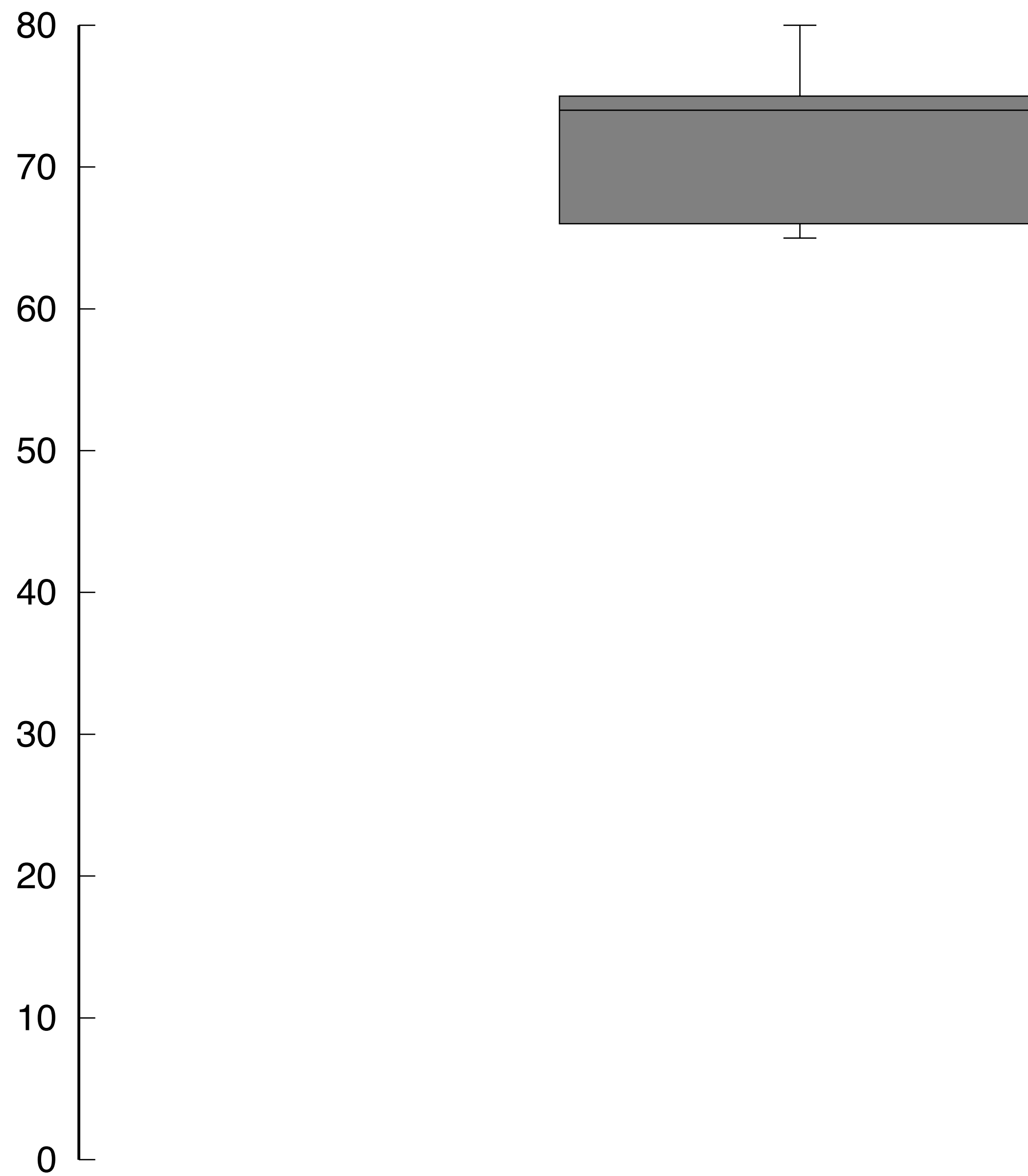
Security

SWE 622, Spring 2017

Distributed Software Engineering

GEORGE MASON UNIVERSITY

HW4

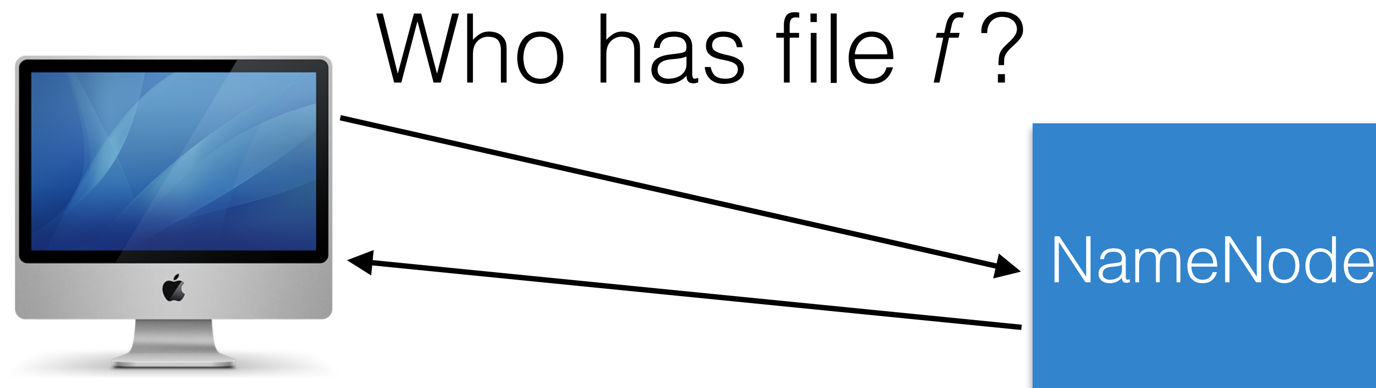


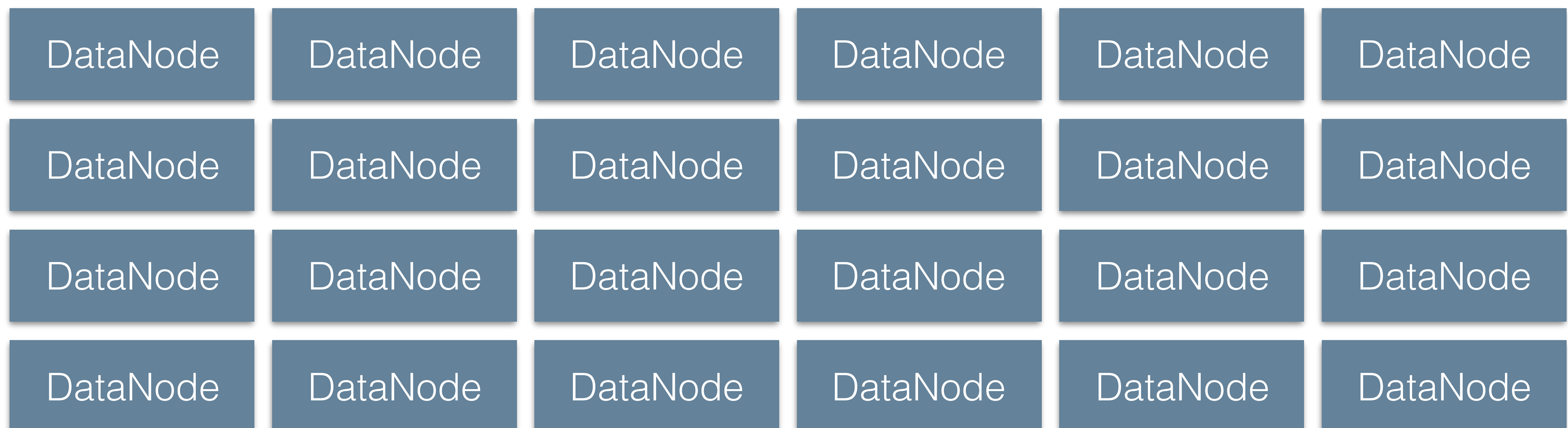
Review: HW 5: Fault Tolerance

- We're going all-in on ZooKeeper here
- Use ZK similar to how HDFS does: each Redis slave will have a dedicated ZK client to determine who the master Redis server is
- Redis master holds a lease that can be renewed perpetually
- When client notices a problem (e.g. WAIT doesn't work right or can't talk to master) it proposes becoming the master
- As long as a client can talk to a quorum of ZK nodes, then they can decide who the leader is
- Clients don't need to vote - just matters that there is exactly one of them

Where do we find data?

The easy answer: master metadata server (GFS, HDFS, etc)

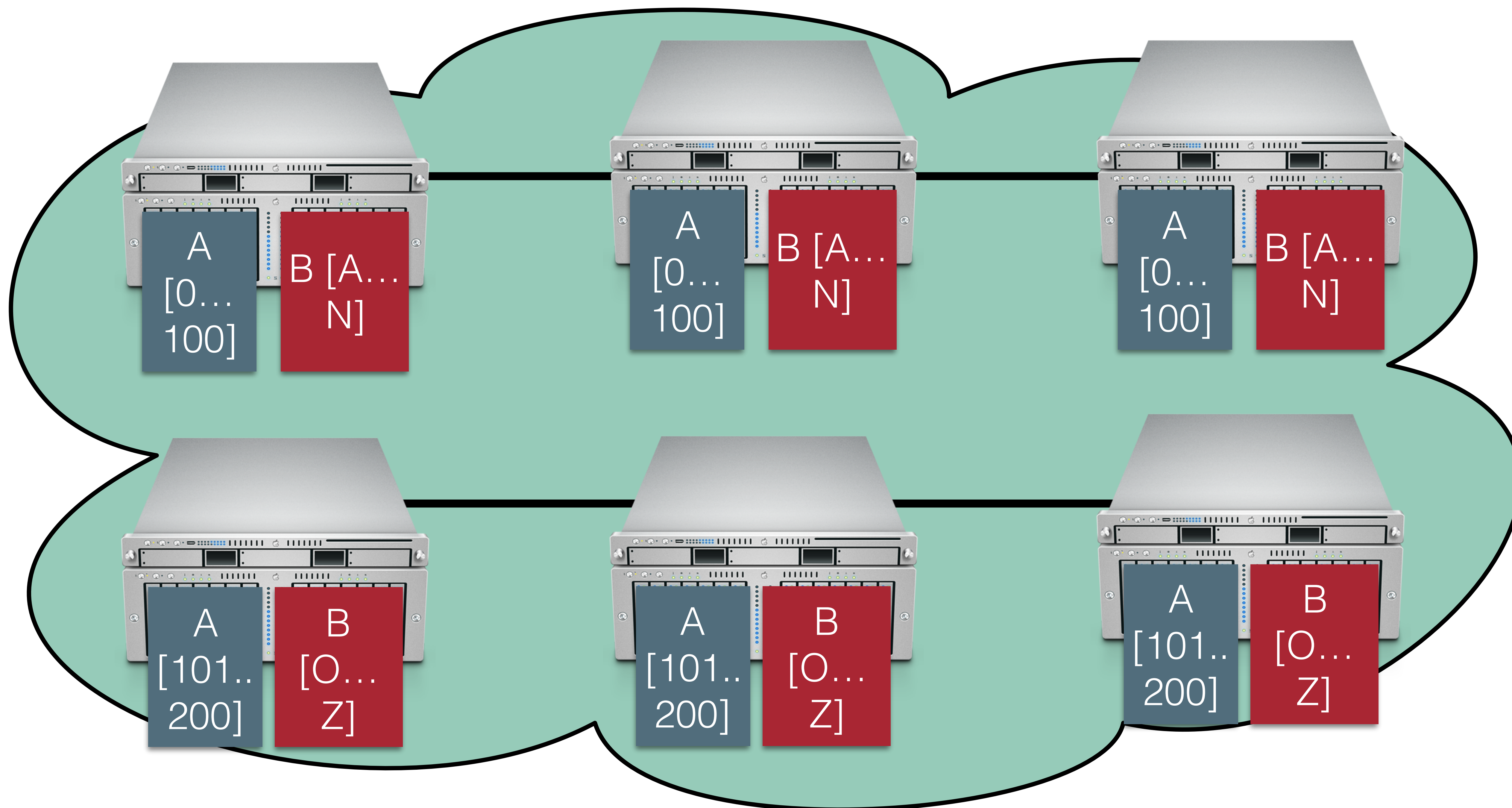




Where do we find data?

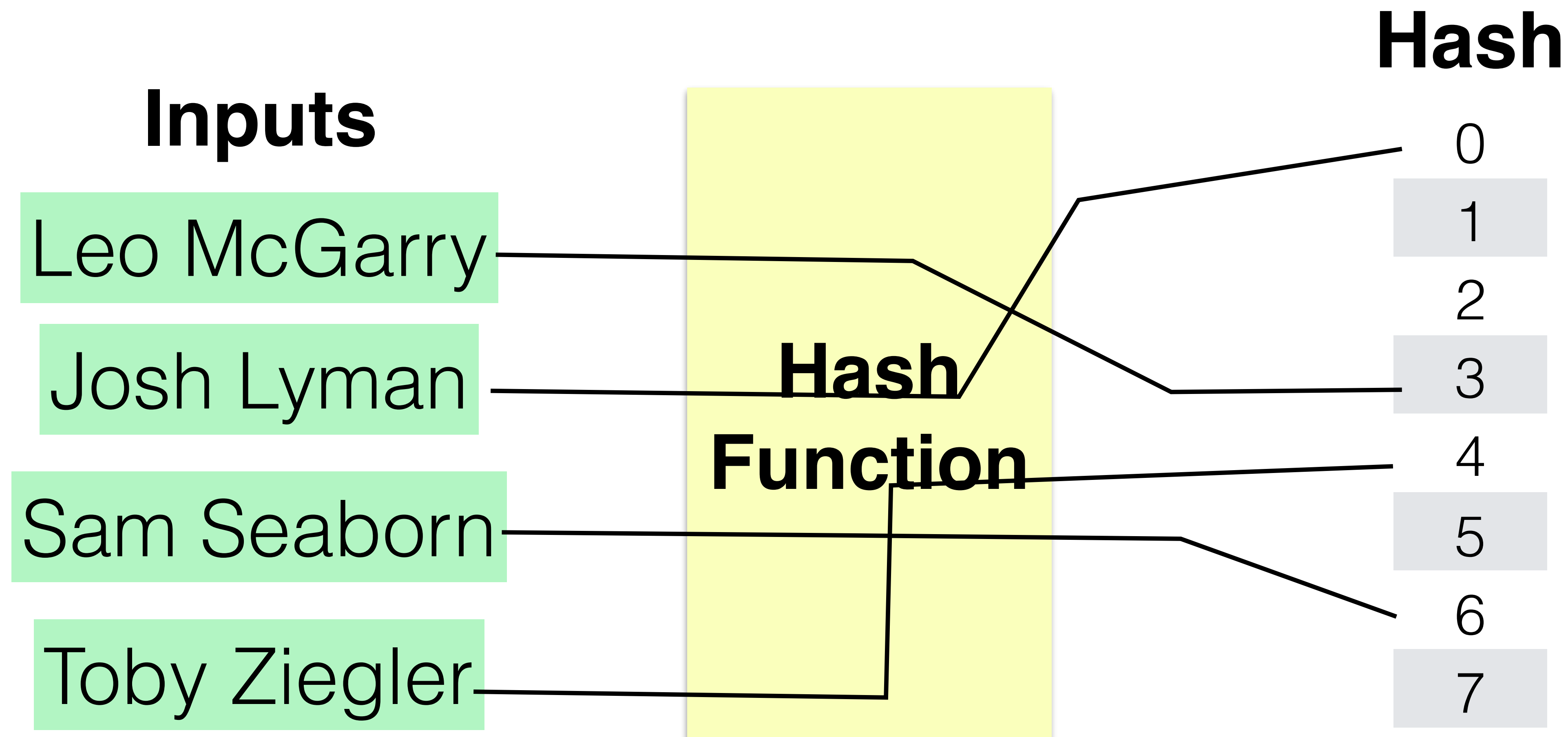
- What's bad with the single master picture?
- HDFS/GFS leverage the fact that there is relatively **little** metadata, **lots** of data (e.g. few files, each file is large)
- What if there is really only metadata?
- How can we build a system with high performance **without** having this single server that knows where data is stored?

Partitioning + Replication



Hashing

- Compresses data: maps a variable-length input to a fixed-length output
- Relatively easy to compute
- Example:

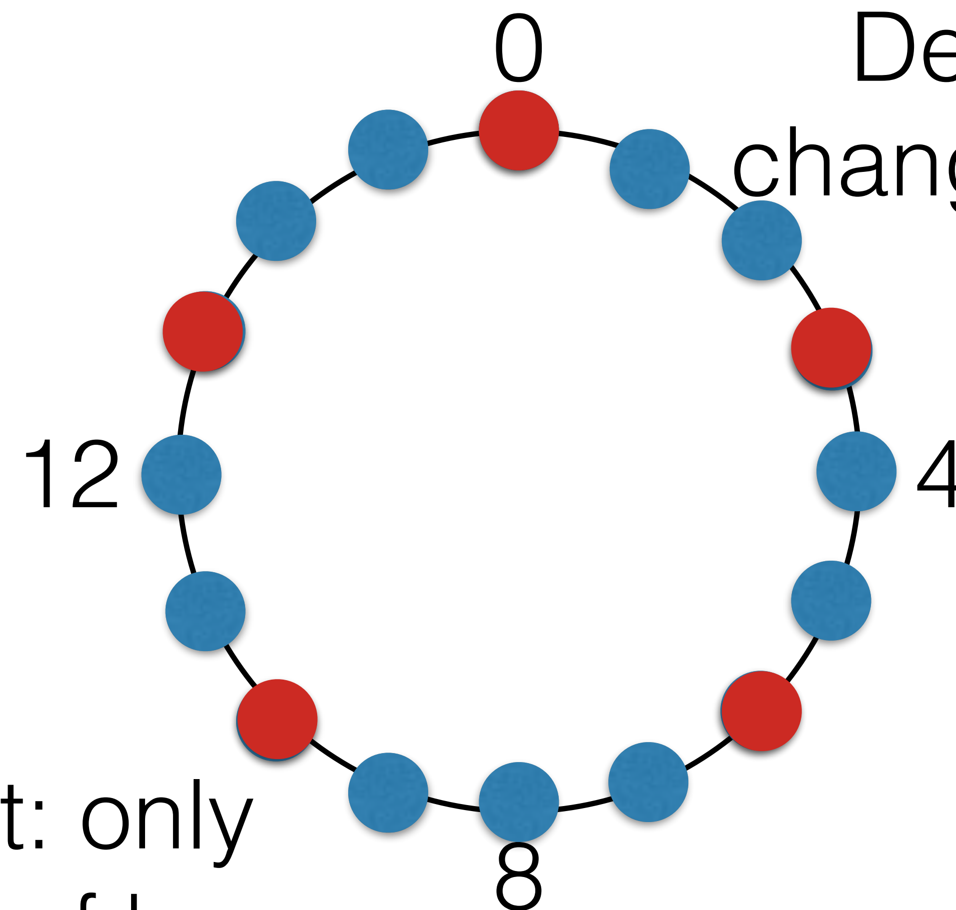


Hashing for Partitioning

Input		Hash Result		Server ID
Some big long piece of text or database key	*hash()=*	900405	*% 20 =*	5

Consistent Hashing

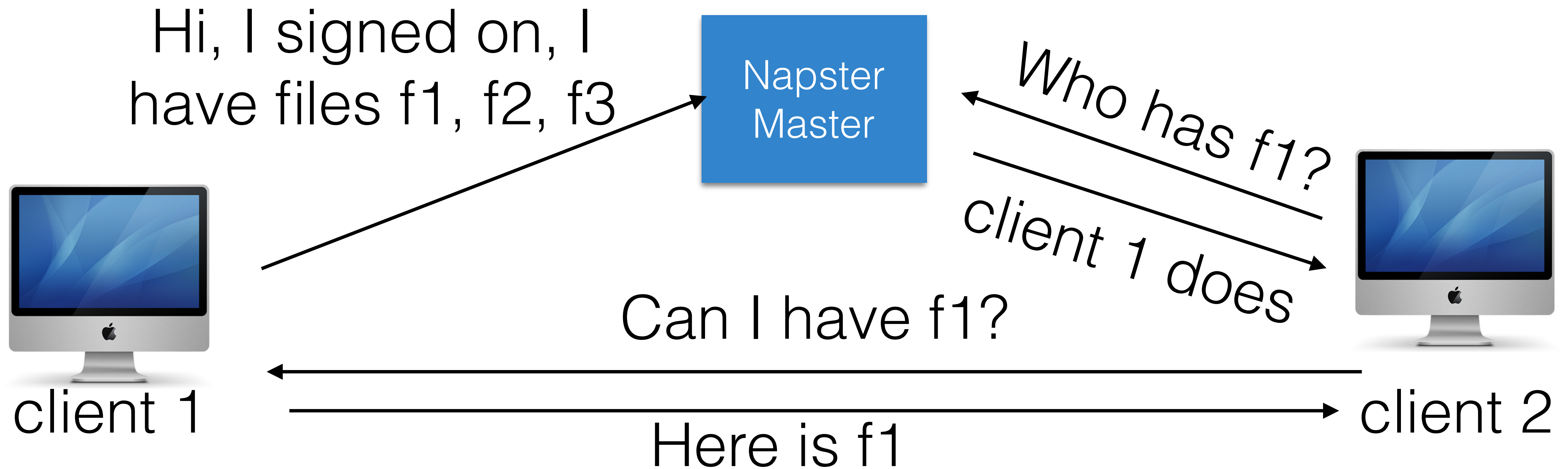
It is relatively smooth: adding a new bucket doesn't change that much



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

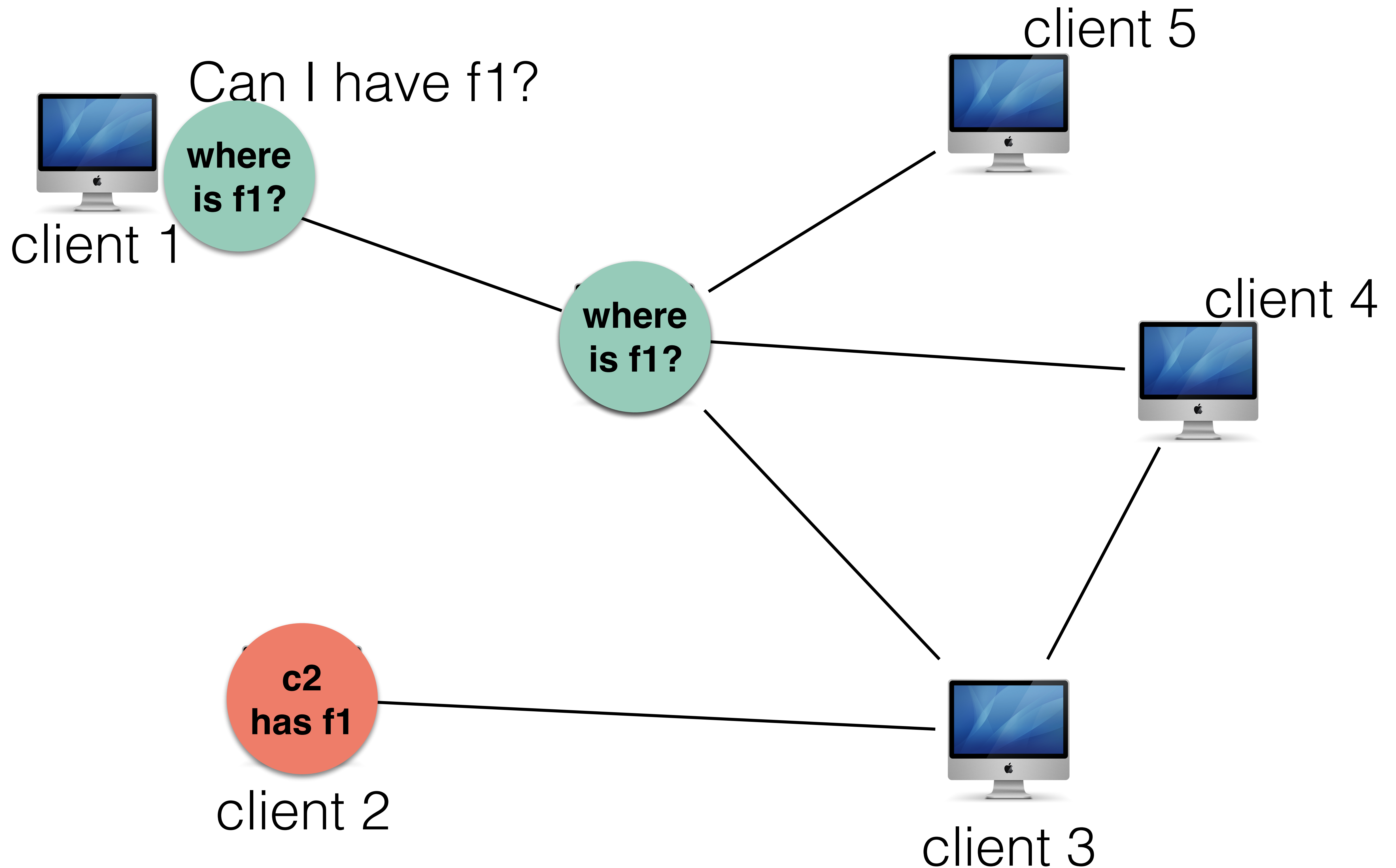
Add new bucket: only changes location of keys 7,8,9,10

Napster

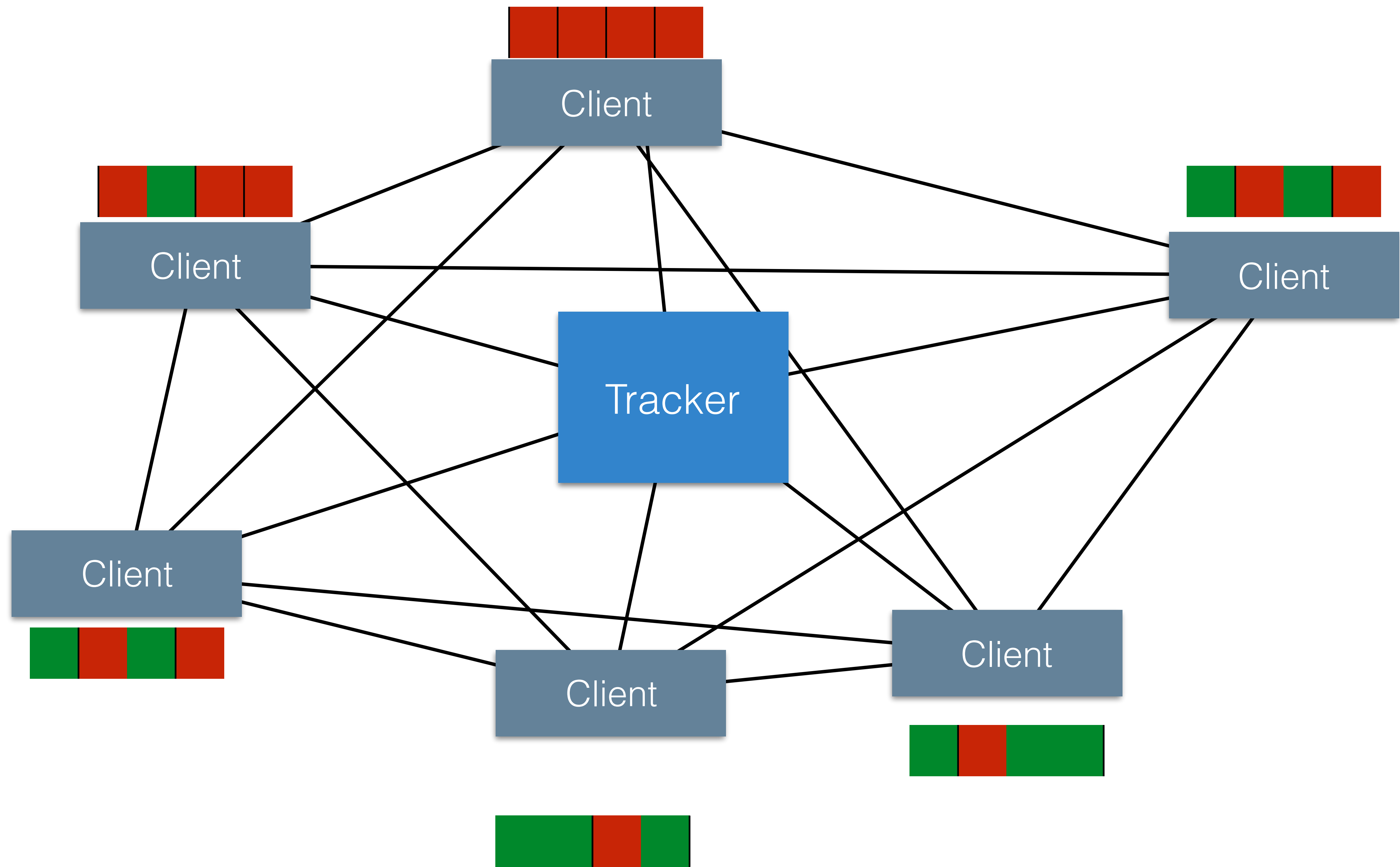


Doesn't everything just look like GFS, even things that predated it? :)

Gnutella 1.0



BitTorrent



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)

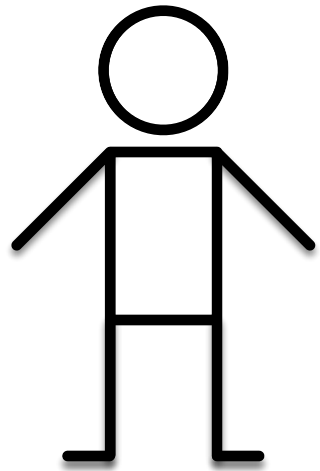
Today

- Security in Distributed Systems
- Failure Models
- Attack Models
- Byzantine Fault Tolerance
- Blockchains and Bitcoin

Threat Models

- What is being defended?
  - What resources are important to defend?
  - What malicious actors exist and what attacks might they employ?
- Who do we trust?
  - What entities or parts of system can be considered secure and trusted
  - Have to trust **something**!

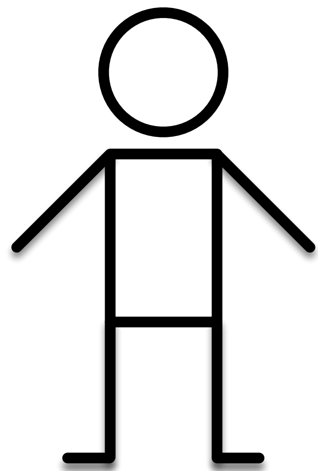
Web Threat Models: Big Picture



client page
(the "user")

server

Web Threat Models: Big Picture

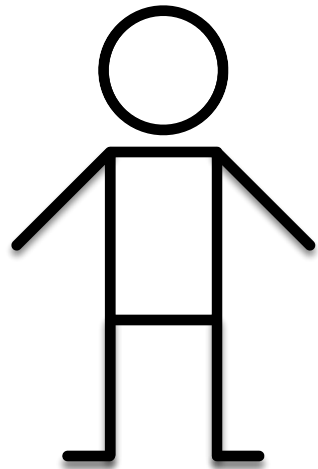


client page
(the "user")

server

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

Web Threat Models: Big Picture



HTTP Request

HTTP Response

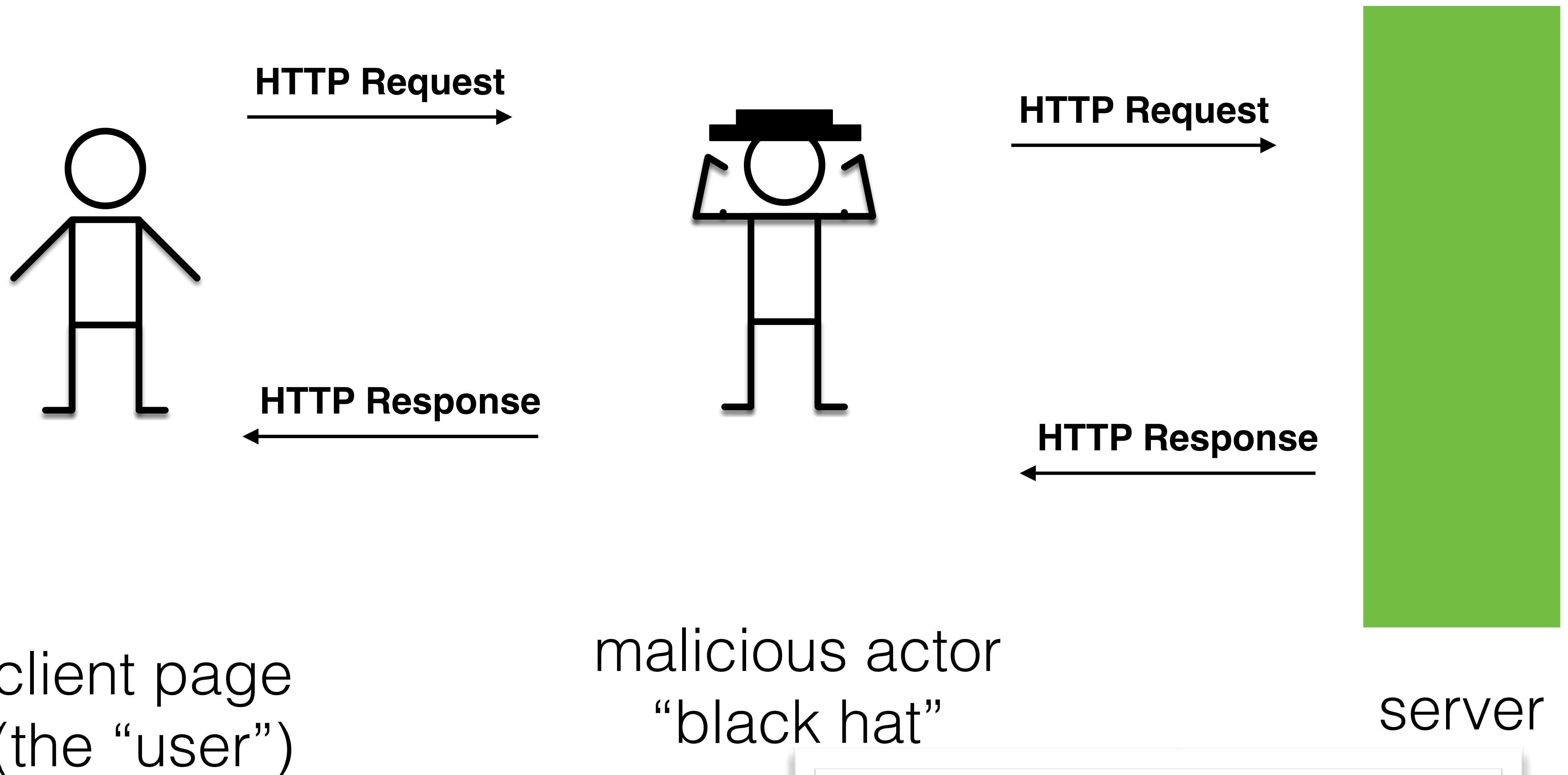
client page
(the "user")

server

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

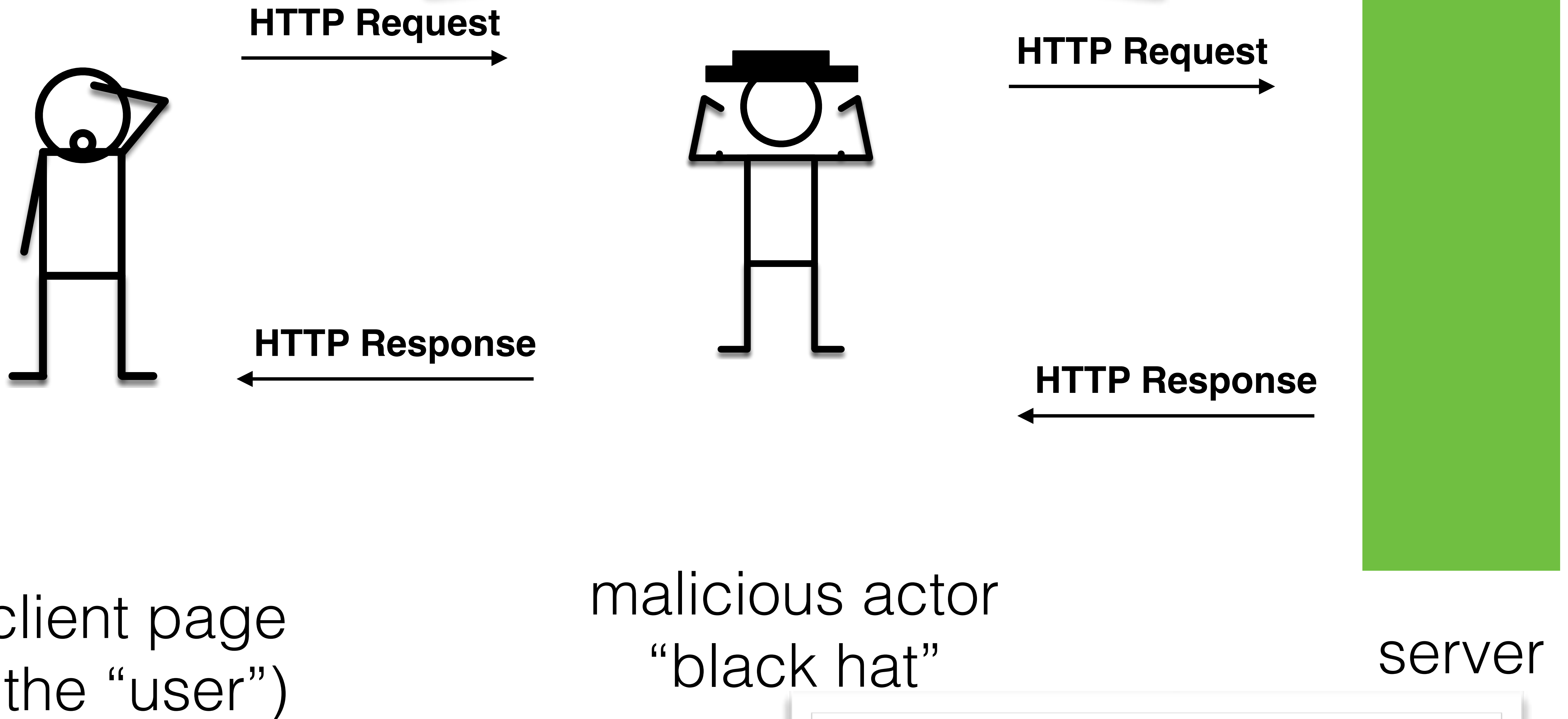
Do I trust that this request *really* came from the user?

Web Threat Models: Big Picture



Web Threat Models: Big Picture

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?

Other Threat Models

- Is our network well behaved?
- Is our network malicious?
- Who can access our system?
- Are our users well-behaved?
- Are our users malicious?
- Is our system well behaved?

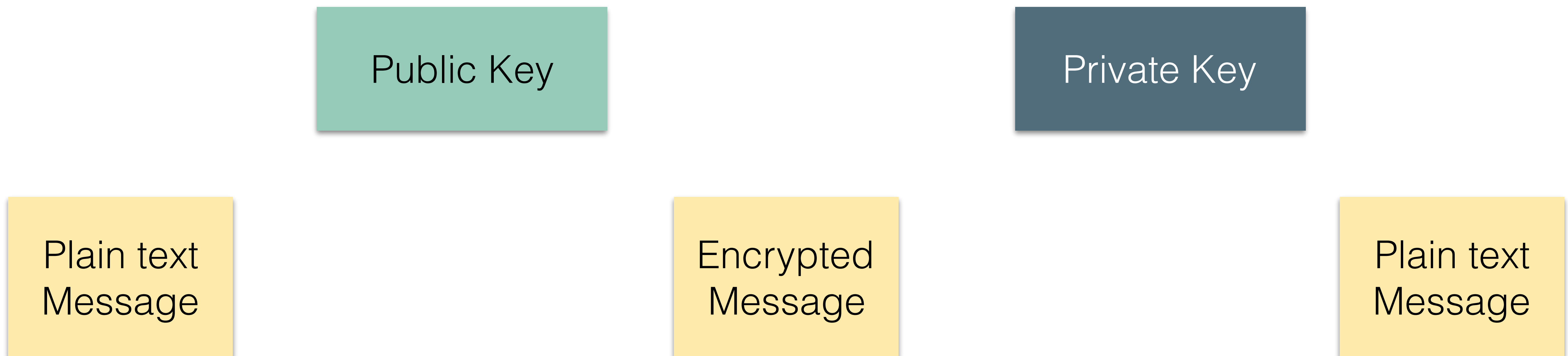
What does it mean for a distributed system to be secure?

- Maintain a secure channel between nodes:
  - Authenticity (Who am I talking to?)
  - Confidentiality (Is my data hidden?)
  - Integrity (Has my data been modified?)
  - Availability (Can I reach the destination?)
- Maintain some security about who participates in the system?

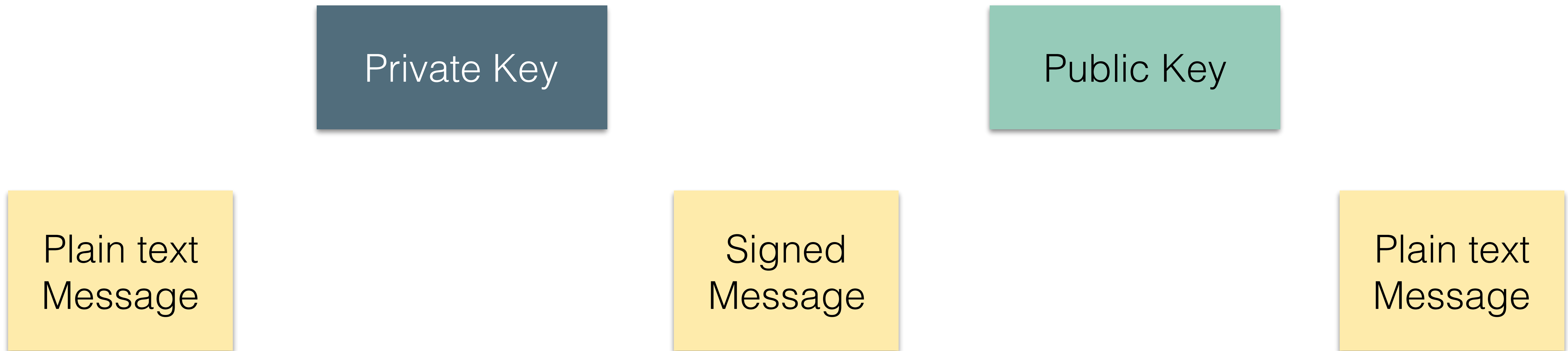
Cryptography Tools

- Symmetric encryption, aka shared secret key
  - $M = D_K (E_K (M))$
  - M is the data, D is decrypt, E is encrypt, and k is the key
  - computationally efficient
- Asymmetric encryption, aka public key/private key
  - $M=D_{K-} (E_{K+} (M))=D_{K+} (E_{K-} (M))$
  - K+ is public key, and k- is private key
  - computationally expensive
- Hashing & MACs
  - $S = H(M)$, or $S = MAC_K (M)$
  - S, aka digest, is a unique representation of data such that an accidental or intentional change to the data will change the representation
  - fixed size and independent of size of M
  - computationally efficient

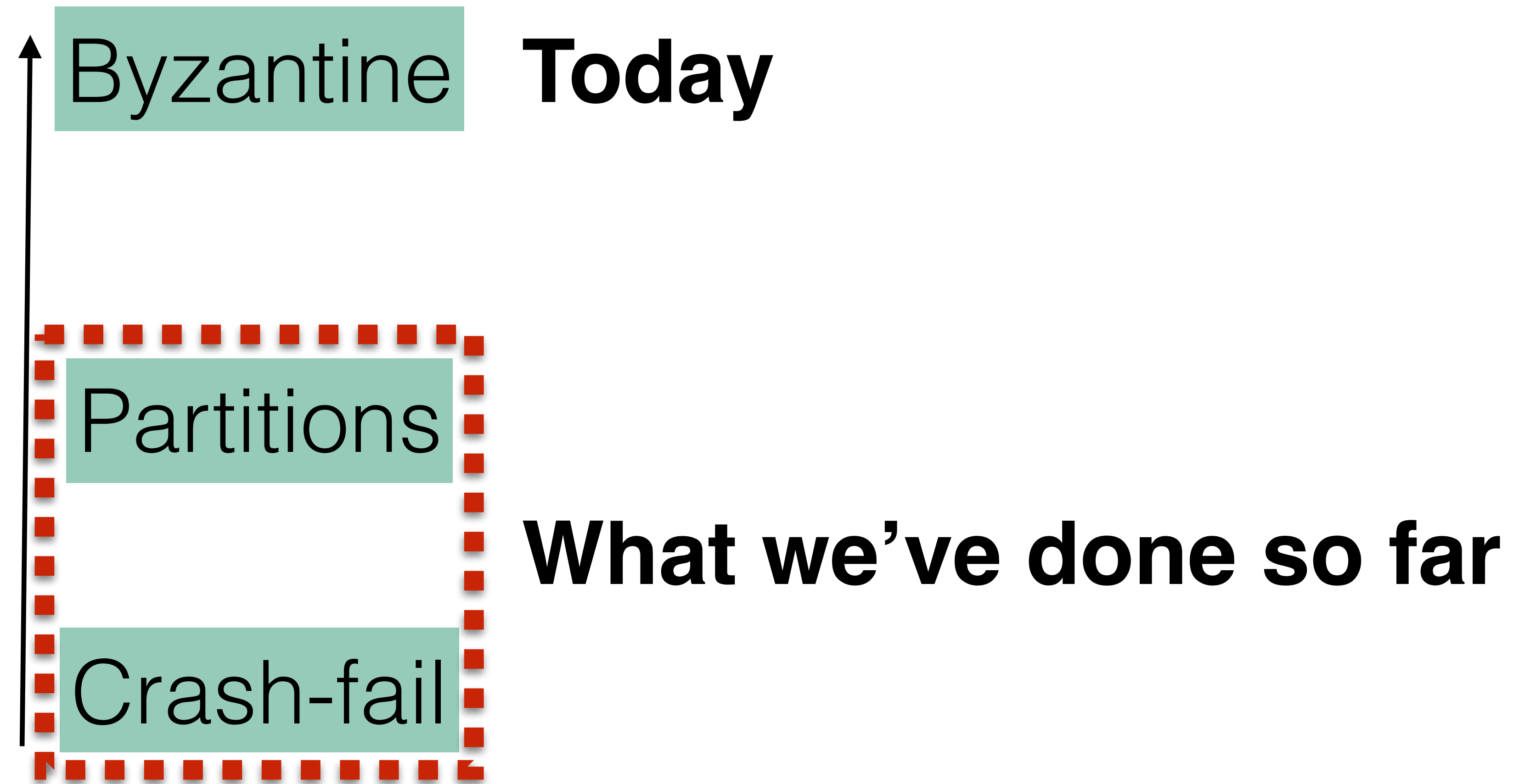
Public Key Cryptography



Public Key Cryptography



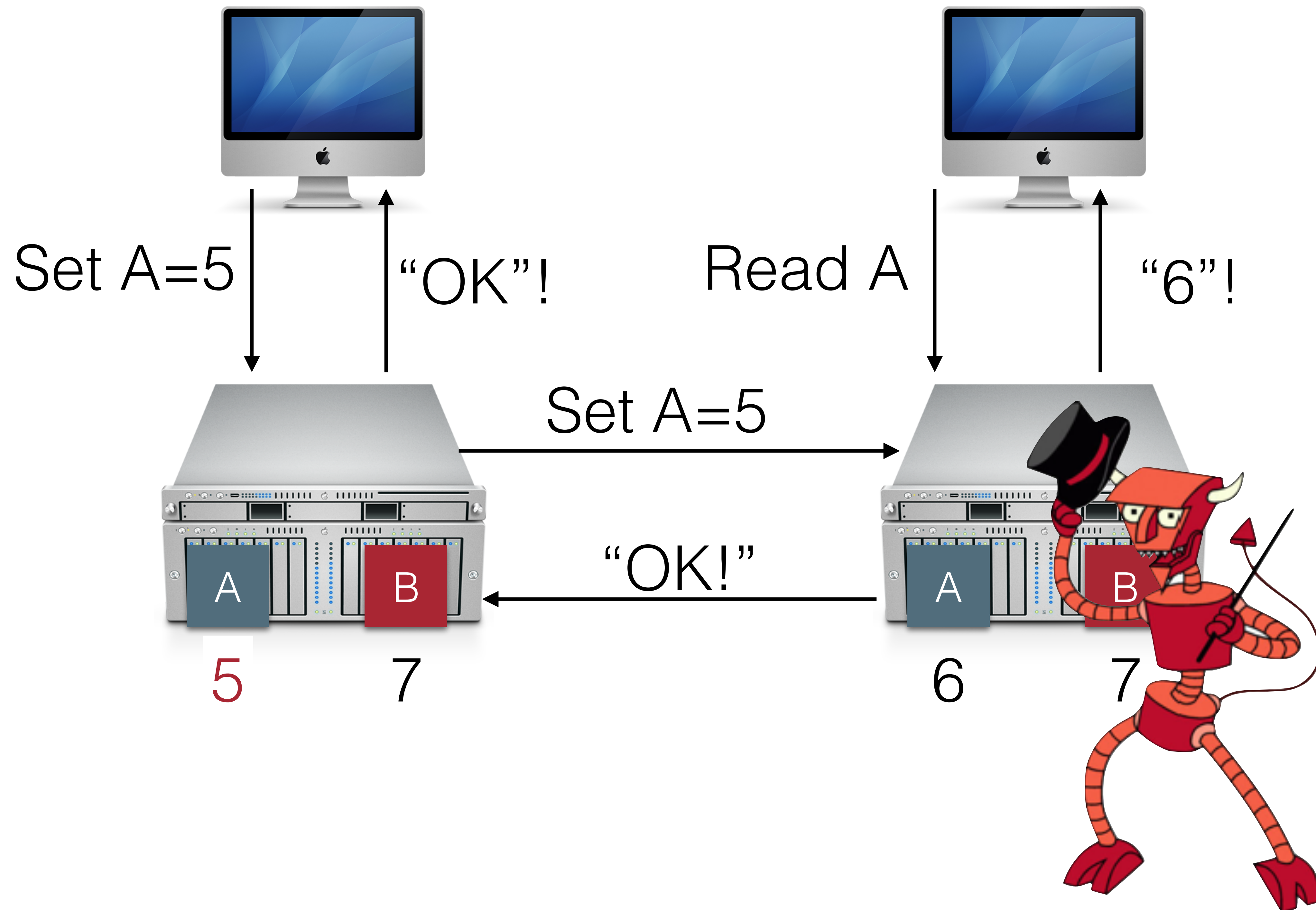
Is *our system* well behaved?



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?**
- Can we tell that it has failed?
  - Probably, using voting? But - expensive?

Byzantine Faults



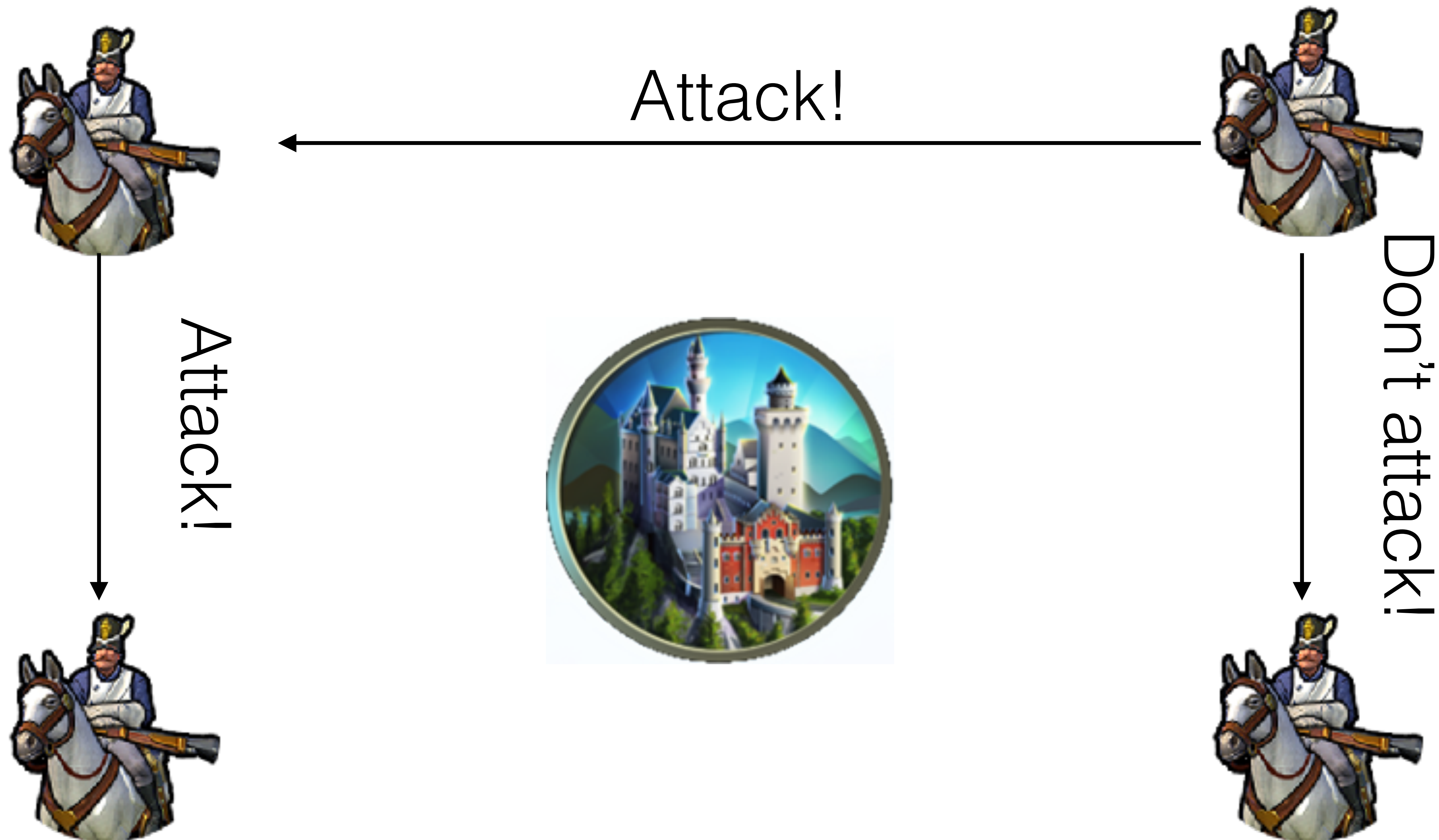
Byzantine Faults in Practice

- 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 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 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…)

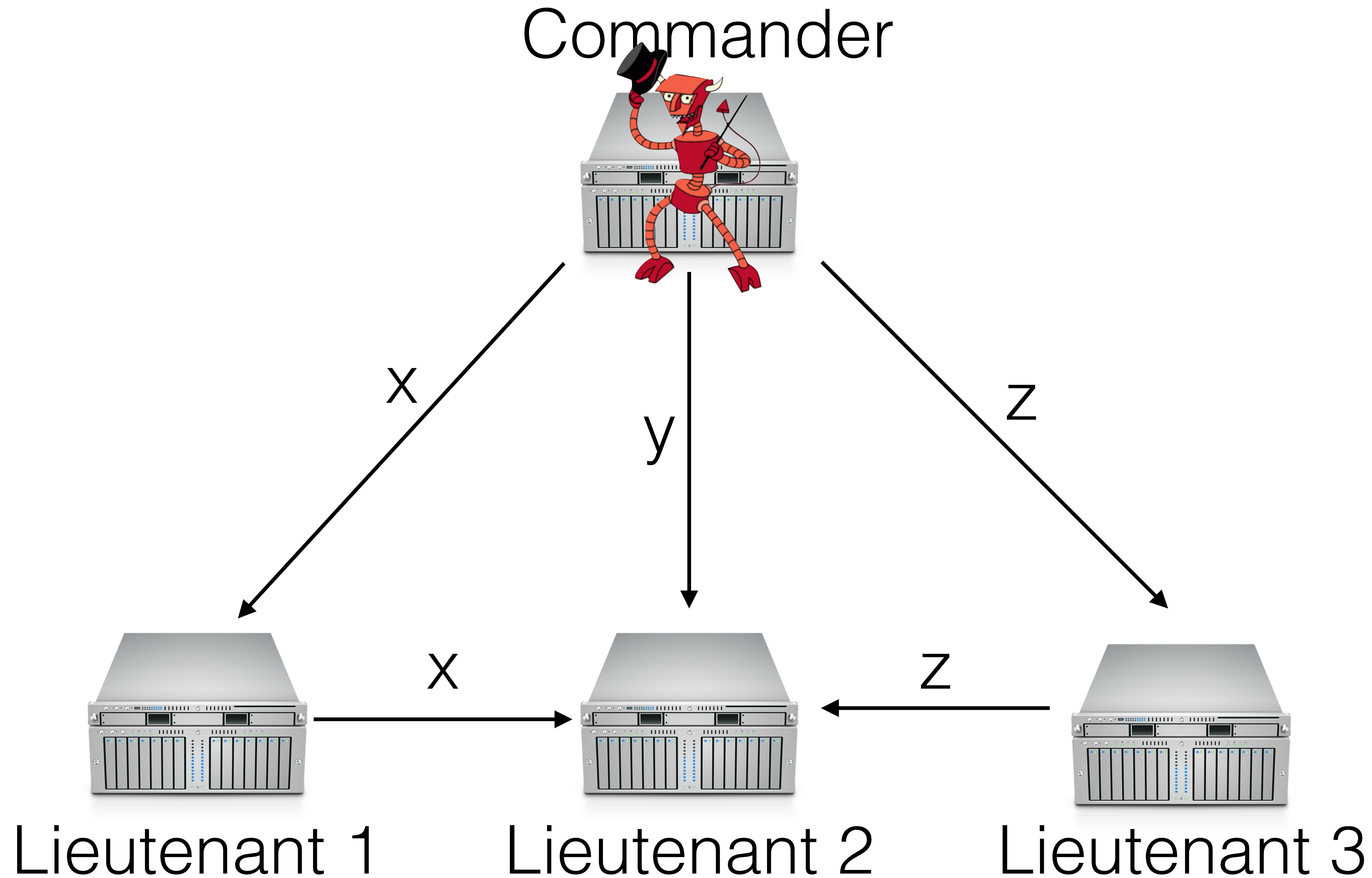
Oral BFT Solution (*m* traitors)

- 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

Signed BFT

- In the oral algorithm, a traitor can lie about the commander’s orders
- Signed BFT adds an additional assumption:
  - Messages are signed; a loyal participant's signature can not be forged; alteration of the messages contents can be detected
  - Anyone can verify a signature
- Algorithm SM(m):
- General signs and sends its value to each lieutenant
- For every lieutenant *i*:
  - If the order they receive has *m* distinct signature on it, then you are done
  - If not, then sign the order, forward to participants who have not signed it

Signed BFT

- Requires 2m+1 nodes to tolerate *m* byzantine faults
- Less messages than the oral approach (only need
- Tricky to implement a system that holds all of the assumptions we set out:
  - Every message sent is delivered correctly
  - Receiver knows who the sender is
  - Absence of a message can be detected
  - Loyal general's signature cannot be forged; any alteration of a signed message can be detected; anyone can verify authenticity of a general's signature

BFT Disclaimers

- Are byzantine failures truly random? (do they occur independently)
- Does not protect against all kinds of attacks against your system
  - E.g. steal sensitive data
- If anybody can join the network, then an adversary could overwhelm the voting process

Bitcoin

- Goal: Build a system for electronic cash, but without having any trust (of government, money holders, money changers)
- What's good (or not) about cash?
    - Portable
    - Can not spend twice
    - Can not repudiate after payment
    - No need for trusted 3rd party to do a single transaction
    - Doesn't work online
    - Easy to steal

What about credit cards (paypal, venmo, square)?

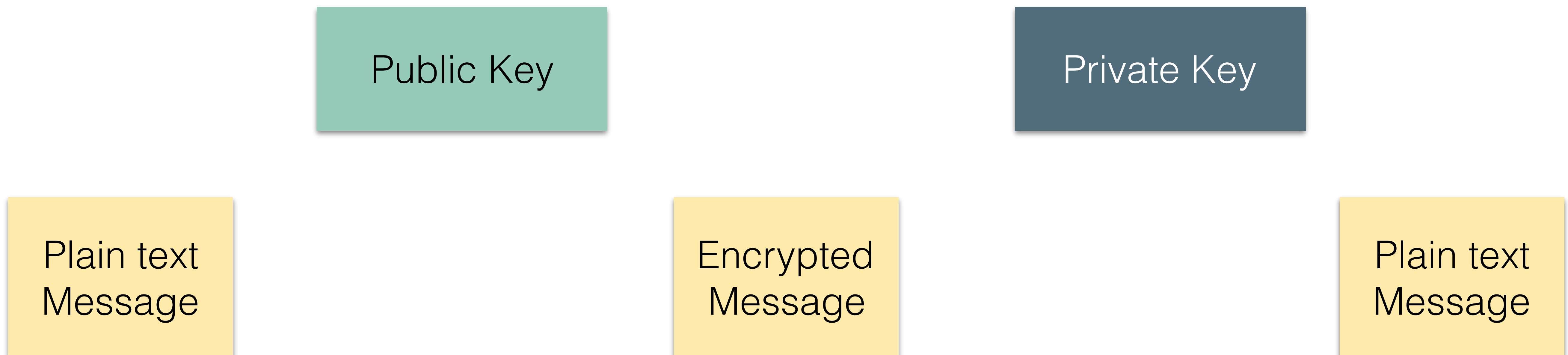
- Works online
- Somewhat hard to steal (need some knowledge)
- Can repudiate
- Requires trusted 3rd party
- Tracks all of your purchases

Bitcoin

- Works online
- Uses crypto-coins
- No central authority for issuing coins or tracking ownership of coins

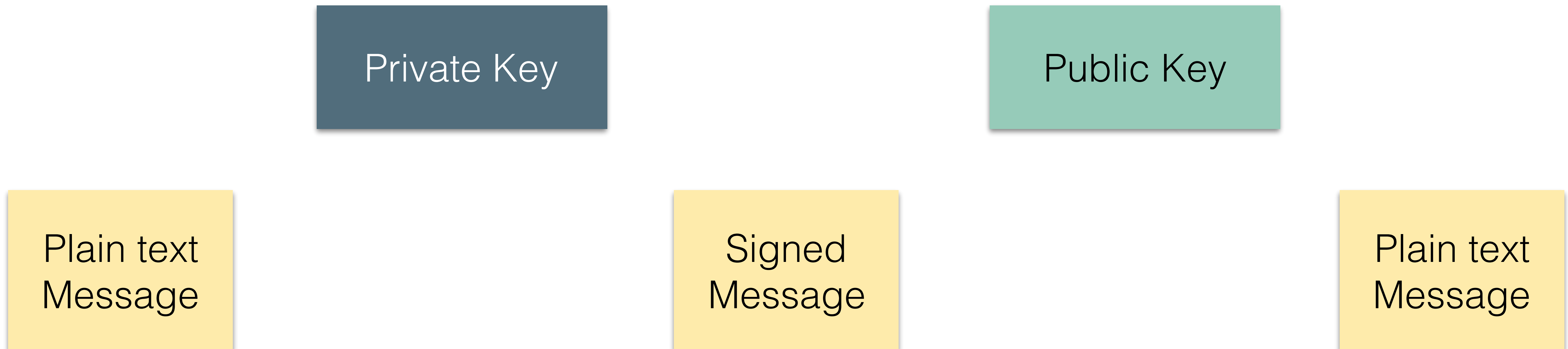
Cryptocurrencies

- Cryptocurrencies are based on public-key encryption
- Encryption review: Using public key, can send message that can only be read by holder of private key



Cryptocurrencies

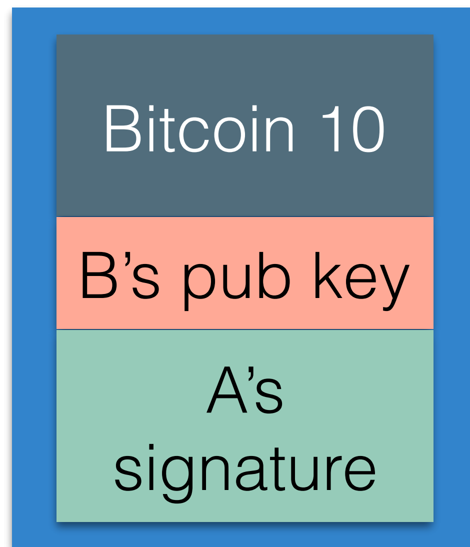
- Cryptocurrencies are based on public-key encryption
- Encryption review: Using private key, can send messages that can be verified came from us (using our public key)



Bitcoin

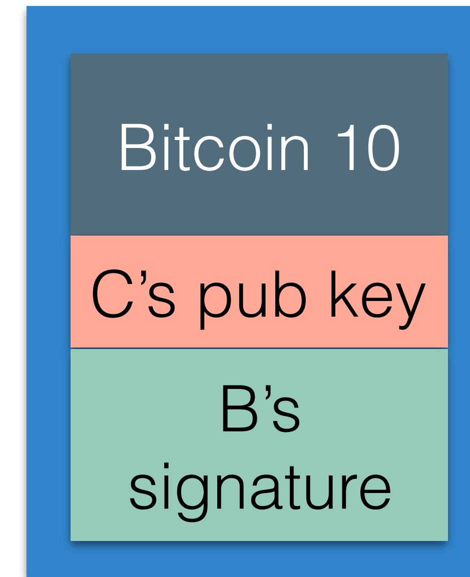
- If I own a bitcoin, then I have the private key that signed it; anyone can verify that I own it
- Transfer some bitcoin (say, #10) from A->B
  - A creates a record that has B's public key, plus the serial # of the coin that A is transferring
  - A signs it with their private key

Bitcoin: Example



Bitcoin Transaction 1

Transfers coin 10 from A to B



Bitcoin Transaction 2

Transfers coin 10 from B to C

Bitcoin

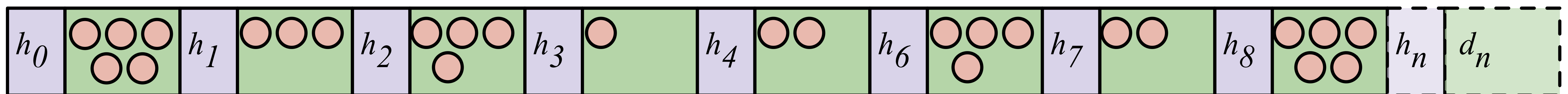
- Problem:
  - Where do the serial numbers come from?
  - How do we know that a coin is only spent once?
- Easy answer - use a bank/central party:
  - Bank issues serial numbers
  - Bank keeps track of who owns each coin; doesn't let you spend the same coin more than once
- Problem:
  - Want decentralized.

Blockchains

- Idea: make everyone that participates keep track of all records as a common log
- Each participant stores a replica of the log, broadcasts transactions to peers
- How do we keep the peers up to date though?
  - Paxos?
    - Requires everyone is trusted to not corrupt the log
  - Byzantine fault tolerant paxos?
    - Requires 2/3 trusted to not corrupt the log
    - How do you move forward even if you find corruption?
    - How easy is it to overwhelm the network with malicious colluding nodes?

Blockchains

- Solution: make it hard for participants to take over the network; provide rewards for participants so they will still participate
- Each participant stores the entire record of transactions as blocks
- Each block contains some number of transactions and the *hash* of the previous block
- All participants follow a set of rules to determine if a new block is valid



Blockchains

- How do we limit participation?
- Require a “proof of work”
- For the network to accept a new block, it must meet the following requirement:
  - hash(block,*nonce*) < target
  - *target* is picked a priori
  - *nonce* is a random value that the client is trying to guess

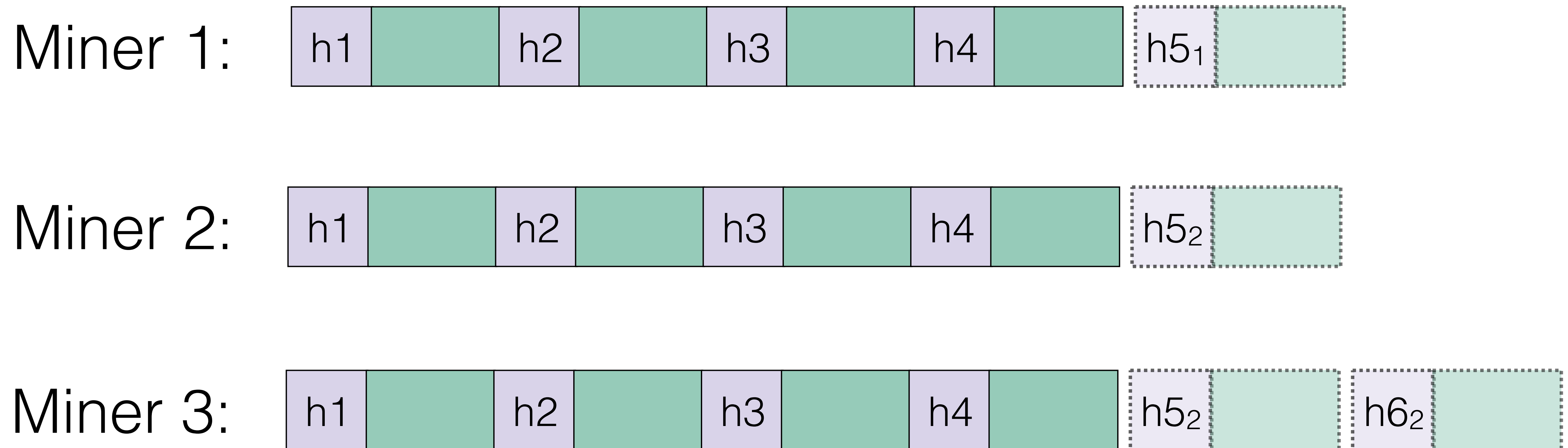
Proof of work

- Reminder: hashing
  - Takes some arbitrarily long input, produces a fixed-length
  - Same input gives same output
  - Making a subtle change in input can result in unpredictable change of output
- Proof of work:
  - hash(block data, nonce) < target
  - Requires brute force

Proof of work

- Each node that is trying to make a new block is called a *miner*
- Participants who want to make a transaction need to do so with the help of a miner, who will put it in a block
- Miners get paid to create blocks:
  - Transaction fees (roughly ~$0.10)
  - Reward for making a new block (currently 12.5 btc)

Blockchain's view of consensus



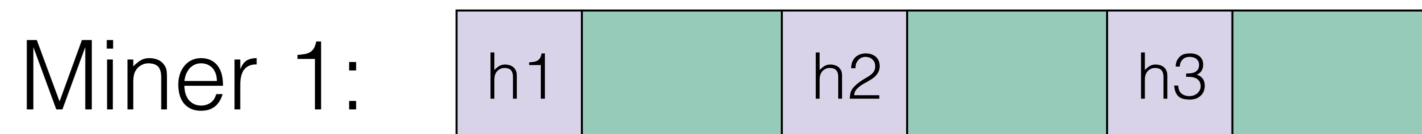
"Longest chain rule"

When is a block truly safe?

Attacks

With massive computation power, can rewrite history: nobody can prove which way it was supposed to be

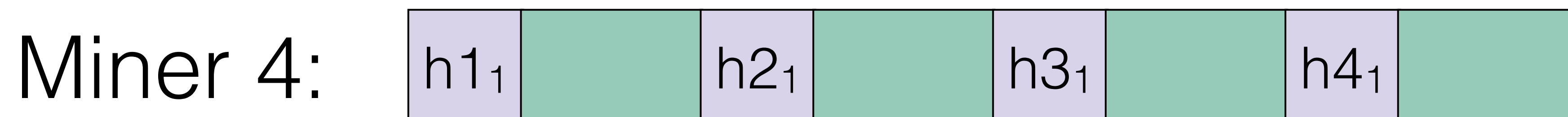
Miner 1:



Miner 2:

Miner 3:

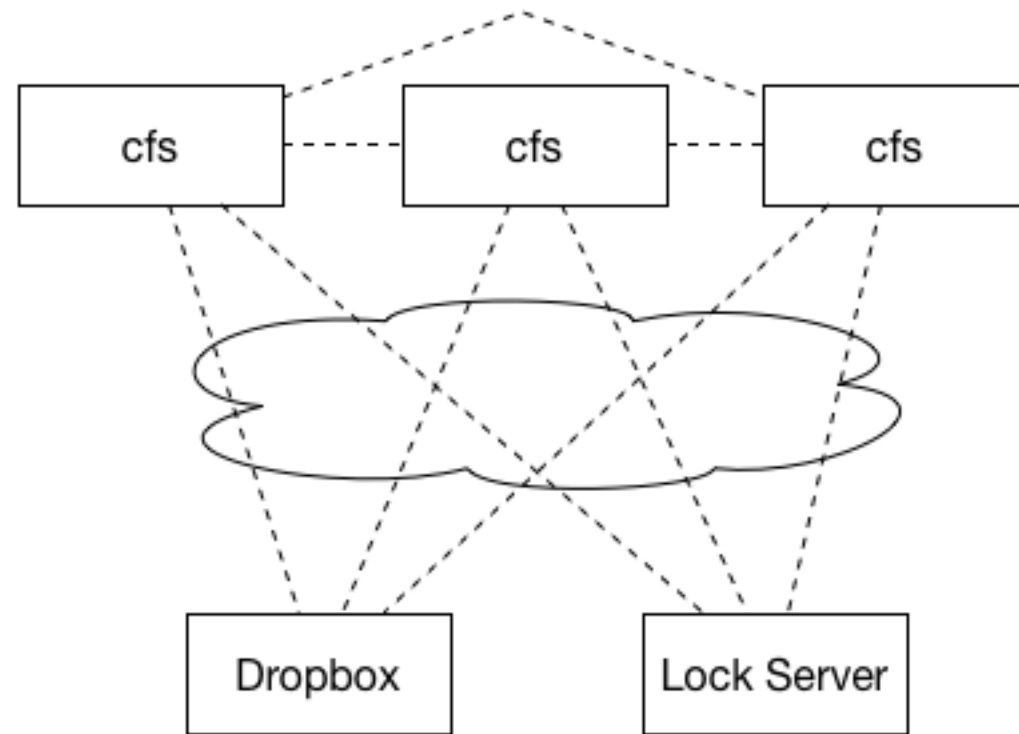
Miner 4:



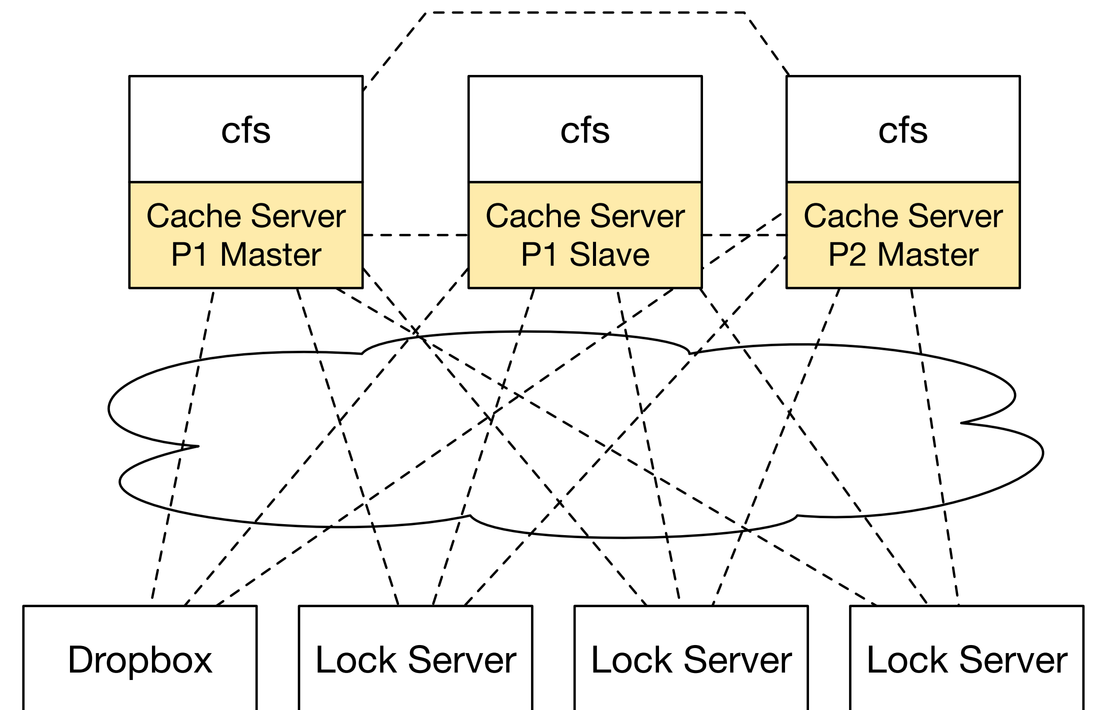
Blockchain & Trust

- Miners don't trust people submitting transactions
  - If you accept an invalid transaction then try to include it in your block, block is rejected
- Miners don't trust each other
  - If you include invalid transactions: rejected
- Nobody trusts miners
  - Requires expending effort to get a new block in

Introducing HW6 - the last one!

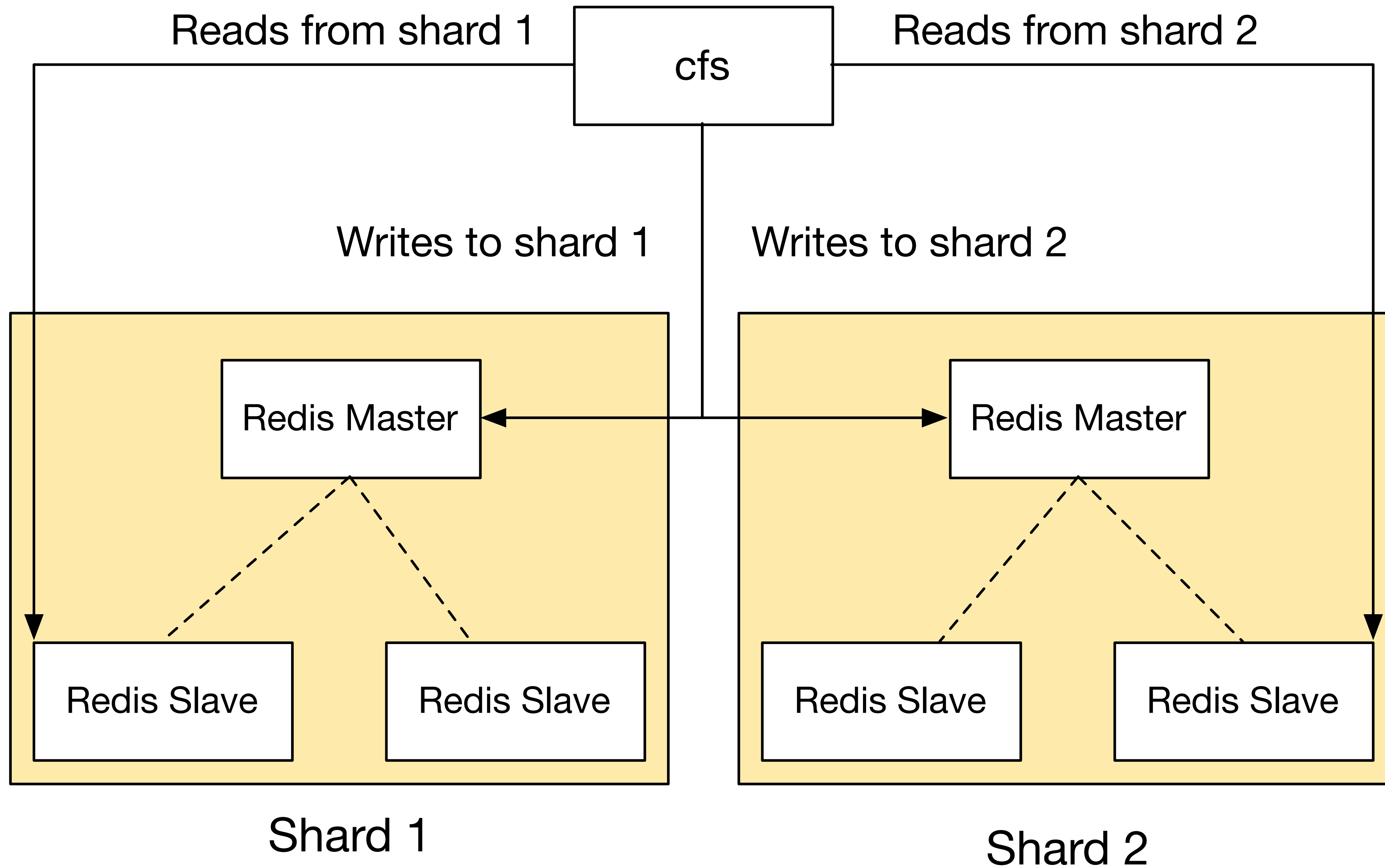


HW2



HW6

HW6



Lab 10

- A building block for HW6
- Curator's GroupMember is really annoying
  - start() is asynchronous
  - getMembers() is cached
- We're going to make our own GroupMember class
  - start() will block until joined
  - getMembers() will not be cached
- Will be used in HW6
- Build on a key building block from Curator: PersistentNode