Exam Review

CS 475, Spring 2019
Concurrent & Distributed Systems

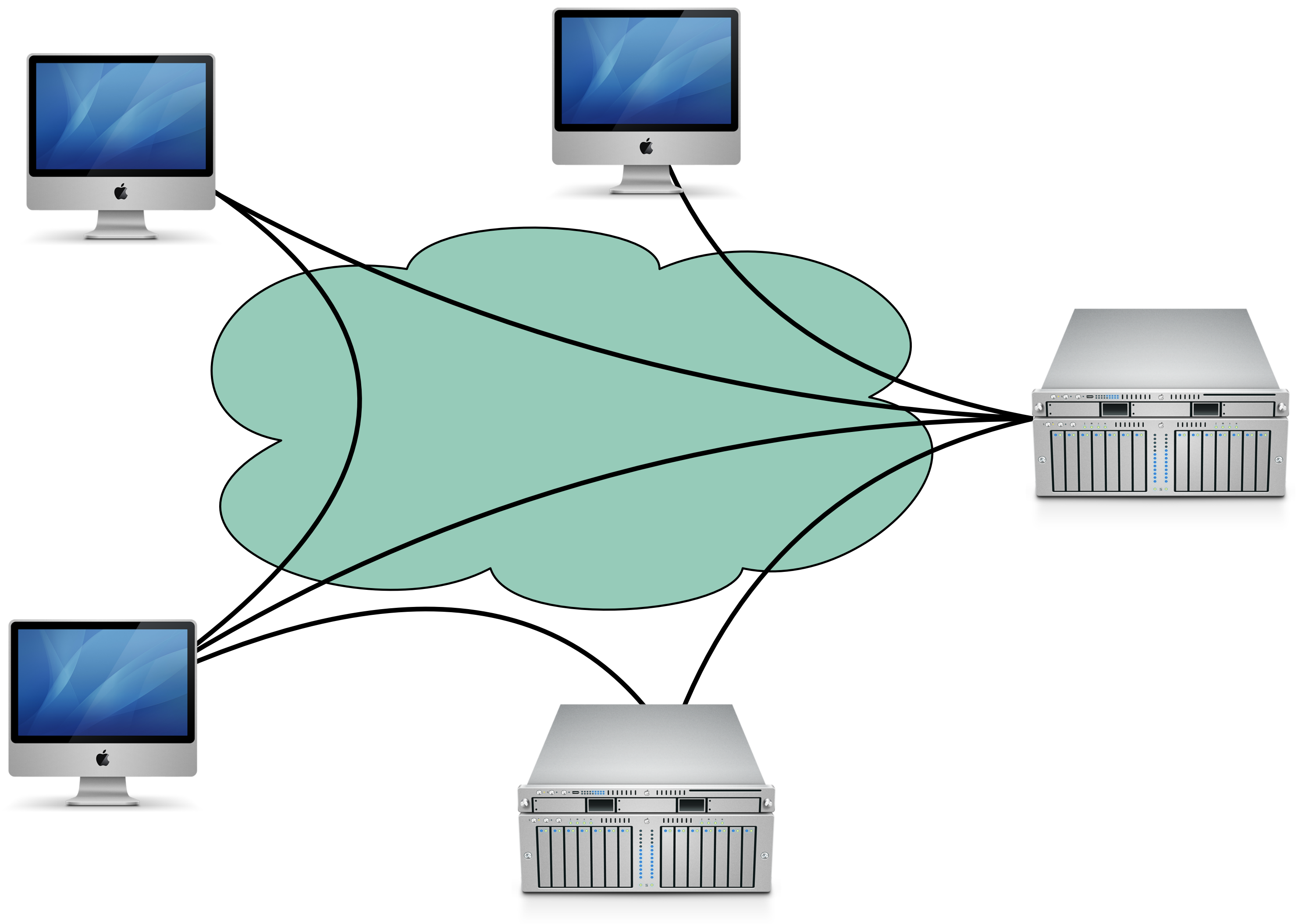
Course Topics

- 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?

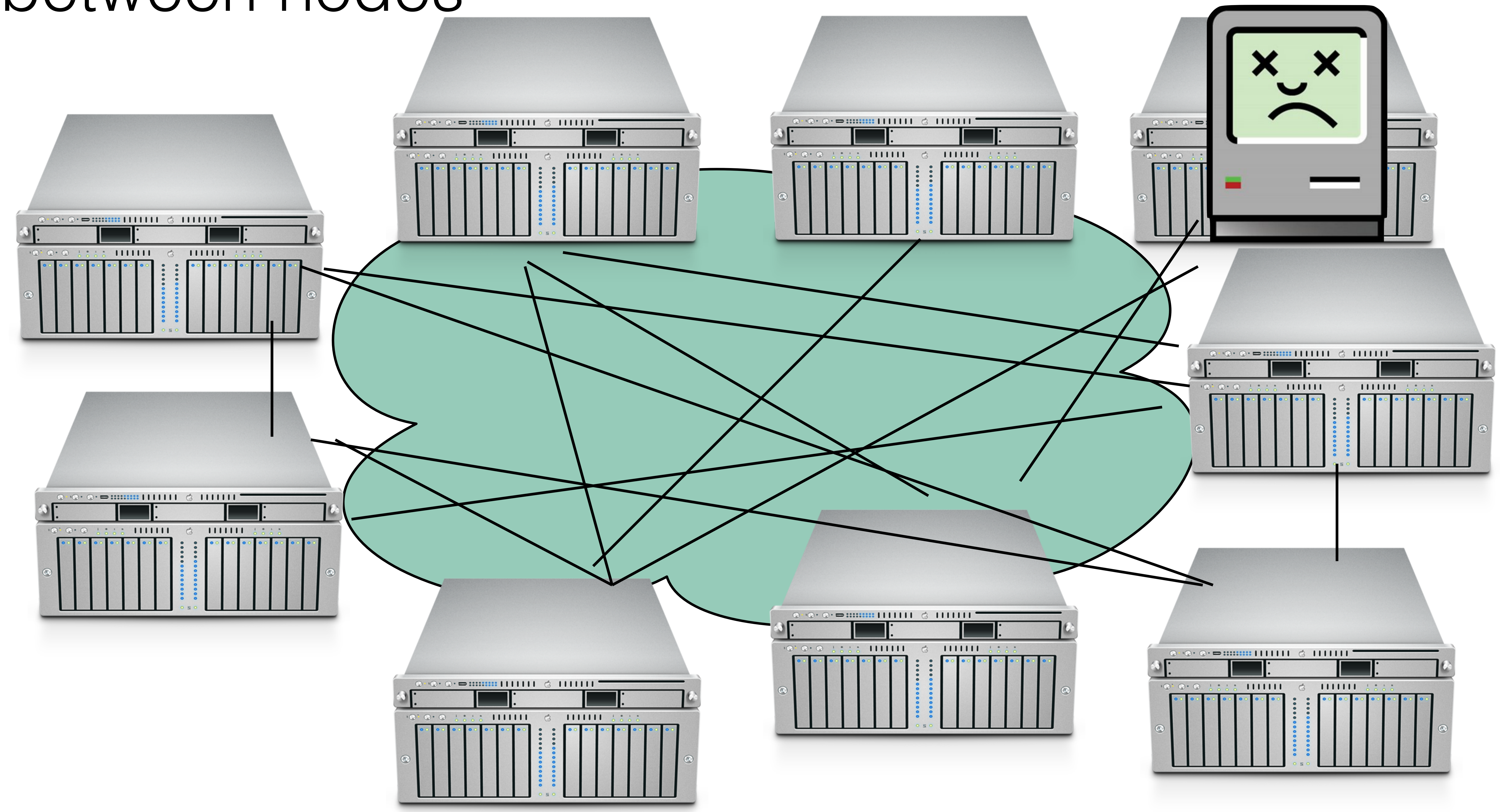
- Scalability
- Performance
- Latency
- Availability
- Fault Tolerance

More machines, more problems

- More machines -> more chance of seeing at least one machine fail
- PLUS, the network may be:
  - Unreliable
  - Insecure
  - Slow
  - Expensive
  - Limited

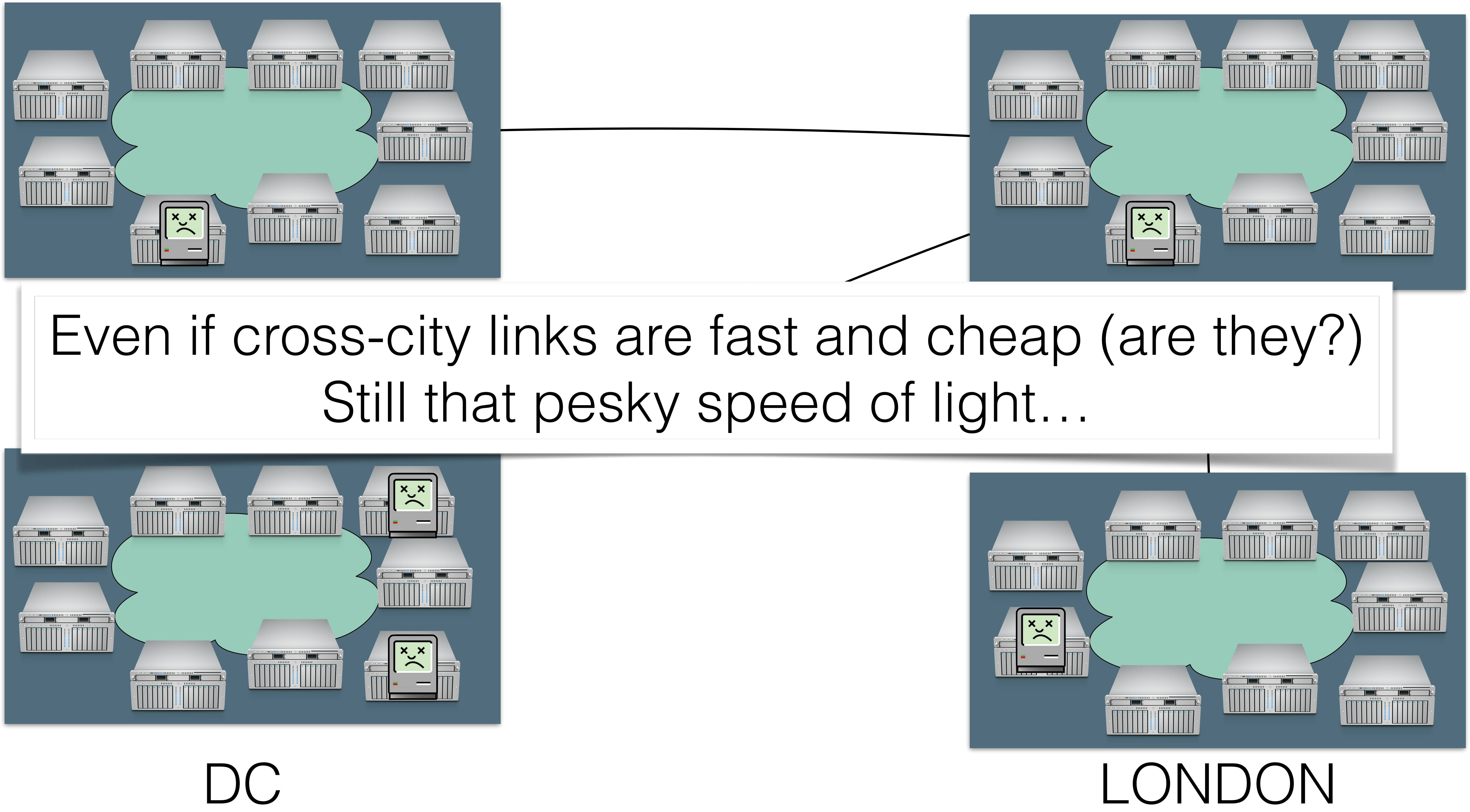
Constraints

- Number of nodes
- Distance between nodes

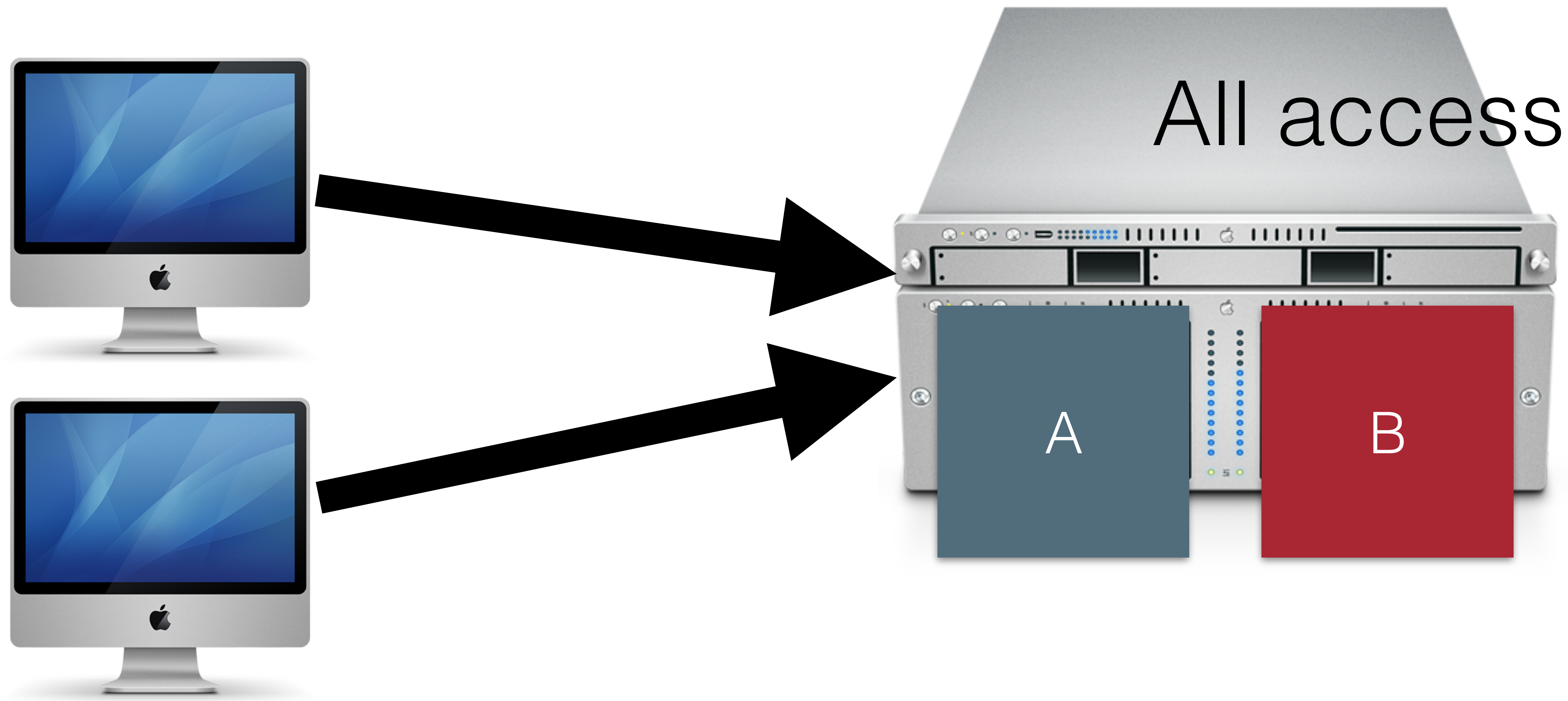


Constraints

- Number of nodes
- Distance between nodes



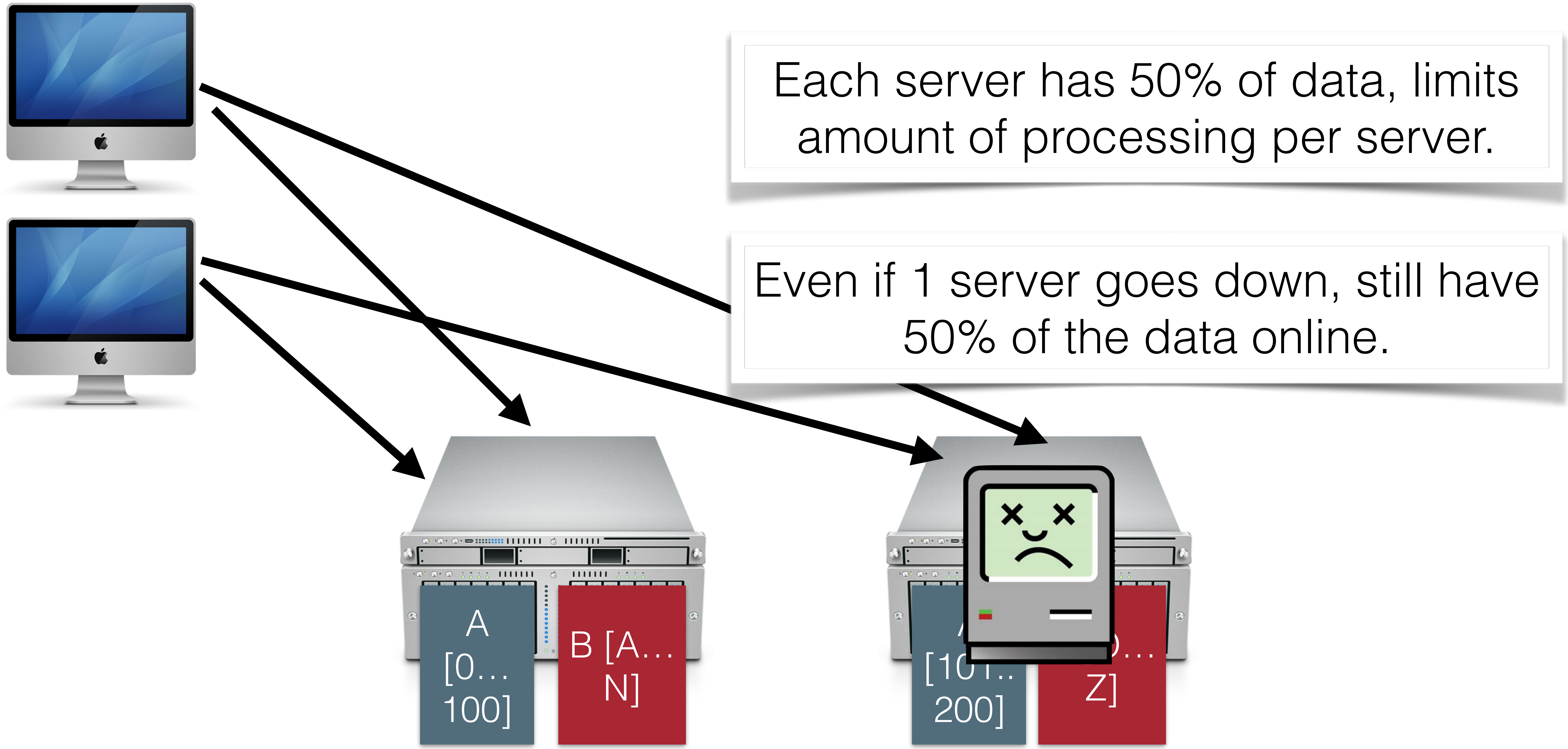
Recurring Solution #1: Partitioning



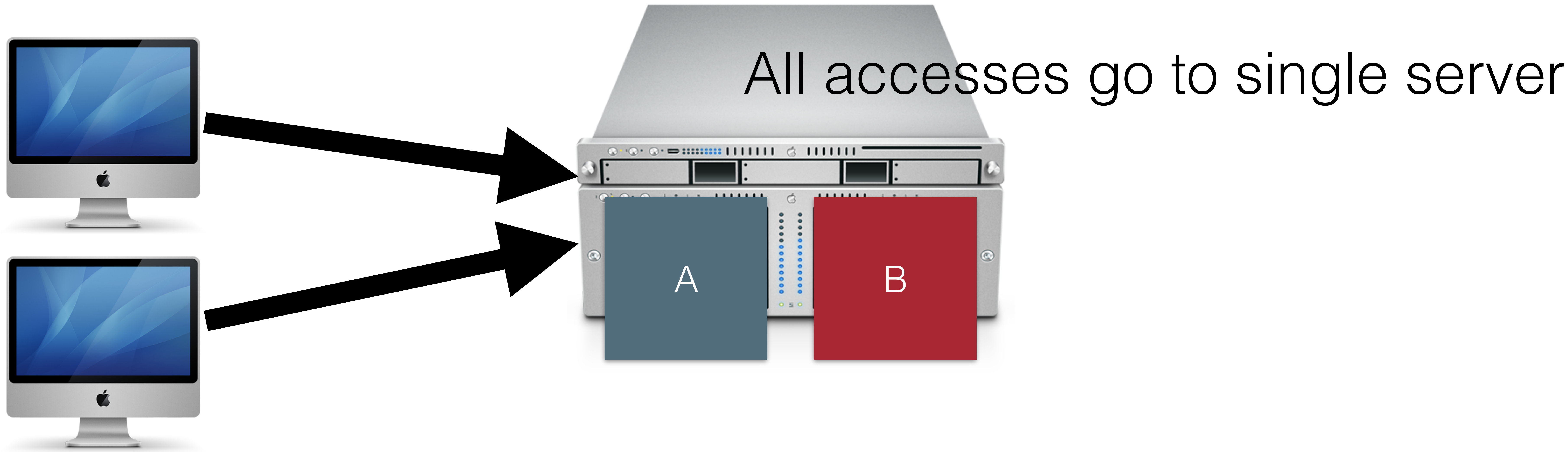
All accesses go to single server

Recurring Solution #1: Partitioning

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

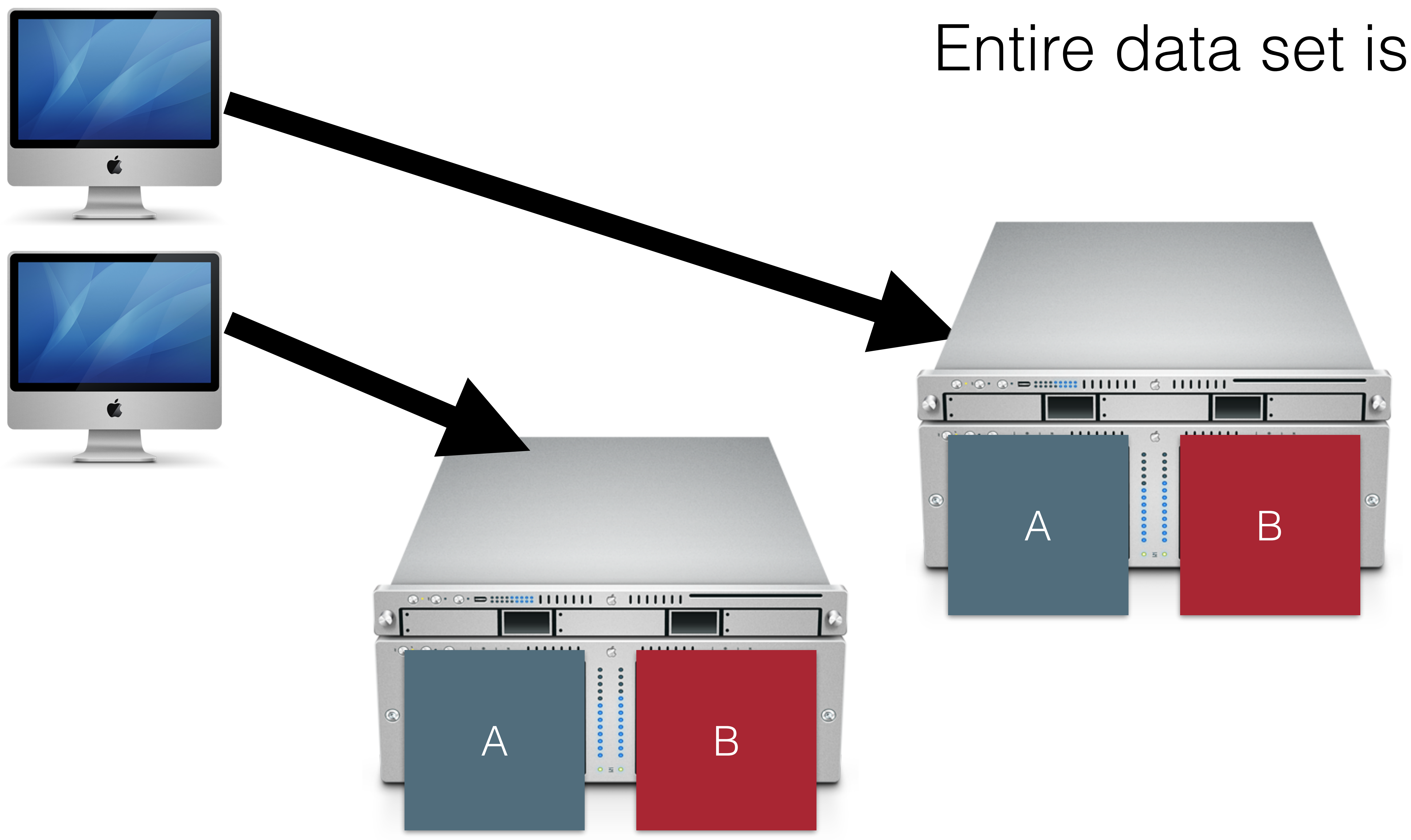


Recurring Solution #2: Replication



Recurring Solution #2: Replication

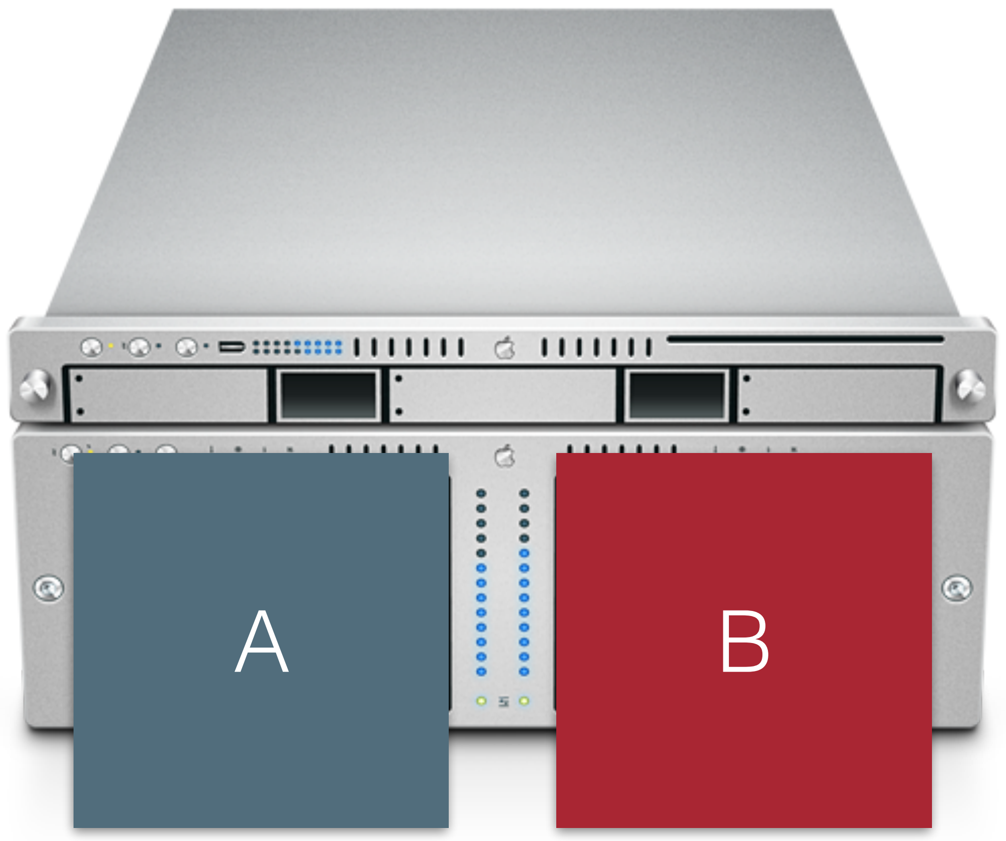
Entire data set is copied



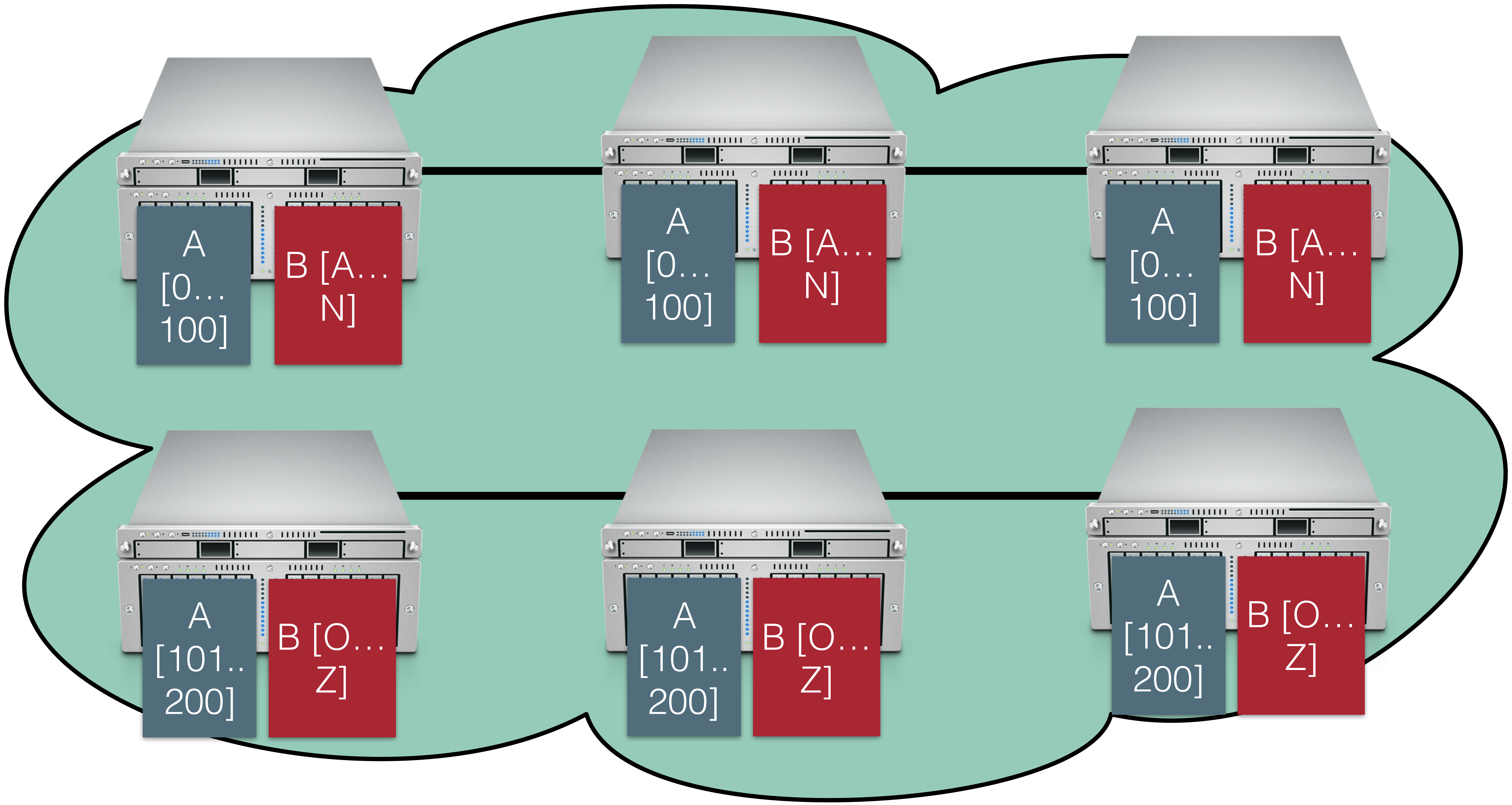
Recurring Solution #2: Replication

- 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

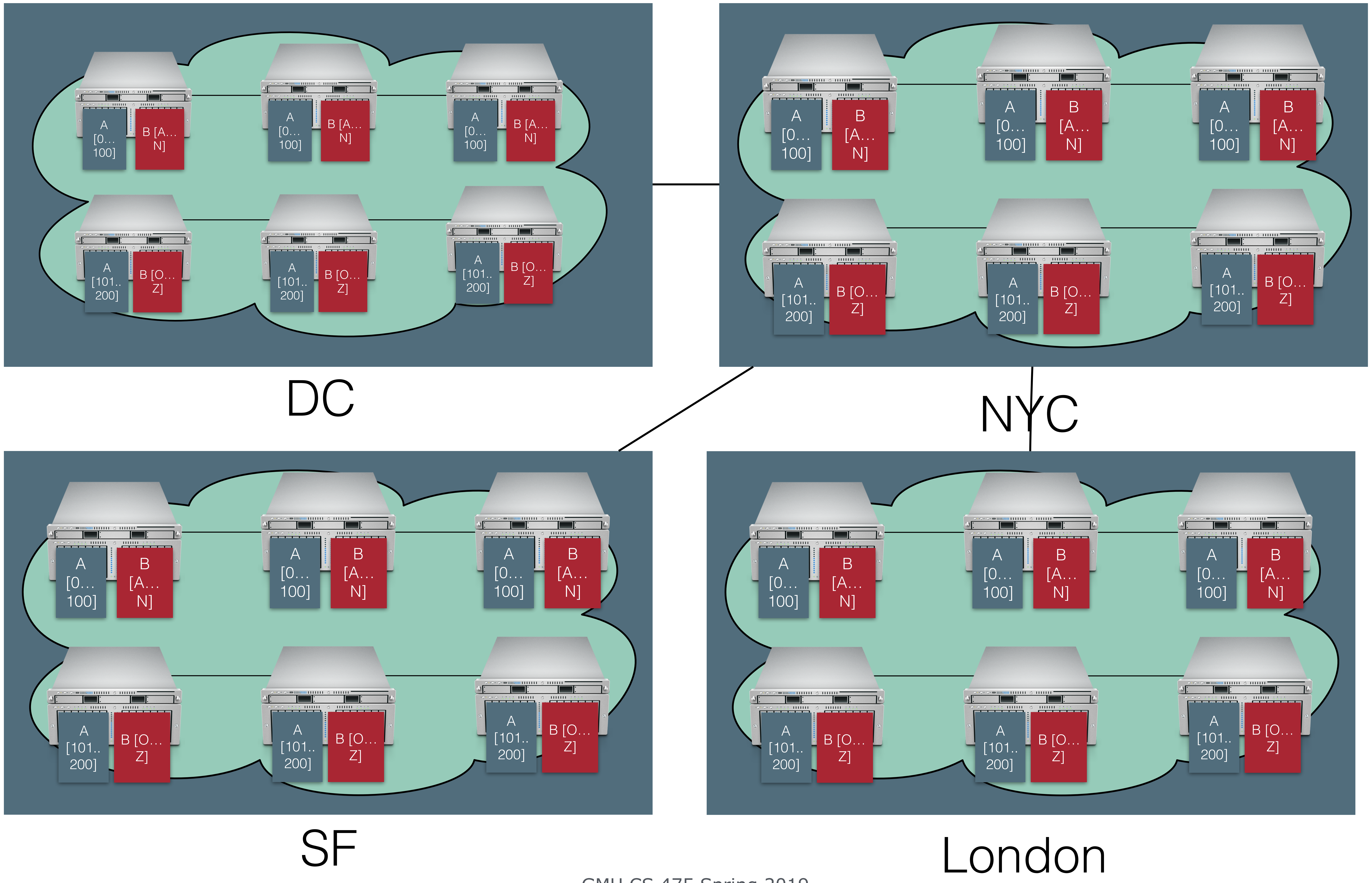
Partitioning + Replication



Partitioning + Replication

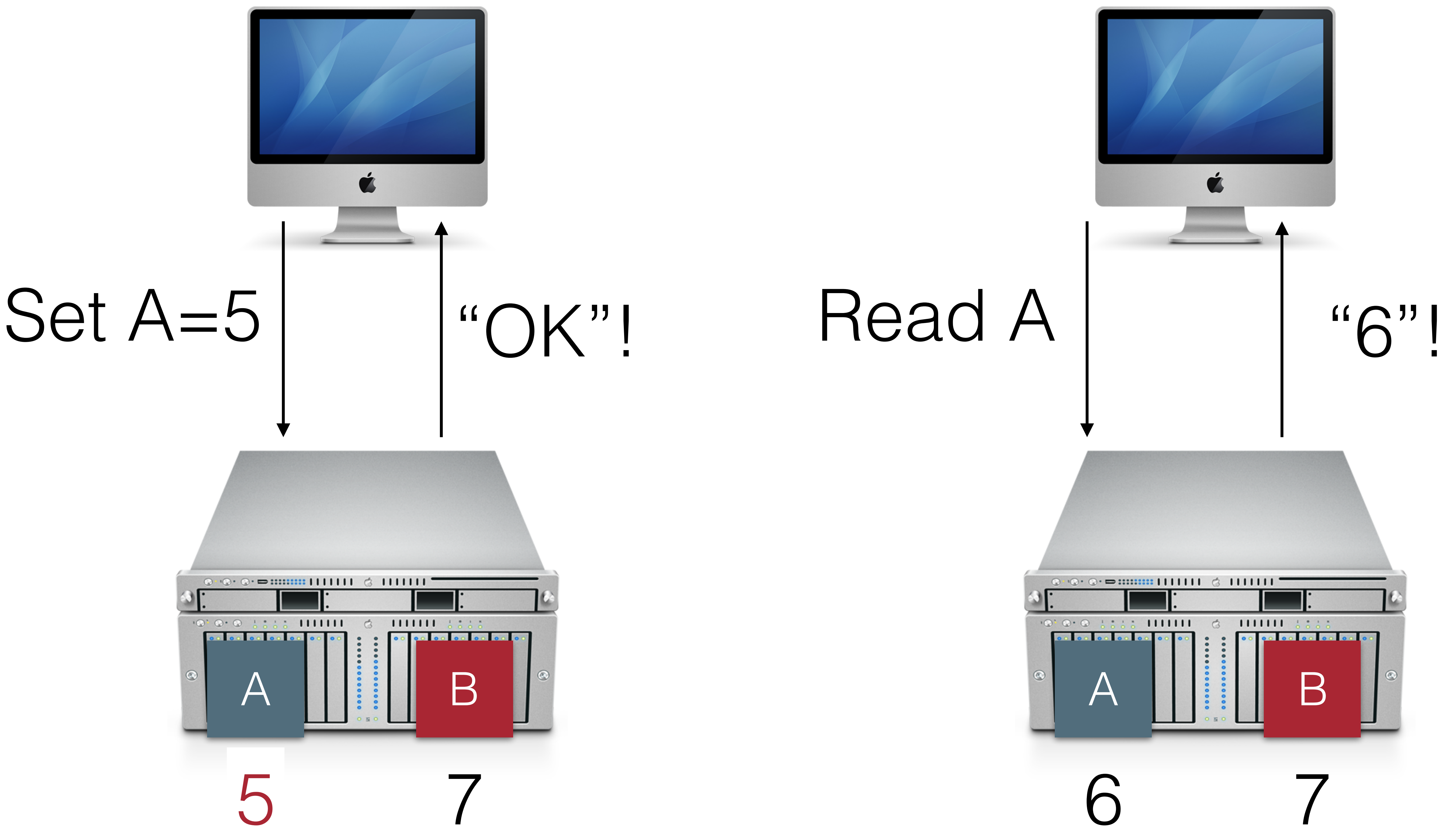


Partitioning + Replication



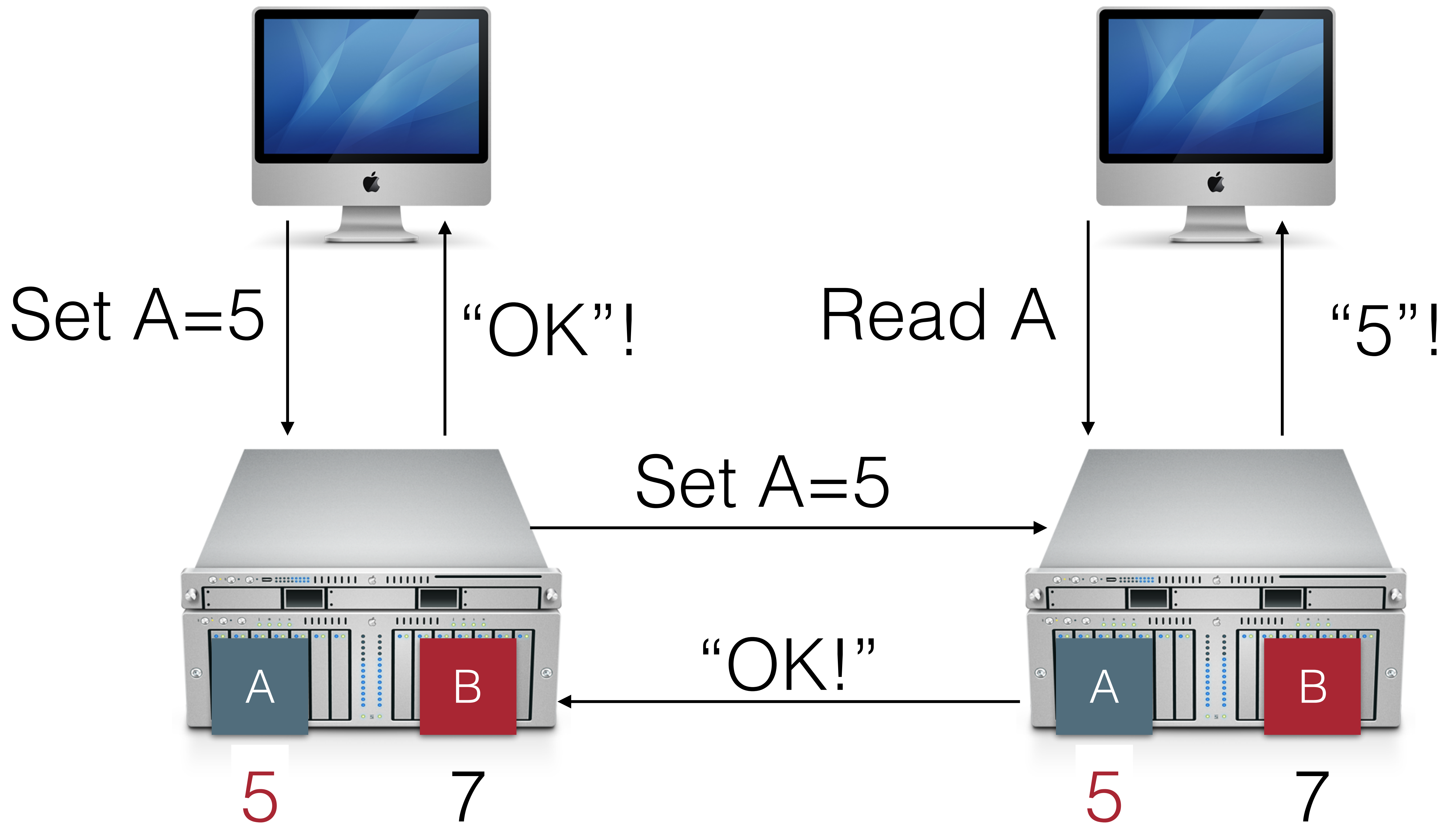
Recurring Problem: Replication

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



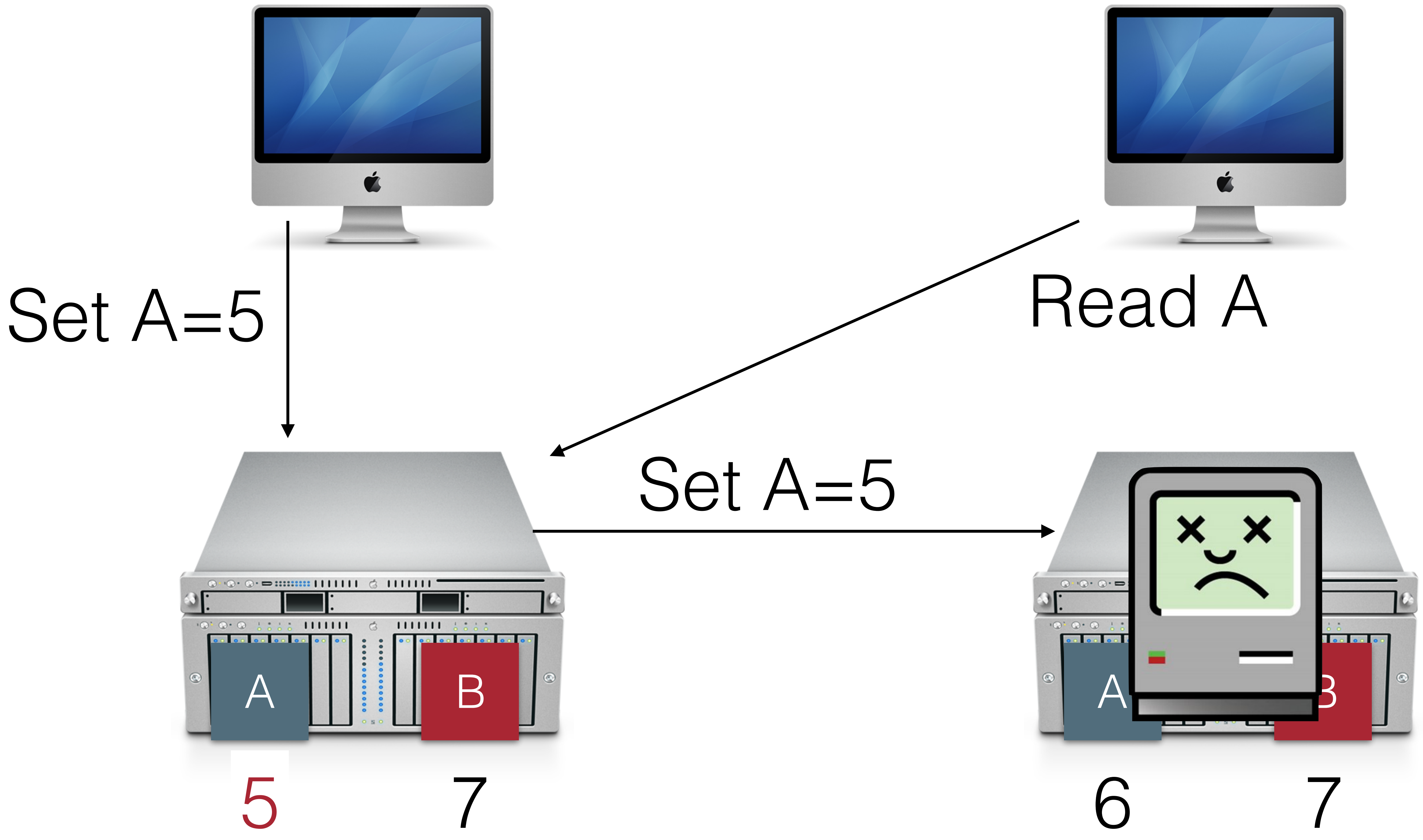
OK, we obviously need to actually do something here to replicate the data… but what?

Sequential Consistency

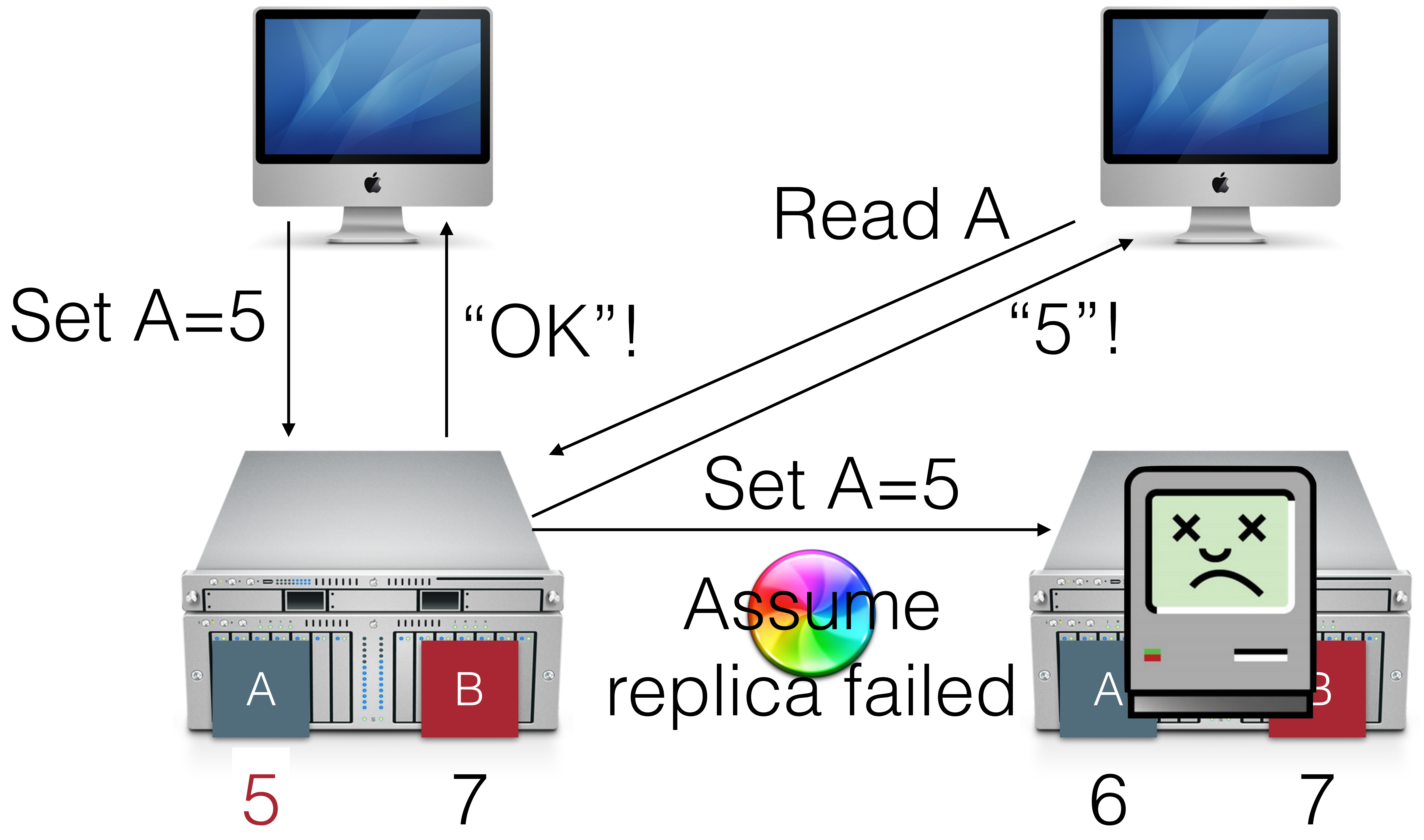


Availability

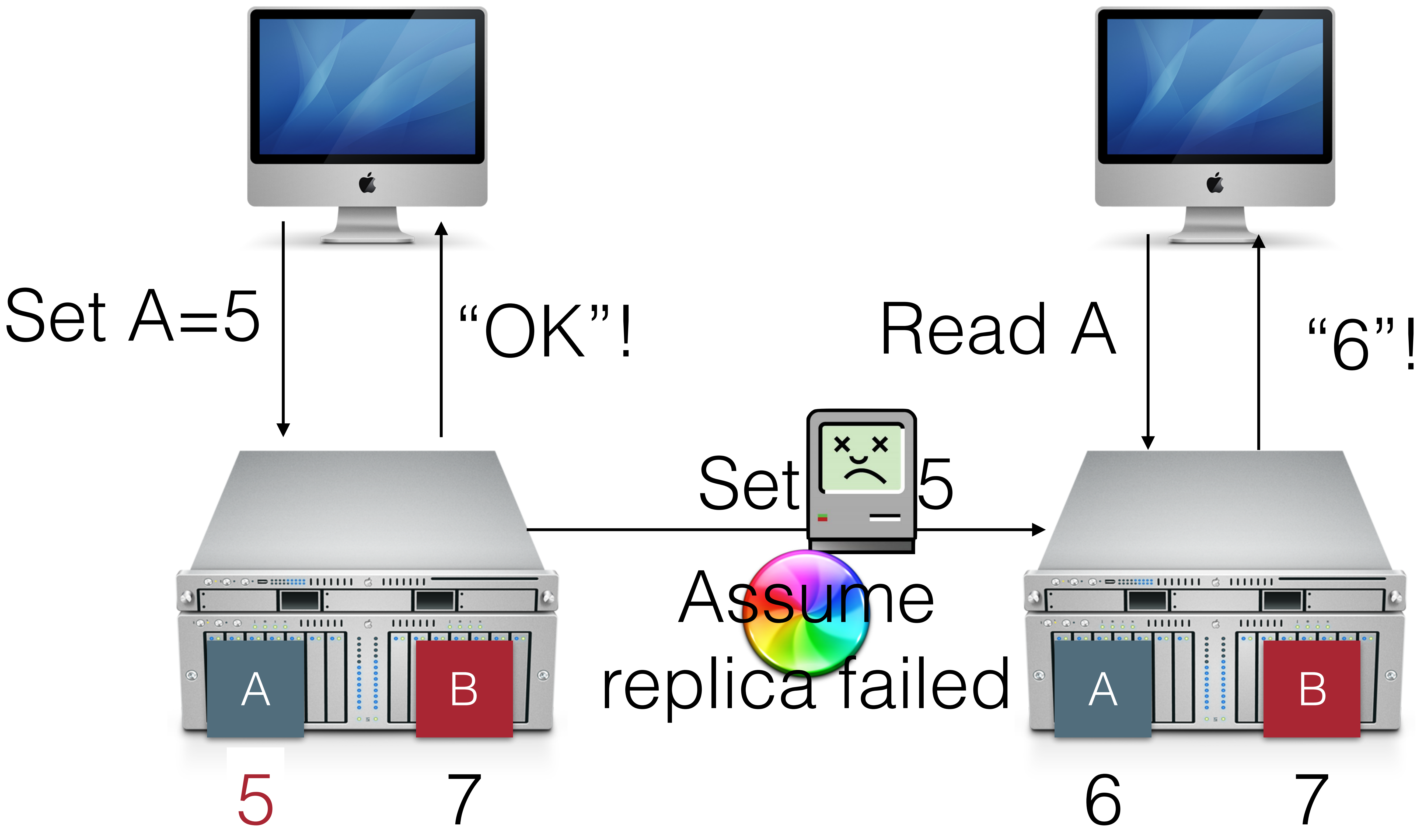
- Our protocol for sequential consistency does NOT guarantee that the system will be available!



Consistent + Available

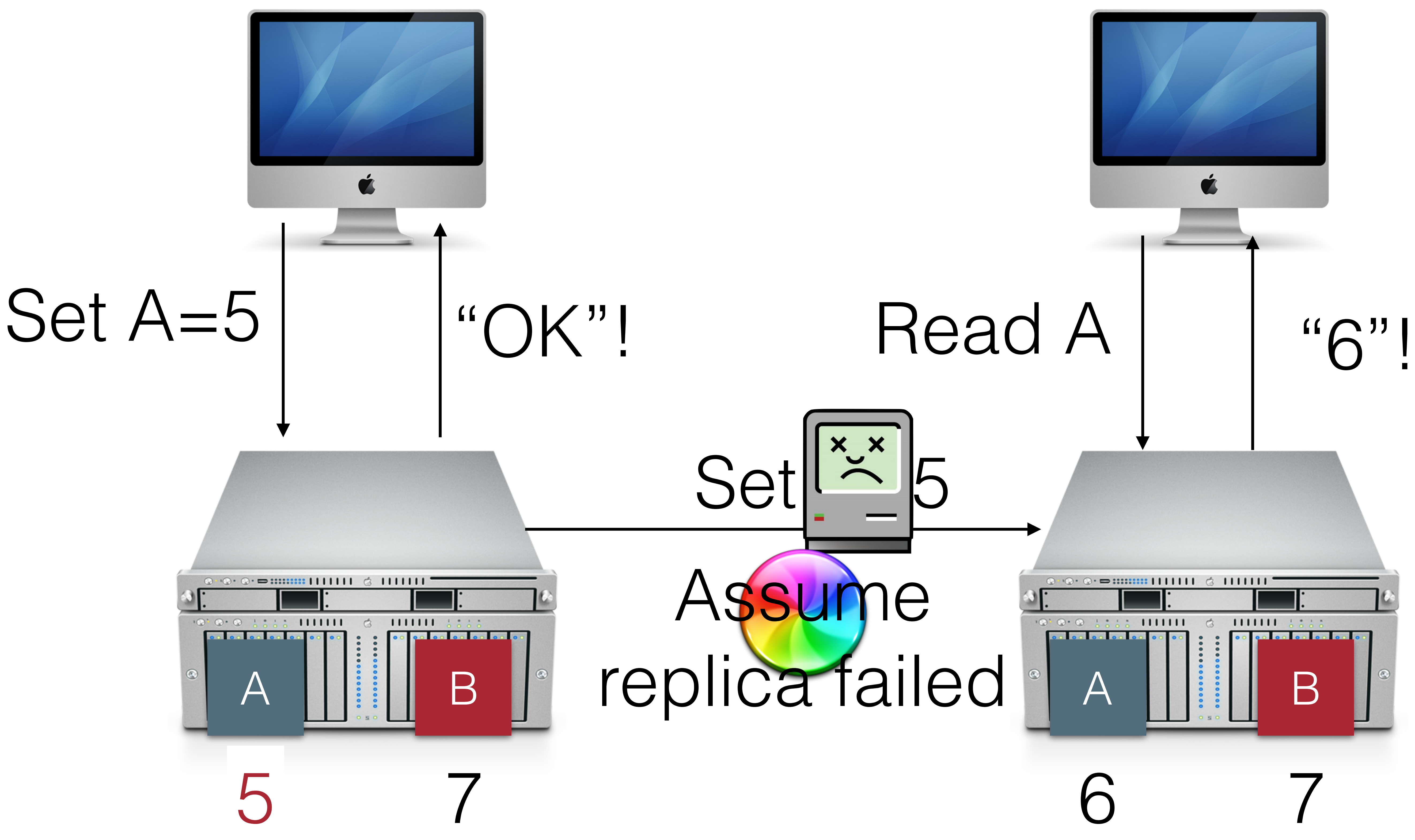


Still broken...

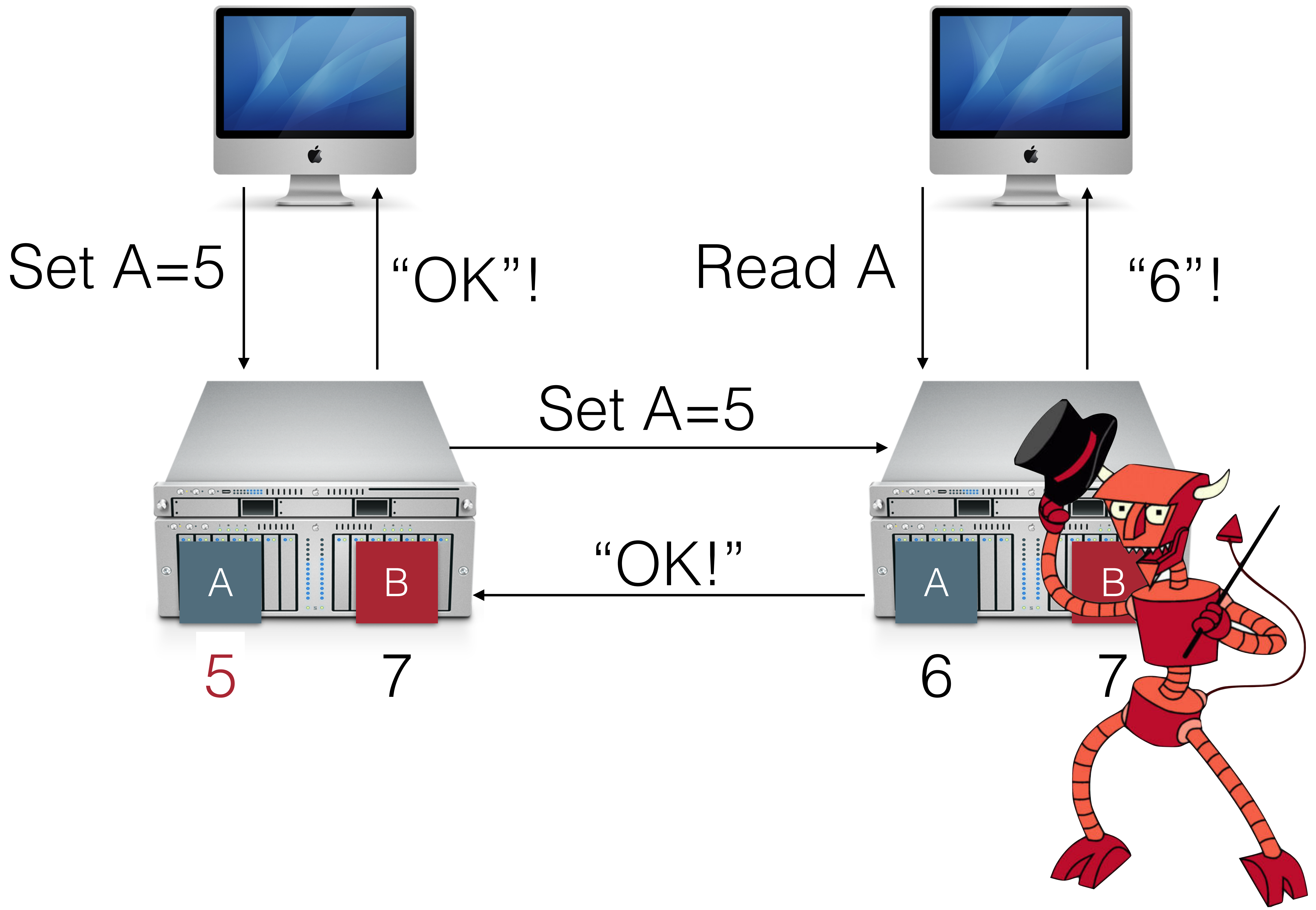


Network Partitions

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



Byzantine Faults



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

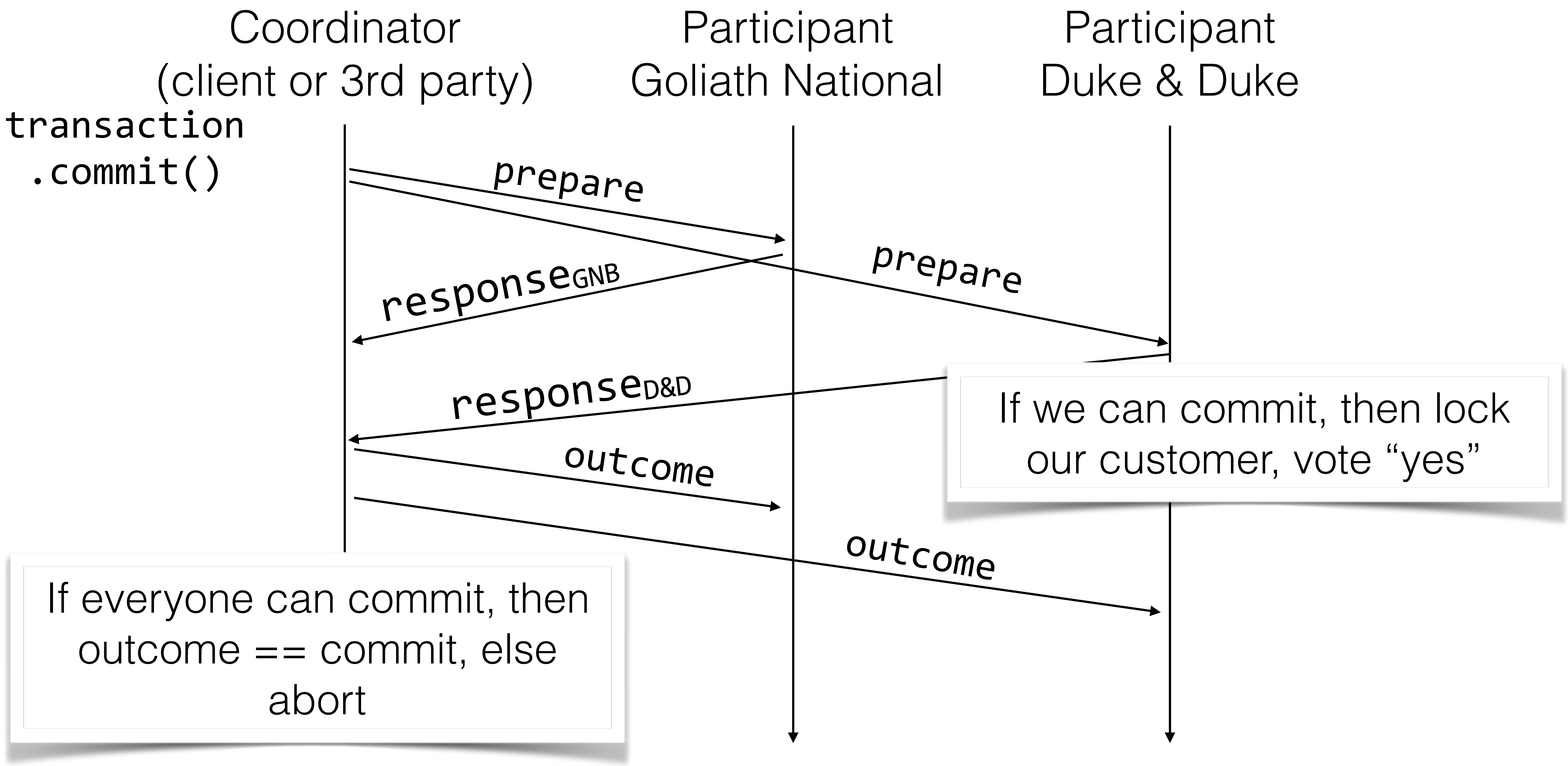
Agreement Generally

- 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

2PC Example

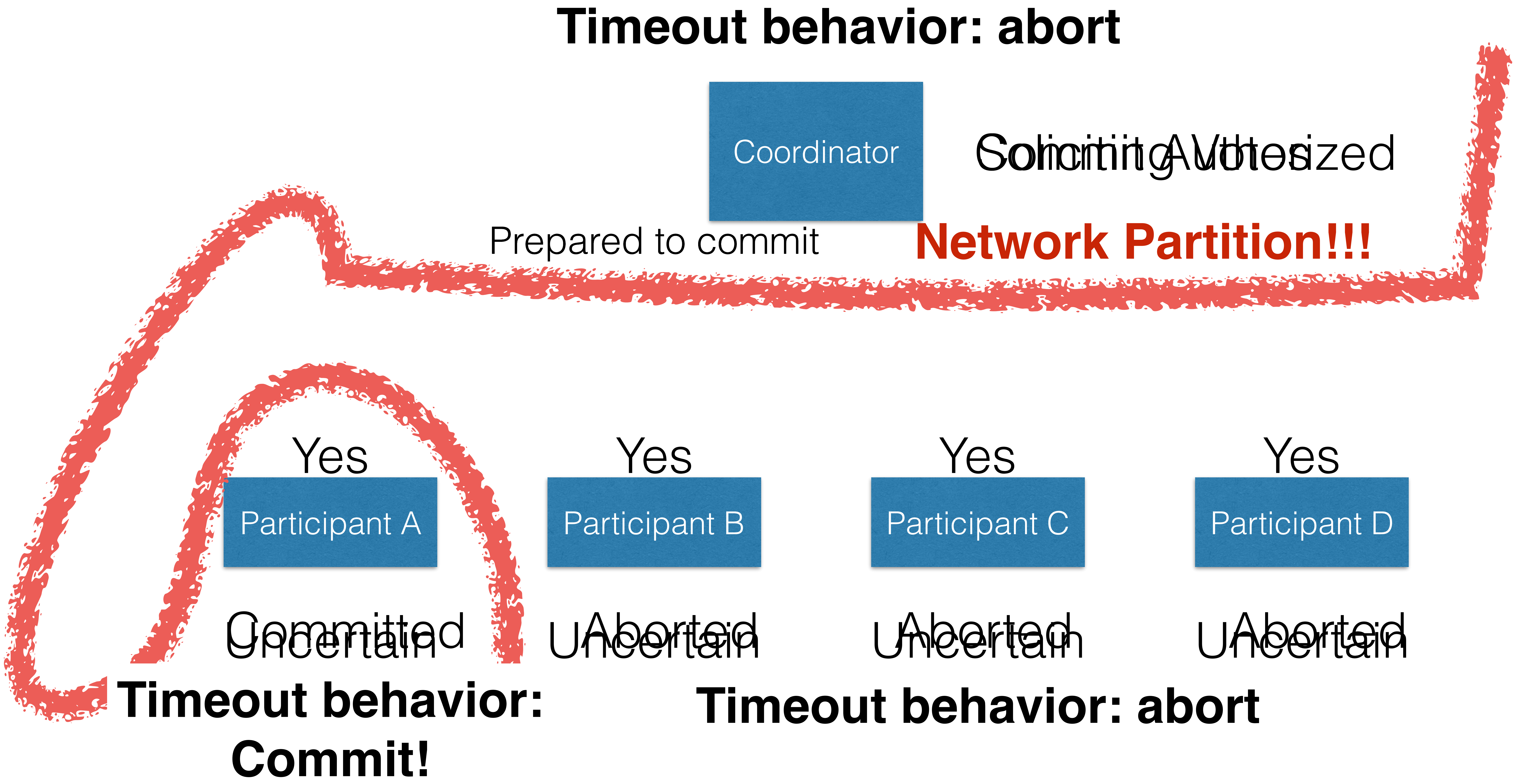

Coordinator
(client or 3rd party)
Participant
Goliath National
Participant
Duke & Duke
transaction
.commit()
prepare
prepare
responseGNB
responseD&D
outcome
outcome
If we can commit, then lock our customer, vote “yes”
If everyone can commit, then outcome == commit, else abort

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!

Partitions

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



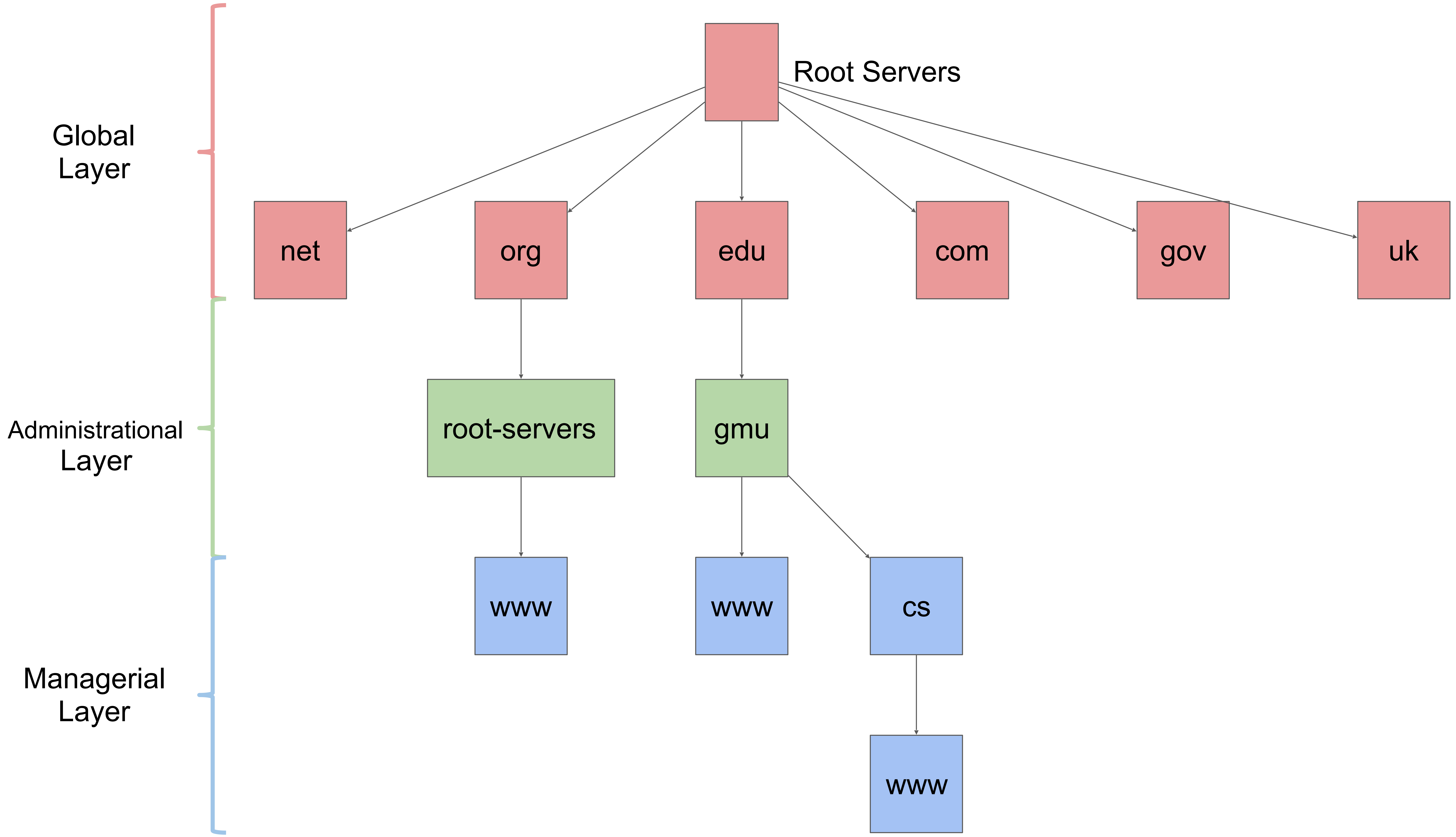
Can we fix it?

- 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

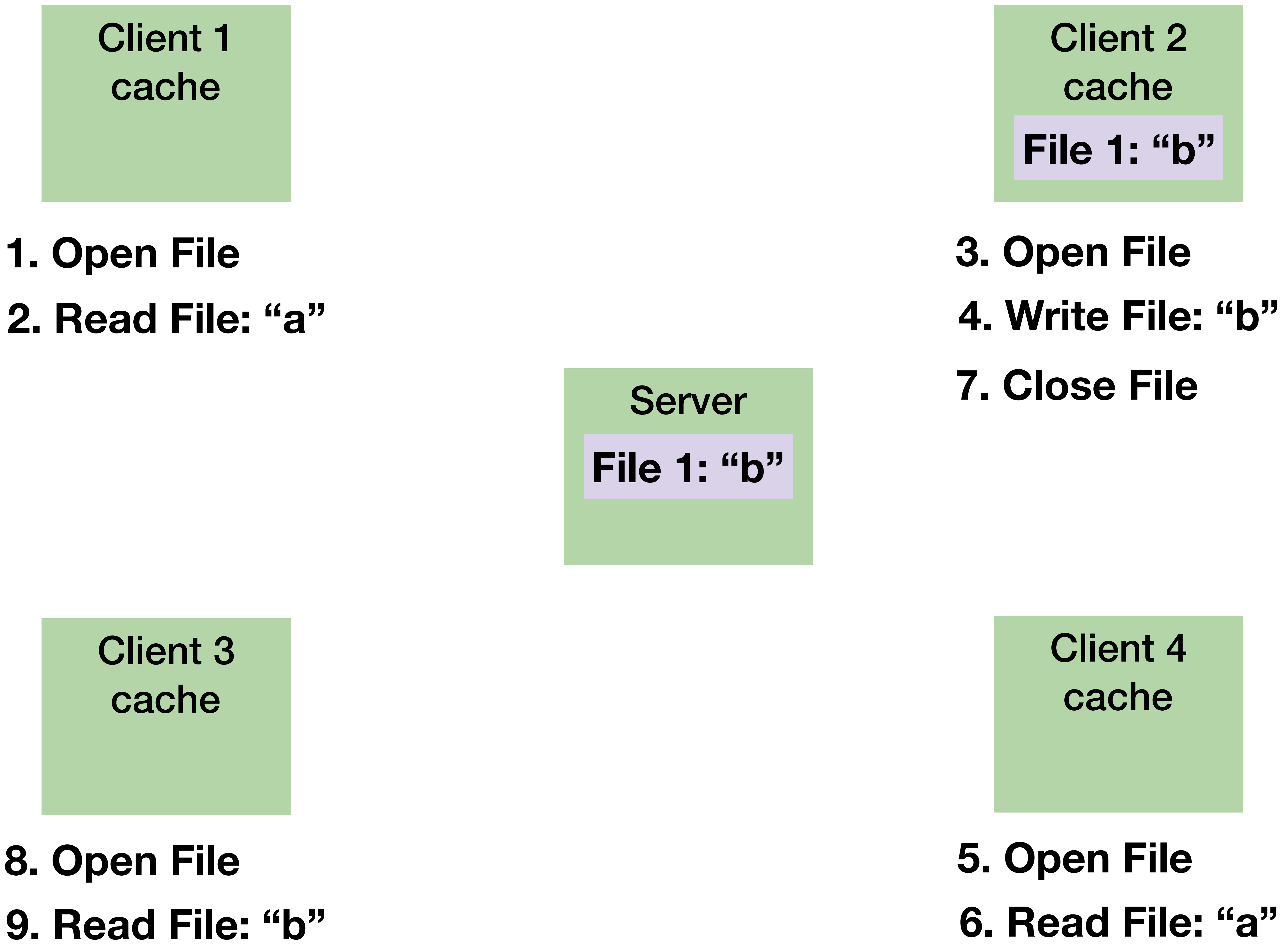
FLP - Intuition

- 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)

Domain Name System

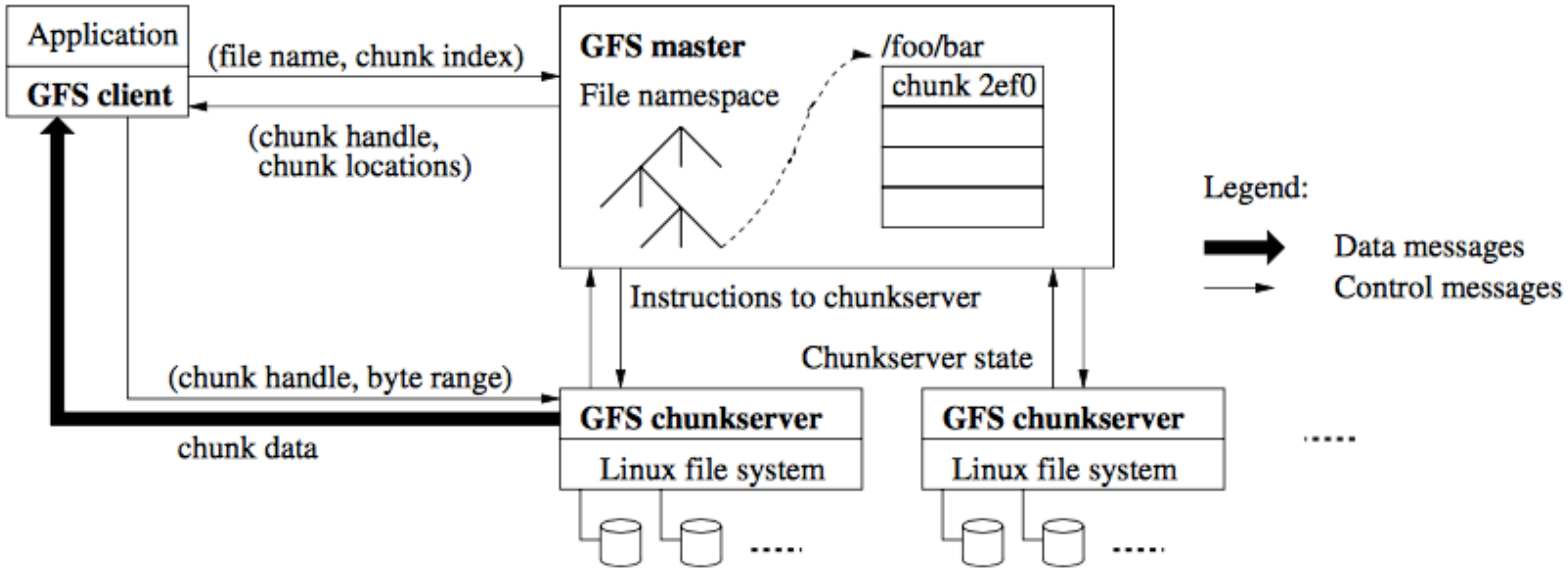


NFS Caching - Close-to-open



Note: in practice, client caches periodically check server to see if still valid

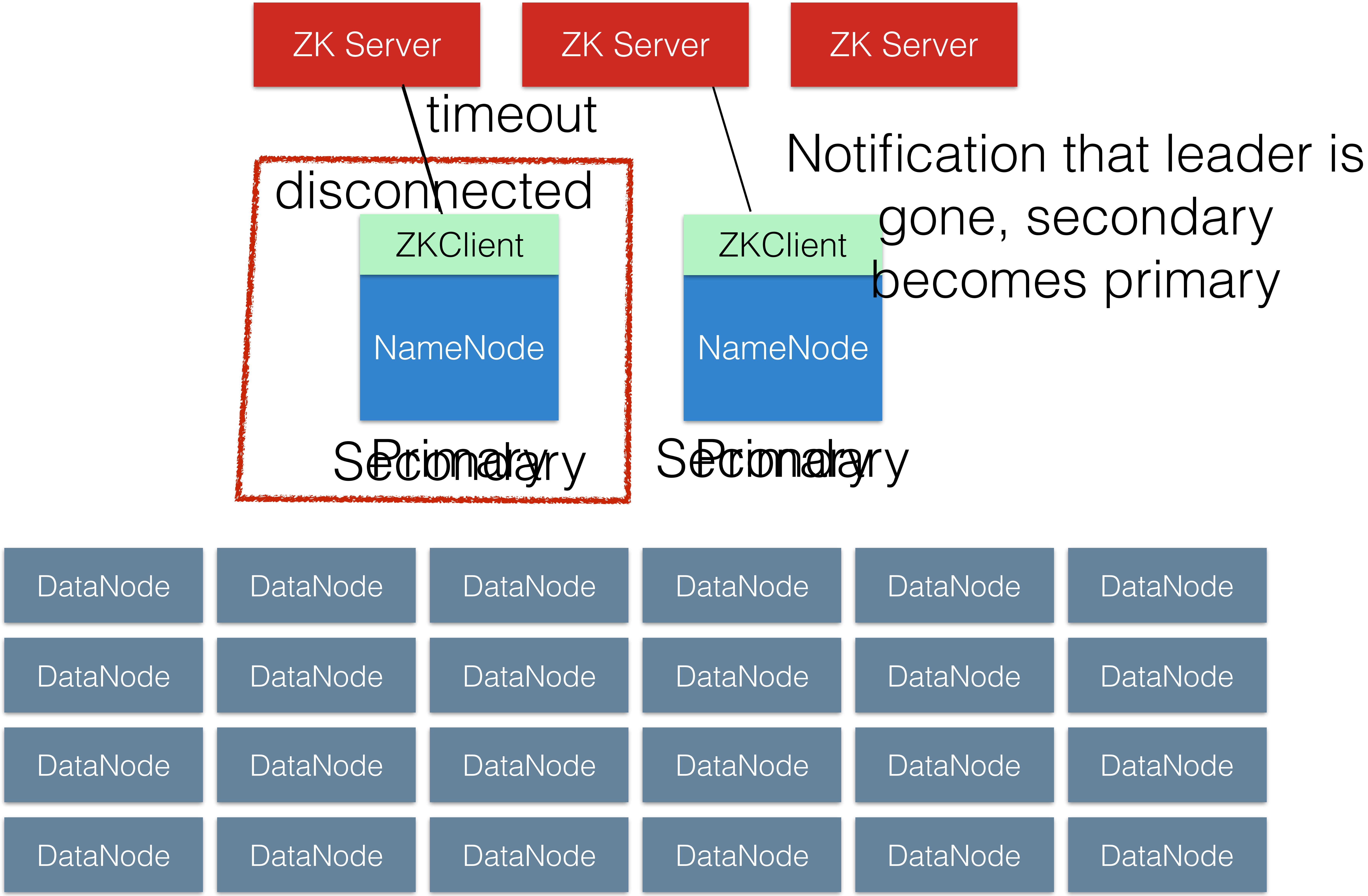
GFS Architecture



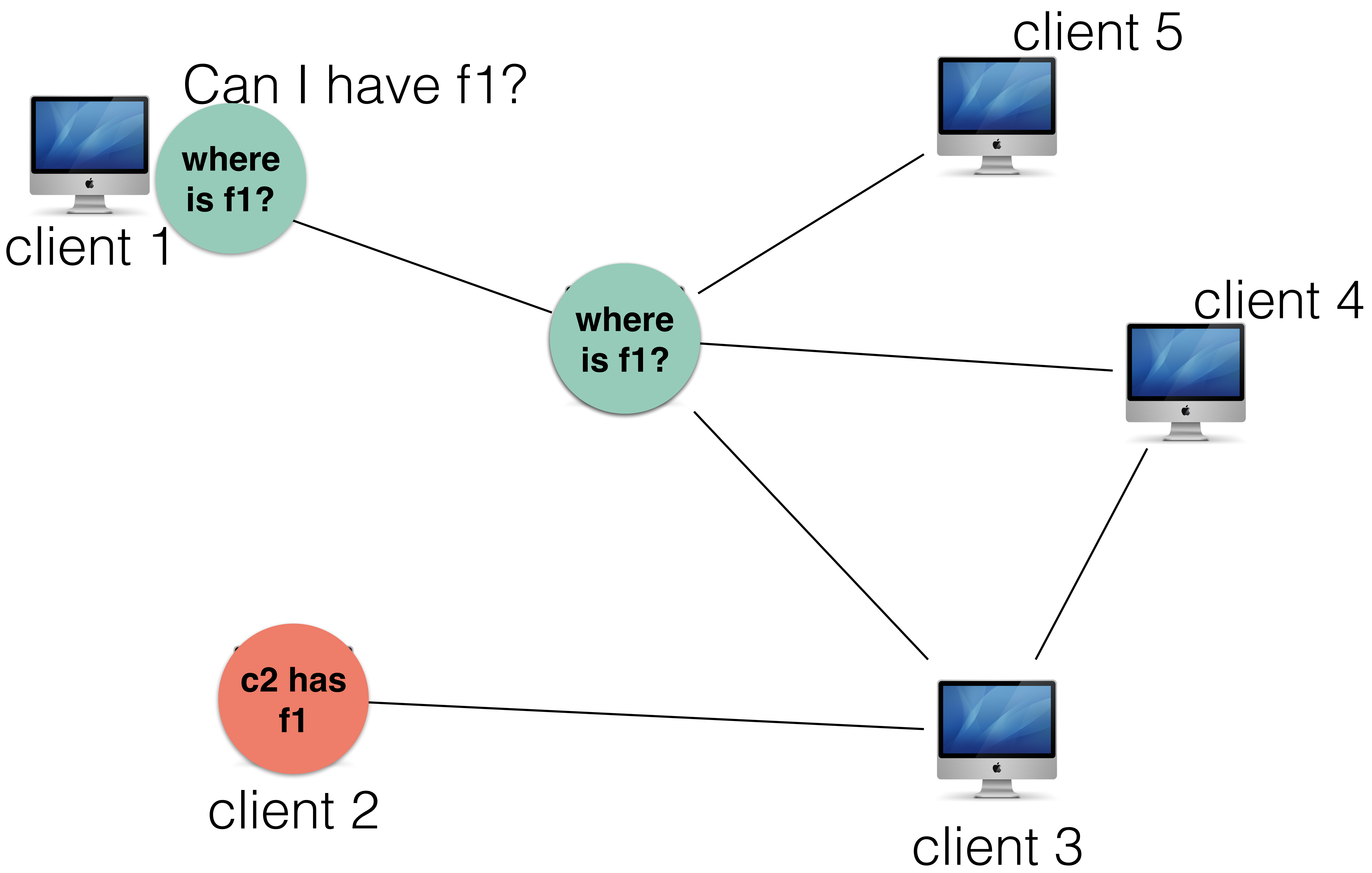
ZooKeeper - Guarantees

- **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

Hadoop + ZooKeeper



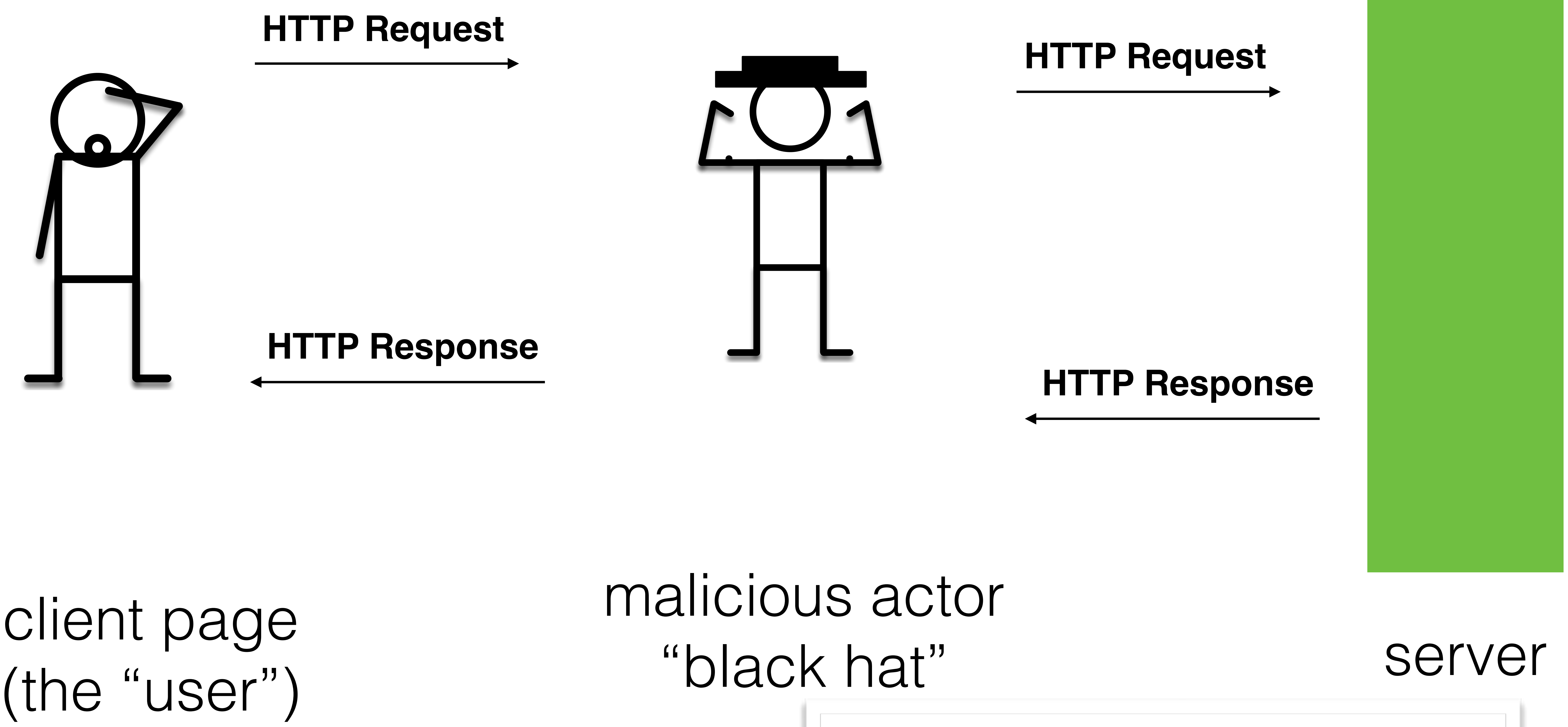
Gnutella 1.0



Can I have f1?
where is f1?
client 1
where is f1?
client 5
client 4
c2 has f1
client 2
client 3

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?