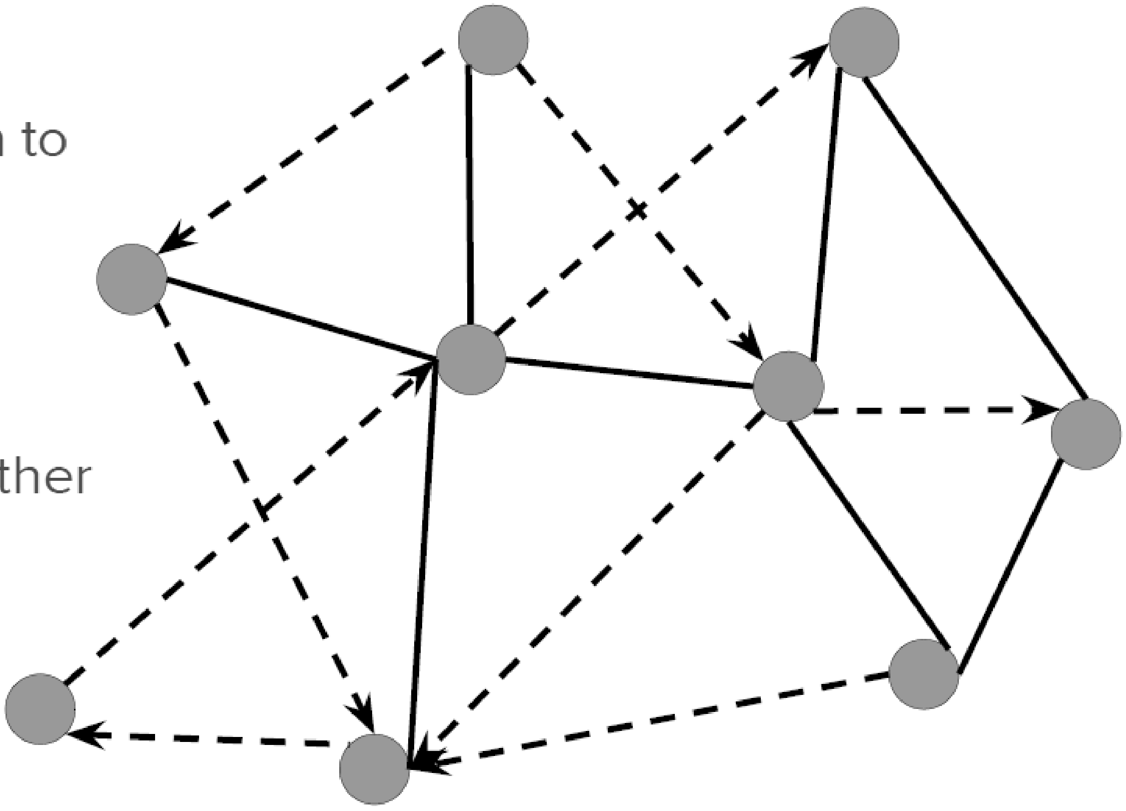
Course Outline

- ❑ Introduction
- ❑ Money and token economy
- ❑ Cryptography
- ❑ Blockchain data structure / Filecoin
- ❑ Mining and PoW
- ❑ Consensus algorithms
- ❑ Bitcoin as a platform
- ❑ Ethereum and smart contracts
- ❑ Distributed applications & enterprise DLT
- ❑ Security
- ❑ Scalability
- ❑ Privacy

Distributed System

Distributed System Definition

- Definition (varies from person to person):
  - A network of independent nodes, each representing a "process," talking to each other via messages
- Purpose:
  - Accomplish a common goal



Consensus Protocol of Distributed System

- Consensus protocol specifies how to get multiple nodes to agree on a value—e.g., if a data item should be added to the blockchain
- Traditional distributed systems target closed systems has developed robust and practical protocols that can tolerate faulty and malicious nodes
- Application to open blockchain revitalized the field
  - Bitcoin enables consensus among an open, decentralized group of nodes via a leader election based on proof-of-work (PoW)
  - Bitcoin is weak consistency - different nodes might end up having different views of the blockchain leading to forks due to
    - ✓ probabilistic leader election process
    - ✓ performance fluctuations in decentralized networks
- Repurpose classical consensus protocols for decentralized open blockchains to achieve strong consistency and similar performance such as mainstream payment systems

PROPERTIES OF MULTIPROCESS PROGRAMS

- **Safety**
  - Formal Definition - The current state guarantees the safety property if all the future states that can be reached from the current state also satisfy the safety property
  - Intuitive Definition - “guarantee that something bad will never happen”
  - In consensus - no two processes decide on different values
- **Liveness**
  - Formal Definition - The current state satisfies liveness if there is some causal path of global steps from the current state to some future state where the liveness property holds true
  - Intuitive Definition - “guarantee that something good happens eventually”
  - In consensus - all processes eventually decide on a value

PROPERTIES OF MULTIPROCESS PROGRAMS

Property	Safety	Liveness
Definition	Never returning a false value	Returning some value

This will *not* happen	This *must* happen

- https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Proving-the-Correctness-of-Multiprocess-Programs.pdf
- **"Proving the Correctness of Multiprocess Programs" – Leslie Lamport, IEEE Transactions on Software Engineering, March 1977**

Network Synchrony Taxonomy

- Synchronous networks - the delays messages may suffer can be bound by some time Δ.
- Asynchronous networks - messages may be delayed arbitrarily, and there exists no reliable bound Δ for their delay.
- Partially synchronous, or eventually synchronous networks, assume that the network at some stage will eventually be synchronous despite potentially a long period of asynchrony

DISTRIBUTED PROPERTIES IN CONSENSUS

- ❑ In an asynchronous network, distributed system cannot guarantee decisions (Liveness) and correct decisions (Safety) immediately
- ❑ Can only guarantee eventual liveness or eventual consistency.
- ❑ Eventual consistency means if no new updates are made to a data item, eventually all accesses to that item will return the last updated value
- ❑ Eventual consistency in terms of proof of work - all nodes on the network will eventually share a common view of the state of the blockchain.

PROOF OF Work – Nakamoto Consensus

- **Proof-of-work**
  - 1993 by Dwork and Naor as a technique for combatting spam mail
- **SAFETY**
  - Forks may happen and resolved by accepting the longest chain
  - Eventual consistency - all nodes will eventually share a common view of the state of the blockchain
  - Security threshold 51% attack
- **LIVENESS**
  - Economical incentivizes for miners to validate and mine blocks, by rewarding them with new coins
  - Verifiable process of weighted random coin-tossing - the winning is in proportion to the resources (hash-rate)

PROOF OF STAKE

- **ASSUMPTION**
  - Mining is done by stakeholders who have the strongest incentives to be good stewards of the system
- **SAFETY**
  - Someone with 51% or more stake has an incentive to do things that would benefit the system as a whole as it will increase the value of the coins they hold
- **LIVENESS**
  - “Validators” chosen randomly based on amount of coin ownership from the group of people who invest in the proof of stake mechanism
  - “Validators” produce a block at each round as they are paid transaction fees for block creation
  - Signers or rest of the nodes commit the block to the blockchain

PROOF OF STAKE

- **Advantages:**
  - Reduces electricity consumption to secure blockchain
  - Amount you likely gain directly proportionate to amount you invest
  - Reduced risk of centralization as no mining hardware involved
- **Issues:**
  - Dislodging 51% power is impossible
  - Nothing at Stake problem
    - ✓ Ethereum tries to get around this by requiring miner to sign the block with the private key corresponding to the transactions that make up the miner's stake
    - ✓ If miner tries to sign two forked chains, a miner later on can input these two signatures as proof of mis-behaviour and collect a portion of this as bounty

Fault Tolerant System - DIGITAL AVIONICS

Super dependable computers pioneered by aircraft manufacturers

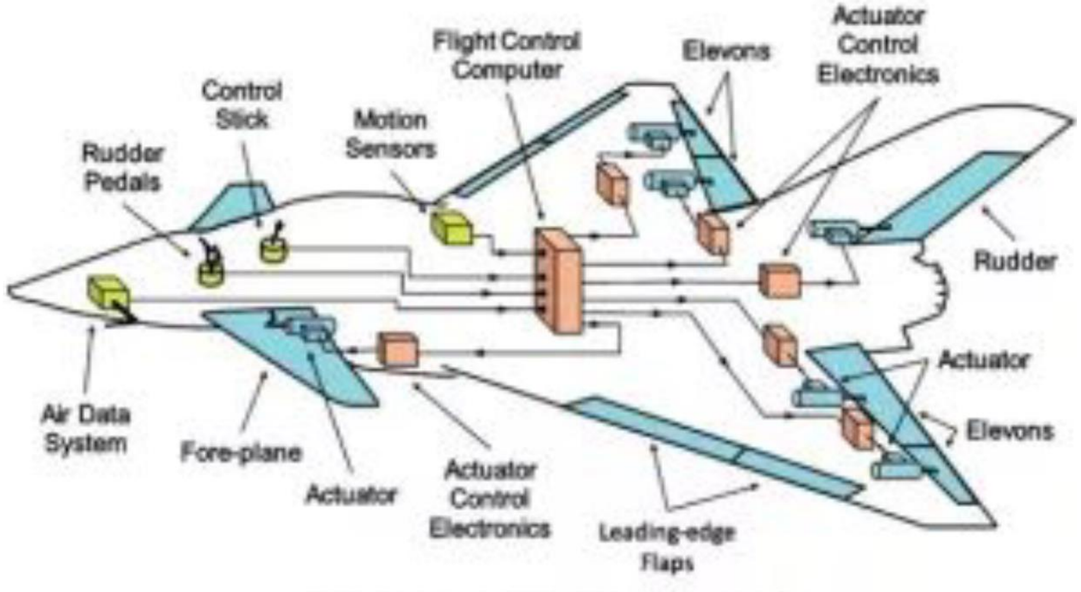
- Safety first, especially…
  - when you’re flying 60-120 million USD aircraft
  - when there’s 100s of passengers

- Fault Tolerant Avionics:
  - http://www.davi.ws/avionics/TheAvionicsHandbook_Cap_28.pdf

DIGITAL AVIONICS - 550 passenger Boeing 777

ARINC 659 SAFEbus network

- ❑ nodes: computers, sensors
  - ❖ duplicate transmitters that control nodes
  - ❖ silence node if transmitters informed of too many errors
- ❑ recipient nodes receive 4 copies of message
- ❑ accept/record only if 4 identical messages



Basic elements of the FWB control system.

DIGITAL AVIONICS - SpaceX Dragon

- NASA requirements for ISS
- Triply redundant computers
- Radiation events changing memory or register values

SOURCE: https://www.universetoday.com/112272/meet-spacexs-new-manned-dragon-cool-animation-shows-how-it-works/

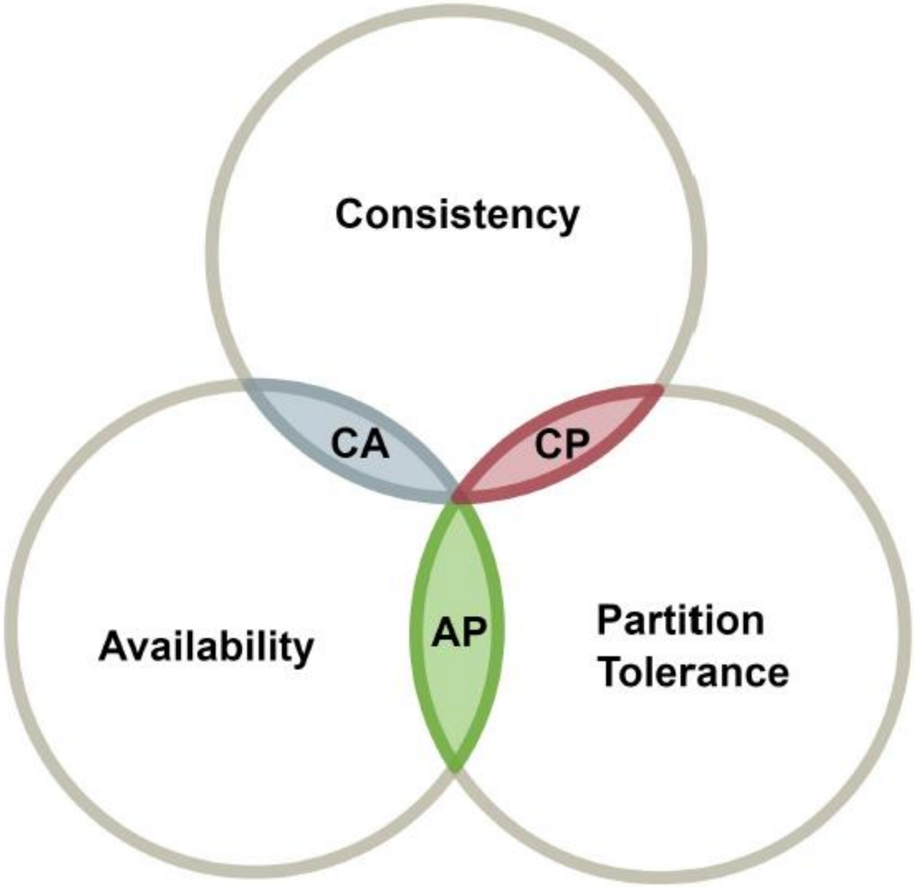
Correctness of a distributed system

Correctness of a distributed system: achieving its intended goal

To ensure correctness, one uses a **consensus algorithm** achieving the following:

- **Validity:** any value decided upon must be proposed by one of the processes
- **Agreement:** all non-faulty processes must agree on the same value
  - Agreement and Validity are *safety* properties: Honest nodes will never decide on trivial, random, or different values
- **Termination:** all non-faulty nodes eventually decide.
  - Termination is a *liveness* property: All nodes eventually decide on a value

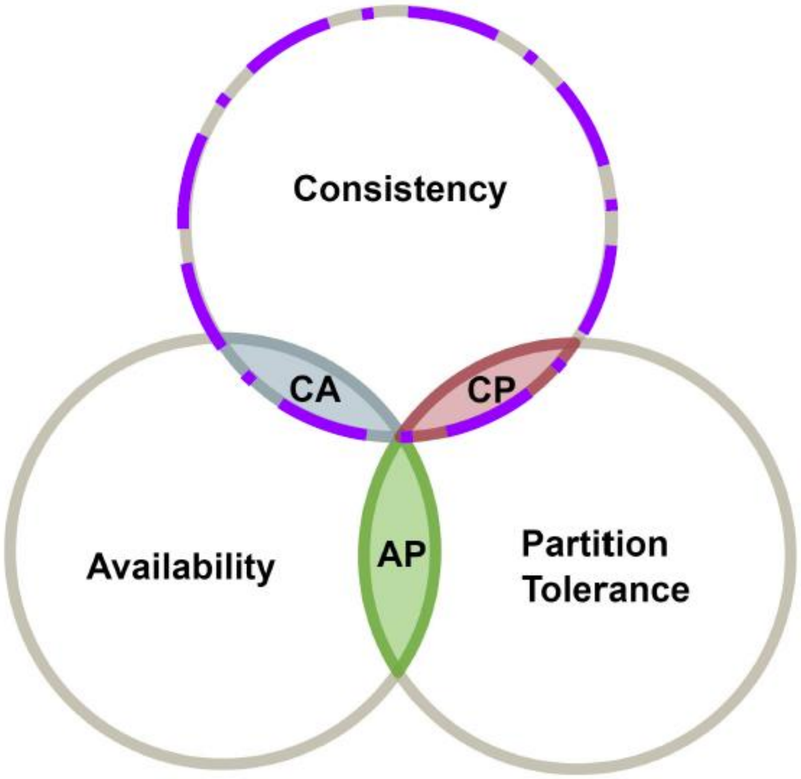
CAP THEOREM



CAP Theorem: Fundamental theorem for any distributed system pertaining to achievable properties

https://www.youtube.com/watch?v=Jw1iFr4v58M

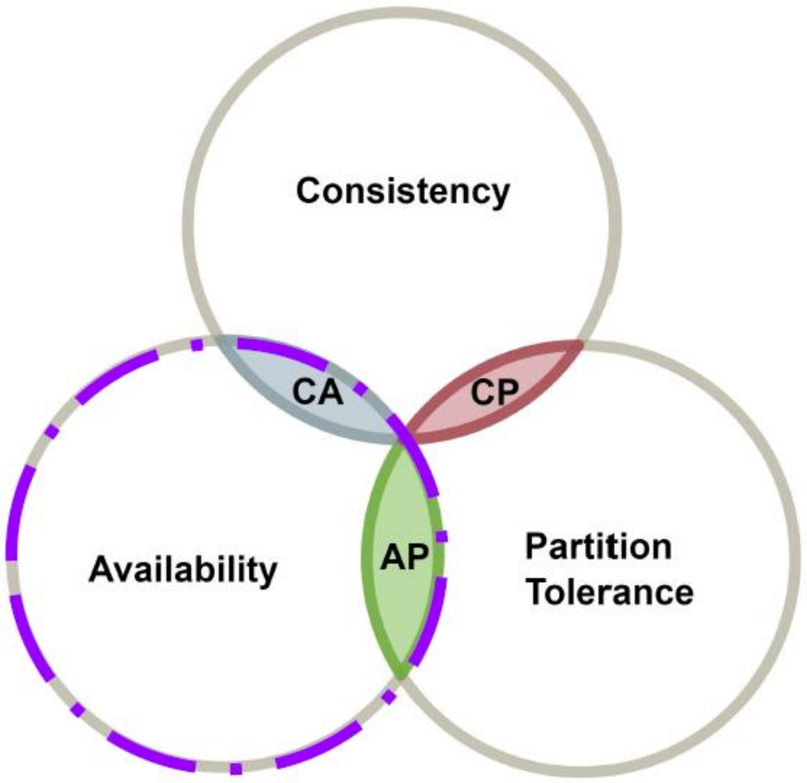
CAP THEOREM



Consistency:
Every node provides the most recent state

https://www.youtube.com/watch?v=Jw1iFr4v58M

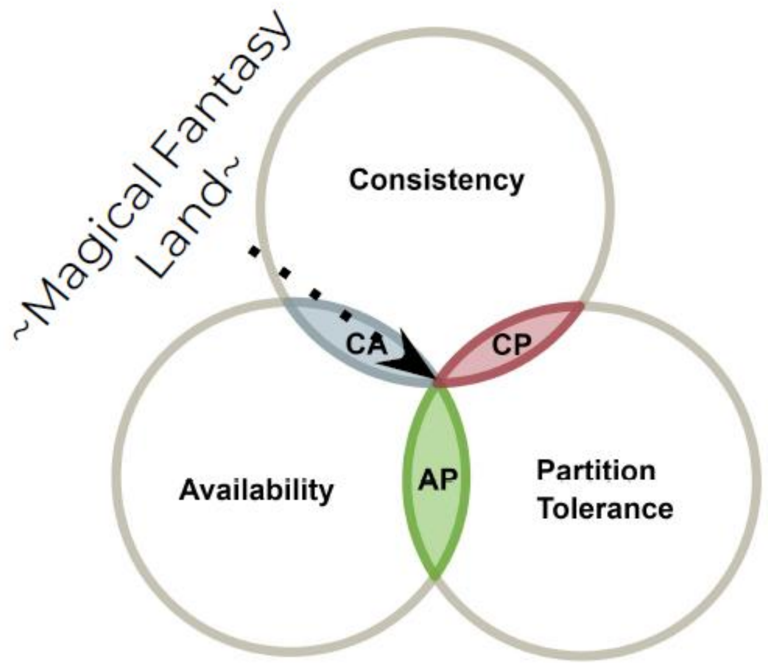
CAP THEOREM



Availability:
Every node has consistent read and write access

https://www.youtube.com/watch?v=Jw1iFr4v58M

CAP THEOREM

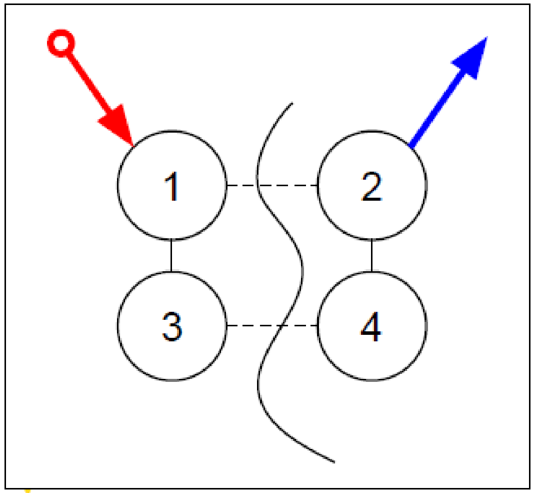


Can only have two of three

https://www.youtube.com/watch?v=Jw1iFr4v58M

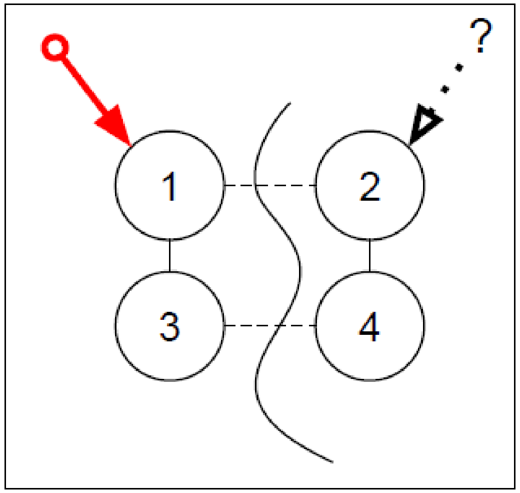
Proof of CAP THEOREM

Partition Tolerant + Available = **Not Consistent**



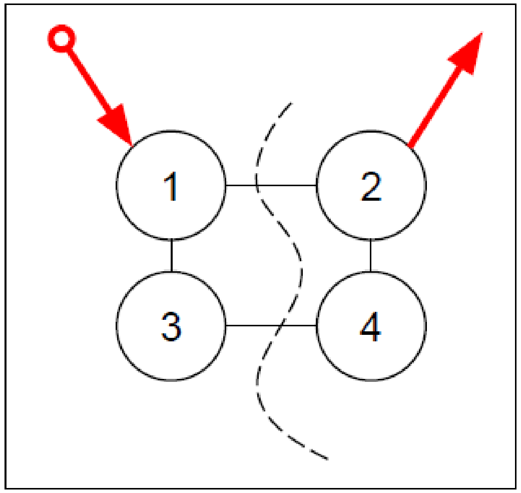
Safety

Partition Tolerant + Consistent = **Not Available**



Liveness

Consistent + Available = **Not Partition Tolerant**



Fault Tolerance

VOTING-BASED CONSENSUS ALGORITHMS

Client-Server Model

Naive Client-Server Algorithm

- ❖ Client sends commands one at a time to server

Problem: message loss or corrupted

Client-Server Algorithm with Acknowledgments

- ❖ Client sends commands one at a time to server
- ❖ Server acknowledges every command
- ❖ If the client does not receive an acknowledgment within a reasonable time, the client resends the command

Problem:

- ❖ both command and ack messages can get lost
- ❖ Sequence number to each message to prevent multiple execution (e.g. reliable protocol TCP)

Client-Server Model

[**Theorem**] If Client-Server Algorithm with Acknowledgments is used with multiple clients and multiple servers, the servers might see the commands in different order, leading to an inconsistent state.

[Proof] Assume two clients u1 and u2, and two servers s1 and s2. Both clients issue a command to update a variable x on the servers, initially x = 0.
Client u1 sends command x = x + 1, and client u2 sends x = 2*x.
With message delay, assume s1 receives the message from u1 first, and s2 from u2 first.
Hence, s1 computes x = (0 + 1)*2 = 2,
and s2 computes x = (0*2) + 1 = 1

State Replication

State Replication: A set of nodes achieves state replication, if all nodes execute a (potentially infinite) sequence of commands c1, c2, c3, … , in the same order.

- Fundamental property for distributed systems - synonymous with the term blockchain
- State replication is trivial with a single server - designate as a serializer

[State Replication with a Serializer Algorithm] Master-Slave replication

- ❑ Clients send commands one at a time to the serializer
- ❑ Serializer forwards commands one at a time to all other servers
- ❑ Once the serializer received all acknowledgments, it notifies the client about the success

Problem: The serializer is a single point of failure

Mutual Exclusive Lock – More Distributed Approach

Two-Phase (2PL) Protocol

by Jim Gray in 1978

```
Phase 1
    Client asks all servers for the lock

Phase 2
    if client receives lock from every server then
        Client sends command reliably to each server, and gives the
        lock back
    else
        Clients gives the received locks back
        Client waits, and then starts with Phase 1 again
    end if
```

Issues with Mutual Exclusion Lock

- ❑ Cannot handle exceptions
- ❑ How about node crashes? …worse than the serializer approach: instead of only one node which must be available, it requires all servers to be responsive
- ❑ What if we only get the lock from a subset of servers? Is a majority of servers enough?
- ❑ What if two or more clients concurrently try to acquire a majority of locks? Do clients have to abandon their already acquired locks, in order not to run into a deadlock?
- ❑ What if they crash before they can release the locks? Do we need a slightly different concept?

Three-Round Protocols – Three-Phase Commit (3PC)

- ❑ 3PC allows recovery — the distributed resource managers being able to release the locks held on resources

- ❑ What if they crash before they can release the locks? Do we need a slightly different concept?

Ticket - Weaker Form of a Lock

[**Ticket**]. A ticket is a weaker form of a lock, with the following properties:

- ❑ **Re-issuable**: A server can issue a ticket, even if previously issued tickets have not yet been returned
- ❑ **Ticket expiration**: If a client sends a message to a server using a previously acquired ticket t, the server will only accept t, if t is the most recently issued ticket

- No problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected, as servers can simply issue new tickets.
- Tickets implemented with a counter: each time a ticket is requested, the counter is increased. When a client tries to use a ticket, the server can determine if the ticket is expired
- Can we replace the locks with tickets? need additional phase, as only the client knows if he has majority of the tickets

Naive Ticket Protocol

```
Phase 1
    Client asks all servers for a ticket
Phase 2
    if a majority of the servers replied then
        Client sends command together with ticket to each server
        Server stores command only if ticket is still valid, and replies to client
    else
        Client waits, and then starts with Phase 1 again
    end if
Phase 3
    if client hears a positive answer from a majority of the servers
    then
        Client tells servers to execute the stored command
    else
        Client waits, and then starts with Phase 1 again
    end if
```

Issues with Naive Ticket Protocol

- Let u1 be the first client that successfully stores its command c1 on a majority of the servers. If u1 is very slow to notify the servers, and another u2 updates the stored command in some servers to c2. Afterwards, u1 tells the servers to execute the command c1. So some servers will execute c1 and others c2
- What if a server, instead of only handing out tickets in Phase 1, also notifies clients about its currently stored command? Then, u2 learns u1 already stored c1, u2 could also store c1. As both store/execute same command, the order does not matter
- What if not all servers have same command stored, which u2 takes?
- …safe to support most recently stored command
- …support the value with majority votes
- Ticket number used to store the command can be used to determine which command was stored most recently
- What if each server uses its own ticket numbers? … clients suggest the ticket numbers themselves - globally consistent ticket number

PAXOS - Overview

- Paxos is a Greek island, but also . . . A CONSENSUS ALGORITHM

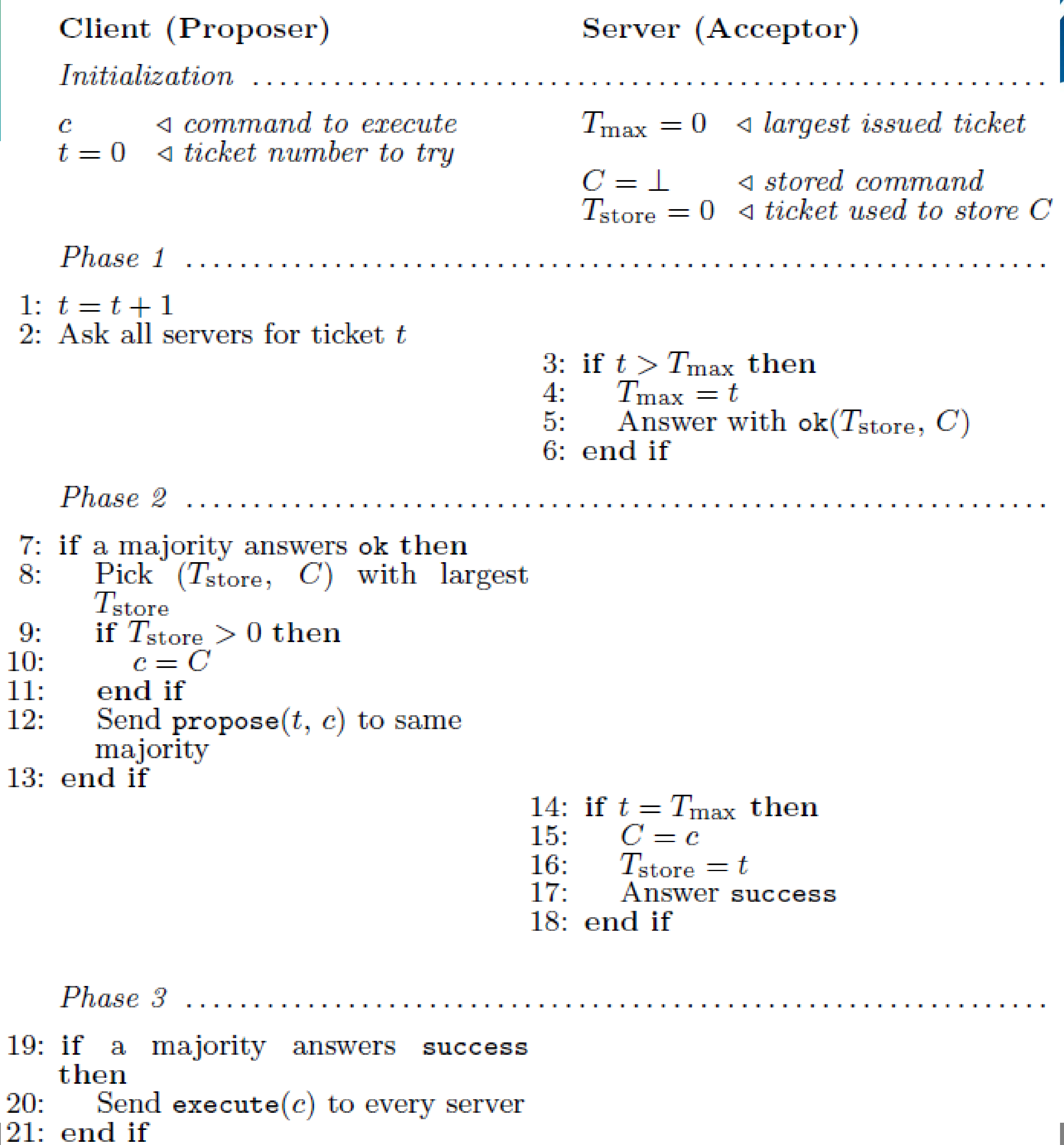
- **Source: http://paxos-map.com/paxos-island-general-information/**

PAXOS TERMINOLOGY

Proposer: advocates a client request, moves protocol forward
Acceptor: a voter
Learner: remembers result, send message to the client

Quorum:
- any majority of Acceptors
- subset of Acceptors, such that any two Quorums share at least one member (they must overlap)

PAXOS Algorithm



```
Client (Proposer)                              Server (Acceptor)
Initialization ..........................................................
$c$         ◁ command to execute               $T_{\max} = 0$     ◁ largest issued ticket
$t = 0$     ◁ ticket number to try
                                               $C = \bot$         ◁ stored command
                                               $T_{\text{store}} = 0$   ◁ ticket used to store $C$

Phase 1 .................................................................
1: $t = t + 1$
2: Ask all servers for ticket $t$
                                               3: if $t > T_{\max}$ then
                                               4:     $T_{\max} = t$
                                               5:     Answer with ok($T_{\text{store}}$, $C$)
                                               6: end if

Phase 2 .................................................................
7: if a majority answers ok then
8:     Pick ($T_{\text{store}}$, $C$) with largest
       $T_{\text{store}}$
9:     if $T_{\text{store}} > 0$ then
10:        $c = C$
11:    end if
12:    Send propose($t$, $c$) to same
       majority
13: end if
                                               14: if $t = T_{\max}$ then
                                               15:     $C = c$
                                               16:     $T_{\text{store}} = t$
                                               17:     Answer success
                                               18: end if

Phase 3 .................................................................
19: if a majority answers success
    then
20:     Send execute($c$) to every server
21: end if
```

PAXOS MAIN IDEA

SAFETY

- guaranteed by a 2 phase mechanism:
- Ph1 :Prepare request
  - Proposer sends a proposal with a unique serial number
  - Each new proposal increments the serial number
  - Acceptor not to look at any proposals with lower serial numbers
  - Acceptor replies with serial number of last proposal accepted by him
- Ph2: Accept request
  - Proposal with maximum responses is accepted

LIVENESS

- Sometimes there are scenarios where decisions cannot be made, consider only 2 nodes in the network
- Instead guarantees failure of nodes at any point won't affect entire network

PAXOS MAIN IDEA

[Lemma]. We call a message propose(t,c) sent by clients on Line 12 a proposal for (t,c). A proposal for (t,c) is chosen, if it is stored by a majority of servers (Line 15). For every issued propose(t',c') with t' > t holds that c' = c, if there was a chosen propose(t,c)

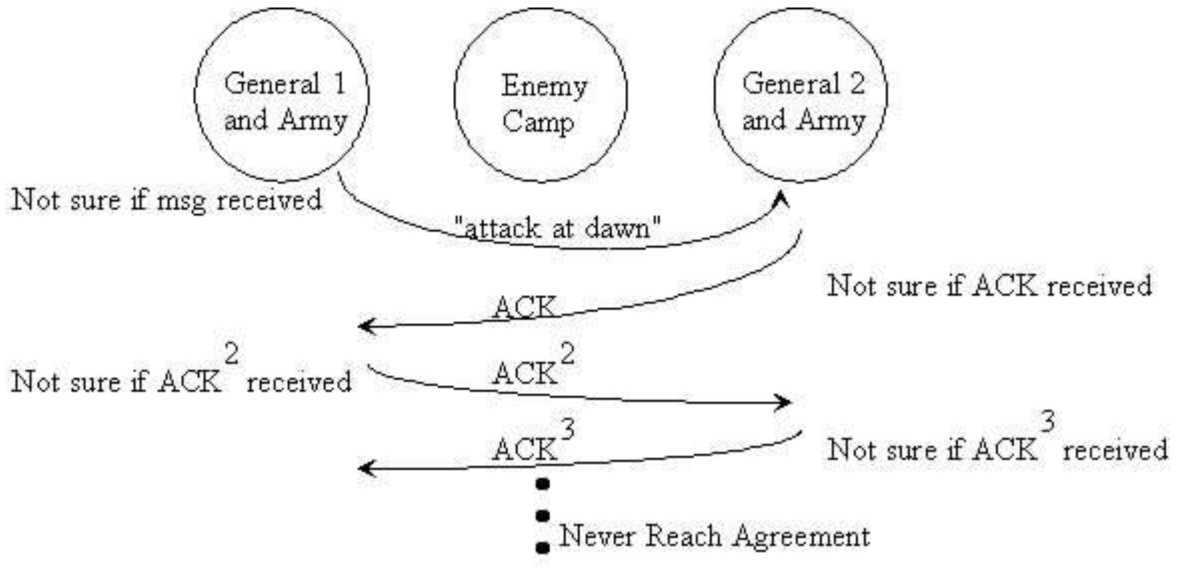
[Theorem]. If a command c is executed by some servers, all servers (eventually) execute c.

PAXOS IN PRACTICE - ASSUMPTIONS AND REAL WORLD USE

- Nodes do not try to subvert the protocol (messages delivered without corruption)
- Only works for fail-stop (no Byzantine failures) faults
- Failed nodes at any point won't affect the entire network
- Good performance (fast)
- Generally used to replicate large sets of data

Two Generals Problems

Since the possibility of the message not getting through is always > 0, the generals can never reach an agreement with 100% confidence.



The Two Generals Problem has been proven to be unsolvable

CONSENSUS Definition

[Consensus]: There are n nodes, of which at most f might crash, i.e., at least n-f nodes are correct. Node i starts with an input value v_i. The nodes must decide for one of those values, satisfying the following properties:

- ❑ Agreement: All correct nodes decide for the same value.
- ❑ Termination: All correct nodes terminate in finite time.
- ❑ Validity: The decision value must be the input value of a node.

Note: Paxos does not guarantee termination. For example, the system can be stuck forever if two clients continuously request tickets, and neither of them manages to acquire a majority

Byzantine Agreement

FAULT CATEGORIES – Failure Models

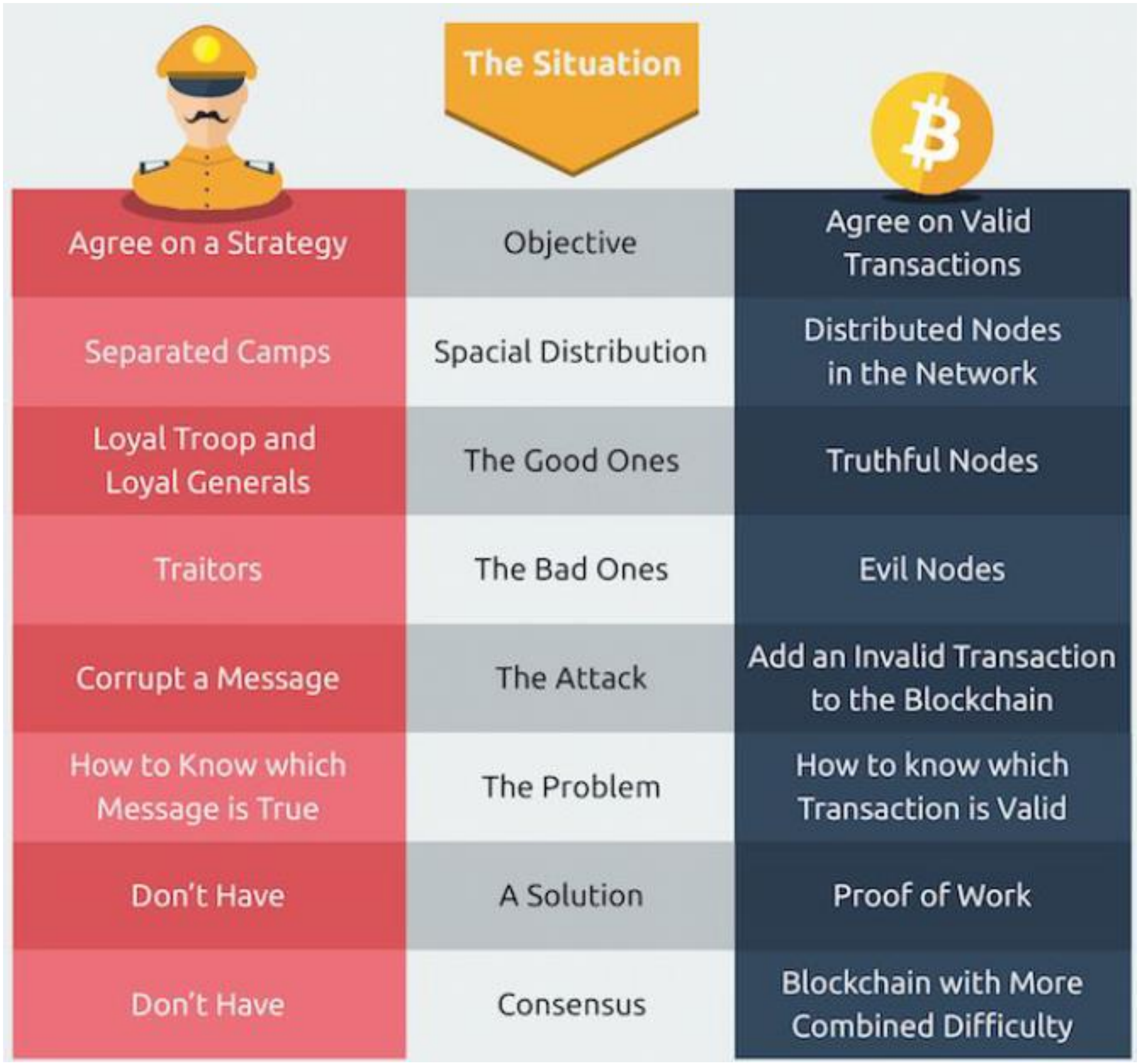
Assumptions of fault tolerant systems vs Byzantine fault tolerant systems:

- **Fail-stop fault:** Nodes can crash, not return values, crash detectable by other nodes

- **Byzantine fault:** Nodes can do all of the above *and* send incorrect/corrupted values, corruption or manipulation harder to detect

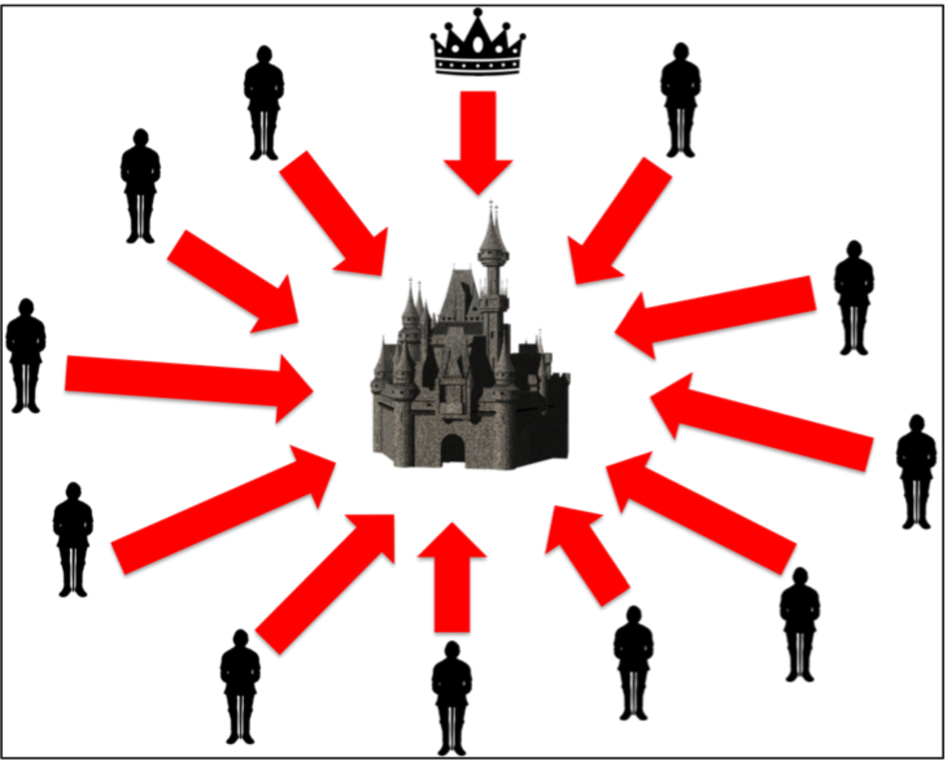
PBFT Problem: Paper released in 1999 by Miguel Castro and Barbara Liskov, for high efficiency (thousands of operations per second)

The Generals and the Bitcoin

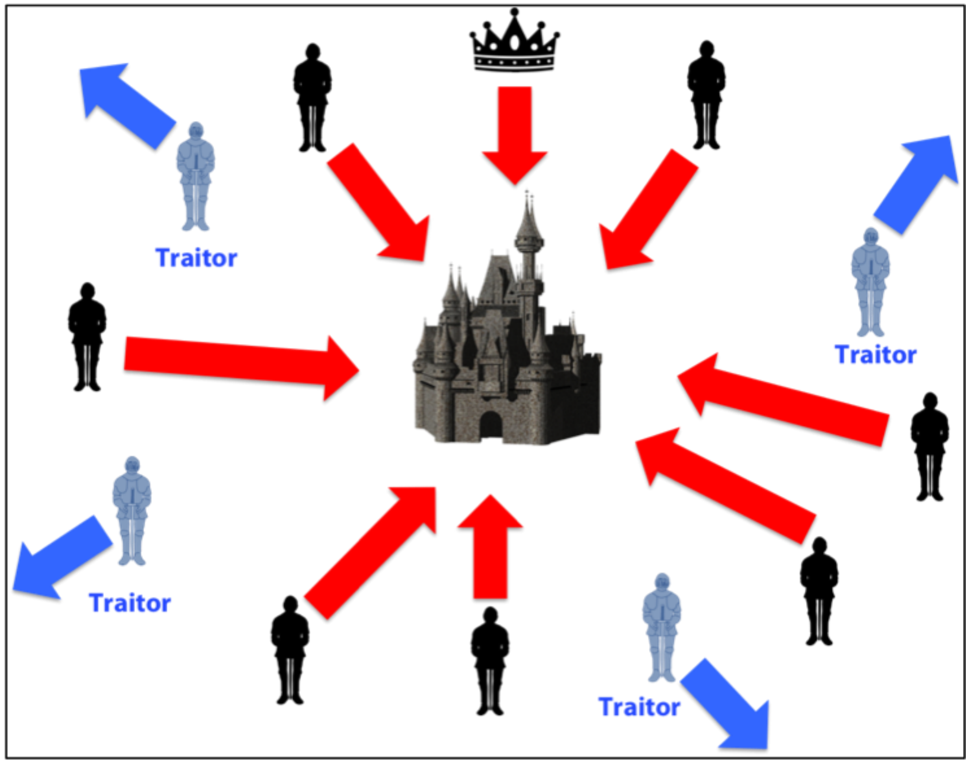


	The Situation	
Agree on a Strategy	Objective	Agree on Valid Transactions
Separated Camps	Spacial Distribution	Distributed Nodes in the Network
Loyal Troop and Loyal Generals	The Good Ones	Truthful Nodes
Traitors	The Bad Ones	Evil Nodes
Corrupt a Message	The Attack	Add an Invalid Transaction to the Blockchain
How to Know which Message is True	The Problem	How to know which Transaction is Valid
Don't Have	A Solution	Proof of Work
Don't Have	Consensus	Blockchain with More Combined Difficulty

Multiple Generals Problem

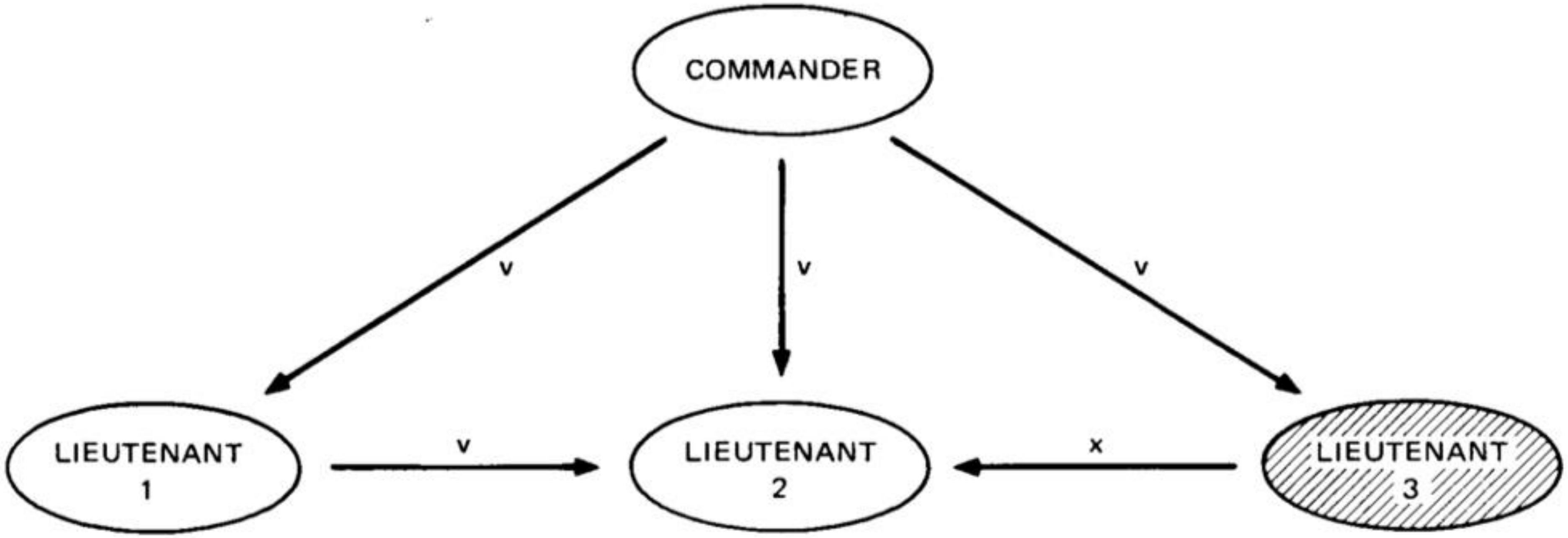


Coordinated Attack Leading to Victory



Uncoordinated Attack Leading to Defeat

Multiple Generals Problem



Theorem: For any *m*, Algorithm *OM(m)* reaches consensus if there are more than *3m* generals and at most *m* traitors

BFT MECHANICS

- Impossible to guarantee safety and liveness if ⅓ or more of generals are traitors
- Solution to all remaining cases: Practical Byzantine Fault Tolerance

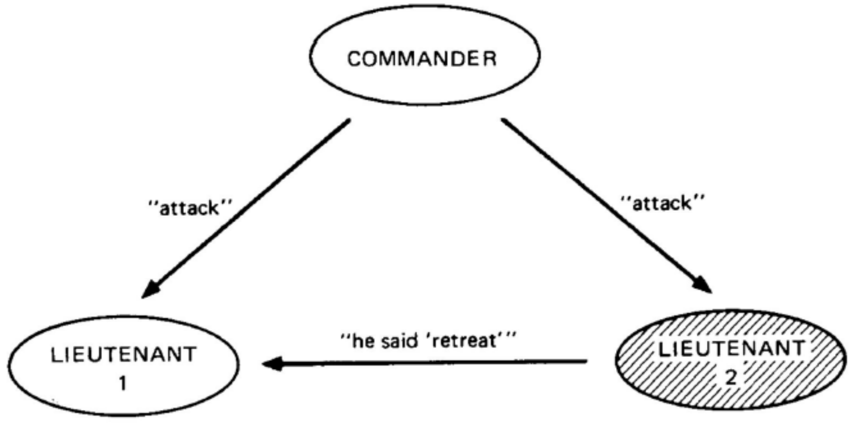


Fig. 1. Lieutenant 2 a traitor.

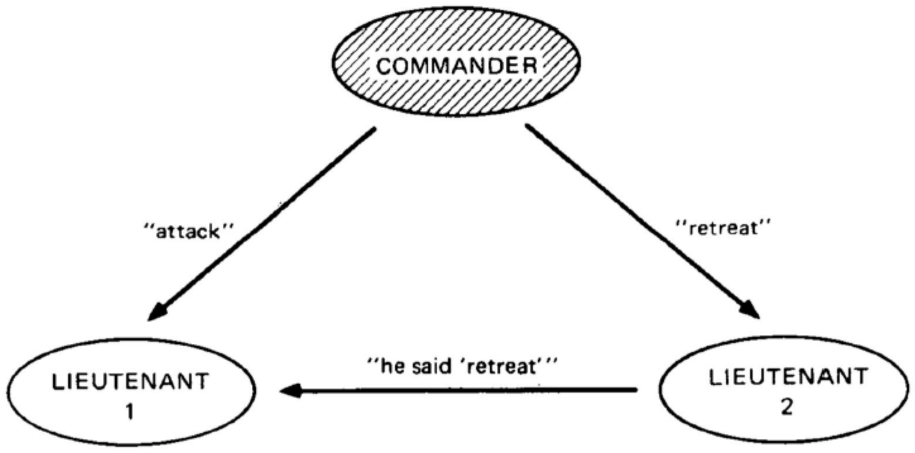


Fig. 2. The commander a traitor.

"The Byzantine Generals Problem" – Leslie Lamport, Robert Shostak, Marshall Pease, SRI International, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982

Compare with Nakamoto Consensus in PoW

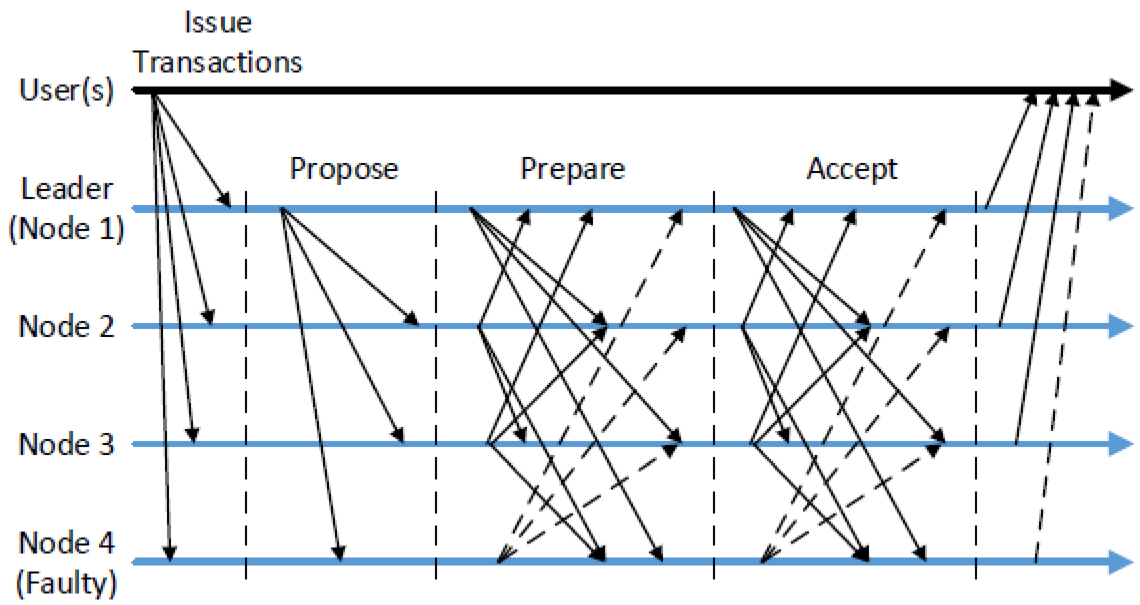
- PoW assumes majority of computing power by honest players
  - These honest players will execute the protocols correctly
  - Weak synchronous – eventual consensus takes a long time
- PoW not suitable for Enterprise Blockchain Apps which need
  - Faster, cheaper settlement
  - Greater scalability, performance
  - Might need identity verification

PRACTICAL BYZANTINE FAULT TOLERANCE

Practical Byzantine Fault Tolerance (PBFT)

- **ASSUMPTION**
  - Medium of communication is reliable and all clients will respond
- **LIVENESS**
  - Clients will receive a reply to every request sent provided the network is functioning
- **SAFETY**
  - The replies received by the clients are correct by vector linearization
- **Useful in File Servers or Automated Control Systems**
- PBFT employ replication to achieve resilience against failures, not scalability - basic protocol requires O(n^2) messages – while PoW follows longest-chain-rule to verify (no all-to-all message, so O(n) message complexity)

PBFT-based Message Pattern



- ❑ Fully connected topology among the consensus nodes
- ❑ Leader-peer hierarchy – leader proposes blocks to peer nodes
- ❑ All-to-all peer messaging phases - proposal accept if majority
- ❑ Three-way handshakes for synchronization require bounded execution time and message latency
- ❑ Guarantee deterministic agreement and liveness with low latency in Byzantine environment

PBFT EXAMPLE

Consider four replicas trying to agree on the value of a single bit (attack/don't attack)

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1			
Replica 2		1		
Replica 3			1	
Replica 4				0

PBFT EXAMPLE

All replicas send their value to the other replicas

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1	1	1	0
Replica 2	1	1	1	0
Replica 3	1	1	1	0
Replica 4	1	1	1	0

PBFT EXAMPLE

- All Replicas send their vectors to all other Replicas
- For example: Replica 2 sends <Replica 2, Replica 3, Replica 4> values to Replica 1

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1	<1,1,0>	<1,1,0>	<0,0,0>
Replica 2	<1,1,0>	1	<1,1,0>	<0,0,0>
Replica 3	<1,1,0>	<1,1,0>	1	<0,0,0>
Replica 4	<1,1,1>	<1,1,1>	<1,1,1>	0

PBFT EXAMPLE

- Most frequent value in the vectors is the agreed upon value

	Replica 1	Replica 2	Replica 3	Replica 4
Replica 1	1	1	1	0
Replica 2	1	1	1	0
Replica 3	1	1	1	0
Replica 4	1	1	1	0

- **Question: in this example we had 4 replicas, one of which was faulty. Would this work with 3 replicas, one of which was faulty?**

Asynchronous Consensus

Asynchronous Model

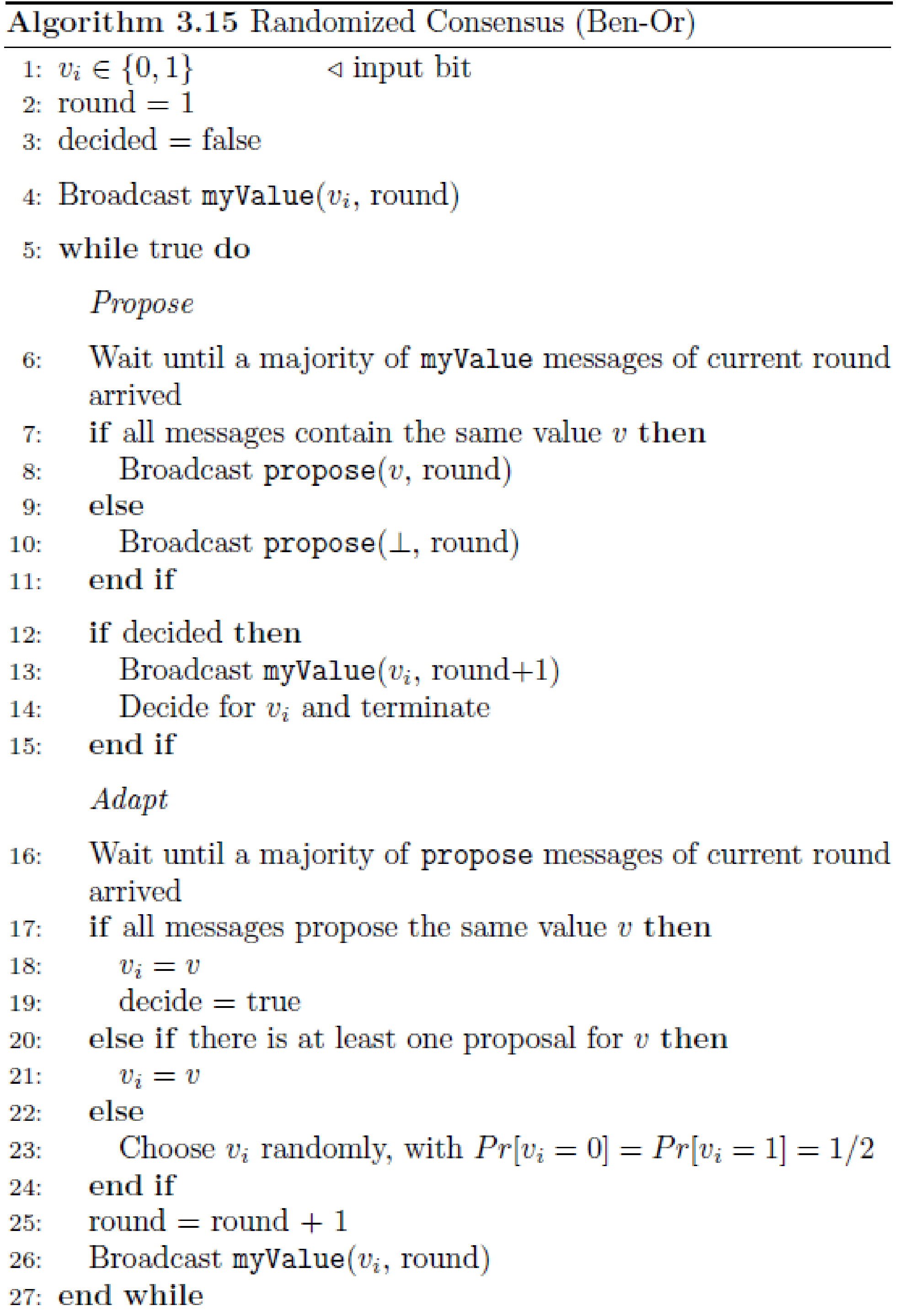
Asynchronous Model, In the asynchronous model, algorithms are event based (“upon receiving message . . . , do . . . "). Nodes do not have access to a synchronized wall-clock. A message sent from one node to another will arrive in a finite but unbounded time.

[**Theorem**] There is no deterministic algorithm which always achieves consensus in the asynchronous model, with $f > 0$.

How can the situation be improved? For example by giving each node access to randomness, i.e., we allow each node to toss a coin.

Randomized Consensus

Randomized Binary Consensus: either all nodes start with the same input bit, or, nodes toss a coin until a large number of nodes get “by chance” the same outcome.



Algorithm 3.15 Randomized Consensus (Ben-Or)

```
$v_i \in \{0,1\}$            ◁ input bit
round = 1
decided = false
Broadcast myValue($v_i$, round)
while true do
    Propose
    Wait until a majority of myValue messages of current round arrived
    if all messages contain the same value $v$ then
        Broadcast propose($v$, round)
    else
        Broadcast propose($\perp$, round)
    end if
    if decided then
        Broadcast myValue($v_i$, round+1)
        Decide for $v_i$ and terminate
    end if
    Adapt
    Wait until a majority of propose messages of current round arrived
    if all messages propose the same value $v$ then
        $v_i = v$
        decide = true
    else if there is at least one proposal for $v$ then
        $v_i = v$
    else
        Choose $v_i$ randomly, with $Pr[v_i = 0] = Pr[v_i = 1] = 1/2$
    end if
    round = round + 1
    Broadcast myValue($v_i$, round)
end while
```

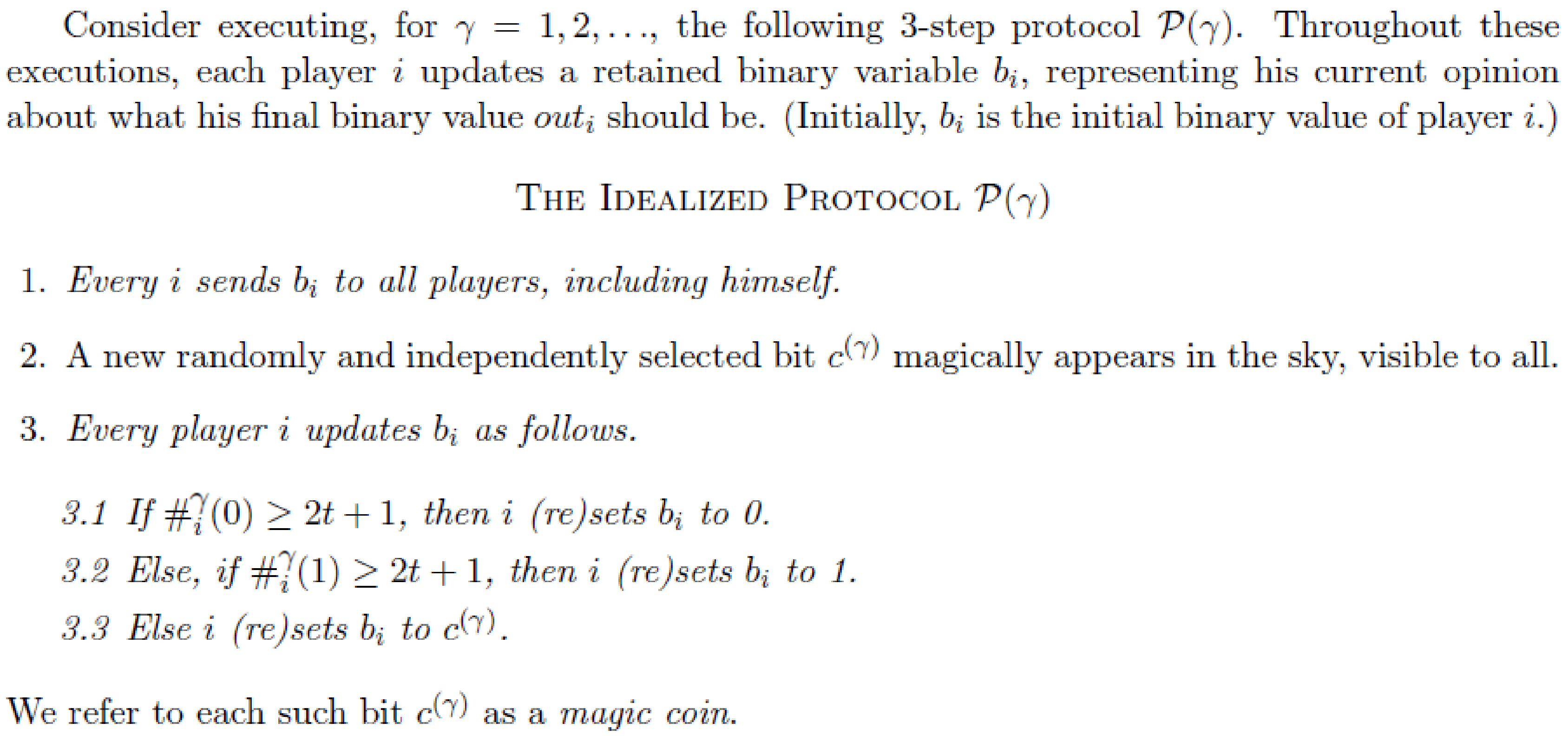
Randomized Consensus

[**Theorem**]. Algorithm 3.15 achieves binary consensus with expected runtime O(2^n) if up to f < n/2 nodes crash.

[**Theorem**]. There is no consensus algorithm for the asynchronous model that tolerates f >= n/2 many failures.

- Algorithm 3.15 solves consensus with optimal fault-tolerance, but it is awfully slow. The problem is rooted in the individual coin tossing.
- It can be improved by tossing a shared coin, which is a random variable that is 0 for all nodes with constant probability, and 1 with constant probability

Shared Magic Coin



Consider executing, for $\gamma = 1, 2, \ldots$, the following 3-step protocol $\mathcal{P}(\gamma)$. Throughout these executions, each player i updates a retained binary variable b_i, representing his current opinion about what his final binary value out_i should be. (Initially, b_i is the initial binary value of player i.)

THE IDEALIZED PROTOCOL $\mathcal{P}(\gamma)$

1. *Every i sends b_i to all players, including himself.*
2. A new randomly and independently selected bit $c^{(\gamma)}$ magically appears in the sky, visible to all.
3. *Every player i updates b_i as follows.*

   3.1 If $\#_i^\gamma(0) \geq 2t+1$, then i (re)sets b_i to 0.
   3.2 Else, if $\#_i^\gamma(1) \geq 2t+1$, then i (re)sets b_i to 1.
   3.3 Else i (re)sets b_i to $c^{(\gamma)}$.

We refer to each such bit $c^{(\gamma)}$ as a *magic coin*.

Theory: Above algorithm will converge with probability 1.

Concrete Coin

The players themselves, without any help from the sky, can by themselves (with digital signatures, hash function H, and random string R), implement a magic coin in a single step.

We identify each player i with his public key. We denote i's digital signature of a string x by $sig_i(x)$. To ensure signers and messages are retrievable from digital signatures, we define

$$SIG_i(x) \triangleq (i, x, sig_i(x)).$$

Single-Step Protocol *ConcreteCoin(γ)*

Each player i

- *Sends the value* $v_i \triangleq SIG_i(R, \gamma)$.
- *Computes the first player, m, such that* $H(v_m) \leq H(v_j)$ *for all players j.*
- *Sets* $c_i^{(\gamma)}$ *to be the least significant bit of* $H(v_m)$.

FEDERATED CONSENSUS

FEDERATED CONSENSUS

In a distributed system, a quorum is a set of nodes sufficient to reach agreement.

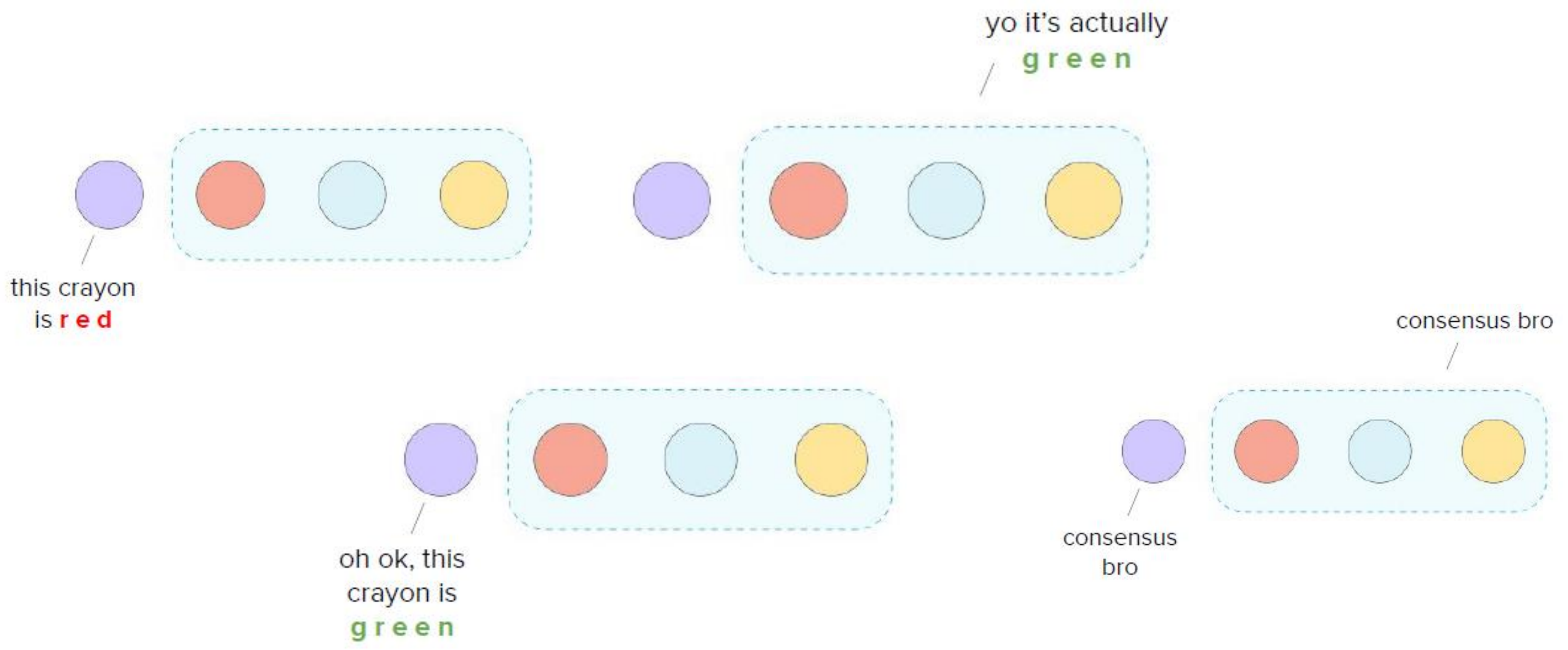
What if you don't necessarily trust certain nodes in the quorum? How can the we still achieve consensus?

FEDERATED BYZANTINE AGREEMENT

Problem: How do we determine quorums in a decentralized way?

Solution: introduce quorum slices

- ❑ Subset of a quorum that can convince one particular node of agreement
- ❑ Individual nodes decide on other participants they trust for information
  - ❖ Example: a particular bank might be reputable
  - ❖ Example: pre-established financial relationships, make sure all parties sign off

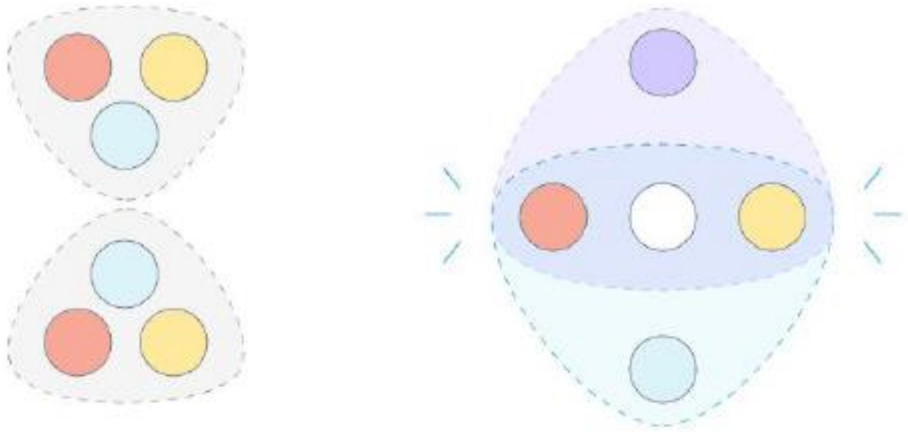


FEDERATED BYZANTINE AGREEMENT

Idea: What happens when multiple quorum slices join together?

We get a **quorum intersection**

- Quorum slices that come together will convince other quorums slices and form a “larger” quorum
- Otherwise **disjoint quorums** agree on different things
  - independent agreement
  - potential for contradictory statements



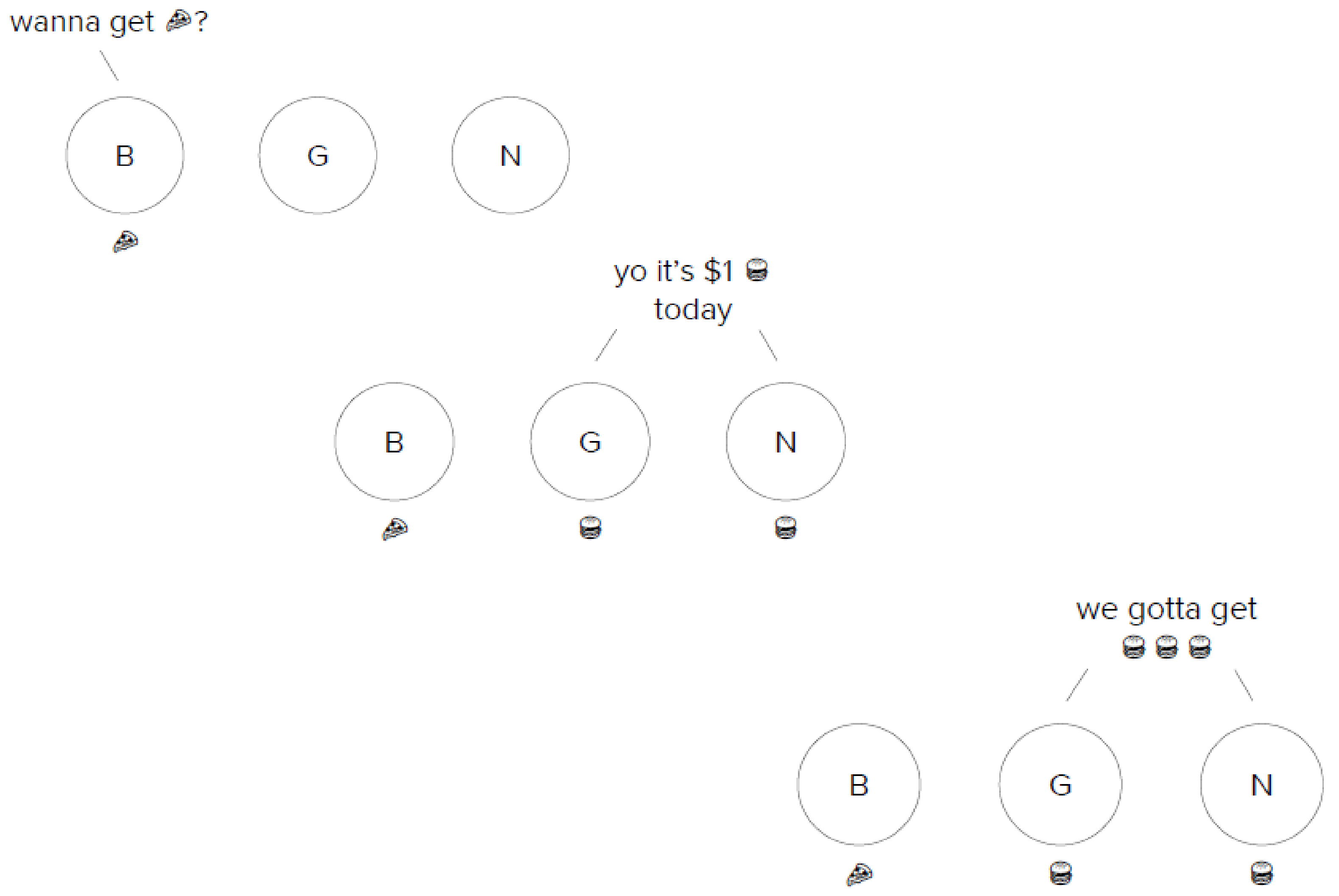
FEDERATED BYZANTINE AGREEMENT

Each node responsible for choosing own quorum slice, ensure:

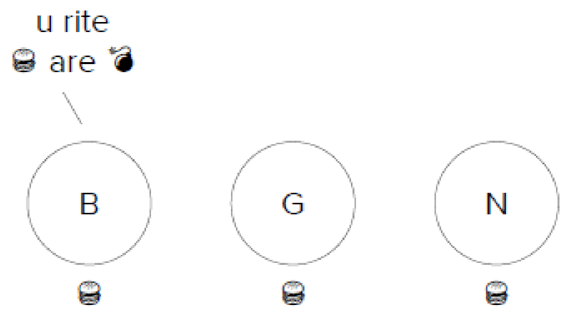
- ❑ Slices large enough
- ❑ Nodes in slices want to maintain reputation

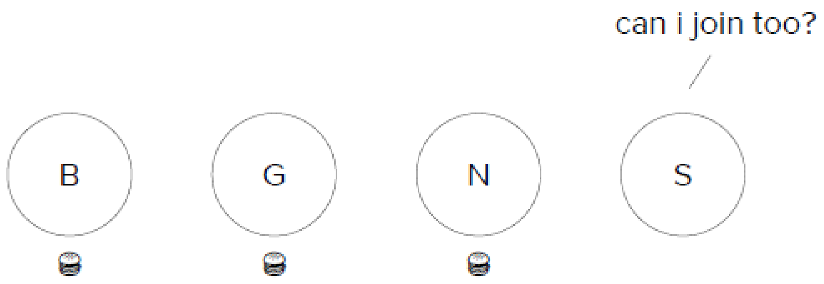
- ❑ **Decentralized control**: no central authority that authorizes consensus
- ❑ **Low latency**: consensus achieved in a few seconds
- ❑ **Flexible trust**: nodes choose who they trust, don’t have to trust the entire network

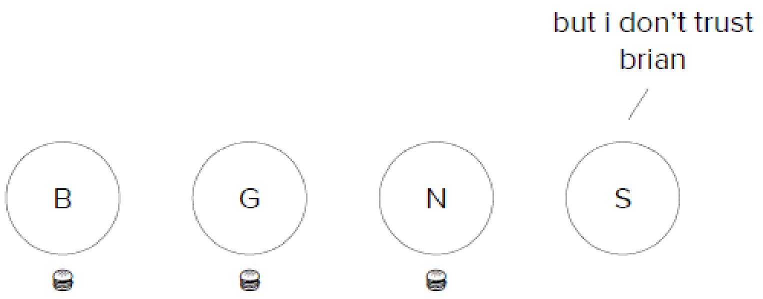
Lunchtime Consensus!



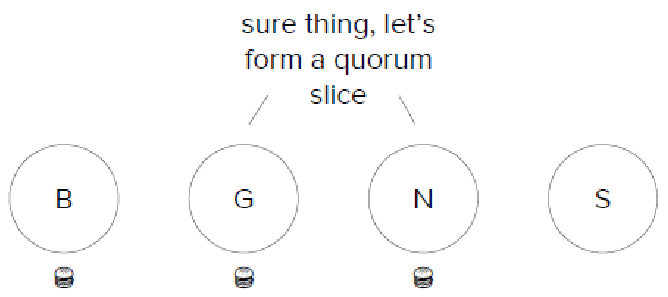
Lunchtime Consensus!

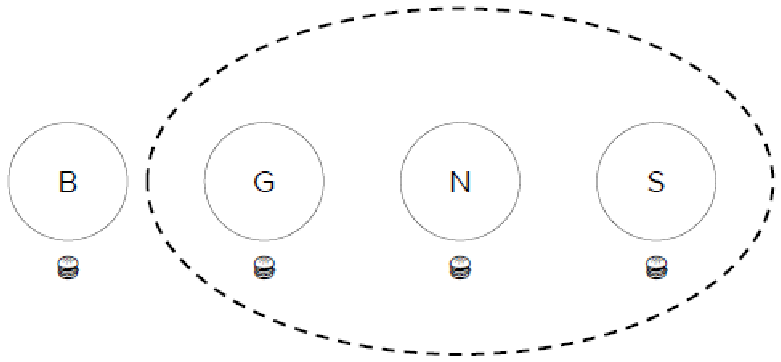


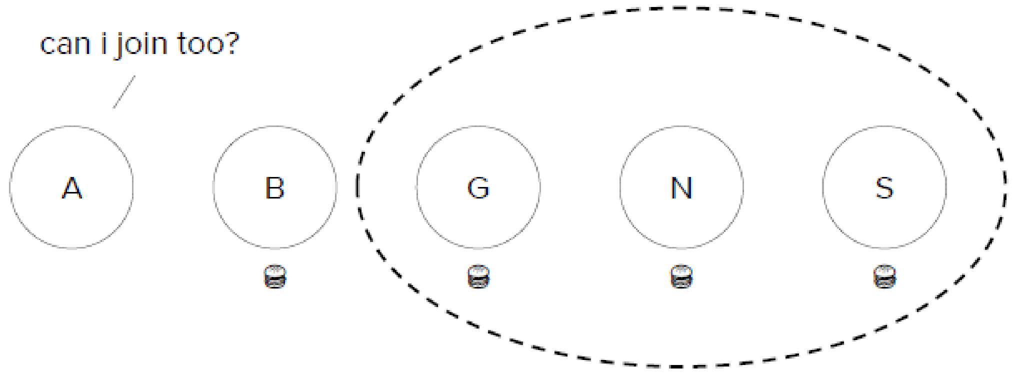




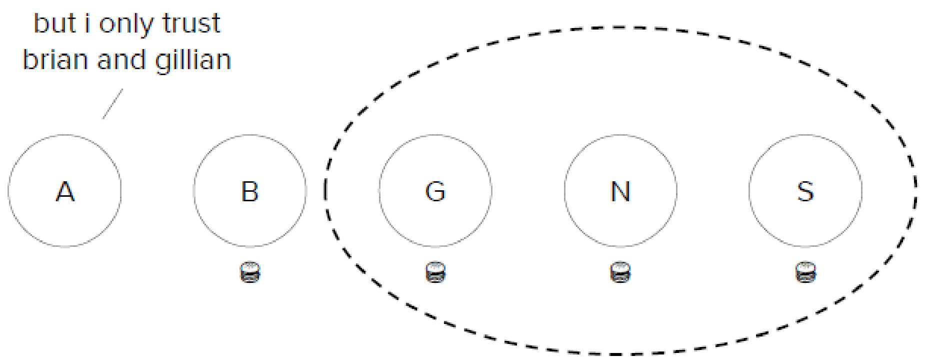
Lunchtime Consensus!

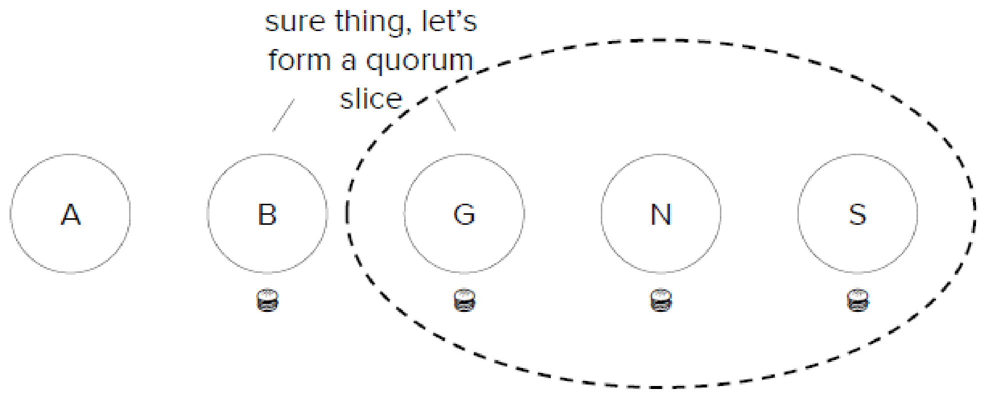


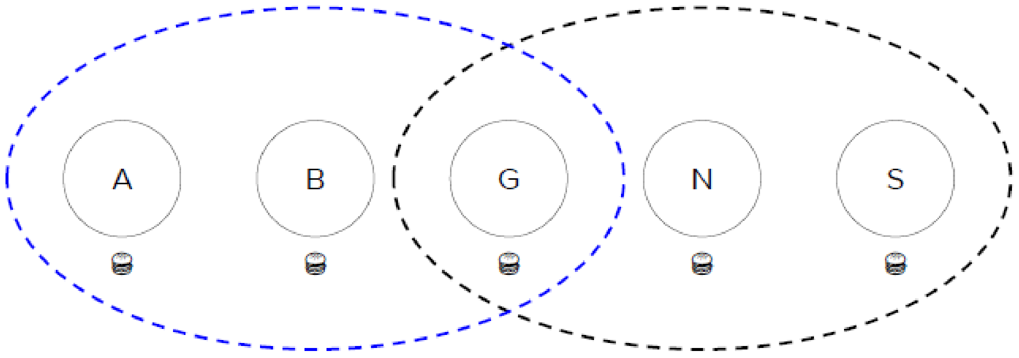




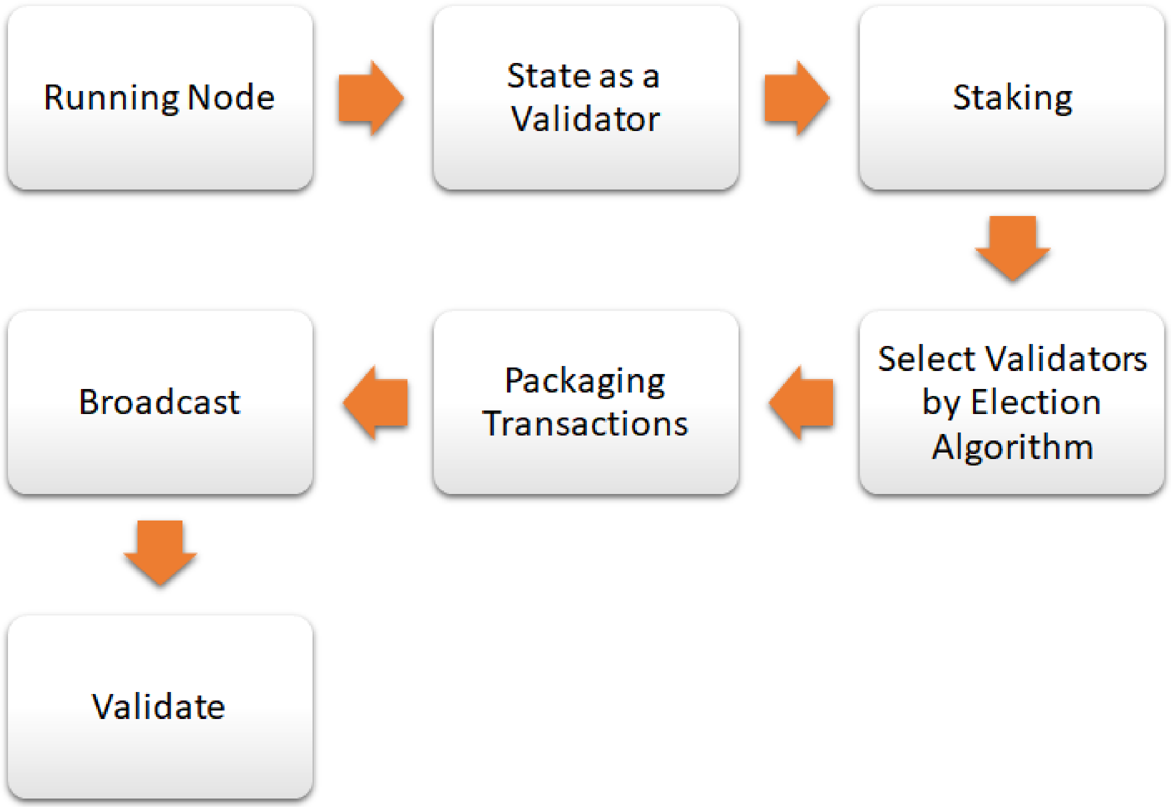
Lunchtime Consensus!







PoS Consensus



Consensus Algorithms in PoS

Proof of Stake (PoS) – Unbiased Leader Election

- ❑ **Ethereum's Casper**
- ❑ **Delegated PoS – BitShares, EOS, Steem**
- ❑ **Algorand - MIT**
- ❑ **Cardano (based on Ouroboros) – U. of Edinburgh**
- ❑ **Dfinity – Switzerland**
- ❑ **Snow White (based on Sleepy Model) - Cornell**

Challenges of PoS Consensus

Elect "unbiased" block proposers or committee:

- **unpredictable and safe against any adversarial attacks**
- **verifiable to everybody**

Possible attacks on PoS consensus models:

- ❑ **Grinding attack –** *influence the slot leader selection process (e.g. randomness dependent on signature of previous block) using computing power*
- ❑ **Desynchronization attack –** *ill-timed issuing of block and offline*
- ❑ **Transaction denial attack –** *censorship attack, w/o liveness*
- ❑ **Eclipse attack -** *subversion in p2p message delivery mechanism*
- ❑ **Bribery attack**
- ❑ **Nothing at Stake -** *simultaneously mining multiple blockchains since little computational effort is dedicated to build a PoS blockchain*
- ❑ **Long Range Attack -** *computing a longer valid chain that starts right after the genesis block*
- ❑ **Network splitting (–> CAP theorem)**

PoS Consensus – Slot Lead Election - Algorand

Algorand:

- **Anyone can join lottery to win the ticket as block proposer or validating committee**
- **Everyone runs own "lottery machine" with public random seed and his private key**
- **Lottery machines = verifiable random functions (VRF) which produces uniformly distributed random values, and a *non-interactive* proof (others can verify using the public key)**

Given an input value *x*, the owner of the secret key SK can compute the function value $y = F_{SK}(x)$ and the proof $p_{SK}(x)$. Using the proof and the public key $PK = g^{SK}$, everyone can check that the value $y = F_{SK}(x)$ was indeed computed correctly, yet this information cannot be used to find the secret key.

$$F_{SK}(x) = e(g,g)^{1/(x+SK)} \quad \text{and} \quad p_{SK}(x) = g^{1/(x+SK)},$$

where $e(\cdot,\cdot)$ is a bilinear map. To verify whether $F_{SK}(x)$ was computed correctly or not, one can check if $e(g^x PK, p_{SK}(x)) = e(g,g)$ and $e(g, p_{SK}(x)) = F_{SK}(x)$.

- **Cryptographic sortition to choose a random subset of users according to per-user weights** w_i/W

PoS Consensus – Election - Algorand

Algorand:

- Cryptographic sortition to choose a random subset of users according to per-user weights - given a set of weights wi and the weight of all users W = SUM(wi) , the probability that user i is selected is proportional to wi /W – to withstand Sybil attacks

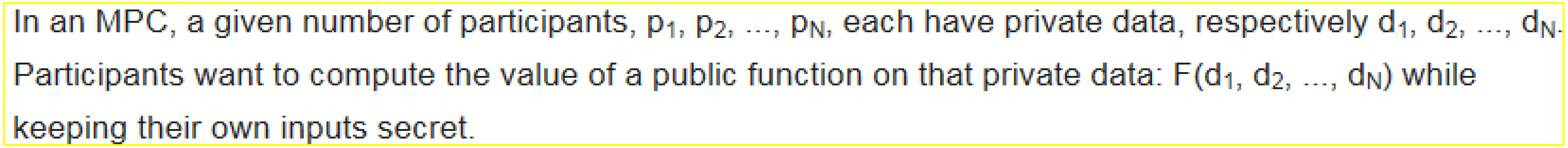
```
procedure Sortition($sk, seed, \tau, role, w, W$):
$\langle hash, \pi \rangle \leftarrow \text{VRF}_{sk}(seed || role)$
$p \leftarrow \frac{\tau}{W}$
$j \leftarrow 0$
while $\frac{hash}{2^{hashlen}} \notin \left[ \sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p) \right)$ do
    $j$++
return $\langle hash, \pi, j \rangle$
```

- **VRF_sk(x) returns two values: a hash and a proof. The hash is a hashlen-bit-long value that is uniquely determined by sk and x. The proof π enables anyone that knows pk to check the hash indeed corresponds to x.**
- **"role" is "proposer or acceptor", a threhold "τ" is expected number of users selected for that role**
- **Pseudo-random value "hash" determines how many sub-users (# times of a user) are selected according to its weight**
- **The probability that exactly k out of the w (the user's weight) sub-users are selected follows the binomial distribution** $B(k; w, p) = \binom{w}{k} p^k (1-p)^{w-k}$, where $\sum_{k=0}^{w} B(k; w, p) = 1$.

PoS Consensus – Election - Cardano

Cardano:

- Multiparty Computation (MPC) to obtain random seed - each one independently performs “coin tossing” and shares with others - eventually they agree on the same final value
- Commitment / Reveal / Recovery phase -> all get the same seed
- Follow-the-Satoshi algorithm (FTS) verifiably picks a coin, and the coin owner becomes a slot leader to propose the block



In an MPC, a given number of participants, $p_1, p_2, \ldots, p_N$, each have private data, respectively $d_1, d_2, \ldots, d_N$. Participants want to compute the value of a public function on that private data: $F(d_1, d_2, \ldots, d_N)$ while keeping their own inputs secret.

- Coin Tossing Game
- n users toss coin to decide each round the bit value - Binary Byzantine Agreement (BBA)

PoS Consensus – Election (Cont’d)

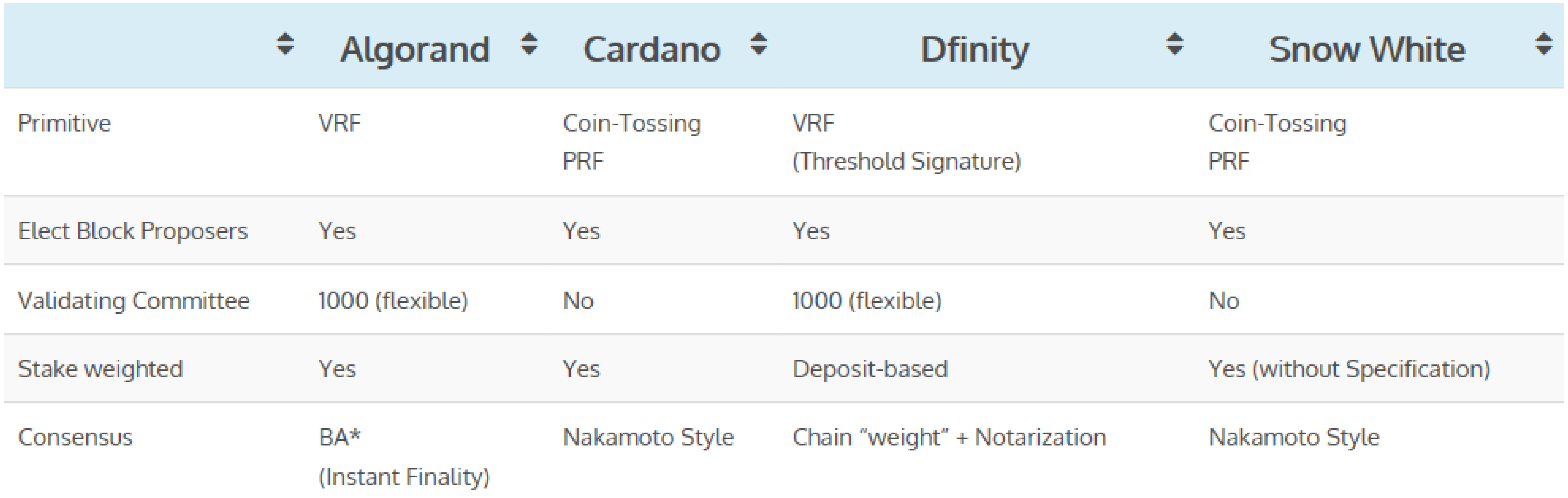
Dfinity:

- **Threshold Signatures - n parties set up a public key (group public key) and each party retains an individual secret (secret key share). t out of n parties are sufficient for creating a signature (group signature) that validates group public key**
- **Signature schemes with unique, non-interactive threshold version is the pairing-based schemes from Boneh of Stanford (2003)**
- **Each one sign and broadcast to peers its signature. Others verify and rebroadcast. Once t correct signatures received, unique group signature can be created.**
- **The random number is H(group signature)**
- **Election algorithm uses a pseudo-random permutation on all registered users, and ranks all block proposals through the given random seed**

Snow White:

- **Similar to Algorand, all users pick a private and a public random seed to fuel their lottery machines.**
- **Users evaluate a pseudo random function (PRF) using their secret string and a public random string inferred from a common reference string (CRS).**
- **If the random string produced by user u is under some threshold, then this user is elected**

PoS Consensus – Comparison



	Algorand	Cardano	Dfinity	Snow White
Primitive	VRF	Coin-Tossing PRF	VRF (Threshold Signature)	Coin-Tossing PRF
Elect Block Proposers	Yes	Yes	Yes	Yes
Validating Committee	1000 (flexible)	No	1000 (flexible)	No
Stake weighted	Yes	Yes	Deposit-based	Yes (without Specification)
Consensus	BA* (Instant Finality)	Nakamoto Style	Chain “weight” + Notarization	Nakamoto Style