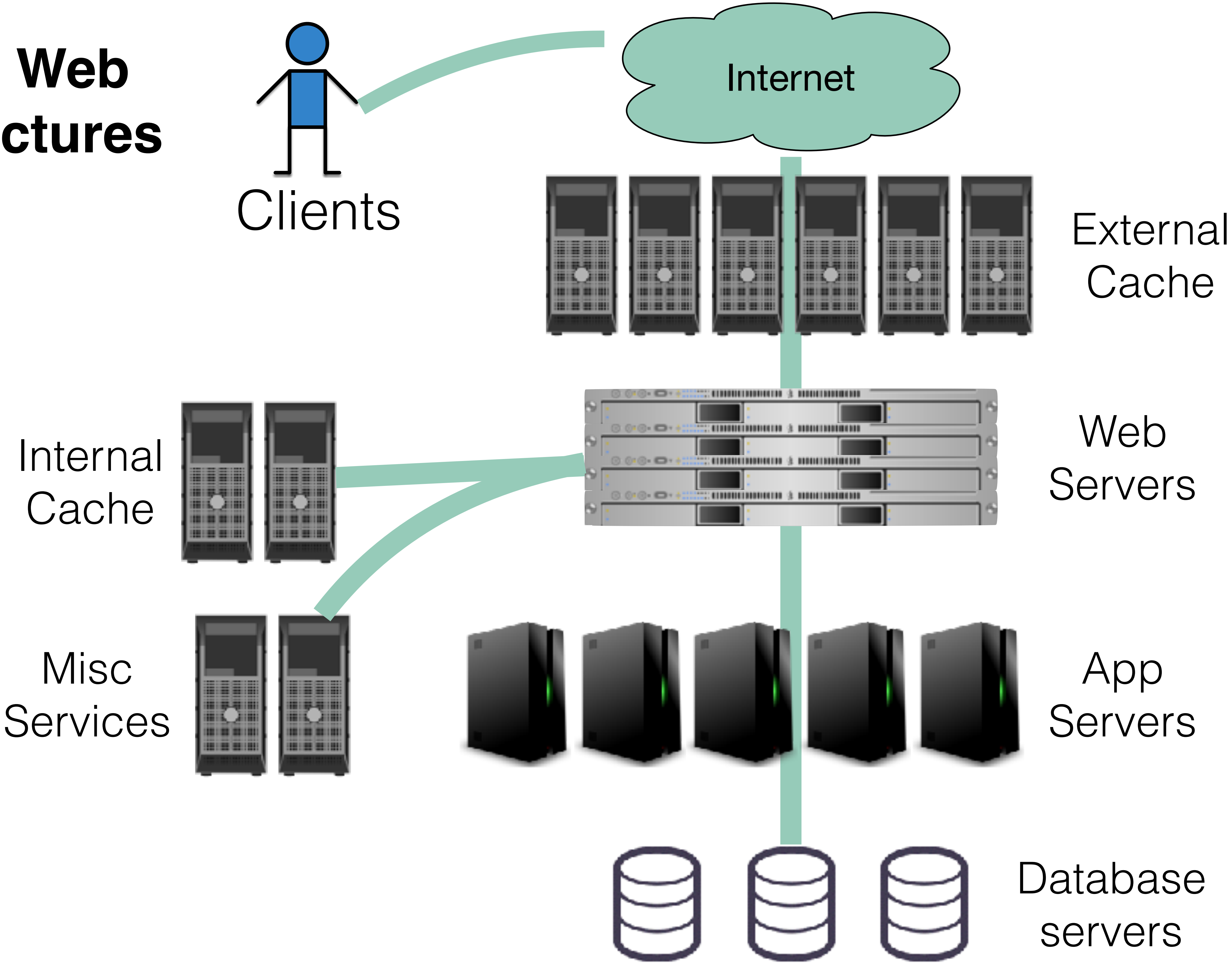
Consistency: Strict & Sequential

SWE 622, Spring 2017
Distributed Software Engineering

GEORGE
MASON
UNIVERSITY

Review: Real Architectures

N-Tier Web Architectures



Clients

Internet

External Cache

Web Servers

Internal Cache

Misc Services

App Servers

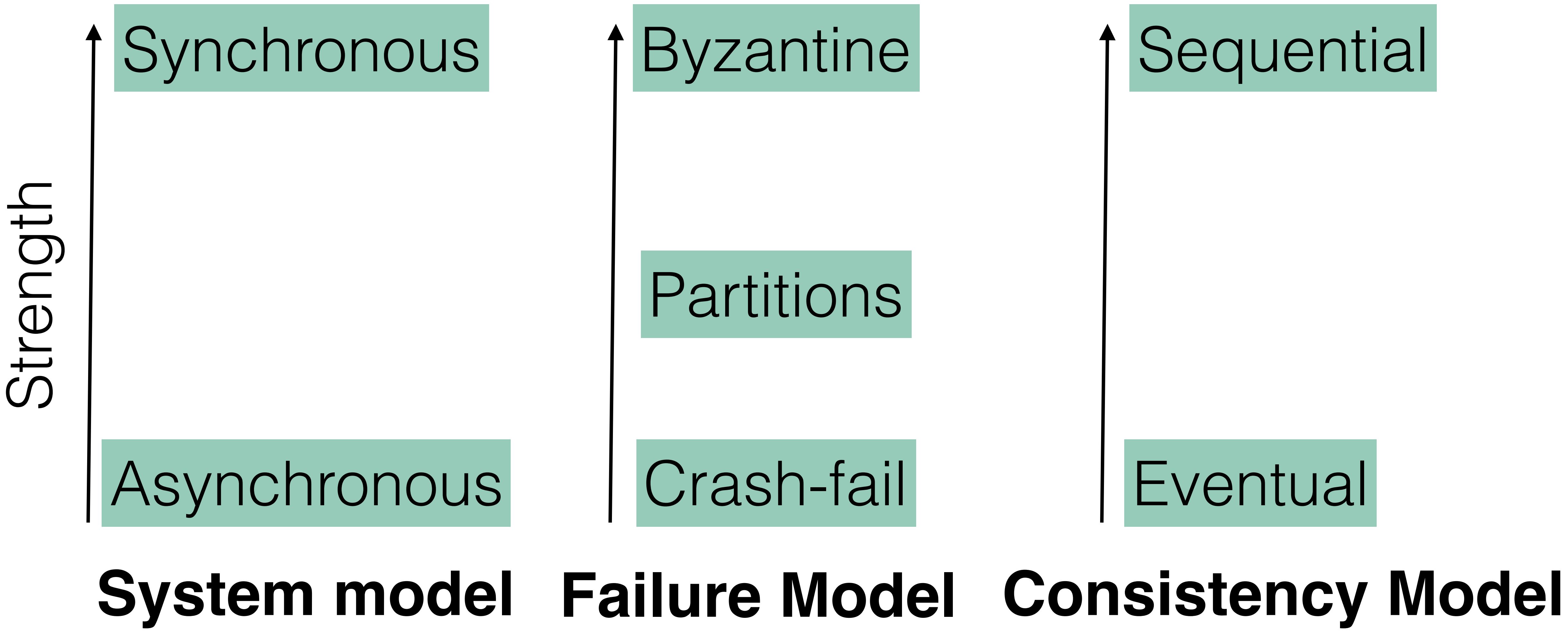
Database servers

Review: Abstractions & Models

To help design our algorithms and systems, we tend to leverage abstractions and models to make assumptions

Generally: Stronger assumptions -> worse performance

Weaker assumptions -> more complicated



Strength

Synchronous

Asynchronous

System model

Byzantine

Partitions

Crash-fail

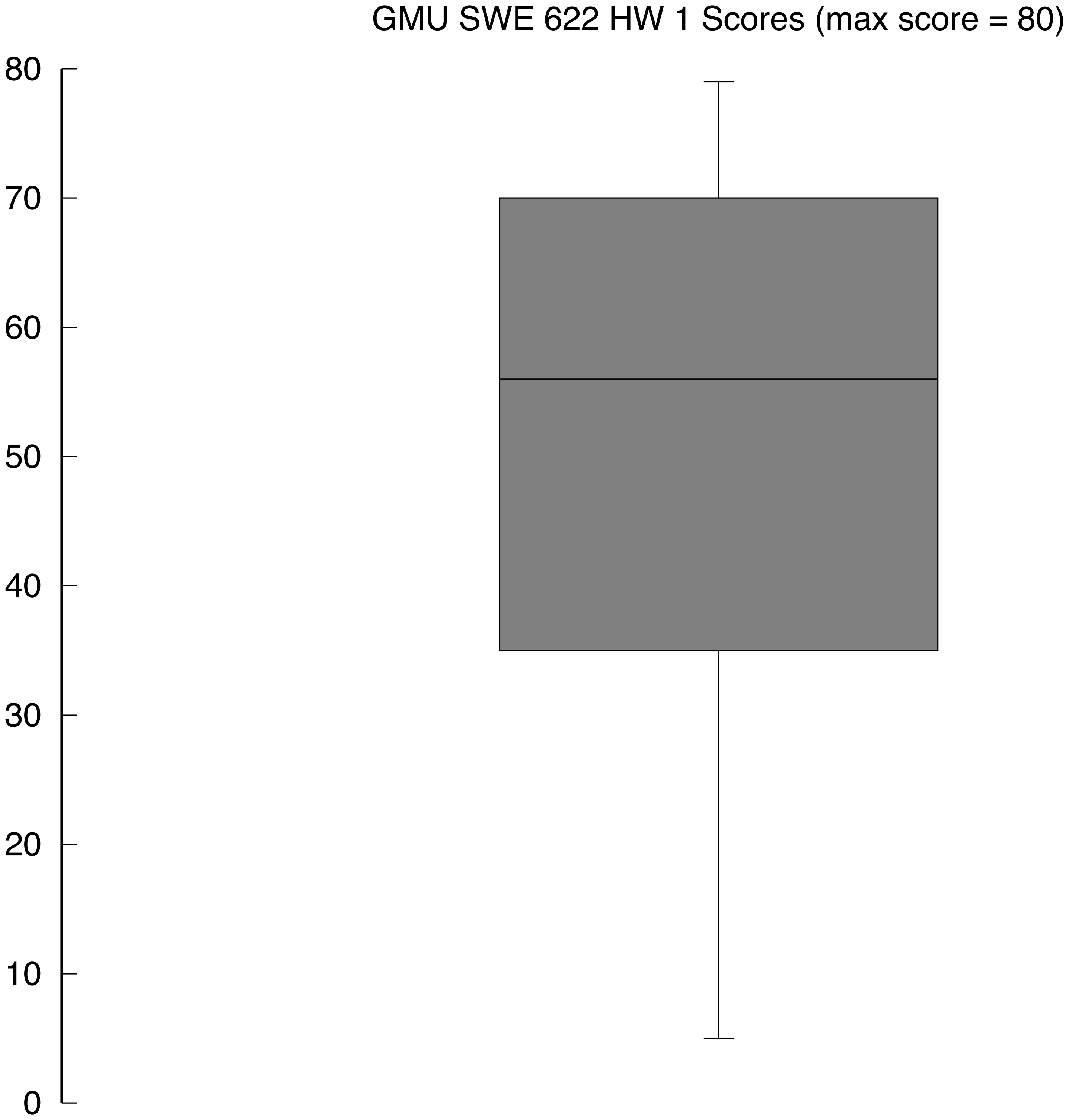
Failure Model

Sequential

Eventual

Consistency Model

HW1 Grades are in Blackboard



Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
  cachedFile = getCachedFileAndPutInCache(path);
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
   cachedFile = getCachedFileAndPutInCache(path);
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Thread 1	Thread 2

Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
  cachedFile = getCachedFileAndPutInCache(path);
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Thread 1	Thread 2
findCachedFile -> null	

Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
  cachedFile = getCachedFileAndPutInCache(path);
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Thread 1	Thread 2
findCachedFile -> null	
getCachedFile -> F1	findCachedFile -> null

Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
  cachedFile = getCachedFileAndPutInCache(path);
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Thread 1	Thread 2
findCachedFile -> null	
getCachedFile -> F1	findCachedFile -> null
	getCachedFile -> F2

Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
  cachedFile = getCachedFileAndPutInCache(path);
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Thread 1	Thread 2
findCachedFile -> null	
getCachedFile -> F1	findCachedFile -> null
	getCachedFile -> F2
F1.lock	F2.lock

Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
  cachedFile = getCachedFileAndPutInCache(path);
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Better: correct, but only 1 thread can fetch a new file at a time

```
synchronized(cache){
 CloudFile cachedFile = findCachedFile(path);
 if(cachedFile == null)
 {
  cachedFile = getCachedFileAndPutInCache(path);
 }
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Review: HW1

```
CloudFile cachedFile = findCachedFile(path);
if(cachedFile == null)
{
   cachedFile = getCachedFileAndPutInCache(path);
}
```

Better: correct, but only 1 thread can fetch a new file at a time

```
synchronized(cache){
 CloudFile cachedFile = findCachedFile(path);
 if(cachedFile == null)
 {
  cachedFile = getCachedFileAndPutInCache(path);
 }
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Review: HW1

```
synchronized(cache){
    CloudFile cachedFile = findCachedFile(path);
}

if(cachedFile == null)
{
    cachedFile = downloadFile(path);
}
synchronized(cache)
{
    if(findCachedFile(path) == null)
        storeToCache(cachedFile);
    else
        cachedFile = findCachedFile(path)
}
cachedFile.lock.readLock().lock();
//Do stuff
cachedFile.lock.readLock().unlock();
```

Best: Release lock, fetch file, then double check that you didn't accidentally fetch it twice

The Sleeping Barber

- Barber:
  - Cuts 1 person’s hair at a time
  - When finished, dismiss customer. Check waiting room for more customers. If more, then cut next customer’s hair. If no more, take a nap
- Customer:
  - Walks in, sees if barber is napping, if so, wakes barber, else, goes to waiting room

The Sleeping Barber

The Sleeping Barber

Barber	Old Customer	New Customer

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	Sees barber cutting hair

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	Sees barber cutting hair
		Goes to waiting room

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	Sees barber cutting hair
		Goes to waiting room
Finishes cutting	Leaves	

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	Sees barber cutting hair
		Goes to waiting room
Finishes cutting	Leaves	
Checks waiting room		

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	Sees barber cutting hair
		Goes to waiting room
Finishes cutting	Leaves	
Checks waiting room		
Escorts to chair, cuts hair		Follows barber to get hair cut

The Sleeping Barber

The Sleeping Barber

Barber	Old Customer	New Customer

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Sees barber cutting hair

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Sees barber cutting hair
Checks waiting room		Walks slowly to waiting room

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Sees barber cutting hair
Checks waiting room		Walks slowly to waiting room
Goes to sleep		Sits in waiting room

The Sleeping Barber

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Gets lock to check on barber

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Gets lock to check on barber
Gets lock to check waiting room (blocked)		Walks slowly to waiting room

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Gets lock to check on barber
Gets lock to check waiting room (blocked)		Walks slowly to waiting room
		Sits in waiting room

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Gets lock to check on barber
Gets lock to check waiting room (blocked)		Walks slowly to waiting room
		Sits in waiting room
		Releases lock

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Gets lock to check on barber
Gets lock to check waiting room (blocked)		Walks slowly to waiting room
		Sits in waiting room
		Releases lock
Acquires lock		

Fix: Barber can not check or new customers while customers are entering

The Sleeping Barber

Barber	Old Customer	New Customer
Cutting Hair	In Chair	
Finishes cutting	Leaves	Gets lock to check on barber
Gets lock to check waiting room (blocked)		Walks slowly to waiting room
		Sits in waiting room
		Releases lock
Acquires lock		
Checks waiting room, finds customer		

Fix: Barber can not check or new customers while customers are entering

Today

- Consistency & Memory Models
- Strict Consistency
- Sequential Consistency
- Distributed Shared Memory

Quiz: What’s the output?

```
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)
        System.out.println(“OK");
    }
}
```

Quiz: What’s the output?

```
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)
        System.out.println(“OK");
    }
}
```

```
"OK"
```

Quiz: What’s the output?

```
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)
        System.out.println(“OK");
    }
}
```

```
"OK"
```

```
"OK"
"OK"
```

Quiz: What’s the output?

```
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)
        System.out.println(“OK");
    }
}
```

```
"OK"
```

```
"OK"
"OK"
```

WTF?

Quiz: What’s the output?

```
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)
         System.out.println(“OK");
    }
}
```

""

"OK"

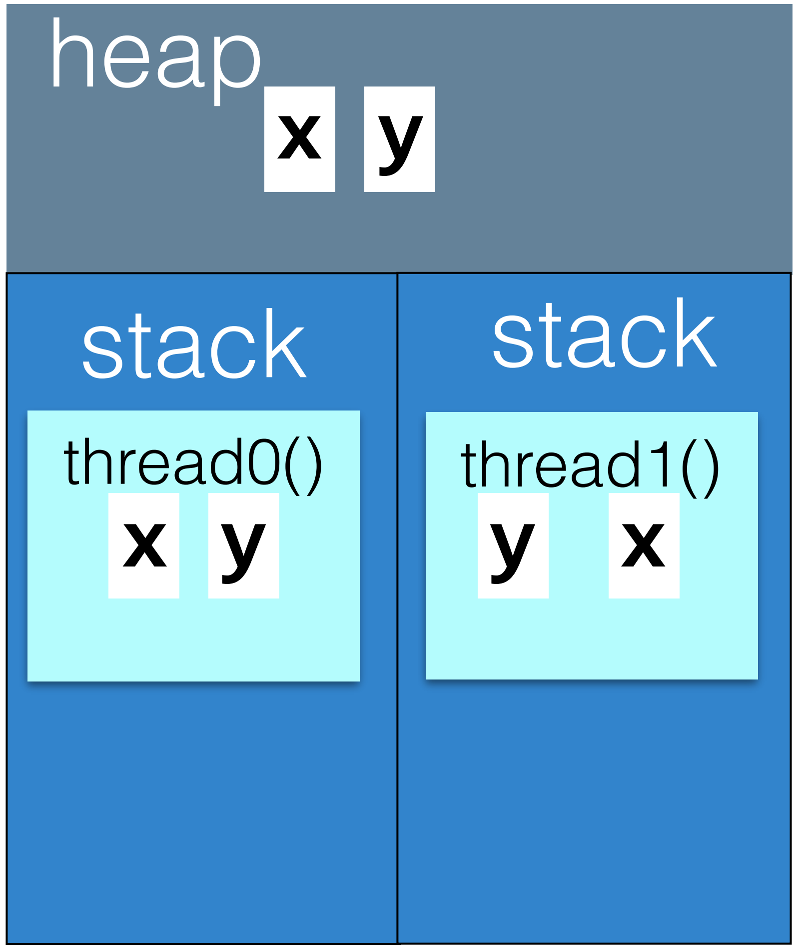
"OK"
"OK"

WTF?

Quiz: What’s the output?

```
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)
        System.out.println(“OK");
  }
}
```



Java Memory Model: Threads are allowed to cache reads and writes

Java Memory Model

CPU 1

thread0()

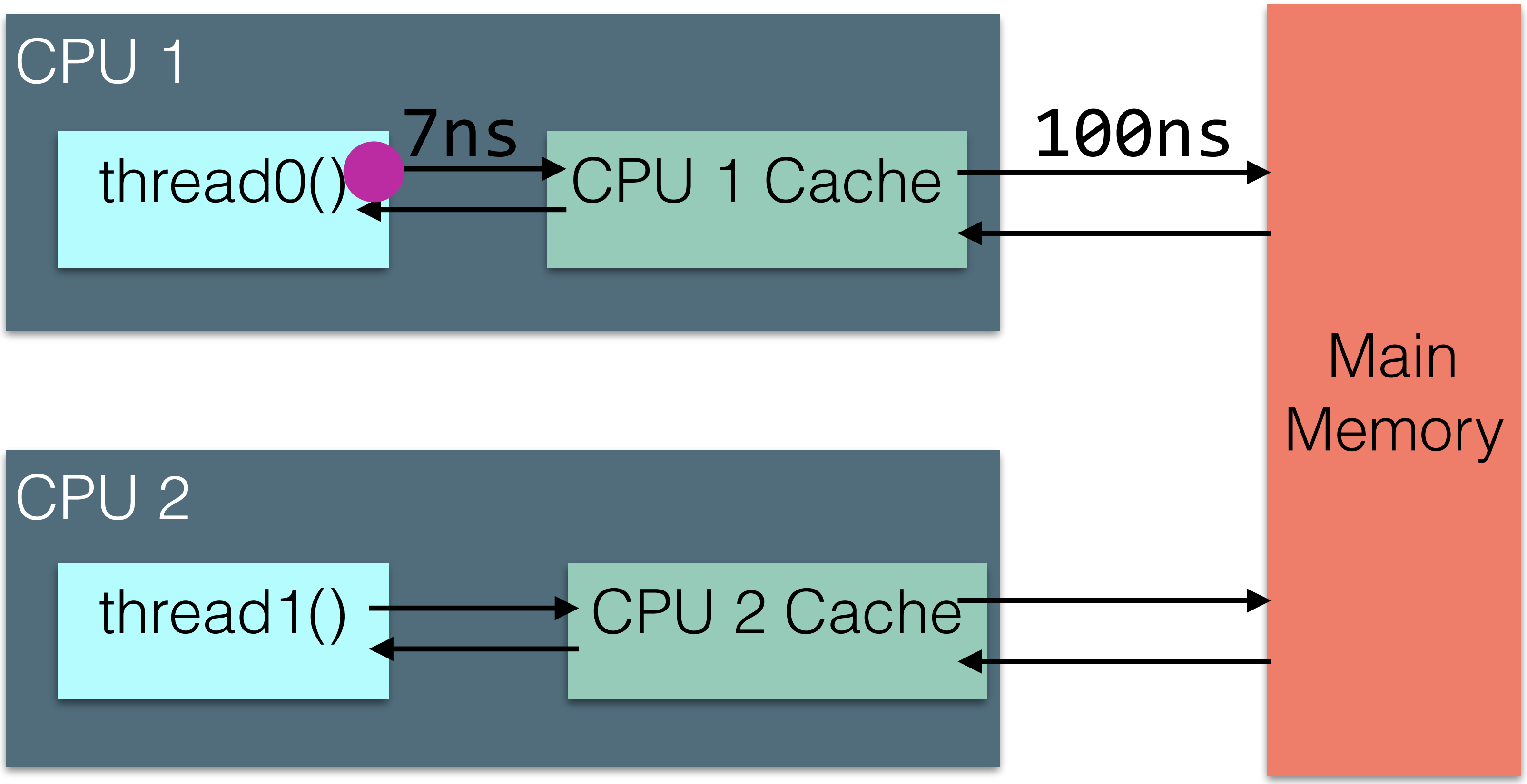
CPU 2

thread1()

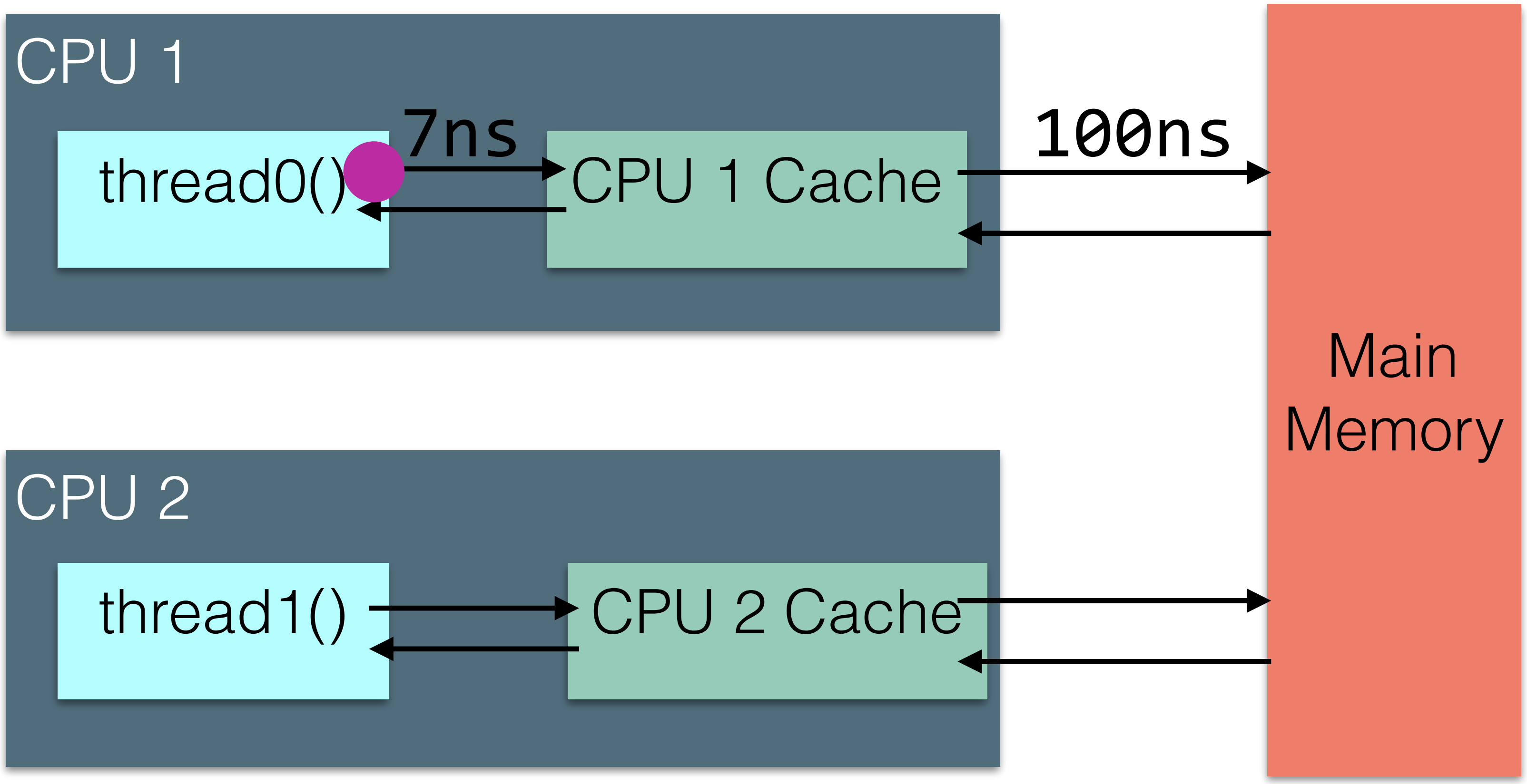
Main Memory

Java Memory Model

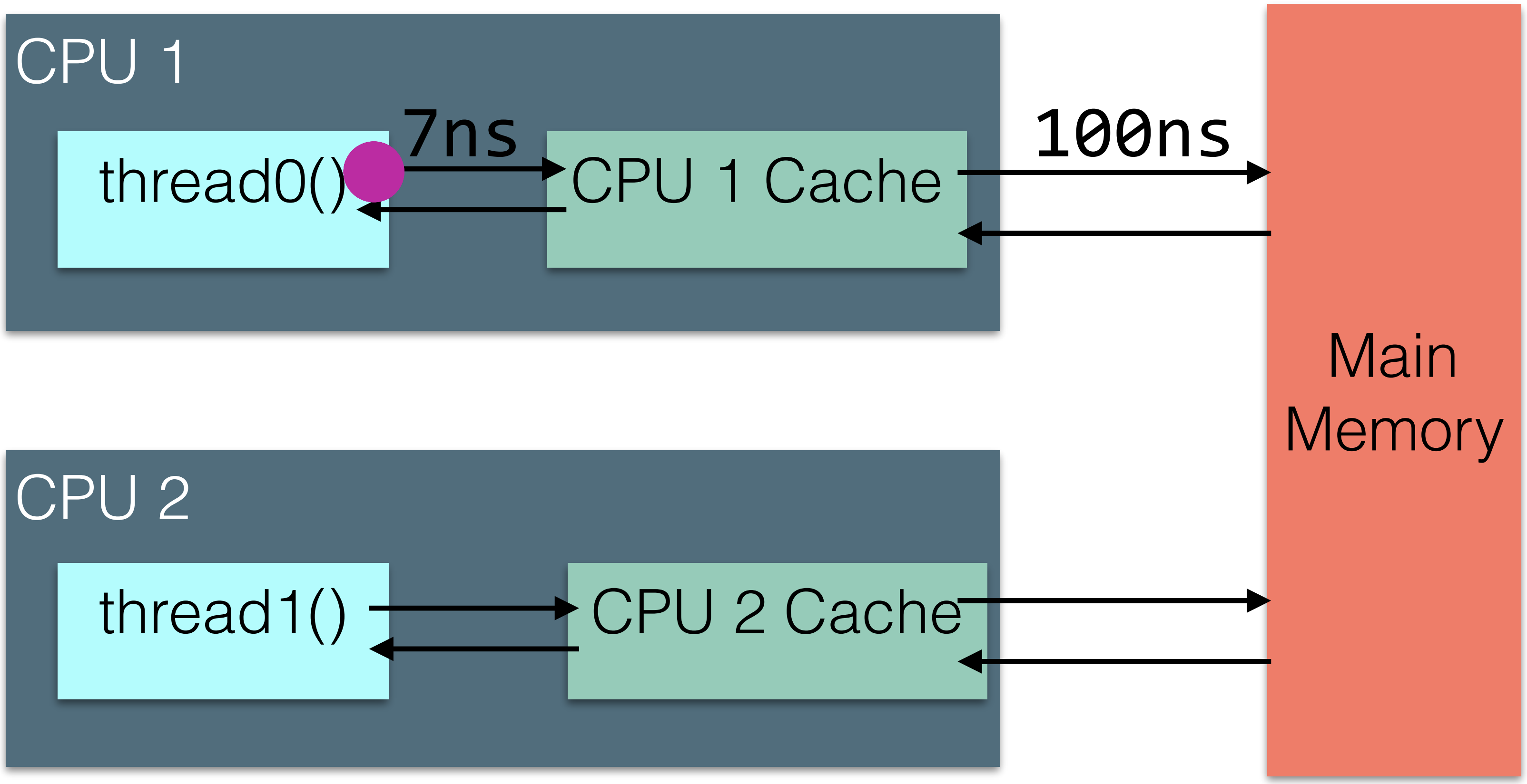
Java Memory Model



Java Memory Model



Java Memory Model



Quiz: What’s the output?

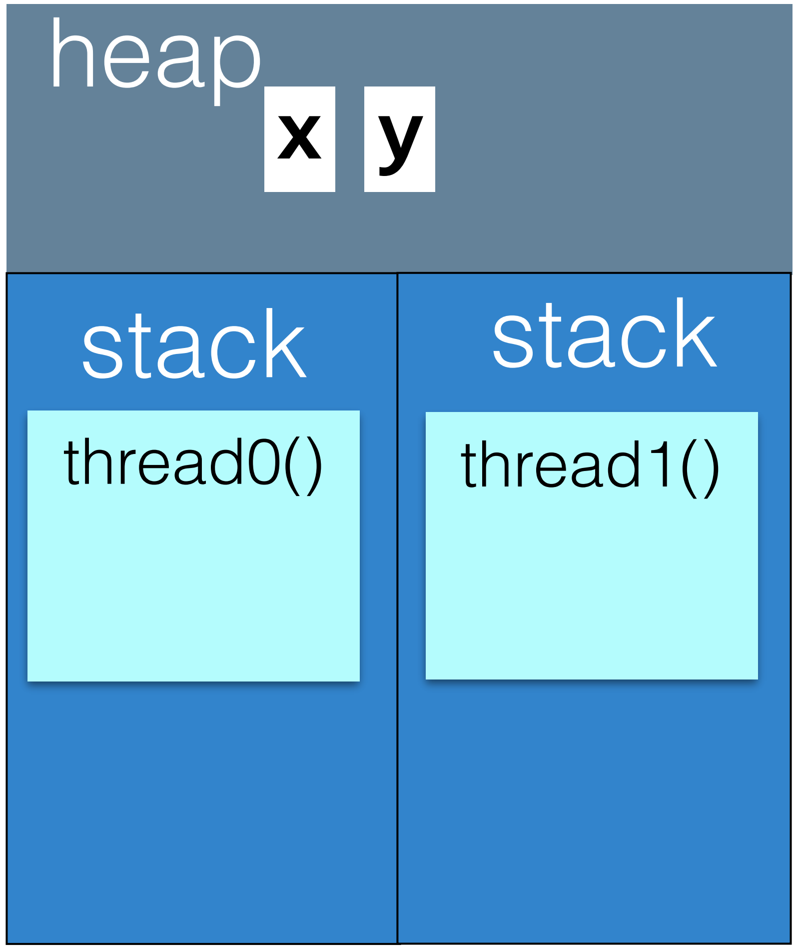
```
class MyObj {
    volatile int x = 0;
    volatile int y = 0;

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

Quiz: What's the output?

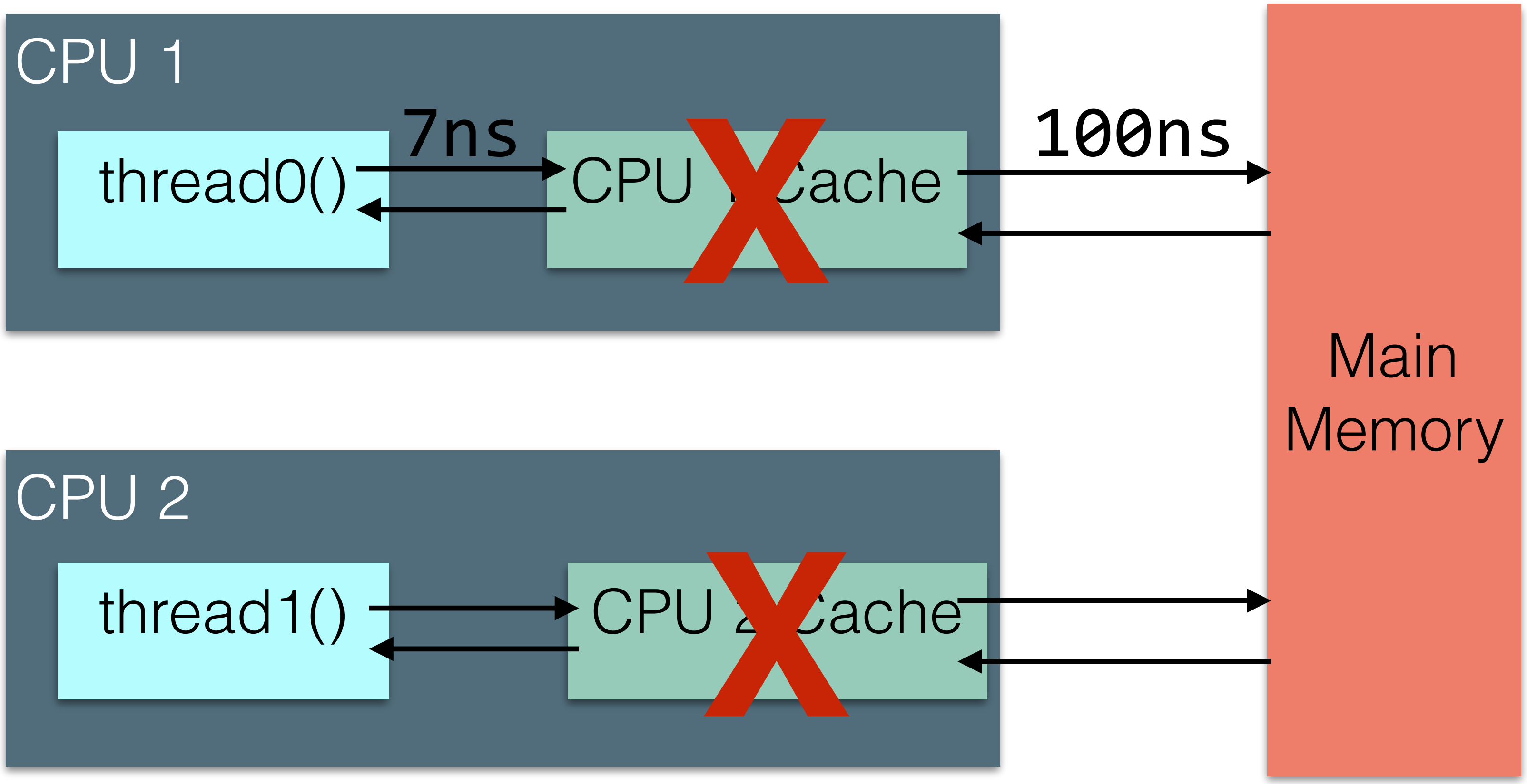
```
class MyObj {
  volatile int x = 0;
  volatile int y = 0;

  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`…)

Strict Consistency

- Each operation is stamped with a global (wall-clock, aka absolute) time
- Rules:
    - Each read sees the latest write
    - All operations on one CPU have time-stamps in execution order

Strict Consistency

```
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");
 }
}
```

	t=0	t=1
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

Strict Consistency

```
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");
 }
}
```

	t=0	t=1
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

	t=0	t=1	
CPU0	W(X) 1	R(Y) 0	
CPU1		W(Y) 1	R(X) 1

Strict Consistency

```
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");
  }
}
```

	t=0	t=1
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

	t=0	t=1	
CPU0	W(X) 1	R(Y) 0	
CPU1		W(Y) 1	R(X) 1

	t=0	t=1
CPU0	W(X) 1	R(Y) 0
CPU1	W(Y) 1	R(X) 0

Strict Consistency

```
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");
  }
}
```

	t=0	**t=1**
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

	t=0	**t=1**	
CPU0	W(X) 1	R(Y) 0	
CPU1		W(Y) 1	R(X) 1

	~~**t=0**~~	~~**t=1**~~
CPU0	~~W(X) 1~~	~~R(Y) 0~~
CPU1	~~W(Y) 1~~	~~R(X) 0~~

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)

Sequential Consistency

Strict Consistency

	t=0	t=1
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

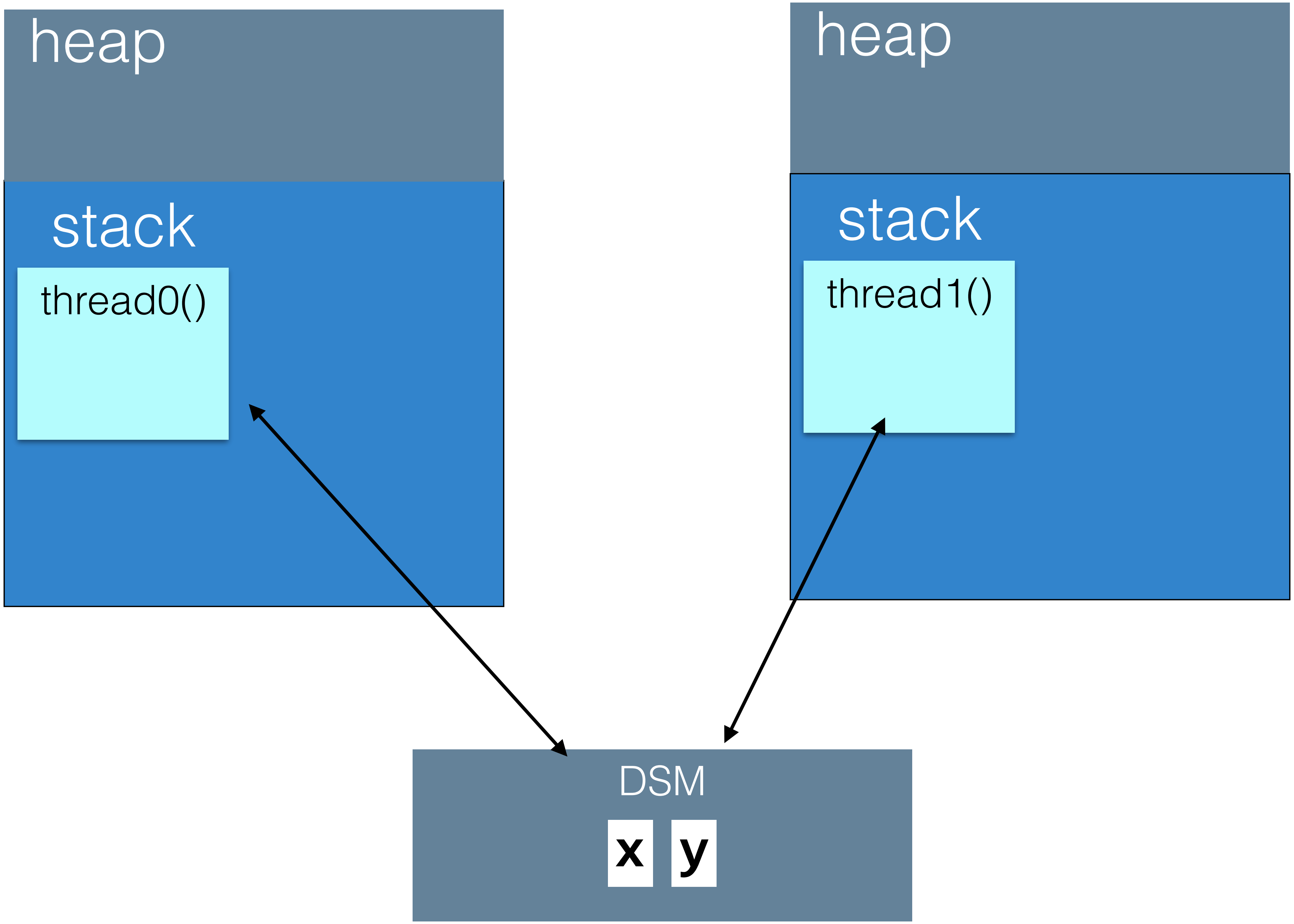
Sequential Consistency

CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

Distributed Shared Memory

- How do multiple nodes communicate in a distributed system?
  - Message passing
    - E.g. RMI
  - Shared memory
    - Make it look like there are just a bunch of threads on one machine

Distributed Shared Memory



heap

stack

thread0()

heap

stack

thread1()

DSM

x y

Naïve DSM

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

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 = 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 = 1;
  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 = 1;

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

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

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

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

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

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

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

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

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

```
ne2 {
= 1;
= 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

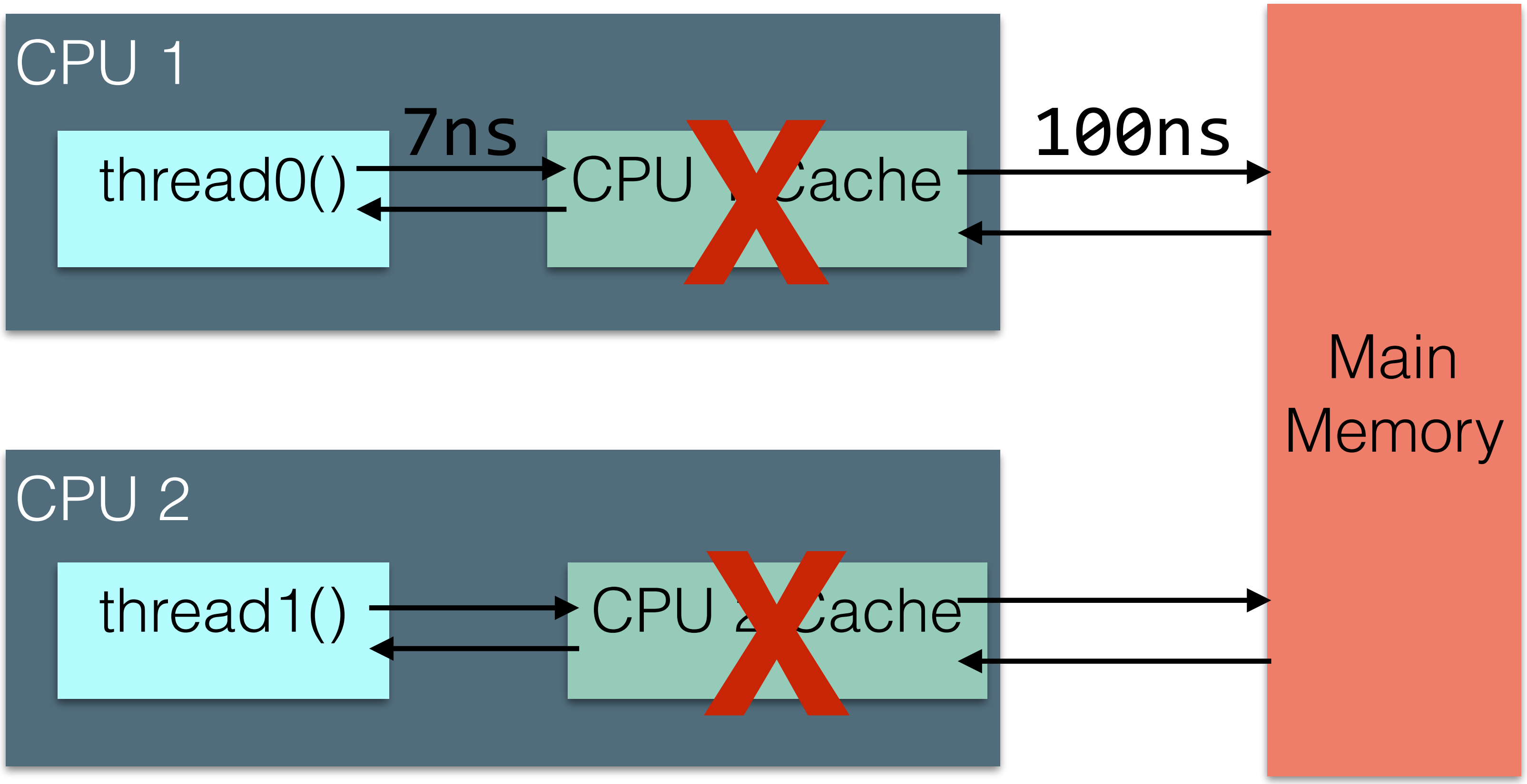
Sequentially Consistent DSM

- 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

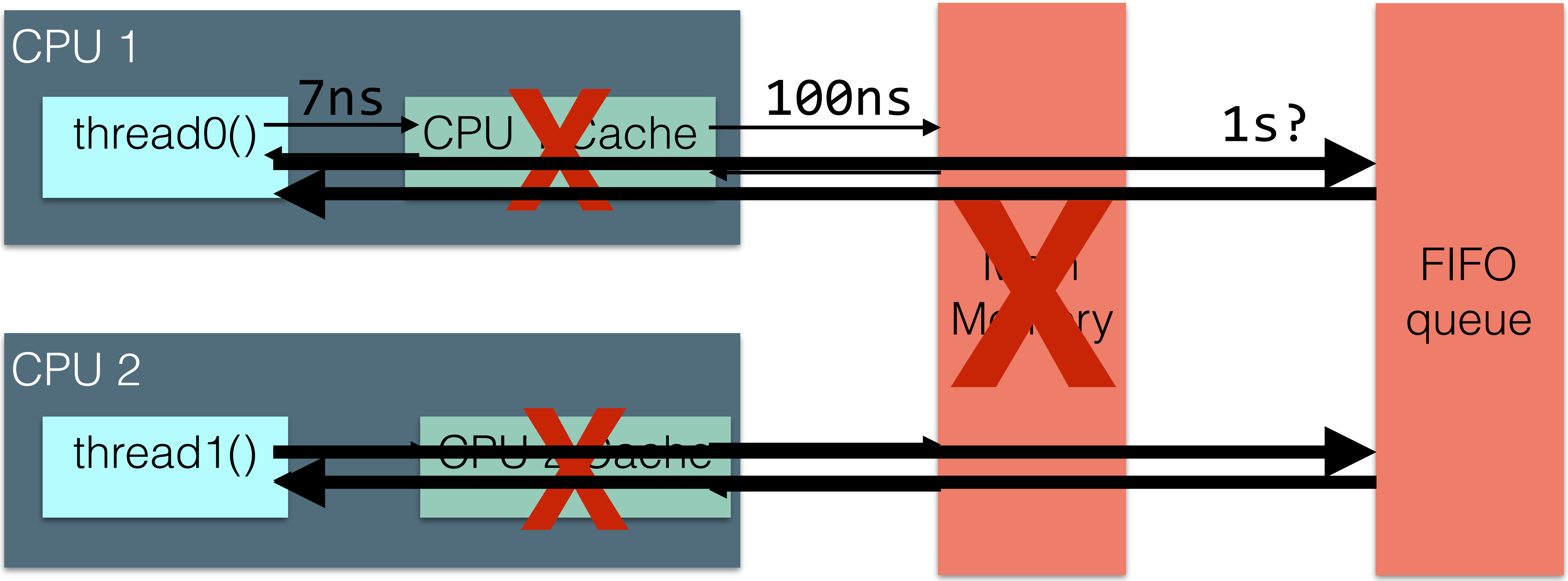
Sequentially Consistent DSM

- 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



Sequentially Consistent DSM



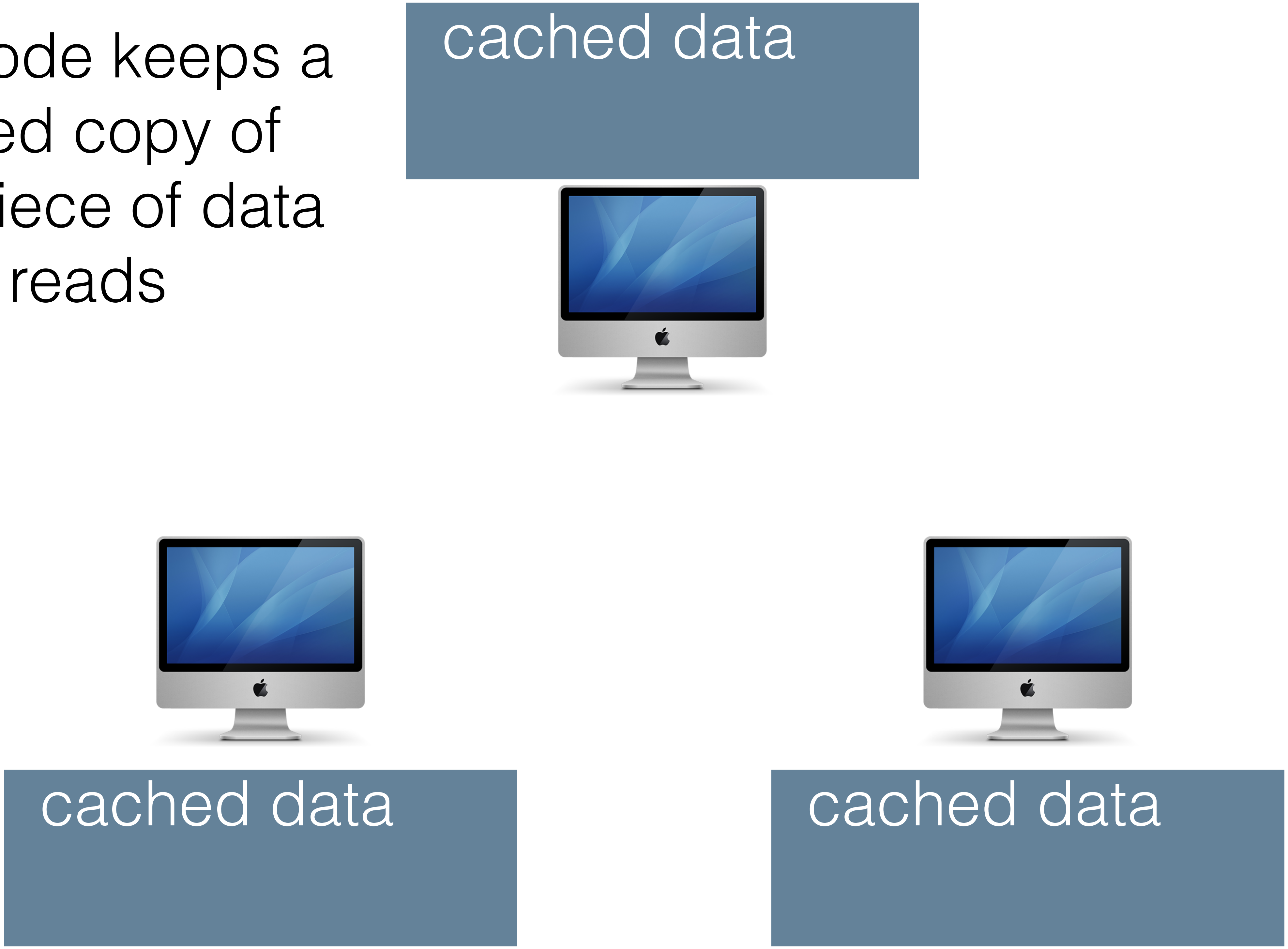
Ivy DSM

- **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

Ivy Architecture

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

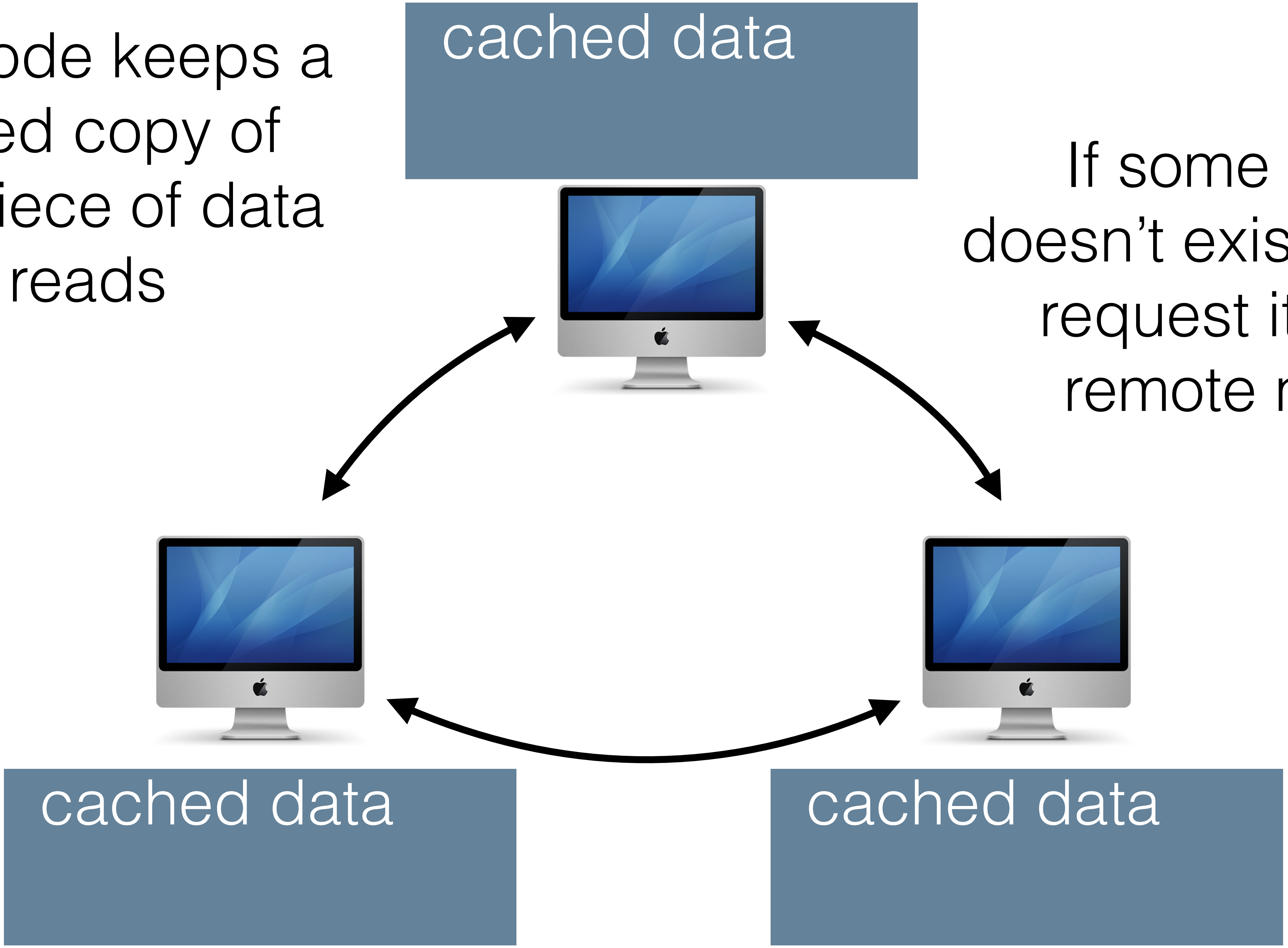


cached data

cached data

cached data

Ivy Architecture



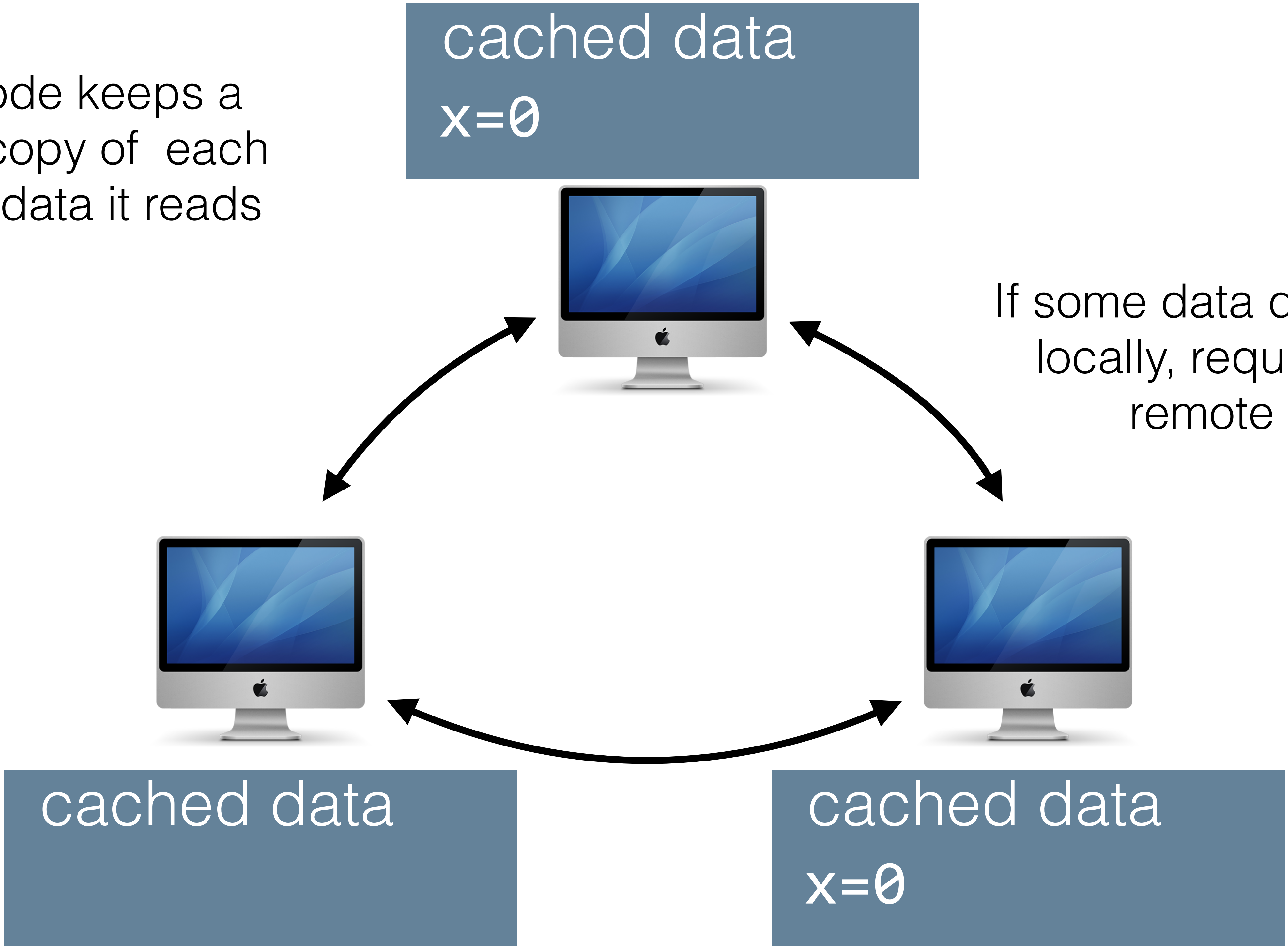
Each node keeps a cached copy of each piece of data it reads

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

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



cached data

x=0

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

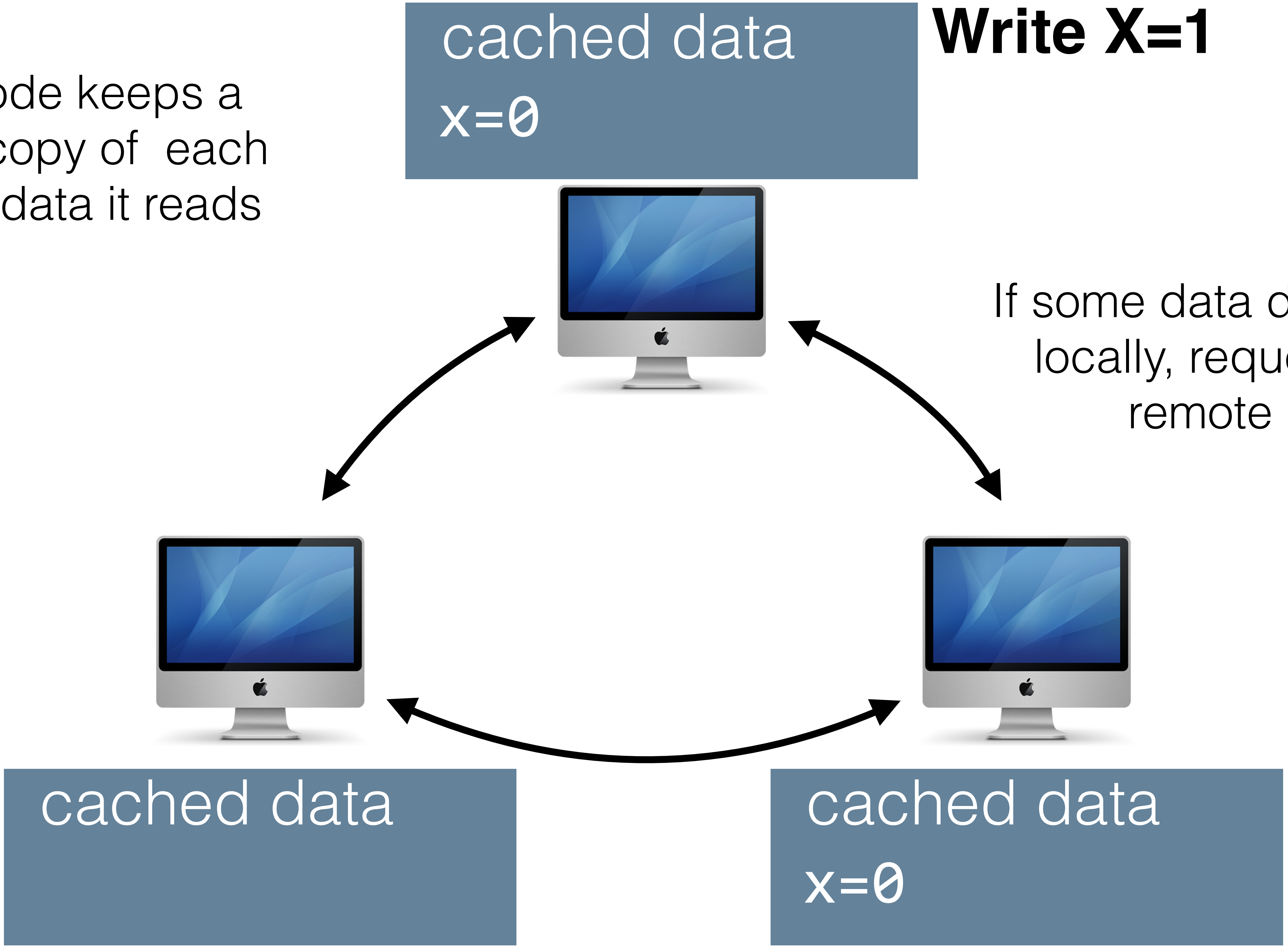
If some data doesn't exist locally, request it from remote node

cached data

cached data

x=0

Ivy Architecture



cached data

`x=0`

Write X=1

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

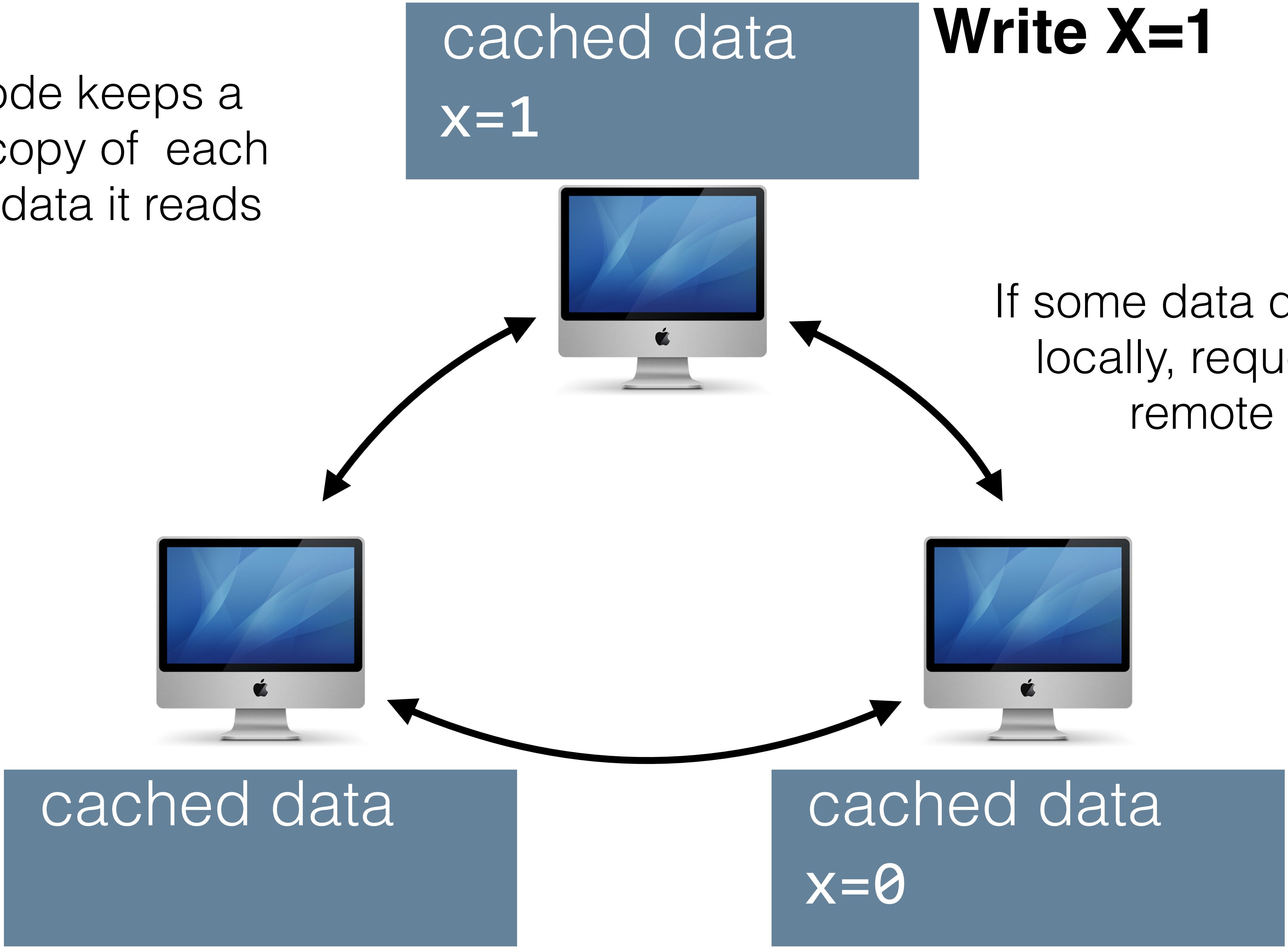
cached data

cached data

`x=0`

Ivy Architecture

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



cached data

x=1

Write X=1

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

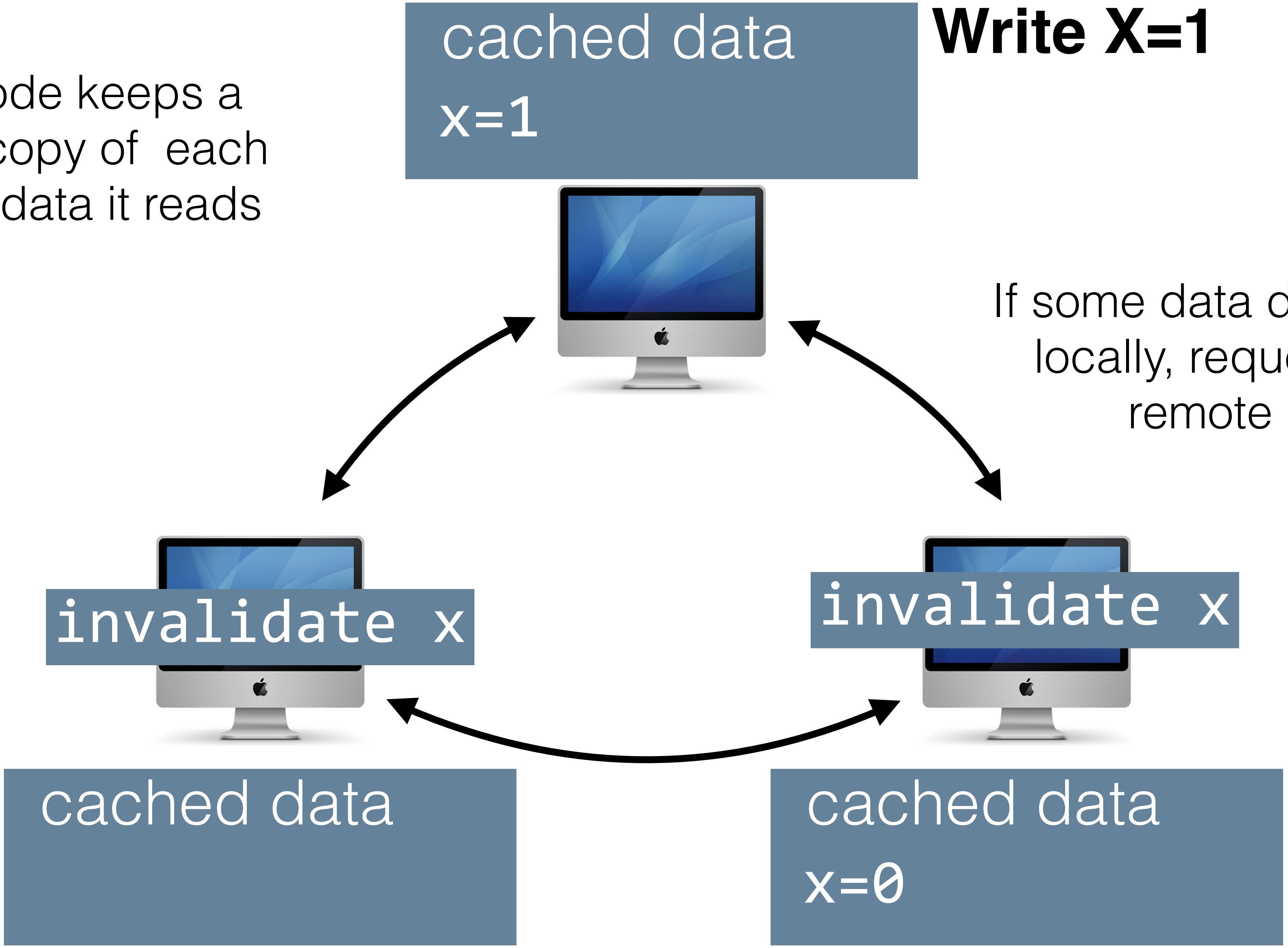
cached data

cached data

x=0

Ivy Architecture

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



cached data

x=1

Write X=1

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

invalidate x

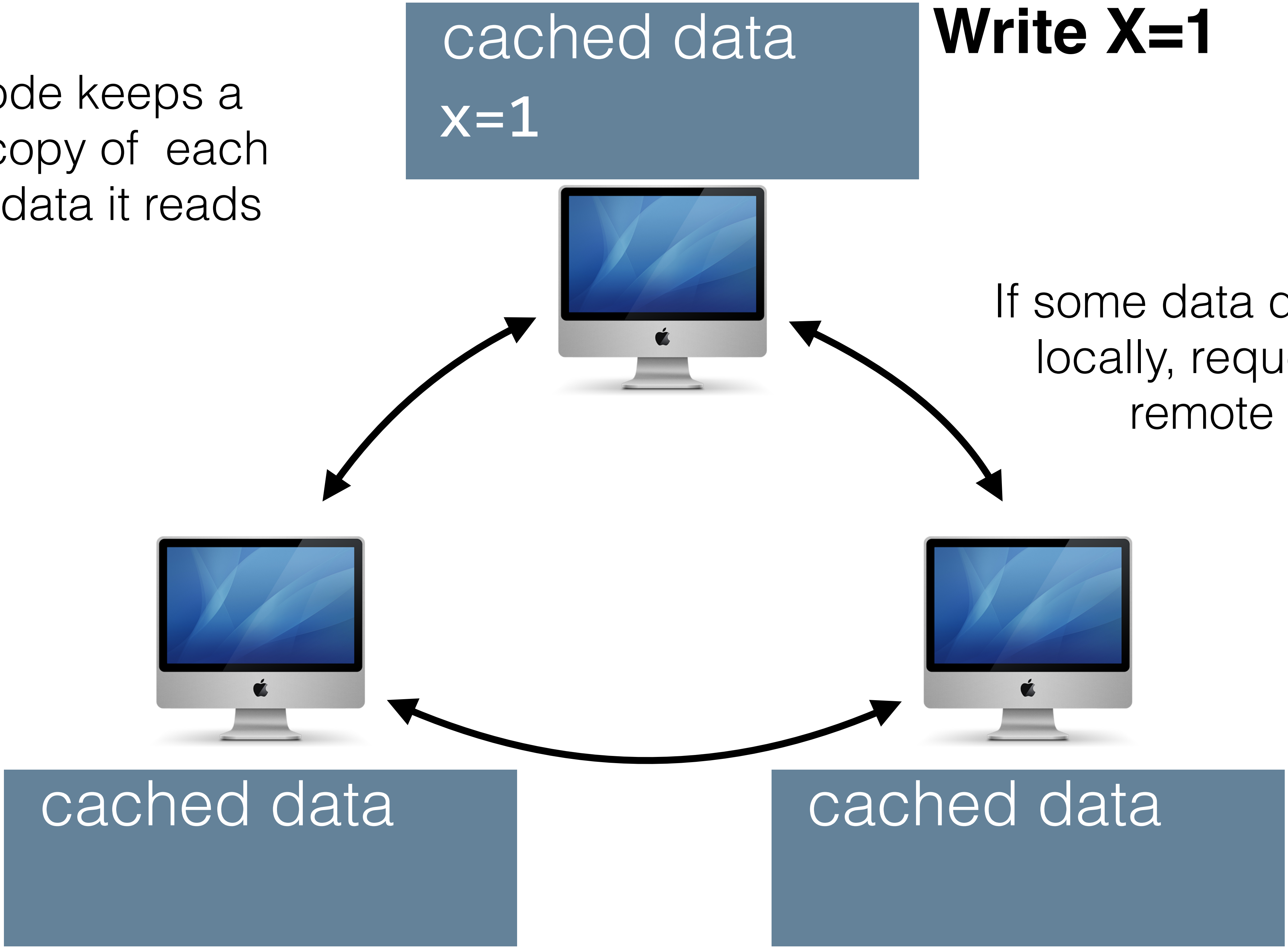
invalidate x

cached data

cached data

x=0

Ivy Architecture



cached data

x=1

Write X=1

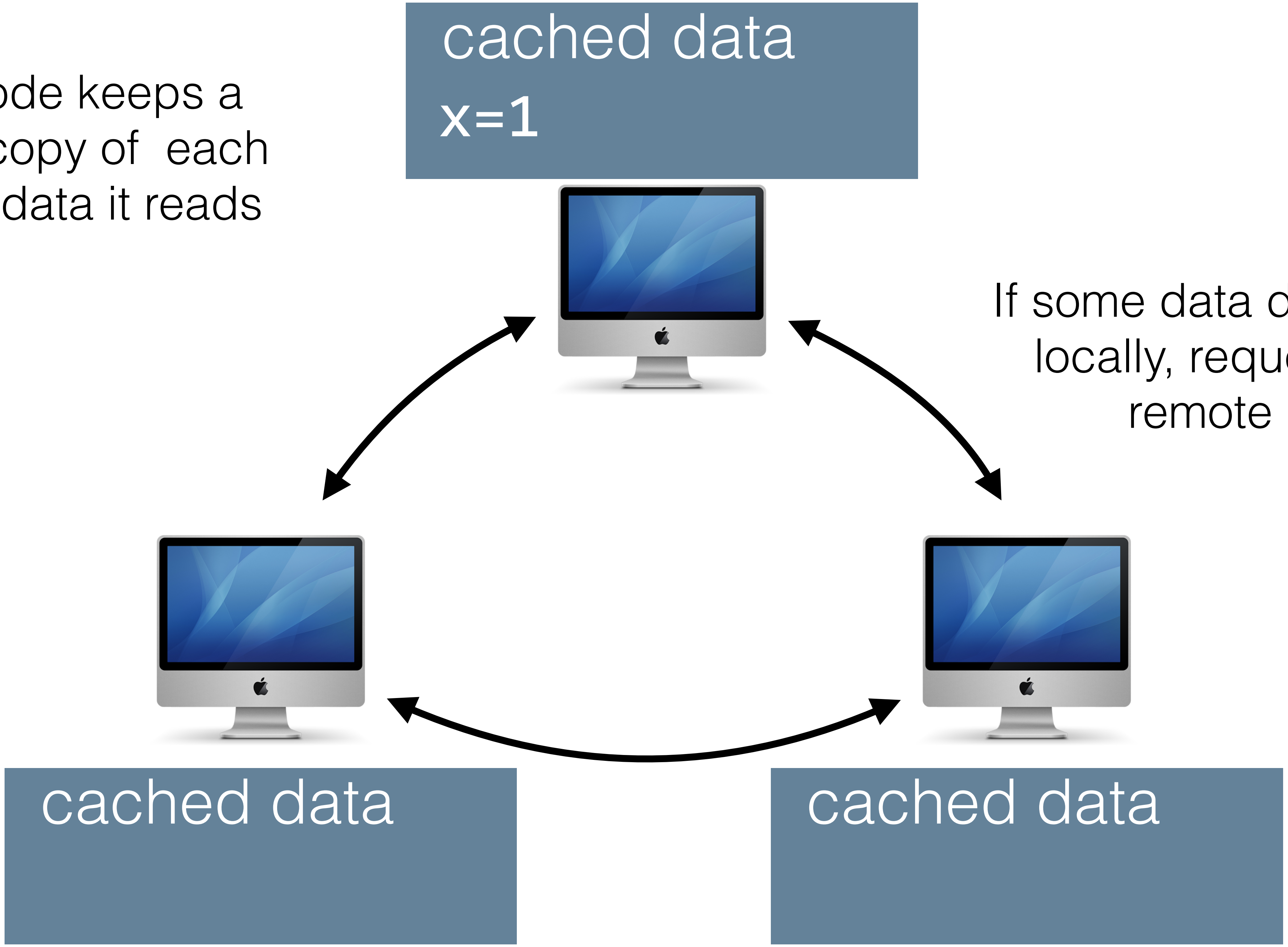
Each node keeps a cached copy of each piece of data it reads

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

cached data

cached data

Ivy Architecture



cached data

x=1

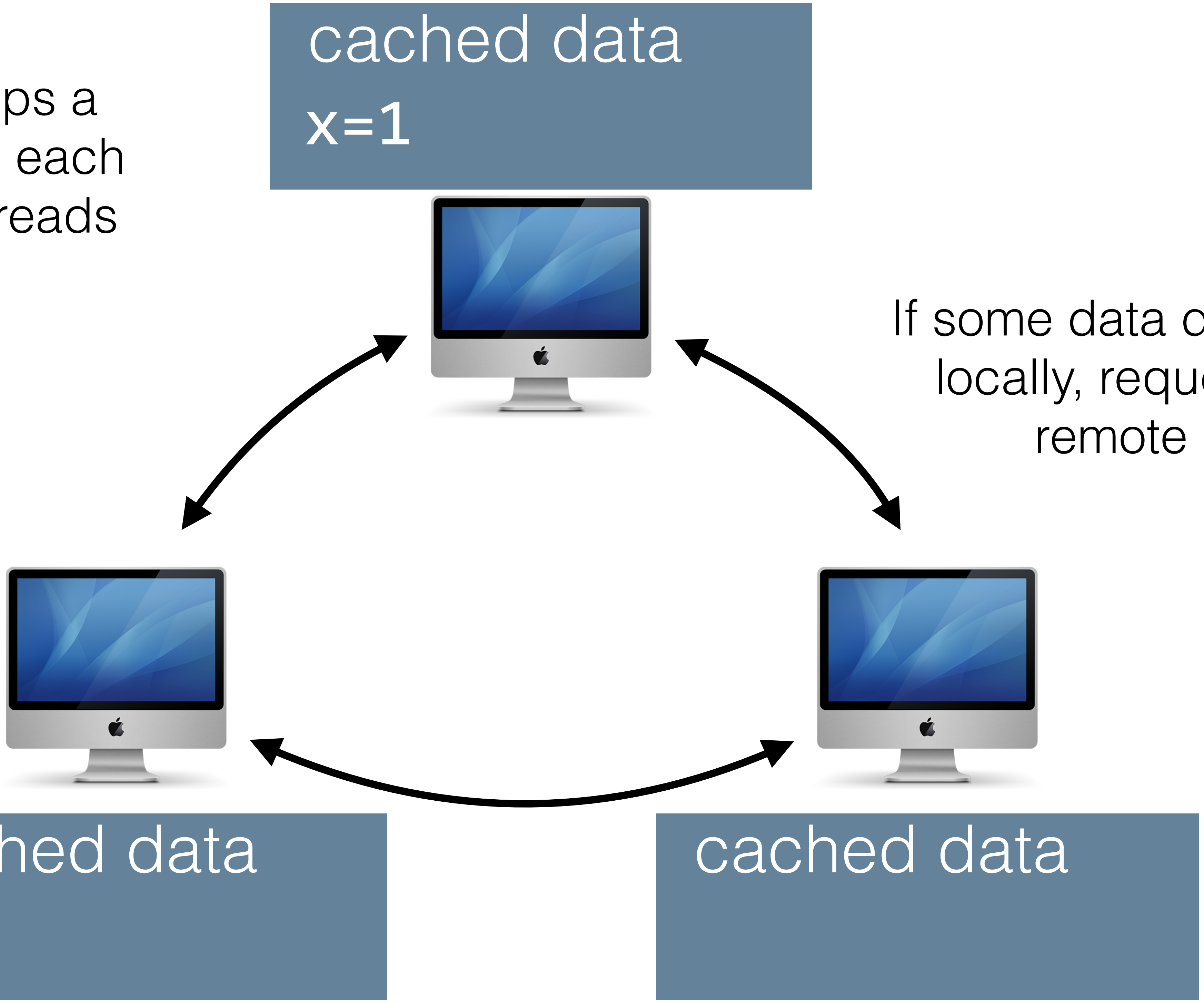
Each node keeps a cached copy of each piece of data it reads

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

cached data

cached data

Ivy Architecture



cached data

x=1

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

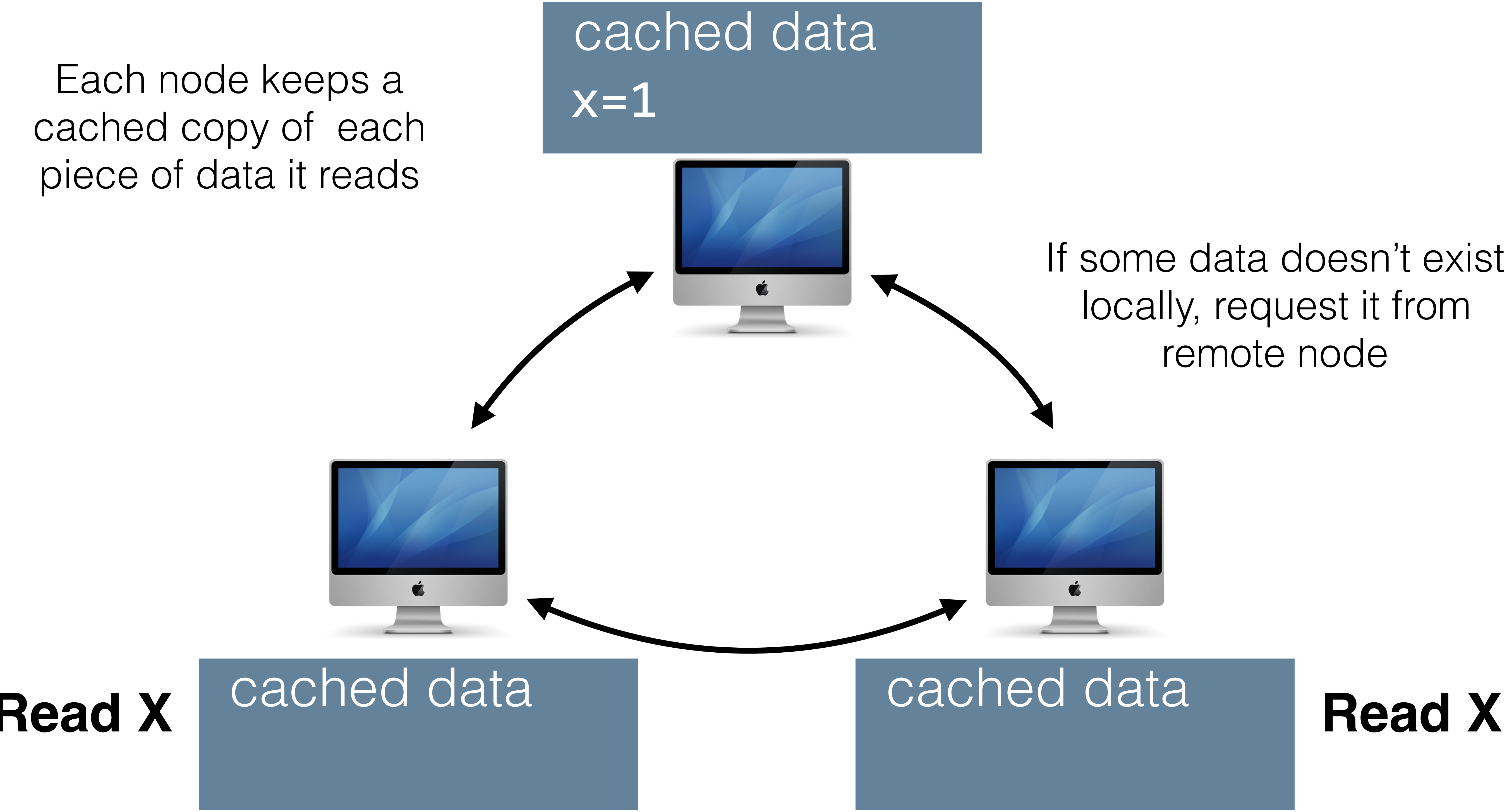
Read X

cached data

cached data

Read X

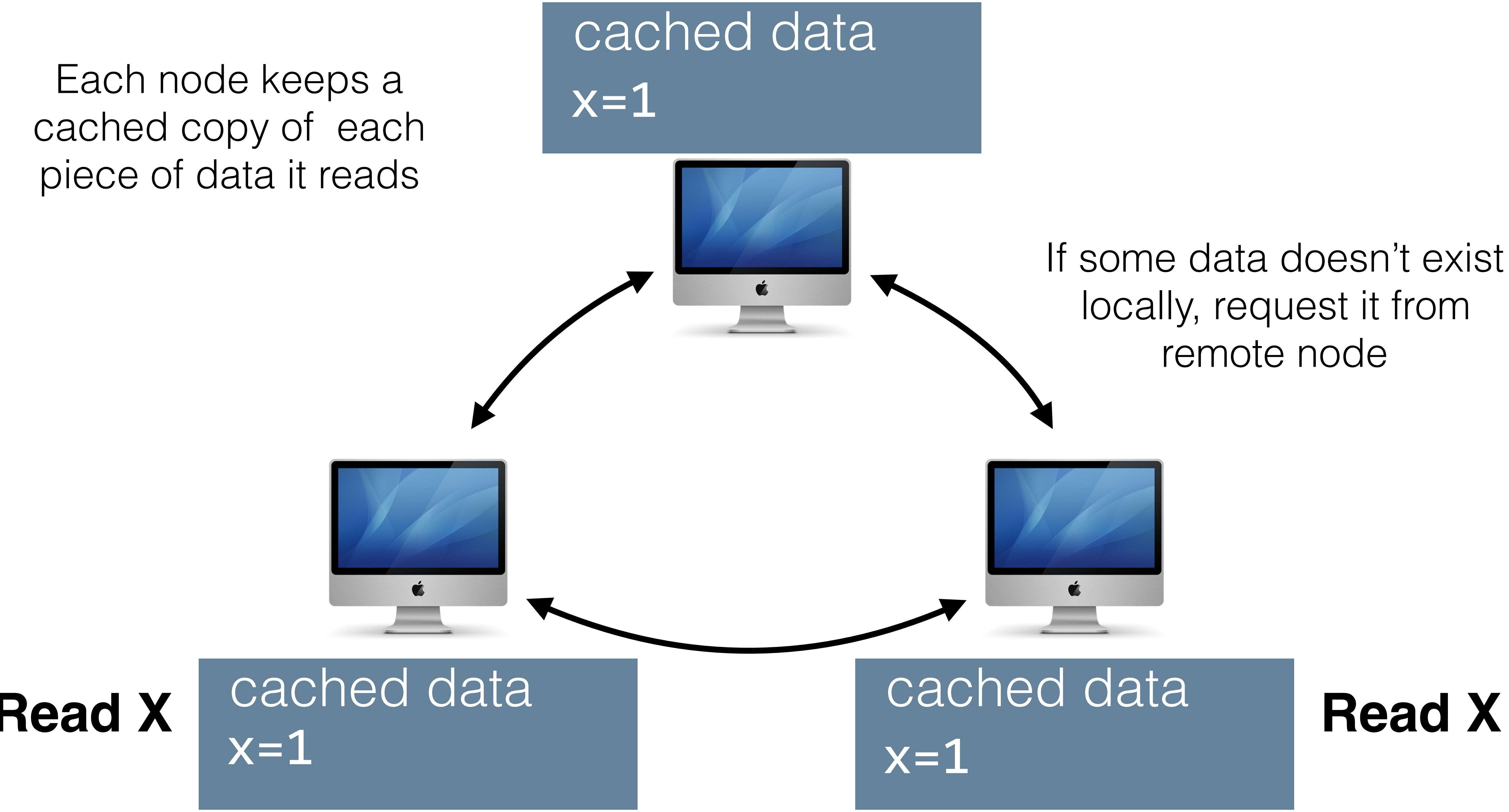
Ivy Architecture



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 Architecture



cached data
x=1

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

Read X

cached data
x=1

cached data
x=1

Read X

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

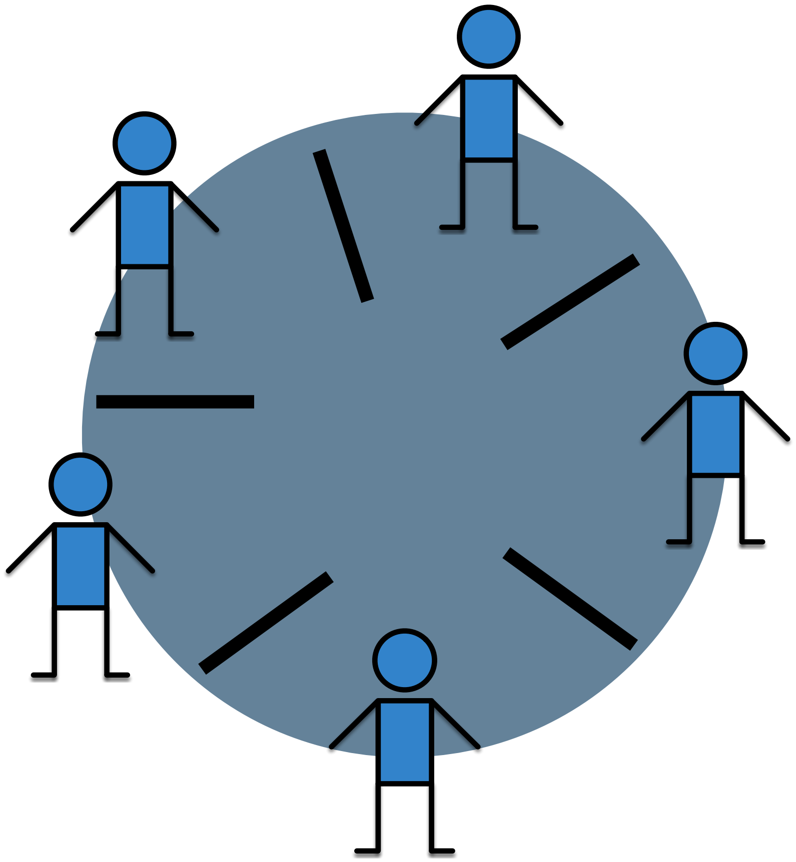
Ivy invariants

- 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 CFS + its lock server? :)

Lab: More Concurrency, lock, wait

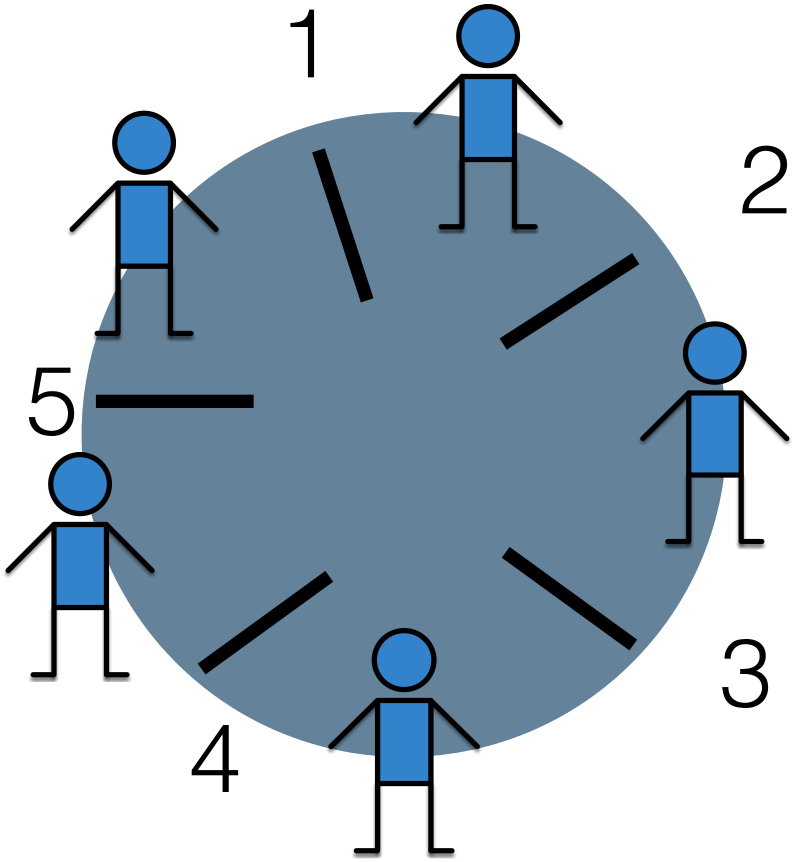
Dining Philosophers

- N philosophers seated around a circular table
  - One chopstick between each philosopher (N chopsticks)
  - A philosopher picks up both chopsticks next to him to eat
  - Philosophers may not pick up both chopsticks at the same time
- How do they all eat without deadlocking or starving?



Dining Philosophers

- Give each chopstick a lock
- Is this enough?
- Could deadlock!
- Solution: always grab right chopstick first?
  - Nope
- Solution: Index chopsticks, always grab biggest # first?
- Solution: Only get them if they are both free?



Your tasks

- I’ve provided an implementation of Dining Philosophers with a little simulator. It deadlocks. Fix it.
- Implement a small simulator for sleeping barber