Permissioned Blockchain
&
Consensus Algorithms - I
(PAXOS)

Permissioned Model

- A blockchain architecture where users are authenticated apriory
- Users know each other
- However, users may not trust each other – Security and consensus are still required.
- Run blockchain among known and identified participants

Permissioned Model

- Moving towards Blockchain 2.0 (smart contracts)
- Support closed network
- Users are authenticated a priory
- Users know each other
- Users may not trust each other
- Security and Consensus are still required
- Run blockchain among known and identified participants
- No mining

Need of permissioned blockchain

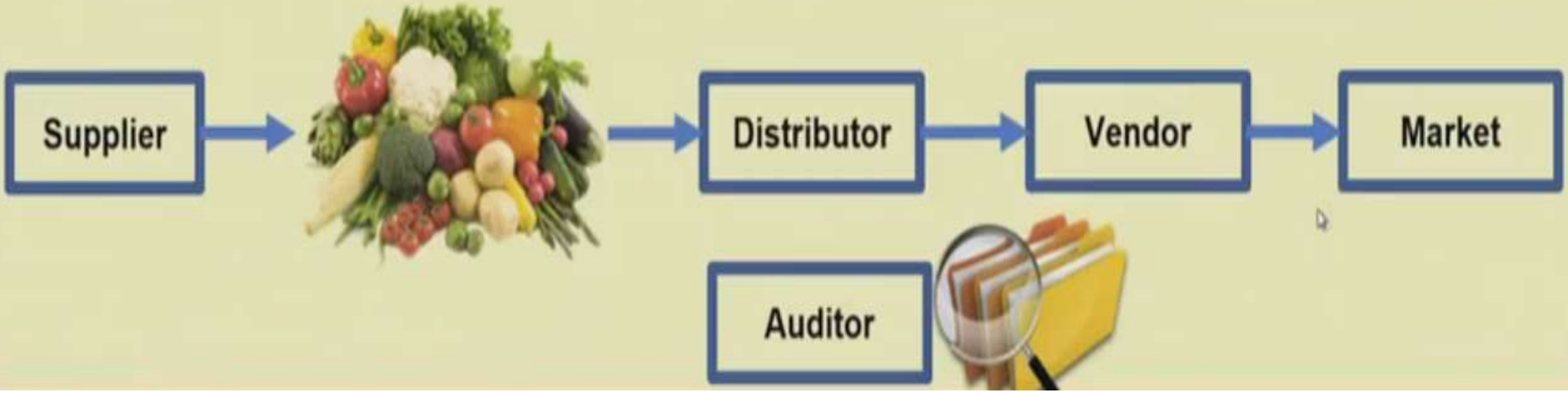
- Unlike courier tracking → central server/database
  - Updates information of article movement
  - Applicable if only one courier company in transition
- If article moving multiple companies:
  - Bluedart → DHL → DTDC

- International post tracking system (say, India to US)
- Issues:
  - Who will deploy the centralize server
  - Suppose Indian post deployed the server → US postal service has to trust on Indian postal server
  - Indian post has to provide access to US postal service

Use Cases

- Particularly interesting for business applications – execute contracts among a closed set of participants

- Example: Provenance tracking of assets



Smart Contract

Smart Contract:

- Self Executing
- Defines terms and conditions of agreement
- Written in Line of codes

Suppose you want to change the control on money spending

- You friend can use money immediately
- You friend can use money 15 days later
- Some other new terms to spend money
- OR certain other conditions satisfies

Smart Contracts

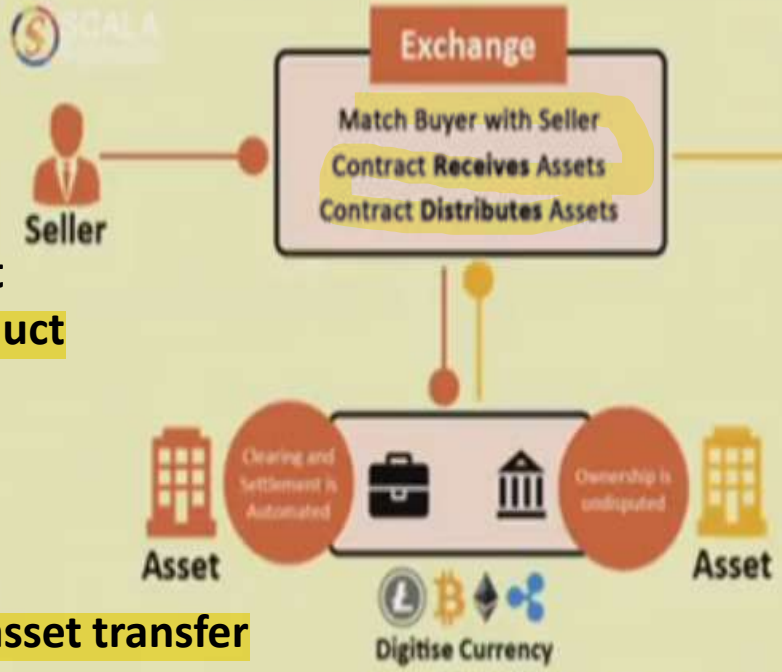
"A self executing contract in which the terms of the agreement between the buyer and the seller is directly written into the lines of code" - http://www.scalablockchain.com/

Remember the **bitcoin scripts** – you can change the script to control how the money that you are transferring to someone can be spend further

- Your friend can use that money immediately
- Your friend can use that money after 2 months

Smart Contracts Of Smart Contracts

- You can extend the script to ensure smart contract execution
  - Execute a transaction only when certain condition is satisfied



Seller wants money first before sending the product

Buyer wants product first before sending the money

Ownership of product/asset transfer

Source: http://www.scalablockchain.com/smartcontract.html

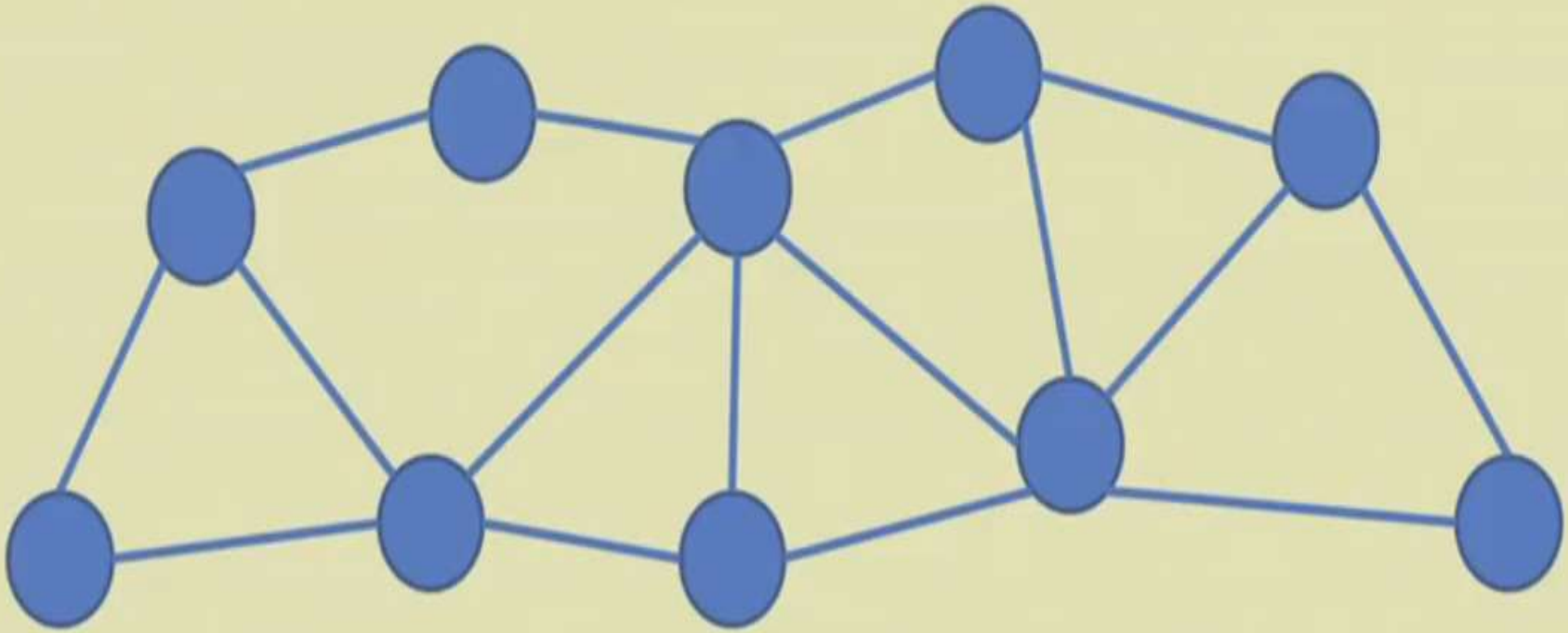
Verifies the money transfer before transferring the ownership

Design Limitations of Smart Contract

- Sequential Execution
- Non deterministic Execution
- Execution on all nodes

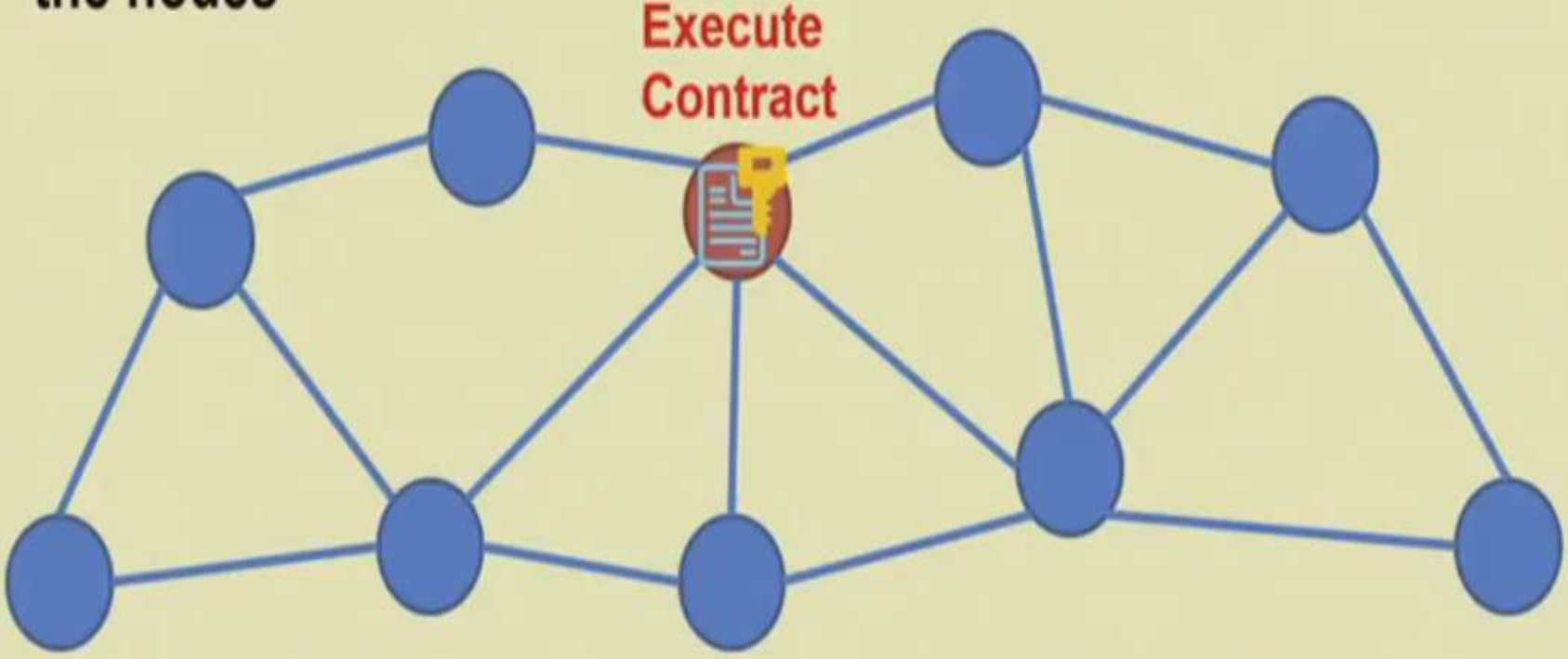
Do We Really Need to Execute Contracts at Each Node

- Not necessary always, **we just need state synchronization across all the nodes**



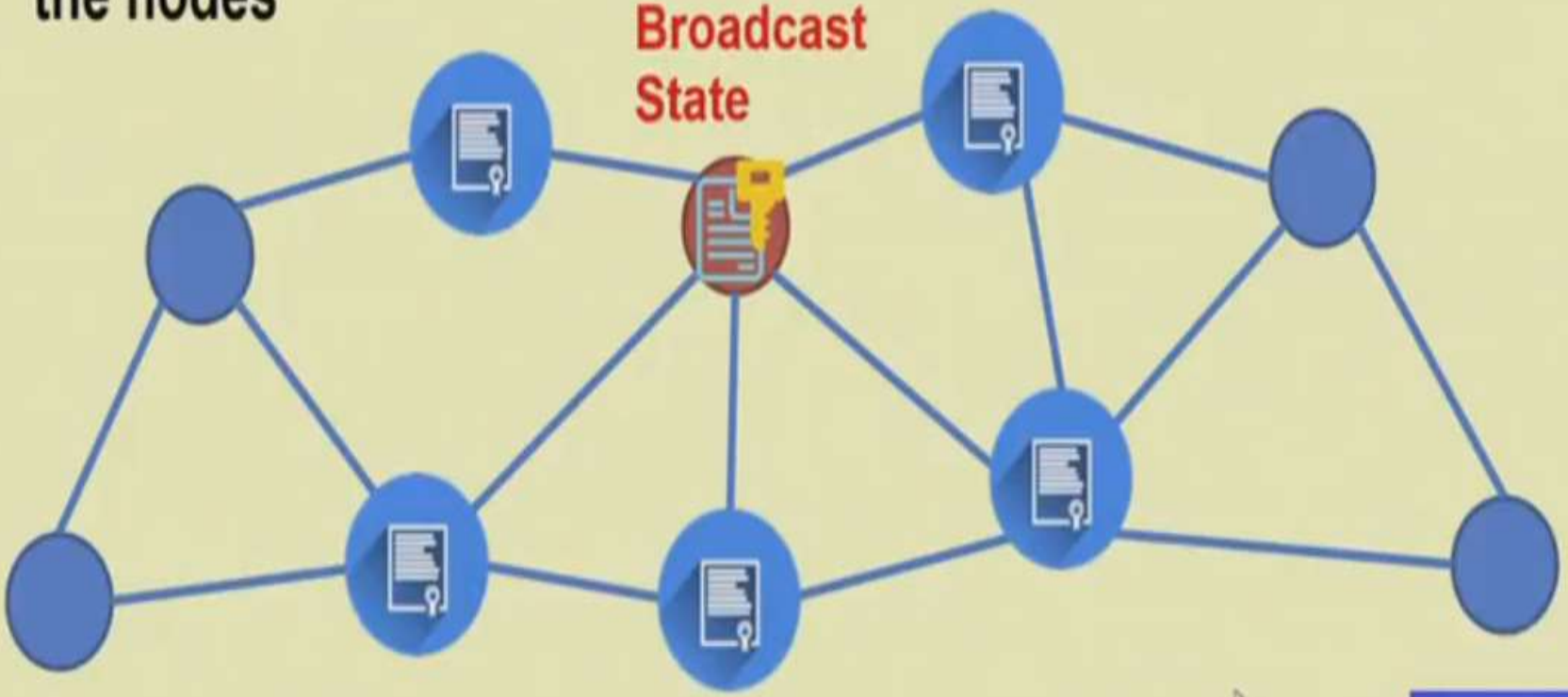
Do We Really Need to Execute Contracts at Each Node

- Not necessary always, **we just need state synchronization across all the nodes**



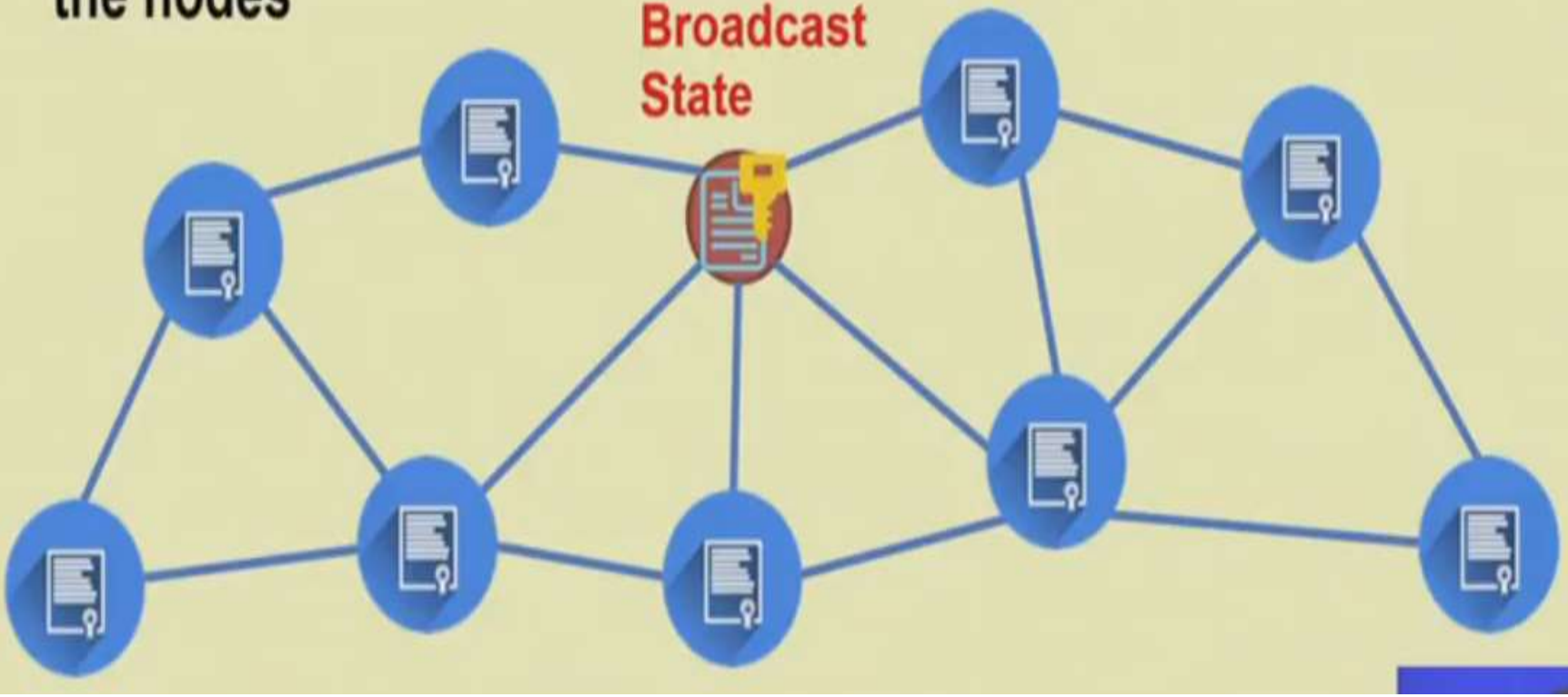
Do We Really Need to Execute Contracts at Each Node

- Not necessary always, **we just need state synchronization across all the nodes**



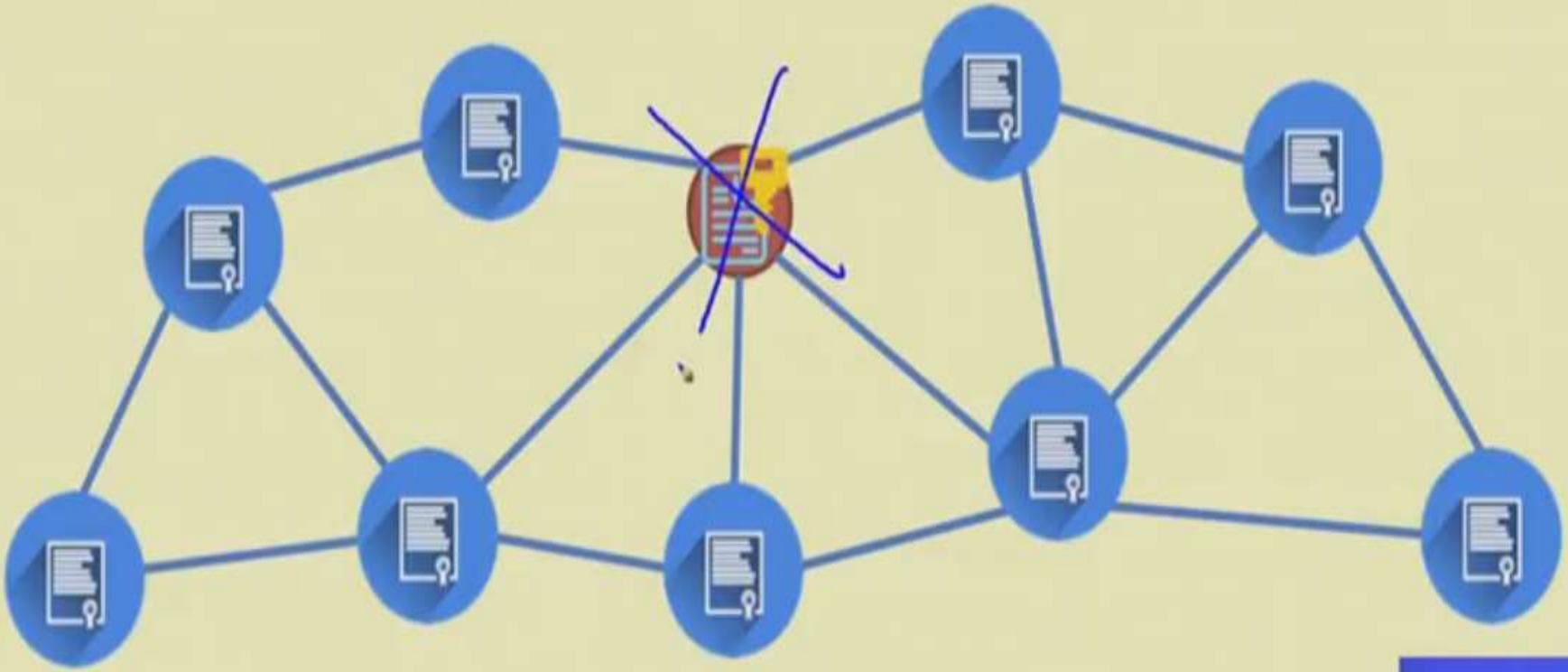
Do We Really Need to Execute Contracts at Each Node

- Not necessary always, **we just need state synchronization across all the nodes**



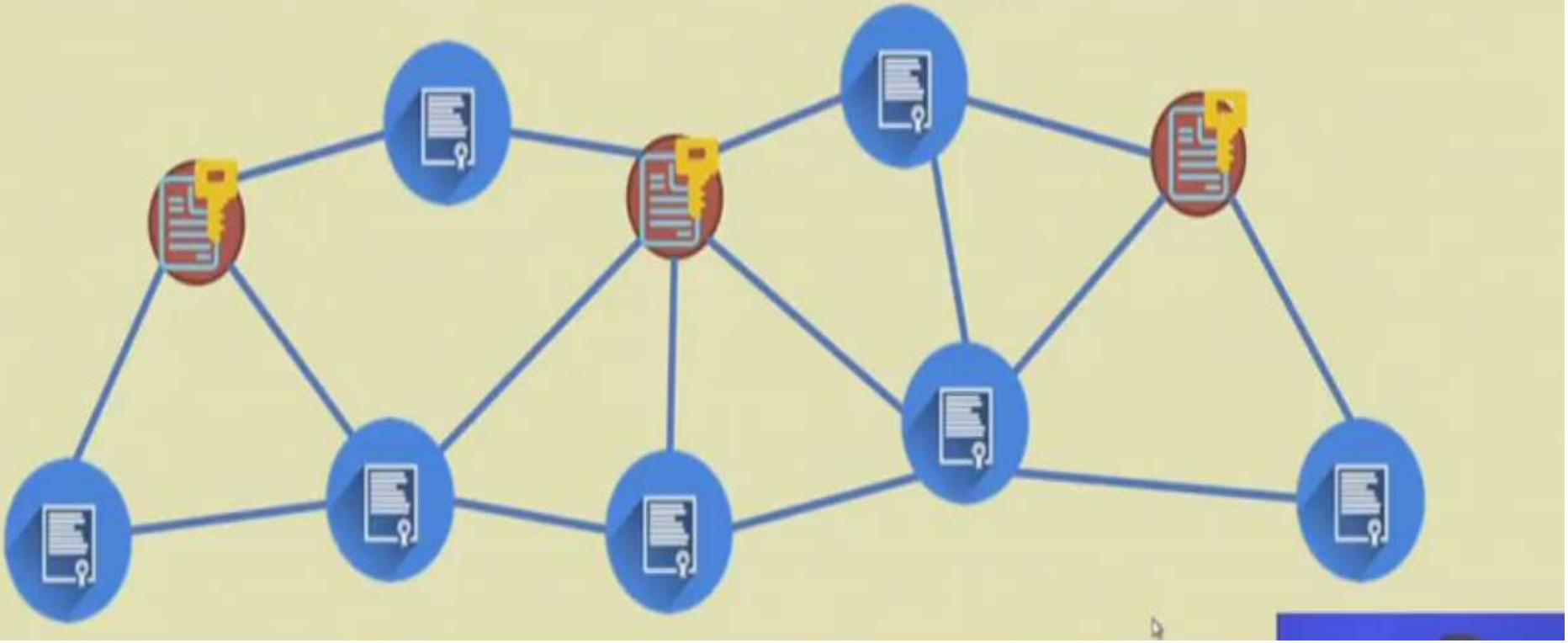
Do We Really Need to Execute Contracts at Each Node

- What if the node that executes the contract is faulty?



Do We Really Need to Execute Contracts at Each Node

- **Use state machine replication** – execute contract at a subset of nodes, and ensure that the same state is propagated to all the nodes



State Machine Replication

State machine

- A set of states (S) based on the system design
- A set of inputs (I)
- A set o outputs (O)
- A transition function $S \times I \rightarrow S$
- A output function $S \times I \rightarrow O$
- A *start* state

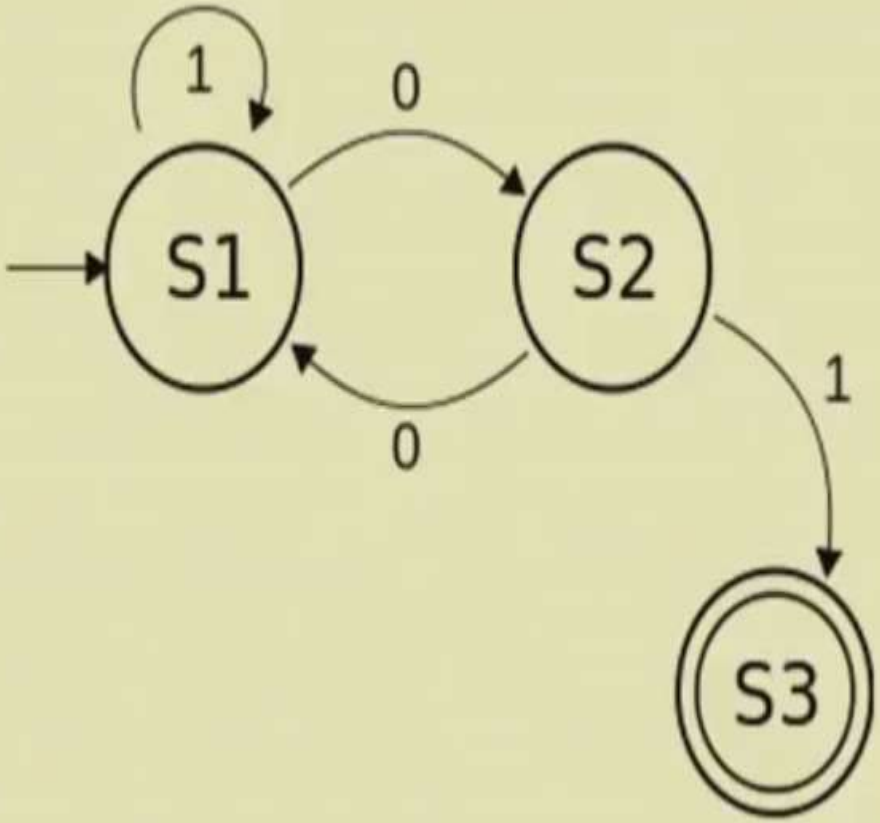
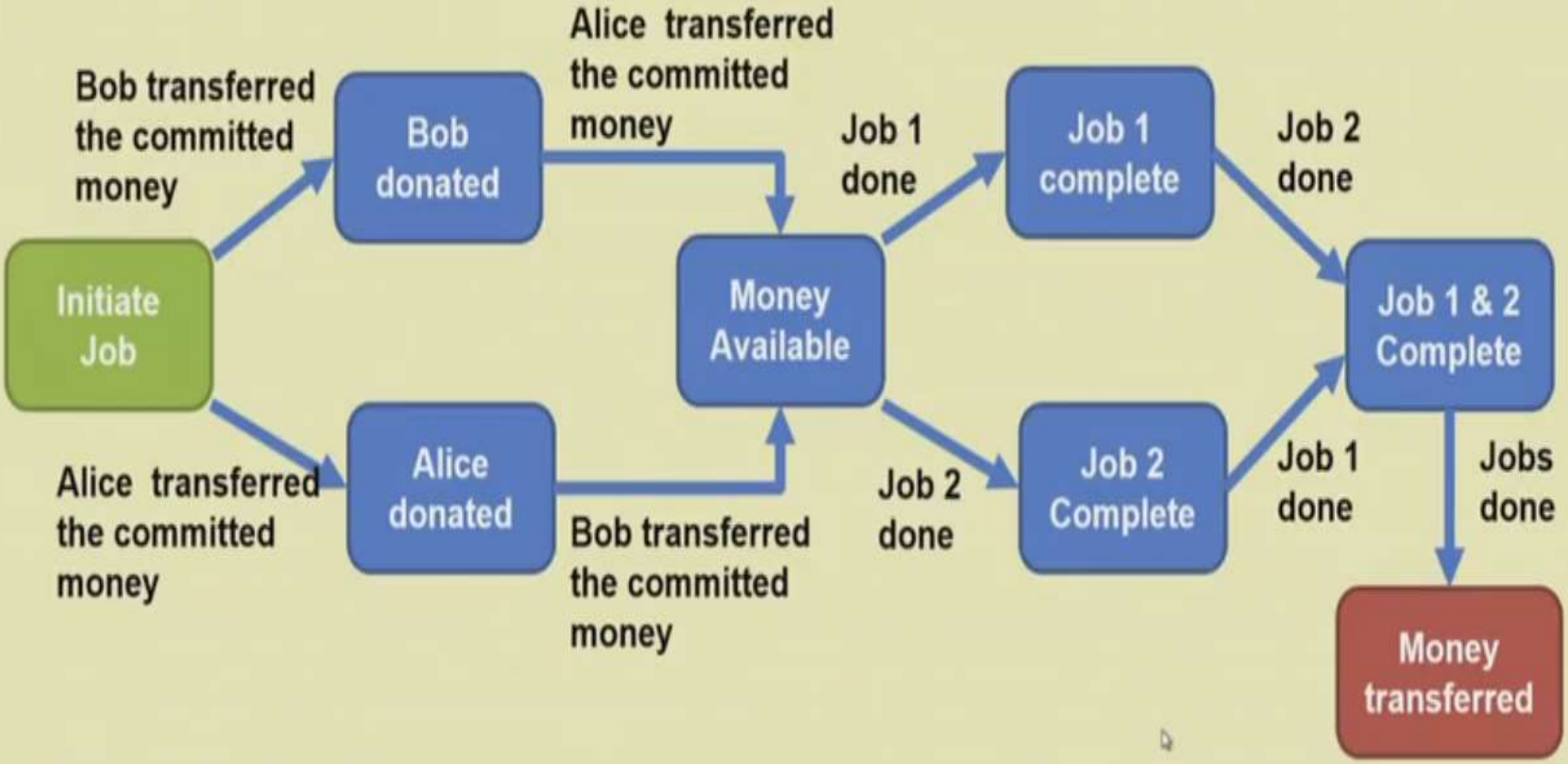


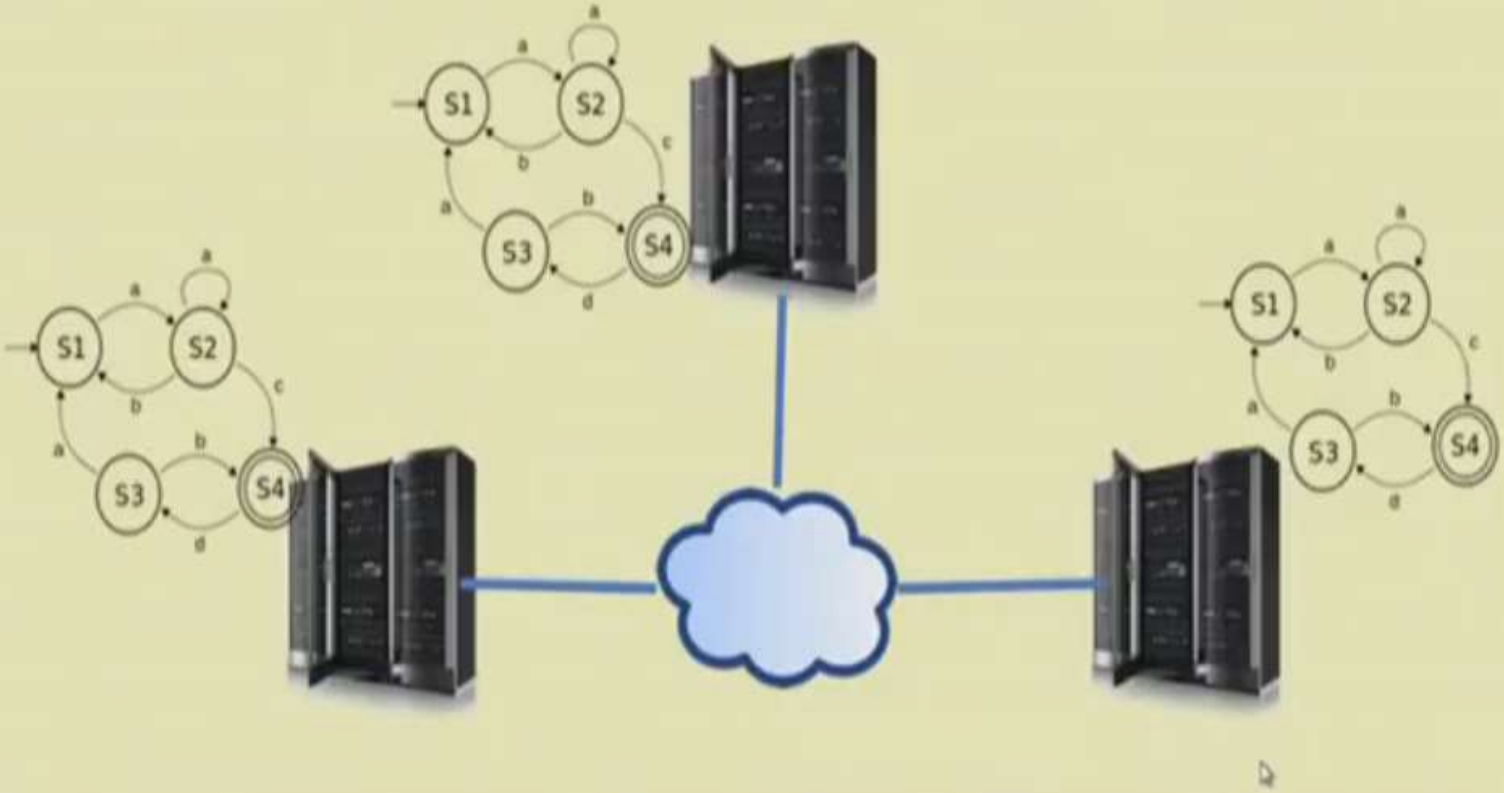
Image source: commons.wikimedia.org

Smart Contract State Machine - Crowd-Funding



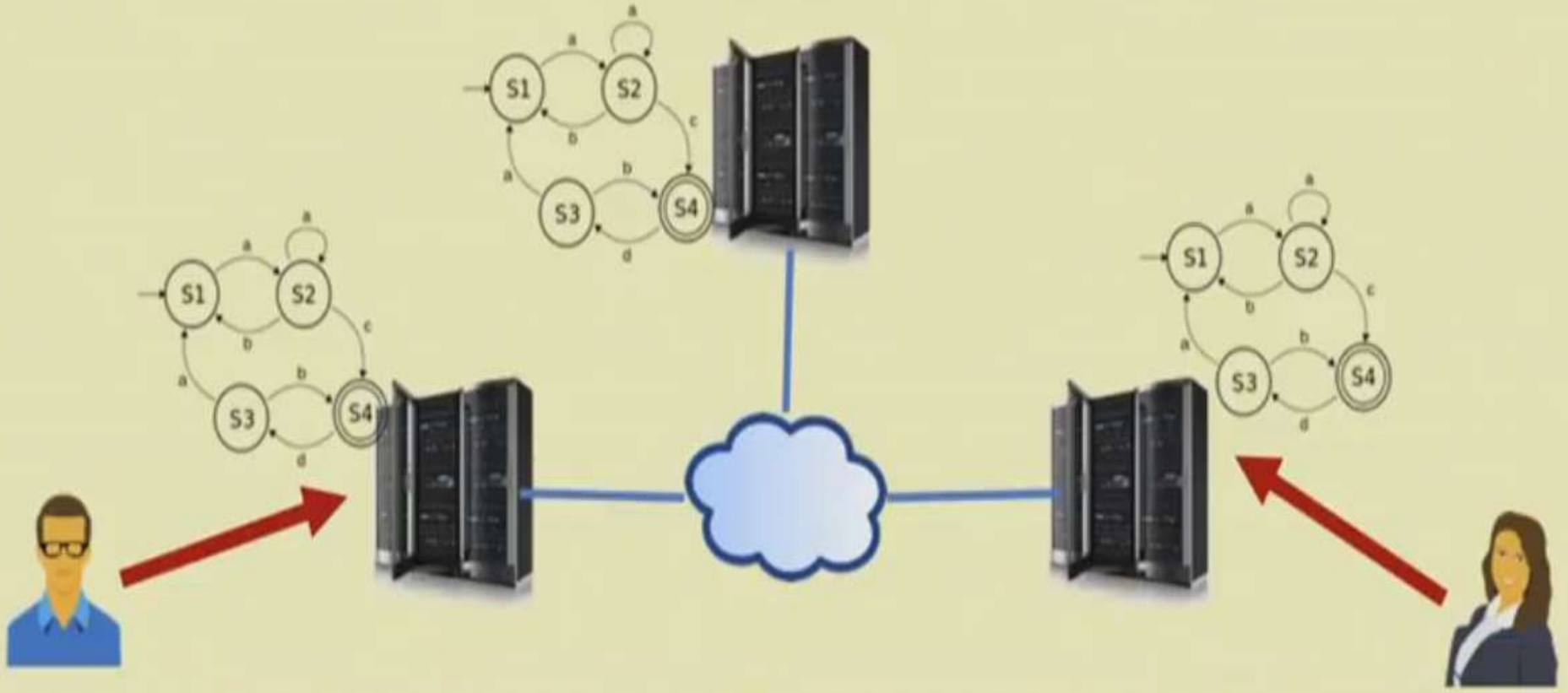
Distributed State Machine Replication

1. Place copies of the state machine on multiple independent servers



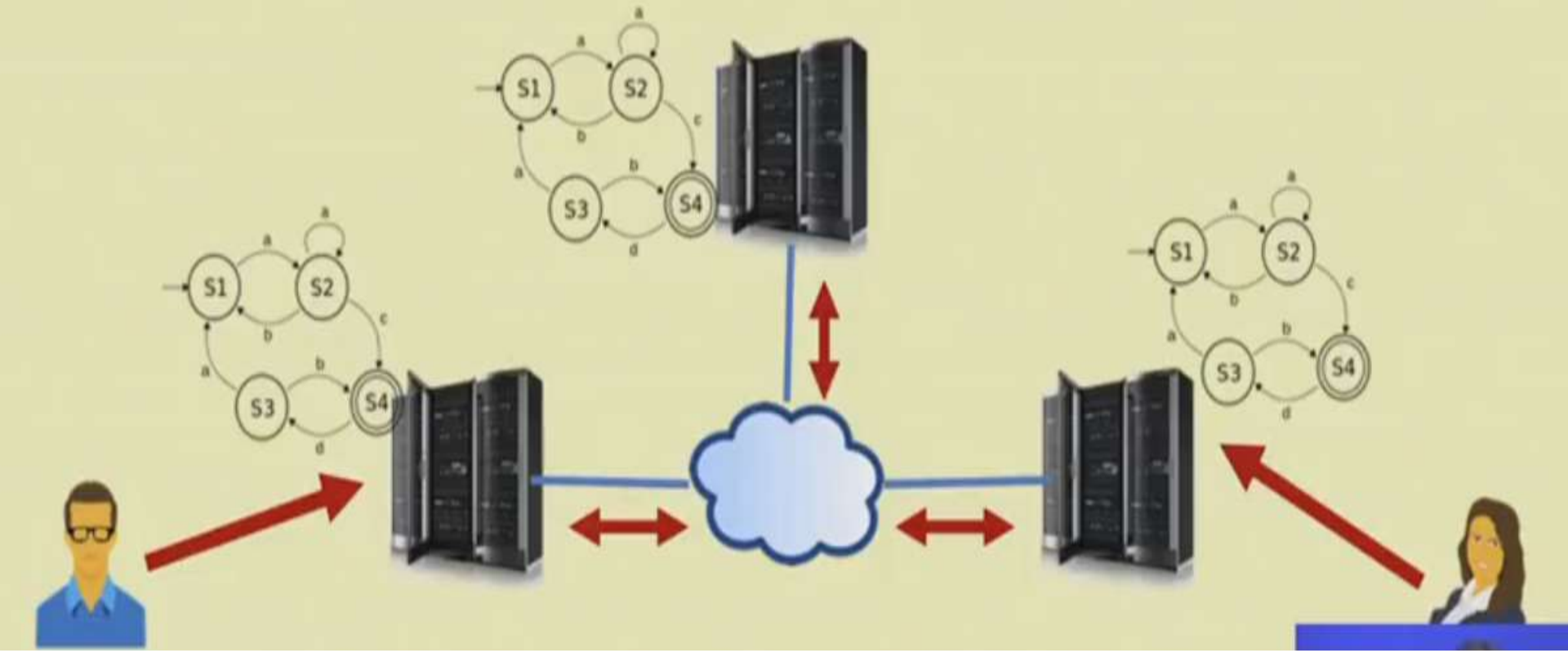
Distributed State Machine Replication

2. Receive client requests, as an input to the state machine



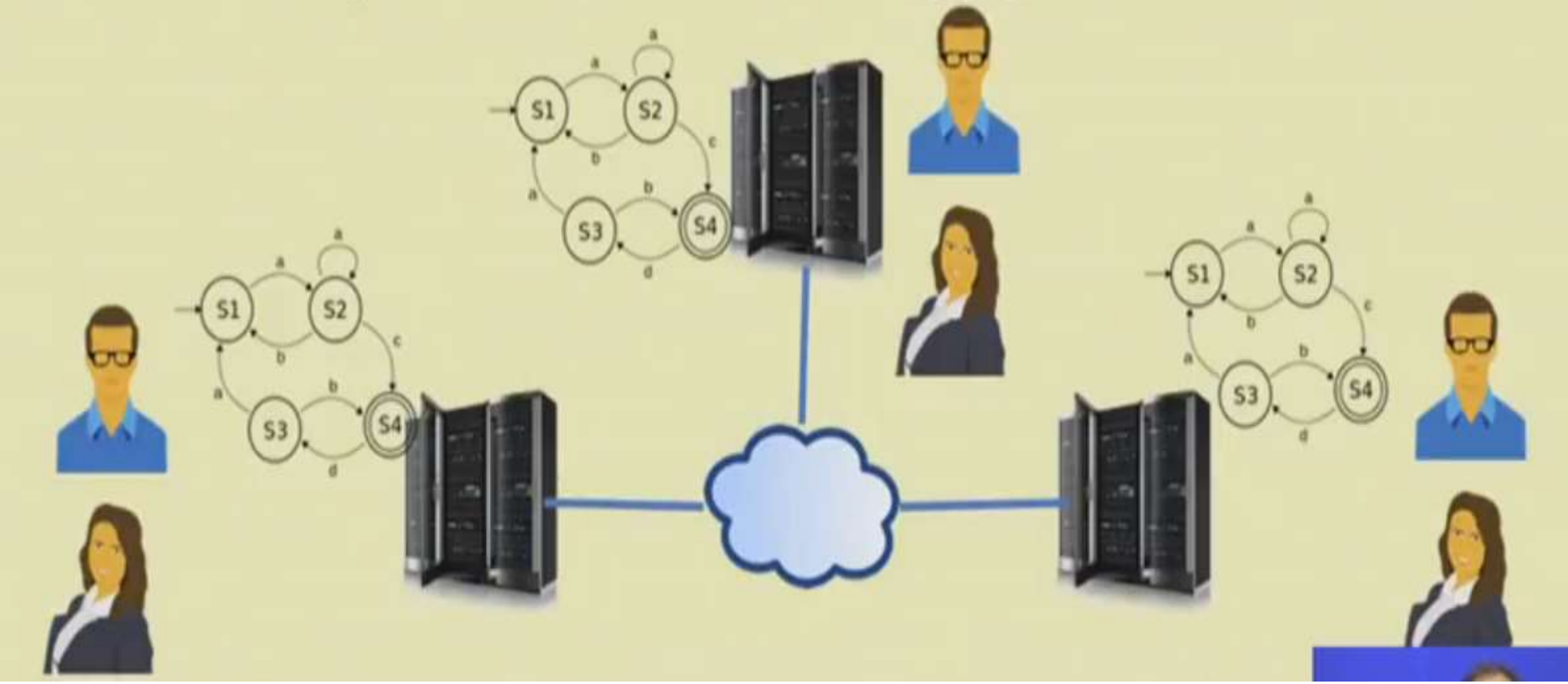
Distributed State Machine Replication

3. Propagate the inputs to all the servers



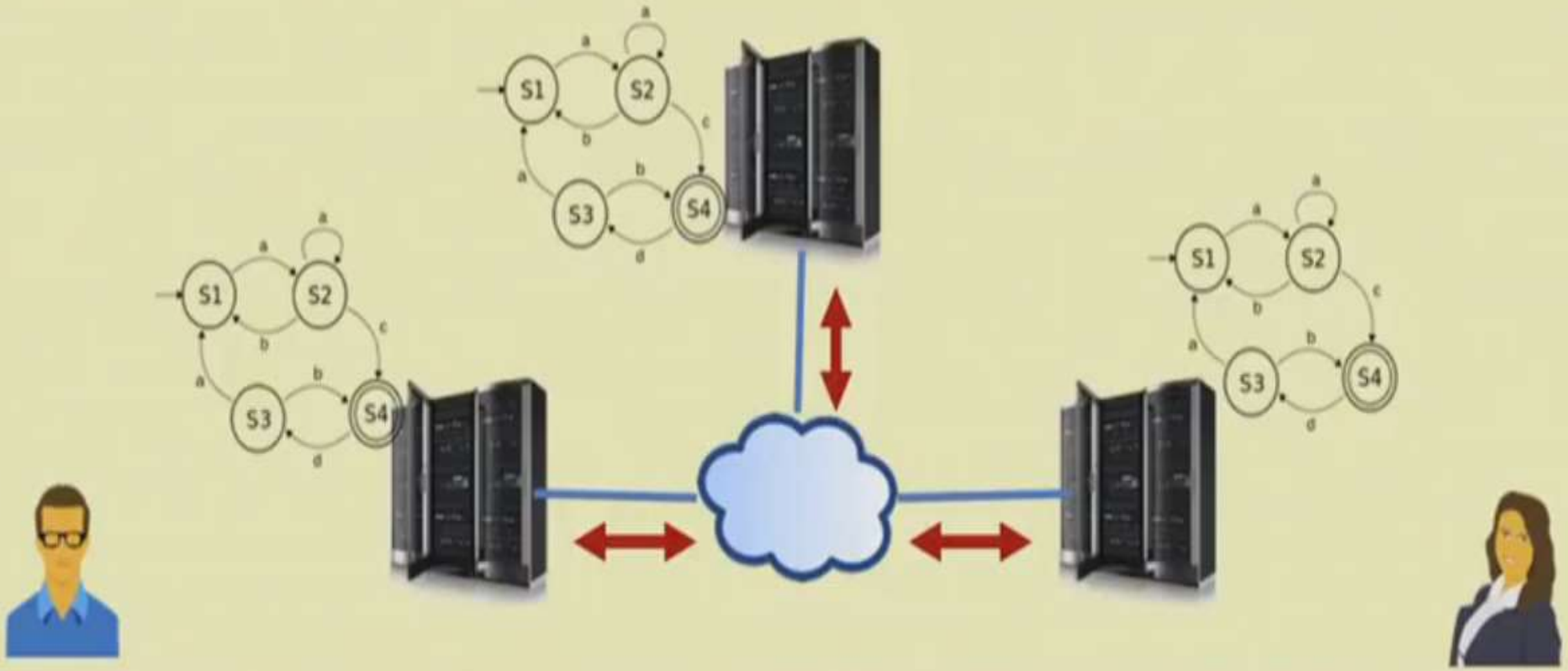
Distributed State Machine Replication

4. Order the inputs based on some ordering algorithm



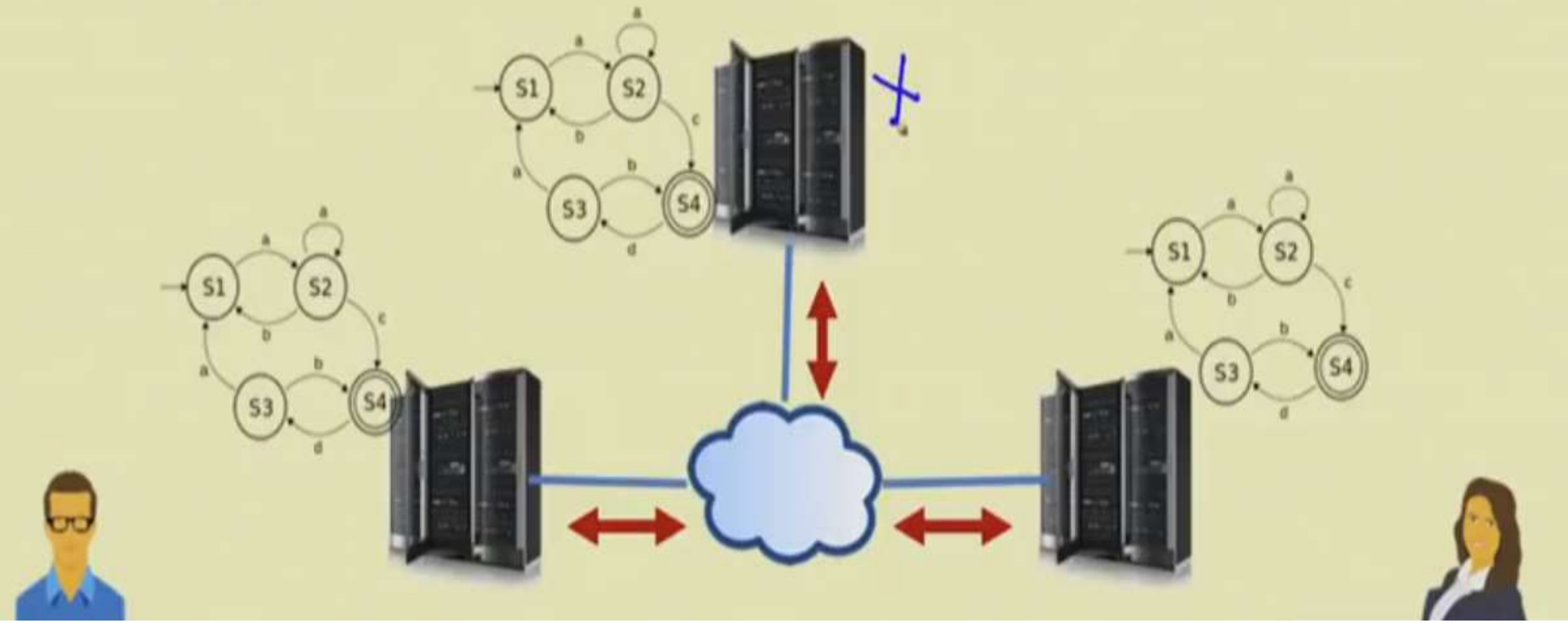
Distributed State Machine Replication

5. Sync the state machines across the servers, to avoid any failure.



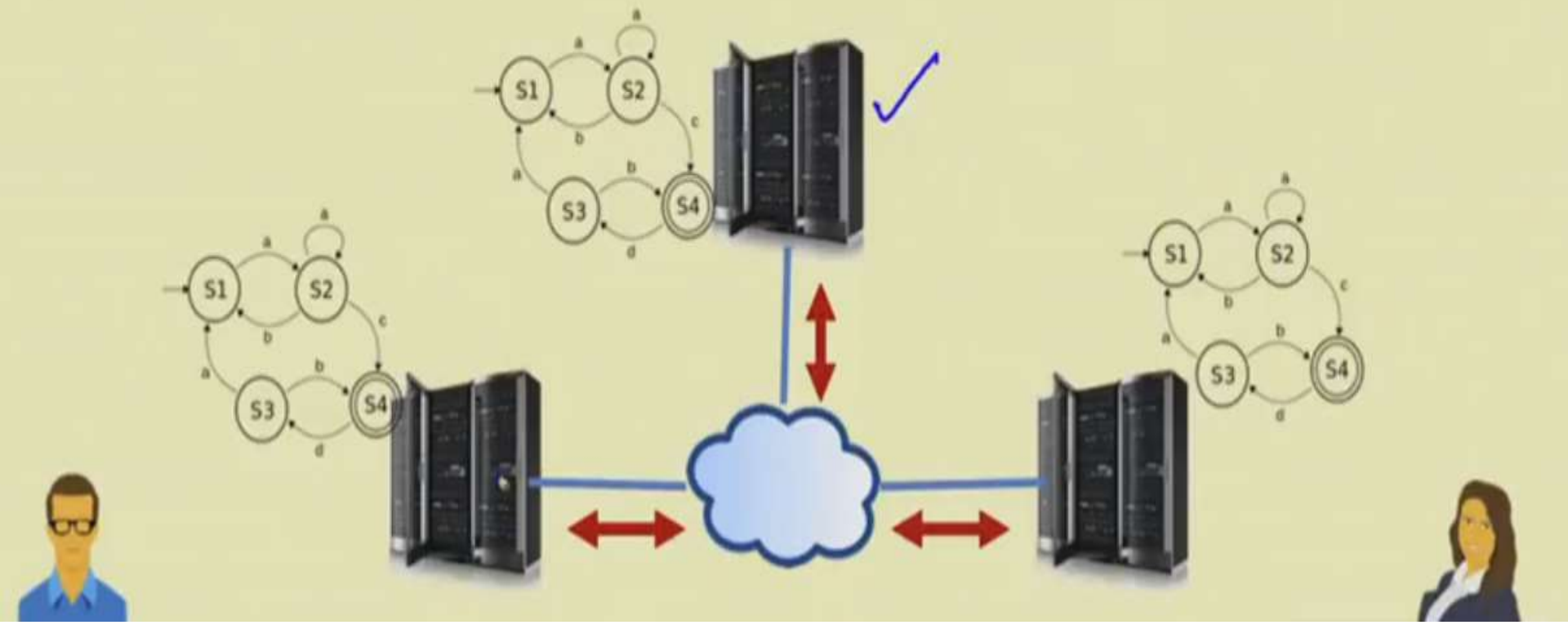
Distributed State Machine Replication

5. Sync the state machines across the servers, to avoid any failure.



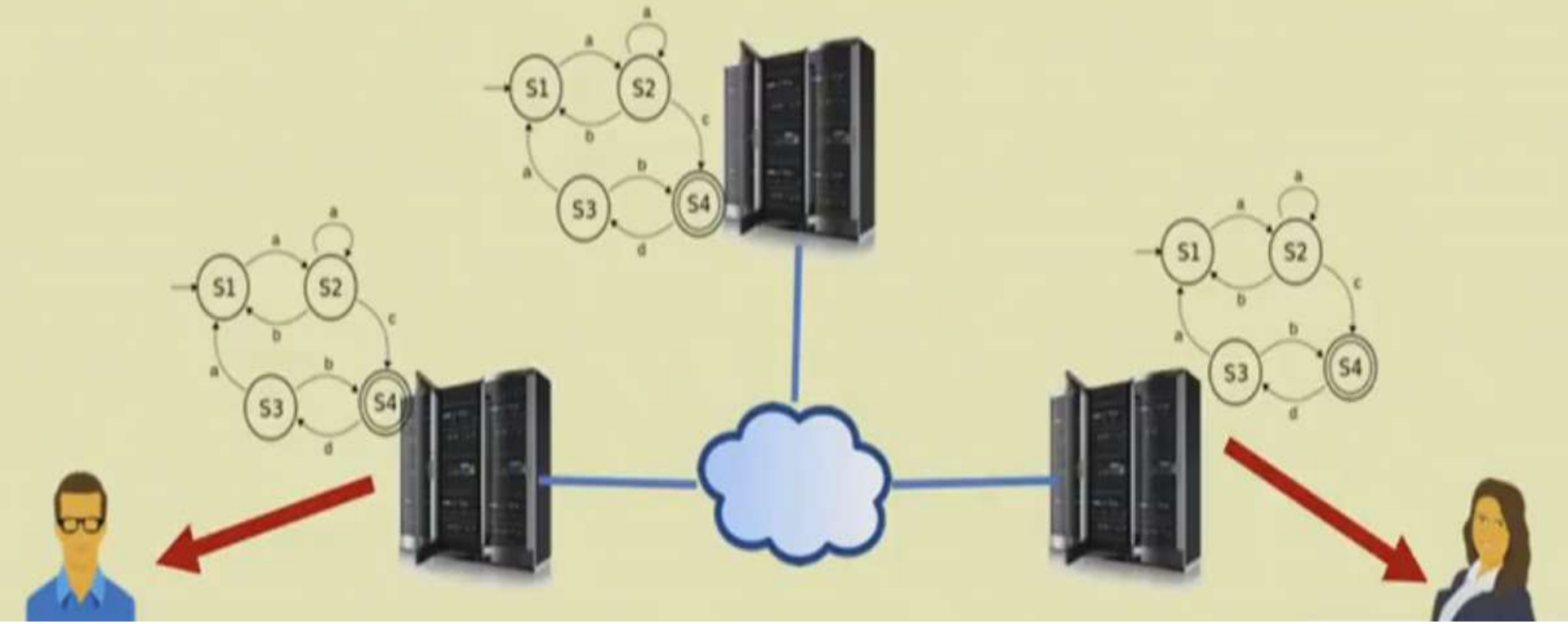
Distributed State Machine Replication

5. Sync the state machines across the servers, to avoid any failure.



Distributed State Machine Replication

6. If output state is produced, inform the clients about the output



Permissioned Blockchain and State Machine Replication

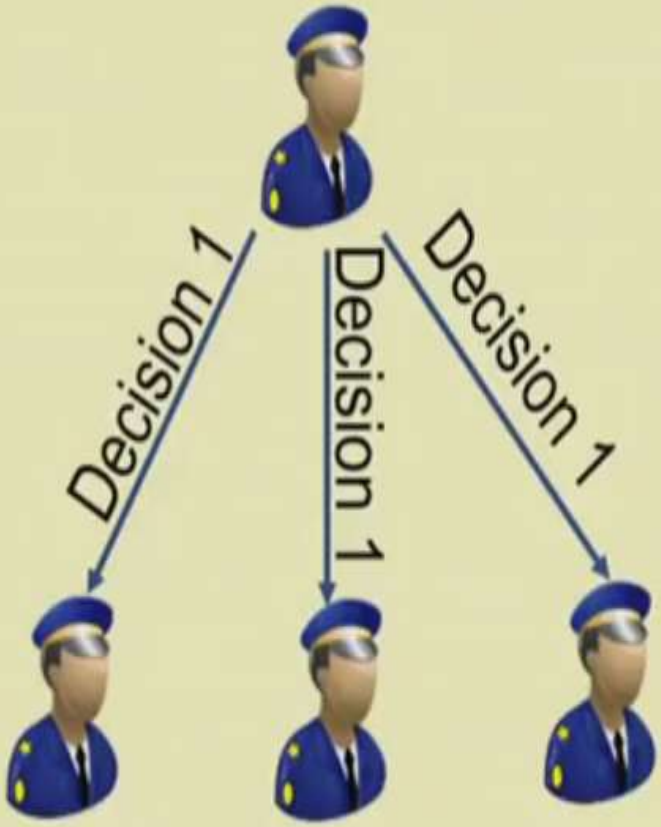
- There is a natural reason to use state machine replication based consensus over permissioned blockchains
  - The network is closed, the nodes know each other, so state replication is possible among the known nodes
  - Avoid the overhead of mining - do not need to spend anything (like power, time, bitcoin) other than message passing
  - However, consensus is still required - machines can be faulty or behave maliciously

Why Distributed Consensus

One Decision Maker

No Consensus

Why Distributed Consensus



Multiple
Decision Maker

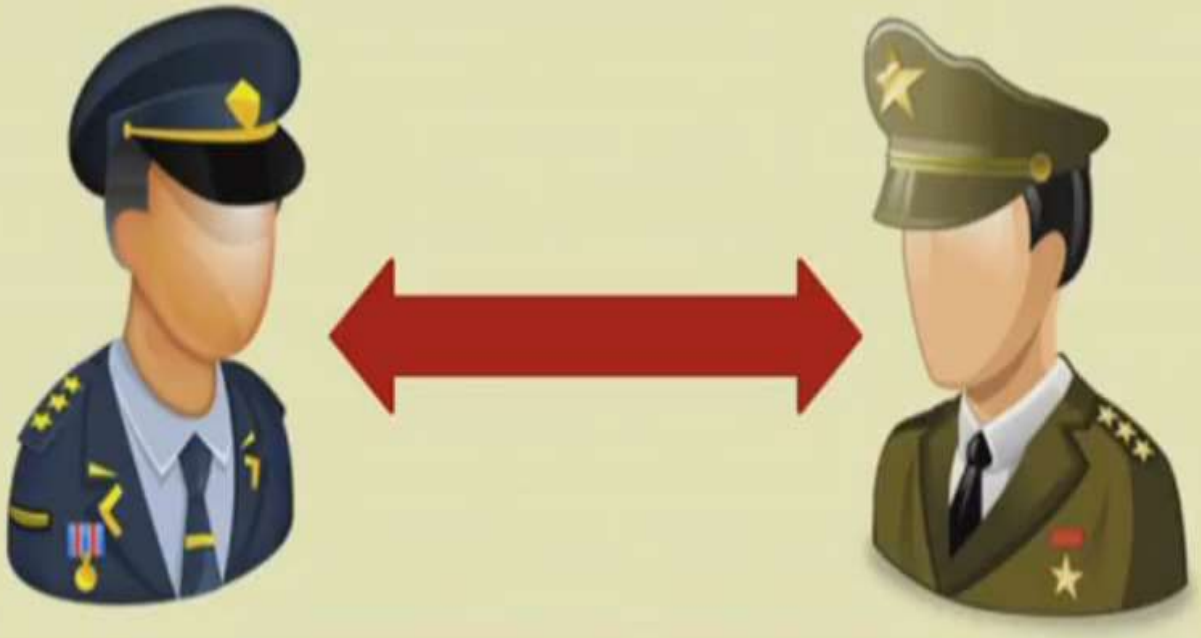
Consensus Required

Why Distributed Consensus

- Reaching agreement in distributed computing
- Replication of common state so that all processes have same view
- Applications:
  - Flight control system: E.g. Boeing 777 and 787
  - Fund transferring system: Bitcoin and cryptocurrencies
  - Leader election/Mutual Exclusion

Why Distributed Consensus

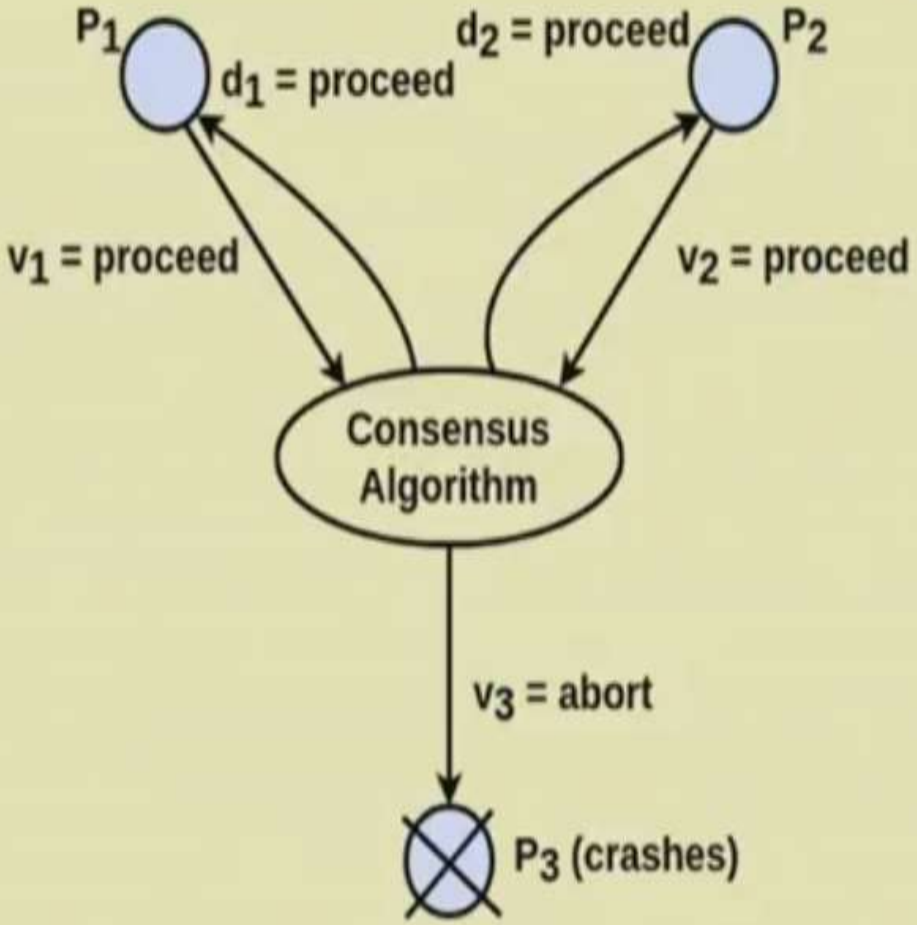
- So, no need of consensus in a single node process.
- **What about when there are two nodes?**
  - Network or partitioned fault, consensus cannot be reached



Faults in Distributed Consensus

- **Crash Fault**
- **Network or Partitioned Faults**
- **Byzantine Faults**
  - malicious behaviour in nodes
  - hardware fault
  - software error

Consensus for three processes



- Each process P_i (i=1,2,...N):
  - **Undecided state:** proposed value v_i from set D
  - **Communication state:** exchange values
  - **Decided state:** set decision variable d_i

Requirements of a Consensus Algorithm

- **Termination:**
  - Eventually each correct process sets its decision variable
- **Agreement:**
  - The decision value of all correct processes is the same
- **Integrity:**
  - If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value

Different Algorithms

- **Crash or Network Faults:**
  - PAXOS
  - RAFT

- **Byzantine Faults (including Crash or Network Failures):**
  - Byzantine fault tolerance (BFT)
  - Practical Byzantine Fault Tolerance (PBFT)

PAXOS
Consensus Algorithm

PAXOS

- Introduced by L. Lamport in 1989 but published in 2001:
  - There was a lot of discussion among the community about the correctness of this particular algorithm.

- Name after **Paxos island** (now paxi) in Greece, where Lamport wrote that the parliament had to function

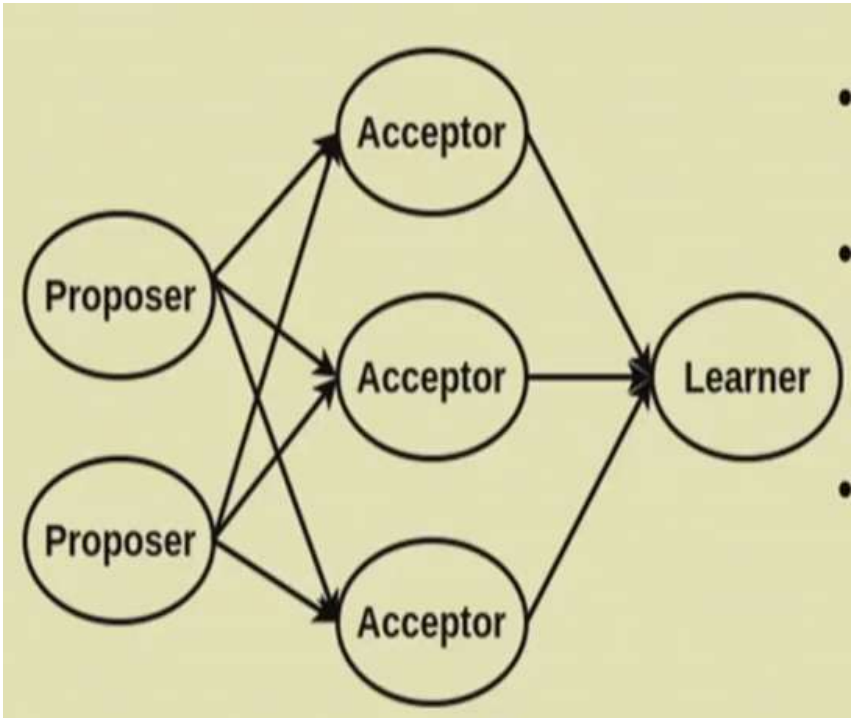
- **Objective:** Choose a single value under crash or network fault

PAXOS

- Based on Majority decision
- **Condition:** All members will follow same decision
- **Example:**
  - Group of students are deciding to go for PIZZA party
  - Two options (or more): Domino's or Pizza-Hut
  - Wait for proposal from other group members
  - If no proposal: Propose for Domino's
- **System Process:**
  - Making a proposal
  - Accepting a value
  - Handling failures

PAXOS

- Three Type of nodes :

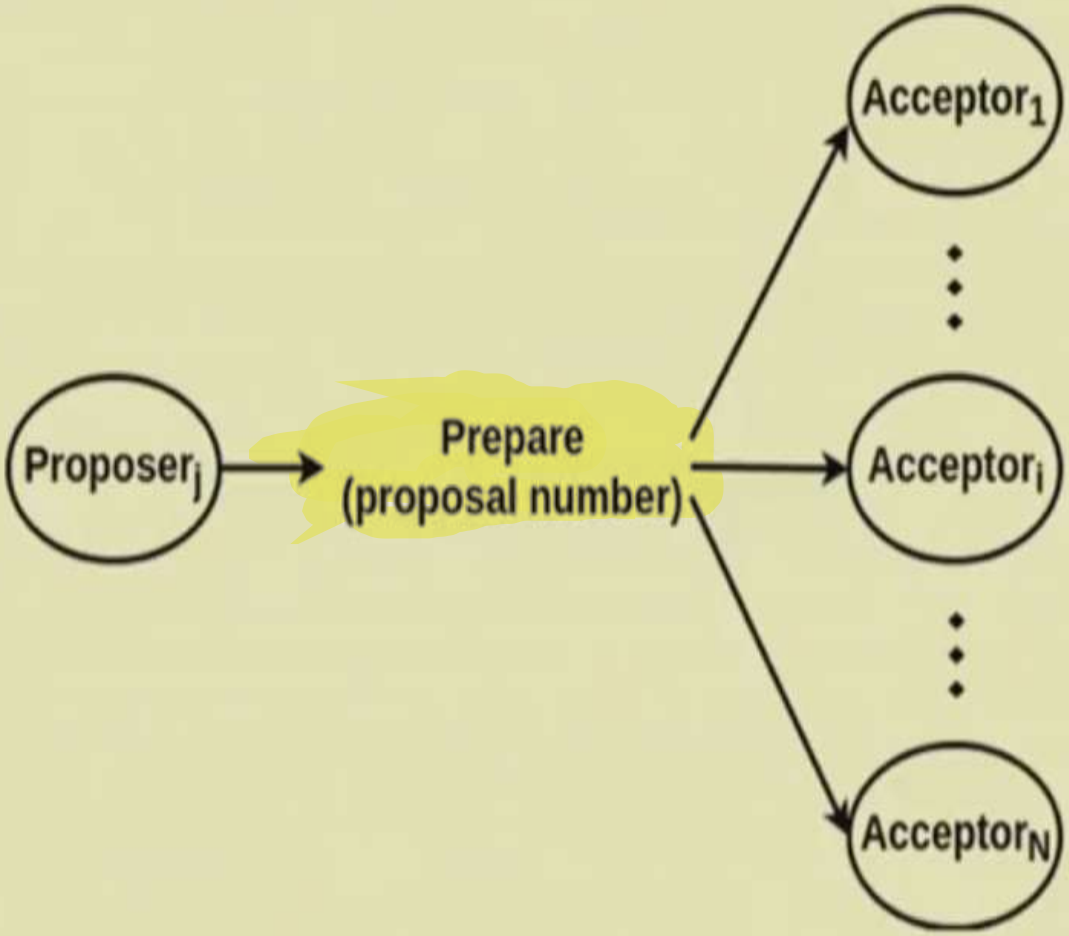


– **Proposers:** Propose a value that should be chosen by consensus

– **Acceptors:** Form the consensus and accept values (Accept/Reject)

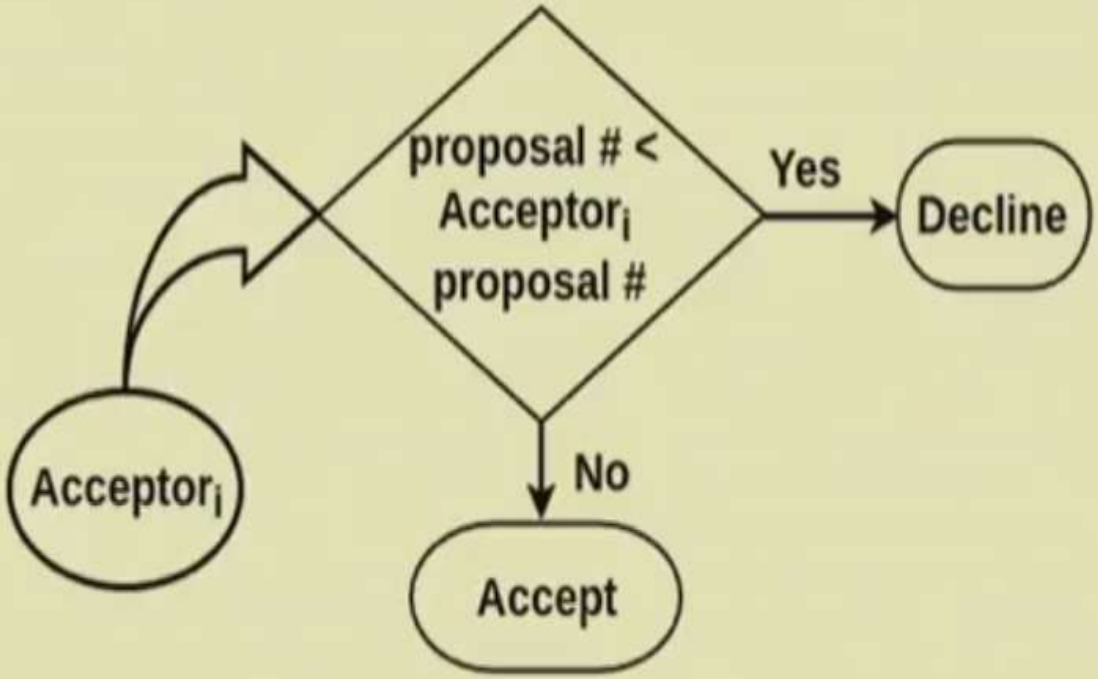
– **Learner:** Learn value chosen by each acceptor (Everyone)

Making a Proposal: Proposer Process



- **proposal number:** form a timeline, biggest number considered up-to-date

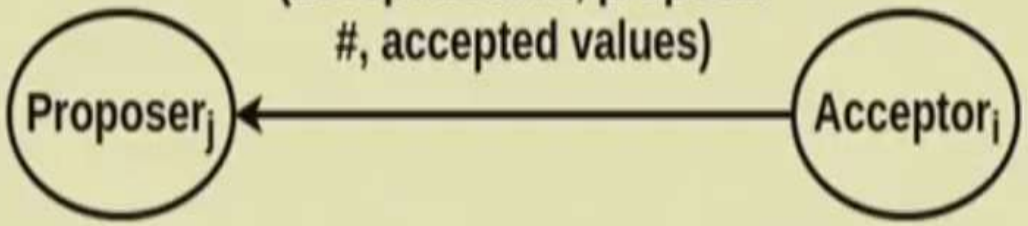
Making a Proposal: Acceptor's Decision Making



- Each acceptor compares received proposal number with the current known values for all proposer's prepare message

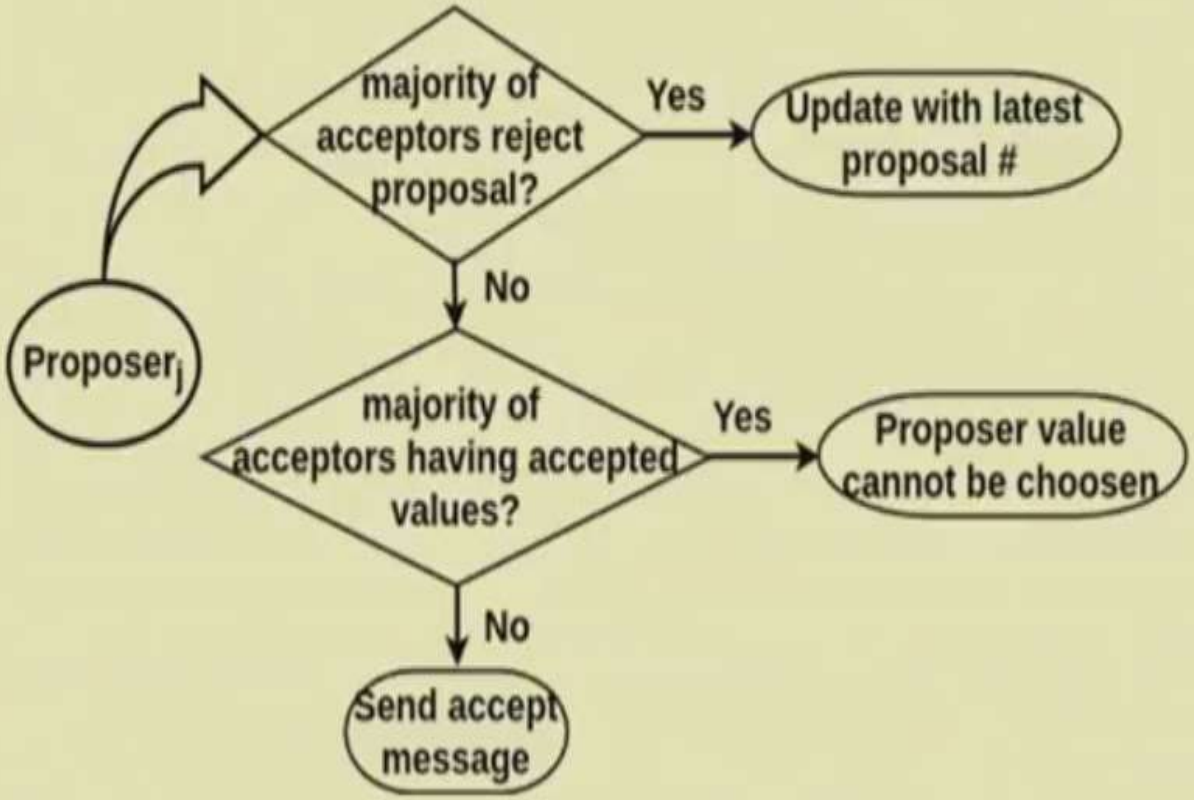
All members are listening to the proposals

Making a Proposal: Acceptor's Message



- **accept/decline:** whether prepare accepted or not
- **proposal number:** biggest number the acceptor has seen
- **accepted values:** already accepted values from other proposer

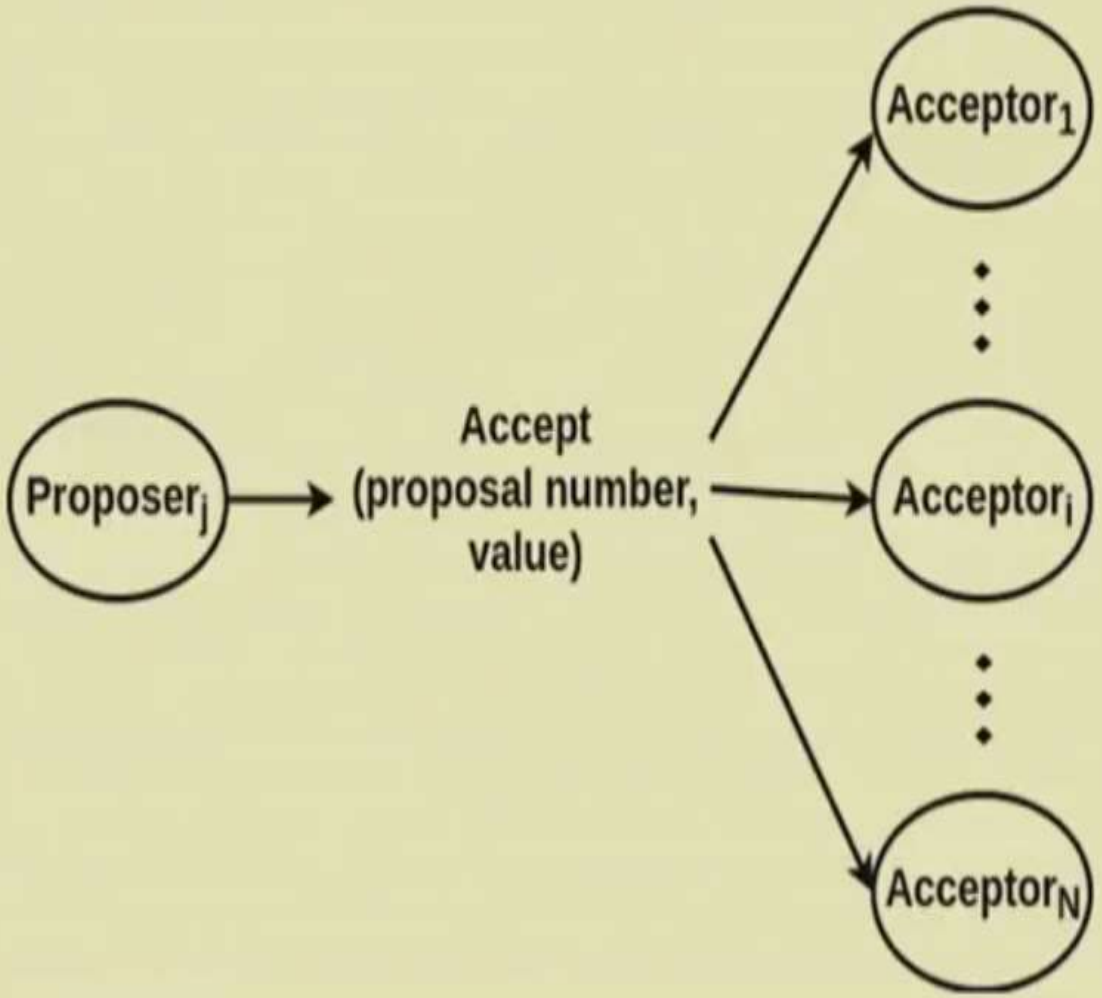
Accepting a Value: Proposer's Decision Making



- Proposer receive a response from **majority** of acceptors before proceeding

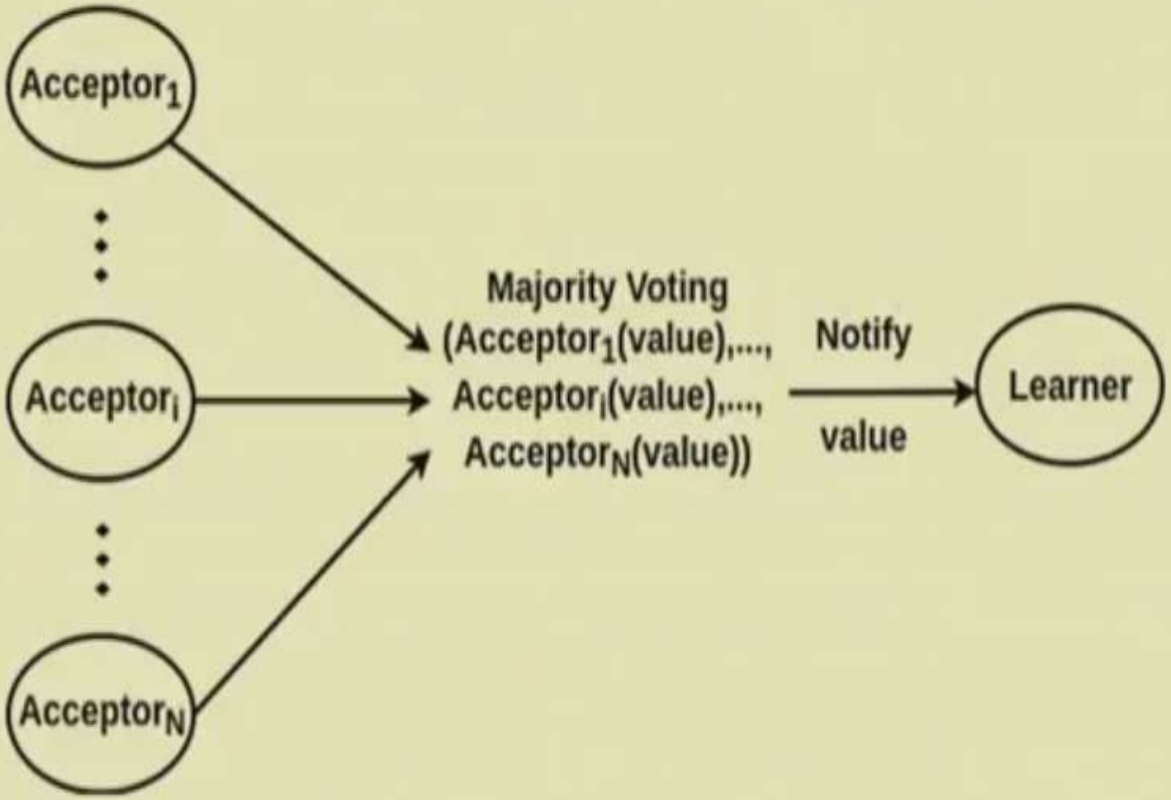
The value that you have shared that is coming to be a consensus. (If accepted by majority of accepters)

Accepting a Value: Accept Message



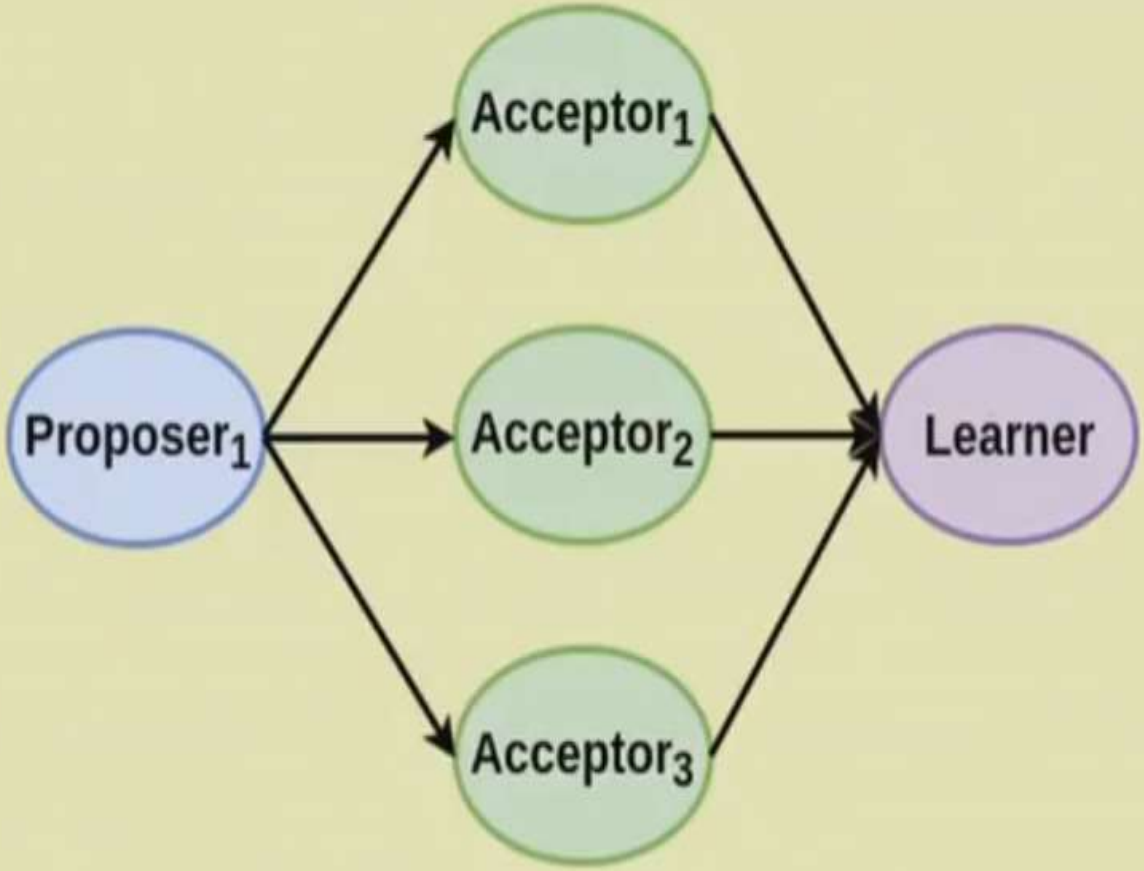
- **proposal number:** same as prepare phase value
- **value:** single value proposed by proposer

Accepting a Value: Notifying Learner



- Each acceptor accept value from any of the proposer
- Notify learner the majority voted value

Single Proposer: No Rejection



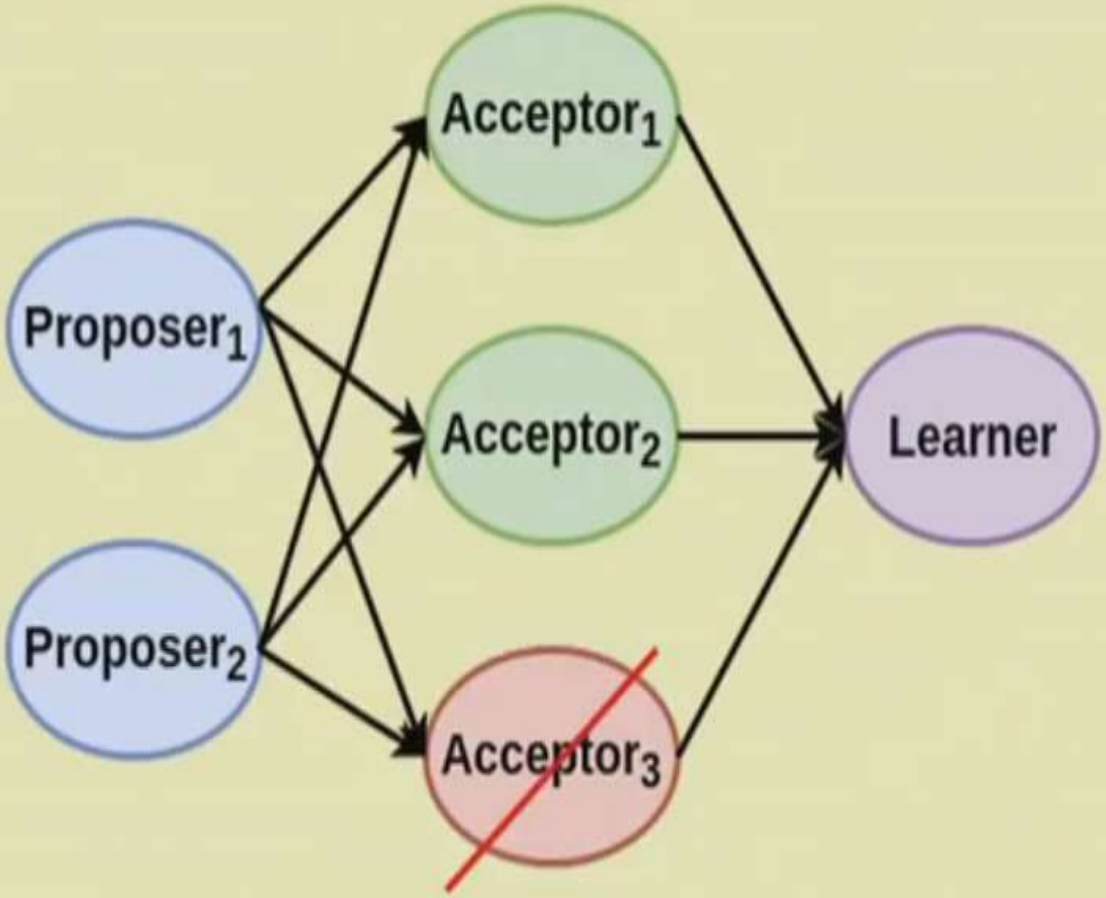
- Proposer always have proposal with biggest number
- No proposal rejected

Possibilities of Failure

- Acceptor Failure:
  - During prepare phase
  - During accept phase

- Proposer Failure
  - During prepare phase
  - During accept phase

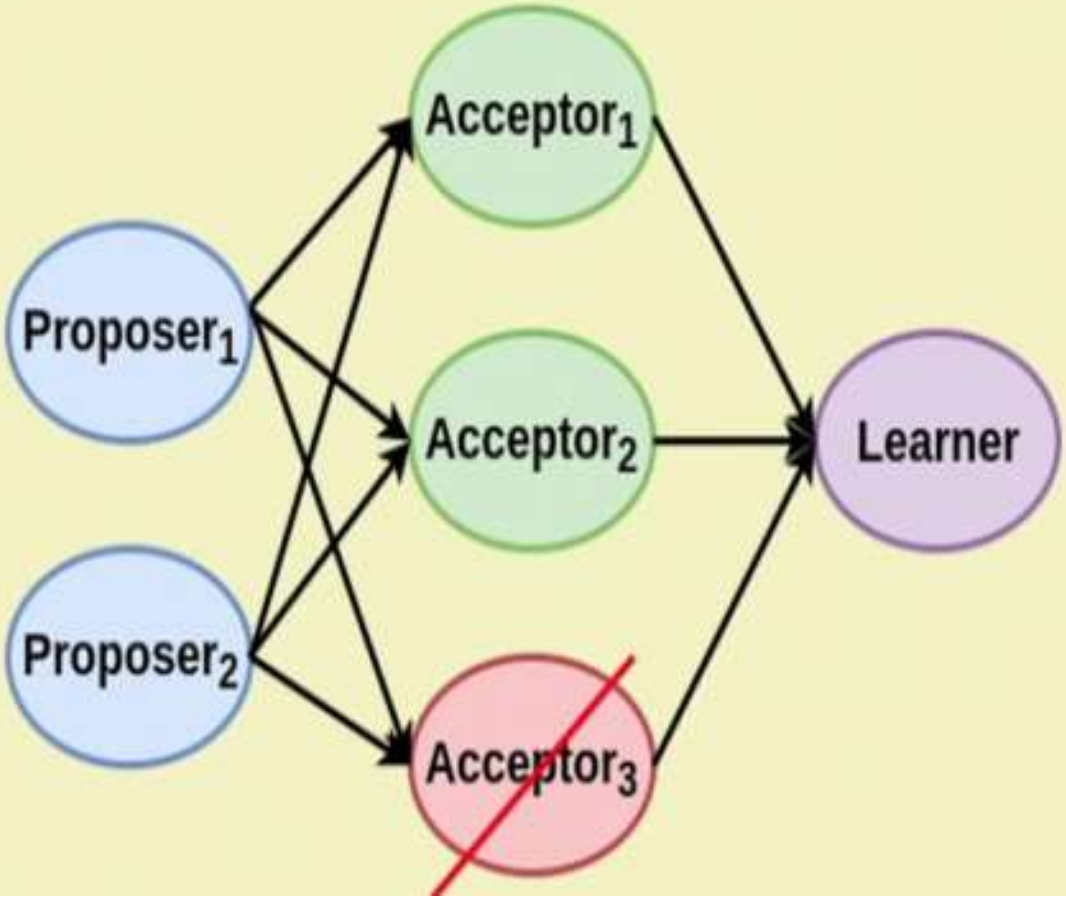
Handling Failure: Acceptor Failure



- **Acceptor fails during prepare**
  - No issues, other acceptor can hear the proposal and vote

let us see that whenever **certain acceptor fails**
If fails **during prepare phase** → No issue

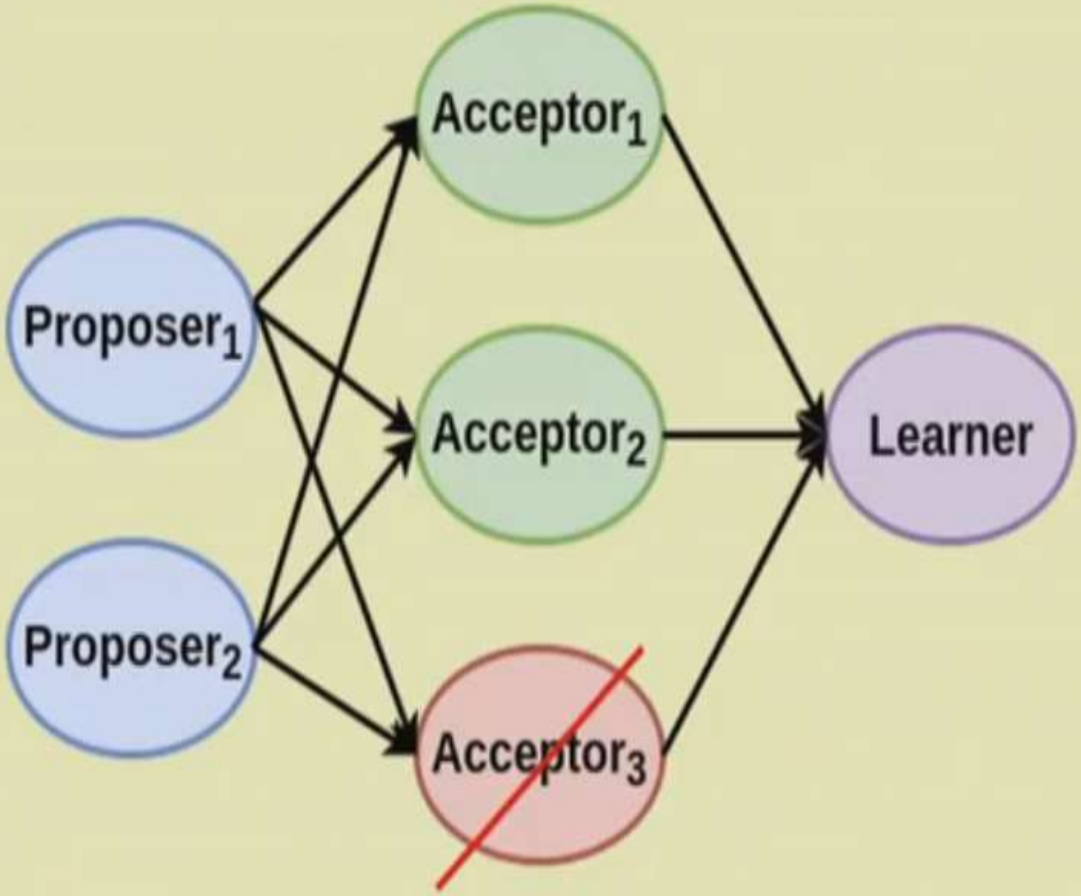
Handling Failure: Acceptor Failure



- **Acceptor fails during accept**
  - Again, no issues, other acceptor can vote for the proposal

let us see that whenever **certain acceptor fails**
If fails **during accept phase** → No issue

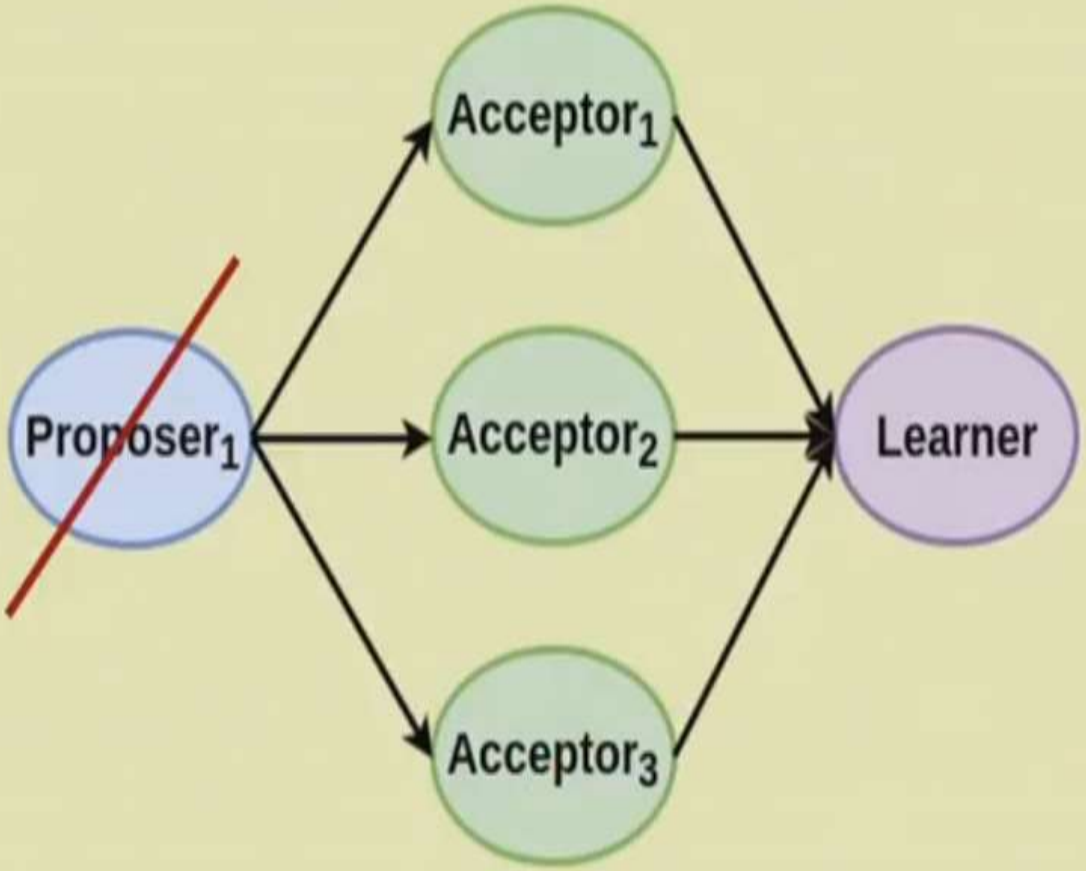
Handling Failure: Acceptor Failure



- More than N/2 - 1 acceptors fail
  - no proposer get a reply
  - no values can be accepted

Ensure majority. E.g. if 24/50 acceptors fails → No issue

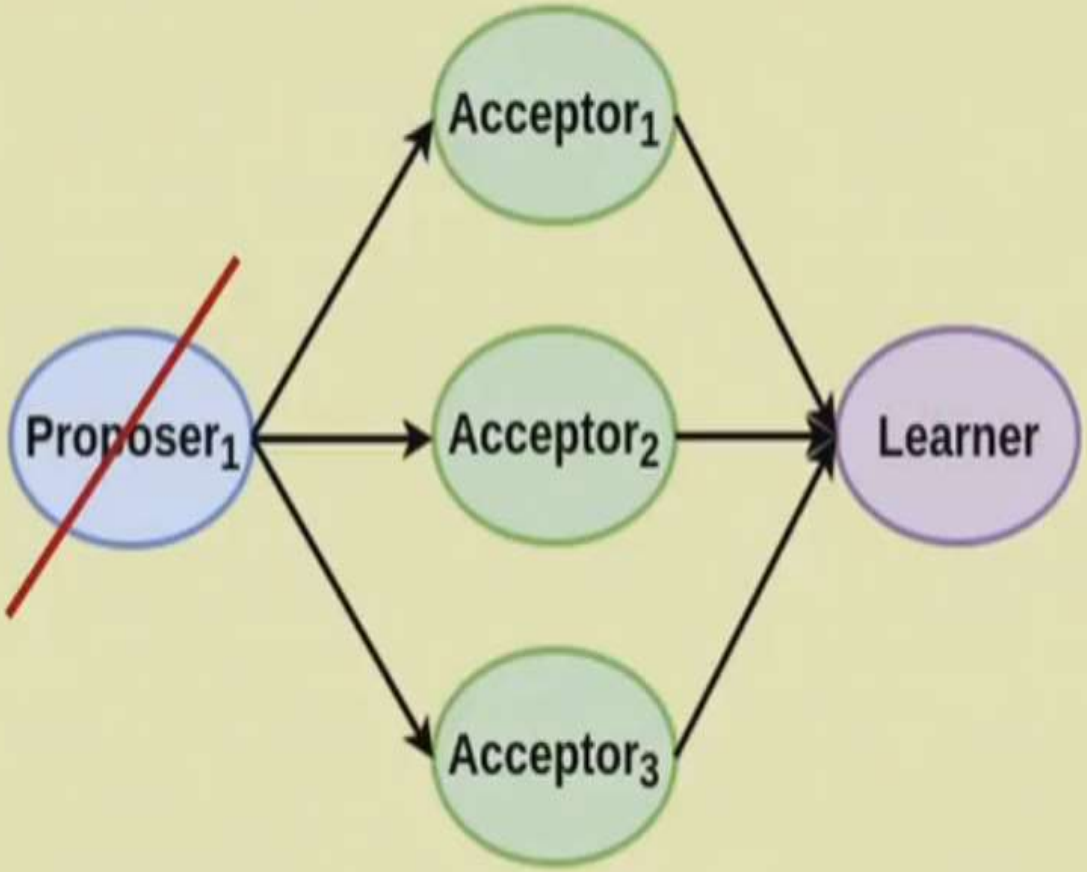
Handling Failure: Proposer Failure



- Proposer fails during prepare phase
  - Acceptors wait, wait, wait, and then someone else become the proposer

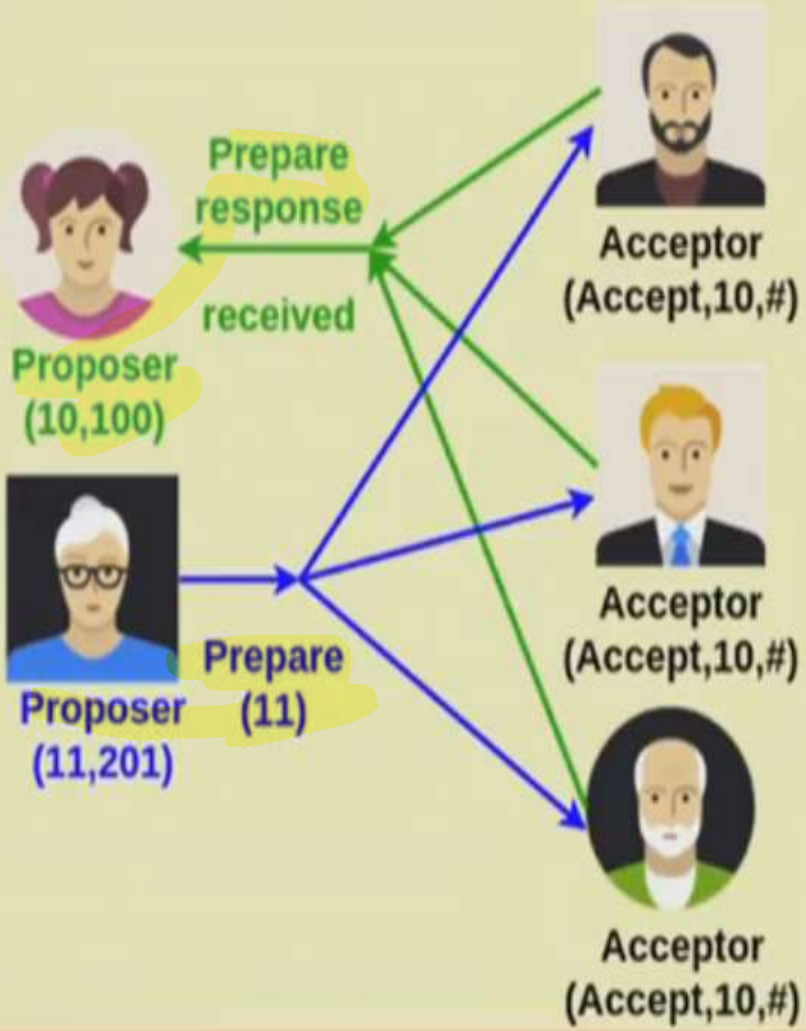
If acceptor do not here for any proposal one of the acceptor, →it becomes the proposer and propose a

Handling Failure: Proposer Failure



- **Proposer fails during accept phase**
  - Acceptors have already agreed upon whether to choose or not to choose the proposal

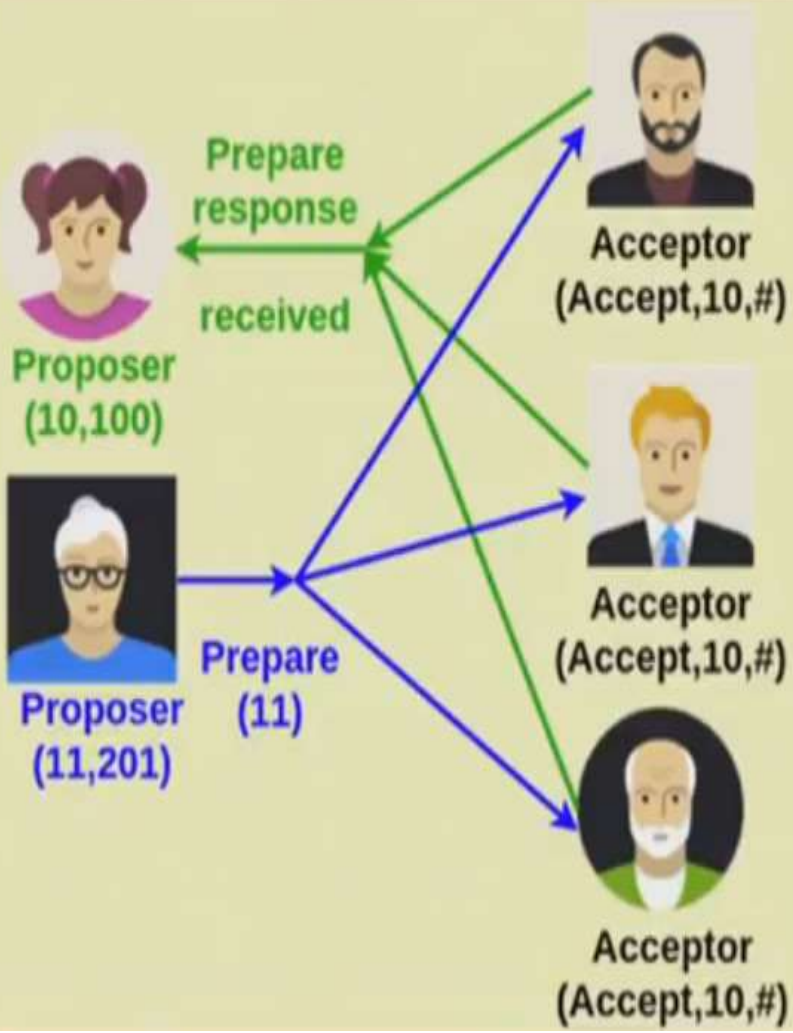
Handling Failure: Dueling Proposers



- Proposer received confirmations to her prepare message from majority
  - yet to send accept messages
- Another proposer sends prepare message with higher proposal number
- Block the first proposer's proposal from being accepted

Assign ID to the proposer

Handling Failure: Dueling Proposers



- Use **leader election** - select one of the proposer as leader
- Paxos can be used for leader election !!

Periodically multiple message forwarding → High complexity

Multi/Repeated-Paxos (sequence of selections) → Higher complexity

THANK YOU