CS 4530 Software Engineering

Lecture 9.2: Strategies for Engineering Distributed Software

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Lesson

By the end of this lesson, you should be able to…

- Describe partitioning and replication as building blocks for distributed systems
- Evaluate the trade-offs between consistency and availability in distributed systems
- Answer the question: how does partitioning and replication help us satisfy requirements for distributed systems?

Recap: Why expand to distributed systems?

- Scalability
- Performance
- Latency
- Availability
- Fault Tolerance

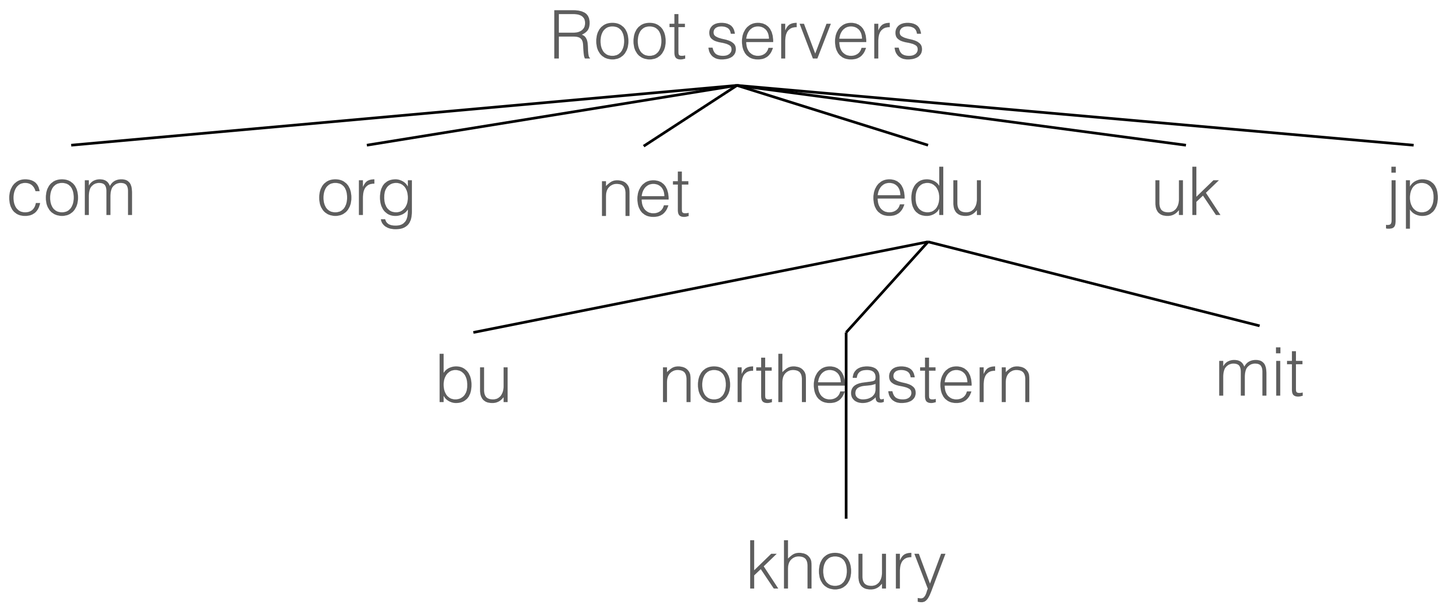
“Distributed Systems for Fun and Profit”, Takada

How do we organize our distributed system?

- This depends to a large degree on whether there is shared state
- Usually, there is some shared state
- How important is it to synchronize?
- What about our DNS example?
  - Domains can be split (e.g., .com, .edu, .info, .eu, .jp, …)
  - Huge volume of requests – multiple nodes need to provide the same mappings, consistently

How to organize DNS

Idea: break apart responsibility for each part of a domain name (**zone**) to a different group of servers



Each zone is a continuous section of name space
Each zone has an associate set of name servers

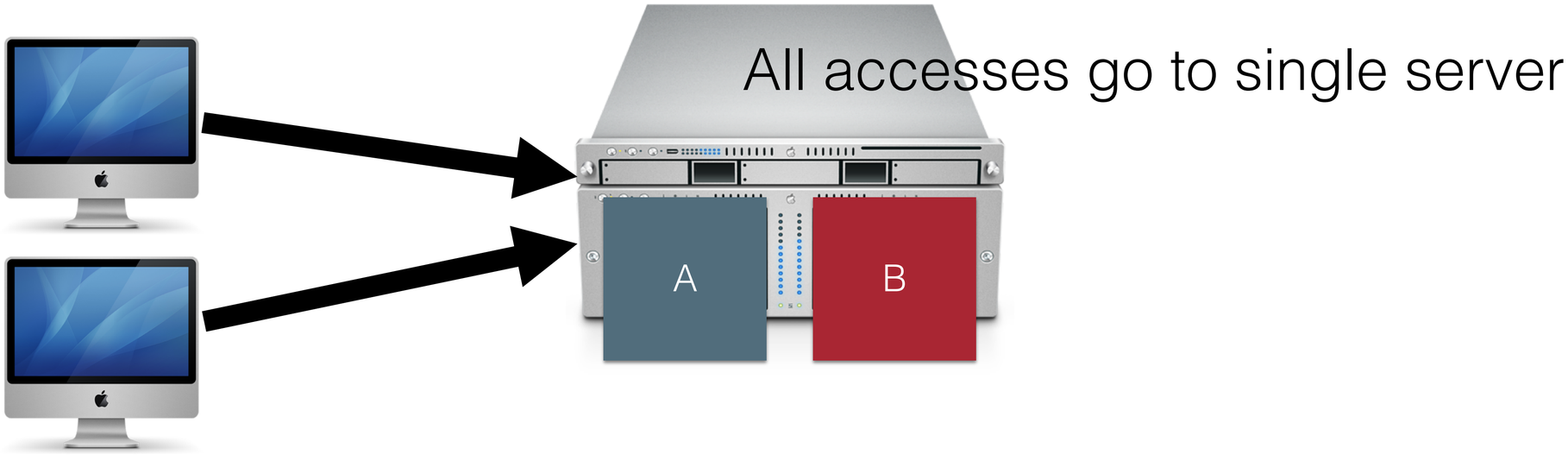
How to organize DNS

Idea: break apart responsibility for each part of a domain name (**zone**) to a different group of servers

In other words, we **partition** the domain names according to the top-level domain.

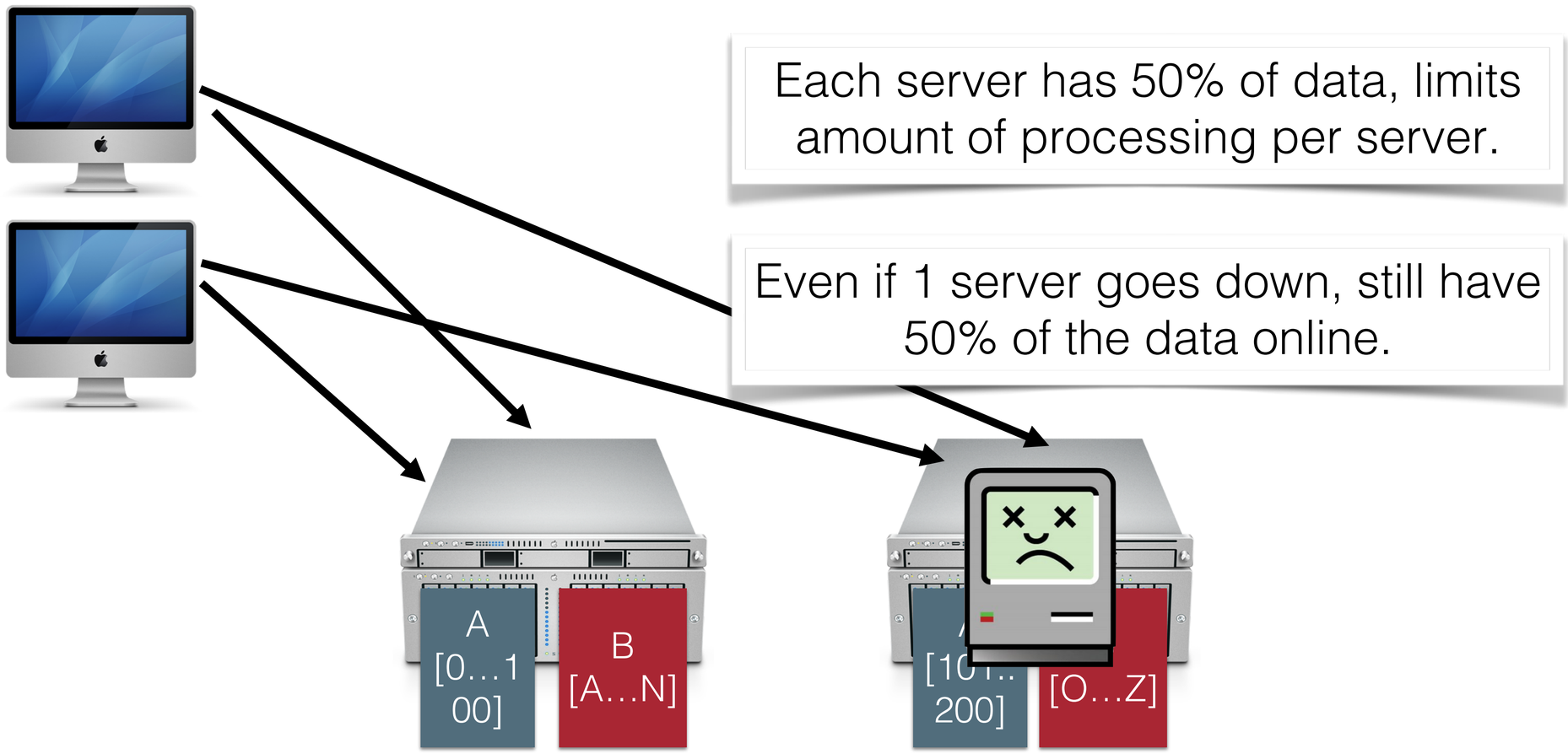
Recurring Solution #1: Partitioning

- Partitioning is a common strategy to distributing a system and its data
- Starting from a non-distributed system:

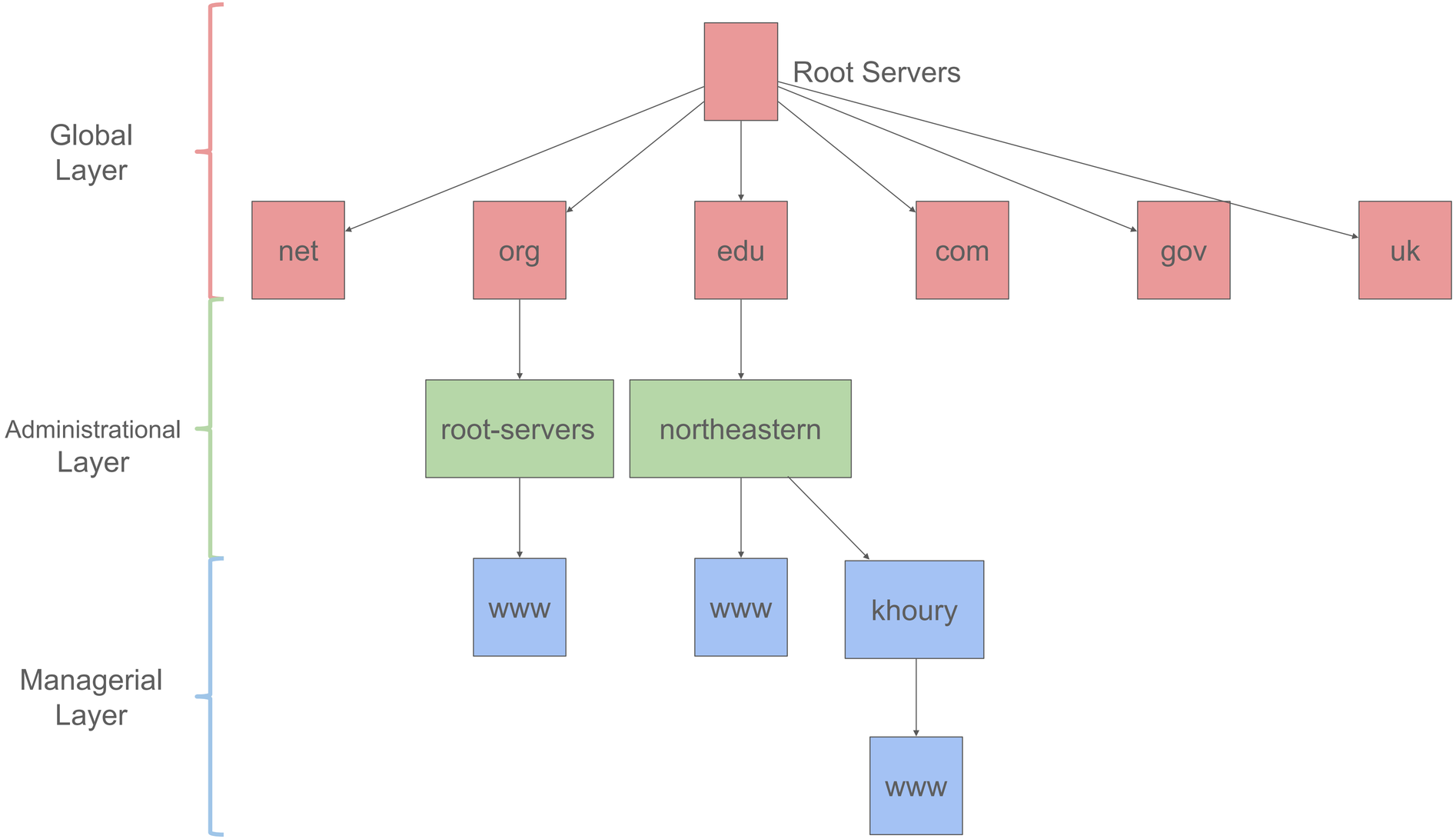


Recurring Solution #1: Partitioning

- Divide data up in some (hopefully logical) way
- Makes it easier to process data concurrently (cheaper reads)

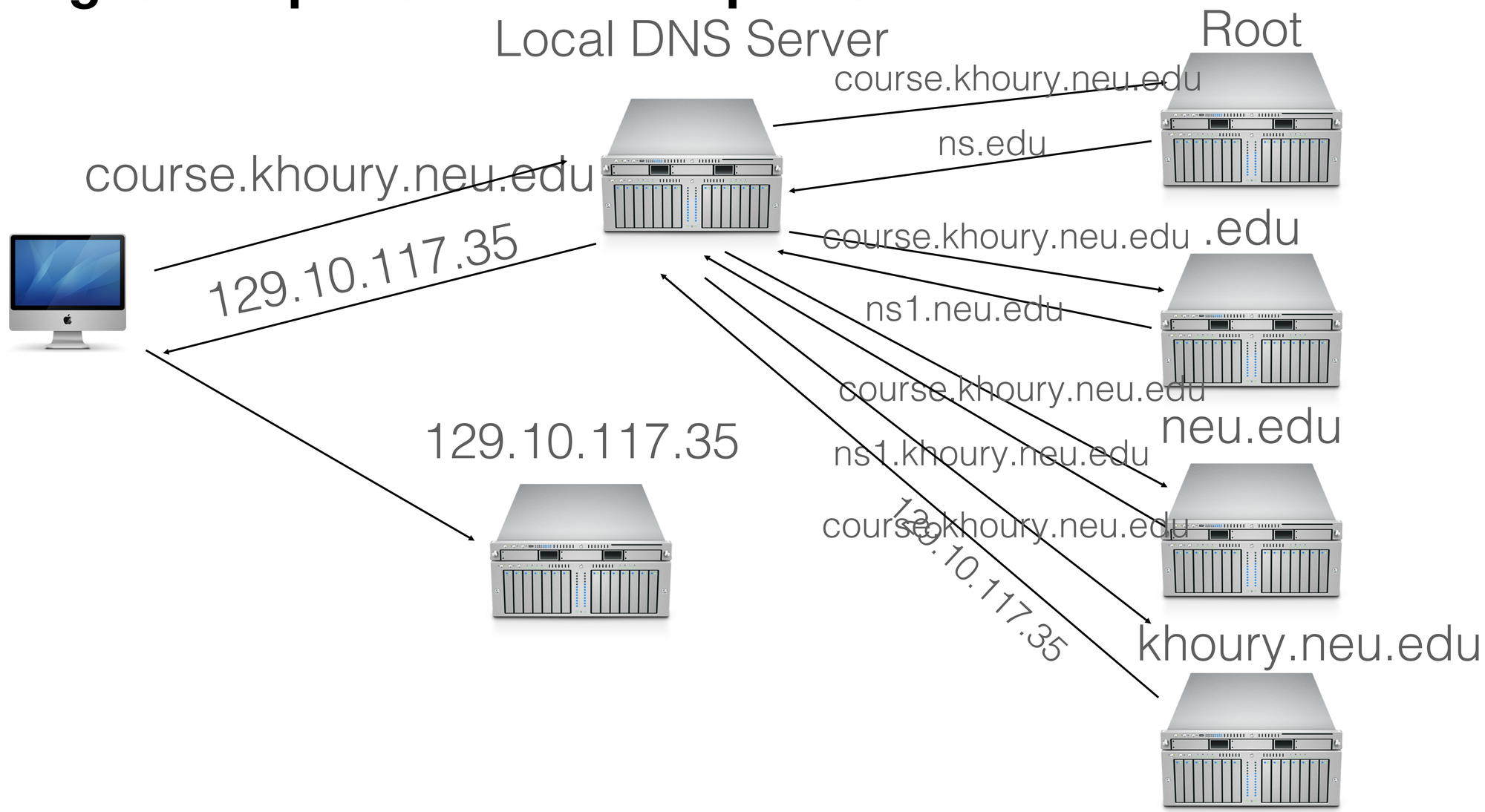


Partitioning DNS



DNS: Example

What might a request look like in practice?

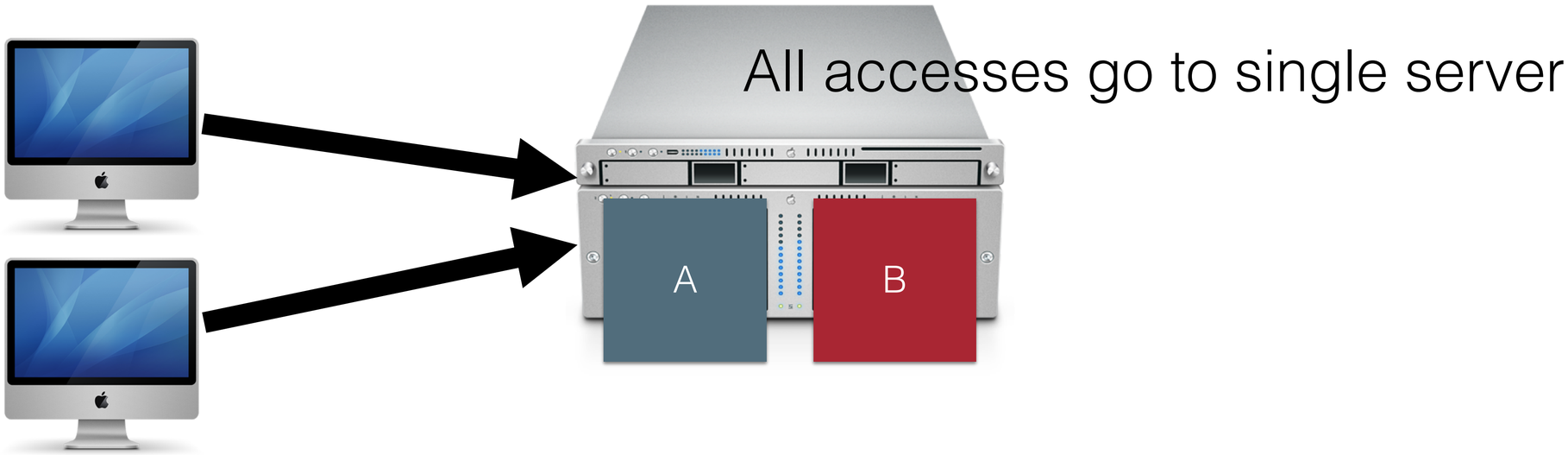


How to deal with volume?

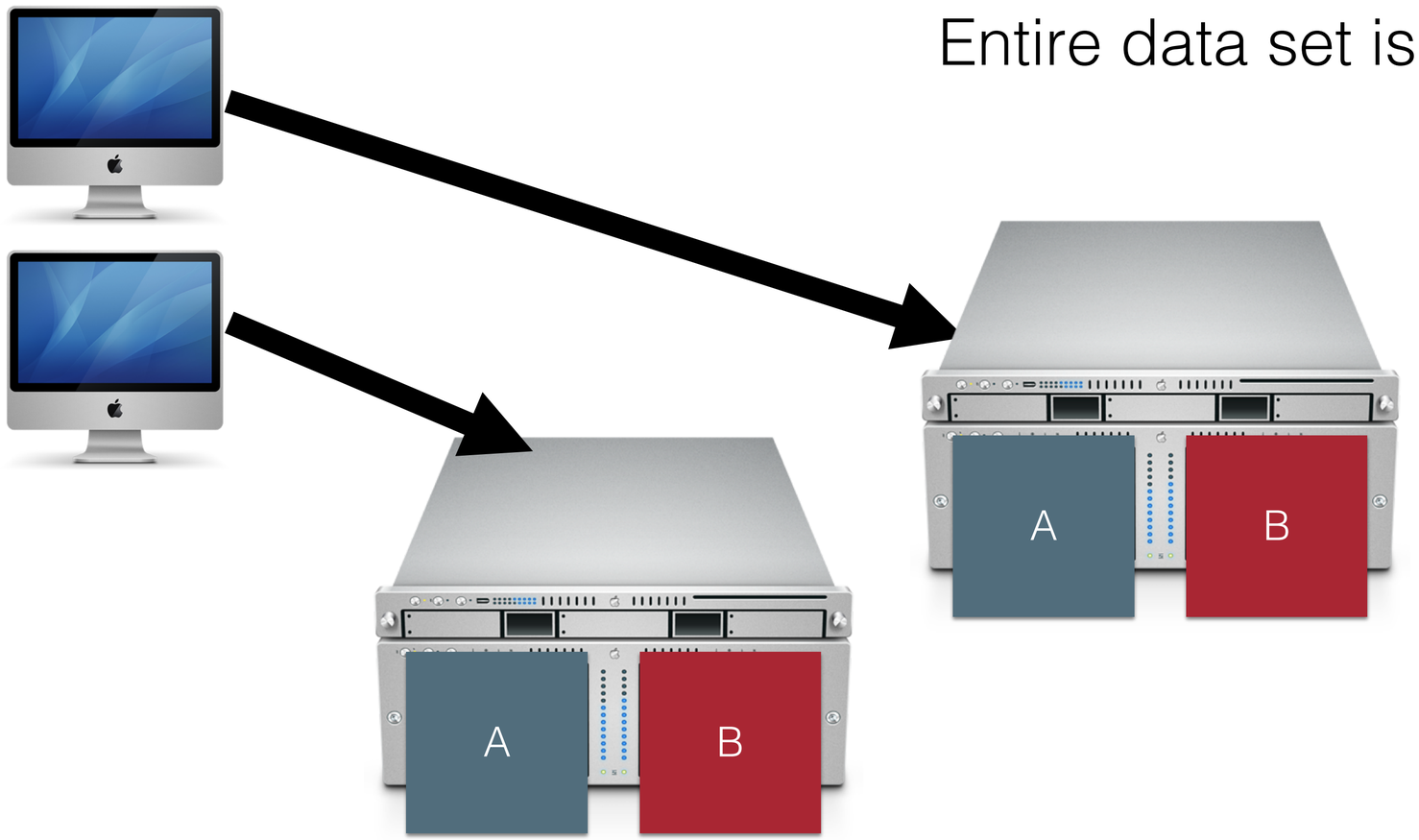
- We successfully distributed requests following the hierarchical nature of domain names
- However, e.g., .com is a very popular TLD – there might be (hundreds of) thousands of requests happening at any given time
- We may need several nodes just servicing .com
- This leads to **replication**

Recurring Solution #2: Replication

- Goal: Any node should be able to process any request
- Again, starting from a non-distributed system:



Recurring Solution #2: Replication



Entire data set is copied

Recurring Solution #2: 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

Replication in DNS – Root Servers

- 13 root servers
  - `[a-m].root-servers.org`
  - E.g., `d.root-servers.org`
- Handled by 12 distinct entities
  - (“a” and “j”) are both Verisign
    - Don’t ask why.

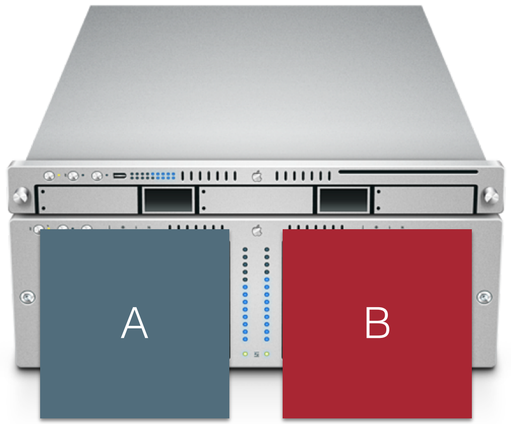
Verisign, Inc.	a
Information Sciences Institute	b
Cogent Communications	c
University of Maryland	d
NASA Ames Research Center	e
Internet Systems Consortium, Inc.	f
U.S. DOD Network Information Center	g
U.S. Army Research Lab	h
Netnod	i
Verisign, Inc.	j
RIPE NCC	k
ICANN	l
WIDE Project	m

There is replication even within the root servers

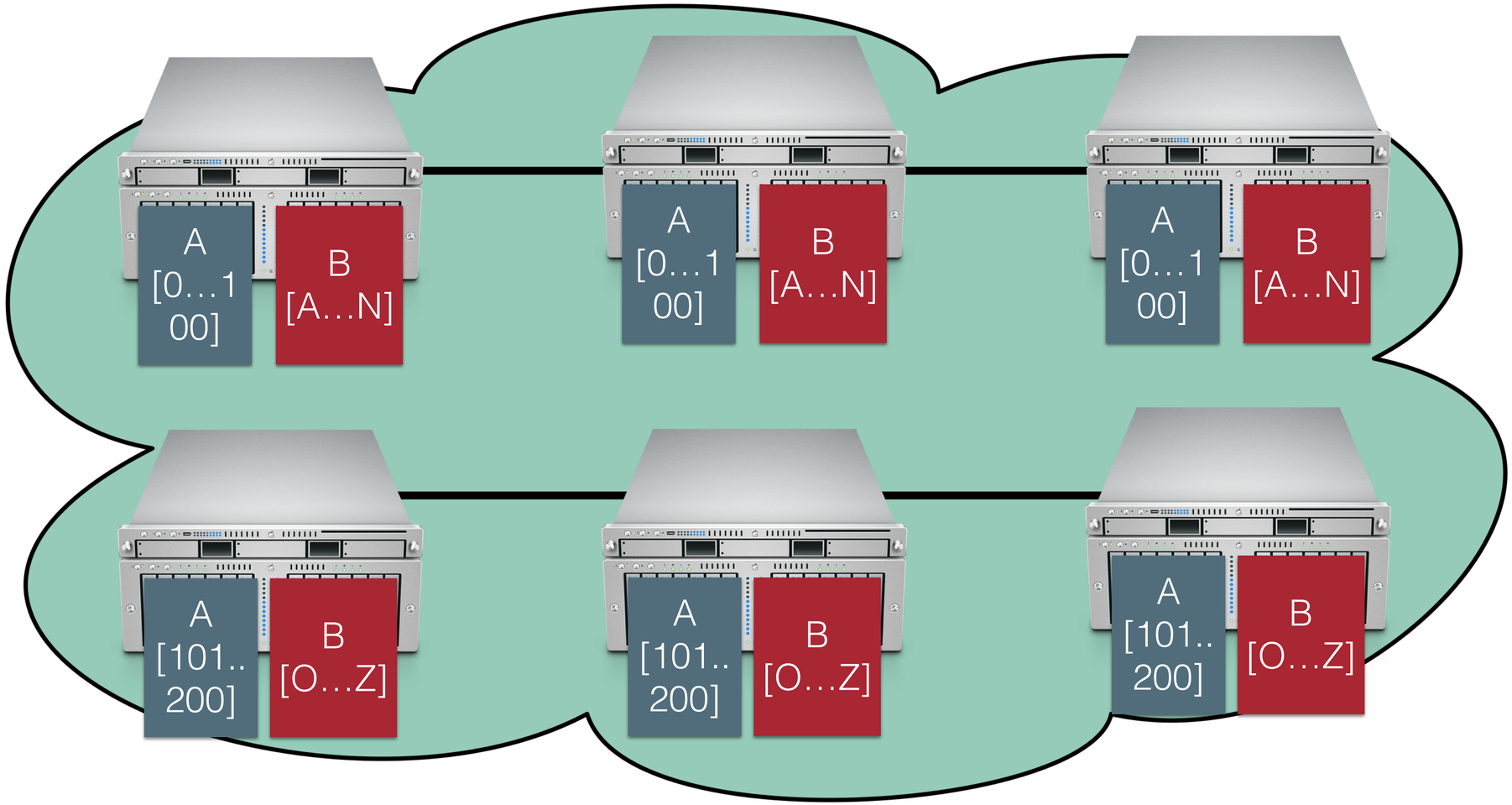
- 13 root servers
  - `[a-m].root-servers.org`
  - E.g., `d.root-servers.org`
- But each root server has multiple copies of the database, which need to be kept in sync.
- Somewhere around 1500 replicas in total.

Partitioning + Replication

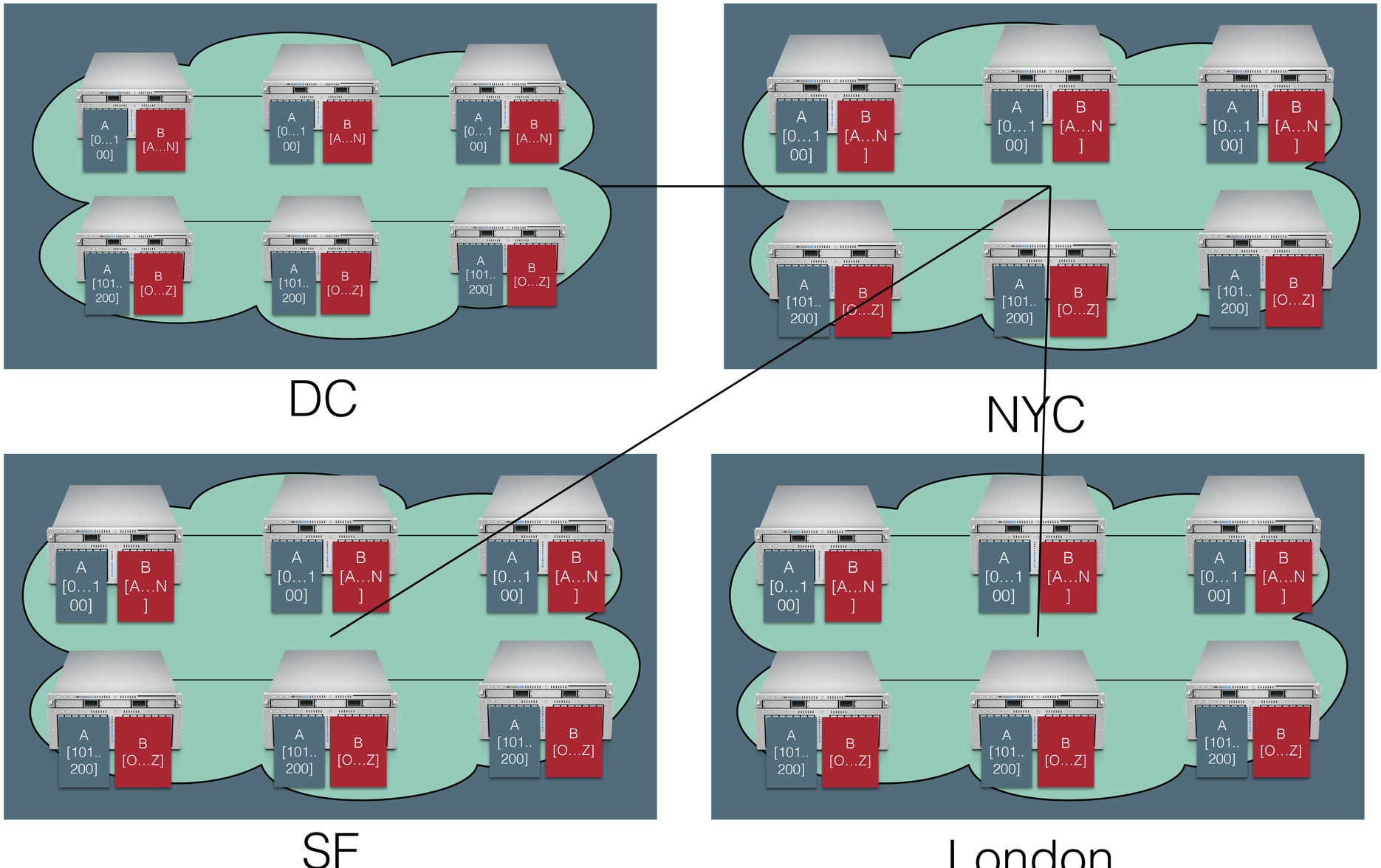
- So, DNS combines both partitioning and replication
- As do most distributed systems



Partitioning + Replication

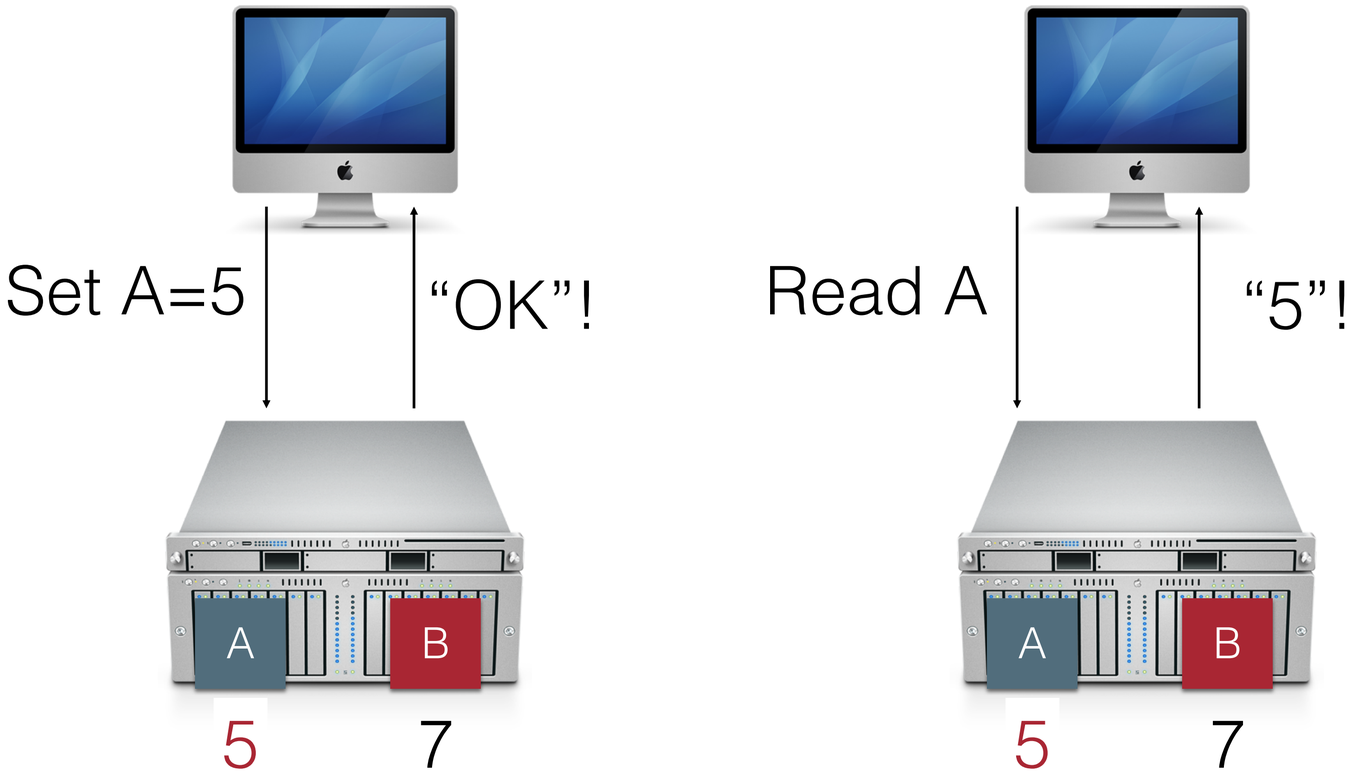


Partitioning + Replication



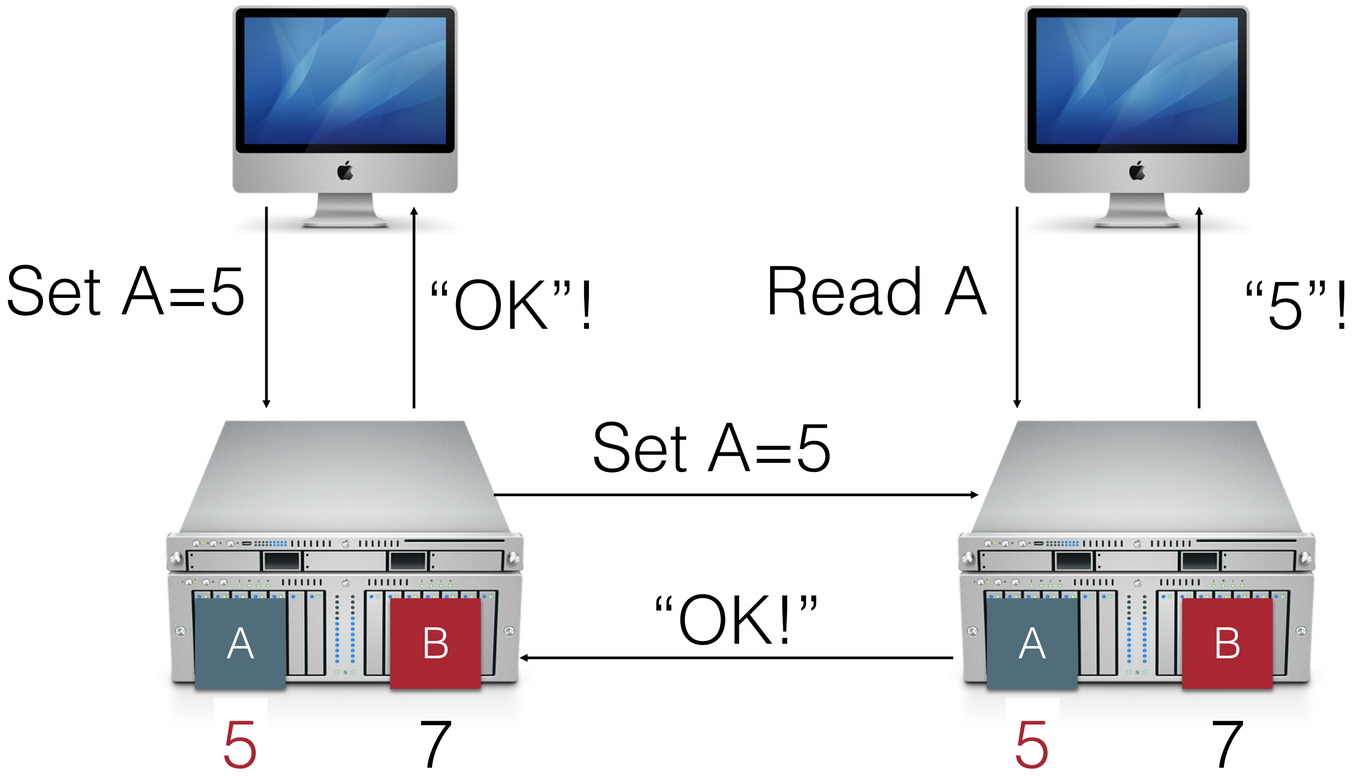
Replication Problem: Consistency

We probably want our system to work like this



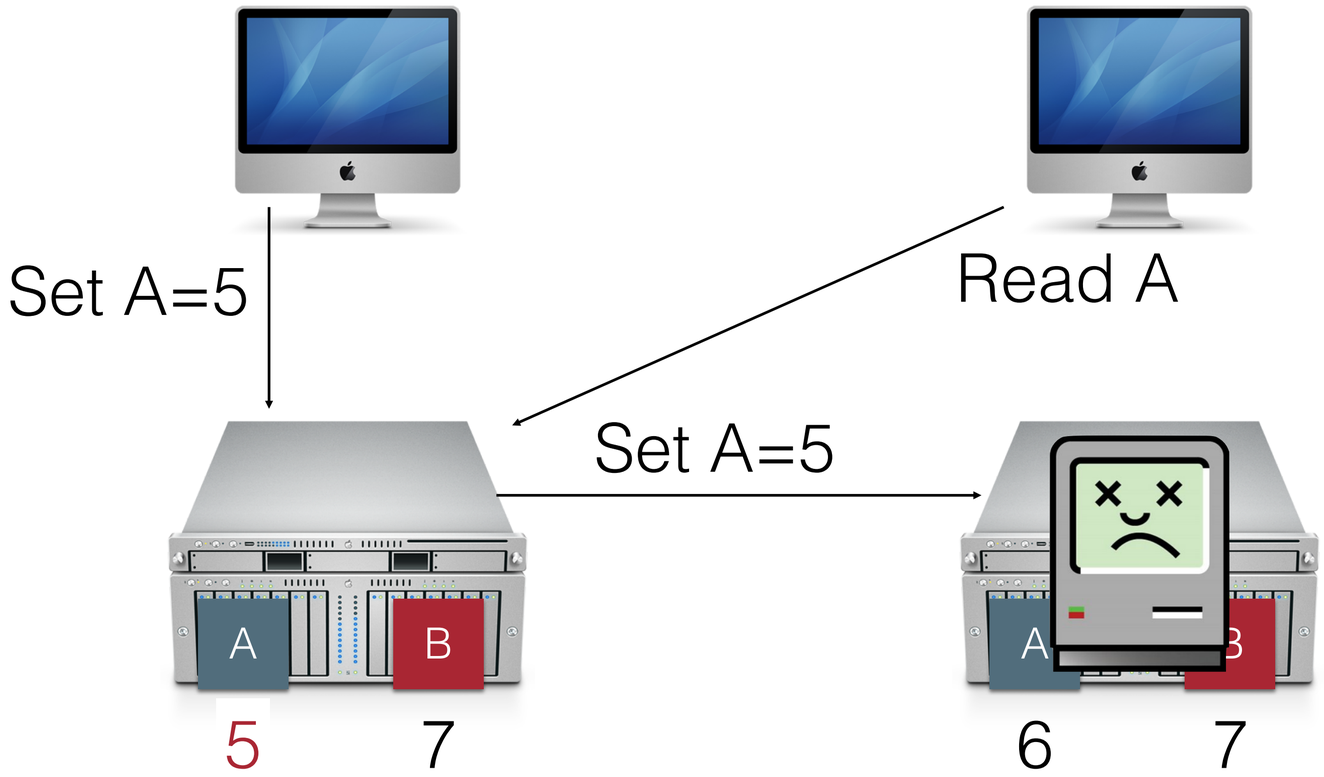
Sequential Consistency

AKA: Behaves like a single machine would



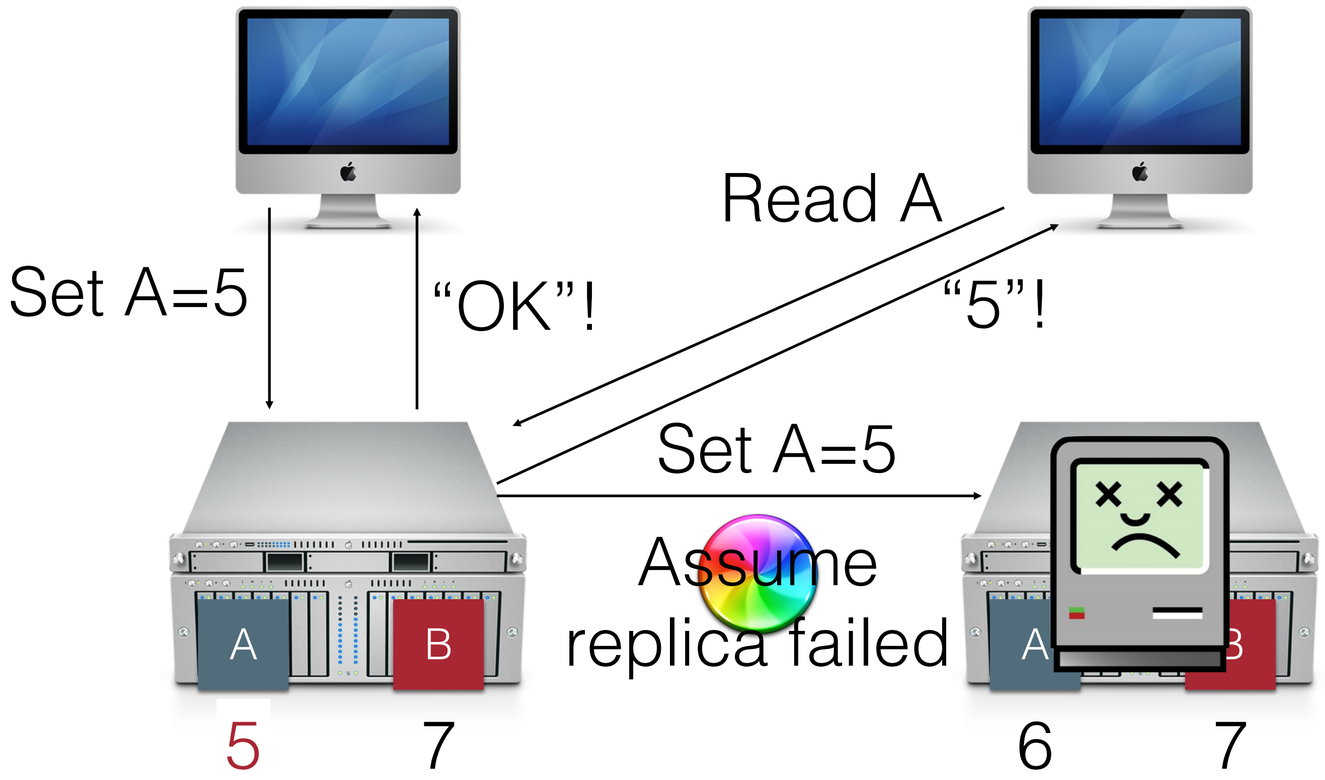
Availability

If at least one node is online, can we still answer a request?

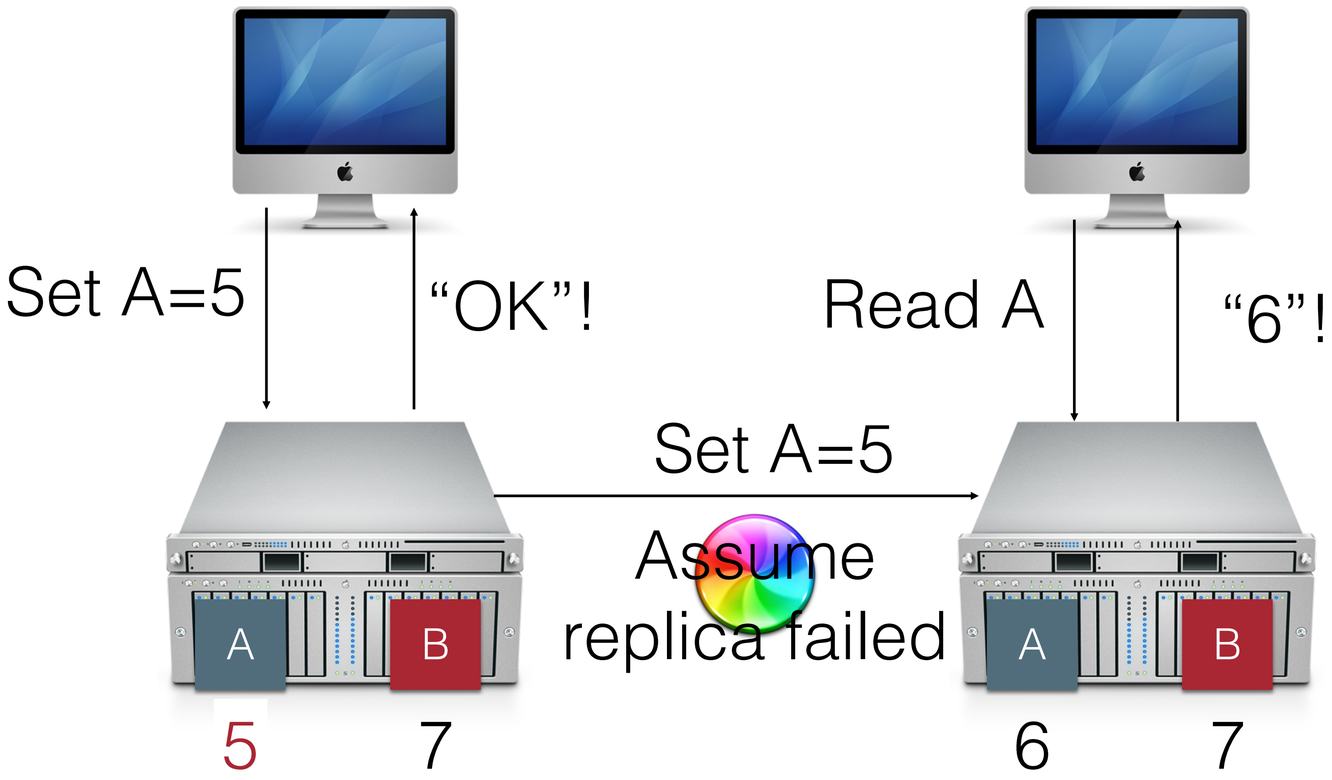


Consistent + Available

On timeout, assume node is crashed



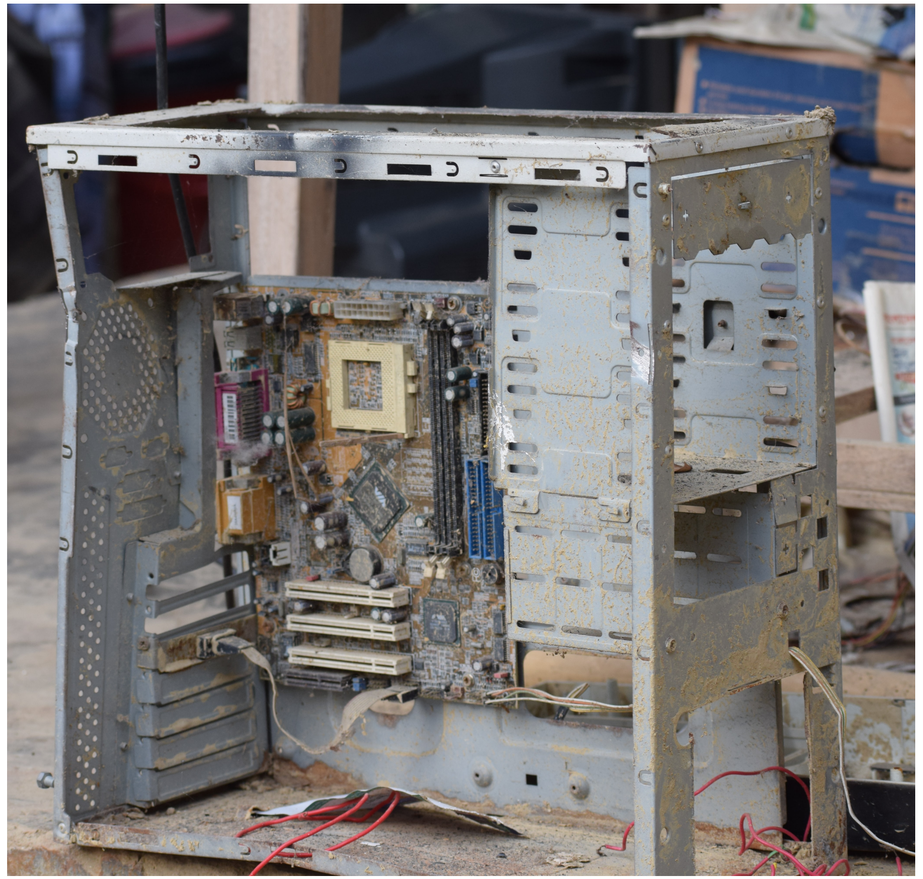
What if *the network* fails?



Shared Fate

Are you still there?

- Two methods/threads/processes running on the same computer generally have **shared fate** [Crashed/not]
- When two machines in a distributed system can’t talk to each other, how do we know if the other is crashed?
- We call this a **split brain** problem



CAP Theorem: Consistency or Availability

- 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)
    - Can’t drop this for a DS - networks can always fail

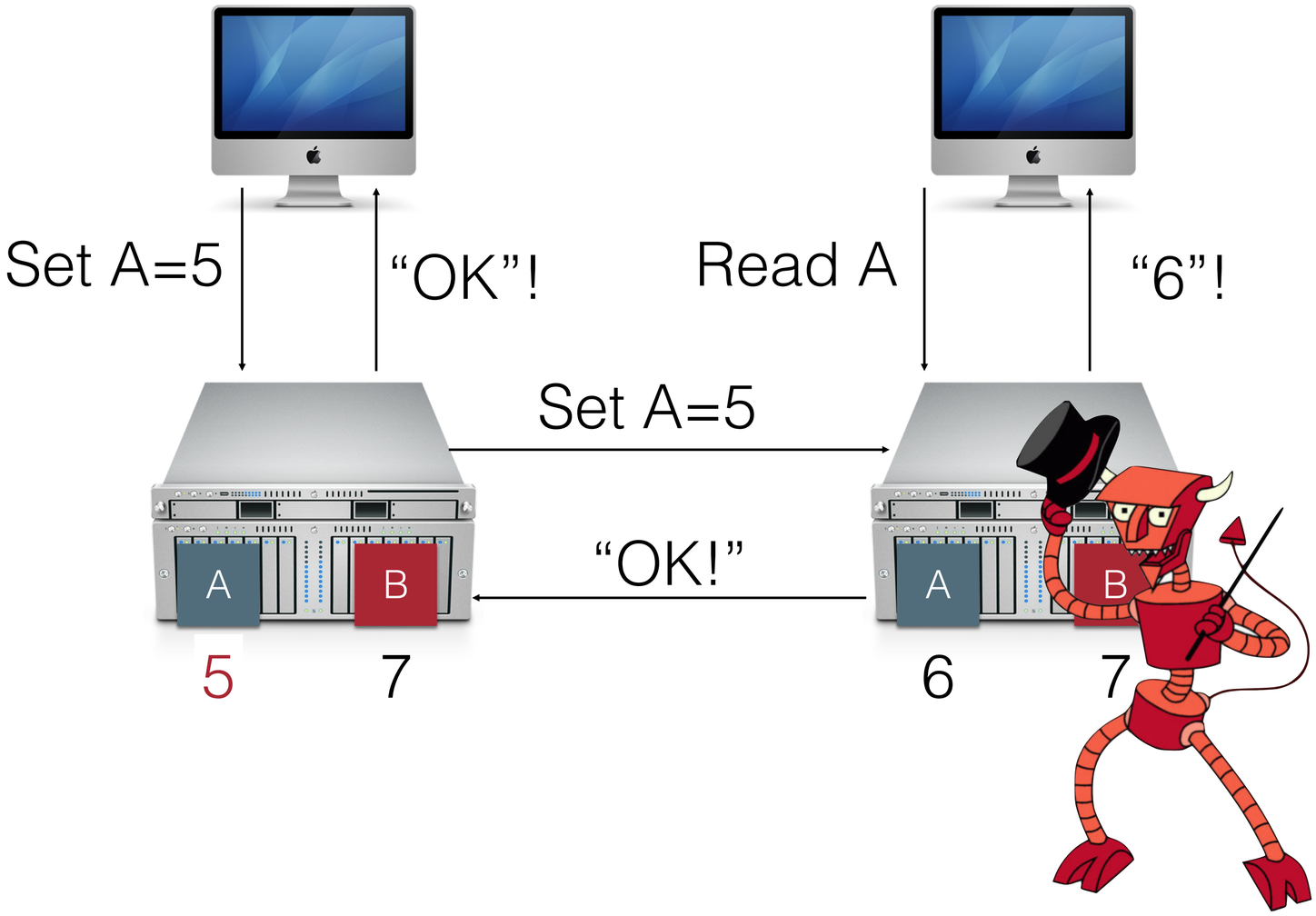
Distributed Software Engineering Abstractions

Key Question: Consistency vs Availability

- Distributed system will never match exact semantics of non-distributed system
- For replication do we value more: guaranteed consistency (looks like a single machine) or guaranteed availability (sometimes read stale data)?
  - For a lock server?
  - For the order of tweets on twitter?
- For partitioning: Where can we draw the line?

Byzantine Faults

Unfortunately, still more things can go wrong



Review: Learning Objectives for this Lesson

By the end of this lesson, you should be able to…

- Describe partitioning and replication as building blocks for distributed systems
- Evaluate the tradeoffs between consistency and availability in distributed systems
- Answer the question: how does partitioning and replication help us satisfy requirements for distributed systems?