Peer to Peer

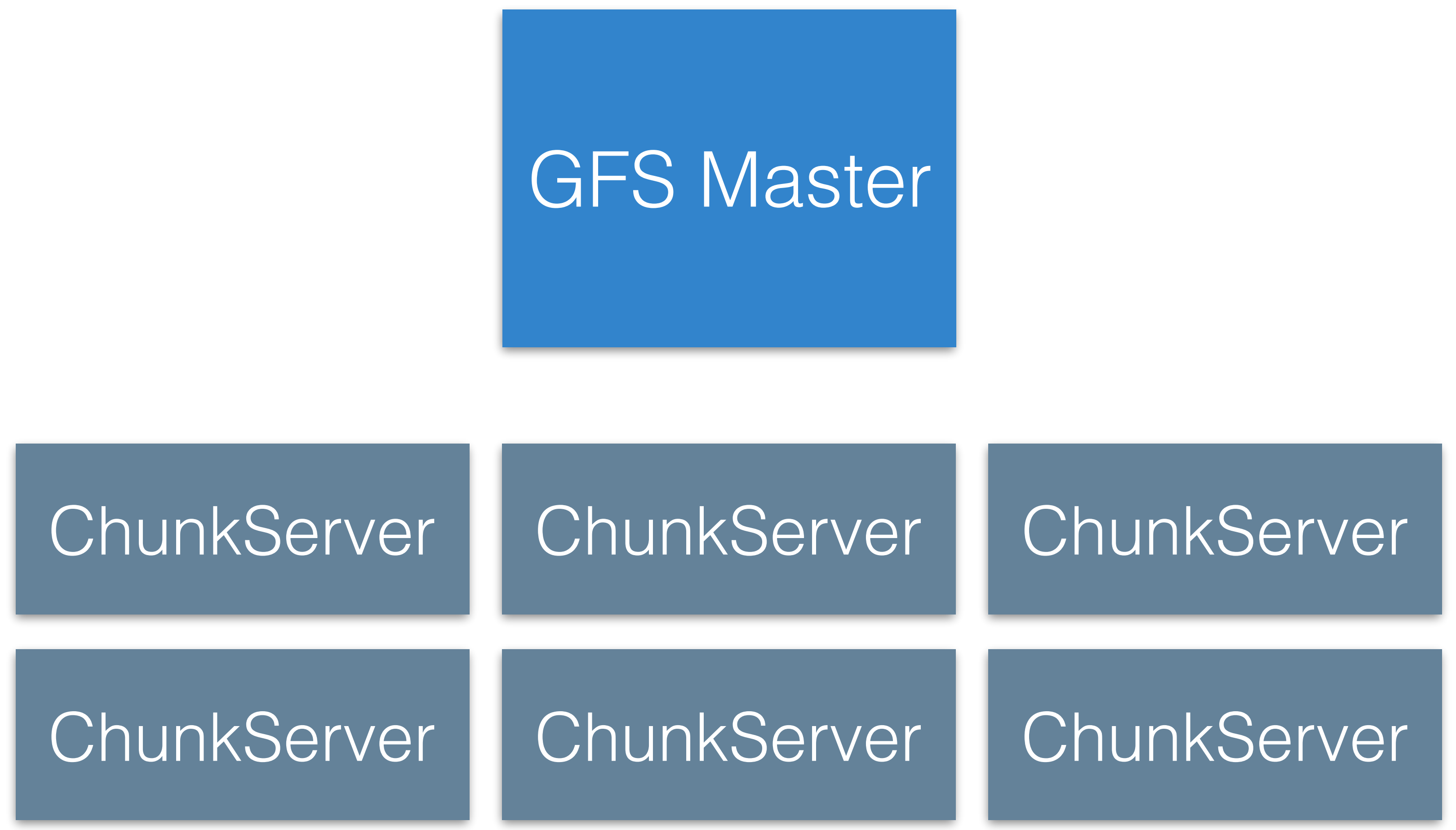
CS 475, Fall 2019
Concurrent & Distributed Systems

Review: Locating Data

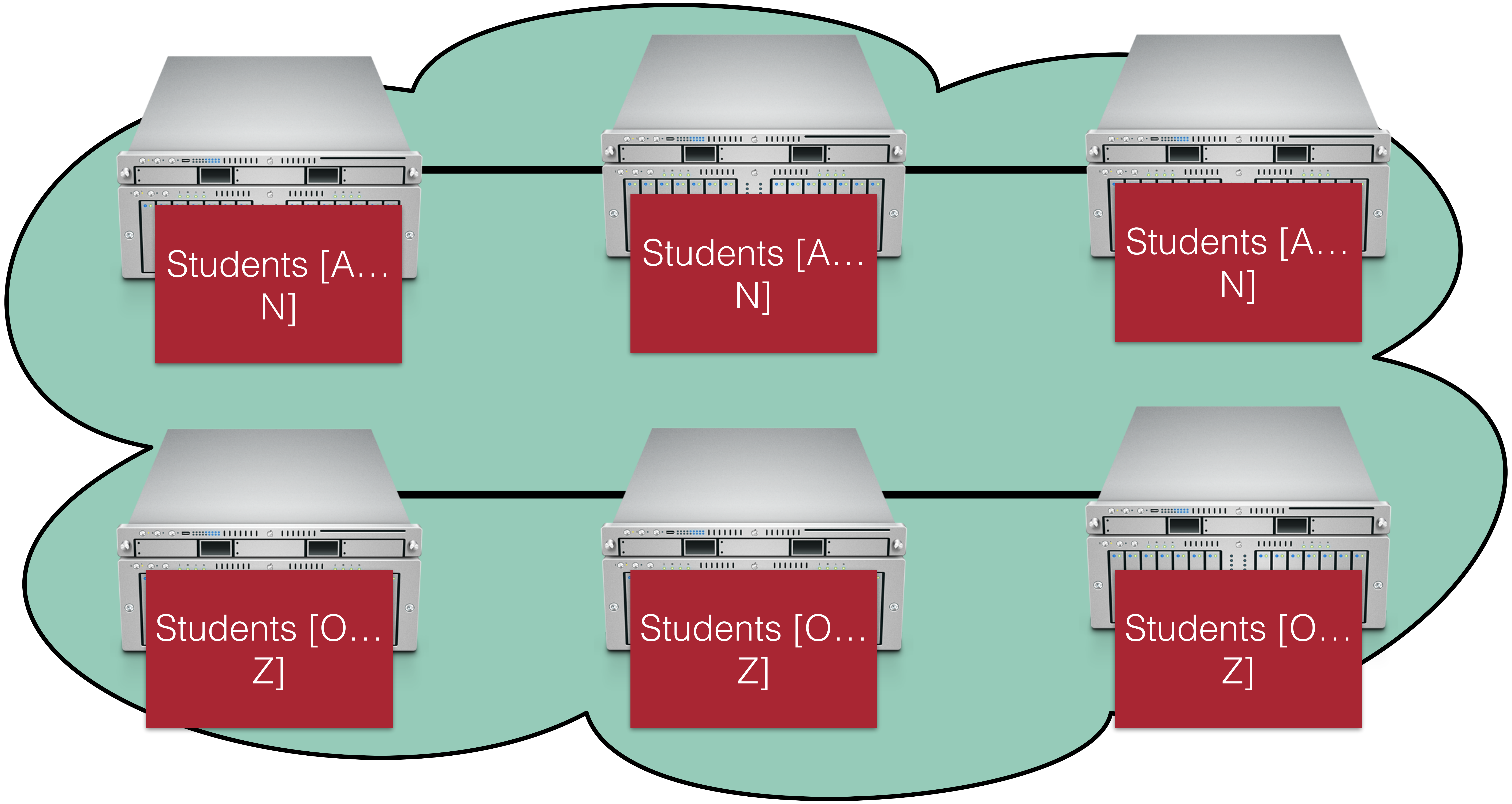
- How do we find data?
- Every answer so far has required some sort of central server
  - DNS lets us resolve names, going through the root servers
  - GFS lets us find chunks that match to files, but need to go through master server
- Why not use the central server to find data?

Review: Why not use a central server to find data?

- Central server is:
  - Point of failure
  - Performance bottleneck
  - Requires bootstrapping

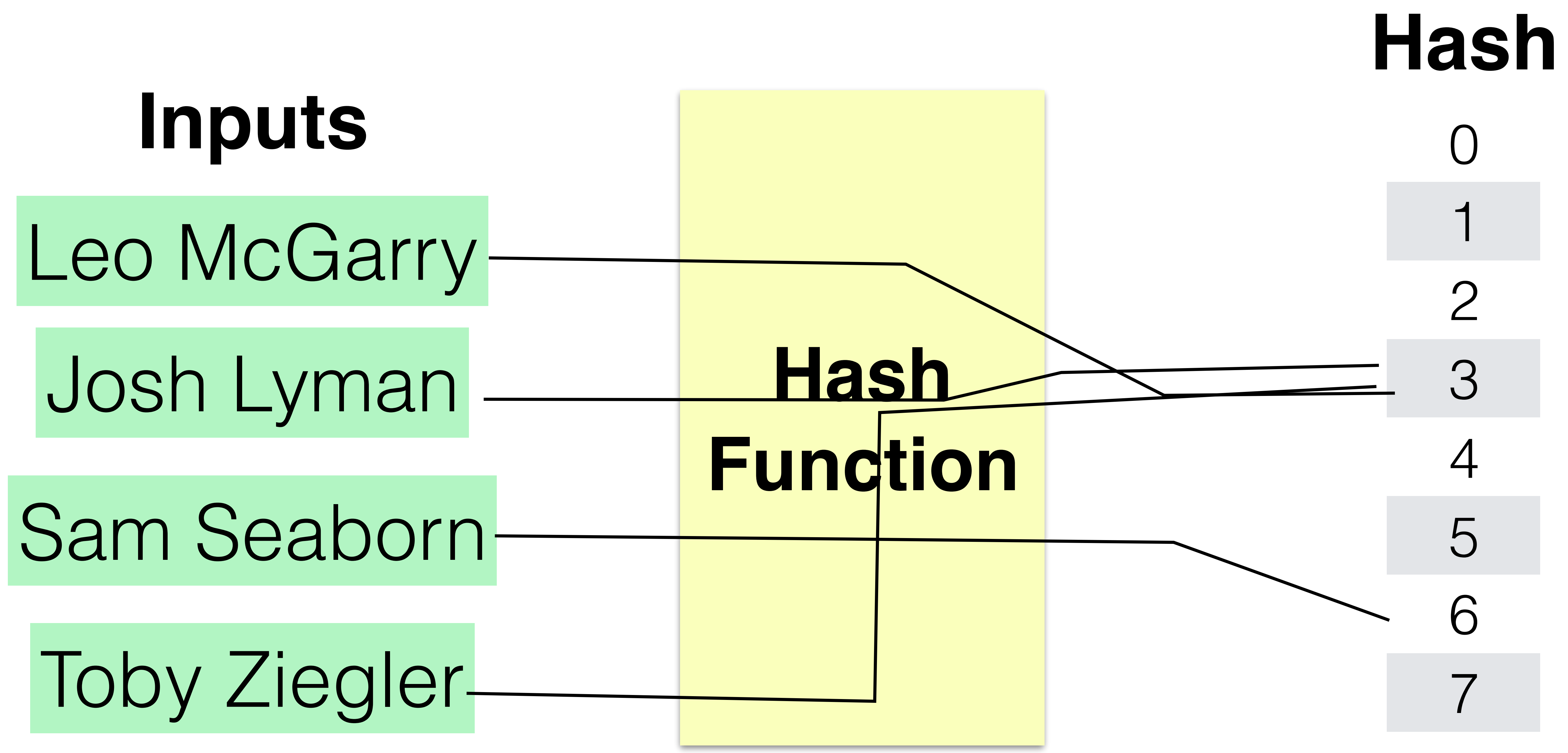


Review: Strawman: Sharding (Partitioning by Key)



Review: Hashing

- The last one mapped every input to a different hash
- Doesn't have to, could be collisions



Review: Consistent Hashing

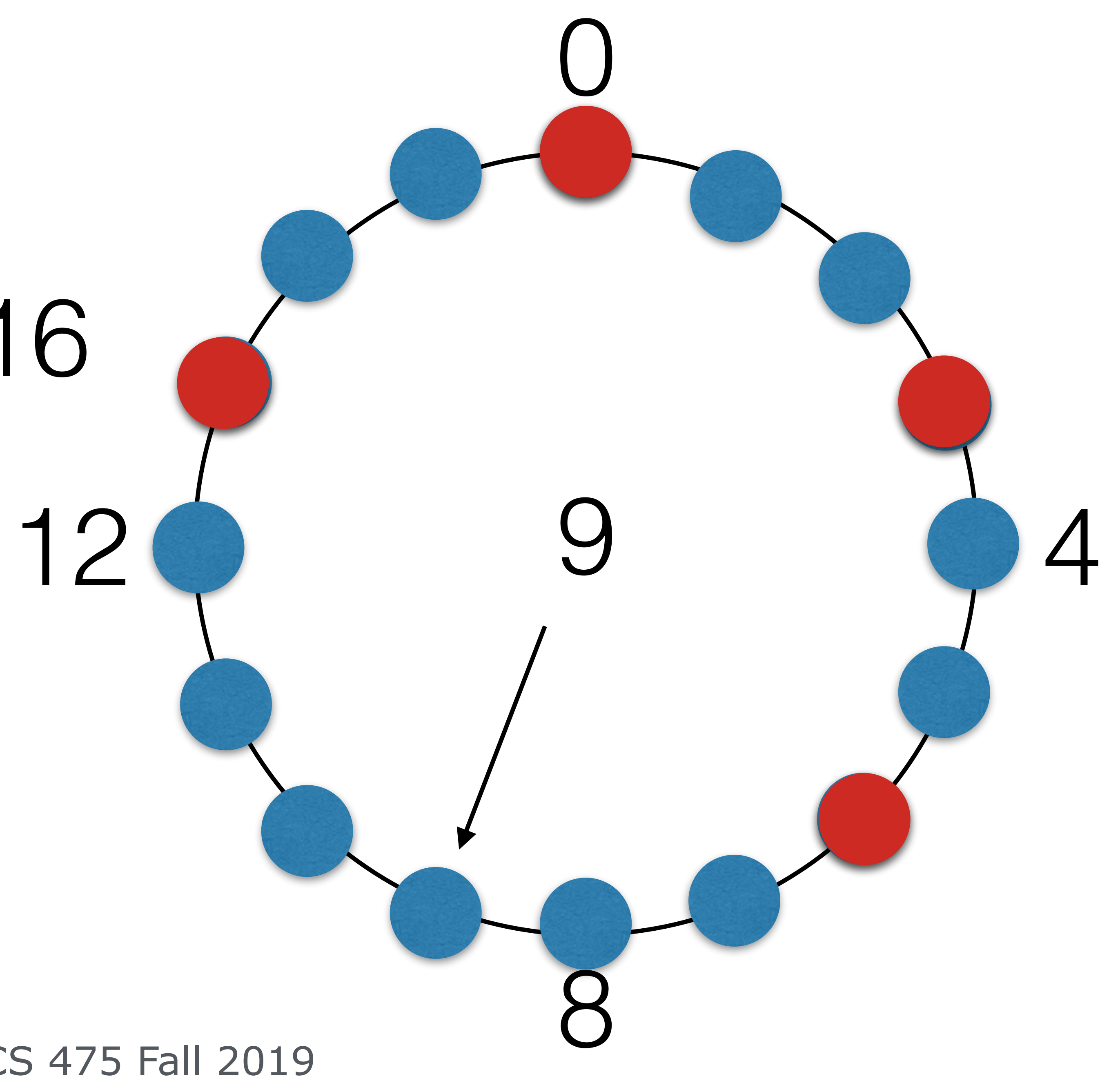
- Construction:
  - Assign each of C hash buckets to random points on mod 2^n circle, where hash key size = n
  - Map object to pseudo-random position on circle
  - Hash of object is the closest clockwise bucket

Example: hash key size is 16

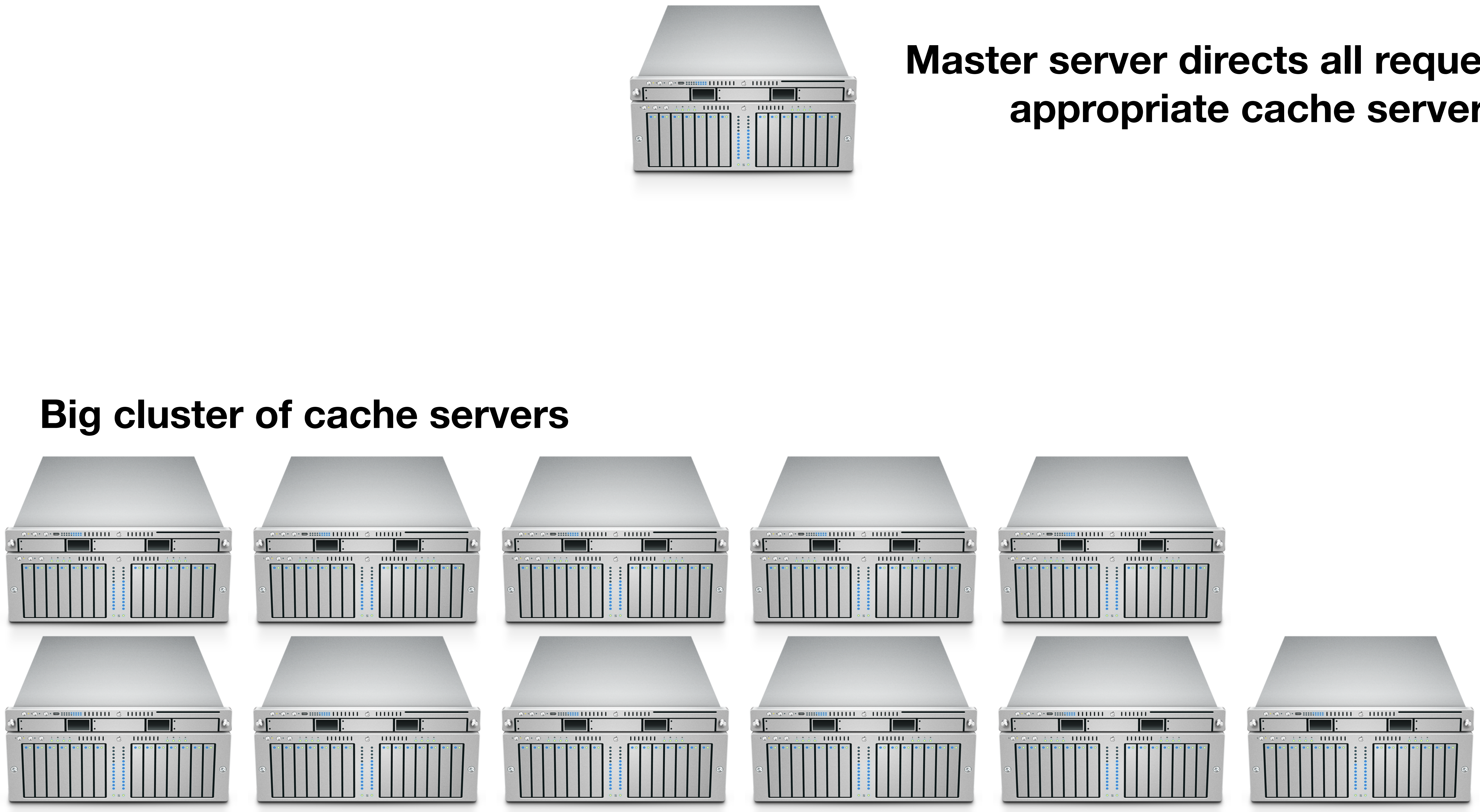
Each 🔵 is a value of hash % 16

Each 🔴 is a bucket

Example: bucket with key 9?

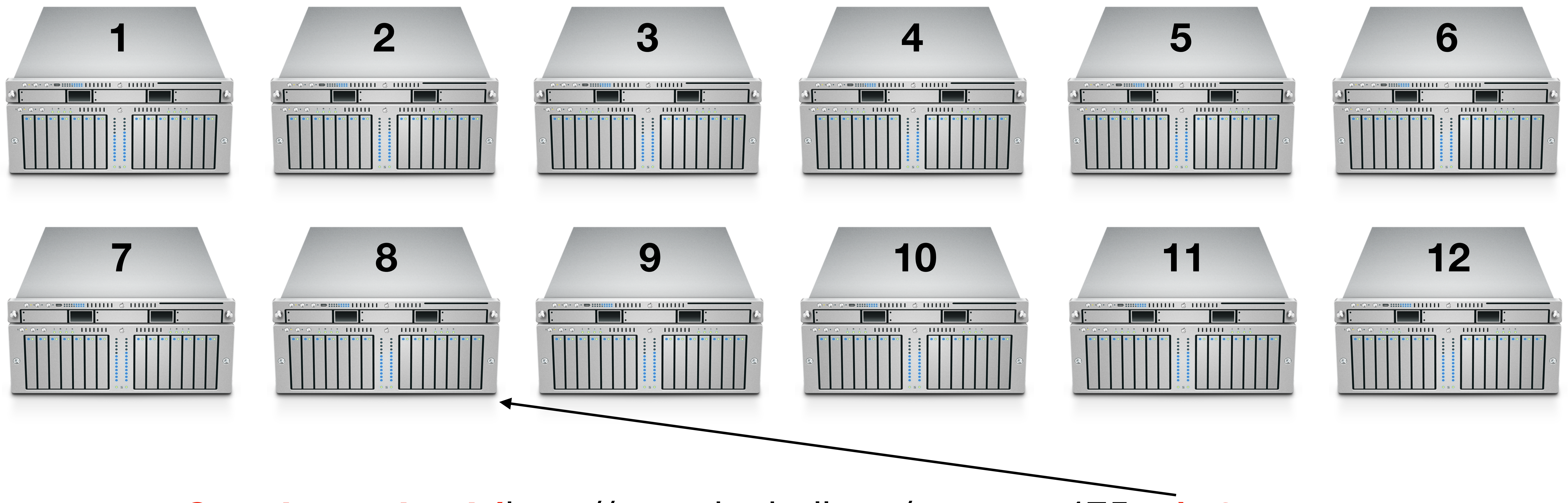


Review: CDN: How to find content?


Master server directs all requests to appropriate cache server
Big cluster of cache servers

Problem: Master becomes a huge bottleneck
Millions of requests, each request needs to be processed incredibly fast

Review: CDN: Finding Content



Consistent hash(http://www.jonbell.net/gmu-cs-475… **)=8**

Today

- Today:
  - How do we get rid of the “master” server that keeps track of metadata? - Pure peer-to-peer systems
  - Begin discussion of Byzantine failures (continue next class)
- Reminder - Project is out!
  - Fault-tolerant, sequentially consistent replicated key value store
  - Can do in a group (1 to 3 students per group)

Why P2P?

- Spreads network/cache costs across users instead of provider
- No server might mean:
  - Easier to deploy
  - Less chance of overload
  - Single failure won’t take down the system
  - Harder to attack

Why not P2P?

- Hard to find data items over millions of users
- Computers might not be as reliable as a managed server
- Less secure (?)

P2P

- Goal: IF there must be a master, all that it knows is the address of a few clients using the system
- Otherwise, everyone talks to each other, figures it out
- Replicate files, store them on clients, let clients find files from each other
- Challenges:
  - Where to find data?
  - What to do when clients come and go?

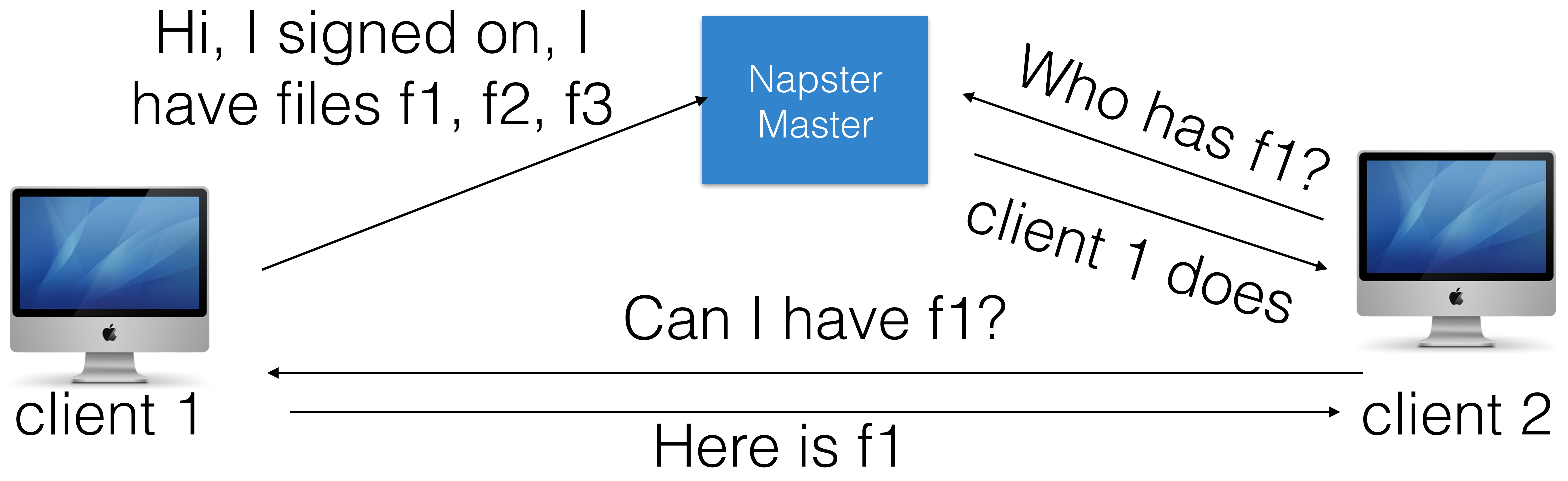
P2P

- Break it down into four operations:
  - **Join** the network and begin participating
  - **Publish** a file to the network, letting others know you have it
  - **Search** for a file that you want
  - **Fetch** a file once it is found

Napster

- Single master (centralized DB) stores metadata and client status
- **Join**: Client contacts master
- **Publish:** Client reports list of files to master
- **Search:** Query the server, find who has the file you want
- **Fetch:** Get directly from that peer client

Napster



Doesn't everything just look like GFS, even things that predated it? :)

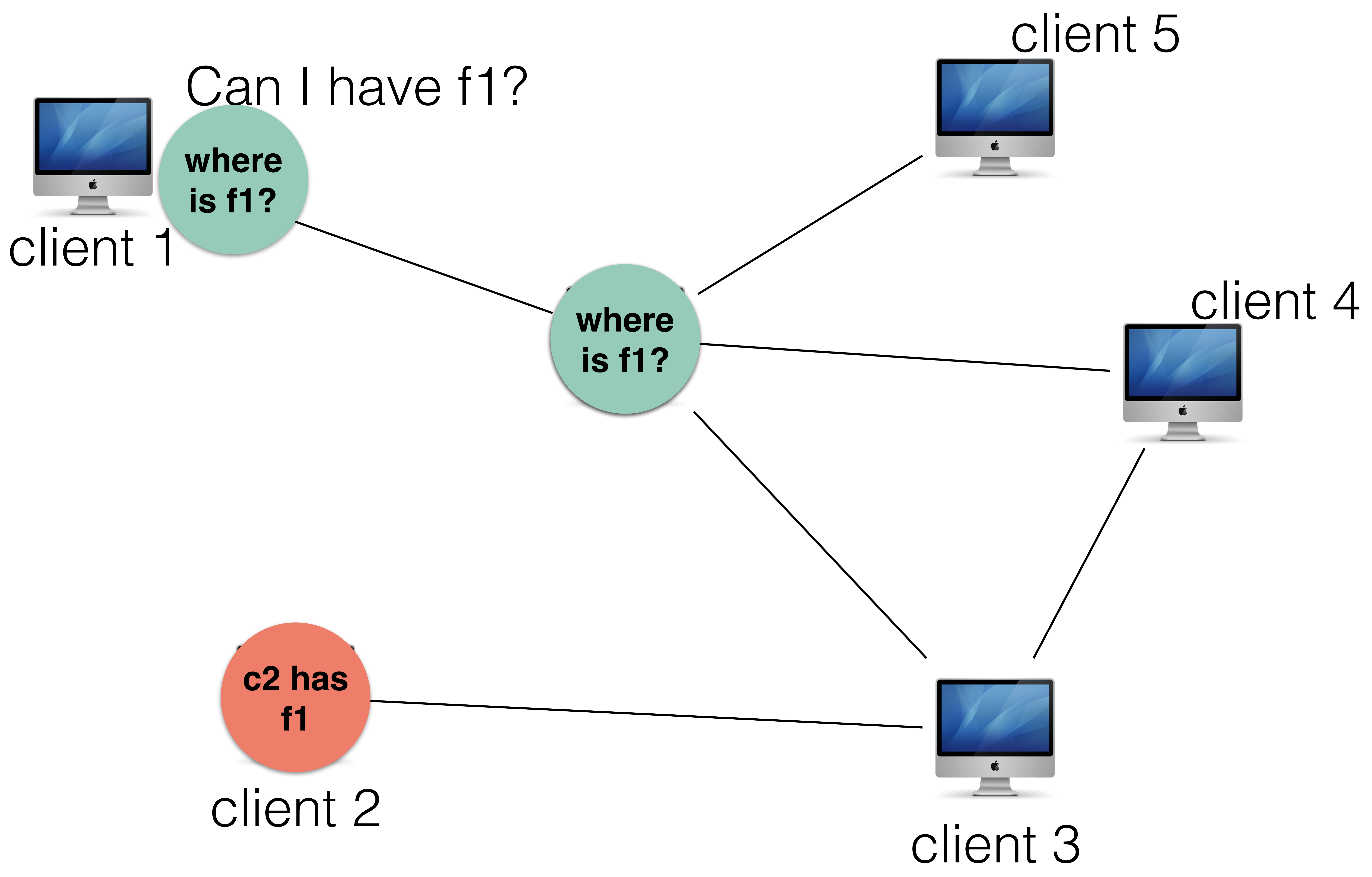
Napster

- The good:
  - Simple
  - Finding a file is really fast, regardless of how many clients there are - master has it all
- The bad:
  - Server becomes a single point of failure
  - Server does a lot of processing
  - Server having all of metadata implies significant legal liabilities

Gnutella 1.0

- **Join:** Client contacts a few other clients to find “neighbors”
  - Requires some initial mechanism to bootstrap
- **Publish:** N/A
- **Search:** Client asks neighbors for file, who ask their neighbors for file, who asks their neighbors out to some depth
- **Fetch:** Clients directly communicate with each other

Gnutella 1.0


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

Gnutella

- This is called "flooding"
- Cool:
  - Fully decentralized
  - Cost of search is distributed - no single node has to search through all of the data
- Bad:
  - Search requires contacting many nodes!
  - Who can know when your search is done?
  - What if nodes leave while you are searching?

BitTorrent

- Goal:
  - Get large files out to as many users as possible, quickly
- Usages:
  - Static bulk content (Big software updates, videos, etc)
- User model is *cooperative*
  - While downloading a large file, also sharing the parts that you have
  - After you get the file, keep sharing for a while too
- Approach relies on a “tracker” *per file*

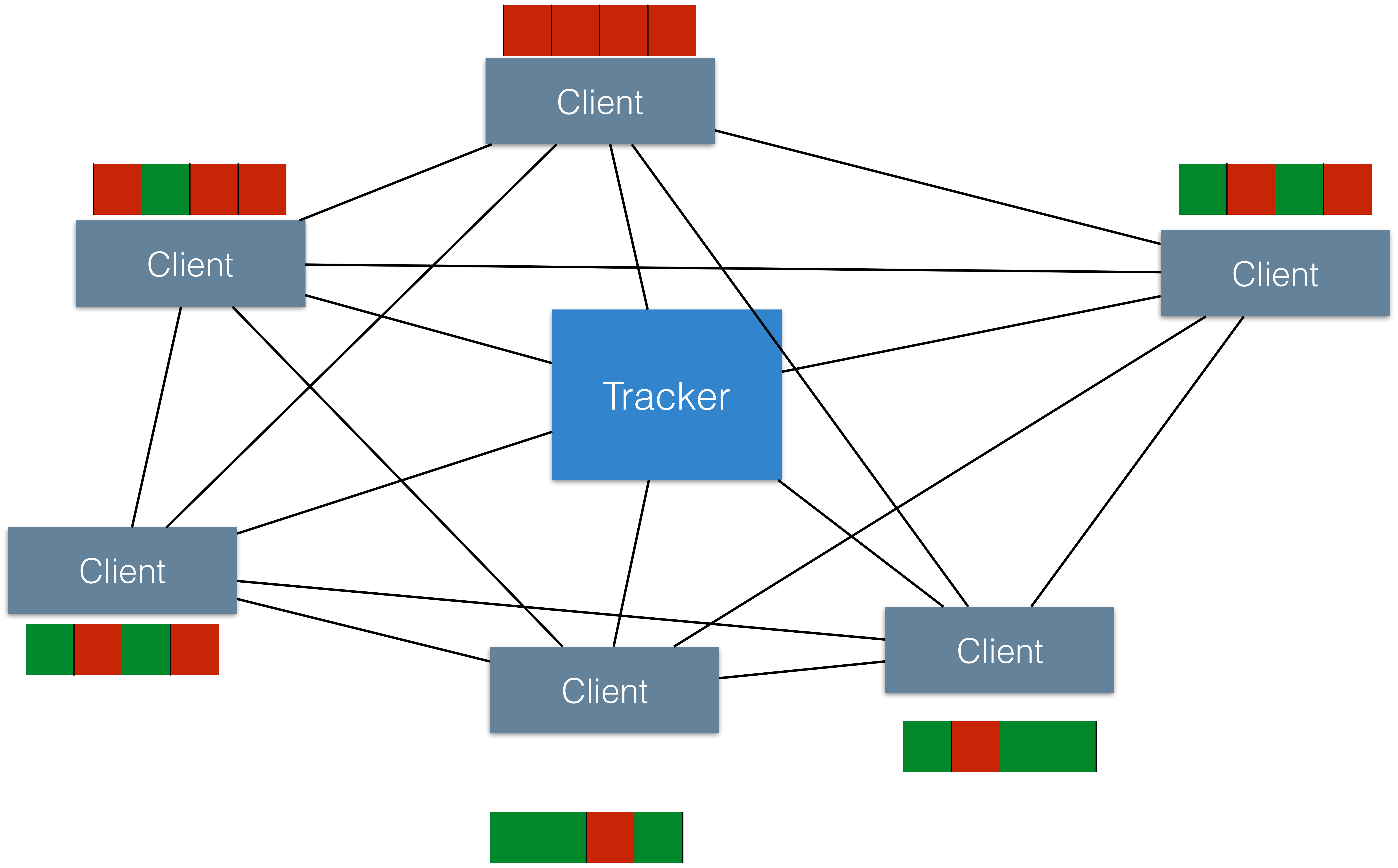
BitTorrent

- "Swarming"
- **Join**: Contact master "tracker," get list of peers
- **Publish**: Run a tracker server
- **Search**: Out-of-band (e.g. google)
- **Fetch**: Download chunks of files from peers

BitTorrent vs Napster

- Focus on **less** files, each of which is **larger**
- Files are broken into chunks -> can get different pieces of a file from different clients
- Anti-freeloading mechanisms - if you don't share, you don't get to play!
  - Since a big file is many chunks, once you get a chunk you can immediately share it with others
- Trackers are still single-points of failure, but assumption is 1 tracker per file

BitTorrent



BitTorrent

- "Tit-for-tat" sharing strategy
- A is getting data from B, C, D
    - A will let the fastest of those get data from A
    - A will be optimistic though, and let nodes who haven't shared anything yet have some data so that they can have a chance to share

DHT (Distributed Hash Table)

- Goal:
  - Guarantee that a file is always found within some bounded and reasonable number of steps
- Abstraction:
  - Create a lookup table, mapping from file to node that has that file (much like Napster)
  - BUT distribute this lookup table amongst the nodes participating (no single master)

DHT

- **Join:** Contact some other node to bootstrap: integrate yourself into the DHT, get a node ID and list of participating nodes
- **Publish:** Tell "mostly the correct" node that you have a file
- **Search:** Query for a file, asking first a "mostly correct" node
- **Fetch:** Contact node that has it directly
- How do we know where to route? Consistent hashing!

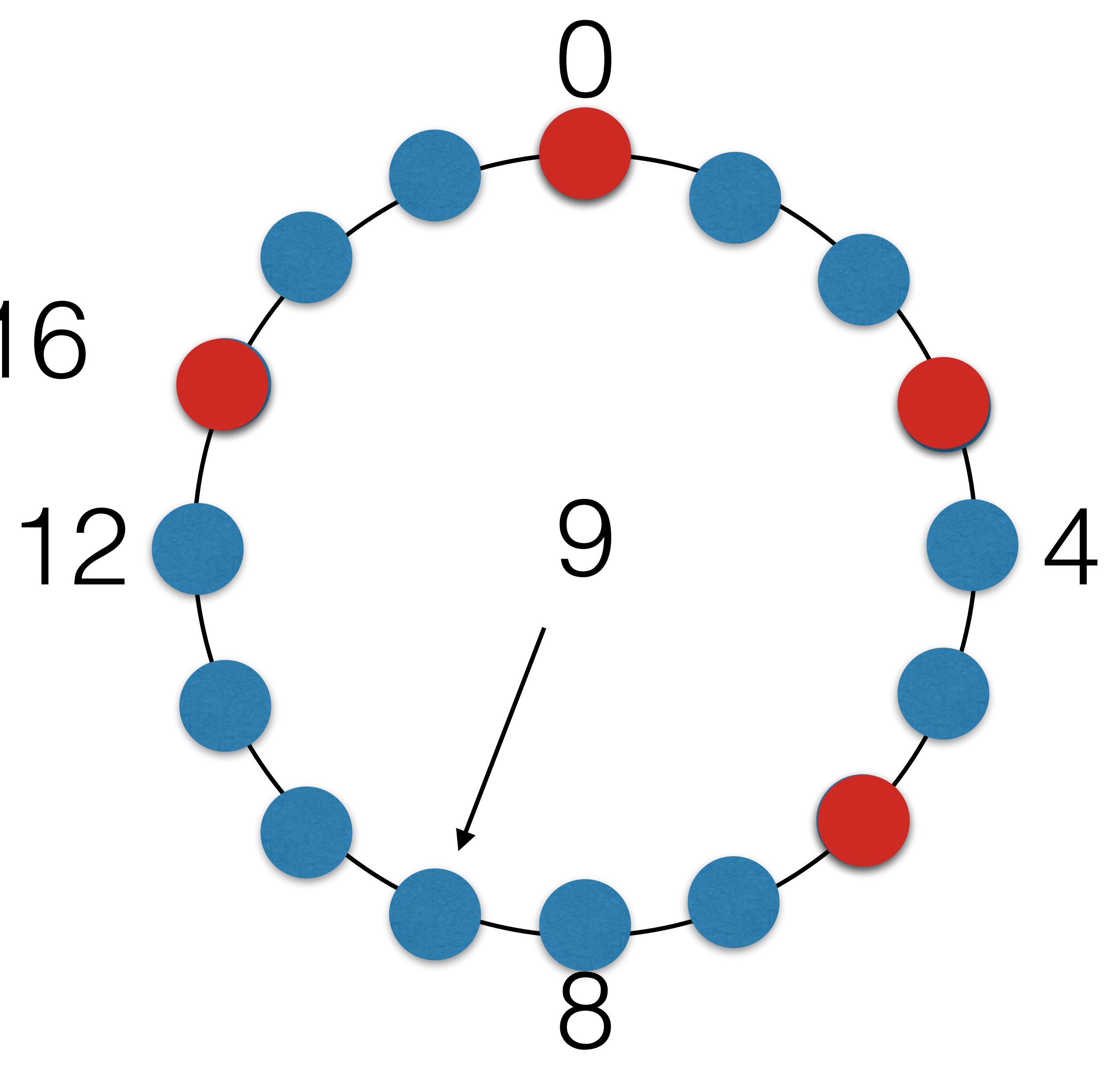
Reminder: Consistent Hashing

Example: hash key size is 16

Each ● is a value of hash % 16

Each ● is a bucket

Example: bucket with key 9?



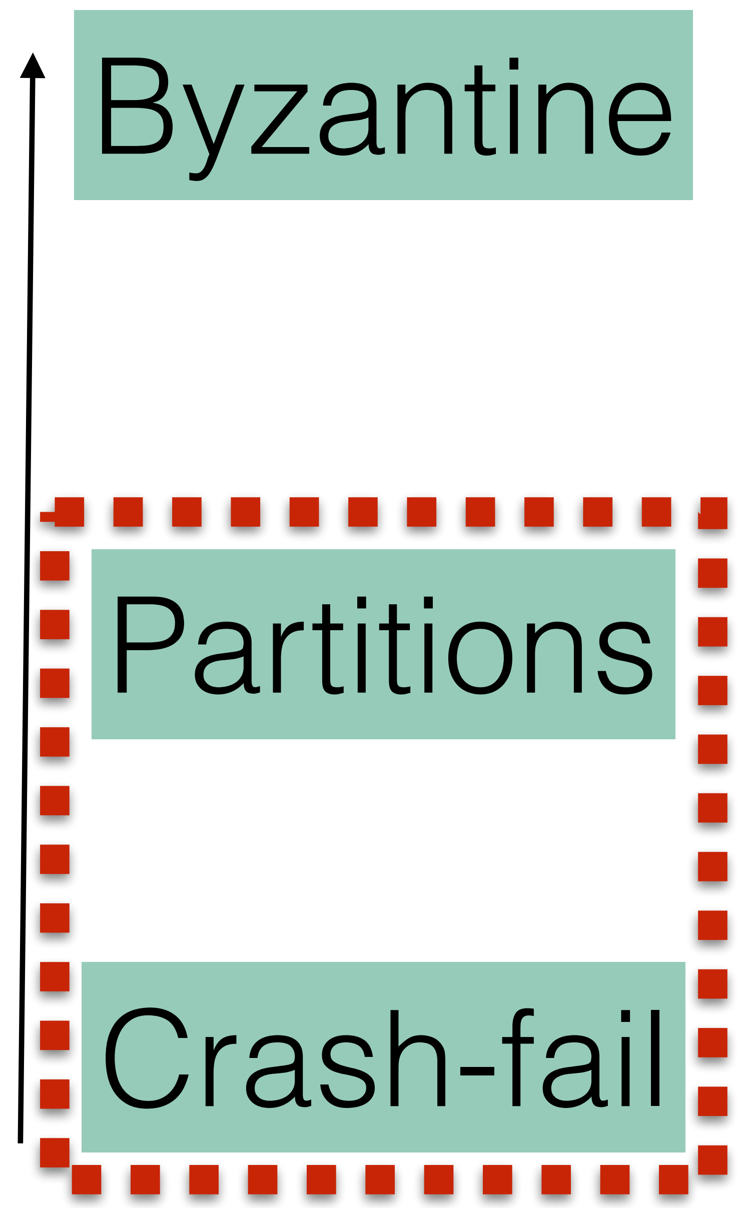
DHT

- Pros:
  - Guarantees that if the data is in the network, you'll find it in log(n) time (compare to Gnutella - pseudo-random search)
  - Good for caching, infrequently written data
- Cons:
  - Can really only match on exact keys
  - The node join/leave story is really bad - if we are distributed across the internet, a node leaving/joining might involve moving hundreds of GBs around

DHT Applications

- Use a DHT instead of a tracker for BitTorrent!
- Bootstrap: find a DHT peer
- Application: As you acquire files or look for files, add those facts into the DHT

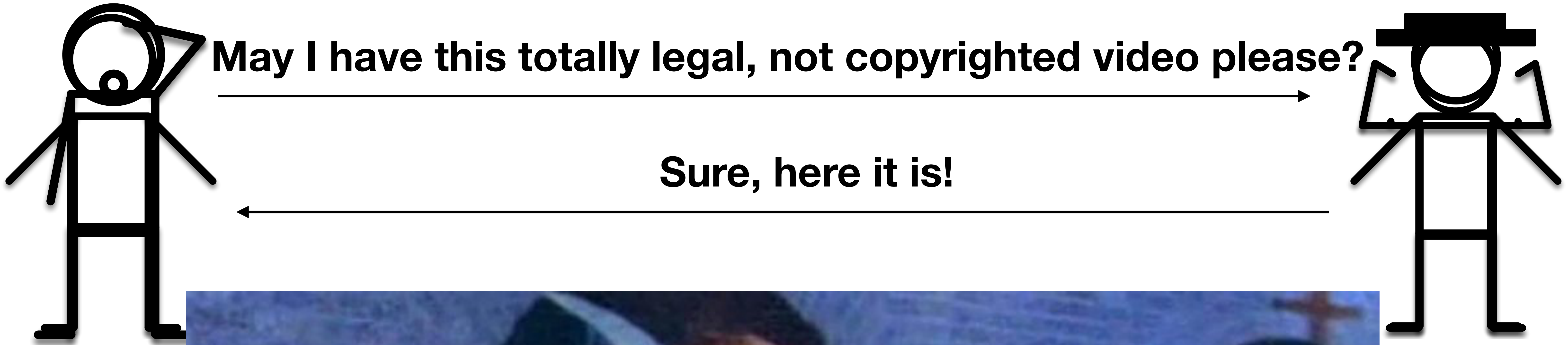
Is *our system* well behaved?



Start talking about for P2P systems

What we’ve done so far

Byzantine Failures in P2P


May I have this totally legal, not copyrighted video please?
Sure, here it is!

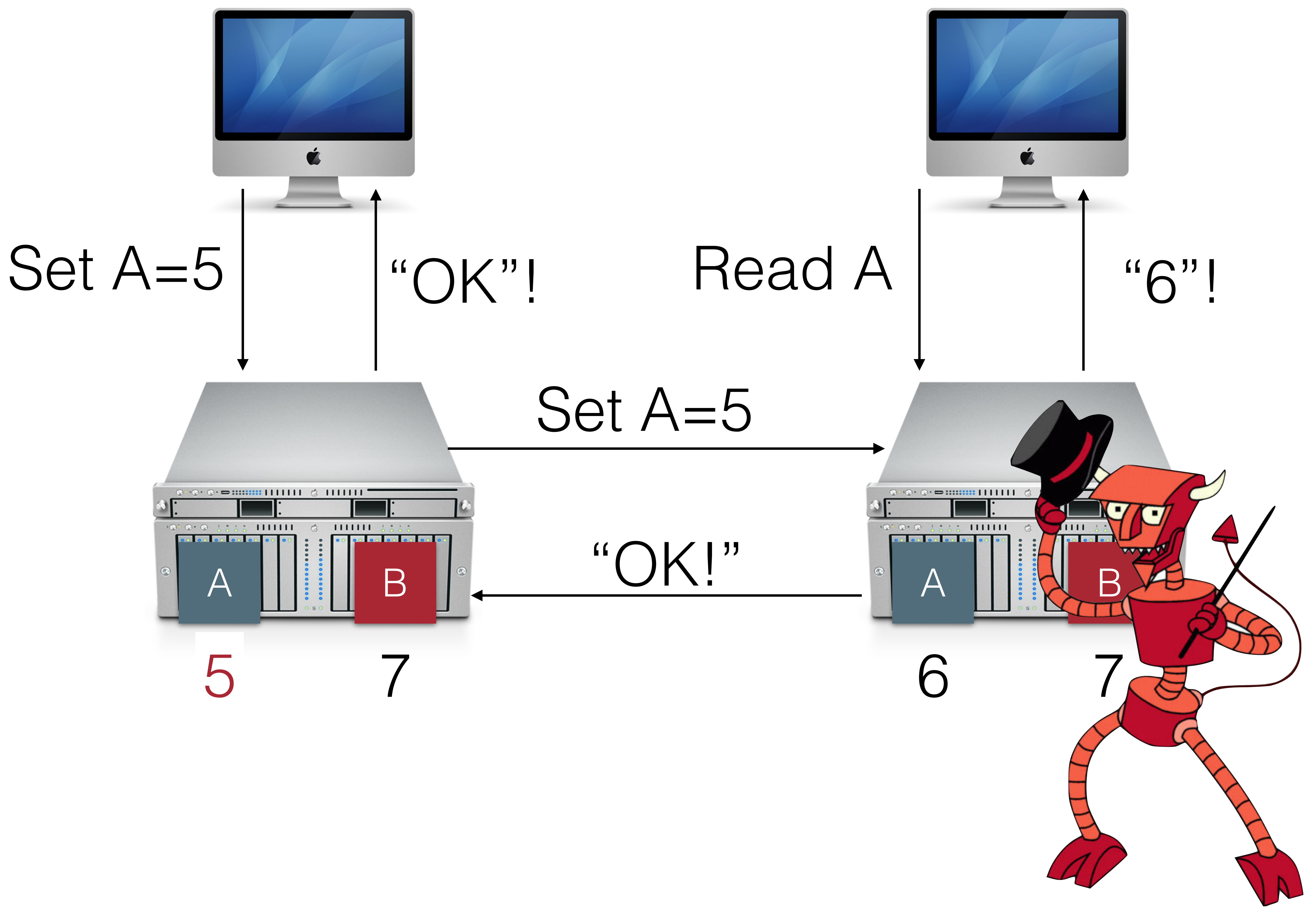
Detecting Failures

- Our expectation so far: Fail-stop
- If a system stops working, it's failed
  - Maybe was network
  - Maybe was computer
  - Hard enough already to tell the difference between temporary (partition) and persistent (node crash)
- What if a node fails but **does not stop responding?**

P2P and Byzantine Faults

- In systems with tracker (e.g. BitTorrent, Napster), **trust the tracker** to tell you the hash of the file (including at the chunk granularity for bit torrent)
- What do you do if you don’t trust the tracker (or there isn’t one)?
- This is the general problem of byzantine faults

Byzantine Faults



Byzantine Faults in Practice

- Many cases in aviation, e.g. 777 fly-by-wire control system
- Pilot gives input to flight computer
- THREE different flight computers
  - AMD, Motorola, Intel
- Each in a different physical location, connected to different electrical circuits, built by different manufacturers
  - Different components vote on the current state of the world and what to do next
  - Tolerates all kinds of failures

Byzantine Faults in Practice 737-MAX Edition

- Hardware designers implemented redundant flight controls to determine if plane was pointing its nose too far up
- Pilots cross-check instruments to double check that the failure of a single instrument doesn't crash the plane
- Because of hardware design, plane needs an always-on autopilot system ("MCAS"), specifically designed to keep the nose of the plane from pointing up too far

Byzantine Faults in Practice 737-MAX Edition

How the new Max flight-control system (MCAS) operates to prevent a stall

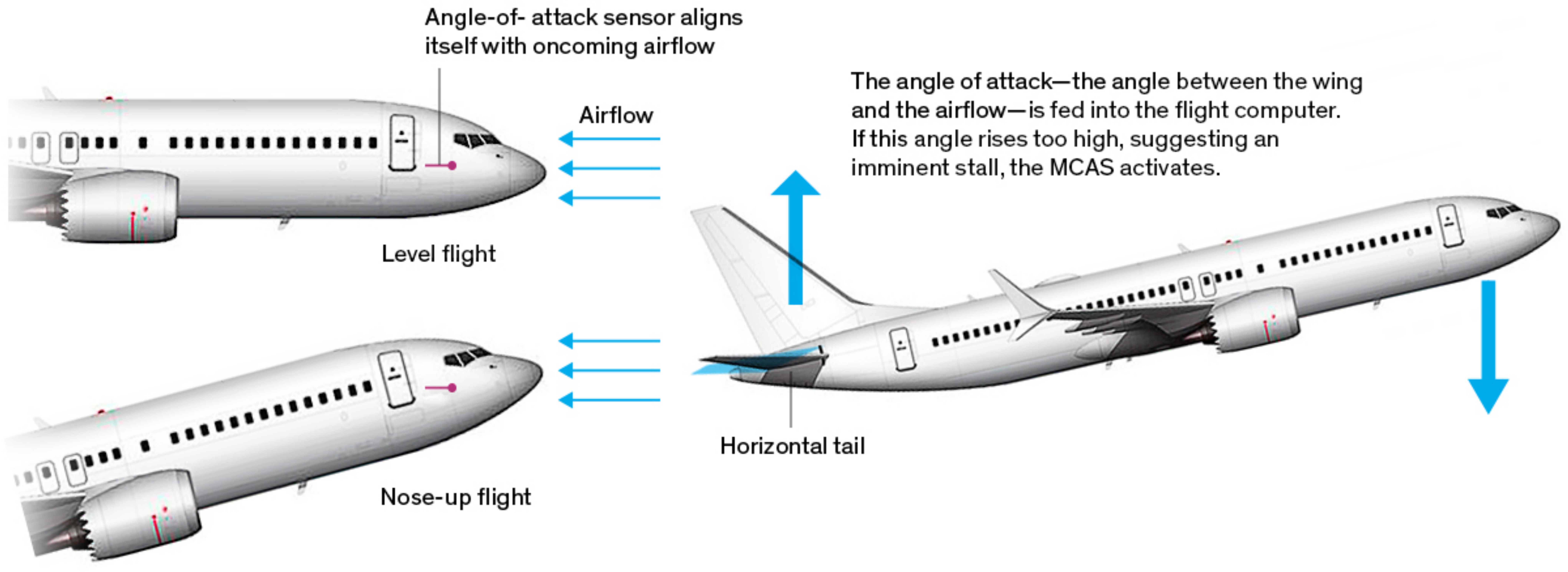

Angle-of- attack sensor aligns itself with oncoming airflow
Airflow
Level flight
Nose-up flight
The angle of attack—the angle between the wing and the airflow—is fed into the flight computer. If this angle rises too high, suggesting an imminent stall, the MCAS activates.
Horizontal tail

Illustration: Norebbo.com

Byzantine Faults in Practice 737-MAX Edition

- MCAS, the thing that can automatically point the plane down does *not* implement any redundancy
- Result: if the single probe used by the MCAS system gave an invalid result, the plane would point straight down to the ground and crash
- Irony: Boeing prided itself on not relying on software controls, and in having high degrees of mechanical redundancy (in contrast to Airbus)
- Nice article: https://spectrum.ieee.org/aerospace/aviation/how-the-boeing-737-max-disaster-looks-to-a-software-developer.amp.html

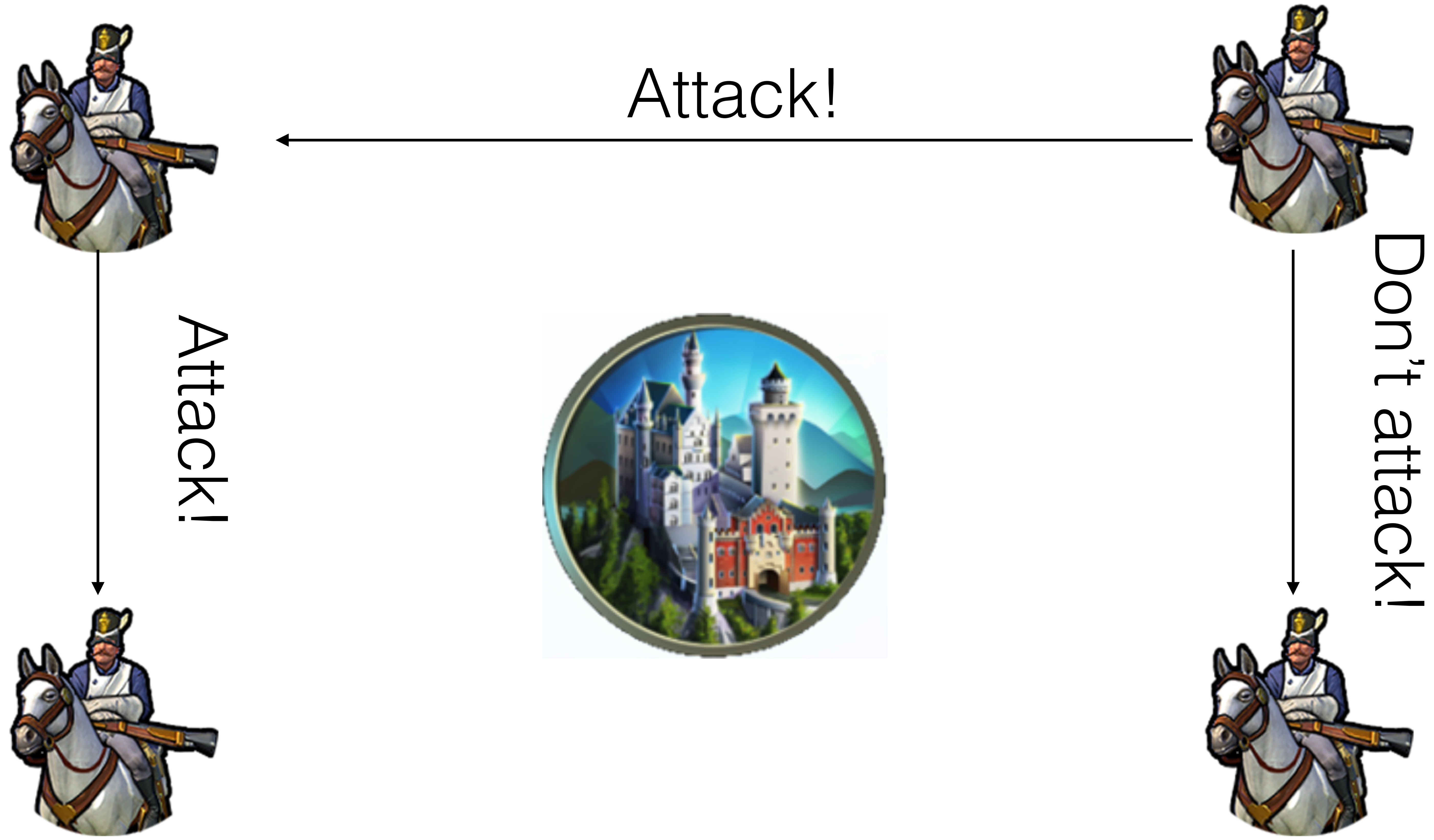
Byzantine Failures

- Very large set of ways in which a system might misbehave
- Bugs (perhaps on a single node)
- Intentional malice (perhaps a single node)
- Conspiracies (multiple bad nodes)

Byzantine General's Problem

- “We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement” - Lamport, Shostak, and Pease, 1980-2

Byzantine Generals Problem



Byzantine Fault Tolerance

- We tend to think of byzantine faults in an *adversarial* model
  - A node gets compromised, an attacker tries to break your protocol
- Adversary could:
  - Control all faulty nodes
  - Be aware of any cryptography keys
  - Read all network messages
  - Force messages to become delayed
- Also could handle bugs
  - Assuming uncorrelated (independent) failures
- How do we detect byzantine faults?

Byzantine Generals: Reduction

- Easier to reason about a single commander (general) sending his order to the others
- "Byzantine Commander Problem":
  - 1 commanding general must send his order to *n-1* lieutenants
  - All loyal lieutenants obey the same order
  - If the commanding general is loyal, every loyal lieutenant obeys the order he sends
- Consider metaphor:
  - General -> node proposing a new value
  - Lieutenants -> participants in agreement process

Byzantine Strawman 1

- *N* servers
- Client sends request to all
- Waits for all *n* to reply, only proceeds if all *n* agree

Byzantine Strawman 1

- Problem: a single evil node can halt the system

Byzantine Strawman 2

- $2f+1$ servers, assume no more than f are faulty
- If client gets $f+1$ matching replies, then OK

Byzantine Strawman 2

- Problem: can't wait for the last f replies (same as previous strawman)
- But what if the first *f* replies were from faulty replicas?

Byzantine Strawman 3

- $3f+1$ servers, of which at most f are faulty
- Clients wait for $2f+1$ replies
  - Take the majority vote from those 2f+1
  - If f are still faulty, then we still have f+1 not-faulty!

Byzantine Fault Tolerance ("Oral messages")

- Assumes conditions similar to if discussion were happening orally, by pairwise conversations between commanders and lieutenants
- Assumptions:
  - Every message is delivered exactly as it was sent
  - Receiver knows who the sender is for every message
  - Absence of a message can be detected (and there is some default assumed value)

Oral BFT Solution (No Traitors)

- Each commander sends the proposed value to every lieutenant
- Each lieutenant accepts that value
- (But that isn’t really fault tolerant…)

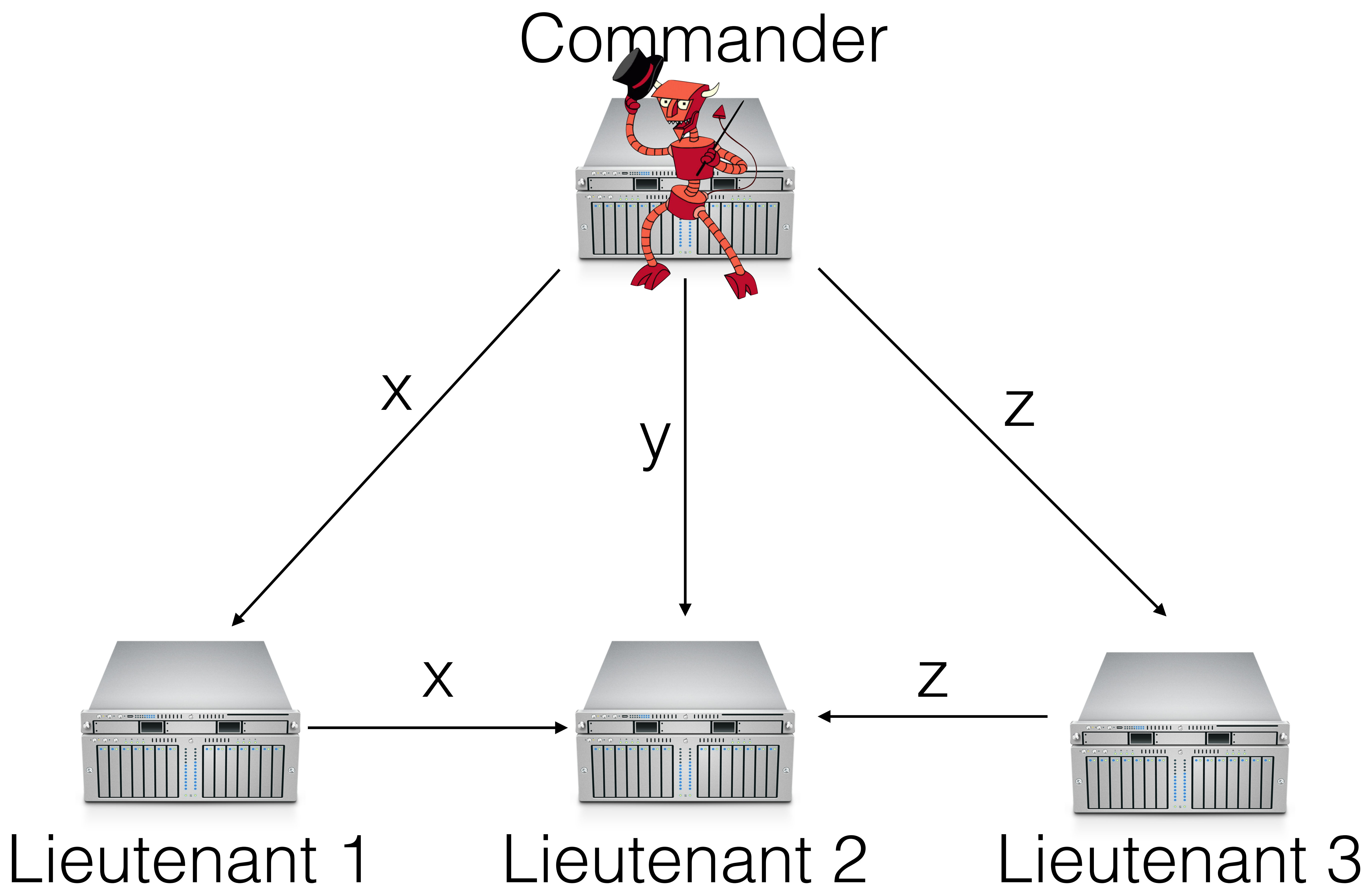
Oral BFT Solution (*m* traitors)

- Our solution: OM(m,S) tolerates *m* traitors in a set of *S* participants
- Commander *i* sends his proposed value v_i to every lieutenant *j*
- Each lieutenant *j* receives some value v_j from the commander (note they might receive different values if commander is traitor!)
- Each lieutenant has a conversation with each other lieutenant to confirm the commander's order, conducting OM(m-1,S-{i}), recursively

Oral BFT Solution (m traitors)

- Example: assume commander *i* is loyal
- Each lieutenant receives the same value from the commander
- Loyal ones could just accept that value, does not matter what traitors do (and hence, we are tolerant as long as a majority of commanders are loyal)
- BUT, maybe commander is not loyal
- Hence, assume commander is a traitor, and conduct a ballot to reach a consensus on what message the commander sent
- But how do you know that the other LIEUTENANTS are loyal? They might lie about what they heard from the commander
- Hence, recurse

Oral BFT Example (n=4, m=1)



Oral BFT

- At best, can tolerate m failures from 3m+1 participants
  - Ensures you always have a majority of valid participants
- If the loyal lieutenants decide the general is a traitor, they need to have some predefined behavior
- This is really expensive (communication)
  - To tolerate *m* traitors among *n* participants, or OM(m), each of n-1 participants will invoke this OM(m-1) times
  - OM(m-1) will cause n-2 participants to call OM(m-2)
  - Overall number of messages: $O(n^m)$
  - Example: tolerate 3 failures from 10 participants: 1,000 messages

This work is licensed under a Creative Commons Attribution-ShareAlike license