Consistency: Relaxed

SWE 622, Spring 2017
Distributed Software Engineering

Review: HW2

- What did we do?
  - Cache->Redis
  - Locks->Lock Server
- Post-mortem feedback: http://b.socrative.com/ click on student login, then SWE622 as room name

Review: 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

Review: 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

Review: 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

Review: 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

Review: 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");
    }
}
```

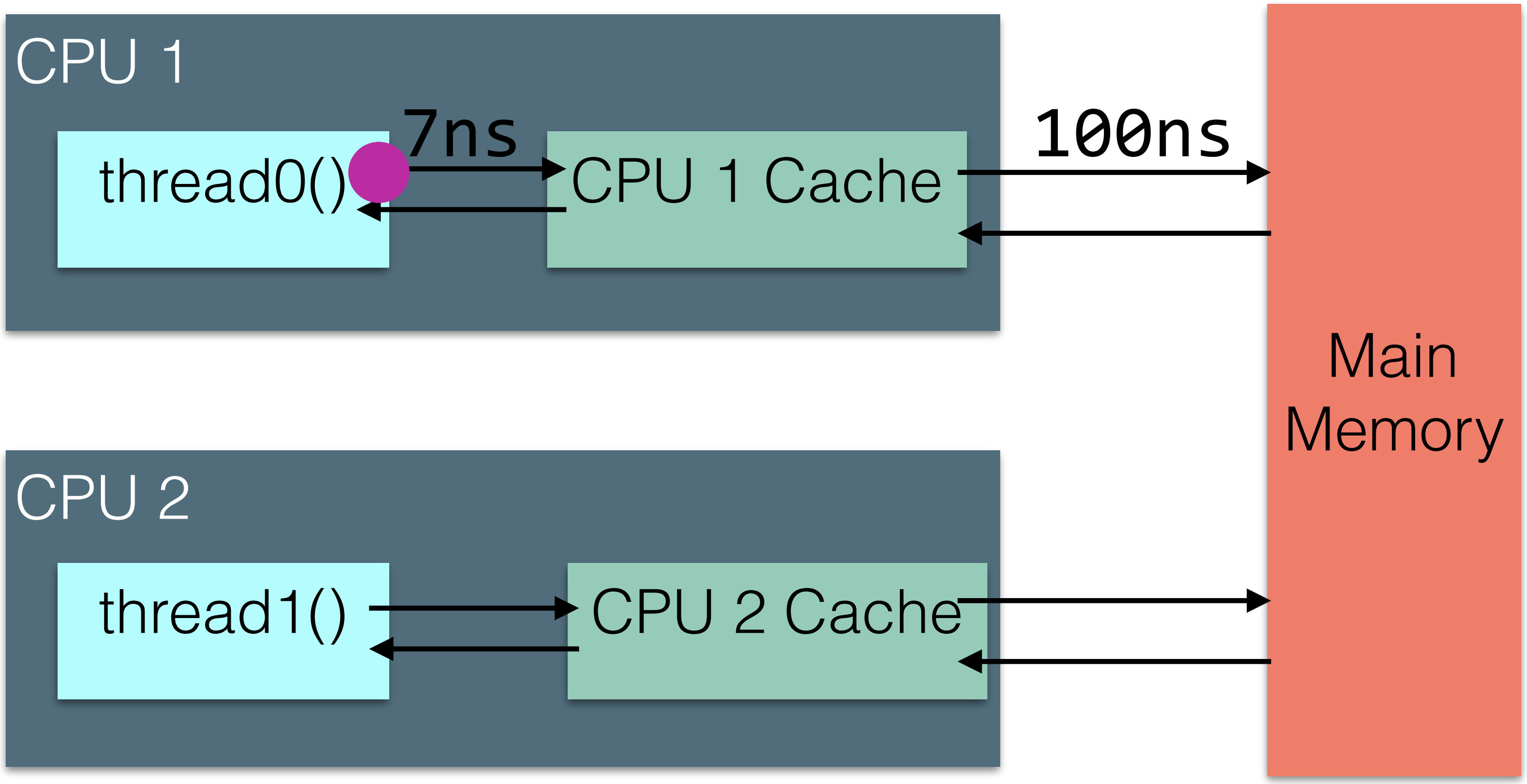
```
""
```

```
"OK"
```

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

WTF?

Review: Java Memory Model



Review: 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`…)

Review: 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: Quiz

- 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)

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

Sequentially consistent but not strictly consistent.

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

Sequential Consistency: Quiz

- 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)

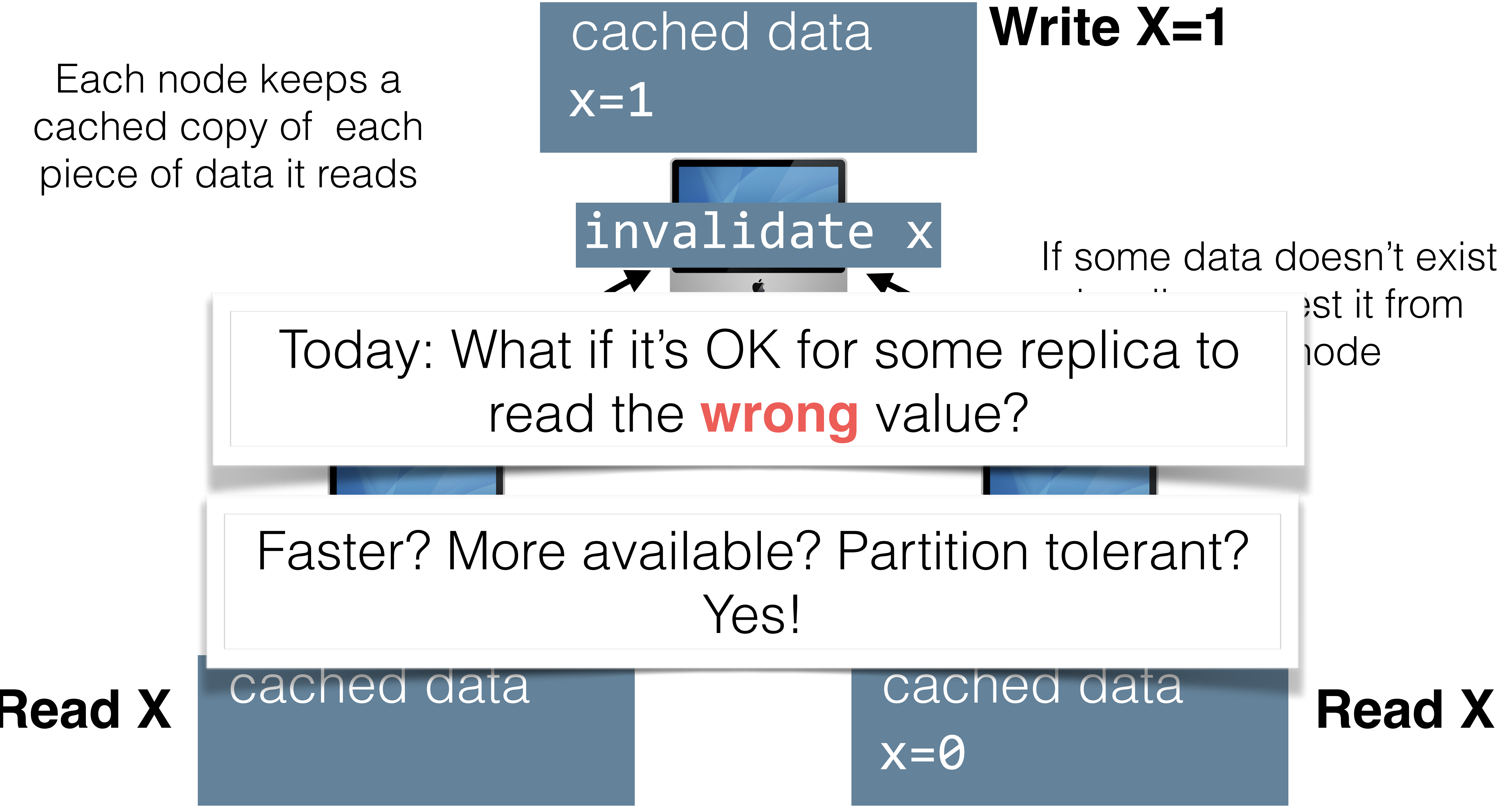
P1	W(X) a				
P2		W(X) b			
P3			R(X) b		R(X) a
P4				R(X) a	R(X) b

Not sequentially consistent

Sequential Consistency: Activity

- http://b.socrative.com/ click on student login, then SWE622 as room name

Review: Ivy Architecture



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

Write X=1

cached data

`x=1`

`invalidate x`

If some data doesn't exist
est it from
ode

Today: What if it's OK for some replica to read the **wrong** value?

Faster? More available? Partition tolerant?
Yes!

Read X

cached data

Read X

cached data

`x=0`

Today

- Are relaxed consistency models easier to fit in CAP?
    - Causal consistency
    - Eventual consistency
- File synchronization
    - Disconnected synchronization
- HW 3

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

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;
= 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
- Recall: Lamport clocks

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.

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

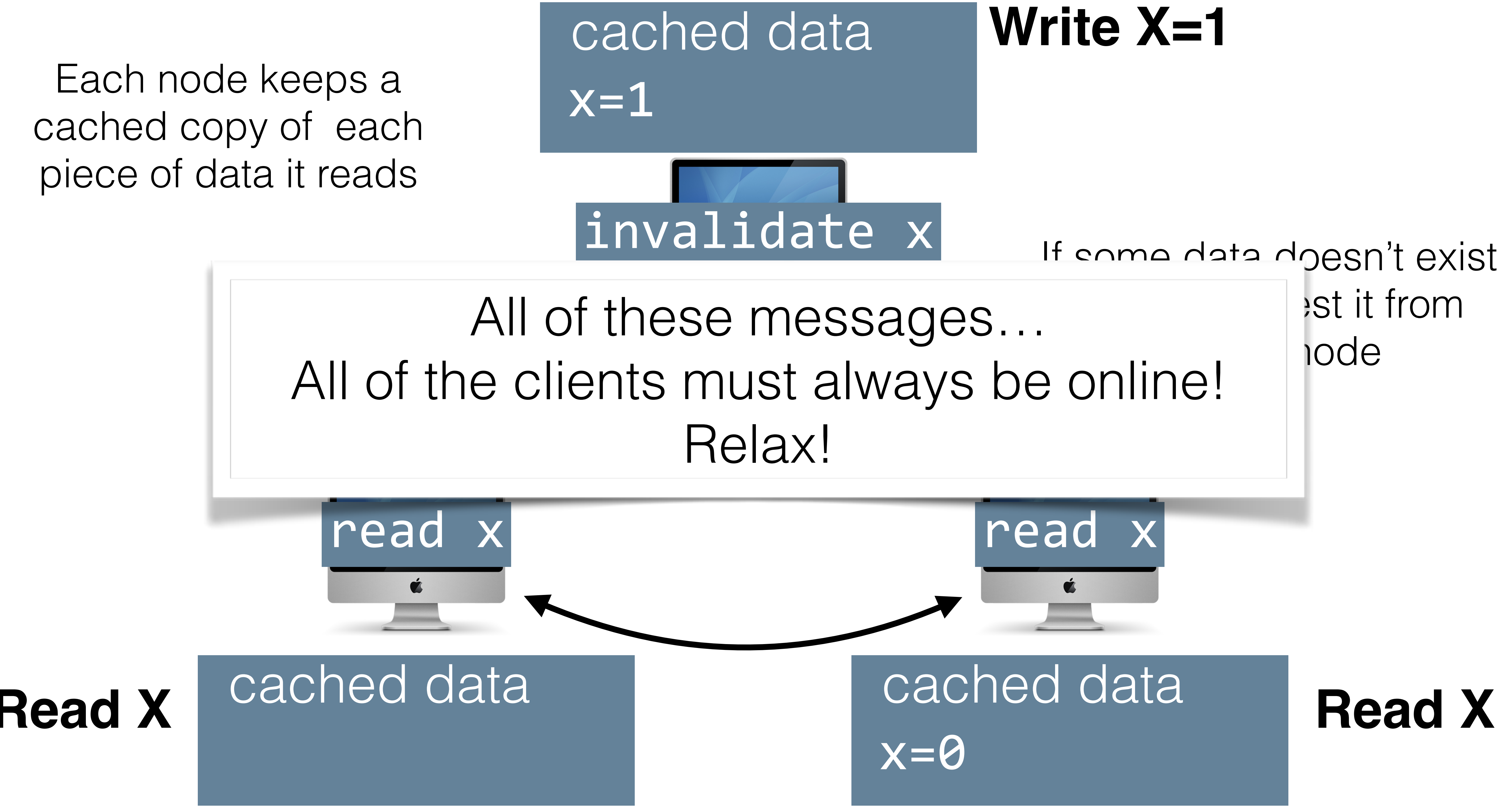
Eventual Consistency

- 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

cached data

x=1

Write X=1

invalidate x

If some data doesn't exist
est it from
node

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

read x

read x

Read X

cached data

cached data

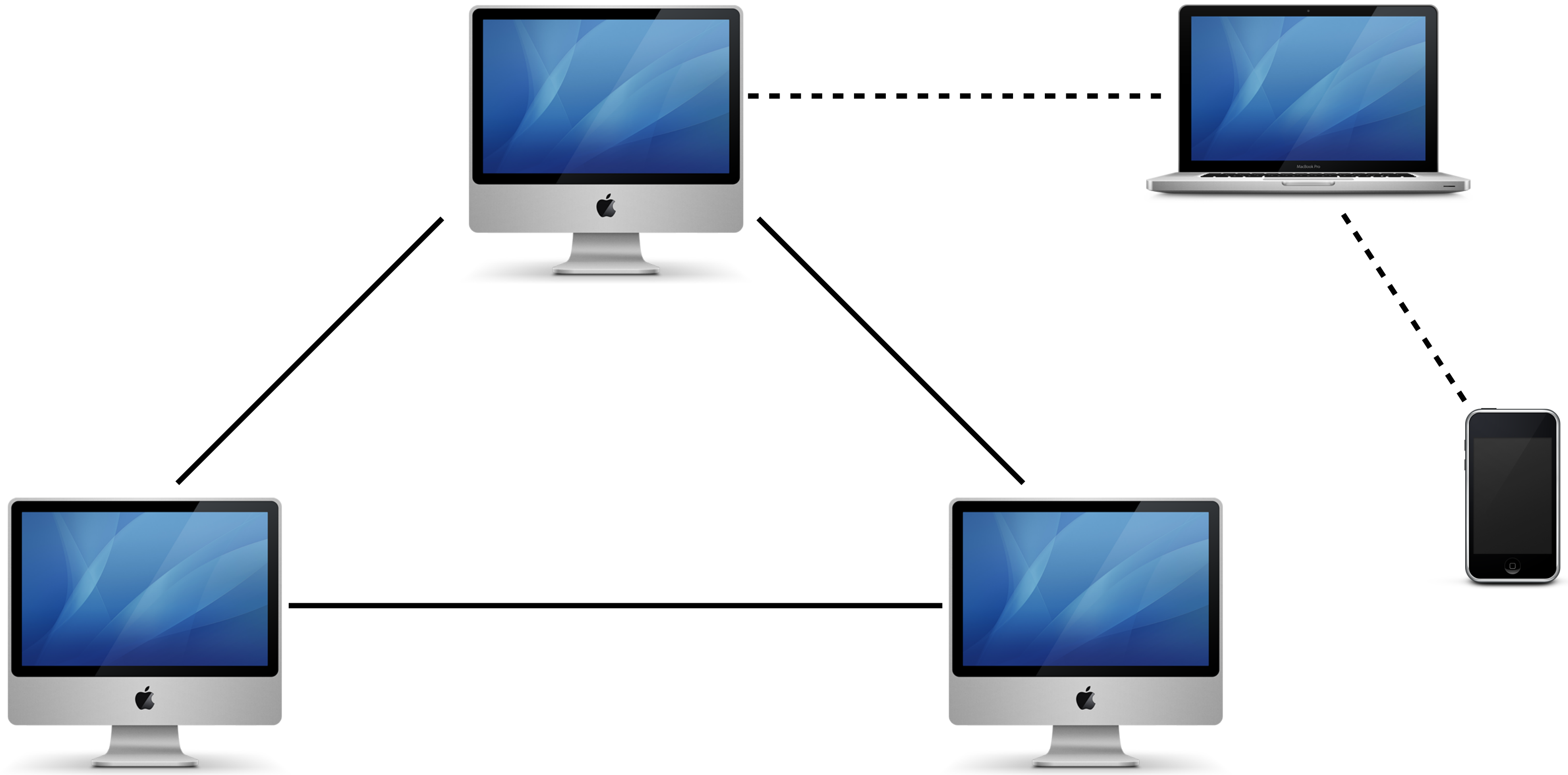
x=0

Read X

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

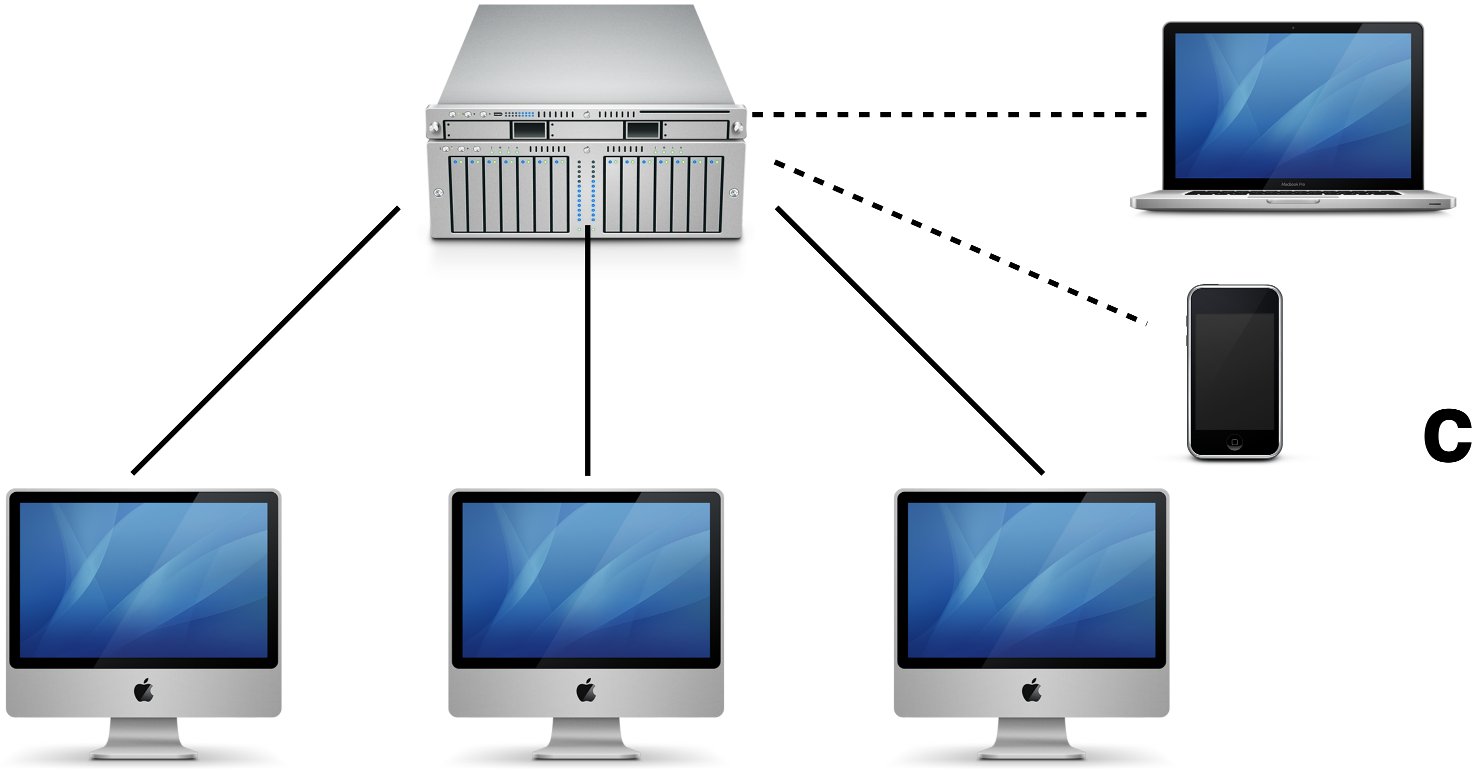
Eventual Consistency: Distributed Filesystem



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?

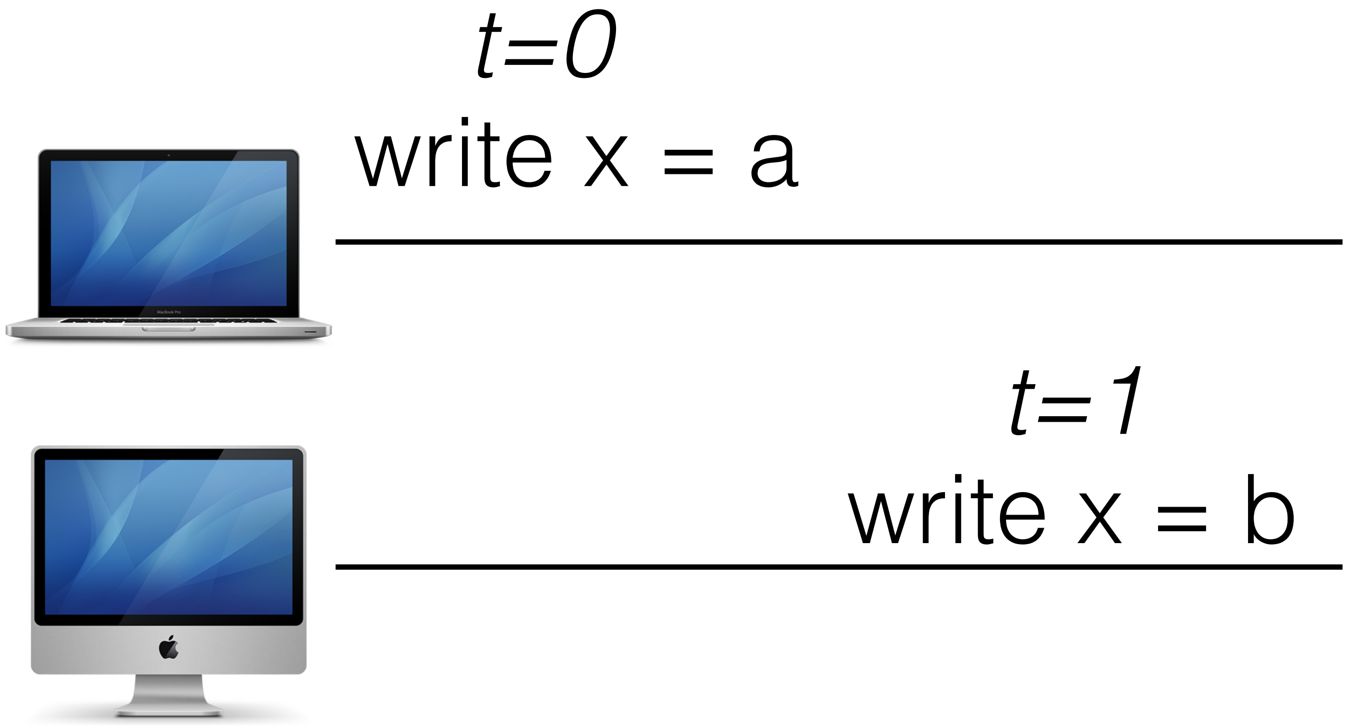
Goal: Everything eventually becomes synchronized.
No lost updates (don't replace new version with old)

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?



t=0

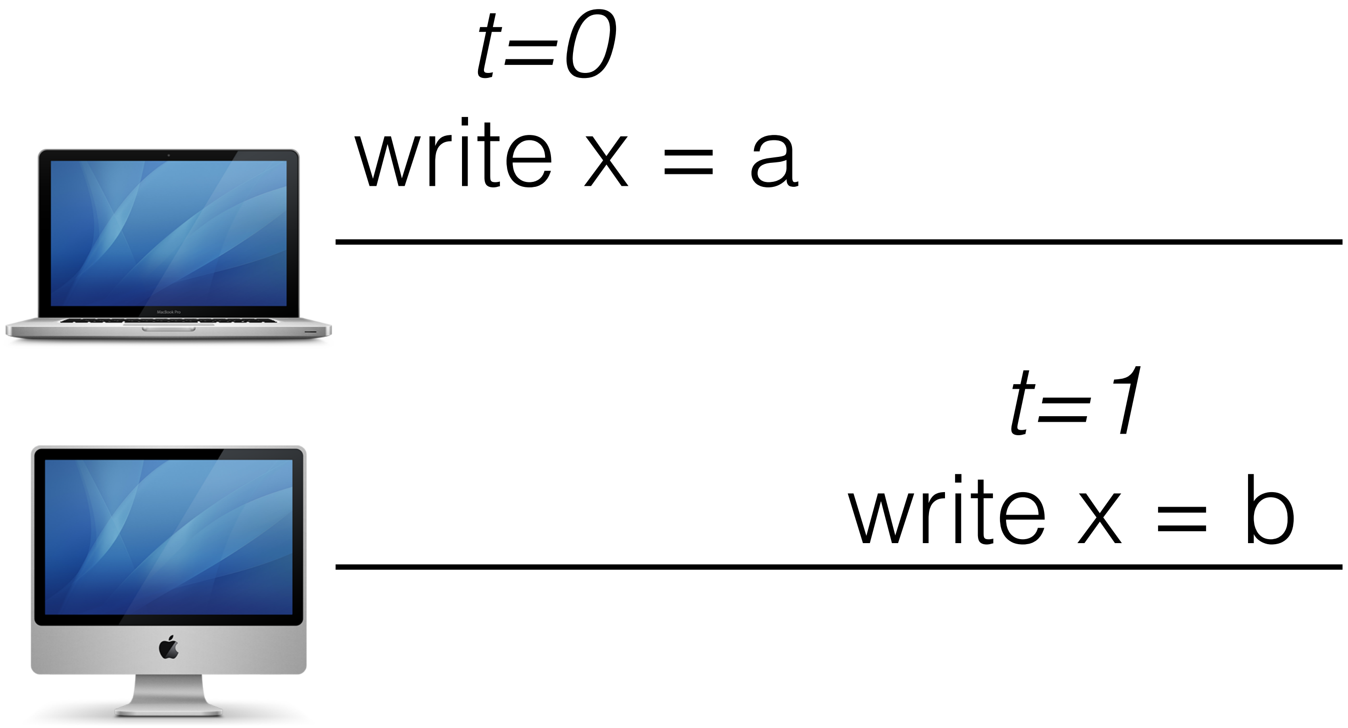
write x = a

t=1

write x = b

Detecting Conflicts

Do we just use timestamps?



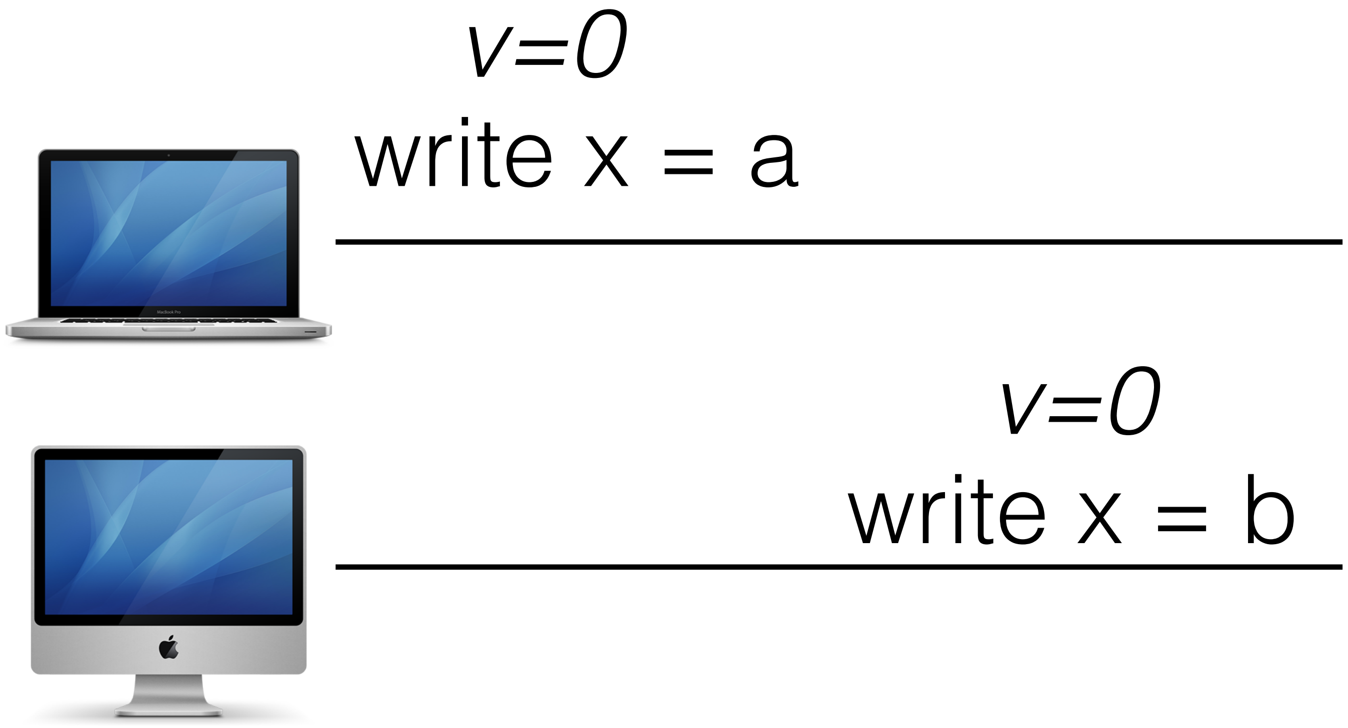
t=0
write x = a

t=1
write x = b

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

Detecting Conflicts

Solution: Track version history on clients



v=0

write x = a

v=0

write x = b

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

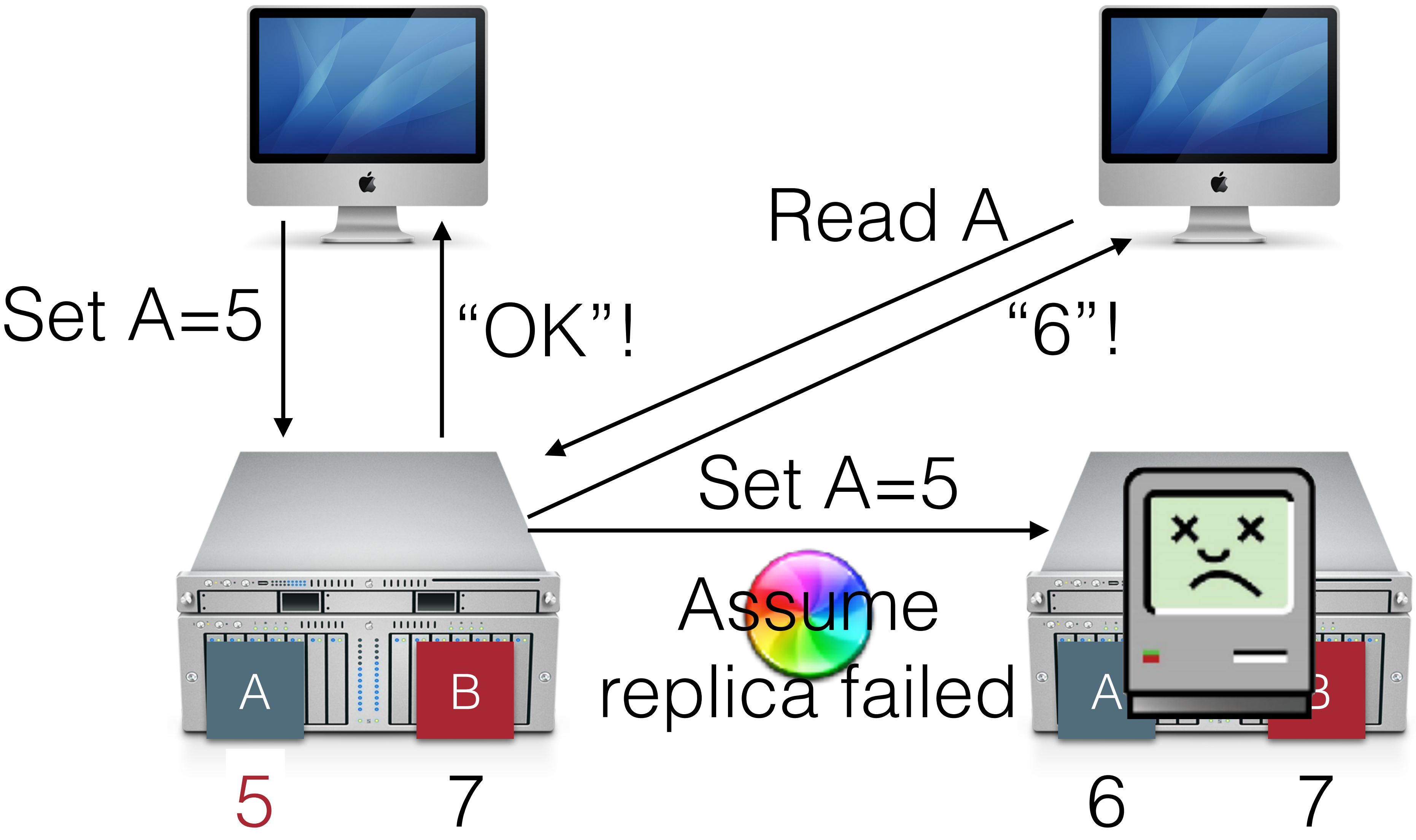
Client-Centric Consistency

- 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

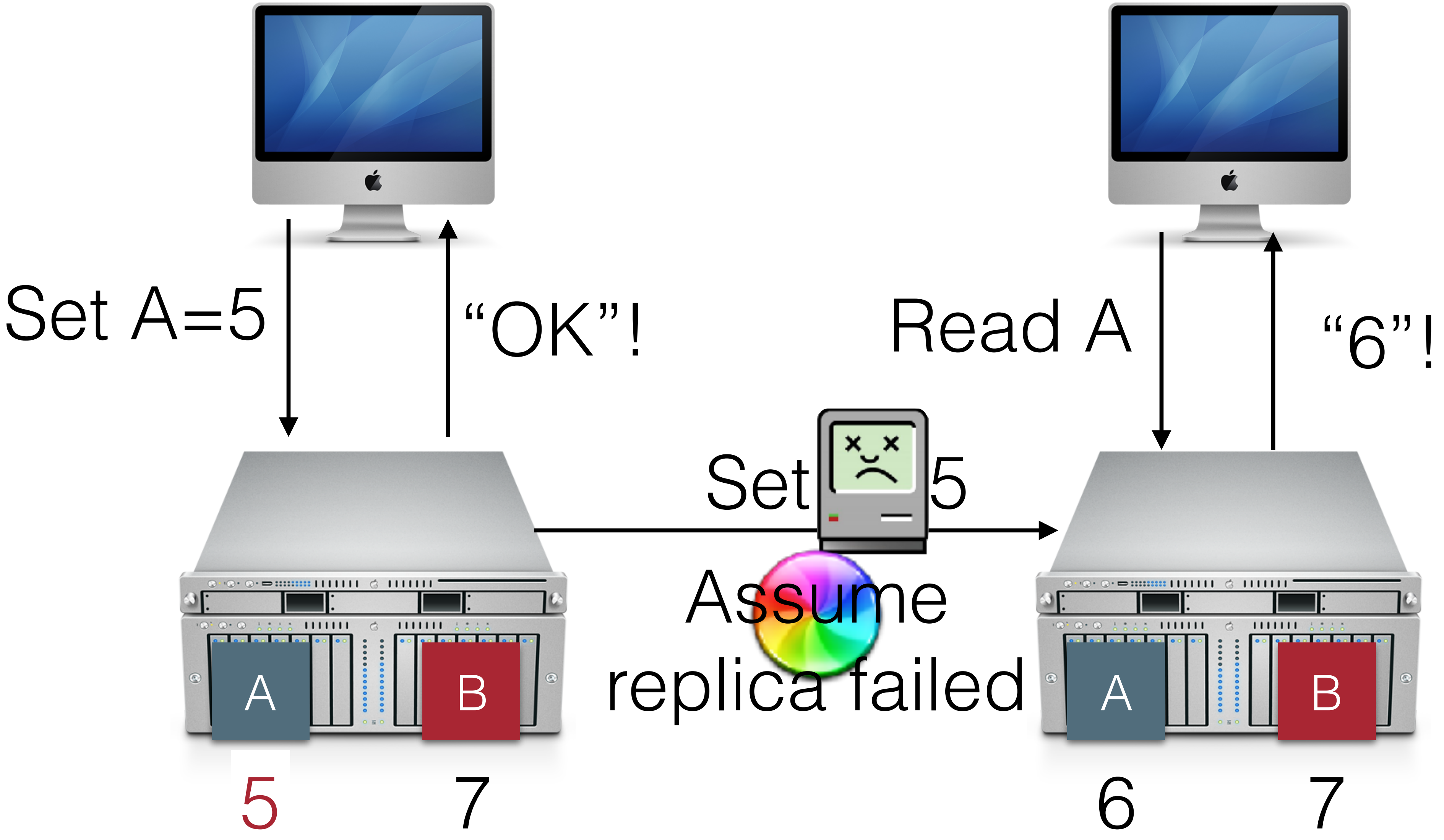
Choosing a consistency model

- 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

Sequentially Consistent + Available



Sequentially Consistent + Not Partition Tolerant



Set A=5

"OK"!

Read A

"6"!

Set 5

Assume replica failed

A

B

5

7

A

B

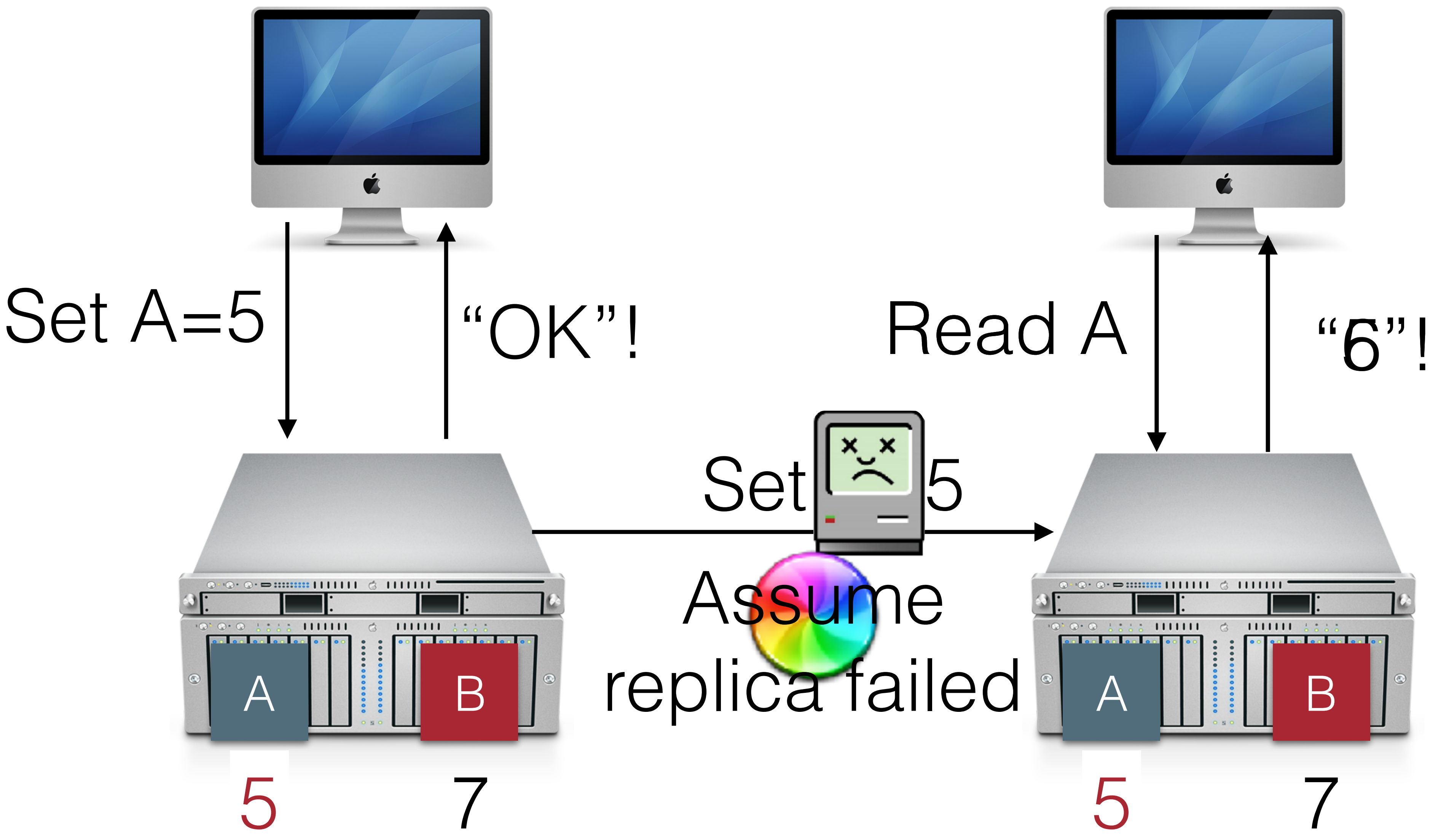
6

7

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…

Eventually Consistent + Available + Partition Tolerant



Set A=5

"OK"!

Read A

"6"!

Set 5

Assume replica failed

A B

5 7

A B

5 7

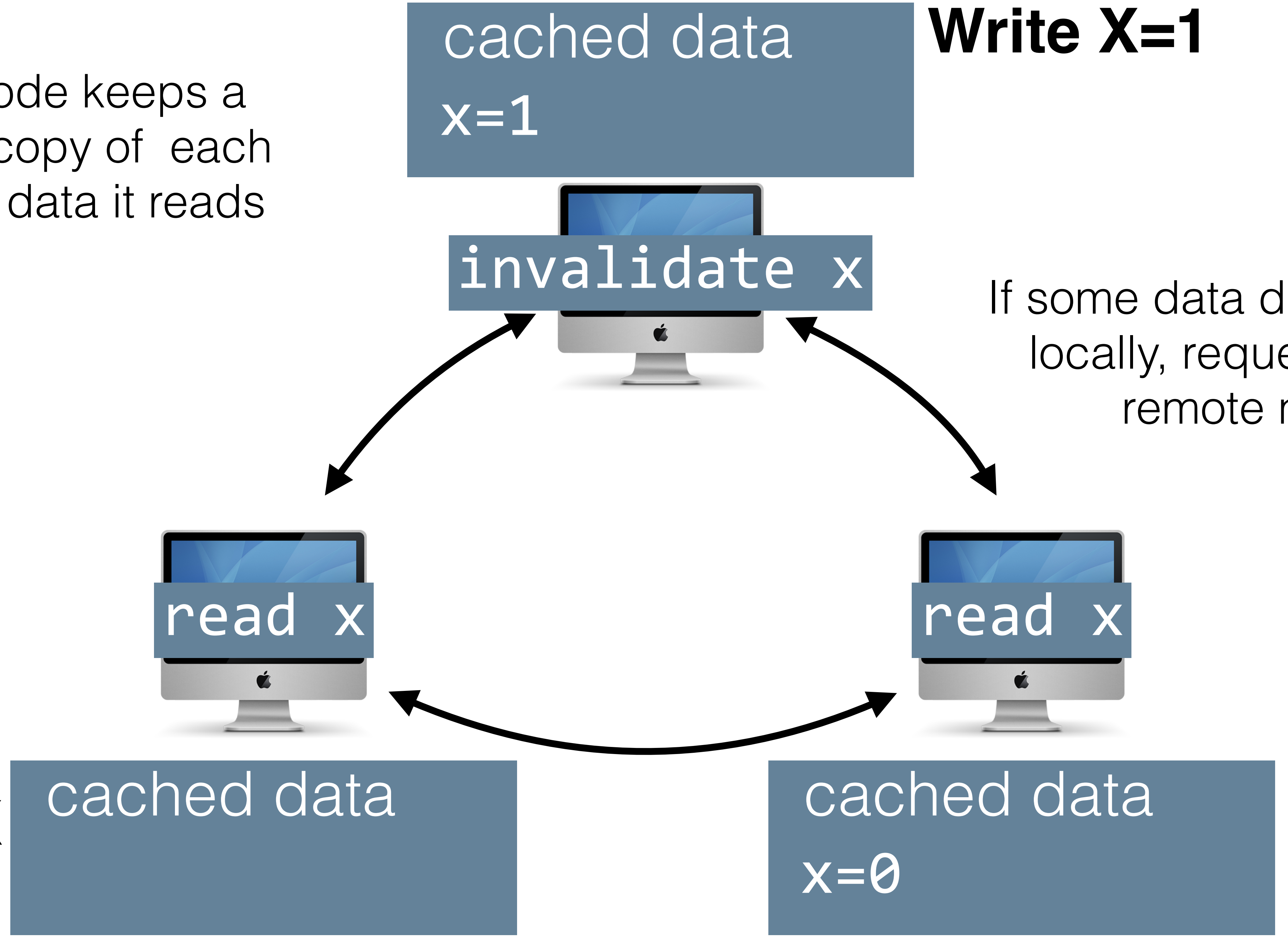
Filesystem consistency

- 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.

Ivy Architecture



cached data

x=1

Write X=1

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

invalidate x

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

read x

read x

Read X

cached data

cached data

x=0

Read X

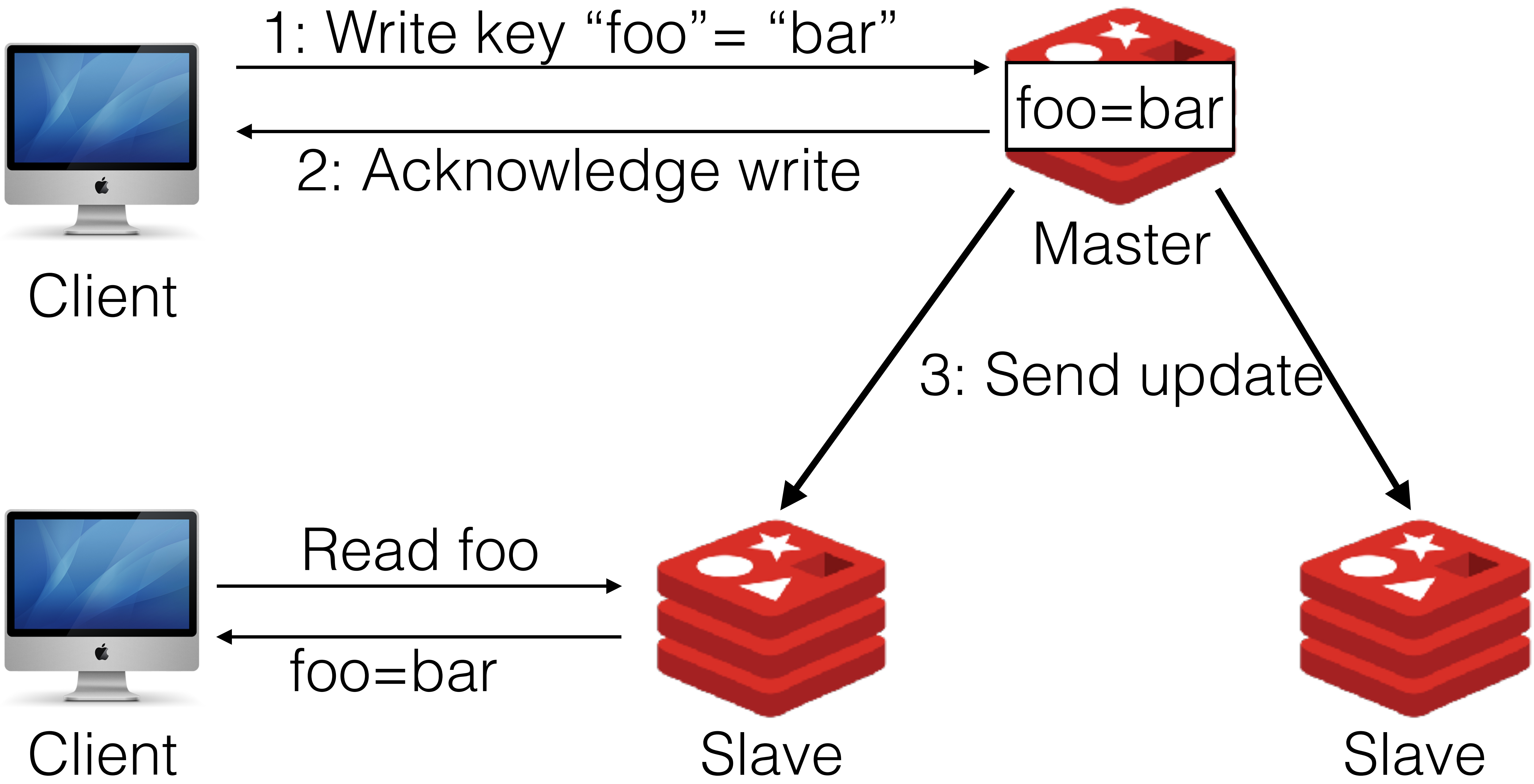
Open-Close Semantics

- This is what AFS (CMU’s internal filesystem) does
- Simplify it. When you open a file, you get a copy.
- When you close it, you commit your changes.
- Allows for stale reads
- But, maybe it’s OK?

Redis Replication

- Redis supports master-slave replication
- What consistency guarantees does it provide and how does it work?
- Replication is **asynchronous**
- Only the master receives writes, slaves are read only
  - Reduces coordination, useful for read-heavy workloads

Redis Replication



1: Write key “foo”= “bar”

foo=bar

2: Acknowledge write

Master

Client

3: Send update

Read foo

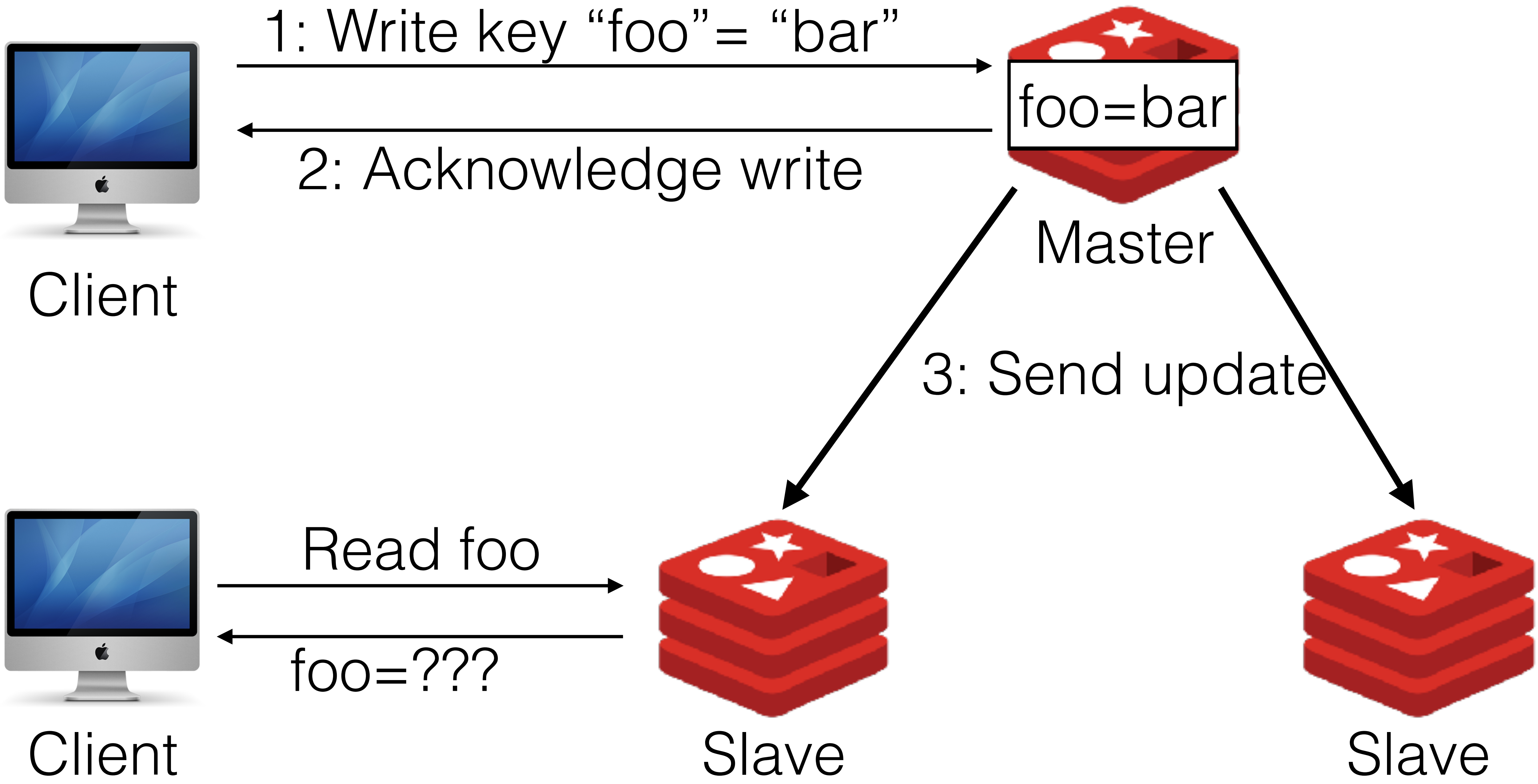
foo=bar

Client

Slave

Slave

Redis Replication



1: Write key “foo”= “bar”

foo=bar

2: Acknowledge write

Master

Client

3: Send update

Read foo

foo=???

Client

Slave

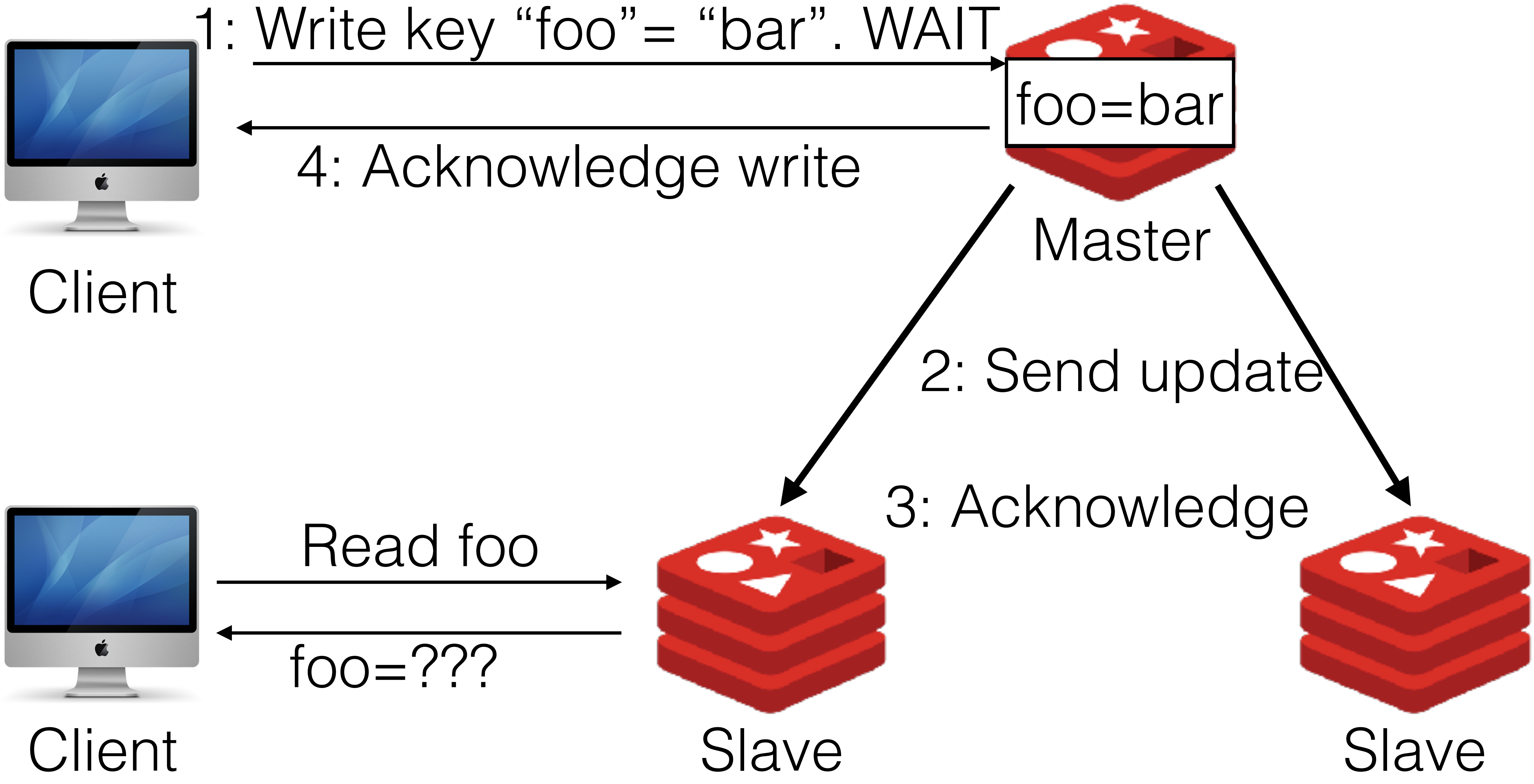
Slave

Redis Replication

	t=0	t=1
Client 1	Write Foo	
Client 2		Read Foo

Nothing tells us/Redis that Client 2 must see Client 1's update (consider this like eventual consistency - eventually client 2 will see it, but it might see a stale one)

Redis “Wait” command



1: Write key “foo”= “bar”. WAIT

foo=bar

4: Acknowledge write

Master

Client

2: Send update

3: Acknowledge

Read foo

foo=???

Client

Slave

Slave

Redis “Wait” command

What does WAIT give us?

	t=0	t=1
Client 1	Write Foo. WAIT	
Client 2		Read Foo

	t=0	t=1
Client 1		Write Foo. WAIT
Client 2	Read Foo	

Client 2 can still read either. But we *know* which happened.

Redis “Wait” command

One more confusing bit:

Is Redis replication + WAIT going to give us sequential consistency?

NO: WAIT only guarantees that the replica *received* the update, *not* that it processed and committed it.

But, it’s probably better than nothing.

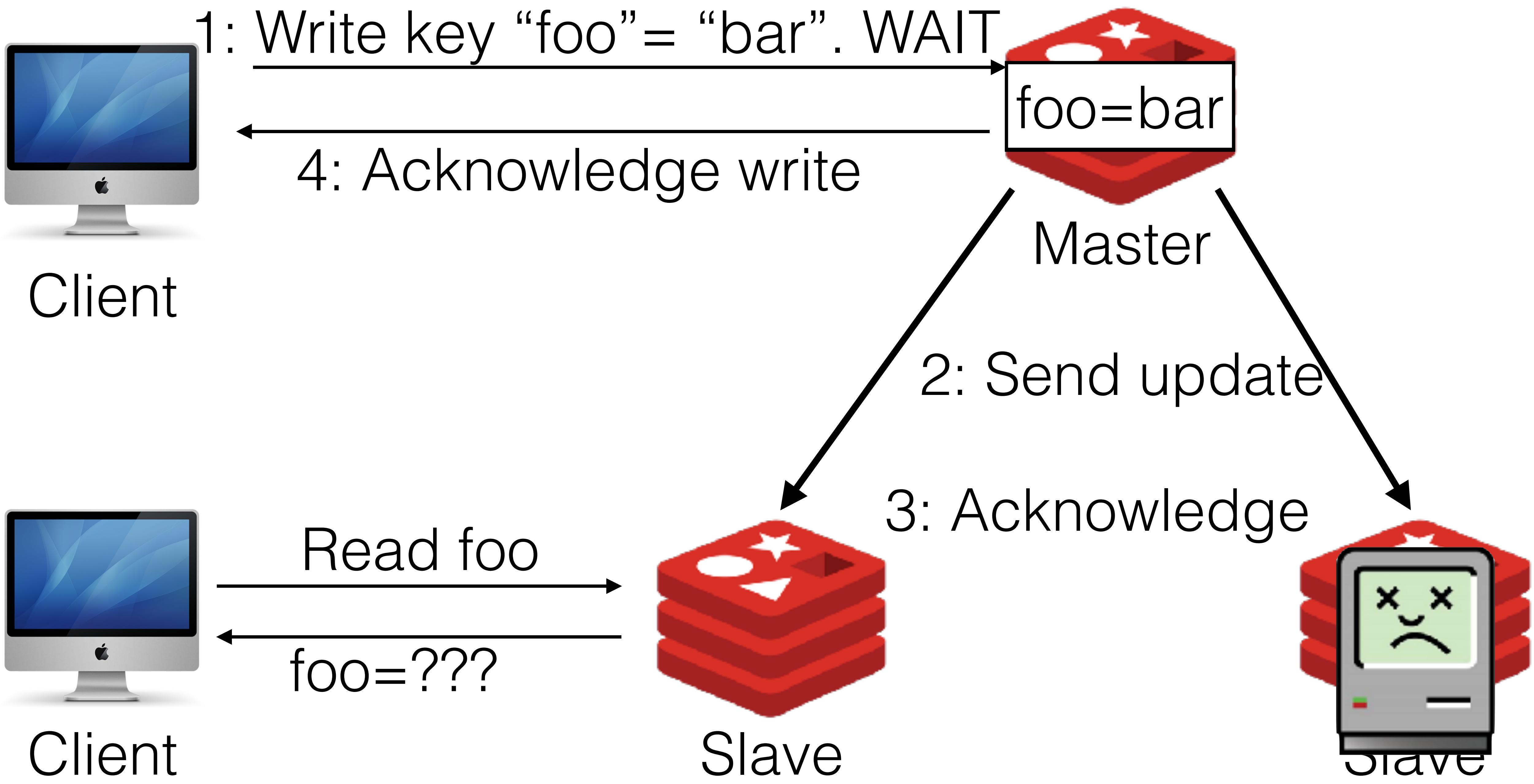
HW3: Replicate Redis

- What happens if Redis fails?
- Solution:
  - Redis has built in replication!
- What consistency guarantees does that provide?
- We want to maintain what we've got.
- You'll use WAIT after writes
- All writes -> master, reads -> slave (note: now each client has its own redis slave)
- Add heartbeat to know how many replicas there are

Lab: Heartbeats

- Goal: make sure all replicas have received most recent update
- Problem: how many replicas are there? How many are online?

Redis “Wait” command



1: Write key “foo”= “bar”. WAIT

foo=bar

4: Acknowledge write

Master

Client

2: Send update

3: Acknowledge

Read foo

foo=???

Client

Slave

Slave

Lab: Heartbeats

- Your task: implement a simple protocol: a server that will keep track of the number of live replicas at any time
- High level question: What’s a “live” replica?
  - Is able to send a message saying its alive?
  - Has seen most recent updates?
  - Has seen most recent updates and processed them?
- For HW3 and this lab: enough to just say the client is “alive”