Exam review

CS 475, Spring 2018
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

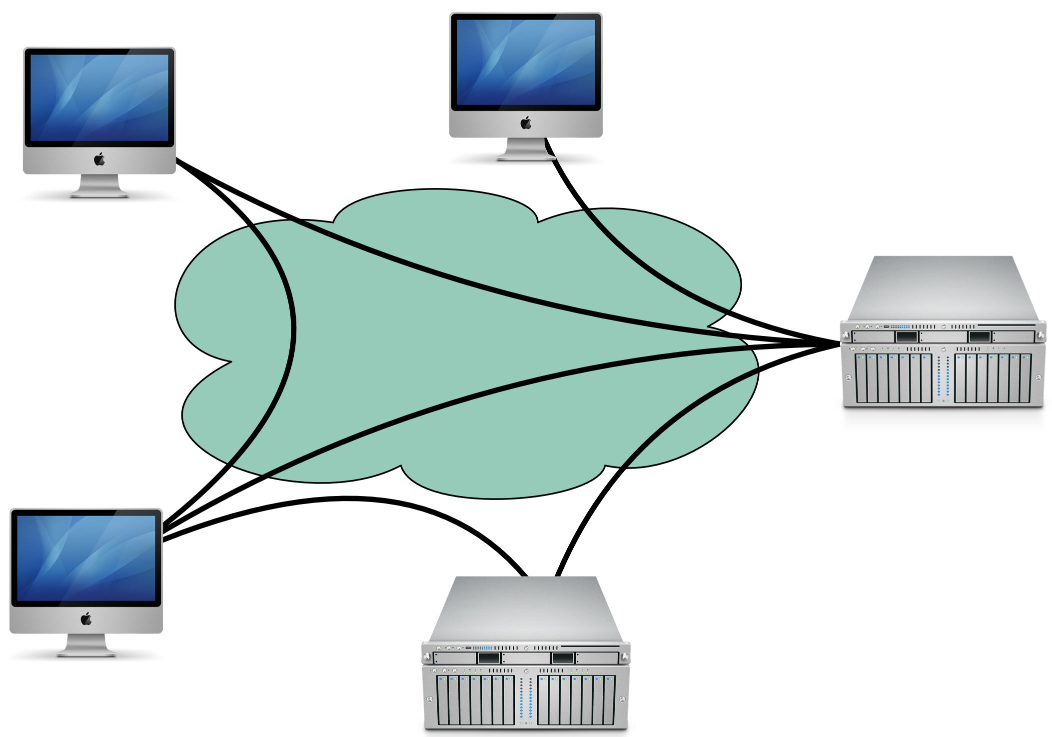
Course Topics

- This course will teach you **how** and **why** to build distributed systems
- Distributed System is “a collection of independent computers that appears to its users as a single coherent system”
- This course will give you theoretical knowledge of the tradeoffs that you’ll face when building distributed systems

Course Topics

How do I run multiple things at once on my computer?

Concurrency, first half of course



How do I run a big task across many computers?

Distributed Systems, second half of course

Concurrency

- Goal: do multiple things, at once, coordinated, on one computer
  - Update UI
  - Fetch data
  - Respond to network requests
  - Improve responsiveness, scalability
- Recurring problems:
  - Coordination: what is shared, when, and how?

Why expand to distributed systems?

- Scalability
- Performance
- Latency
- Availability
- Fault Tolerance

"Distributed Systems for Fun and Profit", Takada

Distributed Systems Goals

- **Scalability**
- Performance
- Latency
- Availability
- Fault Tolerance

"the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth."

"Distributed Systems for Fun and Profit", Takada

Distributed Systems Goals

- Scalability
- **Performance**
- Latency
- Availability
- Fault Tolerance

"is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used."

Distributed Systems Goals

- Scalability
- Performance
- **Latency**
- Availability
- Fault Tolerance

"The state of being latent; delay, a period between the initiation of something and the it becoming visible."

Distributed Systems Goals

- Scalability
- Performance
- Latency
- **Availability**
- Fault Tolerance

"the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable."

Availability = uptime / (uptime + downtime).

Often measured in "nines"

Availability %	Downtime/year
90%	>1 month
99%	< 4 days
99.9%	< 9 hours
99.99%	<1 hour
99.999%	5 minutes
99.9999%	31 seconds

Distributed Systems Goals

- Scalability
- Performance
- Latency
- Availability
- **Fault Tolerance**

"ability of a system to behave in a well-defined manner once faults occur"

What kind of faults?

Disks fail

Networking fails

Power supplies fail

Security breached

Datacenter goes offline

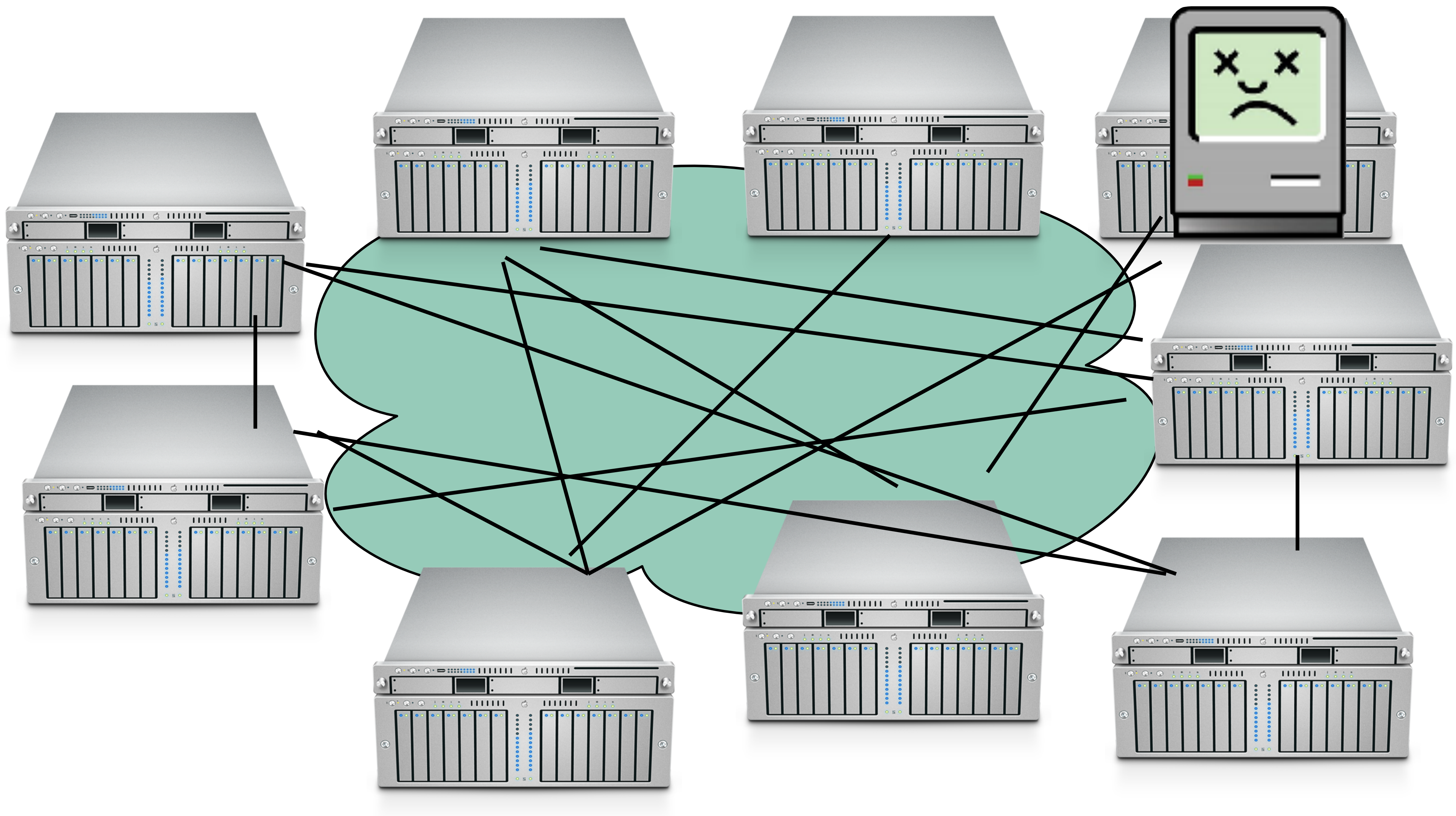
Power goes out

More machines, more problems

- PLUS, the network may be:
  - Unreliable
  - Insecure
  - Slow
  - Expensive
  - Limited

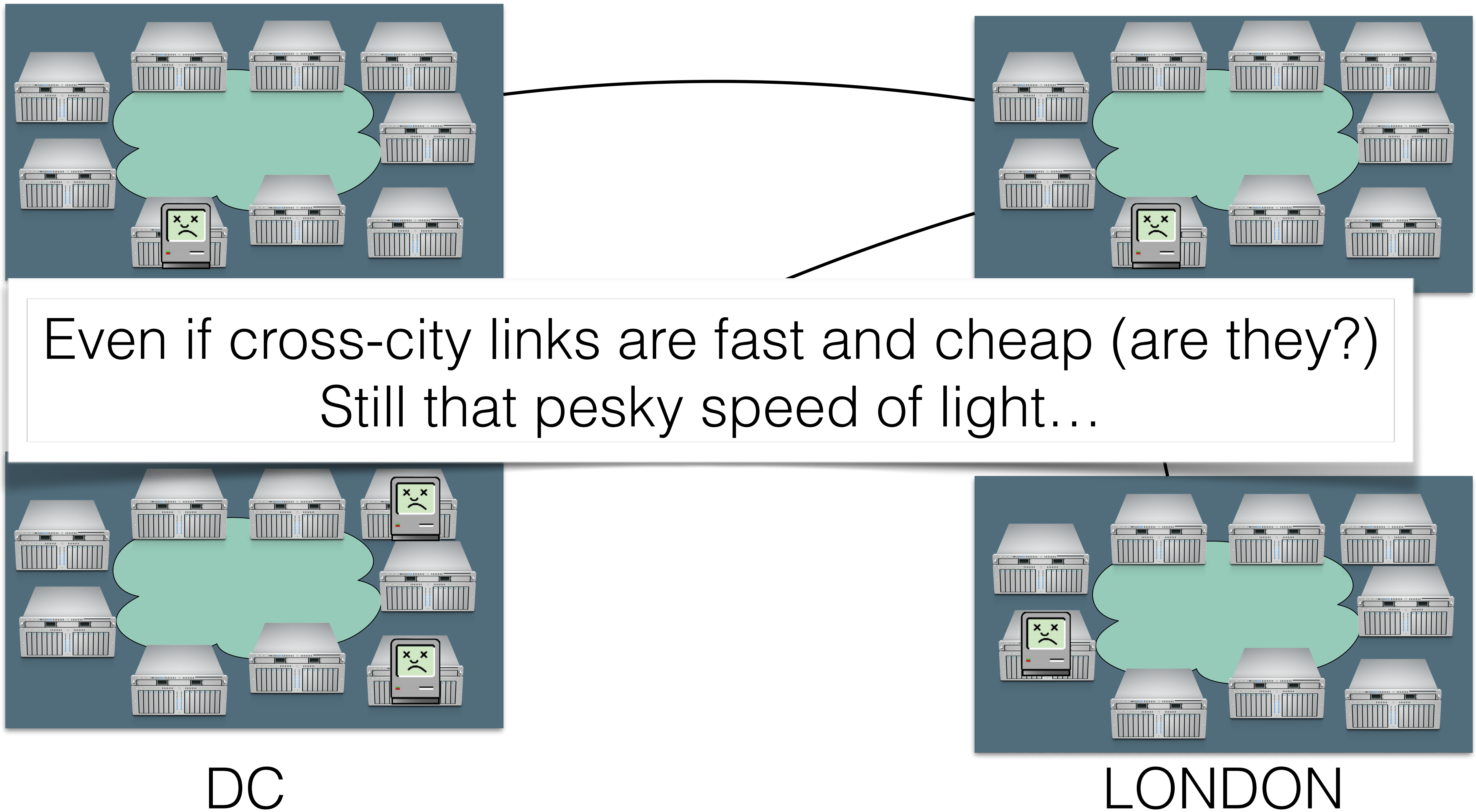
Constraints

- Number of nodes
- Distance between nodes



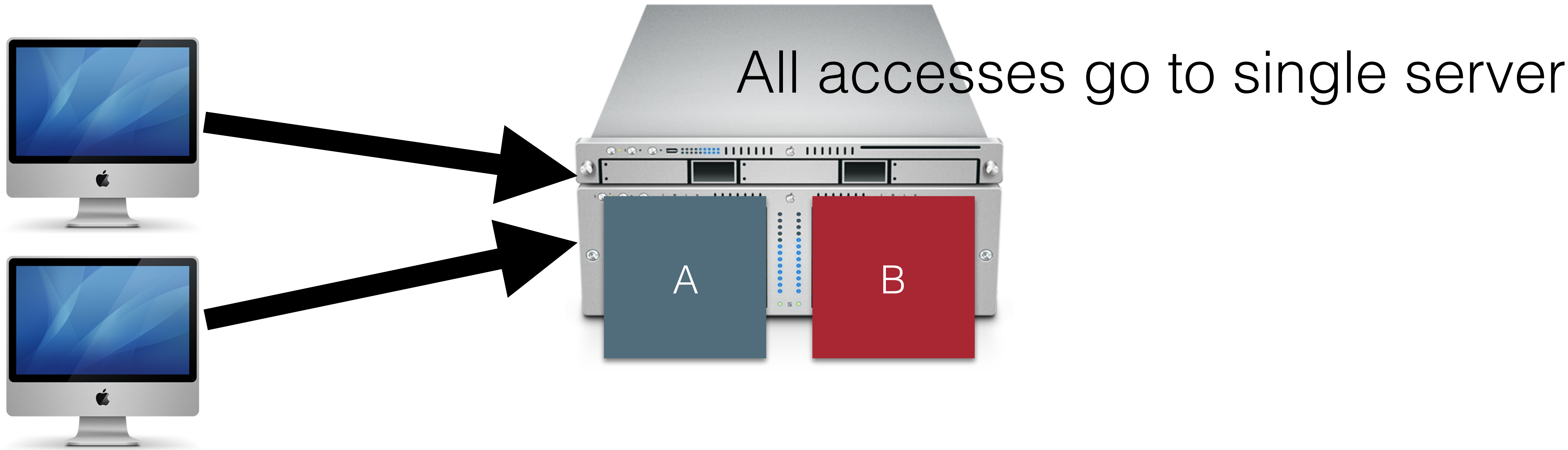
Constraints

- Number of nodes
- Distance between nodes



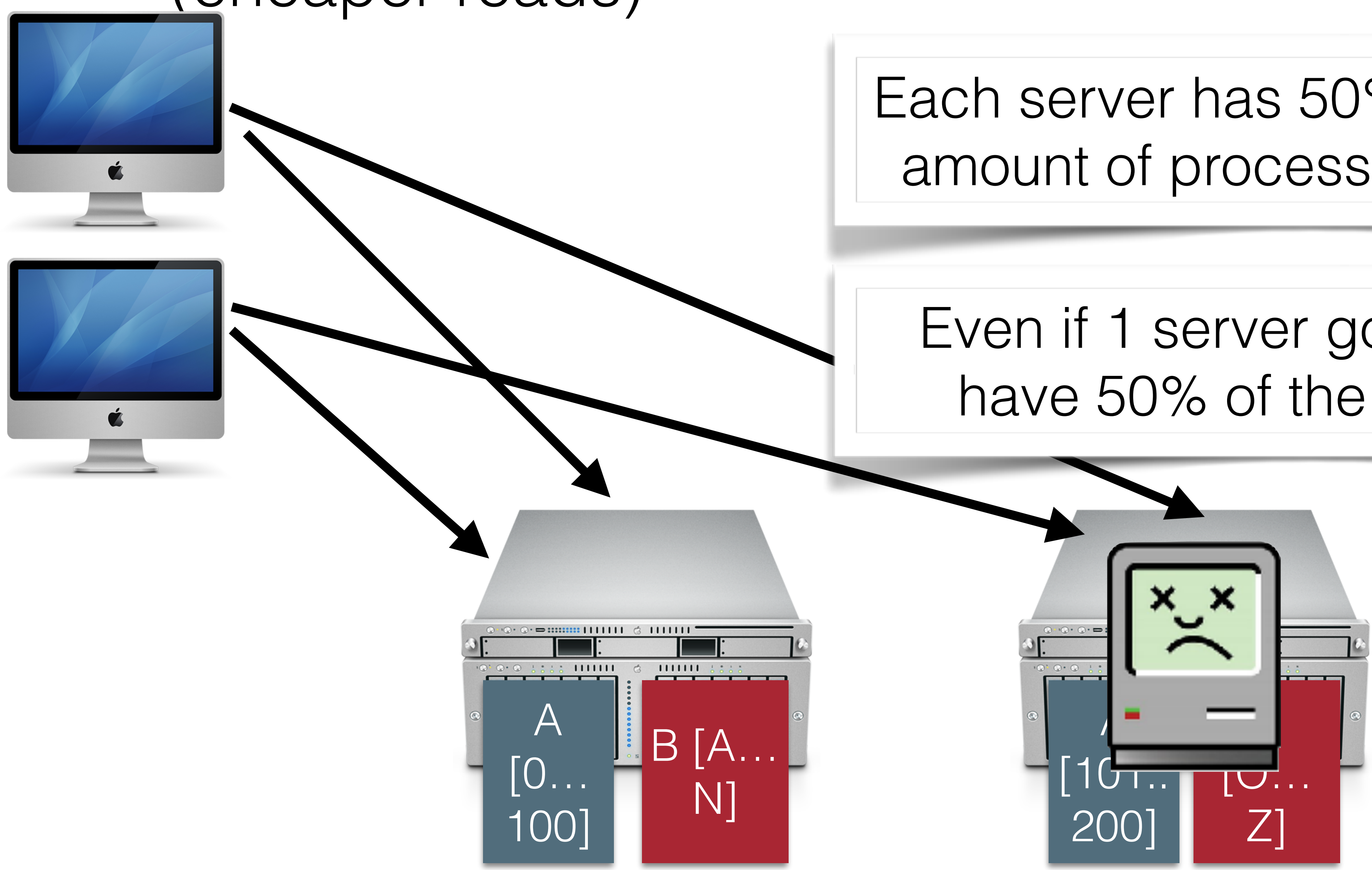
Even if cross-city links are fast and cheap (are they?)
Still that pesky speed of light…

Recurring Solution #1: Partitioning



Recurring Solution #1: Partitioning

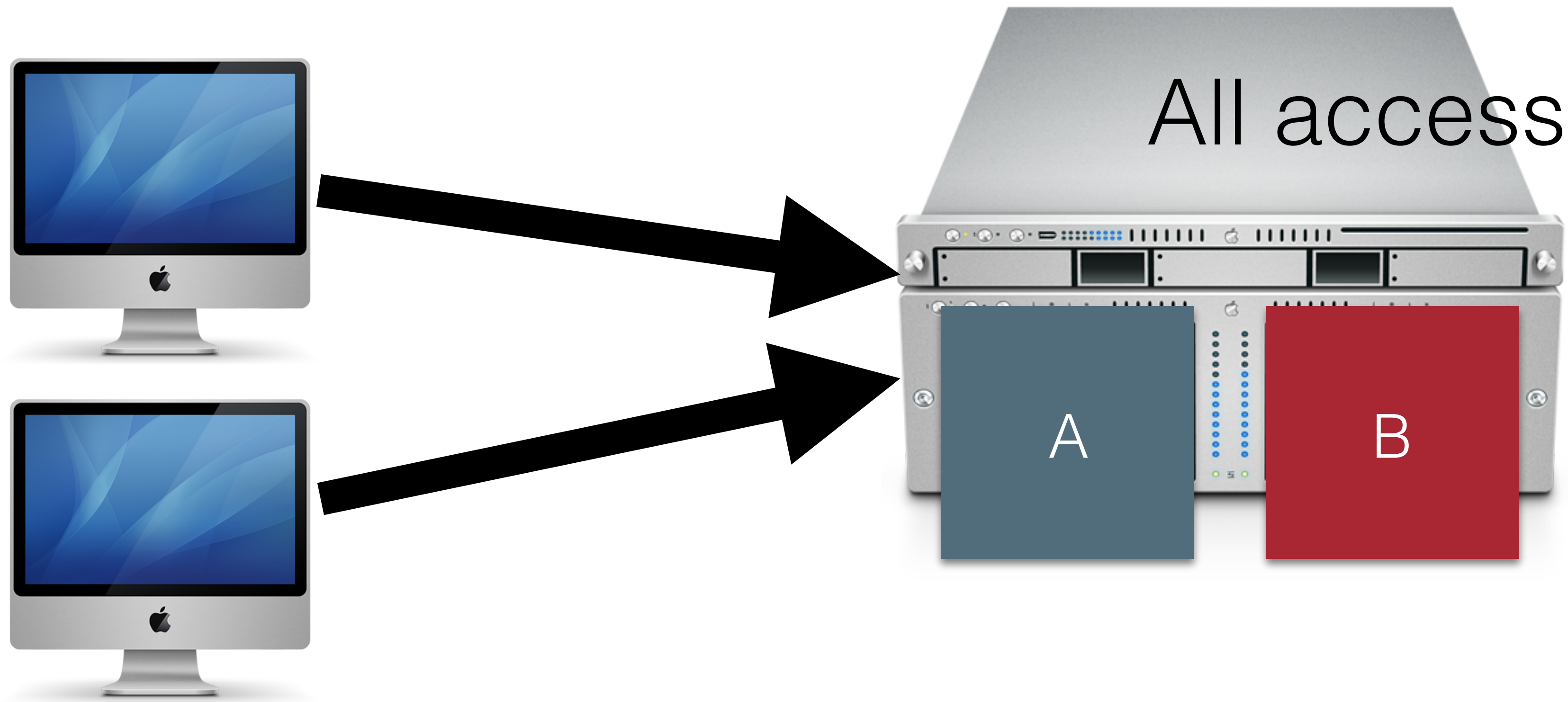
- Divide data up in some (hopefully logical) way
- Makes it easier to process data concurrently (cheaper reads)



Each server has 50% of data, limits amount of processing per server.

Even if 1 server goes down, still have 50% of the data online.

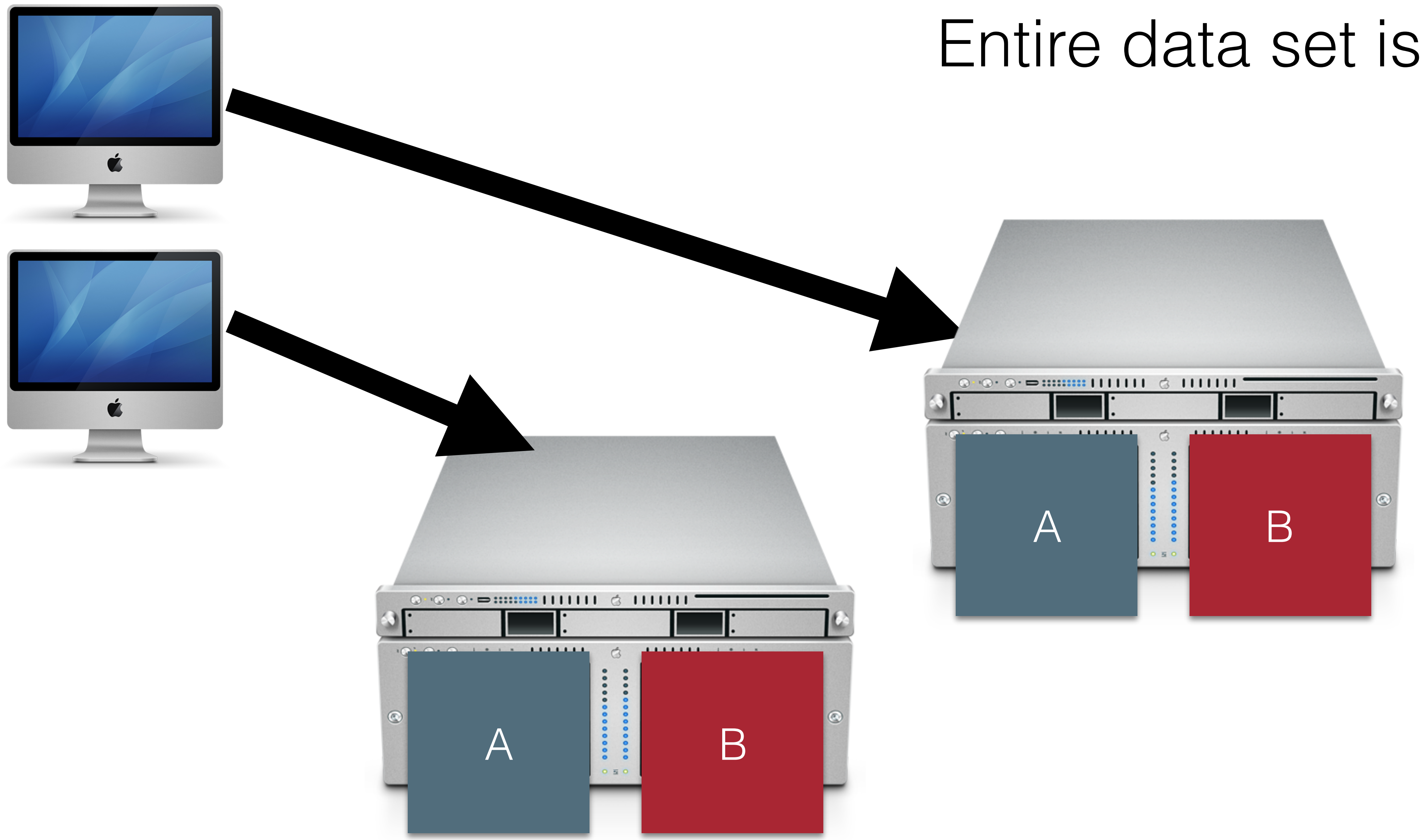
Recurring Solution #2: Replication



All accesses go to single server

Recurring Solution #2: Replication

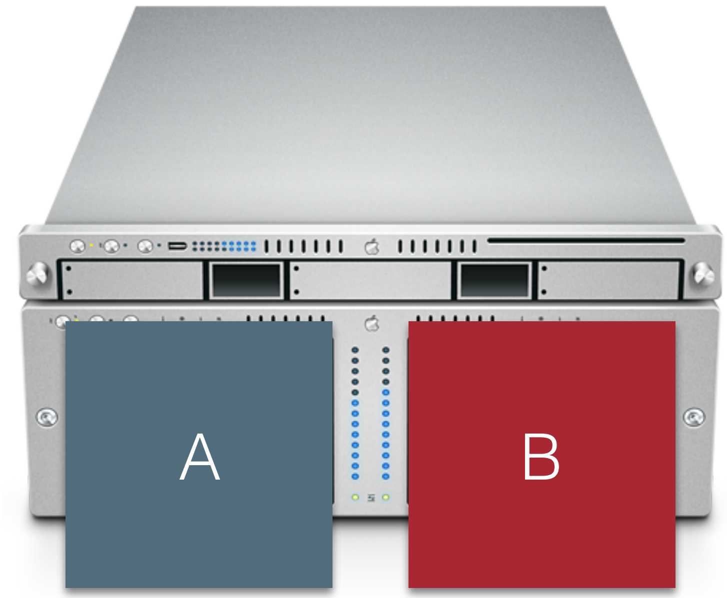
Entire data set is copied



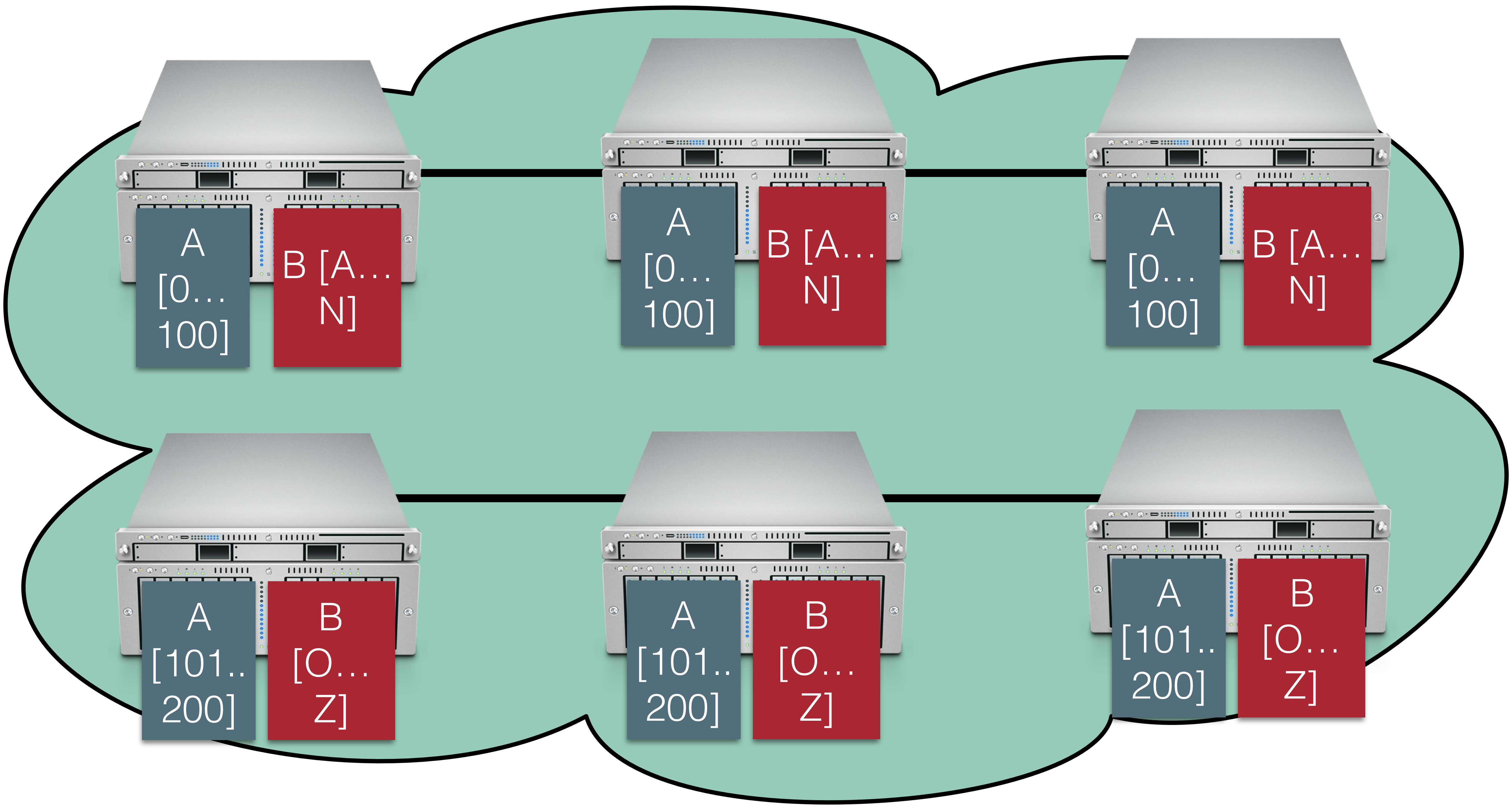
Recurring Solution #2: Replication

- 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

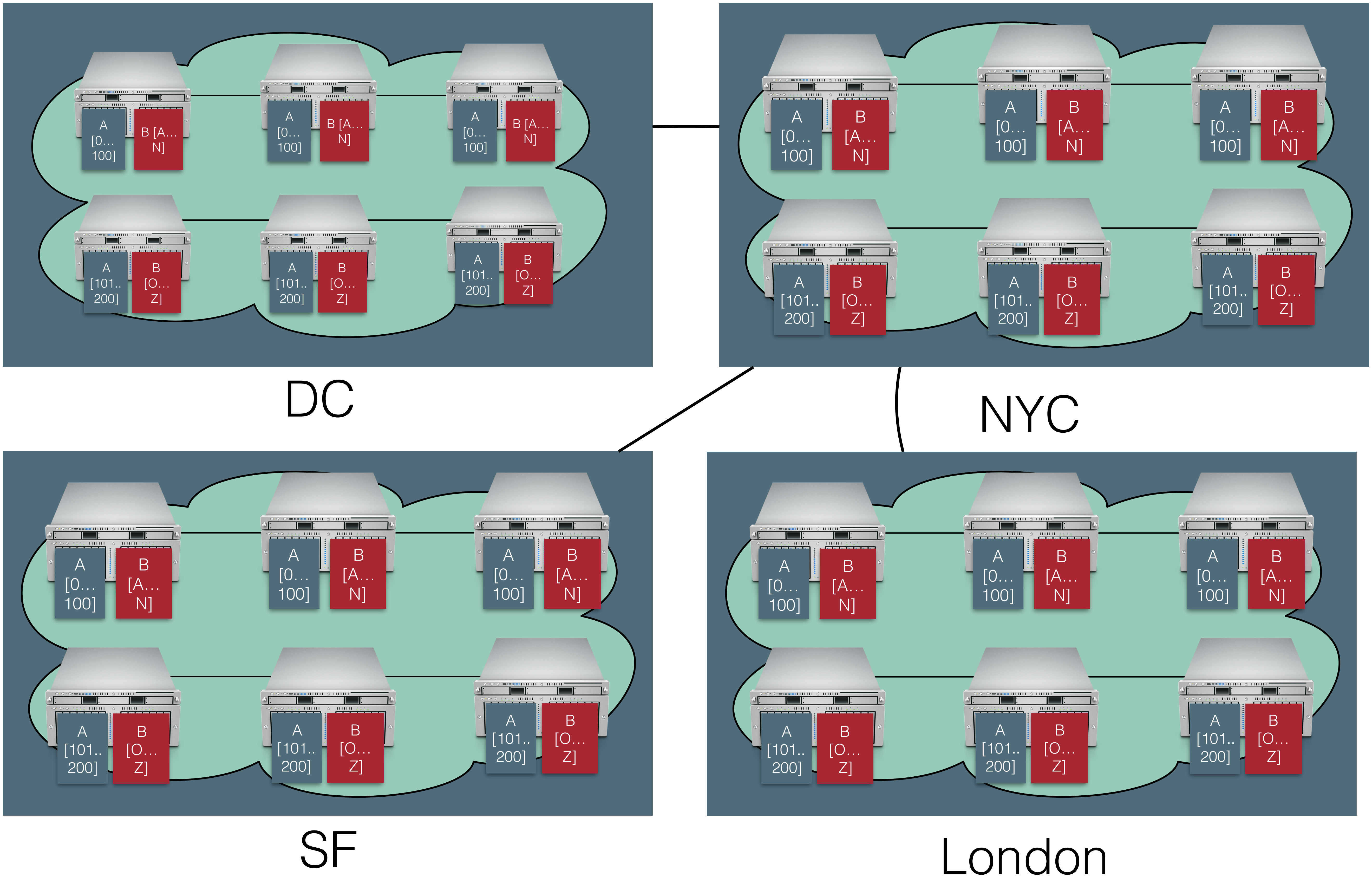
Partitioning + Replication



Partitioning + Replication



Partitioning + Replication



Conventional Hashing + Sharding

- In practice, might use an off-the-shelf hash function, like sha1
- sha1(url) -> 160 bit hash result % 20 -> server ID (assuming 20 servers)
- But what happens when we add or remove a server?
  - Data is stored on what *was* the right server, but now that the number of servers changed, the right server changed too!

Conventional Hashing

Assume we have 10 keys, all integers

0, 3, 6, 9 | 1, 4, 7 | 2, 5, 8

Adding a new server

Conventional Hashing

Assume we have 10 keys, all integers

0, 4, 8 | 1, 5, 9 | 2, 6 | 3, 7

Adding a new server

8/10 keys had to be reshuffled!
Expensive!

Consistent Hashing

- Problem with regular hashing: very sensitive to changes in the number of servers holding the data!
- Consistent hashing will require on average that only K/n keys need to be remapped for K keys with *n* different slots (in our case, that would have been 10/4 = 2.5 [compare to 8])

Consistent Hashing

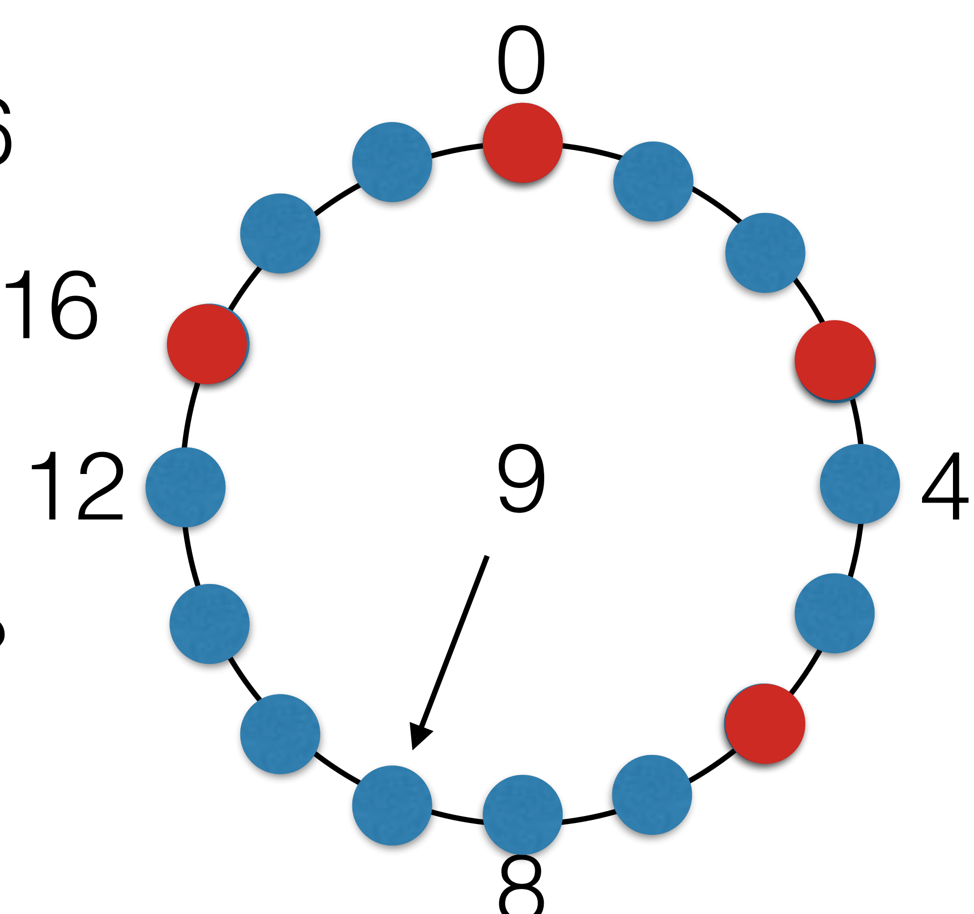
- Construction:
  - Assign each of C hash buckets to random points on mod 2^n circle, where hash key size = n
  - Map object to pseudo-random position on circle
  - Hash of object is the closest clockwise bucket

Example: hash key size is 16

Each (blue dot) is a value of hash % 16

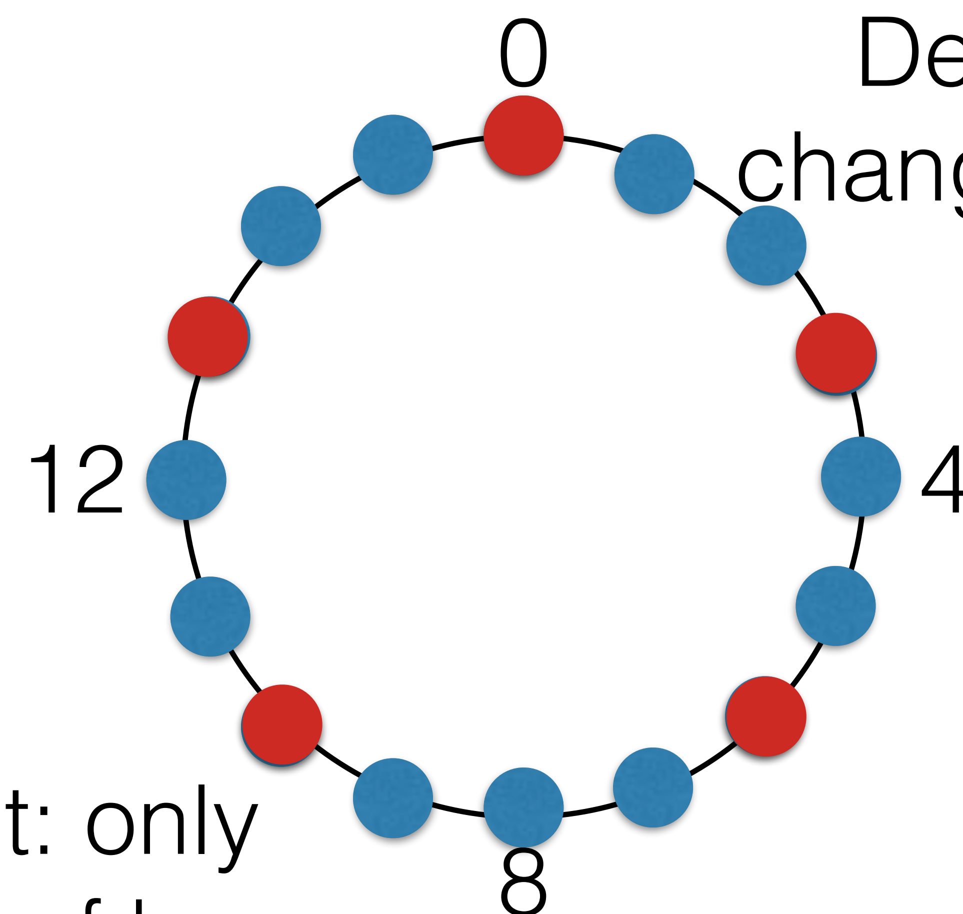
Each (red dot) is a bucket

Example: bucket with key 9?



Consistent Hashing

It is relatively smooth: adding a new bucket doesn't change that much

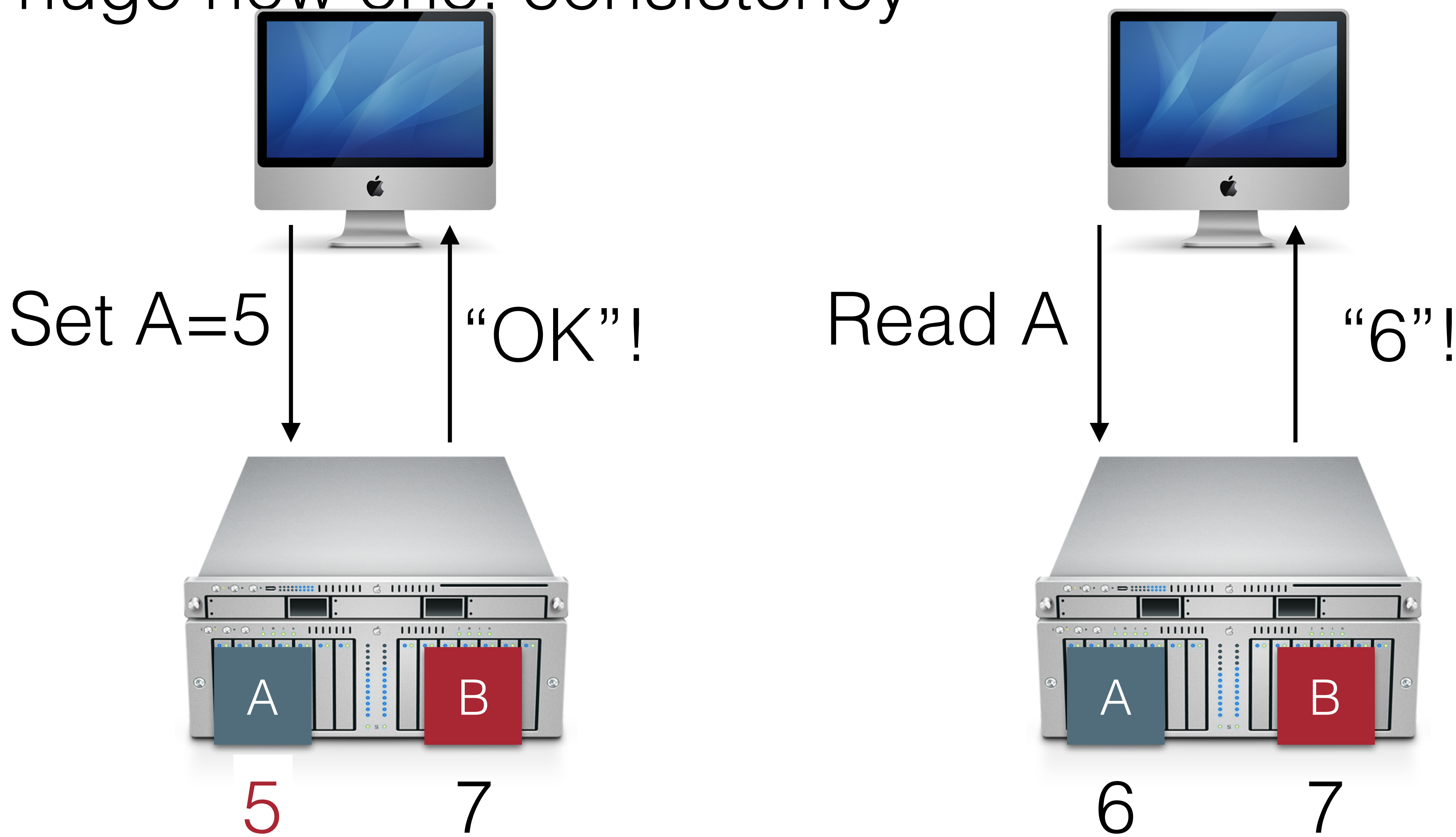


Delete bucket: only changes location of keys 1,2,3

Add new bucket: only changes location of keys 7,8,9,10

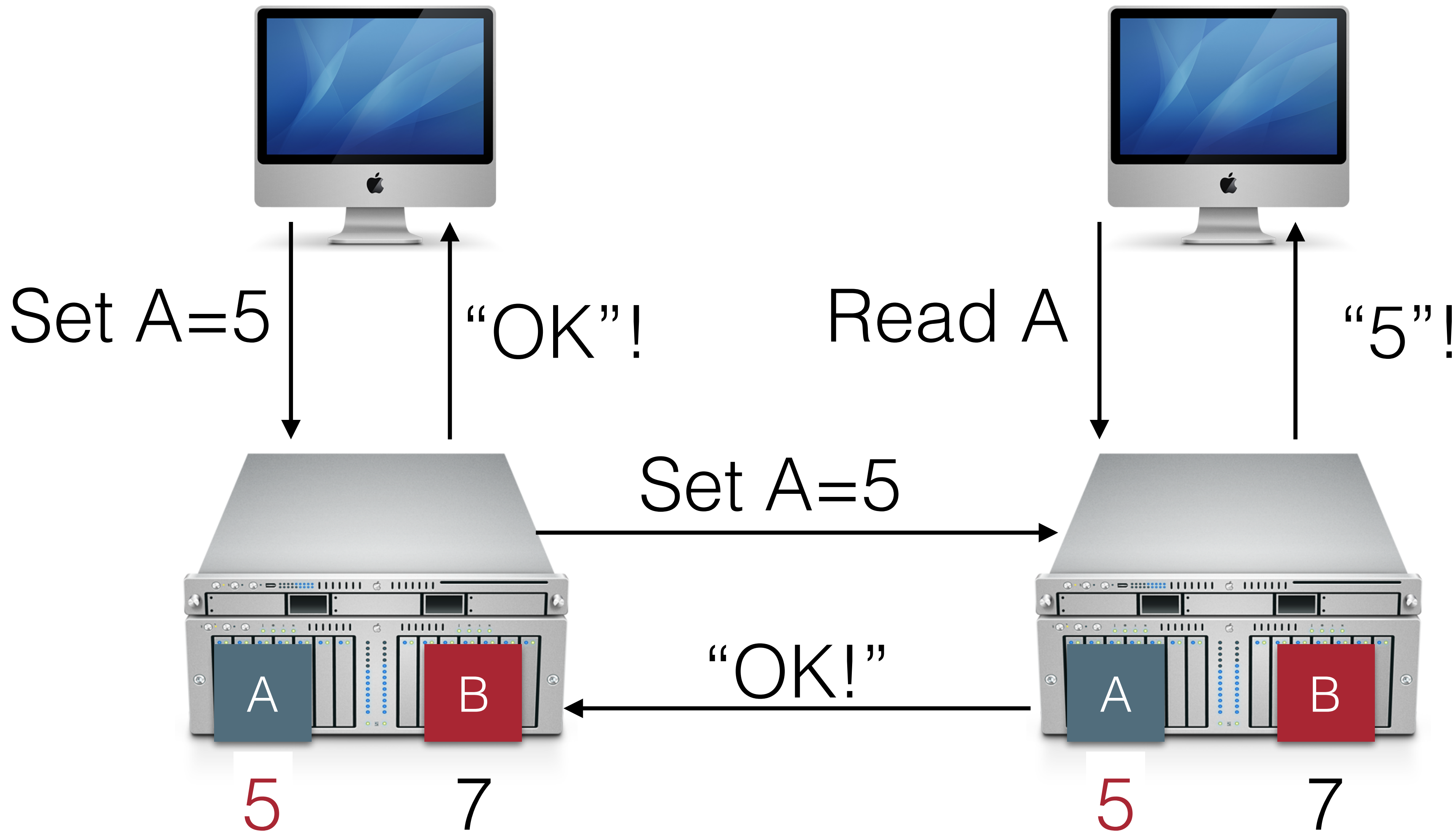
Recurring Problem: Replication

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



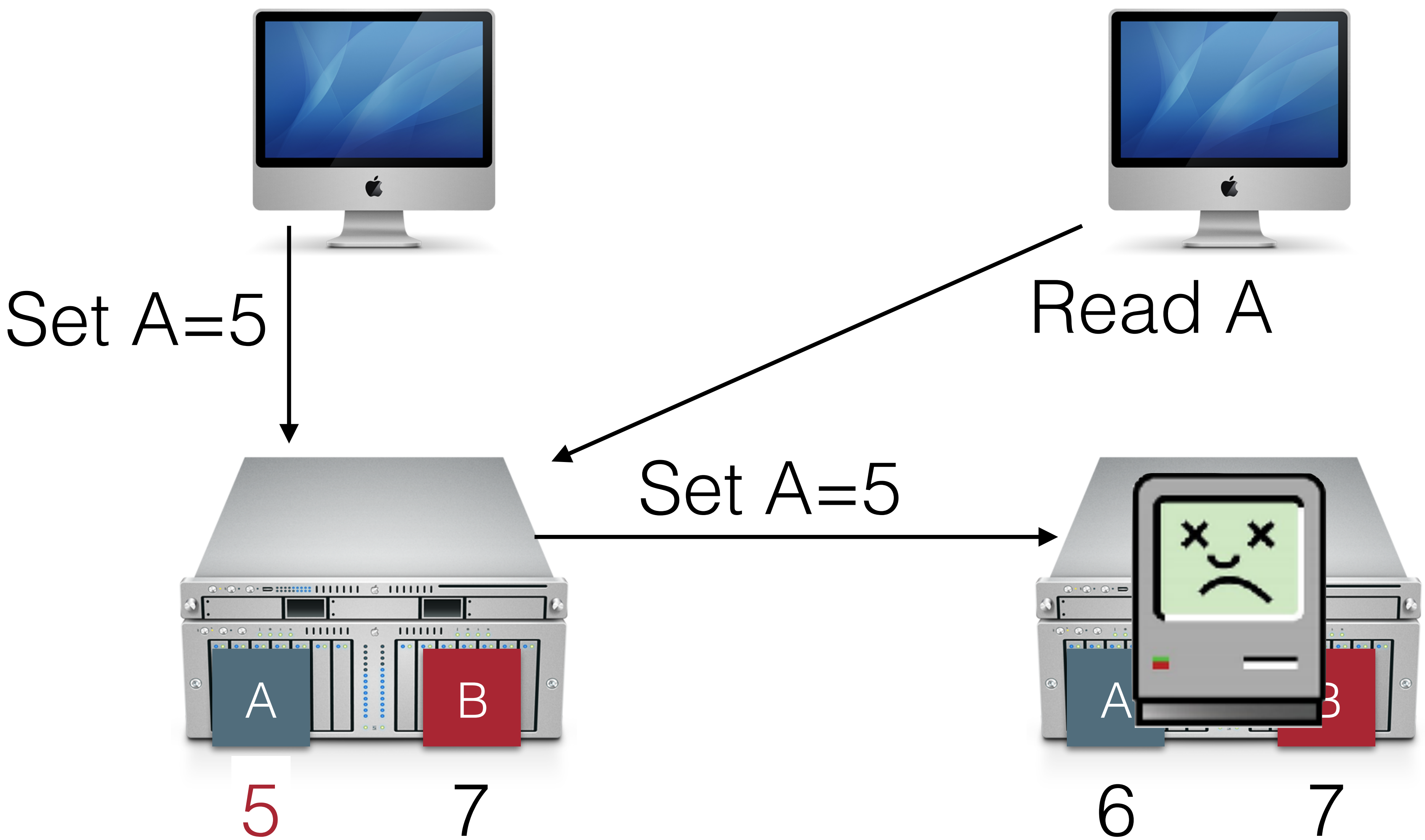
OK, we obviously need to actually do something here to replicate the data… but what?

Sequential Consistency

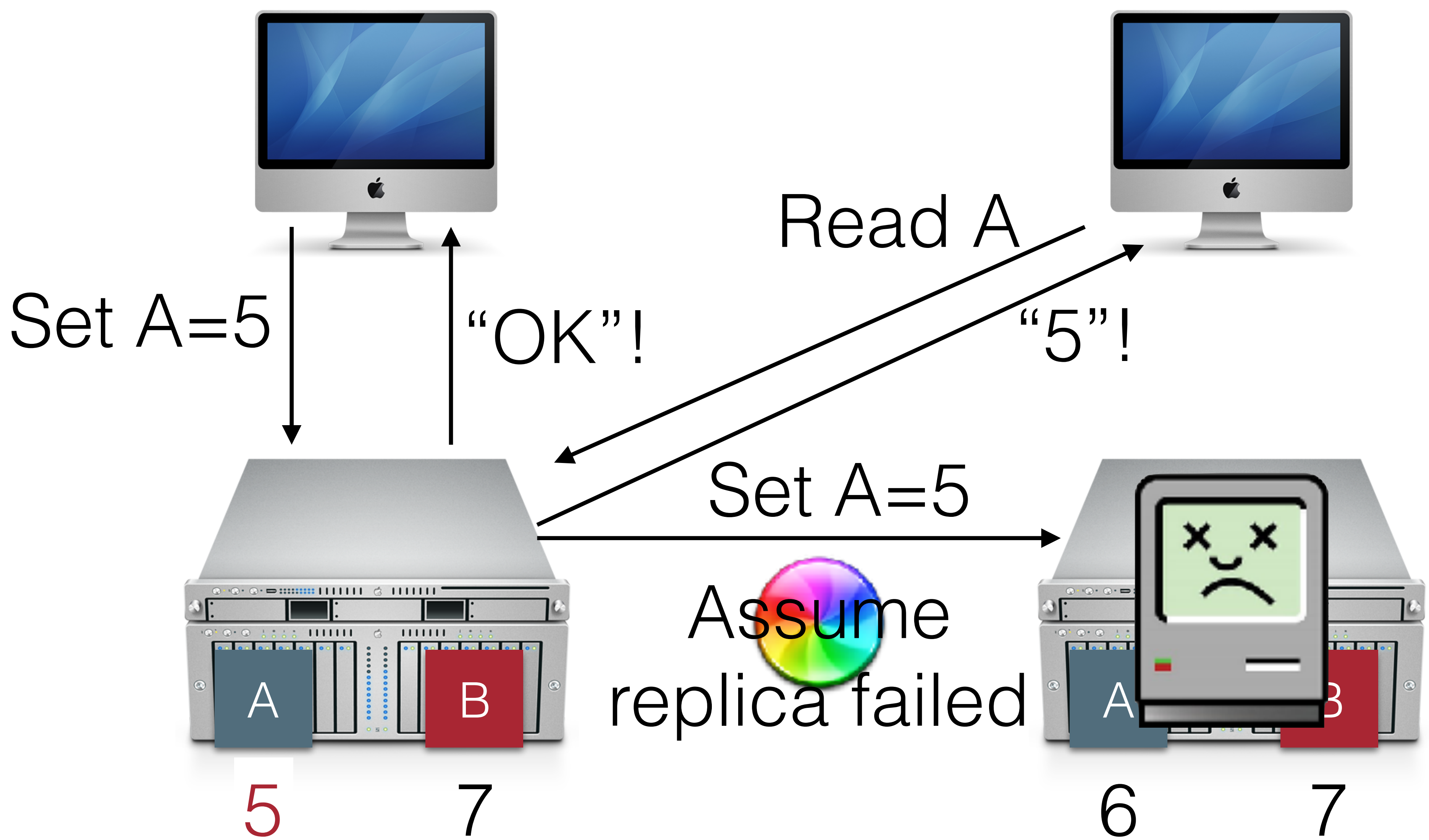


Availability

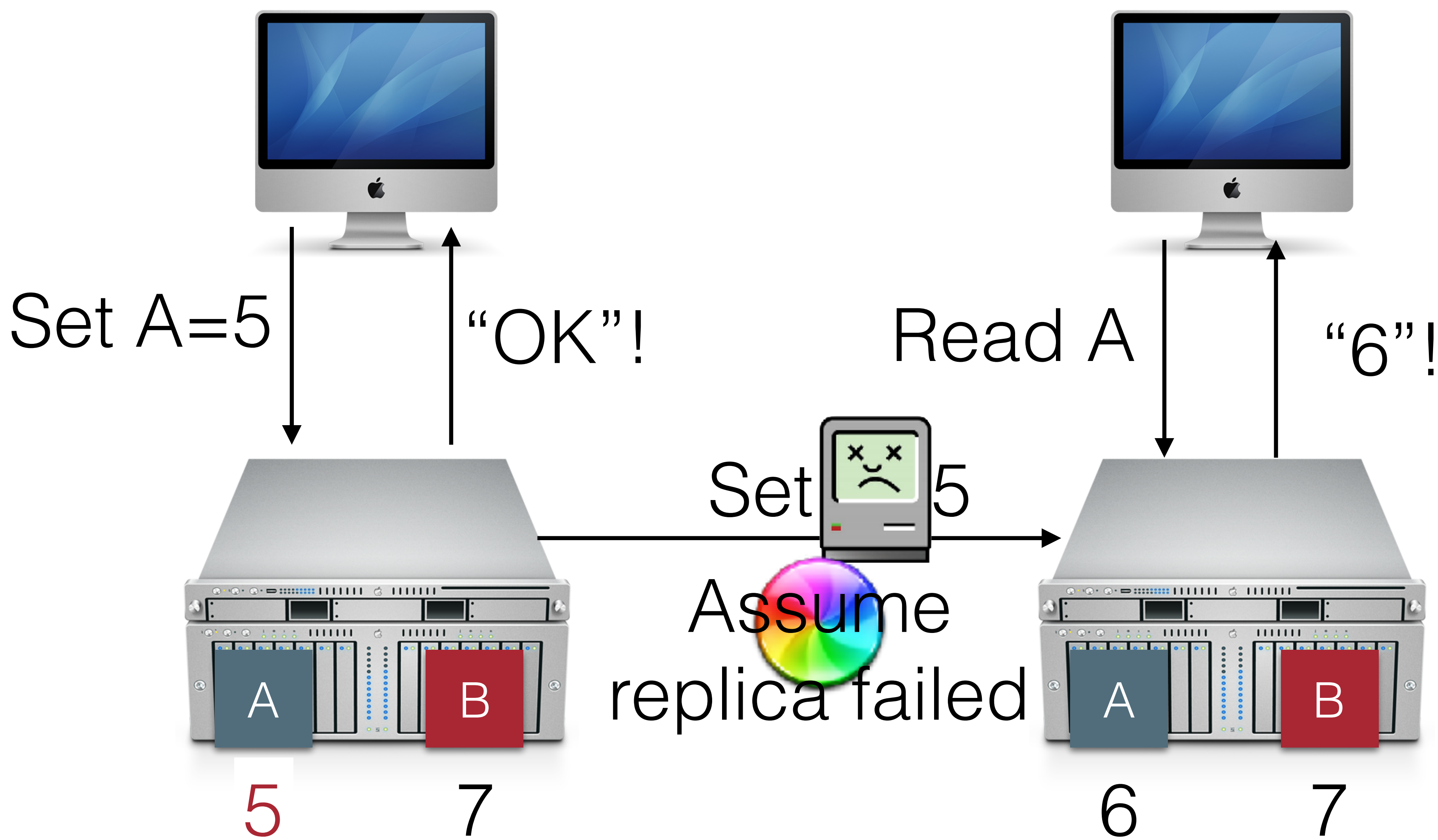
- Our protocol for sequential consistency does NOT guarantee that the system will be available!



Consistent + Available

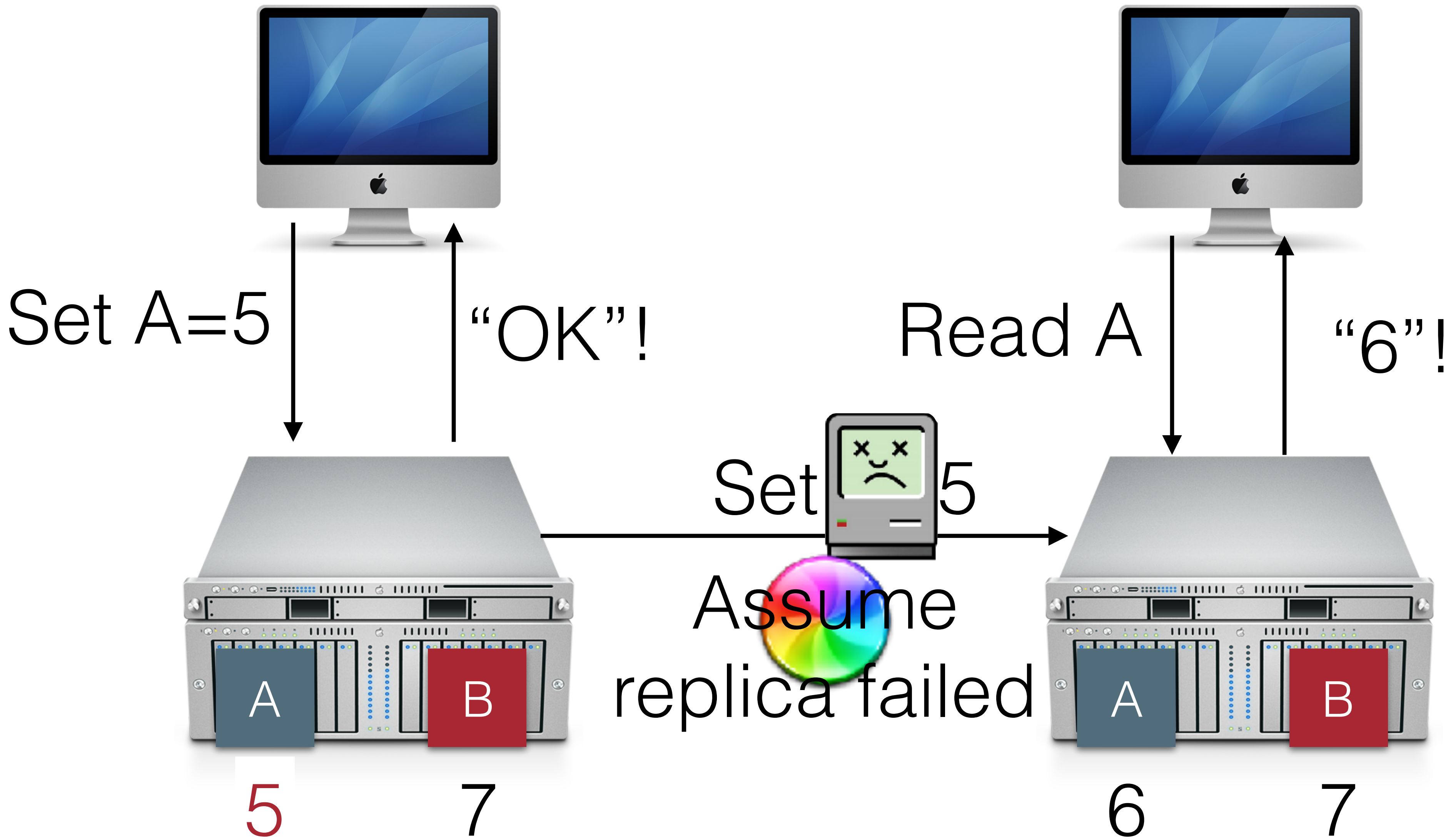


Still broken...



Network Partitions

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



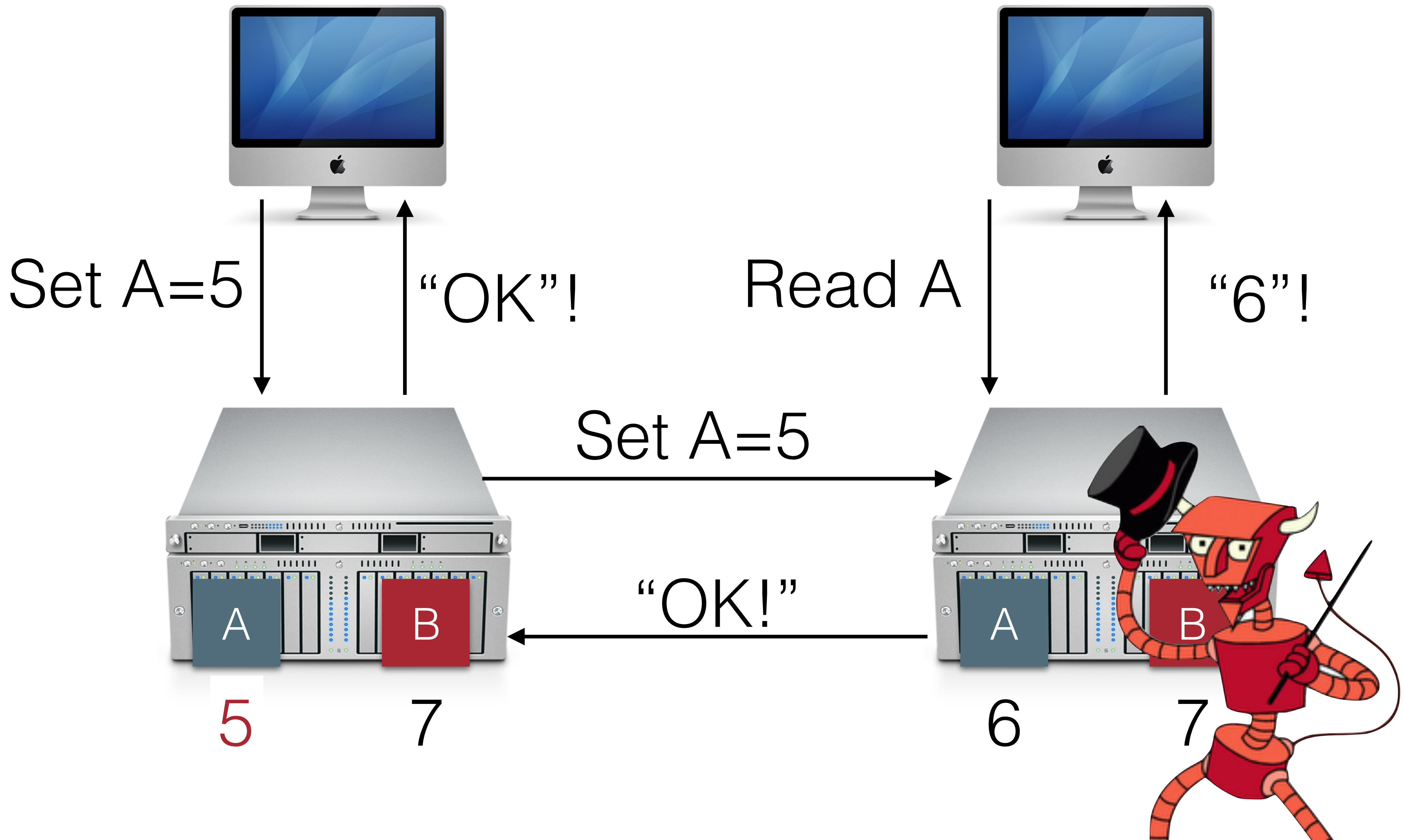
CAP Theorem

- 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

Agreement

- In distributed systems, we have multiple nodes that need to all agree that some object has some state
- Examples:
  - Who owns a lock
  - Whether or not to commit a transaction
  - The value of a file

Agreement Generally

- Most distributed systems problems can be reduced to this one:
  - Despite being separate nodes (with potentially different views of their data and the world)…
  - All nodes that store the same object O must apply all updates to that object in the same order (consistency)
  - All nodes involved in a transaction must either commit or abort their part of the transaction (atomicity)
- Easy?
  - … but nodes can restart, die or be arbitrarily slow
  - … and networks can be slow or unreliable too

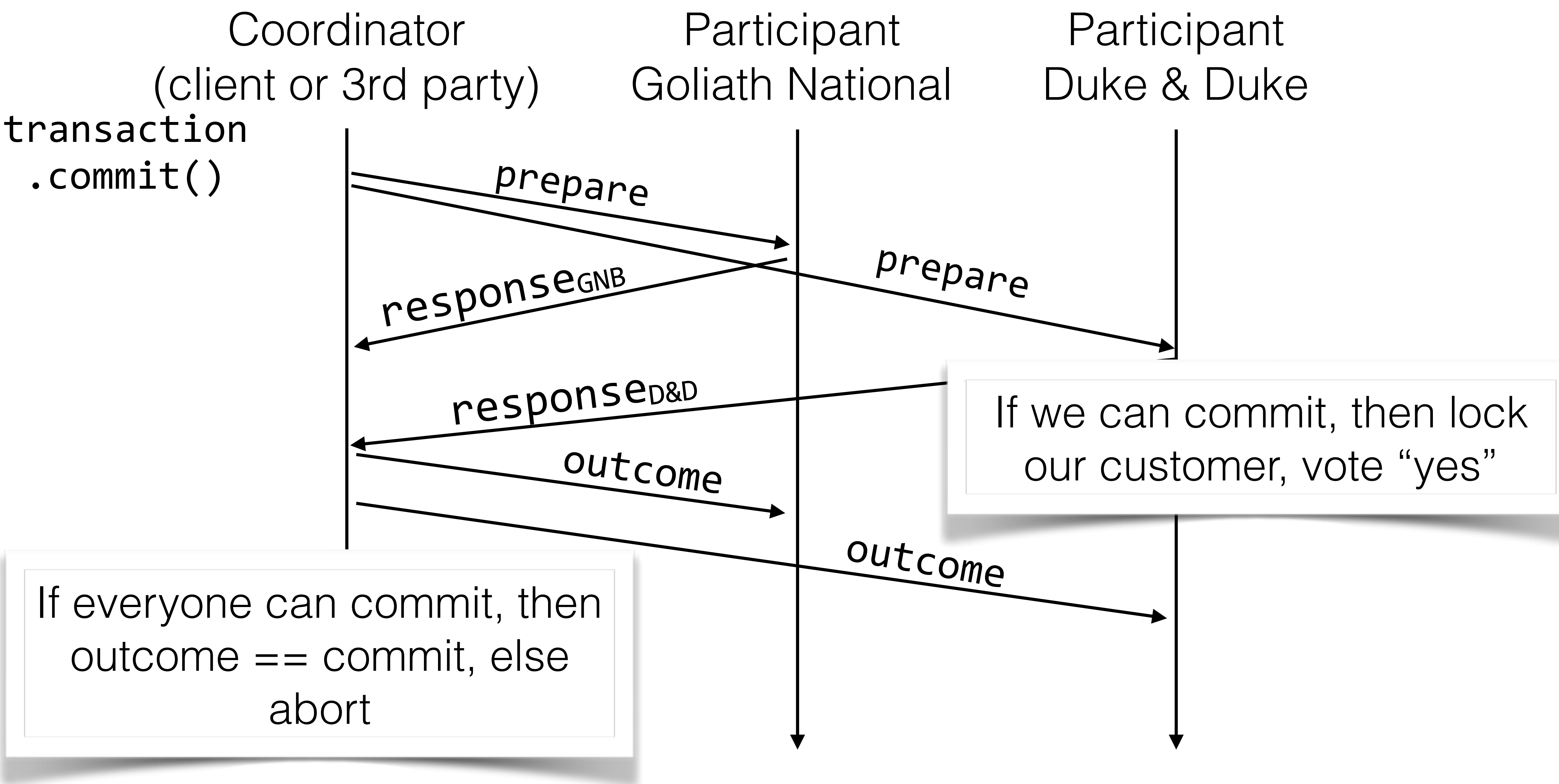
Properties of Agreement

- **Safety** (correctness)
  - All nodes agree on the same value (which was proposed by some node)
- **Liveness** (fault tolerance, availability)
  - If less than N nodes crash, the rest should still be OK

1-Phase Commit

- Naive protocol: coordinator broadcasts out "commit!" continuously until participants all say "OK!"
- Problem: what happens when a participants fails during commit? How do the other participants know that they shouldn't have really committed and they need to abort?

2PC Example


Coordinator
(client or 3rd party)
Participant
Goliath National
Participant
Duke & Duke
transaction
.commit()
prepare
prepare
responseGNB
responseD&D
If we can commit, then lock our customer, vote “yes”
outcome
outcome
If everyone can commit, then outcome == commit, else abort

Timeouts in 2PC

- Example:
  - Coordinator times out waiting for Goliath National Bank's response
  - Bank times out waiting for coordinator's outcome message
- Causes?
  - Network
  - Overloaded hosts
  - Both are very realistic…

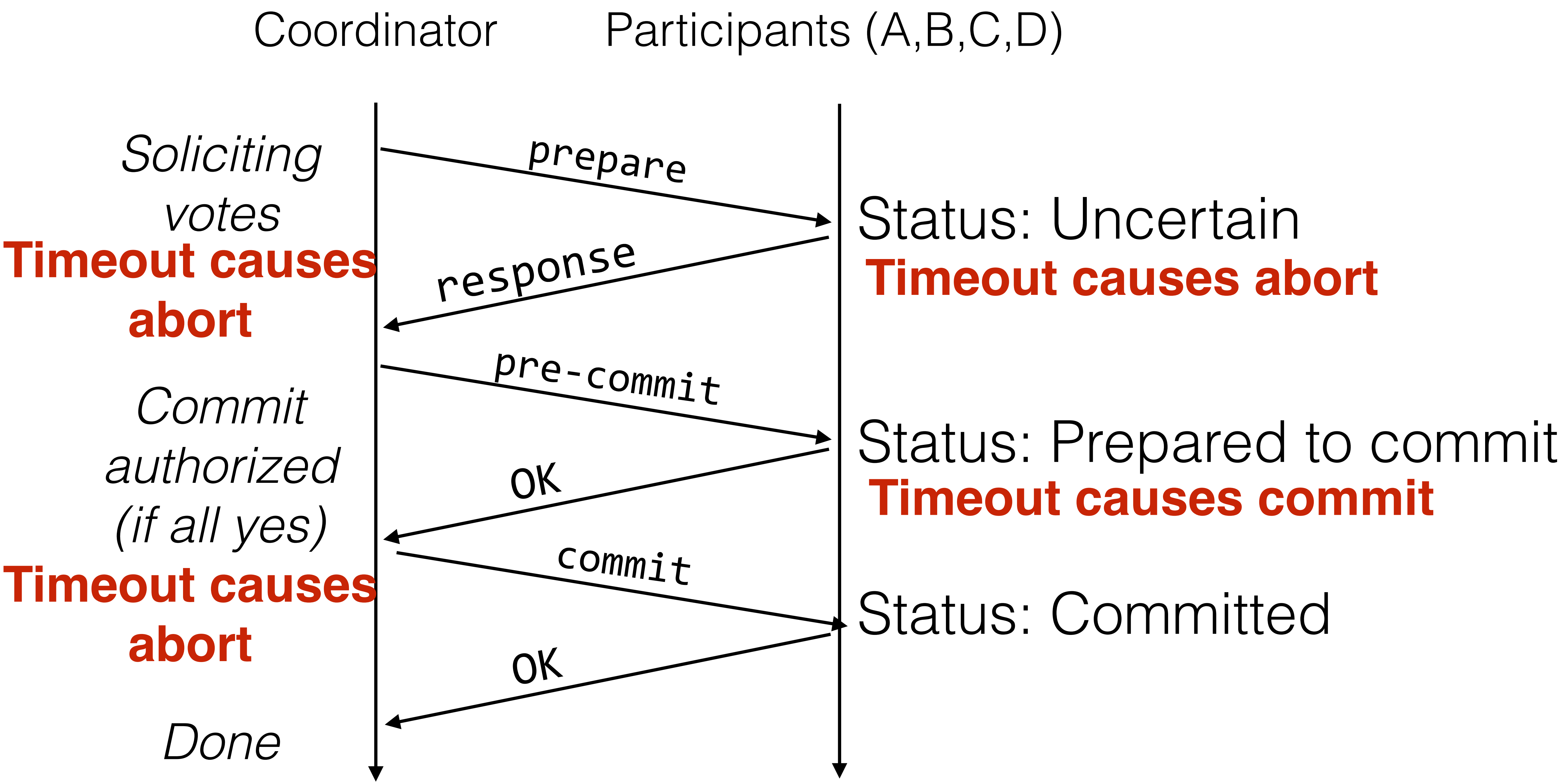
3 Phase Commit

- Goal: Eliminate this specific failure from blocking liveness

3 Phase Commit

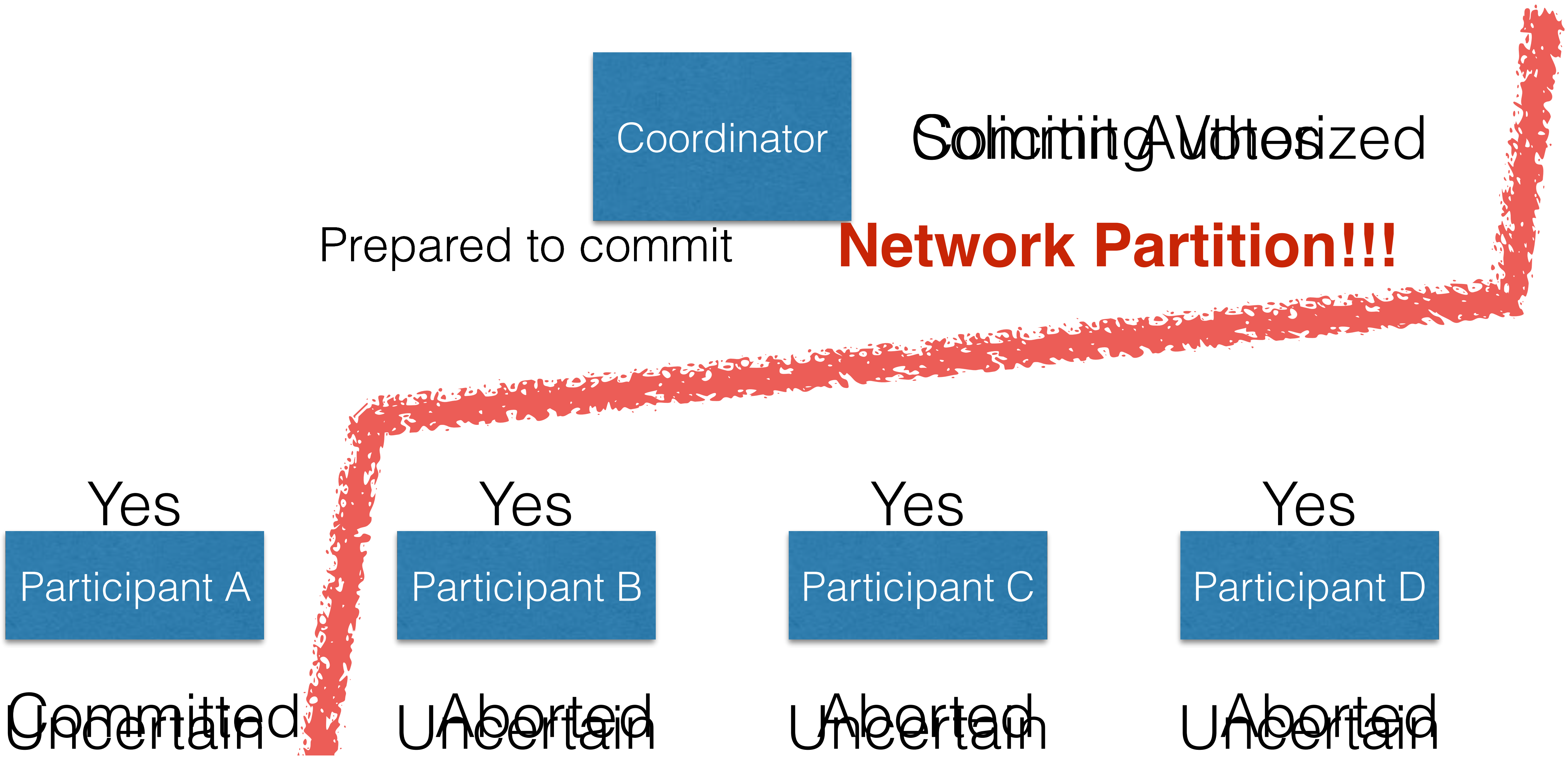
- Goal: Avoid blocking on node failure
- How?
  - Think about how 2PC is better than 1PC
    - 1PC means you can never change your mind or have a failure after committing
    - 2PC **still** means that you can't have a failure after committing (committing is irreversible)
- 3PC idea:
  - Split commit/abort into 2 sub-phases
    - 1: Tell everyone the outcome
    - 2: Agree on outcome
  - Now: EVERY participant knows what the result will be before they irrevocably commit!

3PC Example



Coordinator
Participants (A,B,C,D)
Soliciting votes
Timeout causes abort
Commit authorized (if all yes)
Timeout causes abort
Done
prepare
response
pre-commit
OK
commit
OK
Status: Uncertain
Timeout causes abort
Status: Prepared to commit
Timeout causes commit
Status: Committed

Partitions



Timeout behavior: Commit!

Timeout behavior: abort

Can we fix it?

- Short answer: No.
- Fischer, Lynch & Paterson (FLP) Impossibility Result:
  - Assume that nodes can only fail by crashing, network is reliable but can be delayed arbitrarily
  - Then, there can not be a deterministic algorithm for the consensus problem subject to these failures

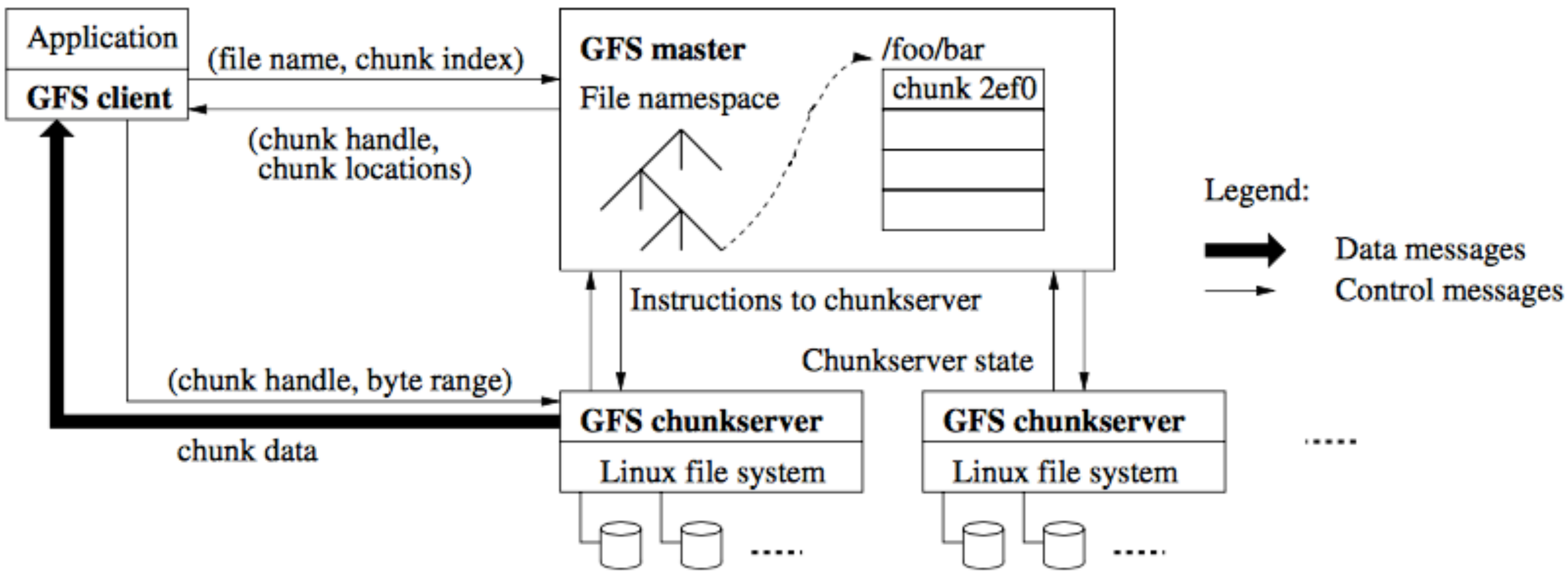
FLP - Intuition

- 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)

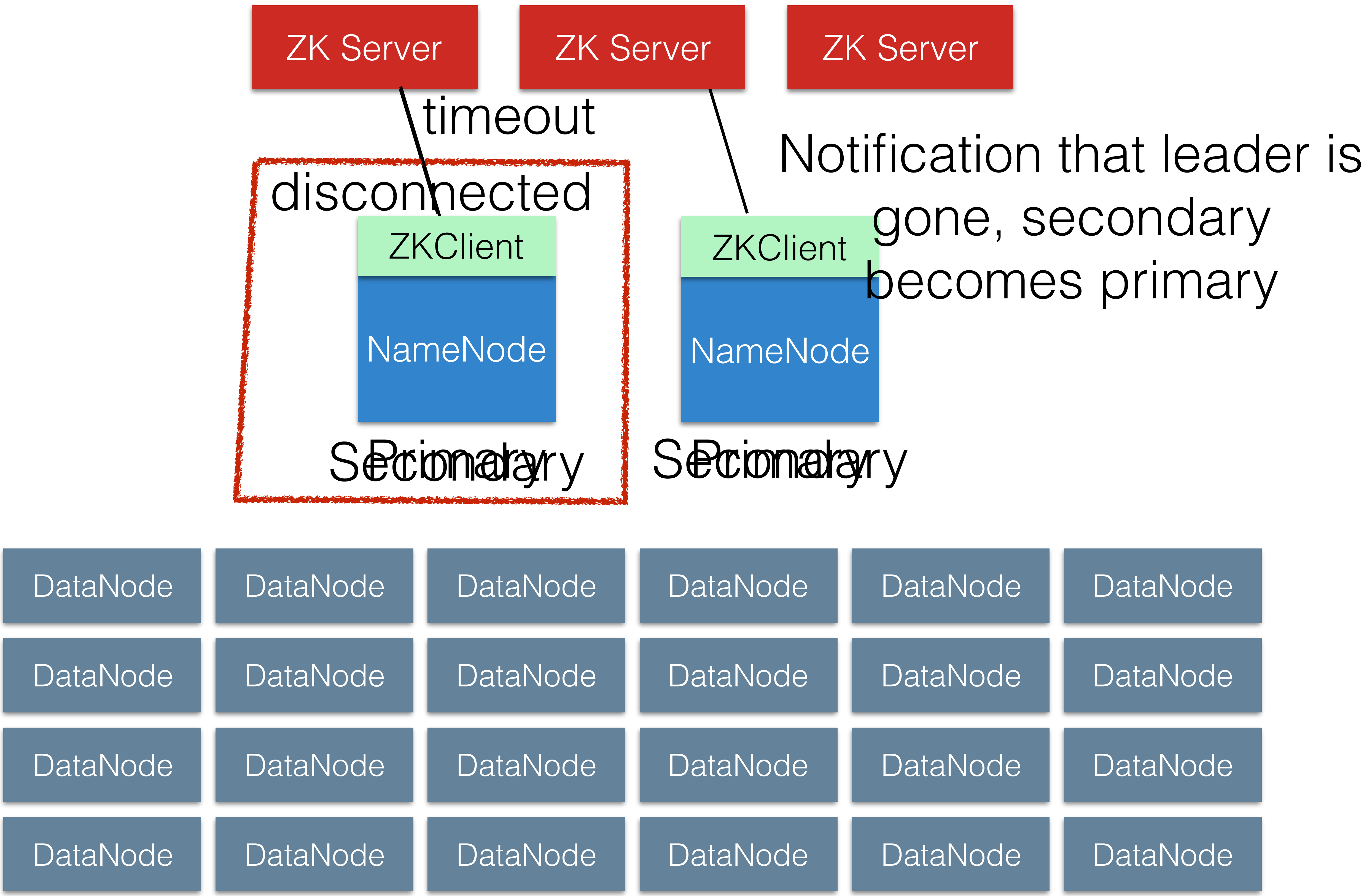
ZooKeeper - Guarantees

- **Liveness guarantees**: if a majority of ZooKeeper servers are active and communicating the service will be available
- **Durability guarantees**: if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover

GFS Architecture

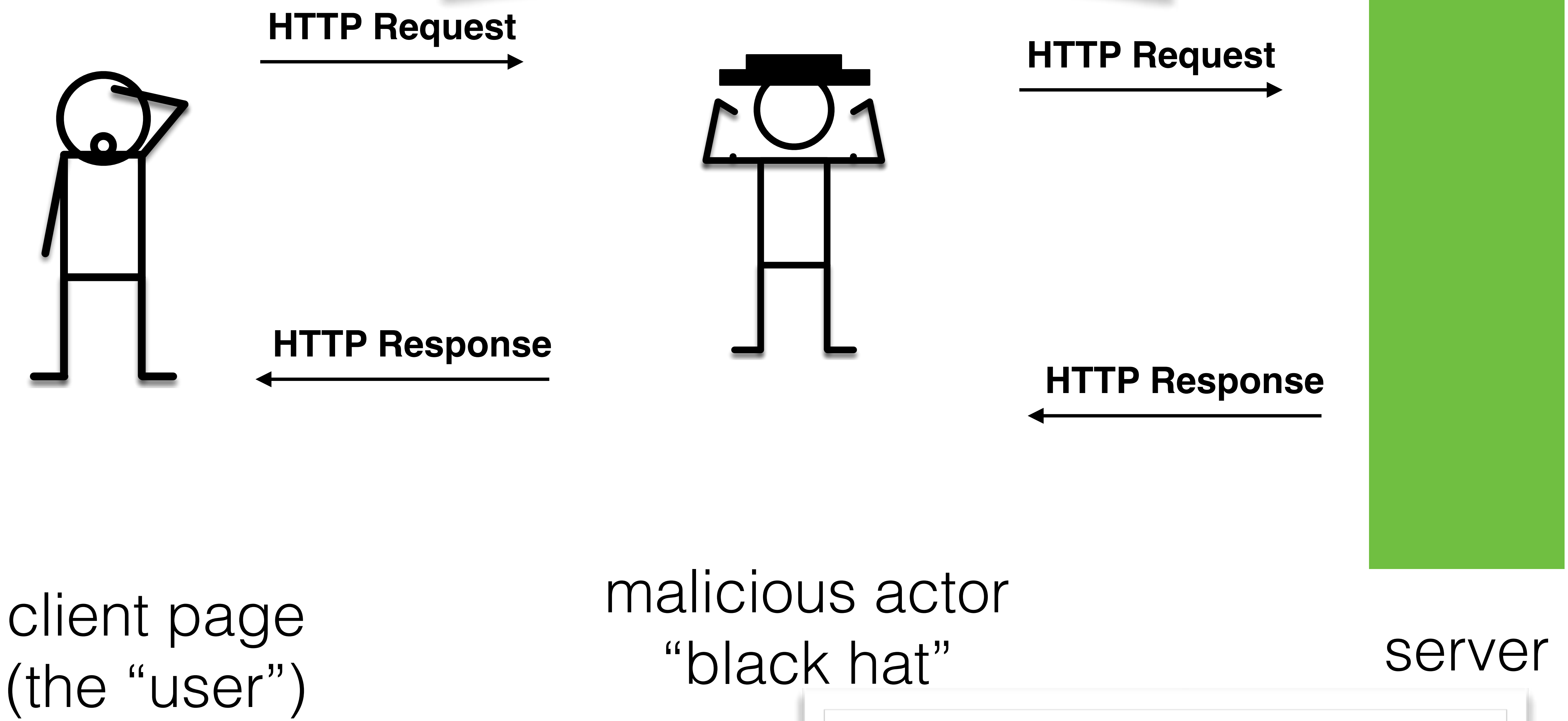


Hadoop + ZooKeeper



Example Threat: Web Server

Might be "man in the middle" that intercepts requests and impersonates user or server.



Do I trust that this response *really* came from the server?

Do I trust that this request *really* came from the user?

Symmetric vs Asymmetric Crypto

	Symmetric Crypto	Asymmetric Crypto
Requires a pre-shared secret	Yes	No
Relative speed	Very fast	Very slow