Byzantine Fault Tolerance

CS 475, Spring 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

Review: 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

Review: 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

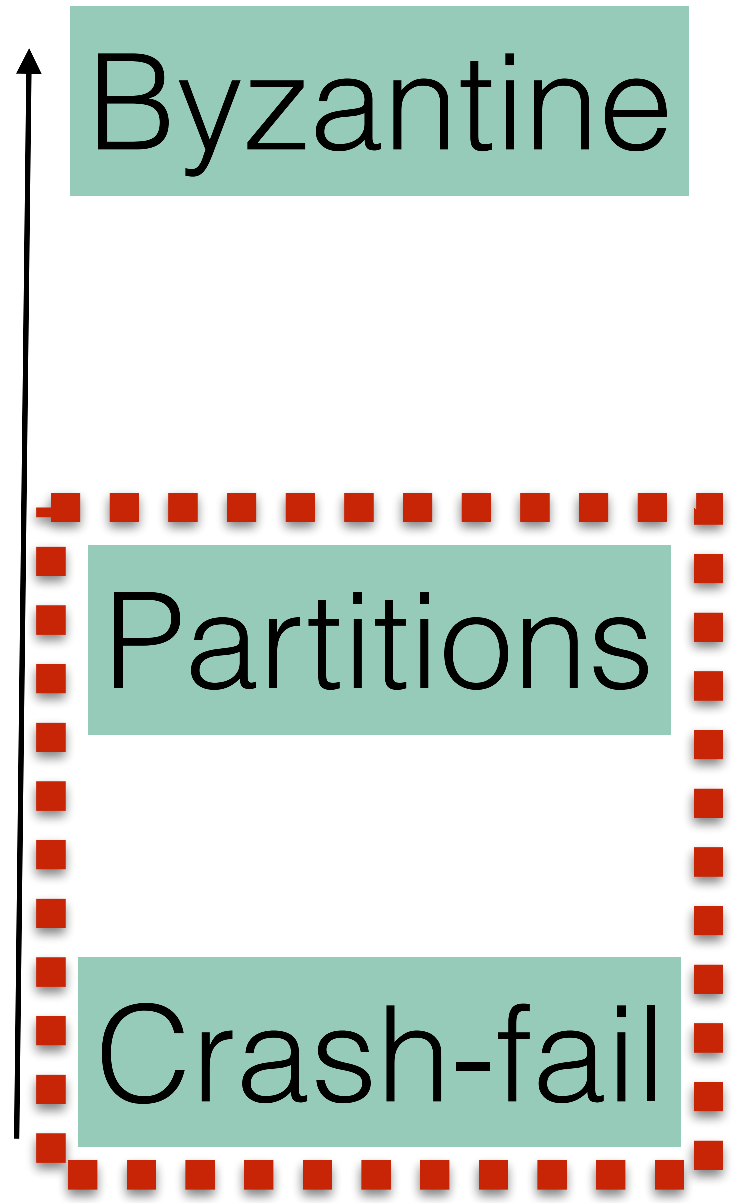
Review: Gnutella

- Gnutella’s search approach 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?

Review: 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*

Is *our system* well behaved?



Start talking about for P2P systems

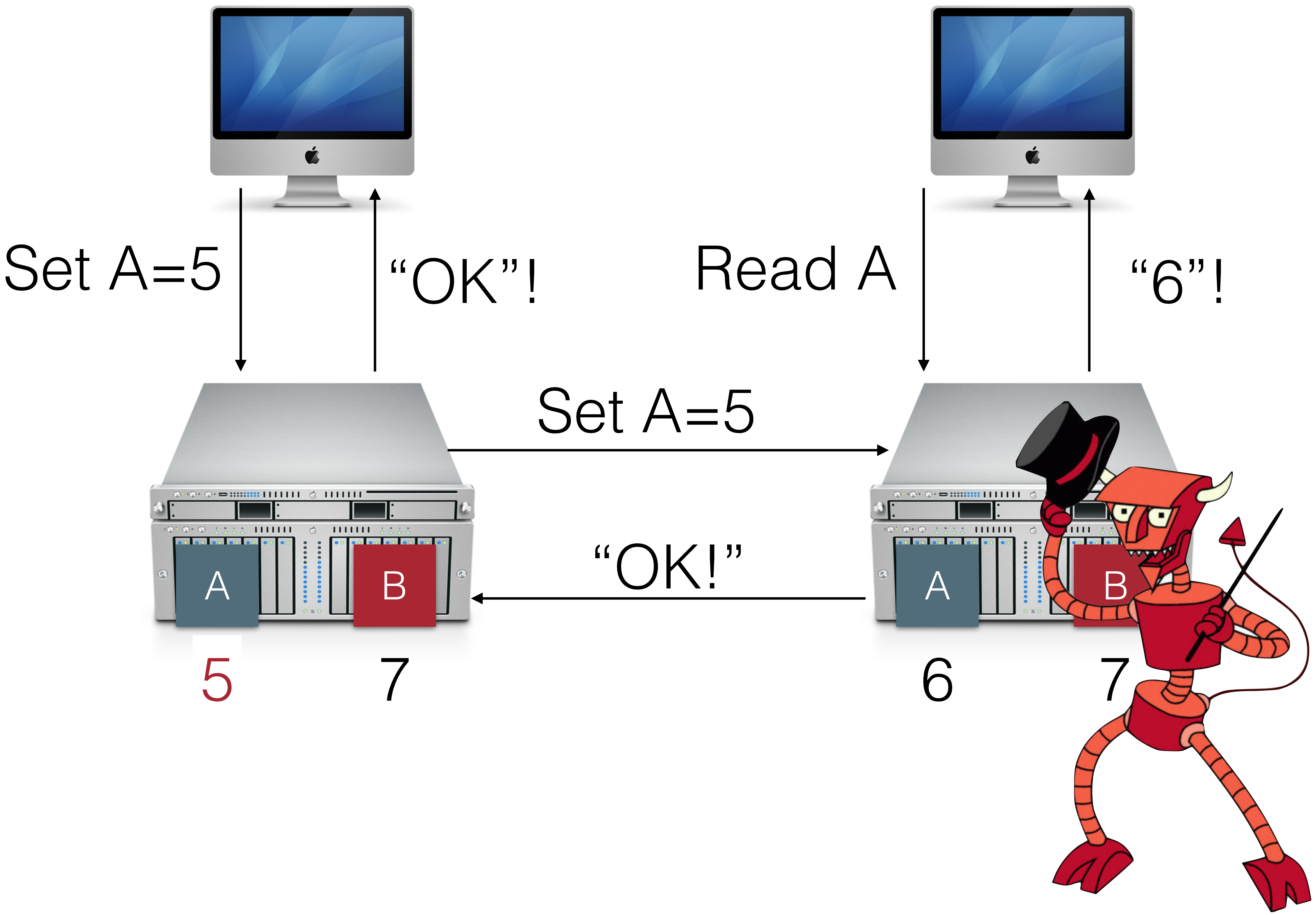
What we’ve done so far

Byzantine Failures in P2P

Today

- Today:
  - Byzantine failures
  - Blockchain
- Reminder - Course evaluation on Wednesday
- Reminder - Project is out!
  - Fault-tolerant, sequentially consistent replicated key value store
  - Can do in a group (1 to 3 students per group)

Byzantine Faults



Byzantine Faults

- 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)

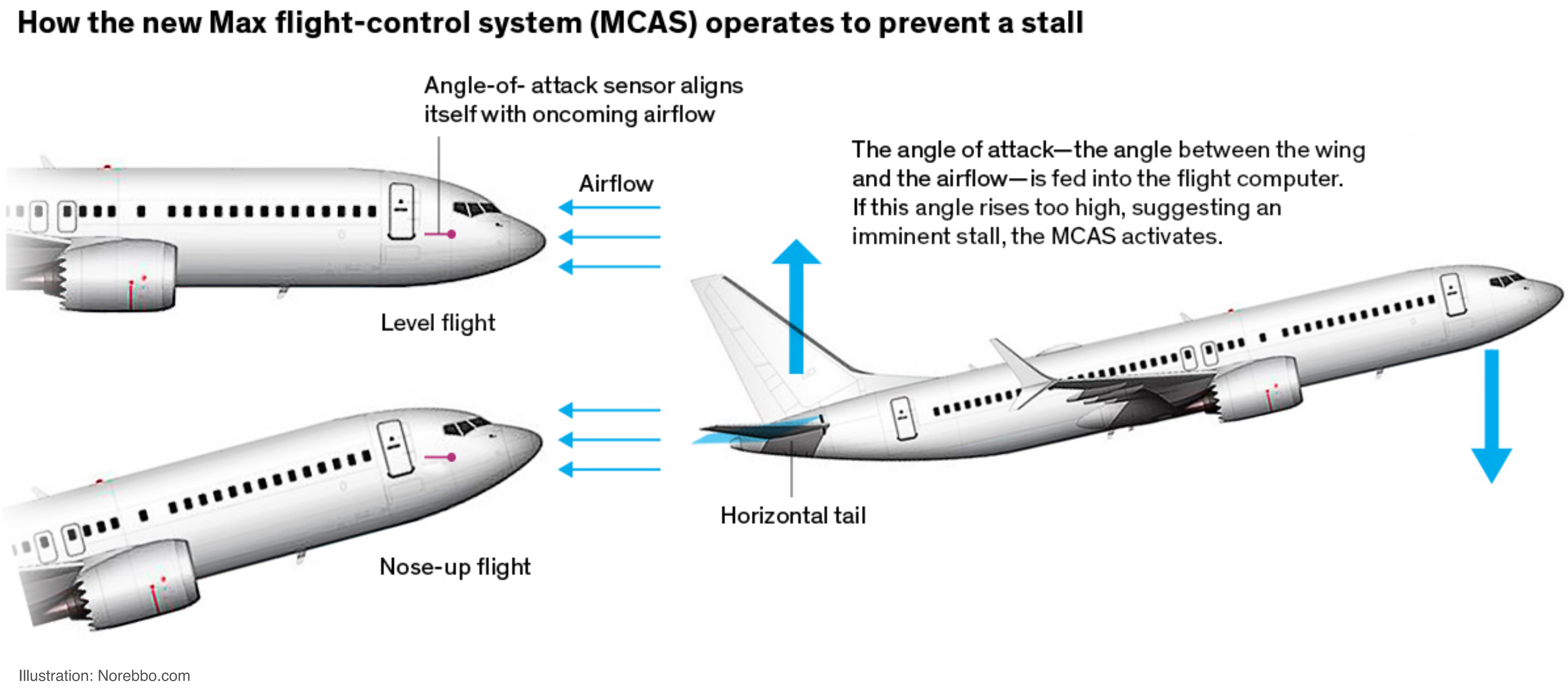
Byzantine Faults in Practice

- 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



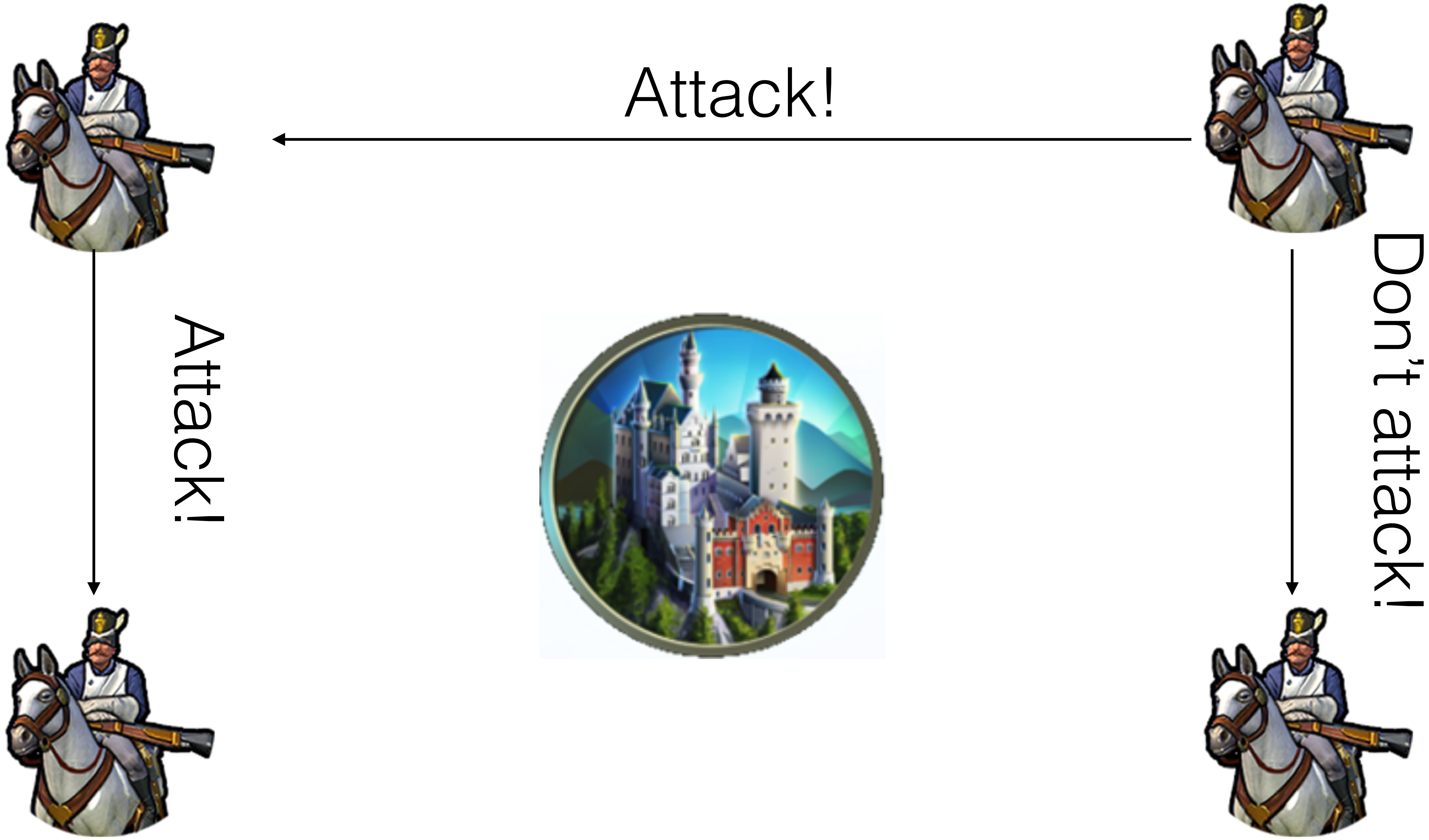
Byzantine Faults in Practice 737-MAX Edition

- 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 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

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
- 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
- Usually considered as one component of a broader threat model

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**!

Byzantine Faults: Summary

- Most systems are *not* byzantine fault tolerant
- In our best case scenario (the signed message BFT protocol), need 2m+1 nodes to tolerate *m* byzantine faults - expensive!
- Nonetheless - important to recognize what failures we *do* and *do not* tolerate
- We'll spend the rest of the lecture discussing a very popular BFT protocol… blockchain

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)?

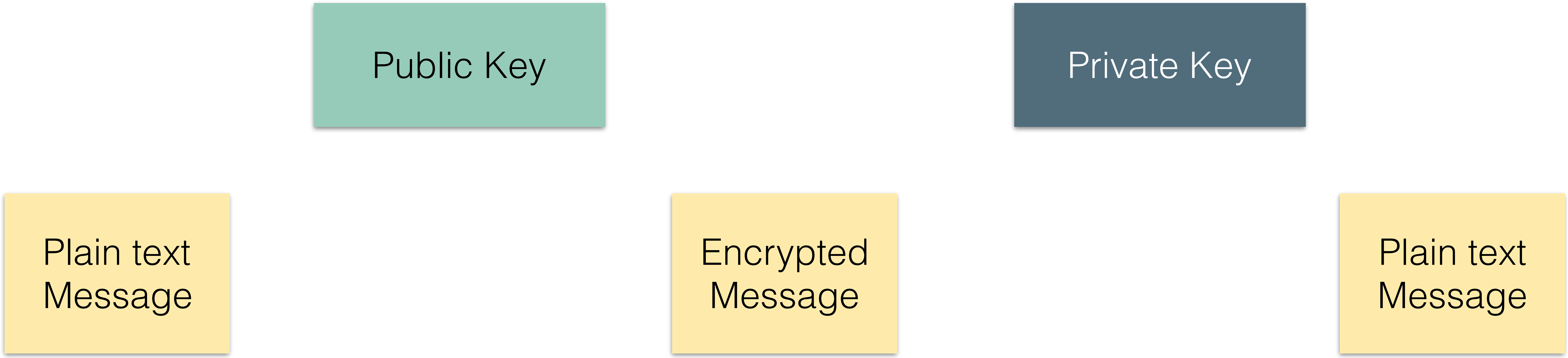
- Works online
- Somewhat hard to steal (need some knowledge)
- Can repudiate
- Requires trusted 3rd party
- Tracks all of your purchases

Bitcoin

- Works online
- Uses crypto-coins
- No central authority for issuing coins or tracking ownership of coins
- Its basis - blockchains - are a form of byzantine-fault-tolerant consensus!

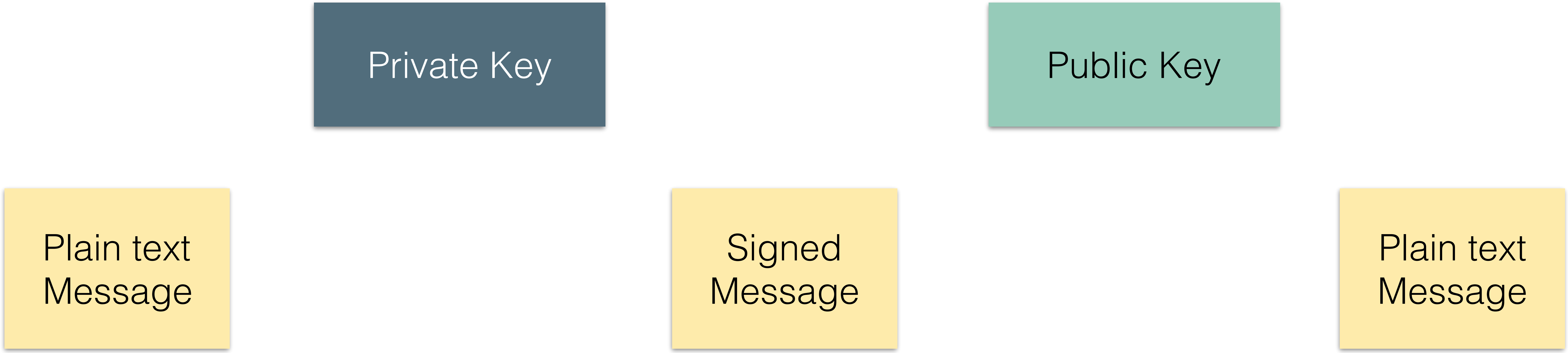
Cryptocurrencies

- 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

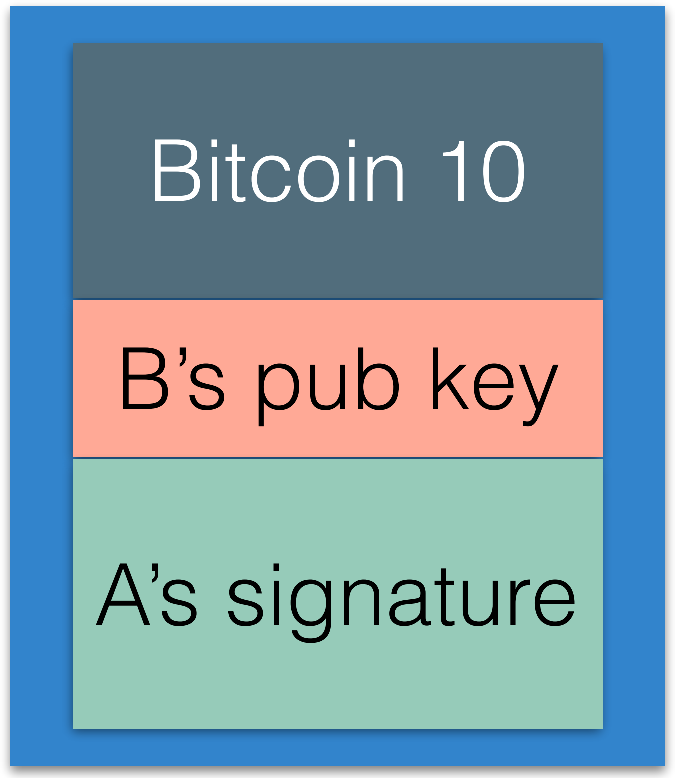
- Cryptocurrencies are based on public-key encryption
- Encryption review: Using private key, can send messages that can be verified came from us (using our public key)



Bitcoin

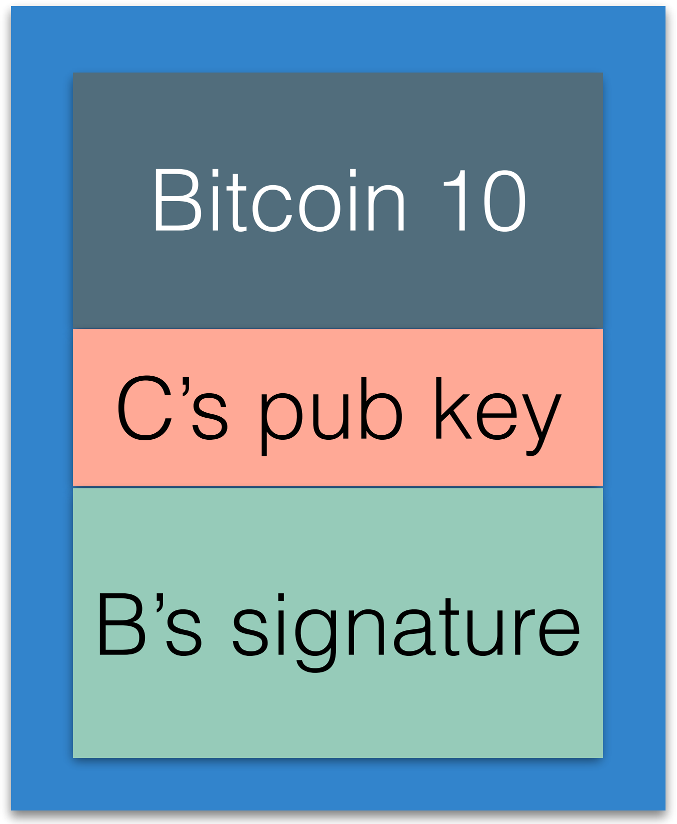
- If I own a bitcoin, then I have the private key that signed it; anyone can verify that I own it
- Transfer some bitcoin (say, #10) from A->B
  - A creates a record that has B's public key, plus the serial # of the coin that A is transferring
  - A signs it with their private key

Bitcoin: Example



Bitcoin Transaction 1

Transfers coin 10 from A to B



Bitcoin Transaction 2

Transfers coin 10 from B to C

Bitcoin

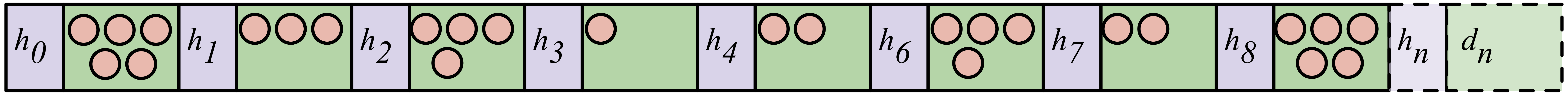
- Problem:
  - Where do the serial numbers come from?
  - How do we know that a coin is only spent once?
- Easy answer - use a bank/central party:
  - Bank issues serial numbers
  - Bank keeps track of who owns each coin; doesn't let you spend the same coin more than once
- Problem:
  - Want decentralized.

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

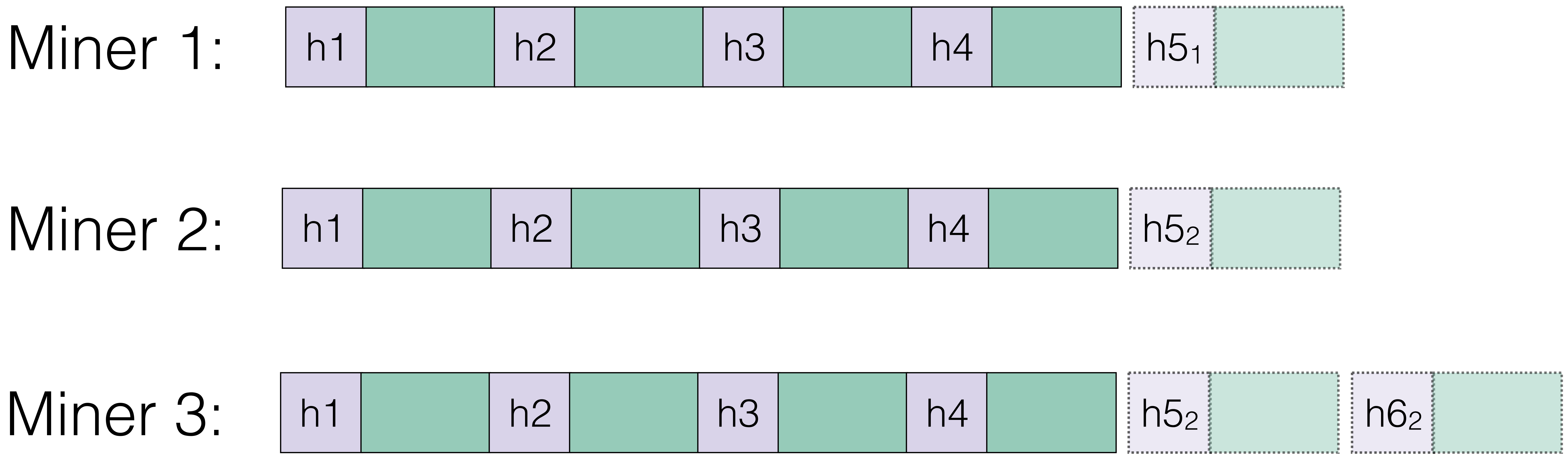
Proof of work

- 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

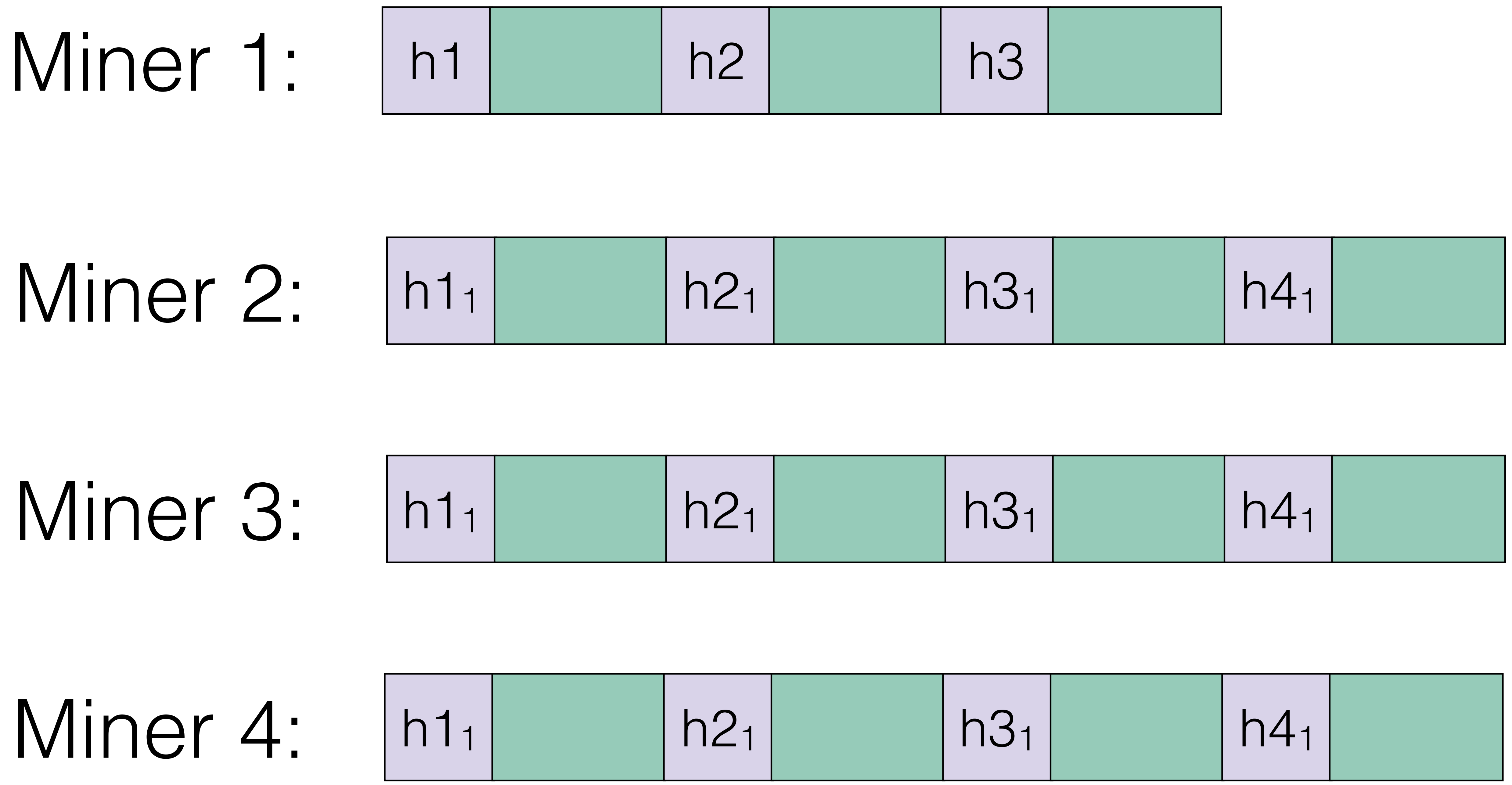


"Longest chain rule"
When is a block truly safe?

Attacks

- Worst case: attacker has 99% of mining capacity

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



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