**UC Santa Barbara**

**Computer Science Department**

# Modern Large-Scale Data Management Systems after 40 Years of Consensus

Mohammad Javad Amiri, Divyakant Agrawal, Amr El Abbadi

The 36th International Conference on Data Engineering (ICDE)

# Reaching Agreement in the Presence of Faults

M. PEASE, R. SHOSTAK, AND L. LAMPORT

*SRI International, Menlo Park, California*

ABSTRACT. The problem addressed here concerns a set of isolated processors, some unknown subset of which may be faulty, that communicate only by means of two-party messages. Each nonfaulty processor has a private value of information that must be communicated to each other nonfaulty processor. Nonfaulty processors always communicate honestly, whereas faulty processors may lie The problem is to devise an algorithm in which processors communicate their own values and relay values received from others that allows each nonfaulty processor to infer a value for each other processor The value inferred for a nonfaulty processor must be that processor's private value, and the value inferred for a faulty one must be consistent with the corresponding value inferred by each other nonfaulty processor

It is shown that the problem is solvable for, and only for, $n \geq 3m + 1$, where $m$ is the number of faulty processors and $n$ is the total number. It is also shown that if faulty processors can refuse to pass on information but cannot falsely relay information, the problem is solvable for arbitrary $n \geq m \geq 0$. This weaker assumption can be approximated in practice using cryptographic methods

KEY WORDS AND PHRASES. agreement, authentication, consistency, distributed executive, fault avoidance, fault tolerance, synchronization, voting

CR CATEGORIES: 3.81, 4.39, 5.29, 5.39, 6.22

© 1980 ACM 0004-5411/80/0400-0228 $00 75

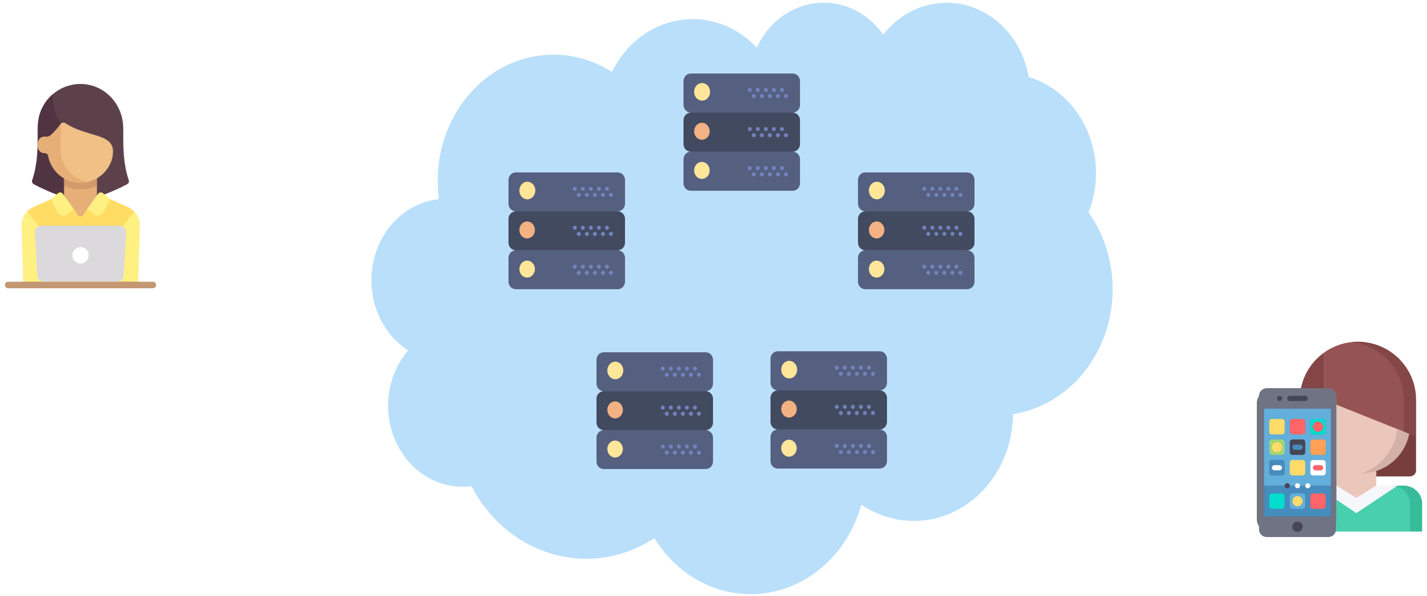
Journal of the Association for Computing Machinery, Vol 27, No 2, April 1980, pp 228–234

# Fault Tolerance

- Build systems that tolerate machine and network faults
- Replicate data on multiple servers to enhance availability
  - Uses State Machine Replication
  - Needs to ensure that replicas remain consistent
  - Needs consensus among different servers

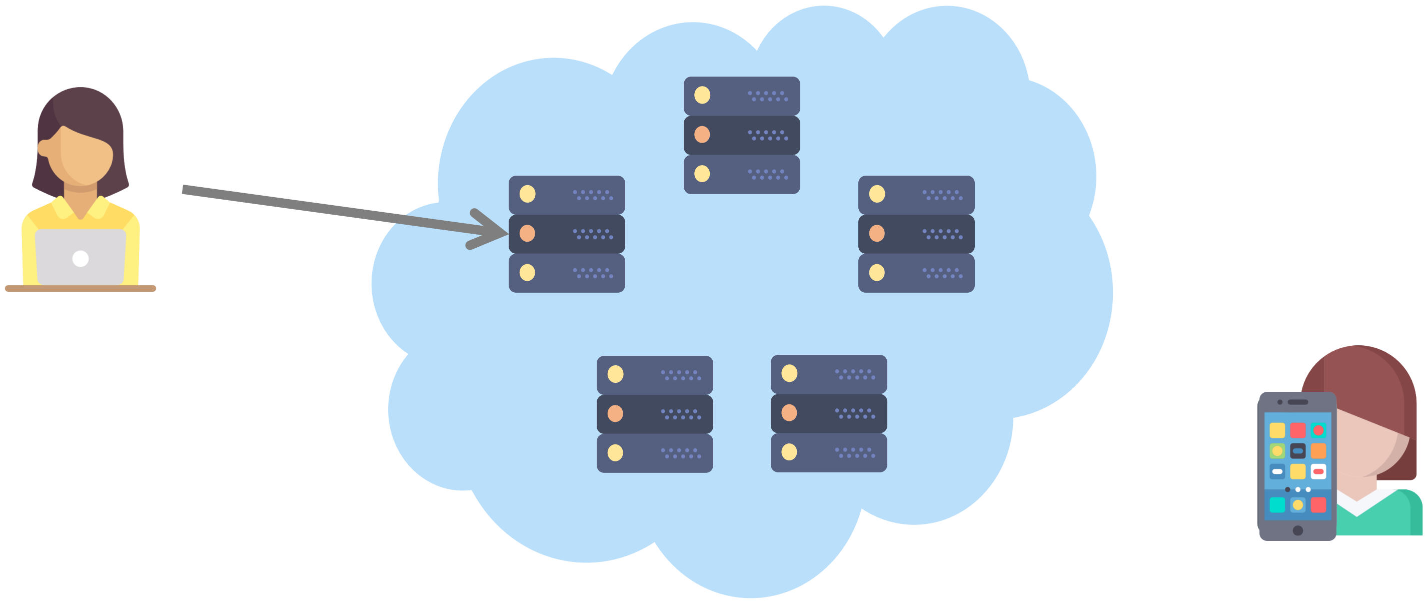
# Fault Tolerance

- Build systems that tolerate machine and network faults
- Replicate data on multiple servers to enhance availability
  - Uses State Machine Replication
  - Needs to ensure that replicas remain consistent
  - Needs consensus among different servers



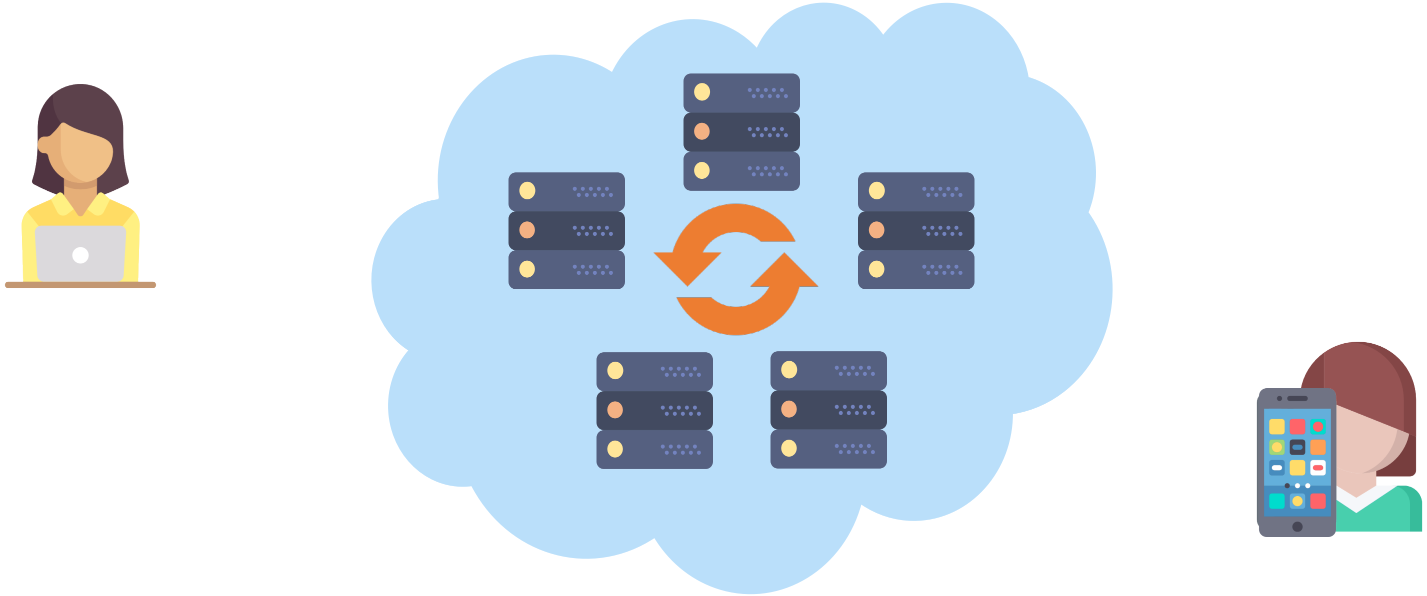
# Fault Tolerance

- Build systems that tolerate machine and network faults
- Replicate data on multiple servers to enhance availability
  - Uses State Machine Replication
  - Needs to ensure that replicas remain consistent
  - Needs consensus among different servers



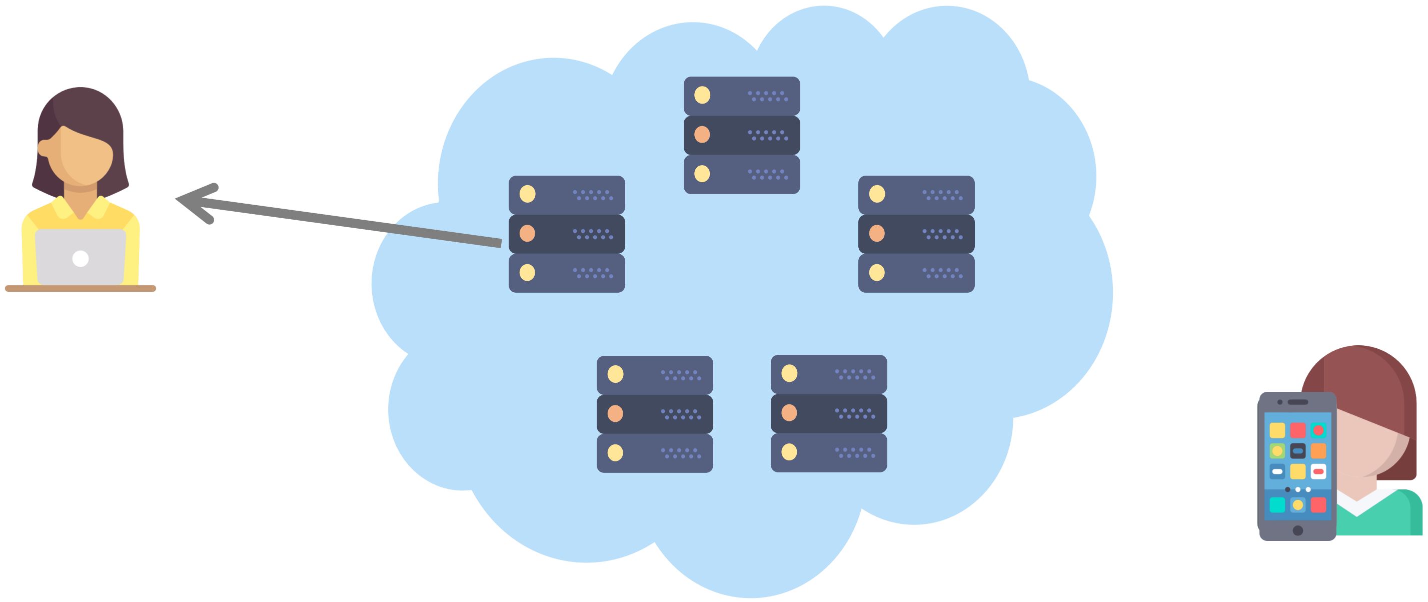
# Fault Tolerance

- Build systems that tolerate machine and network faults
- Replicate data on multiple servers to enhance availability
  - Uses State Machine Replication
  - Needs to ensure that replicas remain consistent
  - Needs consensus among different servers



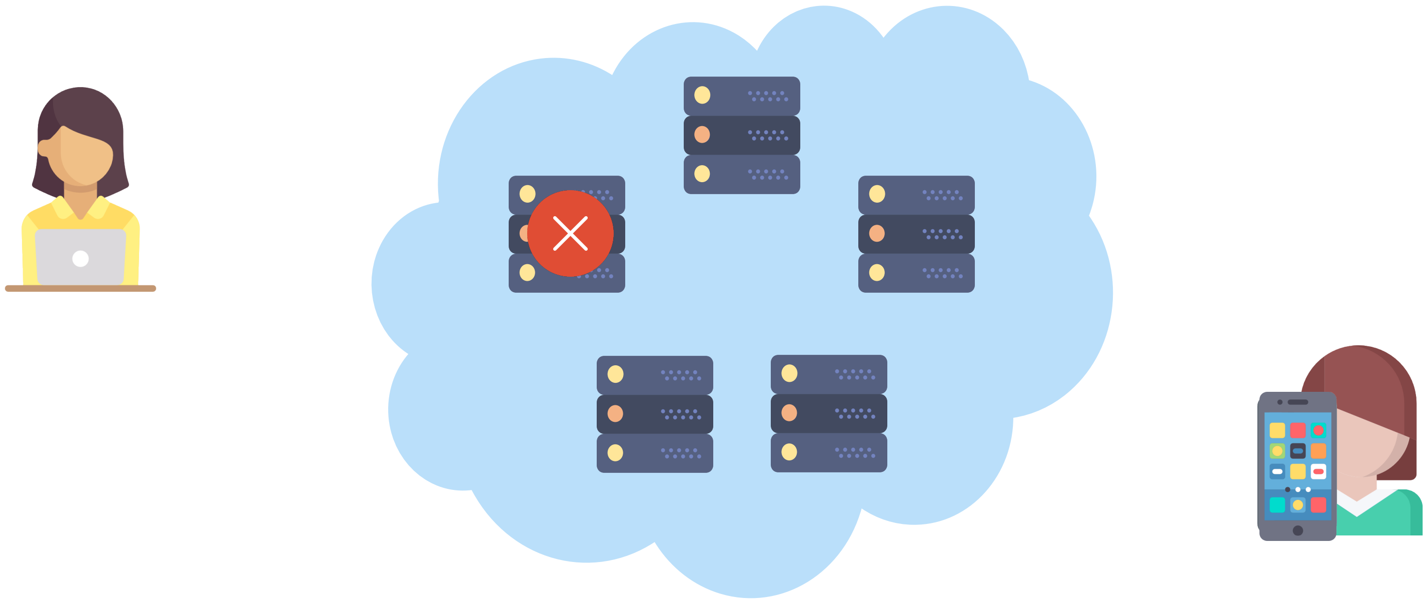
# Fault Tolerance

- Build systems that tolerate machine and network faults
- Replicate data on multiple servers to enhance availability
  - Uses State Machine Replication
  - Needs to ensure that replicas remain consistent
  - Needs consensus among different servers



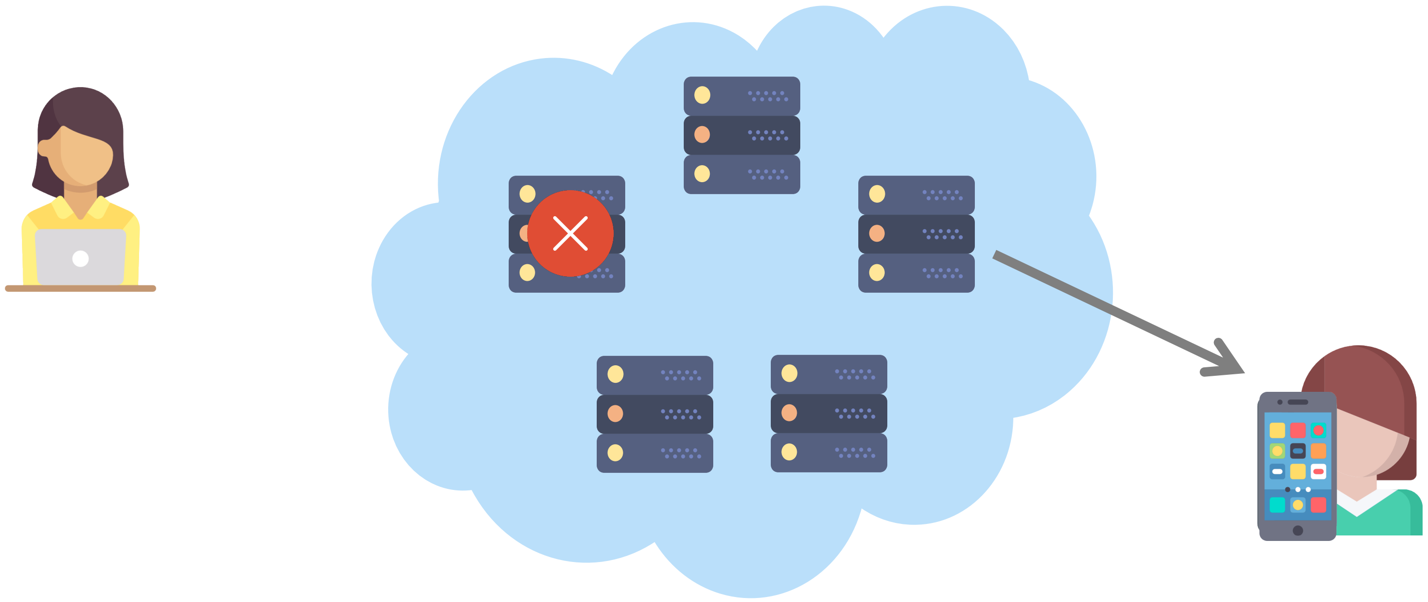
# Fault Tolerance

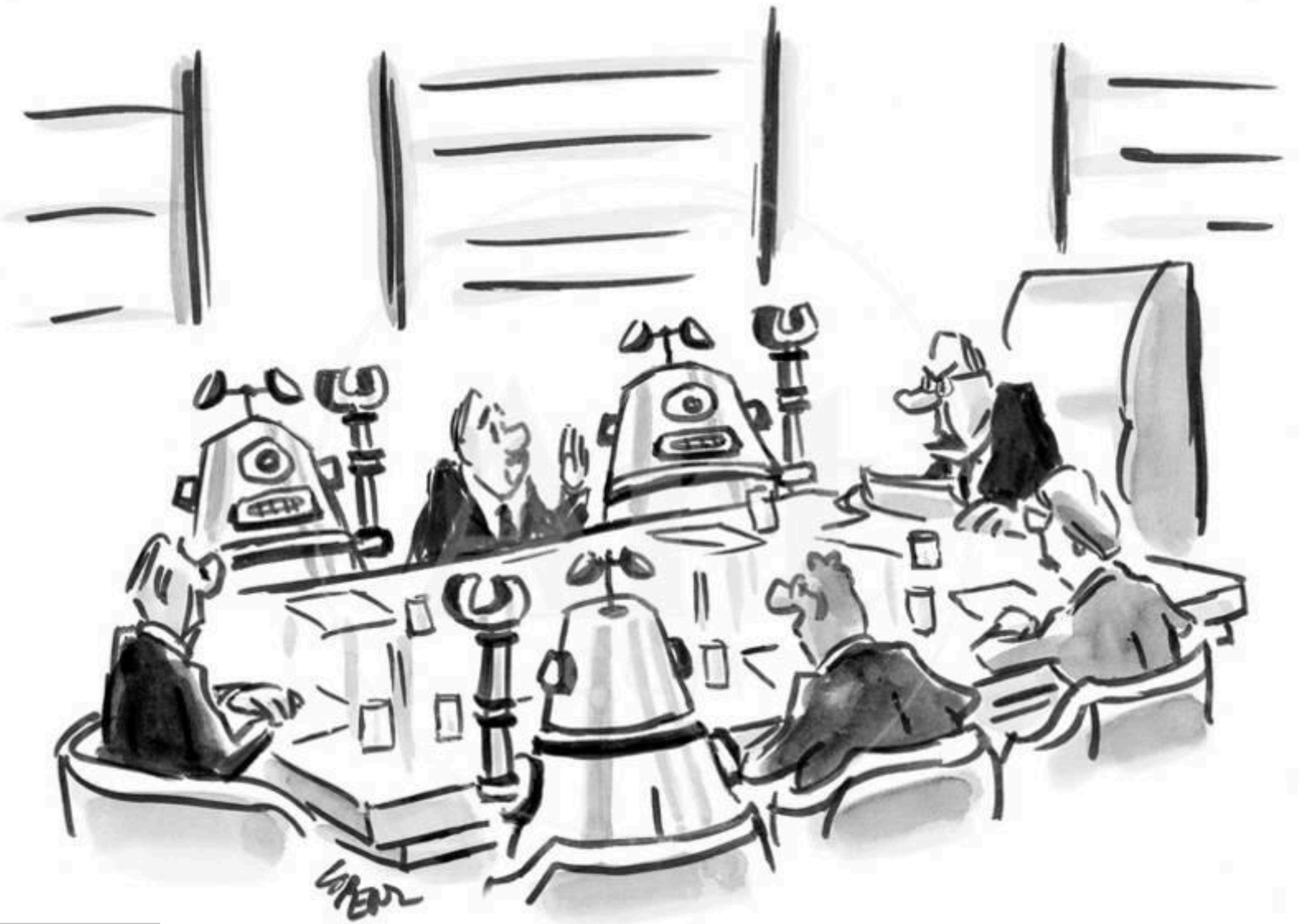
- Build systems that tolerate machine and network faults
- Replicate data on multiple servers to enhance availability
  - Uses State Machine Replication
  - Needs to ensure that replicas remain consistent
  - Needs consensus among different servers



# Fault Tolerance

- Build systems that tolerate machine and network faults
- Replicate data on multiple servers to enhance availability
  - Uses State Machine Replication
  - Needs to ensure that replicas remain consistent
  - Needs consensus among different servers



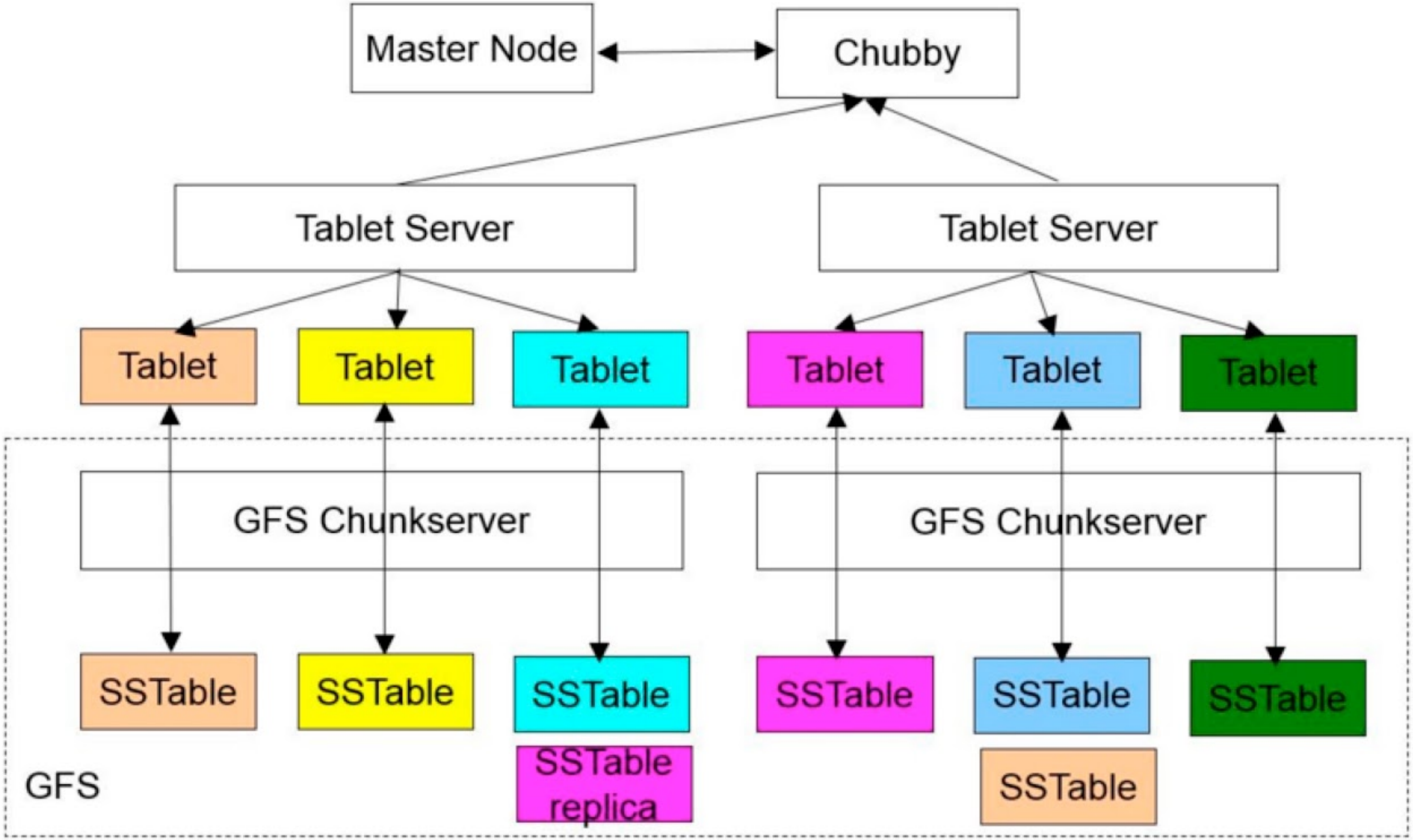


"Jenkins, if I want another yes-man I'll build one."

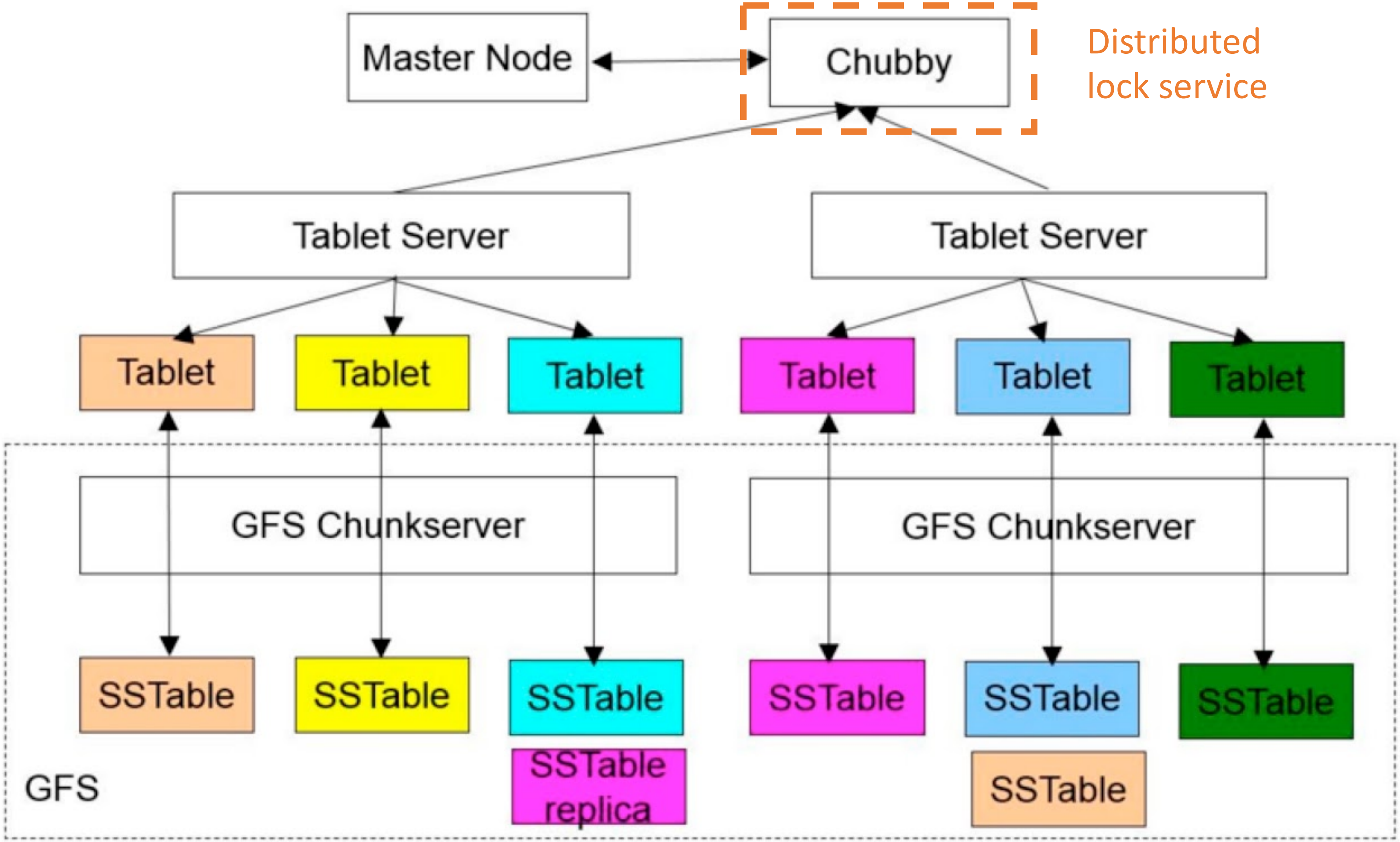
**Consensus Problem**

A set of distributed nodes need to reach agreement on a single value

# Google Bigtable



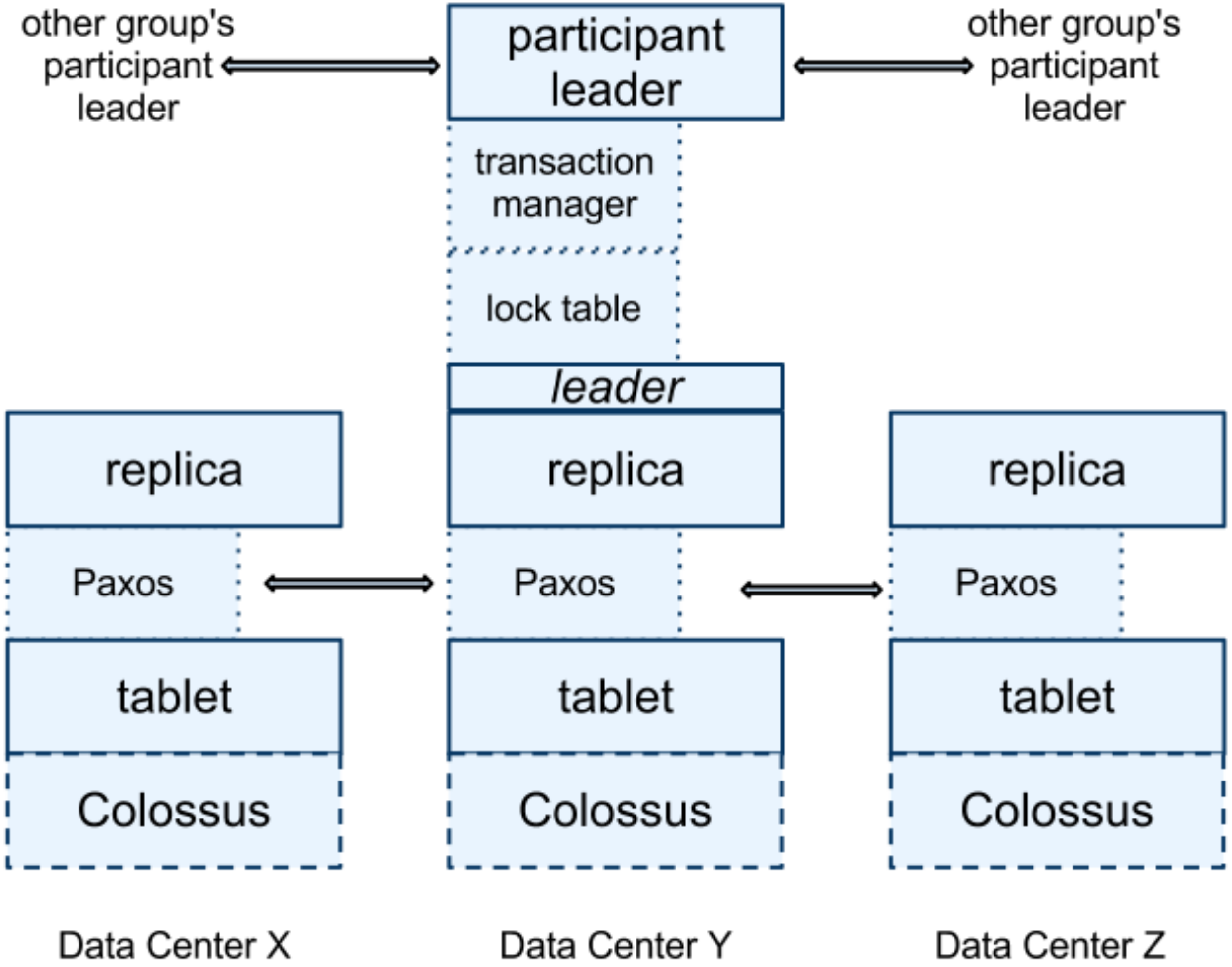
# Google Bigtable



Distributed lock service

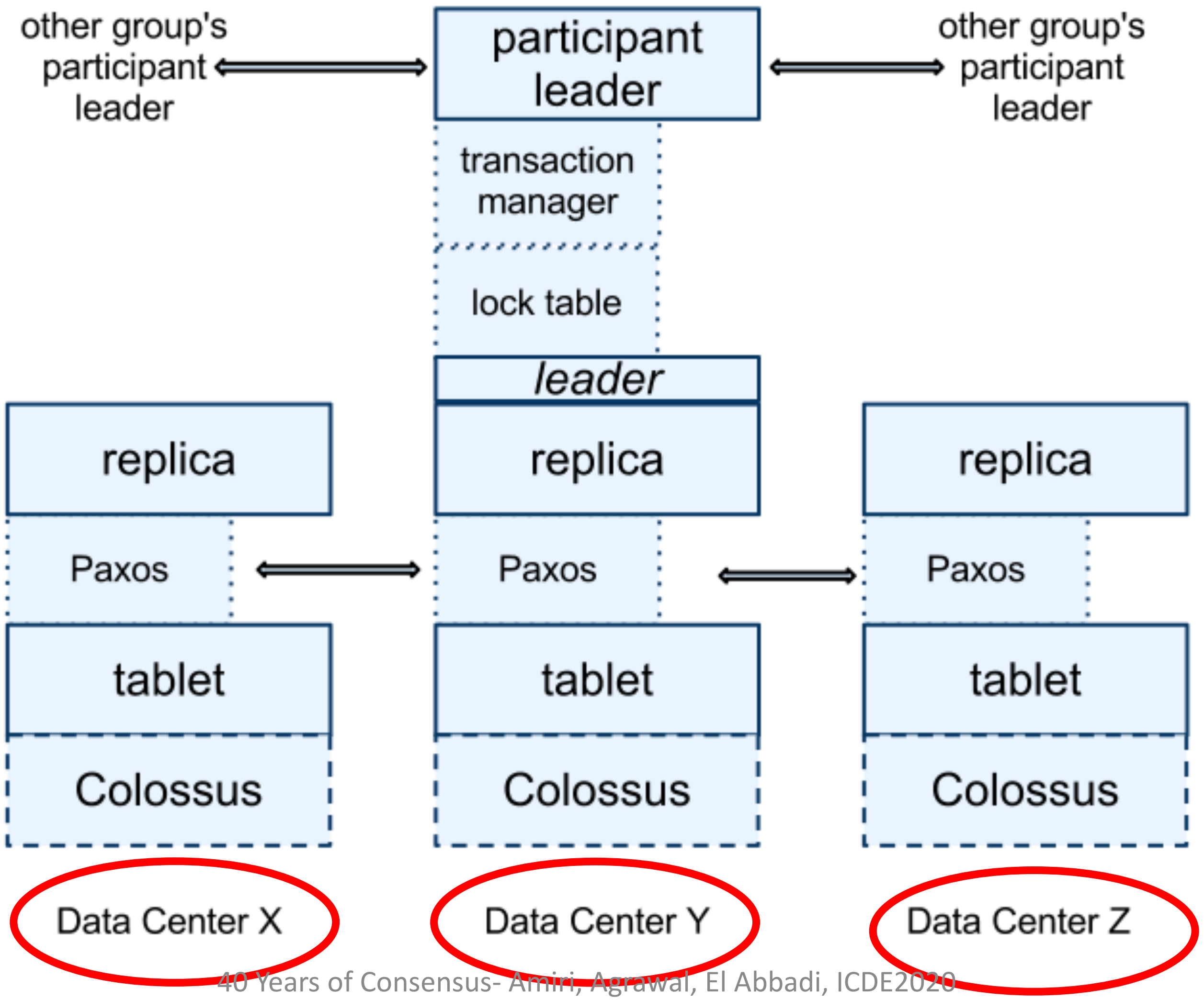
# Google Spanner

Google Cloud Spanner



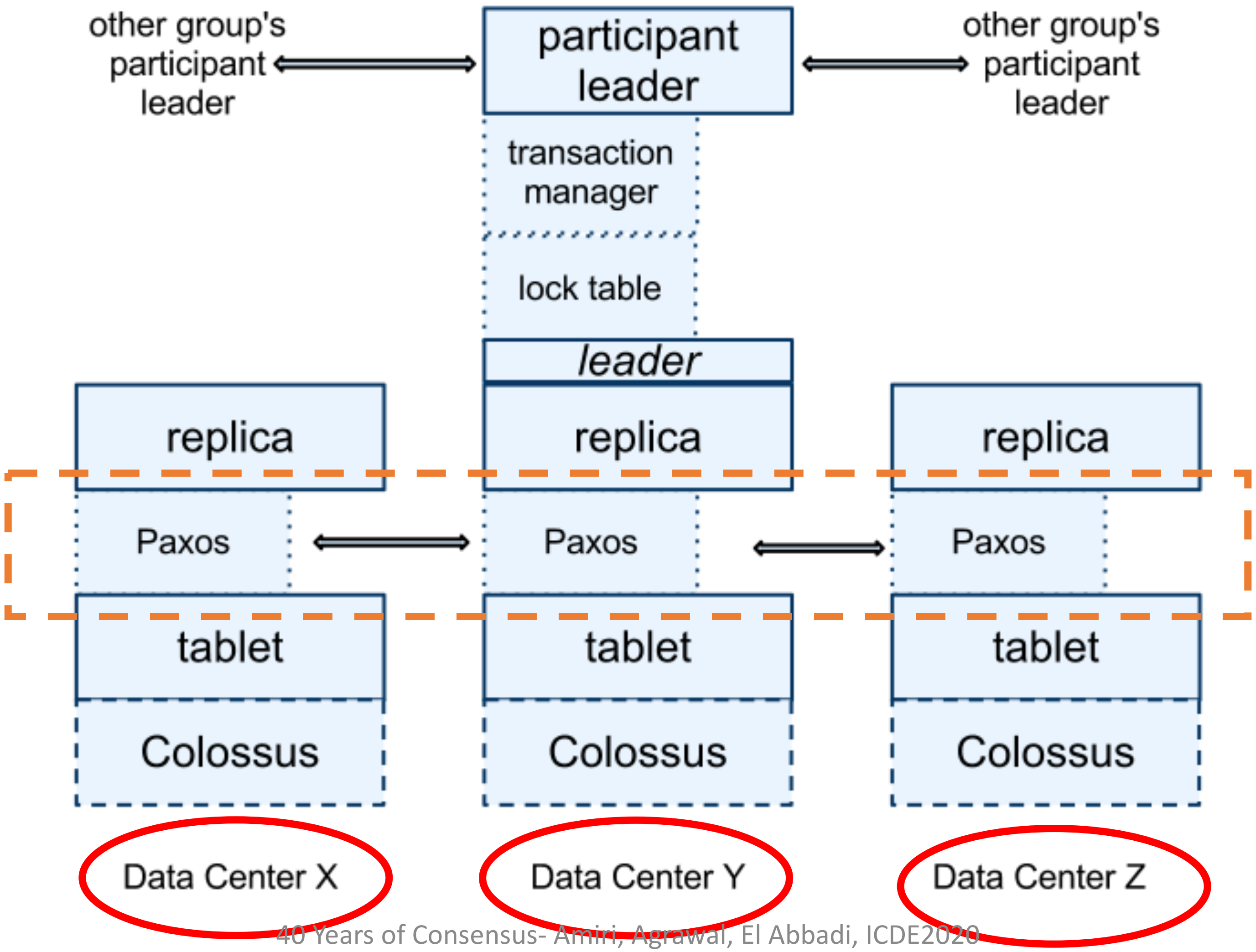
# Google Spanner

Google Cloud Spanner

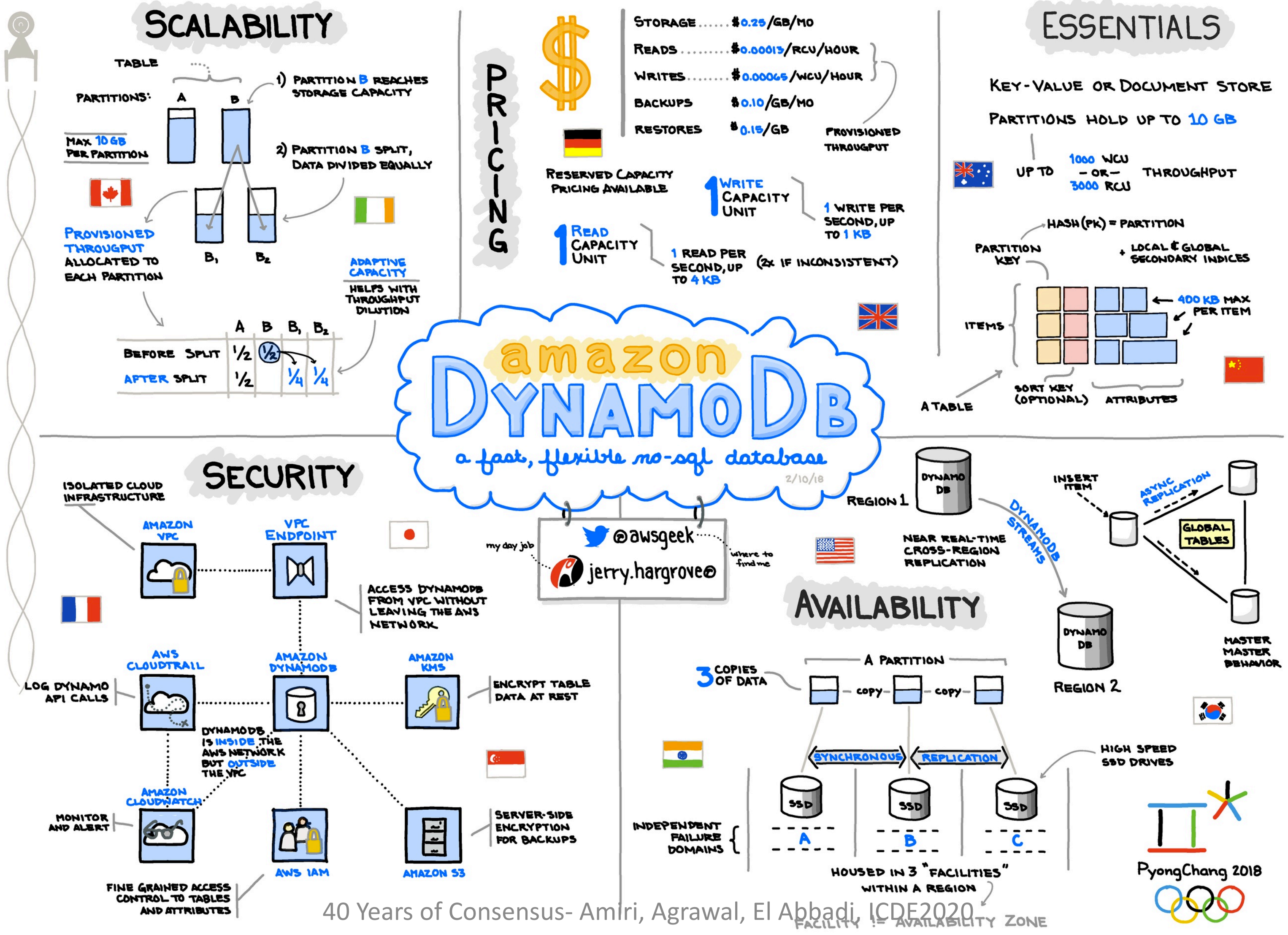


# Google Spanner

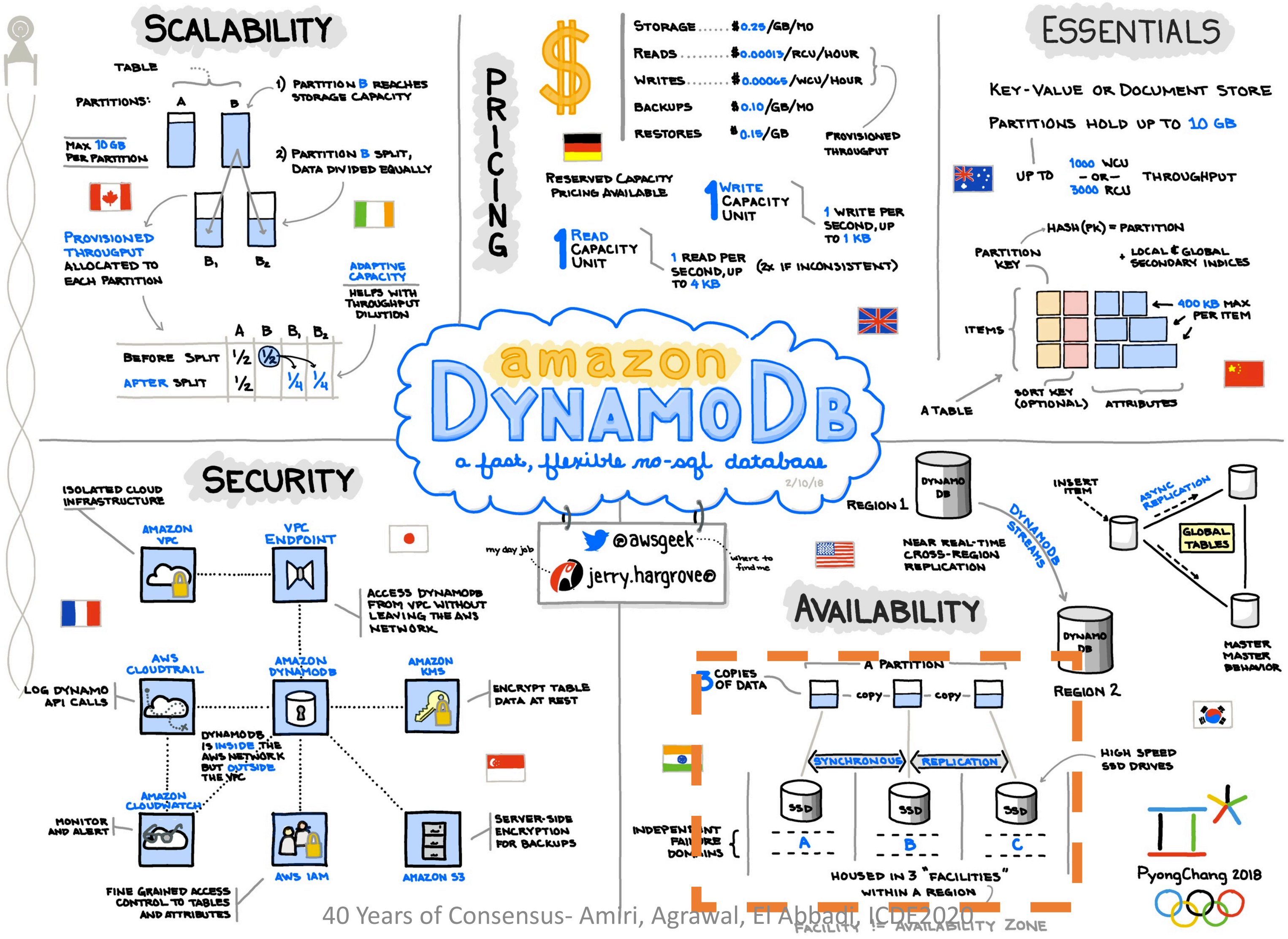
Google Cloud Spanner



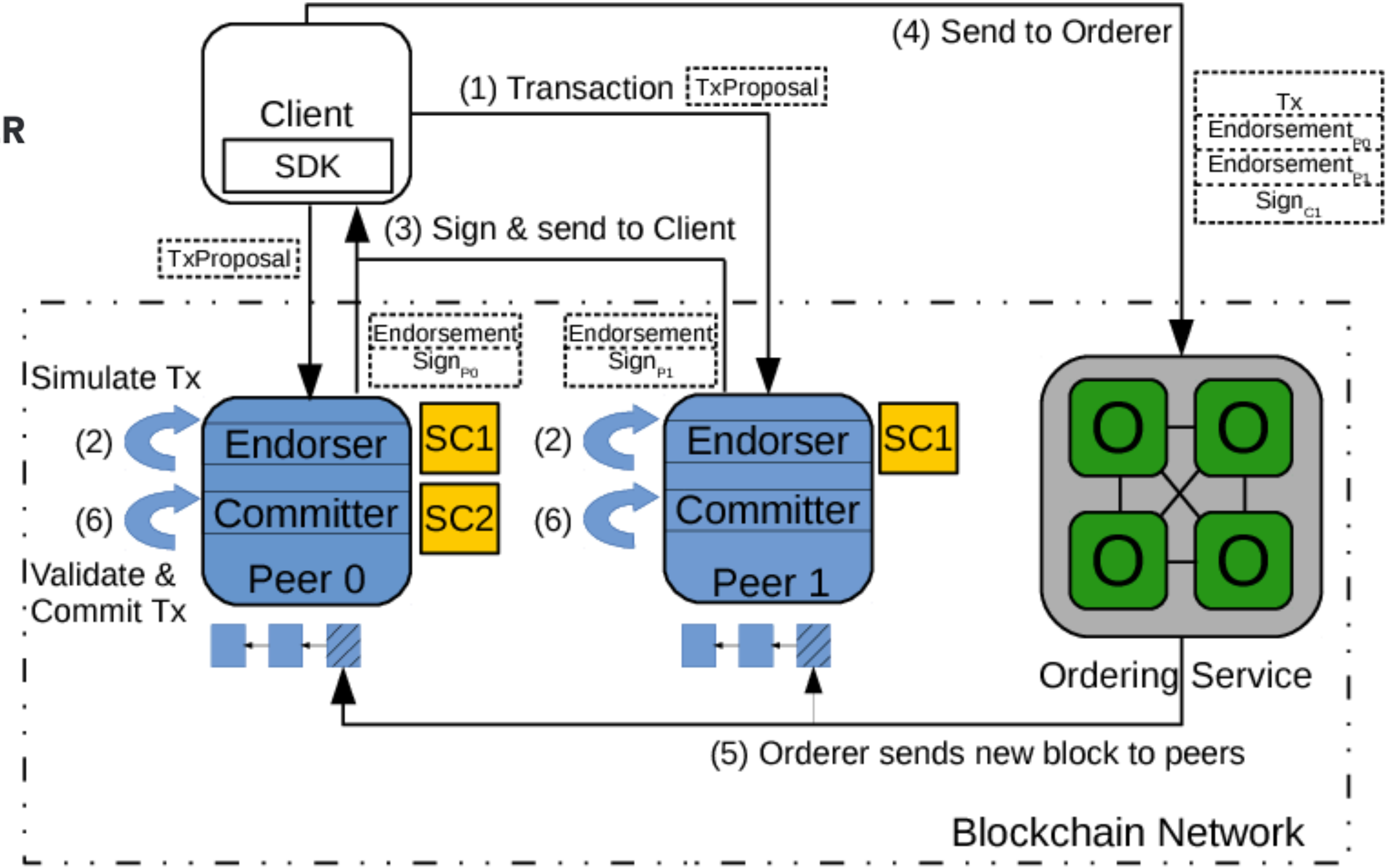
# Amazon DynamoDB



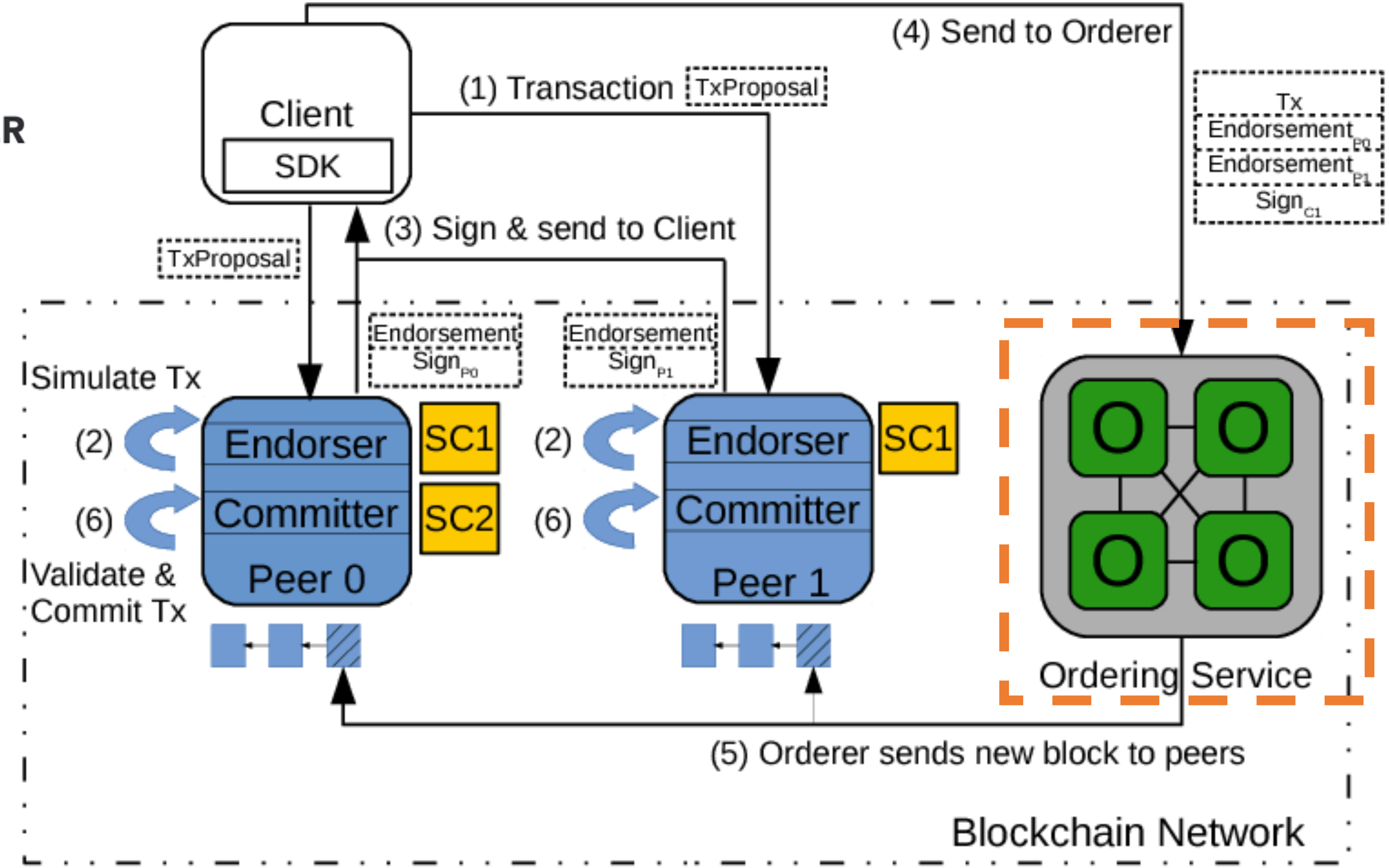
# Amazon DynamoDB



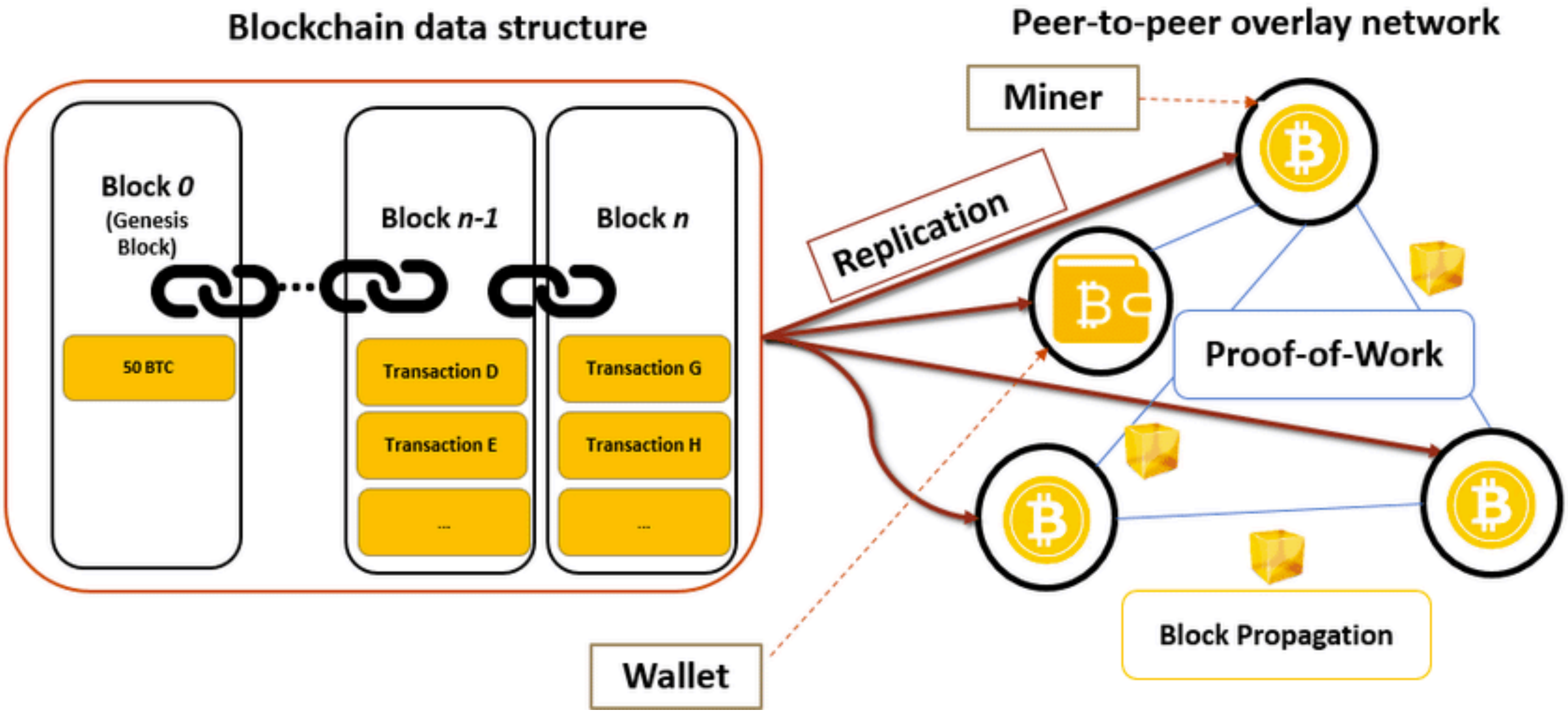
# Hyperledger Fabric (Permissioned Blockchain)



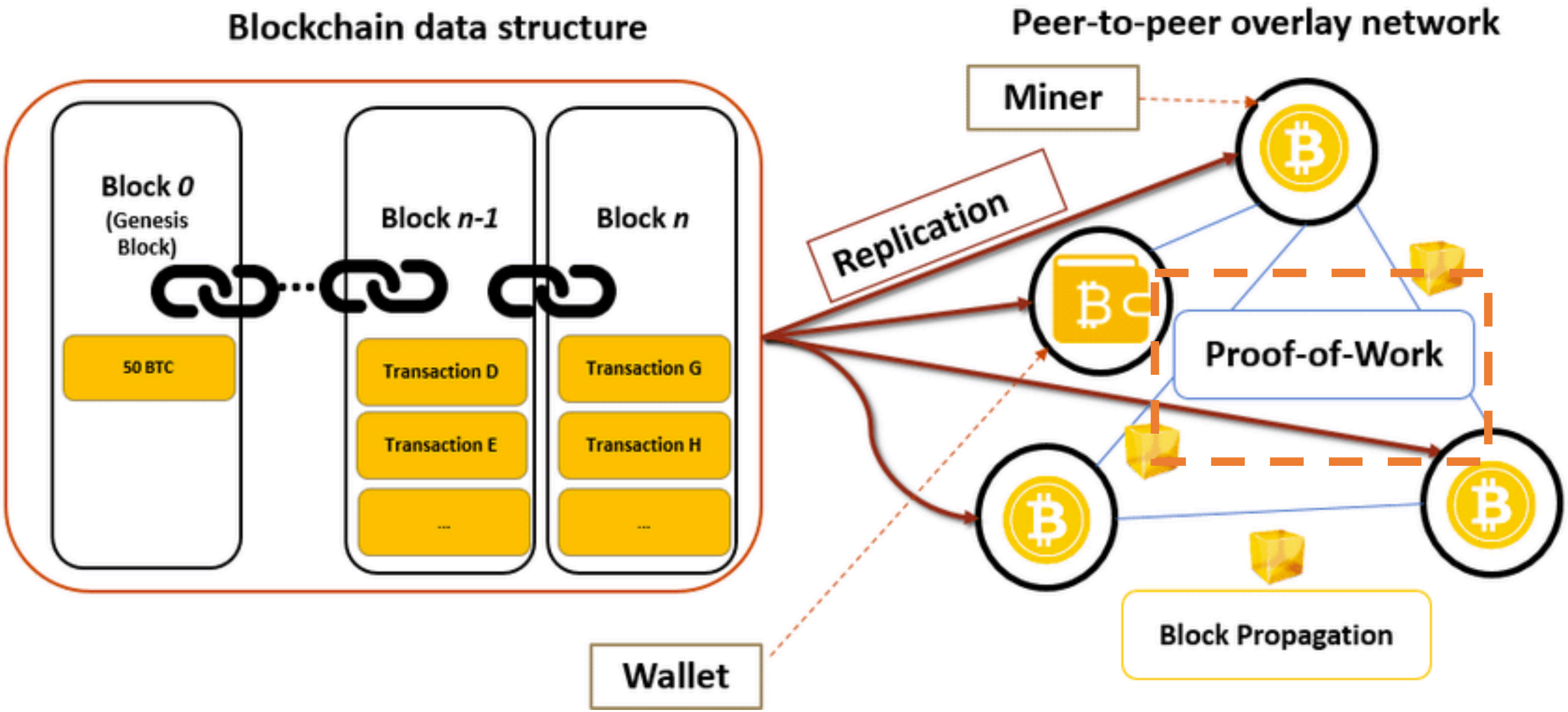
# Hyperledger Fabric (Permissioned Blockchain)



# Bitcoin (Permissionless Blockchain)



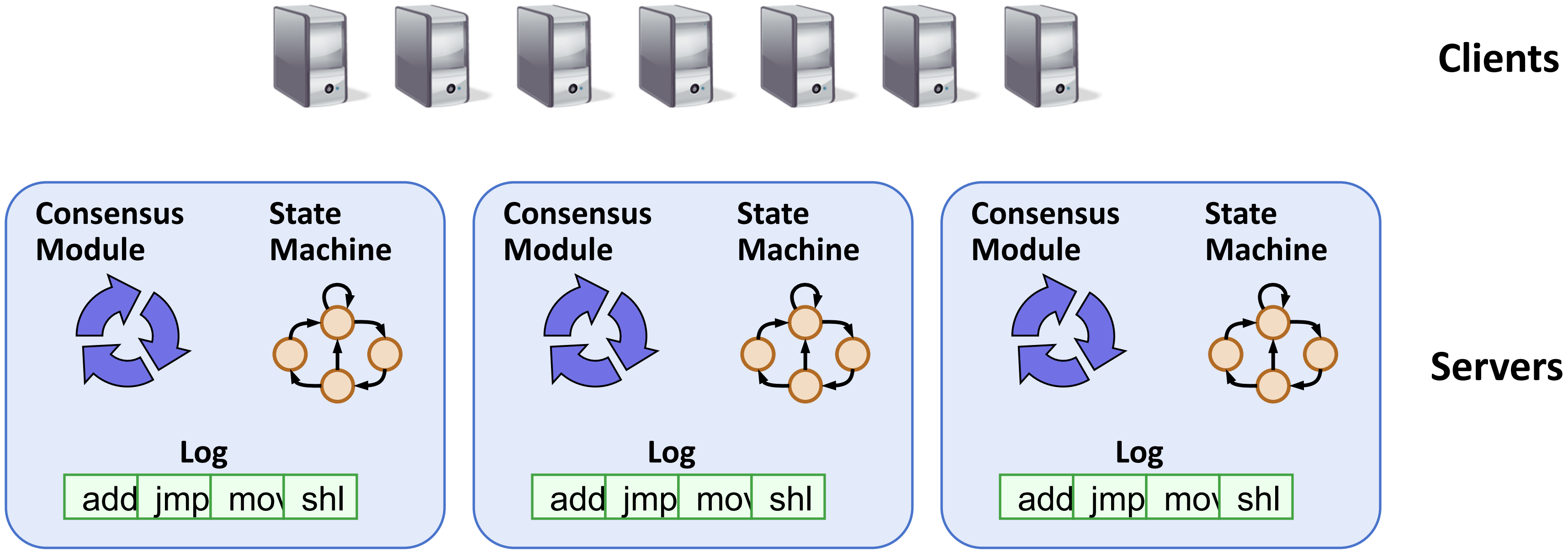
# Bitcoin (Permissionless Blockchain)



# State Machine Replication

**Clients**

# State Machine Replication



**Clients**

**Consensus Module** **State Machine**

**Log**

| add | jmp | mov | shl |
|---|---|---|---|

**Consensus Module** **State Machine**

**Log**

| add | jmp | mov | shl |
|---|---|---|---|

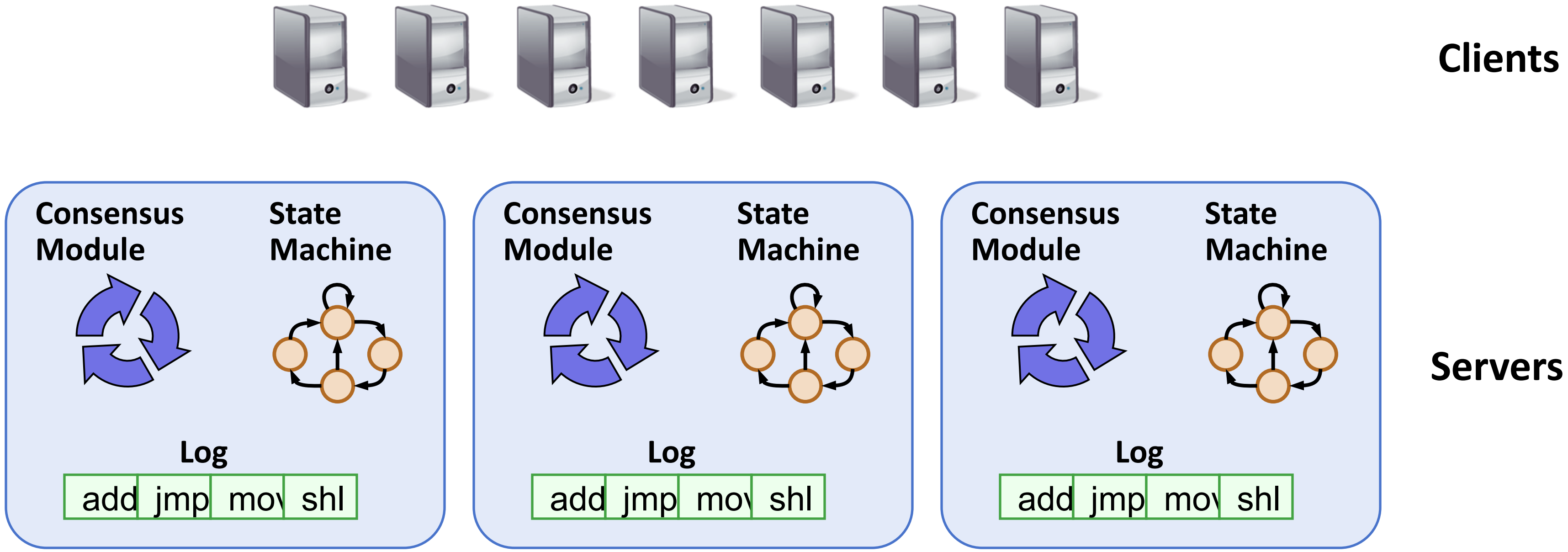
**Consensus Module** **State Machine**

**Log**

| add | jmp | mov | shl |
|---|---|---|---|

**Servers**

# State Machine Replication



Clients

Consensus Module

State Machine

Log

| add | jmp | mov | shl |
|---|---|---|---|

Consensus Module

State Machine

Log

| add | jmp | mov | shl |
|---|---|---|---|

Consensus Module

State Machine

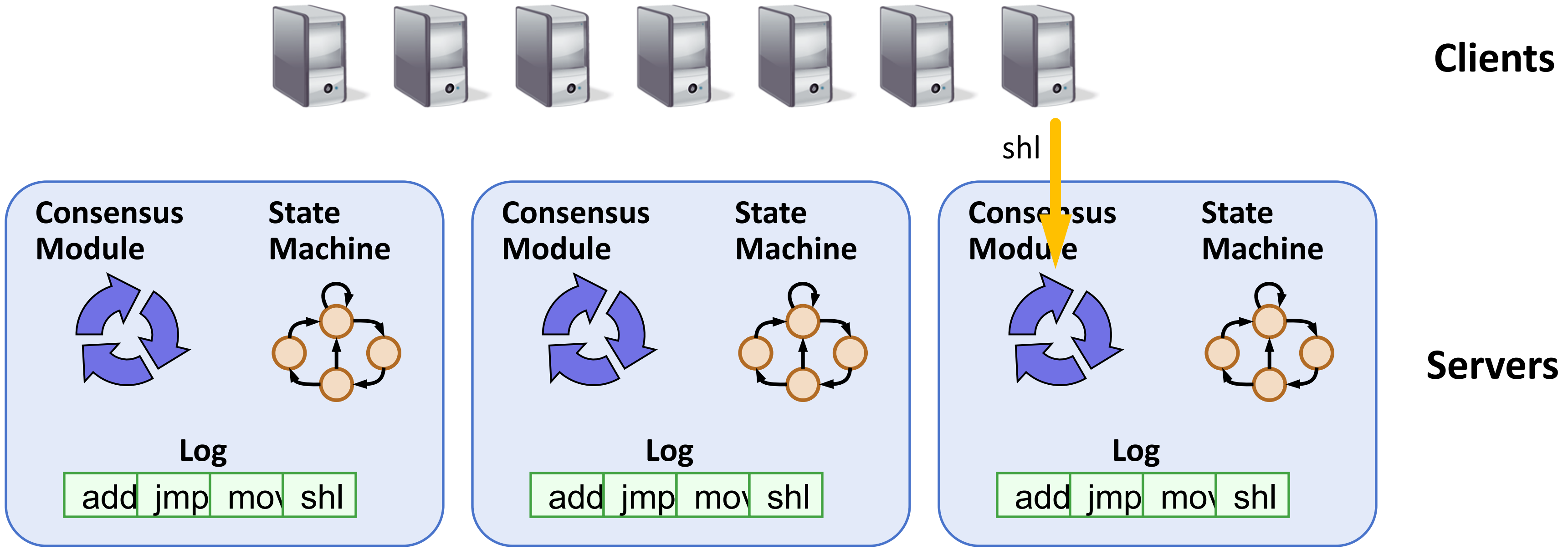
Log

| add | jmp | mov | shl |
|---|---|---|---|

Servers

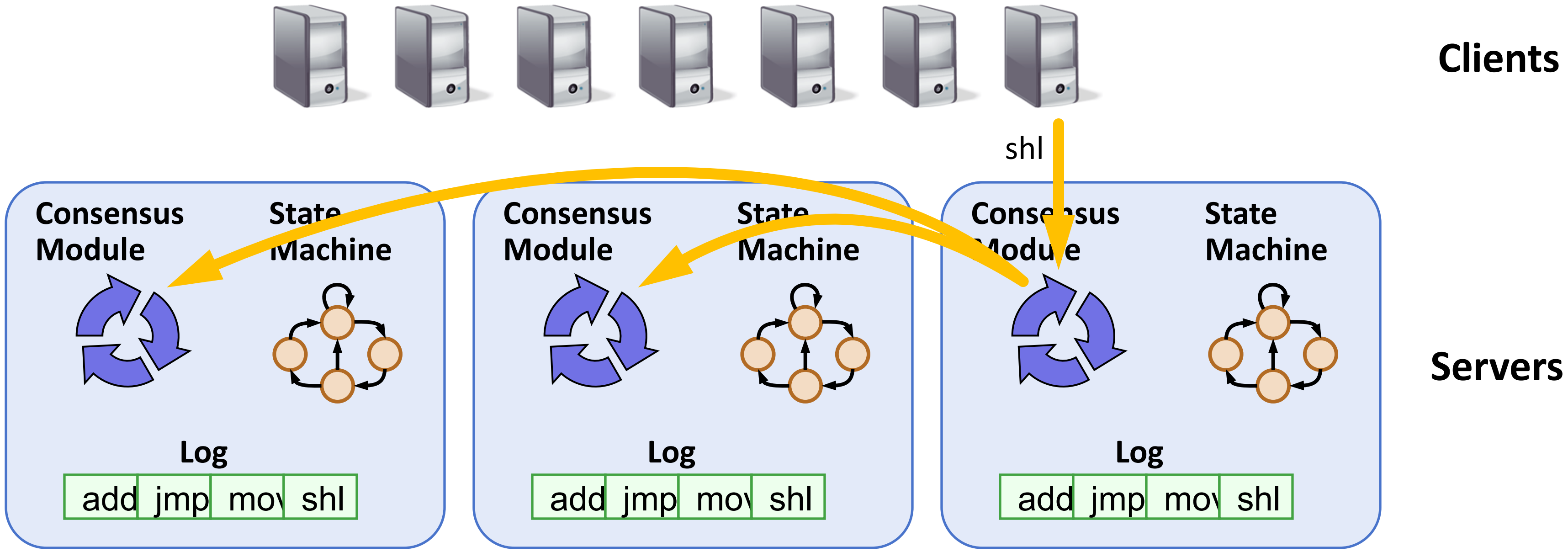
- Replicated log: replicated state machine
  - All servers execute same commands in the same order
  - Commands are deterministic
- Consensus module ensures proper log replication

# State Machine Replication



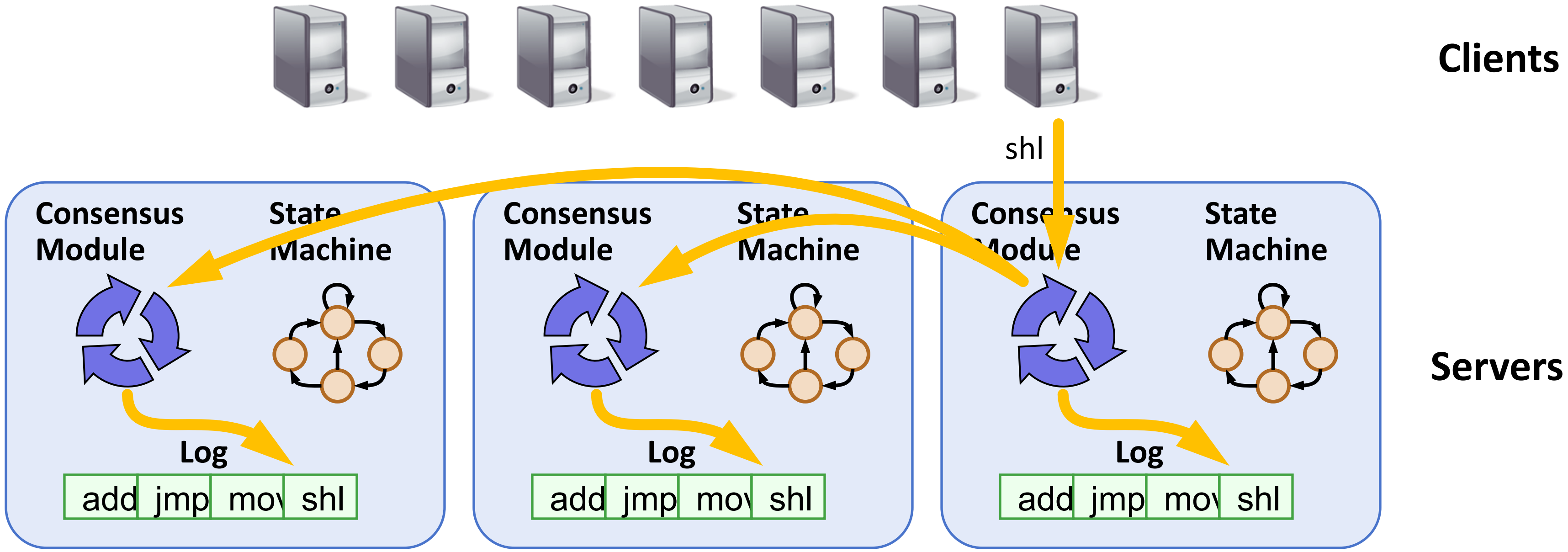
- Replicated log: replicated state machine
  - All servers execute same commands in the same order
  - Commands are deterministic
- Consensus module ensures proper log replication

# State Machine Replication



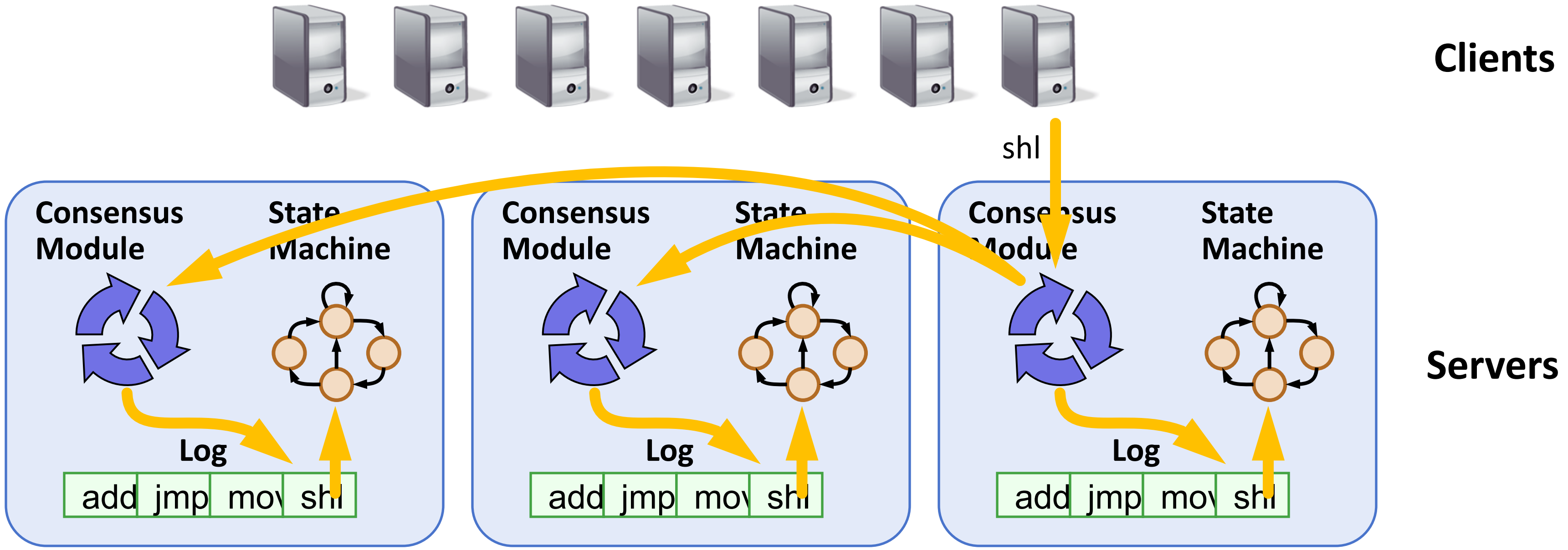
- Replicated log: replicated state machine
  - All servers execute same commands in the same order
  - Commands are deterministic
- Consensus module ensures proper log replication

# State Machine Replication



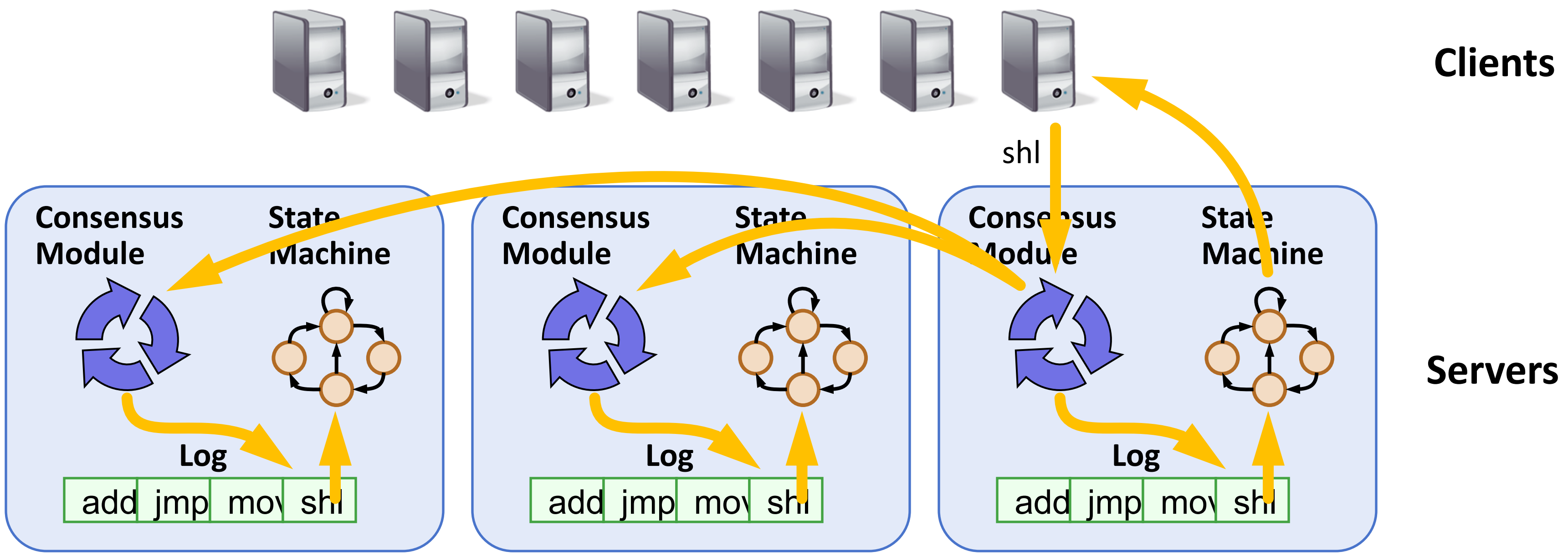
- Replicated log: replicated state machine
  - All servers execute same commands in the same order
  - Commands are deterministic
- Consensus module ensures proper log replication

# State Machine Replication



- Replicated log: replicated state machine
  - All servers execute same commands in the same order
  - Commands are deterministic
- Consensus module ensures proper log replication

# State Machine Replication

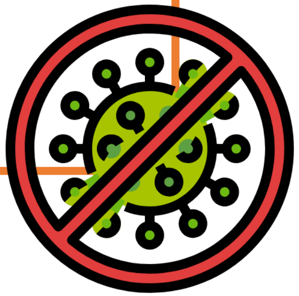


- Replicated log: replicated state machine
  - All servers execute same commands in the same order
  - Commands are deterministic
- Consensus module ensures proper log replication

# Properties

**Safety (bad things never happen)**

- Only a value that has been proposed may be chosen
- Only a single value is chosen
- A node never learns that a value has been chosen unless it has been



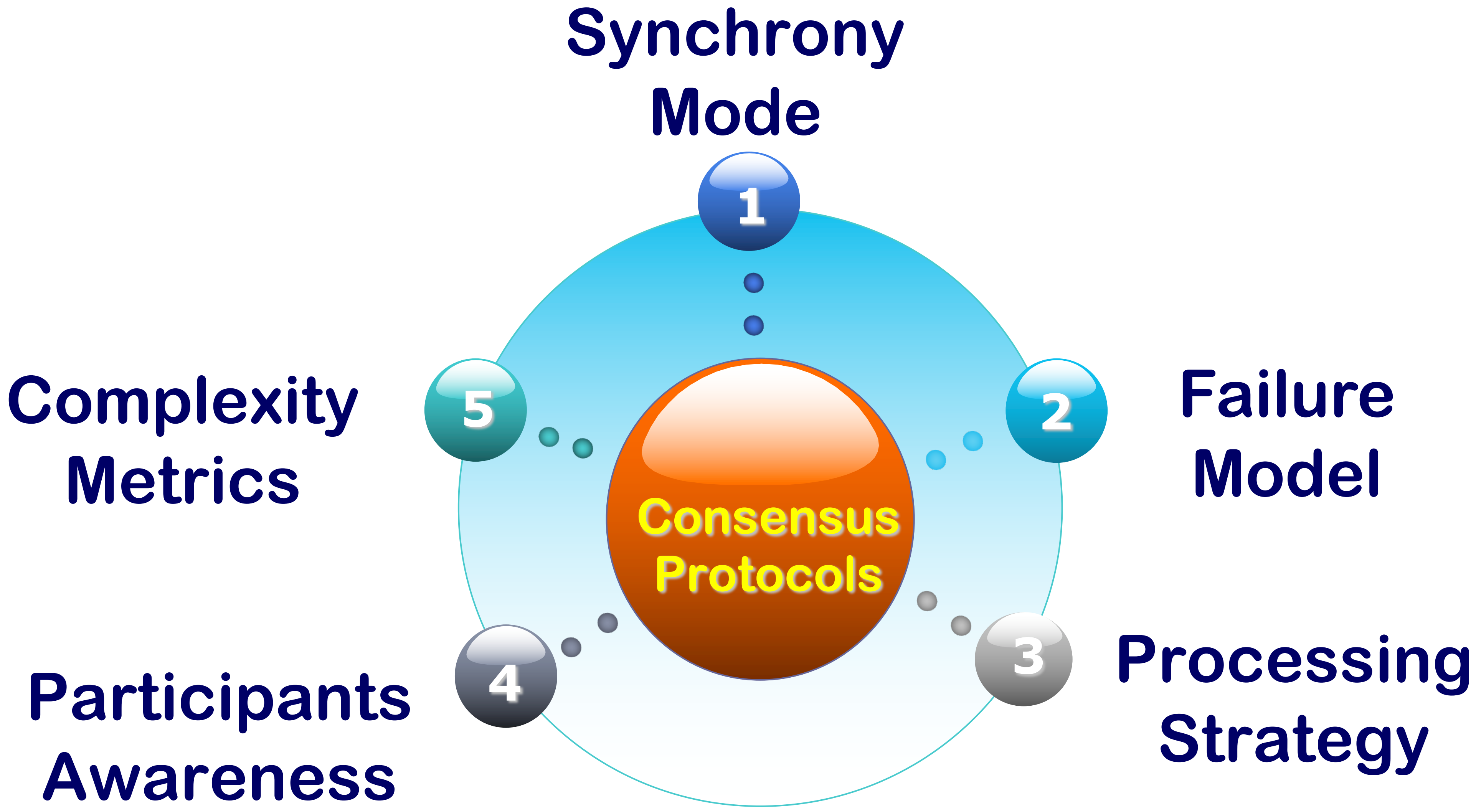
# Properties

**Safety (bad things never happen)**

- Only a value that has been proposed may be chosen
- Only a single value is chosen
- A node never learns that a value has been chosen unless it has been

**Liveness (good things eventually happen)**

- Some proposed value is eventually chosen
- If a value has been chosen, a node can eventually learn the value



Synchrony Mode

1

Failure Model

2

Processing Strategy

3

Participants Awareness

4

Complexity Metrics

5

Consensus Protocols

# First Aspect: Synchrony Mode

**Synchronous System**

- Assume known bounds on message delays and process speeds
- All communication proceeds in rounds.
- In one round, a process may send all the messages it requires, while receiving all messages from others
- No message from one round may influence any messages sent within the same round.

# First Aspect: Synchrony Mode

## Synchronous System

- Assume known bounds on message delays and process speeds
- All communication proceeds in rounds.
- In one round, a process may send all the messages it requires, while receiving all messages from others
- No message from one round may influence any messages sent within the same round.

## Asynchronous System

- There are no bounds on the amount of time a node might take
- There is no global clock nor consistent clock rate
- Each node processes independently of others
- Coordination is achieved via events such as message arrival

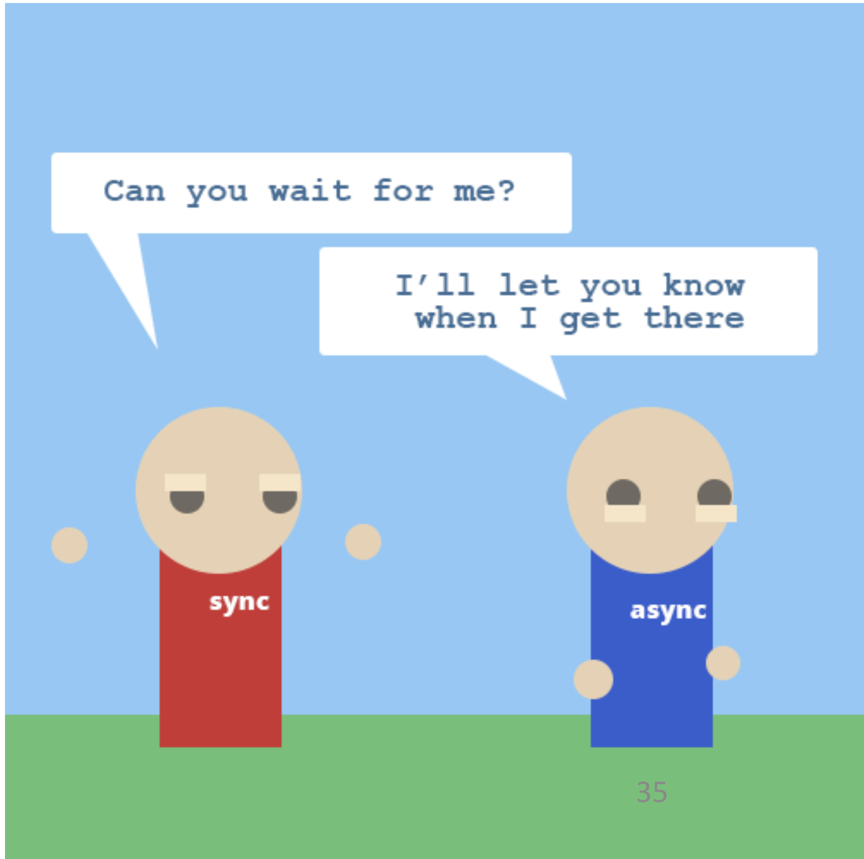
# First Aspect: Synchrony Mode

## Synchronous System

- Assume known bounds on message delays and process speeds
- All communication proceeds in rounds.
- In one round, a process may send all the messages it requires, while receiving all messages from others
- No message from one round may influence any messages sent within the same round.

## Asynchronous System

- There are no bounds on the amount of time a node might take
- There is no global clock nor consistent clock rate
- Each node processes independently of others
- Coordination is achieved via events such as message arrival



# First Aspect: Synchrony Mode

## Synchronous System

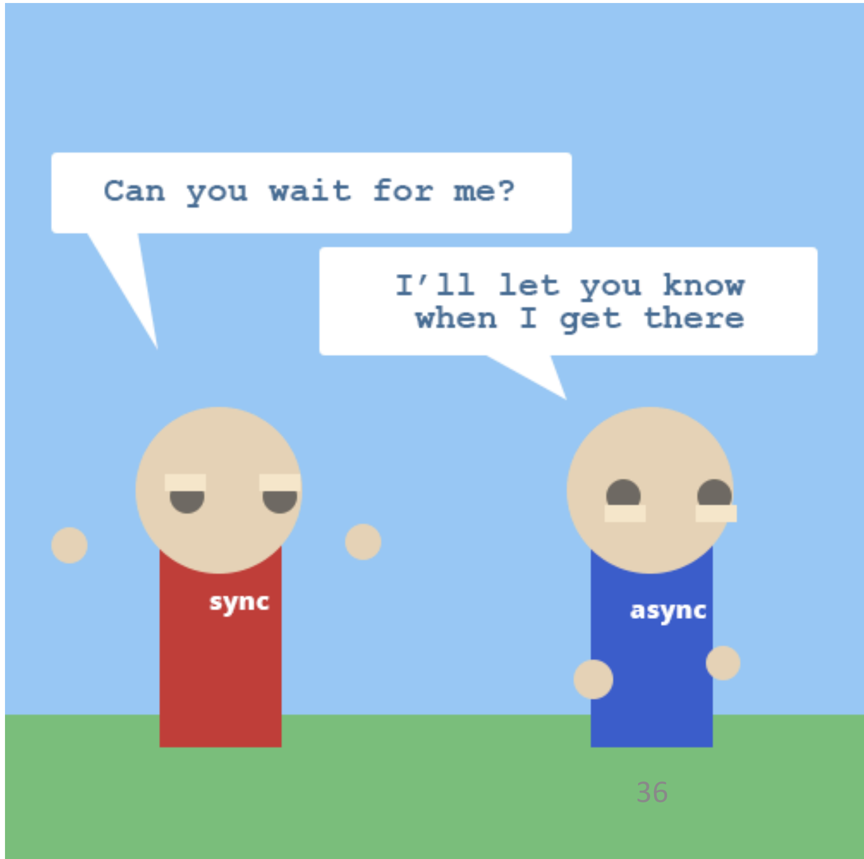
- Assume known bounds on message delays and process speeds
- All communication proceeds in rounds.
- In one round, a process may send all the messages it requires, while receiving all messages from others
- No message from one round may influence any messages sent within the same round.

## Asynchronous System

- There are no bounds on the amount of time a node might take
- There is no global clock nor consistent clock rate
- Each node processes independently of others
- Coordination is achieved via events such as message arrival

## Partially-Synchronous System

- Assumes that among the nodes, there is a subset that can communicate in a timely manner
- Only a limited number of nodes are perceived as arbitrarily slow
- Reasonable in data centers which are more predictable and controllable than an open Internet environment.



# Second Aspect: Failure Model

# Second Aspect: Failure Model

**Crash Failure**

- Nodes operate at arbitrary speed
- May fail by stopping, and may restart
- May not collude, lie, or otherwise attempt to subvert the protocol.

# Second Aspect: Failure Model

**Crash Failure**

- Nodes operate at arbitrary speed
- May fail by stopping, and may restart
- May not collude, lie, or otherwise attempt to subvert the protocol.

**Byzantine Failure**

- Faulty nodes may exhibit arbitrary, potentially malicious, behavior

# Second Aspect: Failure Model

Crash Failure

- Nodes operate at arbitrary speed
- May fail by stopping, and may restart
- May not collude, lie, or otherwise attempt to subvert the protocol.

Byzantine Failure

- Faulty nodes may exhibit arbitrary, potentially malicious, behavior

Hybrid Failure

- Some nodes might crash whereas some nodes behave maliciously.

# Third Aspect: Processing Strategy

# Third Aspect: Processing Strategy

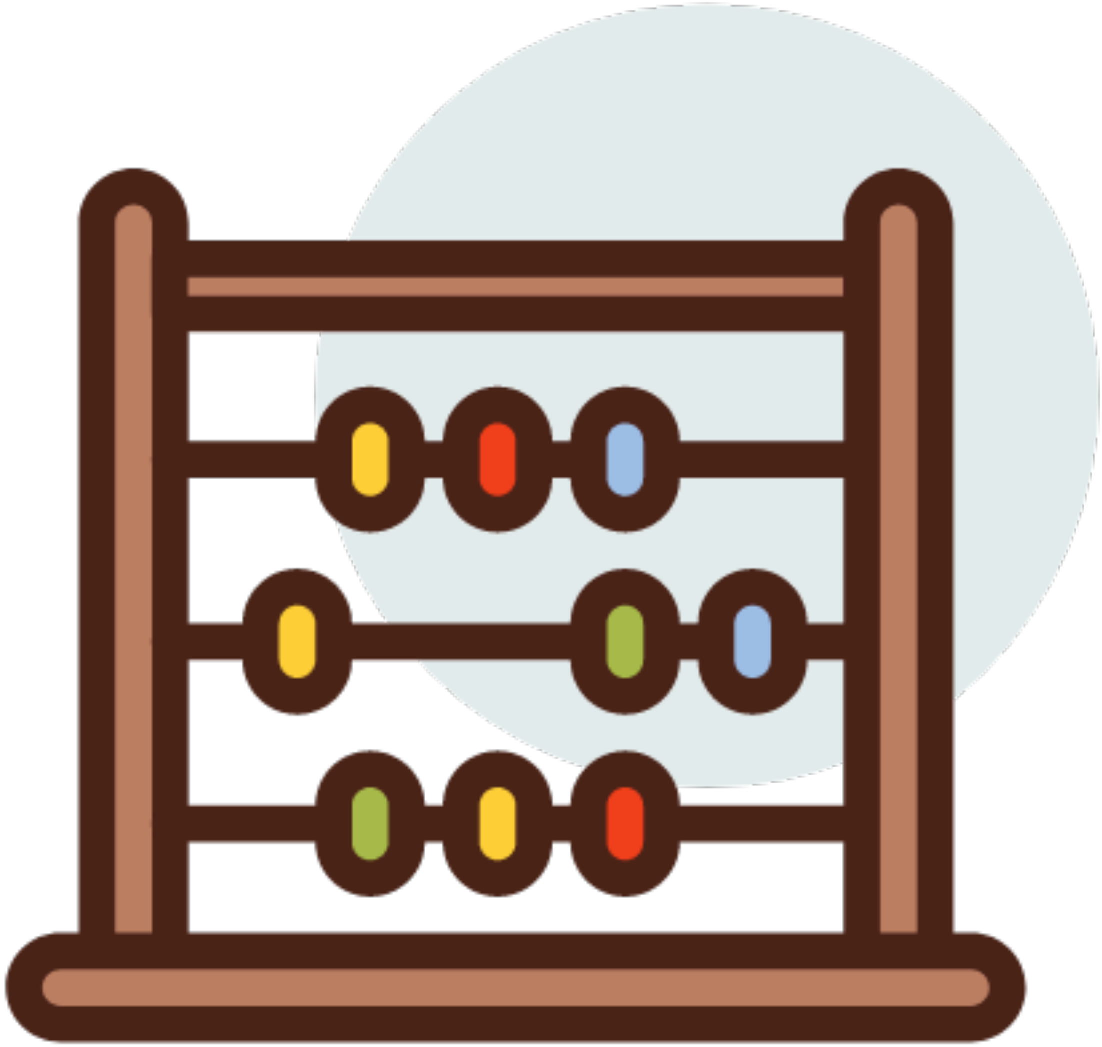
- **Pessimistic**
  - Guarantee from the beginning that all the replicas are identical to each other
  - Robust and designed to tolerate the maximum number of possible concurrent failures
- **Optimistic**
  - Replicas speculatively execute requests without running an agreement protocol to definitively establish the order
  - Replicas can diverge
  - Eventual consistency

# Fourth Aspect: Participant Awareness

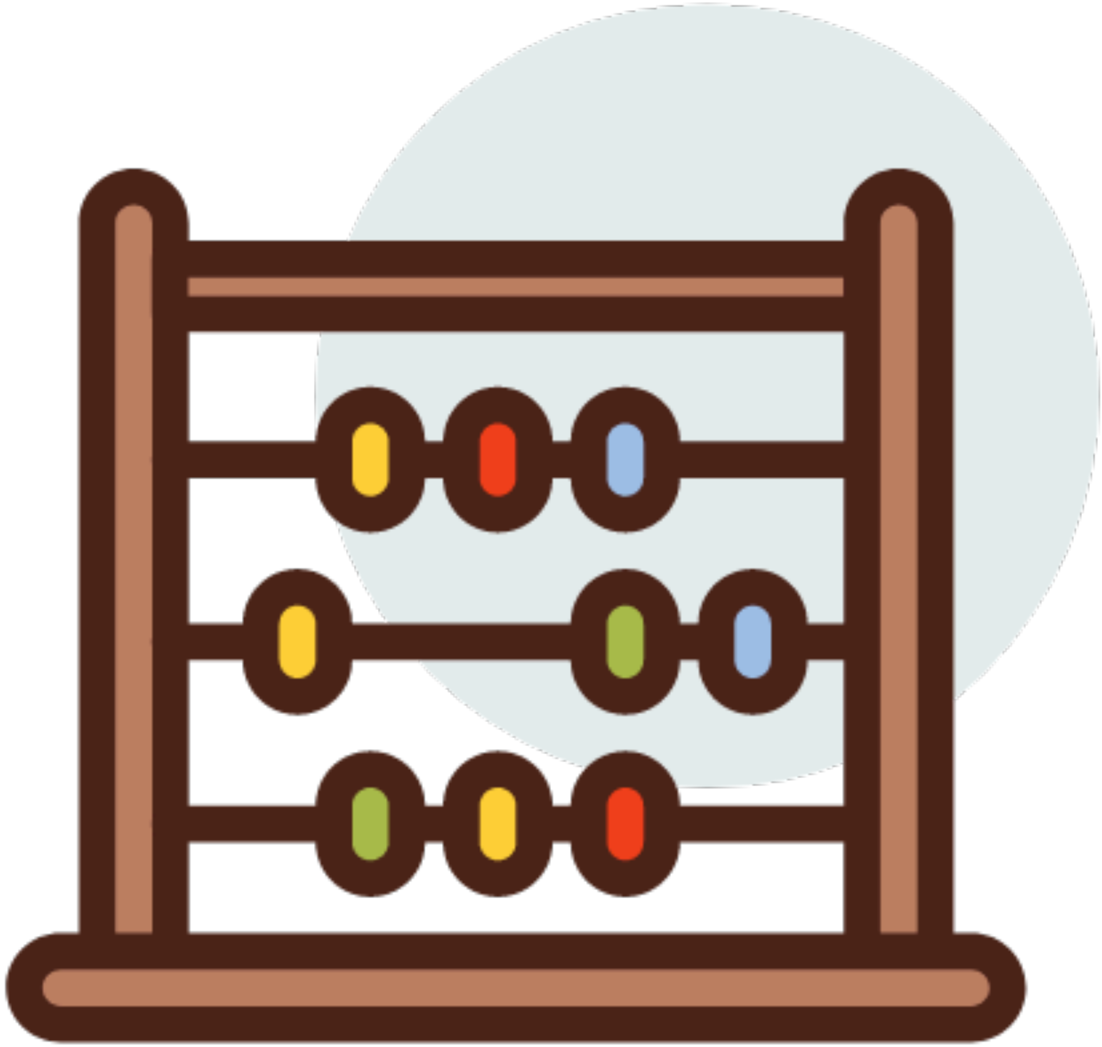
# Fourth Aspect: Participant Awareness

- **Known**
  - The participants are known and identified
  - Assume the maximum number of failures in the system is f
- **Unknown**
  - The set of participants is assumed to be unknown

# Fifth Aspect: Complexity Metrics



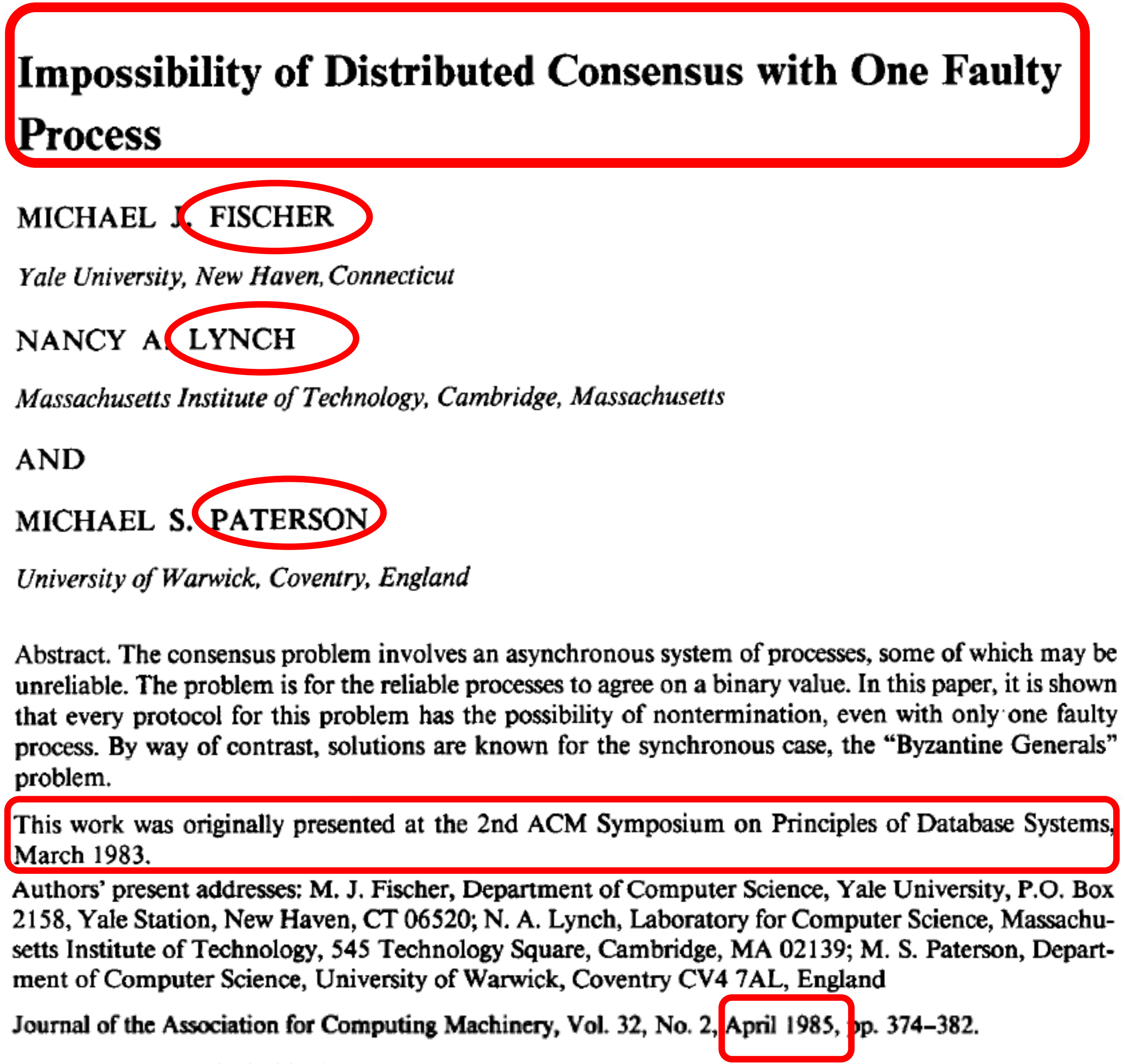
# Fifth Aspect: Complexity Metrics



- Number of nodes
- Number of communication phases
- Message complexity

# FLP Result

No deterministic 1-crash-robust consensus algorithm exists with asynchronous communication



## Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

*Yale University, New Haven, Connecticut*

NANCY A. LYNCH

*Massachusetts Institute of Technology, Cambridge, Massachusetts*

AND

MICHAEL S. PATERSON

*University of Warwick, Coventry, England*

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

This work was originally presented at the 2nd ACM Symposium on Principles of Database Systems, March 1983.

Authors' present addresses: M. J. Fischer, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, CT 06520; N. A. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; M. S. Paterson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, England

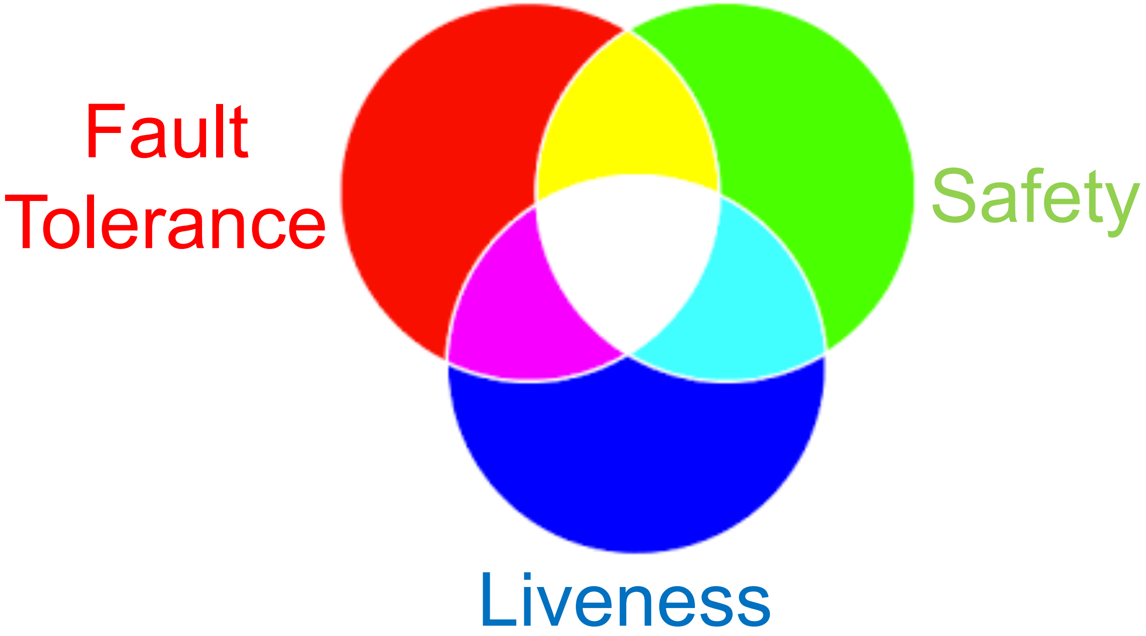
Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp. 374–382.

# FLP Result

- Impossibility of Distributed Consensus with ONE faulty Process

Asynchronous system; but reliable network

  - Process: Crash failures. Max ONE failure
  - Consensus problem: all non faulty processes agree on the same value {0, 1}.

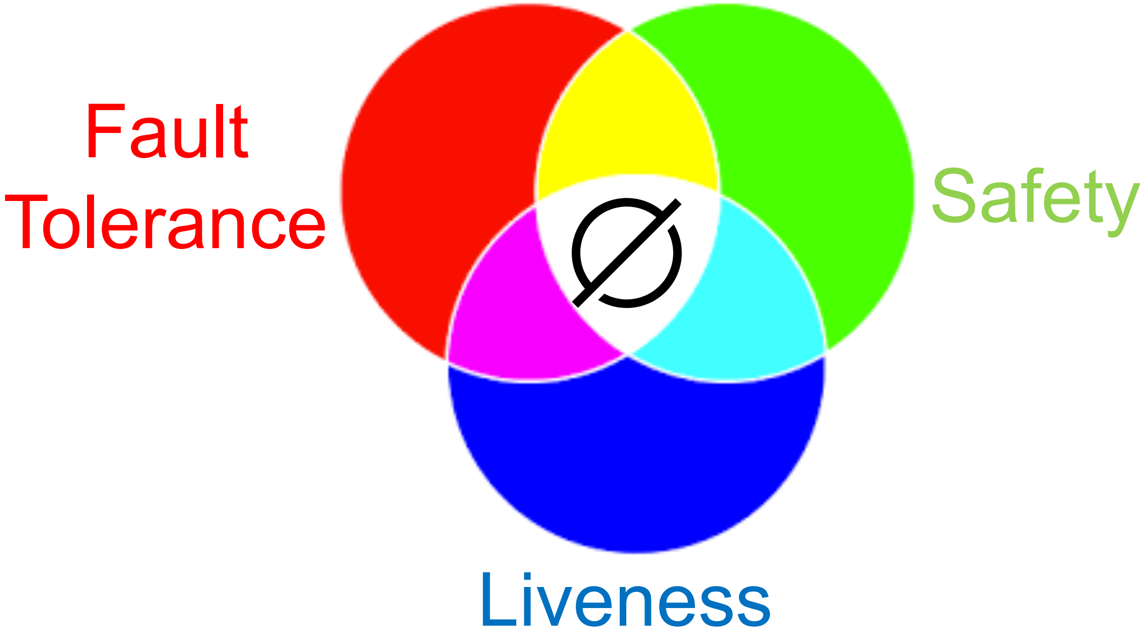


# FLP Result

- Impossibility of Distributed Consensus with ONE faulty Process

Asynchronous system; but reliable network

  - Process: Crash failures.  Max ONE failure
  - Consensus problem:  all non faulty processes agree on the same value {0, 1}.



# How to circumvent FLP result?

Sacrifice determinism

Randomized Byzantine consensus algorithm

Correia, M., Veronese, G. S., Neves, N. F., & Verissimo, P. Byzantine consensus in asynchronous message-passing systems: a survey. *IJCCBS, 2011*

# How to circumvent FLP result?

Sacrifice determinism

Randomized Byzantine consensus algorithm

Adding synchrony assumption

Define bound on message delay, etc.

Correia, M., Veronese, G. S., Neves, N. F., & Verissimo, P. Byzantine consensus in asynchronous message-passing systems: a survey. *IJCCBS, 2011*

# How to circumvent FLP result?

Sacrifice determinism

- Randomized Byzantine consensus algorithm

Adding synchrony assumption

- Define bound on message delay, etc.

Adding oracle (failure detector)

- Adding trusted component

Correia, M., Veronese, G. S., Neves, N. F., & Verissimo, P. Byzantine consensus in asynchronous message-passing systems: a survey. *IJCCBS, 2011*

# How to circumvent FLP result?

Sacrifice determinism

Randomized Byzantine consensus algorithm

Adding synchrony assumption

Define bound on message delay, etc.

Adding oracle (failure detector)

Adding trusted component

Change the problem domain

range of value or set of values

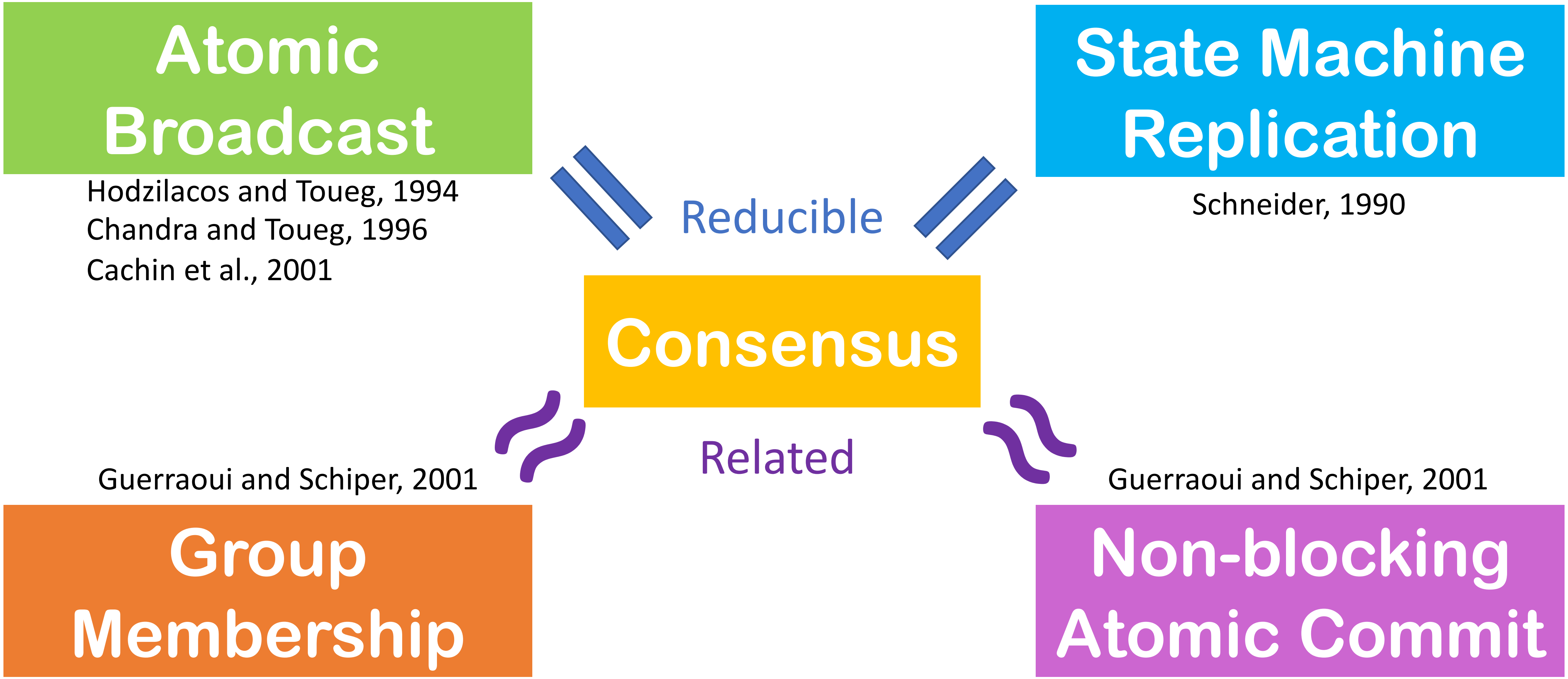
Correia, M., Veronese, G. S., Neves, N. F., & Verissimo, P. Byzantine consensus in asynchronous message-passing systems: a survey. *IJCCBS, 2011*

# Lower Bounds on Number of Processes

# Lower Bounds on Number of Processes

- [Pease, Shostak, Lamport 80] showed 3f+1 lower bound on number of processes for Byzantine Agreement.
- [Dolev 82] showed 2f+1 connectivity bound for BA.
- [Lamport 83] showed 3f+1 lower bound on number of processes for weak BA.
- [Coan, Dolev, Dwork, Stockmeyer 85] showed 3f+1 lower bound for Byzantine firing squad problem.
- [Dolev, Lynch, Pinter, Stark, Weihl 83] claimed 3f+1 bound for approximate BA.
- [Dolev, Halpern, Strong 84] showed 3f+1 lower bound for Byzantine clock synchronization.
- **Easy impossibility proofs for distributed consensus problems [Fischer, Lynch, Merritt PODC 85, DC 86]**

# Equivalent problems to Consensus



Atomic Broadcast

Hodzilacos and Toueg, 1994
Chandra and Toueg, 1996
Cachin et al., 2001

State Machine Replication

Schneider, 1990

Reducible

Consensus

Related

Guerraoui and Schiper, 2001

Group Membership

Guerraoui and Schiper, 2001

Non-blocking Atomic Commit

# PAXOS

Lamport, L. Paxos made simple. ACM Sigact News, 2001

| | | | | |
|---|---|---|---|---|
| Synchronous | [✓ Crash] | | | [✓ 2f+1 nodes] |
| [✓ Asynchronous] | Byzantine | [✓ Pessimistic] | [✓ Known nodes] | [✓ 2 phases] |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | [✓ O(N) Complexity] |

# Paxos Properties

- Paxos guarantees safety.
    - Consensus is a stable property: once reached it is never violated; the agreed value is not changed.

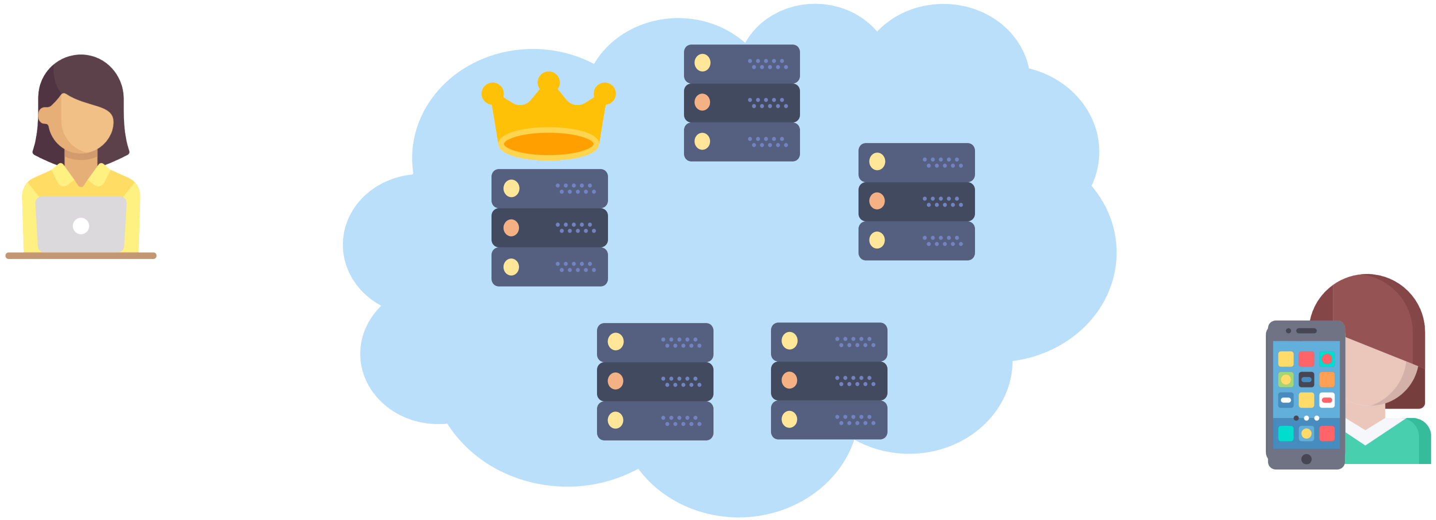
# Paxos Properties

- Paxos guarantees safety.
  - Consensus is a stable property: once reached it is never violated; the agreed value is not changed.

- Paxos does not guarantee liveness.
  - Consensus is reached if “a large enough subnetwork...is non-faulty for a long enough time.”
  - Otherwise Paxos might never terminate.

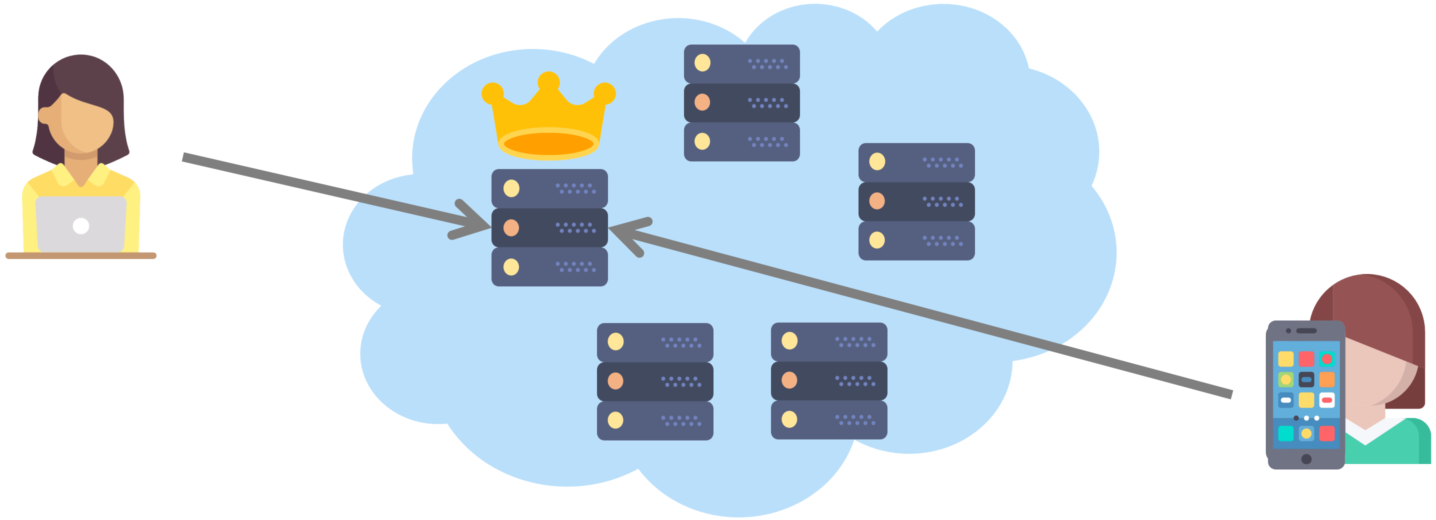
# Paxos Consensus Algorithm

- The clients send updates to the leader
- Leader orders the requests and ‘forwards’ to the replicas
- Leader waits to get acknowledgement of the updates
- Upon receiving ‘enough’ acks, leader sends decision asynchronously



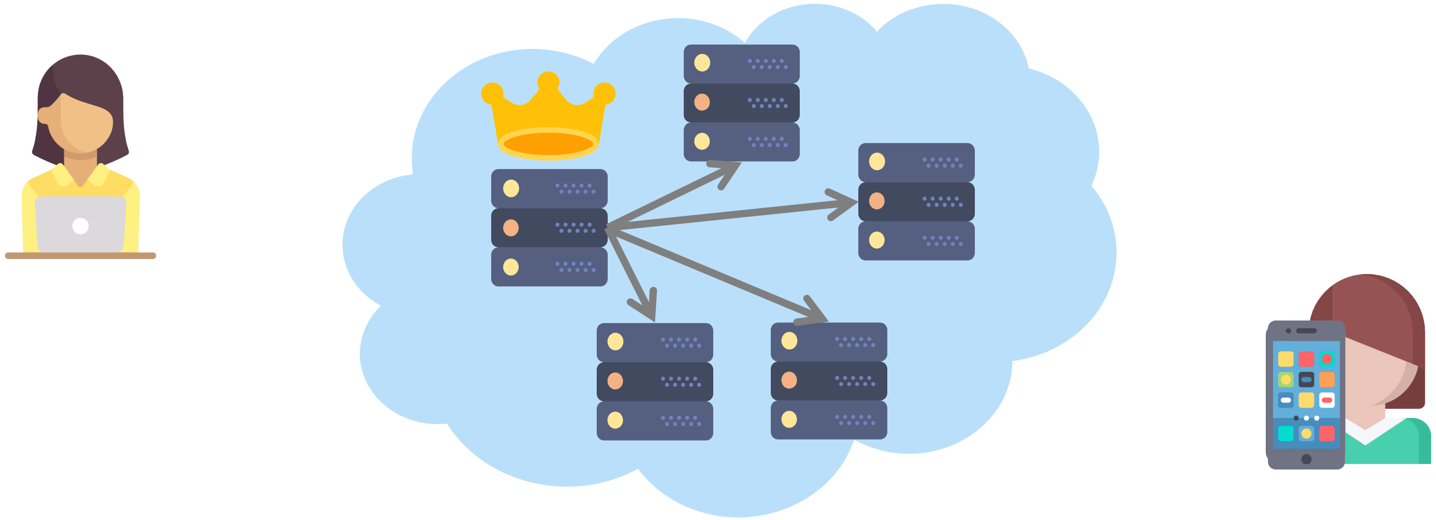
# Paxos Consensus Algorithm

- The clients send updates to the leader
- Leader orders the requests and ‘forwards’ to the replicas
- Leader waits to get acknowledgement of the updates
- Upon receiving ‘enough’ acks, leader sends decision asynchronously



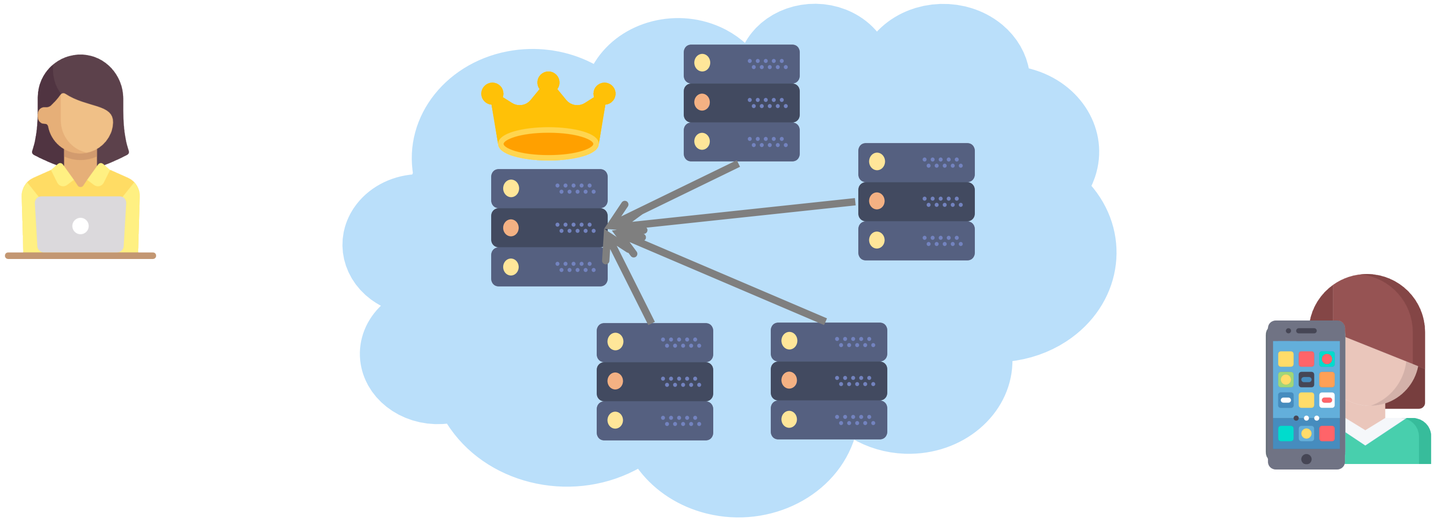
# Paxos Consensus Algorithm

- The clients send updates to the leader
- Leader orders the requests and ‘forwards’ to the replicas
- Leader waits to get acknowledgement of the updates
- Upon receiving ‘enough’ acks, leader sends decision asynchronously



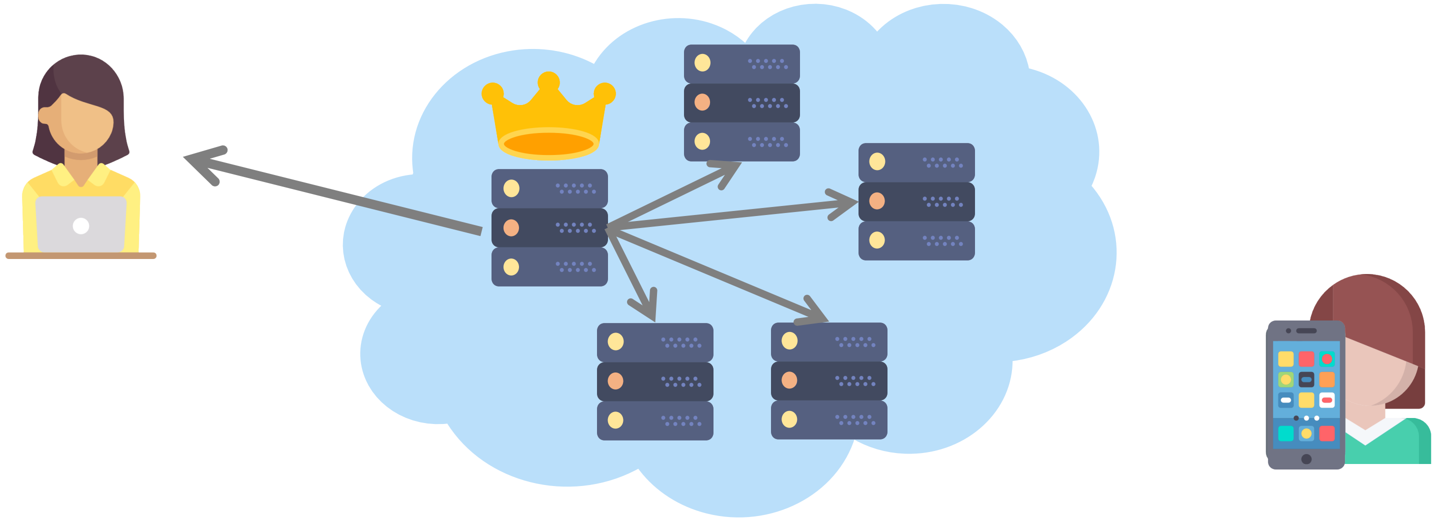
# Paxos Consensus Algorithm

- The clients send updates to the leader
- Leader orders the requests and ‘forwards’ to the replicas
- Leader waits to get acknowledgement of the updates
- Upon receiving ‘enough’ acks, leader sends decision asynchronously



# Paxos Consensus Algorithm

- The clients send updates to the leader
- Leader orders the requests and ‘forwards’ to the replicas
- Leader waits to get acknowledgement of the updates
- Upon receiving ‘enough’ acks, leader sends decision asynchronously

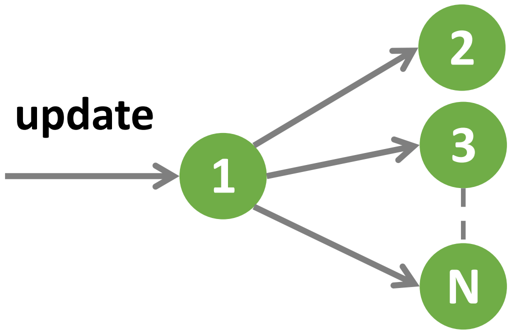


# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**

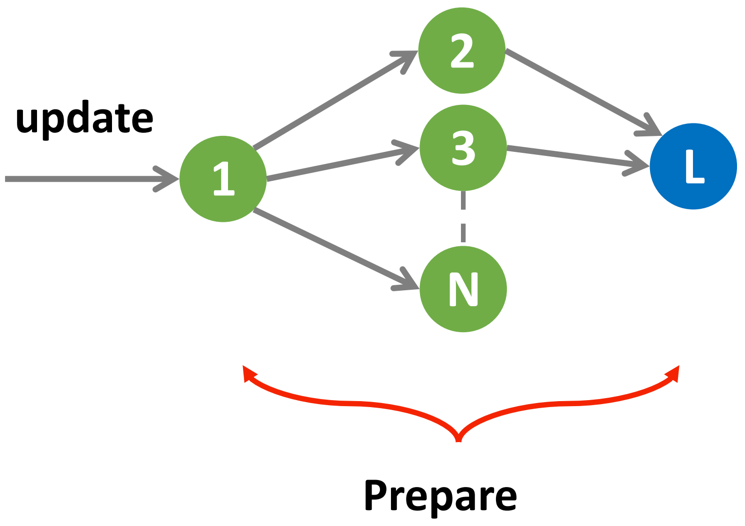
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



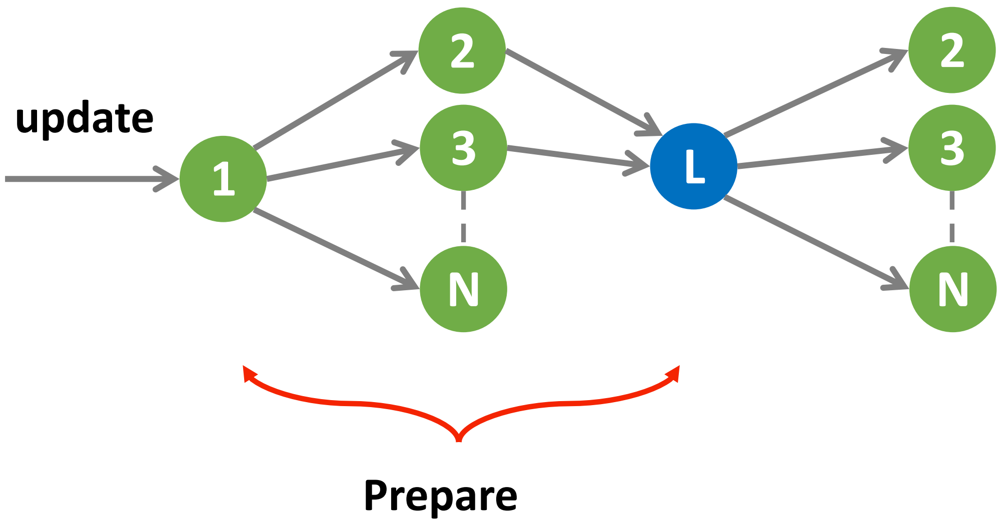
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



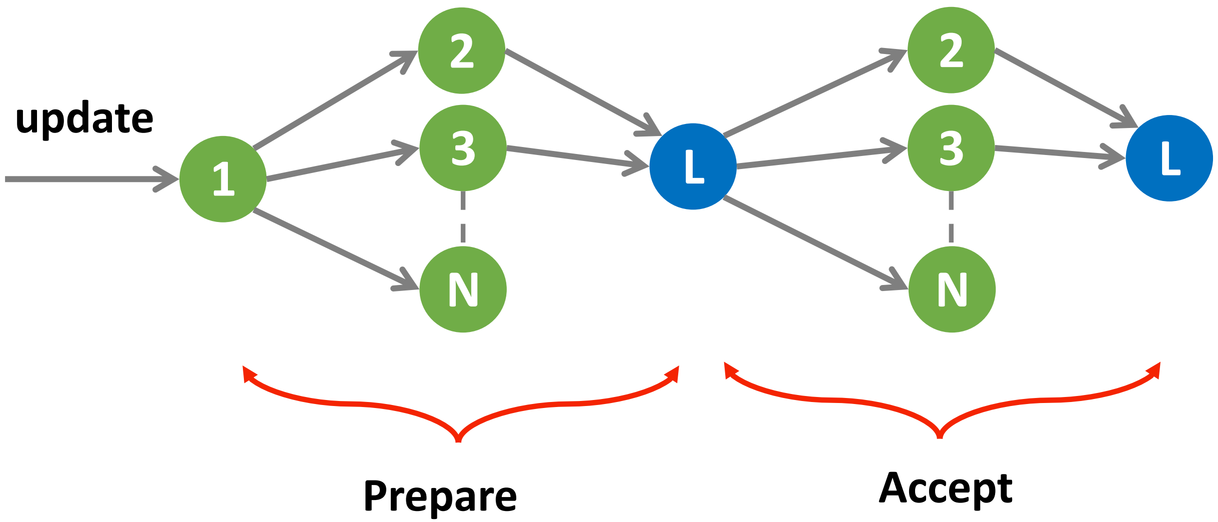
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



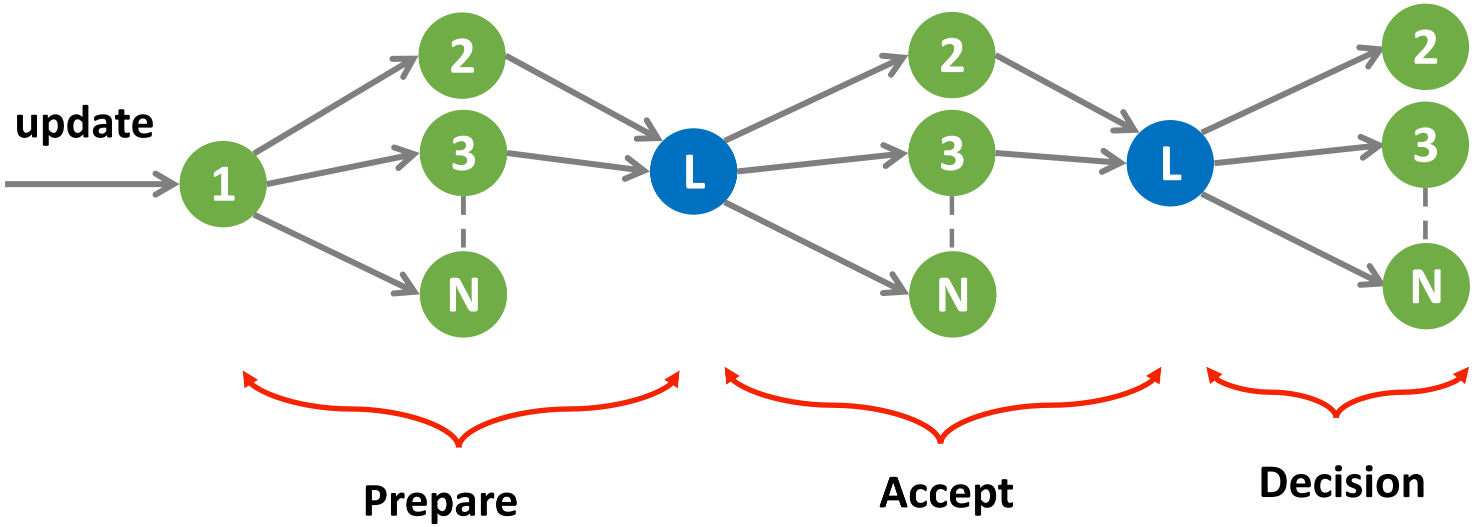
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



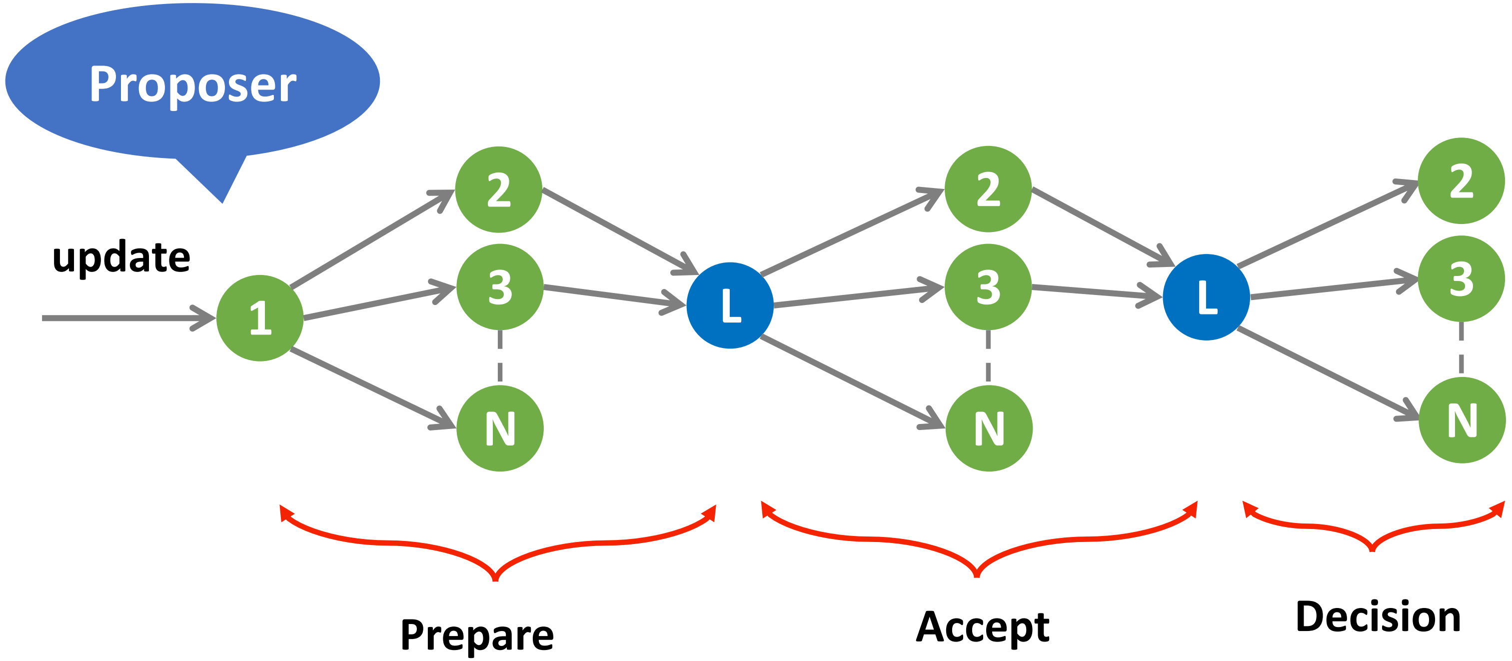
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



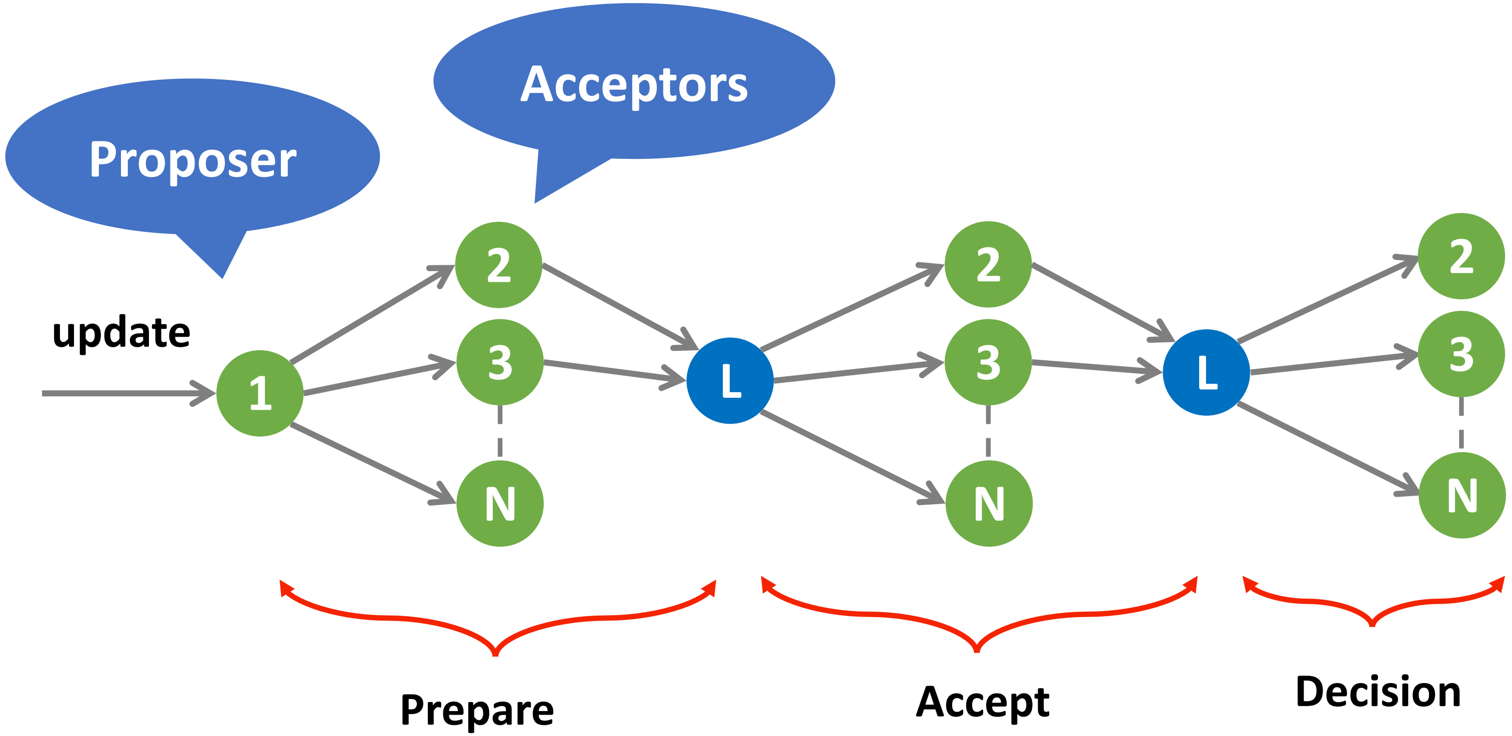
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



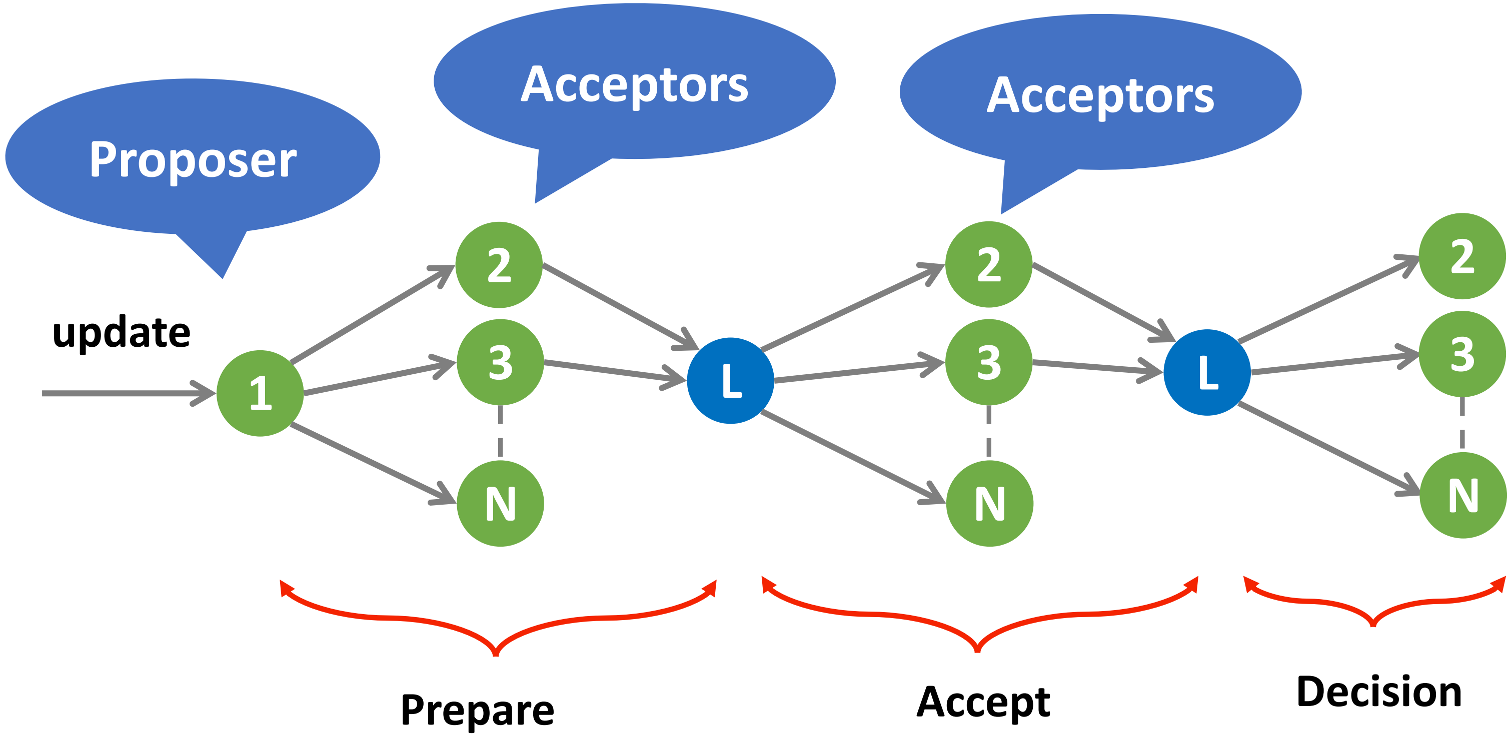
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



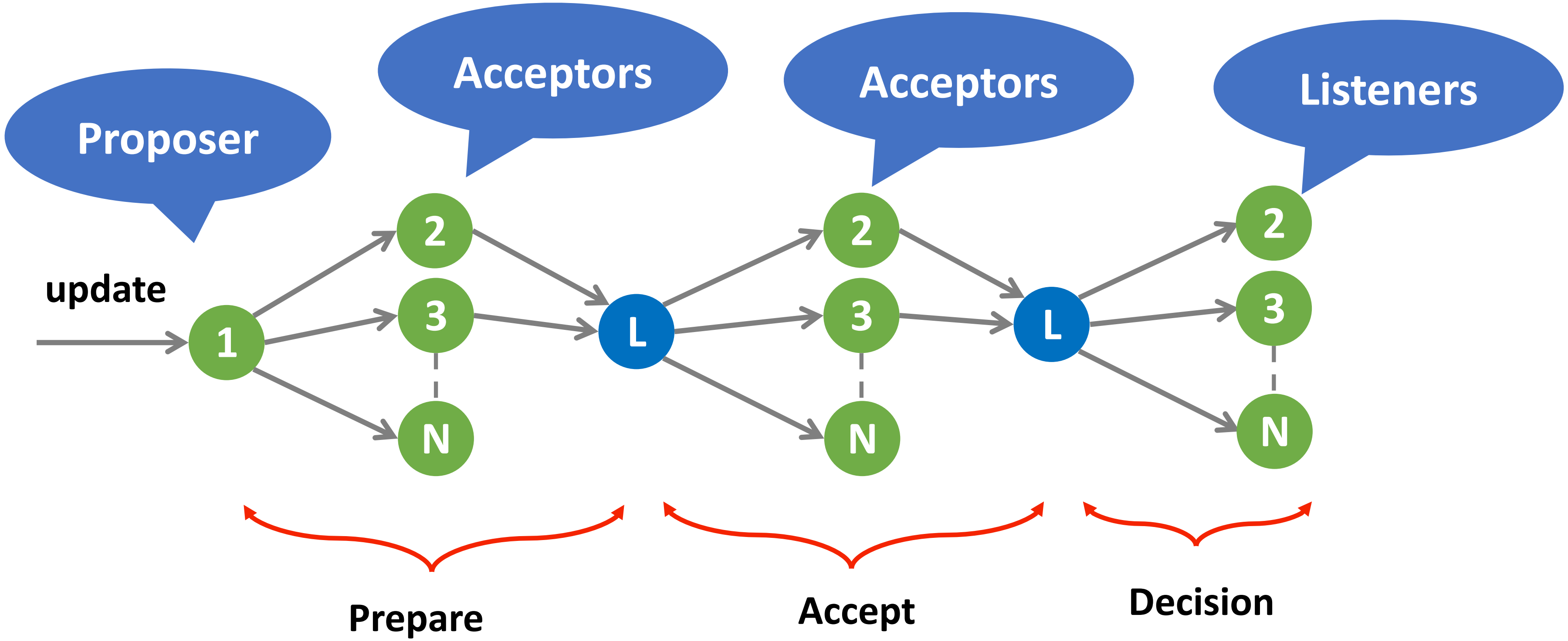
# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



# Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



# Safety Condition

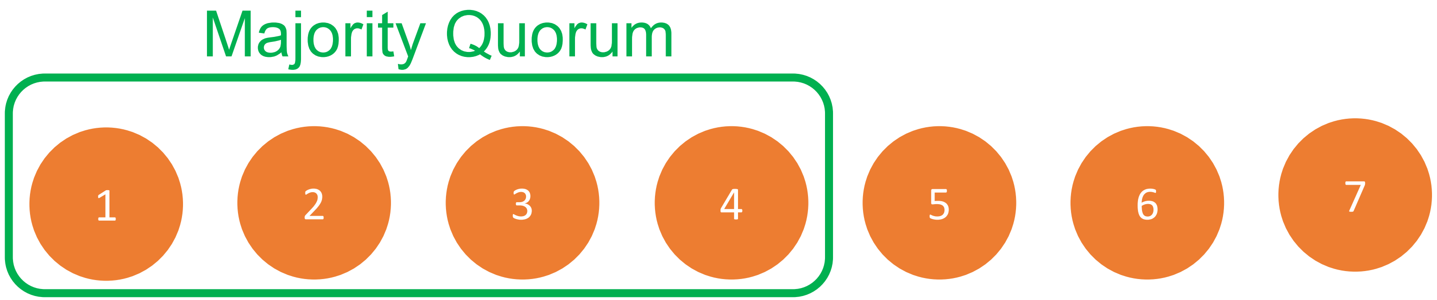
- Any two sets (**quorums**) of acceptors must have **at least one** overlapping acceptor
- This way a new leader will know of a value chosen by old leader through the overlapping acceptor

# Safety Condition

- Any two sets (**quorums**) of acceptors must have **at least one** overlapping acceptor
- This way a new leader will know of a value chosen by old leader through the overlapping acceptor

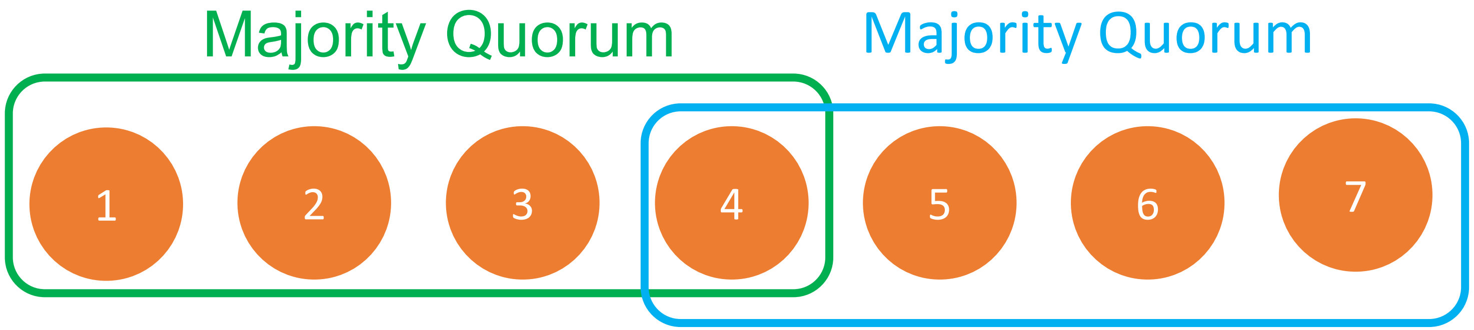
# Safety Condition

- Any two sets (**quorums**) of acceptors must have **at least one** overlapping acceptor
- This way a new leader will know of a value chosen by old leader through the overlapping acceptor



# Safety Condition

- Any two sets (**quorums**) of acceptors must have **at least one** overlapping acceptor
- This way a new leader will know of a value chosen by old leader through the overlapping acceptor



# Paxos is Leader-based

- *Ballots* distinguish among values proposed by different leaders
  - Unique,  locally monotonically increasing
  - Processes respond only to leader with highest ballot

# Paxos is Leader-based

- *Ballots* distinguish among values proposed by different leaders
  - Unique, locally monotonically increasing
  - Processes respond only to leader with highest ballot
- Pairs ⟨**num, process id**⟩ that form a total order.
- $\langle \mathbf{n_1, p_1} \rangle > \langle \mathbf{n_2, p_2} \rangle$
  - If $\mathbf{n_1 > n_2}$
  - Or $\mathbf{n_1 = n_2}$ and $\mathbf{p_1 > p_2}$

# Paxos is Leader-based

- *Ballots* distinguish among values proposed by different leaders
  - Unique,  locally monotonically increasing
  - Processes respond only to leader with highest ballot
- Pairs ⟨**num, process id**⟩ that form a total order.
- ⟨$\mathbf{n_1}$, $\mathbf{p_1}$⟩ **>** ⟨$\mathbf{n_2}$, $\mathbf{p_2}$⟩
  - If $\mathbf{n_1 > n_2}$
  - Or $\mathbf{n_1=n_2}$ and $\mathbf{p_1 > p_2}$
- If latest known ballot is ⟨**n, q**⟩ then
    - p chooses ⟨**n+1, p**⟩

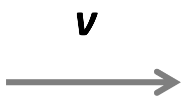
# The First Two Phases of Paxos

- Phase 1: prepare
  - If you *believe you are the leader*
    - Choose new unique ballot number
    - Learn outcome of all smaller ballots from majority

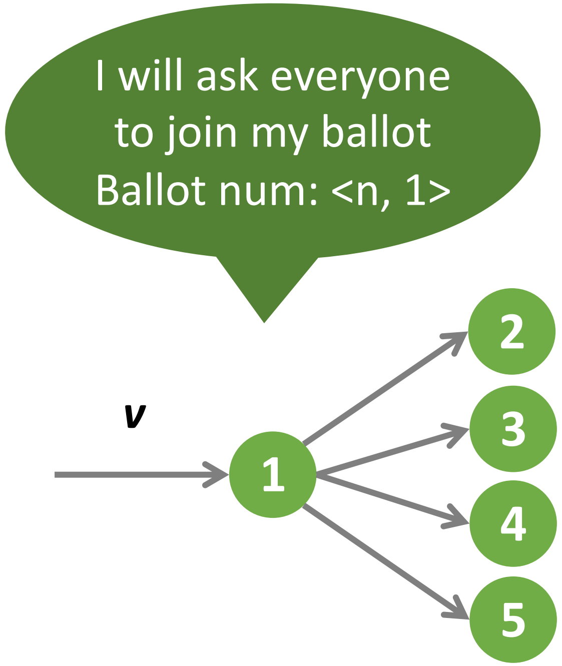
# The First Two Phases of Paxos

- Phase 1: prepare
  - If you *believe you are the leader*
    - Choose new unique ballot number
    - Learn outcome of all smaller ballots from majority
- Phase 2: accept
  - Leader *proposes* a value with its ballot number
  - Leader gets majority to *accept* its proposal
  - A value accepted by a majority can be decided

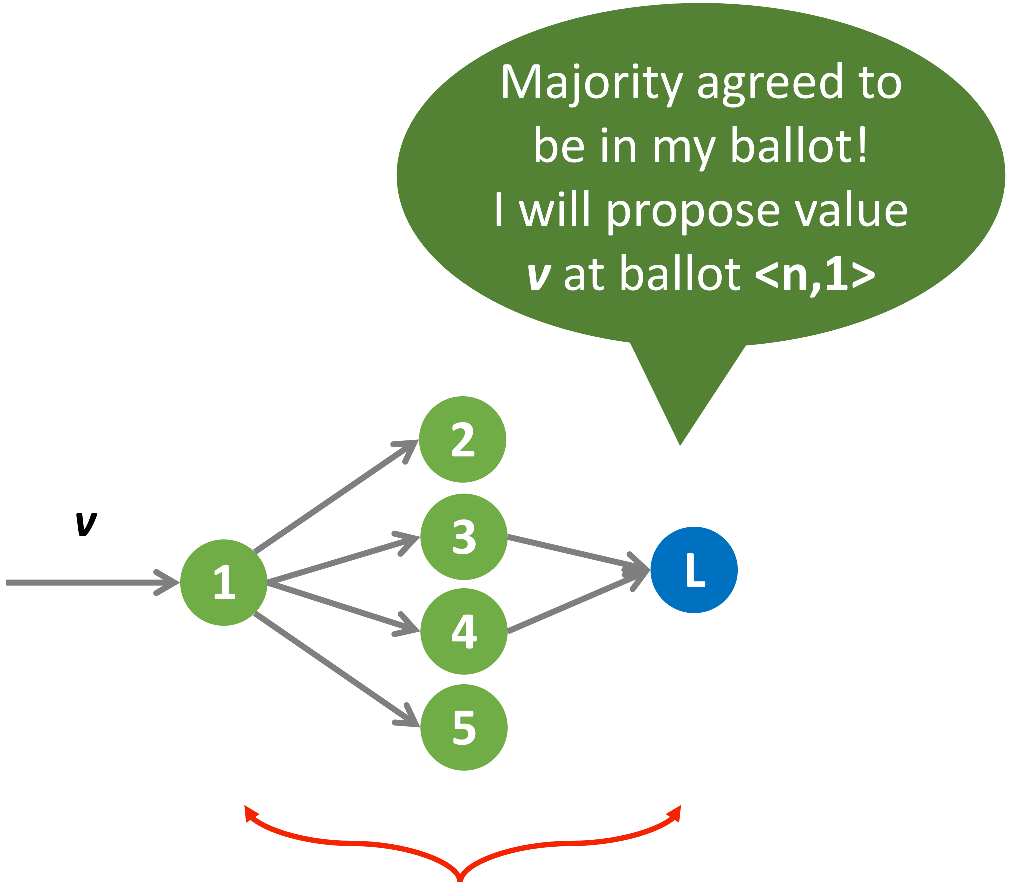
# Leader Crash



# Leader Crash

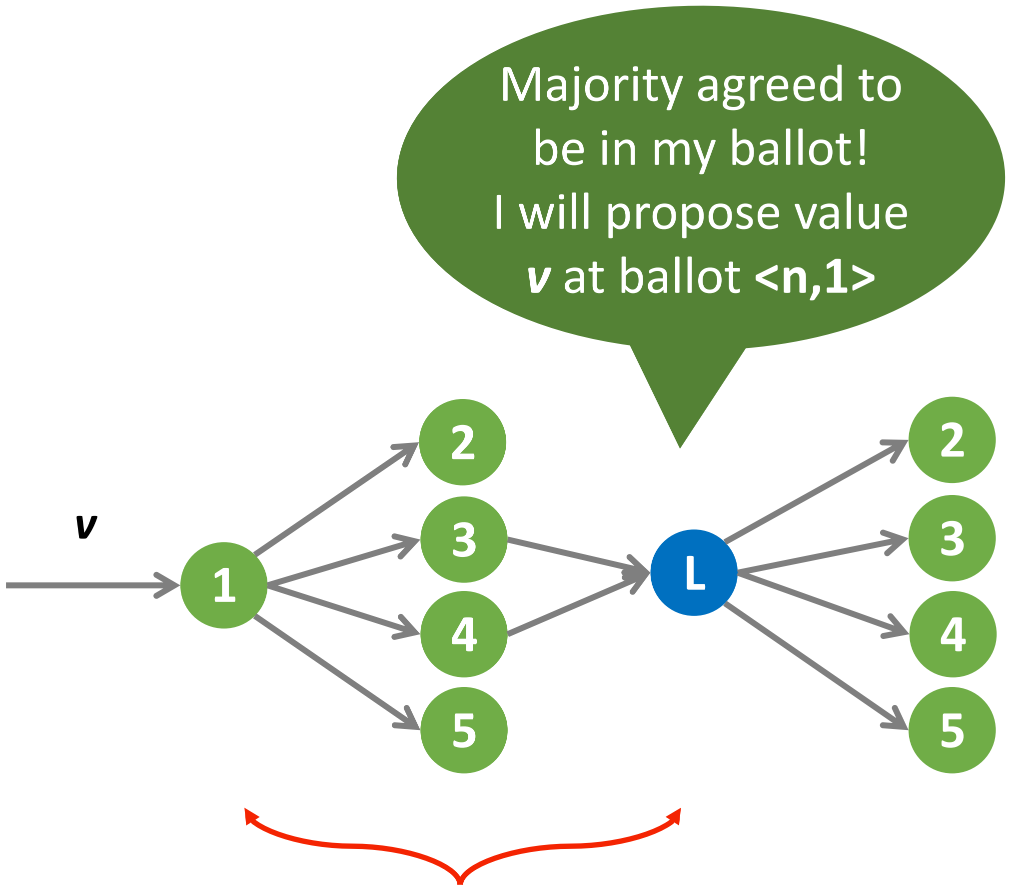


# Leader Crash


Majority agreed to be in my ballot! I will propose value ***v*** at ballot **<n,1>**

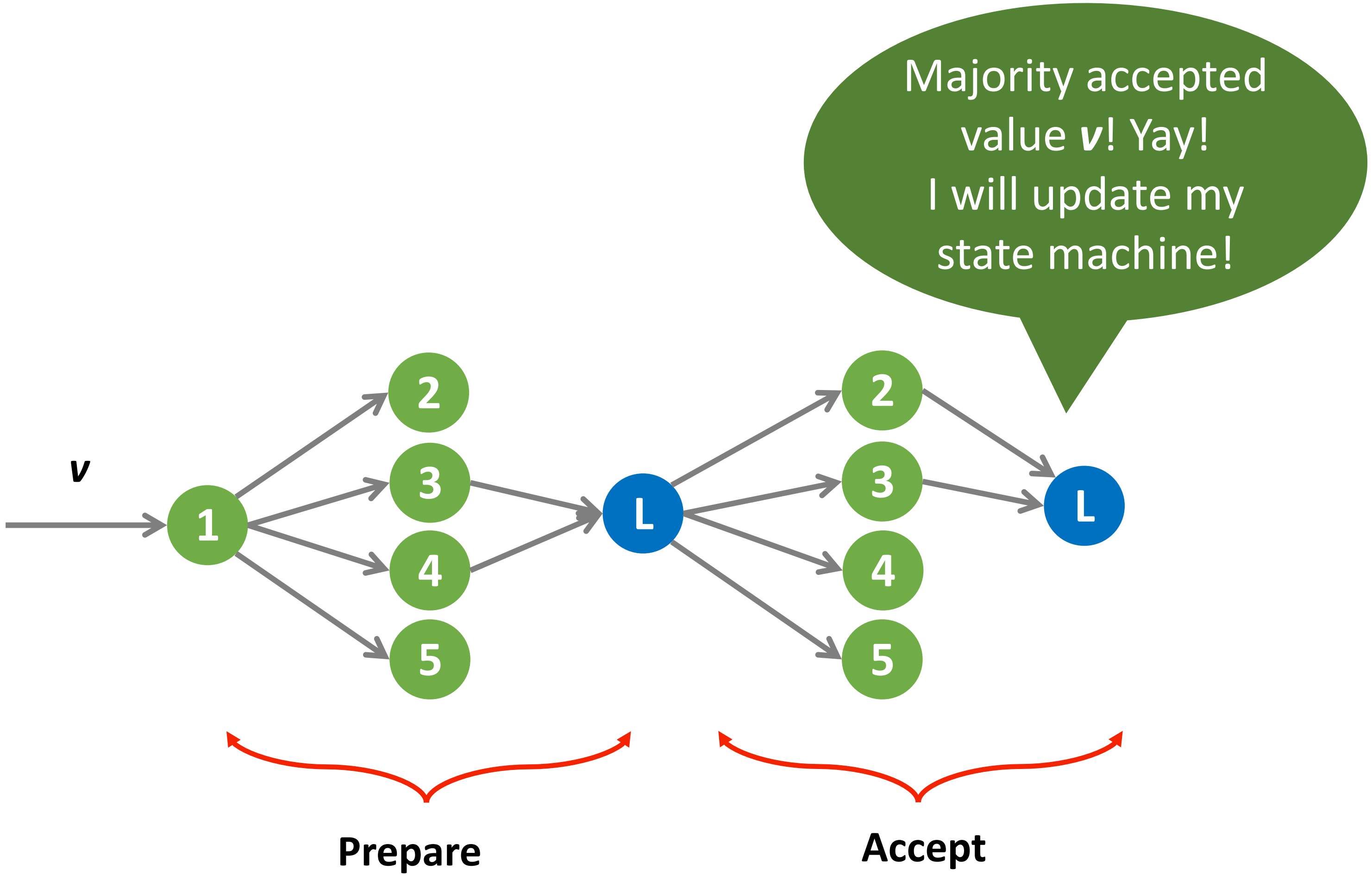
**Prepare**

# Leader Crash

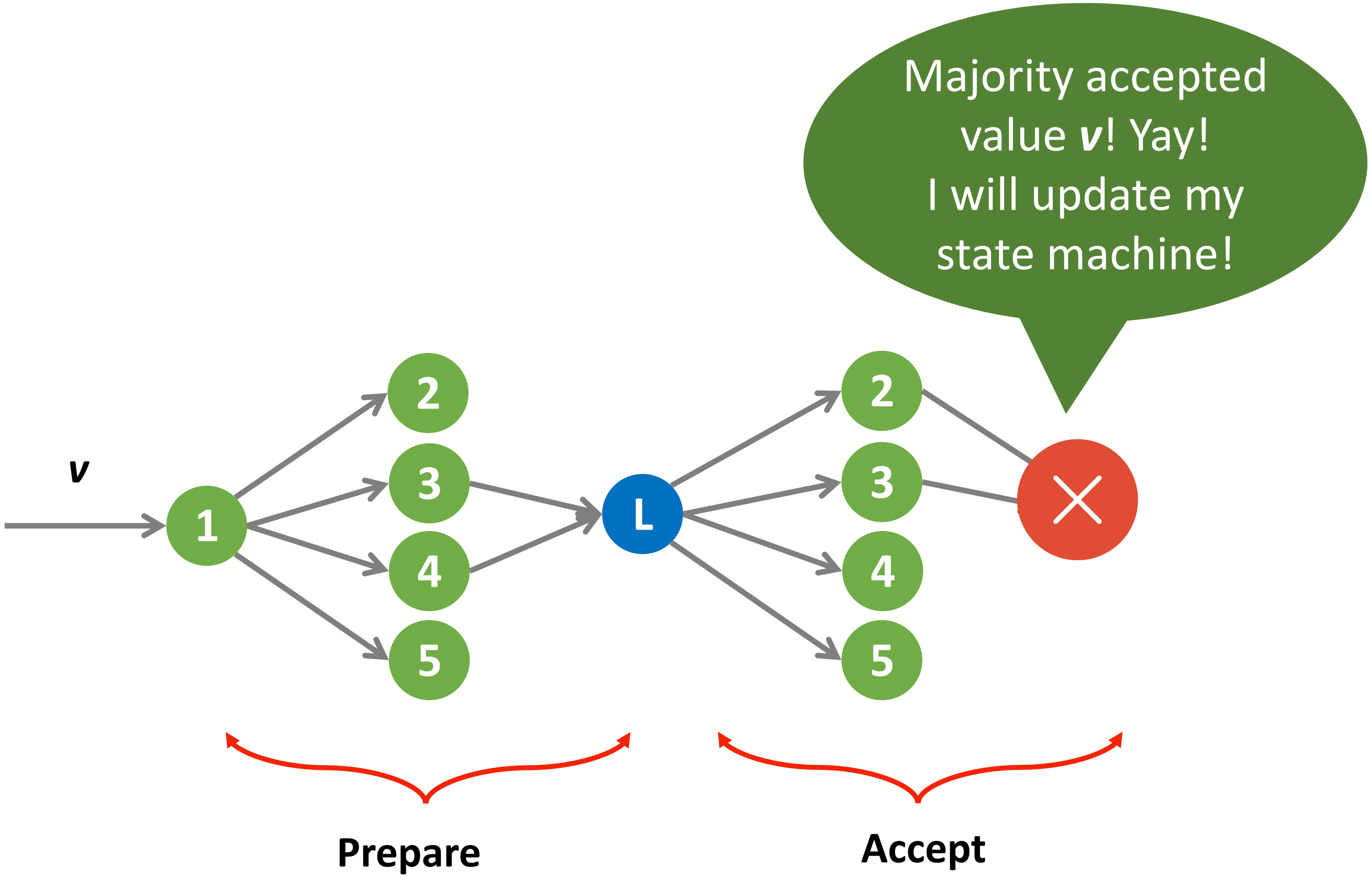

Majority agreed to be in my ballot! I will propose value ***v*** at ballot **<n,1>**

**Prepare**

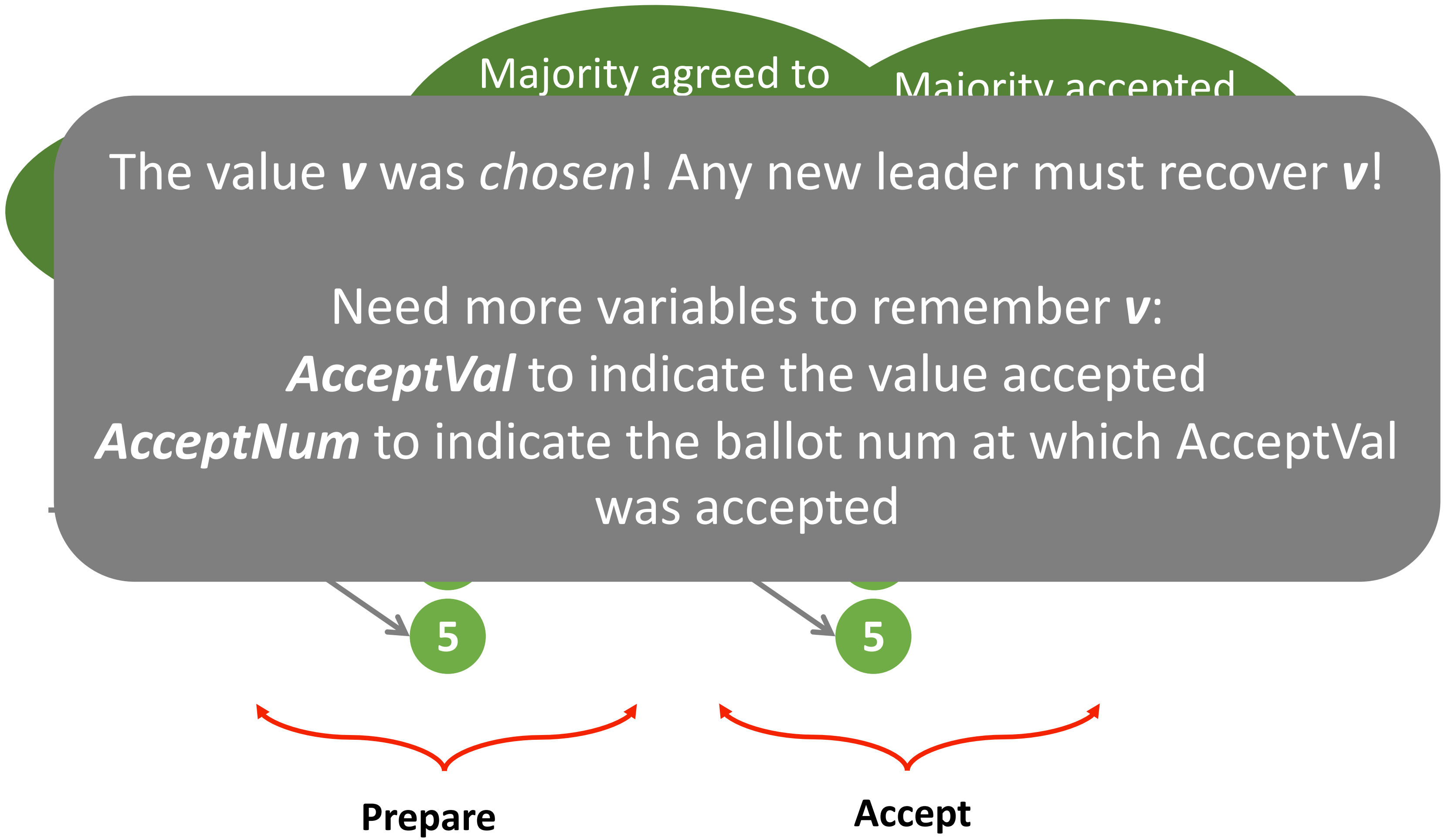
# Leader Crash



# Leader Crash



# Leader Crash



The value ***v*** was *chosen*! Any new leader must recover ***v***!

Need more variables to remember ***v***:
***AcceptVal*** to indicate the value accepted
***AcceptNum*** to indicate the ballot num at which AcceptVal was accepted

# Paxos - Variables

$\text{BallotNum}_i$, initially $\langle 0,0 \rangle$

Latest ballot $p_i$ took part in (phase 1)

$\text{AcceptNum}_i$, initially $\langle 0,0 \rangle$

Latest ballot $p_i$ accepted a value in (phase 2)

$\text{AcceptVal}_i$, initially $\perp$

Latest accepted value (phase 2)

The original version of these Paxos slides are from Idit Keidar several years ago
Thank you. Any errors are mine.

# Phase I: Prepare - Leader

```
if leader then
    BallotNum $\leftarrow \langle$BallotNum.num+1, myId$\rangle$
    send ("prepare", BallotNum) to all
```

# Phase I: Prepare - Leader

```
if leader then
    BallotNum ← ⟨BallotNum.num+1, myId⟩
    send ("prepare", BallotNum) to all
```

- Goal: contact other processes, ask them to join this ballot, and get information about possible past decisions

# Phase I: Prepare - Cohort

- Upon receive ("prepare", bal) from *i*

```
if bal ≥ BallotNum then
    BallotNum ← bal
    send  ("ack", bal, AcceptNum, AcceptVal) to i
```

This is a higher ballot than my current, I better join it

# Phase I: Prepare - Cohort

```
• Upon receive ("prepare", bal) from i
    if $bal \geq BallotNum$ then
        $BallotNum \leftarrow bal$
        send ("ack", bal, AcceptNum, AcceptVal) to i
```

This is a higher ballot than my current, I better join it

This is a promise not to accept ballots smaller than bal in the future

# Phase I: Prepare - Cohort

```
• Upon receive ("prepare", bal) from i
    if $bal \geq BallotNum$ then
        $BallotNum \leftarrow bal$
        send ("ack", bal, AcceptNum, AcceptVal) to i
```

This is a higher ballot than my current, I better join it

This is a promise not to accept ballots smaller than bal in the future

Tell the leader about my latest accepted value and what ballot it was accepted in

# Phase II: Accept - Leader

```
Upon receive ("ack", BallotNum, b, val) from majority
    if all vals = ^ then myVal = initial value
    else myVal = received val with highest b
    send ("accept", BallotNum, myVal) to all   /* proposal */
```

The value accepted in the highest ballot might have been decided, I better propose this value

# Phase II: Accept - Cohort

```
Upon receive ("accept", b, v)
if b ≥ BallotNum then
    AcceptNum ← b; AcceptVal ← v     /* accept proposal */
    send ("accept", b, v) to leader (or to all)
```

This is not from an old ballot

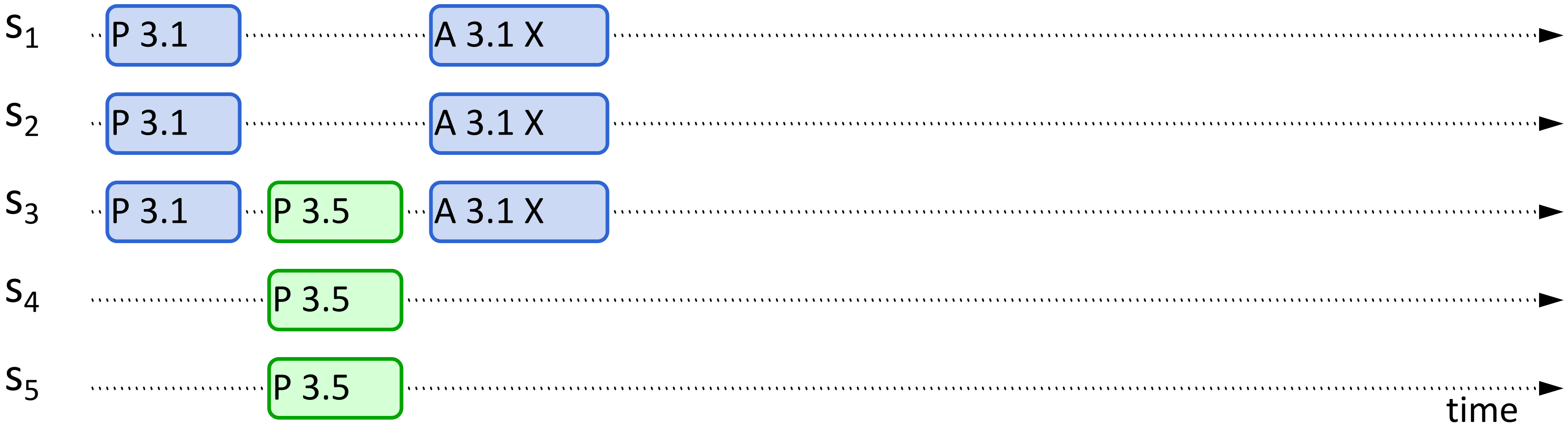
```
Upon receive ("accept", b, v) from majority
    decide v
```

# Liveness

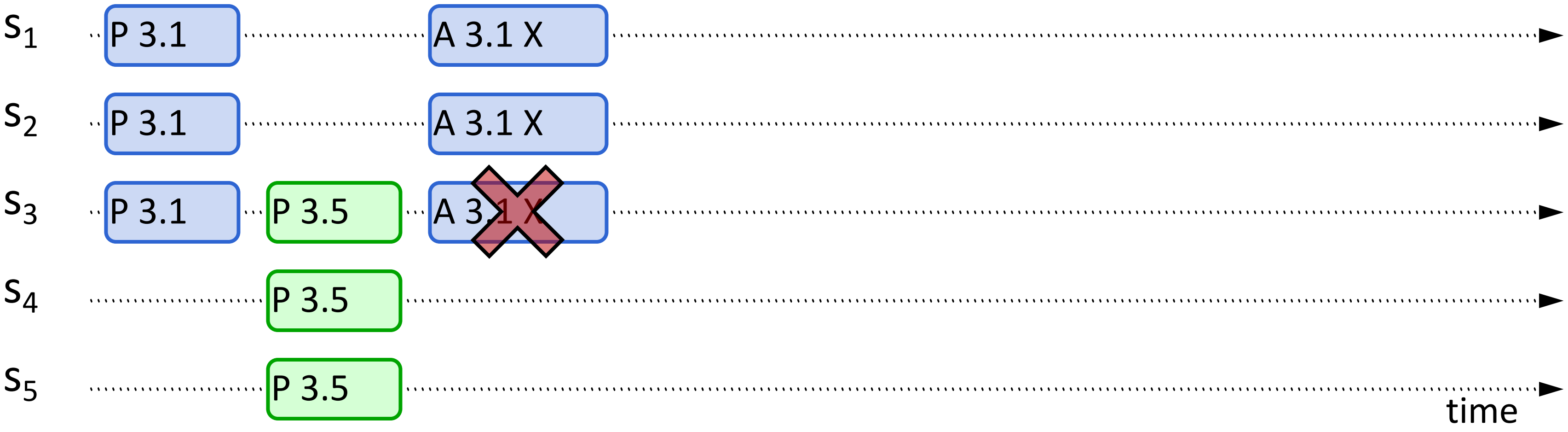
# Liveness

# Liveness

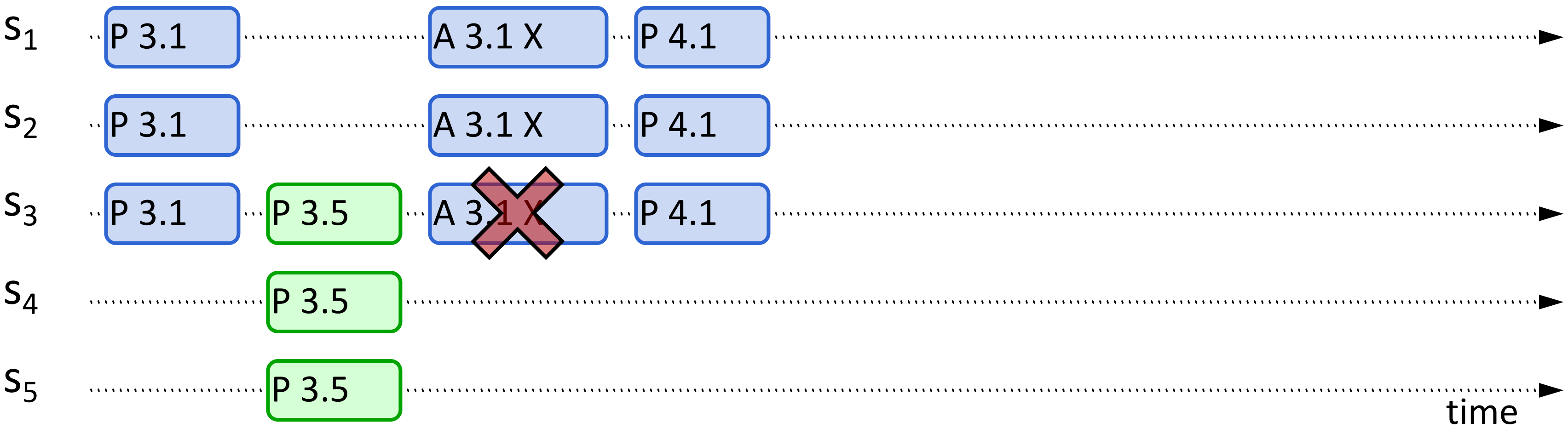
# Liveness



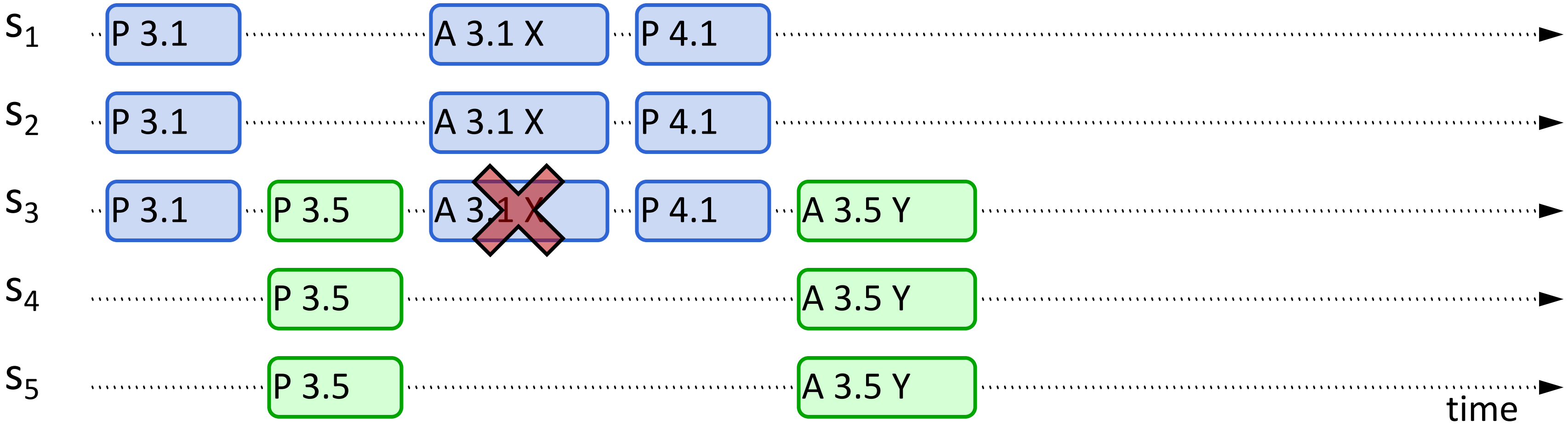
# Liveness



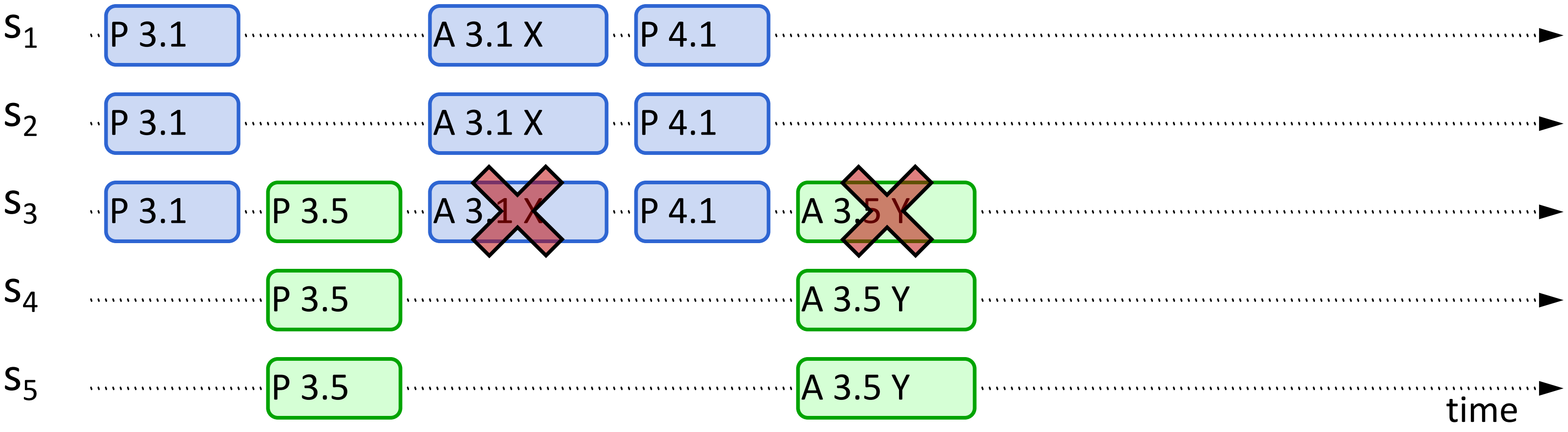
# Liveness


$S_1$
P 3.1
A 3.1 X
P 4.1
$S_2$
P 3.1
A 3.1 X
P 4.1
$S_3$
P 3.1
P 3.5
A 3.1 X
P 4.1
$S_4$
P 3.5
$S_5$
P 3.5
time

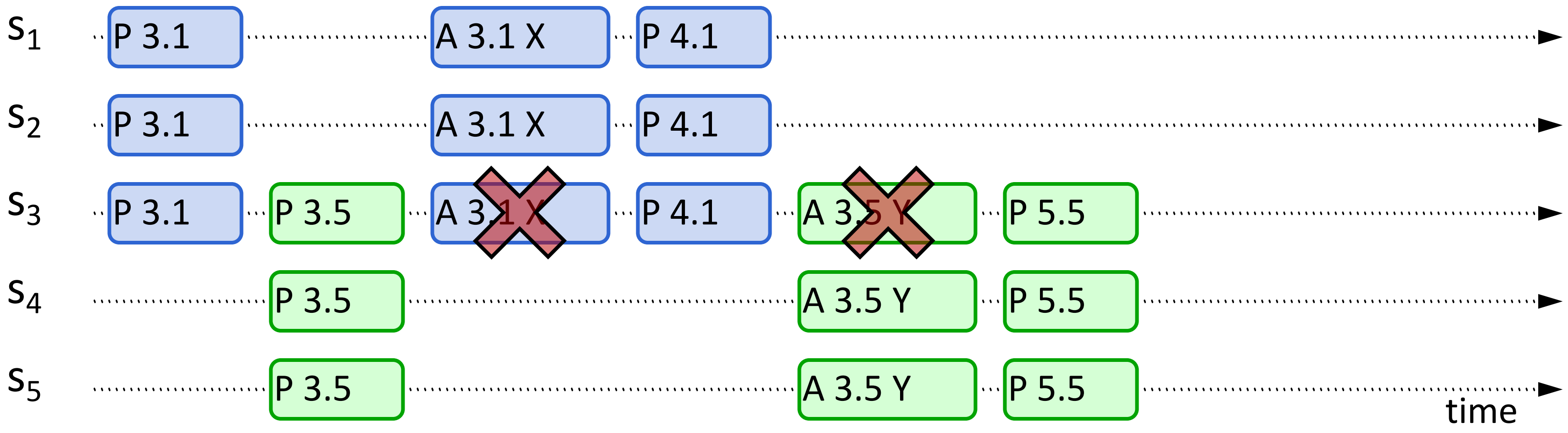
# Liveness


S$_1$
P 3.1
A 3.1 X
P 4.1
S$_2$
P 3.1
A 3.1 X
P 4.1
S$_3$
P 3.1
P 3.5
A 3.1 X
P 4.1
A 3.5 Y
S$_4$
P 3.5
A 3.5 Y
S$_5$
P 3.5
A 3.5 Y
time

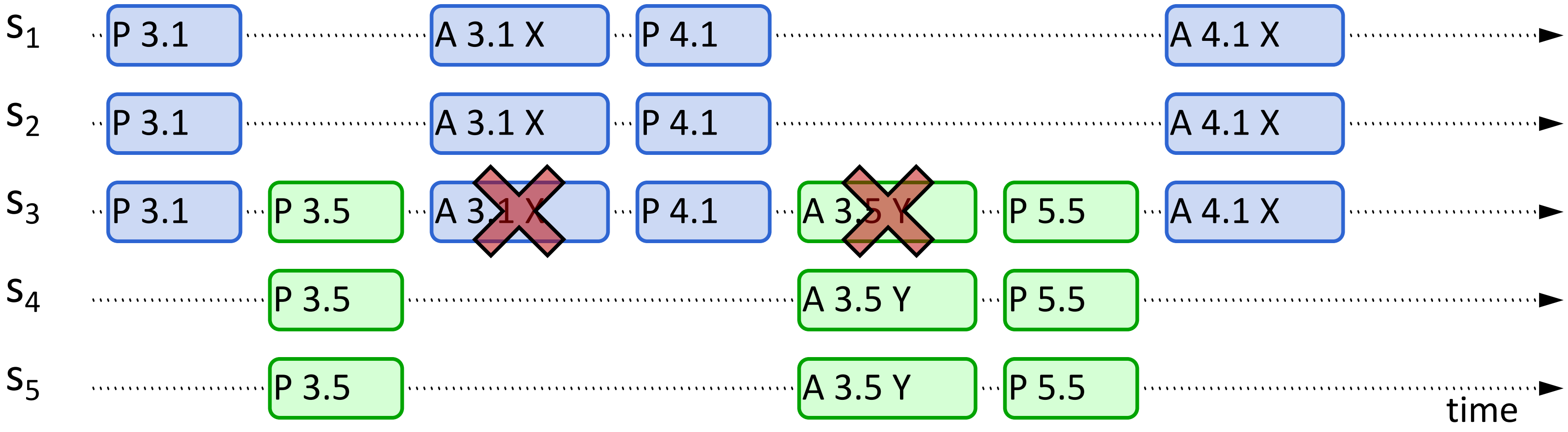
# Liveness


S$_1$
P 3.1
A 3.1 X
P 4.1
S$_2$
P 3.1
A 3.1 X
P 4.1
S$_3$
P 3.1
P 3.5
A 3.1 X
P 4.1
A 3.5 Y
S$_4$
P 3.5
A 3.5 Y
S$_5$
P 3.5
A 3.5 Y
time

# Liveness

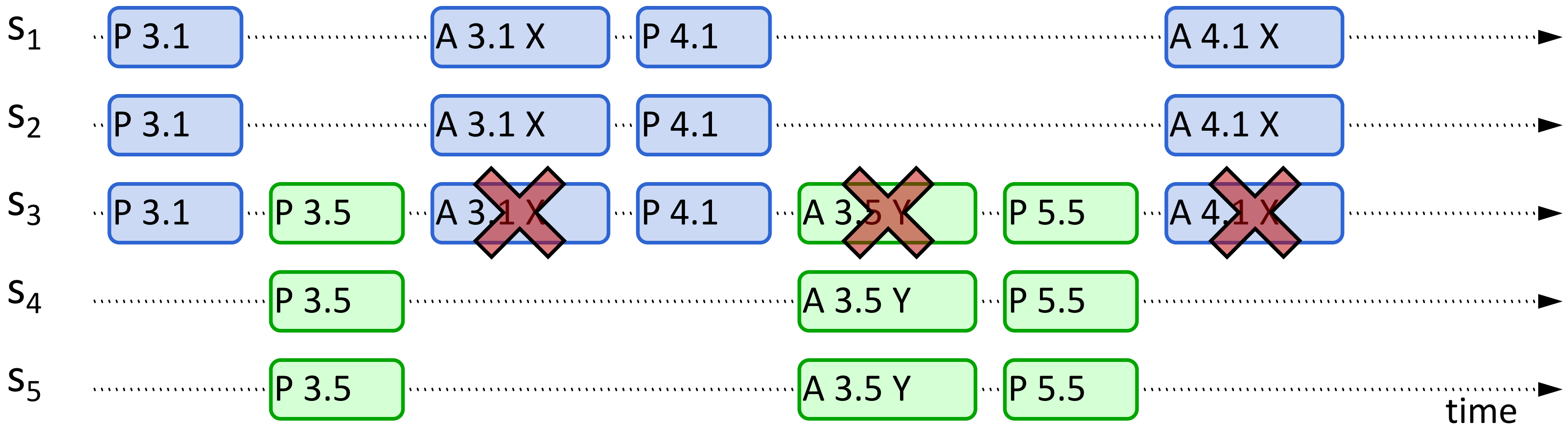

$S_1$
P 3.1
A 3.1 X
P 4.1
$S_2$
P 3.1
A 3.1 X
P 4.1
$S_3$
P 3.1
P 3.5
A 3.1 X
P 4.1
A 3.5 Y
P 5.5
$S_4$
P 3.5
A 3.5 Y
P 5.5
$S_5$
P 3.5
A 3.5 Y
P 5.5
time

# Liveness



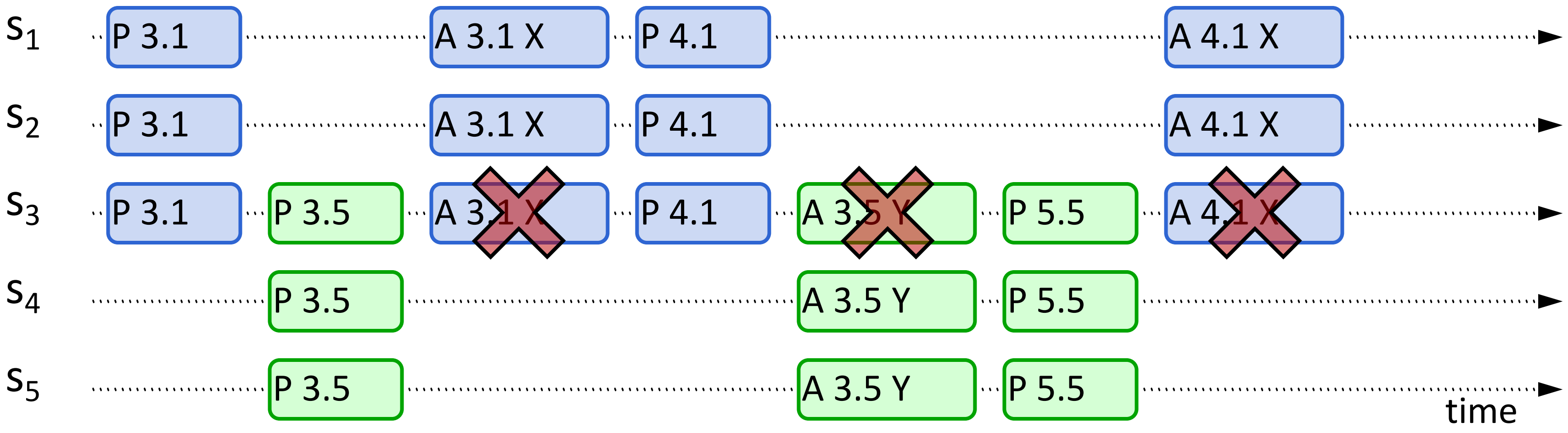
$S_1$
P 3.1
A 3.1 X
P 4.1
A 4.1 X
$S_2$
P 3.1
A 3.1 X
P 4.1
A 4.1 X
$S_3$
P 3.1
P 3.5
A 3.1 X
P 4.1
A 3.5 Y
P 5.5
A 4.1 X
$S_4$
P 3.5
A 3.5 Y
P 5.5
$S_5$
P 3.5
A 3.5 Y
P 5.5
time

# Liveness


$S_1$
P 3.1
A 3.1 X
P 4.1
A 4.1 X
$S_2$
P 3.1
A 3.1 X
P 4.1
A 4.1 X
$S_3$
P 3.1
P 3.5
A 3.1 X
P 4.1
A 3.5 Y
P 5.5
A 4.1 X
$S_4$
P 3.5
A 3.5 Y
P 5.5
$S_5$
P 3.5
A 3.5 Y
P 5.5
time

# Liveness

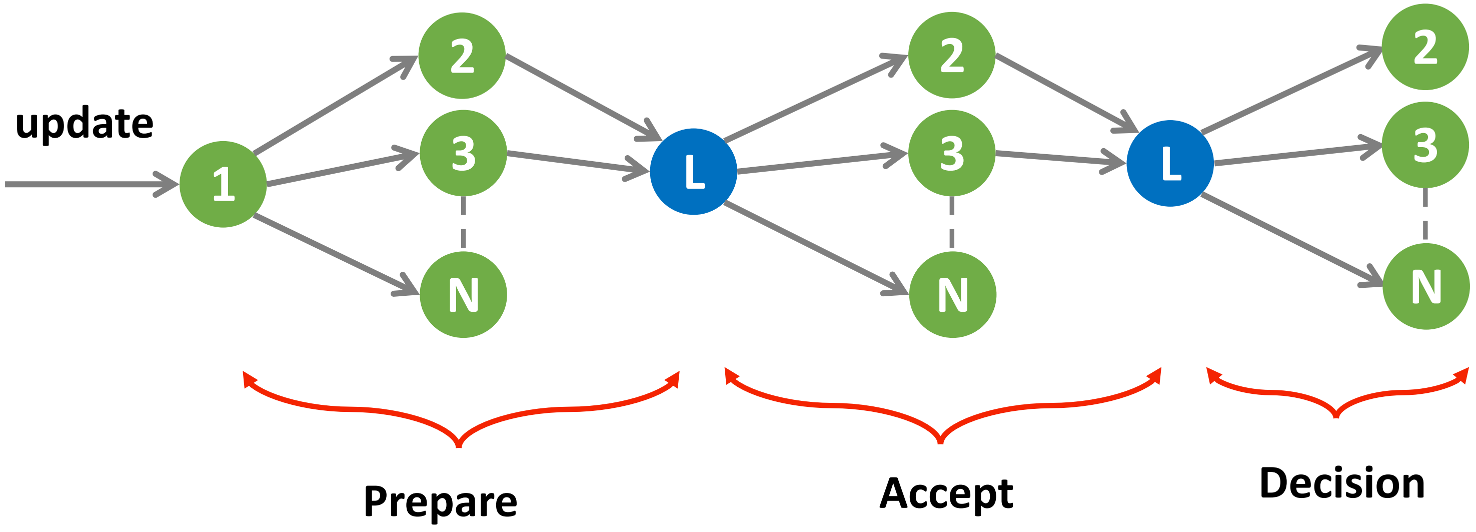
- Competing proposers can livelock:



- One solution: randomized delay before restarting
  - Give other proposers a chance to finish choosing

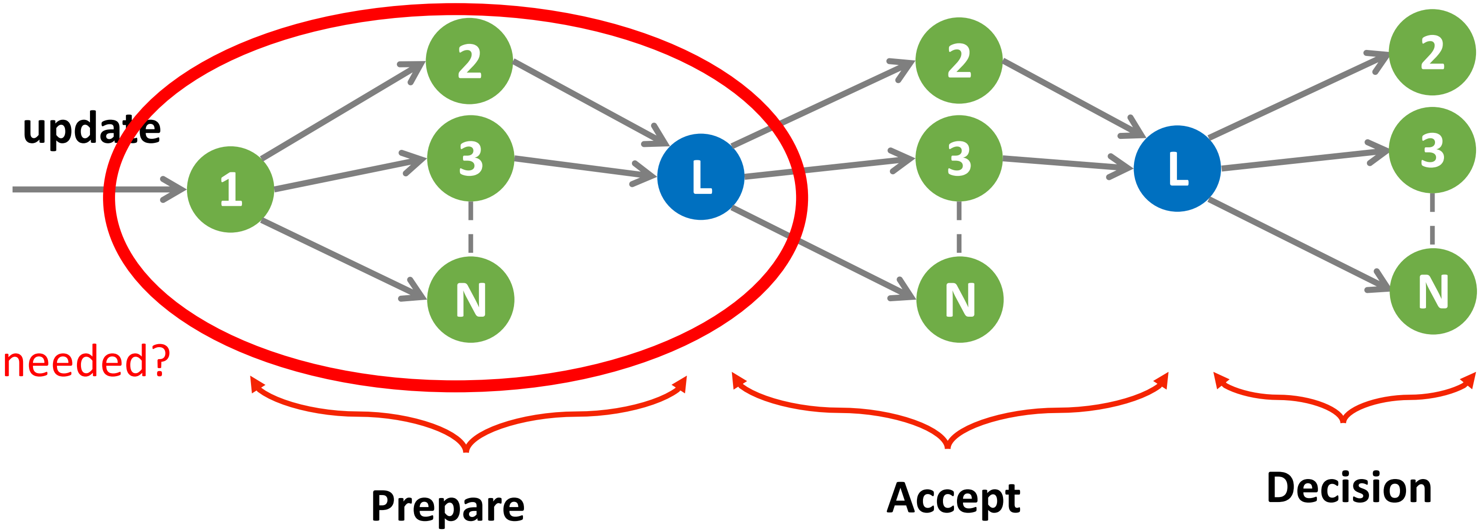
# Recall: Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



# Recall: Paxos Consensus Algorithm

- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



Why is this phase needed?

# Observation

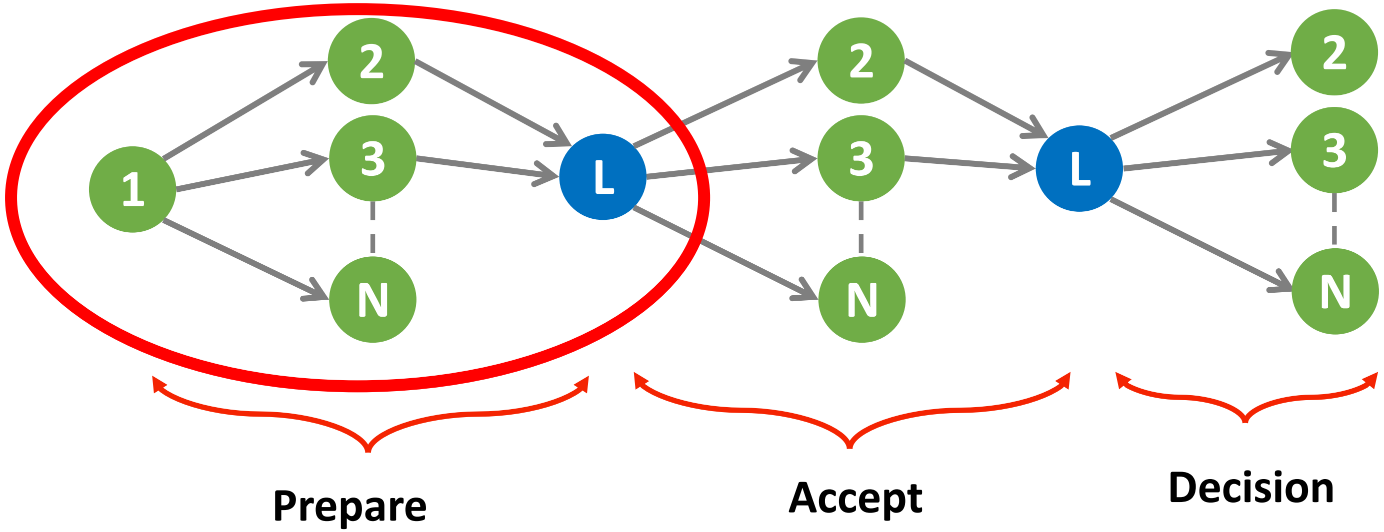
- In Phase 1, no consensus values are sent:
    - Leader chooses largest unique ballot number
    - Gets a majority to “vote” for this ballot number
    - Learns the outcome of all smaller ballots
- In Phase 2, leader proposes its own initial value or latest value it learned in Phase 1

# Optimization

- Run Phase 1 only when the leader changes
  - Phase 1 is called “view change” or “recovery mode”
  - Phase 2 is the “normal mode”
- Each message includes BallotNum (from the last Phase 1) and ReqNum
- Respond only to messages with the “right” BallotNum

# Recall: Paxos Consensus Algorithm

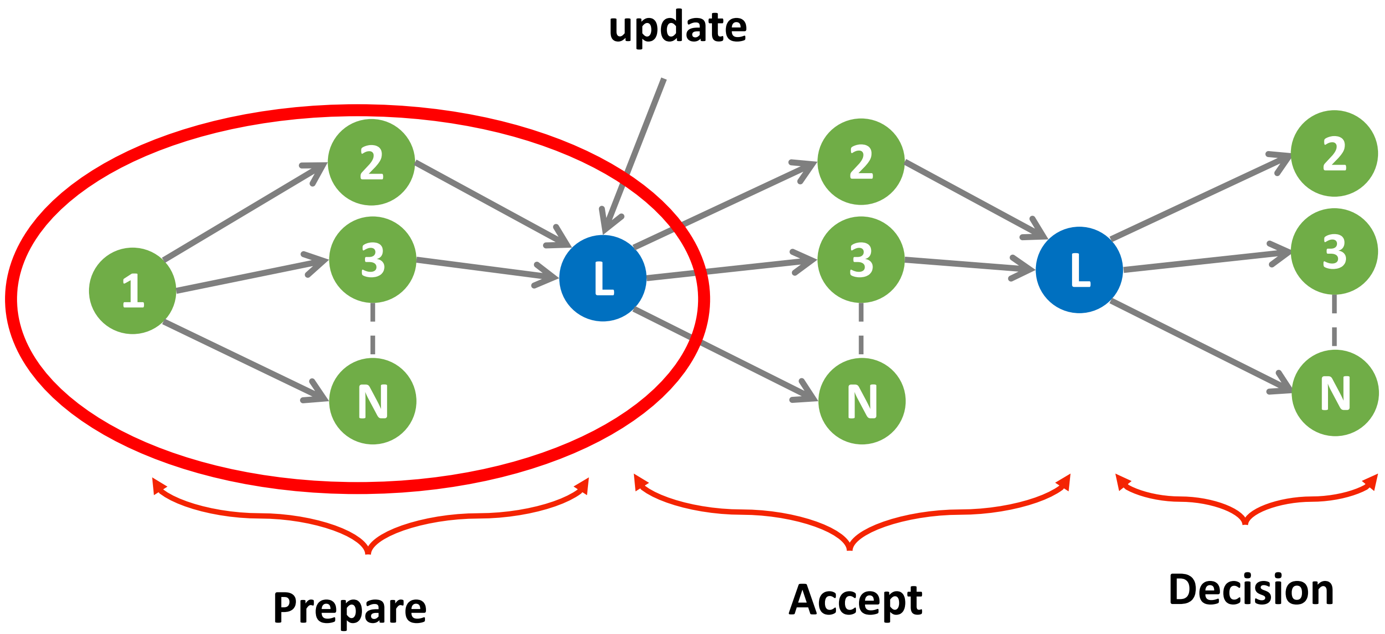
- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



Leader Election

# Recall: Paxos Consensus Algorithm

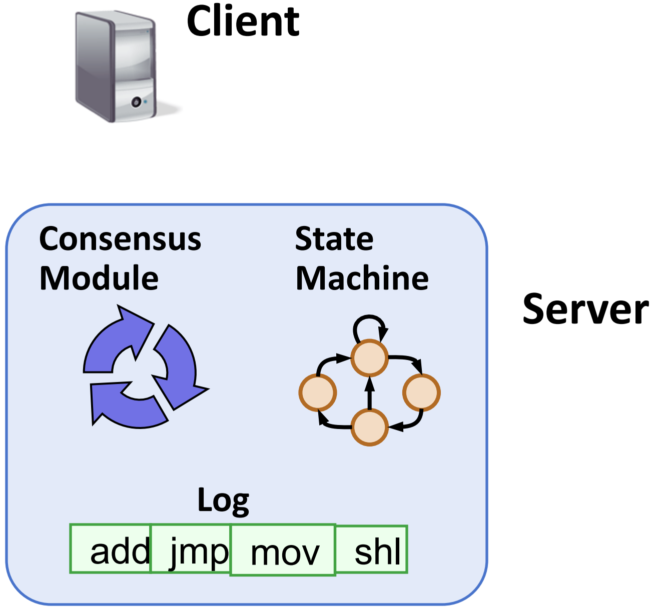
- **Leader Election:** Initially, a leader is elected by a quorum of servers
- **Replication:** Leader replicates new updates on quorum of servers
- **Decision:** Propagates decision to all **asynchronously**



Leader Election

# Multi-Paxos

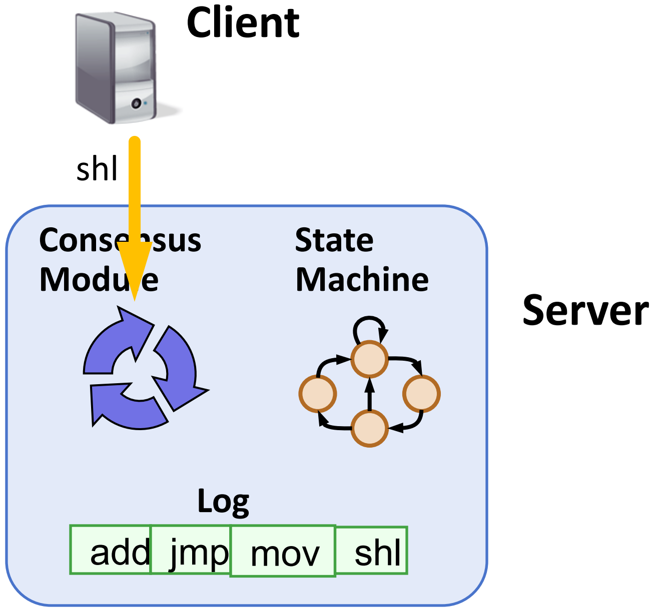
- Separate instance of Basic Paxos for each log entry:
  - Add index argument to Prepare and Accept (selects entry in log)



# Multi-Paxos

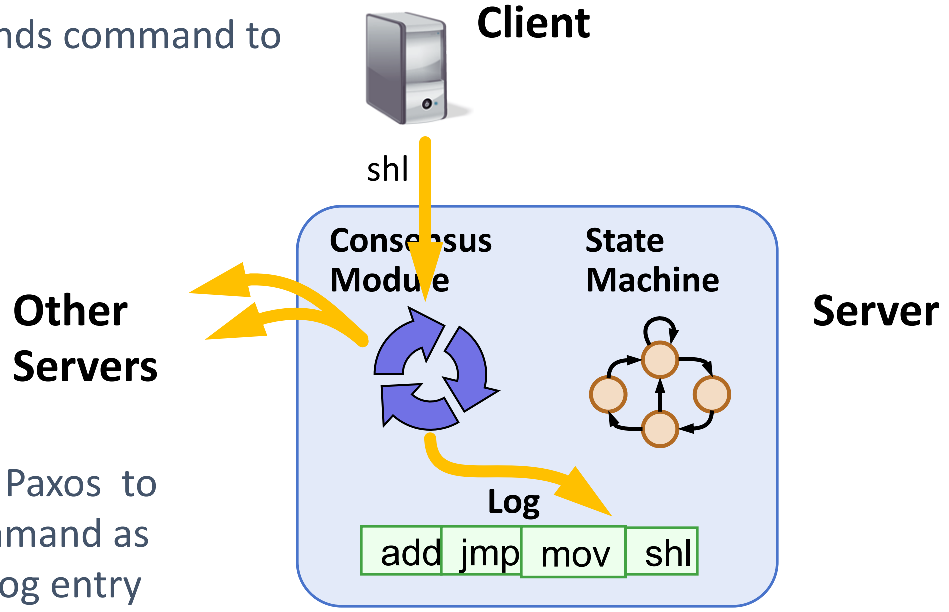
- Separate instance of Basic Paxos for each log entry:
  - Add index argument to Prepare and Accept (selects entry in log)

1. Client sends command to server



# Multi-Paxos

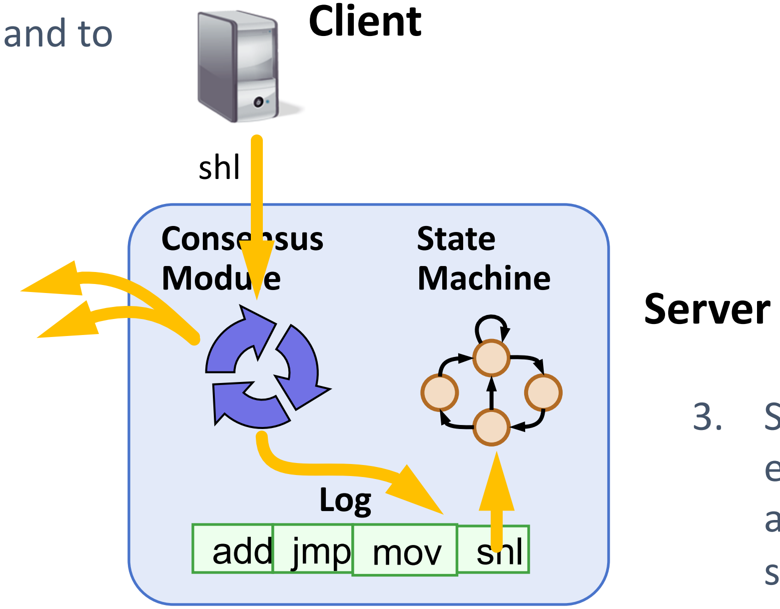
- Separate instance of Basic Paxos for each log entry:
  - Add index argument to Prepare and Accept (selects entry in log)



1. Client sends command to server
2. Server uses Paxos to choose command as value for a log entry

# Multi-Paxos

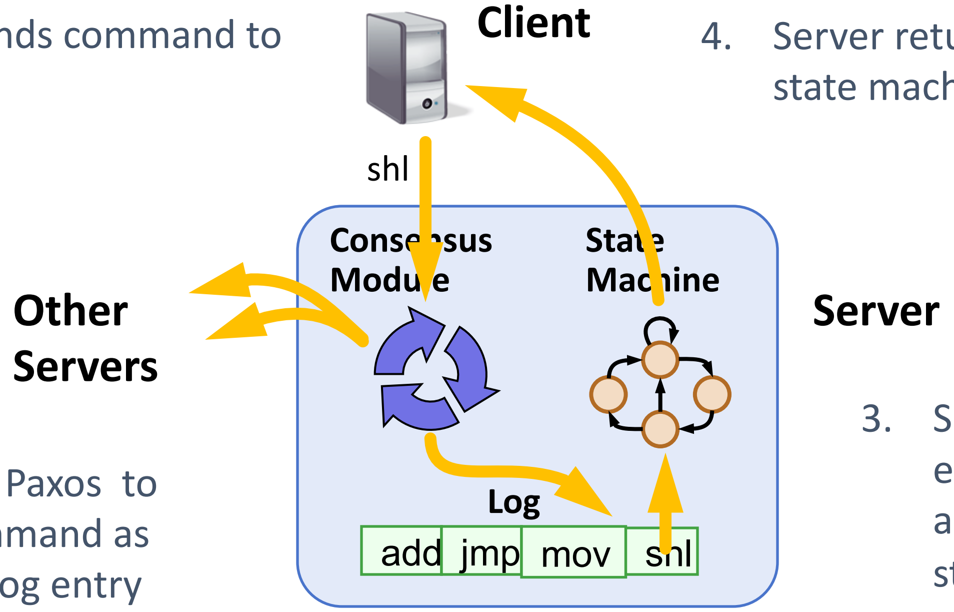
- Separate instance of Basic Paxos for each log entry:
  - Add index argument to Prepare and Accept (selects entry in log)



1. Client sends command to server
2. Server uses Paxos to choose command as value for a log entry
3. Server waits for previous log entries to be applied, then applies new command to state machine

# Multi-Paxos

- Separate instance of Basic Paxos for each log entry:
  - Add index argument to Prepare and Accept (selects entry in log)



1. Client sends command to server
2. Server uses Paxos to choose command as value for a log entry
3. Server waits for previous log entries to be applied, then applies new command to state machine
4. Server returns result from state machine to client

# Raft

Ongaro, D., & Ousterhout, J. In search of an understandable consensus algorithm. In *USENIX* ATC, *2014*

- Equivalent to Paxos in fault-tolerance
- Meant to be more understandable
- Uses a leader approach
- Integrates consensus with log management

Cockroach DB

HYPERLEDGER FABRIC

r3. corda

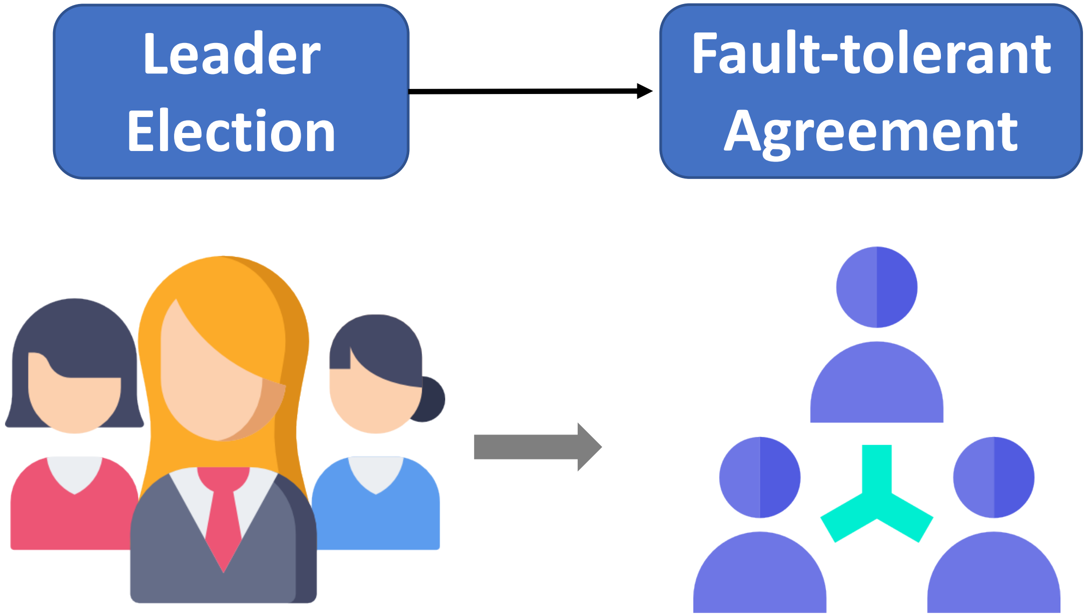
Quorum

| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | 2f+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 2 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O(N) Complexity |

# Abstract Paxos

Leader Election

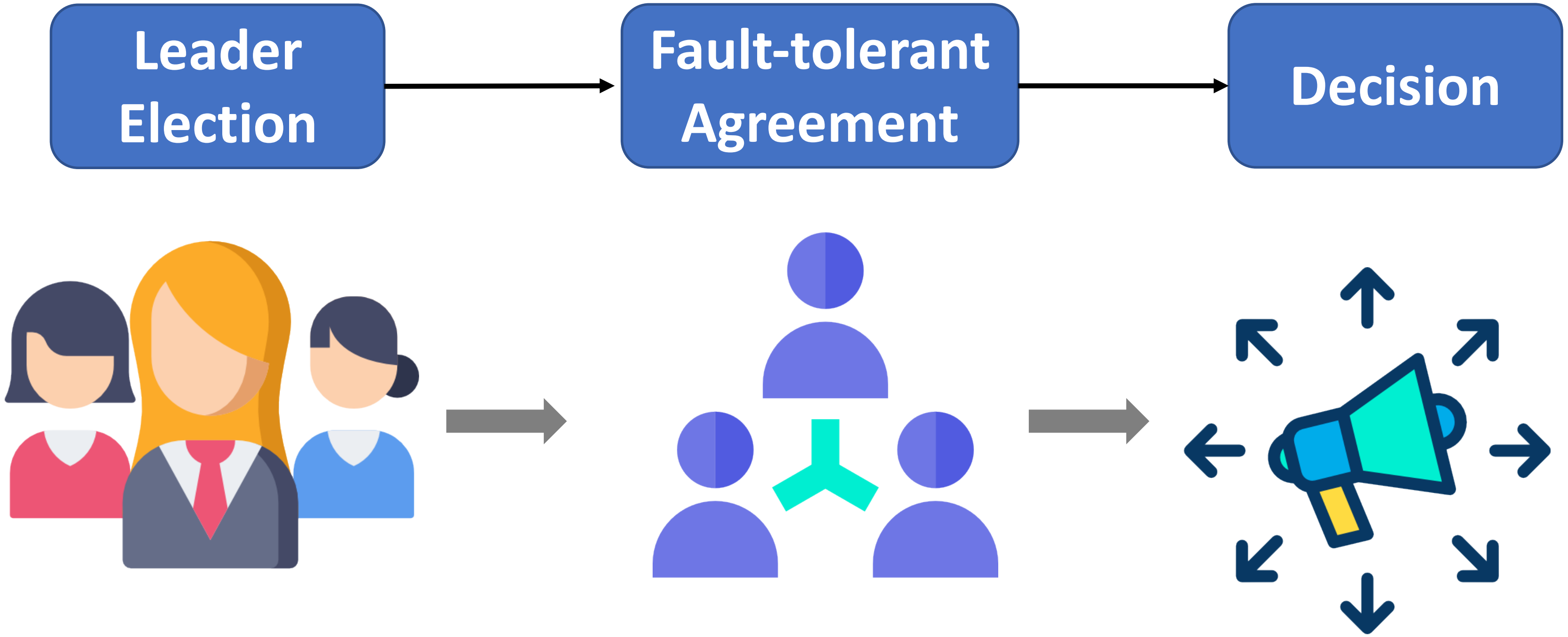
# Abstract Paxos



Leader Election

Fault-tolerant Agreement

# Abstract Paxos



Leader Election

Fault-tolerant Agreement

Decision

# Two Phase Commit

- J. N. Gray. "Notes on data base operating systems." *Operating Systems*. Springer, Berlin, Heidelberg, 1978.
- B. Lampson and H. Sturgis. Crash recovery in a distributed system. Technical report, Xerox PARC Research Report, 1976.

# Two Phase Commit

- A distributed transaction accesses data stored across multiple servers
- 2PC is ***atomic commitment*** protocol: either all servers commit or no server commits

# Two Phase Commit

- A distributed transaction accesses data stored across multiple servers
- 2PC is ***atomic commitment*** protocol: either all servers commit or no server commits

# Two Phase Commit

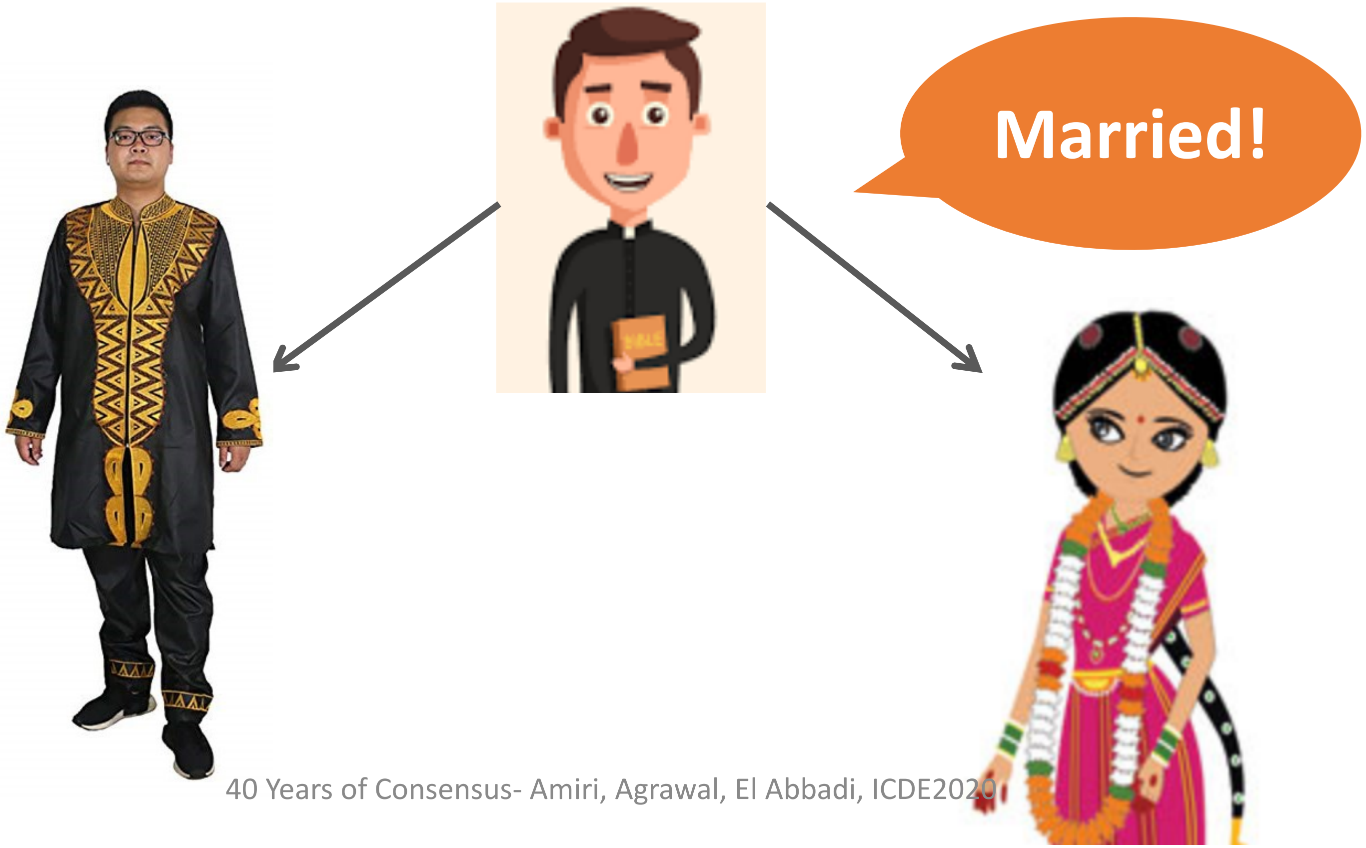
- A distributed transaction accesses data stored across multiple servers
- 2PC is ***atomic commitment*** protocol: either all servers commit or no server commits

# Two Phase Commit

- A distributed transaction accesses data stored across multiple servers
- 2PC is ***atomic commitment*** protocol: either all servers commit or no server commits

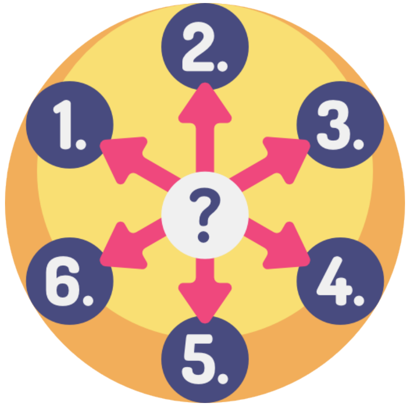
# Two Phase Commit

- A distributed transaction accesses data stored across multiple servers
- 2PC is ***atomic commitment*** protocol: either all servers commit or no server commits

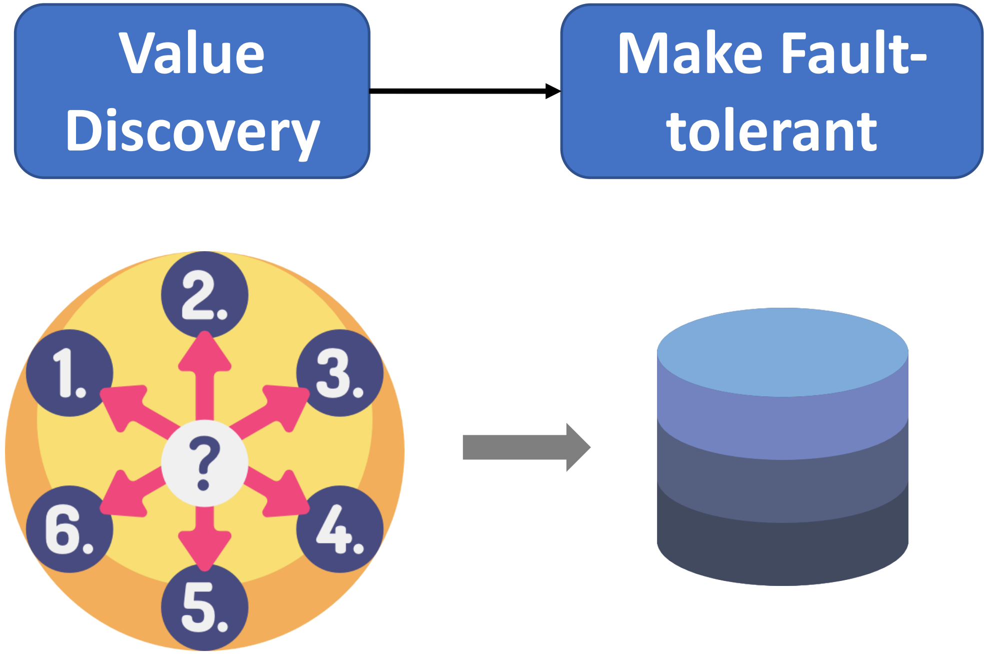


# Abstract 2PC

**Value Discovery**

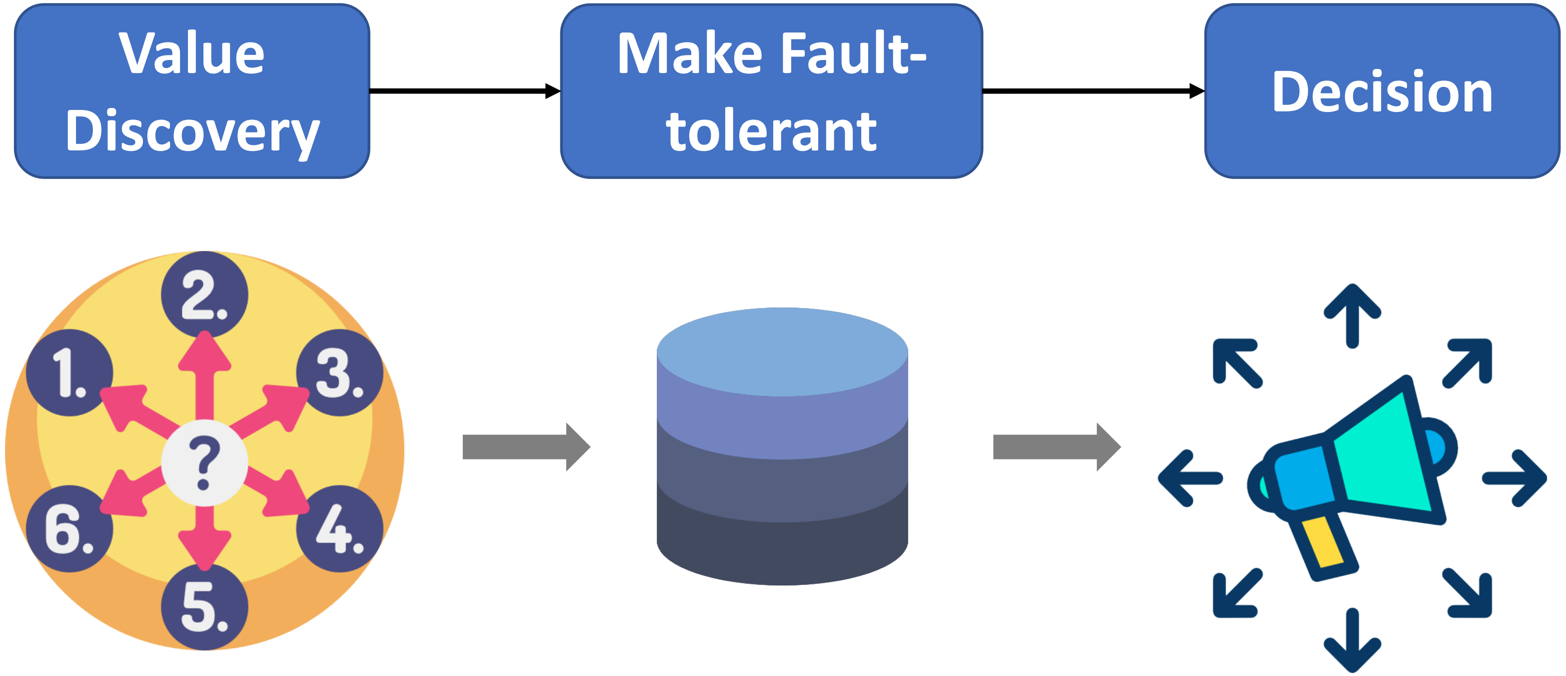


# Abstract 2PC


Value Discovery

Make Fault-tolerant

# Abstract 2PC



Value Discovery

Make Fault-tolerant

Decision

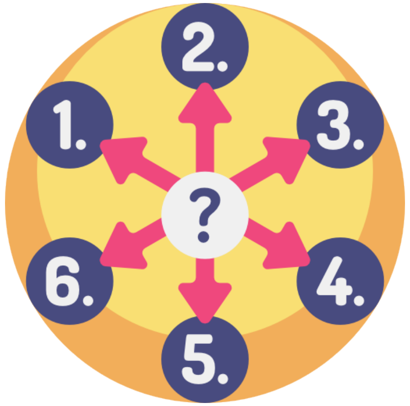
# Abstract 3PC

- 2PC has possibility of Blocking
- 3 Phase Commit:  Replicate decision to cohorts (like Paxos)

# Abstract 3PC

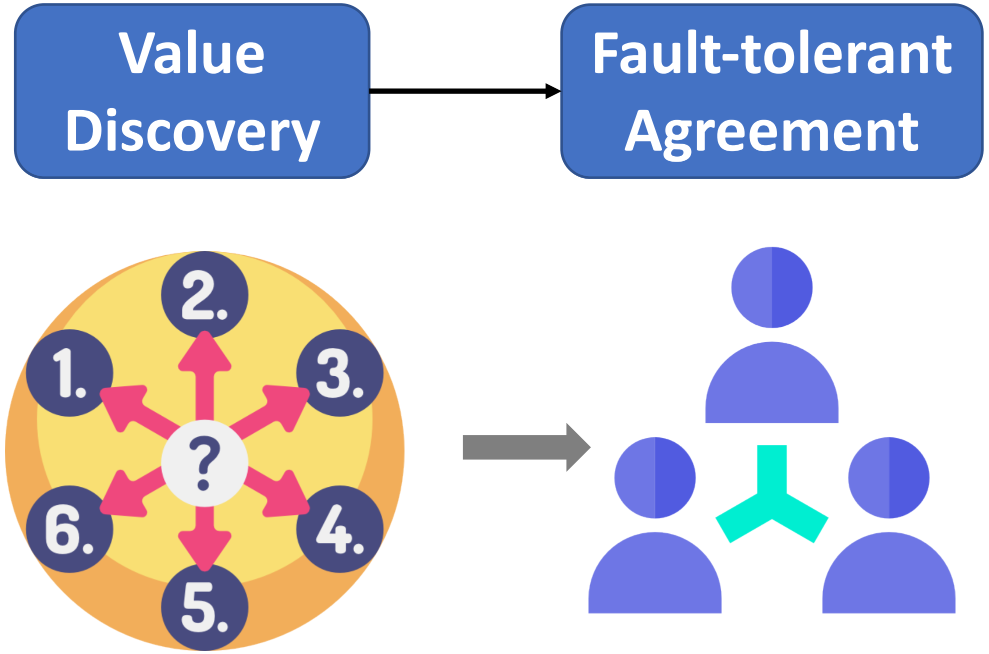
- 2PC has possibility of Blocking
- 3 Phase Commit: Replicate decision to cohorts (like Paxos)

Value Discovery



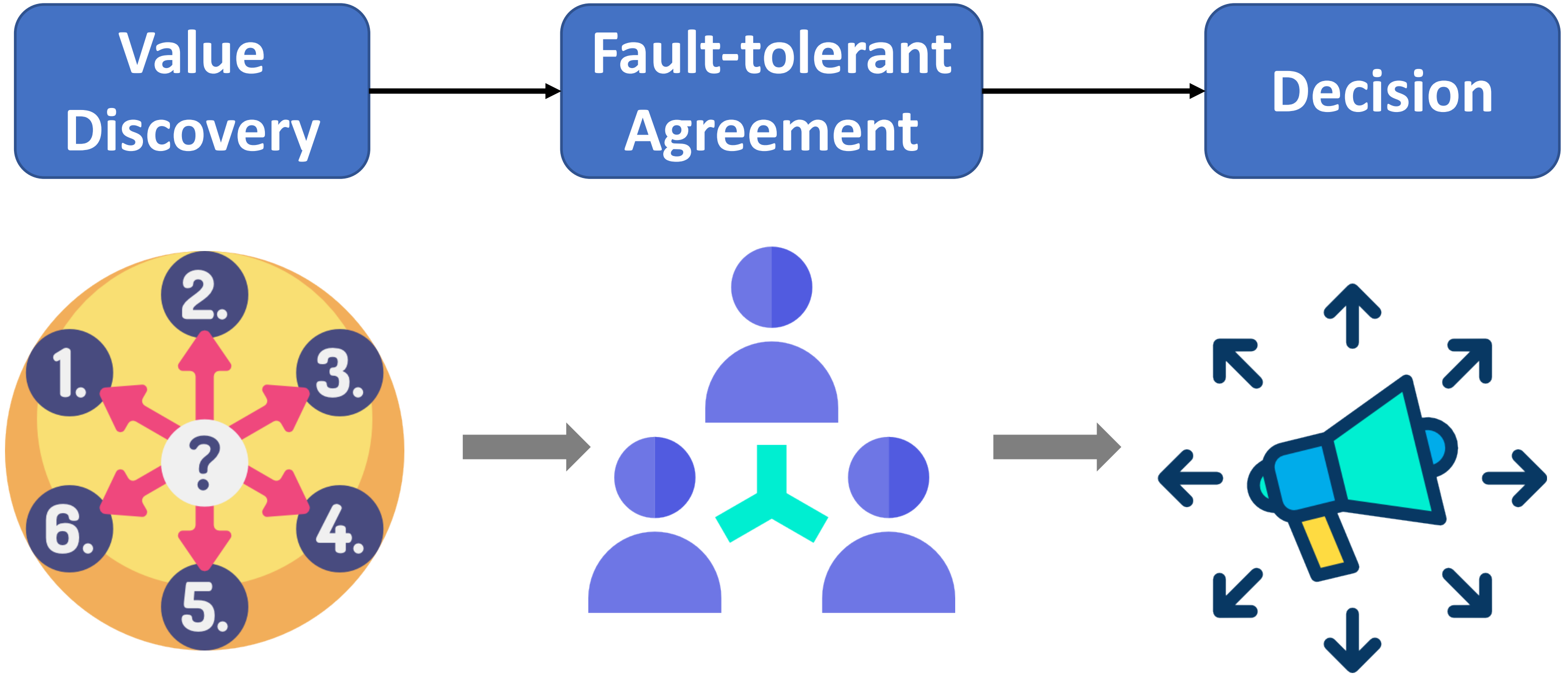
# Abstract 3PC

- 2PC has possibility of Blocking
- 3 Phase Commit: Replicate decision to cohorts (like Paxos)



# Abstract 3PC

- 2PC has possibility of Blocking
- 3 Phase Commit: Replicate decision to cohorts (like Paxos)


Value Discovery

Fault-tolerant Agreement

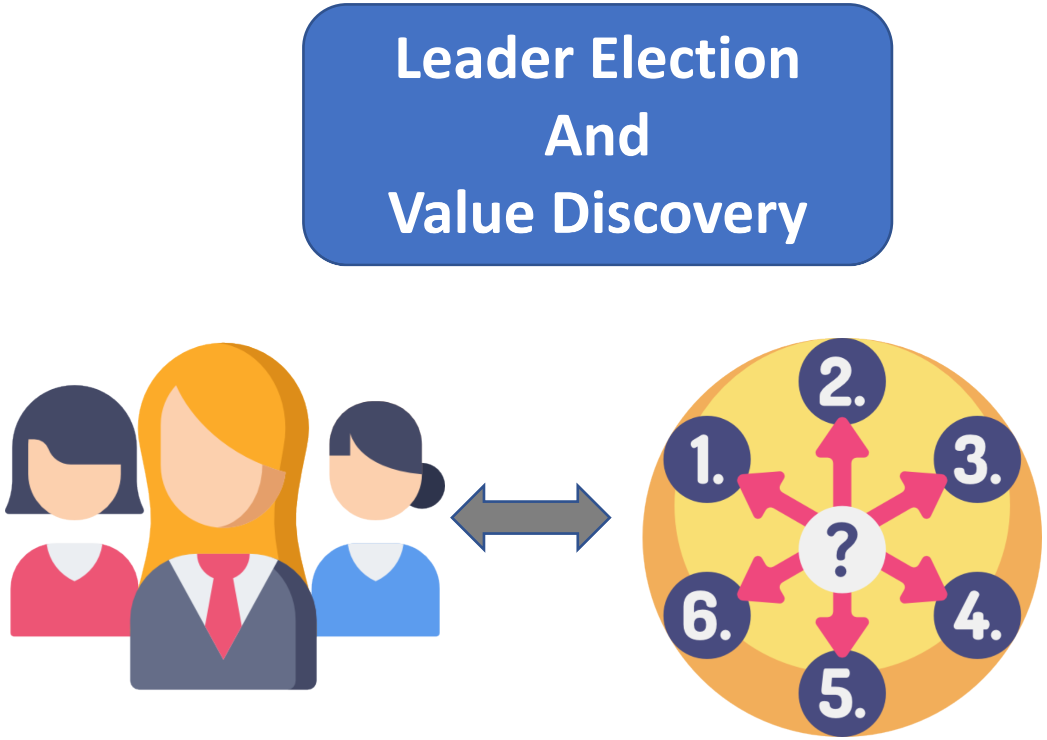
Decision

# Fault-tolerant 3PC (with Termination)

- If leader fails: Elect new leader and execute termination protocol

# Fault-tolerant 3PC (with Termination)

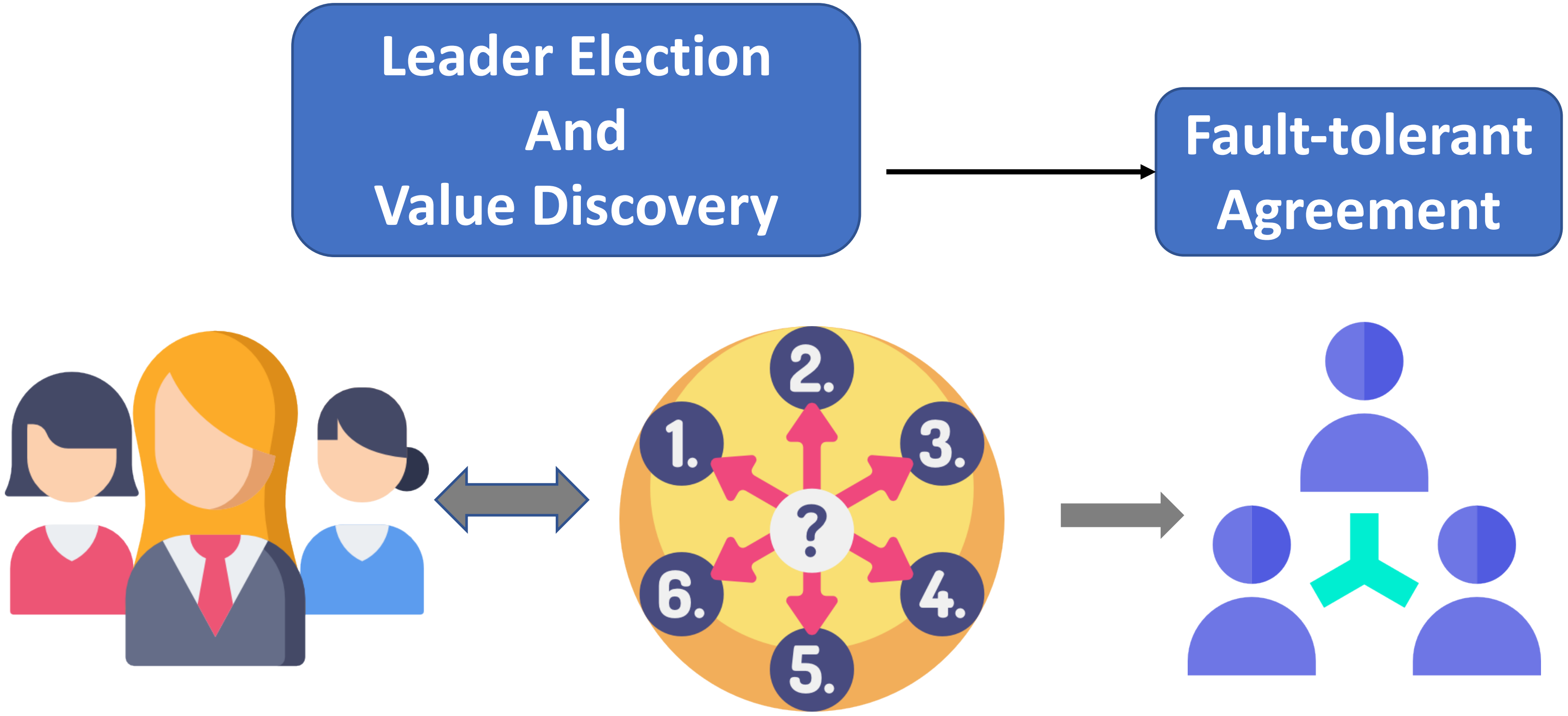
- If leader fails: Elect new leader and execute termination protocol



Leader Election
And
Value Discovery

# Fault-tolerant 3PC (with Termination)

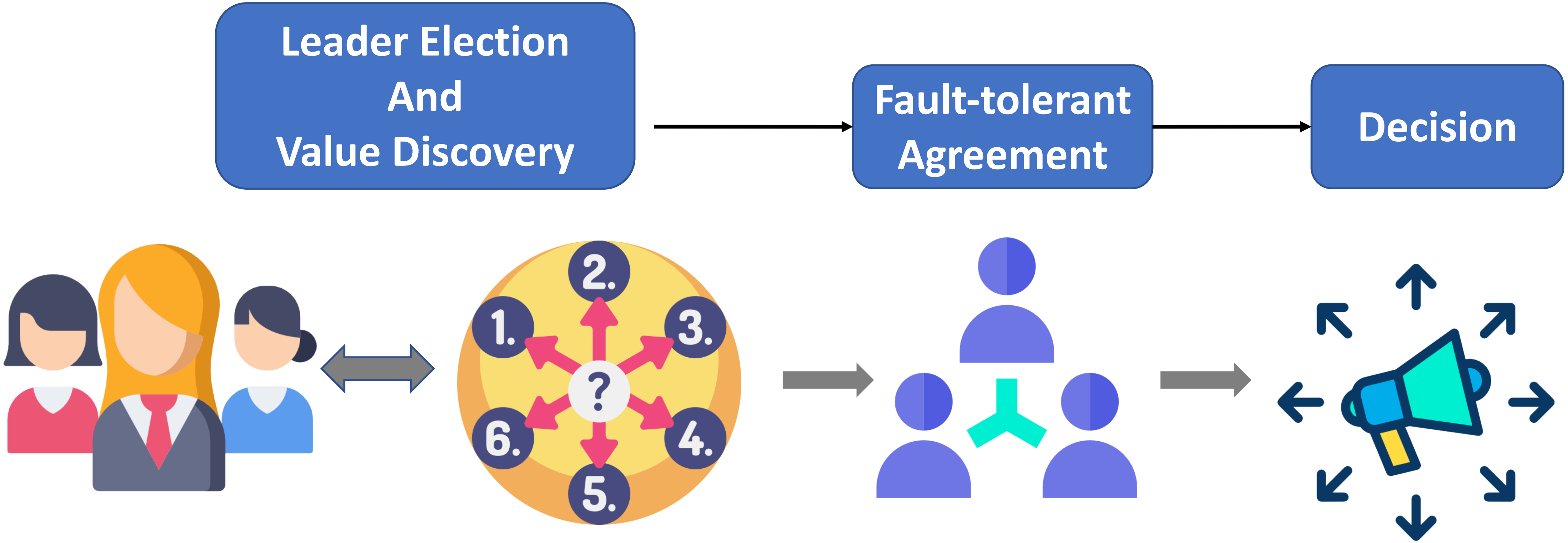
- If leader fails: Elect new leader and execute termination protocol



**Leader Election And Value Discovery** → **Fault-tolerant Agreement**

# Fault-tolerant 3PC (with Termination)

- If leader fails: Elect new leader and execute termination protocol



Leader Election And Value Discovery

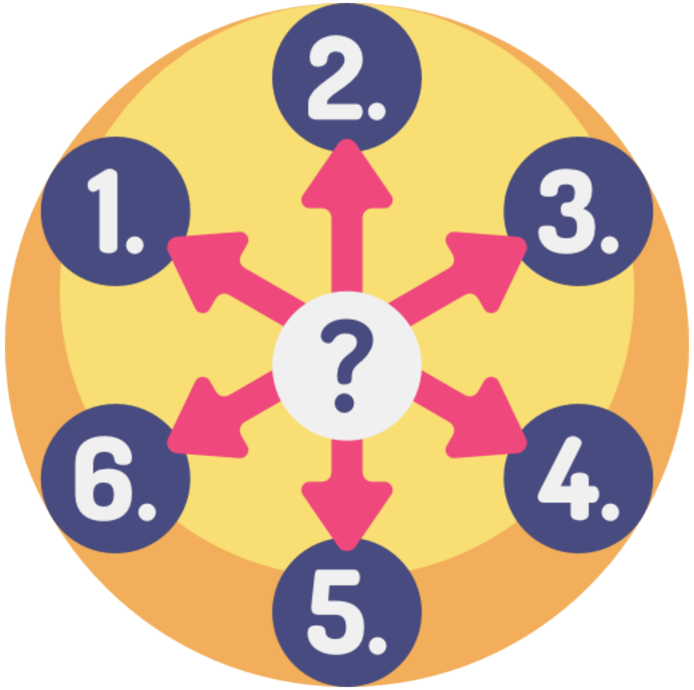
Fault-tolerant Agreement

Decision

# Common phases observed?

- Paxos and 2PC/3PC are **leader**-based protocols

# Common phases observed?



- Paxos and 2PC/3PC are **leader**-based protocols
- Agreement on **a single** value is the main goal

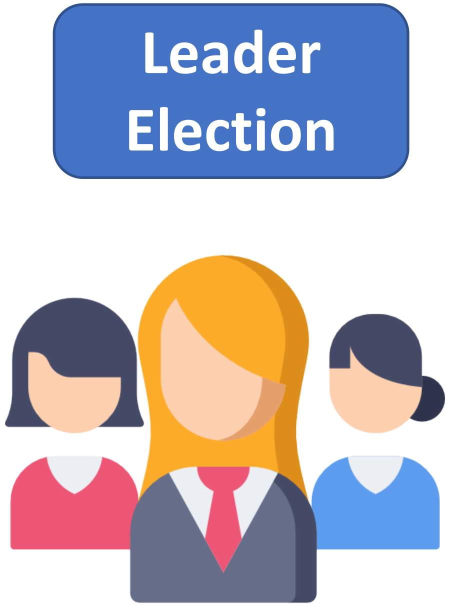
# Common phases observed?

- Paxos and 2PC/3PC are **leader**-based protocols
- Agreement on **a single** value is the main goal
- Both protocols ensure **fault tolerance** on the decided value

# Common phases observed?

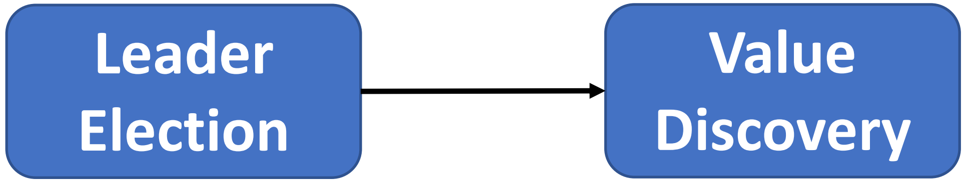
- Paxos and 2PC/3PC are **leader**-based protocols
- Agreement on **a single** value is the main goal
- Both protocols ensure **fault tolerance** on the decided value
- Disseminate the **decision**, typically asynchronously

# Consensus & Commitment (C&C) Framework

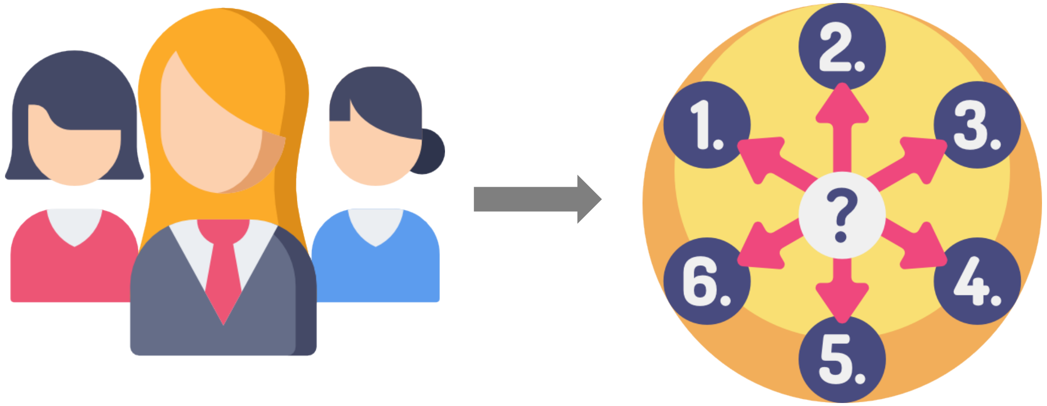


Maiyya, S., Nawab, F., Agrawal, D., & Abbadi, A. E. Unifying consensus and atomic commitment for effective cloud data management. VLDB, 2019.

# Consensus & Commitment (C&C) Framework


Leader Election

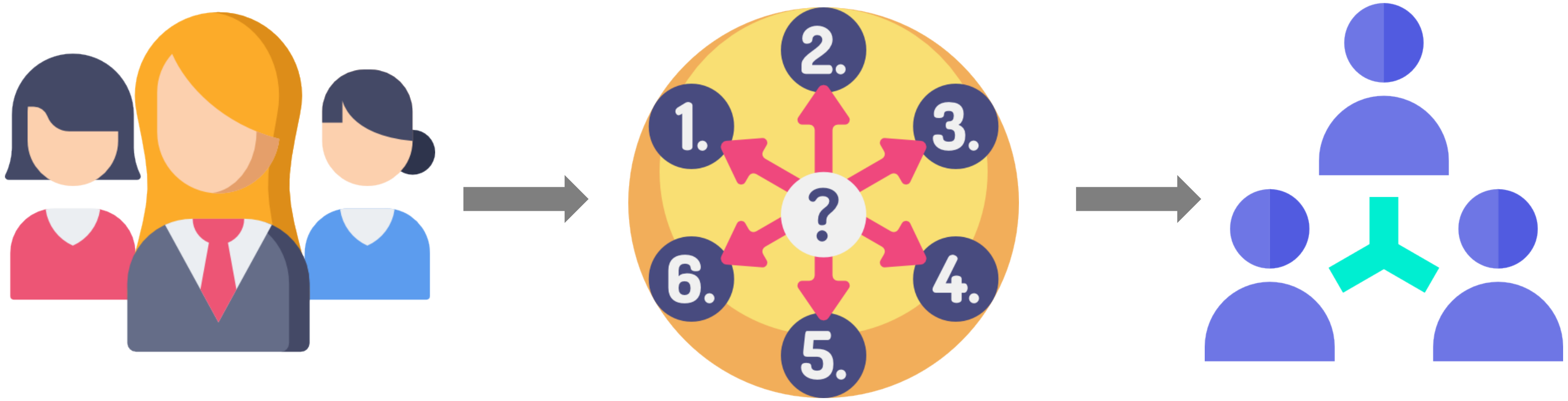
Value Discovery



Maiyya, S., Nawab, F., Agrawal, D., & Abbadi, A. E. Unifying consensus and atomic commitment for effective cloud data management. VLDB, 2019.

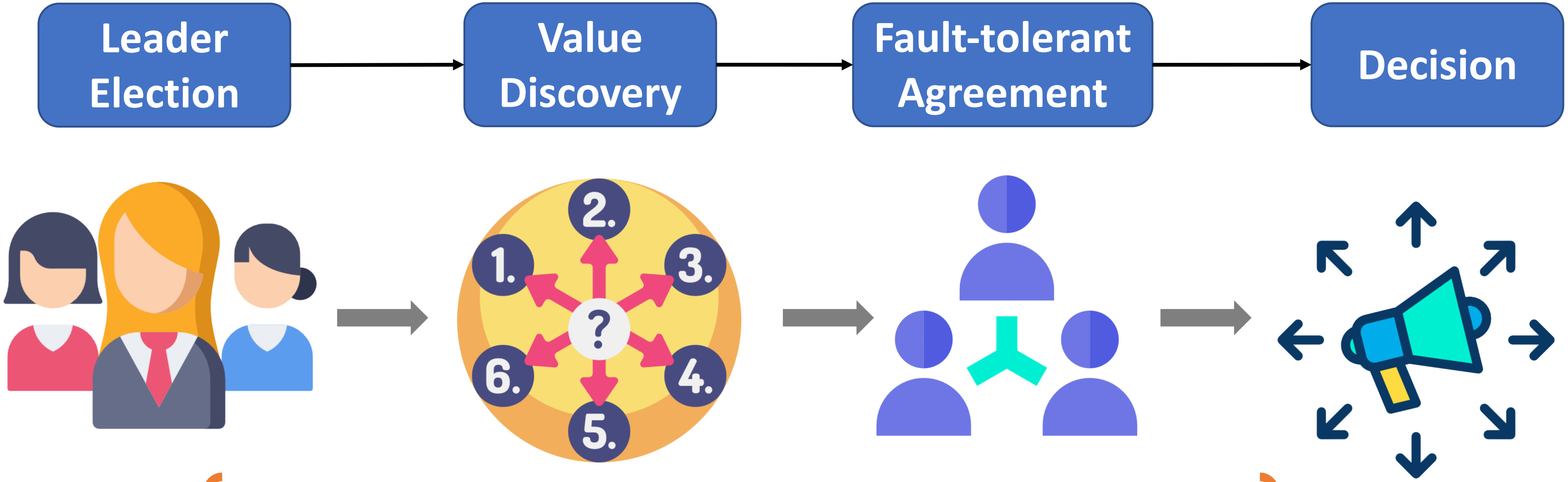
# Consensus & Commitment (C&C) Framework

Leader Election → Value Discovery → Fault-tolerant Agreement



Maiyya, S., Nawab, F., Agrawal, D., & Abbadi, A. E. Unifying consensus and atomic commitment for effective cloud data management. VLDB, 2019.

# Consensus & Commitment (C&C) Framework



Maiyya, S., Nawab, F., Agrawal, D., & Abbadi, A. E. Unifying consensus and atomic commitment for effective cloud data management. VLDB, 2019.

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings

**Three Phase Commit**

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings

**Three Phase Commit**

Leader Election

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings

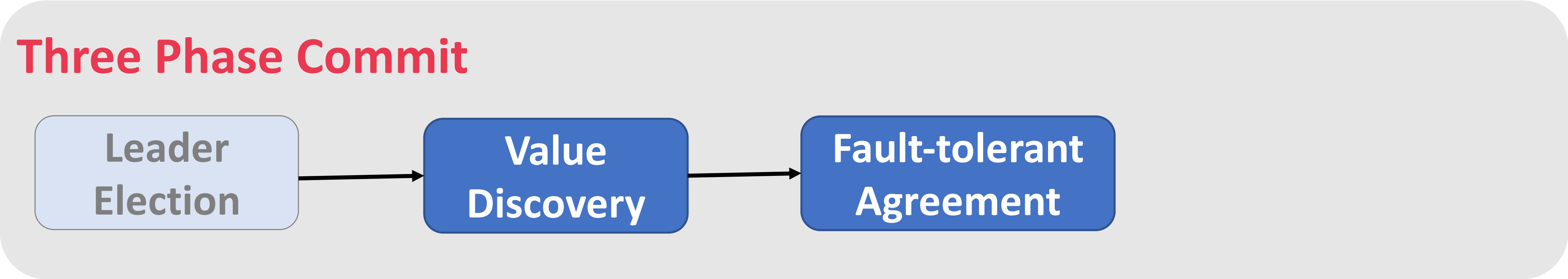
Three Phase Commit

Leader Election

Value Discovery

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings



Three Phase Commit

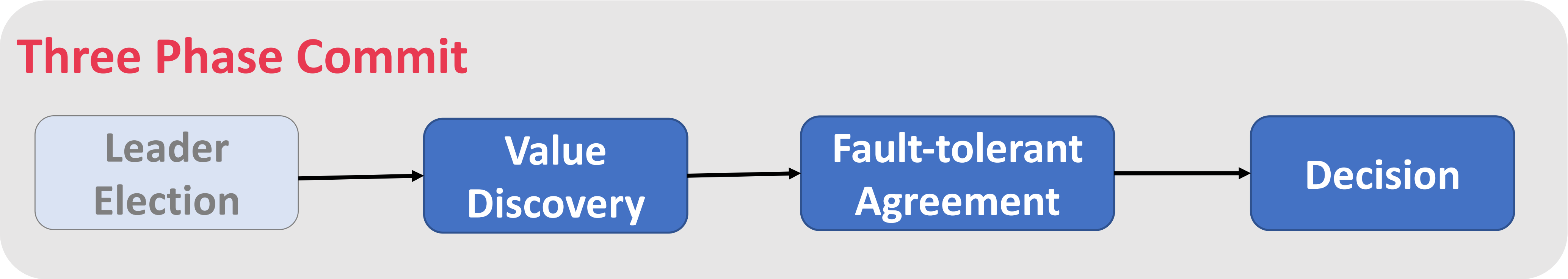
Leader Election

Value Discovery

Fault-tolerant Agreement

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings



Three Phase Commit

Leader Election

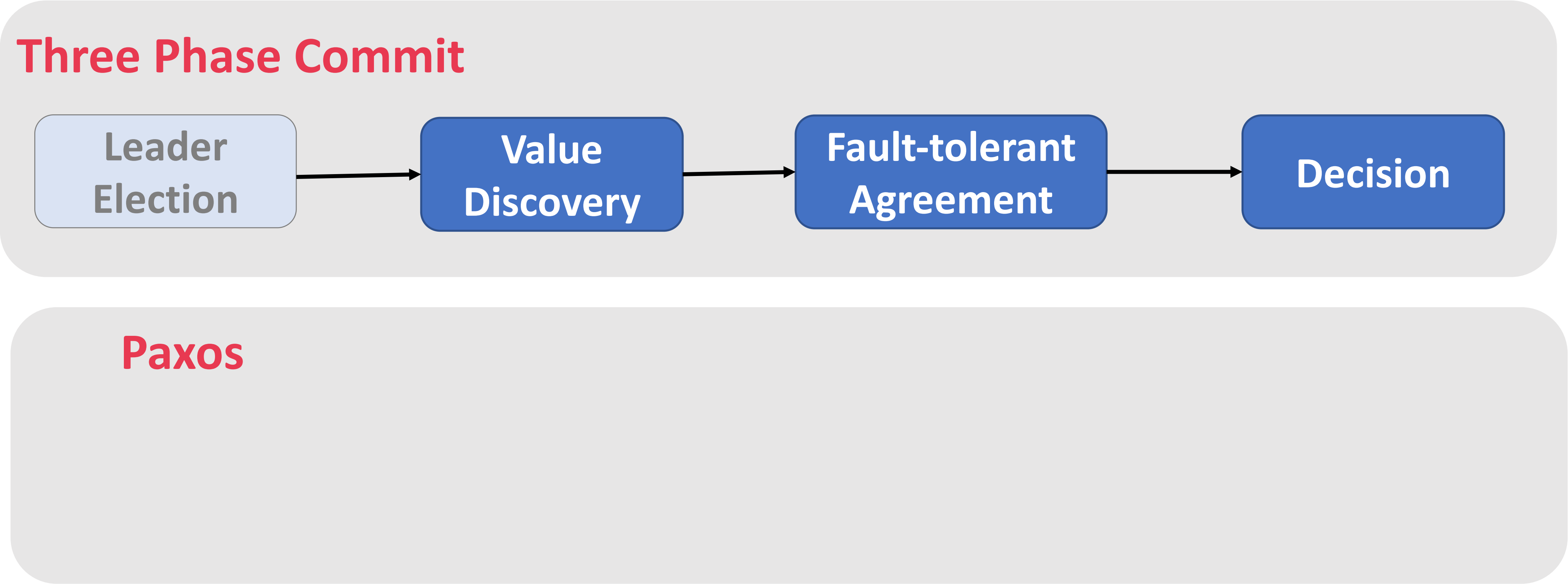
Value Discovery

Fault-tolerant Agreement

Decision

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings



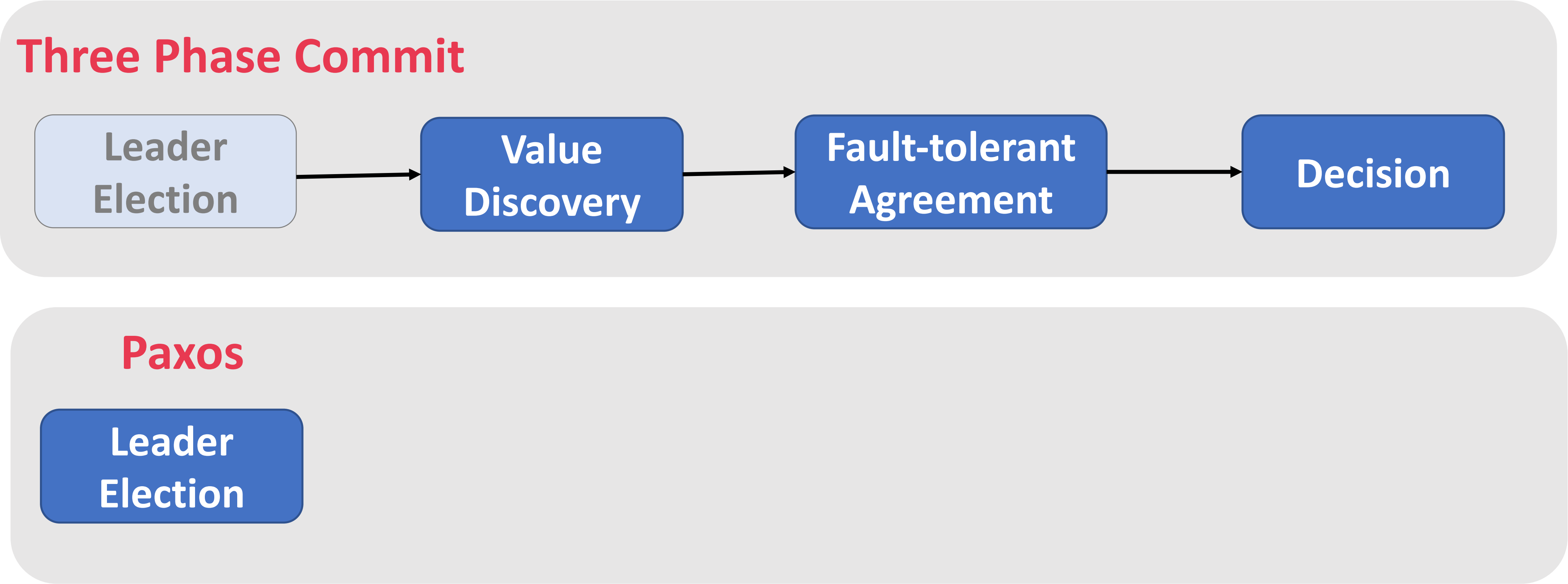
Three Phase Commit

Leader Election → Value Discovery → Fault-tolerant Agreement → Decision

Paxos

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings



Three Phase Commit

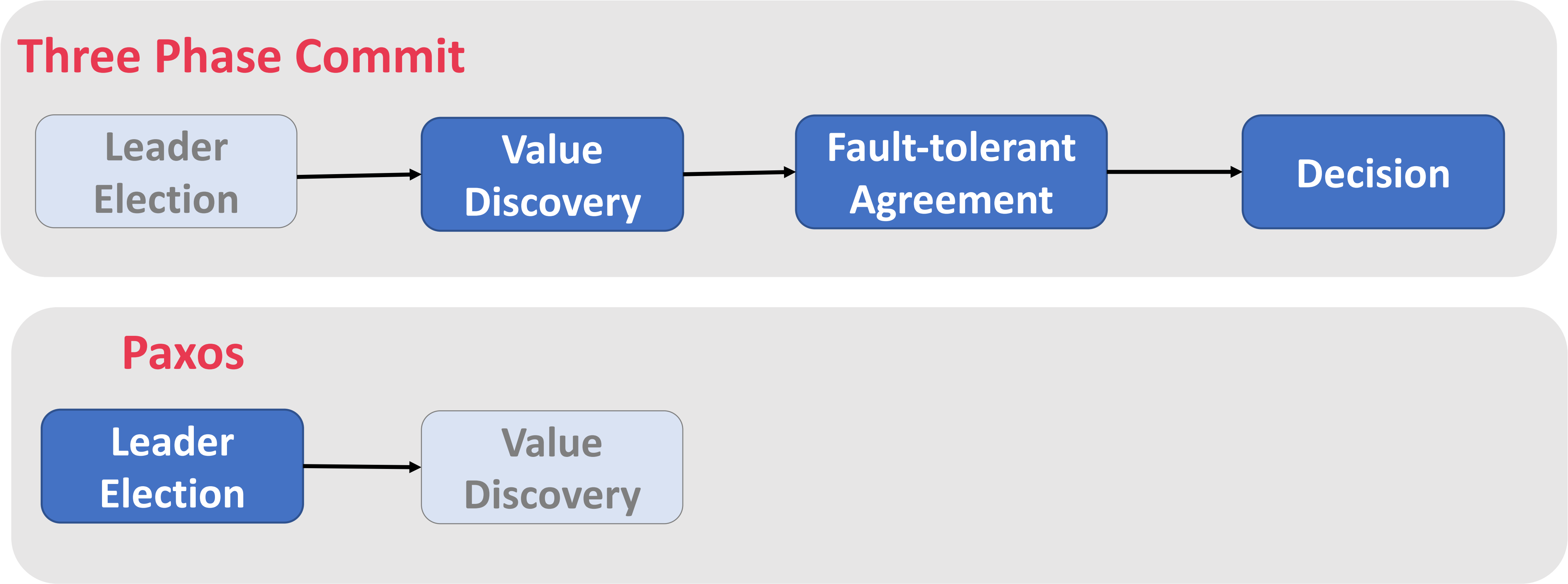
Leader Election → Value Discovery → Fault-tolerant Agreement → Decision

Paxos

Leader Election

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings



Three Phase Commit

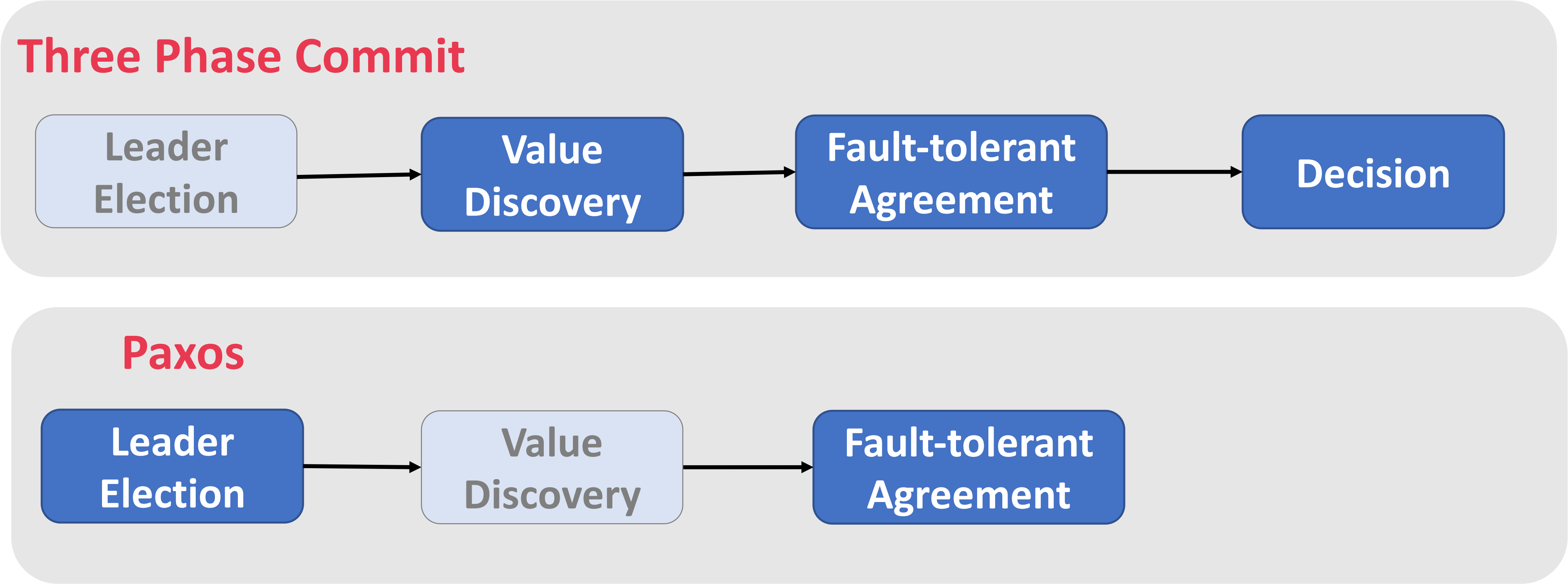
Leader Election → Value Discovery → Fault-tolerant Agreement → Decision

Paxos

Leader Election → Value Discovery

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings



Three Phase Commit

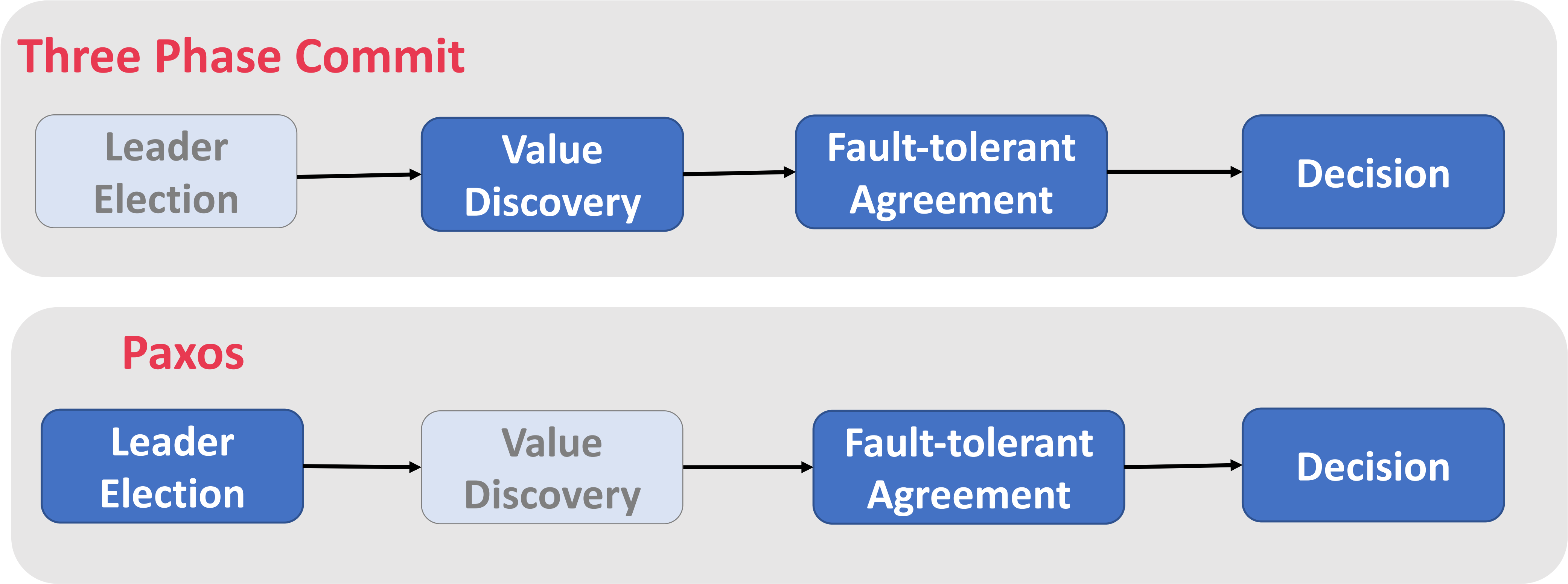
Leader Election → Value Discovery → Fault-tolerant Agreement → Decision

Paxos

Leader Election → Value Discovery → Fault-tolerant Agreement

# C&C Framework

- A pedagogical tool to understand many existing protocols
- Helps us develop insights for protocols in novel settings



Three Phase Commit

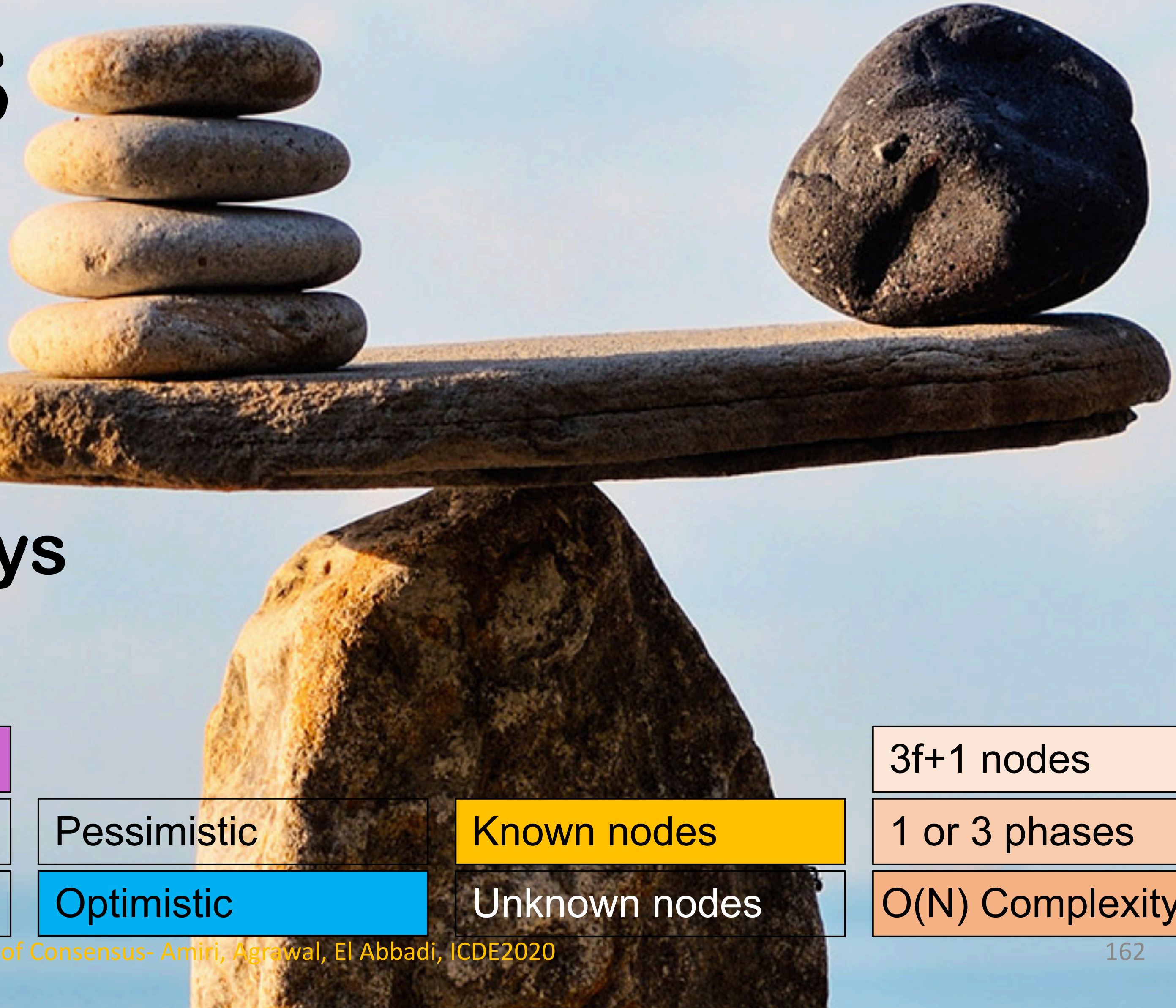
Leader Election → Value Discovery → Fault-tolerant Agreement → Decision

Paxos

Leader Election → Value Discovery → Fault-tolerant Agreement → Decision

# Fast Paxos

Lamport, Leslie. "Fast paxos."
Distributed Computing, 2006



**Reduce messages delays**
**Sacrifice quorum size**

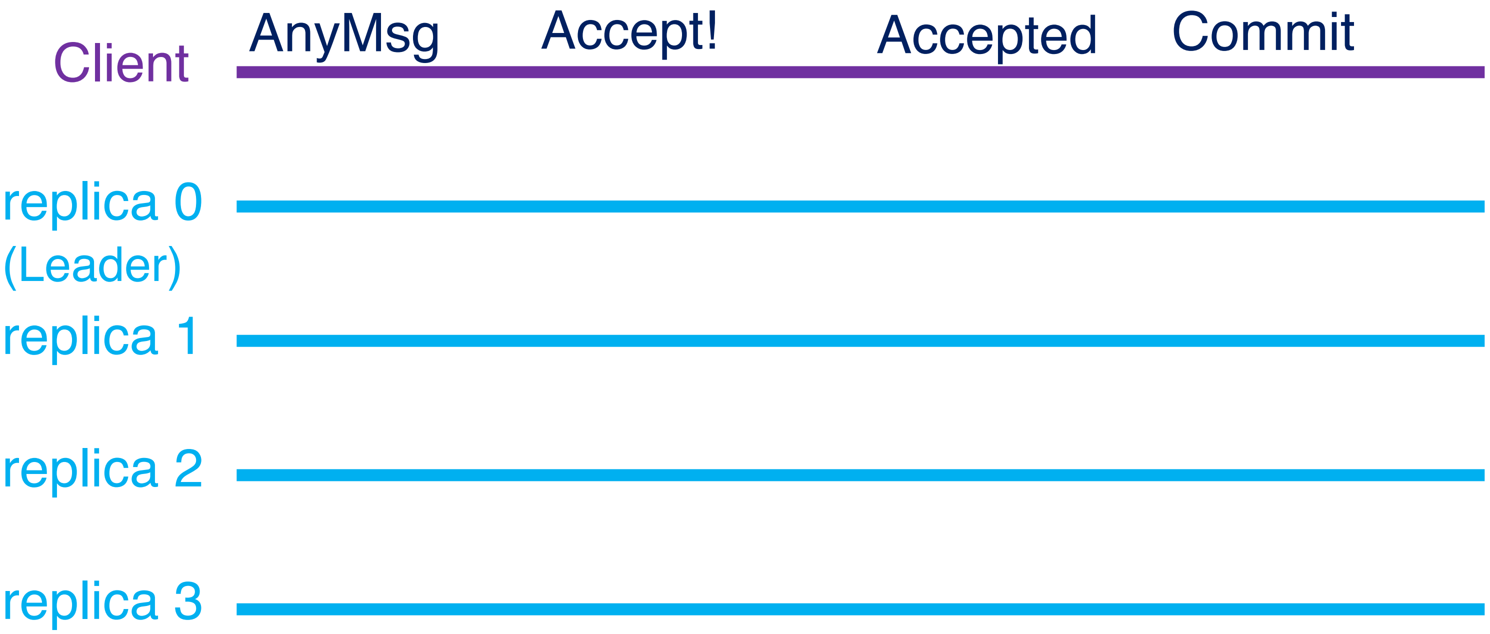
| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | 3f+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 1 or 3 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O(N) Complexity |

# Fast Paxos

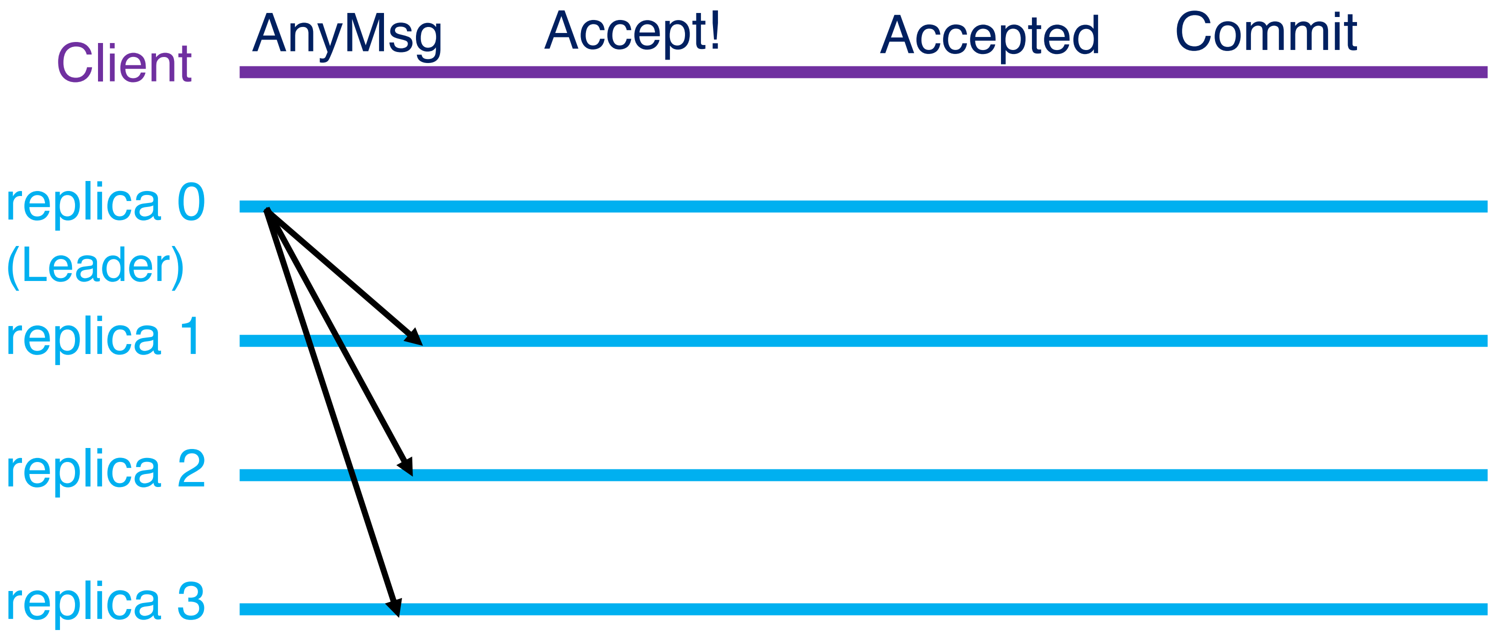
- Generalizes Basic Paxos to reduce end-to-end message delays.
- Basic Paxos: 3 message delays from client request to learning
- Fast Paxos allows 2 message delays where
  1. the system includes *3f+1* nodes (instead of 2f+1)
  2. the Client sends its request to multiple destinations.

- Intuition:
  - If the leader has no value to propose, a client sends an *Accept!* to all nodes.
  - Backups respond as in Basic Paxos, sending *Accepted* messages to the leader

# Fast Round

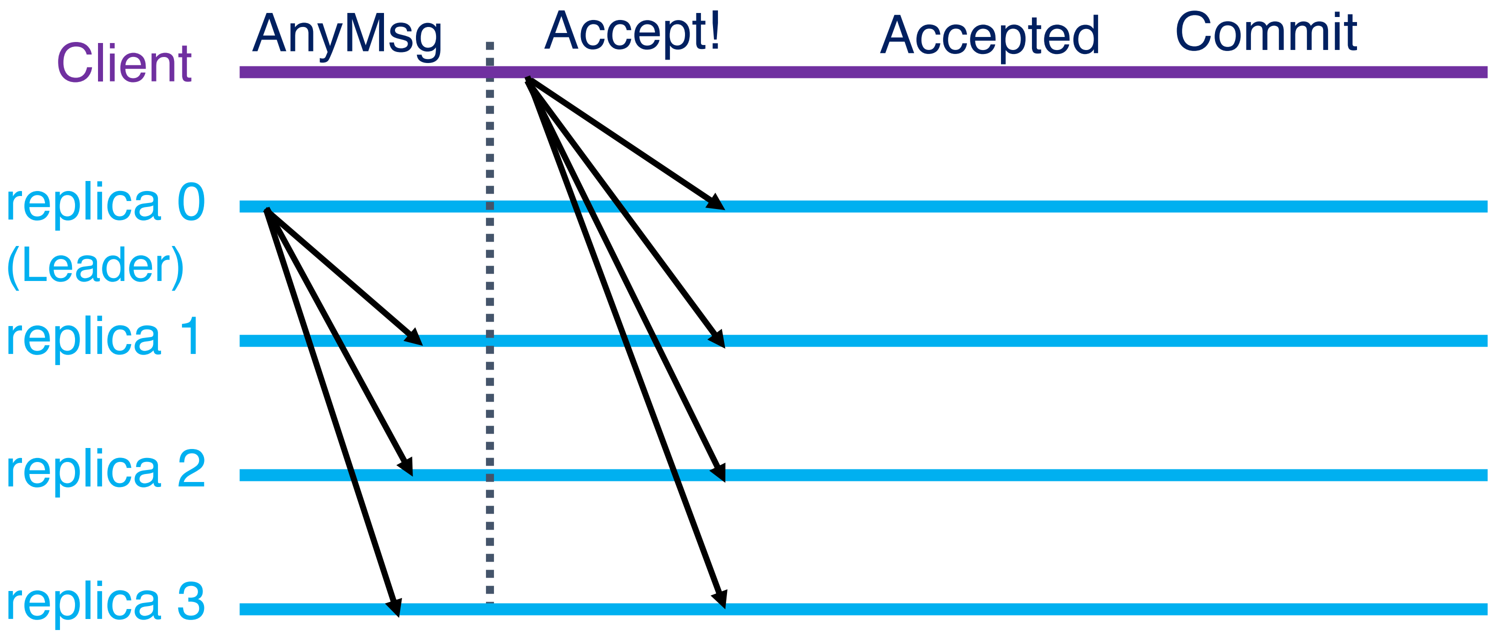

AnyMsg
Accept!
Accepted
Commit
Client
replica 0
(Leader)
replica 1
replica 2
replica 3

# Fast Round



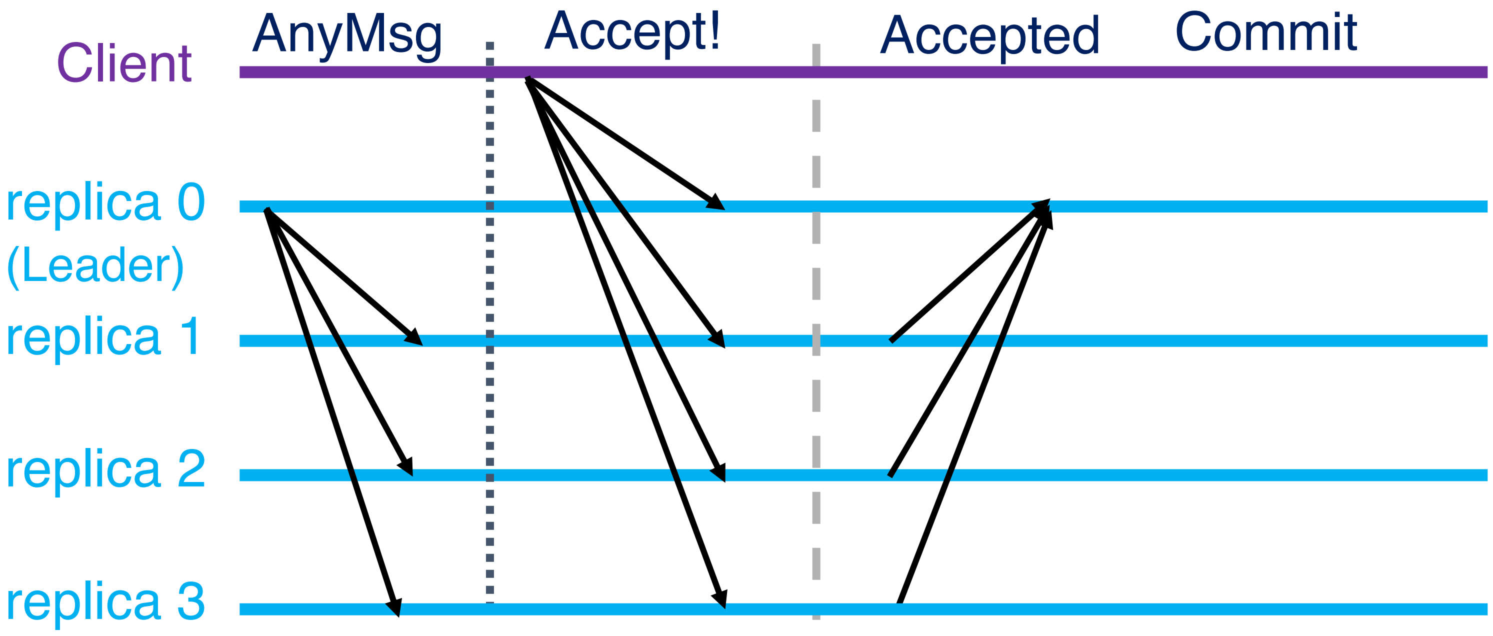
Any Message enables a backup to select its own value (proposed by a client)

# Fast Round



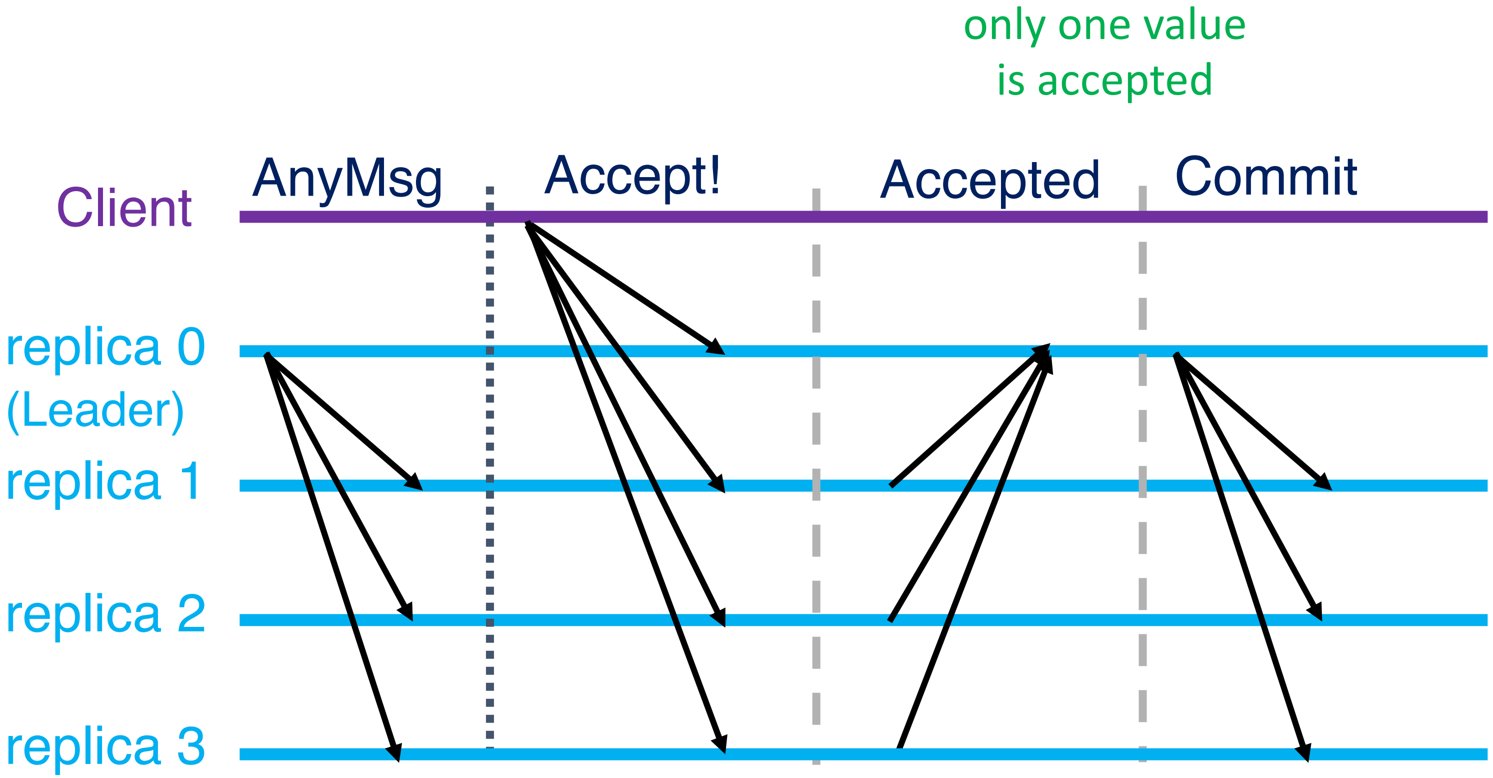
Any Message enables a backup to select its own value (proposed by a client)

# Fast Round



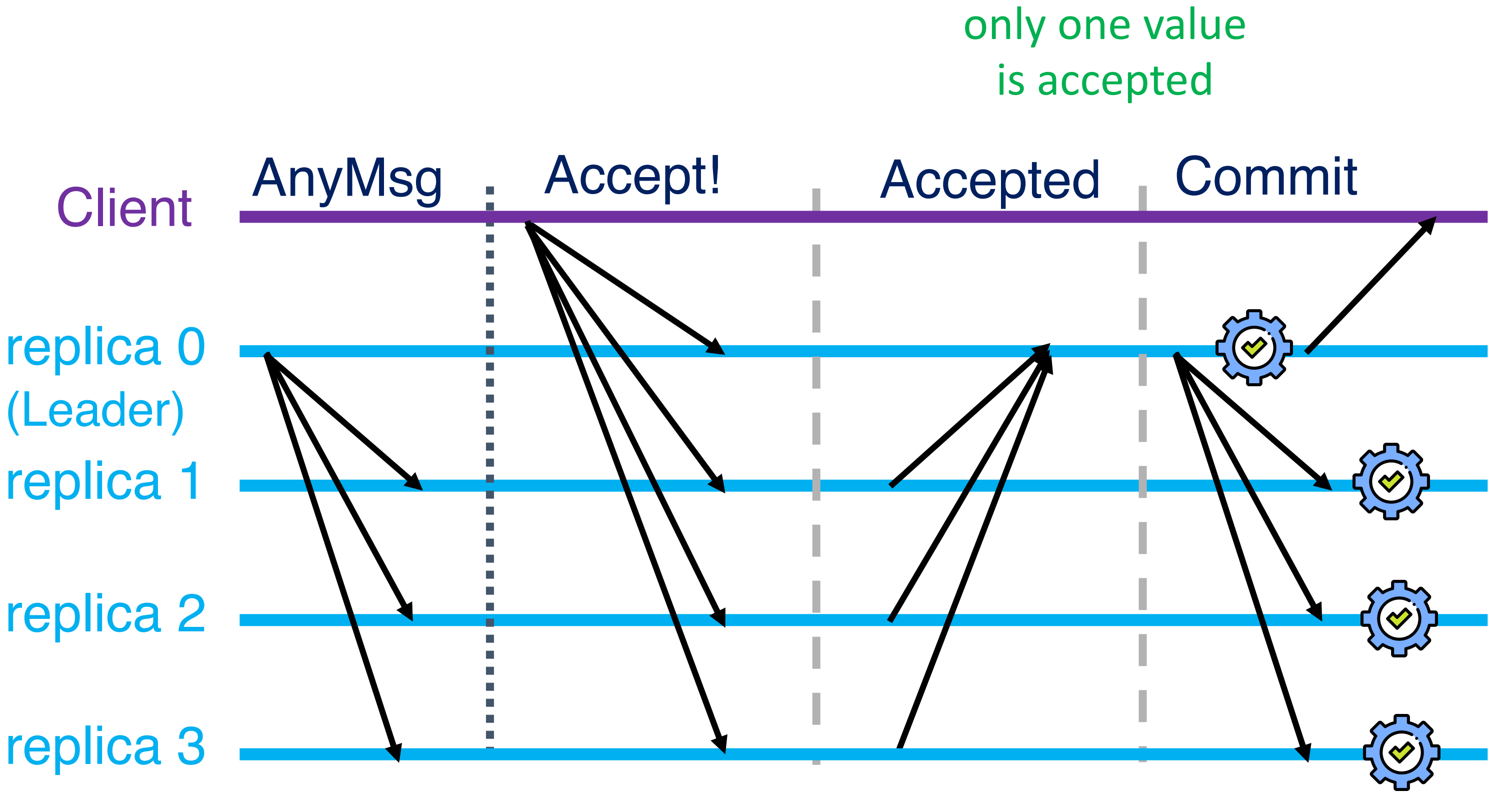
Any Message enables a backup to select its own value (proposed by a client)

# Fast Round



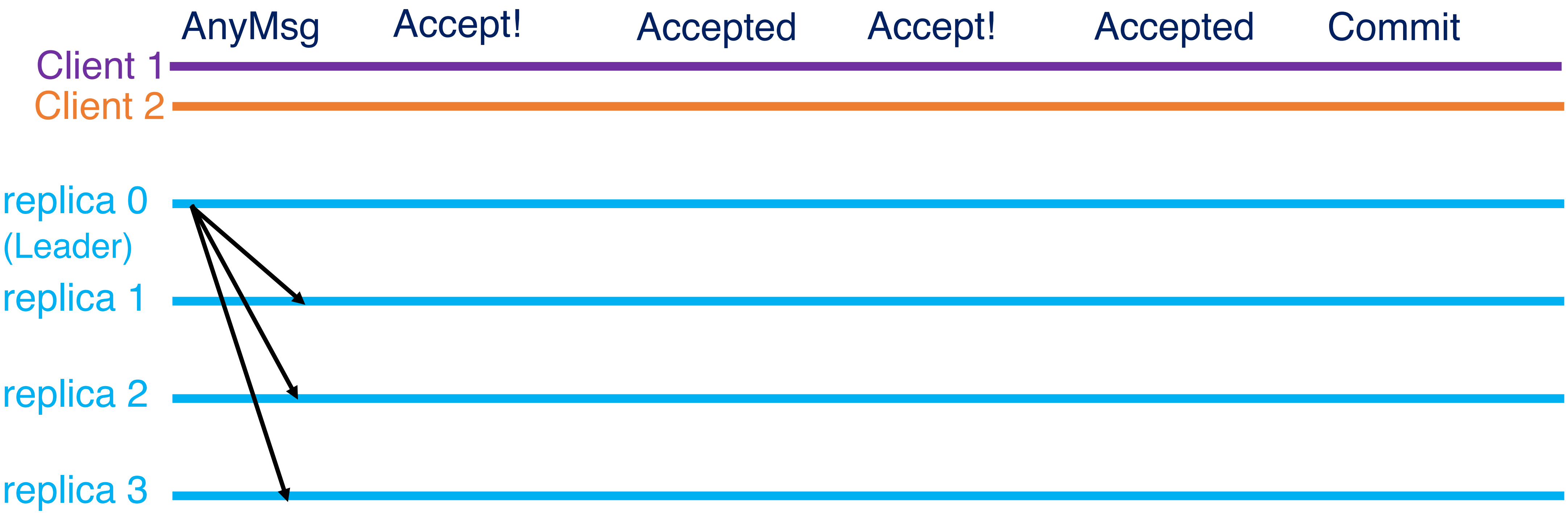
Any Message enables a backup to select its own value (proposed by a client)

# Fast Round

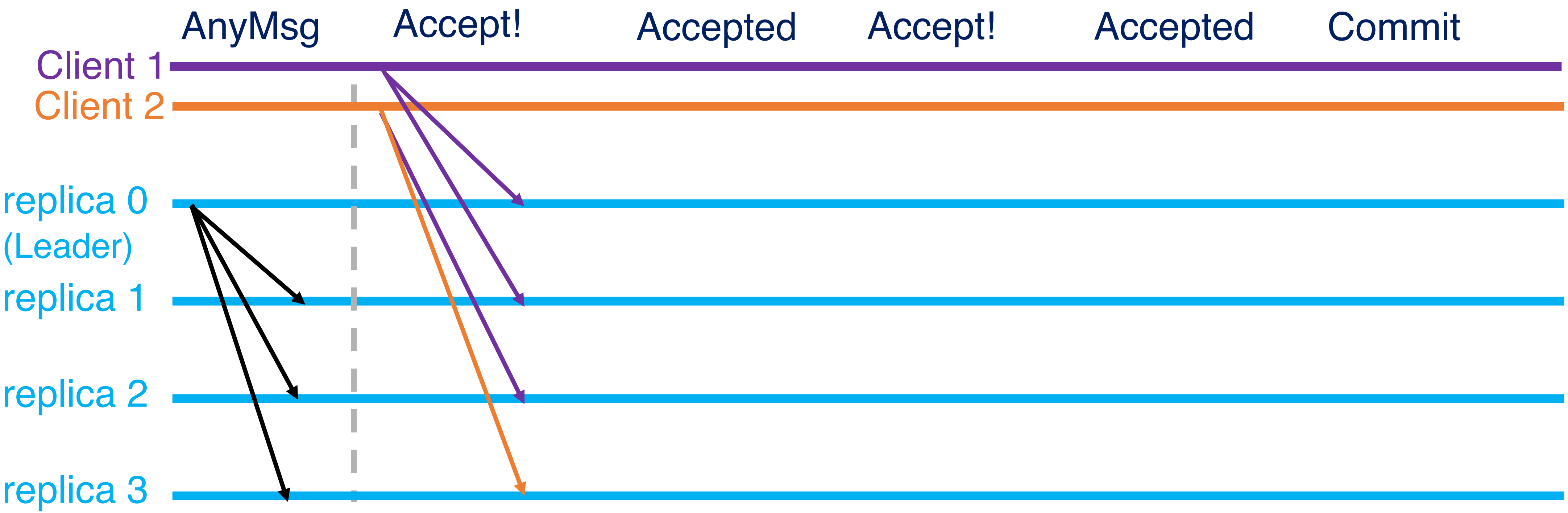


Any Message enables a backup to select its own value (proposed by a client)

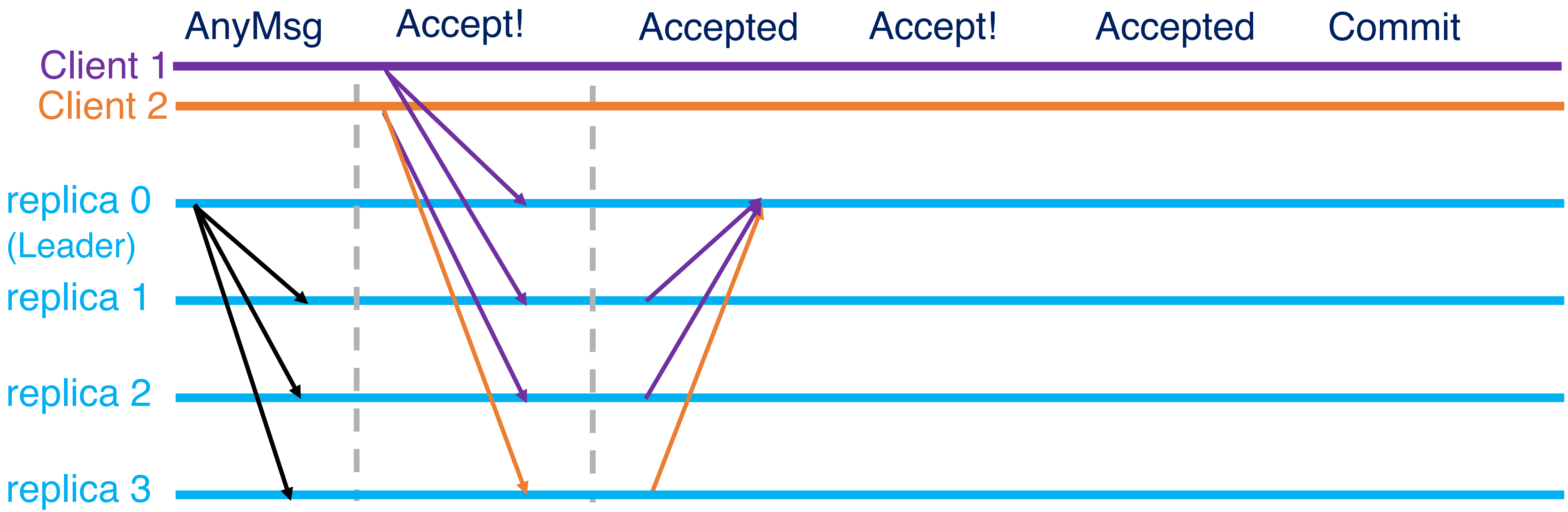
# Collision Happens!



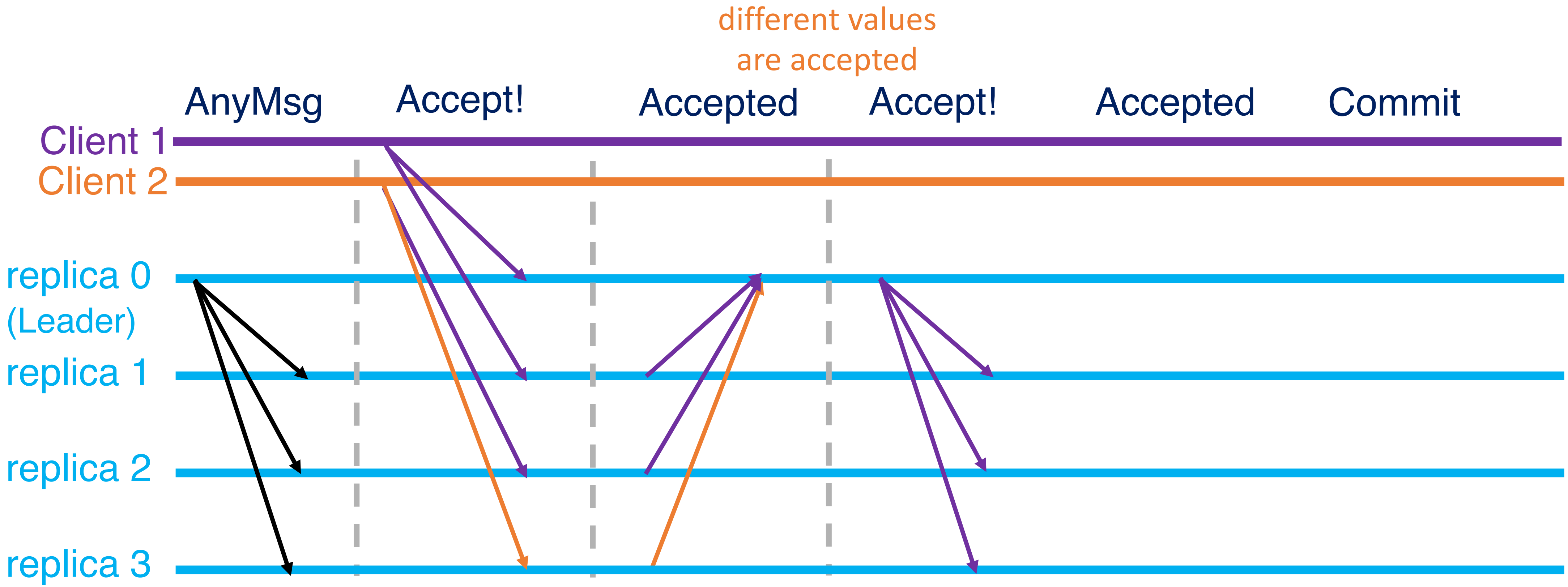
# Collision Happens!



# Collision Happens!

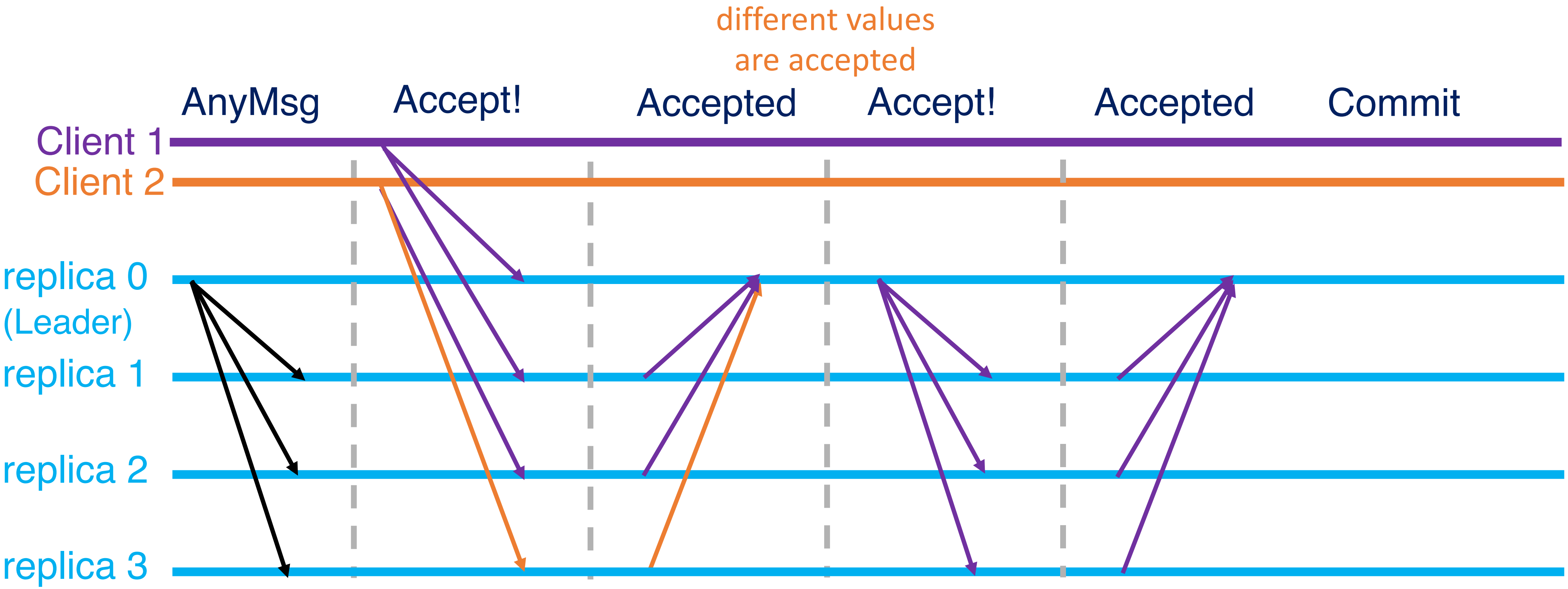


# Collision Happens!



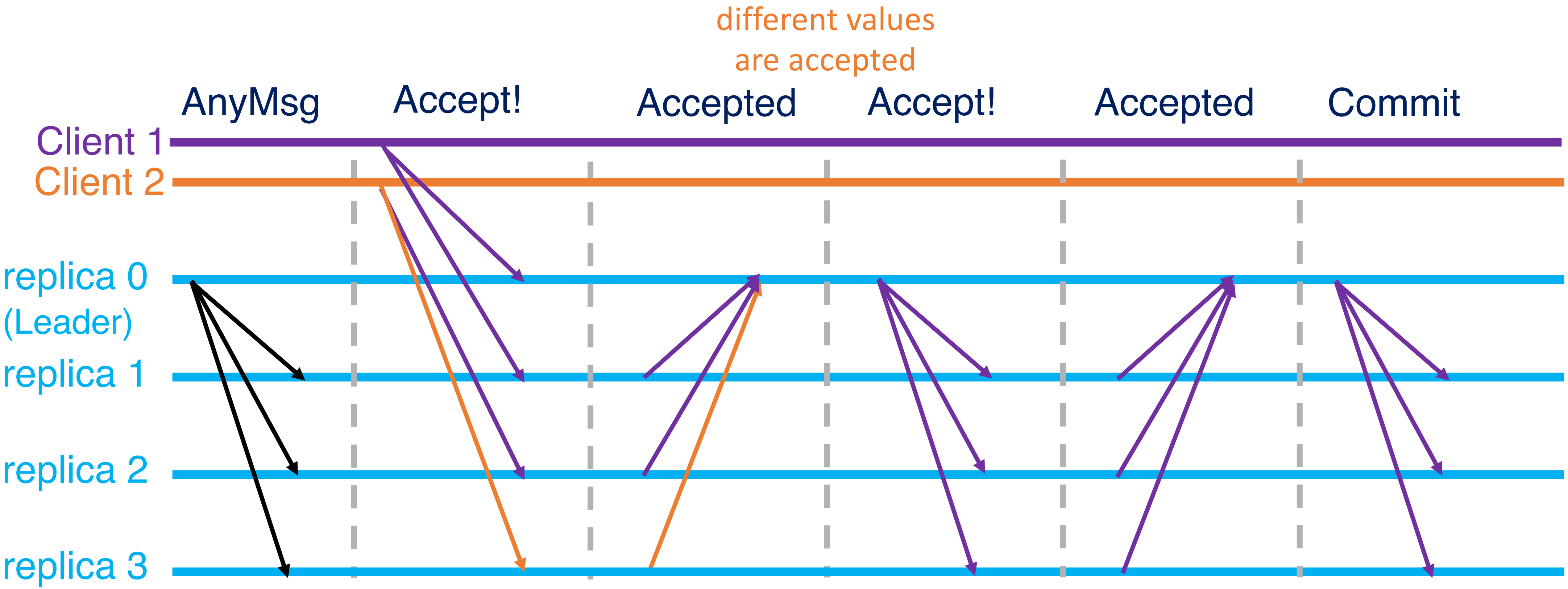
Chooses the value with the majority quorum if exists

# Collision Happens!



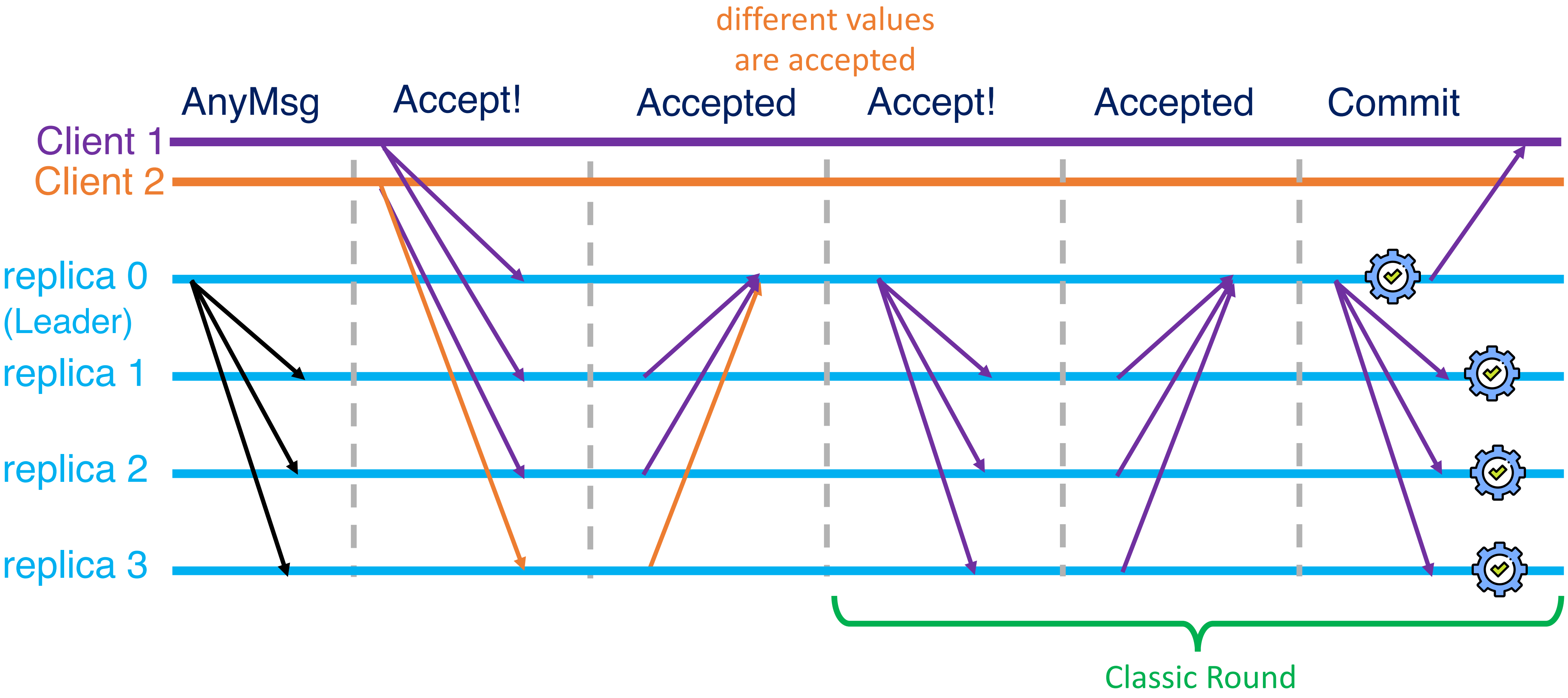
Chooses the value with the majority quorum if exists

# Collision Happens!



Chooses the value with the majority quorum if exists

# Collision Happens!

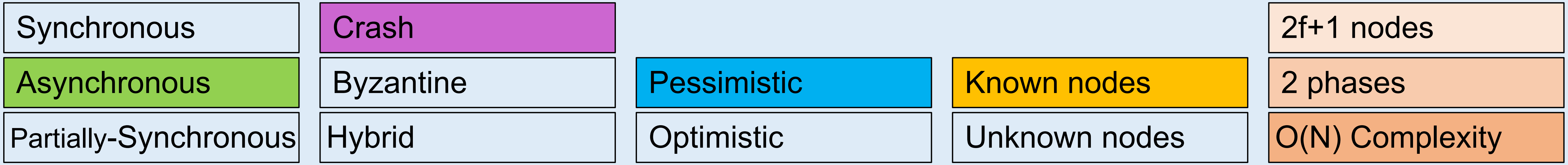


Chooses the value with the majority quorum if exists

# Flexible Paxos

Howard, H., Malkhi, D., & Spiegelman, A. Flexible Paxos: Quorum Intersection Revisited. In *OPODIS, 2017*

**It is not necessary to require all quorums in Paxos to intersect**



| | | | | |
|---|---|---|---|---|
| Synchronous | Crash 🟪 | | | 2f+1 nodes |
| Asynchronous 🟩 | Byzantine | Pessimistic 🟦 | Known nodes 🟨 | 2 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O(N) Complexity |

# Flexible Paxos

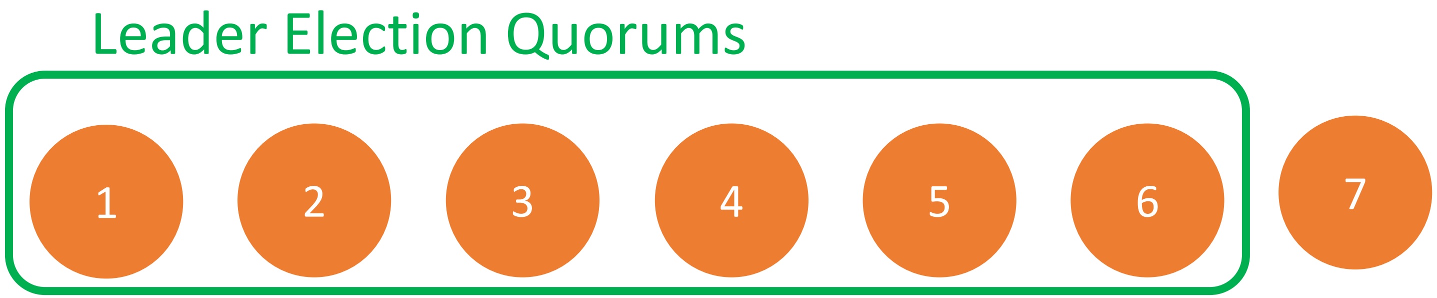
- Majority quorums for BOTH Leader Election AND Replication are too conservative
- Generalized Quorum Condition: only Leader Election Quorums and Replication Quorums must intersect.
  - Decouple Leader Election Quorums from Replication Quorums
  - Arbitrarily small replication quorums as long as Leader Election Quorums intersect with every Replication Quorum
- No changes to Paxos algorithms

# Flexible Paxos

- Majority quorums for BOTH Leader Election AND Replication are too conservative
- Generalized Quorum Condition: only Leader Election Quorums and Replication Quorums must intersect.
  - Decouple Leader Election Quorums from Replication Quorums
  - Arbitrarily small replication quorums as long as Leader Election Quorums intersect with every Replication Quorum
- No changes to Paxos algorithms

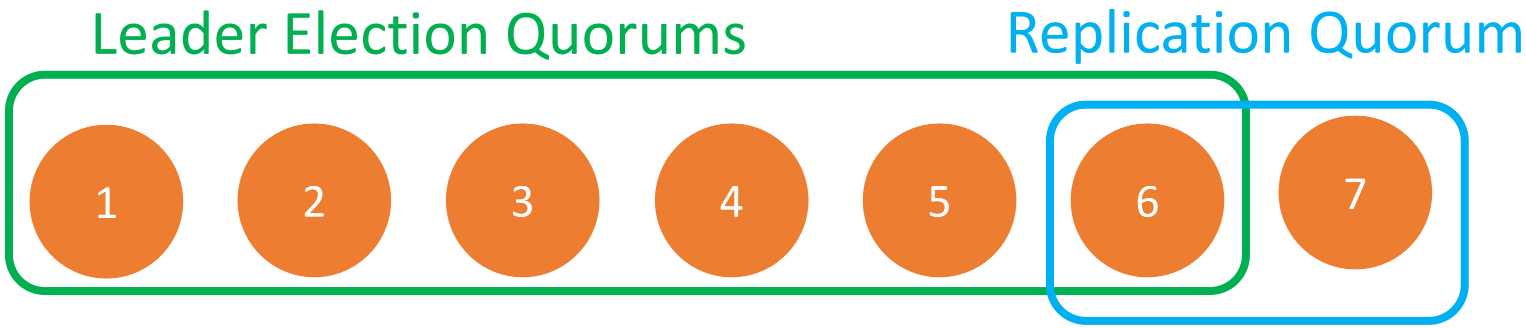
# Flexible Paxos

- Majority quorums for BOTH Leader Election AND Replication are too conservative
- Generalized Quorum Condition: only Leader Election Quorums and Replication Quorums must intersect.
  - Decouple Leader Election Quorums from Replication Quorums
  - Arbitrarily small replication quorums as long as Leader Election Quorums intersect with every Replication Quorum
- No changes to Paxos algorithms

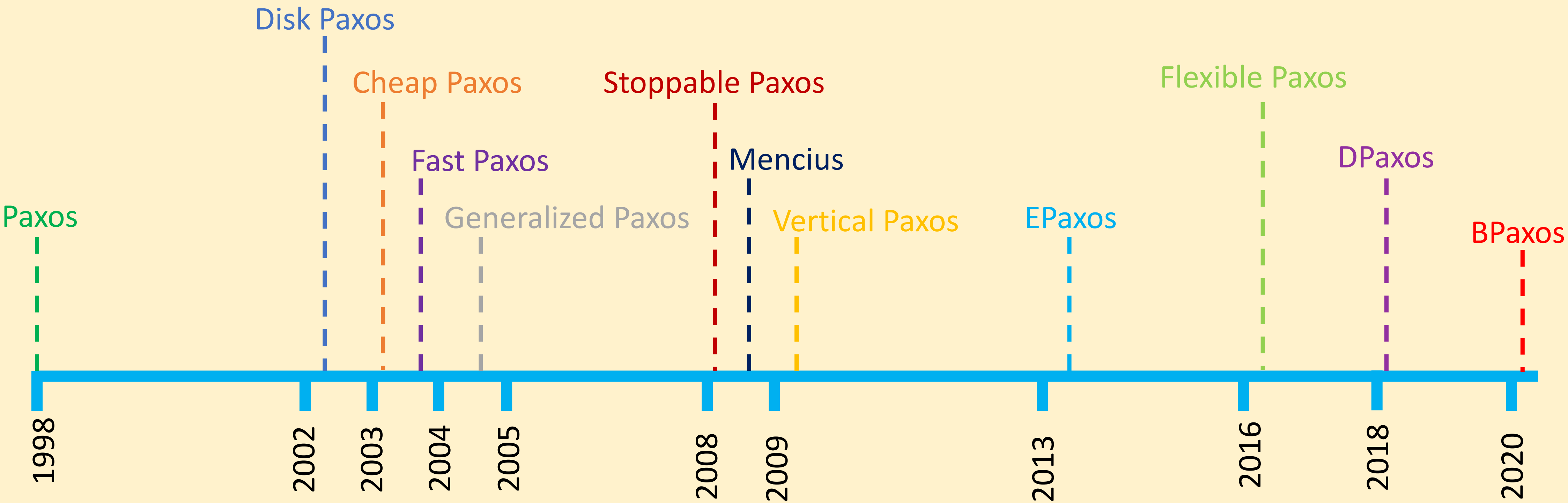


# Flexible Paxos

- Majority quorums for BOTH Leader Election AND Replication are too conservative
- Generalized Quorum Condition: only Leader Election Quorums and Replication Quorums must intersect.
  - Decouple Leader Election Quorums from Replication Quorums
  - Arbitrarily small replication quorums as long as Leader Election Quorums intersect with every Replication Quorum
- No changes to Paxos algorithms



# Paxos Variants



Paxos

Disk Paxos

Cheap Paxos

Fast Paxos

Generalized Paxos

Stoppable Paxos

Mencius

Vertical Paxos

EPaxos

Flexible Paxos

DPaxos

BPaxos

1998

2002

2003

2004

2005

2008

2009

2013

2016

2018

2020

# Paxos in Real Systems

Google
MegaStore

Clustrix

ceph

Google Cloud
Spanner

Doozerd

Bing

OpenReplica

Google
Chubby

wanDISCO

neo4j

XTREEMFS

# Google Spanner

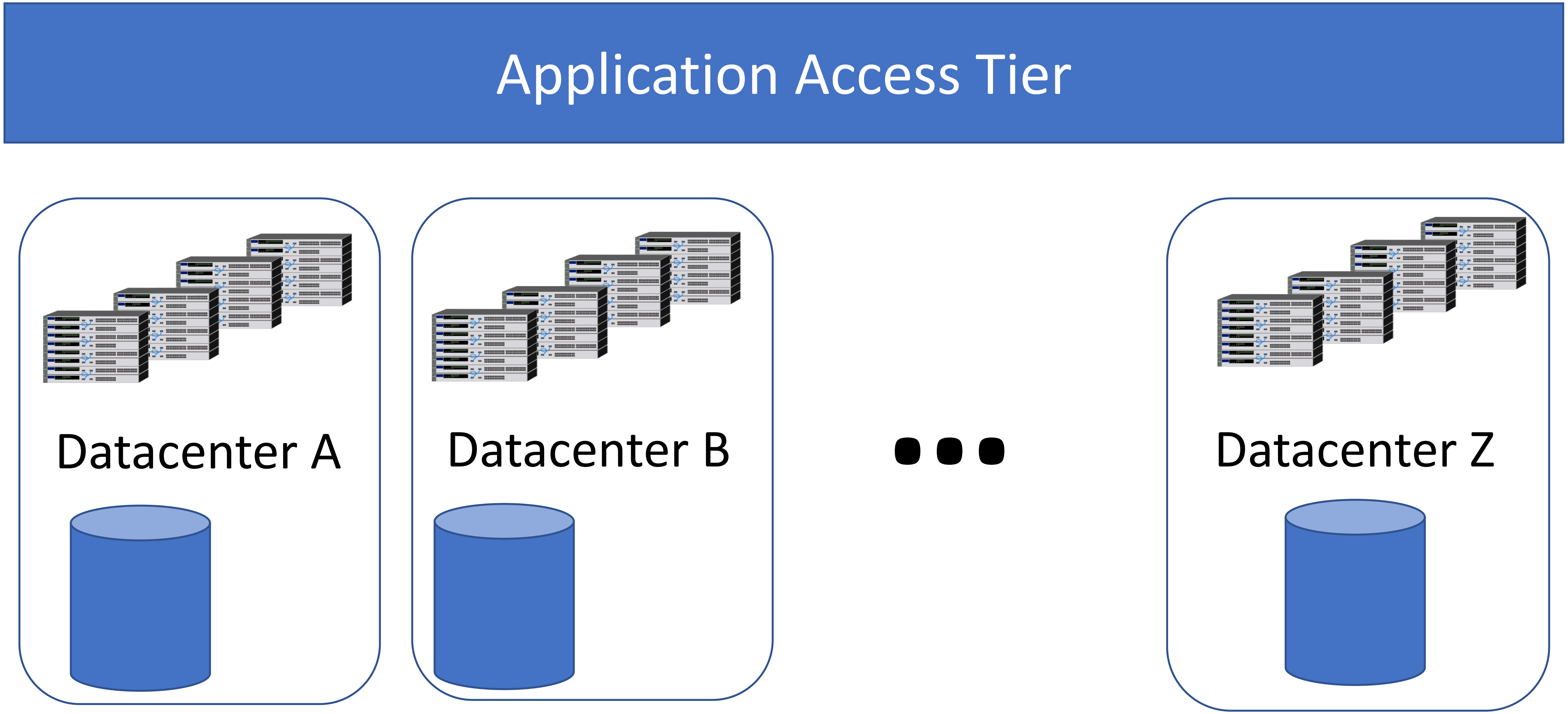
# Google Spanner

Google Cloud Spanner

Application Access Tier

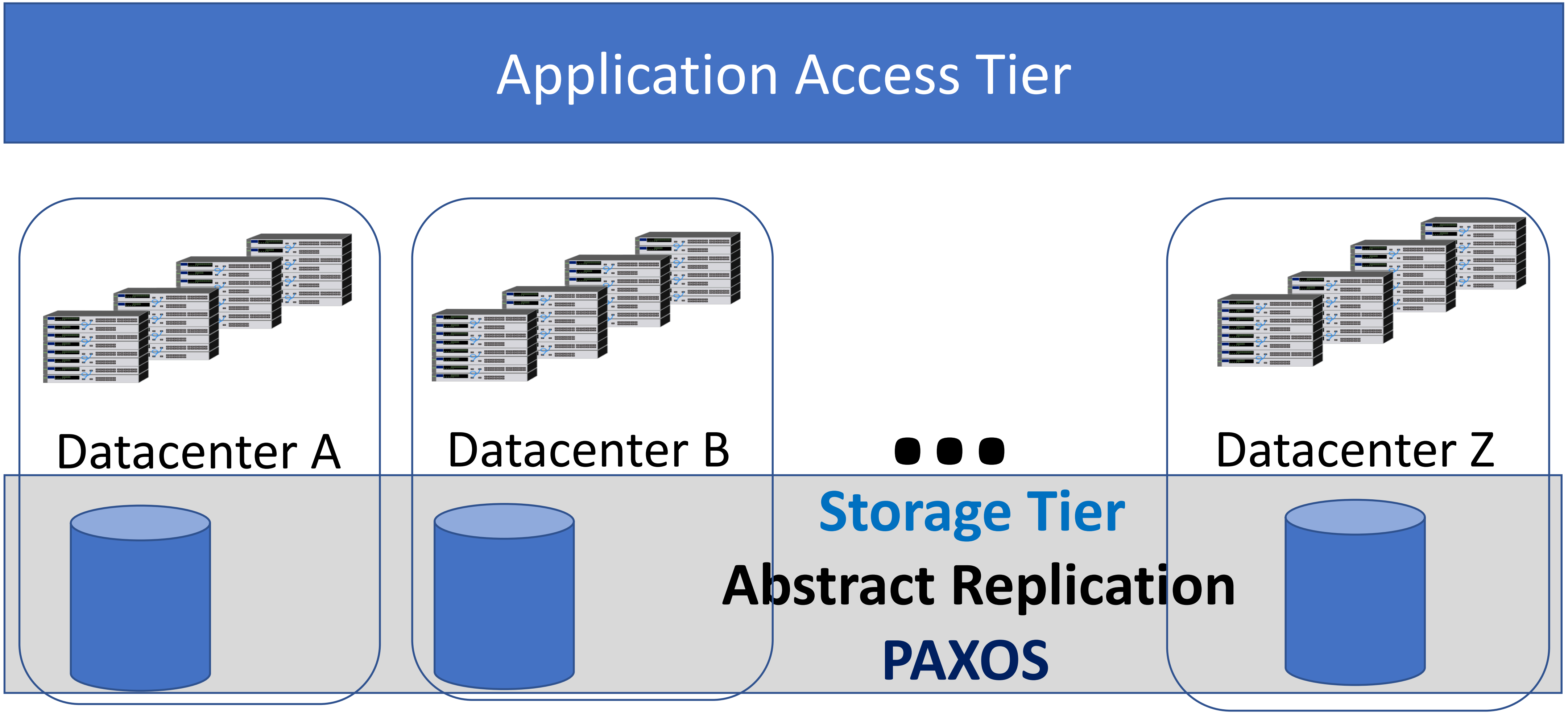
# Google Spanner

Google Cloud Spanner



# Google Spanner

Google Cloud Spanner



Application Access Tier

Datacenter A

Datacenter B

• • •

Datacenter Z

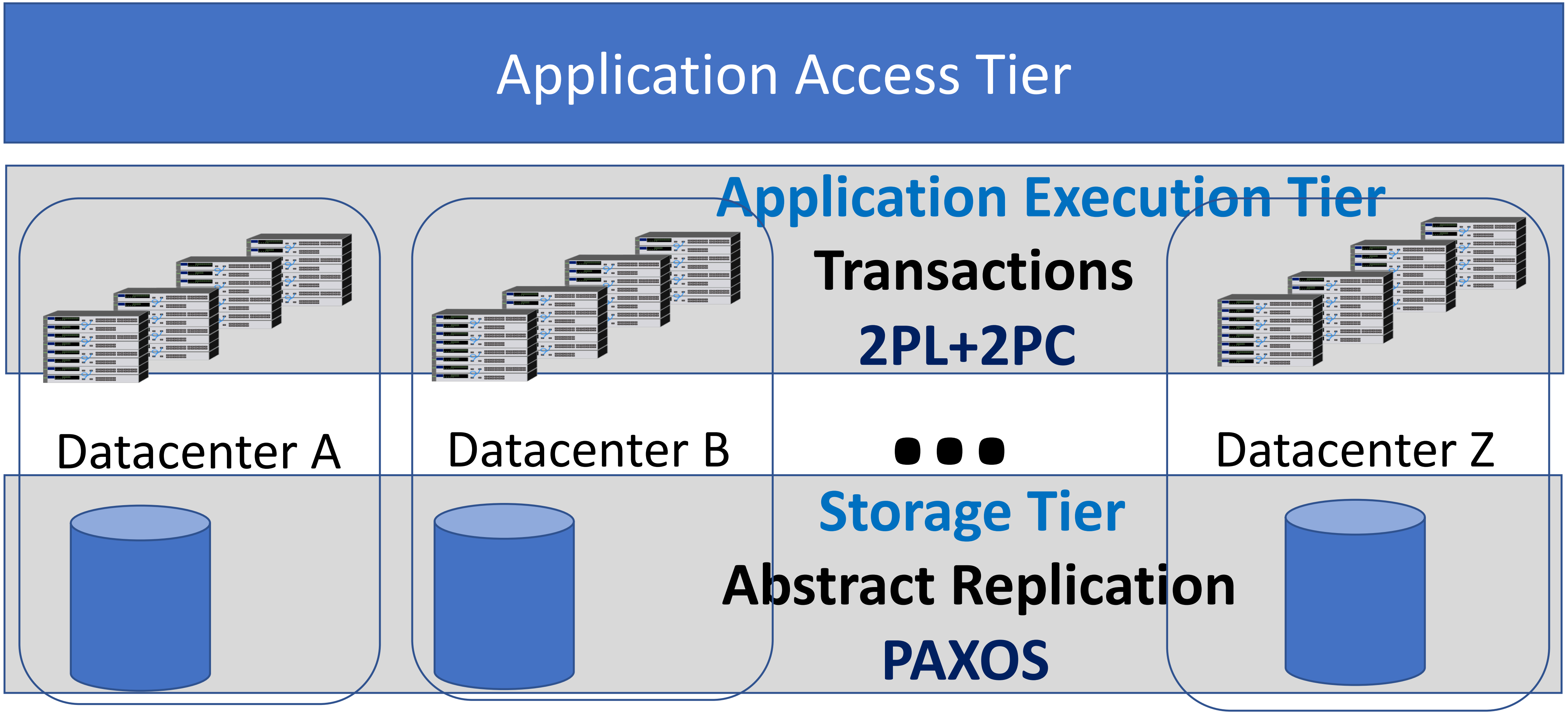
Storage Tier

**Abstract Replication**

**PAXOS**

# Google Spanner

Google Cloud Spanner



Application Access Tier

**Application Execution Tier**

**Transactions**

**2PL+2PC**

Datacenter A

Datacenter B

●●●

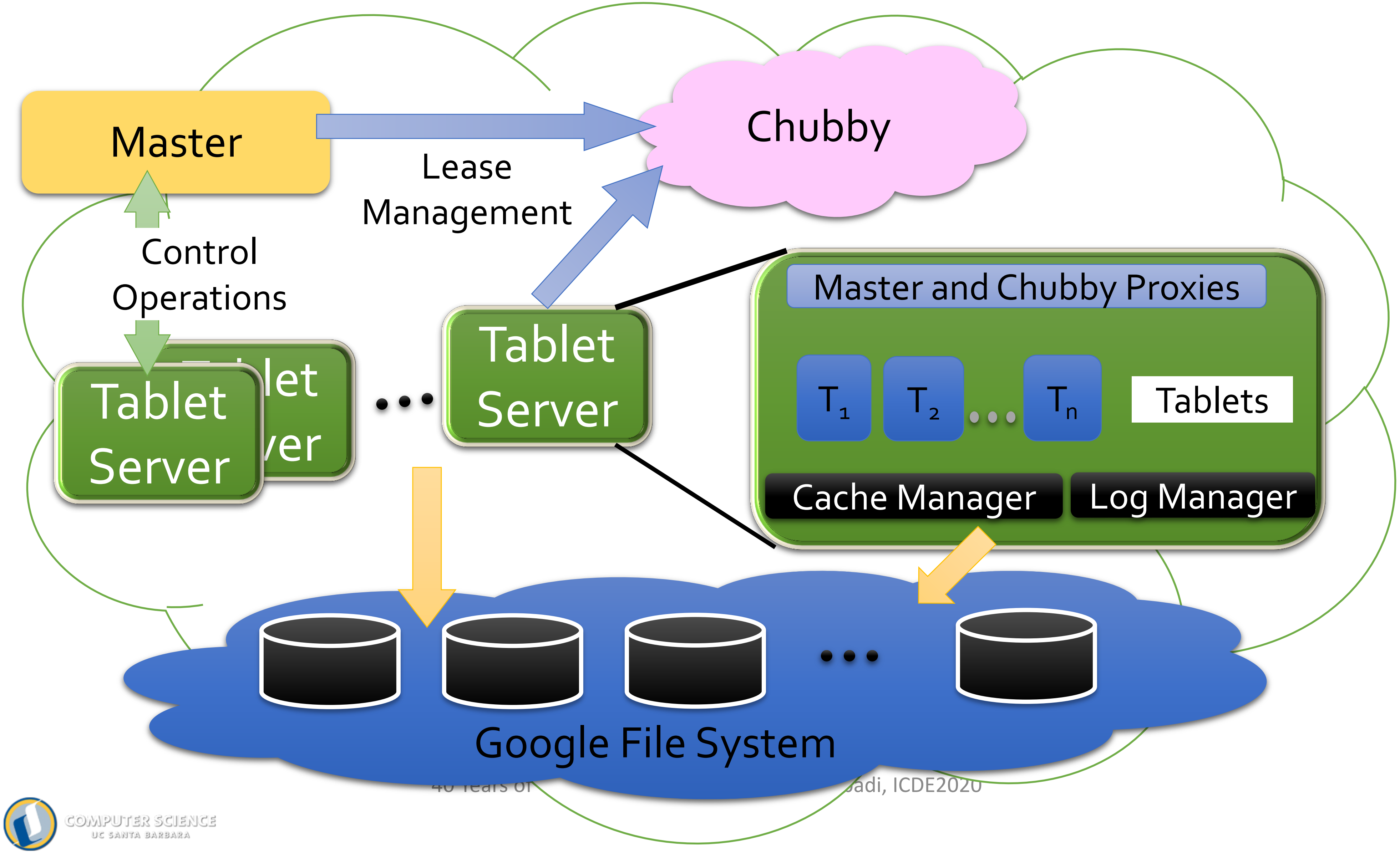
Datacenter Z

**Storage Tier**

**Abstract Replication**

**PAXOS**

# Google Bigtable



Master

Lease Management

Chubby

Control Operations

Tablet Server

Tablet Server

Tablet Server

Master and Chubby Proxies

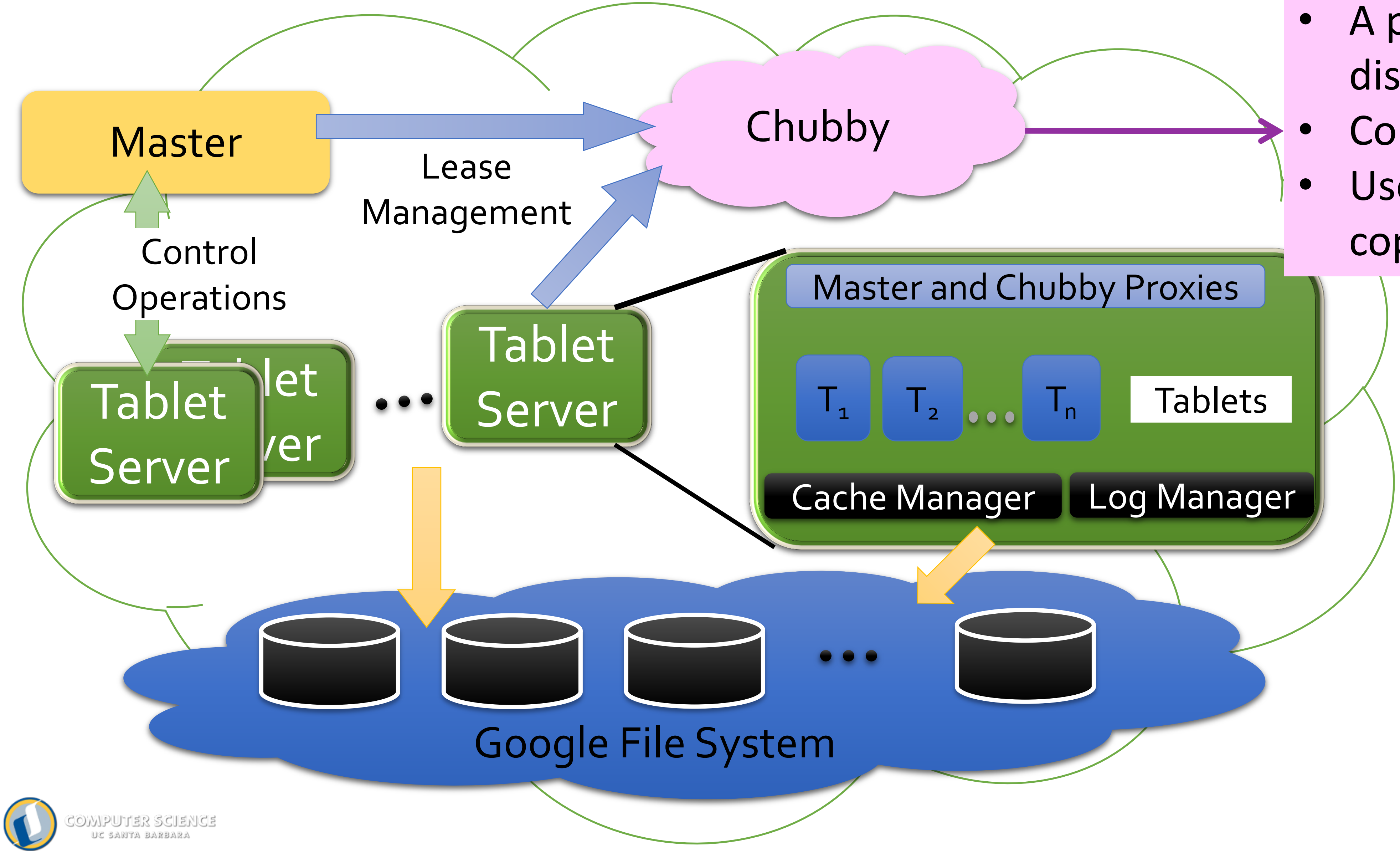
$T_1$ $T_2$ … $T_n$ Tablets

Cache Manager

Log Manager

Google File System

# Google Bigtable



- A persistent and distributed lock service
- Consists of 5 replicas
- Uses Paxos to keep copies consistent

# What if nodes behave maliciously?!

# Reaching Agreement in the Presence of Fault

Pease, Marshall, Robert Shostak, and Leslie Lamport. "Reaching agreement in the presence of faults. *Journal of the ACM (JACM), April* 1980

| | | | | |
|---|---|---|---|---|
| [✓ Synchronous] | Crash | | | [✓ 3f+1 nodes] |
| Asynchronous | [✓ Byzantine] | [✓ Pessimistic] | [✓ Known nodes] | [✓ f+1 phases] |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | [✓ $O(N^2)$ Complexity] |

# Reaching Agreement in the Presence of Fault

# Reaching Agreement in the Presence of Fault

- In a system with f faulty processes, an agreement can be achieved only if 2f+1 correctly functioning processes are present, for a total of 3f+1.
- i.e., An agreement is possible only if more than two-thirds of the processes are working properly.

# Reaching Agreement in the Presence of Fault

- In a system with f faulty processes, an agreement can be achieved only if 2f+1 correctly functioning processes are present, for a total of 3f+1.
- i.e., An agreement is possible only if more than two-thirds of the processes are working properly.
- Model:
  - Processes are synchronous
  - Messages are unicast while preserving ordering
  - Communication delay is bounded
  - There are *N* processes, where each process *i* will provide a value $v_i$ to the others
  - There are at most *f* faulty processes

# Reaching Agreement in the Presence of Fault

- In a system with f faulty processes, an agreement can be achieved only if 2f+1 correctly functioning processes are present, for a total of 3f+1.
- i.e., An agreement is possible only if more than two-thirds of the processes are working properly.
- Model:
  - Processes are synchronous
  - Messages are unicast while preserving ordering
  - Communication delay is bounded
  - There are *N* processes, where each process *i* will provide a value $v_i$ to the others
  - There are at most *f* faulty processes
- Each process *i* constructs a vector V of length N, such that
  - If process *i* is non-faulty, V[i] = i
  - Otherwise, V[*i*] is undefined

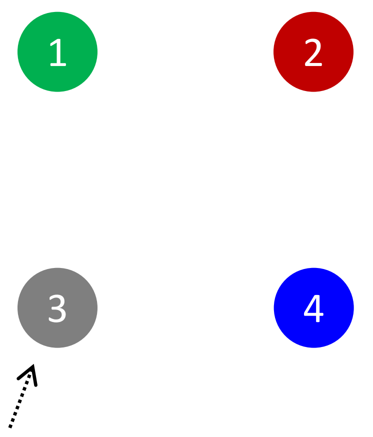
# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others

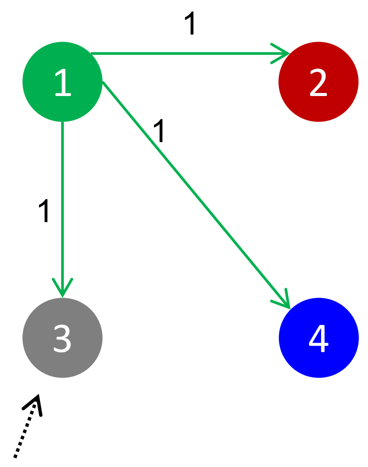


Faulty process

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others

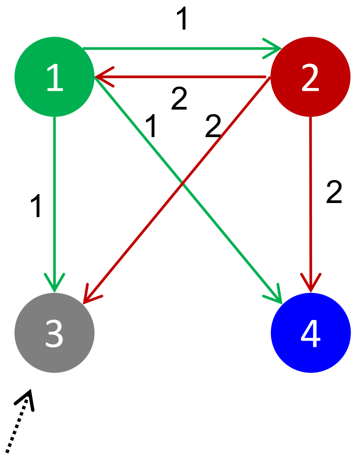


Faulty process

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others

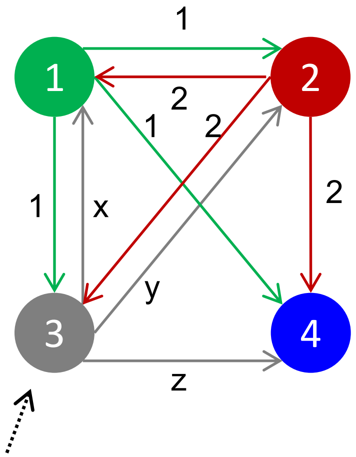


Faulty process

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others

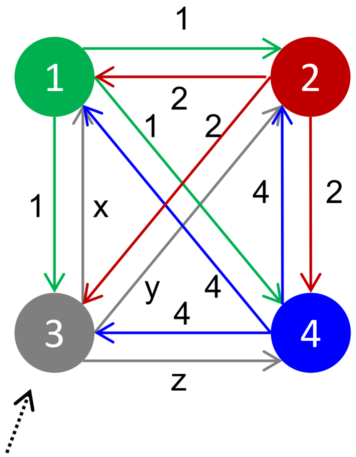


Faulty process

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***<u>Step1:</u>*** Each process sends its value to the others

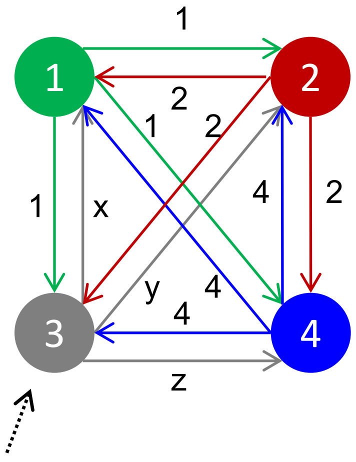


Faulty process

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others

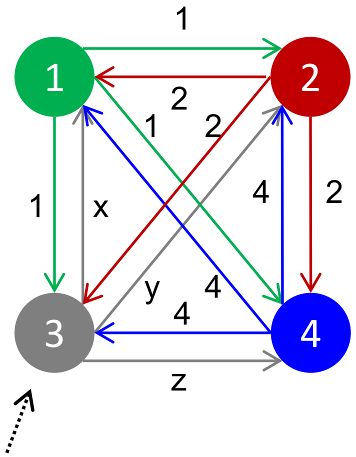


Faulty process

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others



Faulty process

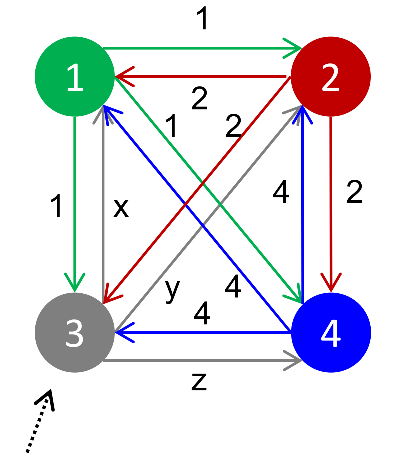
***Step2:*** Each process collects values received in a vector

**1** Got(1, 2, x, 4)
**2** Got(1, 2, y, 4)
**3** Got(1, 2, 3, 4)
**4** Got(1, 2, z, 4)

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others



Faulty process

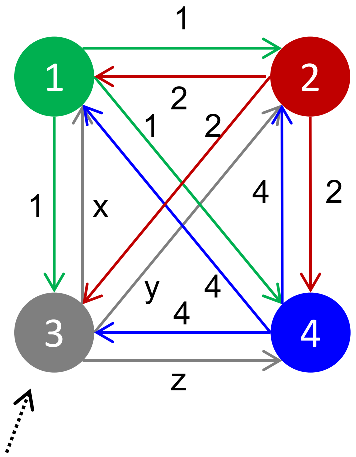
***Step2:*** Each process collects values received in a vector

**1** Got(1, 2, x, 4)
**2** Got(1, 2, y, 4)
**3** Got(1, 2, 3, 4)
**4** Got(1, 2, z, 4)

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others



Faulty process

***Step2:*** Each process collects values received in a vector

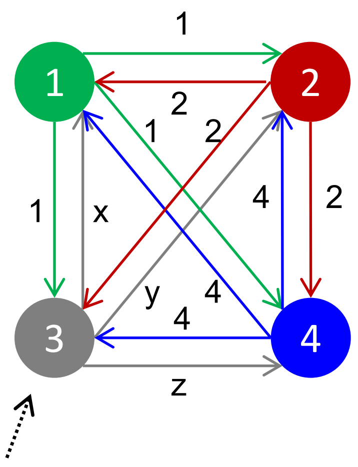
**1** Got(1, 2, x, 4)
**2** Got(1, 2, y, 4)
**3** Got(1, 2, 3, 4)
**4** Got(1, 2, z, 4)

***Step3:*** Every process passes its vector to every other process

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others



Faulty process

***Step2:*** Each process collects values received in a vector

**1** Got(1, 2, x, 4)
**2** Got(1, 2, y, 4)
**3** Got(1, 2, 3, 4)
**4** Got(1, 2, z, 4)

***Step3:*** Every process passes its vector to every other process

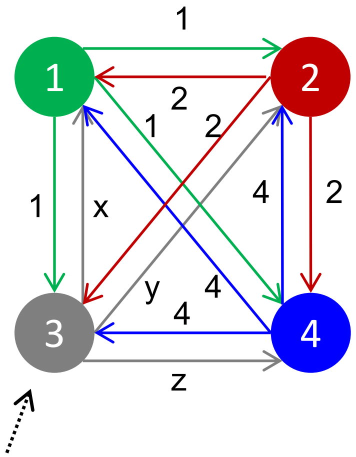
**1** Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others



Faulty
process

***Step2:*** Each process collects values received in a vector

**1** Got(1, 2, x, 4)
**2** Got(1, 2, y, 4)
**3** Got(1, 2, 3, 4)
**4** Got(1, 2, z, 4)

***Step3:*** Every process passes its vector to every other process

**1** Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

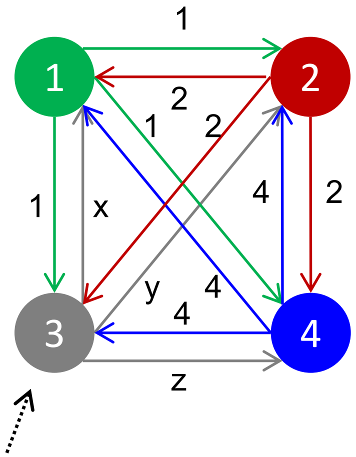
**2** Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

# Reaching Agreement in the Presence of Fault

- Case I: N = 4 and f = 1

***Step1:*** Each process sends its value to the others



Faulty process

***Step2:*** Each process collects values received in a vector

**1** Got(1, 2, x, 4)
**2** Got(1, 2, y, 4)
**3** Got(1, 2, 3, 4)
**4** Got(1, 2, z, 4)

***Step3:*** Every process passes its vector to every other process

**1** Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

**2** Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

**4** Got

(1, 2, x, 4)
(1, 2, y, 4)
(i, j, k, l)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

1 Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

1 Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

2 Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

**2** Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

**2** Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

**4** Got

(1, 2, x, 4)
(1, 2, y, 4)
( i, j, k, l)

**Result Vector:**
(1, 2, UNKNOWN, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

1 Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

2 Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

4 Got

(1, 2, x, 4)
(1, 2, y, 4)
( i,  j, k,  l)

**Result Vector:**
(1, 2, UNKNOWN, 4)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*-th element of each of the newly received vectors
- If any value has a majority, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

**2** Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

**Result Vector:**
(1, 2, UNKNOWN, 4)

**4** Got

(1, 2, x, 4)
(1, 2, y, 4)
( i, j, k, l)

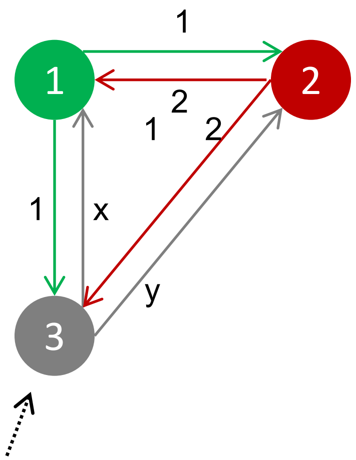
**Result Vector:**
(1, 2, UNKNOWN, 4)

Is 3f+1 optimal?

# Reaching Agreement in the Presence of Fault

- Case II: N = 3 and f = 1

***Step1:*** Each process sends its value to the others



Faulty process

***Step2:*** Each process collects values received in a vector

**1** Got(1, 2, x)
**2** Got(1, 2, y)
**3** Got(1, 2, 3)

***Step3:*** Every process passes its vector to every other process

**1** Got

(1, 2, y)
(a, b, c)

**2** Got

(1, 2, x)
(d, e, f)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y)
(a, b, c)

# Reaching Agreement in the Presence of Fault

**_Step 4:_**

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y)
(a, b, c)

**Result Vector:**
(UNKOWN, UNKNOWN, UNKNOWN)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y)
(a, b, c)

**2** Got

(1, 2, x)
(d, e, f)

**Result Vector:**
(UNKOWN, UNKNOWN, UNKNOWN)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y)
(a, b, c)

**2** Got

(1, 2, x)
(d, e, f)

**Result Vector:**
(UNKOWN, UNKNOWN, UNKNOWN)

**Result Vector:**
(UNKOWN, UNKNOWN, UNKNOWN)

# Reaching Agreement in the Presence of Fault

***Step 4:***

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

**1** Got

(1, 2, y)
(a, b, c)

The algorithm has failed to produce an agreement

**2** Got

(1, 2, x)
(d, e, f)

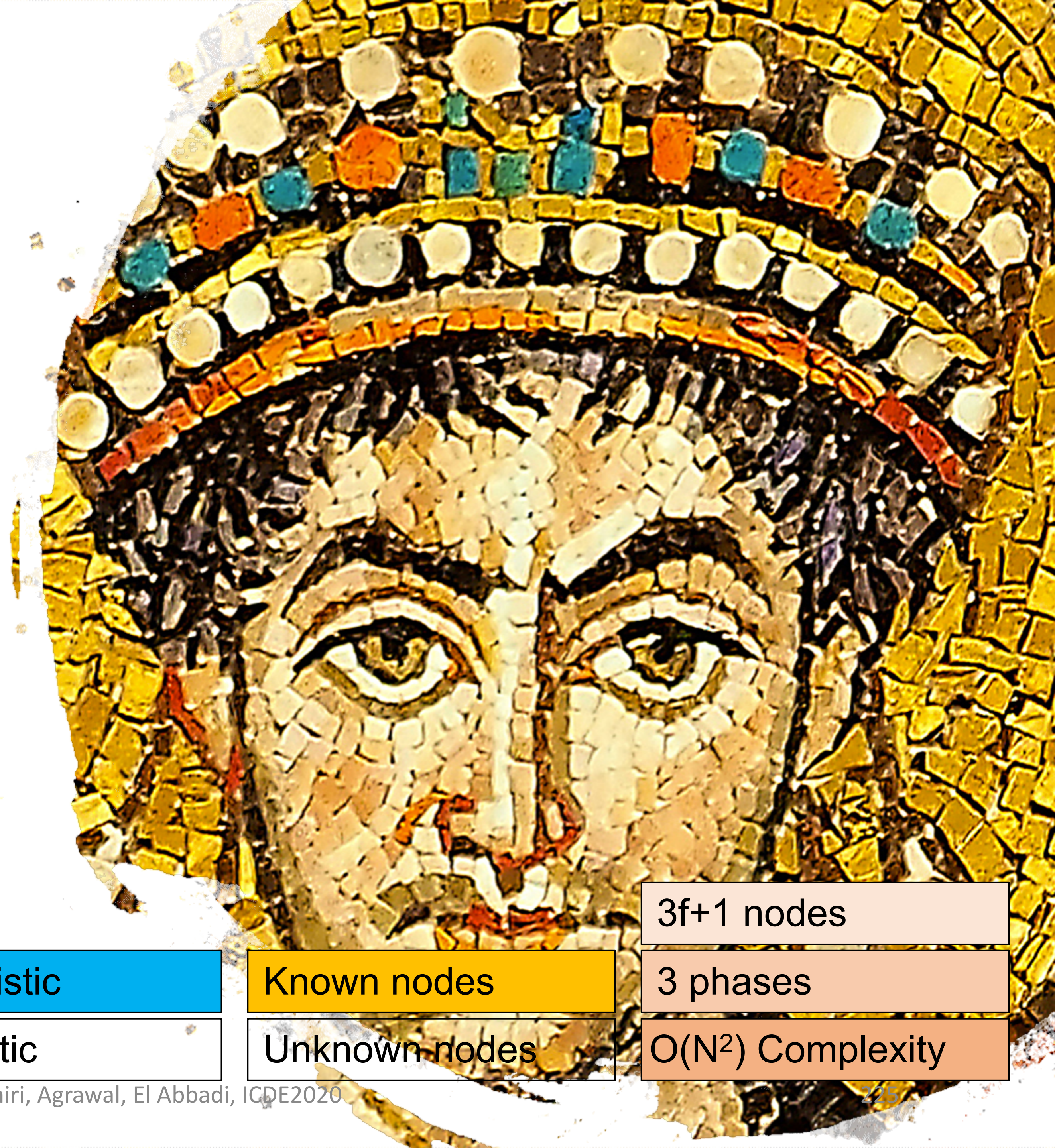
**Result Vector:**
(UNKOWN, UNKNOWN, UNKNOWN)

**Result Vector:**
(UNKOWN, UNKNOWN, UNKNOWN)

# Practical Byzantine Fault Tolerance

Castro, Miguel, and Barbara Liskov.
"Practical Byzantine fault tolerance."

- OSDI, vol. 99, 1999
- ACM Transactions on Computer Systems, 2002

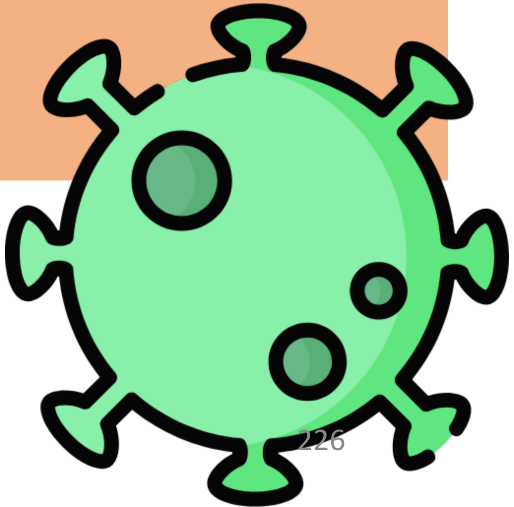


| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | 3f+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 3 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | $O(N^2)$ Complexity |

# Why doesn’t Paxos work with Byzantine nodes?

- Cannot rely on the primary to assign sequence number
  - A Malicious primary can assign the same sequence to different requests!

- Cannot use Paxos for leader election
  - Paxos uses a majority (f+1) accept-quorum to tolerate f benign faults out of 2f+1 nodes
  - Does the intersection of two quorums always contain one honest node?
  - Bad node tells different things to different quorums!
    - E.g. tell N1 accept=val1 and tell N2 accept=val2



# PBFT Main Ideas

- Configuration
    - Use 3f+1 nodes
- To deal with malicious primary
    - Use a 3-phase protocol
- To deal with loss of agreement
    - Use a bigger quorum (2f+1 out of 3f+1 nodes)
- Need to authenticate communications

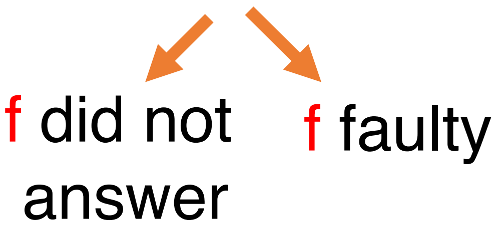
# Failure Assumption

# Failure Assumption

- N > 3f    Why?
- To make any progress must be able to tolerate f failures, i.e., must be able to make progress if only n-f processes respond.
- BUT maybe the f that did not respond are not faulty, but slow (asynchronous systems), and among n-f that responded f are faulty!
- Must have enough responses from non-faulty to outnumber faulty
    - $n-2f > f$ → $n>3f$

# Failure Assumption

- N > 3f Why?
- To make any progress must be able to tolerate f failures, i.e., must be able to make progress if only n-f processes respond.
- BUT maybe the f that did not respond are not faulty, but slow (asynchronous systems), and among n-f that responded f are faulty!
- Must have enough responses from non-faulty to outnumber faulty
  - $n-2f > f \rightarrow n>3f$



f did not answer

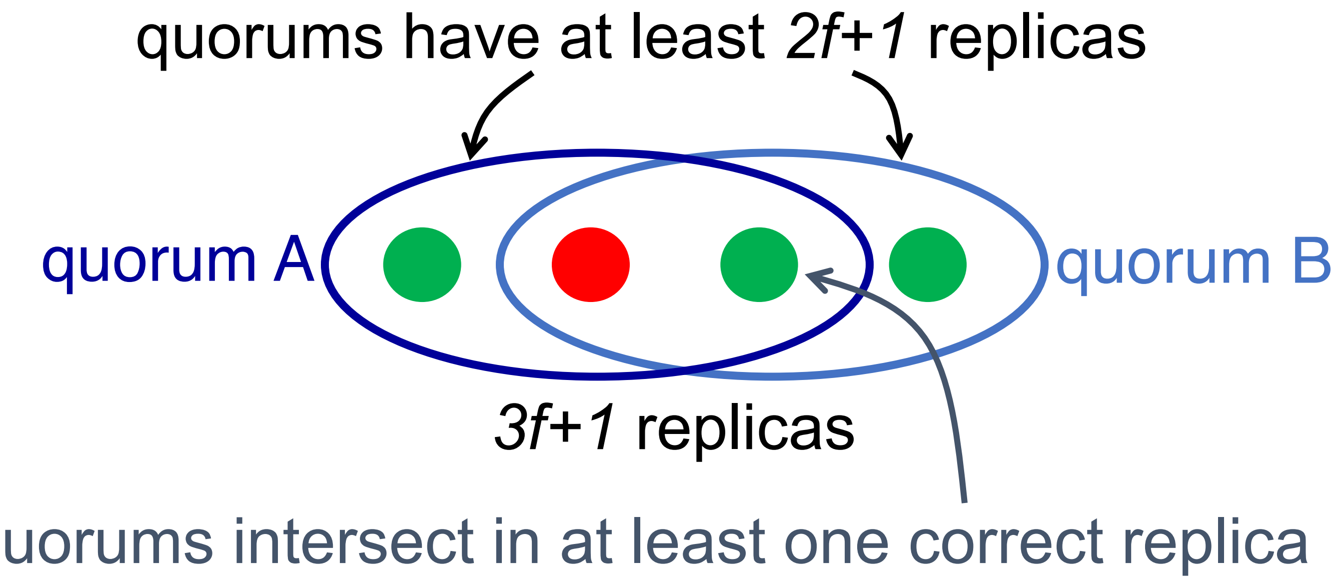
f faulty

# Quorum and Network Size

- $N > 3f$ Why? (Another Argument!)
- Any two Quorums of responses Q need to intersect in at least $f+1$ nodes
  - $Q_1 + Q_2 > N + f$
  - $(N-f) + (N-f) > N + f \Rightarrow N > 3f$

# Quorum and Network Size

- $N > 3f$ Why? (Another Argument!)
- Any two Quorums of responses Q need to intersect in at least $f+1$ nodes
  - $Q_1 + Q_2 > N + f$
  - $(N-f) + (N-f) > N + f \Rightarrow N > 3f$



quorums have at least *2f+1* replicas

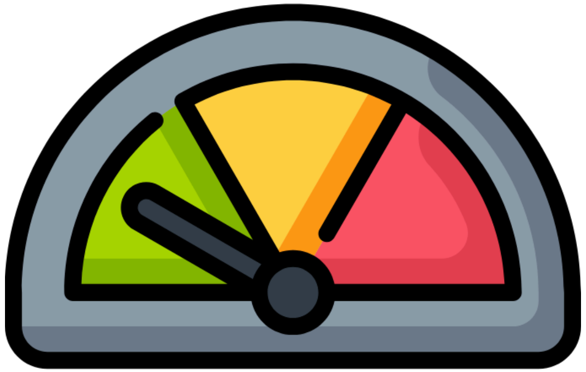
quorum A

quorum B

*3f+1* replicas

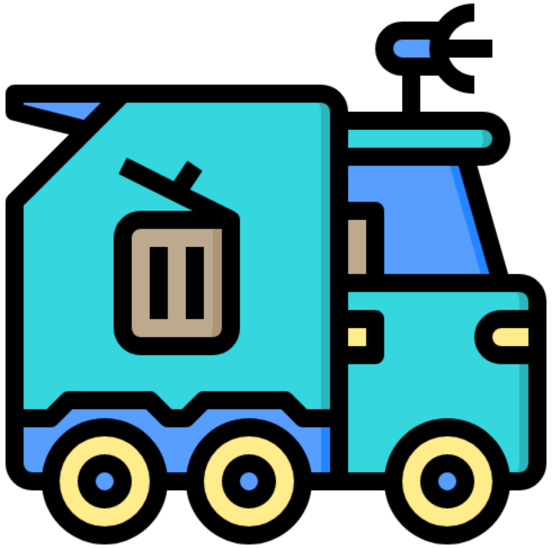
quorums intersect in at least one correct replica

# Algorithm Components



Normal case operation

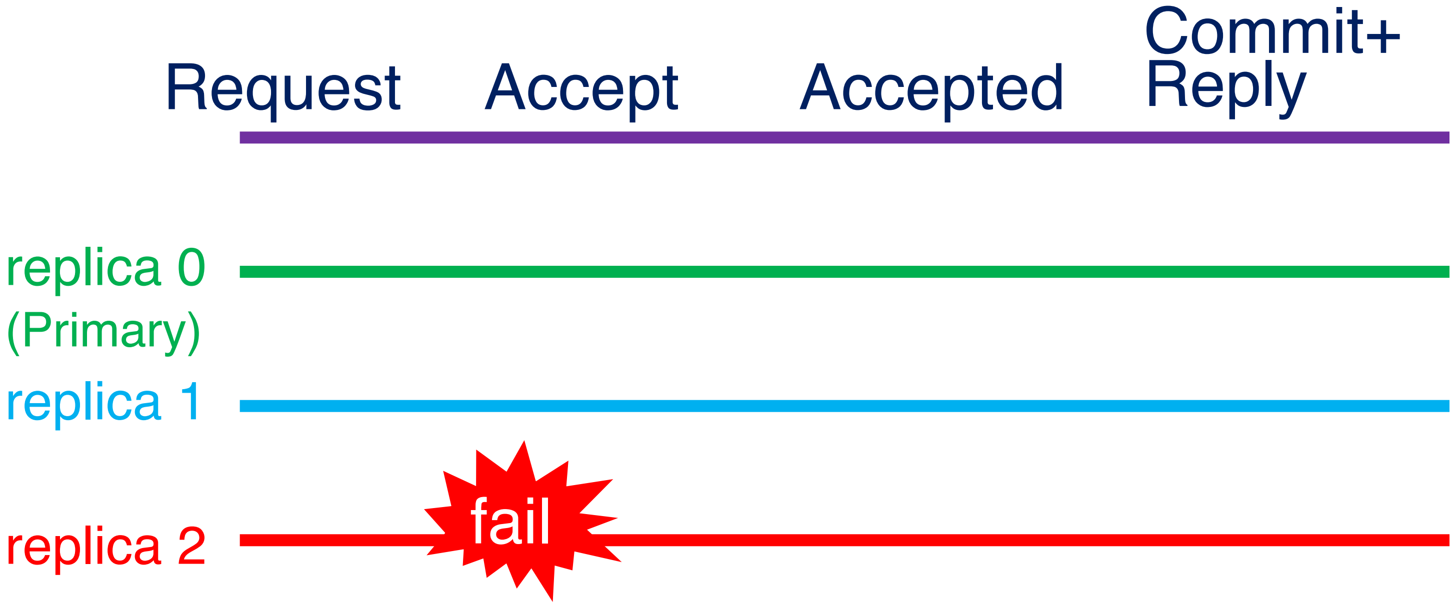
View changes



Garbage collection

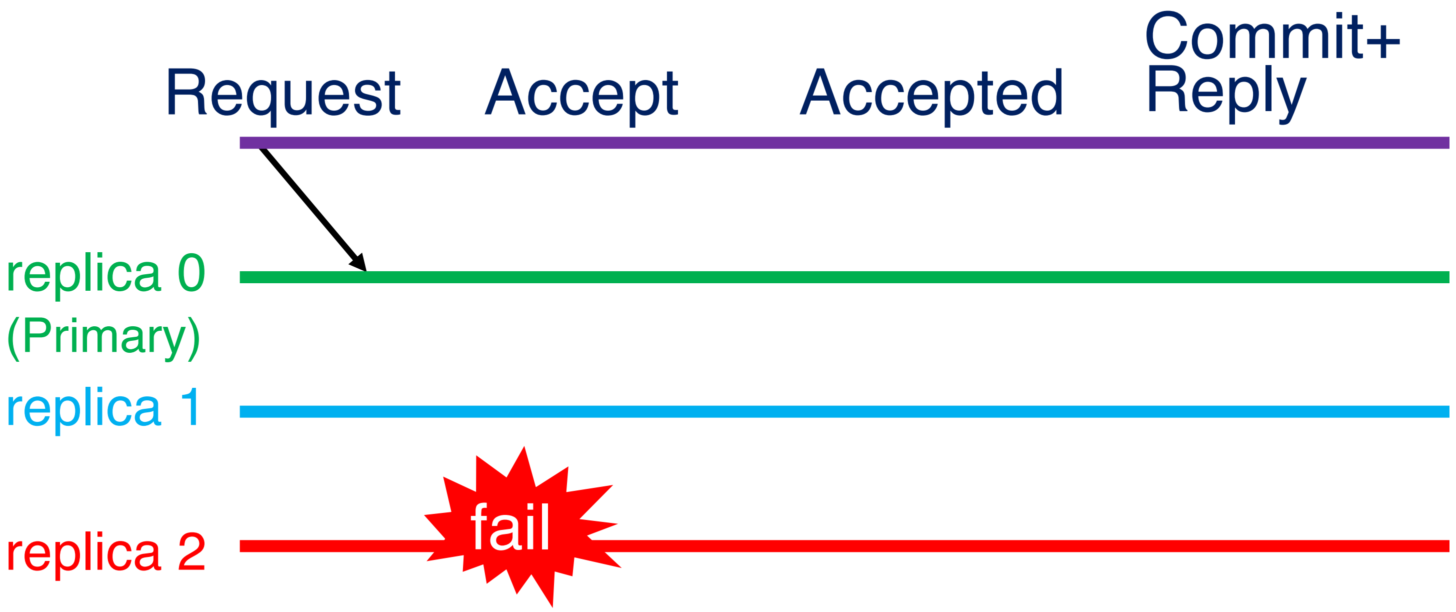
# (Multi-)Paxos Review

- Let's assume the leader is already elected



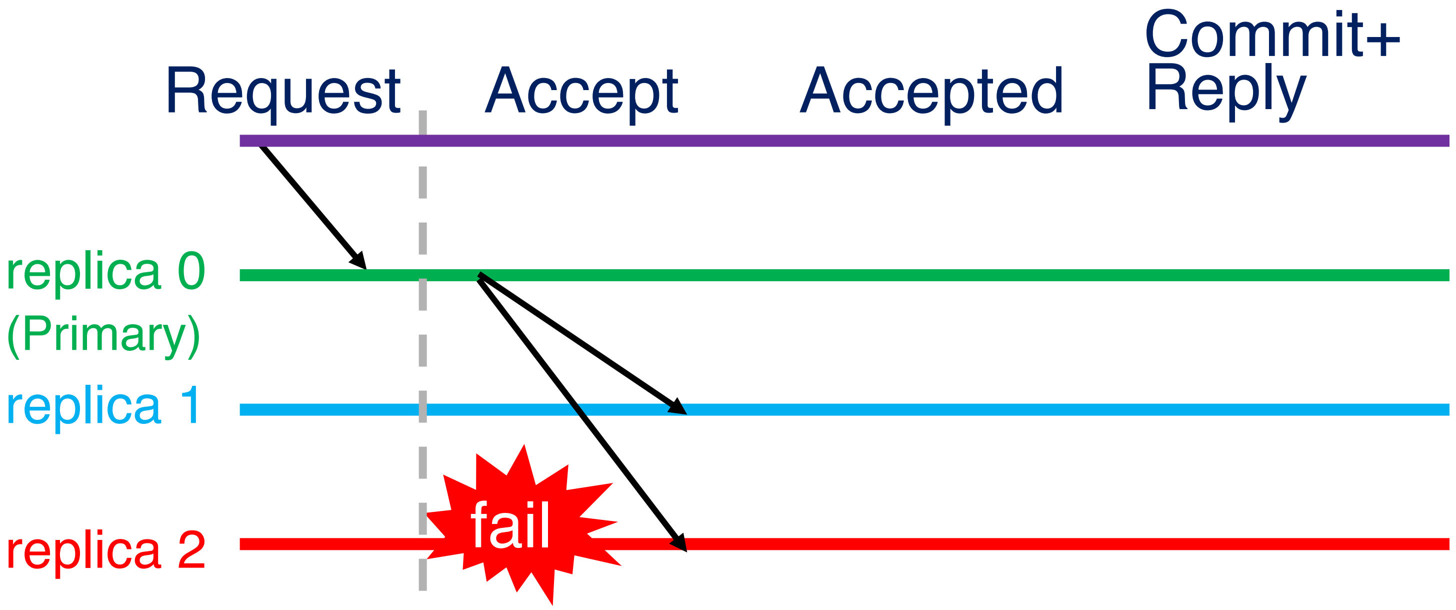
# (Multi-)Paxos Review

- Let’s assume the leader is already elected



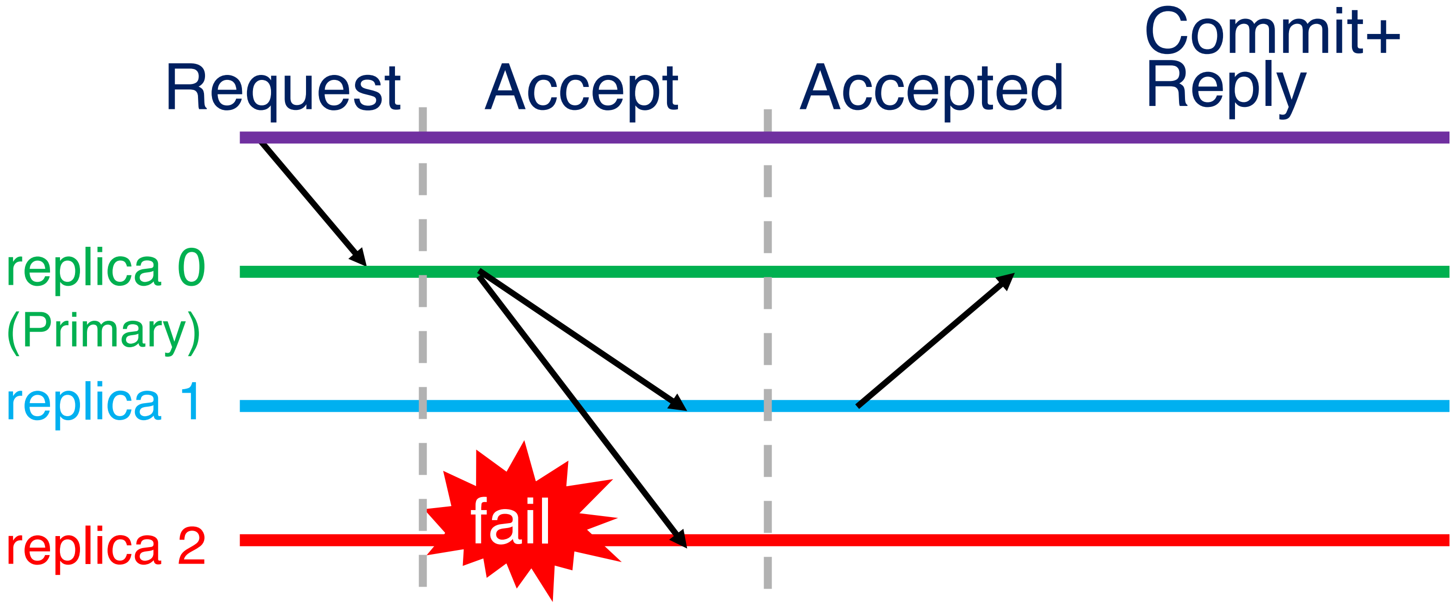
# (Multi-)Paxos Review

- Let’s assume the leader is already elected



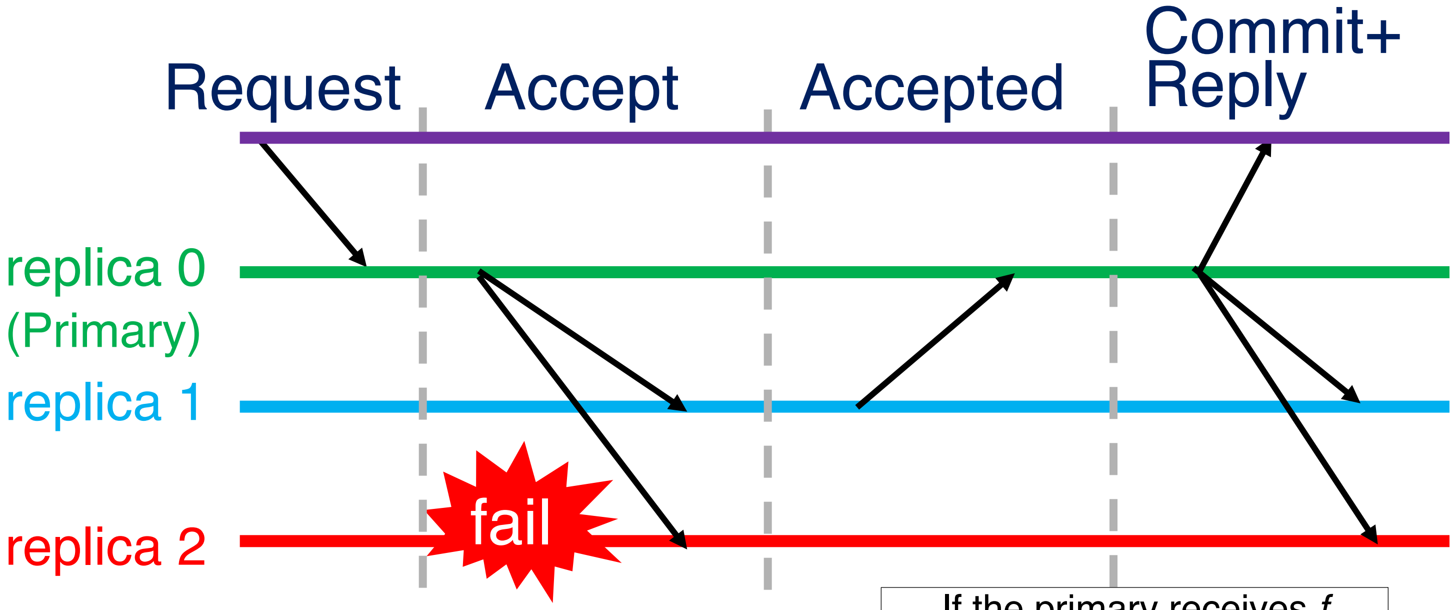
# (Multi-)Paxos Review

- Let's assume the leader is already elected



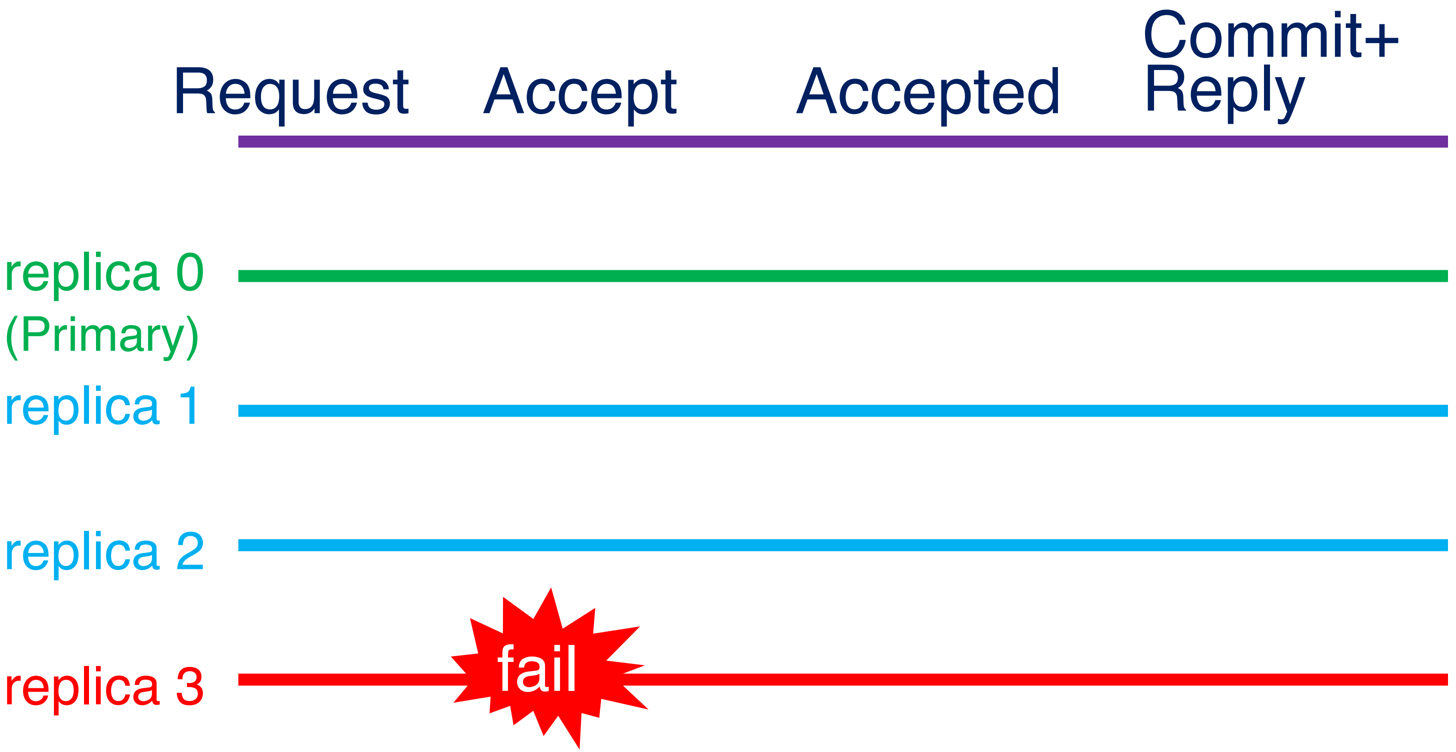
# (Multi-)Paxos Review

- Let’s assume the leader is already elected



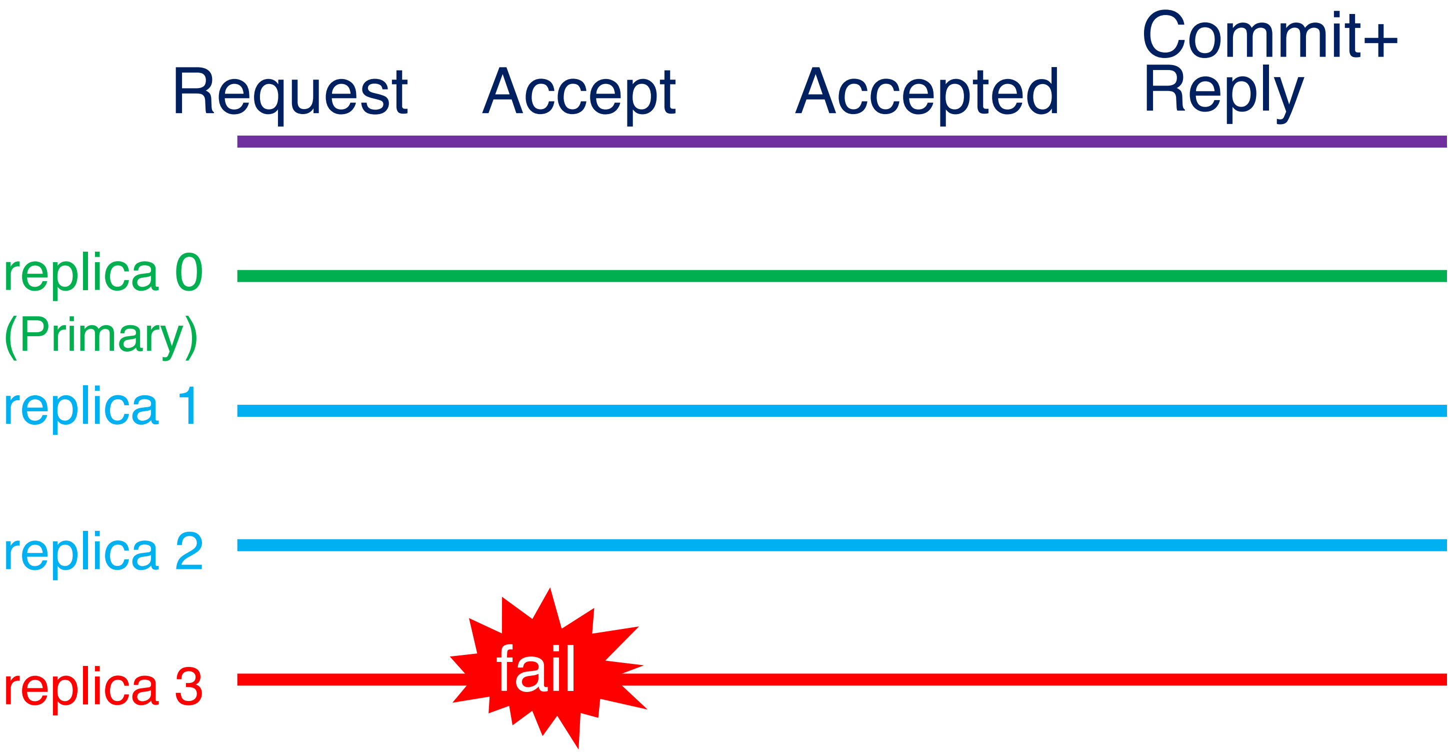
If the primary receives *f*
Matching *accepted* messages
(including itself *f+1*),
it sends *commit* and *reply*

# (Multi-)Paxos with Malicious Backups



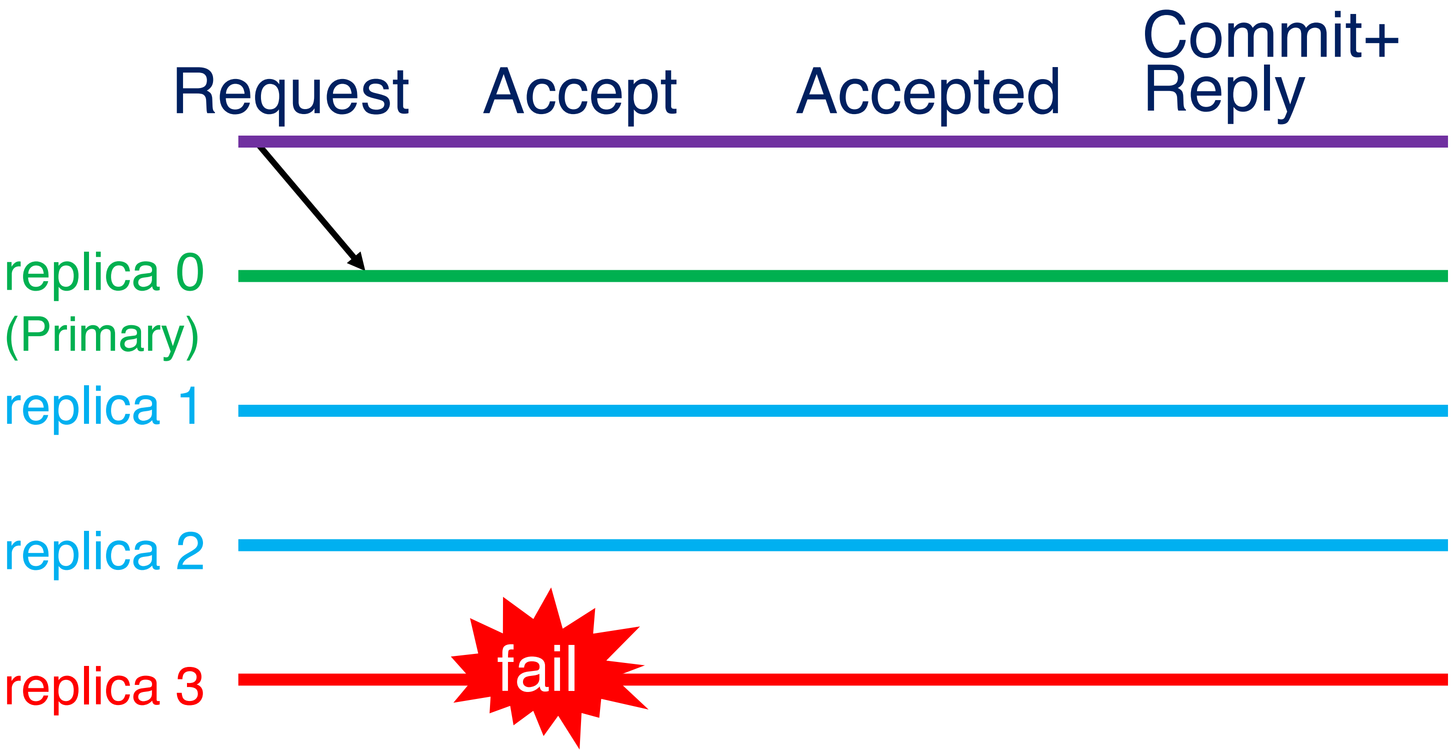
# (Multi-)Paxos with Malicious Backups

- What if f of the backups (not the primary) are malicious?!
  - 3f+1 nodes needed!



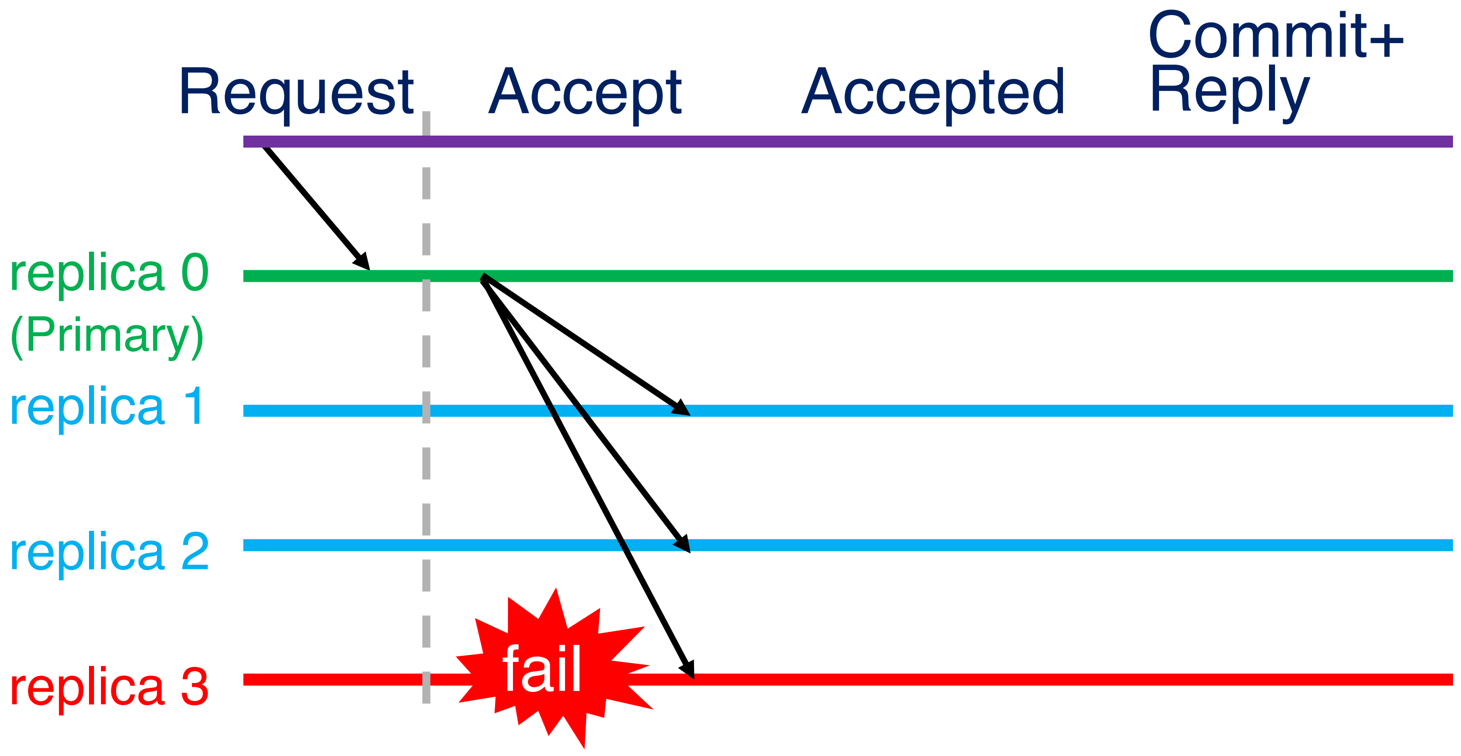
# (Multi-)Paxos with Malicious Backups

- What if f of the backups (not the primary) are malicious?!
  - 3f+1 nodes needed!



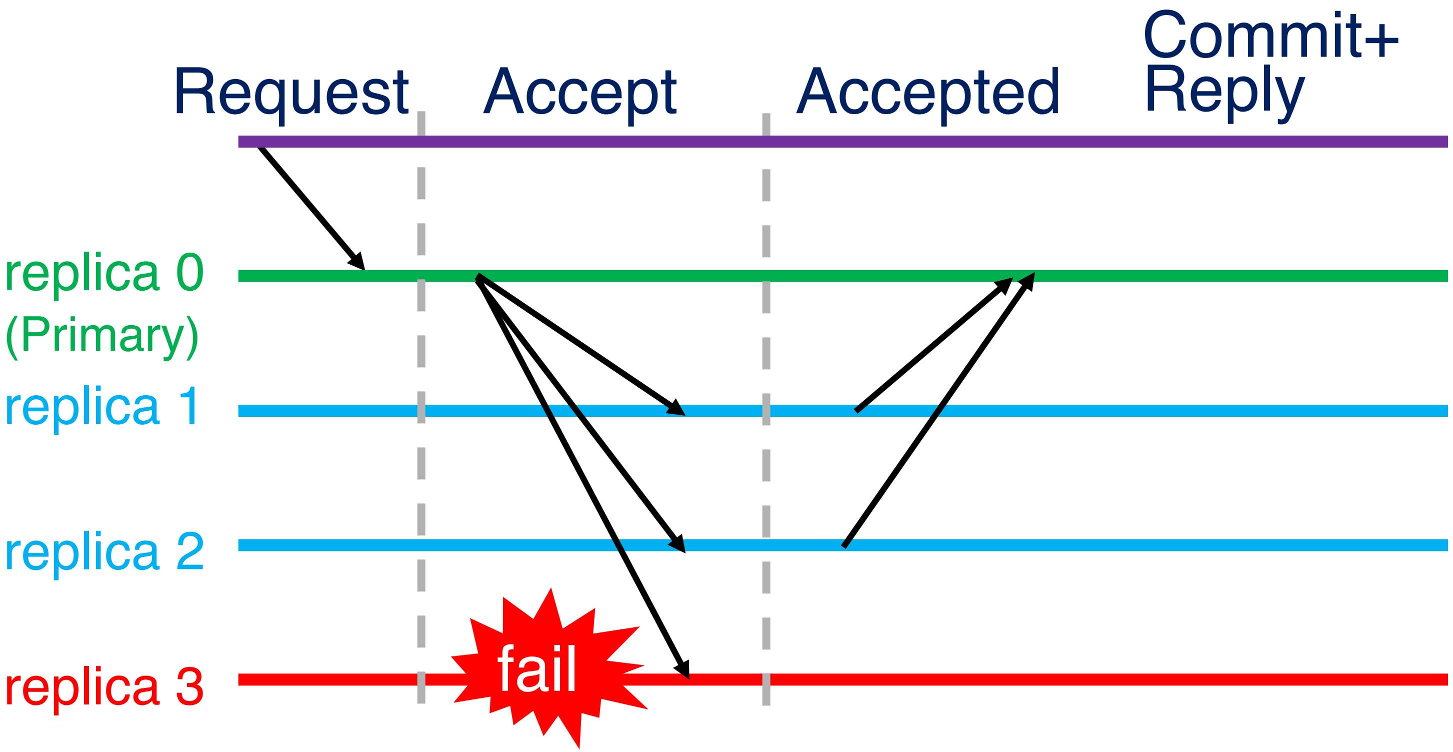
# (Multi-)Paxos with Malicious Backups

- What if f of the backups (not the primary) are malicious?!
  - 3f+1 nodes needed!



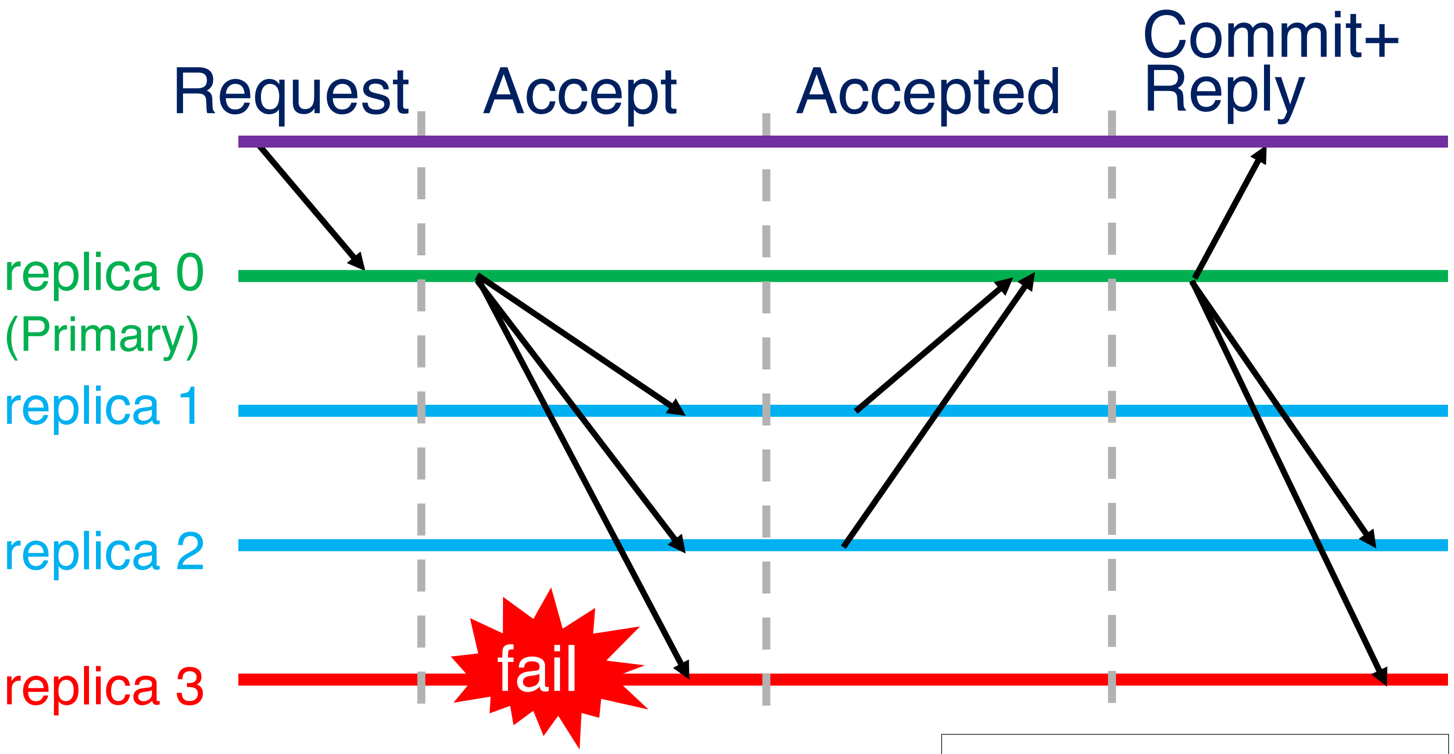
# (Multi-)Paxos with Malicious Backups

- What if f of the backups (not the primary) are malicious?!
  - 3f+1 nodes needed!



# (Multi-)Paxos with Malicious Backups

- What if f of the backups (not the primary) are malicious?!
  - 3f+1 nodes needed!



the primary waits for *2f* Matching *accepted* messages (including itself *2f+1*)

# (Multi-)Paxos Optimization

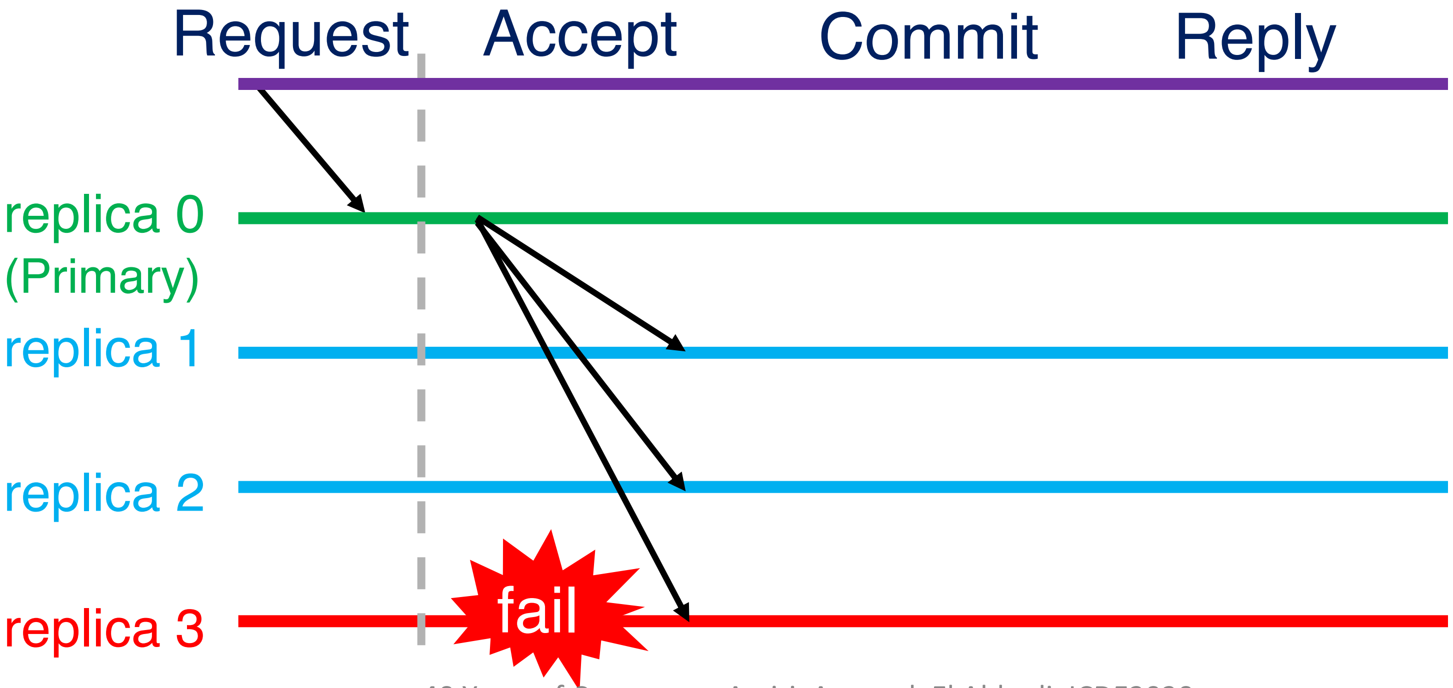
- Can nodes commit earlier?!

# (Multi-)Paxos Optimization

- Can nodes commit earlier?!

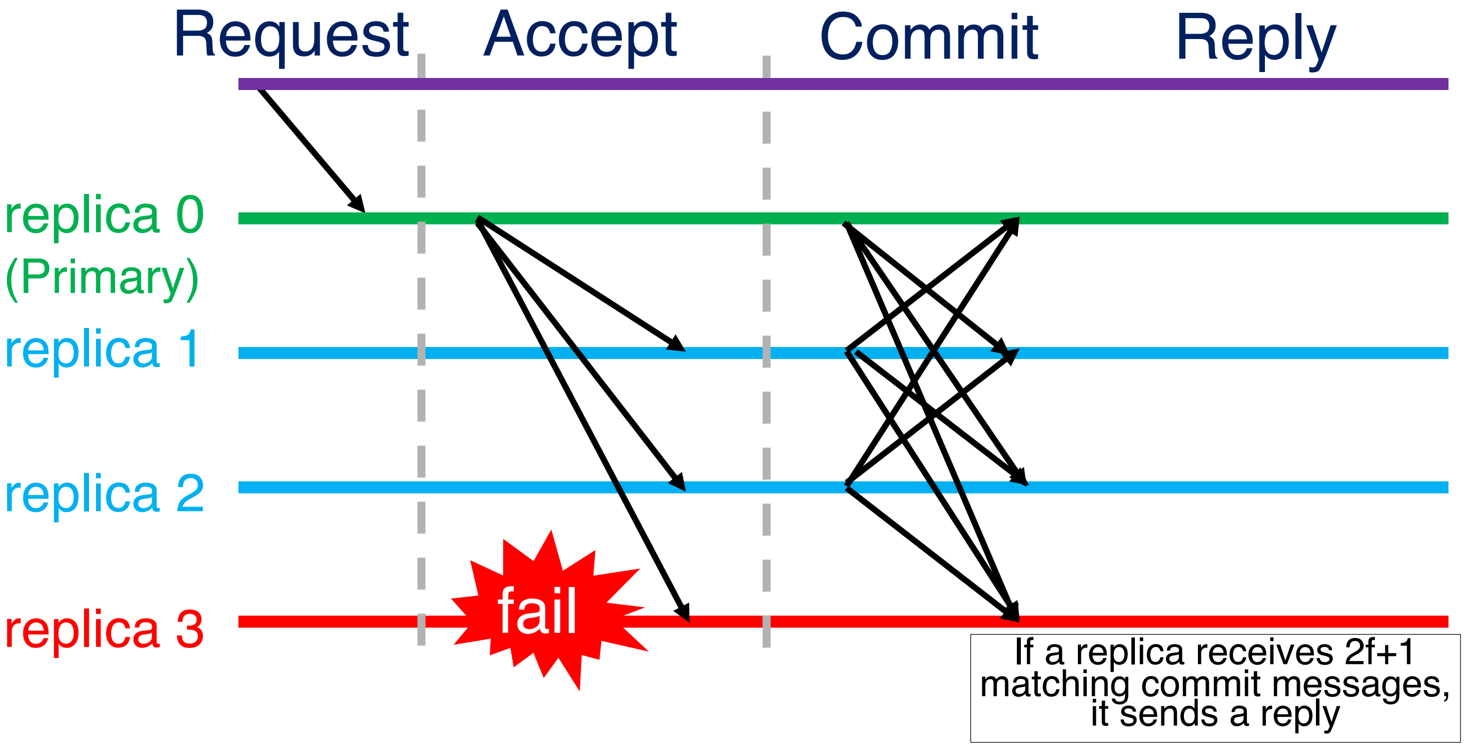
# (Multi-)Paxos Optimization

- Can nodes commit earlier?!



# (Multi-)Paxos Optimization

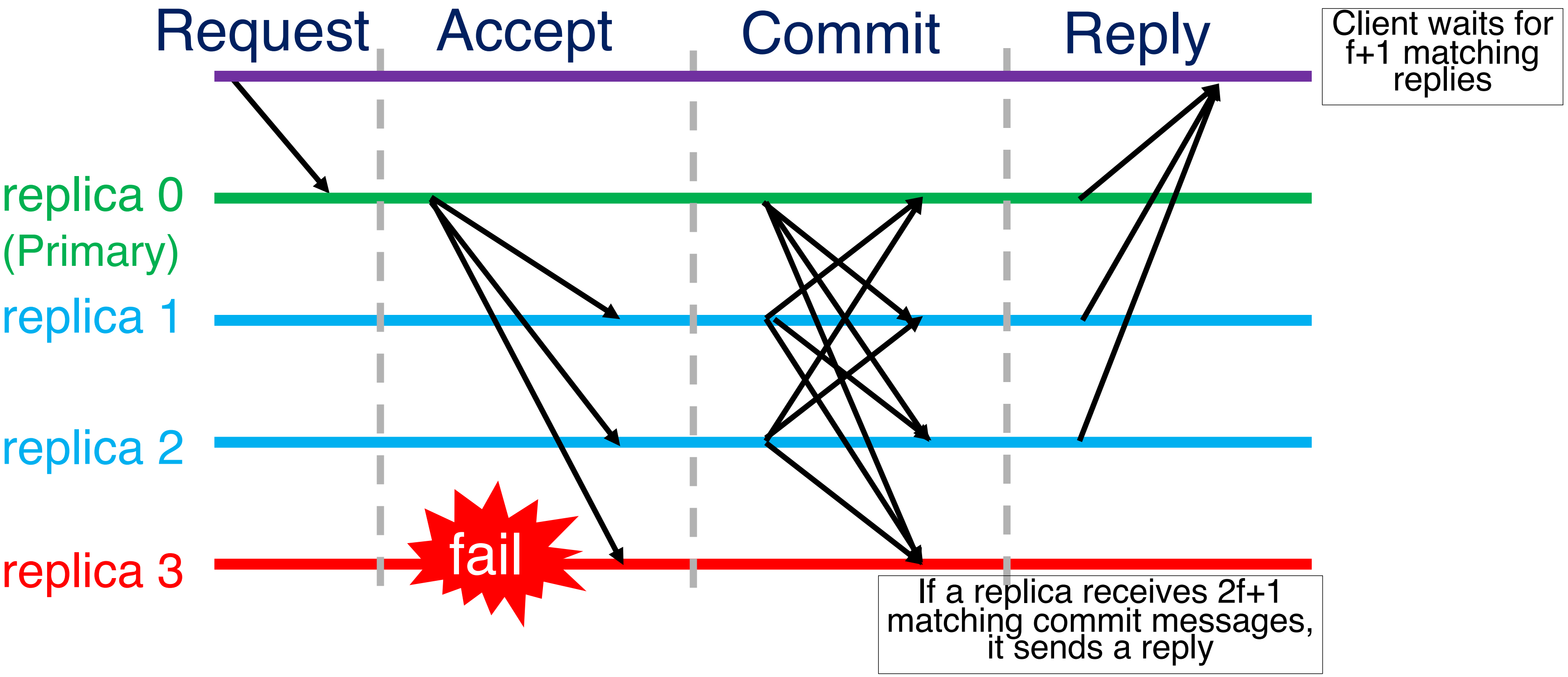
- Can nodes commit earlier?!



If a replica receives 2f+1 matching commit messages, it sends a reply

# (Multi-)Paxos Optimization

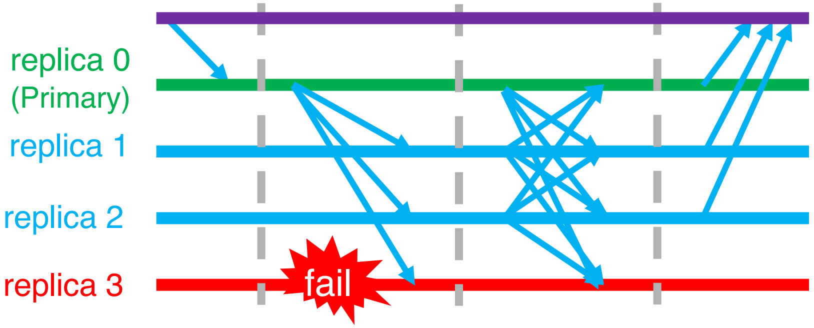
- Can nodes commit earlier?!



Client waits for f+1 matching replies

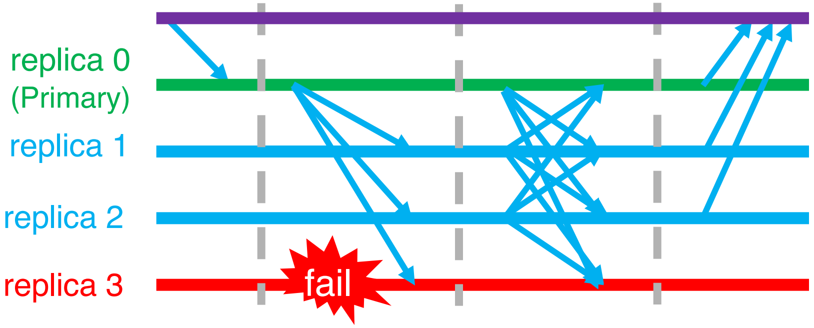
If a replica receives 2f+1 matching commit messages, it sends a reply

# From (Multi-)Paxos to PBFT



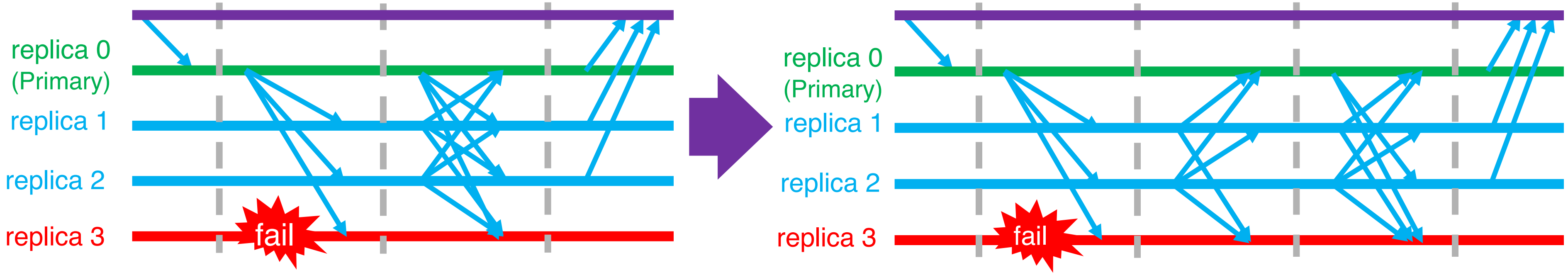
# From (Multi-)Paxos to PBFT

- When a replica receives an *Accept* message, it knows every replica receives the same message
- What if the leader is malicious?!!
  - Assigns different sequence numbers to the same request
  - Assigns the same sequence number to different request
- One more phase of communication is needed!

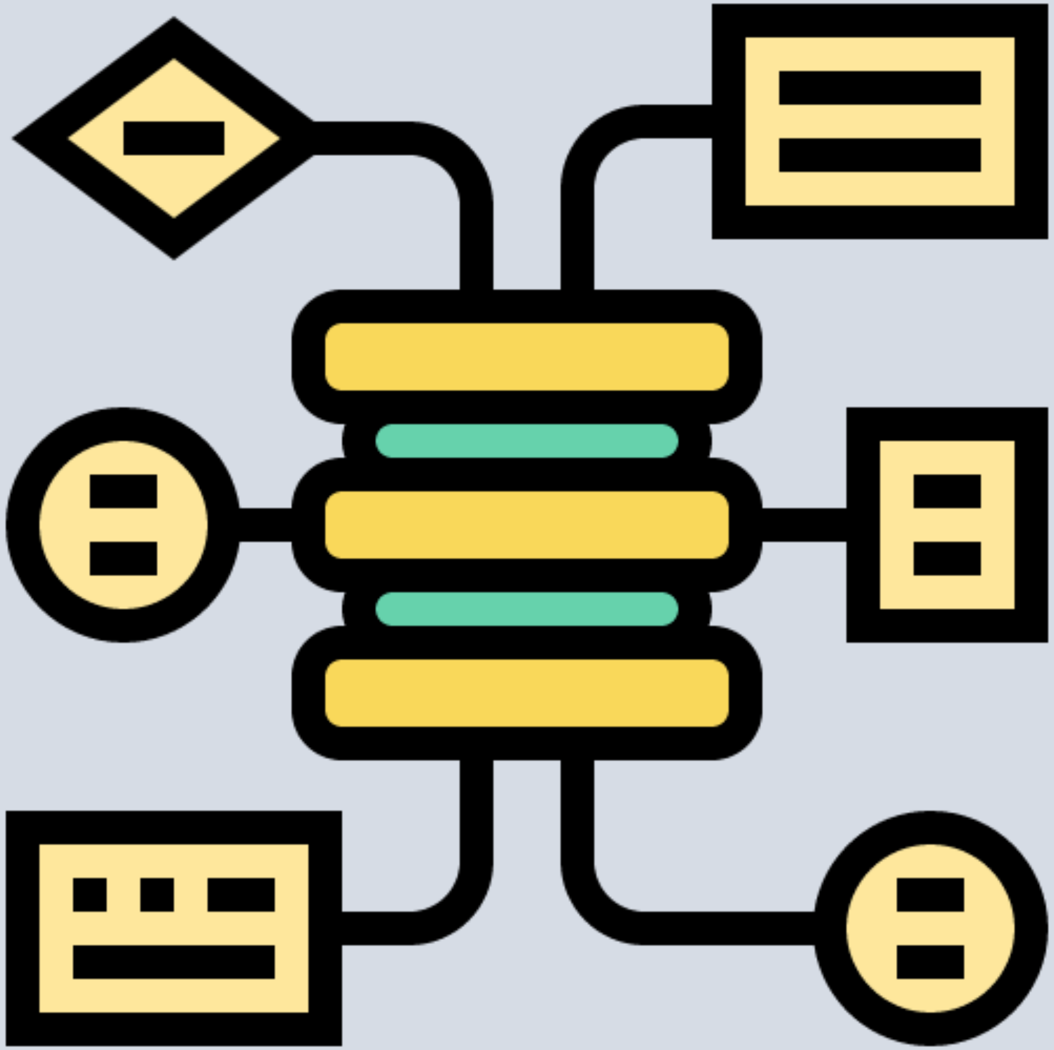


# From (Multi-)Paxos to PBFT

- When a replica receives an *Accept* message, it knows every replica receives the same message
- What if the leader is malicious?!!
  - Assigns different sequence numbers to the same request
  - Assigns the same sequence number to different request
- One more phase of communication is needed!

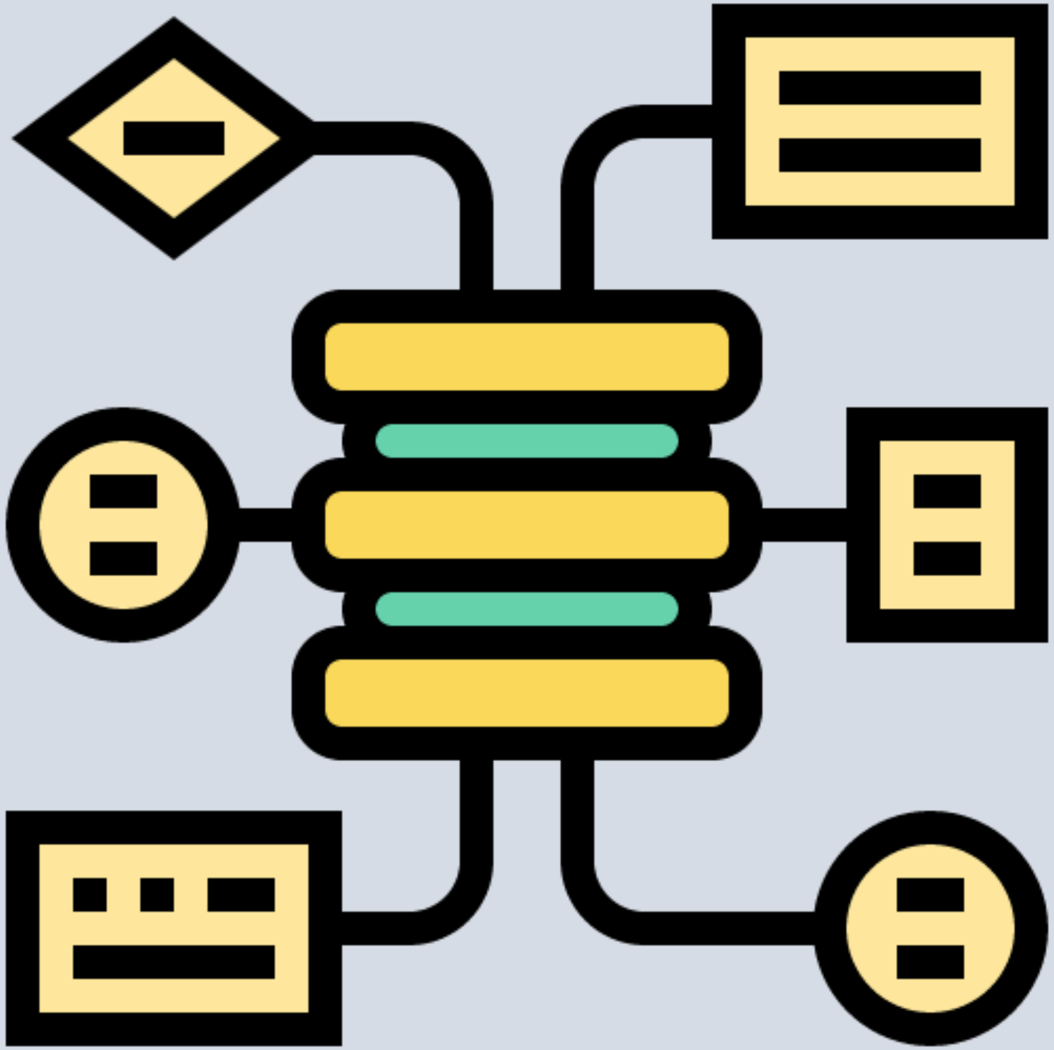


# Normal Case Operation



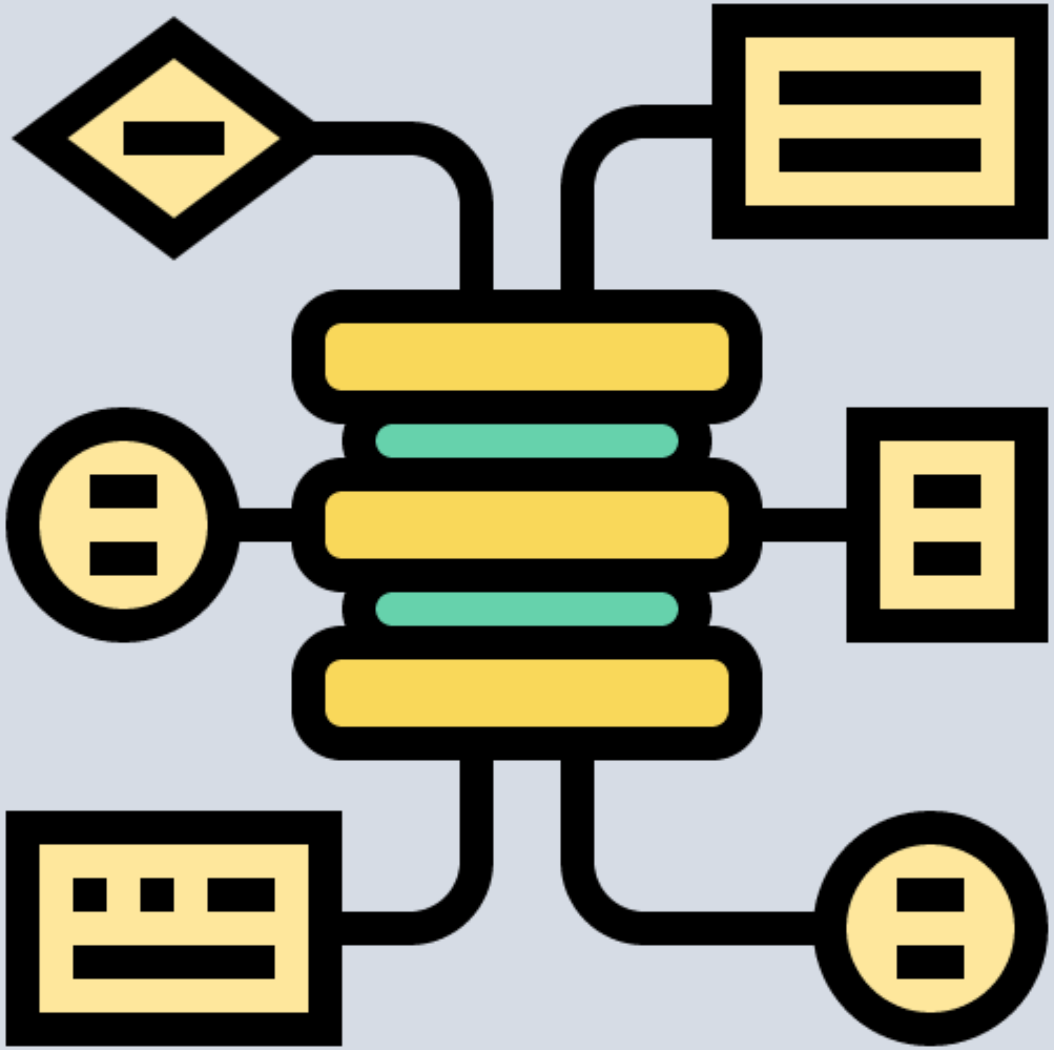
# Normal Case Operation

- The algorithm has three phases:
  - *pre-prepare* picks order of requests
  - *prepare* ensures order within views
  - *commit* ensures order across views



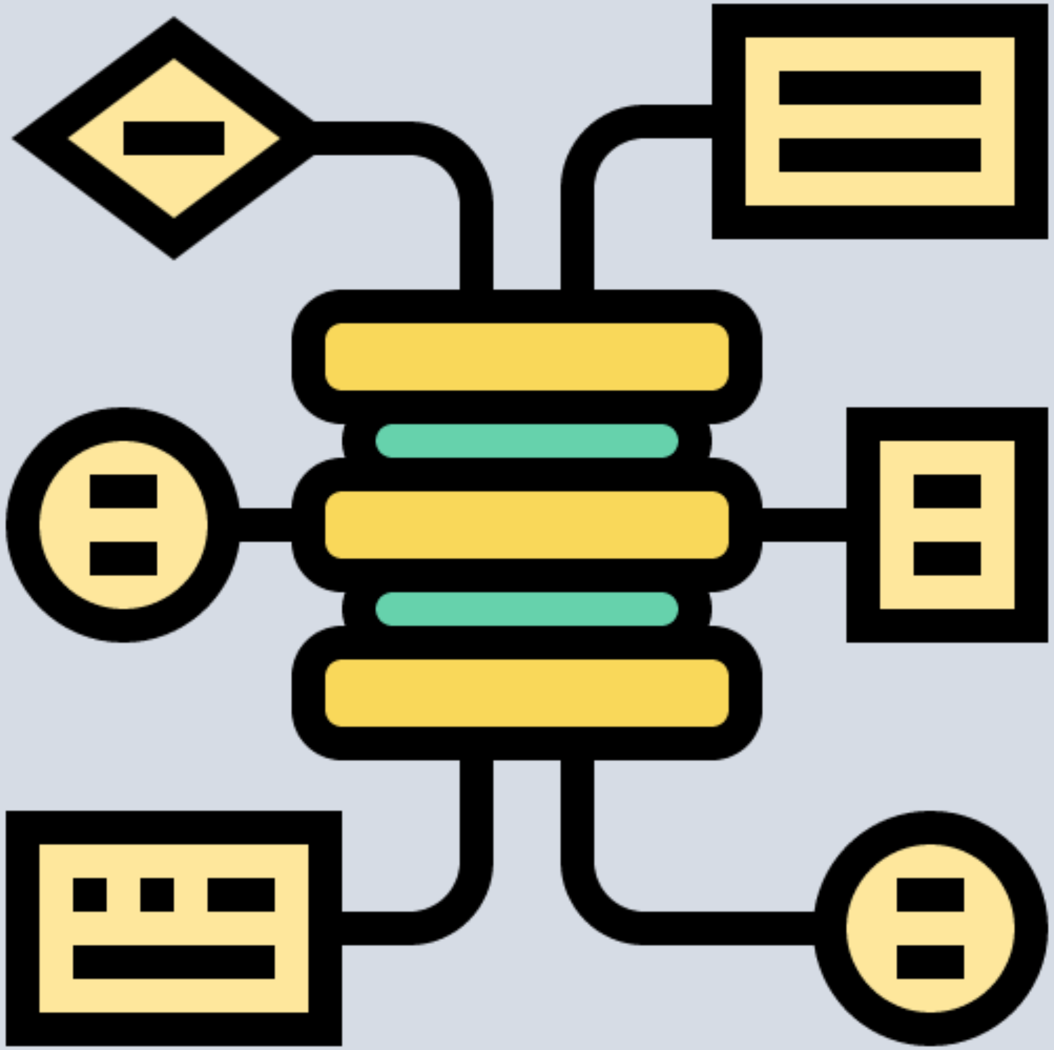
# Normal Case Operation

- The algorithm has three phases:
  - *pre-prepare* picks order of requests
  - *prepare* ensures order within views
  - *commit* ensures order across views
- A replica executes a request m if
  - m is committed
  - all requests with sequence number less than n have been executed

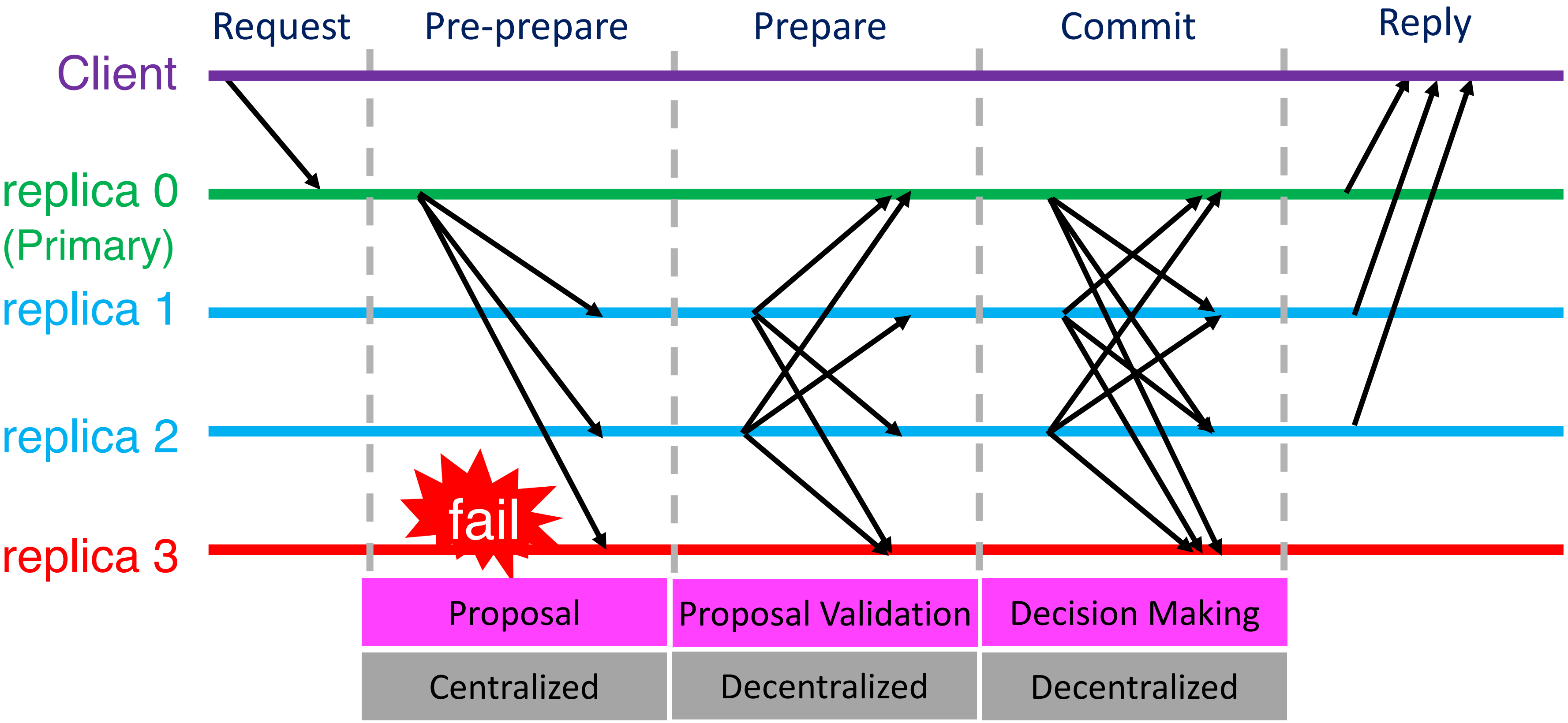


# Normal Case Operation

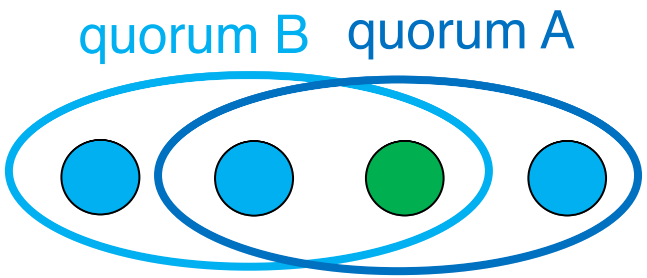
- The algorithm has three phases:
  - *pre-prepare* picks order of requests
  - *prepare* ensures order within views
  - *commit* ensures order across views
- A replica executes a request m if
  - m is committed
  - all requests with sequence number less than n have been executed
- Replicas send a reply to the client
  - Client waits for f+1 matching replies



# PBFT Agreement Protocol Summary



**At Most f Malicious Failures**



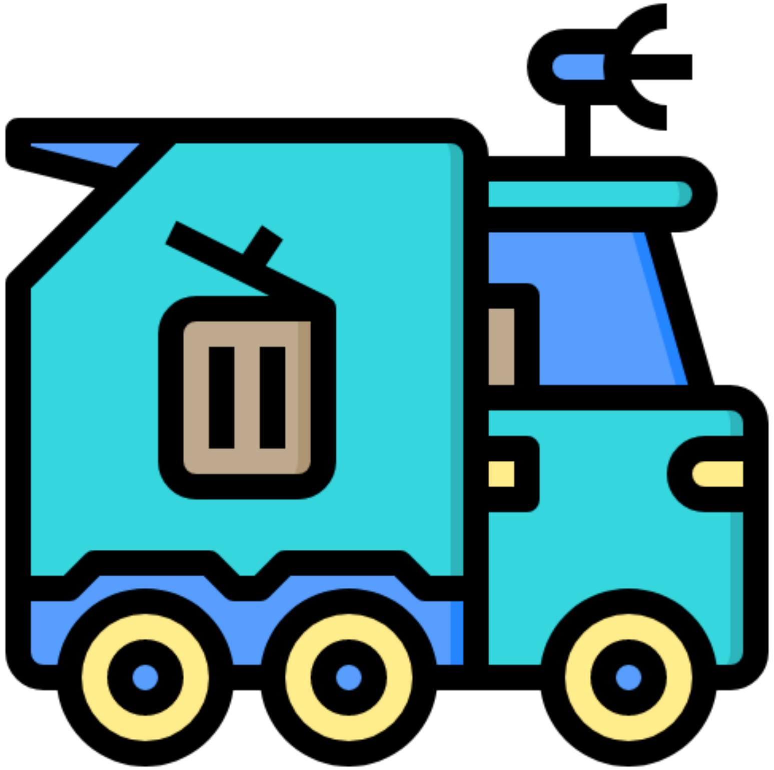
Network: 3f+1
Quorum: 2f+1
Intersection: f+1

# View Change

# View Change

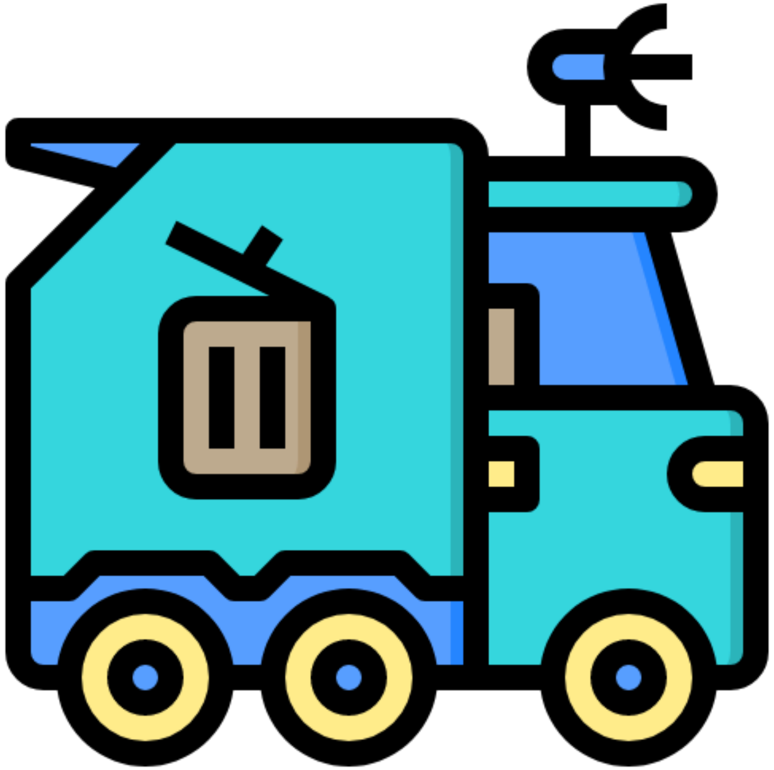
- Provide liveness when primary fails
  - Timeouts trigger view changes
- Request a view change
  - send a viewchange request to all
  - new primary requires 2f+1 viewchange messages to accept new role
  - sends new-view with proof (2f+1 viewchange messages)
- View change has a high complexity: $O(n^3)$

# Garbage Collection



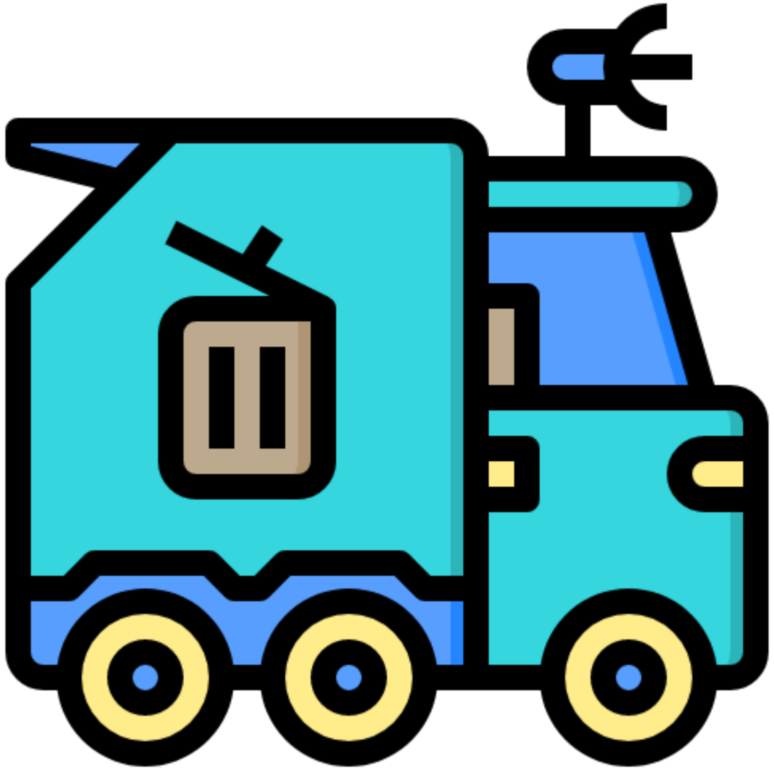
# Garbage Collection

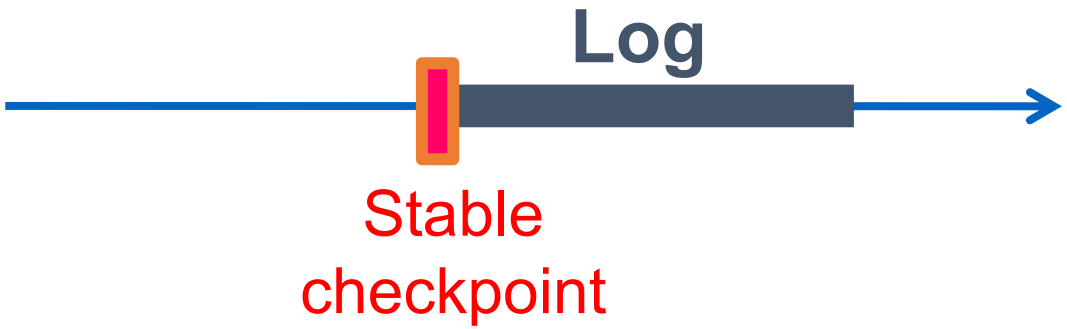
- When to discard messages in the log?
  - periodically checkpoint the state by multicasting CHECKPOINT messages
  - Each node collects *2f+1* checkpoint messages: proof of correctness



# Garbage Collection

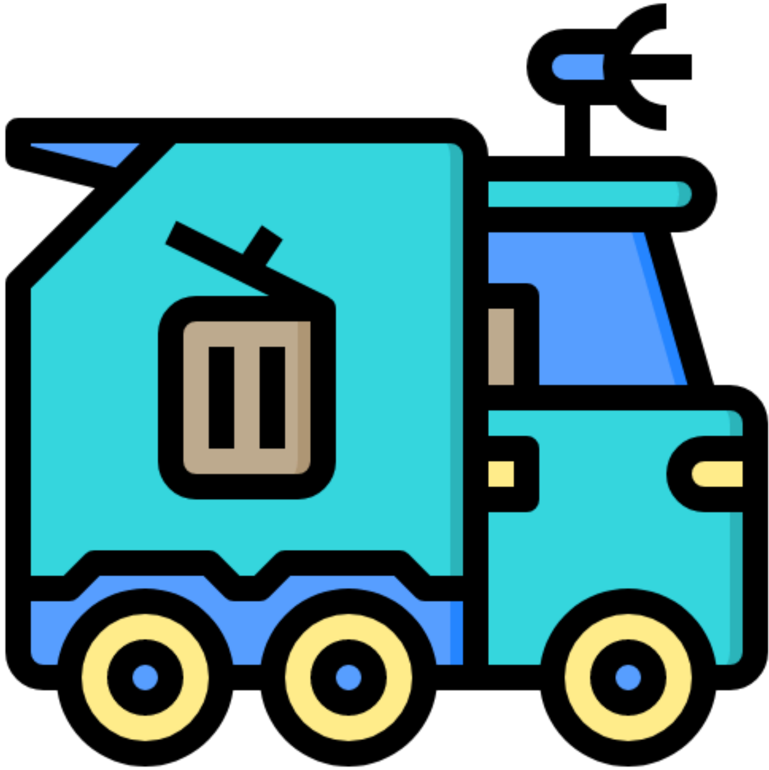
- When to discard messages in the log?
  - periodically checkpoint the state by multicasting CHECKPOINT messages
  - Each node collects *2f+1* checkpoint messages: proof of correctness





# Garbage Collection

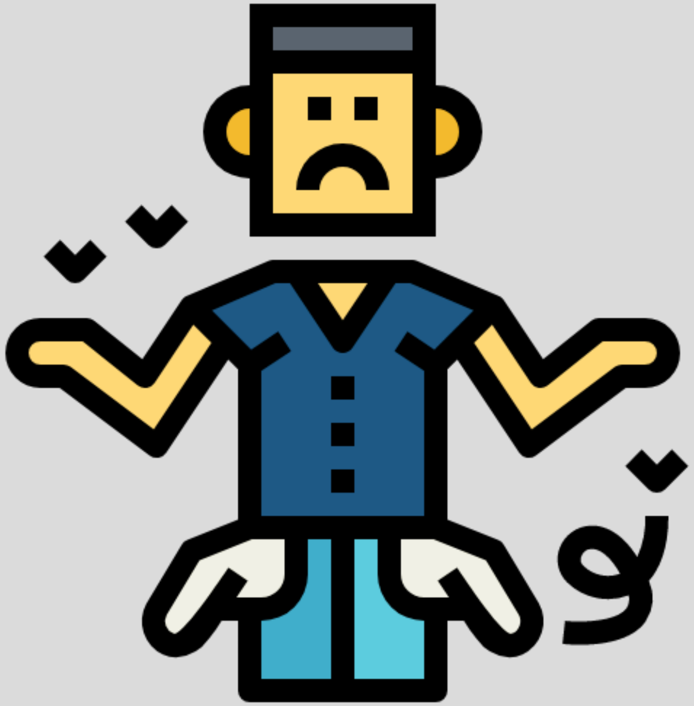
- When to discard messages in the log?
  - periodically checkpoint the state by multicasting CHECKPOINT messages
  - Each node collects *2f+1* checkpoint messages: proof of correctness



discard prior messages and checkpoints

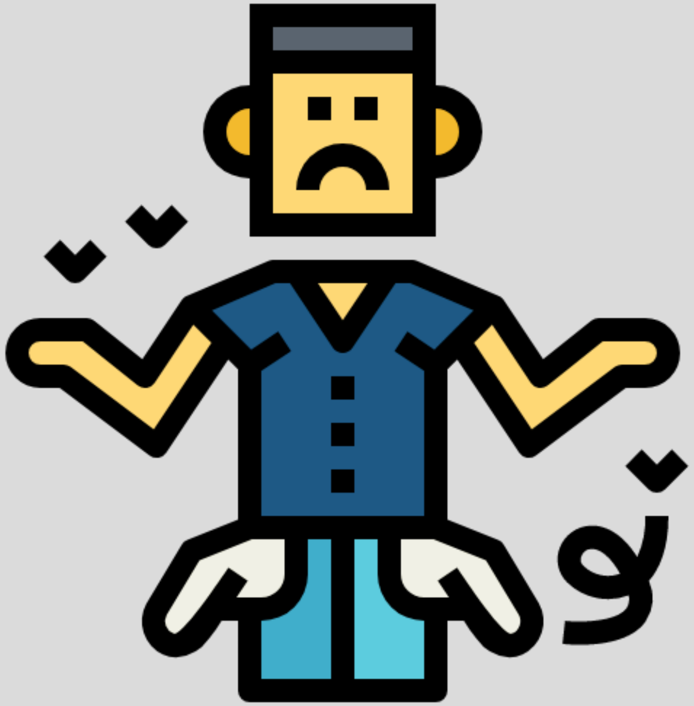
Stable checkpoint

# How to Deal with malicious Failures?



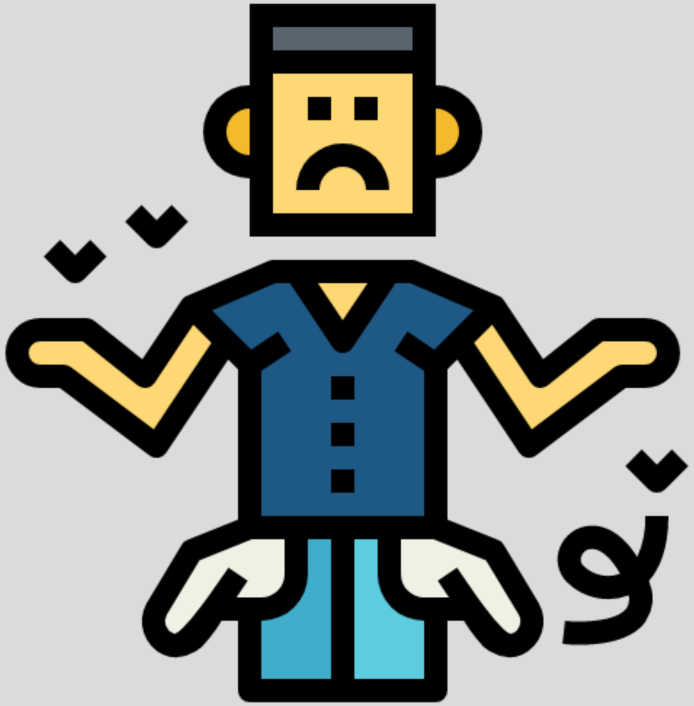
# How to Deal with malicious Failures?

- PBFT is an expensive protocol
  - 3 phases of communication
  - 3f+1 nodes
  - $O(n^2)$ message communication ($O(n^3)$ view change)



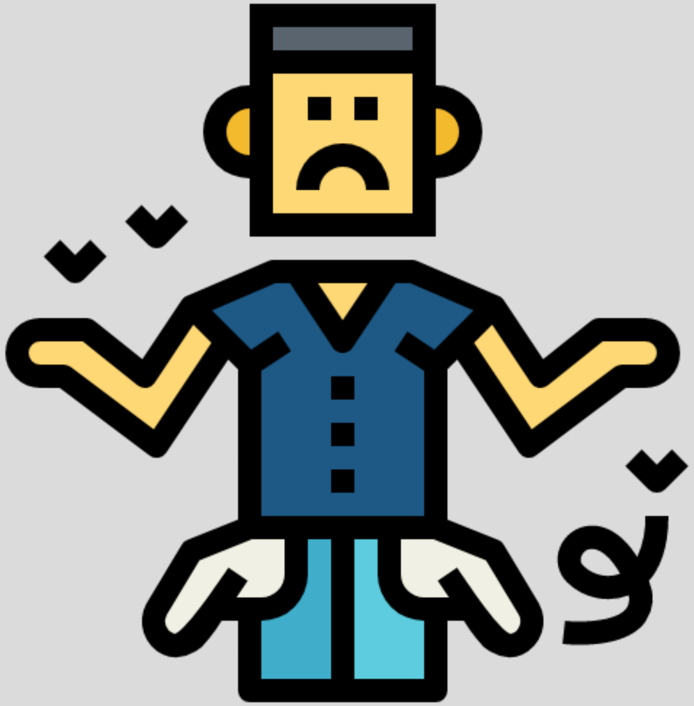
# How to Deal with malicious Failures?

- PBFT is an expensive protocol
  - 3 phases of communication
  - 3f+1 nodes
  - $O(n^2)$ message communication ($O(n^3)$ view change)
- Can an asynchronous protocol perform better?



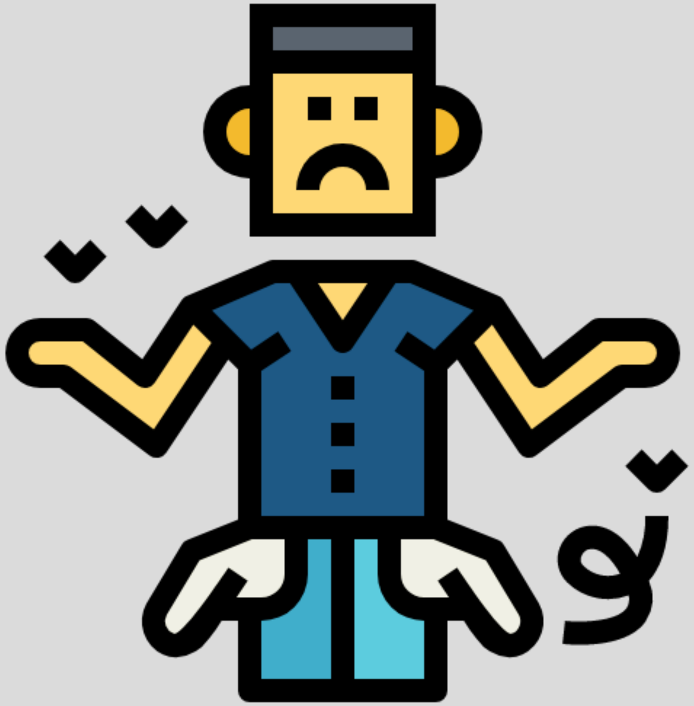
# How to Deal with malicious Failures?

- PBFT is an expensive protocol
  - 3 phases of communication
  - 3f+1 nodes
  - $O(n^2)$ message communication ($O(n^3)$ view change)
- Can an asynchronous protocol perform better?
  - Optimistic Approaches
    - Execute transactions without ordering [Zyzzyva]
    - Active/passive replication [CheapBFT]



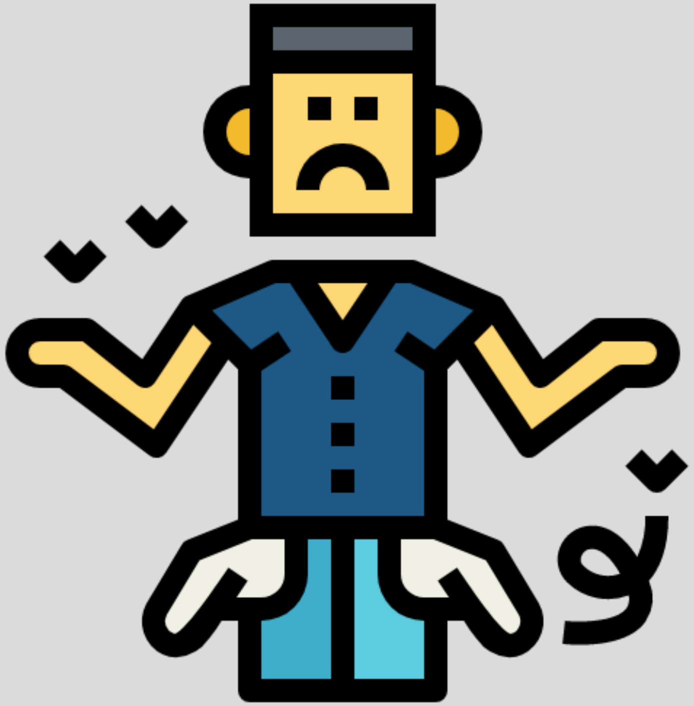
# How to Deal with malicious Failures?

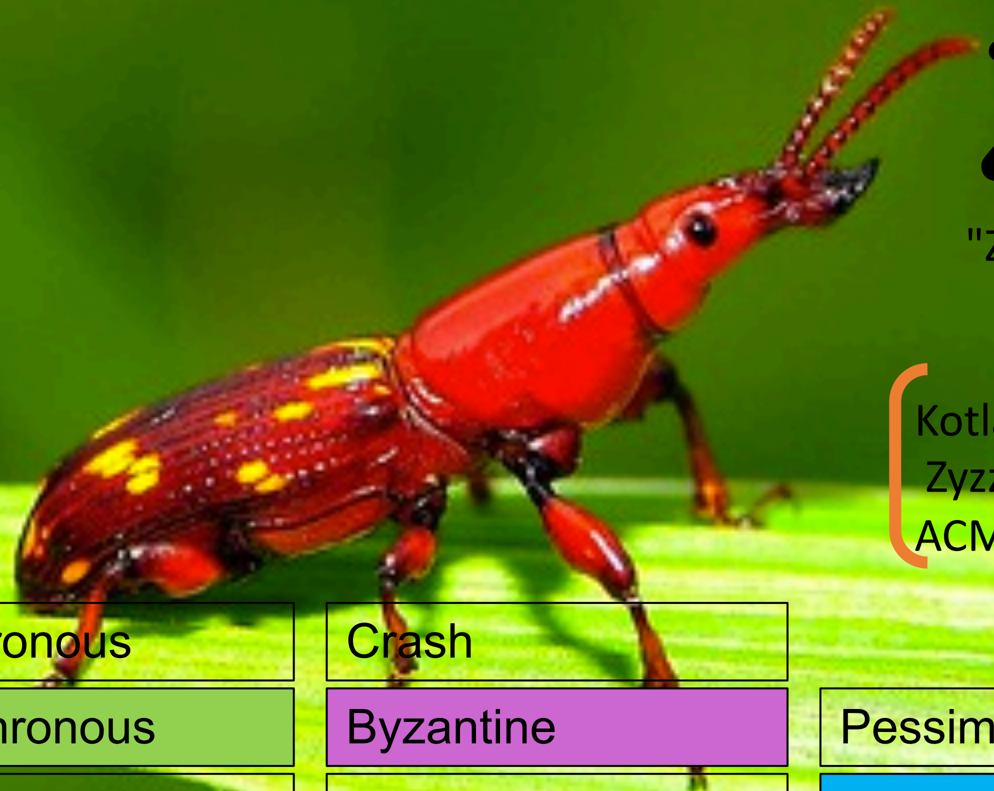
- PBFT is an expensive protocol
  - 3 phases of communication
  - 3f+1 nodes
  - $O(n^2)$ message communication ($O(n^3)$ view change)
- Can an asynchronous protocol perform better?
  - Optimistic Approaches
    - Execute transactions without ordering [Zyzzyva]
    - Active/passive replication [CheapBFT]
  - Restrict the malicious behavior of nodes
    - Trusted hardware [MinBFT][CheapBFT]



# How to Deal with malicious Failures?

- PBFT is an expensive protocol
  - 3 phases of communication
  - 3f+1 nodes
  - $O(n^2)$ message communication ($O(n^3)$ view change)
- Can an asynchronous protocol perform better?
  - Optimistic Approaches
    - Execute transactions without ordering [Zyzzyva]
    - Active/passive replication [CheapBFT]
  - Restrict the malicious behavior of nodes
    - Trusted hardware [MinBFT][CheapBFT]
  - Explore a spectrum of performance Trade-off between different complexity metrics
    - Reduce the number of phases (increase the number of nodes) [FaB]
    - Reduce message complexity (increase the number of phases) [HotStuff]





# ZYZZYVA

"Zyzzyva" is the last word in many English-language dictionaries

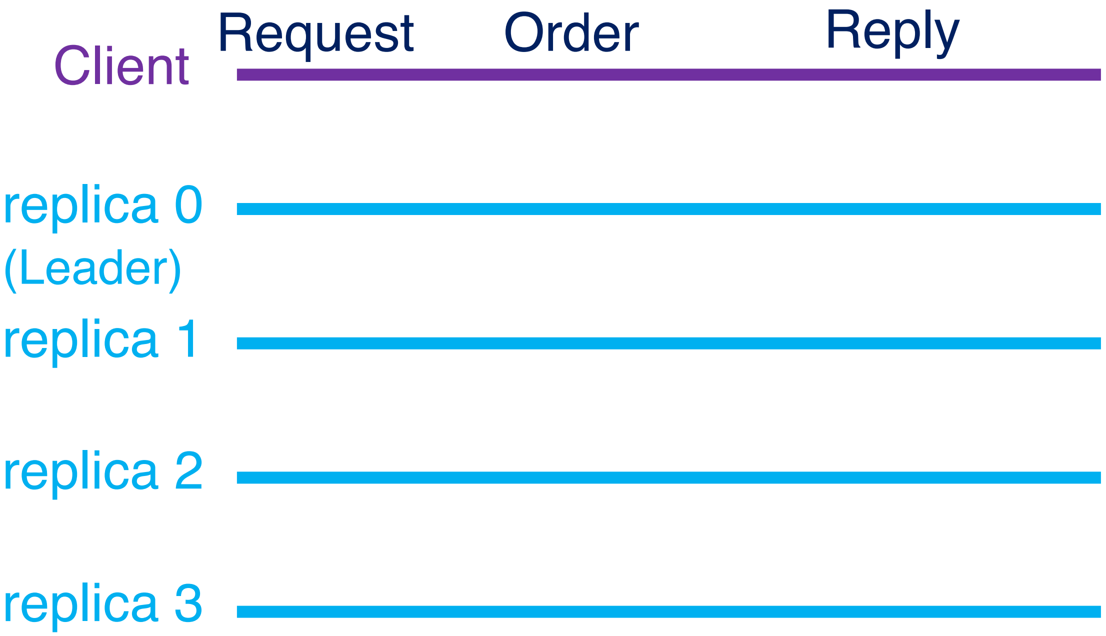
Kotla, R., Alvisi, L., Dahlin, M., Clement, A., & Wong, E.
Zyzzyva: speculative byzantine fault tolerance.
ACM SIGOPS Operating Systems Review, 2007

| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | **3f+1 nodes** |
| **Asynchronous** | **Byzantine** | Pessimistic | **Known nodes** | **1 or 3 phases** |
| Partially-Synchronous | Hybrid | **Optimistic** | Unknown nodes | **O(N) Complexity** |

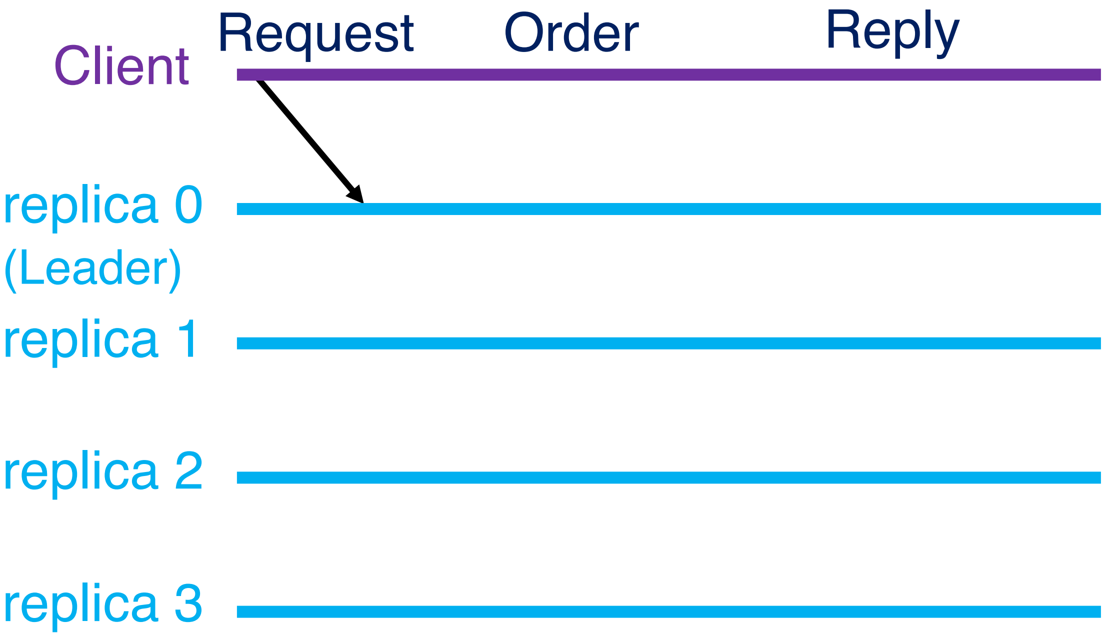
# Zyzzyva: Speculative BFT

- A replica speculatively executes a request as soon as it receives a valid pre-prepare message
- Commitment of a request is moved to the client
  - If a request completes at a client, the request will eventually be committed at the server replicas
- Prepare and commit phases are reduced to a single linear phase
  - View change has one more additional phase

# Zyzzyva Agreement Protocol: Case 1



# Zyzzyva Agreement Protocol: Case 1



Request Order Reply

Client

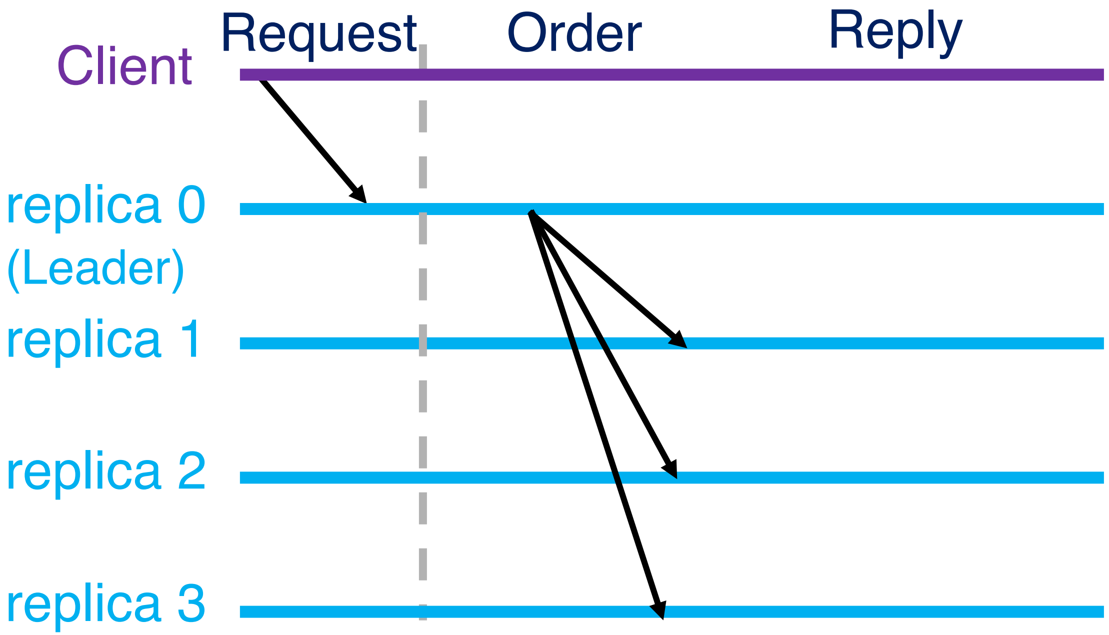
replica 0
(Leader)

replica 1

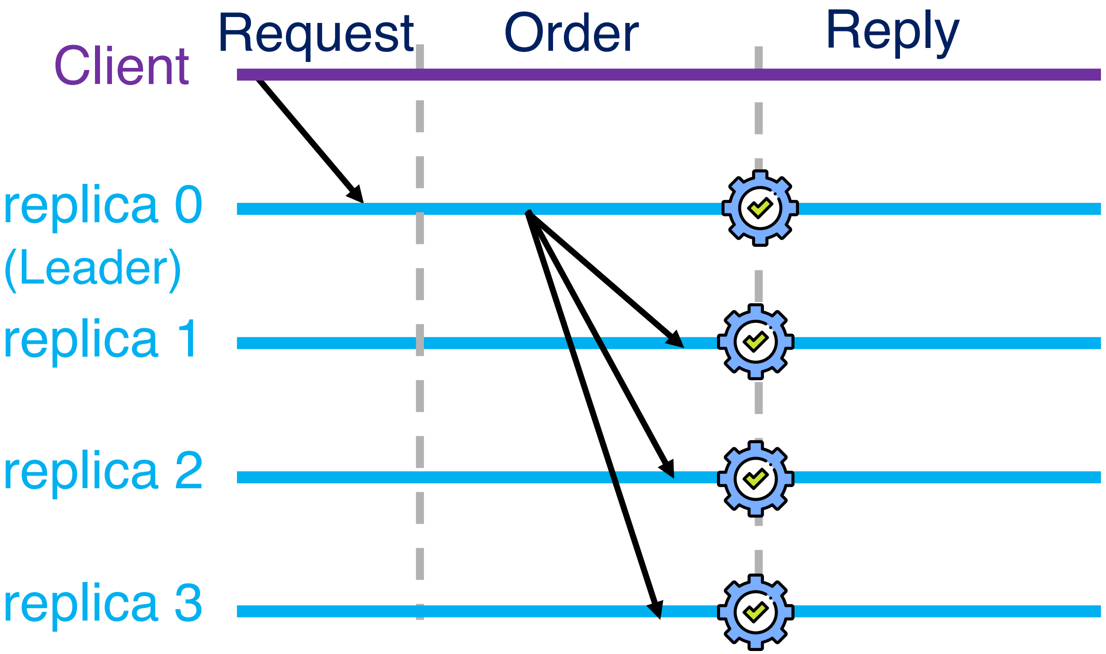
replica 2

replica 3

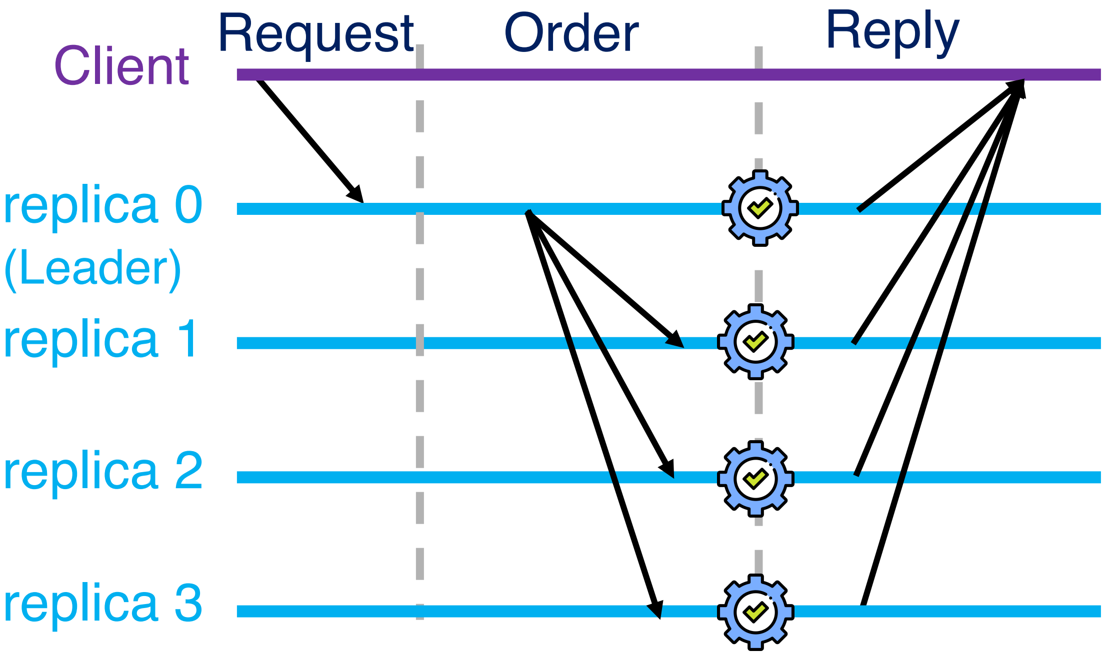
# Zyzzyva Agreement Protocol: Case 1



# Zyzzyva Agreement Protocol: Case 1



# Zyzzyva Agreement Protocol: Case 1

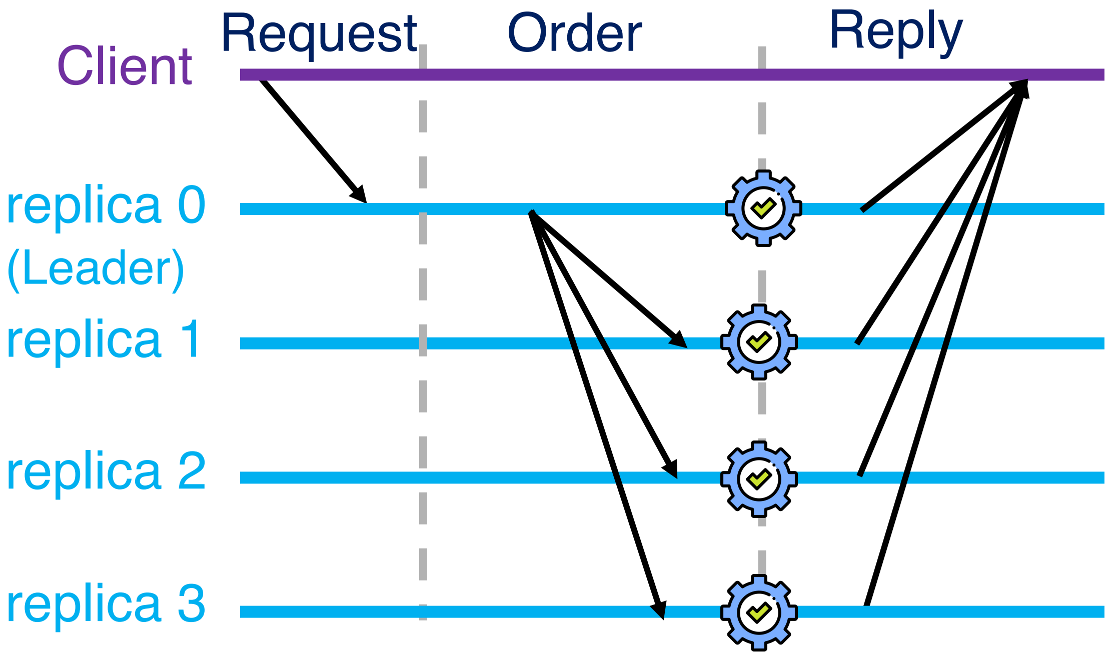


Receives 3f+1
matching replies

# Zyzzyva Agreement Protocol: Case 1

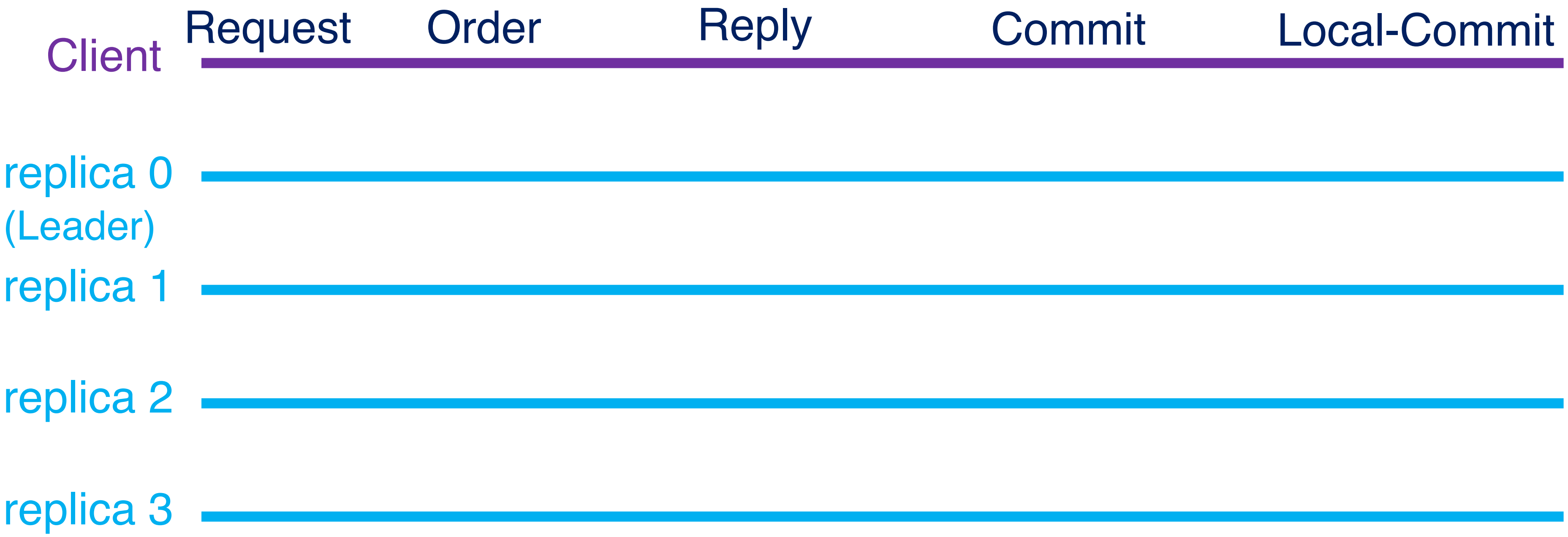
- Client receives 3f+1 matching replies

=> all replicas have executed the request in the same total order



Receives 3f+1 matching replies

# Zyzzyva Agreement Protocol: Case 2



Request Order Reply Commit Local-Commit

Client

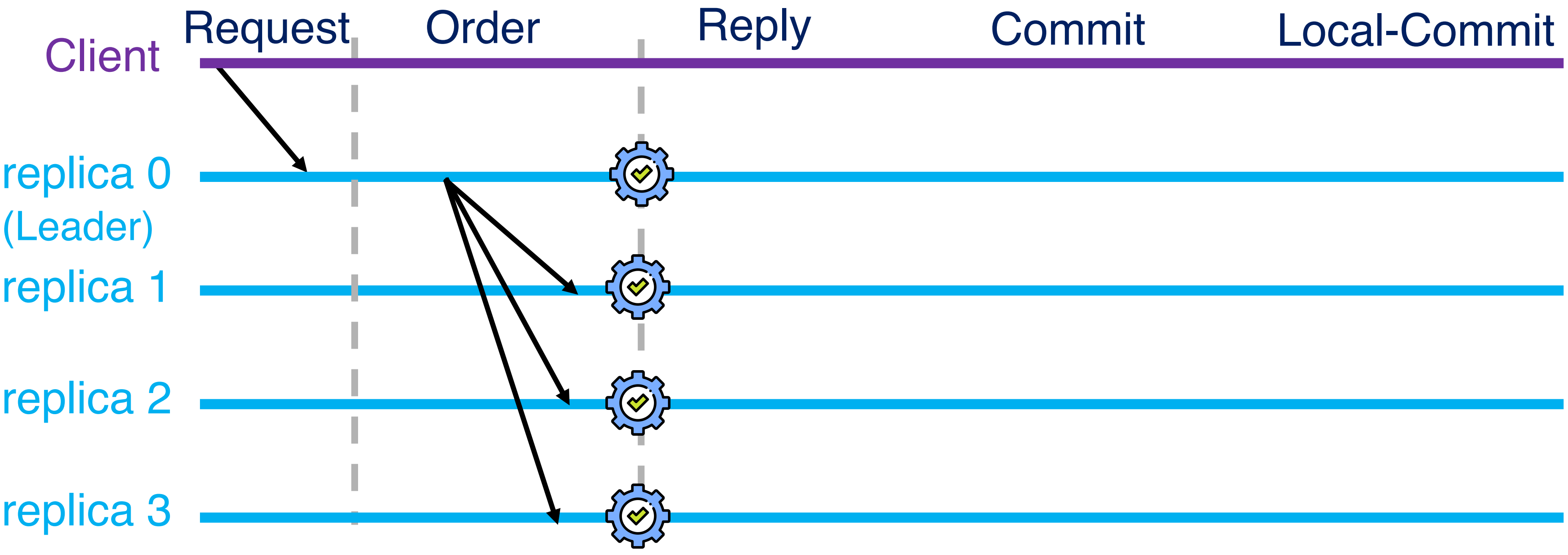
replica 0
(Leader)

replica 1

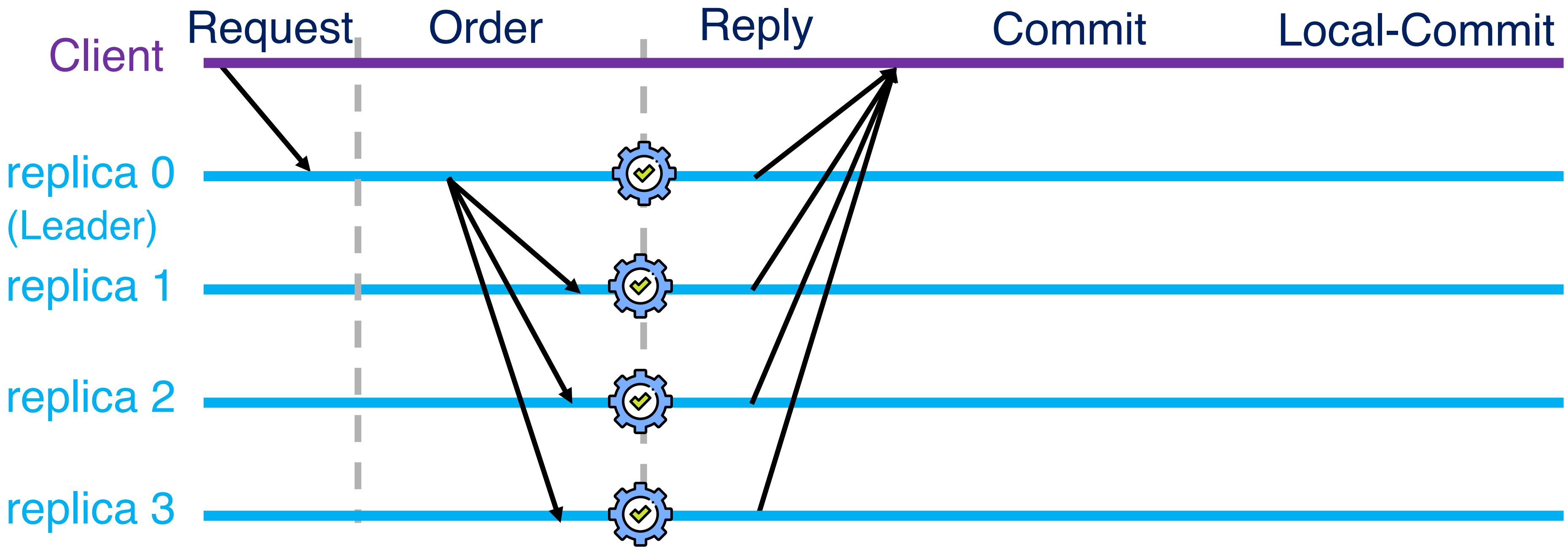
replica 2

replica 3

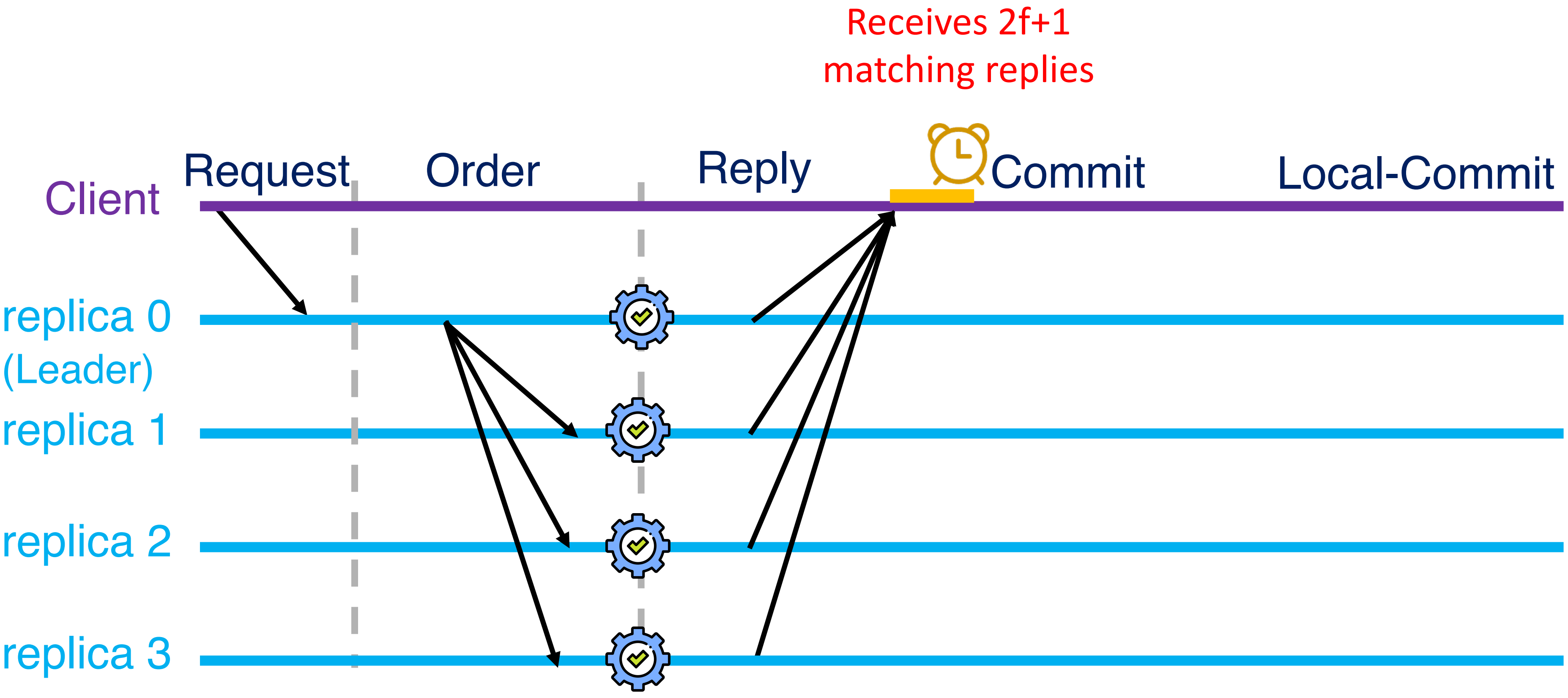
# Zyzzyva Agreement Protocol: Case 2



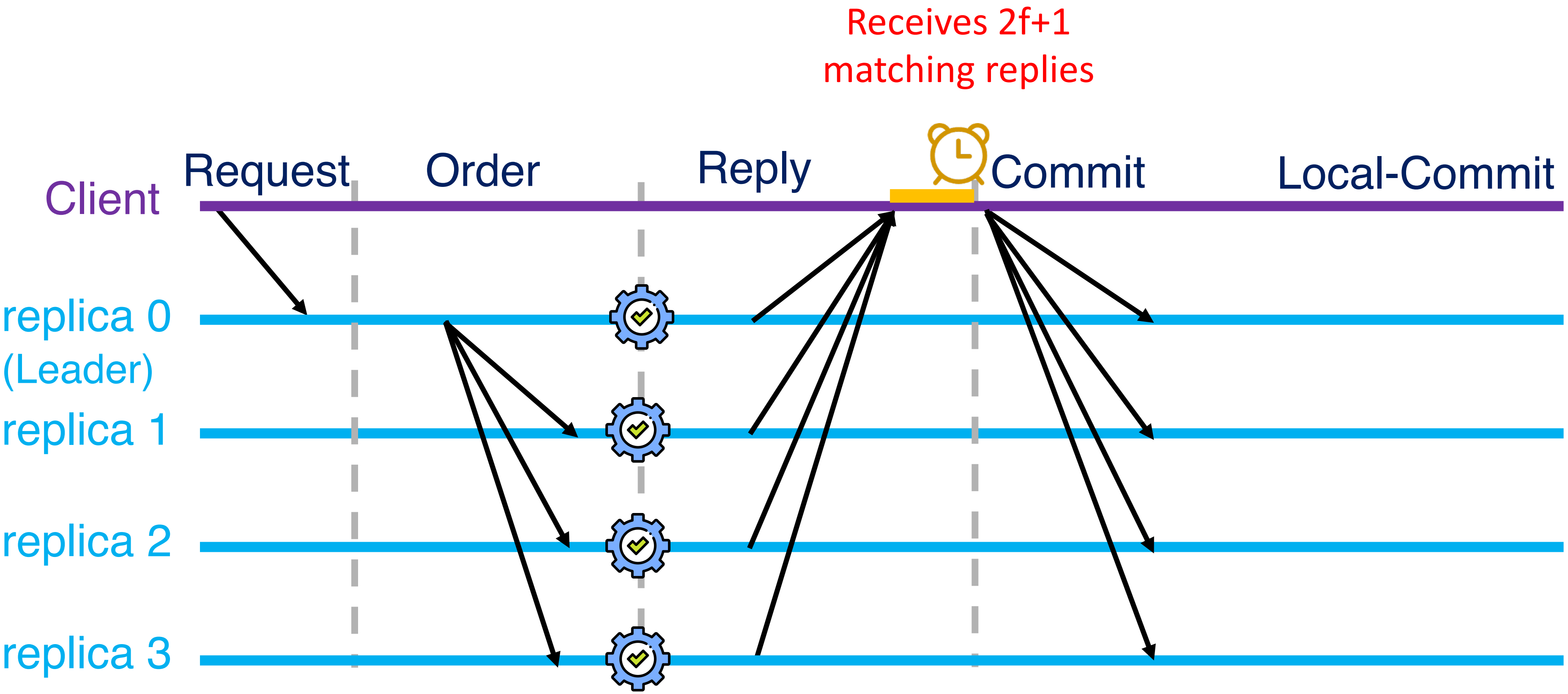
# Zyzzyva Agreement Protocol: Case 2



# Zyzzyva Agreement Protocol: Case 2



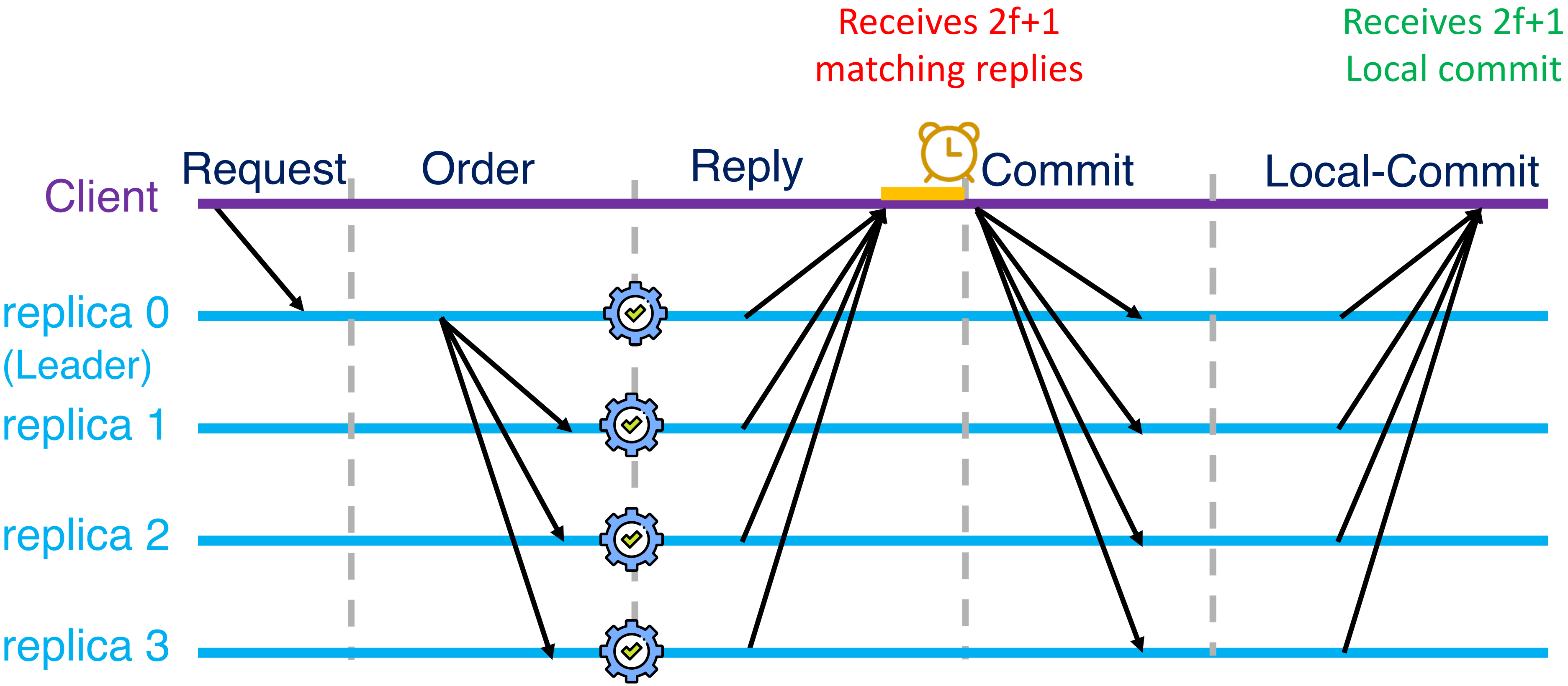
# Zyzzyva Agreement Protocol: Case 2



Commit message contains a commit certificate:

A list of 2f+1 replica ids and their signed messages

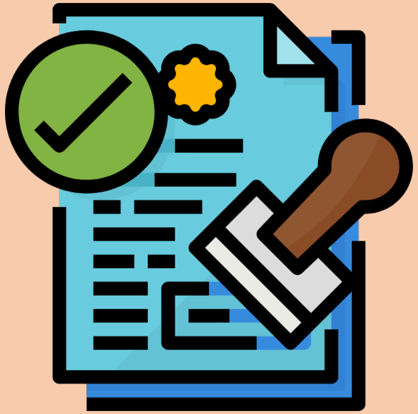
# Zyzzyva Agreement Protocol: Case 2



Commit message contains a commit certificate:

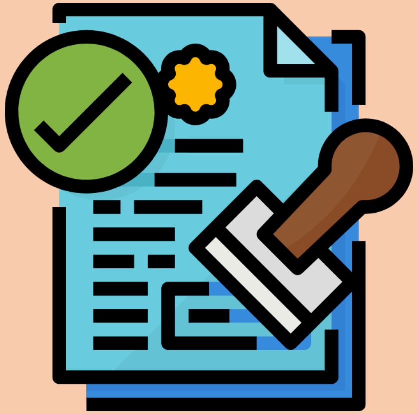
A list of 2f+1 replica ids and their signed messages

# Zyzzyva Summary

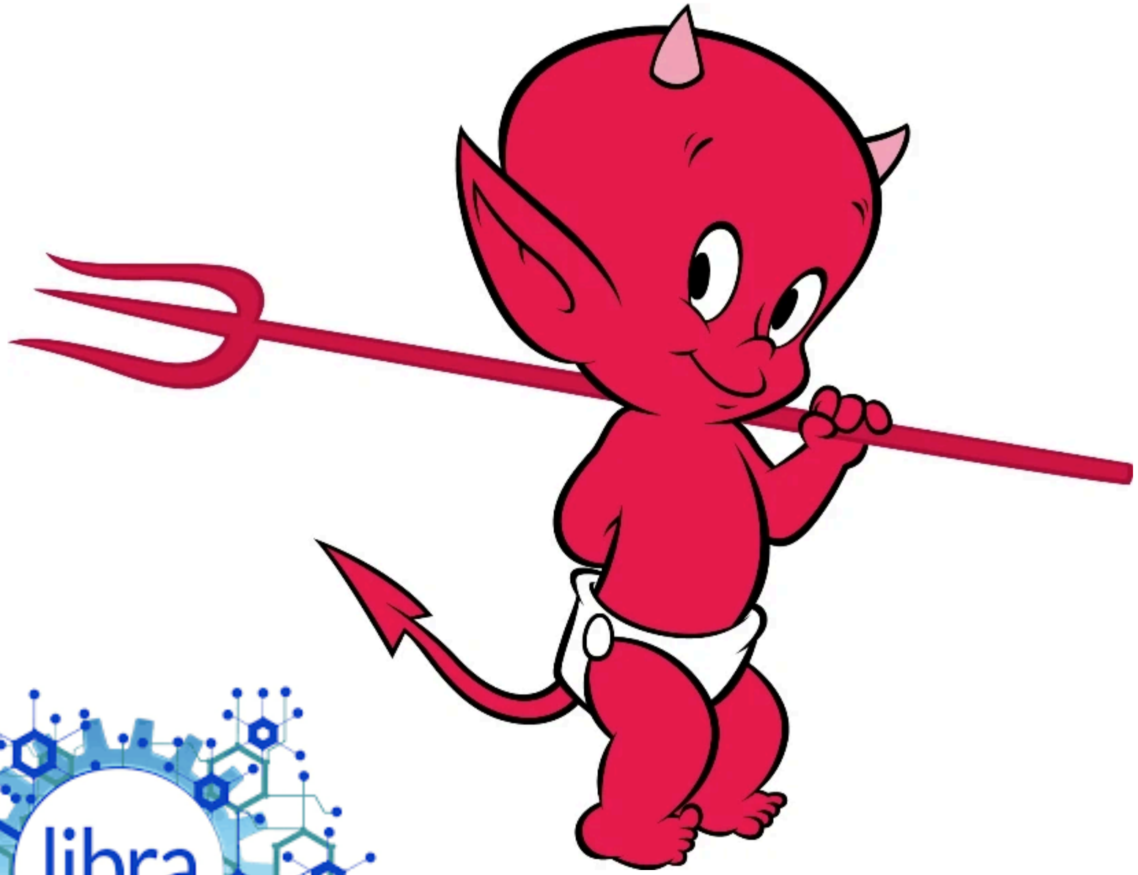


# Zyzzyva Summary

- One round of message exchange during normal operation



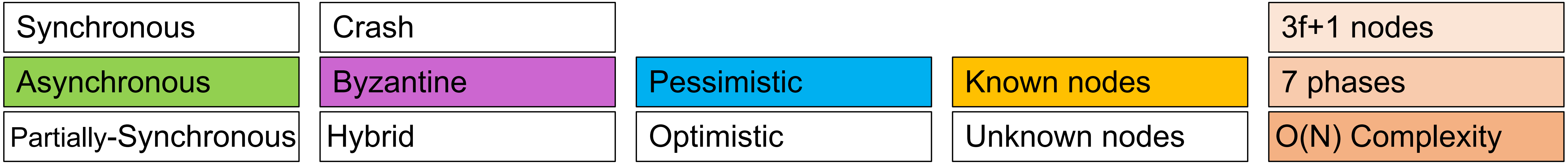
- Impact on view change
  - Need an additional round of message exchange



# HOTSTUFF

Yin, Maofan, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. "HotStuff: Bft consensus with linearity and responsiveness." In *PODC,* 2019.

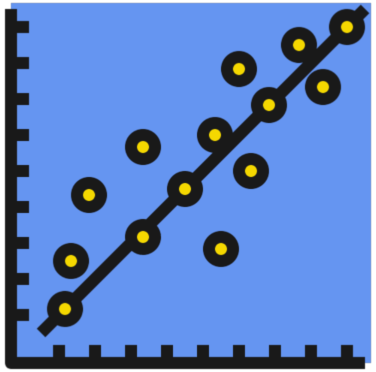
- **Linear Communication**
- **Request Pipelining**
- **Leader Rotation**



| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | 3f+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 7 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O(N) Complexity |

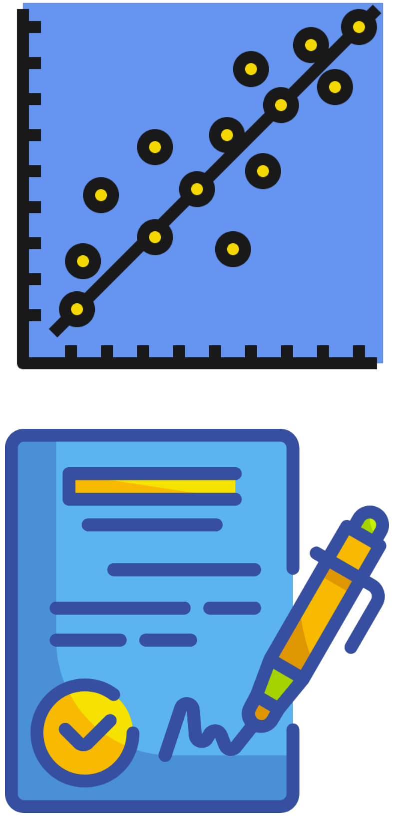
# HotStuff Model

- The same network and quorum size as PBFT
  - 3f+1 nodes in total, Quorums of 2f+1 nodes



# HotStuff Model

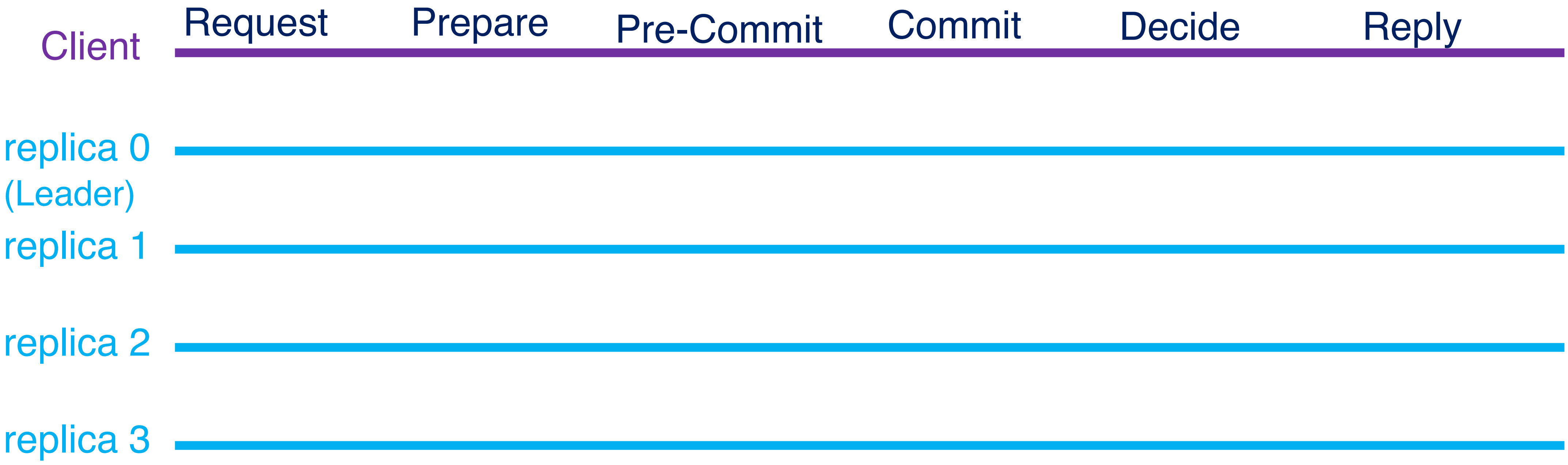
- The same network and quorum size as PBFT
  - 3f+1 nodes in total, Quorums of 2f+1 nodes
- Linear message complexity
  - Increases the number of phases
  - Each n to n phase of PBFT = an n to 1 + a 1 to n phases of Hotstuff
  - The primary uses (k, n)-threshold signature schema



# HotStuff Model

- The same network and quorum size as PBFT
  - 3f+1 nodes in total, Quorums of 2f+1 nodes
- Linear message complexity
  - Increases the number of phases
  - Each n to n phase of PBFT = an n to 1 + a 1 to n phases of Hotstuff
  - The primary uses (k, n)-threshold signature schema
- Leader Rotation
  - A leader is rotated after a single attempt to commit a command/block
  - View-change is part of the normal operation of the system
    - One more phase of communication is needed
    - Linear View change routine
    - PBFT’s View Change has $O(n^3)$ message complexity

# HotStuff Agreement Protocol



Request Prepare Pre-Commit Commit Decide Reply

Client

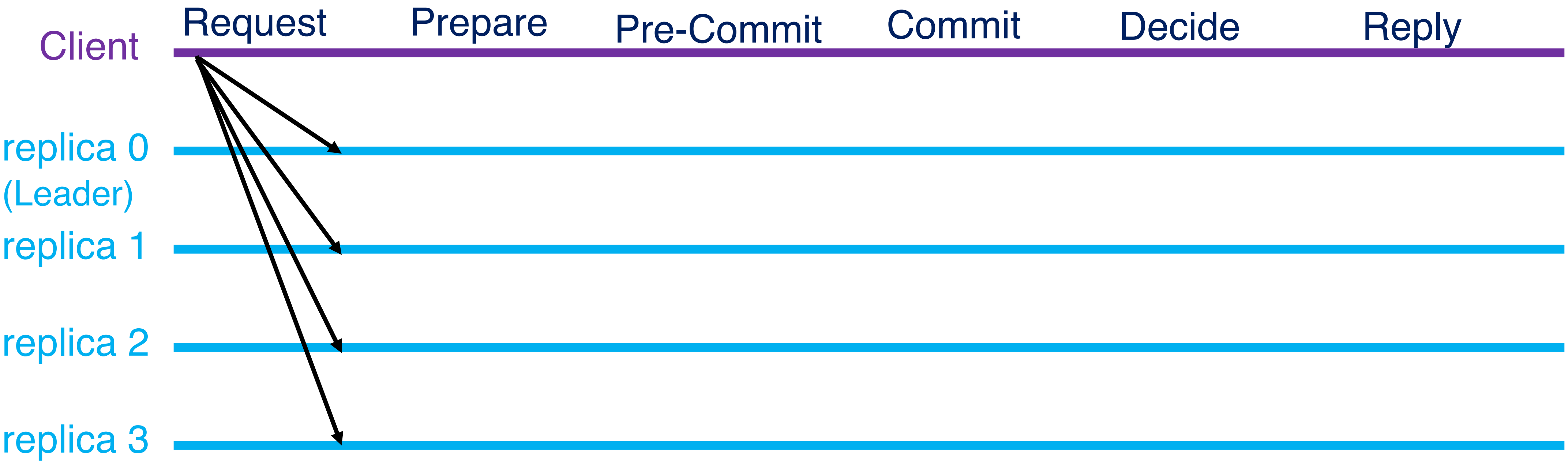
replica 0
(Leader)

replica 1

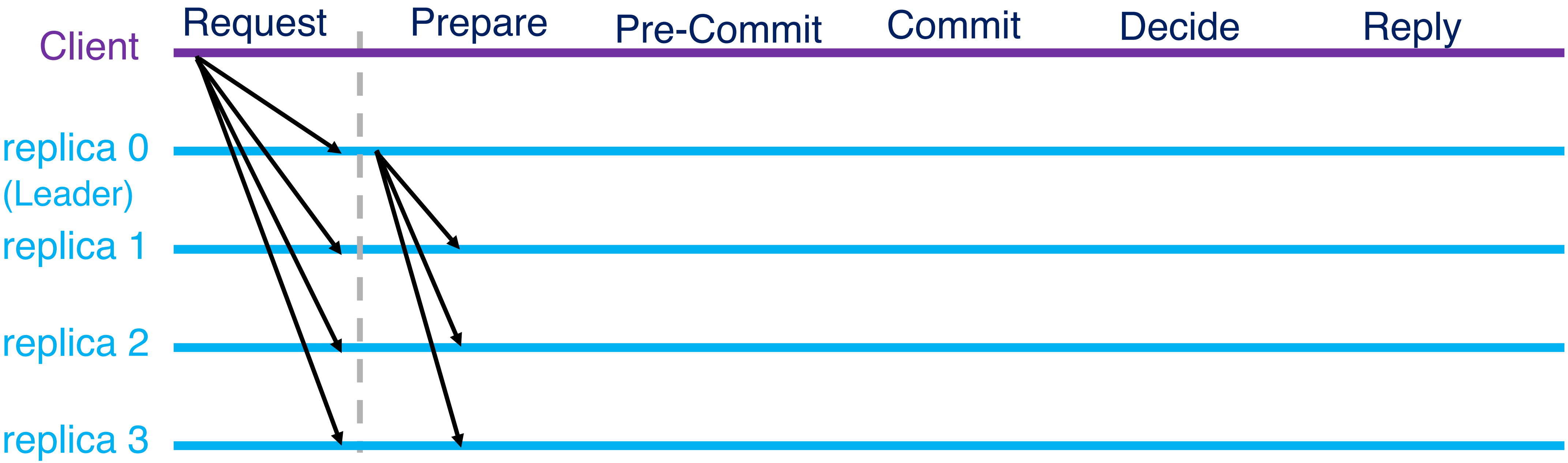
replica 2

replica 3

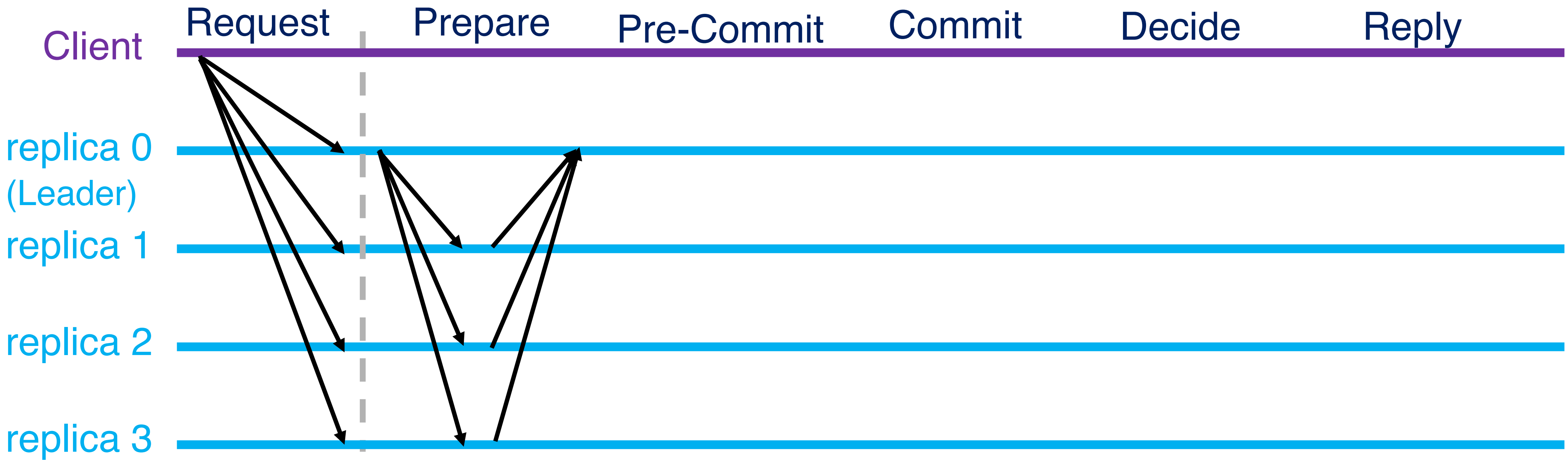
# HotStuff Agreement Protocol



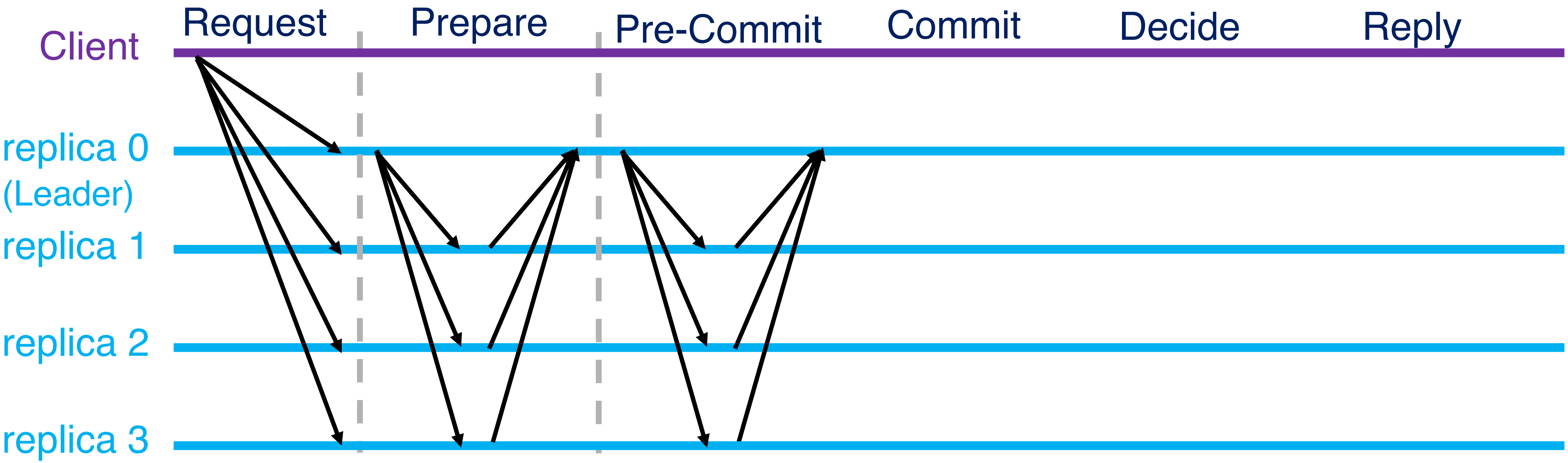
# HotStuff Agreement Protocol



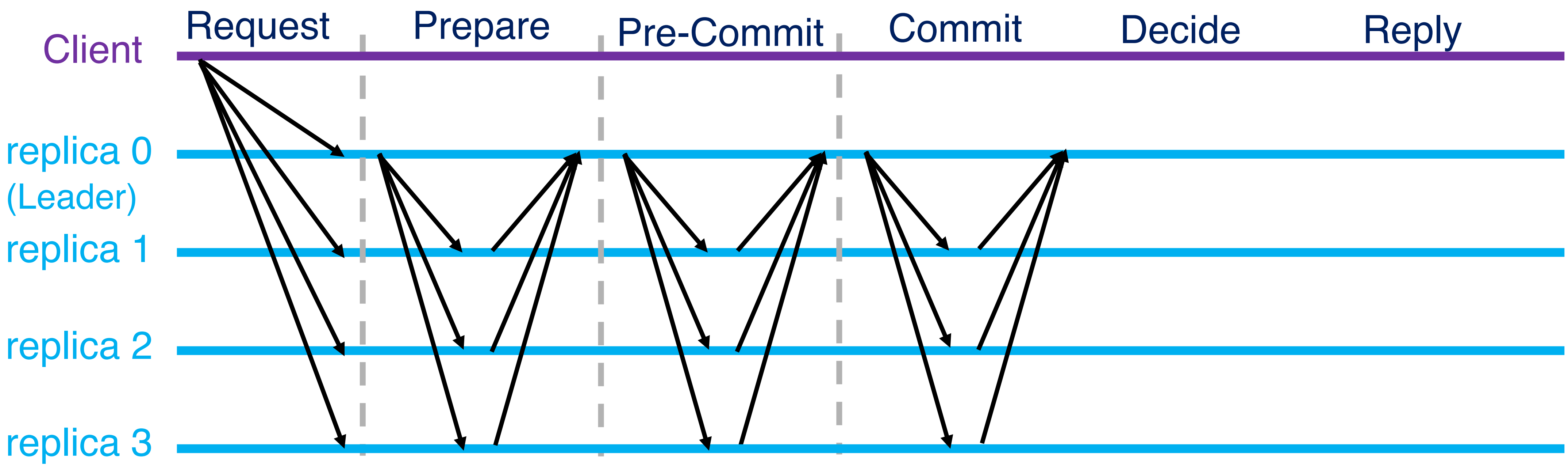
# HotStuff Agreement Protocol



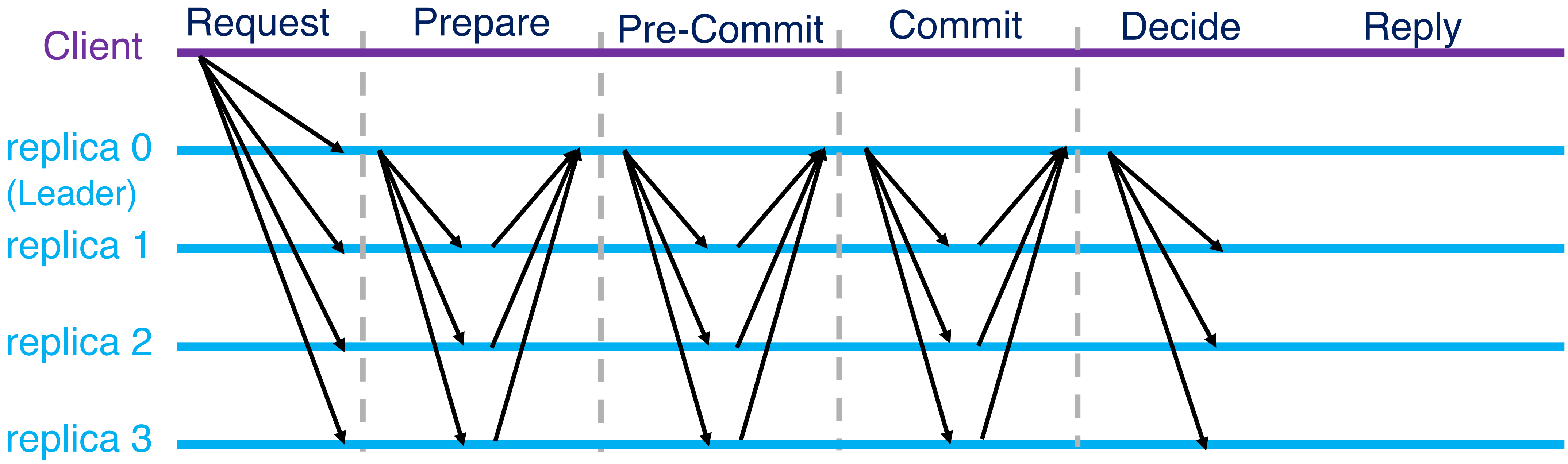
# HotStuff Agreement Protocol



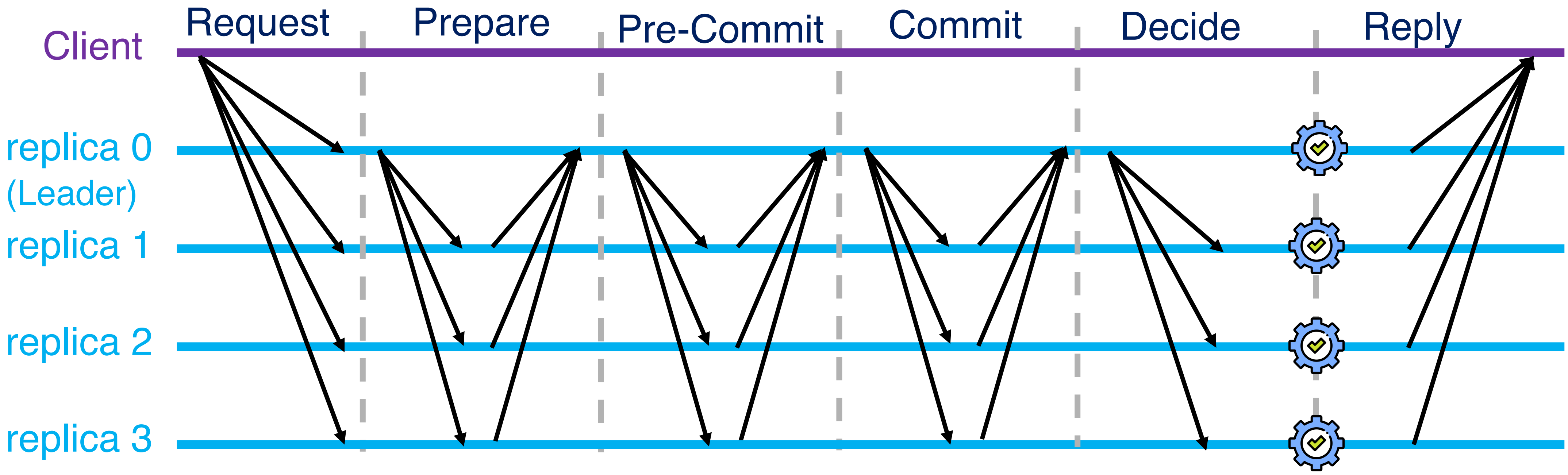
# HotStuff Agreement Protocol



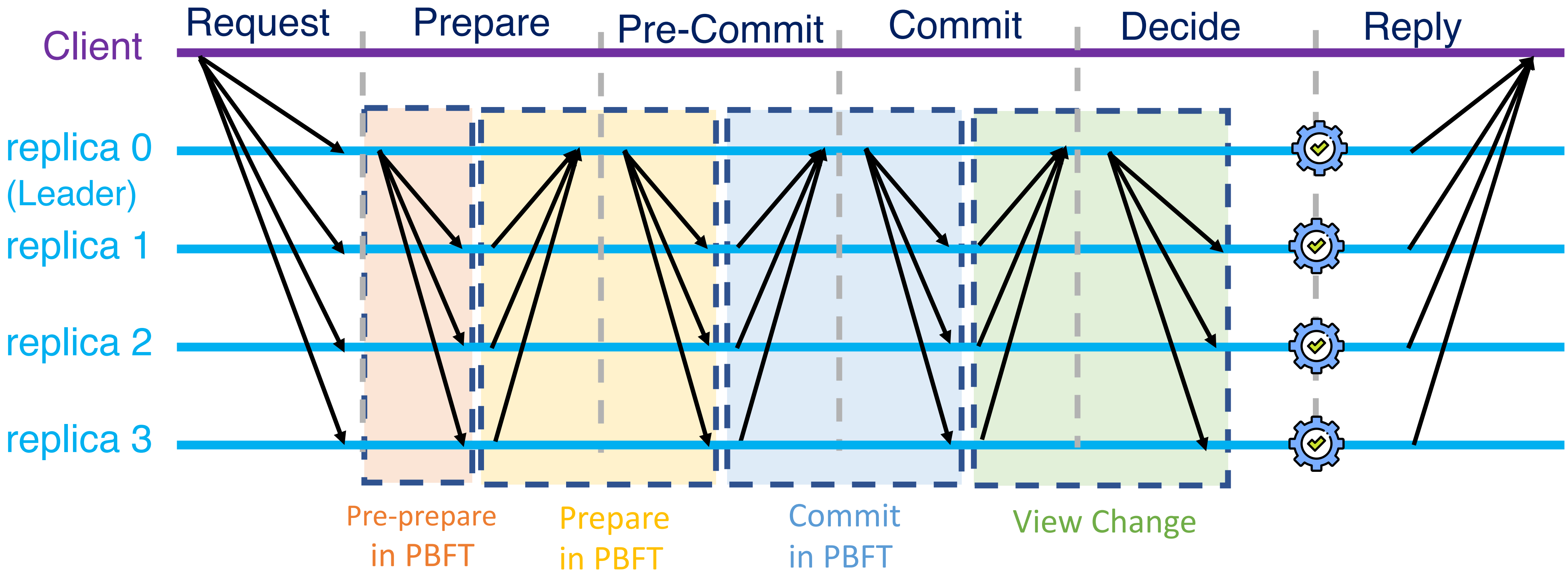
# HotStuff Agreement Protocol



# HotStuff Agreement Protocol



# HotStuff Agreement Protocol



# The Pipeline of HotStuff

clients

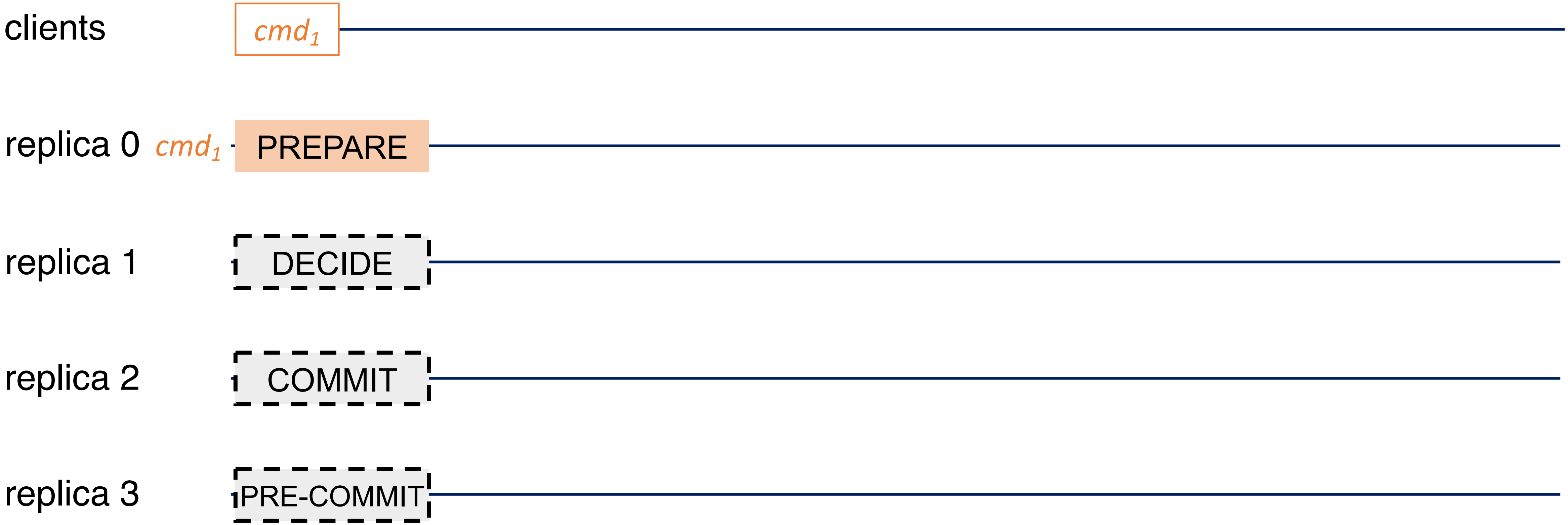
replica 0

replica 1

replica 2

replica 3

# The Pipeline of HotStuff



clients $cmd_1$

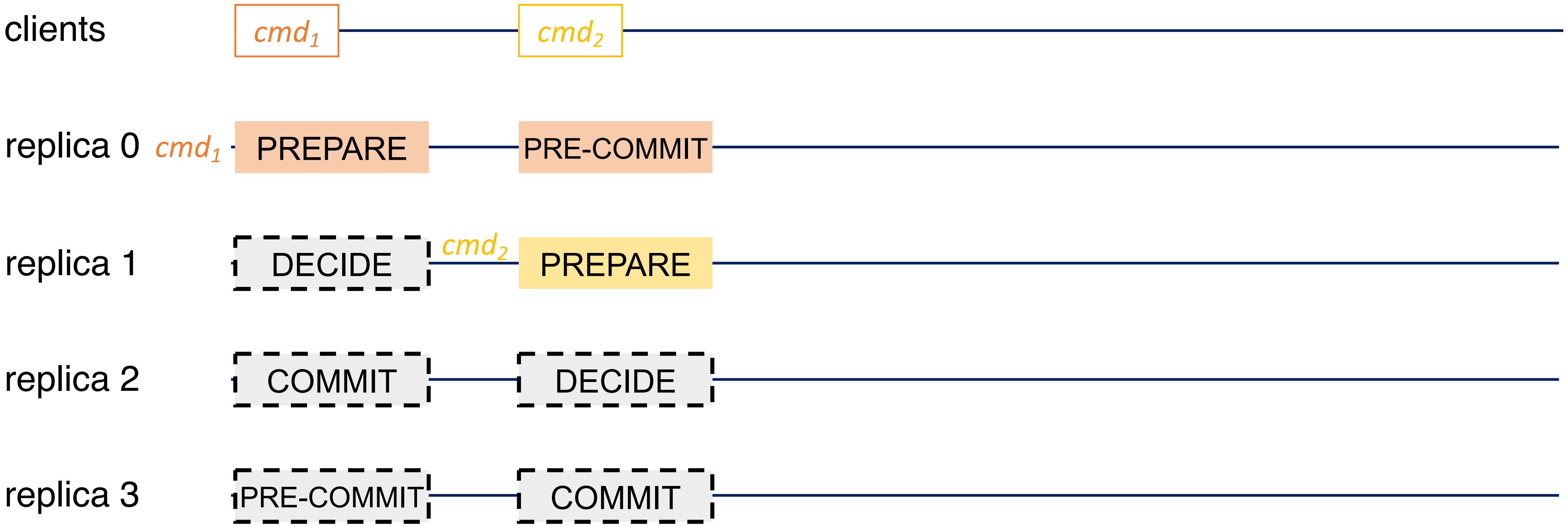
replica 0 $cmd_1$ PREPARE

replica 1 DECIDE

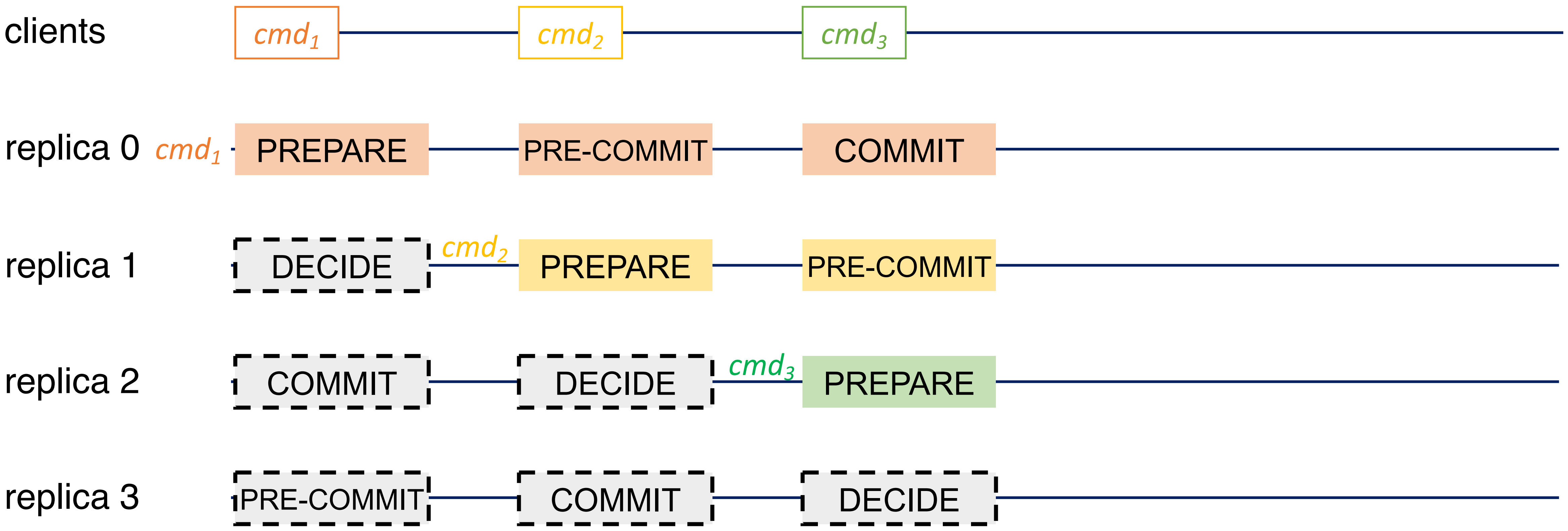
replica 2 COMMIT

replica 3 PRE-COMMIT

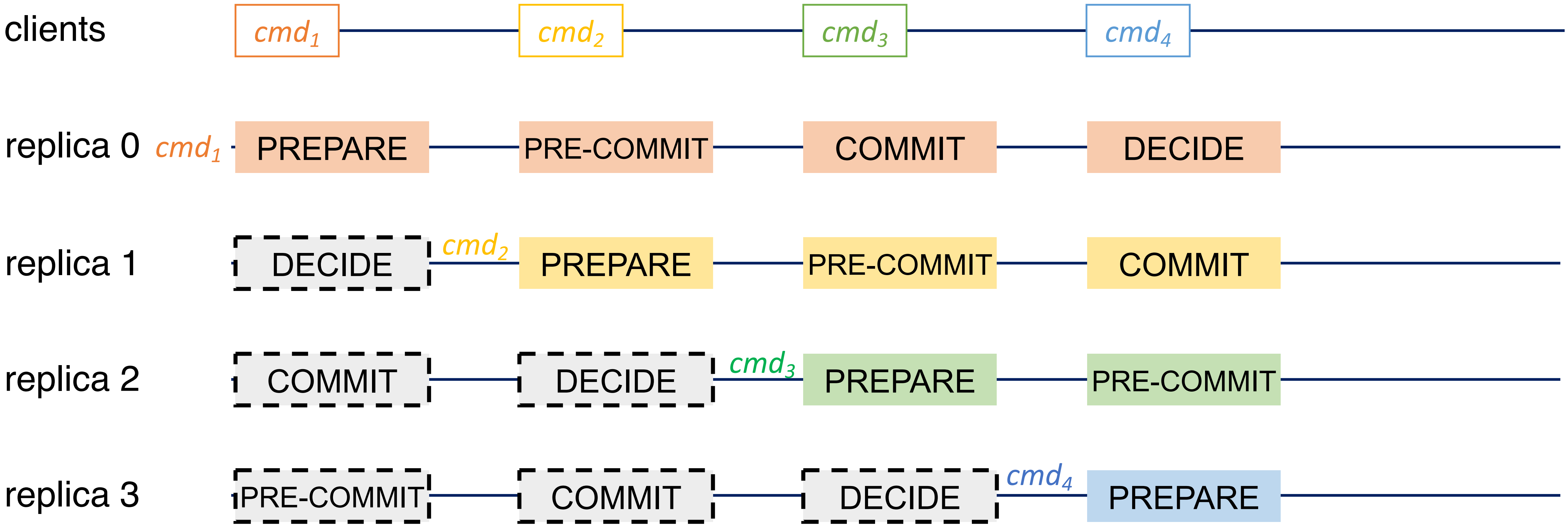
# The Pipeline of HotStuff



# The Pipeline of HotStuff

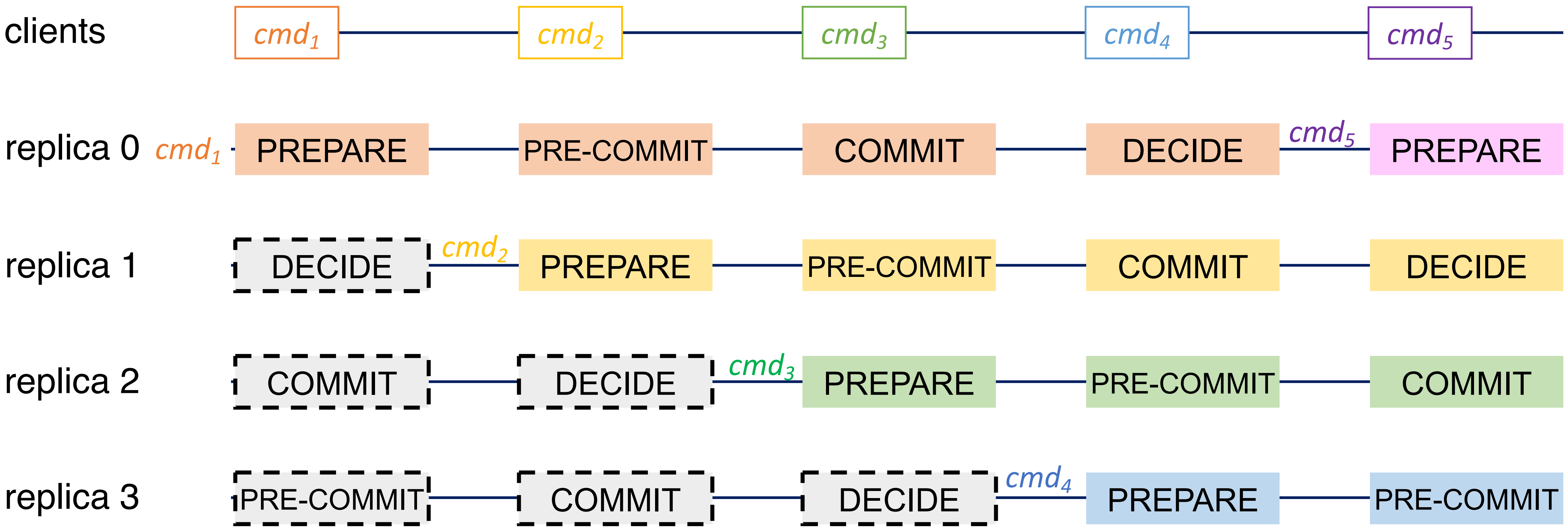


# The Pipeline of HotStuff



| | | | | |
|---|---|---|---|---|
| clients | $cmd_1$ | $cmd_2$ | $cmd_3$ | $cmd_4$ |
| replica 0 | $cmd_1$ PREPARE | PRE-COMMIT | COMMIT | DECIDE |
| replica 1 | DECIDE | $cmd_2$ PREPARE | PRE-COMMIT | COMMIT |
| replica 2 | COMMIT | DECIDE | $cmd_3$ PREPARE | PRE-COMMIT |
| replica 3 | PRE-COMMIT | COMMIT | DECIDE | $cmd_4$ PREPARE |

# The Pipeline of HotStuff



| | | | | | |
|---|---|---|---|---|---|
| clients | | $cmd_1$ | $cmd_2$ | $cmd_3$ | $cmd_4$ | $cmd_5$ |
| replica 0 | $cmd_1$ | PREPARE | PRE-COMMIT | COMMIT | DECIDE | $cmd_5$ PREPARE |
| replica 1 | | DECIDE | $cmd_2$ PREPARE | PRE-COMMIT | COMMIT | DECIDE |
| replica 2 | | COMMIT | DECIDE | $cmd_3$ PREPARE | PRE-COMMIT | COMMIT |
| replica 3 | | PRE-COMMIT | COMMIT | DECIDE | $cmd_4$ PREPARE | PRE-COMMIT |

# MinBFT

Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., & Verissimo, P. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers, 2011.*

**Trusted Hardware**

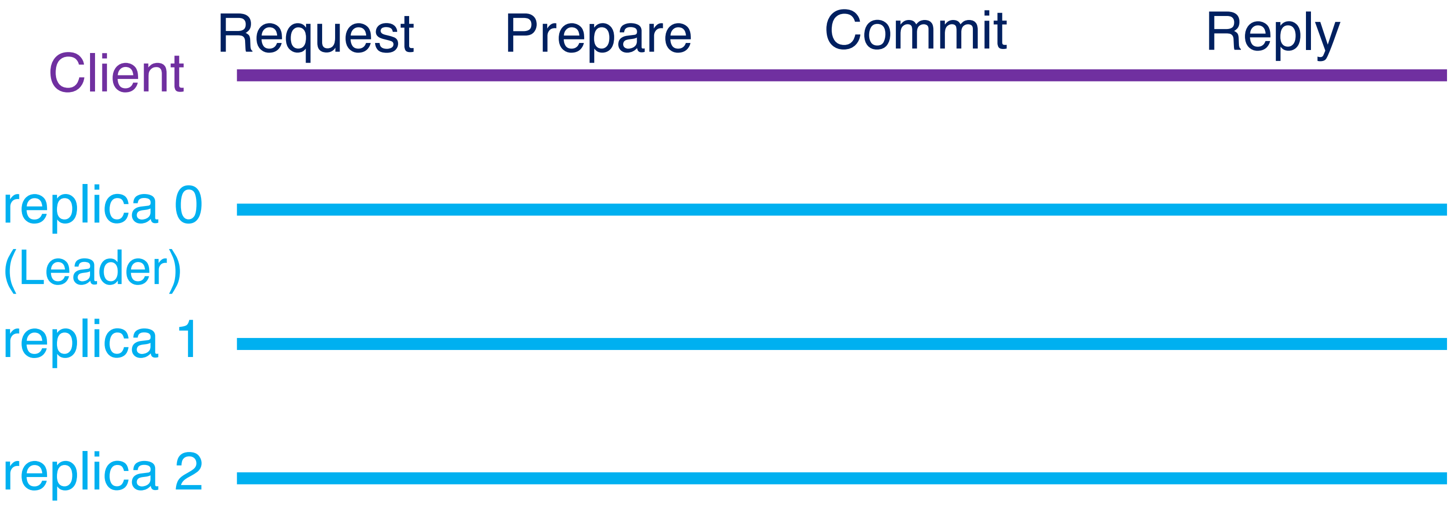
| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | 2f+1 nodes |
| [✓ Asynchronous] | [✓ Byzantine] | [✓ Pessimistic] | [✓ Known nodes] | 2 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O(N) Complexity |

# MinBFT

- Uses a tamper proof component: Unique Sequential Identifier Generator (USIG)
- All nodes use USIG for message authentication and verification to ensure receiving symmetric messages
  - A Byzantine node may decide not to send a message or send it corrupted, but it can not send two different messages to different replicas
- USIG generates unique identifiers for every message
  - Each identifier is assigned incrementally
  - Each identifier is the successor of the previous one.

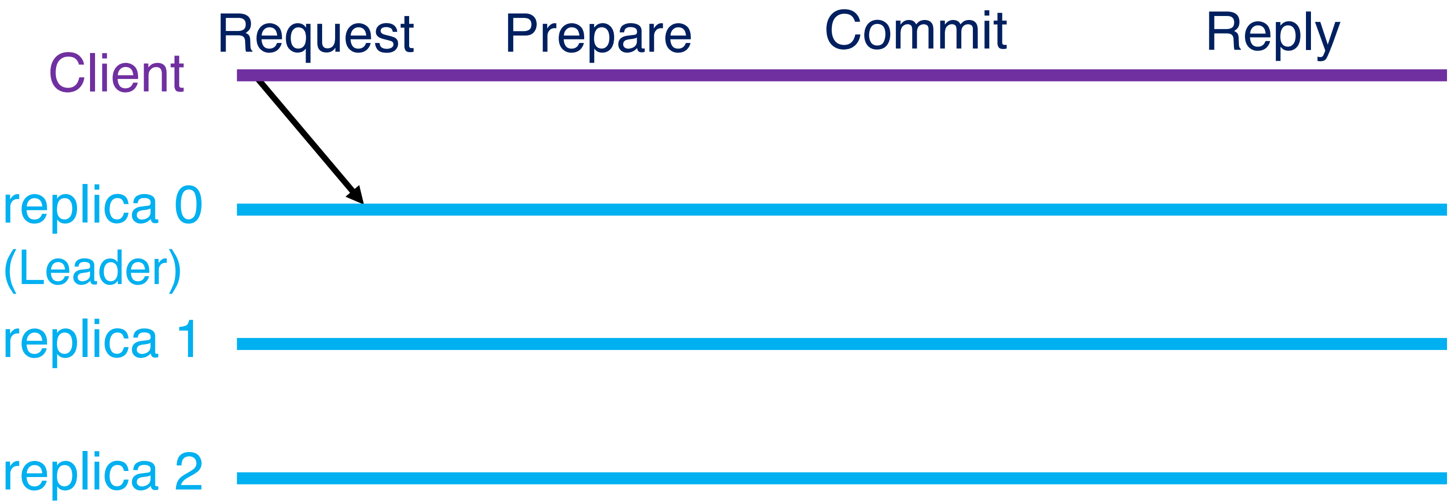
# MinBFT Agreement Protocol

Requires the same number of replicas, communication phases and message complexity as Paxos



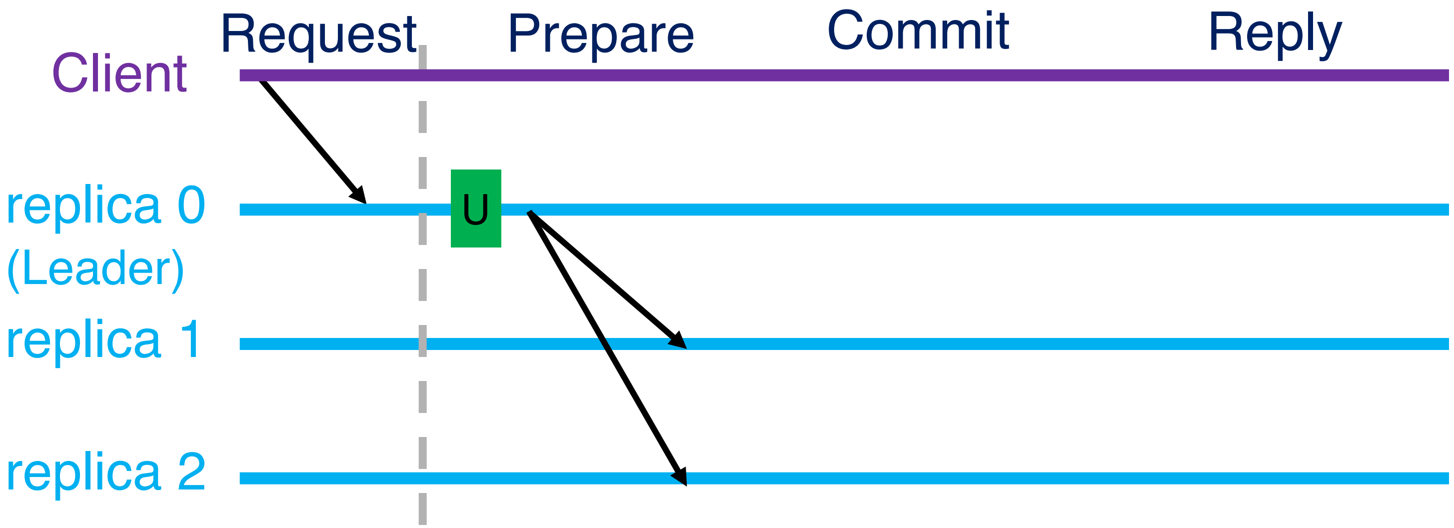
# MinBFT Agreement Protocol

Requires the same number of replicas, communication phases and message complexity as Paxos



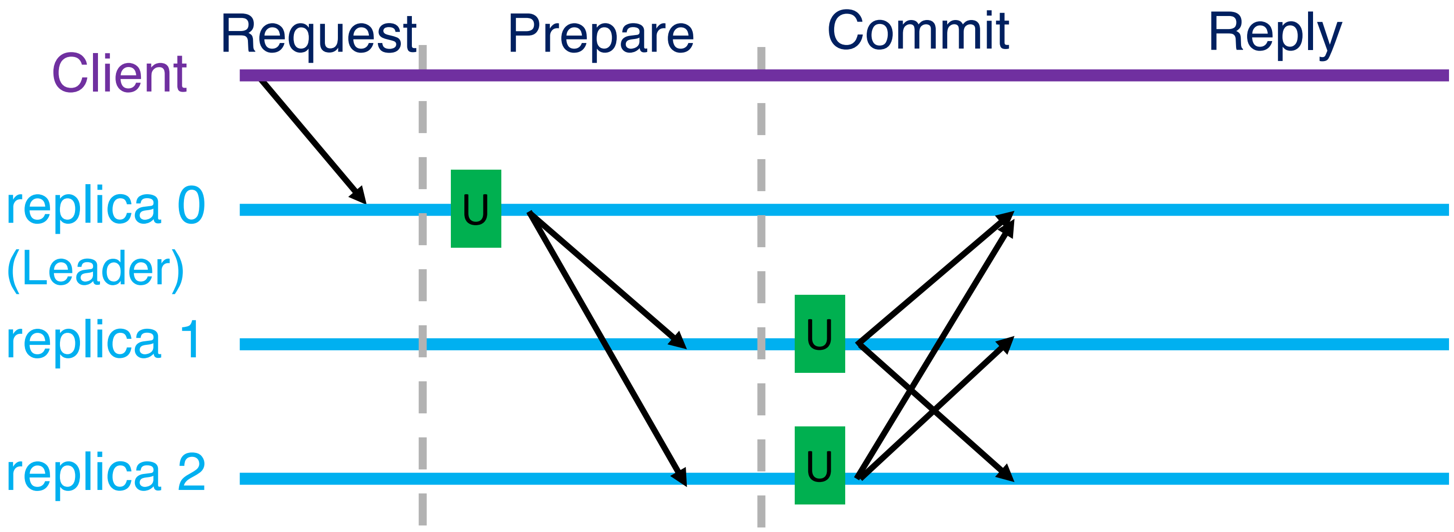
# MinBFT Agreement Protocol

Requires the same number of replicas, communication phases and message complexity as Paxos



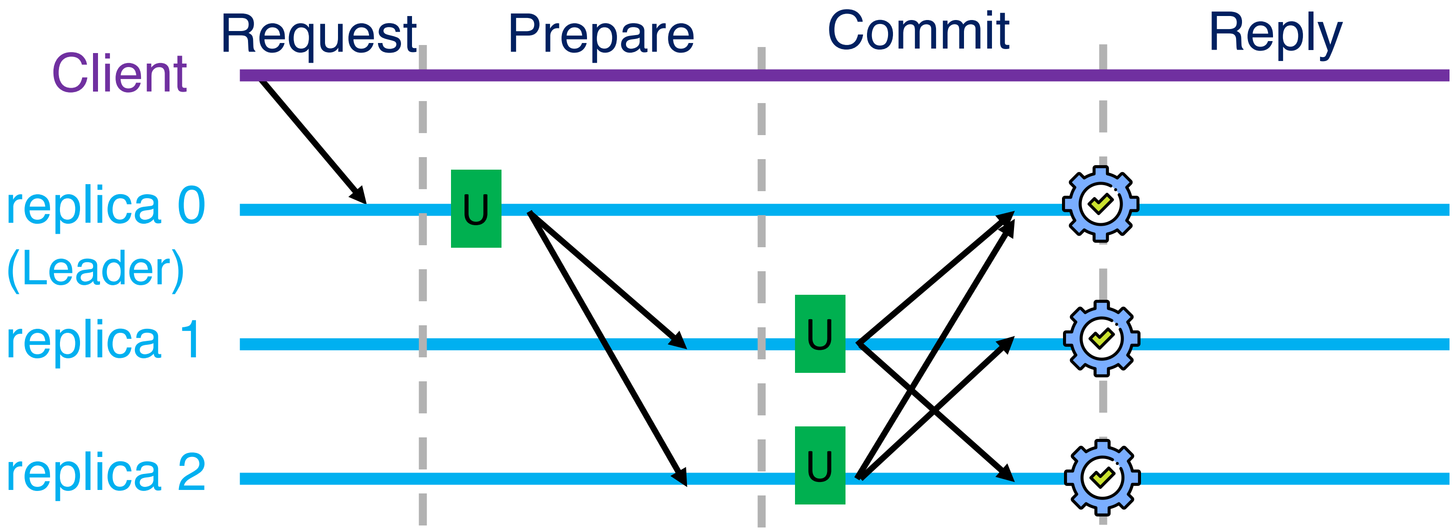
# MinBFT Agreement Protocol

Requires the same number of replicas, communication phases and message complexity as Paxos



# MinBFT Agreement Protocol

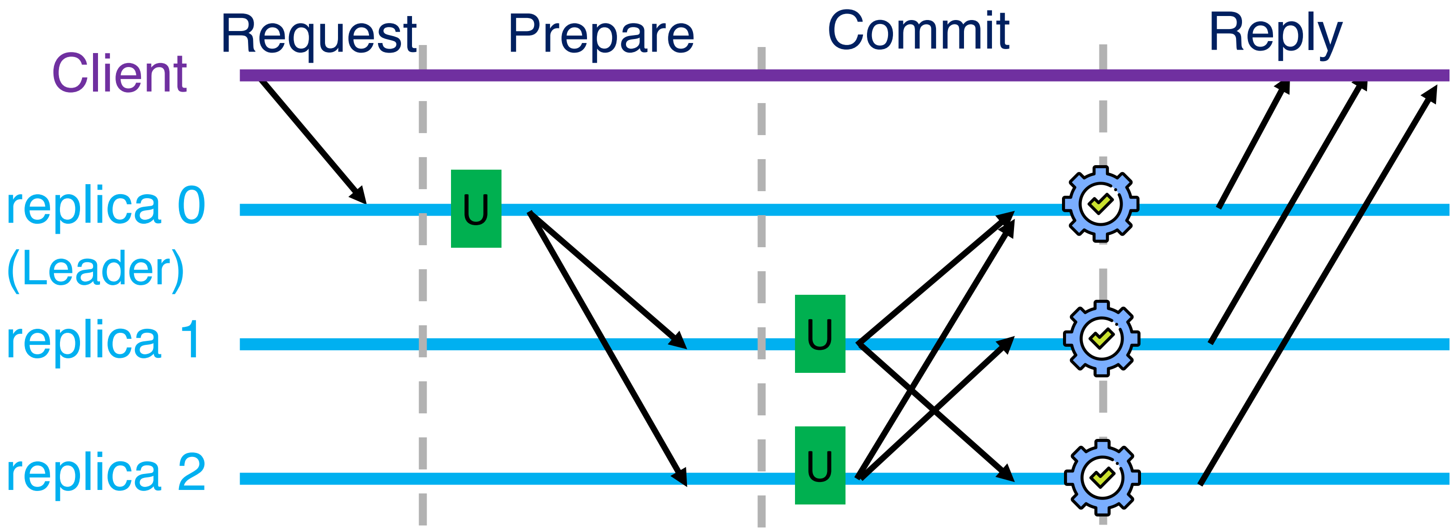
Requires the same number of replicas, communication phases and message complexity as Paxos



COMPUTER SCIENCE
UC SANTA BARBARA

# MinBFT Agreement Protocol

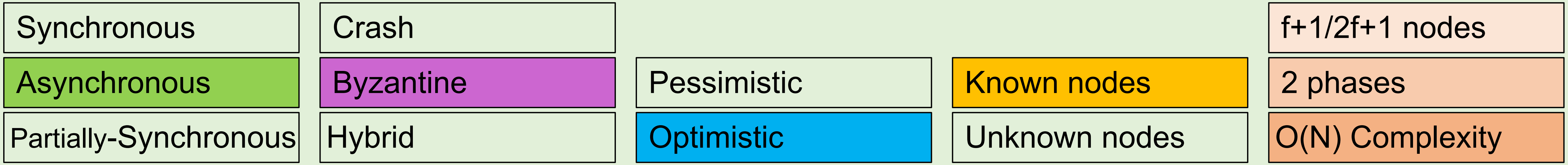
Requires the same number of replicas, communication phases and message complexity as Paxos



# CheapBFT

Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S. V., ... & Stengel, K. CheapBFT: resource-efficient byzantine fault tolerance. In EuroSys, 2012

**Trusted Hardware**
**Active/Passive Replication**



| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | f+1/2f+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 2 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O(N) Complexity |

# CheapBFT

Trusted Hardware (called Cash subsystem)
Assigns a unique counter value to each request
Creates Message Certificate and Checks Message Certificate
CASH system can fail only by crashing

Active Passive Replication
f replicas are passive and needed only when there is a failure

# CheapBFT Agreement Protocol

## 1 CheapTiny

- The default protocol, only f+1 replicas participate
- Only f+1 active replicas are selected.
- All the other replicas go in a passive mode

# CheapBFT Agreement Protocol

## 1 CheapTiny

- The default protocol, only f+1 replicas participate
- Only f+1 active replicas are selected.
- All the other replicas go in a passive mode

## 2 CheapSwitch

- Switches the protocol from cheapTiny to MinBFT if there is any failure

# CheapBFT Agreement Protocol

## 1 CheapTiny

- The default protocol, only f+1 replicas participate
- Only f+1 active replicas are selected.
- All the other replicas go in a passive mode

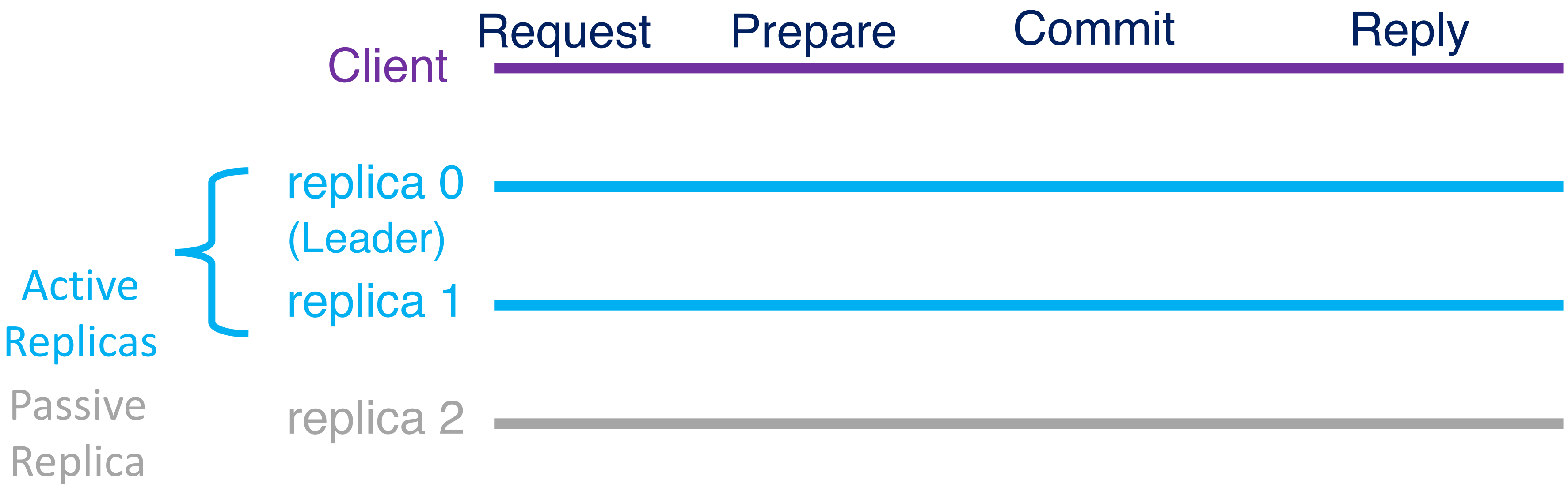
## 2 CheapSwitch

- Switches the protocol from cheapTiny to MinBFT if there is any failure

## 3 MinBFT

- Involve 2f+1 active replicas.
- Eventually, system again switches back to cheapTiny.

# CheapTiny Protocol



Request Prepare Commit Reply

Client

Active Replicas

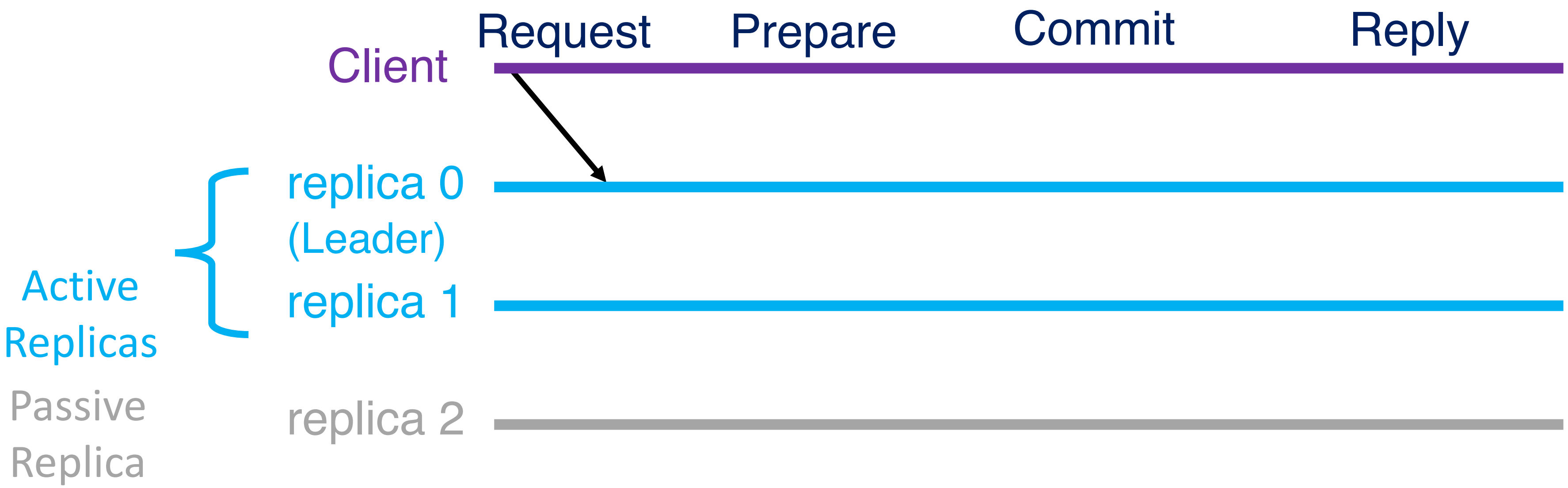
replica 0 (Leader)

replica 1

Passive Replica

replica 2

# CheapTiny Protocol



Request Prepare Commit Reply

Client

replica 0 (Leader)

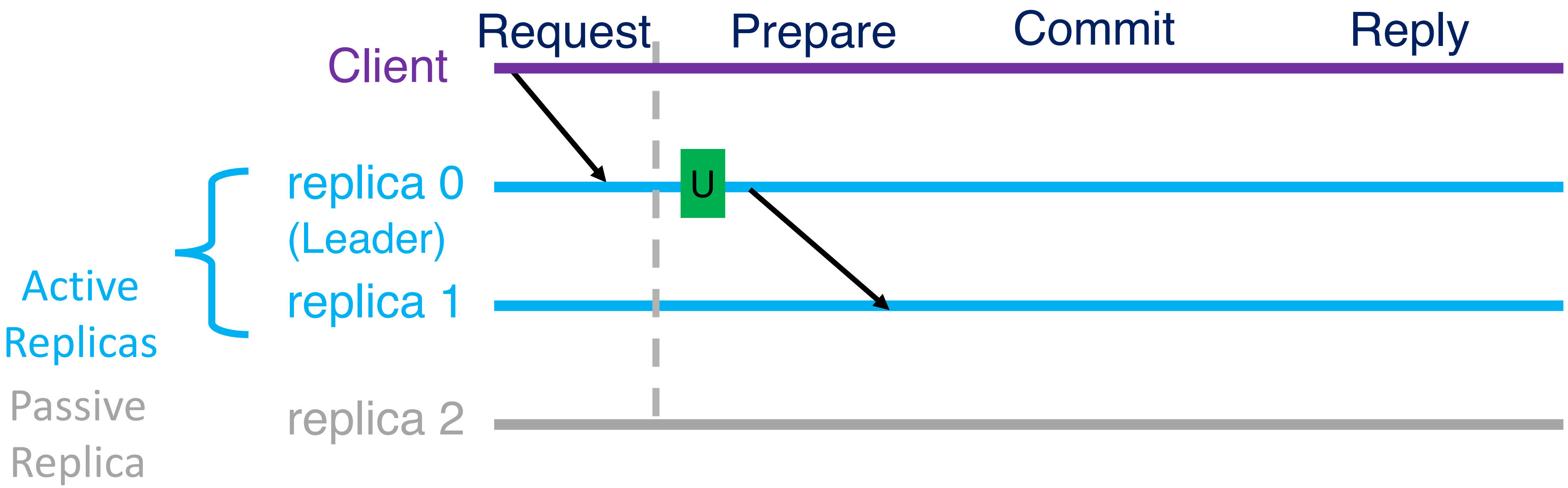
replica 1

replica 2

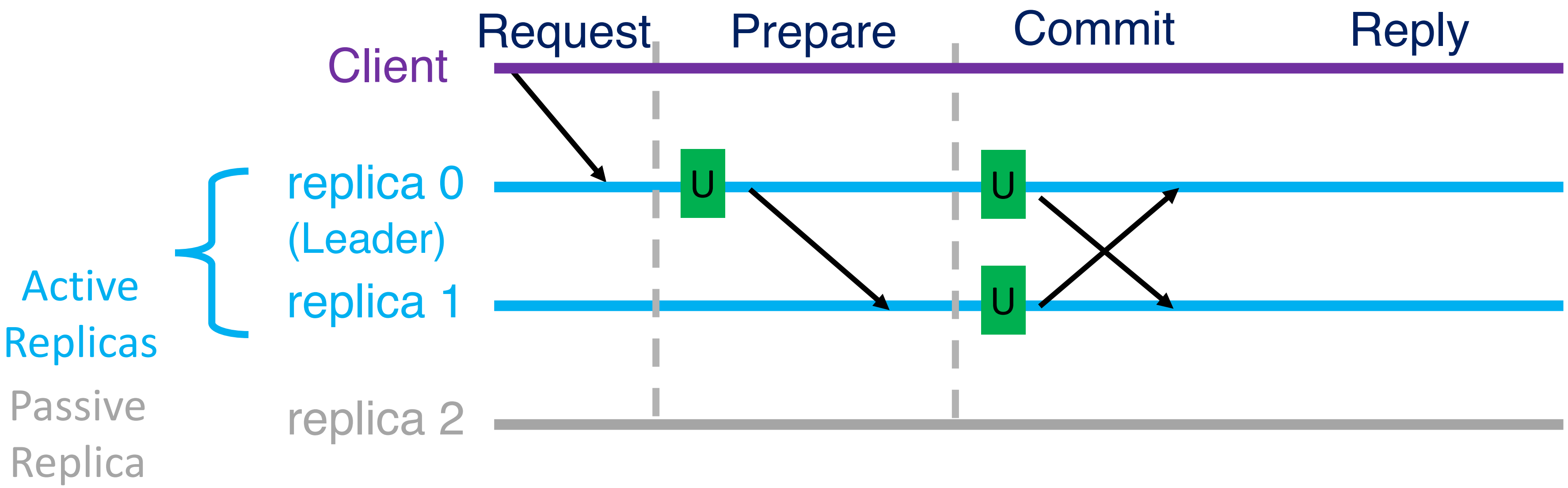
Active Replicas

Passive Replica

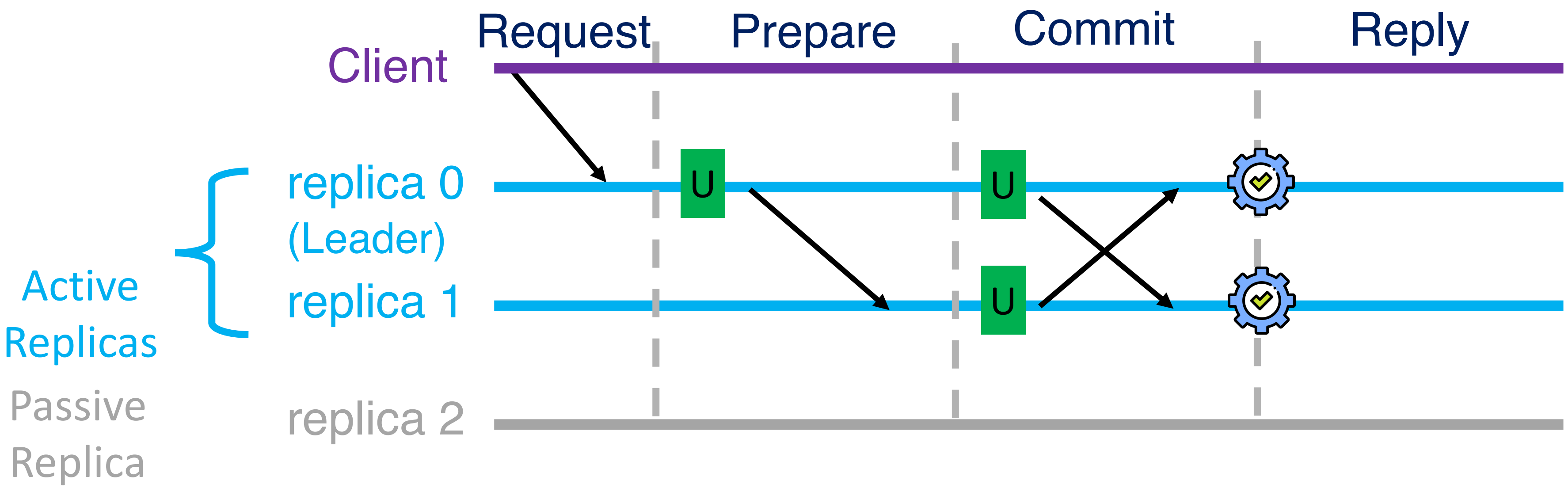
# CheapTiny Protocol



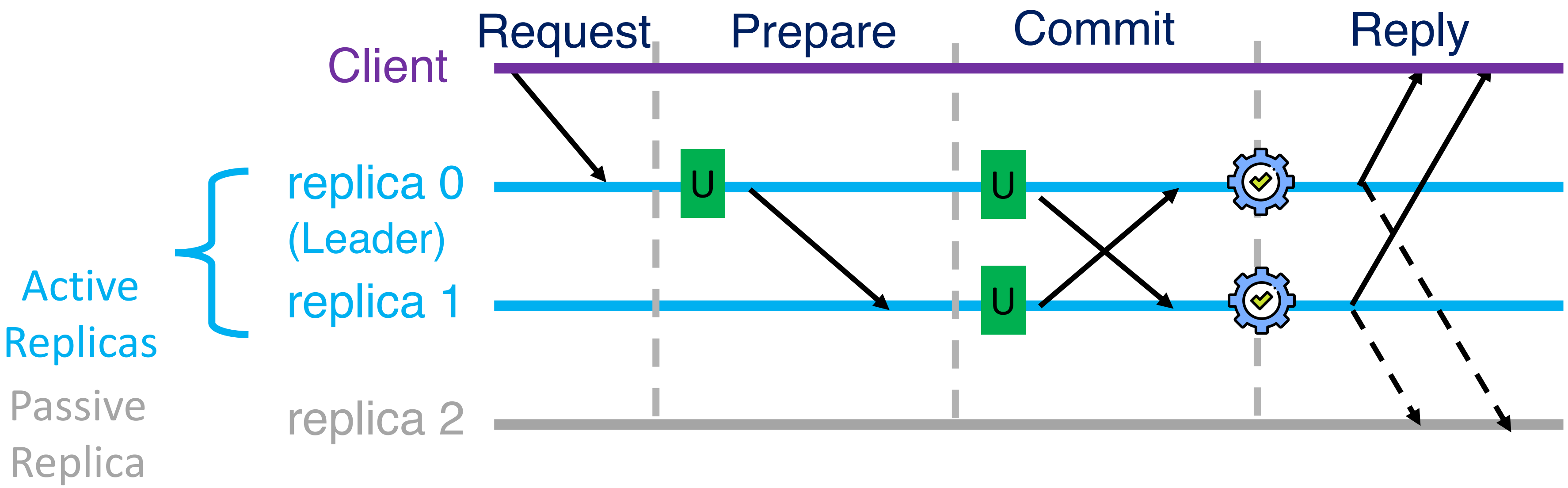
# CheapTiny Protocol



# CheapTiny Protocol



# CheapTiny Protocol



# CheapSwitch Protocol

Any node can request protocol switch by sending a PANIC message to all replicas

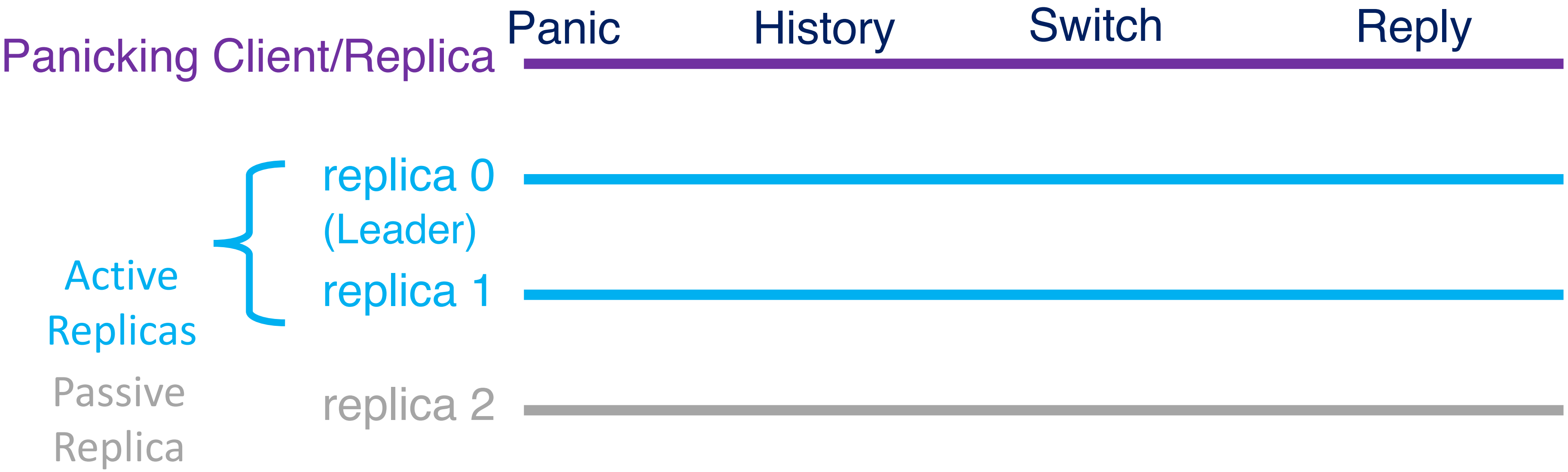
Replicas broadcast the message and wait for Abort History message from the new leader

New Leader creates and broadcasts an Abort History

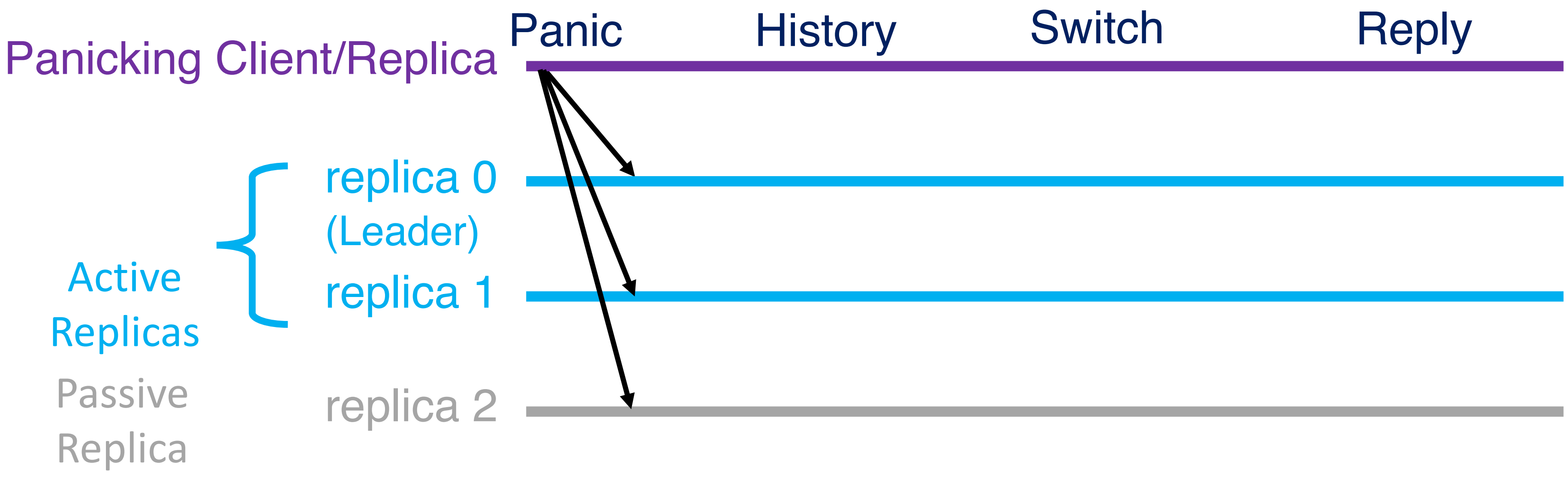
Other Replicas validate the abort history and send Switch messages to all other replicas

After receiving f matching switch messages, the history becomes stable.

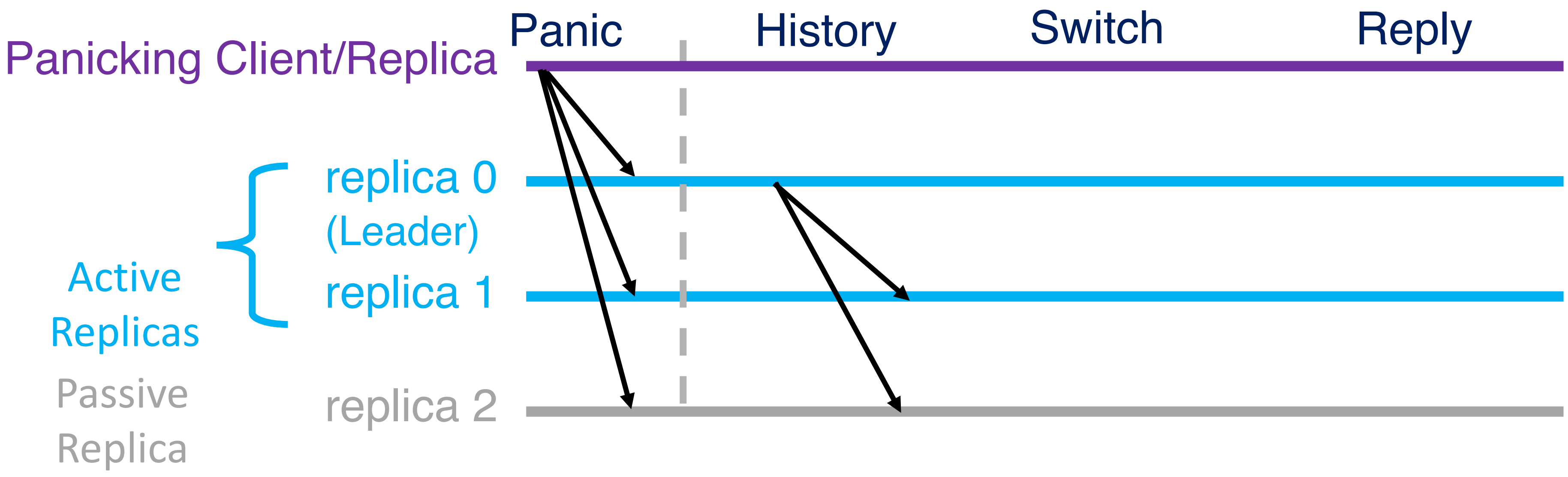
# CheapSwitch Protocol



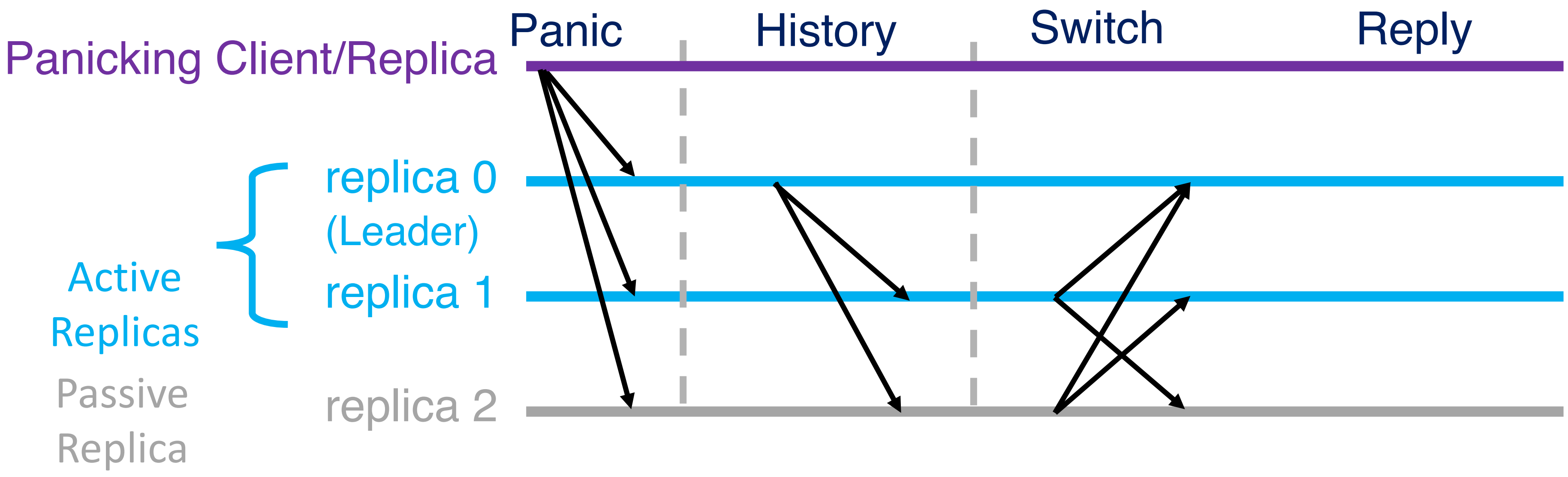
# CheapSwitch Protocol



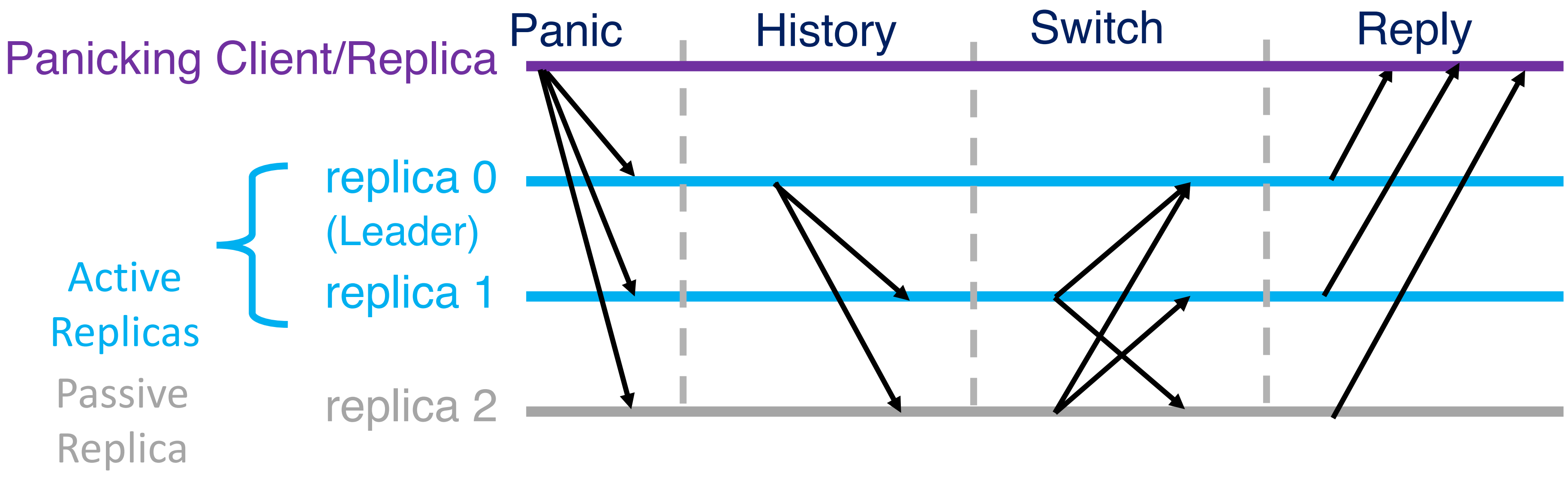
# CheapSwitch Protocol

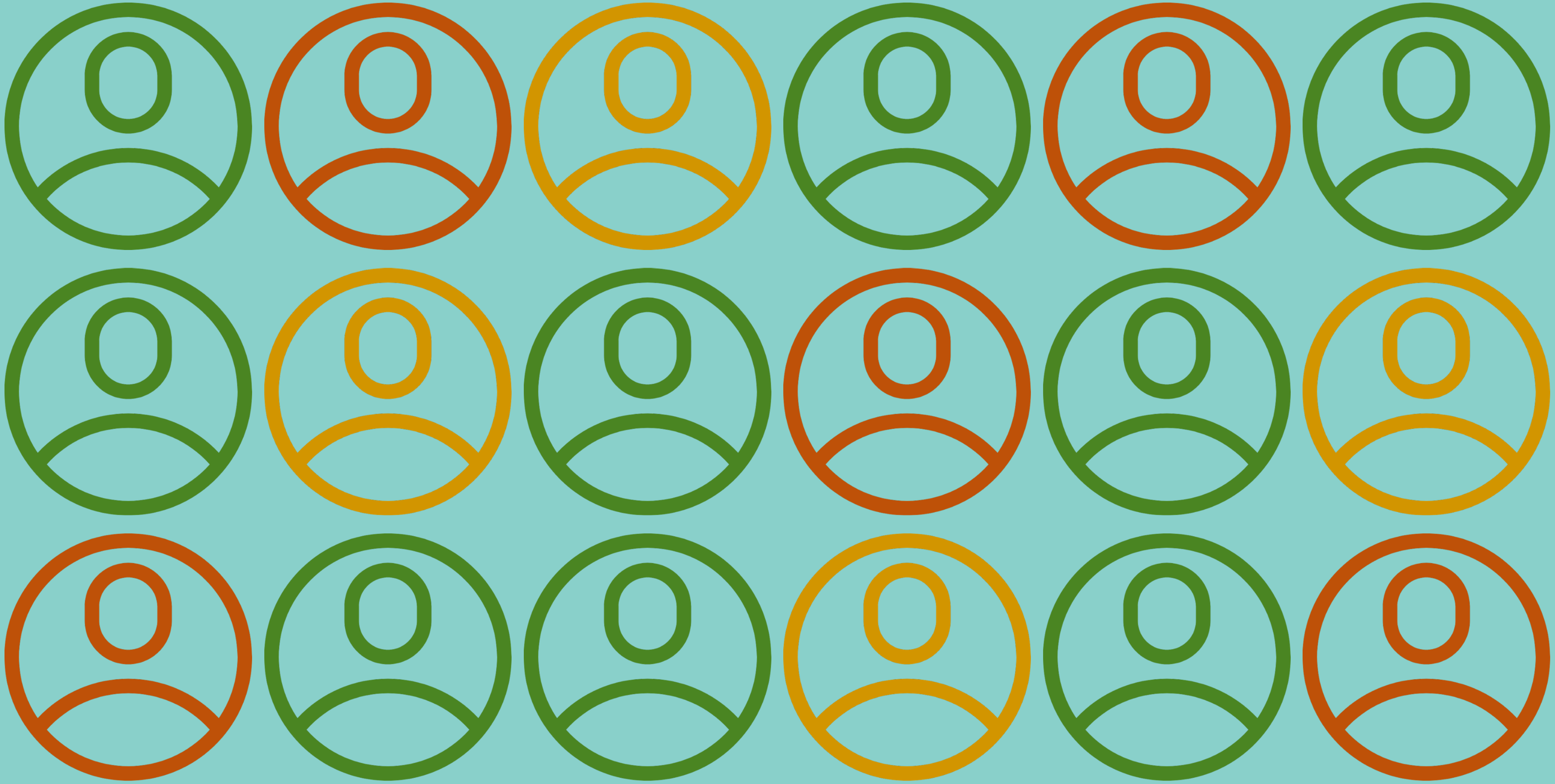


# CheapSwitch Protocol



# CheapSwitch Protocol





**What if a network includes both Crash-only and Byzantine nodes?**

# UpRight

[Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., & Riche, T. Upright cluster services. In *SOSP, 2009.*]

| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | 3m+2c+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 2 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O($N^2$) Complexity |

# UpRight Cluster Services

- Hybrid failure model
  - Tolerates both crash and malicious failure

# UpRight Cluster Services

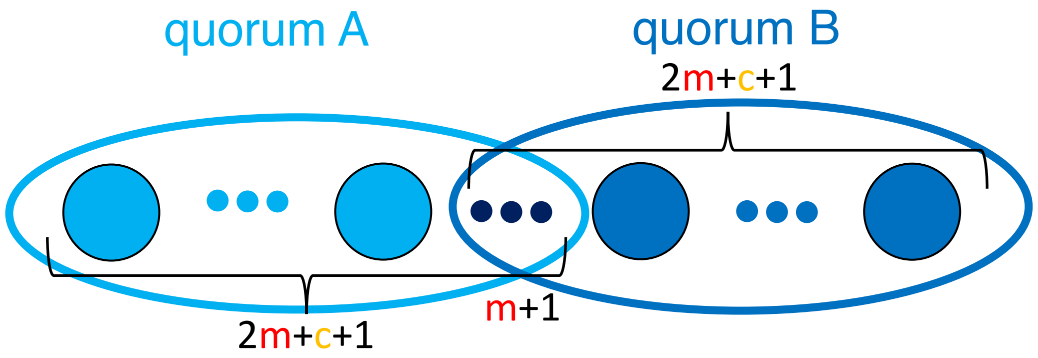
- Hybrid failure model
  - Tolerates both crash and malicious failure
- Request quorum
  - Avoid expensive corner cases with inconsistent client MACs
  - Separate the data path from the control path

# UpRight Cluster Services

- Hybrid failure model
  - Tolerates both crash and malicious failure
- Request quorum
  - Avoid expensive corner cases with inconsistent client MACs
  - Separate the data path from the control path
- Agreement protocol is a combination of
  - Zyzzyva's speculative execution
  - Aardvark’s techniques for robustness
  - Yin et al.'s techniques for separating agreement and execution
    - While agreement requires 3f+1 nodes, execution needs 2f+1 nodes

# UpRight Failure Model

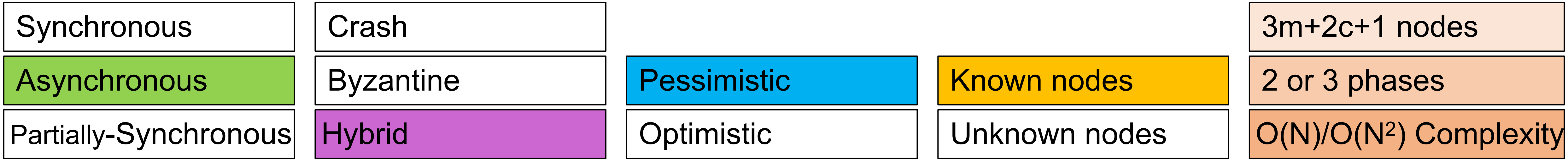
- Tolerate at most m malicious and at most c crash faults
  - Quorum: 2m + c + 1
  - Intersection: m + 1
  - Network: 3m + 2c + 1



# SeeMoRe

SeeMoRe is derived from Seemorq, a benevolent, mythical bird in Persian mythology which appears as a peacock with the head of a dog and the claws of a lion.

Amiri, M. J., Maiyya, S., Agrawal, D., & Abbadi, A. E. Seemore: A fault-tolerant protocol for hybrid cloud environments. *ICDE, 2020*



| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | 3m+2c+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 2 or 3 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | $O(N)/O(N^2)$ Complexity |

# Hybrid Cloud Environment

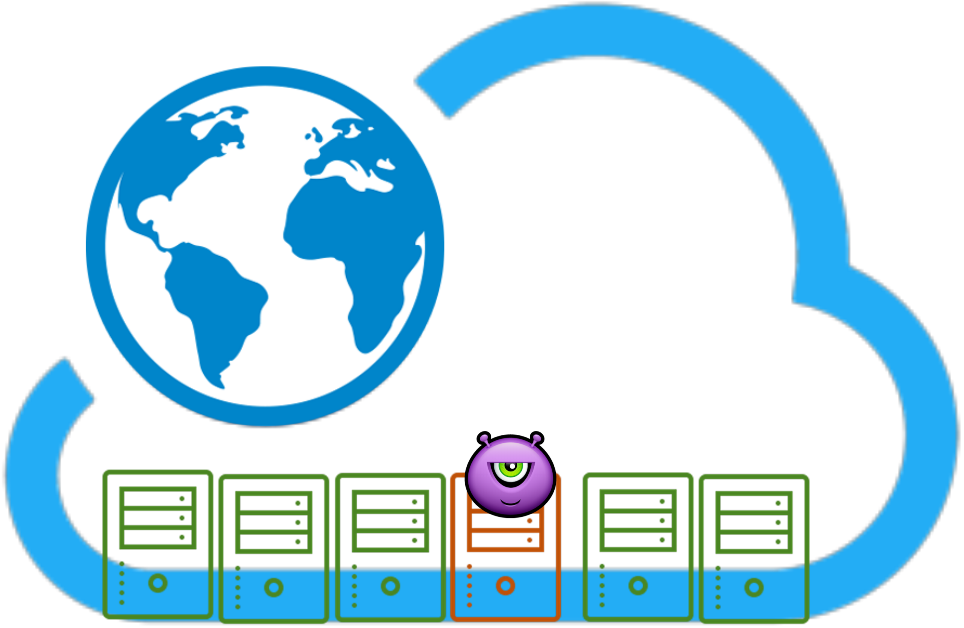
Lack of resources to guarantee fault tolerance

Nodes in the private cloud are trusted (crash-only)

# Hybrid Cloud Environment

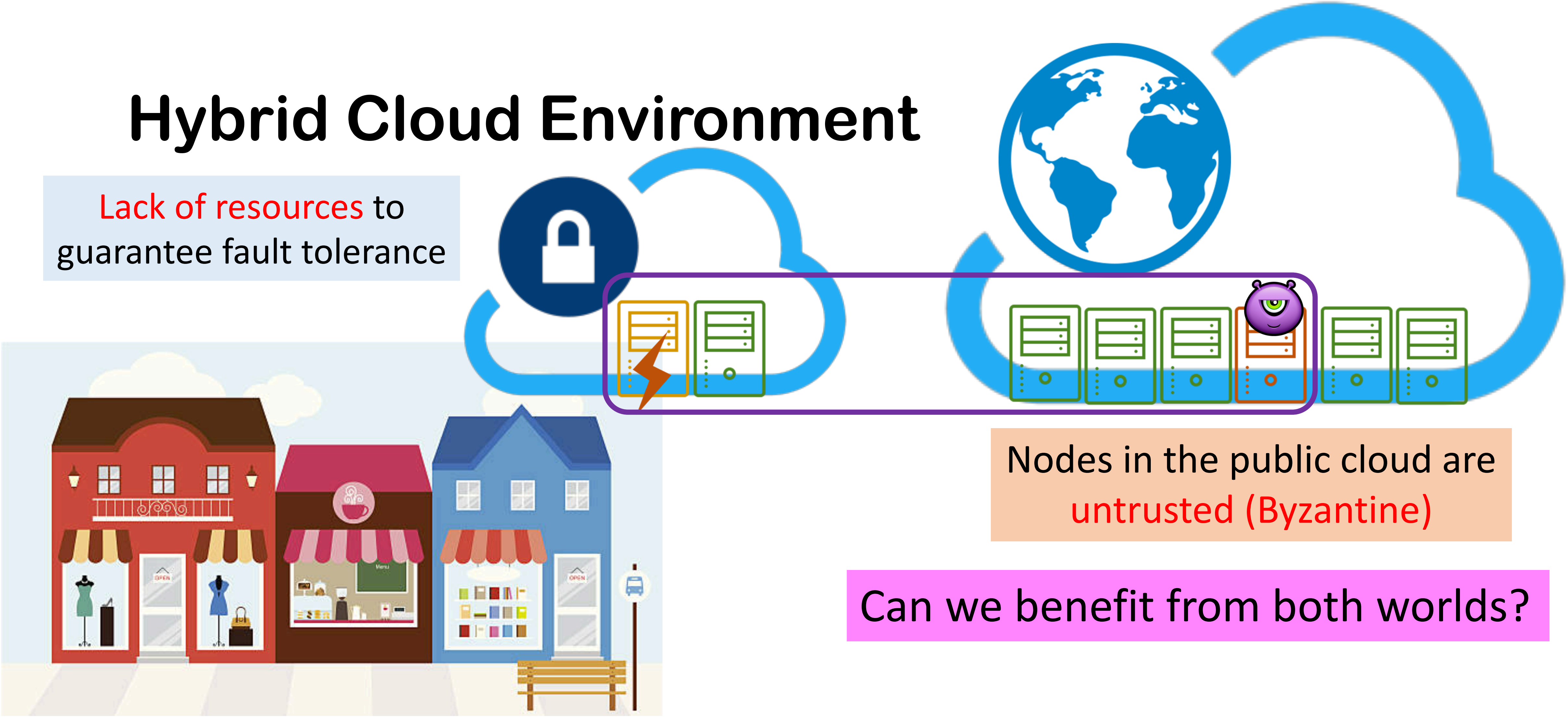
Lack of resources to guarantee fault tolerance

Nodes in the private cloud are trusted (crash-only)



Nodes in the public cloud are untrusted (Byzantine)

# Hybrid Cloud Environment



Lack of resources to guarantee fault tolerance

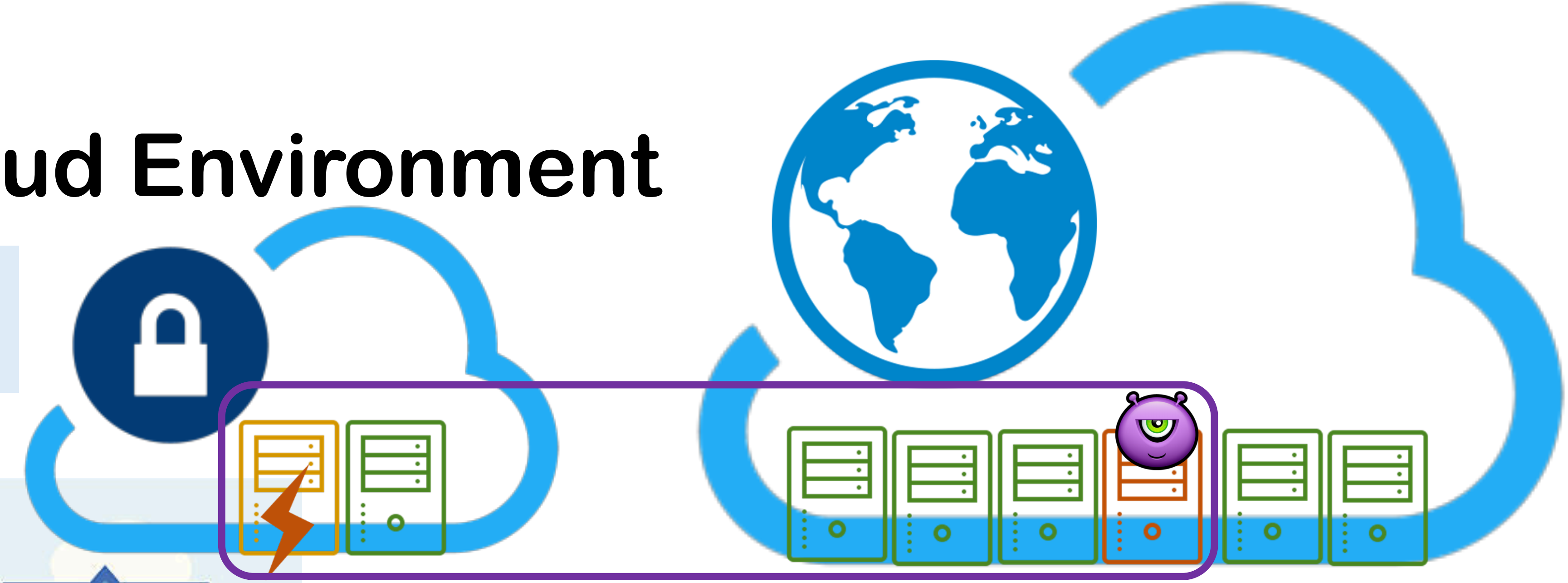
Nodes in the public cloud are untrusted (Byzantine)

Can we benefit from both worlds?

Nodes in the private cloud are trusted (crash-only)

# Hybrid Cloud Environment

Lack of resources to guarantee fault tolerance



Nodes in the public cloud are untrusted (Byzantine)

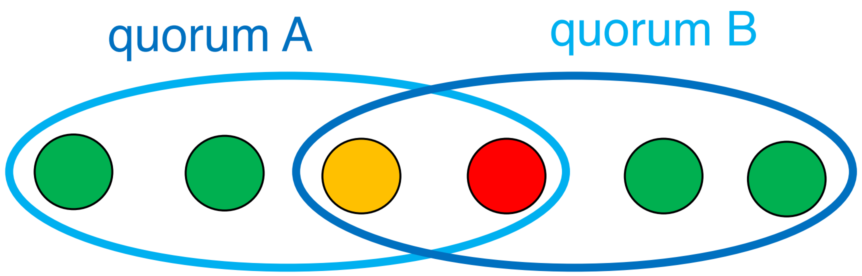
Can we benefit from both worlds?

Nodes in the private cloud are trusted (crash-only)

**SeeMoRe**

# Mode 1: Trusted Primary, Centralized Coordination

- The primary is in the private cloud (Trusted)
- Backups are in both private and public cloud

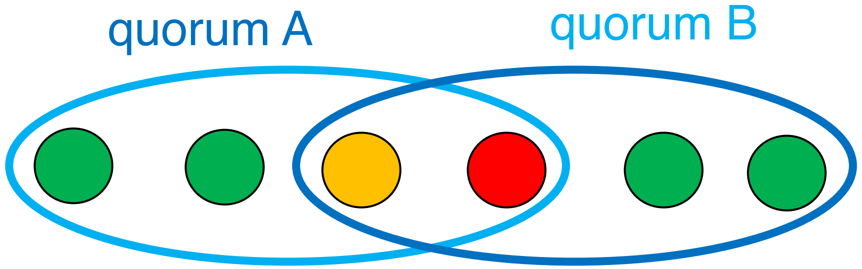


Network: 3m+2c+1
Quorum: 2m+c+1
Intersection: m+1

At most **m Malicious**
and
At most **c crash faults**

# Mode 1: Trusted Primary, Centralized Coordination

- The primary is in the private cloud (Trusted)
- Backups are in both private and public cloud



Network: 3m+2c+1
Quorum: 2m+c+1
Intersection: m+1

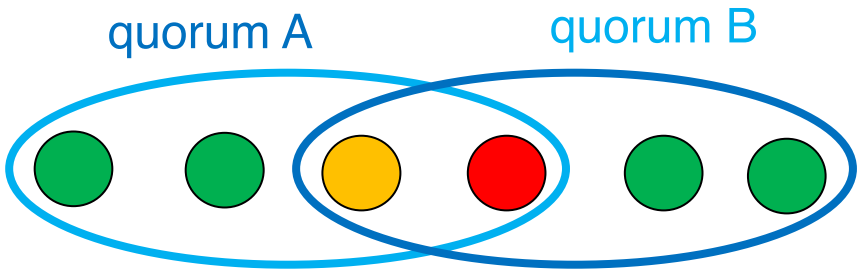
At most **m Malicious**
and
At most **c crash faults**

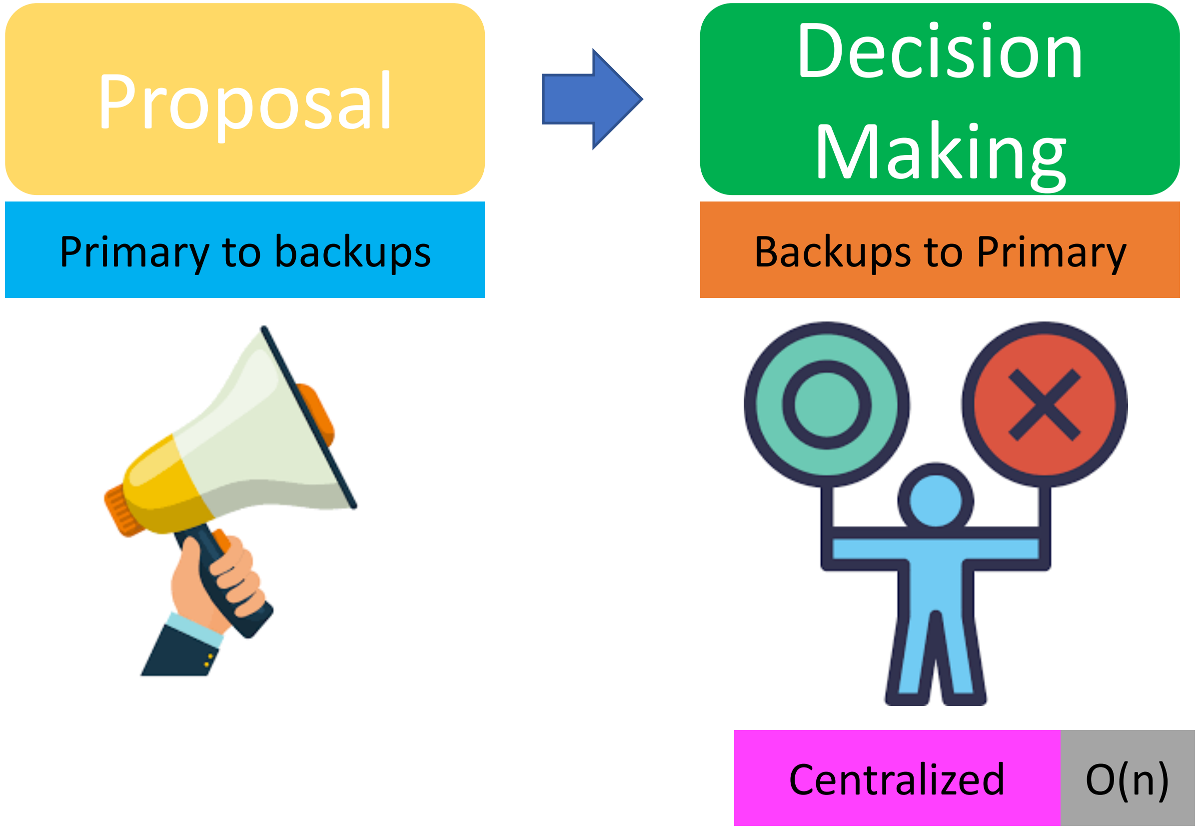
**Proposal**

Primary to backups

# Mode 1: Trusted Primary, Centralized Coordination

- The primary is in the private cloud (Trusted)
- Backups are in both private and public cloud





Network: 3m+2c+1
Quorum: 2m+c+1
Intersection: m+1

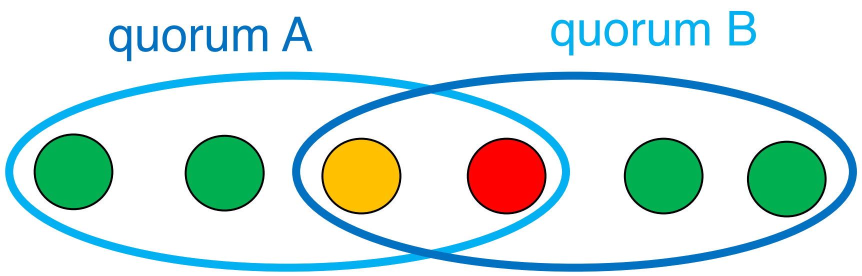
At most **m Malicious**

and

At most **c crash faults**

# Mode 1: Trusted Primary, Centralized Coordination

- The primary is in the private cloud (Trusted)
- Backups are in both private and public cloud



Proposal → Decision Making

Primary to backups

Backups to Primary

Centralized | O(n)

Network: 3m+2c+1
Quorum: 2m+c+1
Intersection: m+1

At most **m Malicious**
and
At most **c crash faults**

**Phases: Two**
**Messages: O(n)**
**Quorum: 2c+m+1**

# Mode 2: Trusted Primary, Decentralized Coordination

- The primary is still in the private cloud (Trusted)
- The private cloud is not involved in the second phase
- Proxy nodes: *3m+1* nodes from the public cloud

Goal:
Reduce the load on the private cloud

# Mode 2: Trusted Primary, Decentralized Coordination

- The primary is still in the private cloud (Trusted)
- The private cloud is not involved in the second phase
- Proxy nodes: *3m+1* nodes from the public cloud

Proposal

Primary to backups

Goal:
Reduce the load on the private cloud

# Mode 2: Trusted Primary, Decentralized Coordination

- The primary is still in the private cloud (Trusted)
- The private cloud is not involved in the second phase
- Proxy nodes: *3m+1* nodes from the public cloud

Goal:
Reduce the load on the private cloud

Proposal → Decision Making

Primary to backups

Proxies to Proxies

Decentralized $O(n^2)$

# Mode 2: Trusted Primary, Decentralized Coordination

- The primary is still in the private cloud (Trusted)
- The private cloud is not involved in the second phase
- Proxy nodes: *3m+1* nodes from the public cloud

Proposal → Decision Making

Primary to backups

Proxies to Proxies

Goal:
Reduce the load on the private cloud

Decentralized $O(n^2)$

**Phases: Two**
**Messages: $O(n^2)$**
**Quorum: 2m+1**

# Mode 3: Untrusted Primary, Decentralized Coordination

- The primary is in the public cloud (Untrusted)
- The private cloud is not involved in any phases
- Proxy nodes: *3m+1* nodes from the public cloud

Goal:
Reduce the load on the private cloud
Reduce latency when there is a large network distance between clouds

# Mode 3: Untrusted Primary, Decentralized Coordination

- The primary is in the public cloud (Untrusted)
- The private cloud is not involved in any phases
- Proxy nodes: *3m+1* nodes from the public cloud

Goal:
Reduce the load on the private cloud
Reduce latency when there is a large network distance between clouds

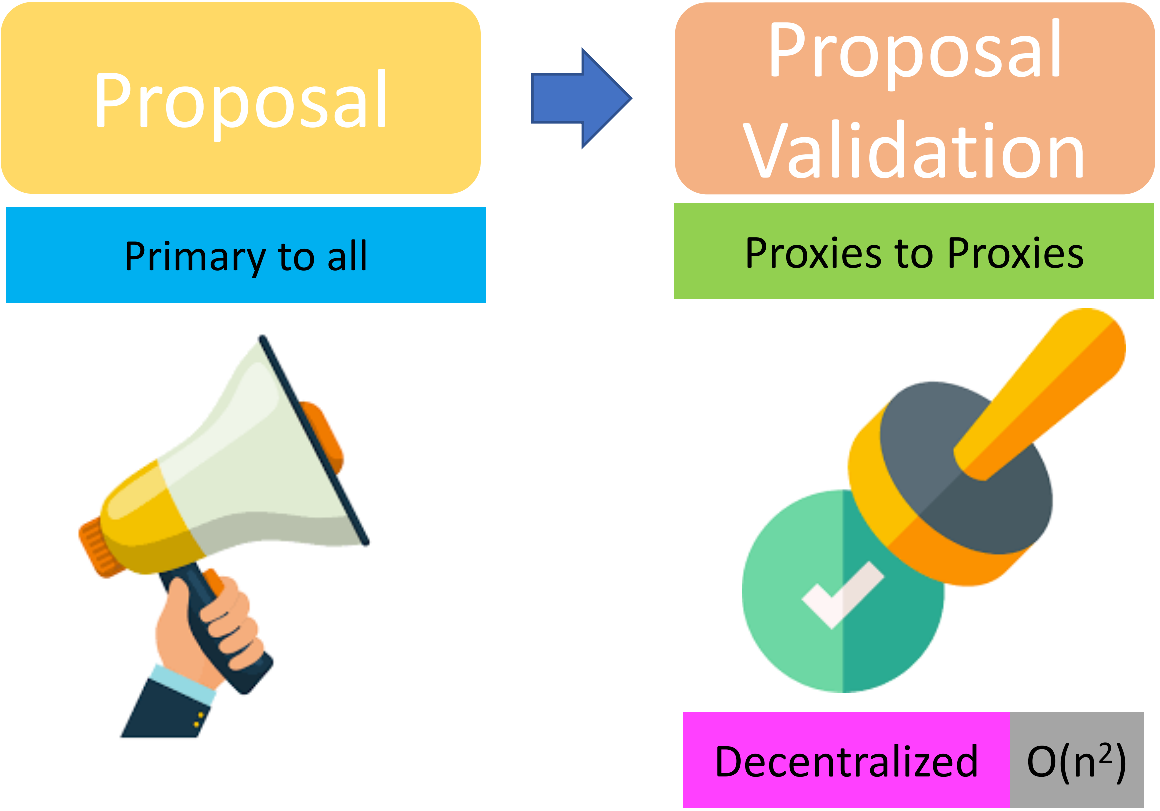
Proposal

Primary to all

# Mode 3: Untrusted Primary, Decentralized Coordination

- The primary is in the public cloud (Untrusted)
- The private cloud is not involved in any phases
- Proxy nodes: *3m+1* nodes from the public cloud

Goal:
Reduce the load on the private cloud
Reduce latency when there is a large network distance between clouds



# Mode 3: Untrusted Primary, Decentralized Coordination

- The primary is in the public cloud (Untrusted)
- The private cloud is not involved in any phases
- Proxy nodes: *3m+1* nodes from the public cloud

Goal:
Reduce the load on the private cloud
Reduce latency when there is a large network distance between clouds

**Proposal** → **Proposal Validation** → **Decision Making**

Proposal

Primary to all

Proposal Validation

Proxies to Proxies

| Decentralized | $O(n^2)$ |
|---|---|

Decision Making

Proxies to Proxies

| Decentralized | $O(n^2)$ |
|---|---|

# Mode 3: Untrusted Primary, Decentralized Coordination

- The primary is in the public cloud (Untrusted)
- The private cloud is not involved in any phases
- Proxy nodes: *3m+1* nodes from the public cloud

Goal:
Reduce the load on the private cloud
Reduce latency when there is a large network distance between clouds

Proposal → Proposal Validation → Decision Making

Proposal: Primary to all

Proposal Validation: Proxies to Proxies

Decentralized | $O(n^2)$

Decision Making: Proxies to Proxies

Decentralized | $O(n^2)$

**Phases: Three**
**Messages: $O(n^2)$**
**Quorum: 2m+1**

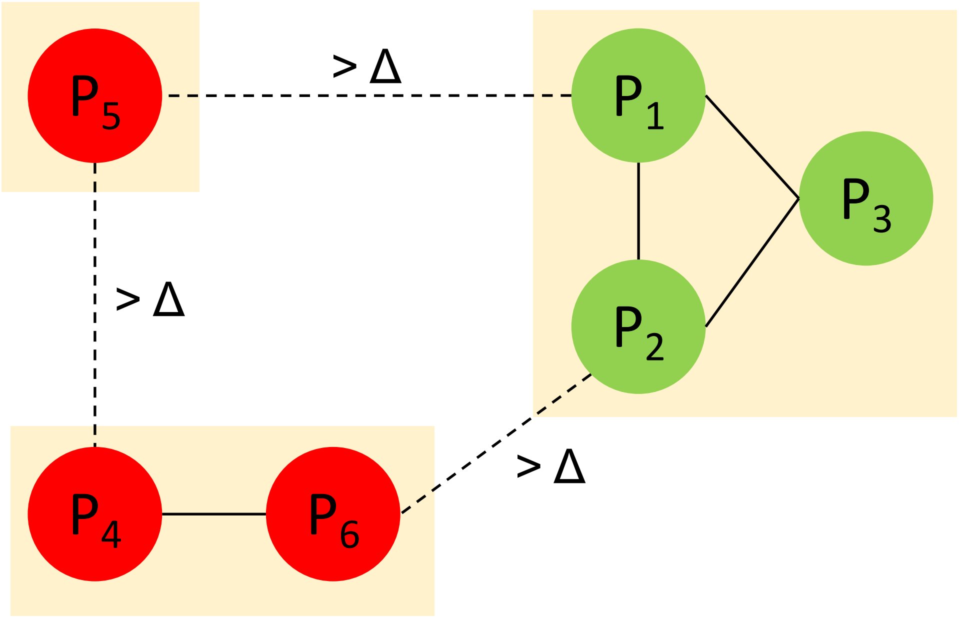
# XFT

Liu, S., Viotti, P., Cachin, C., Quéma, V., & Vukolić, M. XFT: Practical fault tolerance beyond crashes. In OSDI, 2016

| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | f+1/2f+1 nodes |
| Asynchronous | Byzantine | Pessimistic | Known nodes | 2 phases |
| Partially-Synchronous | Hybrid | Optimistic | Unknown nodes | O($N^2$) Complexity |

# Partially Synchronous

- Replica p is partitioned if p is not in the largest subset of replicas, in which every pair of replicas can communicate among each other within delay Δ.
- replica p is synchronous if p is not partitioned



# Failures and Anarchy

# Failures and Anarchy

- XFT considers three types of failures:
  - c: Number of crash failure
  - m: Number of non-crash (Byzantine) failure
  - p: Number of correct, but partitioned replicas

# Failures and Anarchy

- XFT considers three types of failures:
  - c: Number of crash failure
  - m: Number of non-crash (Byzantine) failure
  - p: Number of correct, but partitioned replicas

- Anarchy: The system is in anarchy at a given moment s iff
  - $m(s) > 0$
  - $f = c(s) + m(s) + p(s) > \left\lfloor \frac{n-1}{2} \right\rfloor$

# Failures and Anarchy

- XFT considers three types of failures:
  - c: Number of crash failure
  - m: Number of non-crash (Byzantine) failure
  - p: Number of correct, but partitioned replicas

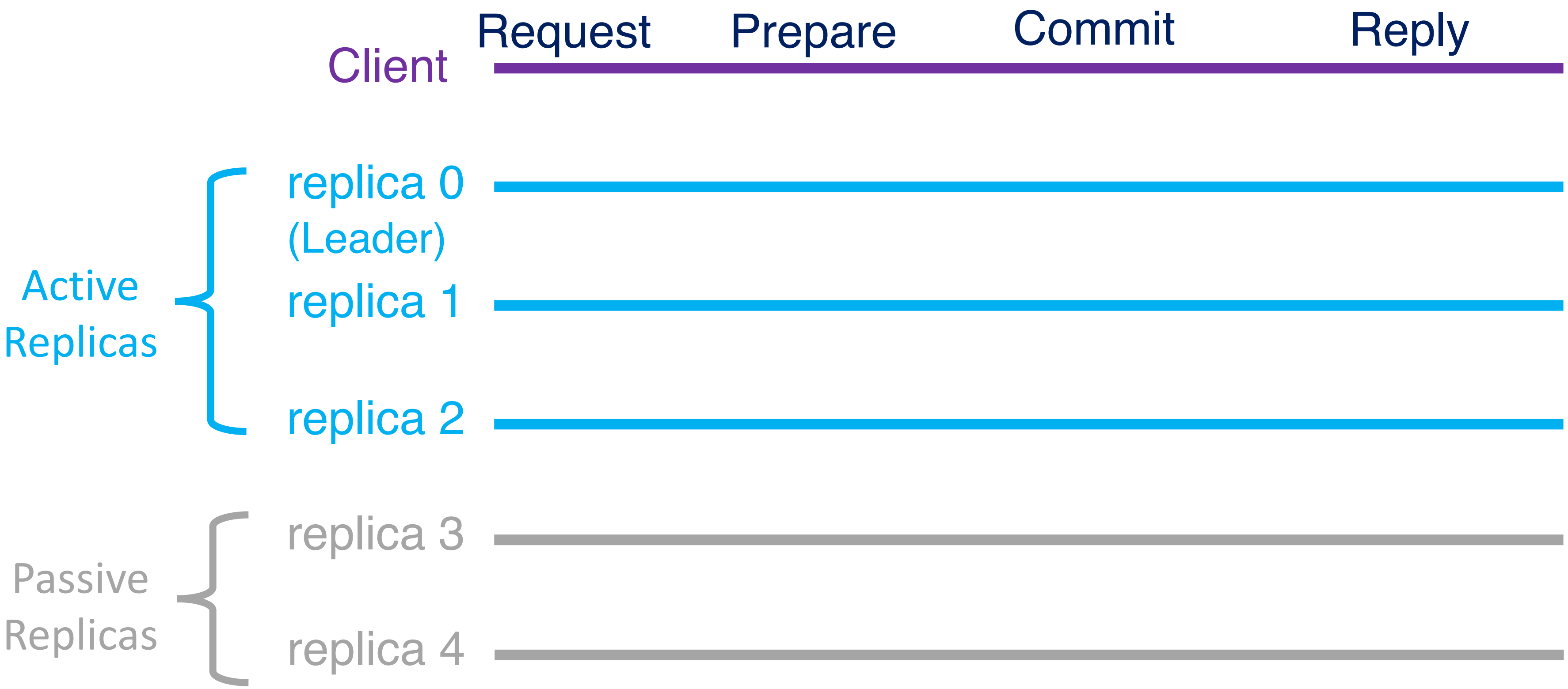
- Anarchy: The system is in anarchy at a given moment s iff
  - $m(s) > 0$
  - $f = c(s) + m(s) + p(s) > \left\lfloor \frac{n-1}{2} \right\rfloor$

XFT satisfies safety in executions in which the system is never in anarchy

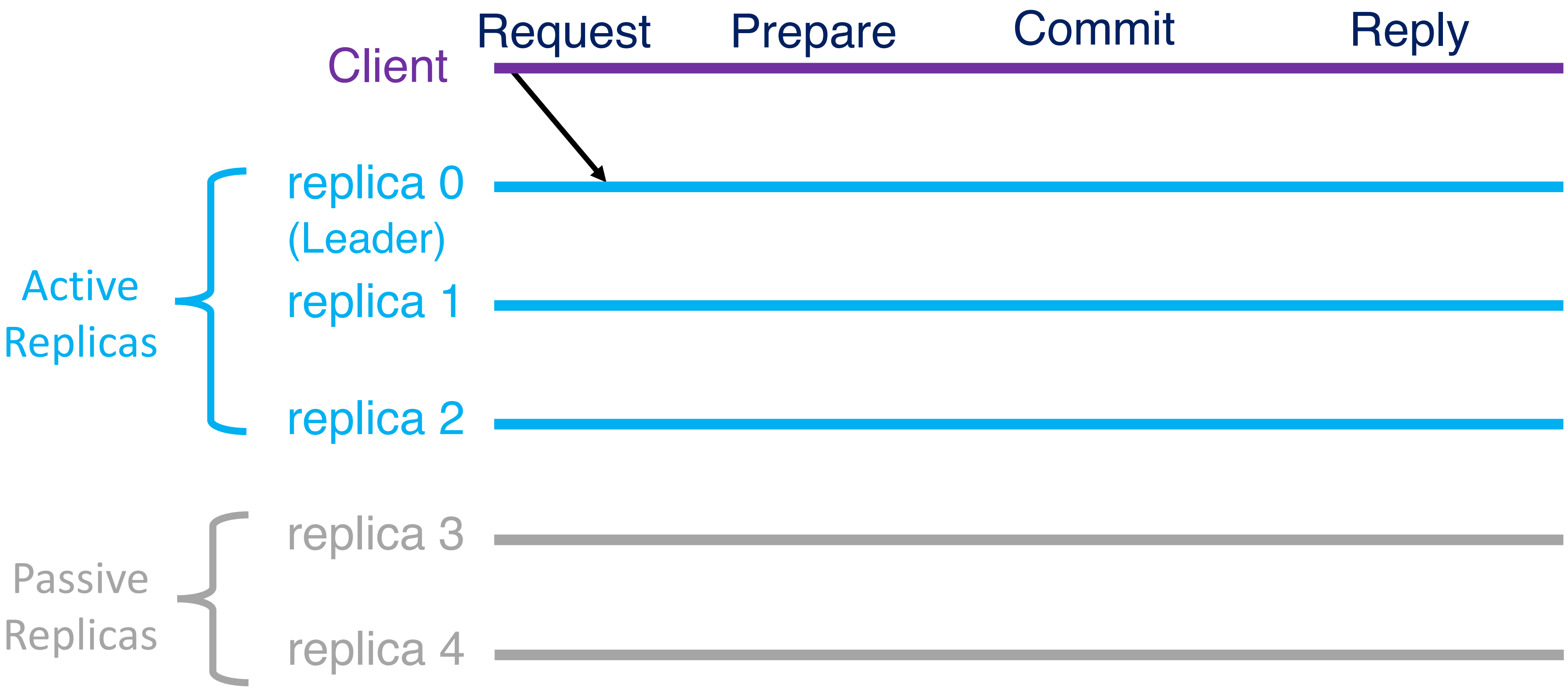
# XFT Agreement Protocol (XPaxos)

- Network includes 2f + 1 replicas where f is network + machine faults
- Uses the active/passive replication technique
- Optimistically replicates requests on only f+1 replicas, called a synchronous group
- A view is changed when there is a failure within the synchronous group.
- The view change reconfigures the entire synchronous group

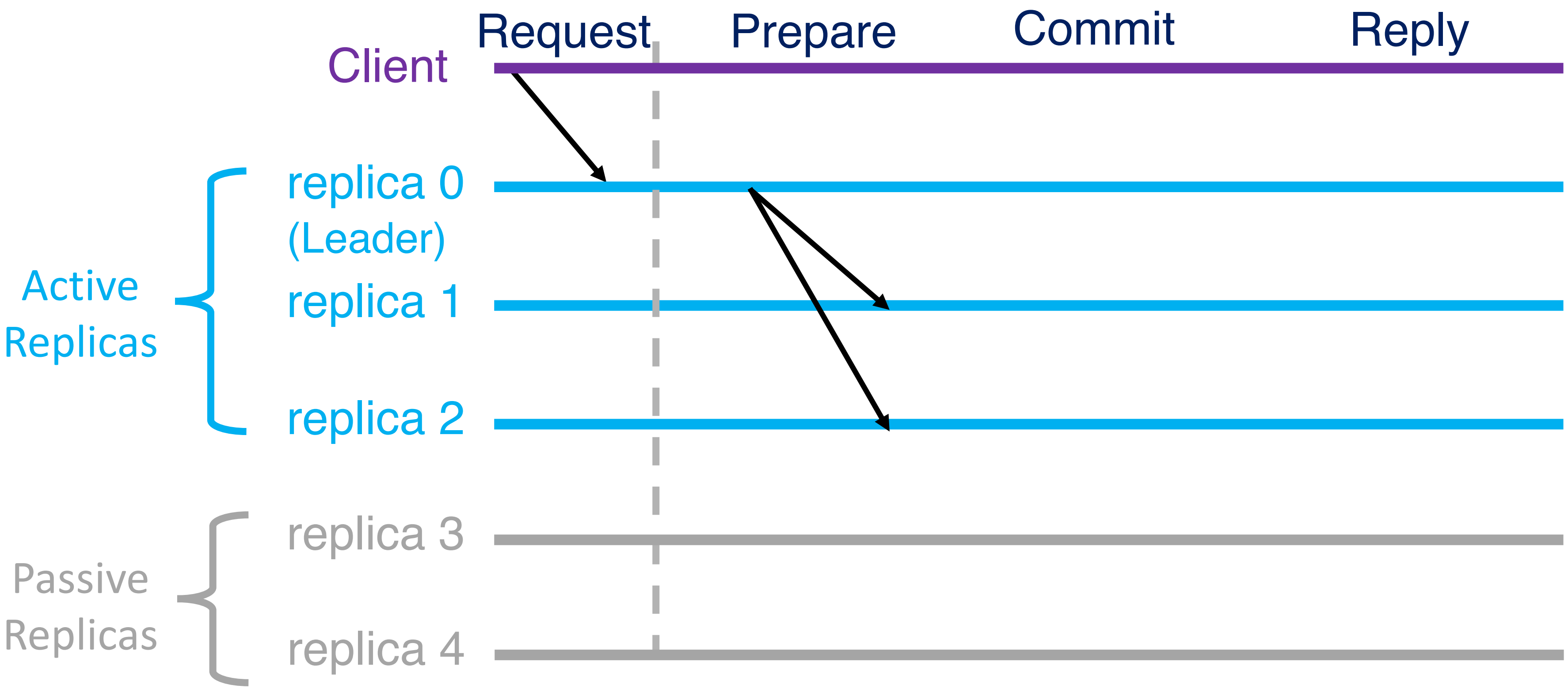
# XFT Common Case Protocol



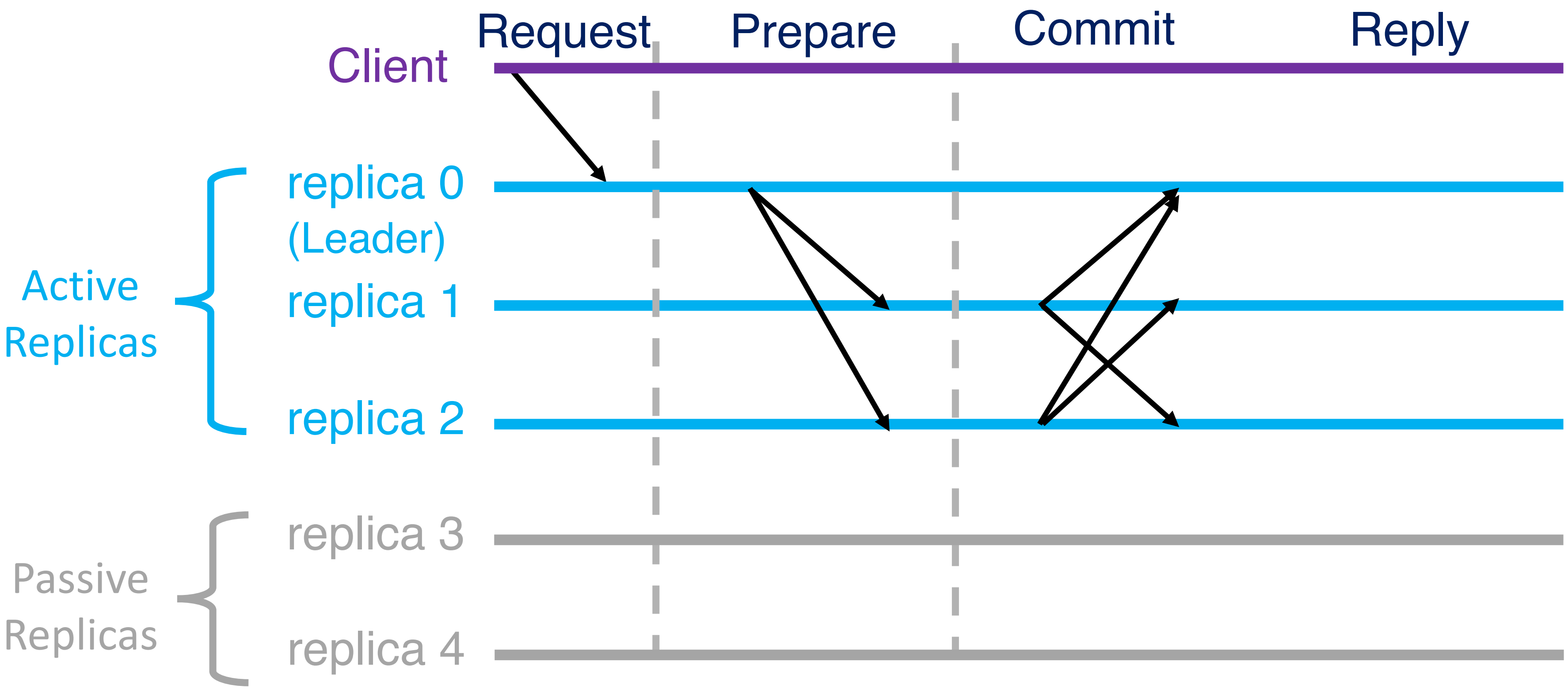
# XFT Common Case Protocol



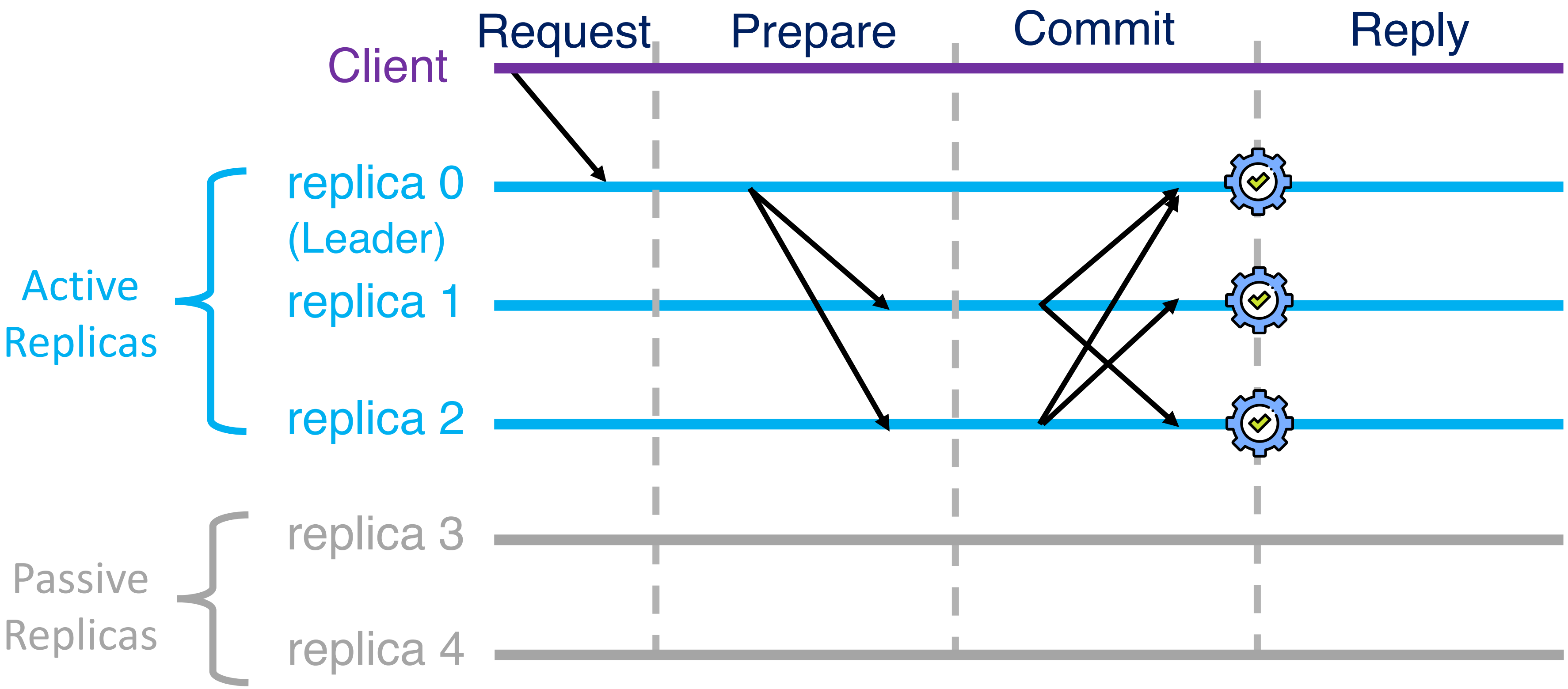
# XFT Common Case Protocol



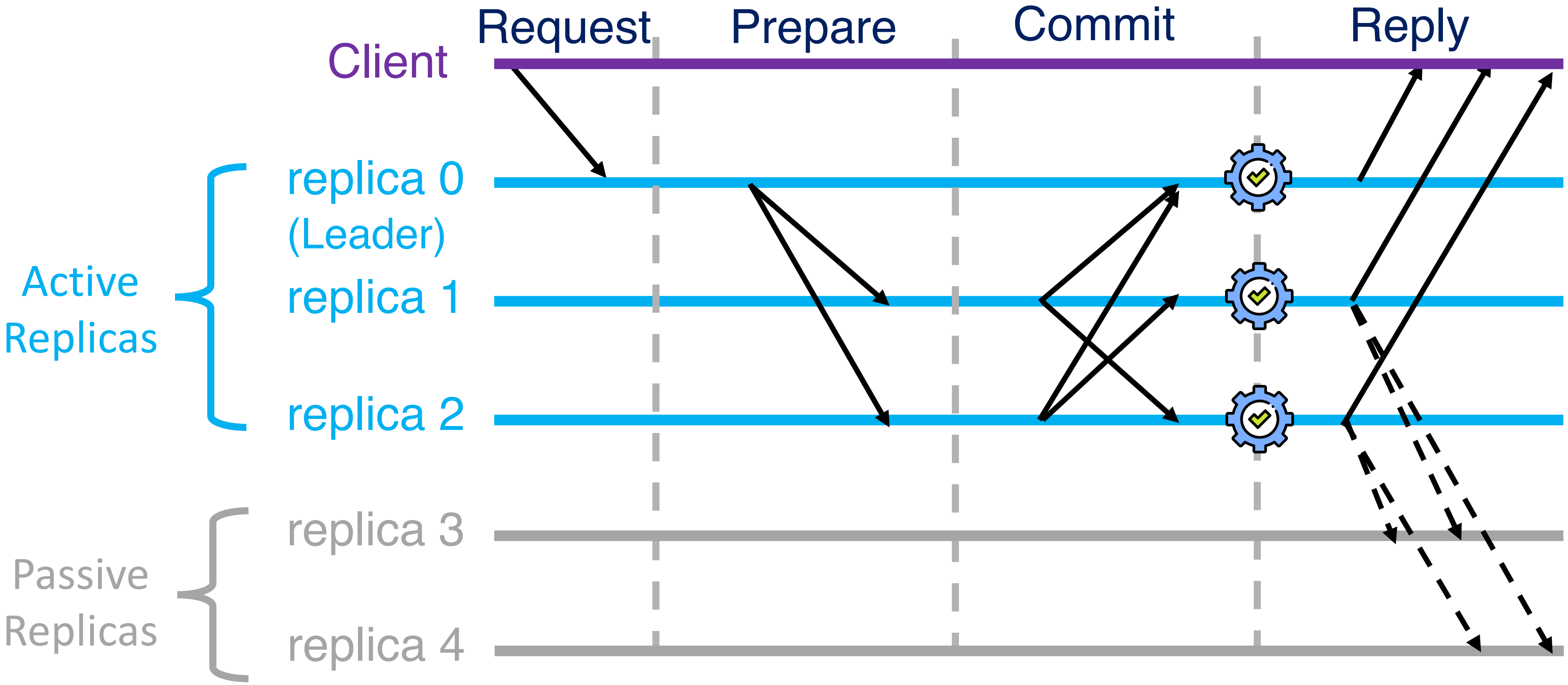
# XFT Common Case Protocol



# XFT Common Case Protocol



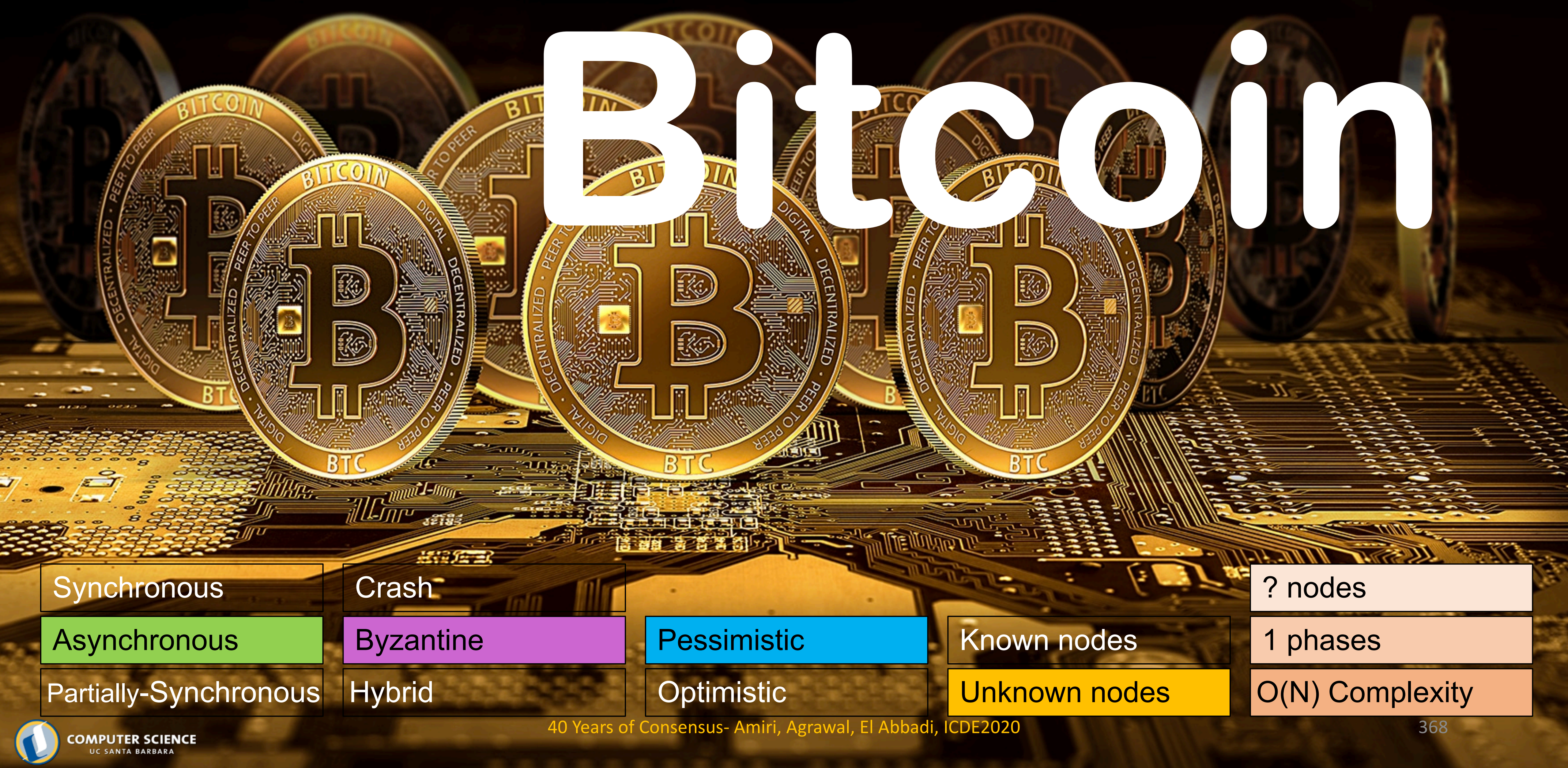
# XFT Common Case Protocol



# What if the participants are unknown?!

[Nakamoto, Satoshi. Bitcoin: A peer-to-peer electronic cash system, 2008]

# Bitcoin



| | | | | |
|---|---|---|---|---|
| Synchronous | Crash | | | [✓ ? nodes] |
| [✓ Asynchronous] | [✓ Byzantine] | [✓ Pessimistic] | Known nodes | [✓ 1 phases] |
| Partially-Synchronous | Hybrid | Optimistic | [✓ Unknown nodes] | [✓ O(N) Complexity] |

# What is a Blockchain?

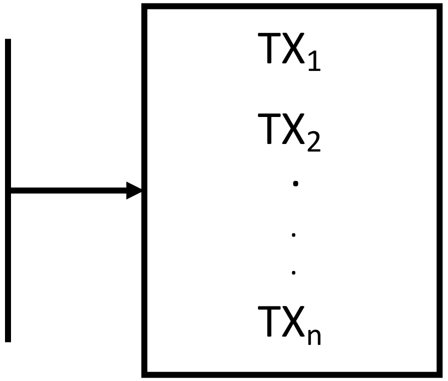
- Signed Transactions are grouped into blocks
- Blocks are chained to each other through pointers (Hence blockchain)
- How is the ledger tamper-free?

  Blocks are connected through **hash-pointers**

# What is a Blockchain?

- Signed Transactions are grouped into blocks
- Blocks are chained to each other through pointers (Hence blockchain)
- How is the ledger tamper-free?

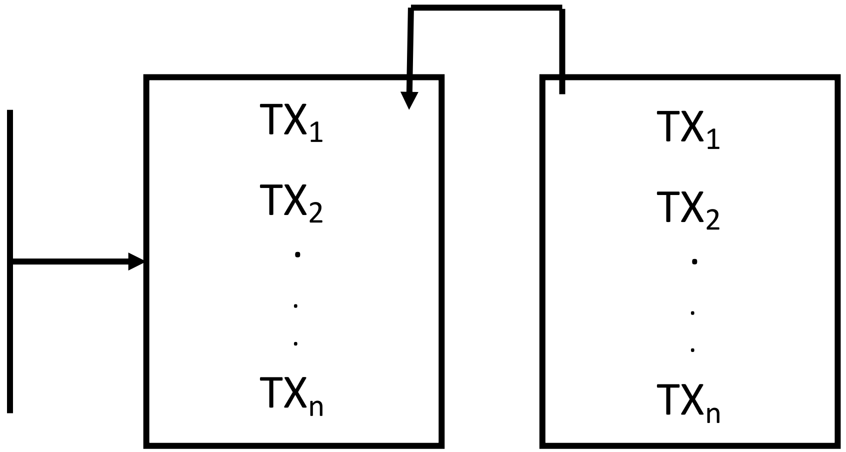
  Blocks are connected through **hash-pointers**



# What is a Blockchain?

- Signed Transactions are grouped into blocks
- Blocks are chained to each other through pointers (Hence blockchain)
- How is the ledger tamper-free?

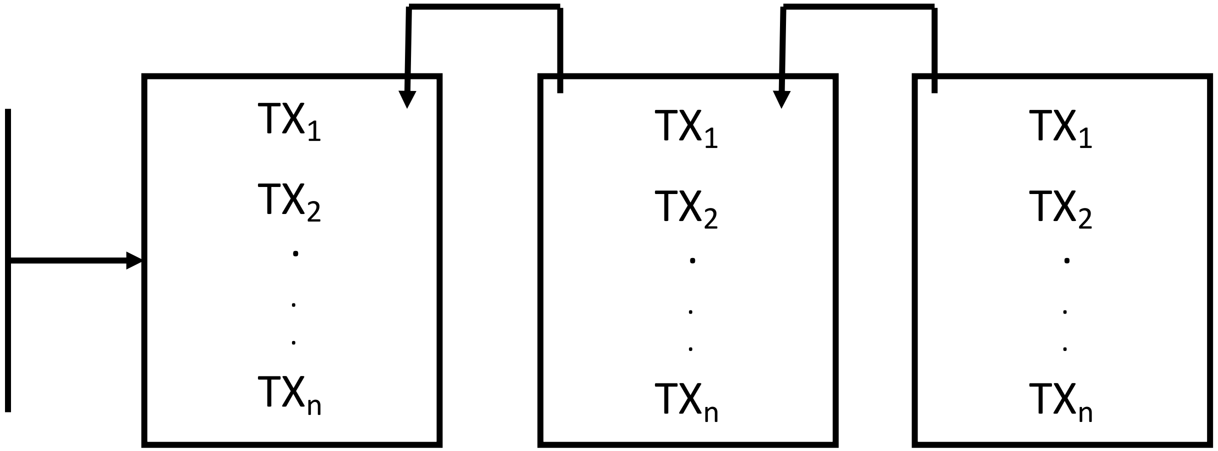
  Blocks are connected through **hash-pointers**



# What is a Blockchain?

- Signed Transactions are grouped into blocks
- Blocks are chained to each other through pointers (Hence blockchain)
- How is the ledger tamper-free?

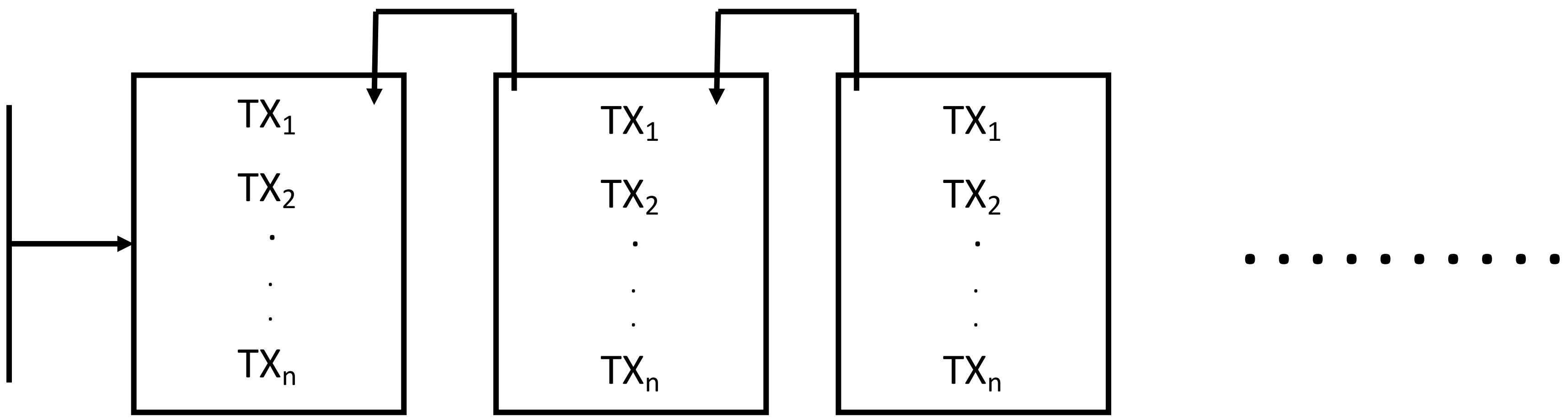
  Blocks are connected through **hash-pointers**



# What is a Blockchain?

- Signed Transactions are grouped into blocks
- Blocks are chained to each other through pointers (Hence blockchain)
- How is the ledger tamper-free?

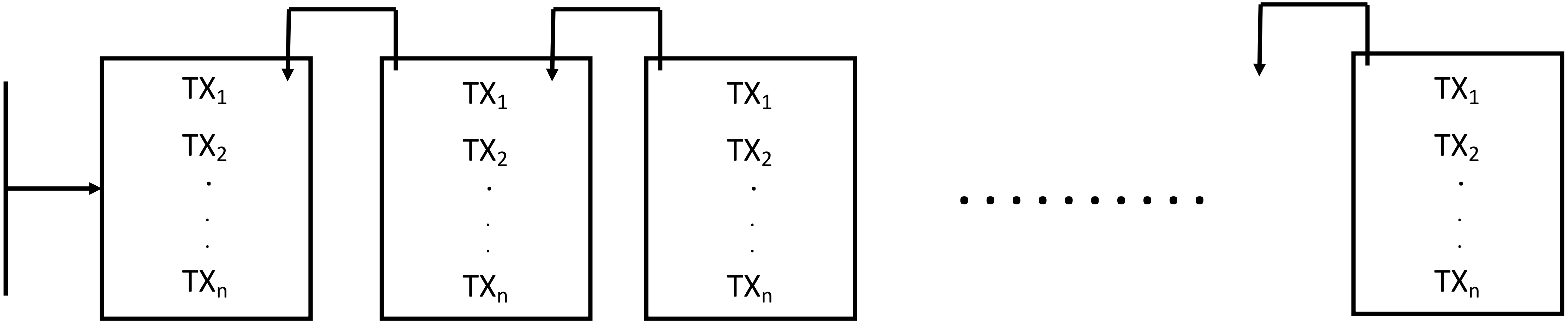
  Blocks are connected through **hash-pointers**



# What is a Blockchain?

- Signed Transactions are grouped into blocks
- Blocks are chained to each other through pointers (Hence blockchain)
- How is the ledger tamper-free?

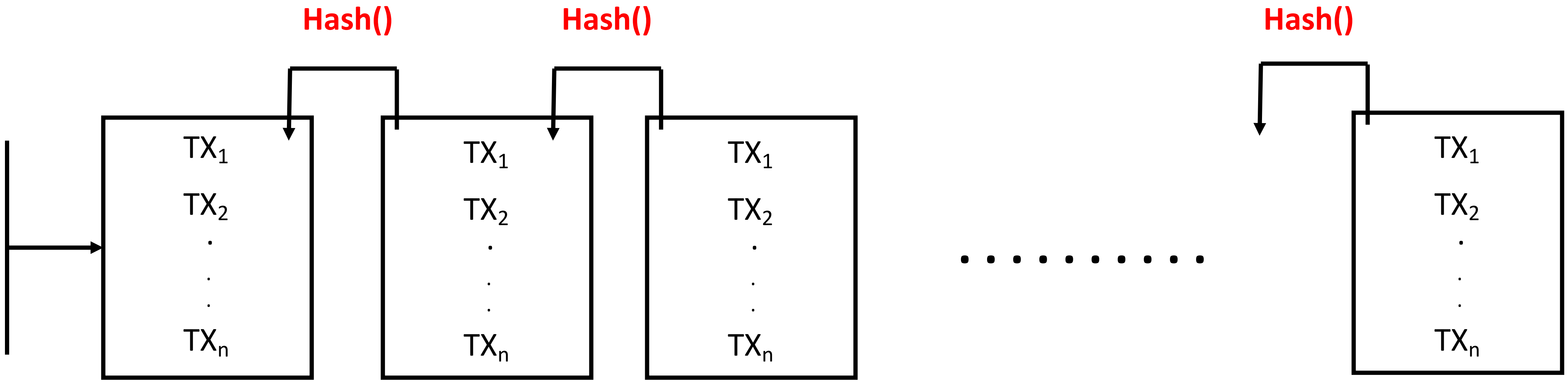
  Blocks are connected through **hash-pointers**



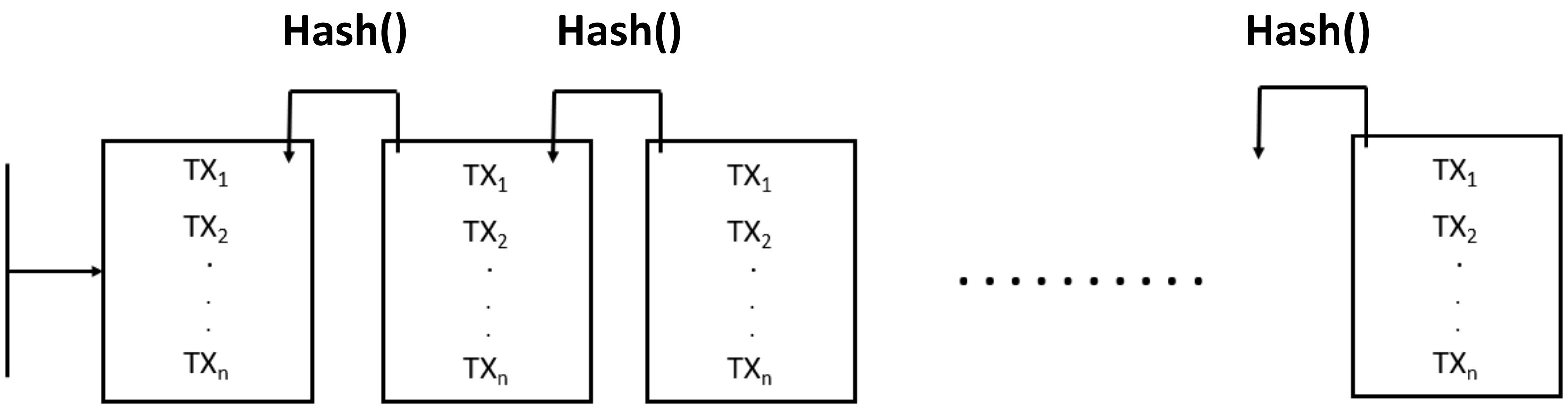
# What is a Blockchain?

- Signed Transactions are grouped into blocks
- Blocks are chained to each other through pointers (Hence blockchain)
- How is the ledger tamper-free?

  Blocks are connected through **hash-pointers**

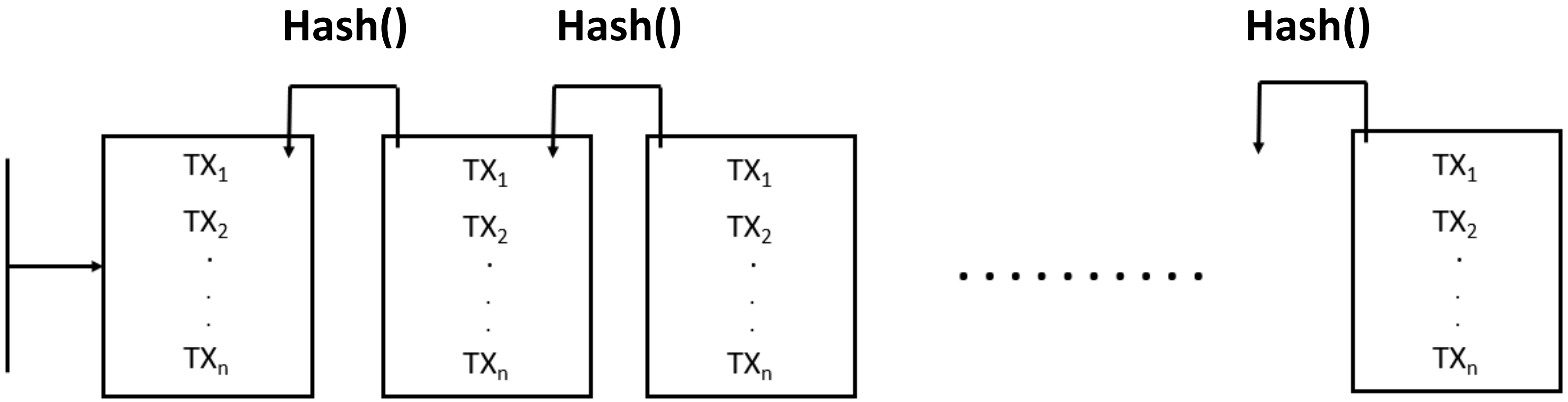


# Making Progress



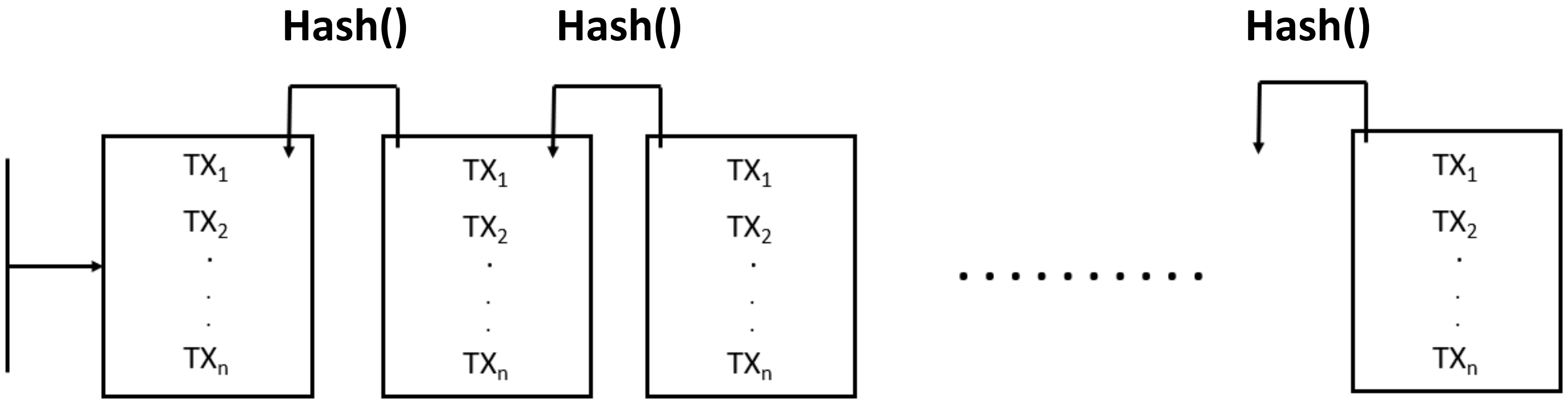
# Making Progress

- To make progress:
  - Network nodes **validate** new transactions are consistent.



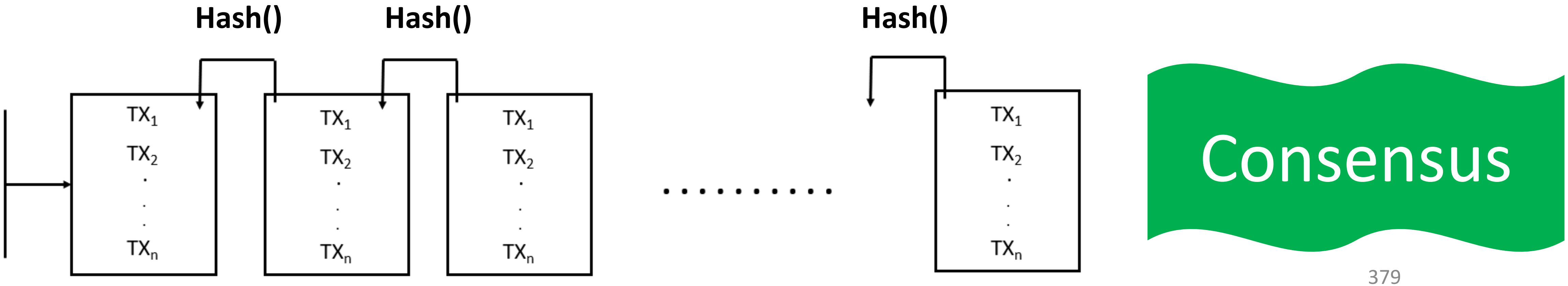
# Making Progress

- To make progress:
  - Network nodes **validate** new transactions are consistent.
  - Network nodes need to agree on the next block to be added to the blockchain



# Making Progress

- To make progress:
  - Network nodes **validate** new transactions are consistent.
  - Network nodes need to agree on the next block to be added to the blockchain



# Permissionless Blockchains have Unknown Number of Participants

# Reach Consensus Using Mining

Permissionless Blockchains have **Unknown Number** of Participants

# Reach Consensus Using Mining

## Replace Communication with Computation!!

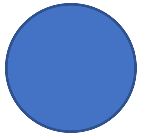
Permissionless Blockchains have **Unknown Number** of Participants

# Proof of Work Consensus

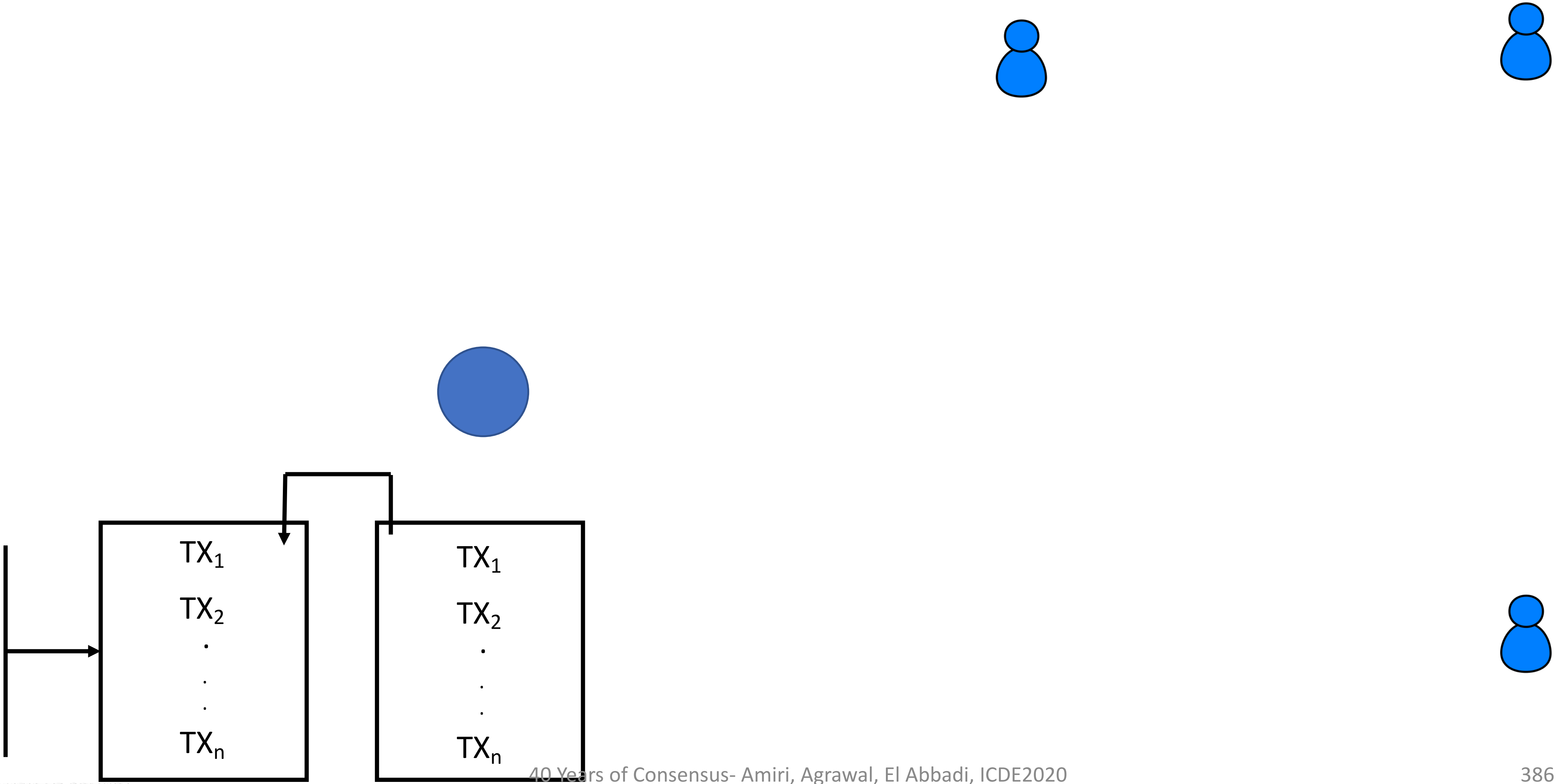
- Intuitively, network nodes race to solve a puzzle
- This puzzle is computationally expensive
- Once a network node finds (mines) a solution:
    - It adds its block of transactions to the blockchain
    - It multi-casts the solution to other network nodes
    - Other network nodes accept and verify the solution

# Mining Details

# Mining Details



# Mining Details



$TX_1$

$TX_2$

.

.

.

$TX_n$

$TX_1$

$TX_2$

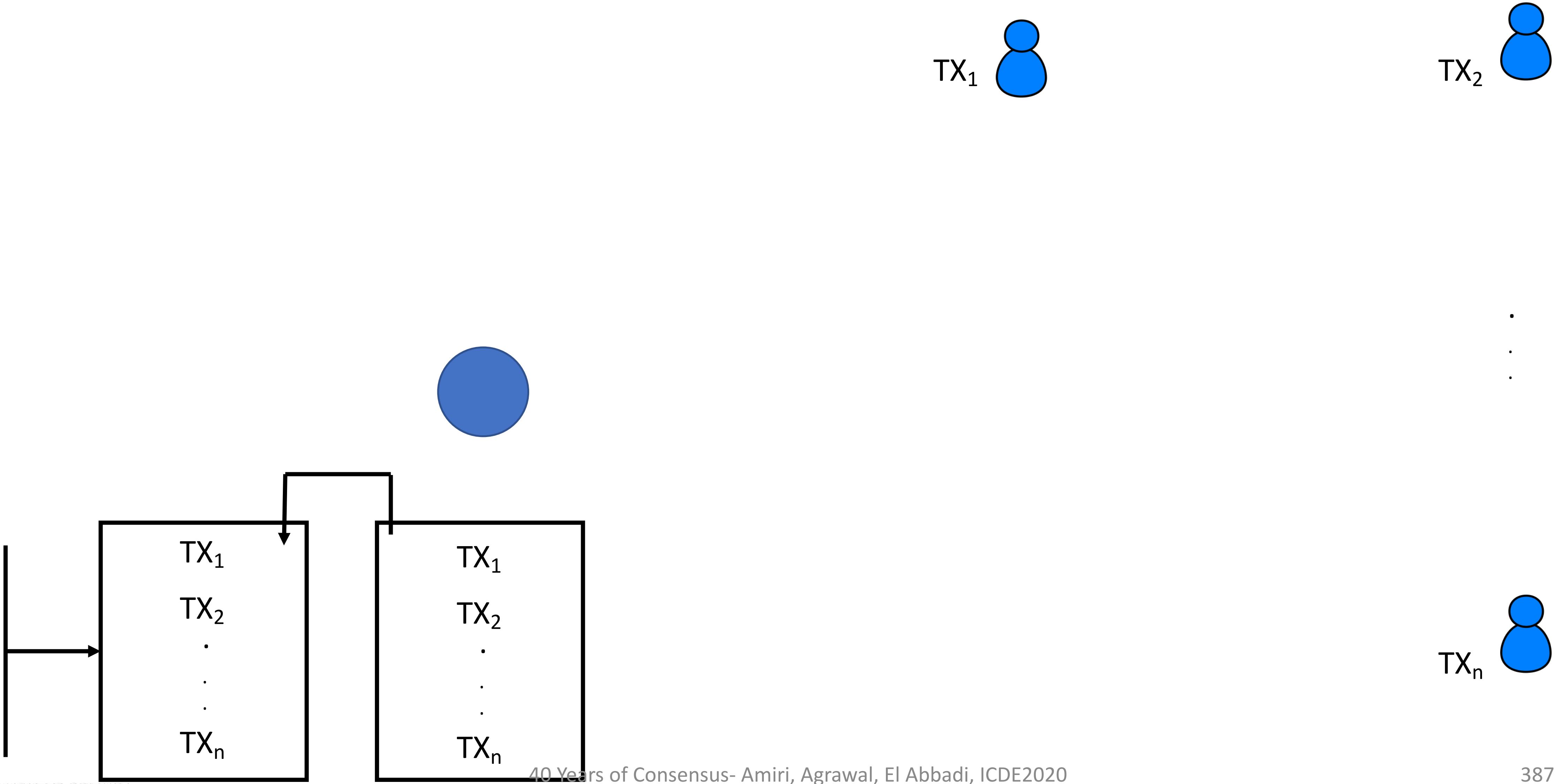
.

.

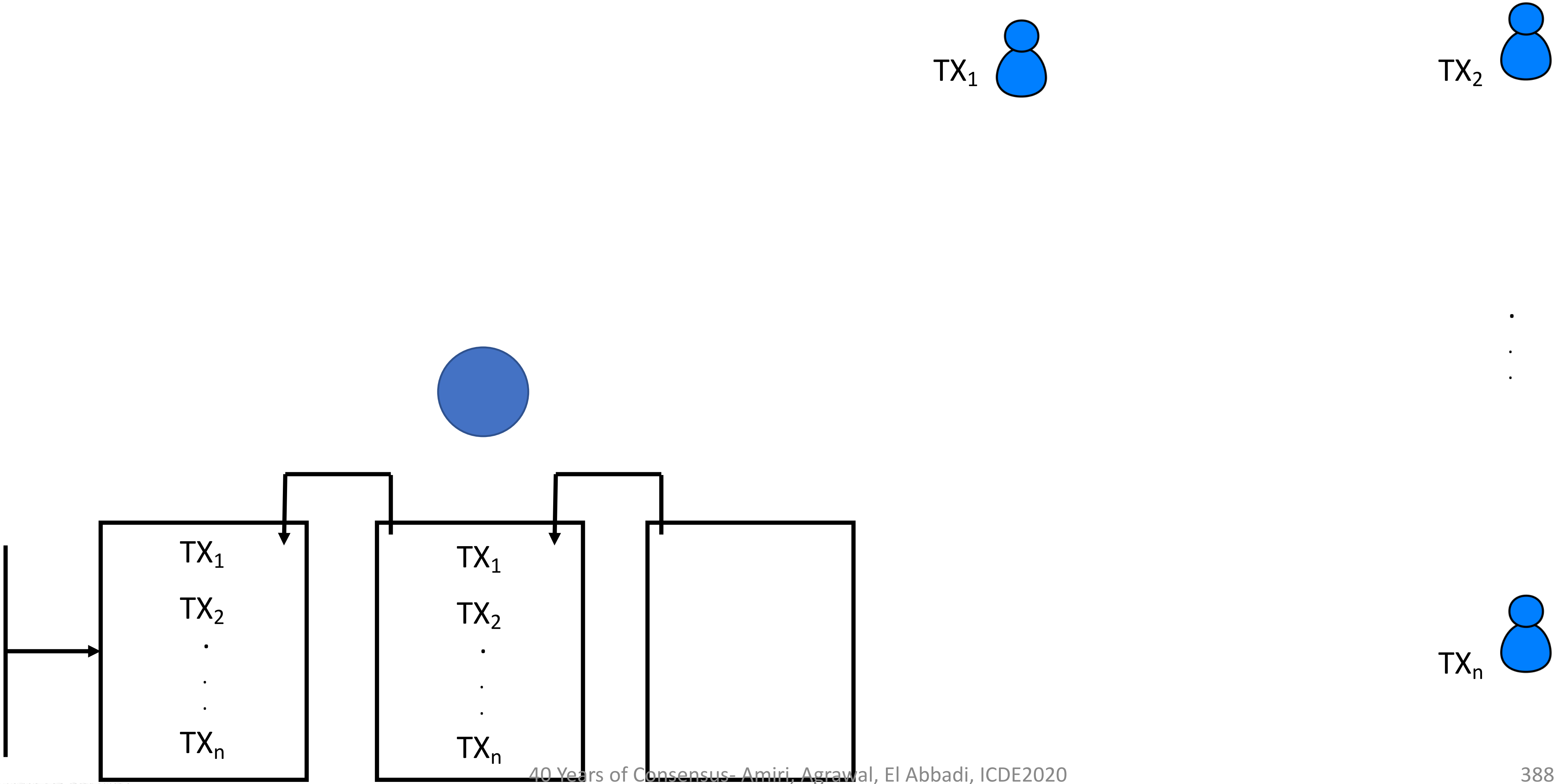
.

$TX_n$

# Mining Details



# Mining Details



$TX_1$

$TX_2$

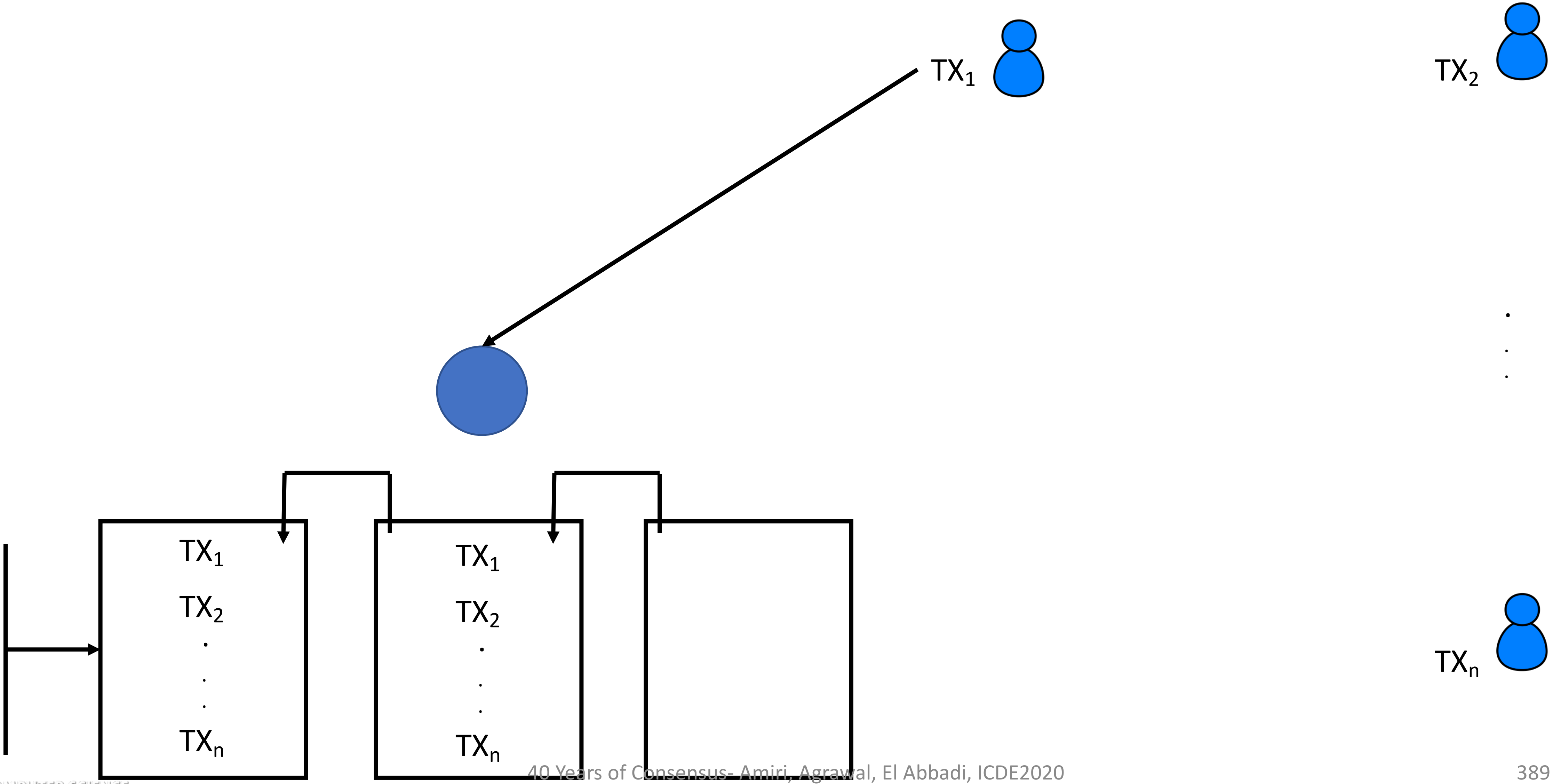
.
.
.

$TX_n$

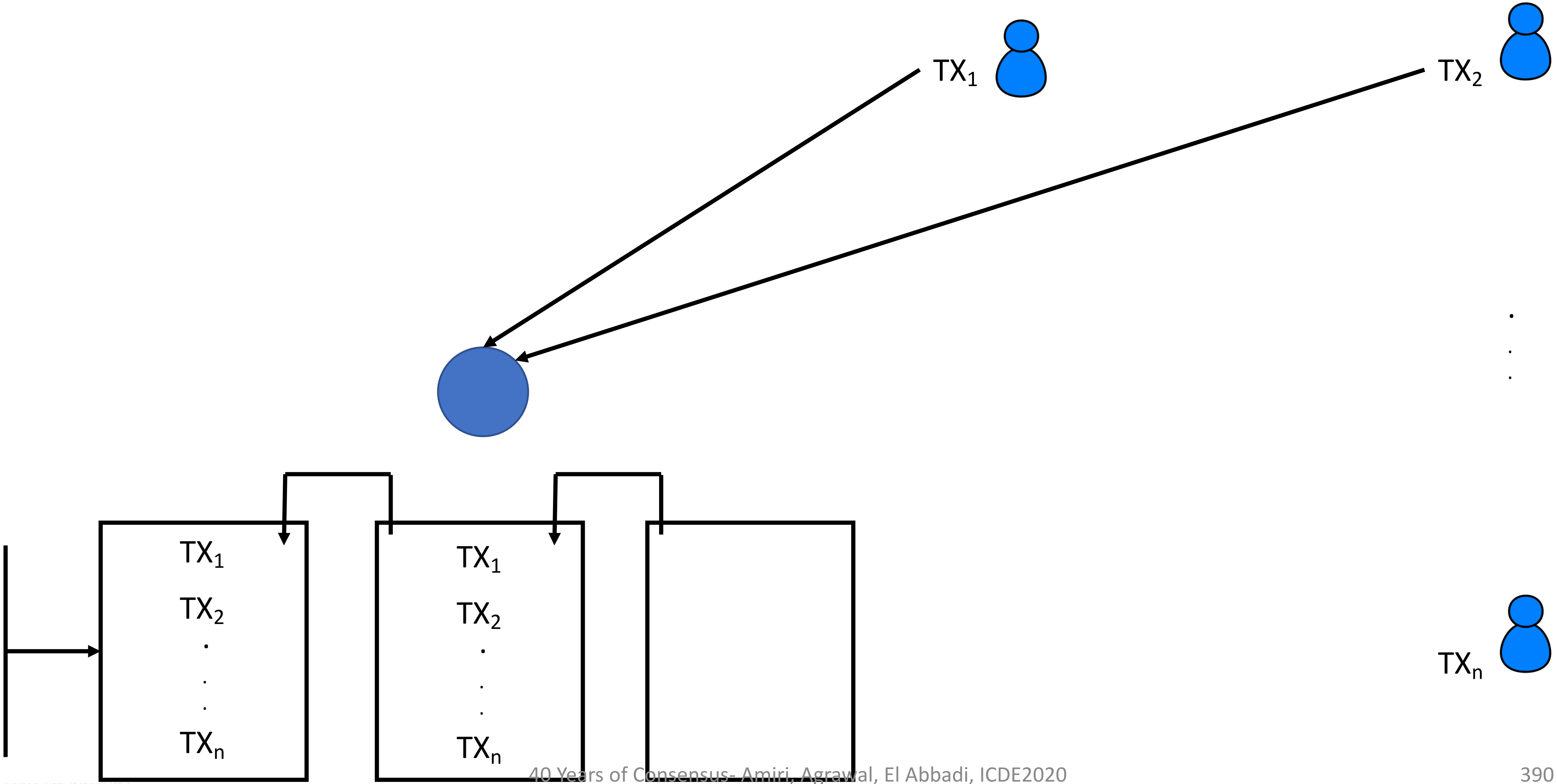
$TX_1$
$TX_2$
.
.
.
$TX_n$

$TX_1$
$TX_2$
.
.
.
$TX_n$

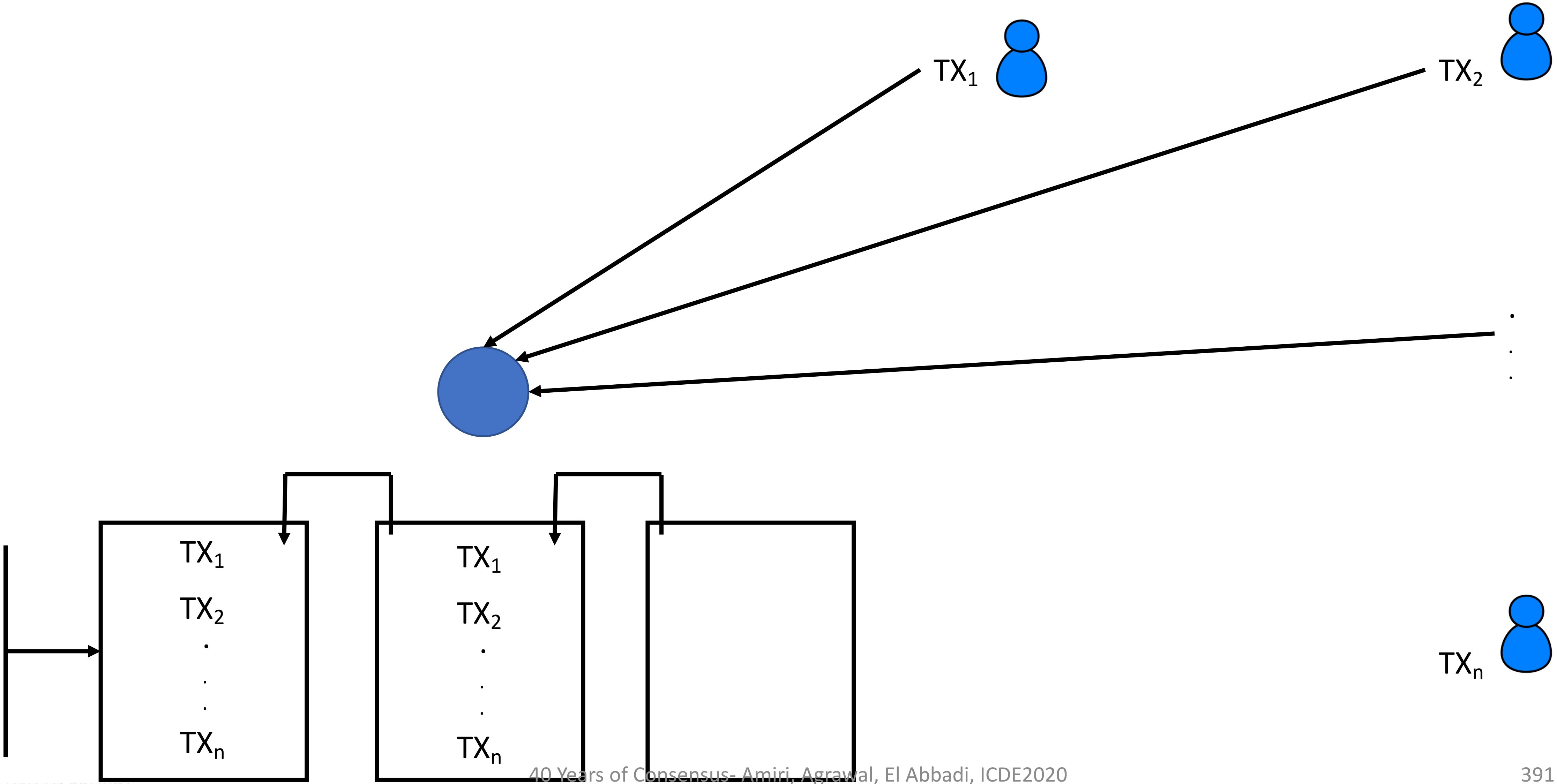
# Mining Details



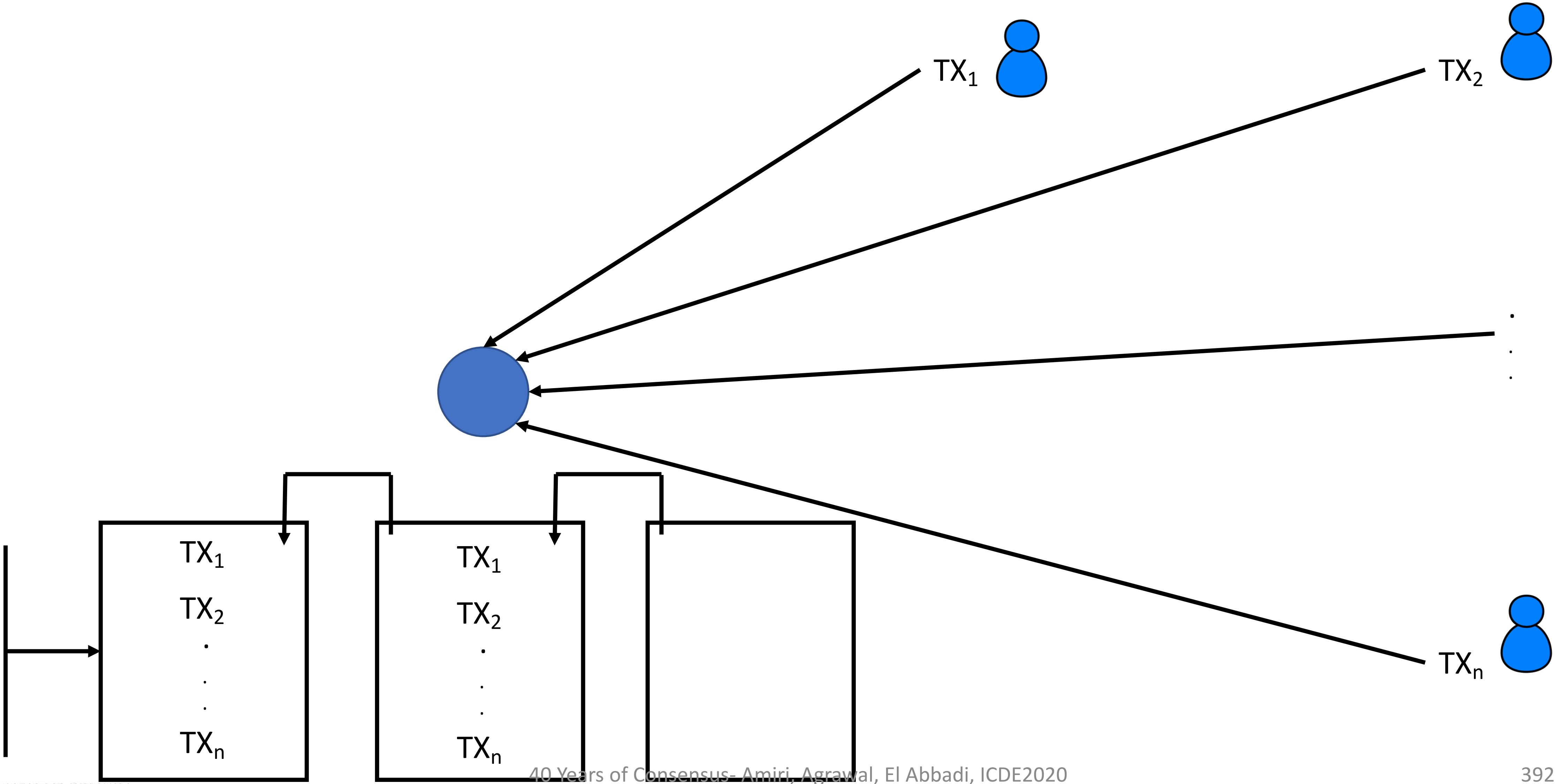
# Mining Details



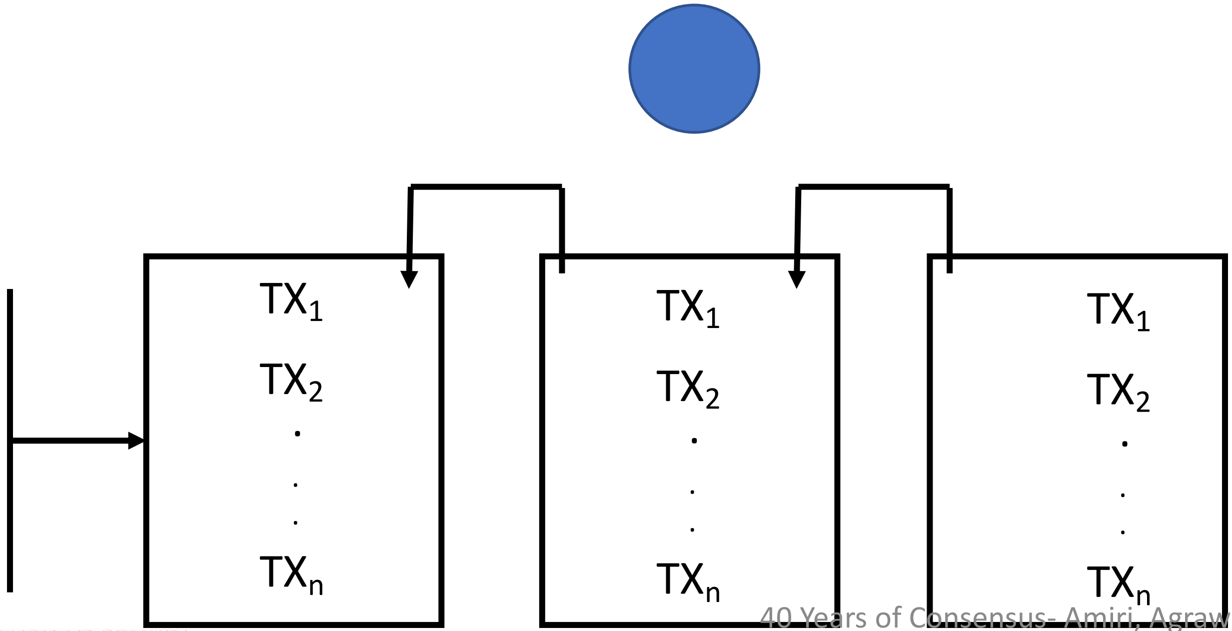
# Mining Details



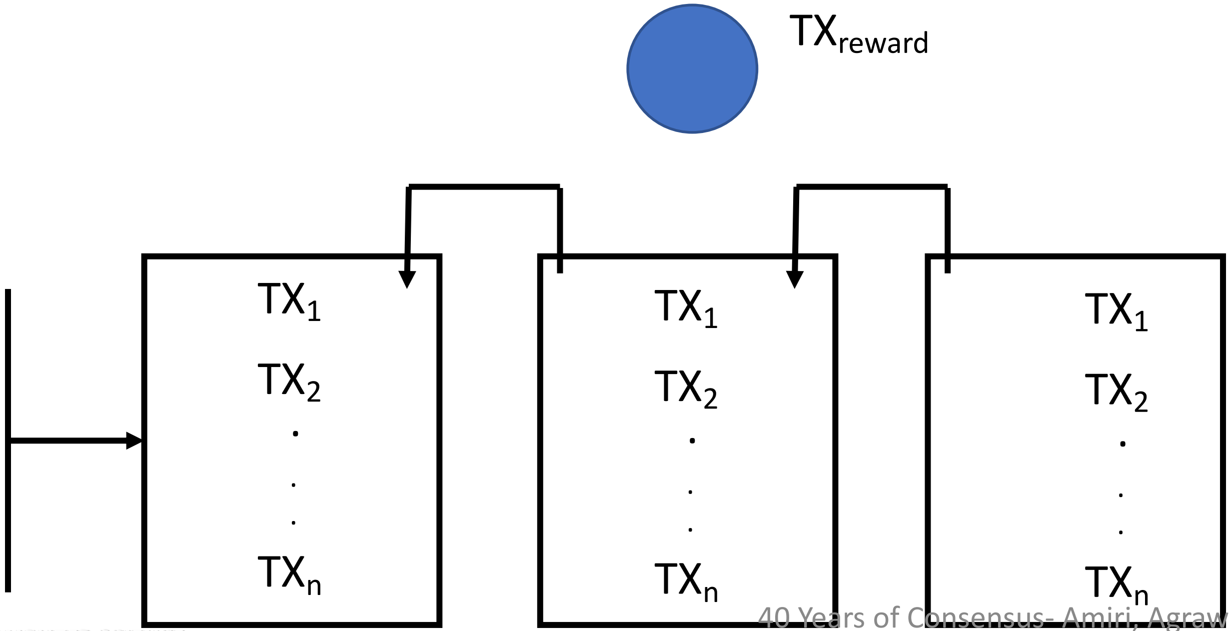
# Mining Details



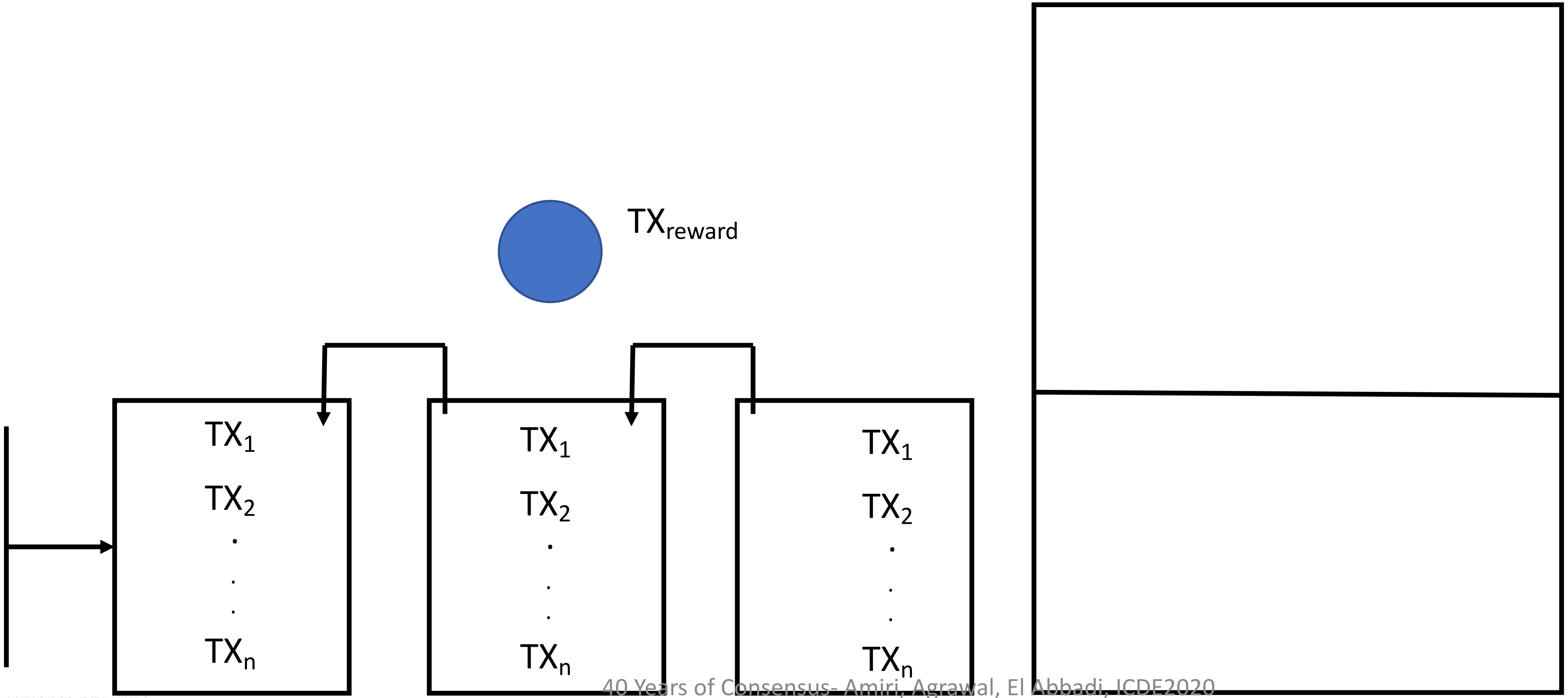
# Mining Details



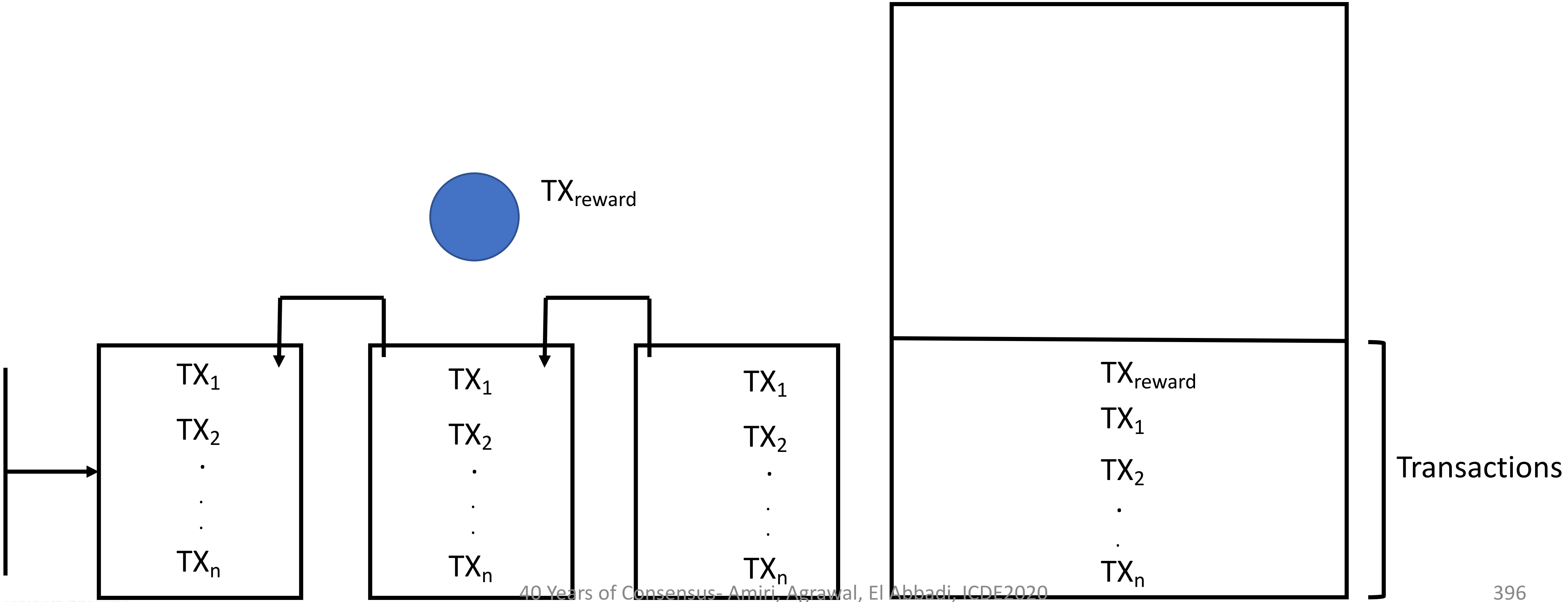
# Mining Details



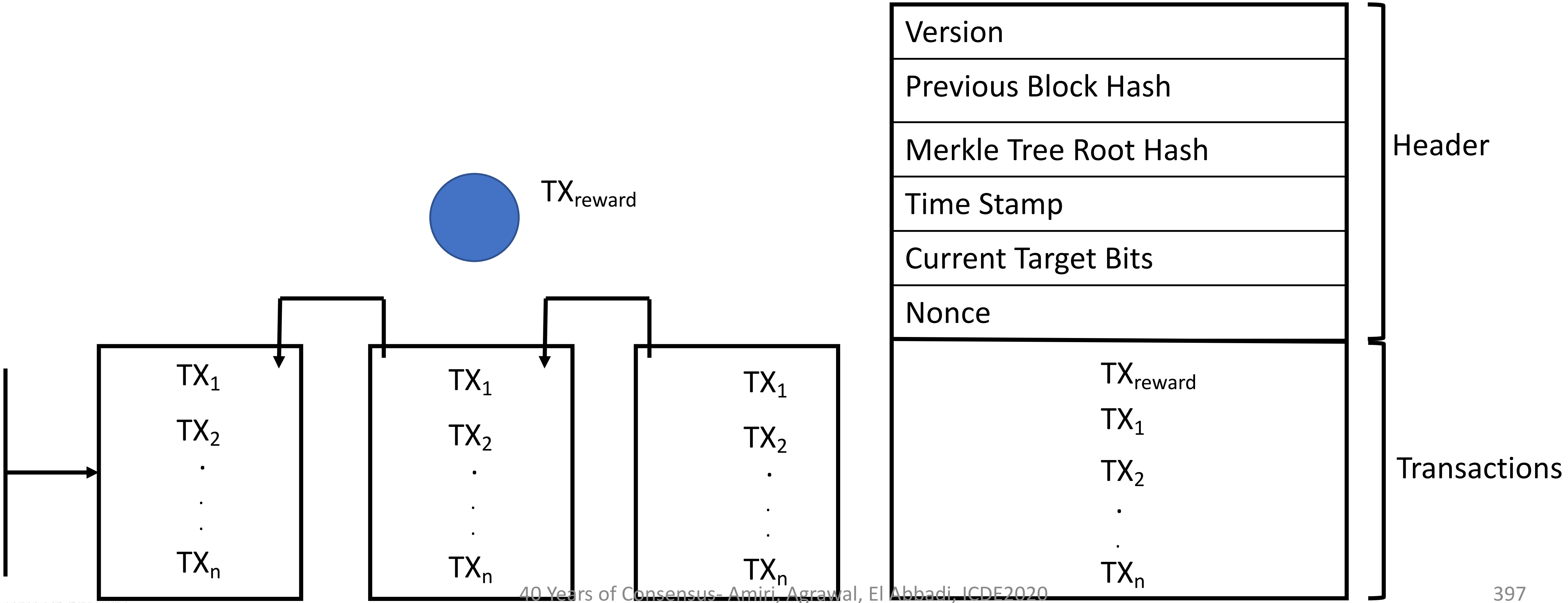
# Mining Details



# Mining Details



# Mining Details



$TX_{reward}$

$TX_1$
$TX_2$
.
.
.
$TX_n$

$TX_1$
$TX_2$
.
.
.
$TX_n$

$TX_1$
$TX_2$
.
.
.
$TX_n$

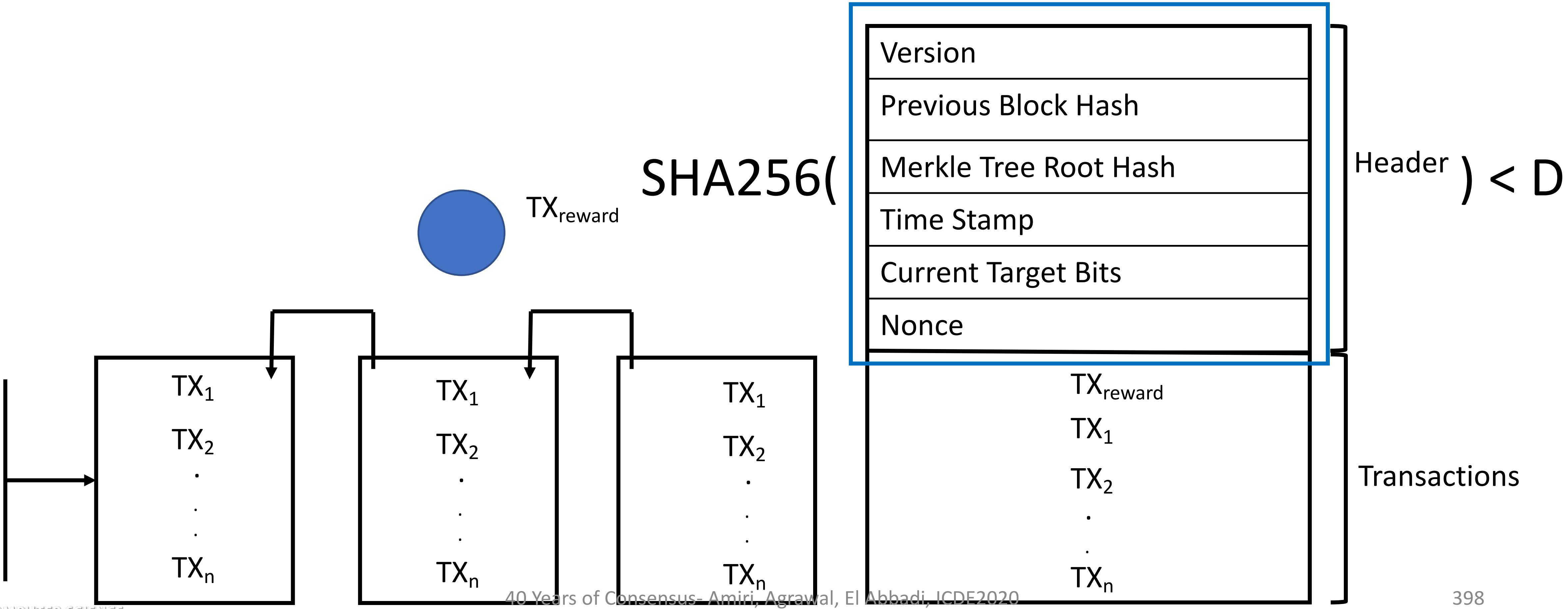
Version
Previous Block Hash
Merkle Tree Root Hash
Time Stamp
Current Target Bits
Nonce

Header

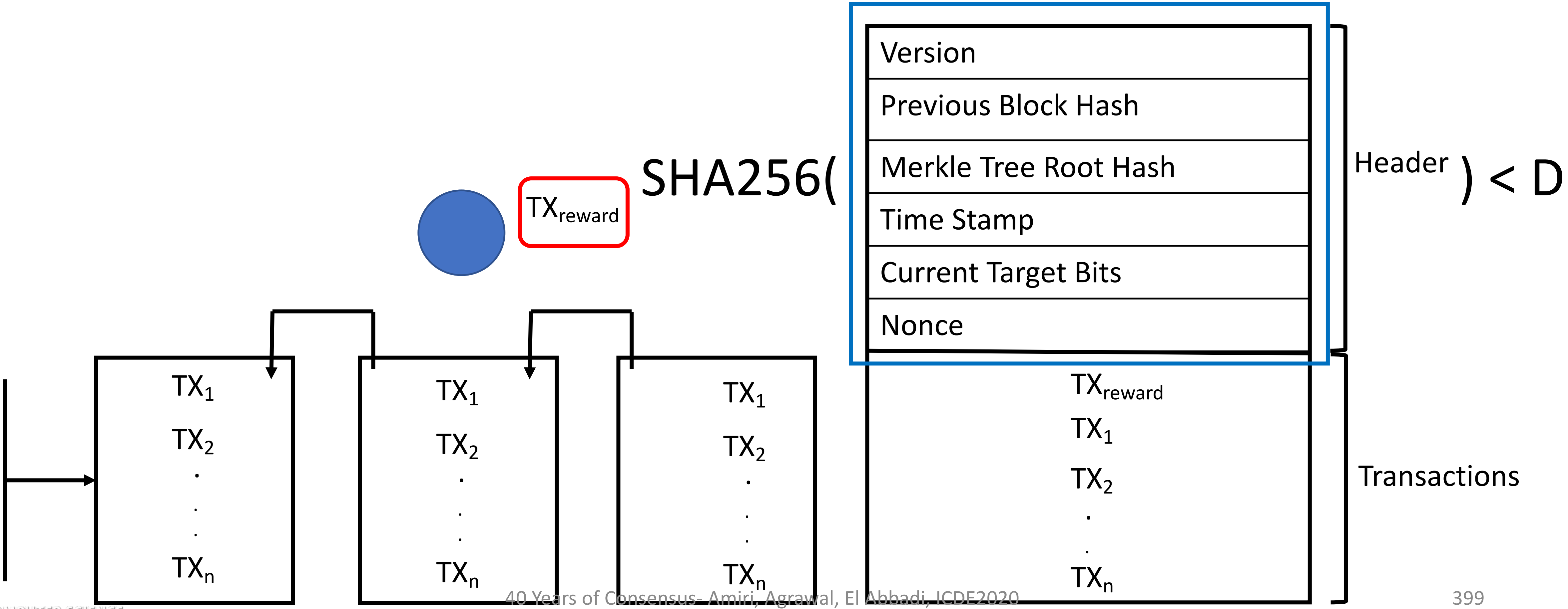
$TX_{reward}$
$TX_1$
$TX_2$
.
.
$TX_n$

Transactions

# Mining Details



# Mining Details



$TX_{reward}$

SHA256(

Version

Previous Block Hash

Merkle Tree Root Hash

Time Stamp

Current Target Bits

Nonce

Header ) < D

$TX_{reward}$

$TX_1$

$TX_2$

.

.

$TX_n$

Transactions

$TX_1$

$TX_2$

.

.

.

$TX_n$

$TX_1$

$TX_2$

.

.

.

$TX_n$

$TX_1$

$TX_2$

.

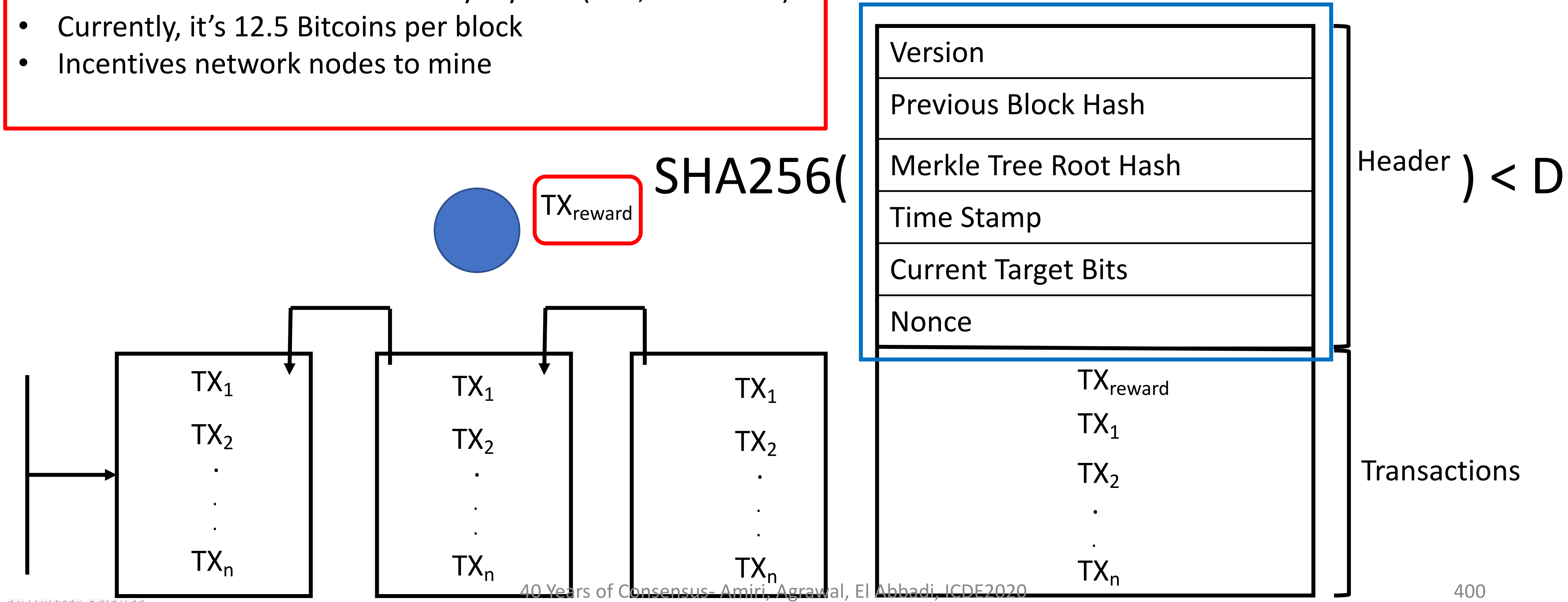
.

.

$TX_n$

# Mining Details

- $TX_{reward}$ is self signed (also called coinbase transaction)
- $TX_{reward}$ is bitcoin’s way to create new coins
- The reward value is halved every 4 years (210,000 blocks)
- Currently, it’s 12.5 Bitcoins per block
- Incentives network nodes to mine



$TX_{reward}$

SHA256( Version | Previous Block Hash | Merkle Tree Root Hash | Time Stamp | Current Target Bits | Nonce ) < D

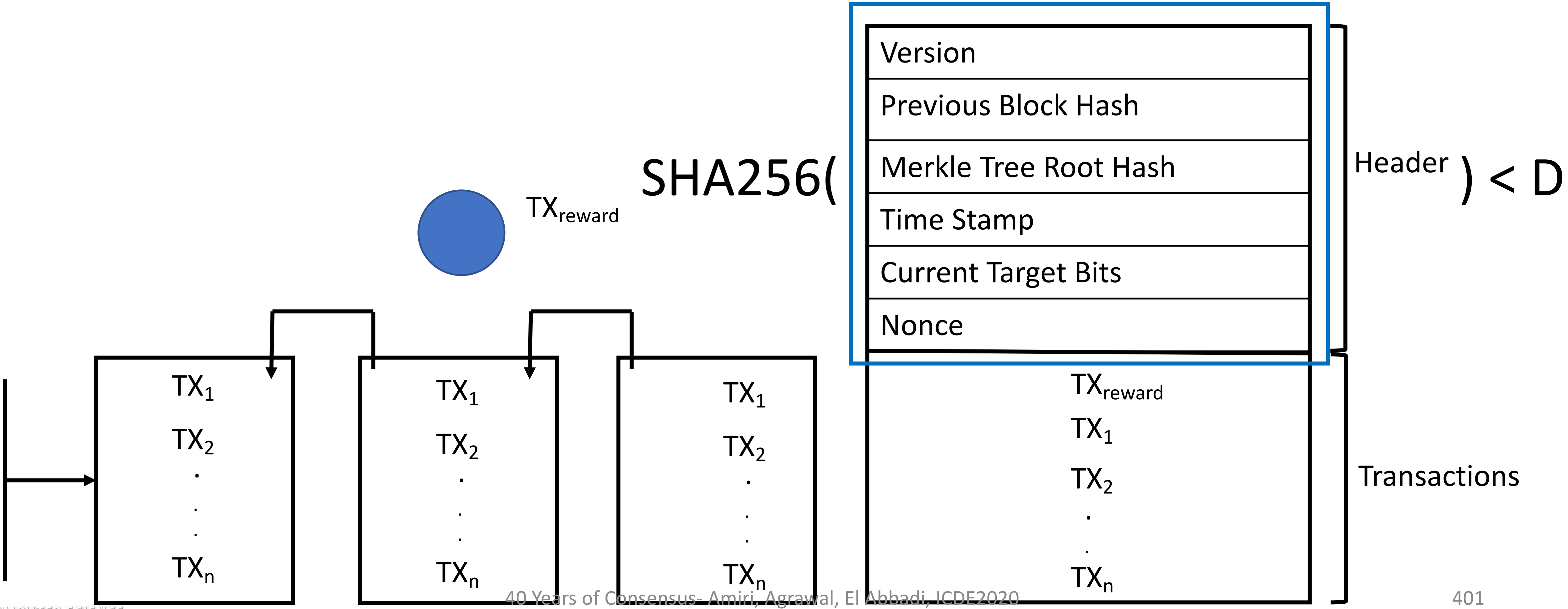
Header

Transactions

$TX_{reward}$, $TX_1$, $TX_2$, …, $TX_n$

$TX_1$, $TX_2$, …, $TX_n$

# Mining Details



$TX_{reward}$

SHA256(

Version

Previous Block Hash

Merkle Tree Root Hash

Time Stamp

Current Target Bits

Nonce

Header ) < D

$TX_{reward}$

$TX_1$

$TX_2$

.

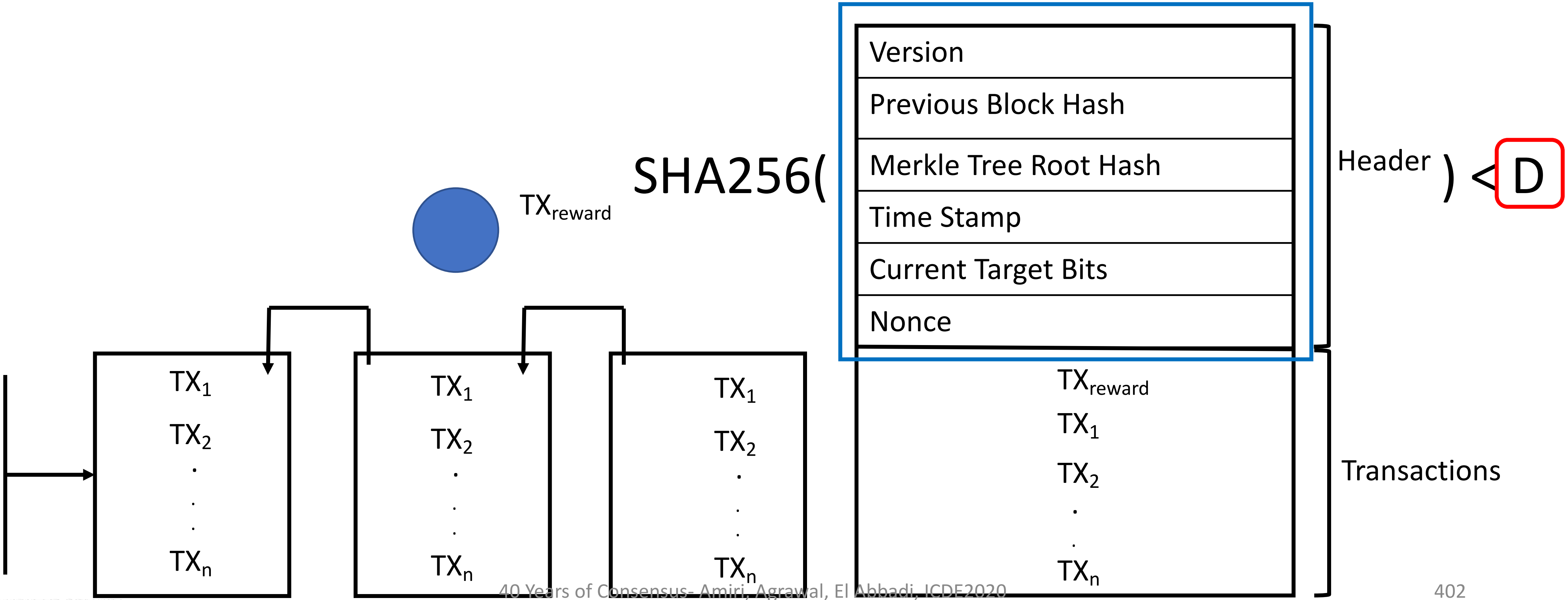
.

.

$TX_n$

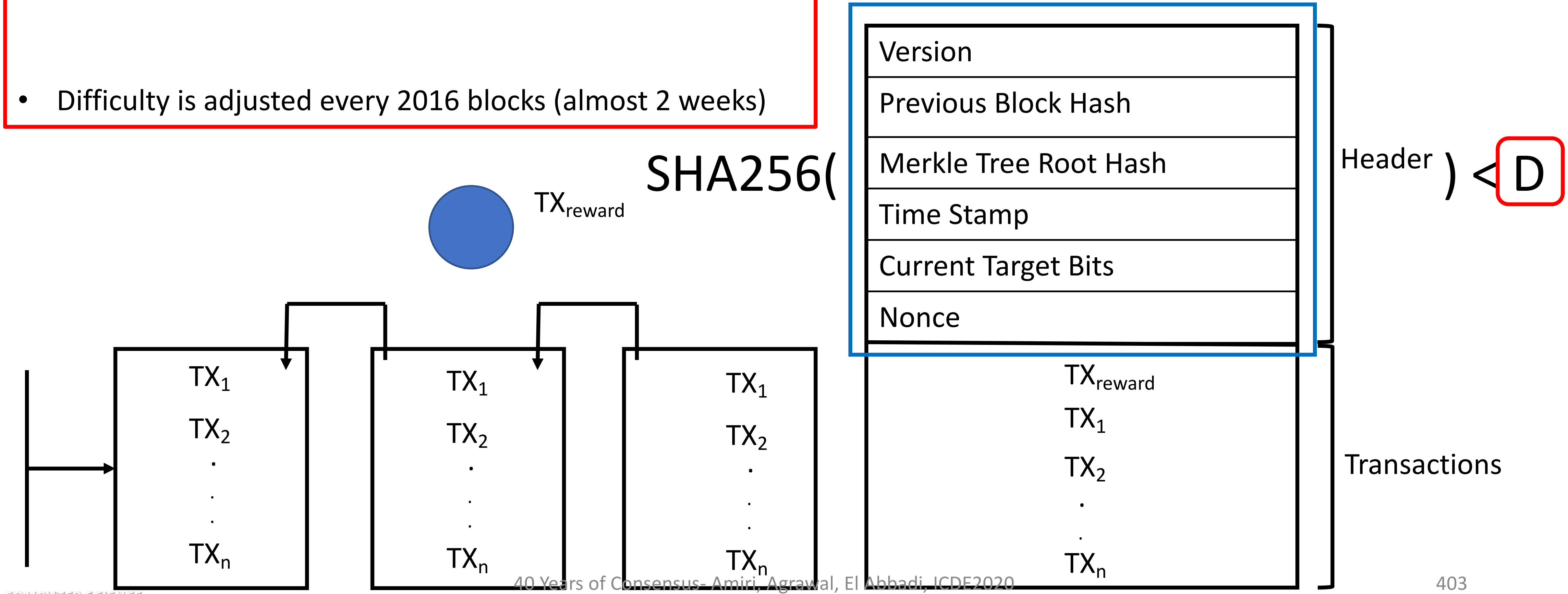
Transactions

# Mining Details



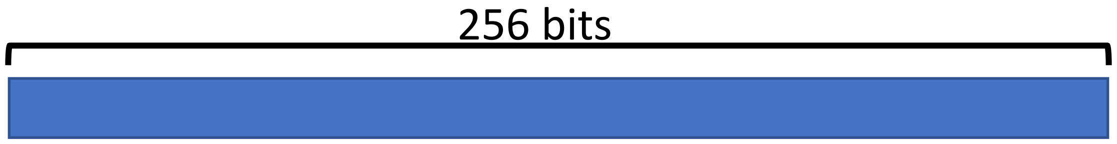
# Mining Details

- D: dynamically adjusted difficulty
- Difficulty is adjusted every 2016 blocks (almost 2 weeks)

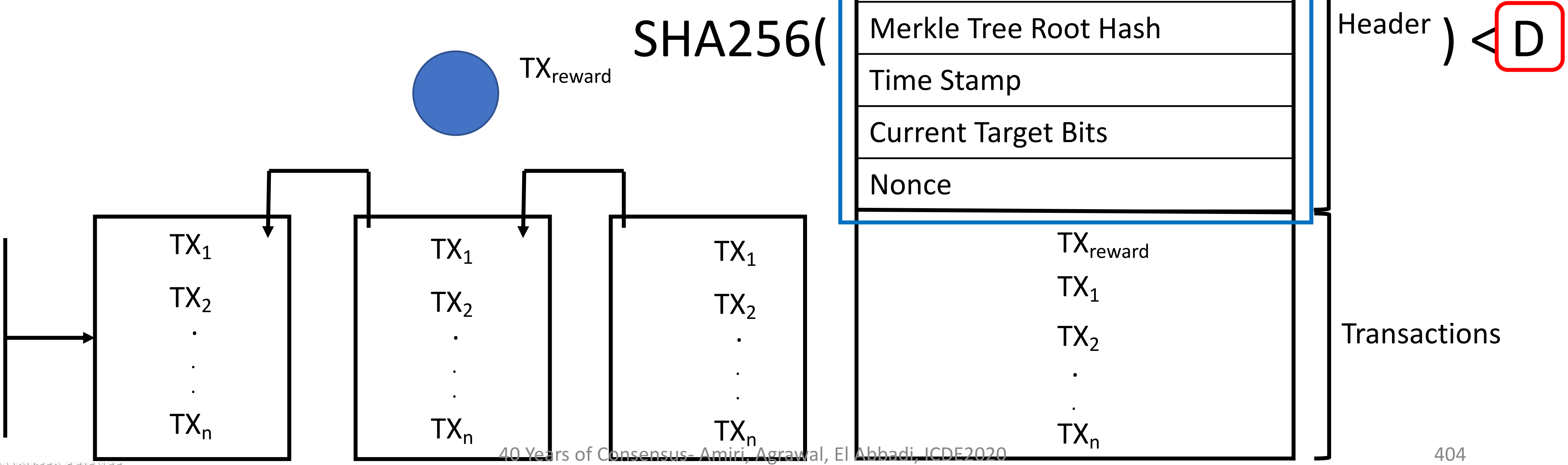


# Mining Details

- D: dynamically adjusted difficulty


256 bits

- Difficulty is adjusted every 2016 blocks (almost 2 weeks)



$\text{TX}_{reward}$

SHA256(

Version

Previous Block Hash

Merkle Tree Root Hash

Time Stamp

Current Target Bits

Nonce

Header ) < D

$\text{TX}_{reward}$

$\text{TX}_1$

$\text{TX}_2$

.

.

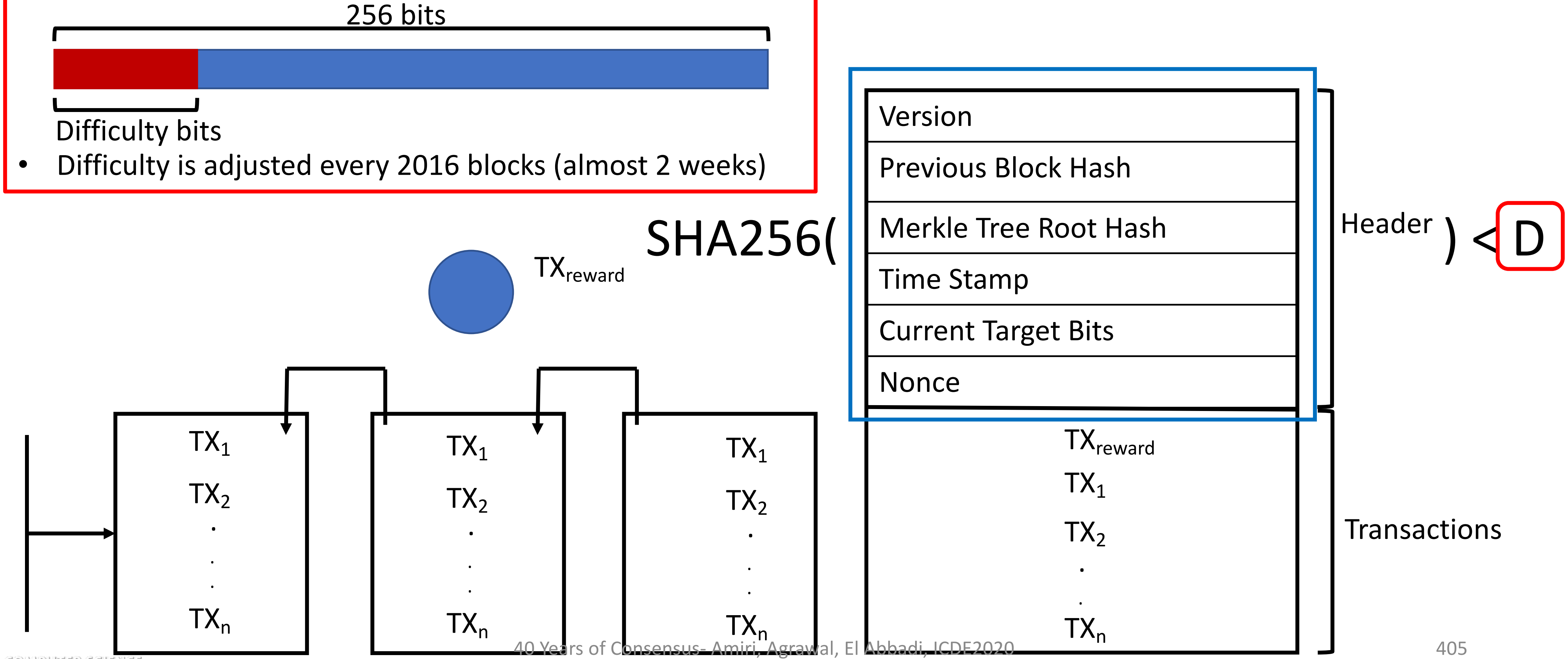
.

$\text{TX}_n$

Transactions

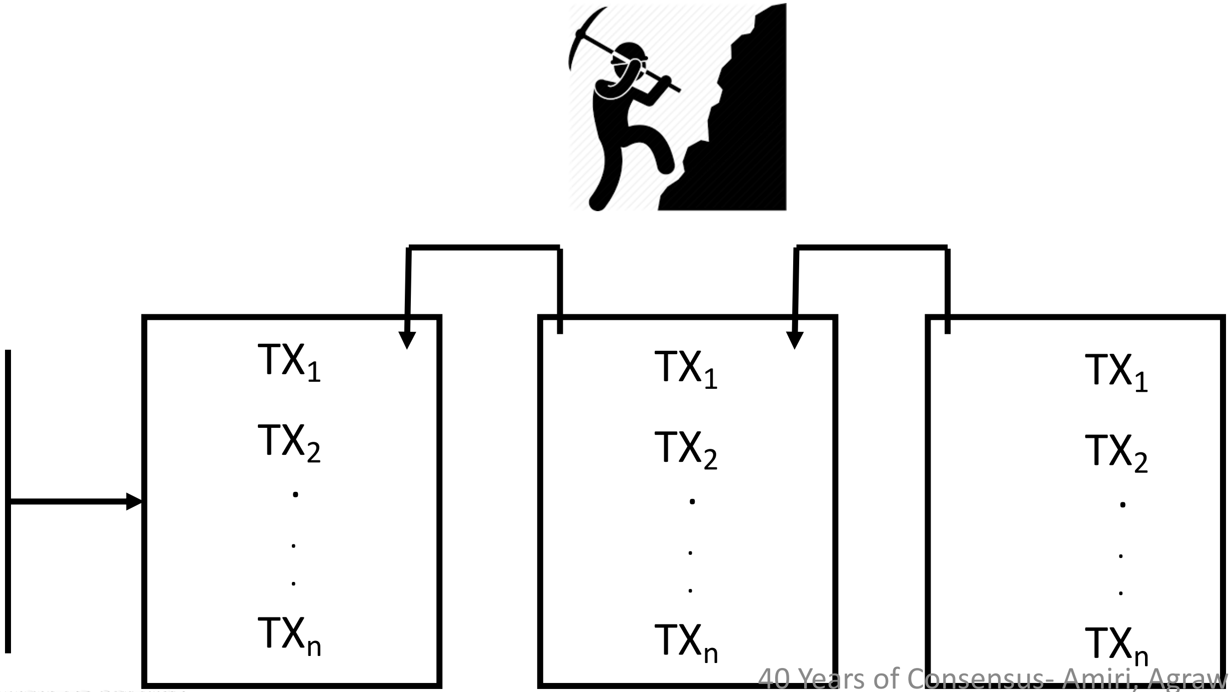
# Mining Details

- D: dynamically adjusted difficulty



- Difficulty is adjusted every 2016 blocks (almost 2 weeks)

# Mining Details: Block Contents



$TX_1$

$TX_2$

.

.

.

$TX_n$

$TX_1$

$TX_2$

.

.

.

$TX_n$

$TX_1$

$TX_2$

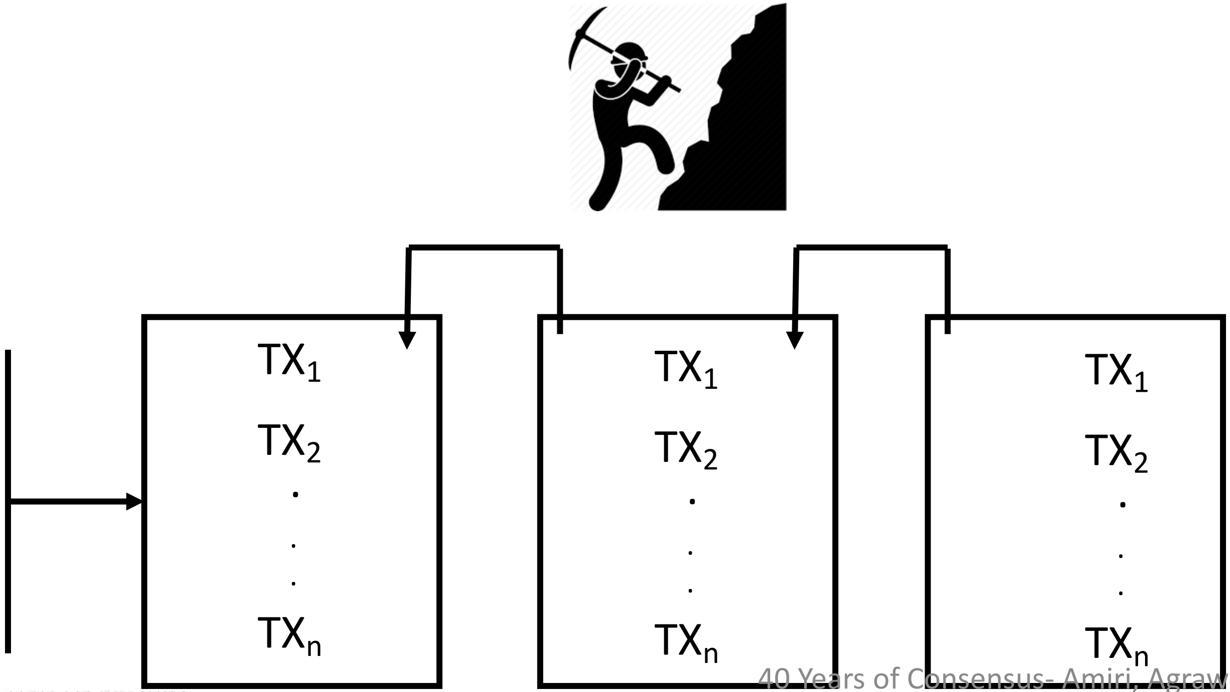
.

.

.

$TX_n$

# Mining Details: Block Contents



$TX_{reward}$

$TX_1$

$TX_2$

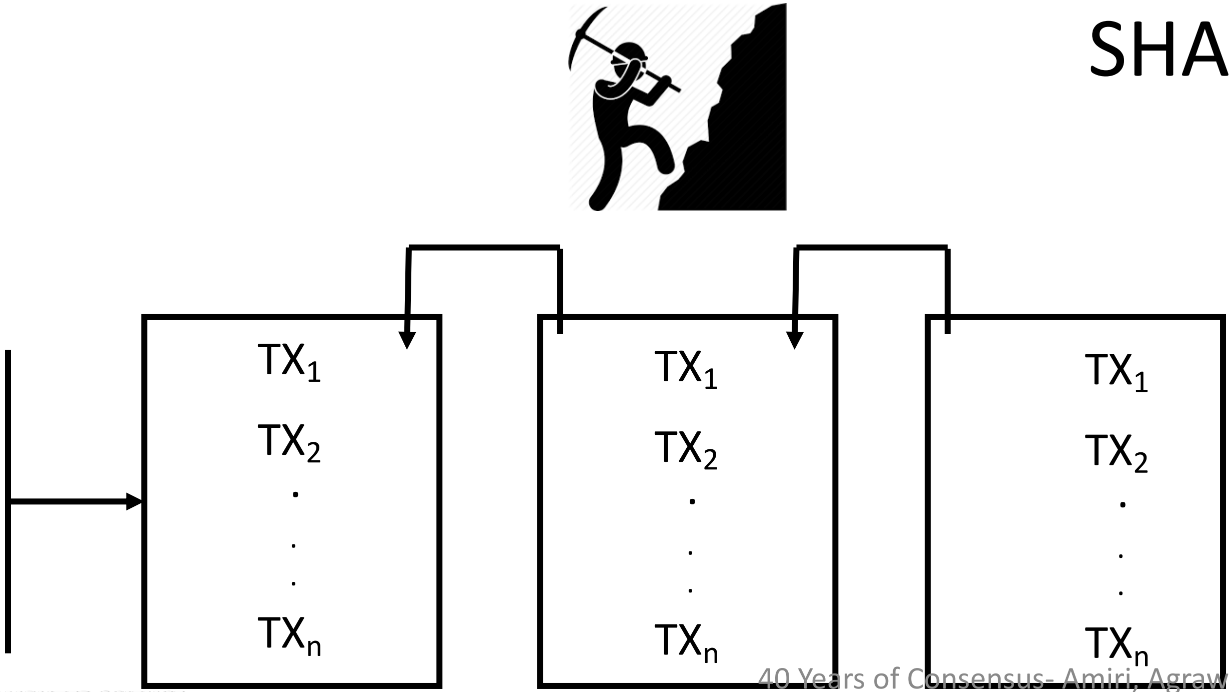
.

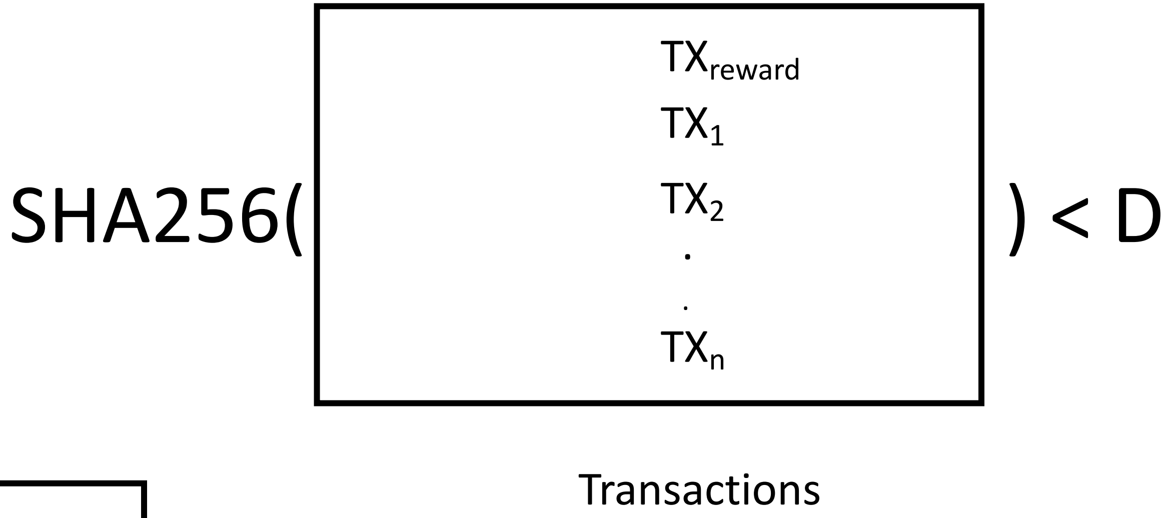
.

$TX_n$

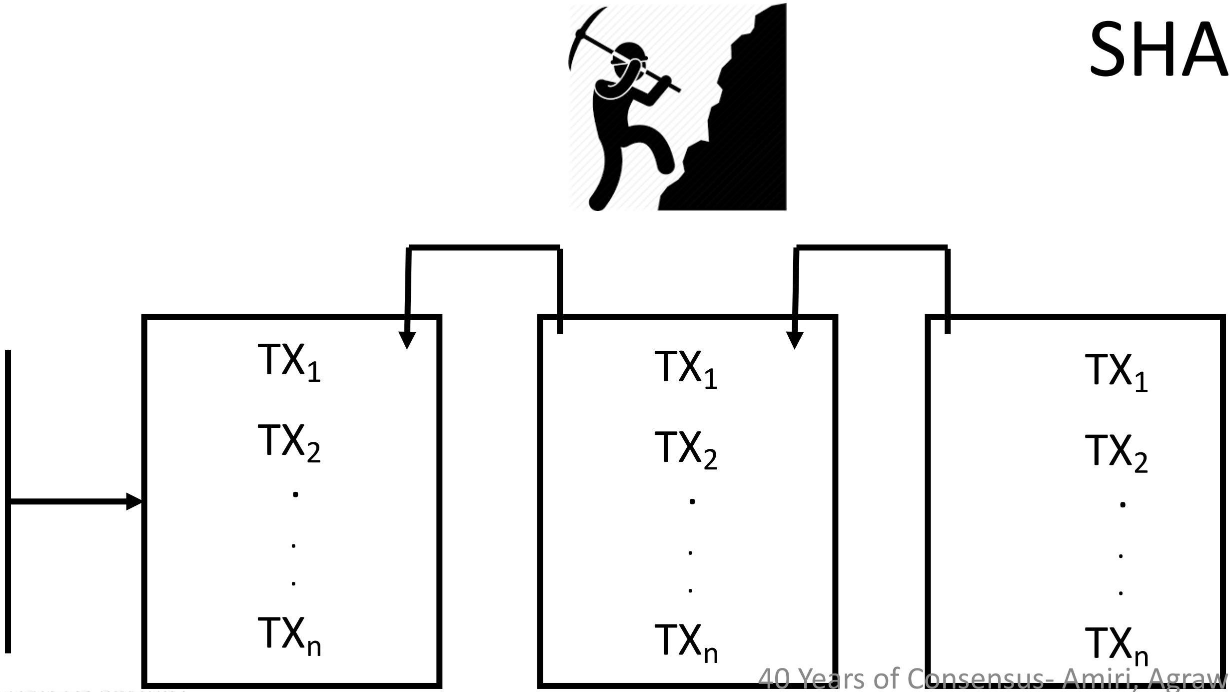
Transactions

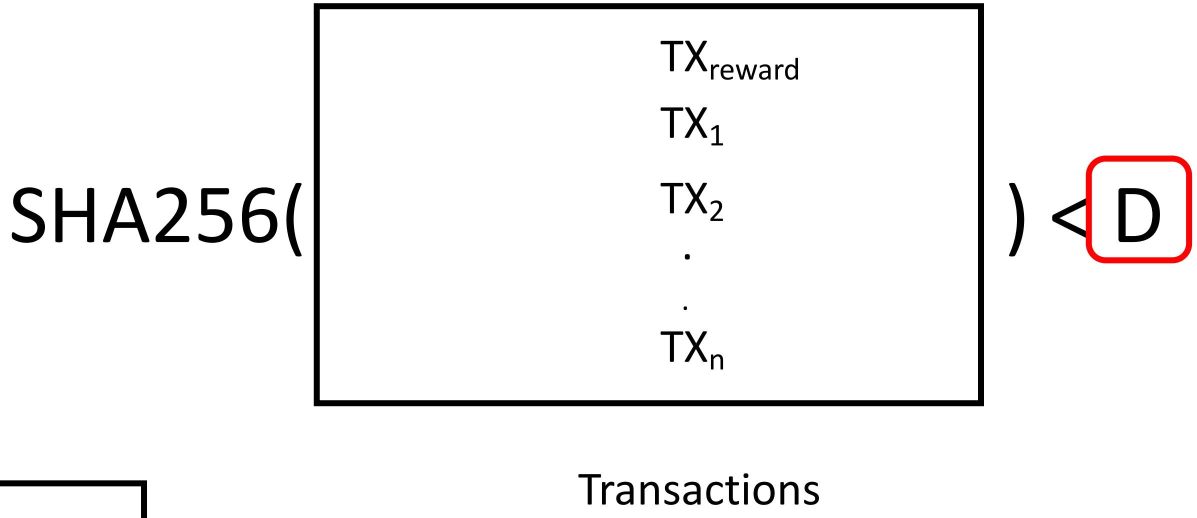
# Mining Details: Block Contents





# Mining Details: Block Contents

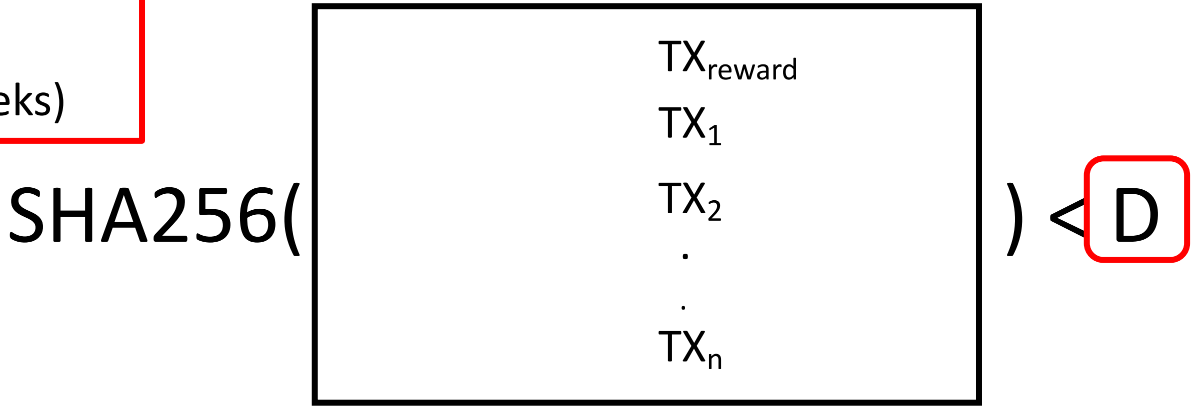




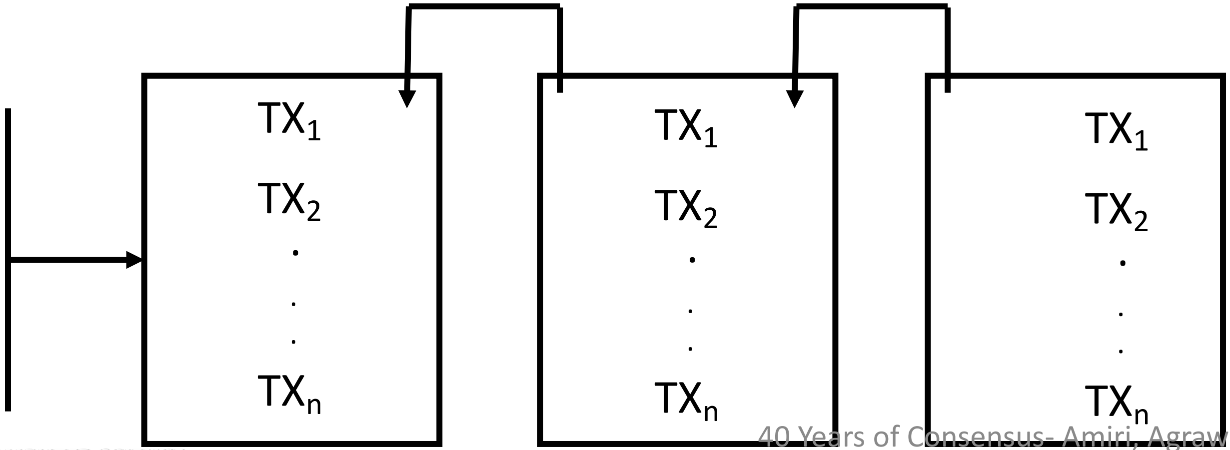
# Mining Details: Block Contents

- D: dynamically adjusted difficulty

- Difficulty is adjusted every 2016 blocks (almost 2 weeks)

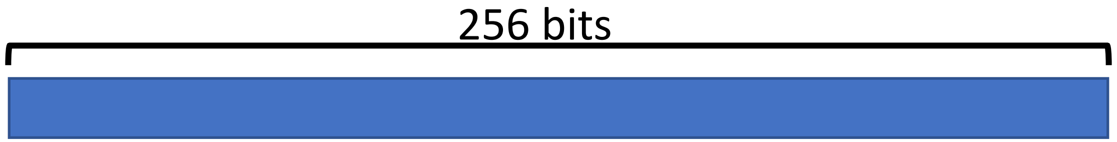


Transactions

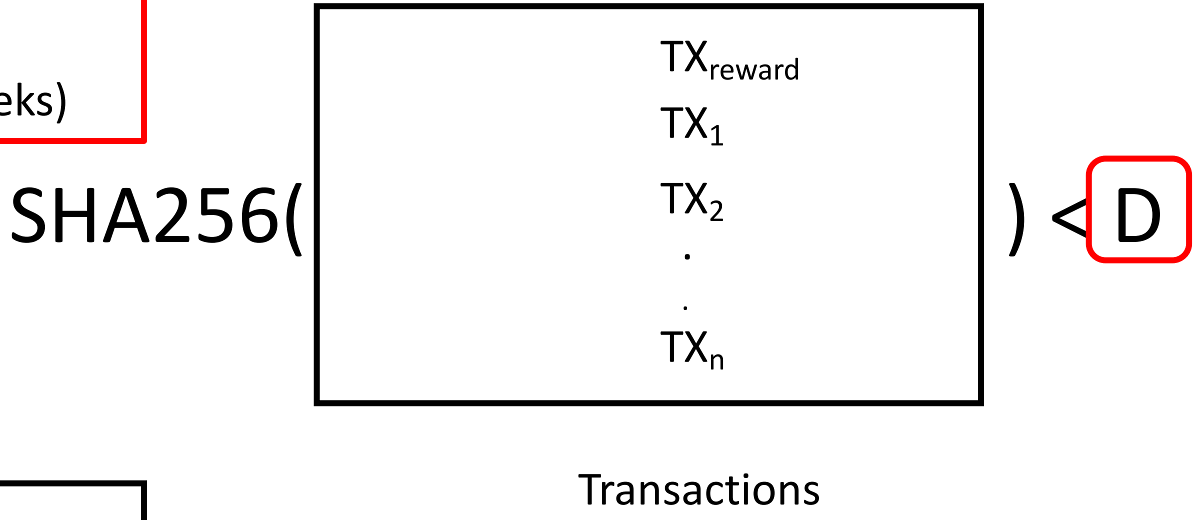


# Mining Details: Block Contents

- D: dynamically adjusted difficulty

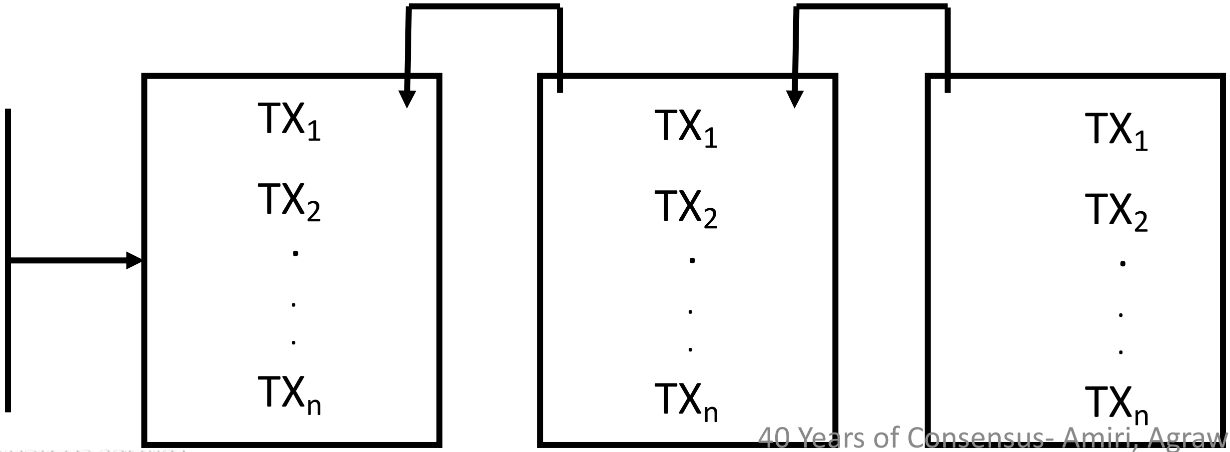


- Difficulty is adjusted every 2016 blocks (almost 2 weeks)



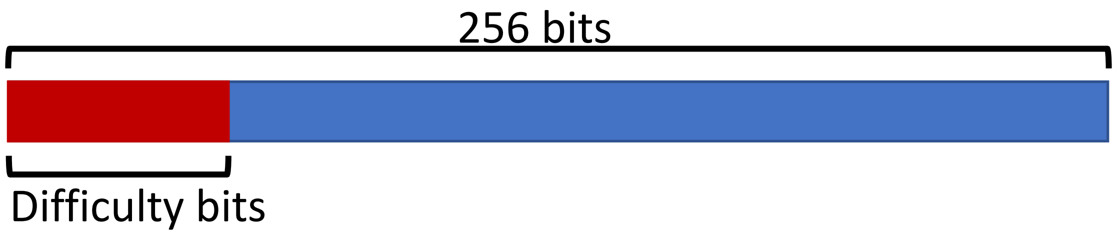
SHA256( $TX_{reward}$ $TX_1$ $TX_2$ . . $TX_n$ ) < D

Transactions

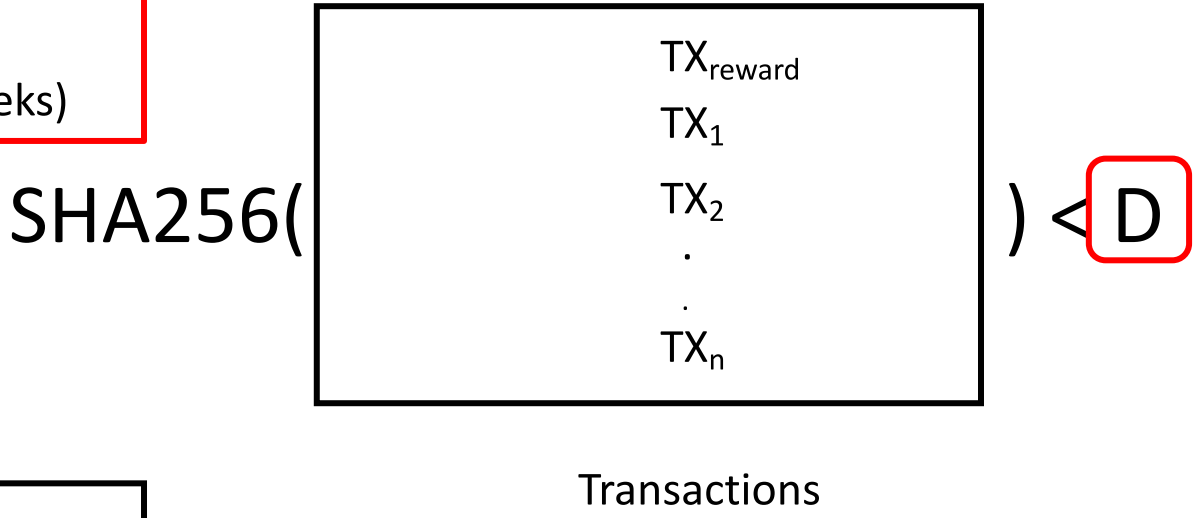


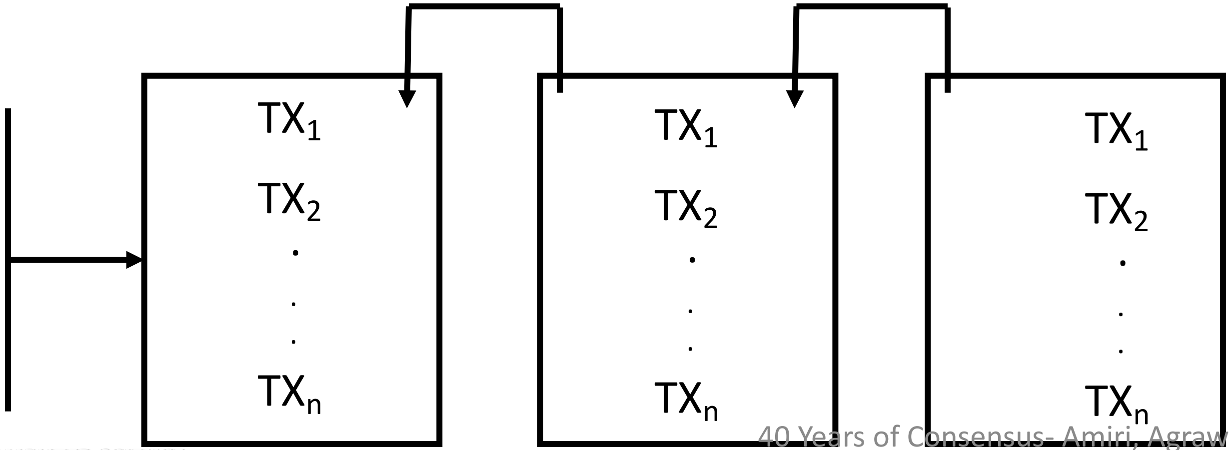
# Mining Details: Block Contents

- D: dynamically adjusted difficulty



- Difficulty is adjusted every 2016 blocks (almost 2 weeks)





# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$\underbrace{\texttt{000000000000000000}}_{\text{18 zeros}}\texttt{cf3620d570d08d1799a1cafbbfae512fdba2124665eca0}$

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$\underbrace{000000000000000000}_{\text{18 zeros}}$cf3620d570d08d1799a1cafbbfae512fdba2124665eca0

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDF64F2DF1B032429C72

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$\underbrace{000000000000000000}_{\text{18 zeros}}$cf3620d570d08d1799a1cafbbfae512fdba2124665eca0

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDF64F2DF1B0324

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$\underbrace{000000000000000000}_{\text{18 zeros}}$cf3620d570d08d1799a1cafbbfae512fdba2124665eca0

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDBF64F2DF1B0324

SHA256(V,P,M,T,C,1) =
DF64342507E785FDC0D4C776D7142BB2BC6467F09E0040A3E9F65E38872A45D8

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$$\underbrace{000000000000000000}_{\text{18 zeros}}\text{cf3620d570d08d1799a1cafbbfae512fdba2124665eca0}$$

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDBF64F2DF1B0324

SHA256(V,P,M,T,C,1) =
DF64342507E785FDC0D4C776D7142BB2BC6467F09E0040A3E9F65E38872

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$\underbrace{000000000000000000}_{\text{18 zeros}}$cf3620d570d08d1799a1cafbbfae512fdba2124665eca0

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDBF64F2DF1B0324

SHA256(V,P,M,T,C,1) =
DF64342507E785FDC0D4C776D7142BB2BC6467F09E0040A3E9F65E38872

SHA256(V,P,M,T,C,2) =
0000000CC7F94221B95F4E606E037D31C10417435DEE60A61C627B64324590FE

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$\underbrace{000000000000000000}_{\text{18 zeros}}$cf3620d570d08d1799a1cafbbfae512fdba2124665eca0

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDBF64F2DF1B0324

SHA256(V,P,M,T,C,1) =
DF64342507E785FDC0D4C776D7142BB2BC6467F09E0040A3E9F65E38872

SHA256(V,P,M,T,C,2) =
$\underbrace{0000000}_{\text{7 zeros}}$CC7F94221B95F4E606E037D31C10417435DEE60A61C627B64324

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

$\underbrace{000000000000000000}_{\text{18 zeros}}\text{cf3620d570d08d1799a1cafbbfae512fdba2124665eca0}$

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDBF64F2DF1B0324

SHA256(V,P,M,T,C,1) =
DF64342507E785FDC0D4C776D7142BB2BC6467F09E0040A3E9F65E38872

SHA256(V,P,M,T,C,2) =
$\underbrace{0000000}_{\text{7 zeros}}\text{CC7F94221B95F4E606E037D31C10417435DEE60A61C627B64324}$

.

.

.

SHA256(V,P,M,T,C,01F04A1C) =
$\underbrace{000000000000000000}_{\text{18 zeros}}\text{1E3BFE56AD29732B81128B79356442C8B87F6CED8B6610}$

# Mining Details

- Find a nonce that results in SHA256(block) < Difficulty

| | |
|---|---|
| Version (4B) | 02000000 |
| Previous Block Hash (32B) | 25F947B7C18A1E4E2DF96D0D4368DFC24<br>AA9C4EC8C3D6B51A4C4935409D58FED |
| Merkle Tree Root Hash (32B) | 4E04D109A3A7A0460AD2DFD95A4F0FAA<br>145F3249BEE9F371F8204D16C01D4921 |
| Time Stamp (4B) | 5C9F3E20 |
| Current Target Bits (4B) | 172E6117 |
| Nonce (4B) | |
| | $TX_{reward}$<br>$TX_1$<br>.<br>.<br>$TX_n$ |

Difficulty is a function of Current Target Bits (Largest possible Target/Current Target)

000000000000000000cf3620d570d08d1799a1cafbbfae512fdba2124665eca0

18 zeros

SHA256(V,P,M,T,C,0) =
BD72804EE251889F9013C100767999B57E92EC5B6ADBDBF64F2DF1B0324

SHA256(V,P,M,T,C,1) =
DF64342507E785FDC0D4C776D7142BB2BC6467F09E0040A3E9F65E38872

SHA256(V,P,M,T,C,2) =
0000000CC7F94221B95F4E606E037D31C10417435DEE60A61C627B64324

7 zeros

.

.

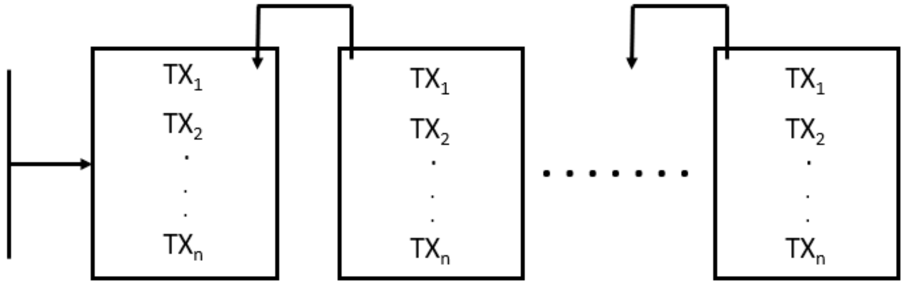
.

SHA256(V,P,M,T,C,01F04A1C) =
0000000000000000001E3BFE56AD29732B81128B79356442C8B87F6CED8

18 zeros

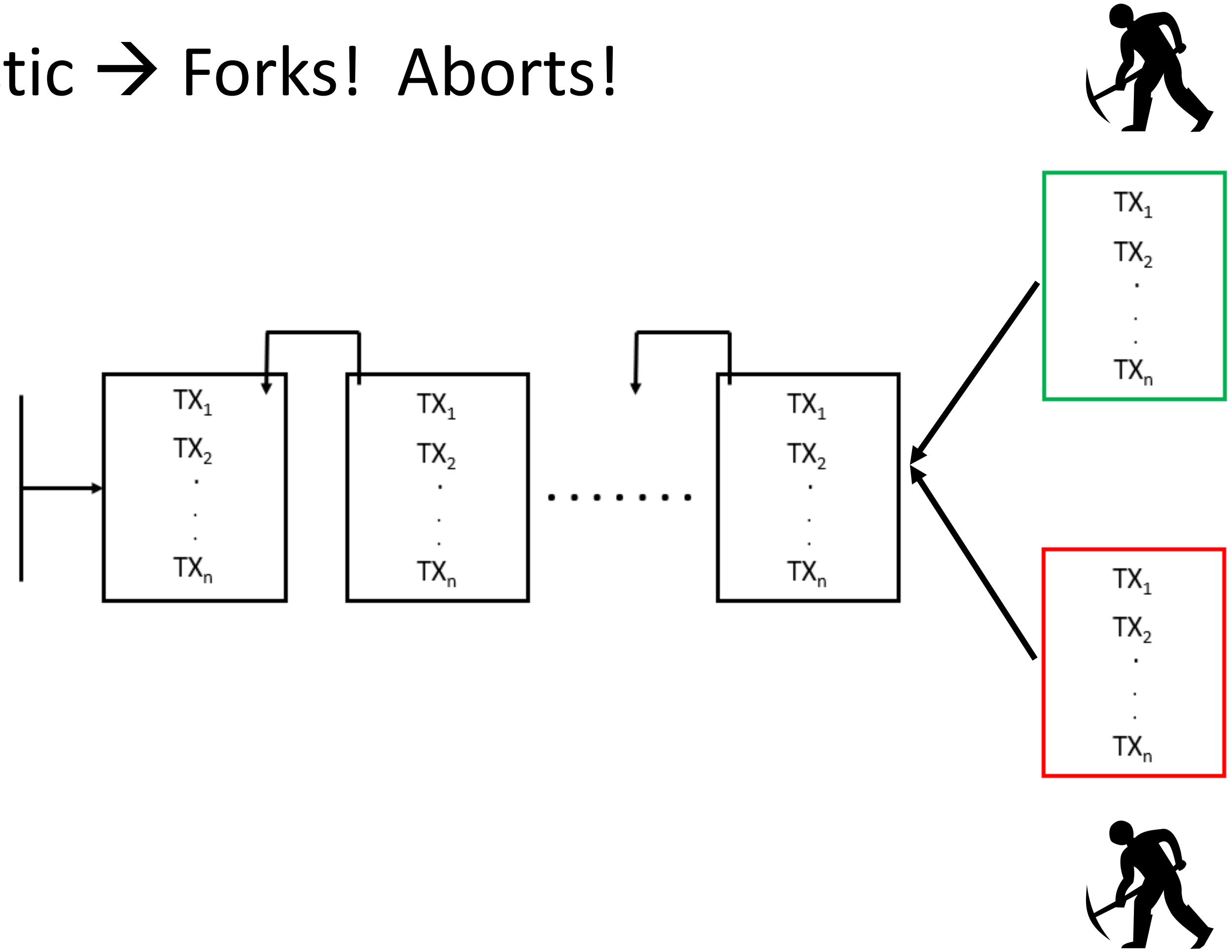
# Bitcoin Forks

- Mining is probabilistic → Forks! Aborts!



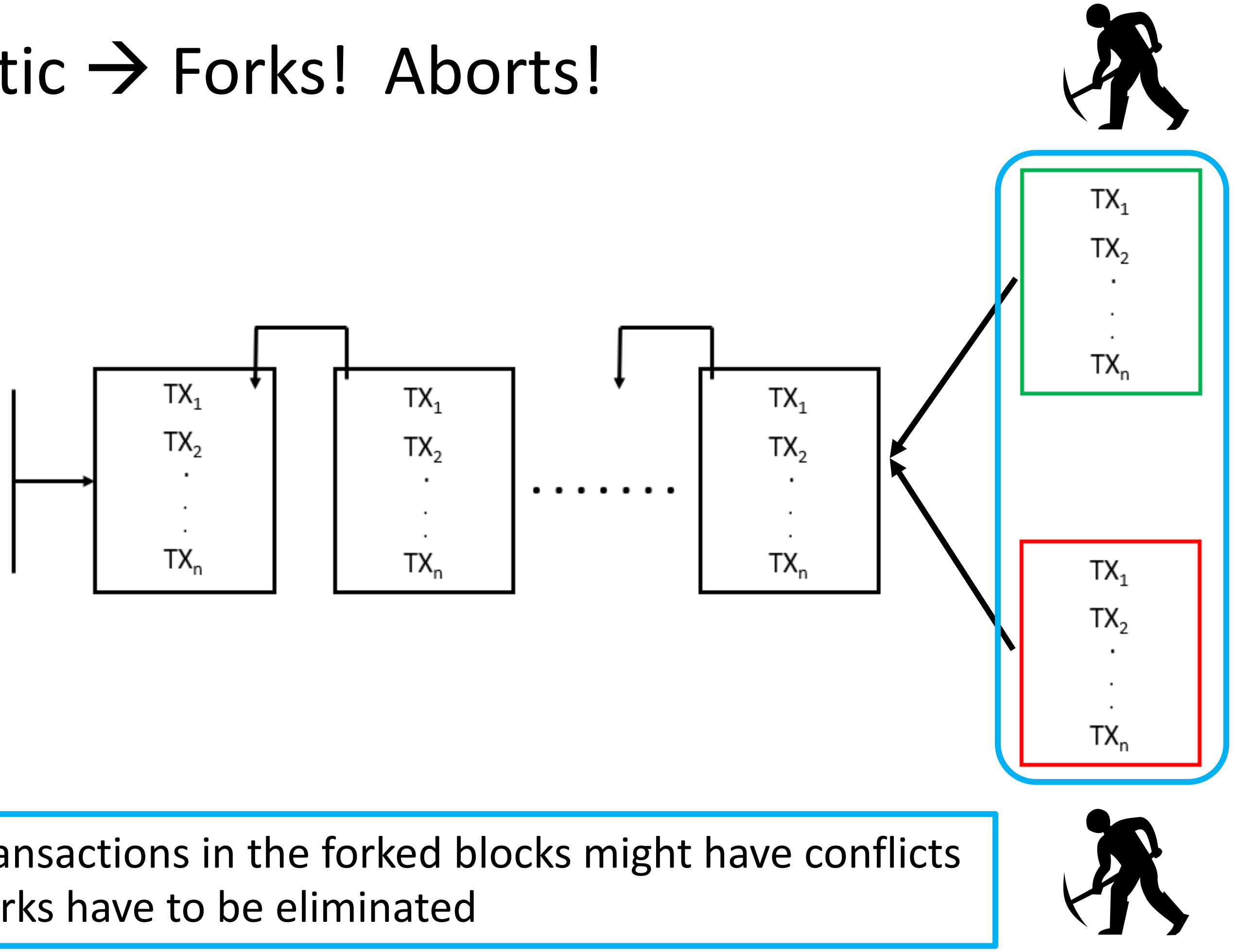
# Bitcoin Forks

- Mining is probabilistic → Forks! Aborts!



# Bitcoin Forks

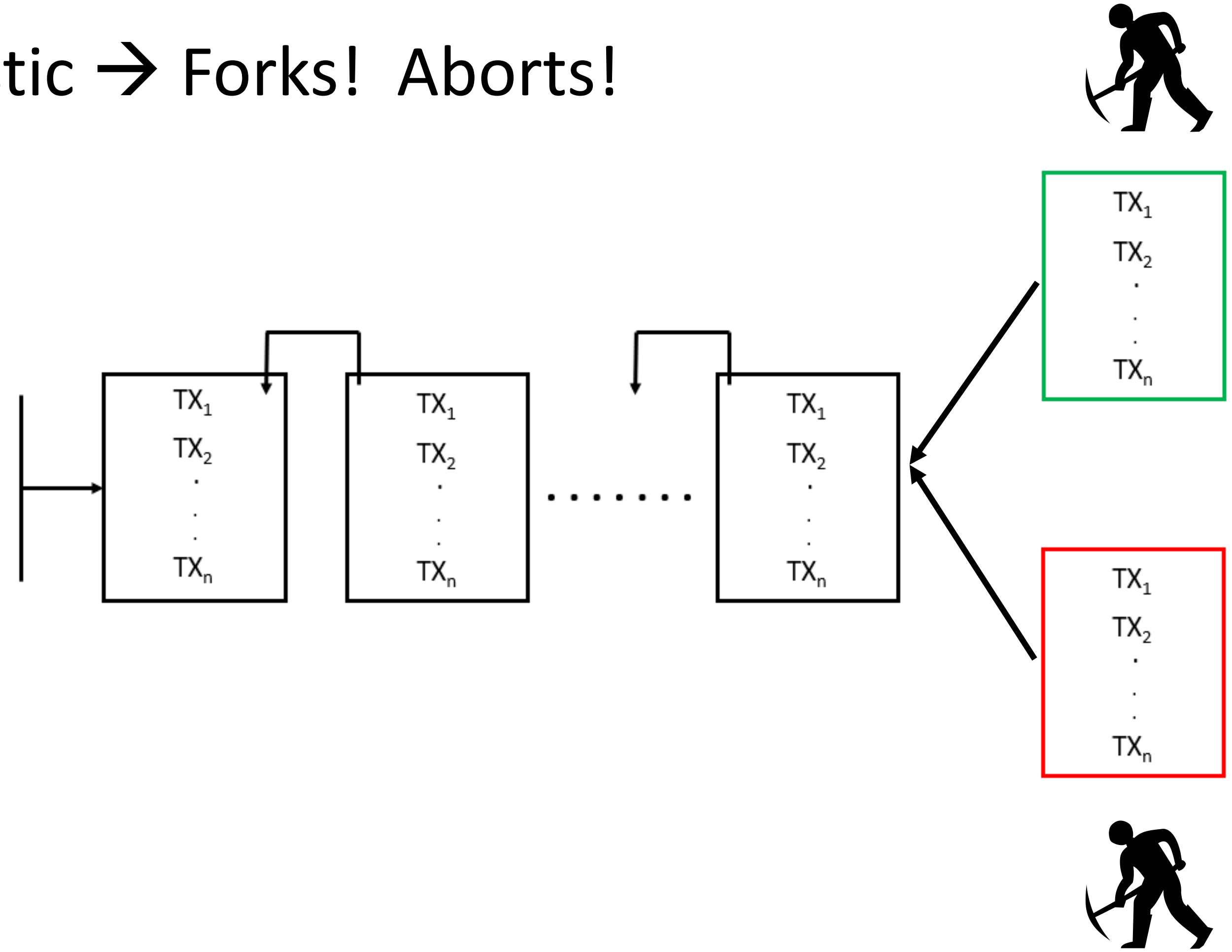
- Mining is probabilistic → Forks! Aborts!



- Transactions in the forked blocks might have conflicts
- Forks have to be eliminated

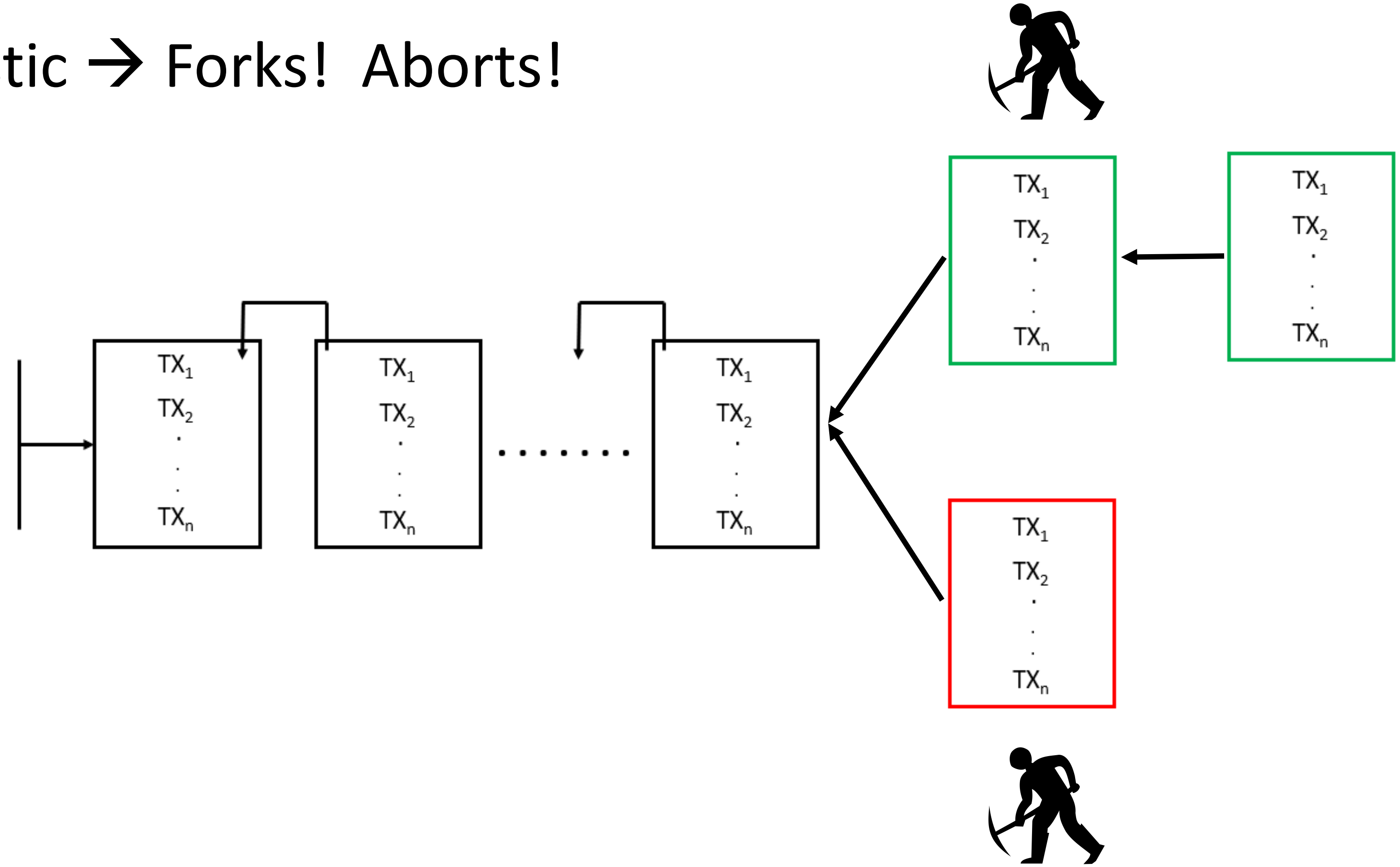
# Bitcoin Forks

- Mining is probabilistic → Forks! Aborts!



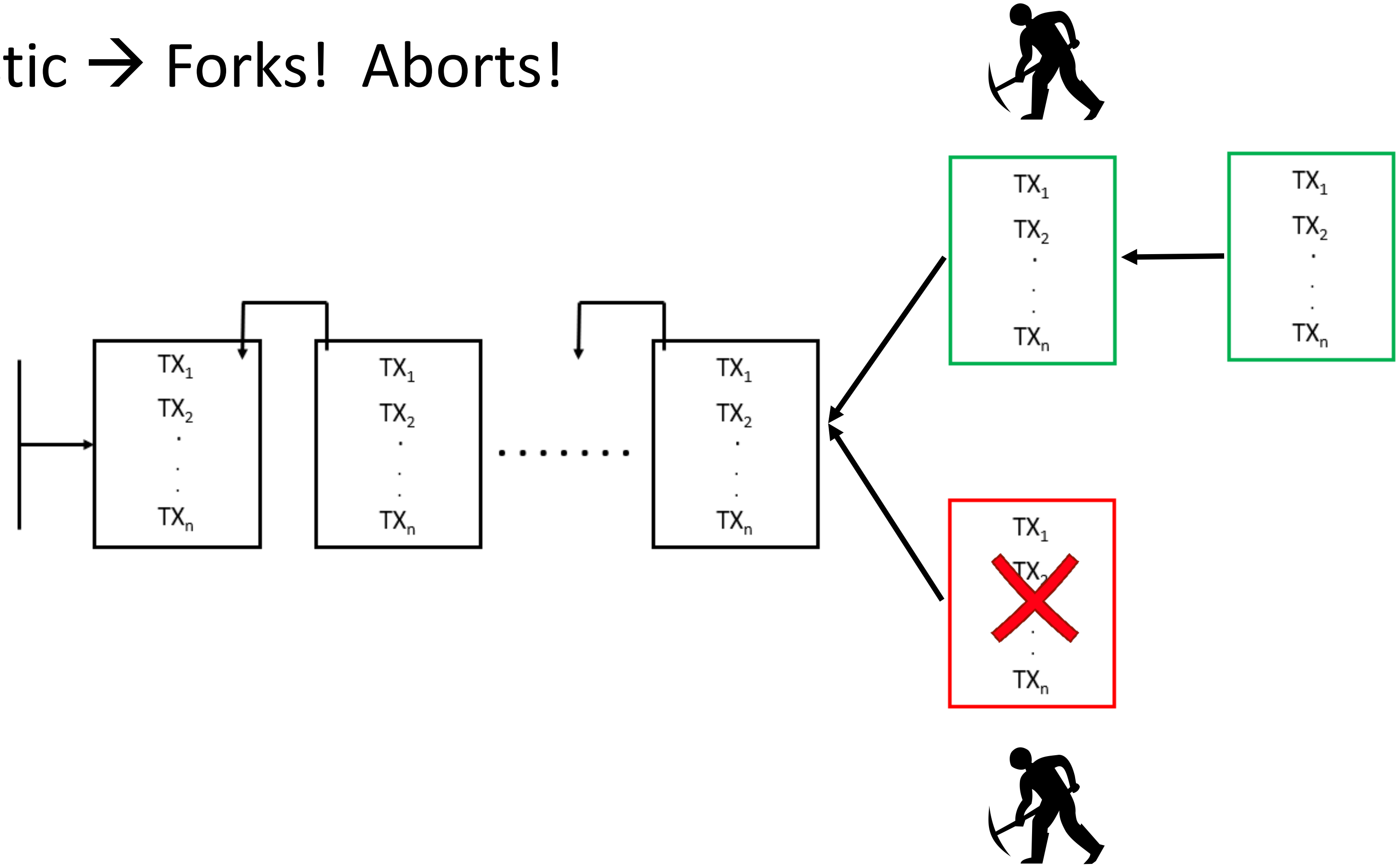
# Bitcoin Forks

- Mining is probabilistic → Forks! Aborts!



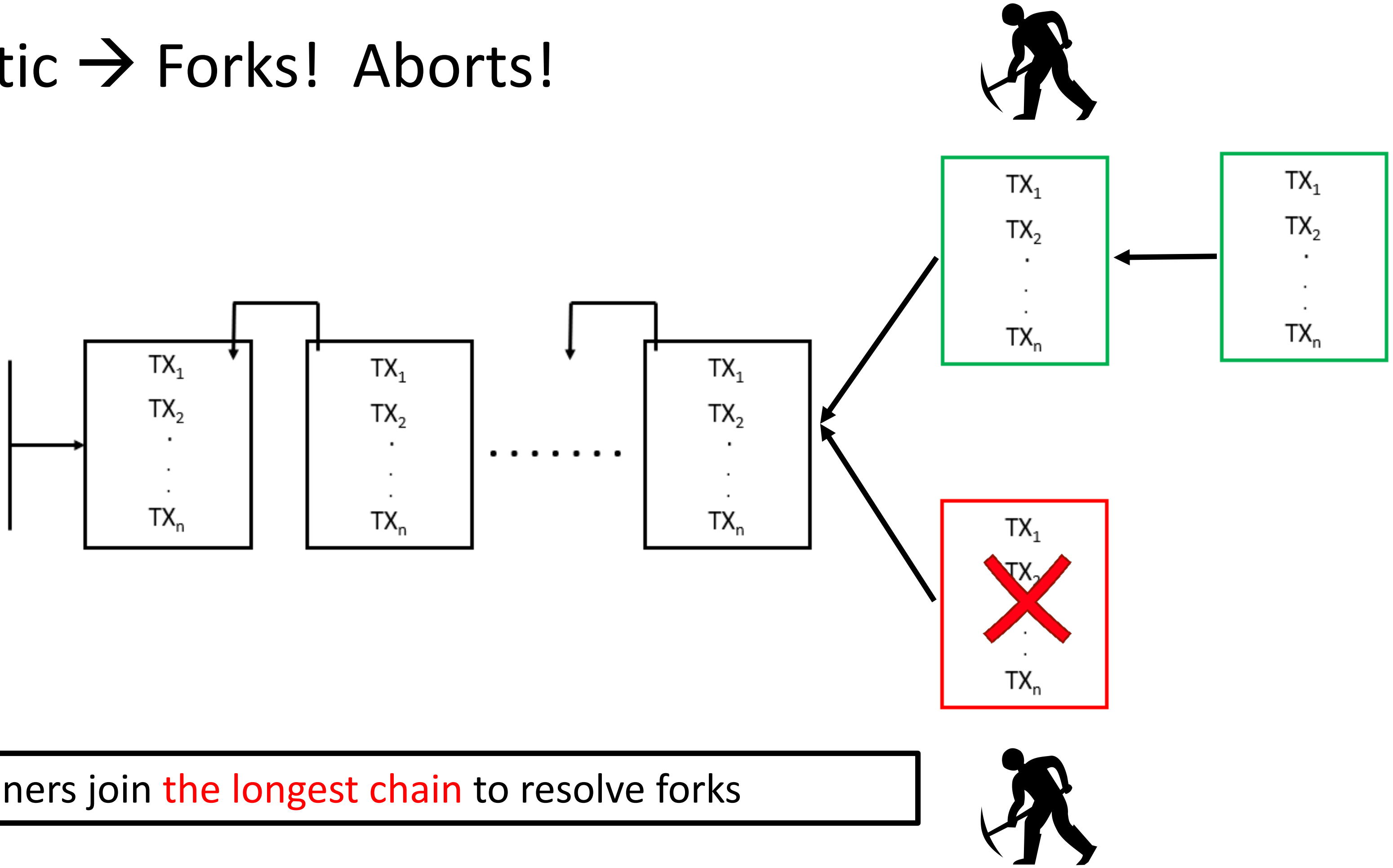
# Bitcoin Forks

- Mining is probabilistic → Forks! Aborts!



# Bitcoin Forks

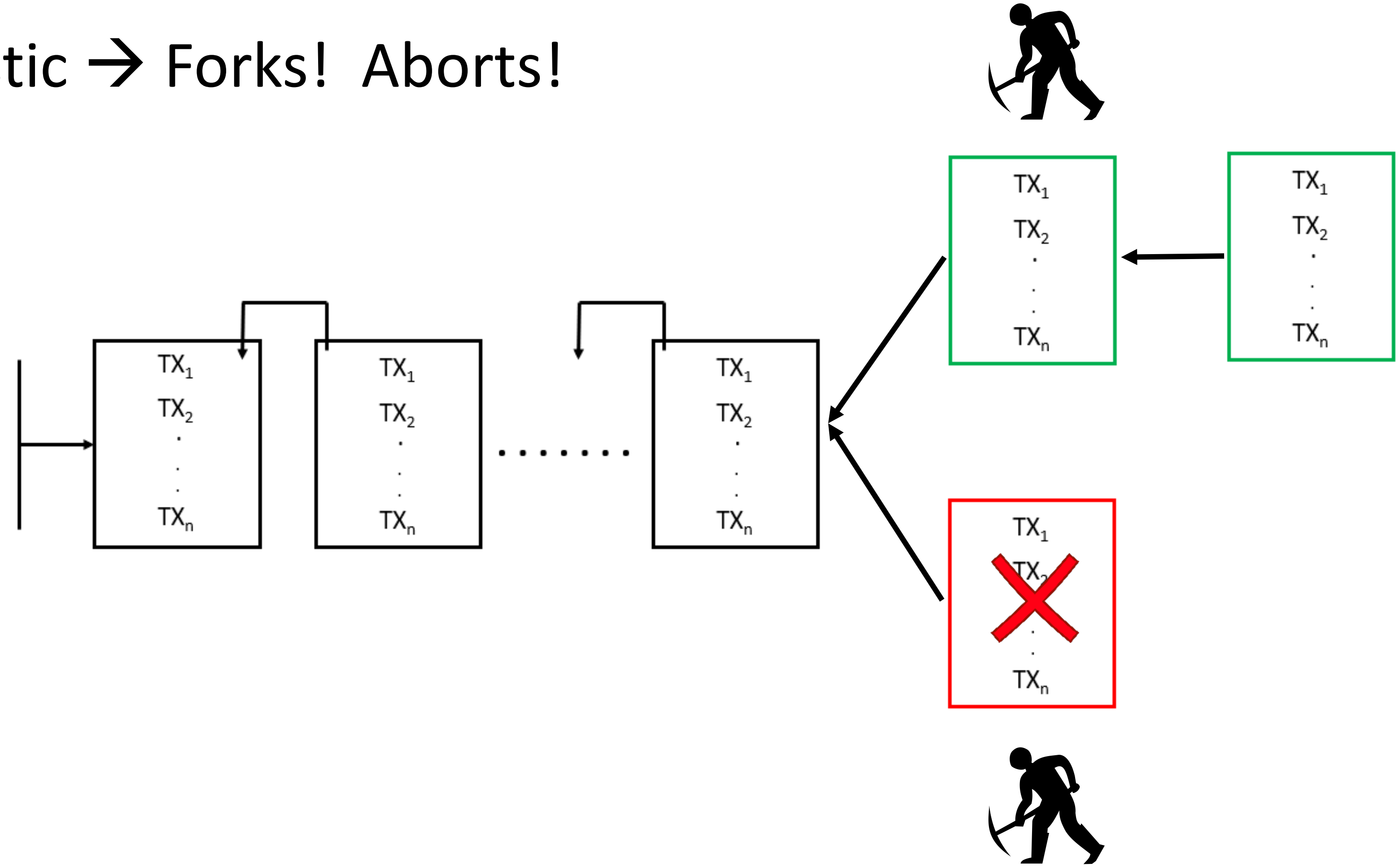
- Mining is probabilistic → Forks! Aborts!



- Miners join the longest chain to resolve forks

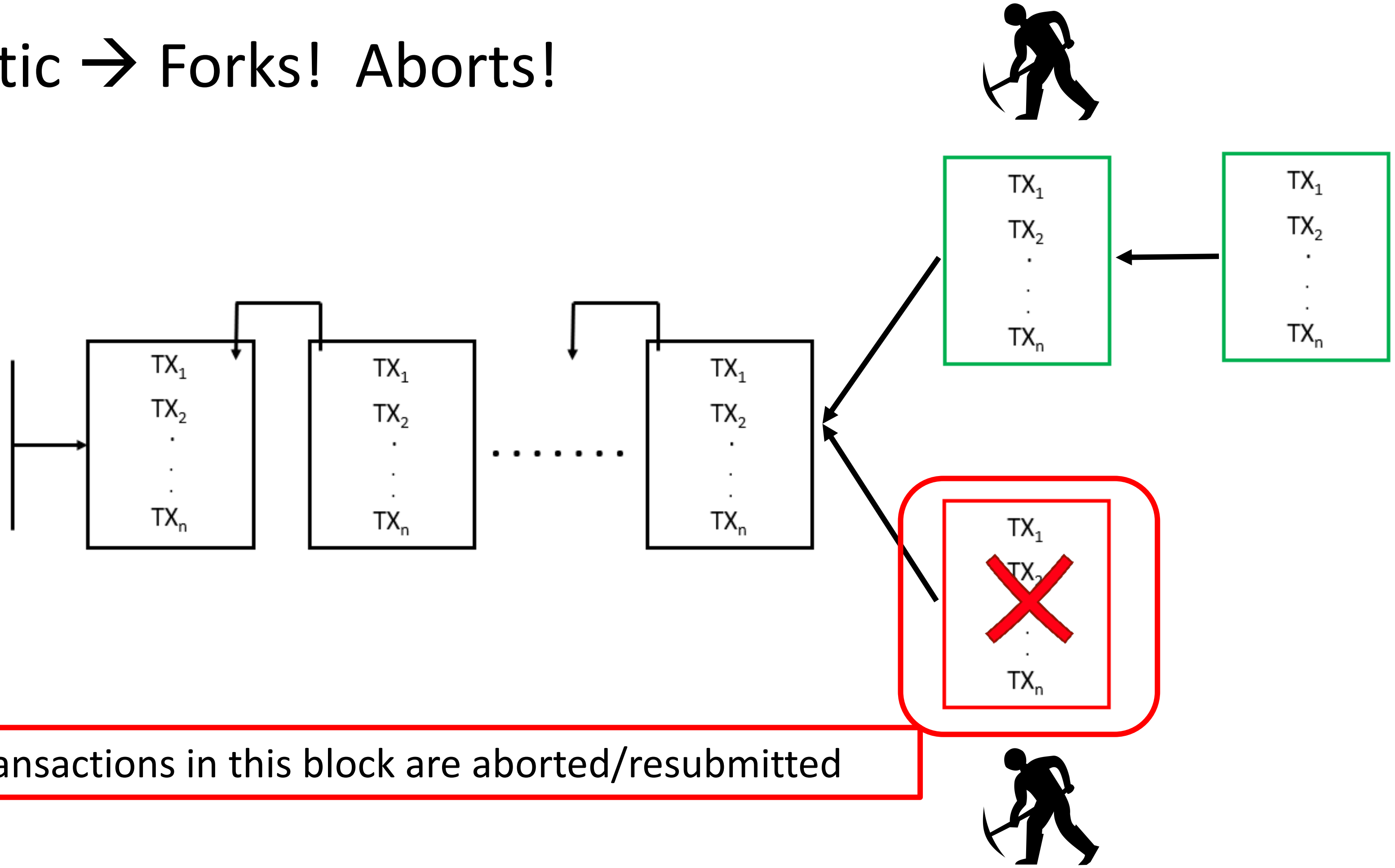
# Bitcoin Forks

- Mining is probabilistic → Forks! Aborts!



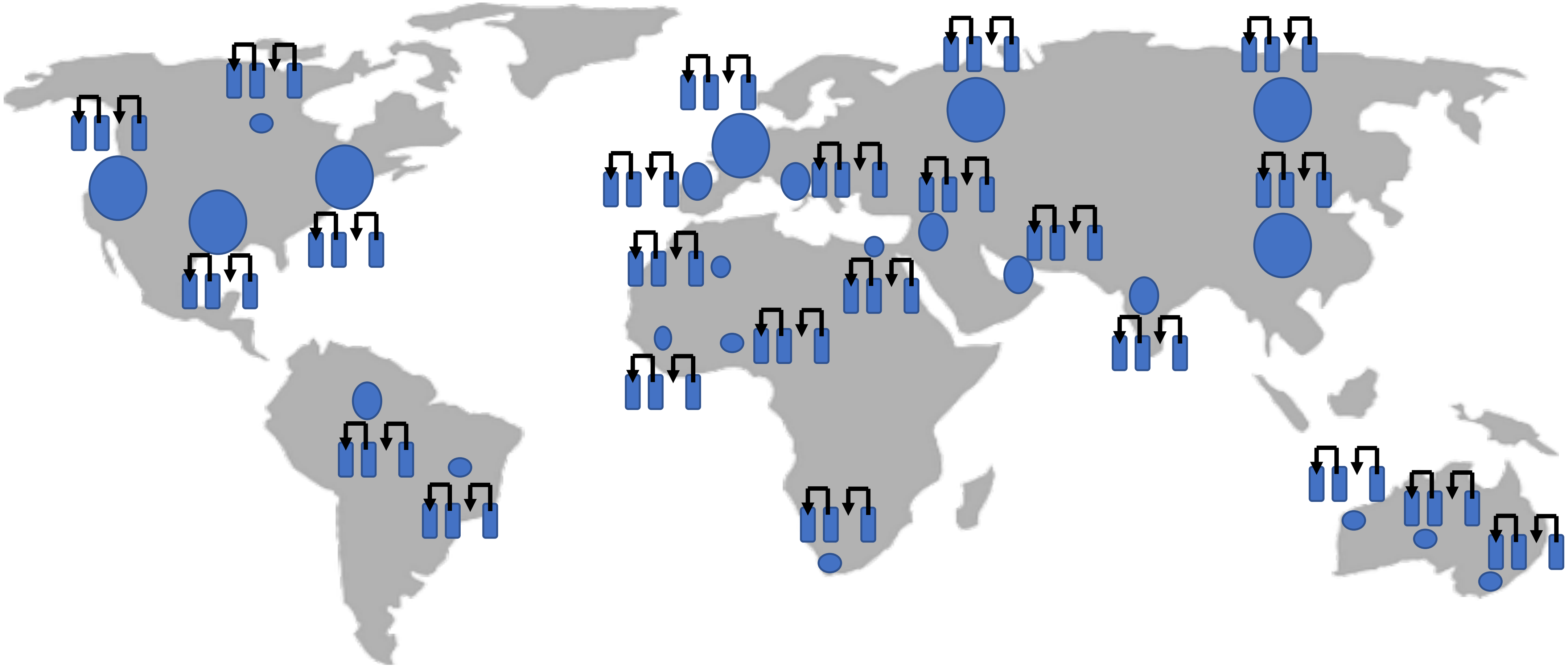
# Bitcoin Forks

- Mining is probabilistic → Forks! Aborts!

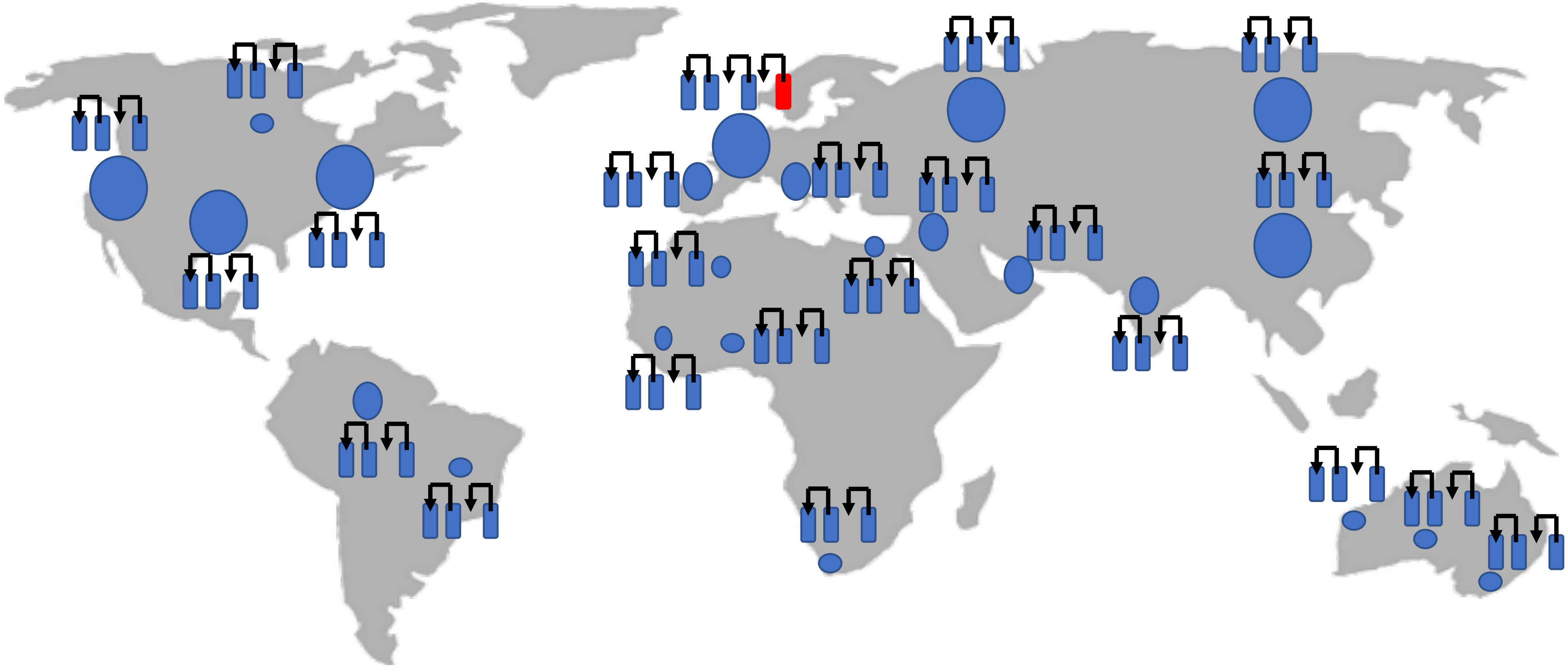


- Transactions in this block are aborted/resubmitted

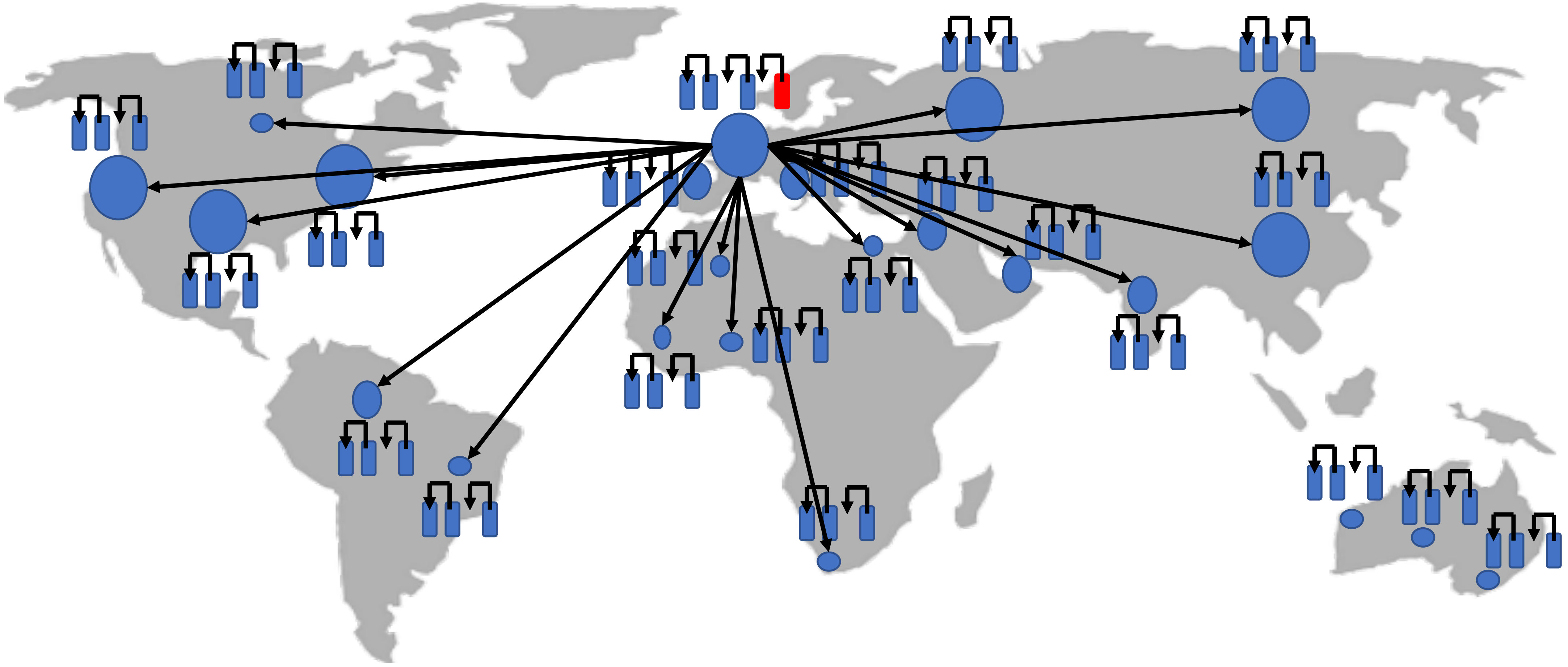
# Mining Big Picture



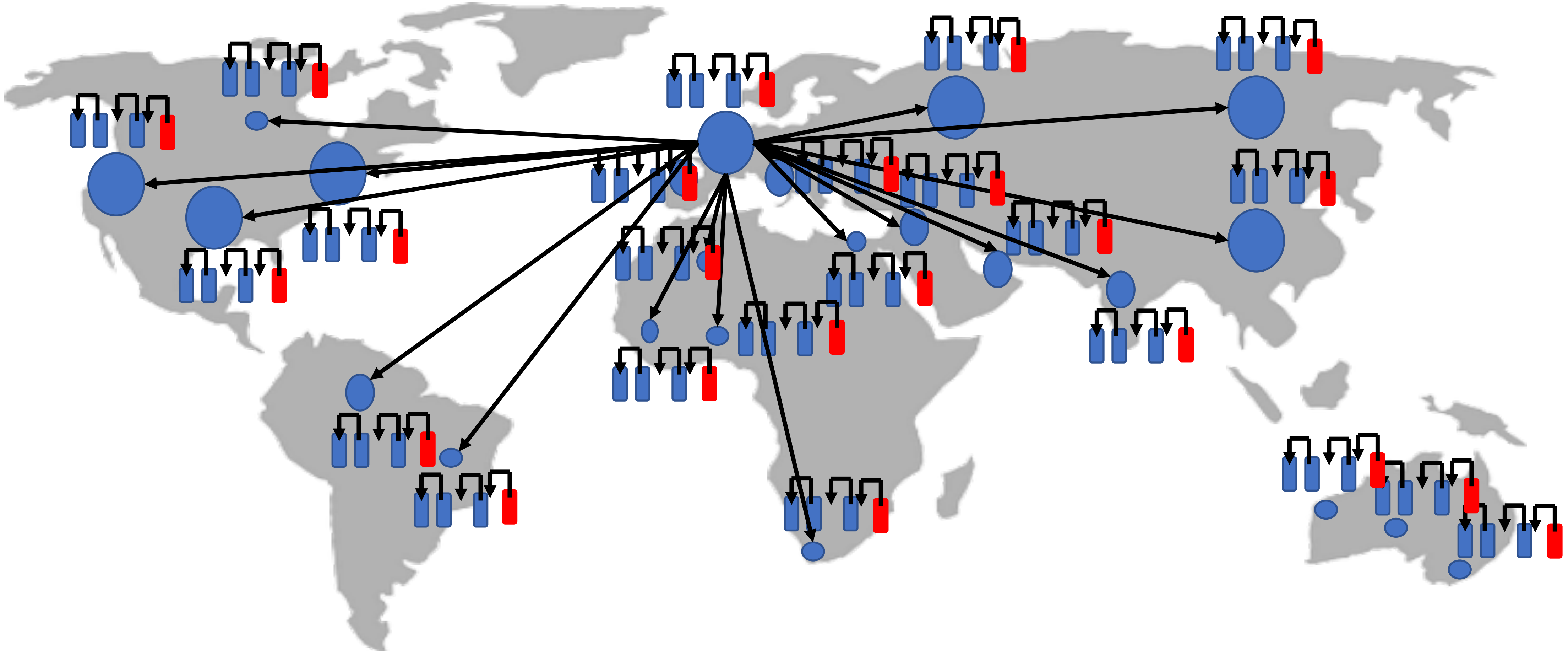
# Mining Big Picture



# Mining Big Picture

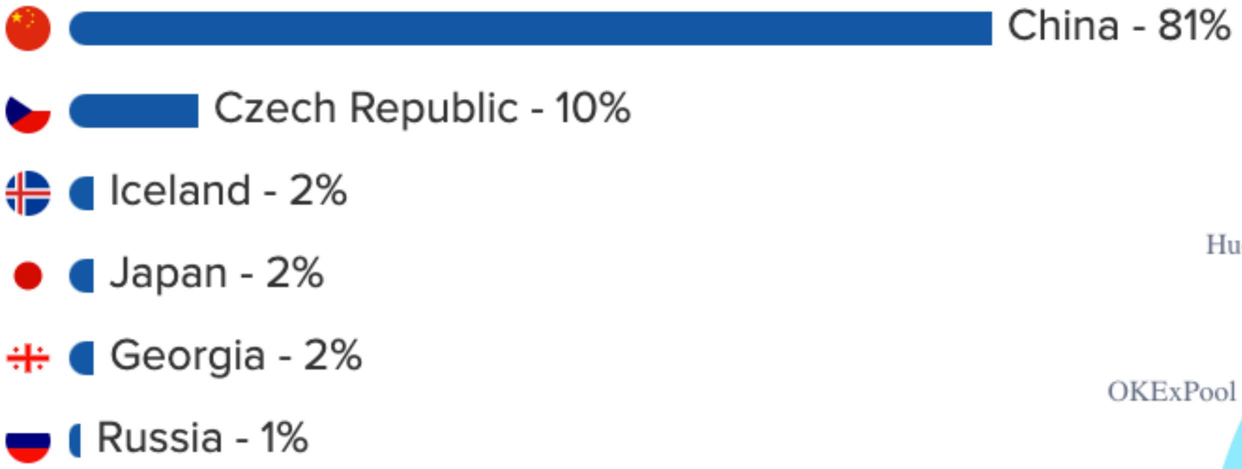


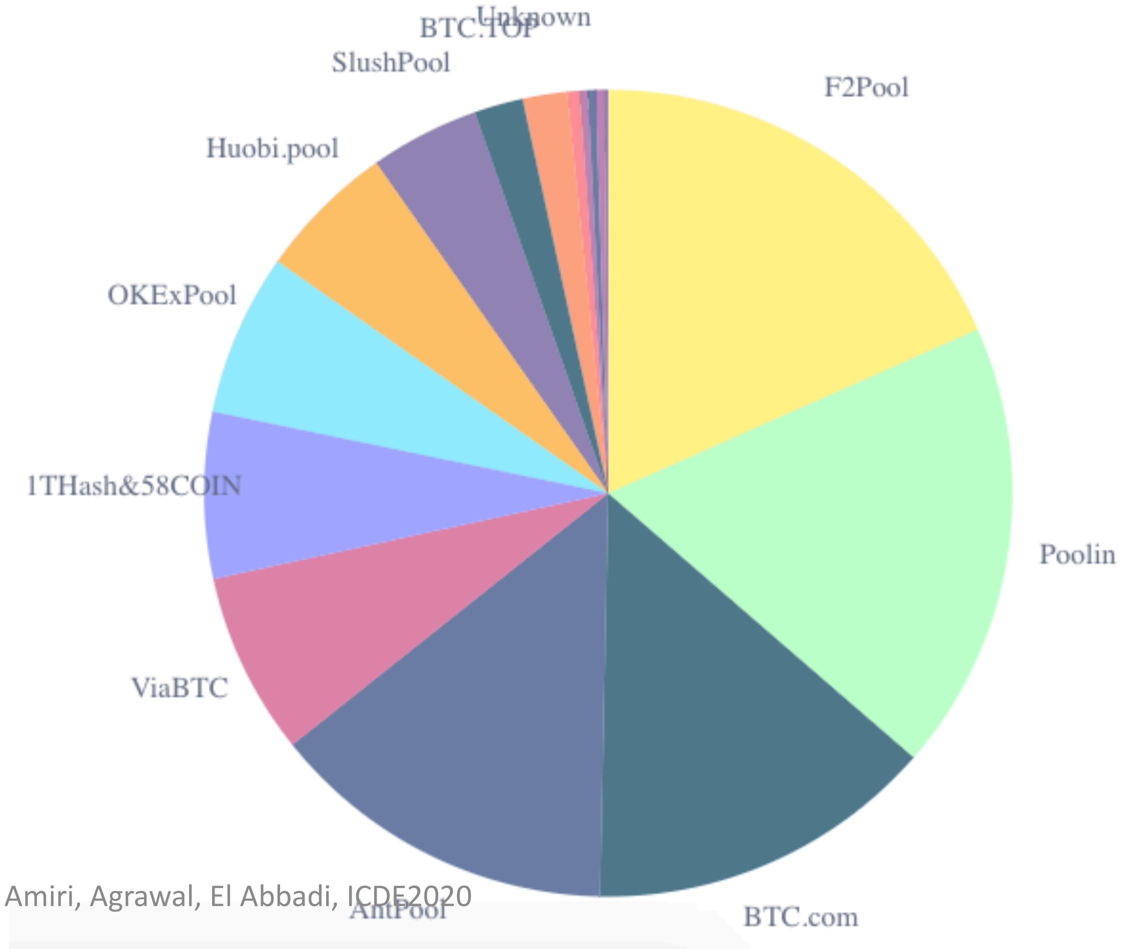
# Mining Big Picture



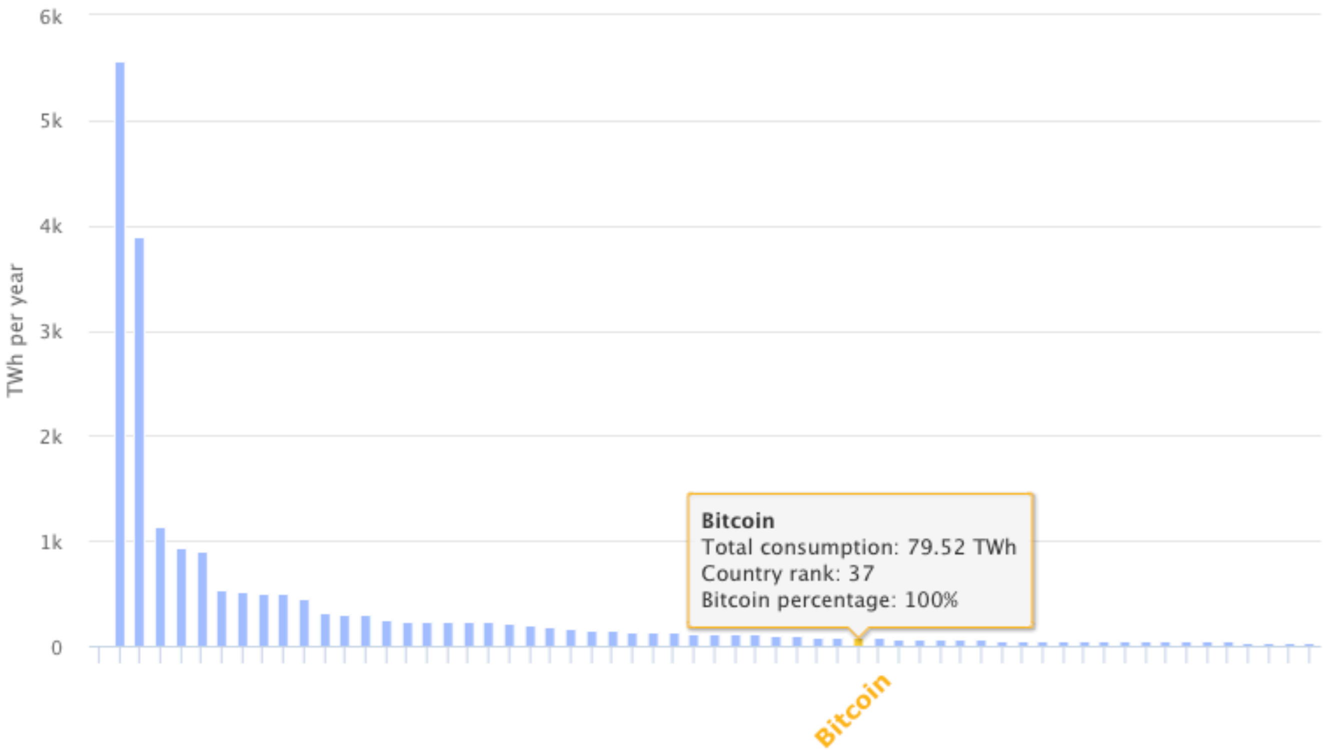
# First Issue: Mining Centralization

- Chinese pools control ~81% of the network hash rate





# Second Issue: PoW consumes lots of electricity



| Chile | Philippines | Bitcoin | Belgium | Finland |
|---|---|---|---|---|
| 73.22<br>TWh per year | 78.30<br>TWh per year | 79.52<br>TWh per year | 82.16<br>TWh per year | 82.79<br>TWh per year |

The amount of electricity used annually by the Bitcoin network could **satisfy the energy needs** of the University of Cambridge for ...

**442 years**

The amount of electricity consumed every year by **always-on but inactive home devices in the USA** alone could ...

power the Bitcoin network for

**2.8 years**

The amount of electricity consumed by the Bitcoin network in one year could **power all tea kettles used to boil water** for ...

United Kingdom

**17 years**

Europe (incl. UK)

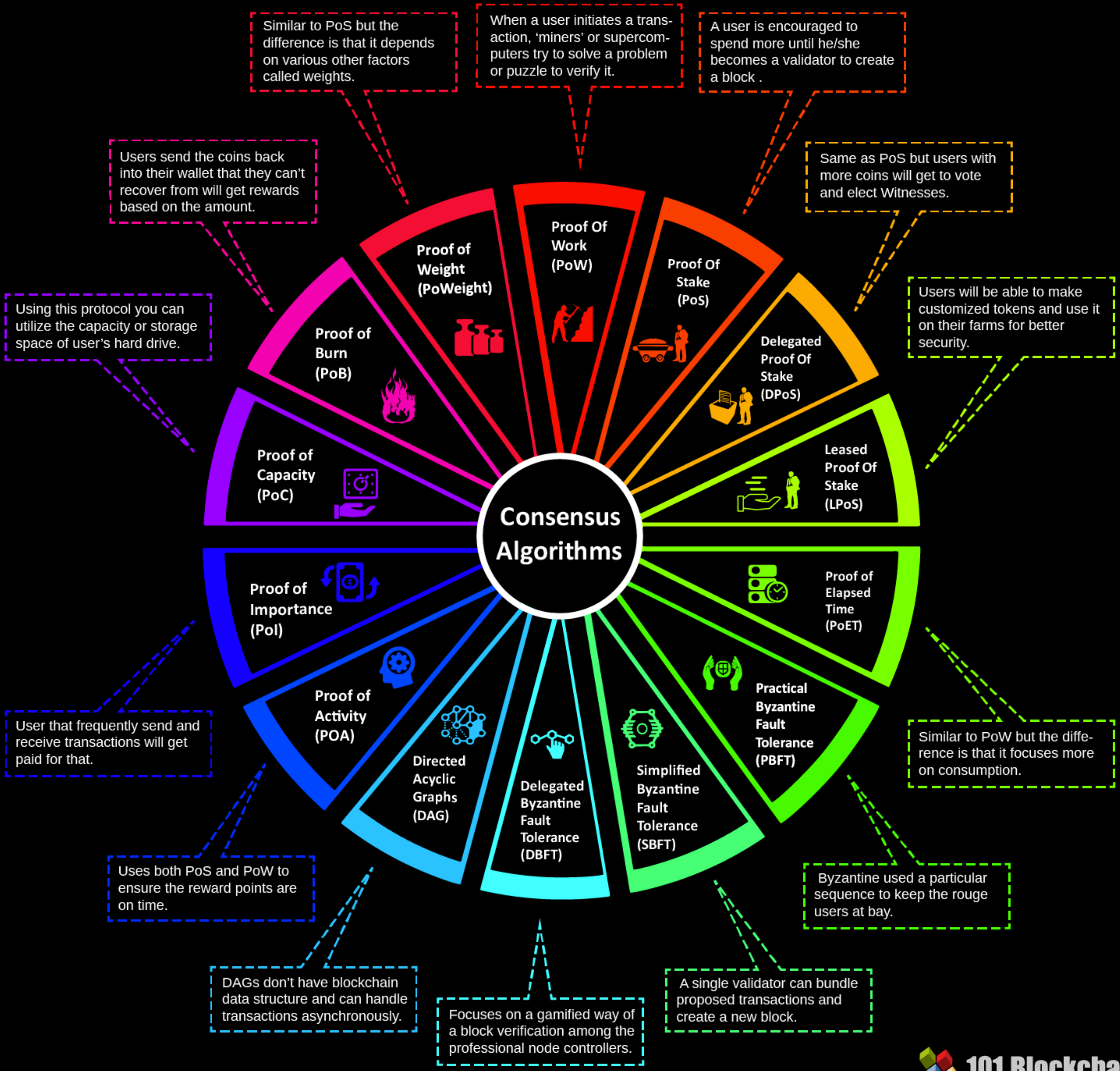
**2.6 years**

# Other Issues

Weak finality guarantees

Selfish mining and other attacks

Suboptimal light client support



# Consensus Algorithms

**Proof Of Work (PoW)**
When a user initiates a transaction, 'miners' or supercomputers try to solve a problem or puzzle to verify it.

**Proof Of Stake (PoS)**
A user is encouraged to spend more until he/she becomes a validator to create a block .

**Delegated Proof Of Stake (DPoS)**
Same as PoS but users with more coins will get to vote and elect Witnesses.

**Leased Proof Of Stake (LPoS)**
Users will be able to make customized tokens and use it on their farms for better security.

**Proof of Elapsed Time (PoET)**
Similar to PoW but the difference is that it focuses more on consumption.

**Practical Byzantine Fault Tolerance (PBFT)**
Byzantine used a particular sequence to keep the rouge users at bay.

**Simplified Byzantine Fault Tolerance (SBFT)**
A single validator can bundle proposed transactions and create a new block.

**Delegated Byzantine Fault Tolerance (DBFT)**
Focuses on a gamified way of a block verification among the professional node controllers.

**Directed Acyclic Graphs (DAG)**
DAGs don't have blockchain data structure and can handle transactions asynchronously.

**Proof of Activity (POA)**
Uses both PoS and PoW to ensure the reward points are on time.

**Proof of Importance (PoI)**
User that frequently send and receive transactions will get paid for that.

**Proof of Capacity (PoC)**
Using this protocol you can utilize the capacity or storage space of user's hard drive.

**Proof of Burn (PoB)**
Users send the coins back into their wallet that they can't recover from will get rewards based on the amount.

**Proof of Weight (PoWeight)**
Similar to PoS but the difference is that it depends on various other factors called weights.

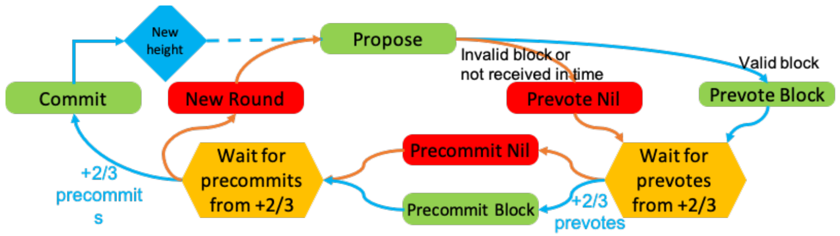
101 Blockchains
Created by 101blockchains.com

# PROOF OF STAKE

# Proof of Stake

- A stakeholder who has *p* fraction of the coins in circulation creates a new block with *p* probability
- Don’t the rich get richer?
  - Randomized block selection
    - Combination of a random number and the stake size
  - Coin age-based selection
    - The number of coins * the number of days the coins have been held.
    - Coins that have been unspent for at least 30 days begin competing for the next block.
    - Older and larger sets of coins have a greater probability of signing the next block.
    - The probability of finding the next block reaches a maximum after 90 days

**Tendermint**



- Has its own consensus protocol
- Extends PBFT with leader rotation

**libra**

- LibraBFT
- A variant of HotStuff

**IBM**

**HYPERLEDGER**

- Pluggable consensus protocols
- Can use PBFT, Paxos, etc.
- Default: Raft

**MultiChain**

- Uses PBFT
- Incorporates leader rotation

**Quorum**

- Introduced by JP Morgan
- A Raft-based consensus
- A PBFT-like called Istanbul BFT

Caper
FastFabric
ResilientDB
Corda
AHL
ParBlockchain
SharPer
Cosmos

# THANK YOU!

## Questions?