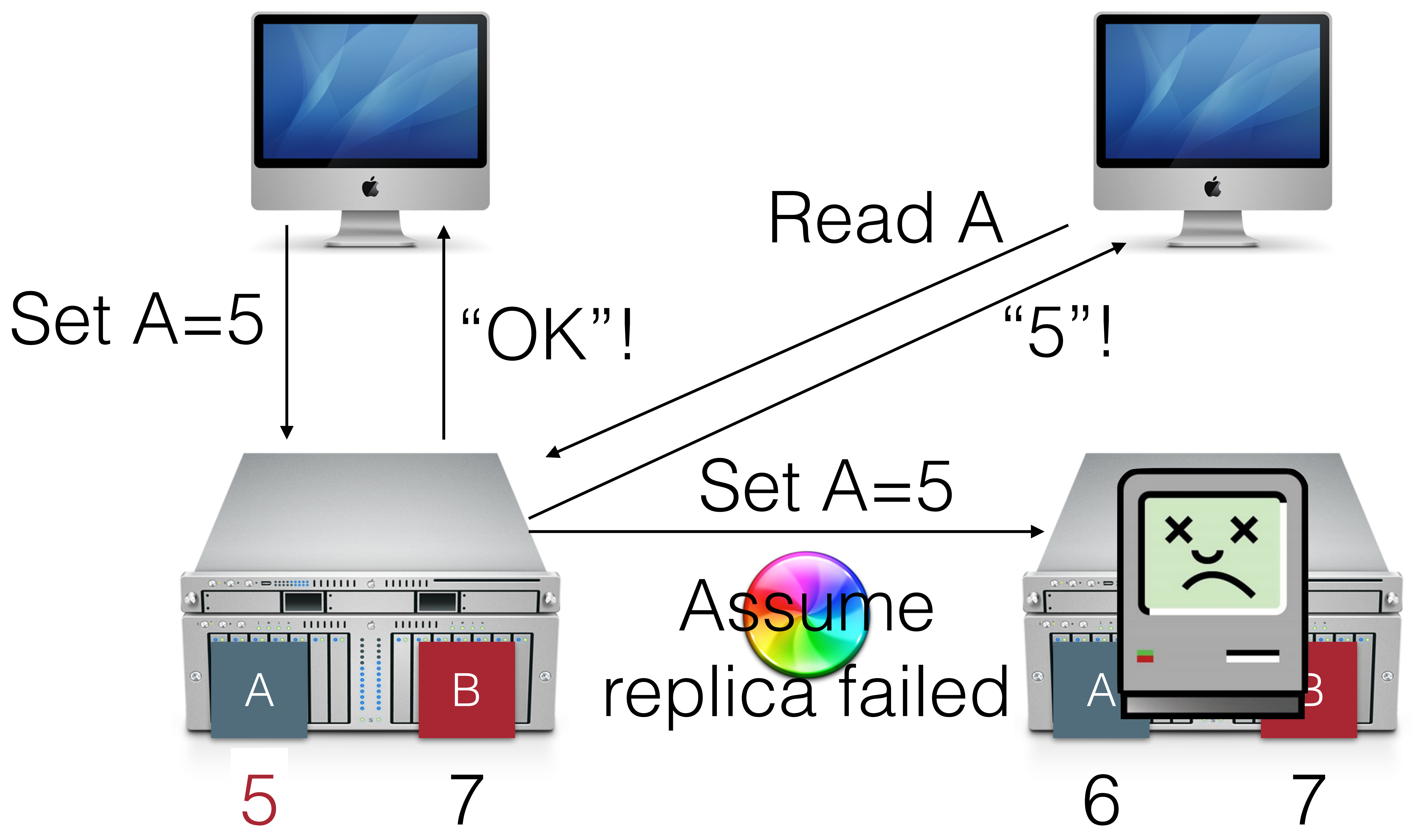


Availability

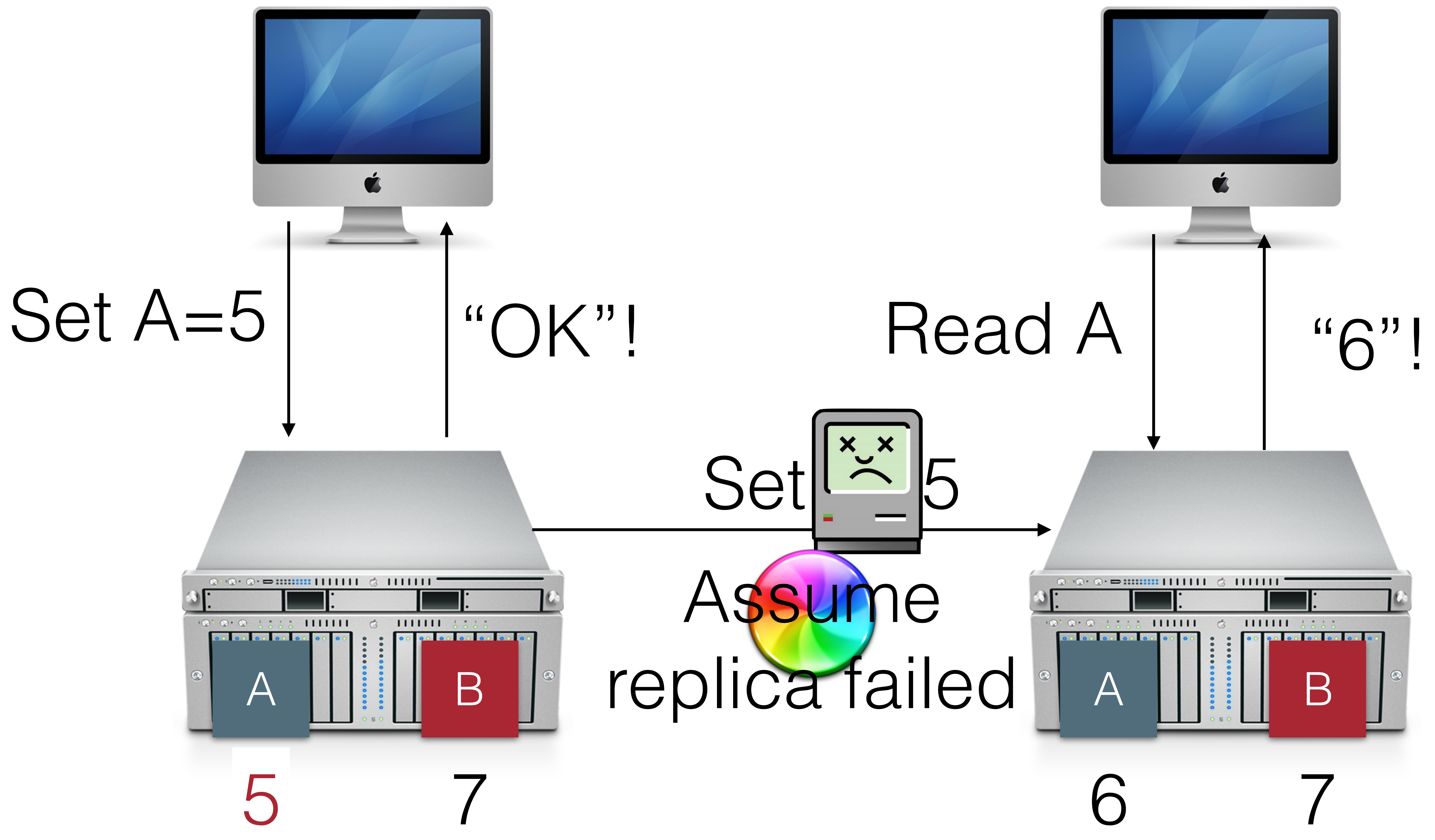
- Our protocol for sequential consistency does NOT guarantee that the system will be available!



Consistent + Available

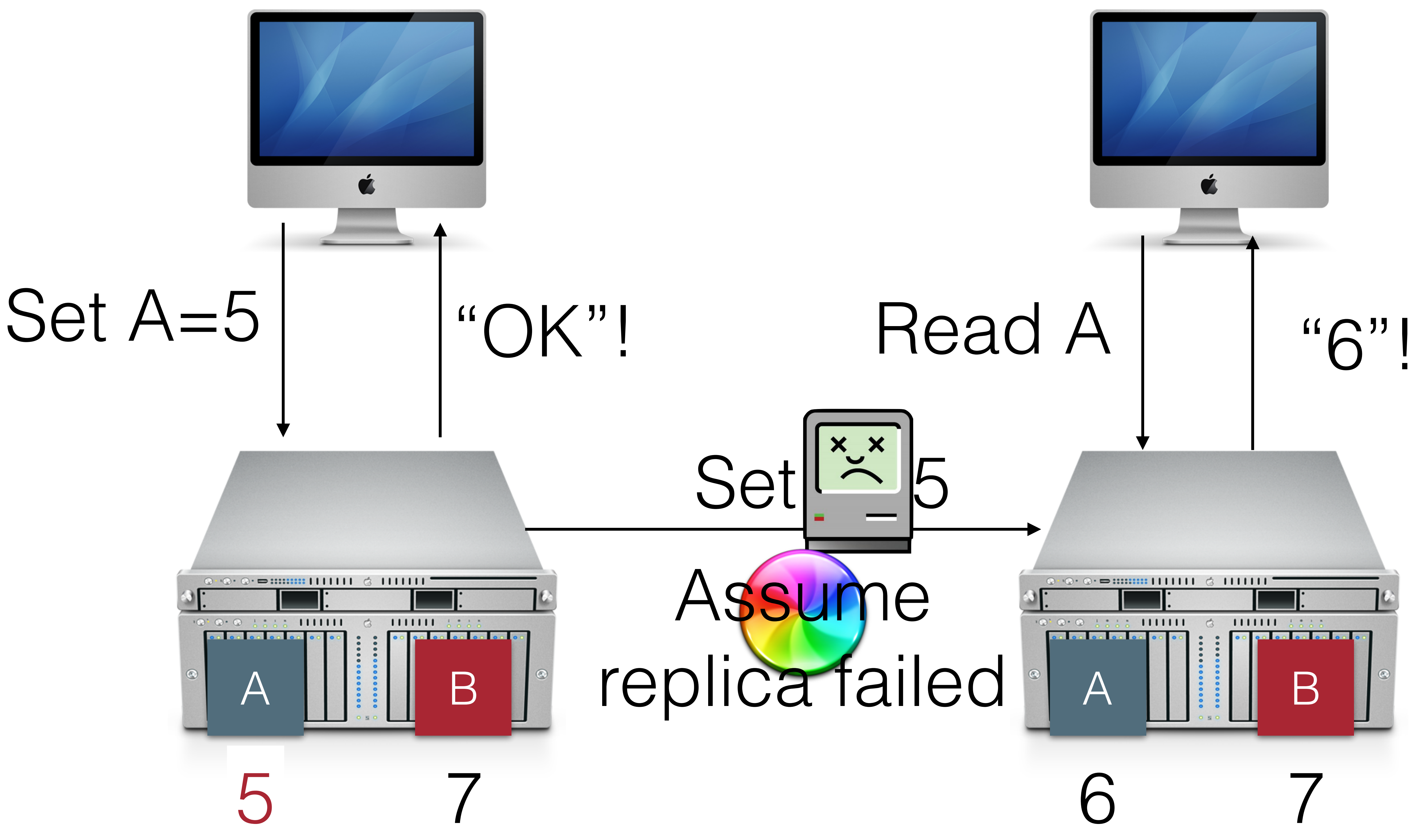


Still broken...



Network Partitions

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



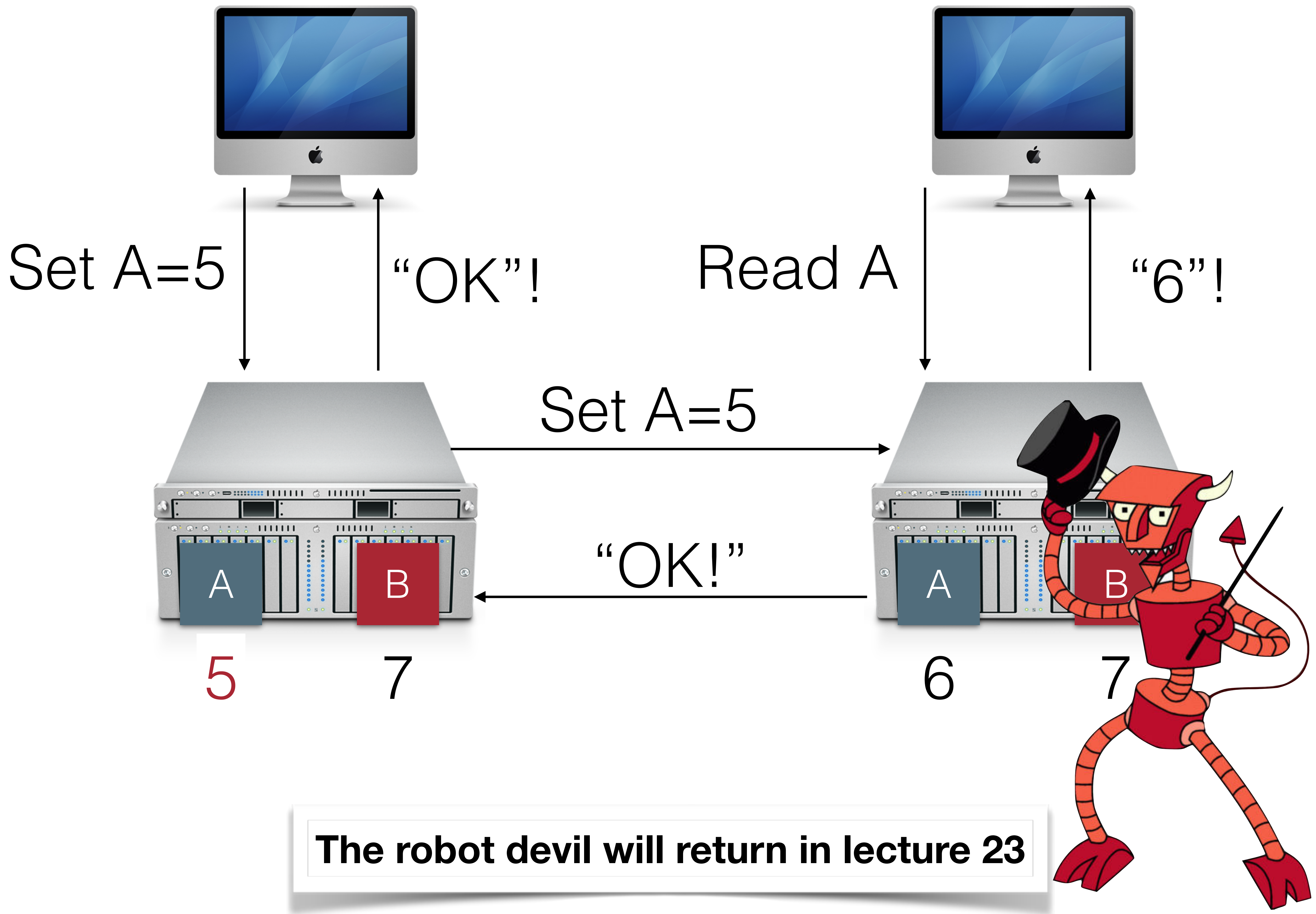
CAP Theorem

- Pick two of three:
  - Consistency: All nodes see the same data at the same time (strong 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***
  - If you relax your consistency guarantee (we'll talk about in a few weeks), you might be able to guarantee THAT…

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 strong consistency guarantee

Still broken...



The robot devil will return in lecture 23

This work is licensed under a Creative Commons Attribution-ShareAlike license