Stellar

# Federated Byzantine Agreement

Giuliano Losa,
Researcher, Stellar Development Foundation
SPTDC 2026

April 2026

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

I promise to exchange any GGOLD token for a gold bar if asked

real gold bar

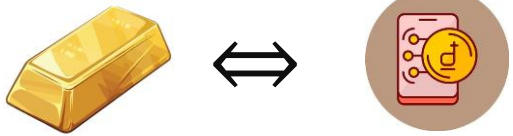
⟺

digital GGOLD token

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

I promise to exchange any GGOLD token for a gold bar if asked



real gold bar

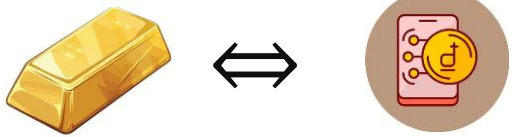
digital GGOLD token

Problem

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

I promise to exchange any GGOLD token for a gold bar if asked



real gold bar

digital GGOLD token

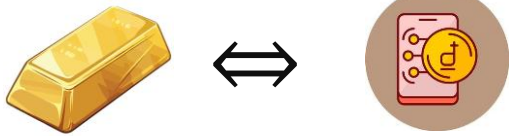
Problem

- Somebody redeems a (real) gold bar

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

I promise to exchange any GGOLD token for a gold bar if asked



real gold bar digital GGOLD token

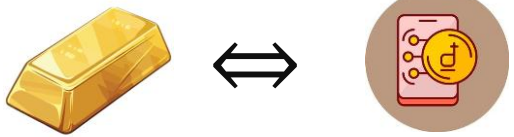
Problem

- Somebody redeems a (real) gold bar
- Stakers decide to revert transactions to an earlier point (e.g., DAO hack)

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

I promise to exchange any GGOLD token for a gold bar if asked



real gold bar digital GGOLD token

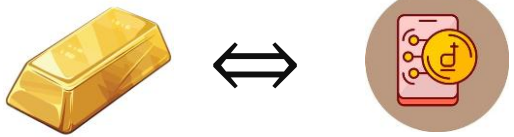
Problem

- Somebody redeems a (real) gold bar
- Stakers decide to revert transactions to an earlier point (e.g., DAO hack)
- I cannot take the (real) gold bar back

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

I promise to exchange any GGOLD token for a gold bar if asked



real gold bar digital GGOLD token

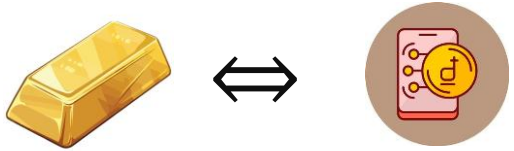
Problem

- Somebody redeems a (real) gold bar
- Stakers decide to revert transactions to an earlier point (e.g., DAO hack)
- I cannot take the (real) gold bar back
- Now I have one less gold bar than GGOLD tokens in circulation

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

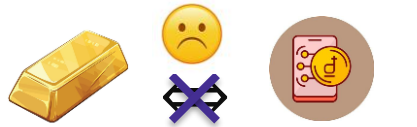
I promise to exchange any GGOLD token for a gold bar if asked



real gold bar digital GGOLD token

Problem

- Somebody redeems a (real) gold bar
- Stakers decide to revert transactions to an earlier point (e.g., DAO hack)
- I cannot take the (real) gold bar back
- Now I have one less gold bar than GGOLD tokens in circulation

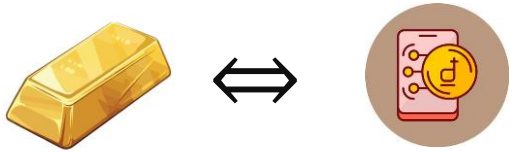


real gold bar digital GGOLD token

# A problem with real-world assets on blockchains

Say I sell gold bars, but I hand out GGOLD tokens on Ethereum instead of the gold bars

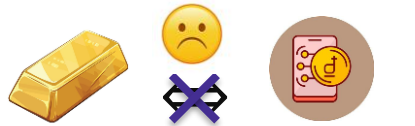
I promise to exchange any GGOLD token for a gold bar if asked



real gold bar digital GGOLD token

Problem

- Somebody redeems a (real) gold bar
- Stakers decide to revert transactions to an earlier point (e.g., DAO hack)
- I cannot take the (real) gold bar back
- Now I have one less gold bar than GGOLD tokens in circulation



real gold bar digital GGOLD token

Stakers have economic incentive not to do that (~USD $100 Billion staked), but what if the assets are worth more?

# A problem with real-world assets on blockchains

Say the central bank of a country wants to issue 10 trillion USD worth of digital currency (a CBDC) on Ethereum

*$10 trillion*

⇔

digital DOLLART token

Total ETH staked is in the order of USD $100 Billion

Now stakers' economic incentives might not be enough to secure the system

# How to safely tokenize 10 trillion USD worth of assets?

We want:

- **Open access:** anyone can issue and trade digital tokens representing real-world assets
- **Cross-issuer atomicity:** tokens from multiple issuers can be exchanged and traded atomically
- **Issuer-enforced finality (IEF):** issuers can prevent transactions in their issued tokens from being reversed or undone

# How to safely tokenize 10 trillion USD worth of assets?

We want:

- **Open access:** anyone can issue and trade digital tokens representing real-world assets
- **Cross-issuer atomicity:** tokens from multiple issuers can be exchanged and traded atomically
- **Issuer-enforced finality (IEF):** issuers can prevent transactions in their issued tokens from being reversed or undone

(Note that this is about protecting issuers from anonymous miners or stakers, and not about protecting users from issuers; the latter is done by courts of law in the real world)

# Can private payment systems do it?

Examples: PayPal, WeChat Pay, Venmo, etc.

- ✓ Have issuer-enforced finality (PayPal can block transactions and controls its ledger)
- ▪ Not open access (cannot issue new tokens)
- ▪ Cross-issuer atomic transactions are impossible (one cannot atomically swap Paypal USD for WeChat Pay CNY)

# Can domestic payment networks do it?

Examples: PIX (Brazil), FedNow (USA), SEPA (EU), etc.

- ✓ Have issuer-enforced finality
- ✓ Have cross-issuer atomic transactions
- ▪ Not open access (limited to one jurisdiction)


# Can public PoW/PoS blockchains do it?

Example: Bitcoin, Ethereum, Solana, Sui, Aptos, etc.

- ✓ Have open access
- ✓ Have cross-issuer atomicity
- ▪ But not issuer-enforced finality: miners or stakers are in control of finality

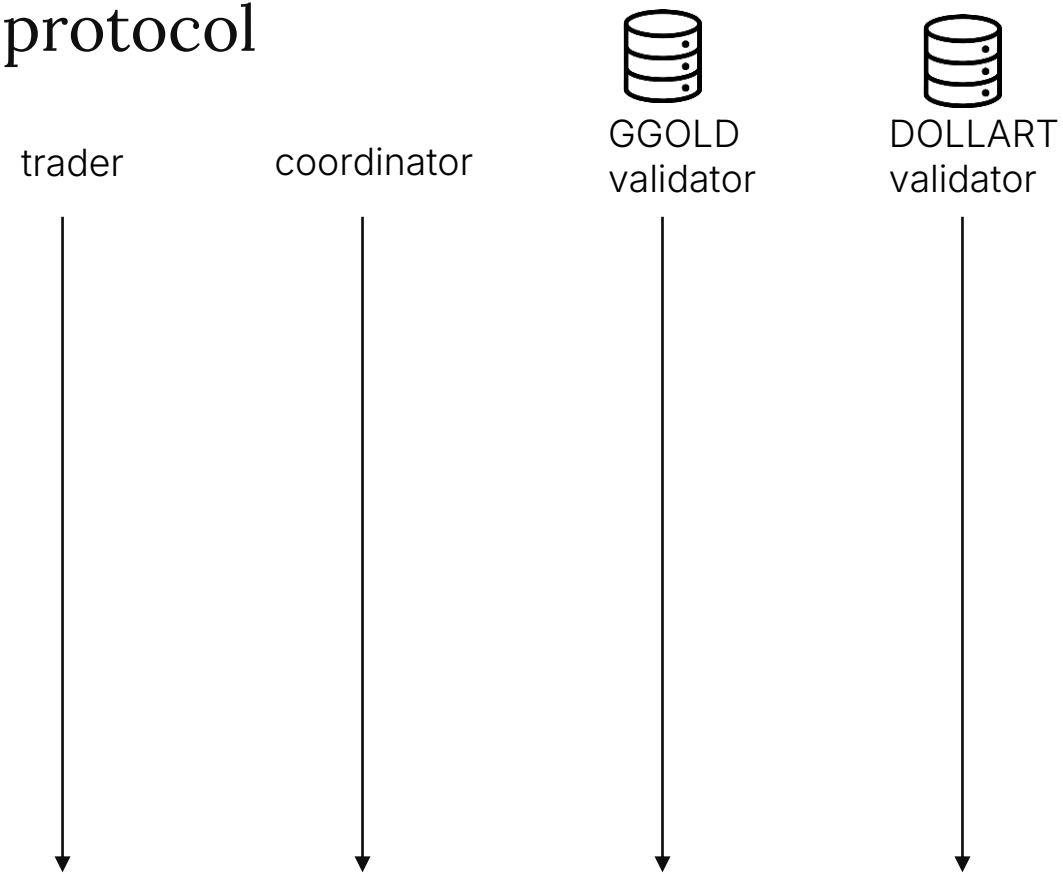
# Idea: let token issuers host their own authoritative ledgers

- I run my own transaction *validator* with the authoritative copy of the GGOLD ledger
- The central bank runs its own *validator* with the authoritative copy of its DOLLART ledger

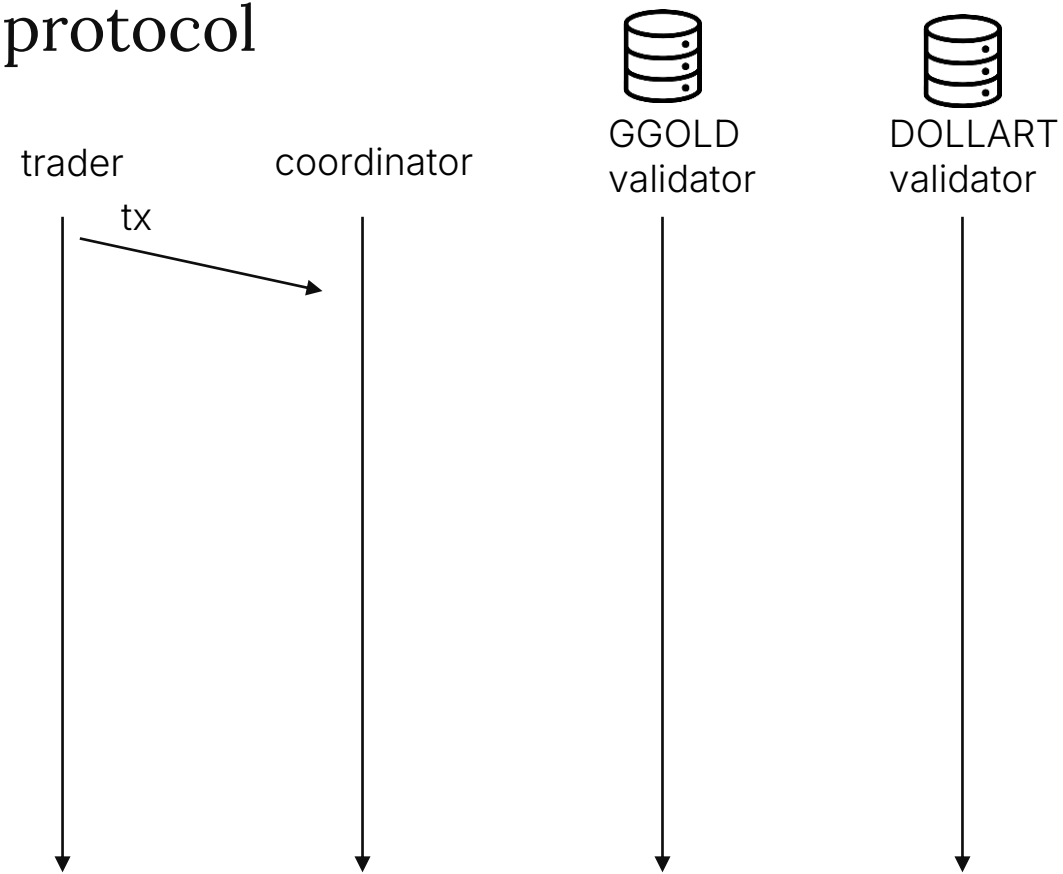
⇒ *We have issuer-enforced finality*

How do we coordinate cross-issuer transactions?

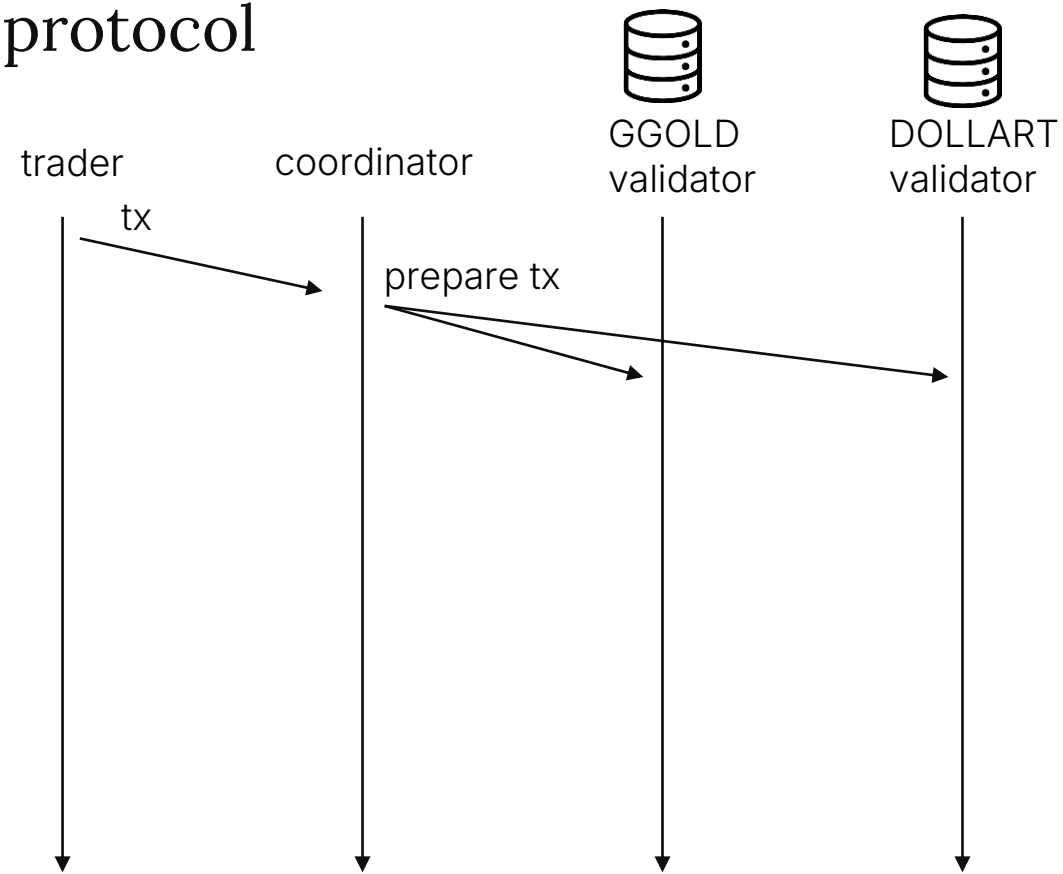
# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



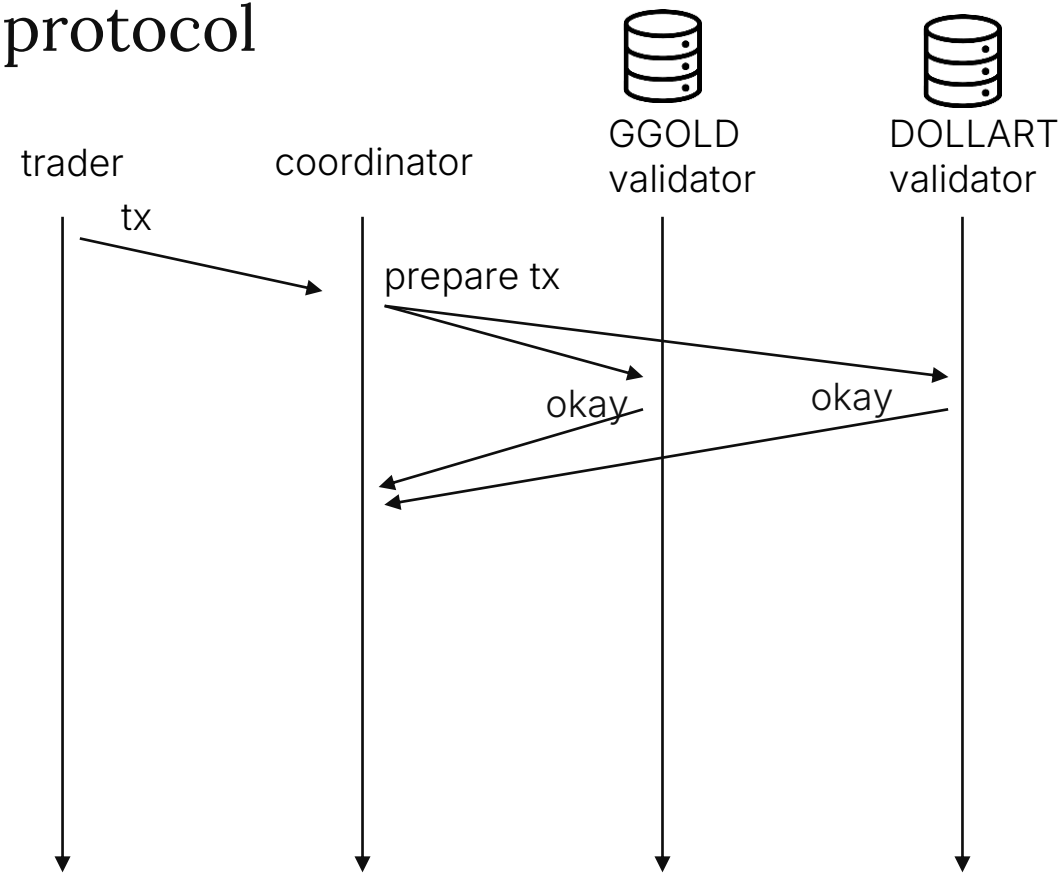
# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



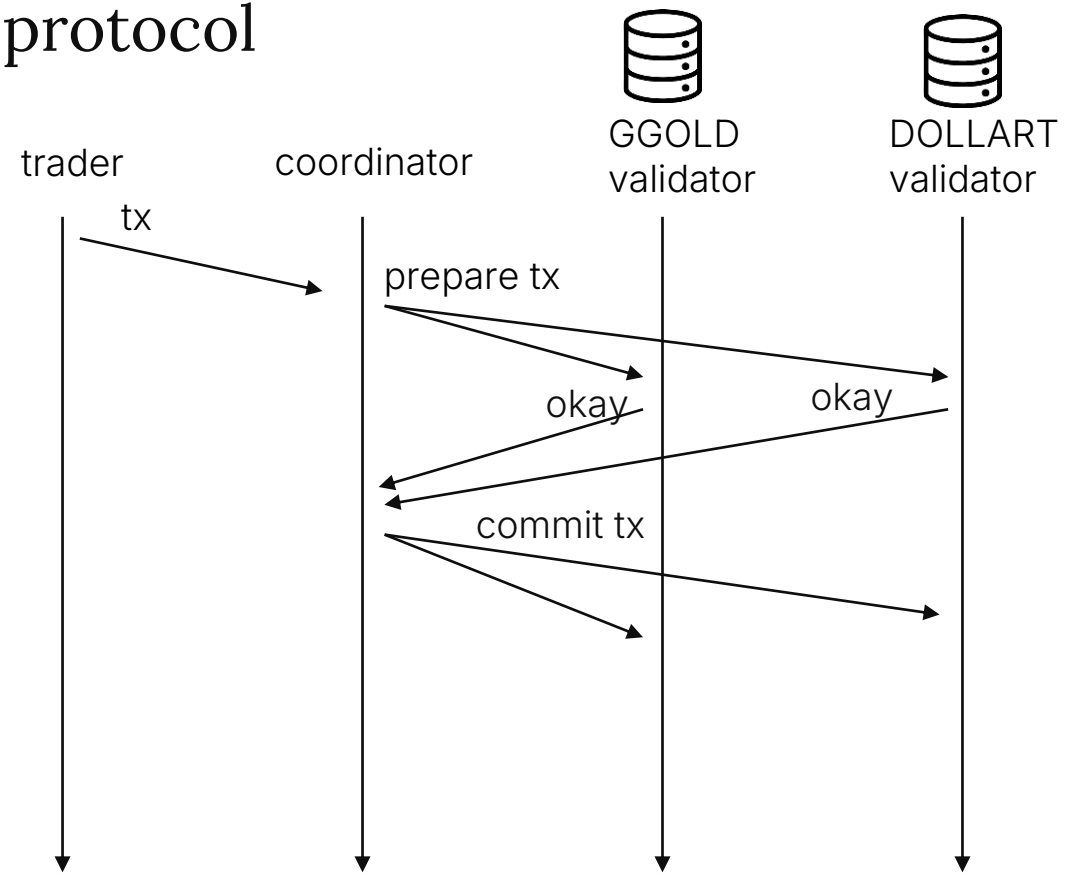
# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



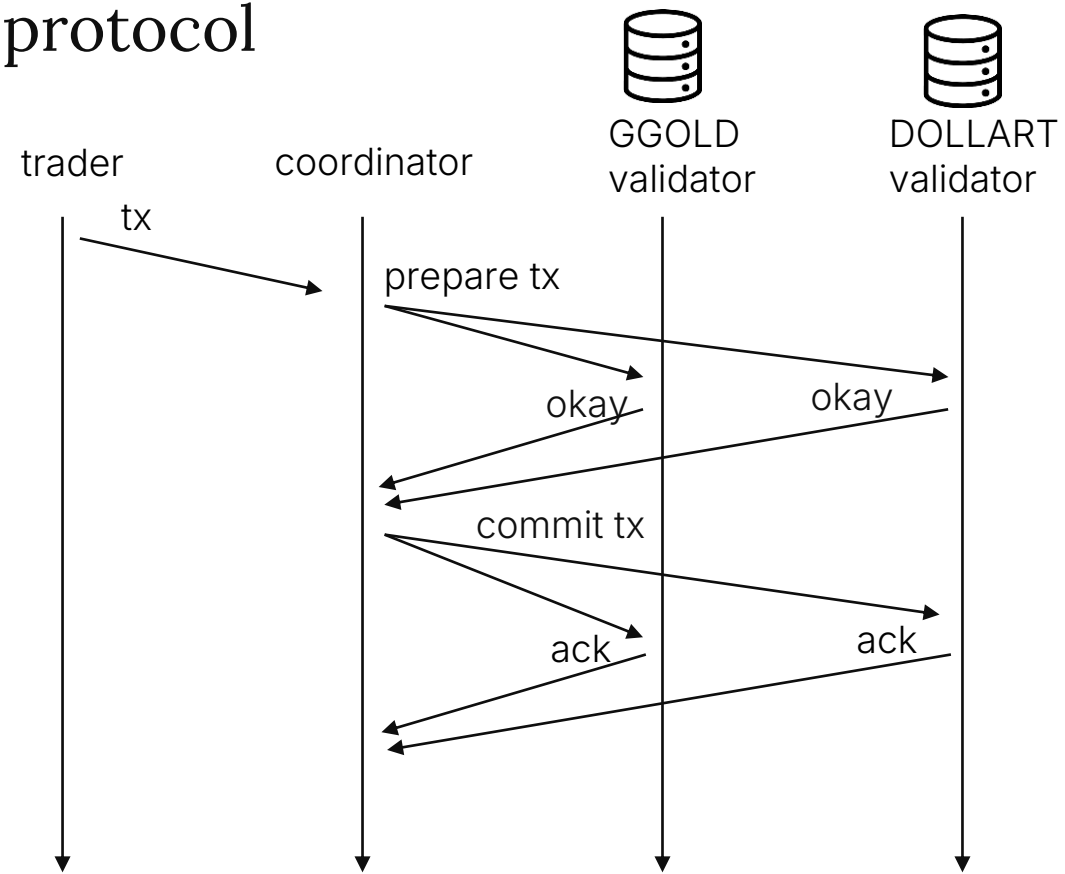
# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



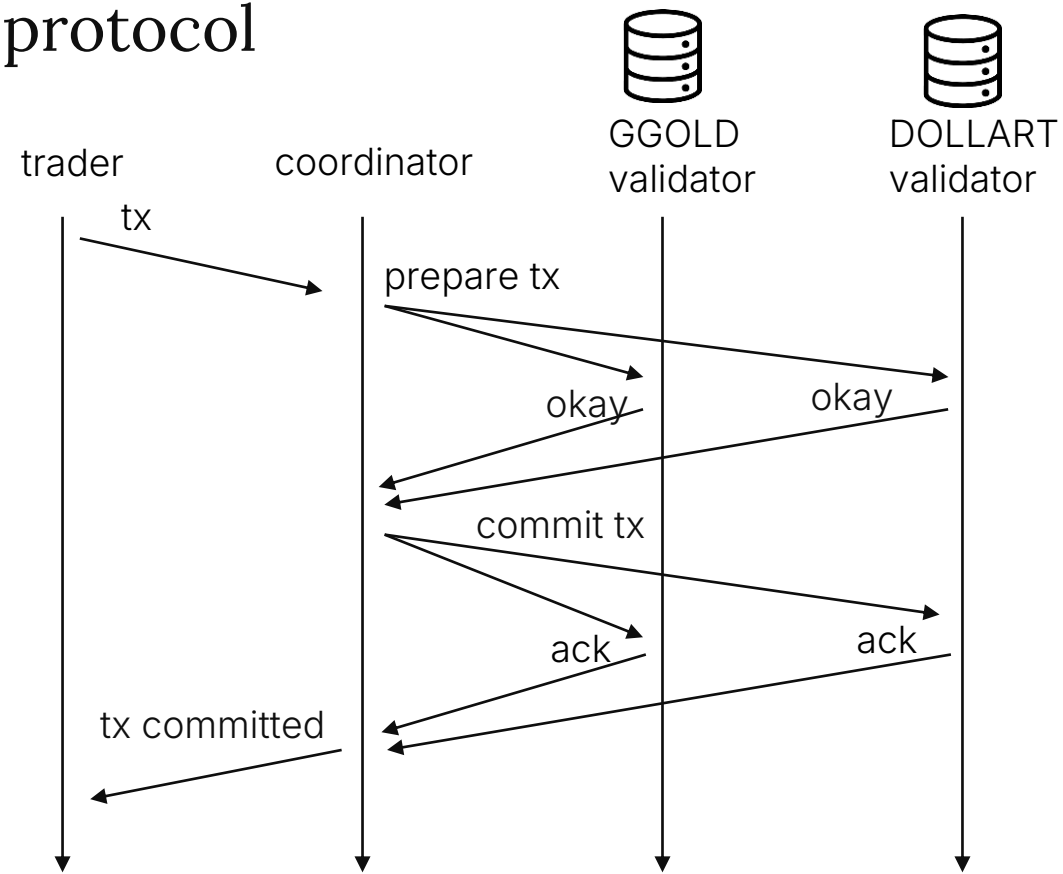
# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



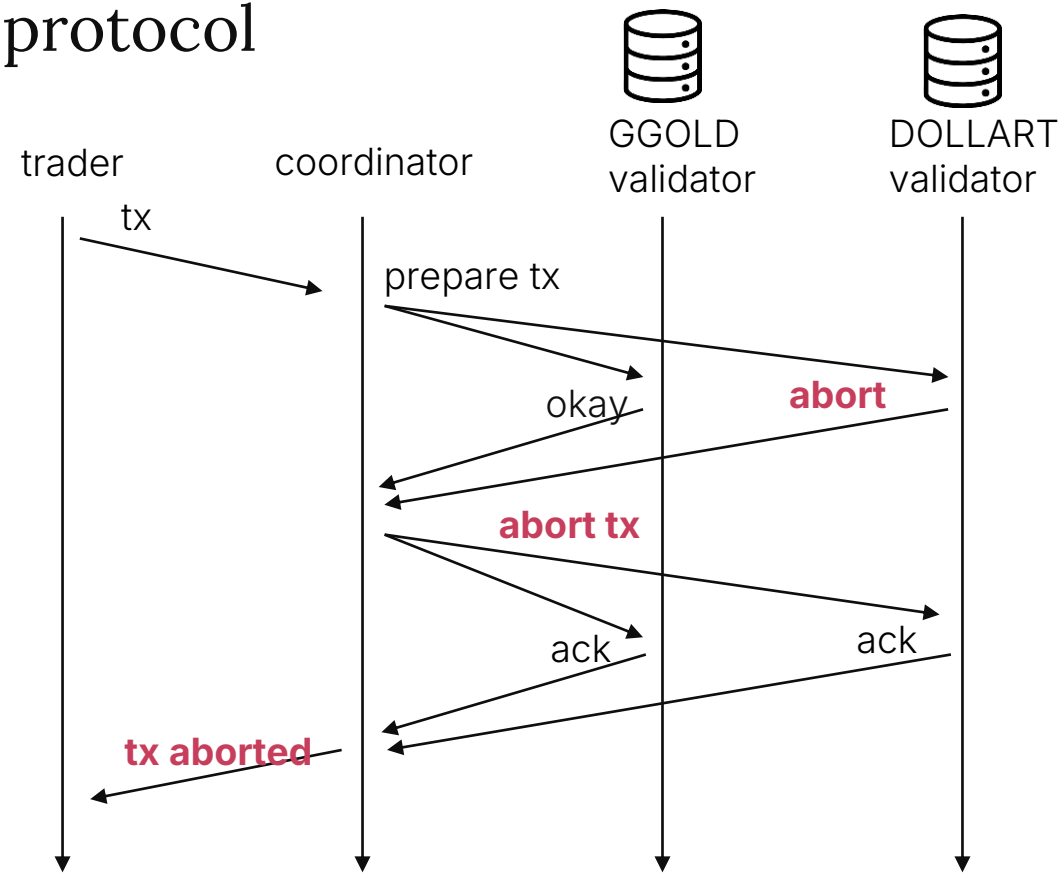
# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



# Idea: coordinate cross-issuer transactions with a 2-phase commit protocol



# Problems with the 2-phase commit approach

- ✓ Issuer-enforced finality
- ✓ Cross-issuer atomic transactions
- Not open access
  - We need to agree in advance on who will participate and who will be the coordinator
  - What about designating a global coordinator? Now we have a dictator...

# Federated Byzantine Agreement

# Federated Byzantine Agreement

- Sets called ***Quorums,*** that are given the authority to commit transactions, emerge from the local agreement requirements of the validators

# Federated Byzantine Agreement

- Sets called ***Quorums,*** that are given the authority to commit transactions, emerge from the local agreement requirements of the validators
- We can use the graph structure to elect leaders/coordinators

# Federated Byzantine Agreement

- Validators each define locally who they want to agree with

- Sets called ***Quorums,*** that are given the authority to commit transactions, emerge from the local agreement requirements of the validators
- We can use the graph structure to elect leaders/coordinators

# Federated Byzantine Agreement

- Validators each define locally who they want to agree with

- Sets called *Quorums,* that are given the authority to commit transactions, emerge from the local agreement requirements of the validators
- We can use the graph structure to elect leaders/coordinators

# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ➢ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- Quorums in Federated Byzantine Agreement systems
- Reliable Voting in Federated Byzantine Agreement systems
- Total-Order Broadcast by porting Simplex to FBA

# Model: Message-Passing Network with Byzantine Failures

We consider a set of validators $\mathcal{V}$ in a message passing network where:

# Model: Message-Passing Network with Byzantine Failures

We consider a set of validators $\mathcal{V}$ in a message passing network where:

- Each validator is either well-behaved or Byzantine faulty;
  we write $\mathcal{V} = \mathcal{W} \uplus \mathcal{F}$
  - $\mathcal{V}$ may be known (closed systems, PoS) with $|\mathcal{V}| = N$ and $|\mathcal{F}| = f$
  - $\mathcal{V}$ may be unknown (Bitcoin, FBA)
  - $\mathcal{F}$ is always unknown

# Model: Message-Passing Network with Byzantine Failures

We consider a set of validators $\mathcal{V}$ in a message passing network where:

- Each validator is either well-behaved or Byzantine faulty; we write $\mathcal{V} = \mathcal{W} \uplus \mathcal{F}$
  - $\mathcal{V}$ may be known (closed systems, PoS) with $|\mathcal{V}| = N$ and $|\mathcal{F}| = f$
  - $\mathcal{V}$ may be unknown (Bitcoin, FBA)
  - $\mathcal{F}$ is always unknown

# Model: Message-Passing Network with Byzantine Failures

We consider a set of validators $\mathcal{V}$ in a message passing network where:

- Each validator is either well-behaved or Byzantine faulty; we write $\mathcal{V} = \mathcal{W} \uplus \mathcal{F}$
  - $\mathcal{V}$ may be known (closed systems, PoS) with $|\mathcal{V}| = N$ and $|\mathcal{F}| = f$
  - $\mathcal{V}$ may be unknown (Bitcoin, FBA)
  - $\mathcal{F}$ is always unknown
- Each validator has a unique public/private key pair and signs their messages

# Model: Message-Passing Network with Byzantine Failures

We consider a set of validators $\mathcal{V}$ in a message passing network where:

- Each validator is either well-behaved or Byzantine faulty; we write $\mathcal{V} = \mathcal{W} \uplus \mathcal{F}$
  - $\mathcal{V}$ may be known (closed systems, PoS) with $|\mathcal{V}| = N$ and $|\mathcal{F}| = f$
  - $\mathcal{V}$ may be unknown (Bitcoin, FBA)
  - $\mathcal{F}$ is always unknown
- Each validator has a unique public/private key pair and signs their messages
- Validators have accurate local clocks

# Partial Synchrony

The system alternates unpredictably between ***synchronous*** and ***asynchronous*** periods

# Partial Synchrony

The system alternates unpredictably between ***synchronous*** and ***asynchronous*** periods

- In ***synchronous periods***, messages between well-behaved validators are delivered reliably in at most Δ time units
  - In closed systems, we can use point-to-point TCP connections
  - In open systems, we often use a gossip protocol over a p2p overlay

# Partial Synchrony

The system alternates unpredictably between ***synchronous*** and ***asynchronous*** periods

- In ***synchronous periods***, messages between well-behaved validators are delivered reliably in at most Δ time units
  - In closed systems, we can use point-to-point TCP connections
  - In open systems, we often use a gossip protocol over a p2p overlay
- In ***asynchronous periods***, message delay is unpredictable and messages may be dropped

# Partial Synchrony

The system alternates unpredictably between ***synchronous*** and ***asynchronous*** periods

- In ***synchronous periods***, messages between well-behaved validators are delivered reliably in at most Δ time units
  - In closed systems, we can use point-to-point TCP connections
  - In open systems, we often use a gossip protocol over a p2p overlay
- In ***asynchronous periods***, message delay is unpredictable and messages may be dropped

> Dwork, Lynch, and Stockmeyer. ***Consensus in the Presence of Partial Synchrony***. Journal of the ACM, 1988.

# Total-Order Broadcast (TOB)

# Total-Order Broadcast (TOB)

Validators repeatedly propose values and each validator must commit to a growing log* of values such that:

*A log of values is just a sequence of values. In a blockchain, values are blocks of transactions.

# Total-Order Broadcast (TOB)

Validators repeatedly propose values and each validator must commit to a growing log* of values such that:

*Validity:* Well-behaved validators only put valid† values in their log

*A log of values is just a sequence of values. In a blockchain, values are blocks of transactions.

†Typically, valid means properly signed, no duplicates, etc.

# Total-Order Broadcast (TOB)

Validators repeatedly propose values and each validator must commit to a growing log* of values such that:

***Validity:*** Well-behaved validators only put valid† values in their log

***Agreement:*** For every two logs committed by well-behaved validators, one is a prefix of the other

*A log of values is just a sequence of values. In a blockchain, values are blocks of transactions.

†Typically, valid means properly signed, no duplicates, etc.

# Total-Order Broadcast (TOB)

Validators repeatedly propose values and each validator must commit to a growing log* of values such that:

*Validity:* Well-behaved validators only put valid† values in their log

*Agreement:* For every two logs committed by well-behaved validators, one is a prefix of the other

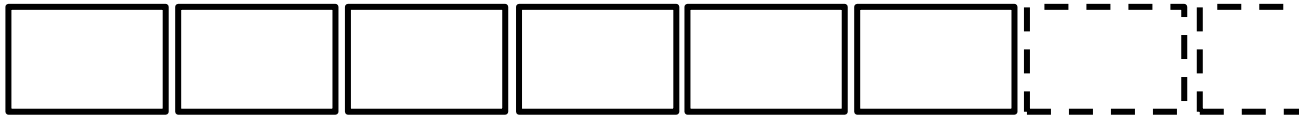
*Progress:* If the system remains synchronous long enough, every well-behaved validator eventually adds one more value proposed by a well-behaved validator to its committed log

*A log of values is just a sequence of values. In a blockchain, values are blocks of transactions.

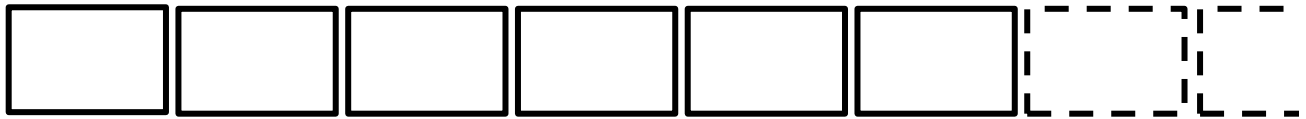
†Typically, valid means properly signed, no duplicates, etc.

# We can think of TOB as filling consecutive positions in local arrays

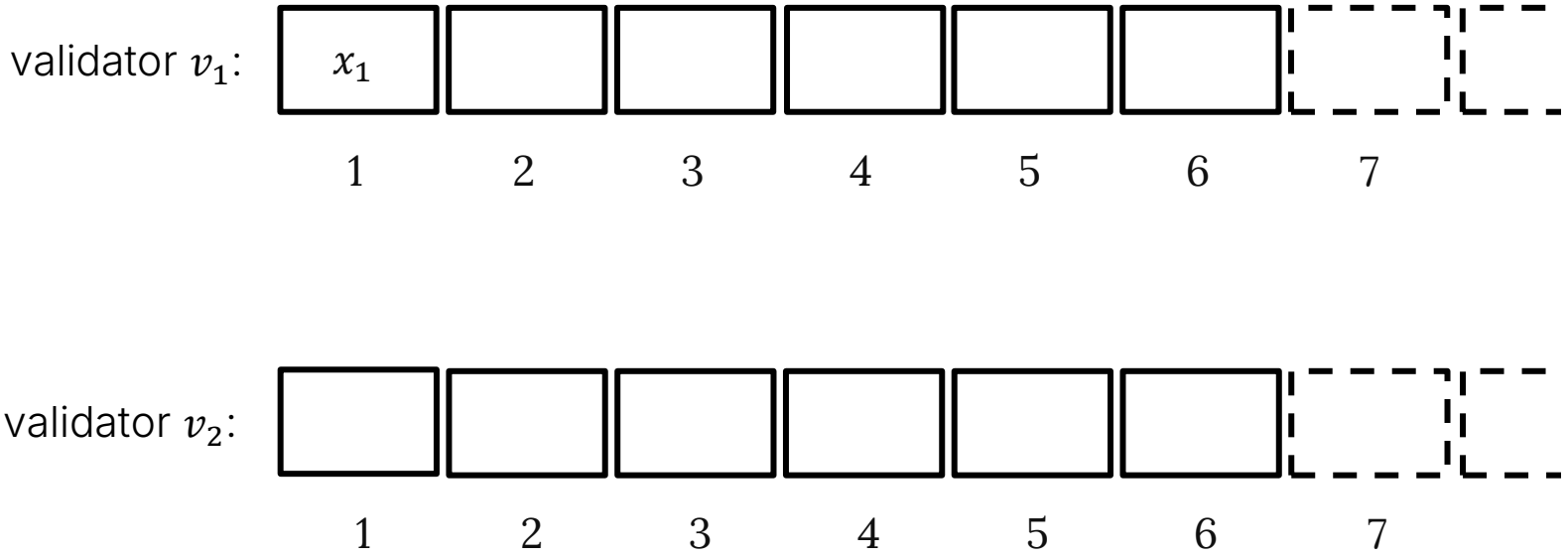
validator $v_1$:



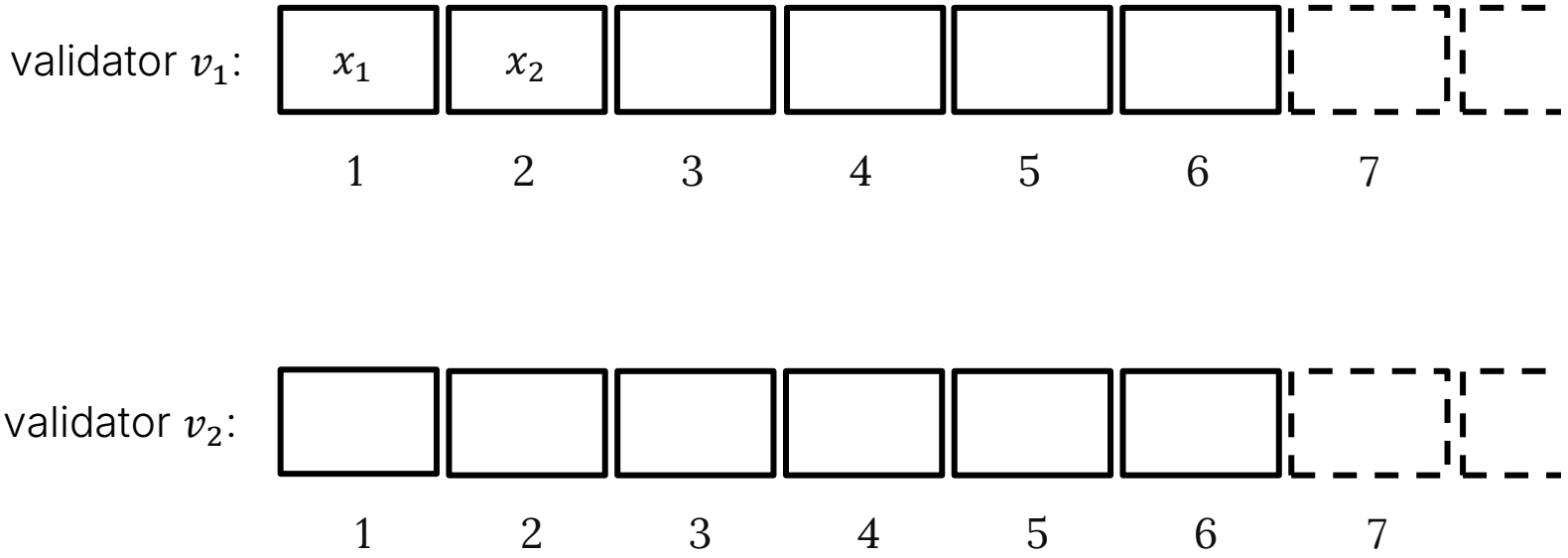

validator $v_2$:



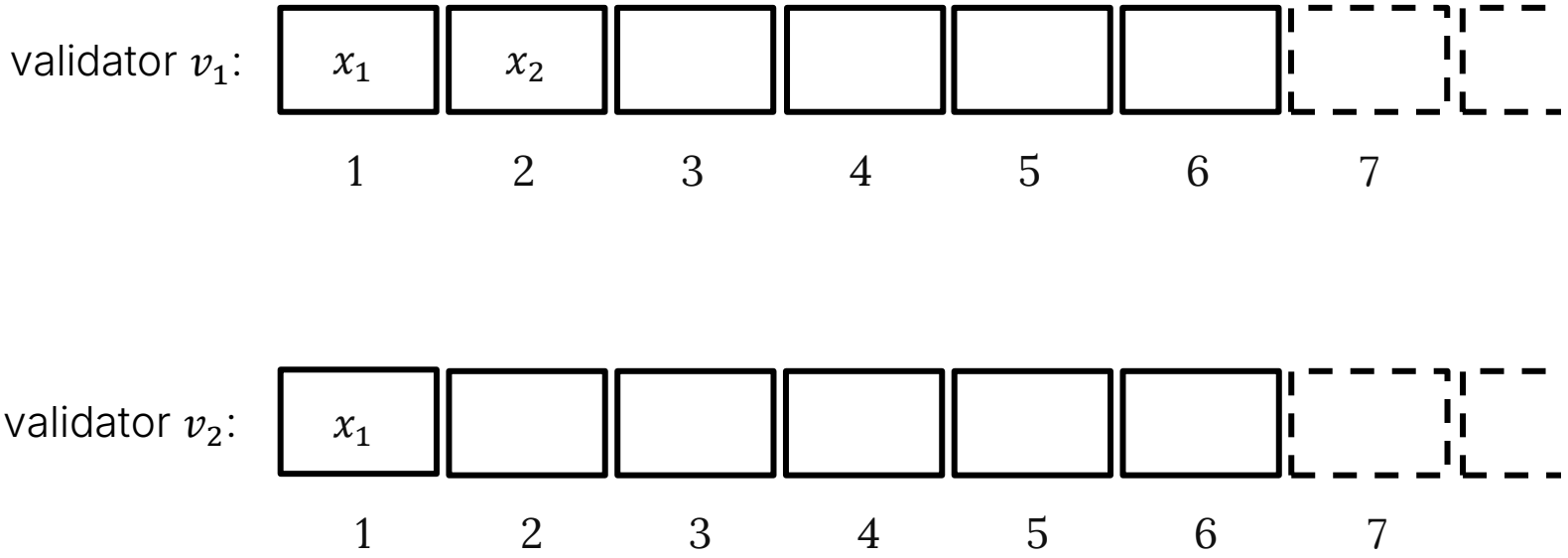
# We can think of TOB as filling consecutive positions in local arrays



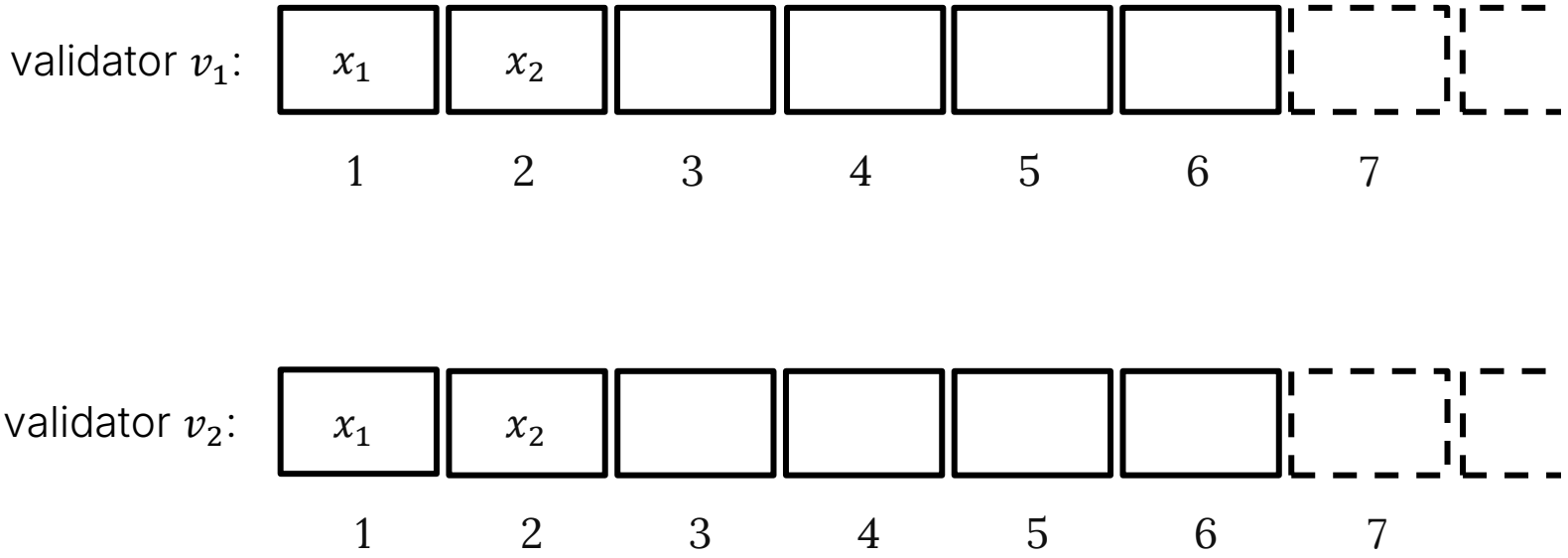
# We can think of TOB as filling consecutive positions in local arrays



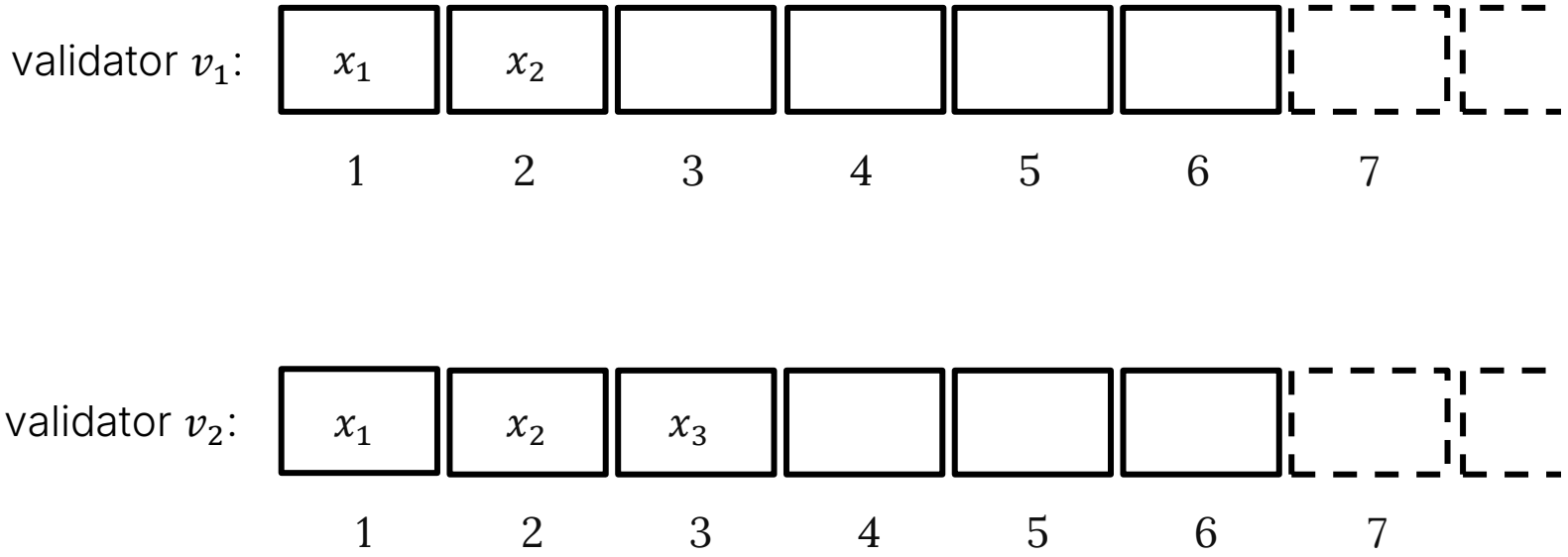
# We can think of TOB as filling consecutive positions in local arrays



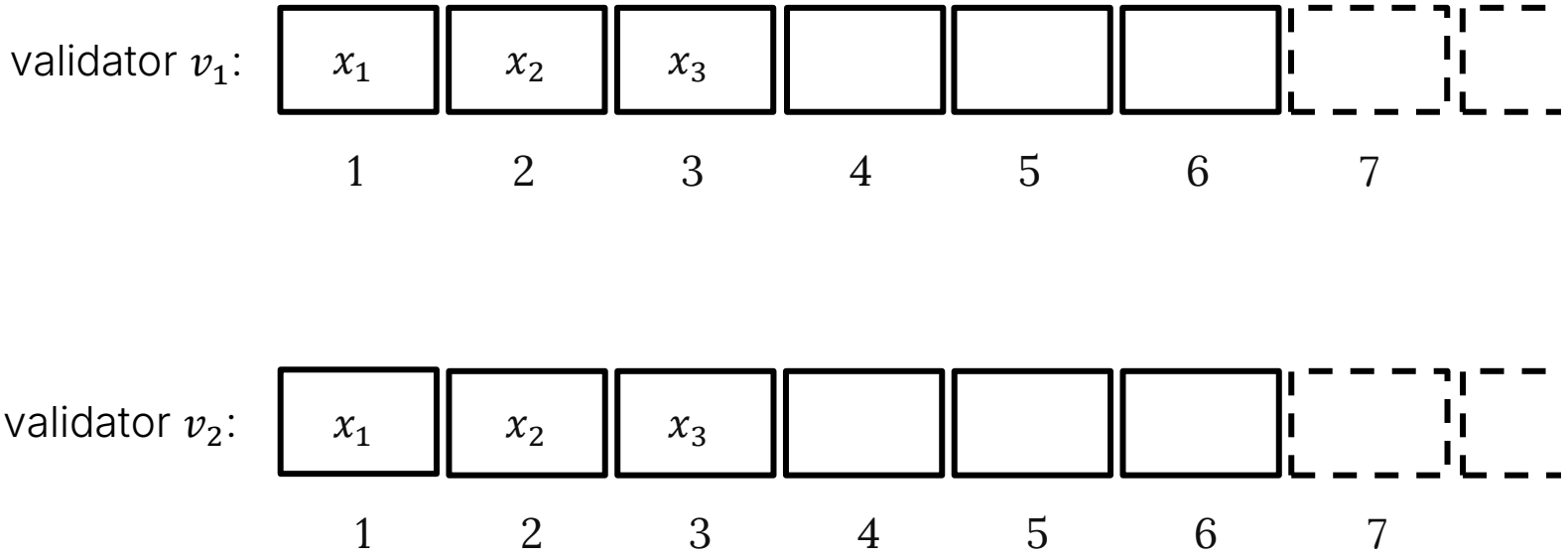
# We can think of TOB as filling consecutive positions in local arrays



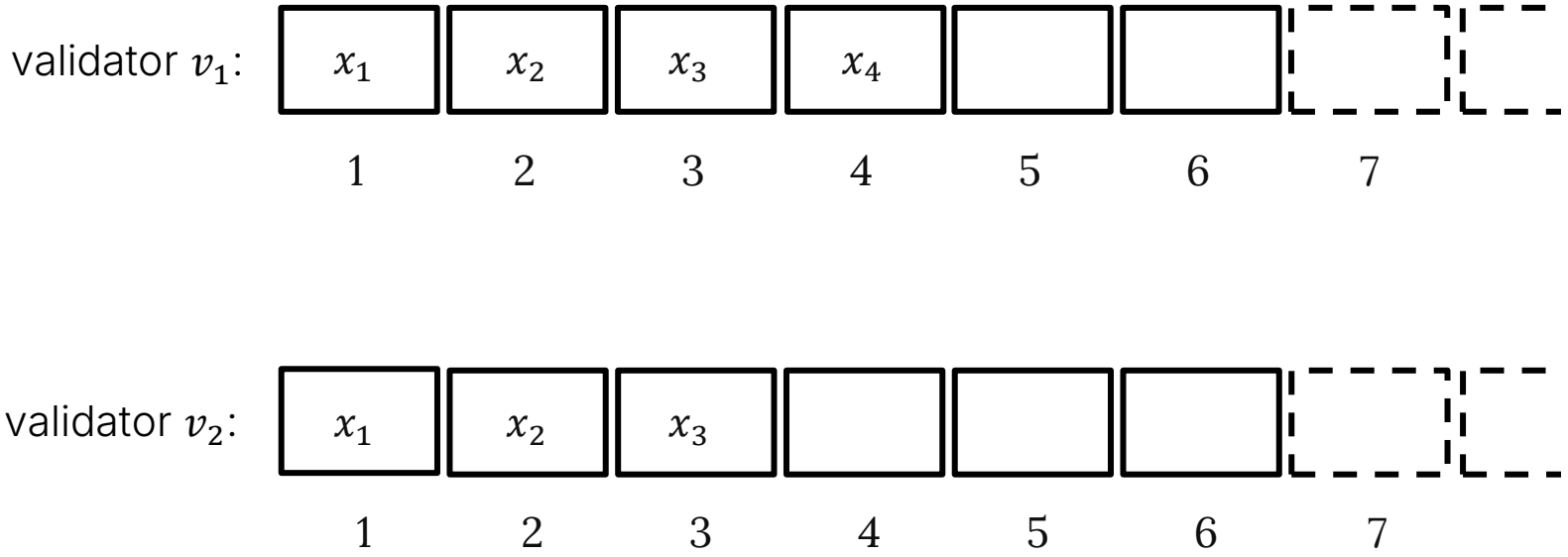
# We can think of TOB as filling consecutive positions in local arrays



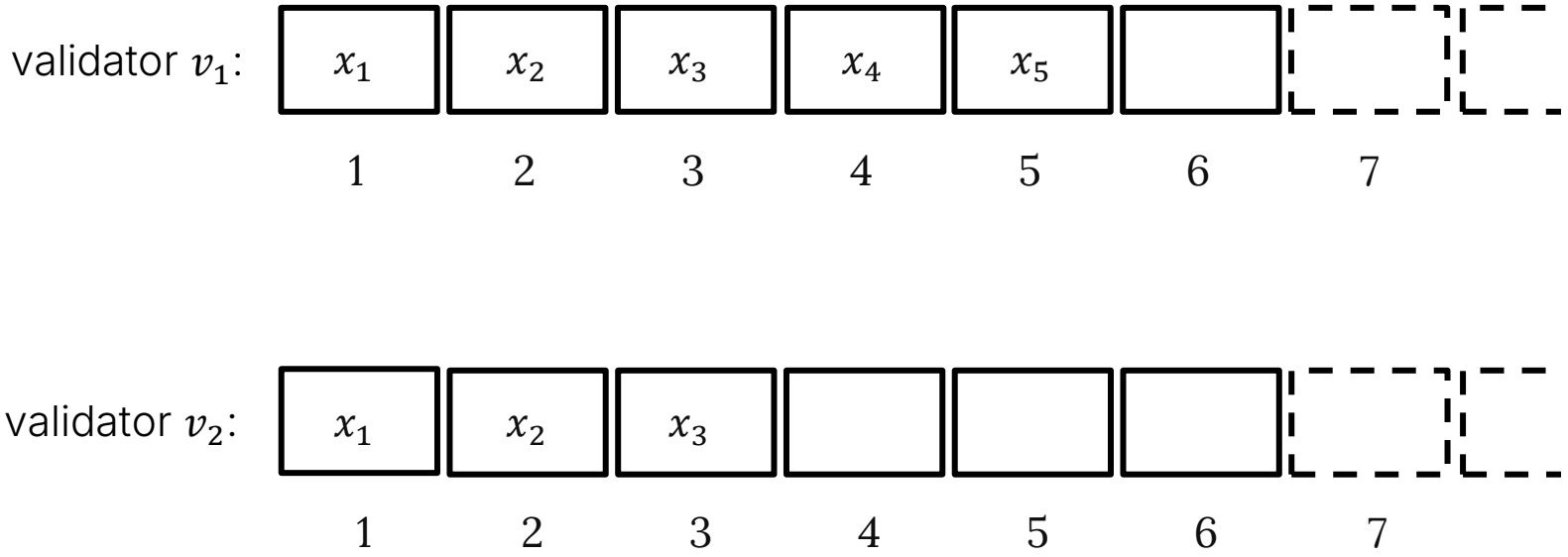
# We can think of TOB as filling consecutive positions in local arrays



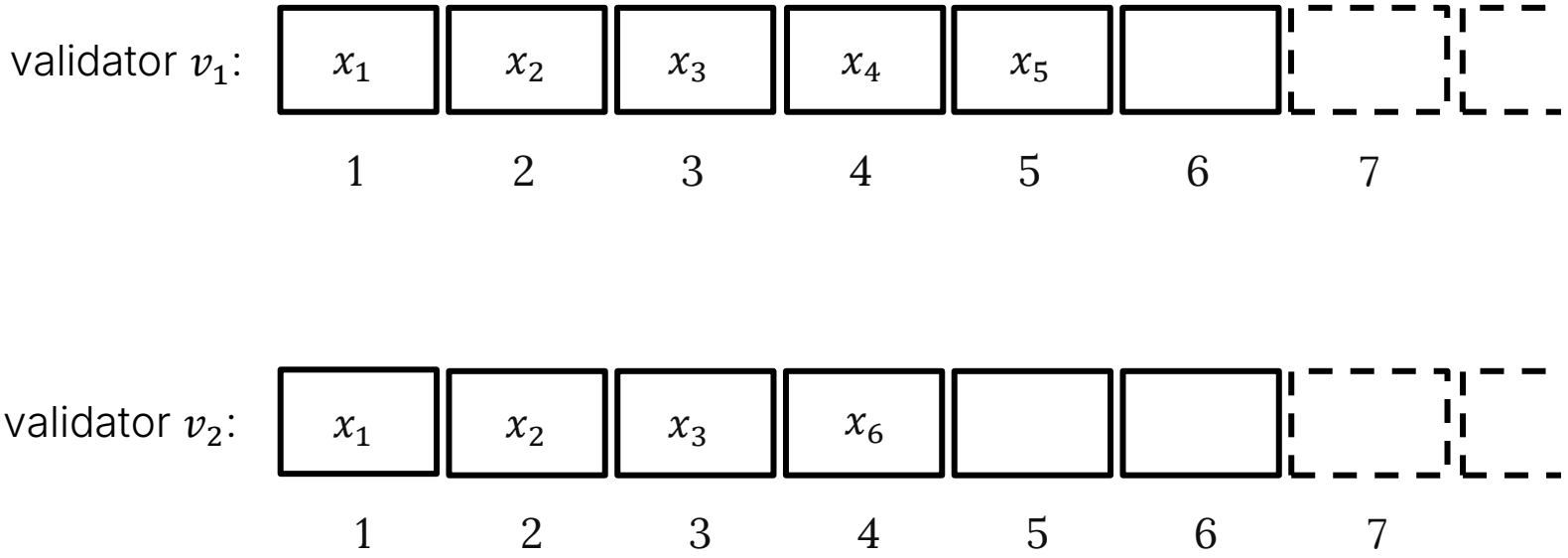
# We can think of TOB as filling consecutive positions in local arrays



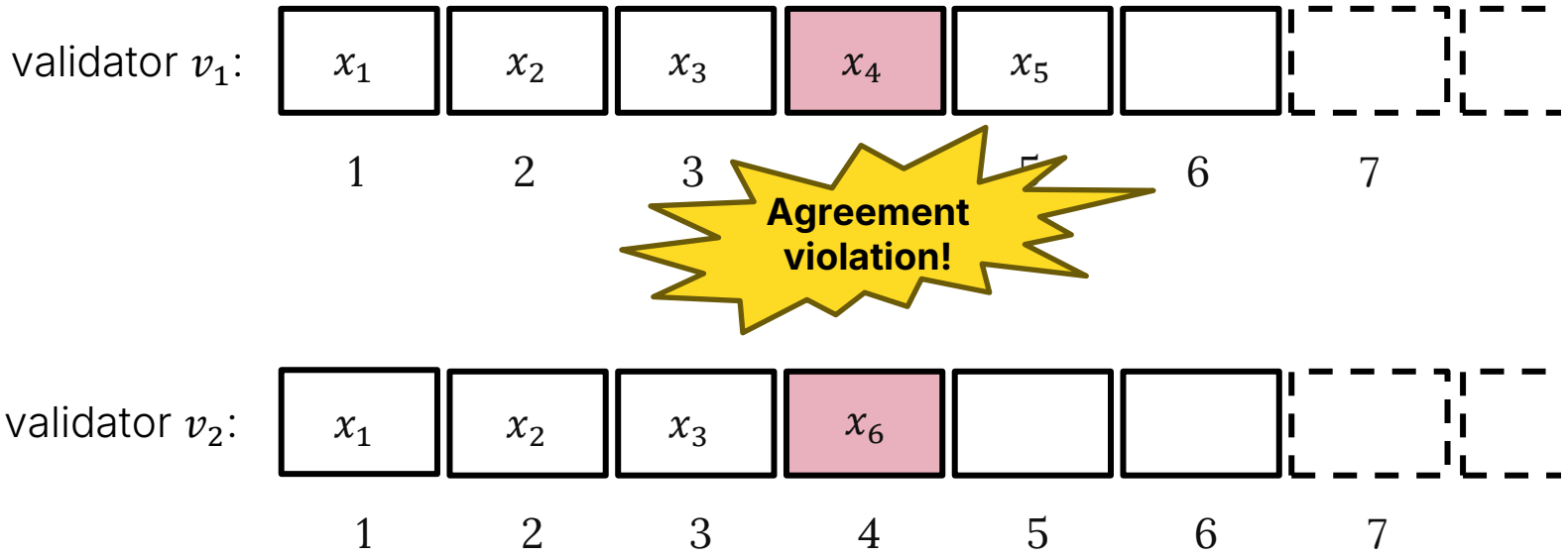
# We can think of TOB as filling consecutive positions in local arrays



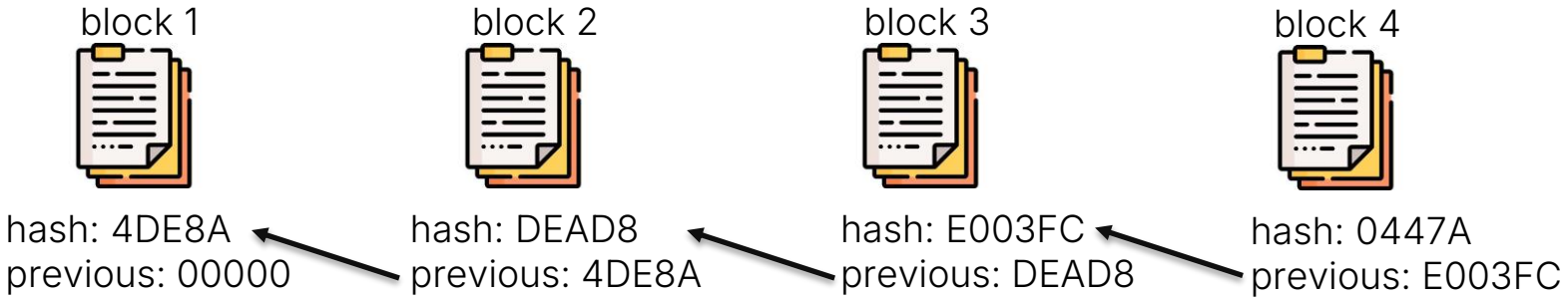
# We can think of TOB as filling consecutive positions in local arrays



# We can think of TOB as filling consecutive positions in local arrays

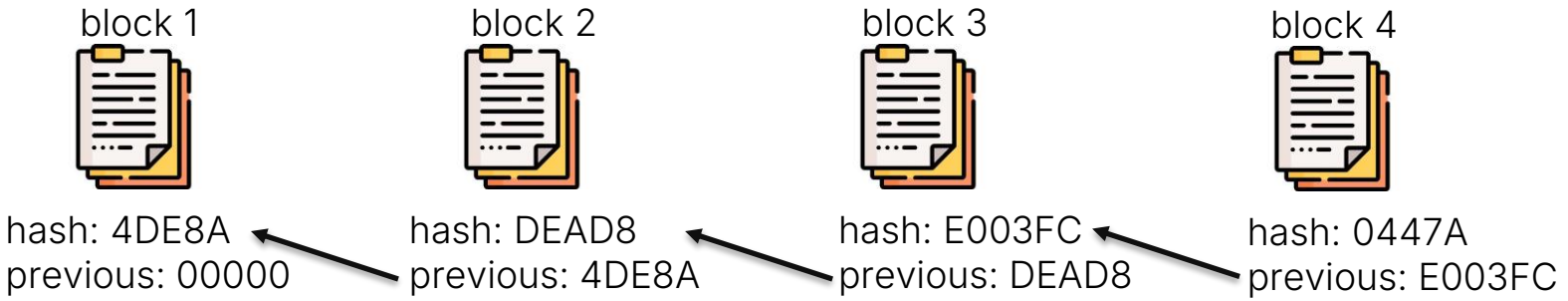


# With Total-Order Broadcast (TOB), we can implement blockchains



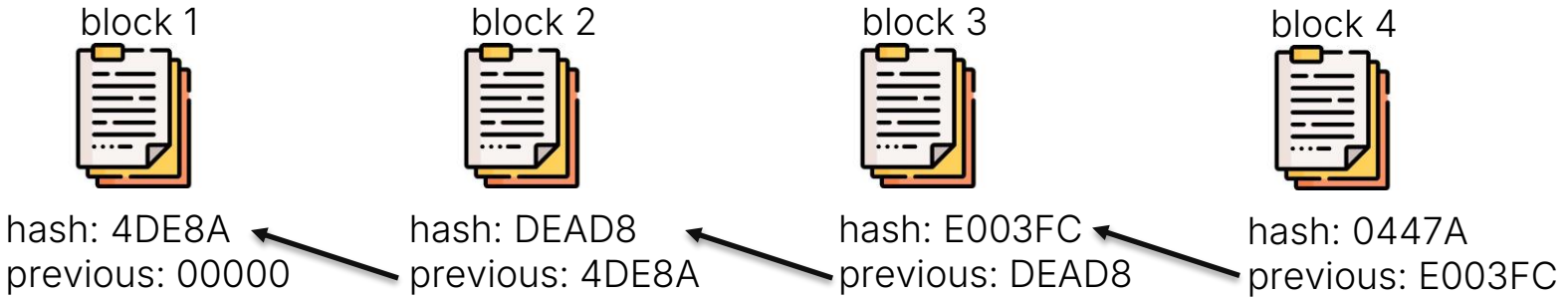
# With Total-Order Broadcast (TOB), we can implement blockchains

- *values* are *blocks*, which contain a list of transactions submitted to the validators by users



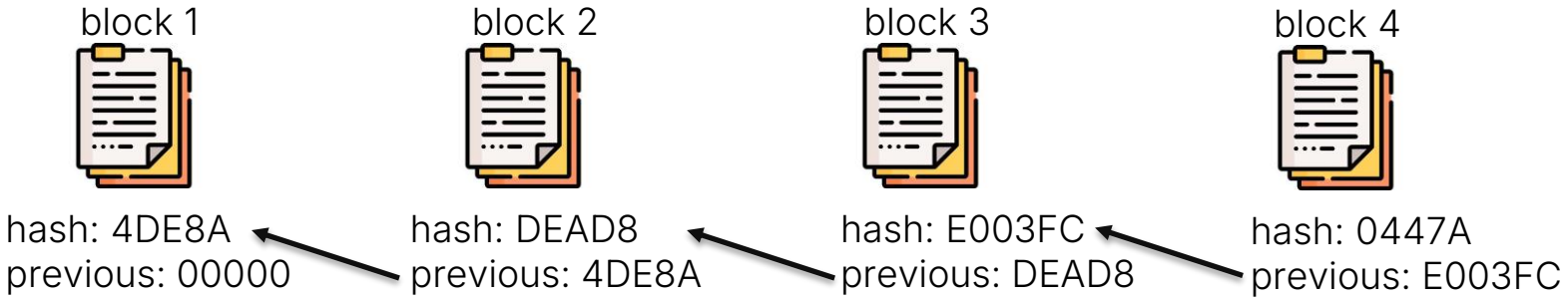
# With Total-Order Broadcast (TOB), we can implement blockchains

- *values* are *blocks*, which contain a list of transactions submitted to the validators by users
- Validators execute the blocks in the order given by their committed logs, and send back execution results to clients



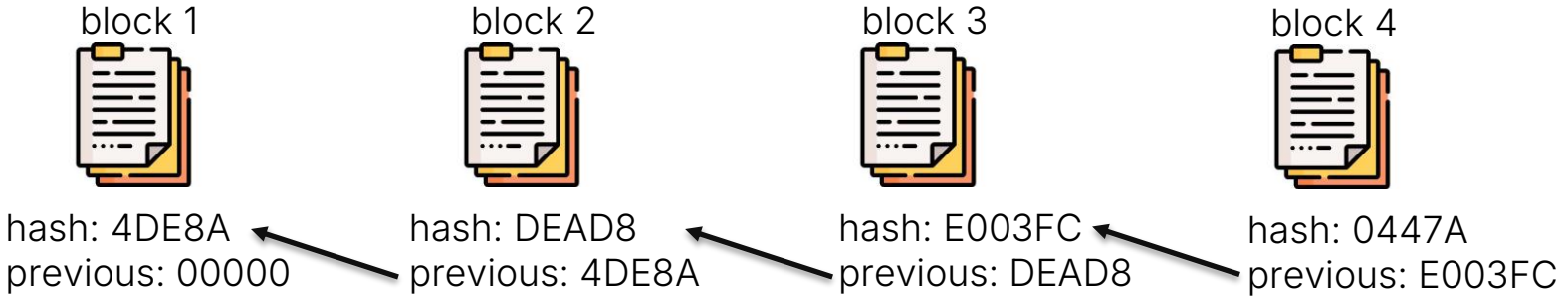
# With Total-Order Broadcast (TOB), we can implement blockchains

- *values* are *blocks*, which contain a list of transactions submitted to the validators by users
- Validators execute the blocks in the order given by their committed logs, and send back execution results to clients
- Each block also contains a cryptographic hash of the previous block, so a block determines a ledger state



# With Total-Order Broadcast (TOB), we can implement blockchains

- *values* are *blocks*, which contain a list of transactions submitted to the validators by users
- Validators execute the blocks in the order given by their committed logs, and send back execution results to clients
- Each block also contains a cryptographic hash of the previous block, so a block determines a ledger state
- The system inherits the fault tolerance of the TOB protocol



# Byzantine Quorum systems

A Byzantine quorum system (BQS) is a collection of sets of validators called quorums, where we assume:

# Byzantine Quorum systems

A Byzantine quorum system (BQS) is a collection of sets of validators called quorums, where we assume:

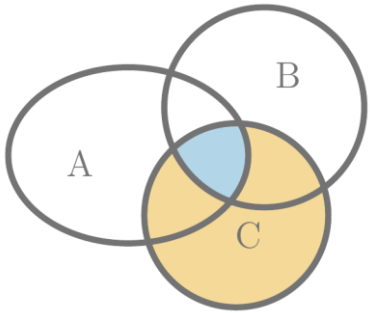
*Quorum Availability (QA)*: at least one (unknown) quorum is well-behaved (meaning all its members are well-behaved)

# Byzantine Quorum systems

A Byzantine quorum system (BQS) is a collection of sets of validators called quorums, where we assume:

*Quorum Availability (QA)*: at least one (unknown) quorum is well-behaved
(meaning all its members are well-behaved)

*Quorum Intersection (QI):* every 3 quorums intersect*
(so, every two quorums have a well-behaved member in common)




* also called “Property B³” in the context of fail-prone systems

# Byzantine Quorum systems

A Byzantine quorum system (BQS) is a collection of sets of validators called quorums, where we assume:

*Quorum Availability (QA)*: at least one (unknown) quorum is well-behaved
(meaning all its members are well-behaved)

*Quorum Intersection (QI):* every 3 quorums intersect*
(so, every two quorums have a well-behaved member in common)

Example:
3 out of 4

* also called “Property B³” in the context of fail-prone systems

# Byzantine Quorum systems

A Byzantine quorum system (BQS) is a collection of sets of validators called quorums, where we assume:

*Quorum Availability (QA)*: at least one (unknown) quorum is well-behaved
(meaning all its members are well-behaved)

*Quorum Intersection (QI):* every 3 quorums intersect*
(so, every two quorums have a well-behaved member in common)

In PoS we have weighted validators and any set of collective weight > 2/3 is a quorum

* also called “Property B³” in the context of fail-prone systems

# Reliable Voting

Each validator may cast a unique vote and may confirm a unique value, such that:

# Reliable Voting

Each validator may cast a unique vote and may confirm a unique value, such that:

***Unanimity:*** If all well-behaved validators vote for $x$, then they all eventually confirm $x$

# Reliable Voting

Each validator may cast a unique vote and may confirm a unique value, such that:

***Unanimity:*** If all well-behaved validators vote for $x$, then they all eventually confirm $x$

***Validity:*** If a well-behaved validator confirms $x$, then a well-behaved validator voted for $x$

# Reliable Voting

Each validator may cast a unique vote and may confirm a unique value, such that:

*Unanimity:* If all well-behaved validators vote for $x$, then they all eventually confirm $x$

*Validity:* If a well-behaved validator confirms $x$, then a well-behaved validator voted for $x$

*Agreement:* No two well-behaved validators confirm different values

# Reliable Voting

Each validator may cast a unique vote and may confirm a unique value, such that:

*Unanimity:* If all well-behaved validators vote for $x$, then they all eventually confirm $x$

*Validity:* If a well-behaved validator confirms $x$, then a well-behaved validator voted for $x$

*Agreement:* No two well-behaved validators confirm different values

*Totality:* If a well-behaved validator confirms $x$, then all well-behaved validators eventually confirm $x$

# Reliable Voting

Each validator may cast a unique vote and may confirm a unique value, such that:

*Unanimity:* If all well-behaved validators vote for $x$, then they all eventually confirm $x$

*Validity:* If a well-behaved validator confirms $x$, then a well-behaved validator voted for $x$

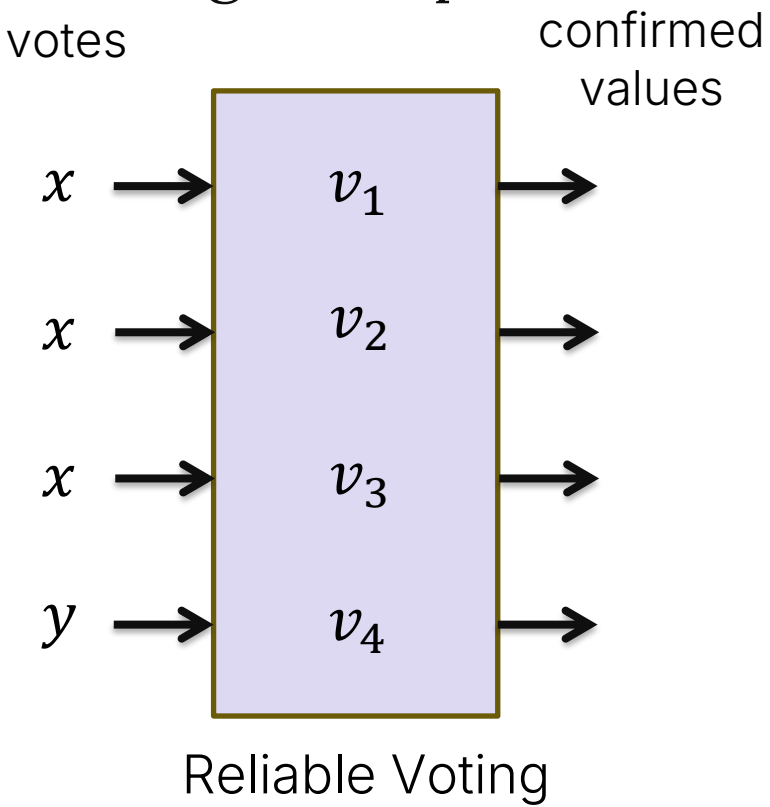
*Agreement:* No two well-behaved validators confirm different values

*Totality:* If a well-behaved validator confirms $x$, then all well-behaved validators eventually confirm $x$

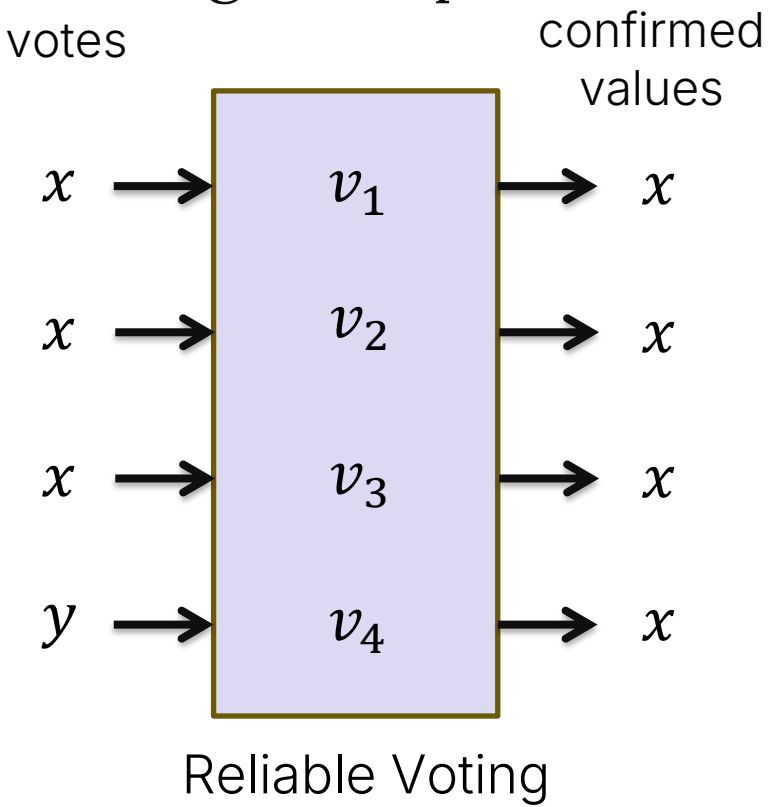
Notes*:*

- Reliable Voting is allowed to get stuck
- This is like Reliable Broadcast but without a broadcaster

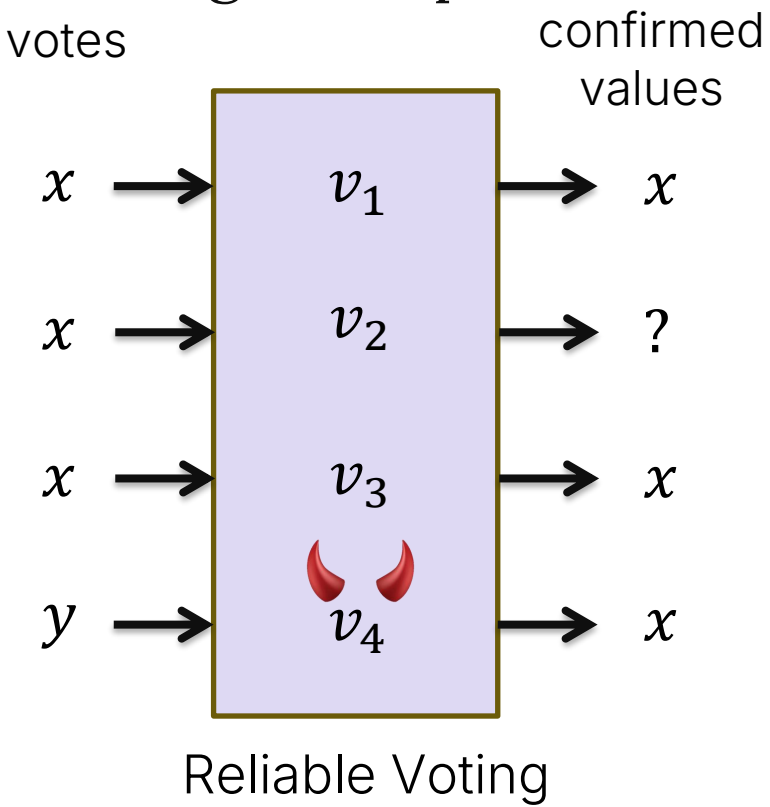
# Reliable Voting: examples



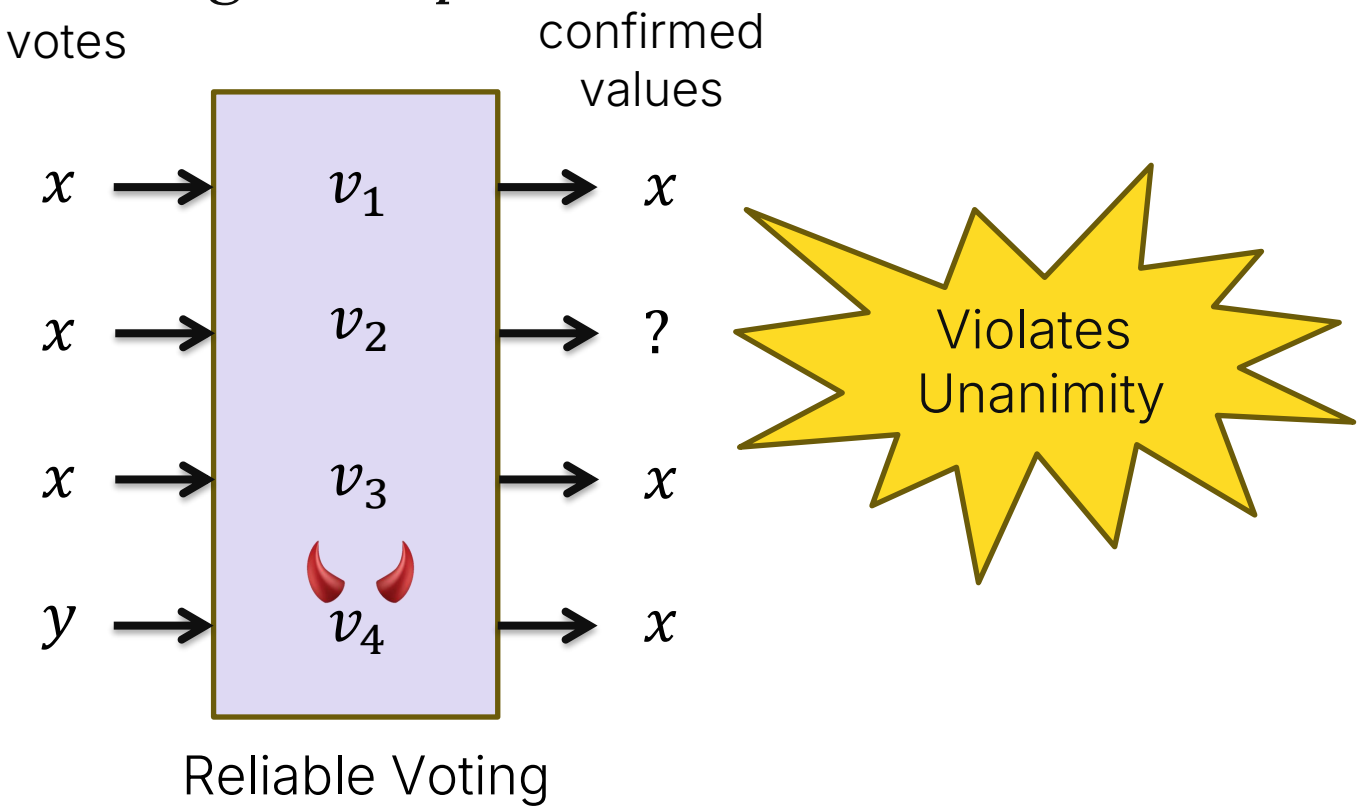
# Reliable Voting: examples



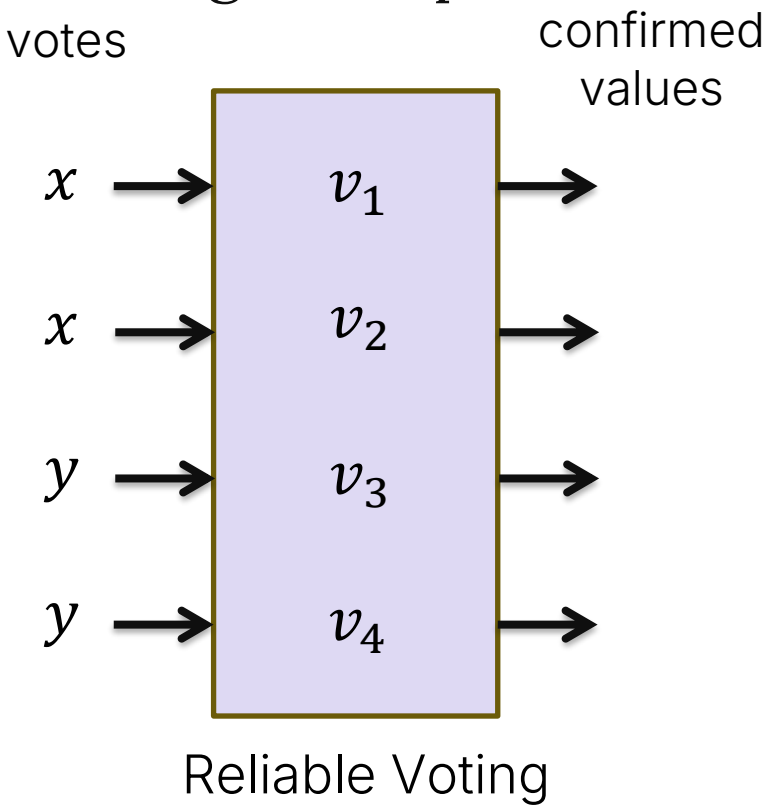

# Reliable Voting: examples



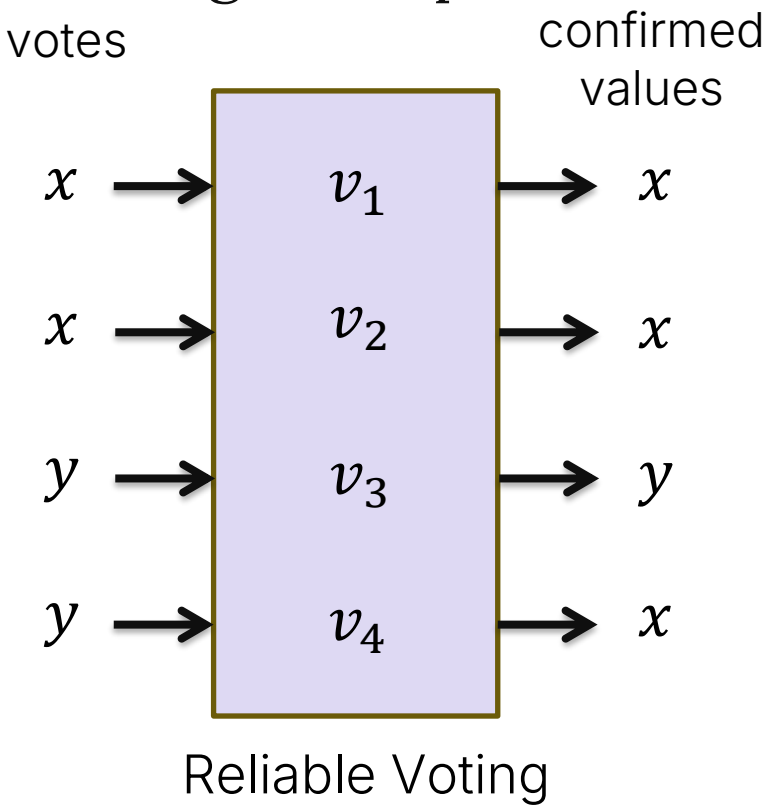
# Reliable Voting: examples



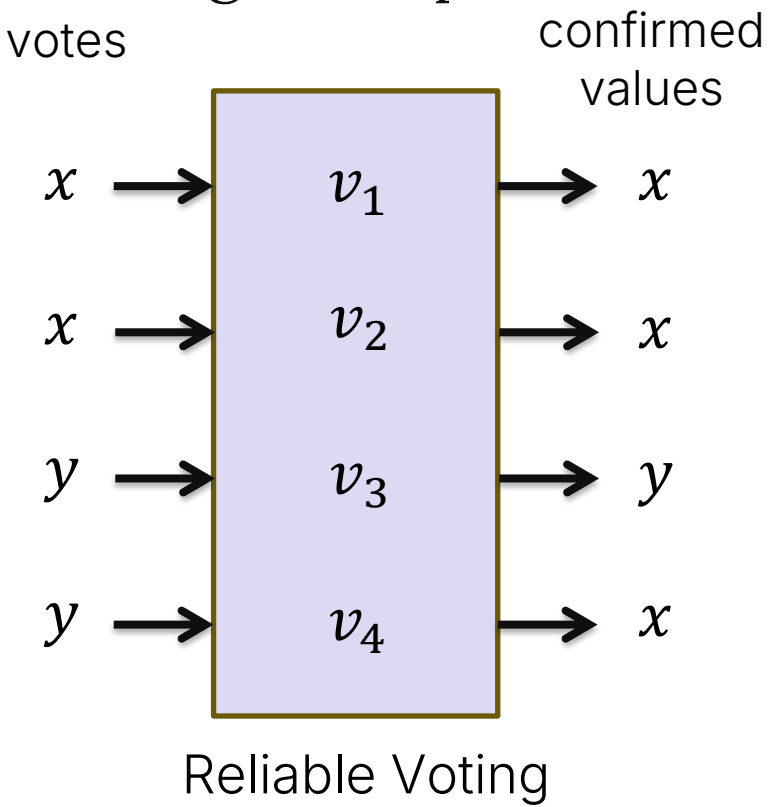
# Reliable Voting: examples



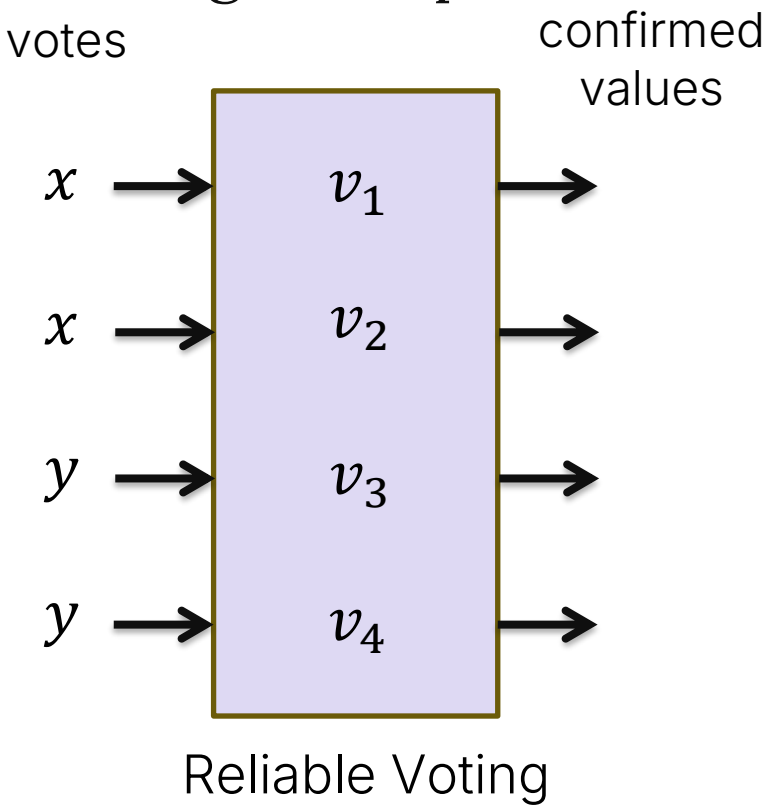
# Reliable Voting: examples



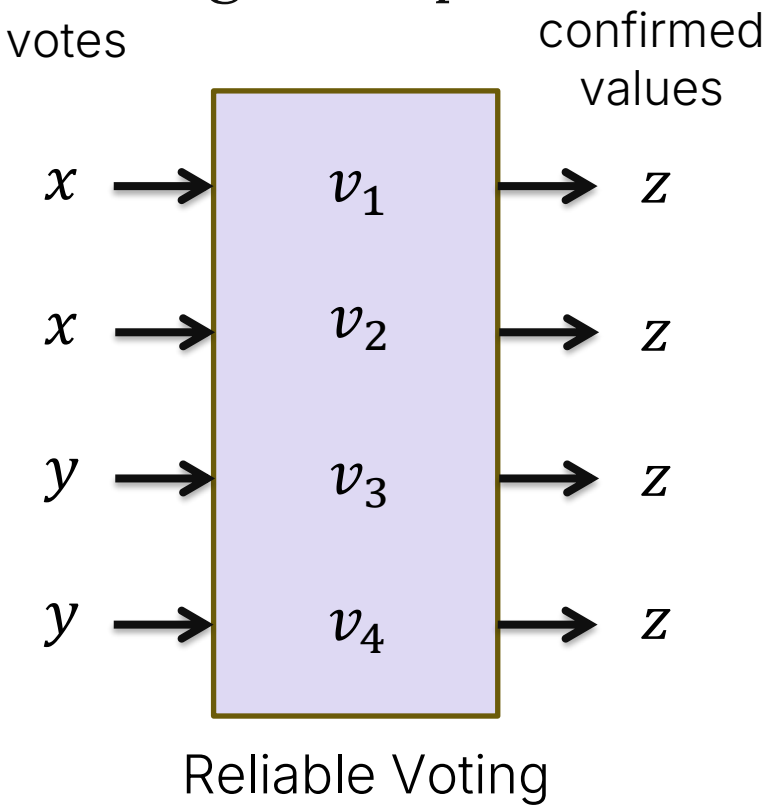
# Reliable Voting: examples



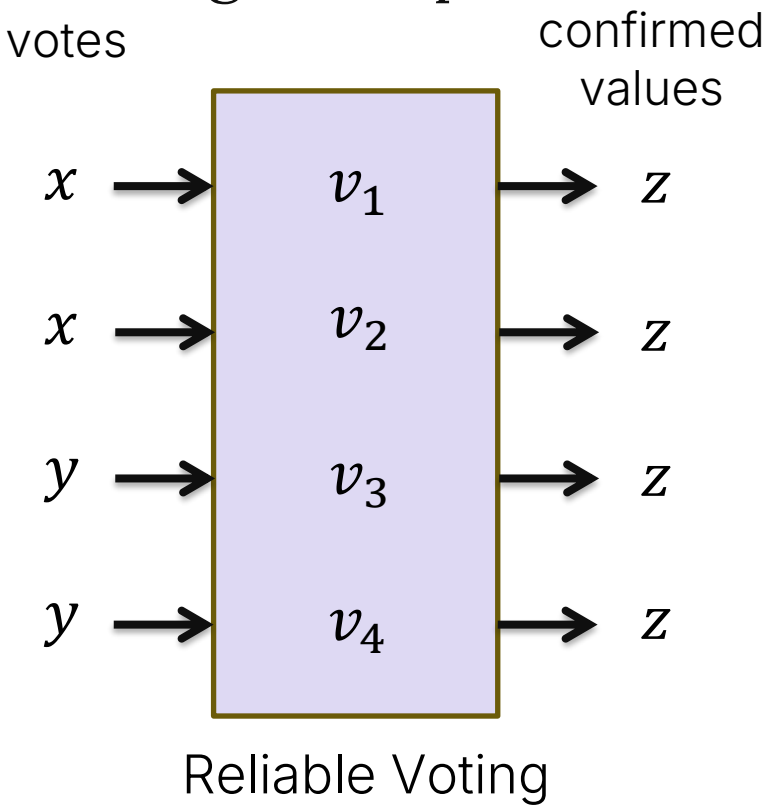

# Reliable Voting: examples



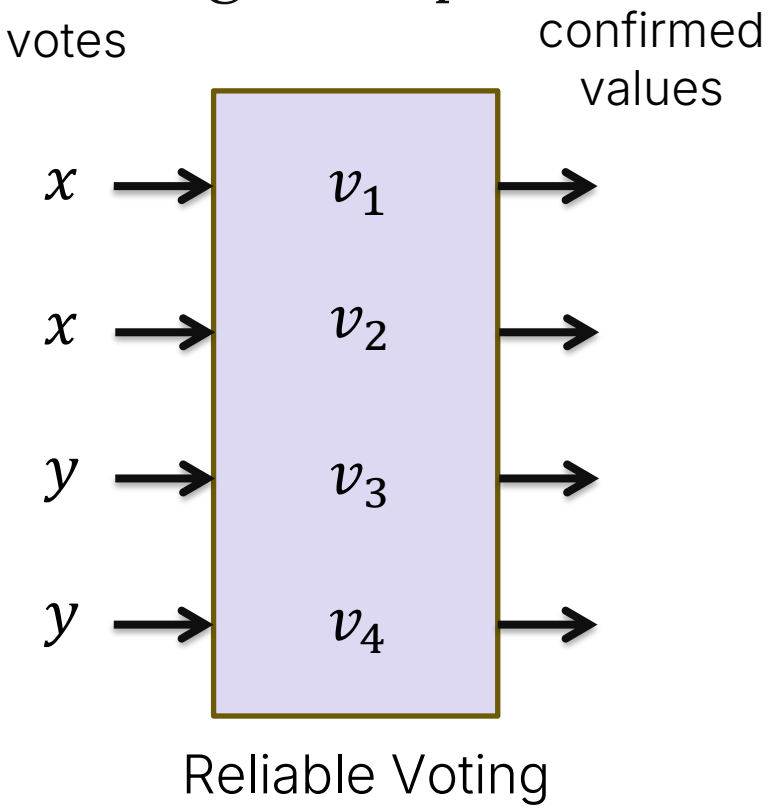
# Reliable Voting: examples



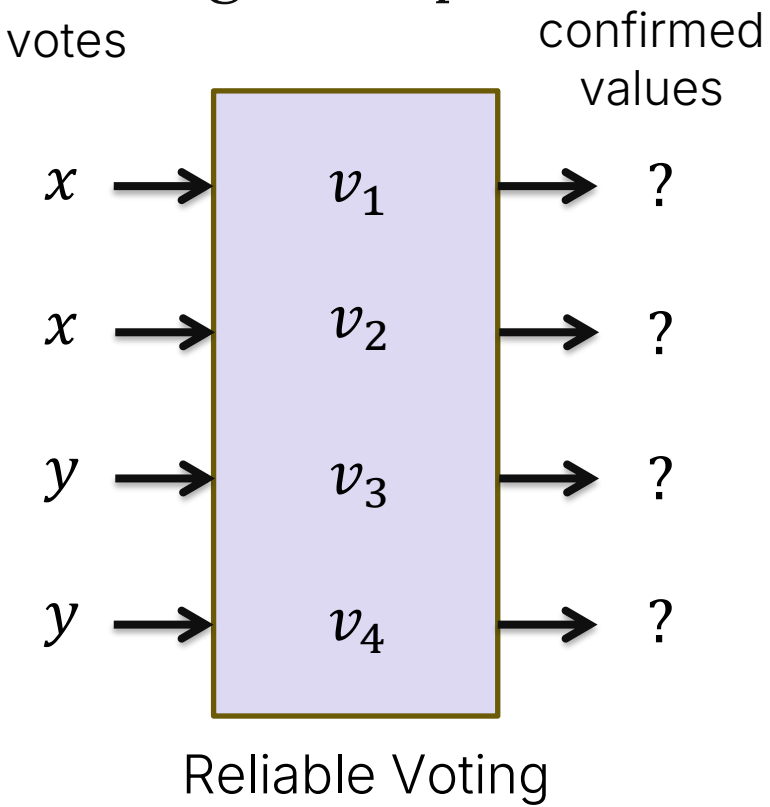
# Reliable Voting: examples



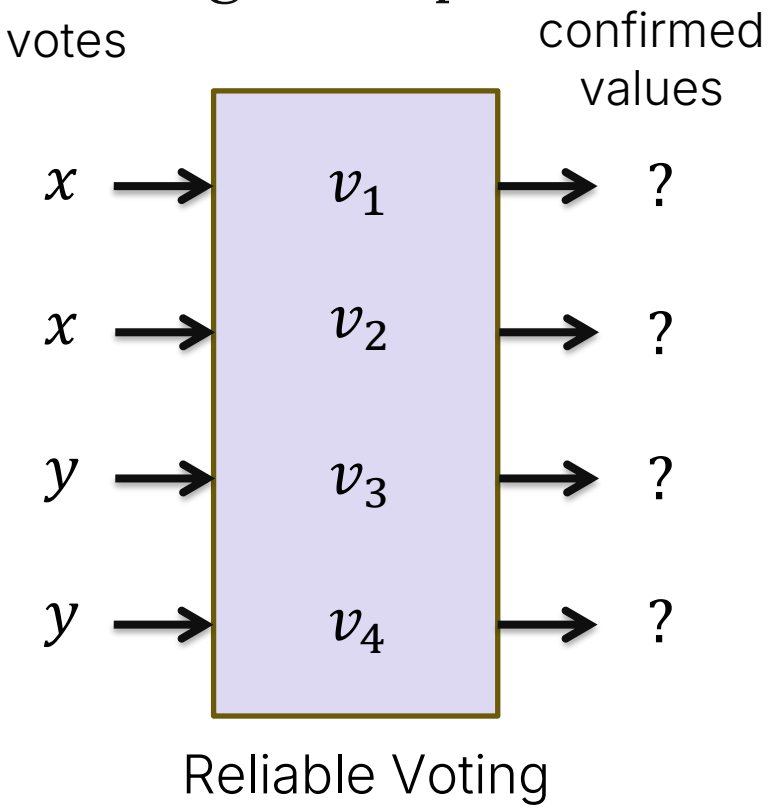

# Reliable Voting: examples



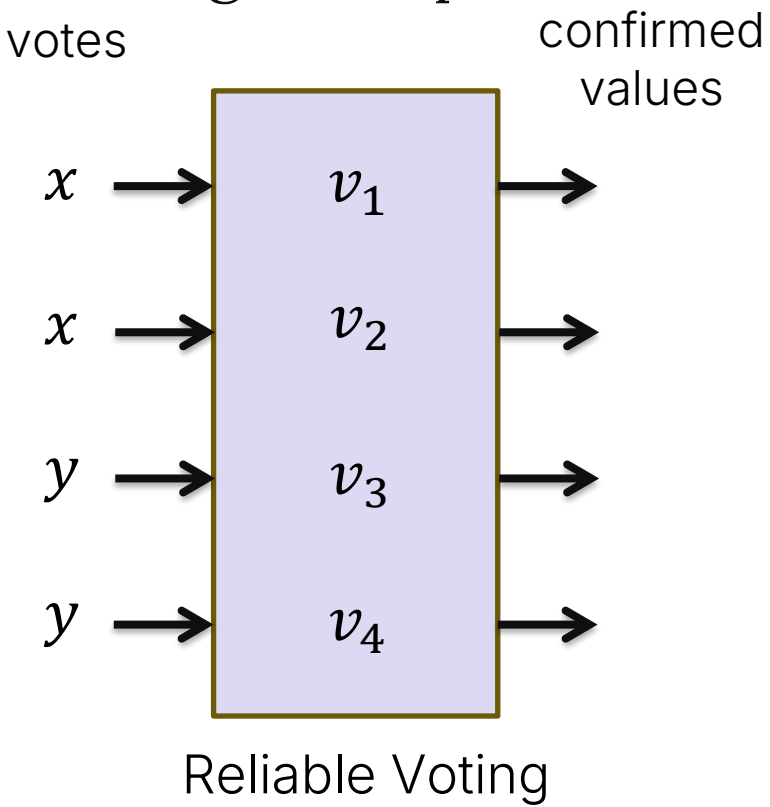
# Reliable Voting: examples



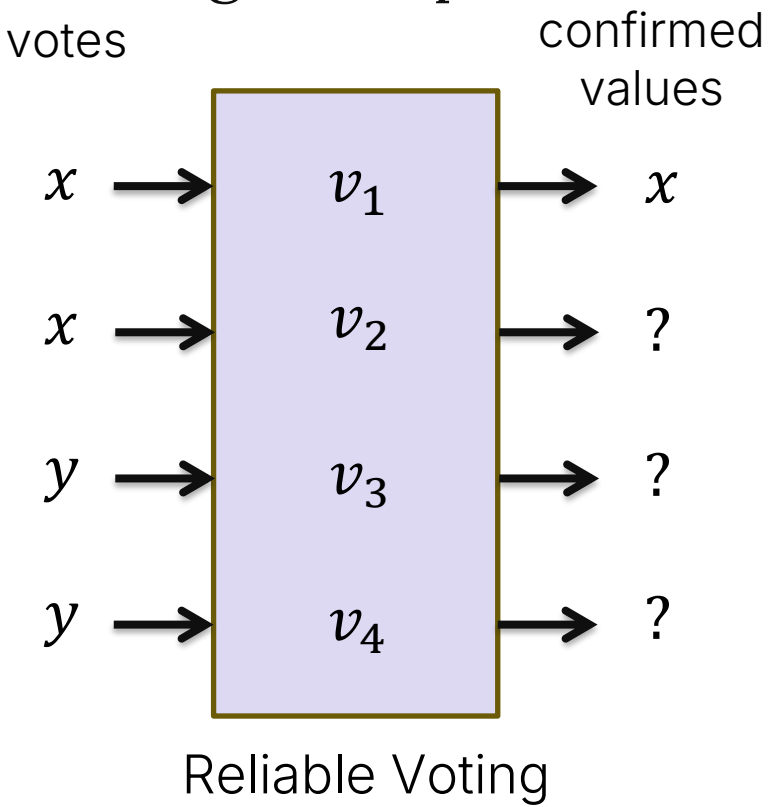
# Reliable Voting: examples



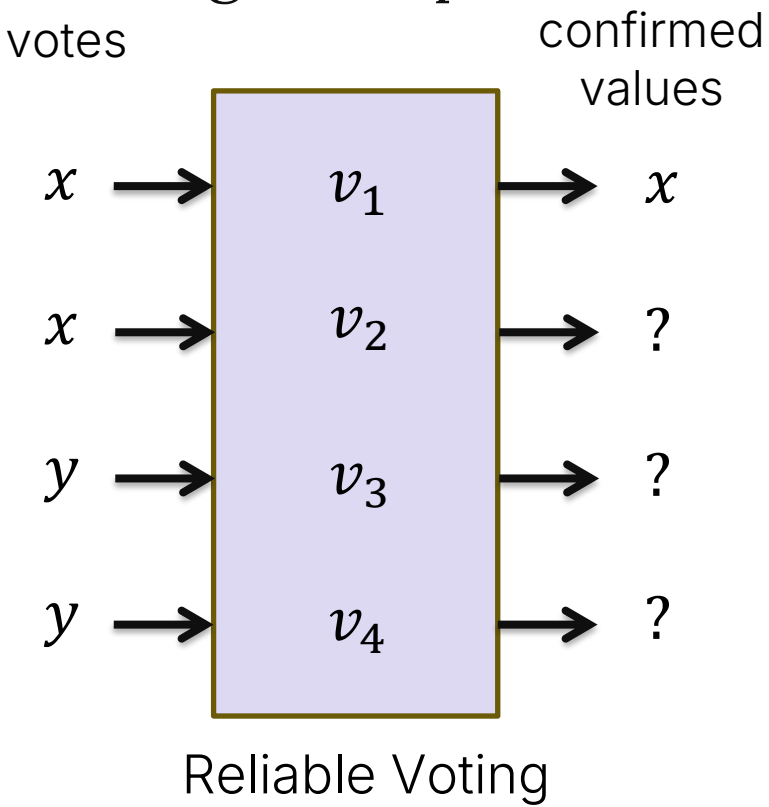

# Reliable Voting: examples

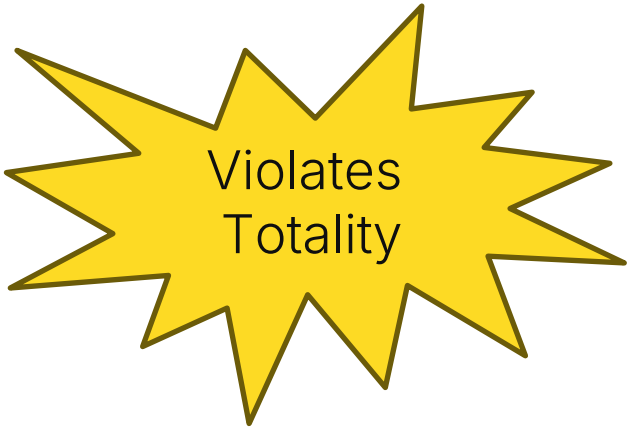


# Reliable Voting: examples



# Reliable Voting: examples



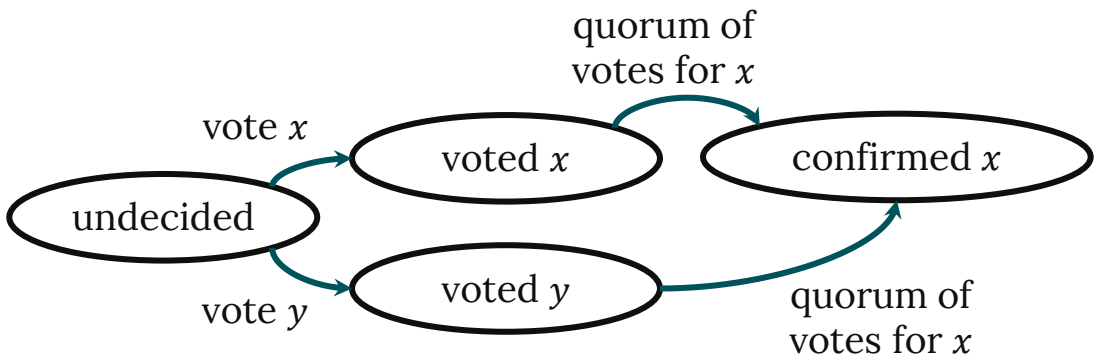



# Reliable Voting using quorums is simple

- Each validator broadcasts its vote
- A validator confirms a value when it learns a quorum unanimously voted for it
- Validators re-broadcast every vote they receive

# Reliable Voting using quorums is simple

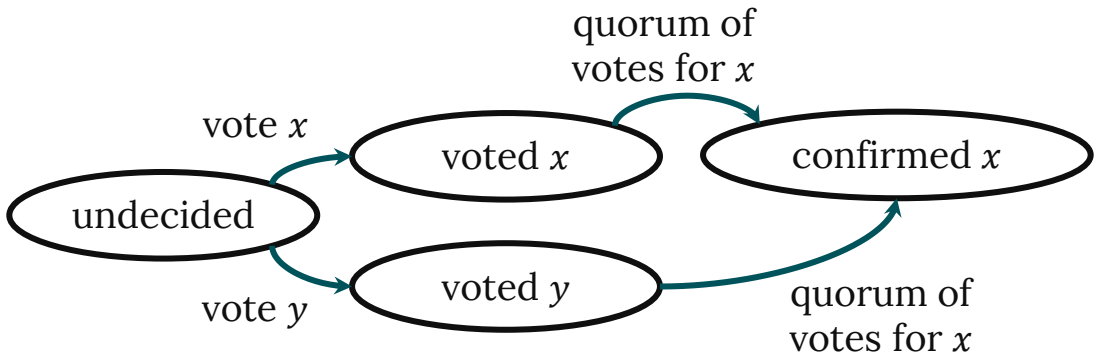
- Each validator broadcasts its vote
- A validator confirms a value when it learns a quorum unanimously voted for it
- Validators re-broadcast every vote they receive



# Reliable Voting using quorums is simple

- Each validator broadcasts its vote
- A validator confirms a value when it learns a quorum unanimously voted for it
- Validators re-broadcast every vote they receive

- Quorum Intersection ⇒ Agreement
- Agreement + Quorum availability ⇒ Unanimity + Totality
- Quorum intersection + Quorum availability ⇒ Validity



# From Reliable Voting to TOB with Simplex

Simplex is a popular and simple blockchain consensus algorithm

- Chan and Pass, ***Simplex consensus: A simple and fast consensus protocol***, TCC 2023
- Shoup, ***Sing a song of Simplex***, DISC 2024

We will see how we can phrase Simplex in terms of Reliable Voting + Leader Election, which then makes it easy to port Simplex to the FBA model

# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ✓ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- ➢ Quorums in Federated Byzantine Agreement systems
- • Reliable Voting in FBA systems
- • Total-Order Broadcast by porting Simplex to FBA

# In FBA, validators declare local, unilateral agreement requirements

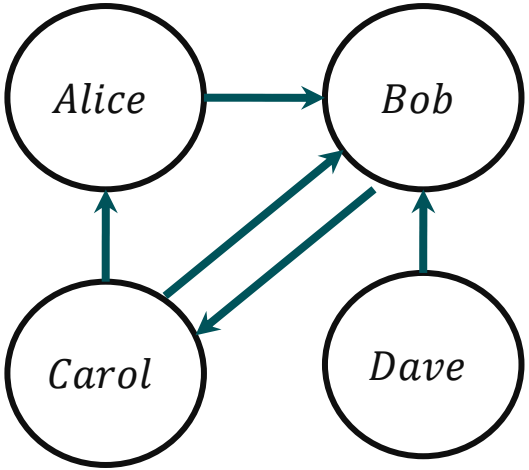
Example

Alice requires agreement from Bob

Bob requires agreement from Carol

Carol requires agreement from Alice and Bob

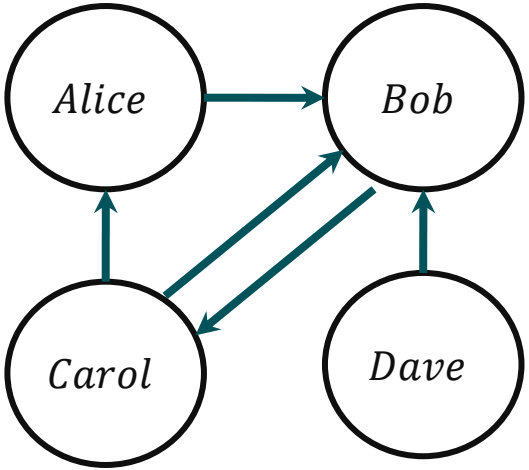
Dave requires agreement from Bob




"Alice → Bob" means Alice requires agreement from Bob

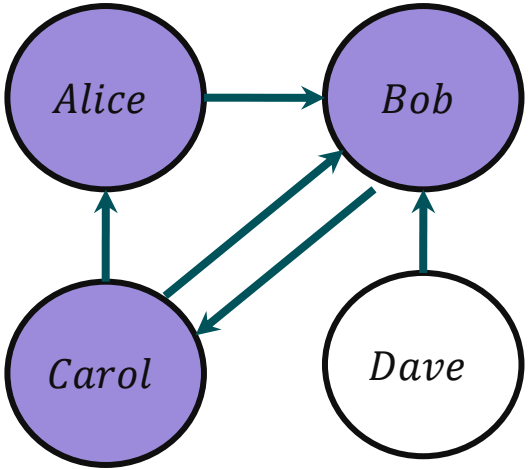
# Quorums emerge from the local agreement requirements of each validator

We call the sets that can make a decision together *quorums*



# Quorums emerge from the local agreement requirements of each validator

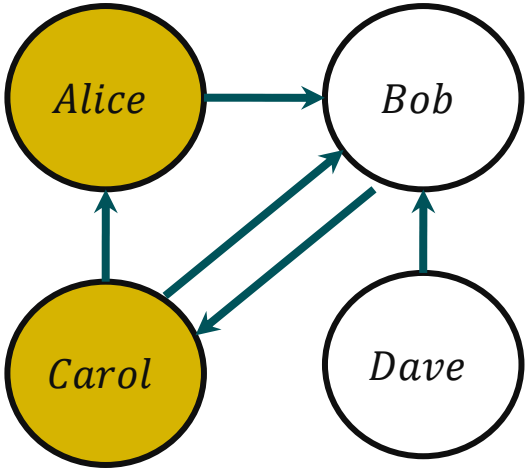
We call the sets that can make a decision together *quorums*




a quorum

# Quorums emerge from the local agreement requirements of each validator

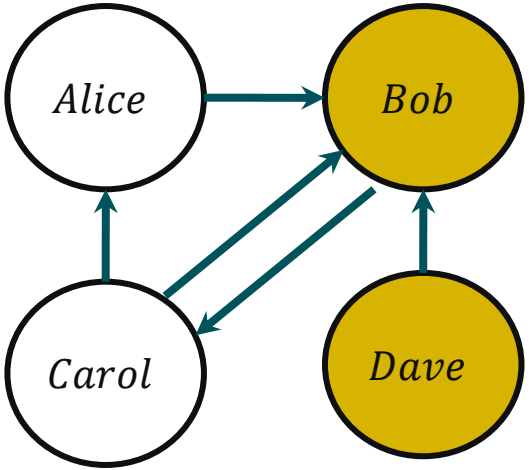
We call the sets that can make a decision together *quorums*




not a quorum

# Quorums emerge from the local agreement requirements of each validator

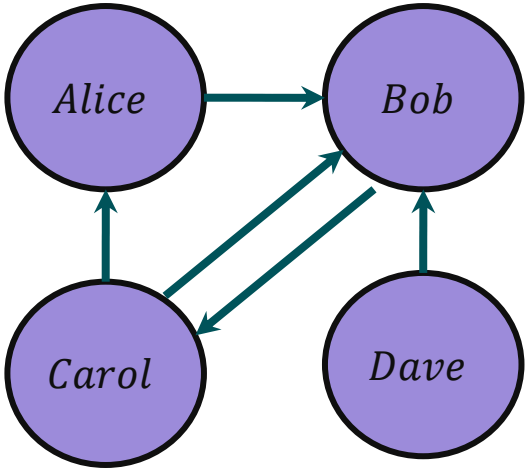
We call the sets that can make a decision together *quorums*




not a quorum

# Quorums emerge from the local agreement requirements of each validator

We call the sets that can make a decision together *quorums*




a quorum

# For fault-tolerance, we allow specifying multiple alternative agreement requirements, called quorum slices

Every validator $v$ picks a set of **quorum slices** (**slices** for short)
$\mathbf{S}_v = \{S_1, S_2, \ldots, S_N\}$ where each slice is a set of validators

$v$ requires unanimous agreement from at least one of its quorum slices

# For fault-tolerance, we allow specifying multiple alternative agreement requirements, called quorum slices

Every validator $v$ picks a set of **quorum slices** (**slices** for short)
$\mathbf{S}_v = \{S_1, S_2, \ldots, S_N\}$ where each slice is a set of validators

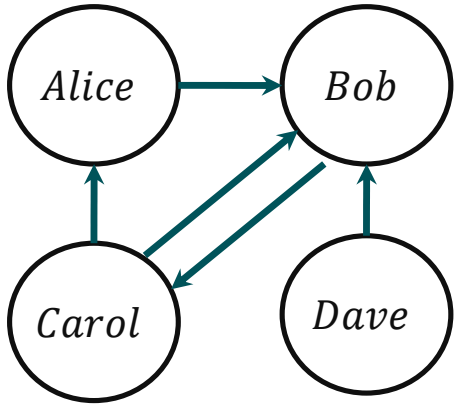
$v$ requires unanimous agreement from at least one of its quorum slices

$$\boldsymbol{S}_{Alice} = \{\{Bob\}\}$$

$$\mathbf{S}_{Bob} = \{\{Carol\}\}$$

$$\boldsymbol{S}_{Carol} = \{\{Alice, Bob\}\}$$

$$\boldsymbol{S}_{Dave} = \{\{Bob\}\}$$



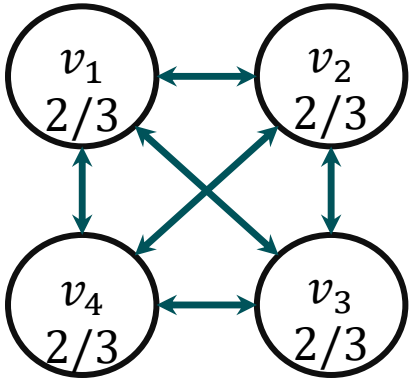
# For fault-tolerance, we allow specifying multiple alternative agreement requirements, called quorum slices

Every validator $v$ picks a set of **quorum slices** (**slices** for short)
$\mathbf{S}_v = \{S_1, S_2, \ldots, S_N\}$ where each slice is a set of validators

$v$ requires unanimous agreement from at least one of its quorum slices

$$\mathbf{S}_{v_i} = \{S \subseteq \boldsymbol{\mathcal{V}} \setminus \{v_i\}. |S| = 2\}$$

e.g. $\boldsymbol{S}_{v_1} = \{\{v_2, v_3\}, \{v_3, v_4\}, \{v_2, v_4\}\}$




2/3 means requires agreement from 2 out of the 3 outgoing edges

# Quorums emerge from quorum slices

$Q$ is a quorum when $Q \neq \emptyset$ and $Q$ contains a slice of each of its members:

# Quorums emerge from quorum slices

$Q$ is a quorum when $Q \neq \emptyset$ and $Q$ contains a slice of each of its members:

$$Q \neq \emptyset \wedge \forall v \in Q. \exists S \in \boldsymbol{S}_v. S \subseteq Q$$

# Quorums emerge from quorum slices

$Q$ is a quorum when $Q \neq \emptyset$ and $Q$ contains a slice of each of its members:

$$Q \neq \emptyset \wedge \forall v \in Q. \exists S \in \boldsymbol{S}_v. S \subseteq Q$$

Intuition: every validator in $Q$ can get the agreement it needs from inside $Q$

# Quorums emerge from quorum slices

$Q$ is a quorum when $Q \neq \emptyset$ and $Q$ contains a slice of each of its members:

$$Q \neq \emptyset \wedge \forall v \in Q. \exists S \in \boldsymbol{S}_v. S \subseteq Q$$

Intuition: every validator in $Q$ can get the agreement it needs from inside $Q$

From now on we use capital $Q$ for quorums ($Q, Q', Q_1$, etc.)

# Quorums emerge from quorum slices

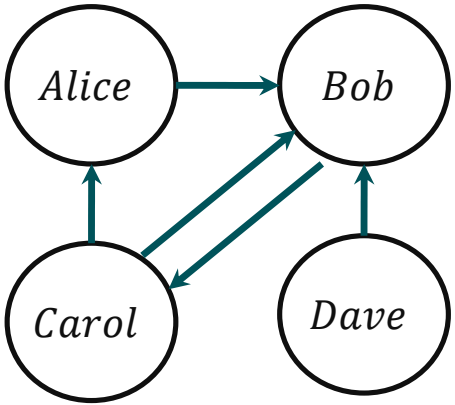
$Q$ is a quorum when $Q \neq \emptyset$ and $Q$ contains a slice of each of its members:

$$Q \neq \emptyset \wedge \forall v \in Q. \exists S \in \boldsymbol{S}_v. S \subseteq Q$$

Intuition: every validator in $Q$ can get the agreement it needs from inside $Q$

From now on we use capital $Q$ for quorums ($Q, Q', Q_1$, etc.)

$$\boldsymbol{S}_{Alice} = \{\{Bob\}\}$$
$$\boldsymbol{S}_{Bob} = \{\{Carol\}\}$$
$$\boldsymbol{S}_{Carol} = \{\{Alice, Bob\}\}$$
$$\boldsymbol{S}_{Dave} = \{\{Bob\}\}$$



# Quorums emerge from quorum slices

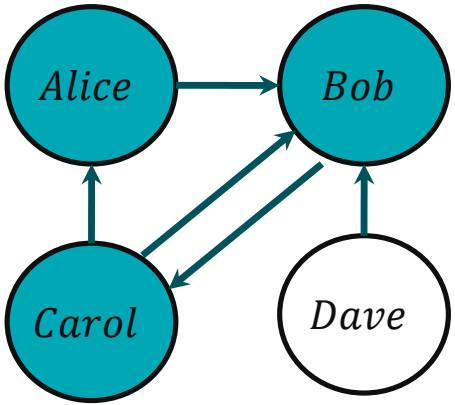
$Q$ is a quorum when $Q \neq \emptyset$ and $Q$ contains a slice of each of its members:

$$Q \neq \emptyset \wedge \forall v \in Q. \exists S \in \boldsymbol{S}_v. S \subseteq Q$$

Intuition: every validator in $Q$ can get the agreement it needs from inside $Q$

From now on we use capital $Q$ for quorums ($Q, Q', Q_1$, etc.)

$$\boldsymbol{S}_{Alice} = \{\{Bob\}\}$$
$$\boldsymbol{S}_{Bob} = \{\{Carol\}\}$$
$$\boldsymbol{S}_{Carol} = \{\{Alice, Bob\}\}$$
$$\boldsymbol{S}_{Dave} = \{\{Bob\}\}$$



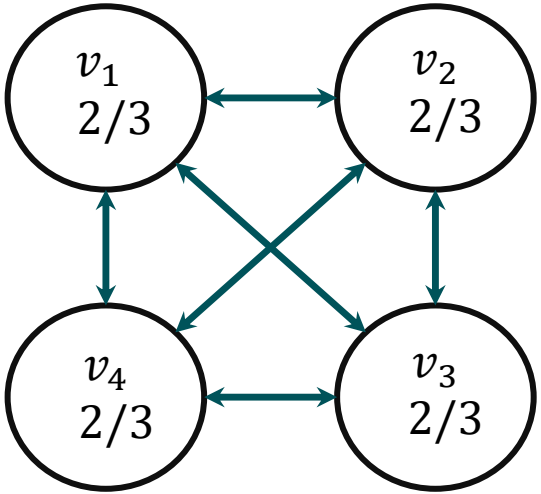

$\{Alice, Bob, Carol\}$ is the minimal quorum

$\{Alice, Bob, Carol, Dave\}$ is also a quorum

# Example 2

Take $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, each requiring agreement from 2 others:

$$\mathbf{S}_{v_i} = \{S \subseteq \boldsymbol{\mathcal{V}} \setminus \{v_i\}. |S| = 2\}$$



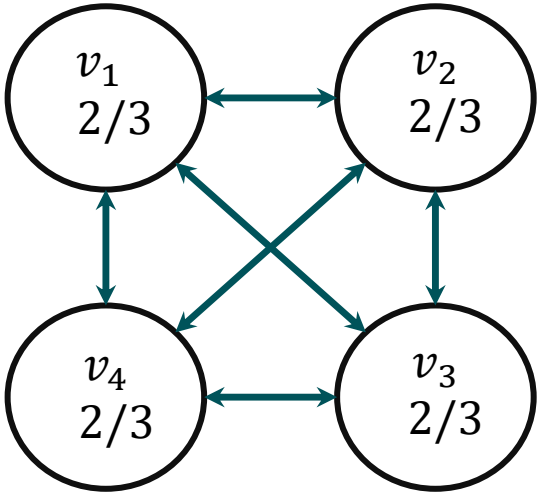

2/3 means requires agreement from 2 out of the 3 outgoing edges

# Example 2

Take $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, each requiring agreement from 2 others:

$$\mathbf{S}_{v_i} = \{S \subseteq \boldsymbol{\mathcal{V}} \setminus \{v_i\}. |S| = 2\}$$

- The quorums are the sets of at least 3 validators



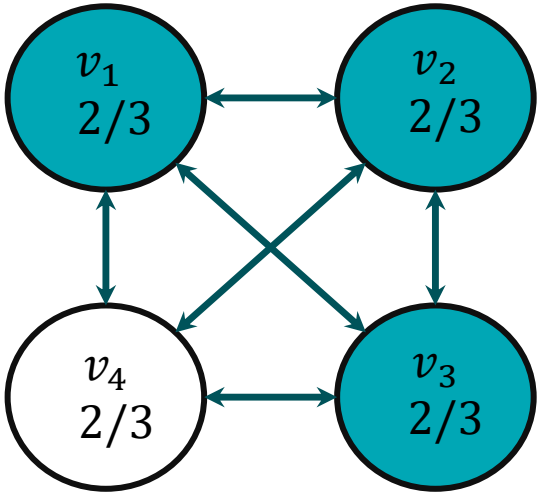

2/3 means requires agreement from 2 out of the 3 outgoing edges

# Example 2

Take $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, each requiring agreement from 2 others:

$$\mathbf{S}_{v_i} = \{S \subseteq \boldsymbol{\mathcal{V}} \setminus \{v_i\}. |S| = 2\}$$

- The quorums are the sets of at least 3 validators
  - $\{v_1, v_2, v_3\}$ is a quorum



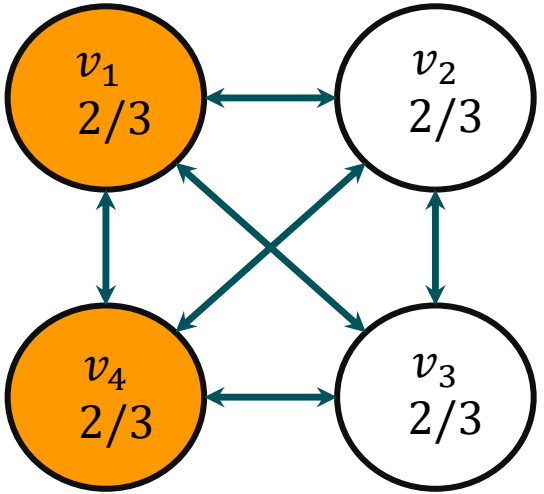

2/3 means requires agreement from 2 out of the 3 outgoing edges

# Example 2

Take $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, each requiring agreement from 2 others:

$$\mathbf{S}_{v_i} = \{S \subseteq \boldsymbol{\mathcal{V}} \setminus \{v_i\}. |S| = 2\}$$

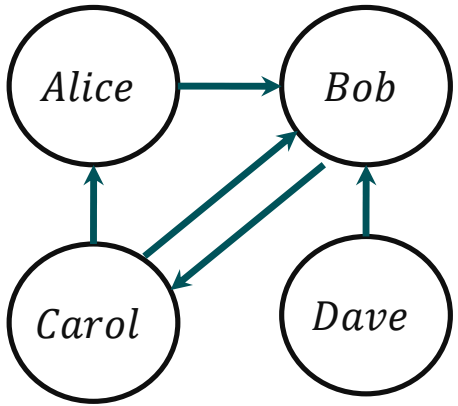
- The quorums are the sets of at least 3 validators
  - $\{v_1, v_2, v_3\}$ is a quorum
  - $\{v_1, v_2\}$ is not a quorum




2/3 means requires agreement from 2 out of the 3 outgoing edges

# Quorums are subjective and local

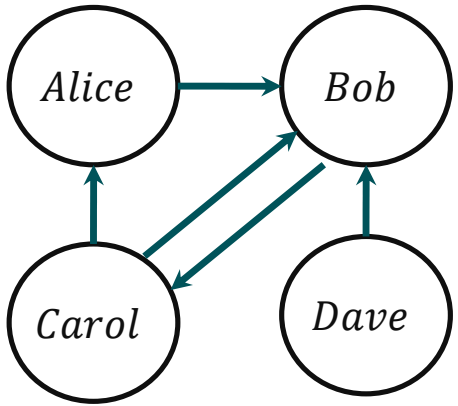
A quorum $Q$ is a ***quorum of*** a validator $v$ when $v \in Q$



# Quorums are subjective and local

A quorum $Q$ is a ***quorum of*** a validator $v$ when $v \in Q$

Intuition: if $v \in Q$, we are sure $v$ can get the agreement it needs from inside $Q$

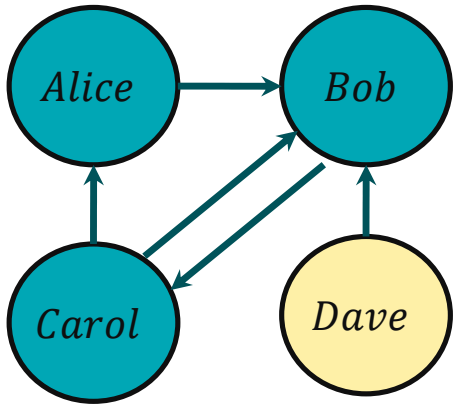


# Quorums are subjective and local

A quorum $Q$ is a ***quorum of*** a validator $v$ when $v \in Q$

Intuition: if $v \in Q$, we are sure $v$ can get the agreement it needs from inside $Q$

$\{Alice, Bob, Carol\}$ is not a quorum of $Dave$



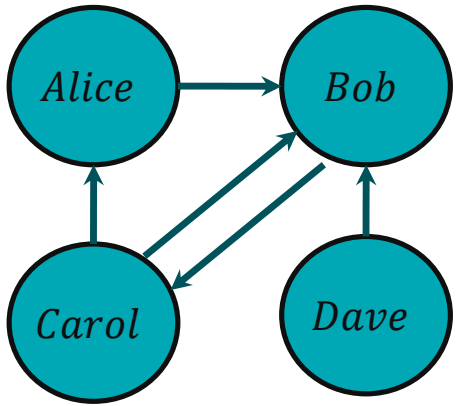
# Quorums are subjective and local

A quorum $Q$ is a ***quorum of*** a validator $v$ when $v \in Q$

Intuition: if $v \in Q$, we are sure $v$ can get the agreement it needs from inside $Q$

$\{Alice, Bob, Carol\}$ is not a quorum of $Dave$

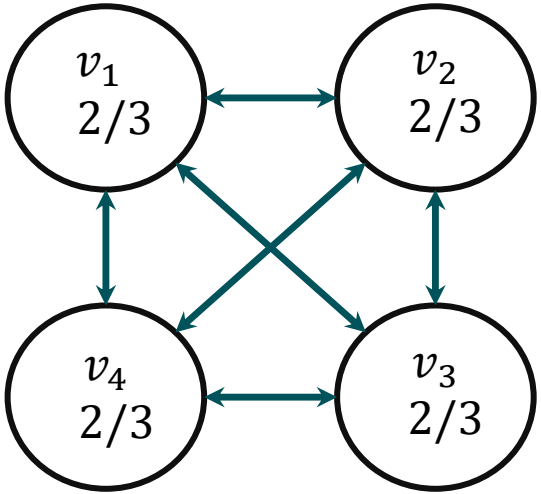
$\{Alice, Bob, Carol, Dave\}$ is a quorum of $Dave$



# Quorums are subjective and local

Take $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, each requiring agreement from 2 others:

$$\mathbf{S}_{v_i} = \{S \subseteq \boldsymbol{\mathcal{V}} \setminus \{v_i\}. |S| = 2\}$$



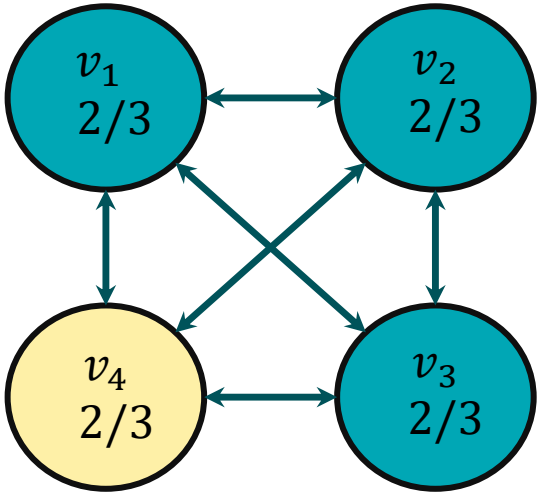

2/3 means requires agreement from 2 out of the 3 outgoing edges

# Quorums are subjective and local

Take $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, each requiring agreement from 2 others:

$$\mathbf{S}_{v_i} = \{S \subseteq \boldsymbol{\mathcal{V}} \setminus \{v_i\}. |S| = 2\}$$

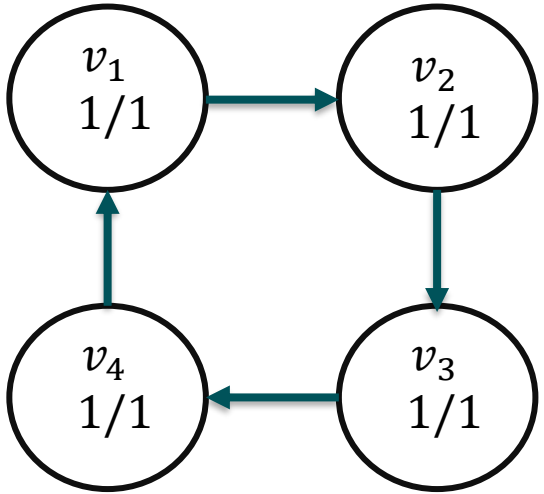
- Not all quorums are quorums of every validator
  - $\{v_1, v_2, v_3\}$ is a quorum but not a quorum of $v_4$




2/3 means requires agreement from 2 out of the 3 outgoing edges

# Example 3

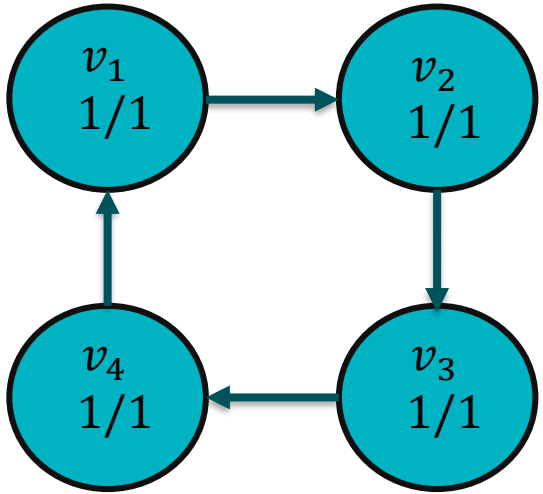
Now each validator has a single singleton slice consisting of the next validator in the ring: $\boldsymbol{S}_{v_i} = \{\{v_{(i\%4)+1}\}\}$



# Example 3

Now each validator has a single singleton slice consisting of the next validator in the ring: $\boldsymbol{S}_{v_i} = \{\{v_{(i\%4)+1}\}\}$

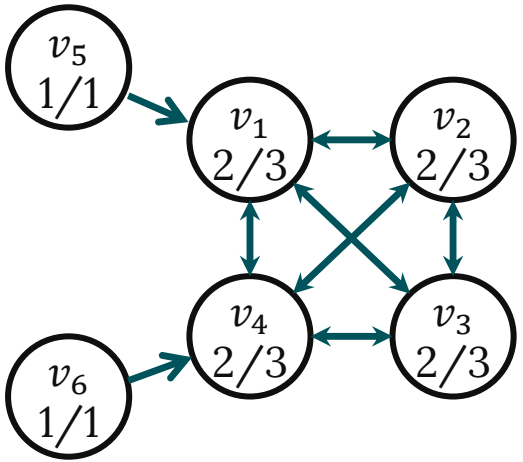
The only quorum is $\boldsymbol{\mathcal{V}}$ itself



# Example 4

We add $v_5$ and $v_6$ with
$S_{v_5} = \{\{v_1\}\}$ and $S_{v_6} = \{\{v_4\}\}$

The minimal quorums are still every 3 out of $\{v_1, v_2, v_3, v_4\}$



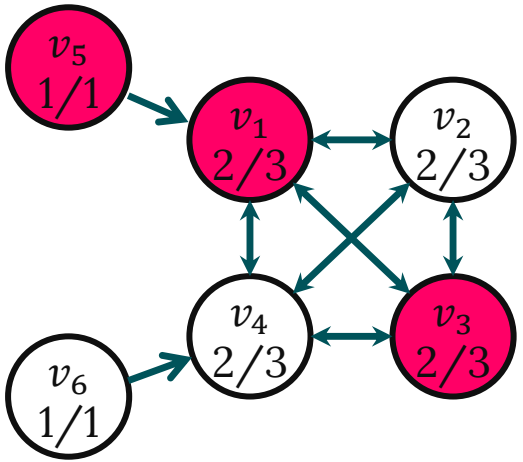
# Example 4

We add $v_5$ and $v_6$ with
$S_{v_5} = \{\{v_1\}\}$ and $S_{v_6} = \{\{v_4\}\}$

The minimal quorums are still every 3 out of $\{v_1, v_2, v_3, v_4\}$

Note:

- $\{v_5, v_1, v_3\}$ is not a quorum



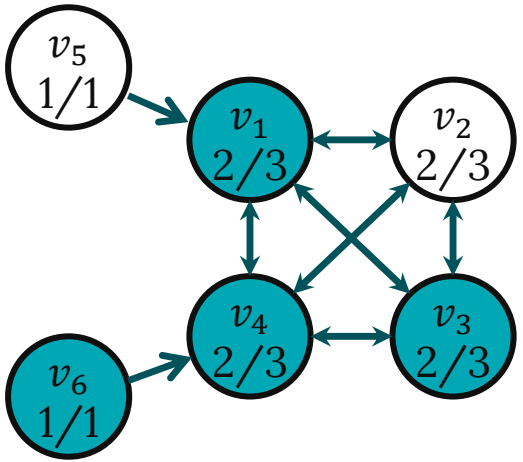
# Example 4

We add $v_5$ and $v_6$ with
$S_{v_5} = \{\{v_1\}\}$ and $S_{v_6} = \{\{v_4\}\}$

The minimal quorums are still every 3 out of $\{v_1, v_2, v_3, v_4\}$

Note:

- $\{v_5, v_1, v_3\}$ is not a quorum
- $\{v_6, v_1, v_3, v_4\}$ is a quorum of $v_6$ but not of $v_2$ or $v_5$



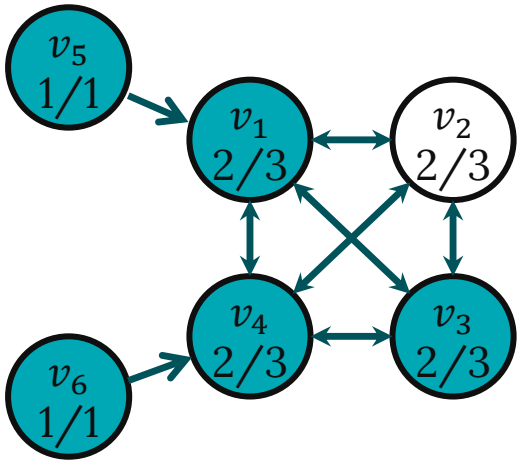
# Example 4

We add $v_5$ and $v_6$ with
$S_{v_5} = \{\{v_1\}\}$ and $S_{v_6} = \{\{v_4\}\}$

The minimal quorums are still every 3 out of $\{v_1, v_2, v_3, v_4\}$

Note:

- $\{v_5, v_1, v_3\}$ is not a quorum
- $\{v_6, v_1, v_3, v_4\}$ is a quorum of $v_6$ but not of $v_2$ or $v_5$
- $\{v_6, v_5, v_1, v_3, v_4\}$ is a quorum of both $v_5$ and $v_6$ but not of $v_2$



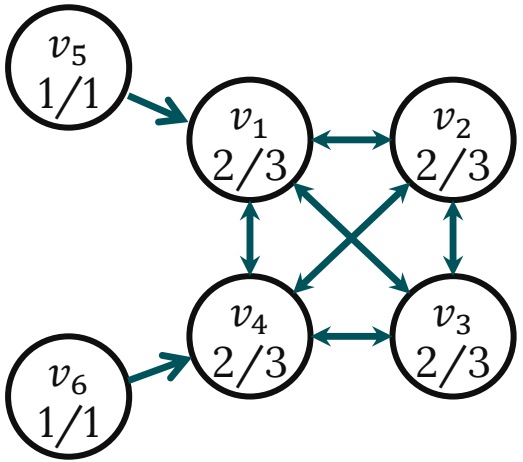
# Example 4

We add $v_5$ and $v_6$ with
$S_{v_5} = \{\{v_1\}\}$ and $S_{v_6} = \{\{v_4\}\}$

The minimal quorums are still every 3 out of $\{v_1, v_2, v_3, v_4\}$

Note:

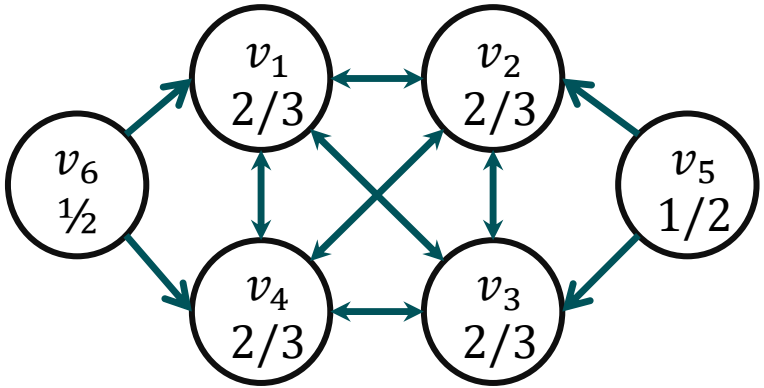
- $\{v_5, v_1, v_3\}$ is not a quorum
- $\{v_6, v_1, v_3, v_4\}$ is a quorum of $v_6$ but not of $v_2$ or $v_5$
- $\{v_6, v_5, v_1, v_3, v_4\}$ is a quorum of both $v_5$ and $v_6$ but not of $v_2$




Note $v_5$ and $v_6$ have disjoint slices but all their quorums intersect

# Example 5

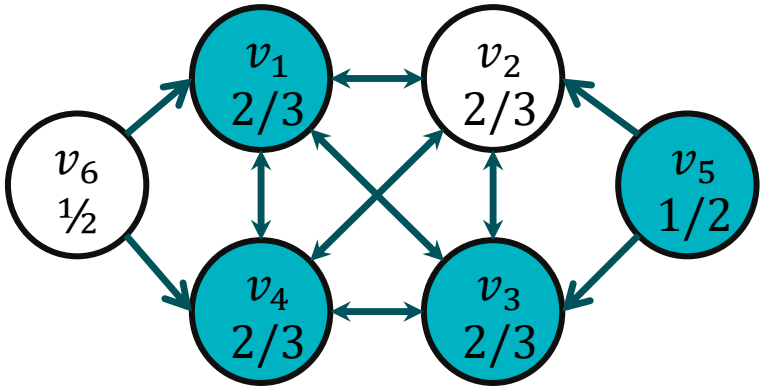
Now assume $\boldsymbol{S}_{v_5} = \{\{v_1\}, \{v_4\}\}$ and $\boldsymbol{S}_{v_6} = \{\{v_2\}, \{v_3\}\}$



# Example 5

Now assume $\boldsymbol{S}_{v_5} = \{\{v_1\}, \{v_4\}\}$ and $\boldsymbol{S}_{v_6} = \{\{v_2\}, \{v_3\}\}$

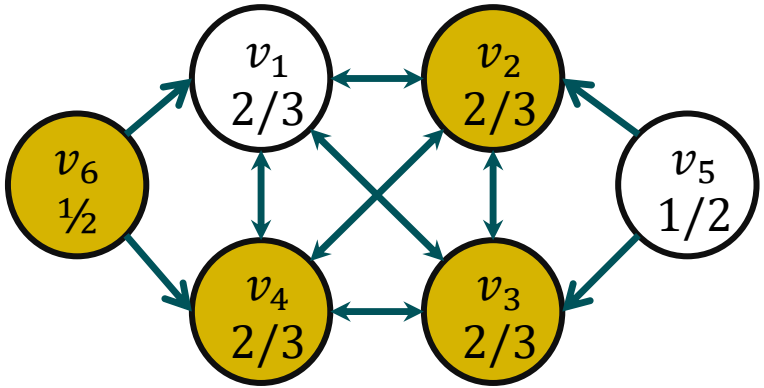
- $\{v_5, v_1, v_3, v_4\}$ is a quorum of $v_5$



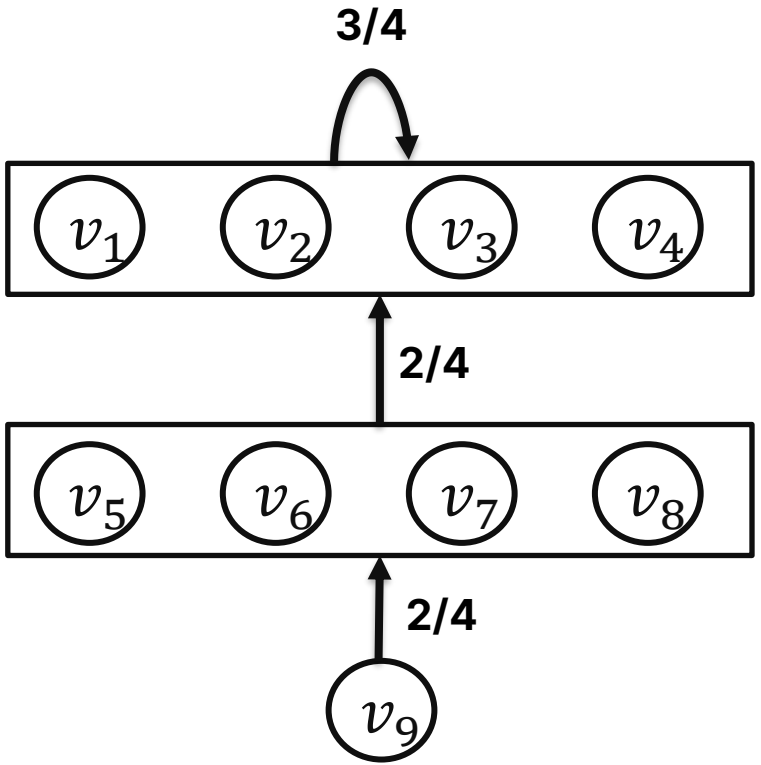
# Example 5

Now assume $\boldsymbol{S}_{v_5} = \{\{v_1\}, \{v_4\}\}$ and $\boldsymbol{S}_{v_6} = \{\{v_2\}, \{v_3\}\}$

- $\{v_5, v_1, v_3, v_4\}$ is a quorum of $v_5$
- $\{v_6, v_4, v_2, v_3\}$ is a quorum of $v_6$

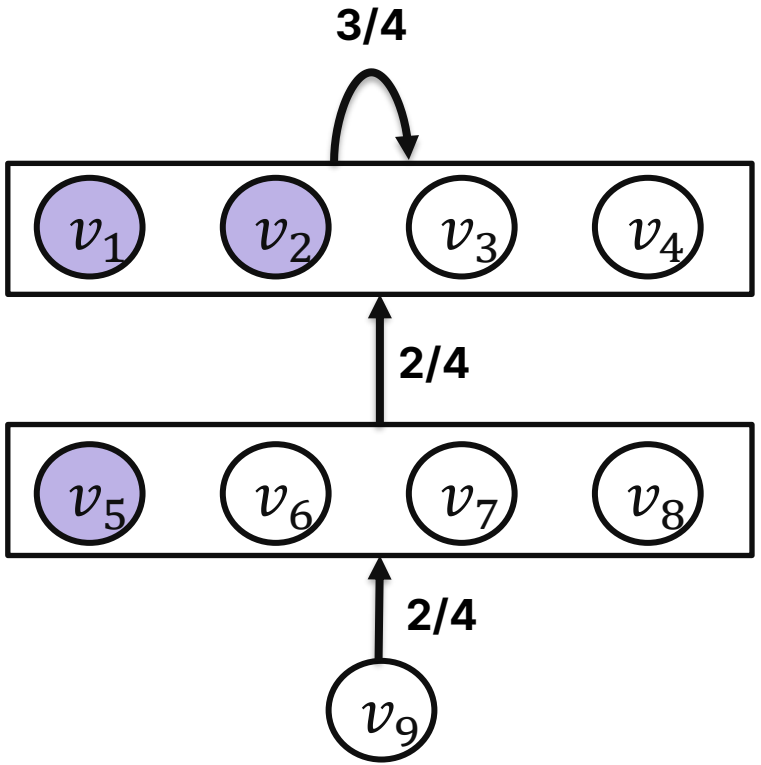


# Example 6



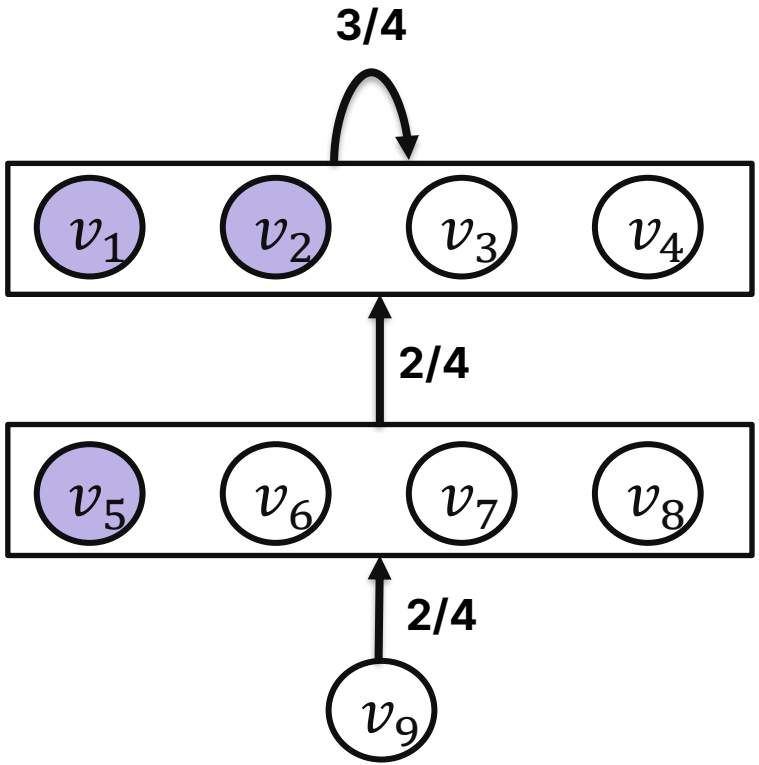
# Example 6

- Is $\{v_5, v_1, v_2\}$ a quorum?



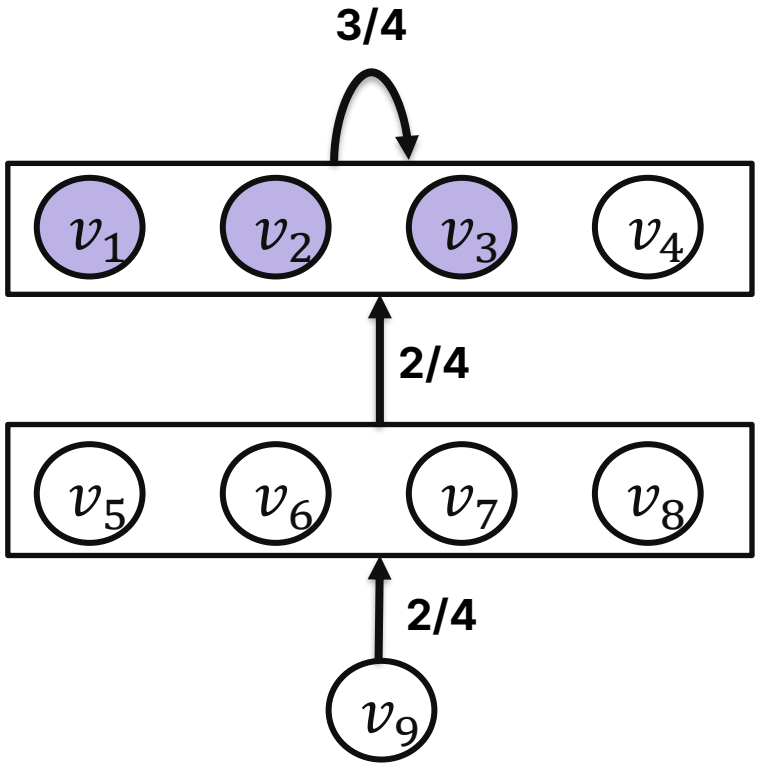
# Example 6

- Is $\{v_5, v_1, v_2\}$ a quorum?
  - No, $v_1$ and $v_2$ do not have slices in $\{v_5, v_1, v_2\}$



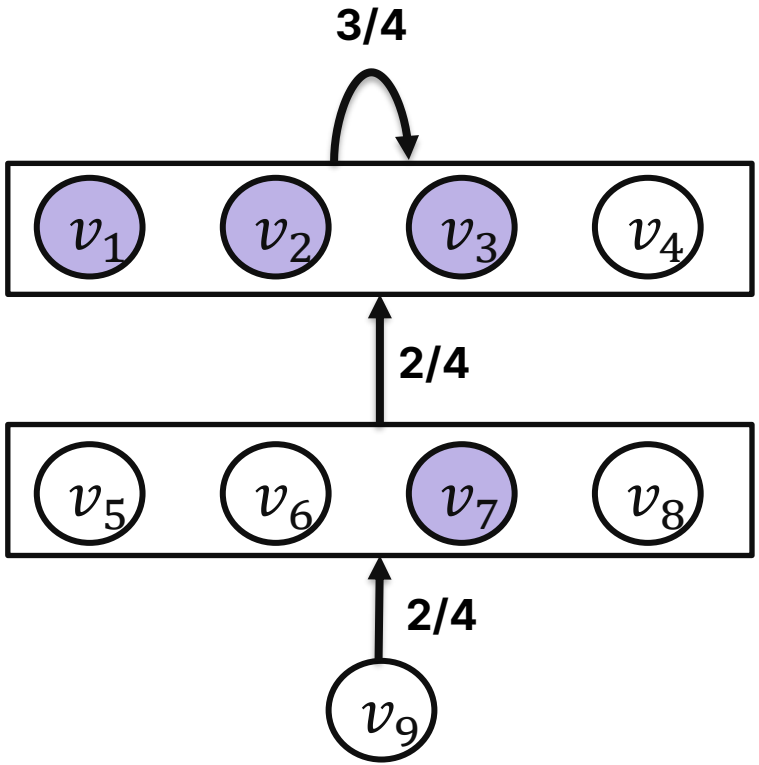
# Example 6

- Is $\{v_5, v_1, v_2\}$ a quorum?
  - No, $v_1$ and $v_2$ do not have slices in $\{v_5, v_1, v_2\}$
- $\{v_1, v_2, v_3\}$ is a quorum



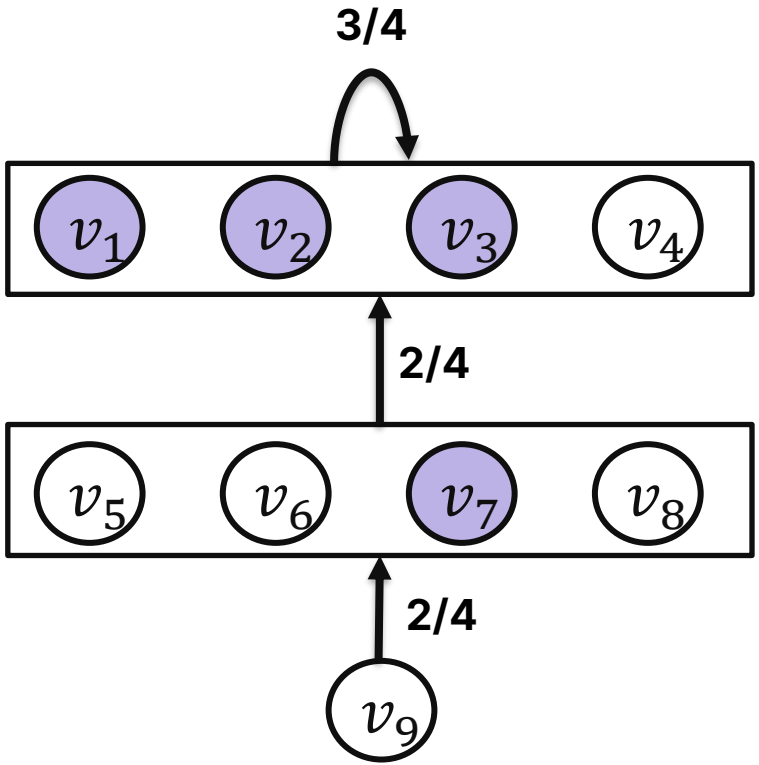
# Example 6

- Is $\{v_5, v_1, v_2\}$ a quorum?
  - No, $v_1$ and $v_2$ do not have slices in $\{v_5, v_1, v_2\}$
- $\{v_1, v_2, v_3\}$ is a quorum
- Is $\{v_1, v_2, v_3, v_7\}$ a quorum?



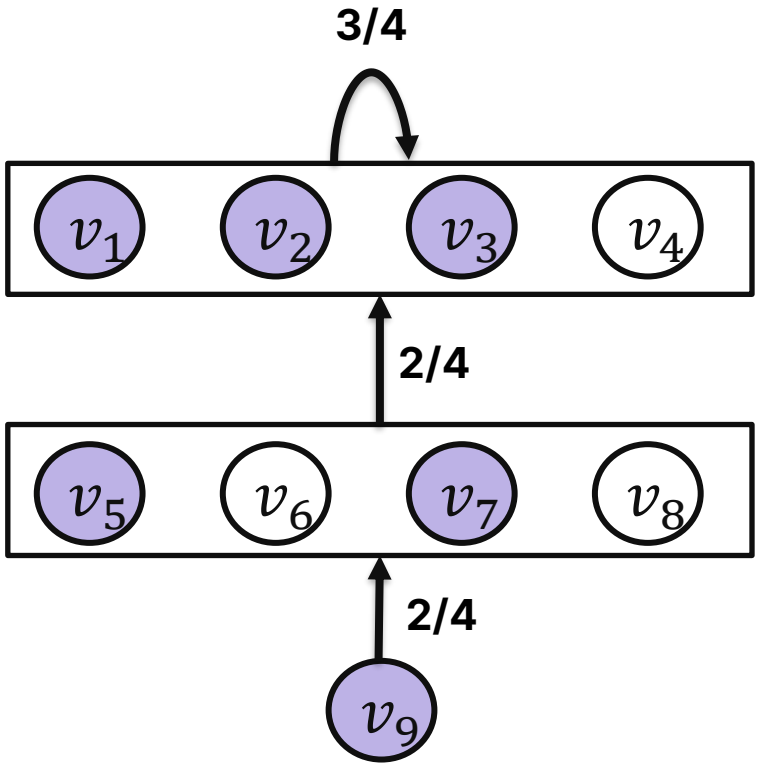
# Example 6

- Is $\{v_5, v_1, v_2\}$ a quorum?
  - No, $v_1$ and $v_2$ do not have slices in $\{v_5, v_1, v_2\}$
- $\{v_1, v_2, v_3\}$ is a quorum
- Is $\{v_1, v_2, v_3, v_7\}$ a quorum?
  - Yes



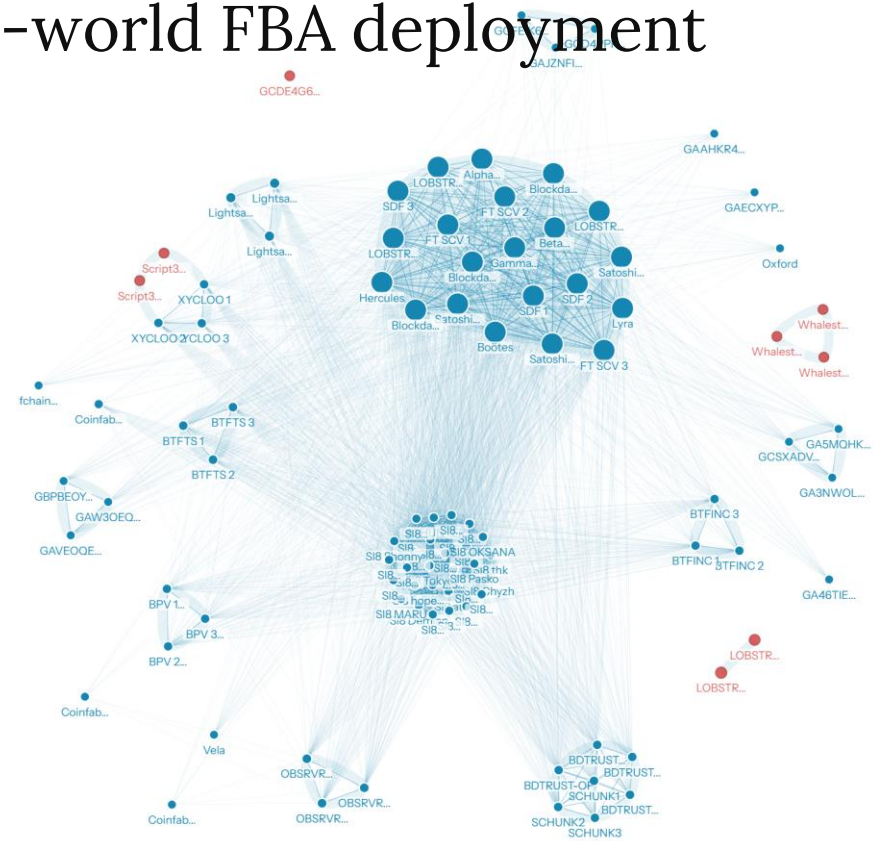
# Example 6

- Is $\{v_5, v_1, v_2\}$ a quorum?
  - No, $v_1$ and $v_2$ do not have slices in $\{v_5, v_1, v_2\}$
- $\{v_1, v_2, v_3\}$ is a quorum
- Is $\{v_1, v_2, v_3, v_7\}$ a quorum?
  - Yes
- $\{v_1, v_2, v_3, v_5, v_7, v_8\}$ is a quorum



# The Stellar network is a real-world FBA deployment

Currently 92 validators belonging to 17 organizations.



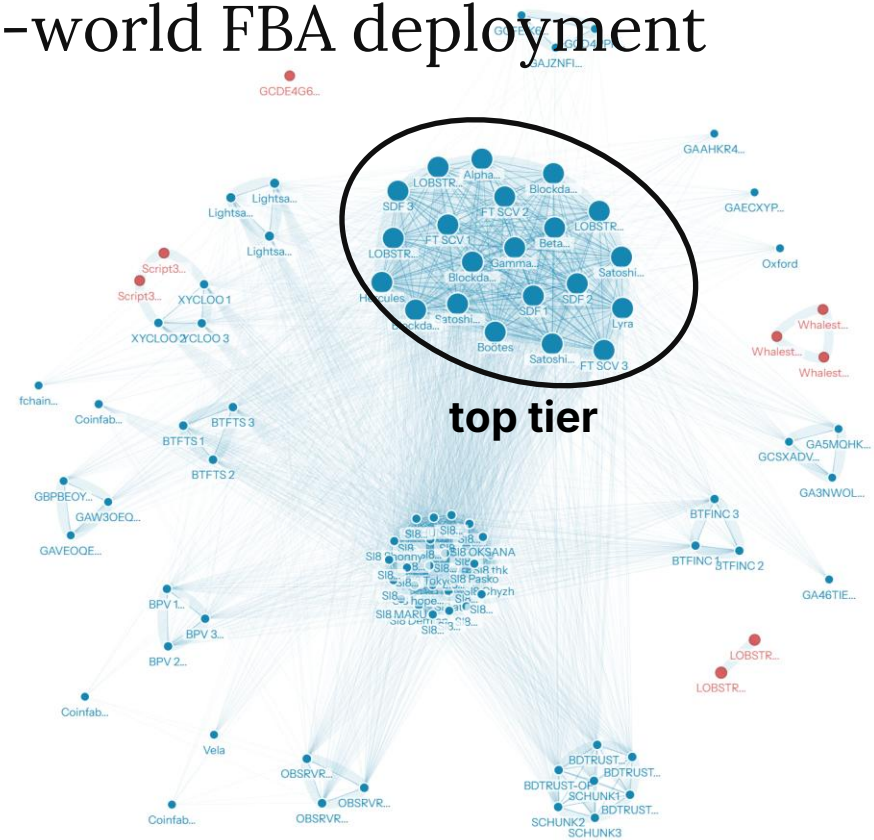

https://radar.withobsrvr.com

# The Stellar network is a real-world FBA deployment

Currently 92 validators belonging to 17 organizations.

7 “top-tier” organizations, each running 3 validators, require agreement from each other

Other organizations, classified as second tier, require agreement from the top tier and/or a few other second-tier organizations



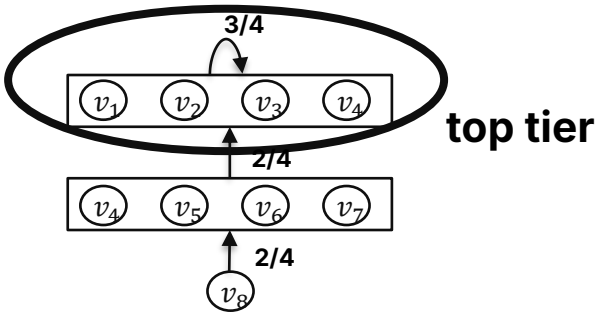

https://radar.withobsrvr.com

# The Stellar network is a real-world FBA deployment

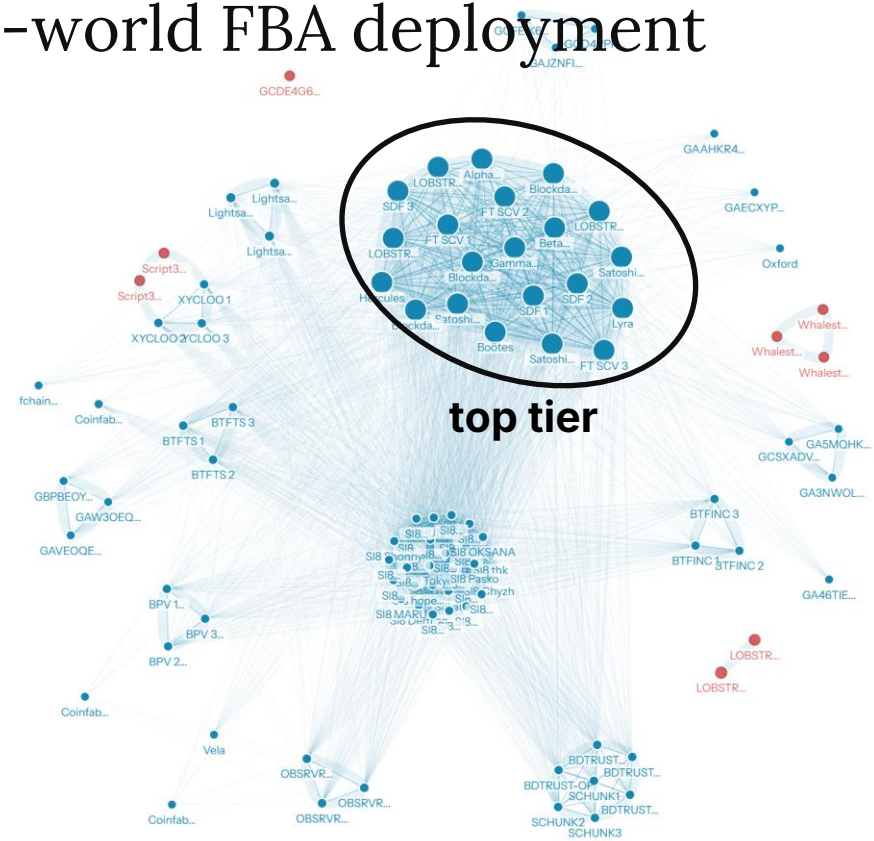
Currently 92 validators belonging to 17 organizations.

7 “top-tier” organizations, each running 3 validators, require agreement from each other

Other organizations, classified as second tier, require agreement from the top tier and/or a few other second-tier organizations




top tier




https://radar.withobsrvr.com

# Issuer-enforced finality in the Stellar network and the Internet Hypothesis

- In practice today, few asset issuers run validators (only one major issuer runs validators)...
- Nevertheless, validators have incentives to require agreement with other validators they deem trustworthy to ensure quorum intersection
- The Internet Hypothesis: like the Internet, a global FBA system will be highly connected and fault tolerant
  - Currently, all quorums intersect with some margin (3 organizations)

# Deployment experience

- Deployed in 2015
- For a few years, all validators just configured their quorum slices as ½ of the validators of the Stellar Development Foundation (SDF)
- The SDF pushed the community towards decentralization, but this led to a problem in 2019:
  - Many validators picked quorum slices that were too big
  - A popular monitoring tool underreported validator down-time
  - ➢ Unreliable validators caused the whole system to stop because no quorum was available
  - ➢ Chaotic, manual reconfiguration ensued and the system lost quorum intersection and validators diverged, requiring manual restart to a know good state
- As a result, SDF implemented a slice-generator tool, and monitoring tools improved

# The simplified slice generator

# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

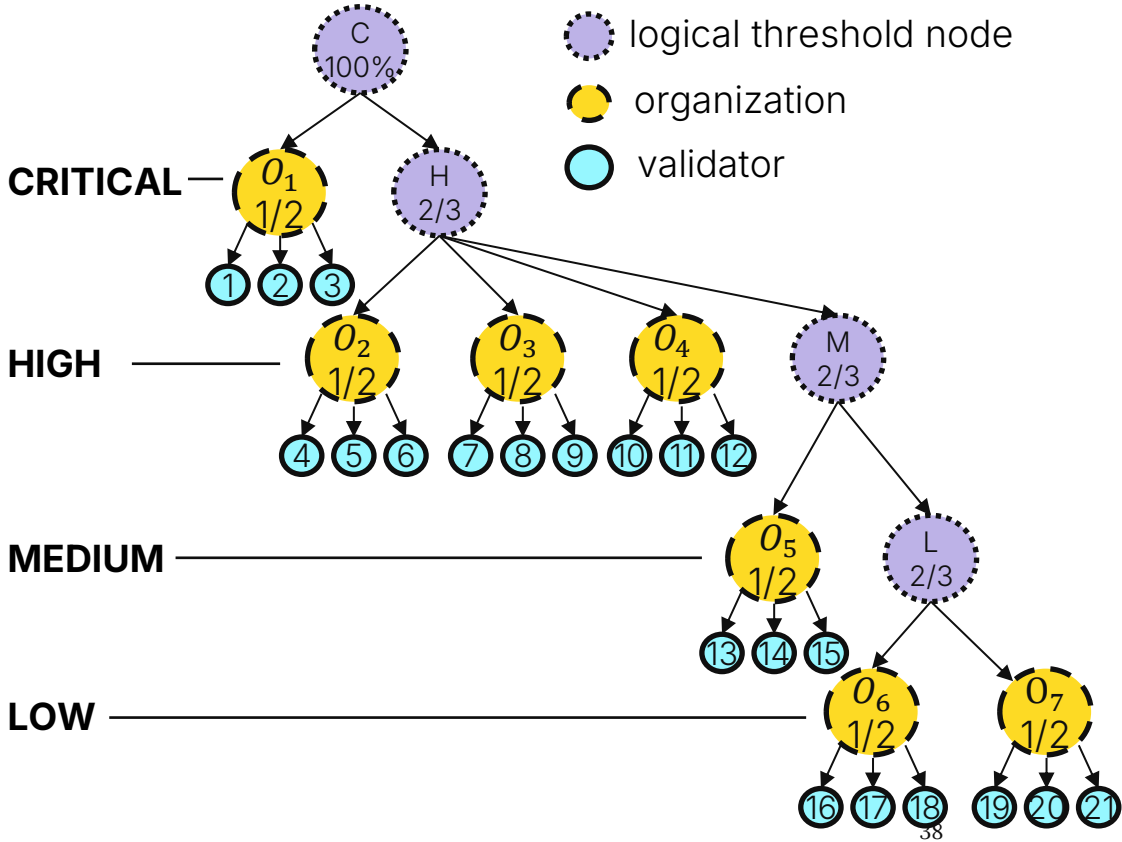
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



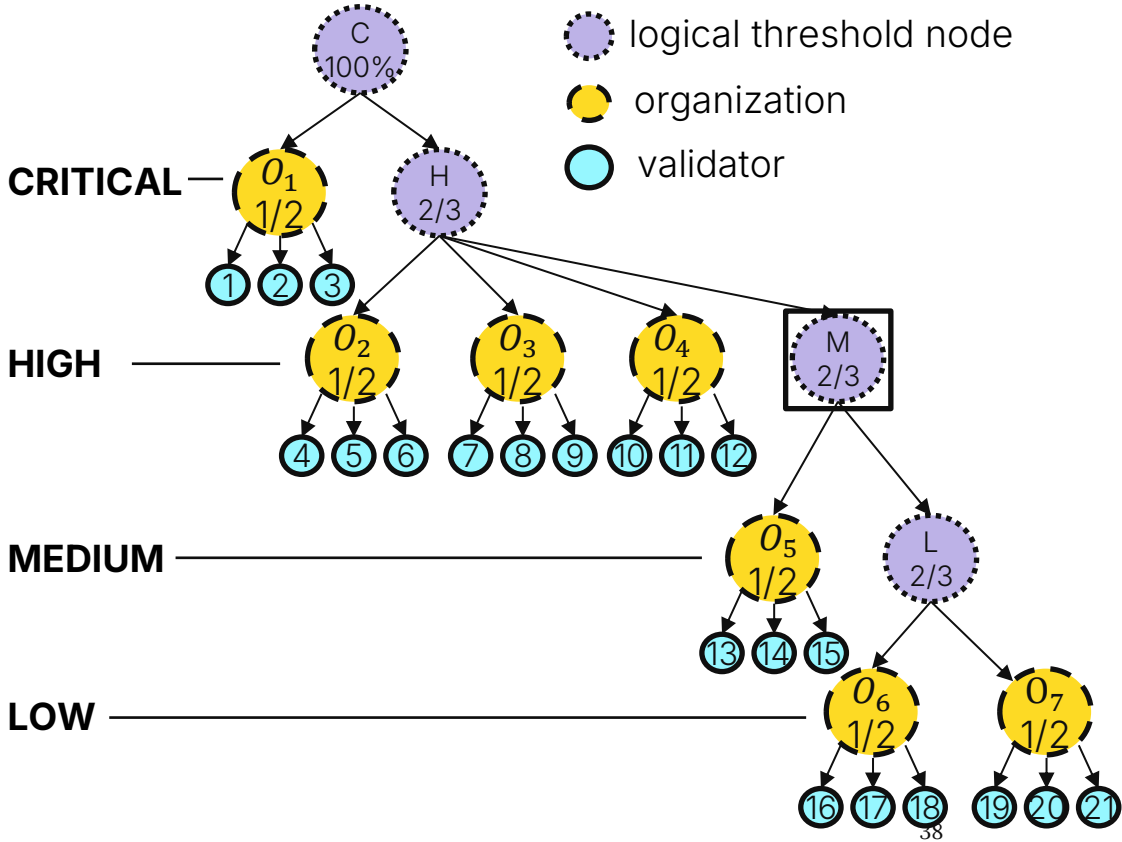
# The simplified slice generator

Organizations run multiple validators (typically 3)

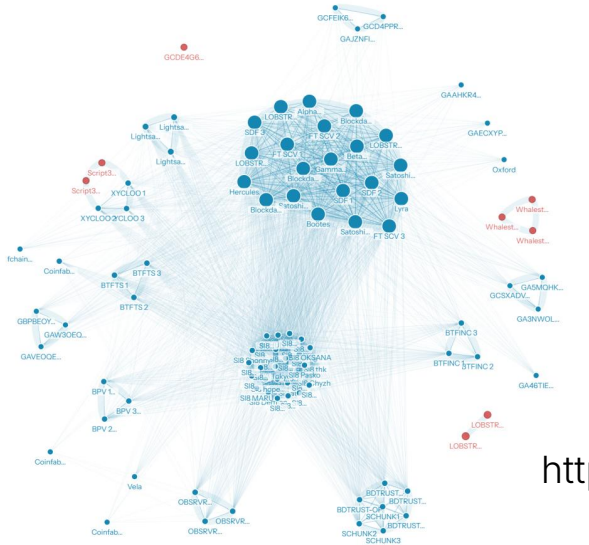
A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

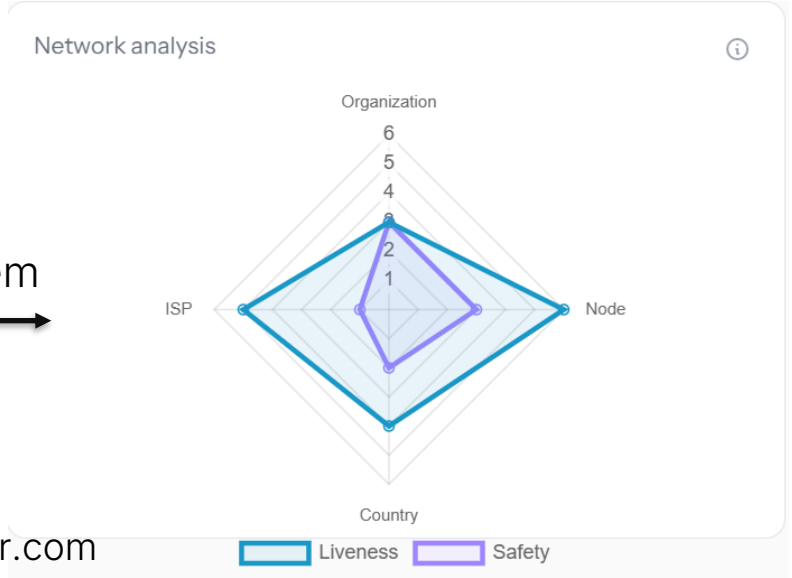
The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



# We can compute the minimal “splitting sets” and “blocking sets”



NP-hard problem




https://radar.withobsrvr.com

Uses search algorithms from
https://github.com/trudi-group/fbas_analyzer

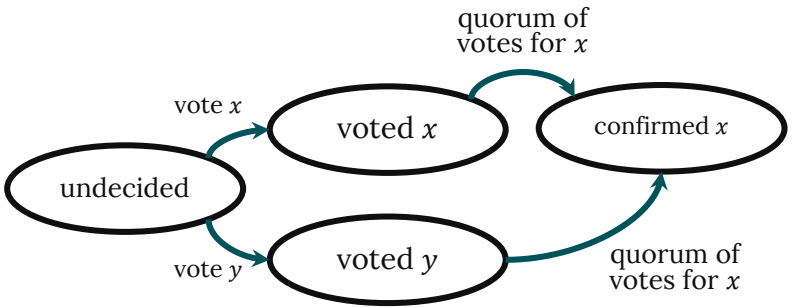
Alternative SAT-based implementation at
https://github.com/nano-o/python-fbas

# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ✓ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- ✓ Quorums in Federated Byzantine Agreement systems
- ➢ Reliable Voting in FBA systems
  - ➢ A simple, safe straw-man protocol
  - Quorum intersection is not that simple
  - Consensus clusters
  - Obtaining Totality: the Federated Voting protocol
- Total-Order Broadcast by porting Simplex to FBA

# How do we implement Reliable Voting in FBA?

Assuming quorum intersection and availability, can we reuse the Reliable Voting algorithm to ensure Unanimity, Validity, Agreement, Totality?



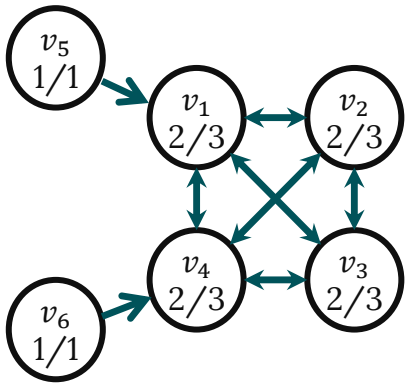

+ Re-broadcast all received votes

Does this algorithm still work?

# Two initial issues

How do validators learn each other's slices?

Slices are local and ***a priori*** unknown to other validators




How do validators compute quorums from slices?

We have a definition, not an algorithm

$$\mathrm{v} \in \mathrm{Q} \wedge \forall v' \in Q. \exists S \in \boldsymbol{S}_{v\prime}. S \subseteq Q$$

# Checking whether a quorum unanimously voted for a value

# Checking whether a quorum unanimously voted for a value

- Validators attach their slices to every message they send

# Checking whether a quorum unanimously voted for a value

- Validators attach their slices to every message they send
- For each value, a validator remembers the latest vote message received for that value from each validator

# Checking whether a quorum unanimously voted for a value

- Validators attach their slices to every message they send
- For each value, a validator remembers the latest vote message received for that value from each validator
- Upon receiving a new vote for a value $x$, we proceed by elimination:

# Checking whether a quorum unanimously voted for a value

- Validators attach their slices to every message they send
- For each value, a validator remembers the latest vote message received for that value from each validator
- Upon receiving a new vote for a value $x$, we proceed by elimination:
  1. Let $candidate$ be the set of validators that we know voted for $x$

# Checking whether a quorum unanimously voted for a value

- Validators attach their slices to every message they send
- For each value, a validator remembers the latest vote message received for that value from each validator
- Upon receiving a new vote for a value $x$, we proceed by elimination:
  1. Let $candidate$ be the set of validators that we know voted for $x$
  2. Let $remaining$ be the subset of $candidates$ that have a slice included in $candidates$

# Checking whether a quorum unanimously voted for a value

- Validators attach their slices to every message they send
- For each value, a validator remembers the latest vote message received for that value from each validator
- Upon receiving a new vote for a value $x$, we proceed by elimination:
  1. Let $candidate$ be the set of validators that we know voted for $x$
  2. Let $remaining$ be the subset of $candidates$ that have a slice included in $candidates$
  3. If $\text{remaining} = candidates$, we have a quorum

# Checking whether a quorum unanimously voted for a value

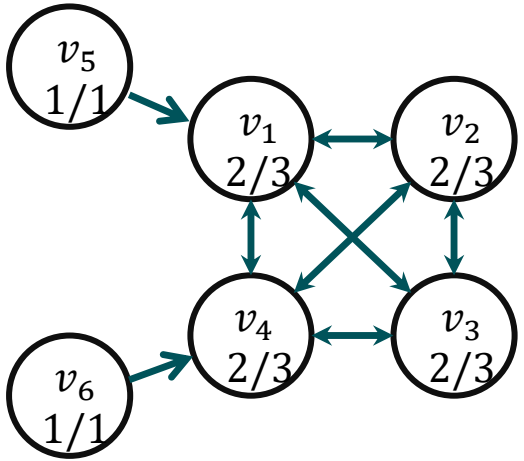
- Validators attach their slices to every message they send
- For each value, a validator remembers the latest vote message received for that value from each validator
- Upon receiving a new vote for a value $x$, we proceed by elimination:
  1. Let $candidate$ be the set of validators that we know voted for $x$
  2. Let $remaining$ be the subset of $candidates$ that have a slice included in $candidates$
  3. If $\mathbf{remaining} = candidates$, we have a quorum
     1. return $self \in candidates$

# Checking whether a quorum unanimously voted for a value

- Validators attach their slices to every message they send
- For each value, a validator remembers the latest vote message received for that value from each validator
- Upon receiving a new vote for a value $x$, we proceed by elimination:
  1. Let $candidate$ be the set of validators that we know voted for $x$
  2. Let $remaining$ be the subset of $candidates$ that have a slice included in $candidates$
  3. If $\mathbf{remaining} = candidates$, we have a quorum
     1. return $self \in candidates$
  4. Else set $candidates := remaining$ and go to 2

# Example 1

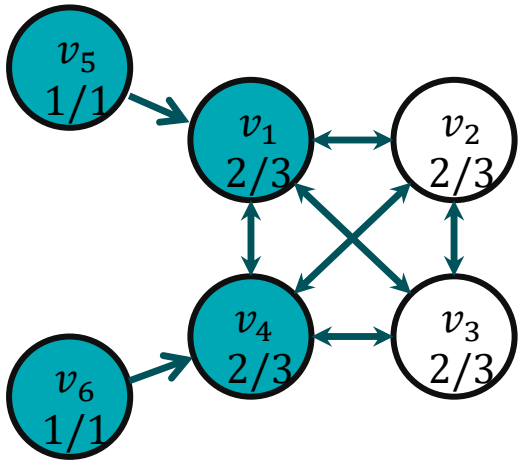
Consider at $v_5$:



# Example 1

Consider at $v_5$:

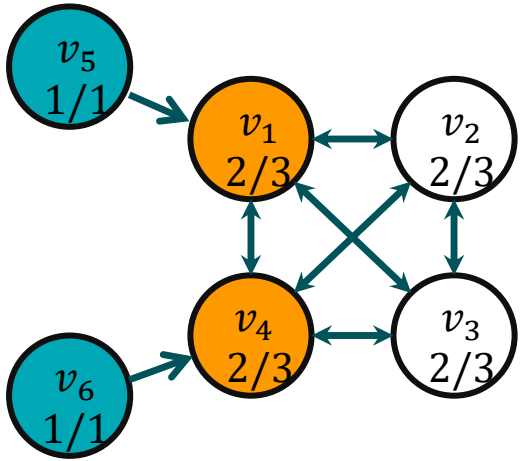
- $candidates = \{v_6, v_5, v_1, v_4\}$



# Example 1

Consider at $v_5$:

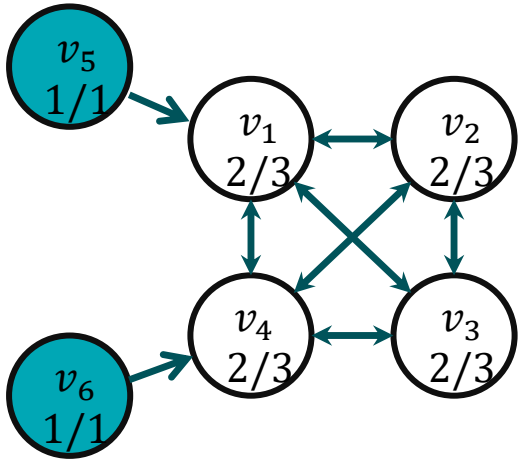
- $candidates = \{v_6, v_5, v_1, v_4\}$
  - $\{v_1\}$ and $\{v_4\}$ have no slice in $\{v_6, v_5, v_1, v_4\}$



# Example 1

Consider at $v_5$:

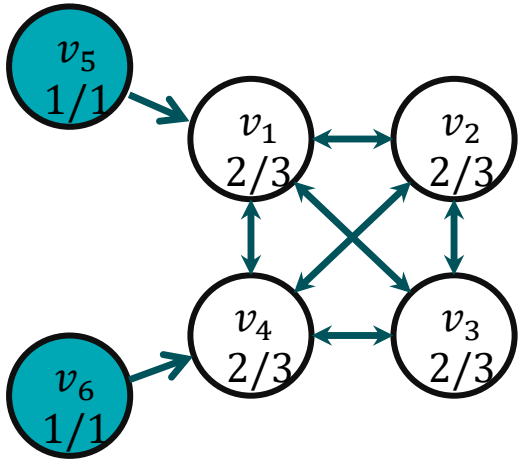
- $candidates = \{v_6, v_5, v_1, v_4\}$
  - $\{v_1\}$ and $\{v_4\}$ have no slice in $\{v_6, v_5, v_1, v_4\}$
  - $remaining = \{v_6, v_5\} \neq candidates$



# Example 1

Consider at $v_5$:

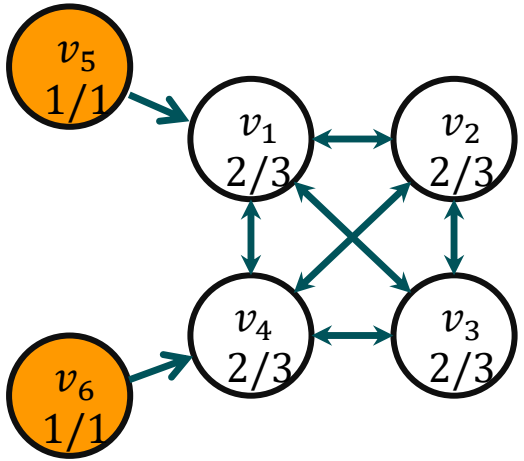
- $candidates = \{v_6, v_5, v_1, v_4\}$
  - $\{v_1\}$ and $\{v_4\}$ have no slice in $\{v_6, v_5, v_1, v_4\}$
  - $remaining = \{v_6, v_5\} \neq candidates$
- $candidates = \{v_6, v_5\}$



# Example 1

Consider at $v_5$:

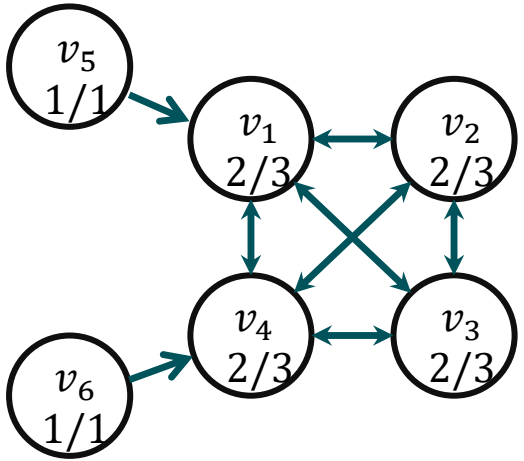
- $candidates = \{v_6, v_5, v_1, v_4\}$
  - $\{v_1\}$ and $\{v_4\}$ have no slice in $\{v_6, v_5, v_1, v_4\}$
  - $remaining = \{v_6, v_5\} \neq candidates$
- $candidates = \{v_6, v_5\}$
  - $\{v_6\}$ and $\{v_5\}$ have no slice in $\{v_6, v_5\}$



# Example 1

Consider at $v_5$:

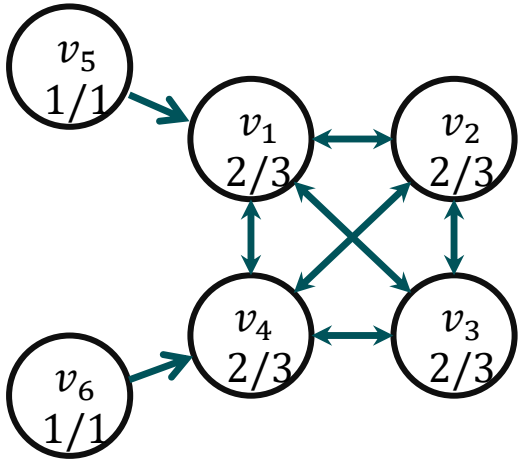
- $candidates = \{v_6, v_5, v_1, v_4\}$
  - $\{v_1\}$ and $\{v_4\}$ have no slice in $\{v_6, v_5, v_1, v_4\}$
  - $remaining = \{v_6, v_5\} \neq candidates$
- $candidates = \{v_6, v_5\}$
  - $\{v_6\}$ and $\{v_5\}$ have no slice in $\{v_6, v_5\}$
  - $remaining = \emptyset \neq candidates$



# Example 1

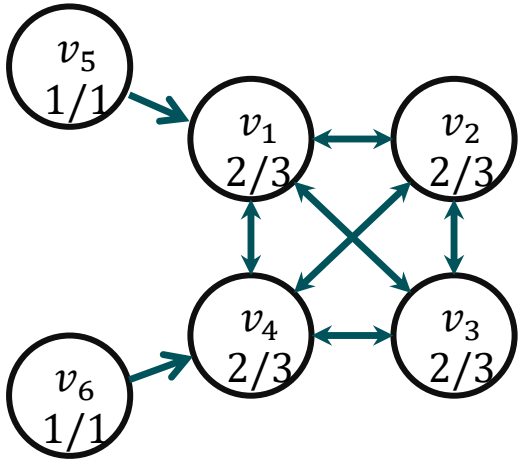
Consider at $v_5$:

- $candidates = \{v_6, v_5, v_1, v_4\}$
  - $\{v_1\}$ and $\{v_4\}$ have no slice in $\{v_6, v_5, v_1, v_4\}$
  - $remaining = \{v_6, v_5\} \neq candidates$
- $candidates = \{v_6, v_5\}$
  - $\{v_6\}$ and $\{v_5\}$ have no slice in $\{v_6, v_5\}$
  - $remaining = \emptyset \neq candidates$
- fixpoint ∅ reached
  - $\{v_5\}$ has no slice included in ∅
  - return false



# Example 2

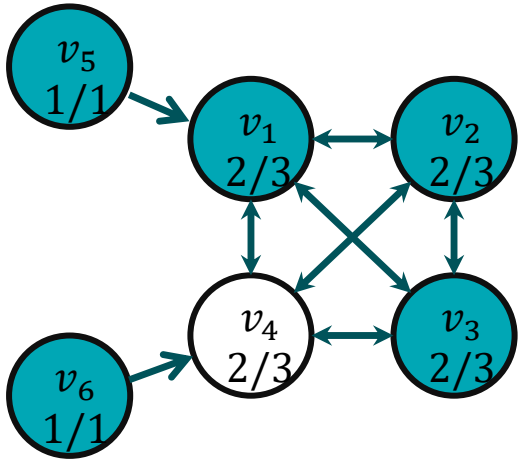
Consider at $v_5$:



# Example 2

Consider at $v_5$:

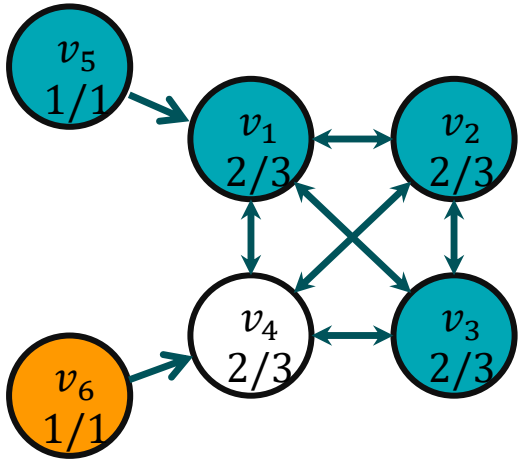
- $candidates = \{v_6, v_5, v_1, v_2, v_3\}$



# Example 2

Consider at $v_5$:

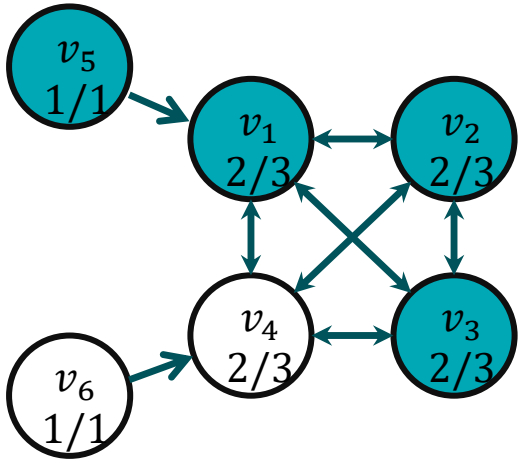
- $candidates = \{v_6, v_5, v_1, v_2, v_3\}$
  - $\{v_6\}$ has no slice in $\{v_6, v_5, v_1, v_2, v_3\}$



# Example 2

Consider at $v_5$:

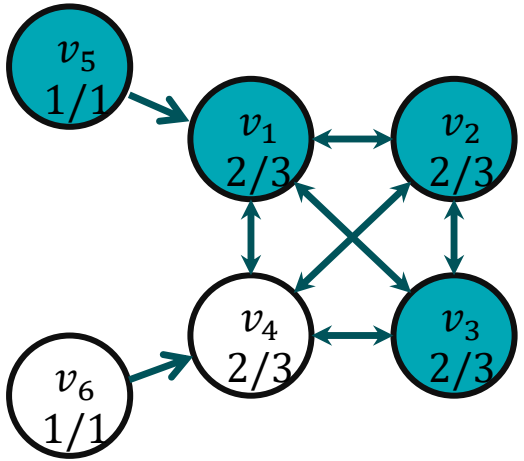
- $candidates = \{v_6, v_5, v_1, v_2, v_3\}$
  - $\{v_6\}$ has no slice in $\{v_6, v_5, v_1, v_2, v_3\}$
  - $remaining = \{v_5, v_1, v_2, v_3\} \neq candidates$



# Example 2

Consider at $v_5$:

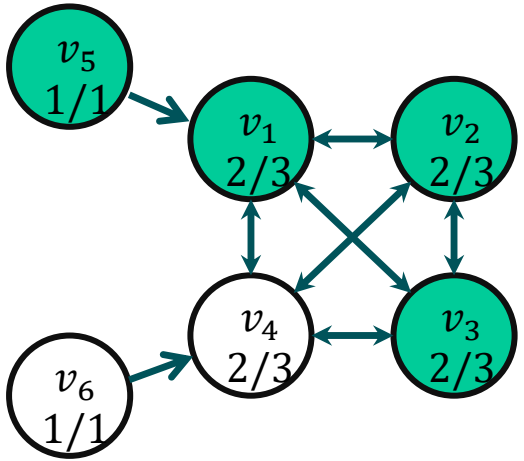
- $candidates = \{v_6, v_5, v_1, v_2, v_3\}$
  - $\{v_6\}$ has no slice in $\{v_6, v_5, v_1, v_2, v_3\}$
  - $remaining = \{v_5, v_1, v_2, v_3\} \neq candidates$
- $candidates = \{v_5, v_1, v_2, v_3\}$
  - $\{v_5, v_1, v_2, v_3\}$ all have a slice in $\{v_5, v_1, v_2, v_3\}$
  - $remaining = \{v_5, v_1, v_2, v_3\} = candidates$



# Example 2

Consider at $v_5$:

- $candidates = \{v_6, v_5, v_1, v_2, v_3\}$
  - $\{v_6\}$ has no slice in $\{v_6, v_5, v_1, v_2, v_3\}$
  - $remaining = \{v_5, v_1, v_2, v_3\} \neq candidates$
- $candidates = \{v_5, v_1, v_2, v_3\}$
  - $\{v_5, v_1, v_2, v_3\}$ all have a slice in $\{v_5, v_1, v_2, v_3\}$
  - $remaining = \{v_5, v_1, v_2, v_3\} = candidates$
- fixpoint reached
  - $\{v_5\}$ has a slice included in$\{v_5, v_1, v_2, v_3\}$
  - return true



# We solved our two initial issues

## How do validators learn each other's slices?

Validators attach their slices to every message they send

## How do validators compute quorums from slices?

We check whether a set of vote messages reached quorum threshold using a linear-time elimination procedure

# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ✓ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- ✓ Quorums in Federated Byzantine Agreement systems
- ➢ Reliable Voting in FBA systems
  - ✓ A simple, safe straw-man protocol
  - ➢ Quorum intersection is not that simple
  - Consensus clusters
  - Obtaining Totality: the Federated Voting protocol
- Total-Order Broadcast by porting Simplex to FBA

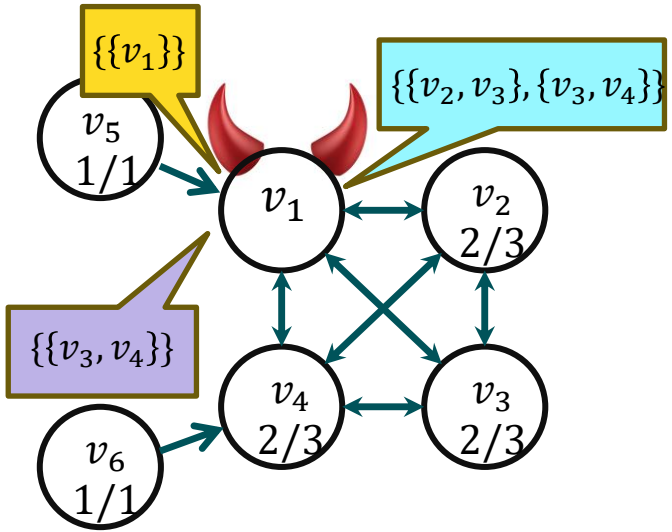
# Does quorum intersection suffice to ensure Agreement?

# Does quorum intersection suffice to ensure Agreement?

- We use slices received over the network to compute whether a value was voted for unanimously by a quorum

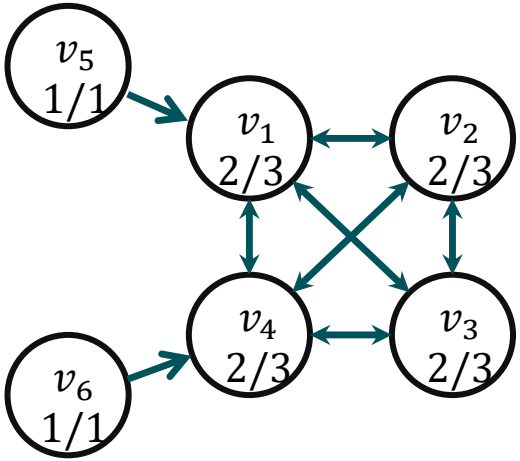
# Does quorum intersection suffice to ensure Agreement?

- We use slices received over the network to compute whether a value was voted for unanimously by a quorum
- Faulty validators can manipulate this by forging or lying about their slices
- “Perceived” quorums depend on Byzantine behavior



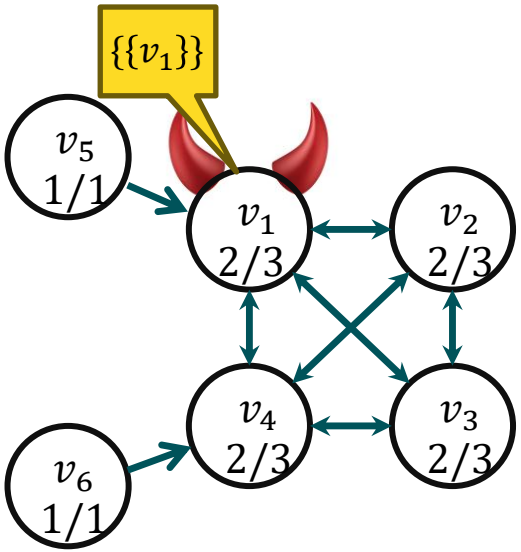
# Example: faulty validators can “cut” quorums

- Initially, every quorum includes 3 validators out of $\{v_1, v_2, v_3, v_4\}$
- So, one Byzantine failure among $\{v_1, v_2, v_3, v_4\}$ should be okay



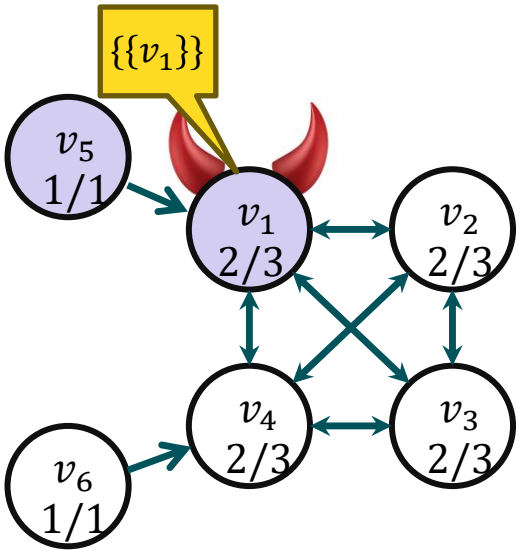
# Example: faulty validators can “cut” quorums

- Initially, every quorum includes 3 validators out of $\{v_1, v_2, v_3, v_4\}$
- So, one Byzantine failure among $\{v_1, v_2, v_3, v_4\}$ should be okay
- But, say $v_1$ is faulty and reports slices $\{\{v_1\}\}$



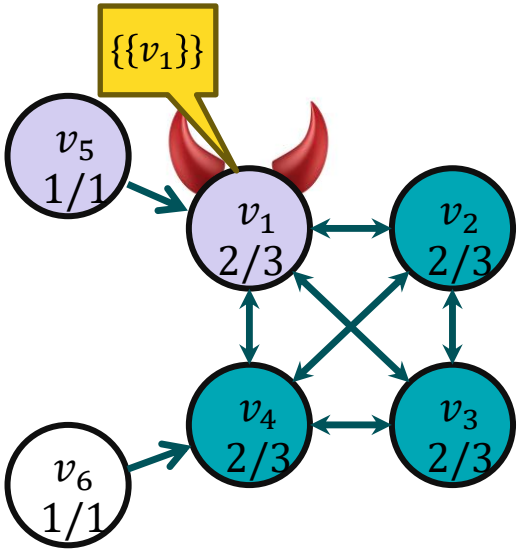
# Example: faulty validators can “cut” quorums

- Initially, every quorum includes 3 validators out of $\{v_1, v_2, v_3, v_4\}$
- So, one Byzantine failure among $\{v_1, v_2, v_3, v_4\}$ should be okay
- But, say $v_1$ is faulty and reports slices $\{\{v_1\}\}$
- Then $v_5$ thinks $\{v_1, v_5\}$ is one of its quorums



# Example: faulty validators can “cut” quorums

- Initially, every quorum includes 3 validators out of $\{v_1, v_2, v_3, v_4\}$
- So, one Byzantine failure among $\{v_1, v_2, v_3, v_4\}$ should be okay
- But, say $v_1$ is faulty and reports slices $\{\{v_1\}\}$
- Then $v_5$ thinks $\{v_1, v_5\}$ is one of its quorums
- $\{v_2, v_3, v_4\}$ is a disjoint quorum
- Quorum intersection actually fails!



# “Possible quorums” account for worst-case Byzantine behavior

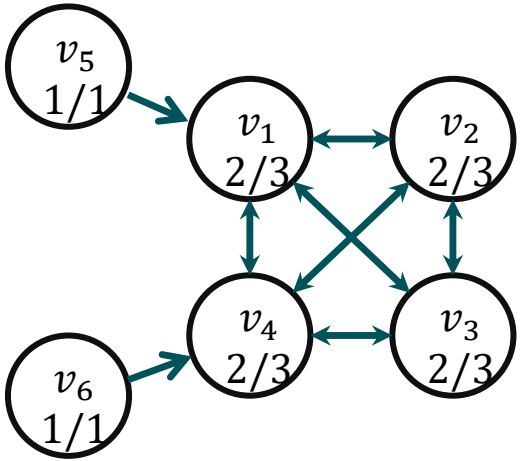
Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

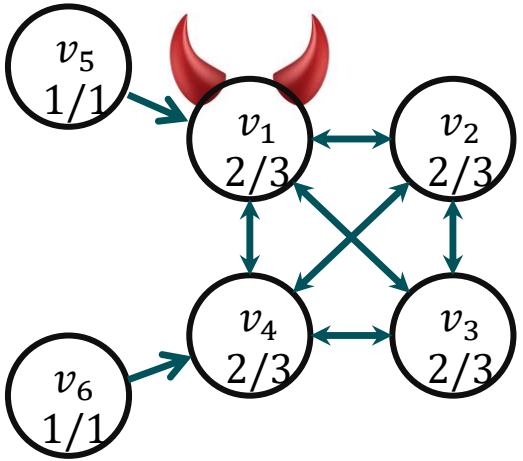


# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_1$ is faulty then



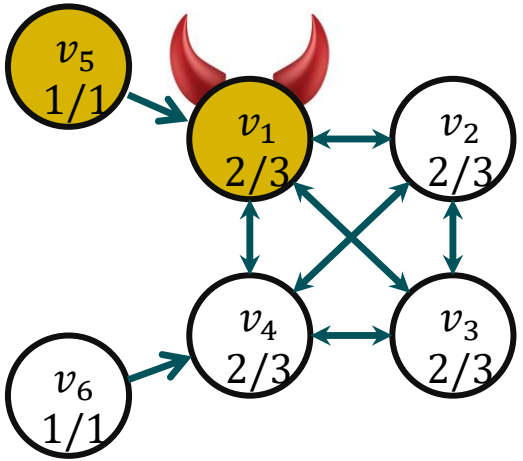
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_1$ is faulty then

- $\{v_5, v_1\}$ is a possible quorum



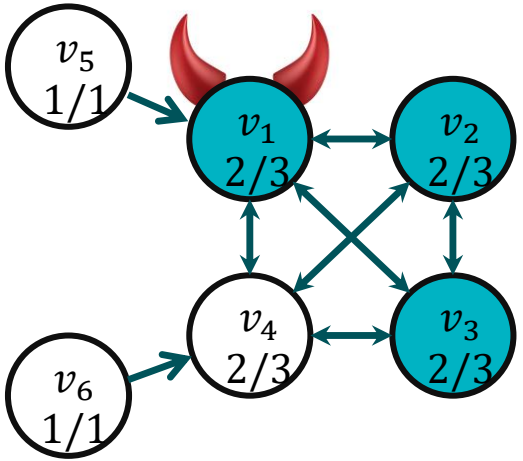
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_1$ is faulty then

- $\{v_5, v_1\}$ is a possible quorum
- Any 3 out of $\{v_1, v_2, v_3, v_4\}$ are a possible quorum



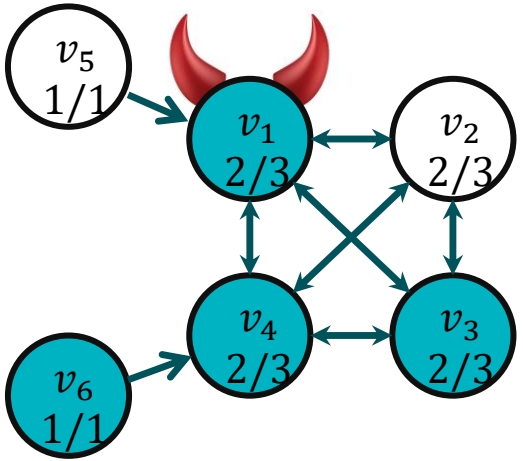
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_1$ is faulty then

- $\{v_5, v_1\}$ is a possible quorum
- Any 3 out of $\{v_1, v_2, v_3, v_4\}$ are a possible quorum
- $\{v_6, v_4\}$ and any two out of $\{v_1, v_2, v_3\}$ are a possible quorum

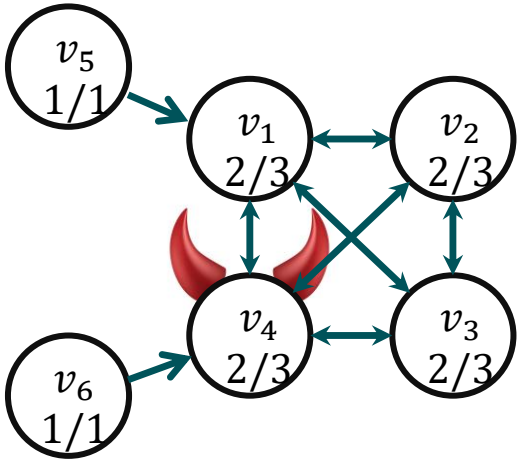


# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_2$ is faulty then



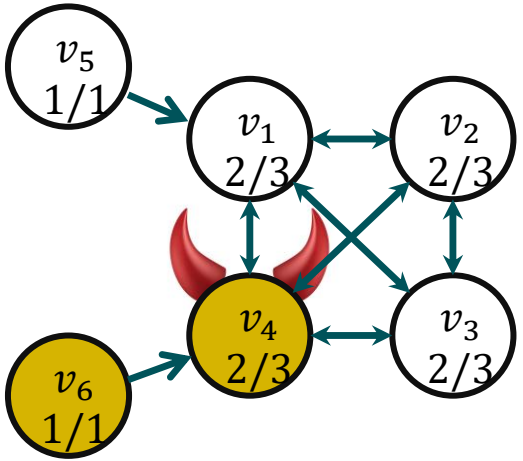
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_2$ is faulty then

- $\{v_6, v_2\}$ is now a possible quorum



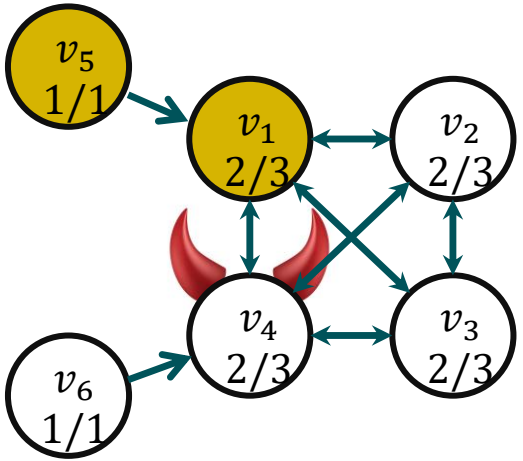
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_2$ is faulty then

- $\{v_6, v_2\}$ is now a possible quorum
- What about $\{v_5, v_6\}$?



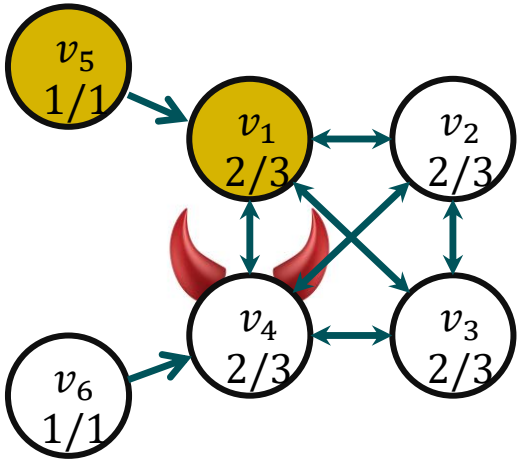
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_2$ is faulty then

- $\{v_6, v_2\}$ is now a possible quorum
- What about $\{v_5, v_6\}$?
  - Not a possible quorum

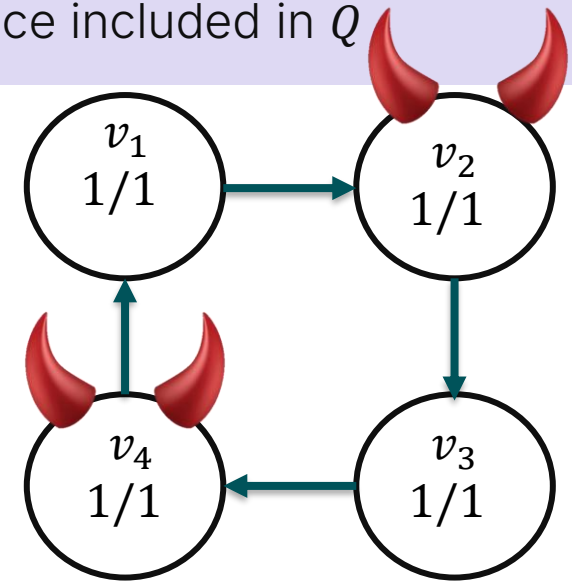


# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_2$ and $v_4$ are faulty then



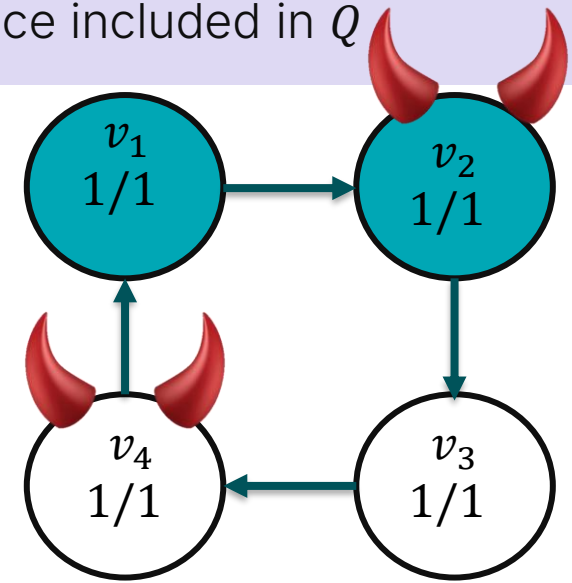
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_2$ and $v_4$ are faulty then

- $\{v_1, v_2\}$ is a possible quorum
  - ✓ $v_1$ has a slice included in $\{v_1, v_2\}$
  - ✓ $v_2$ is faulty



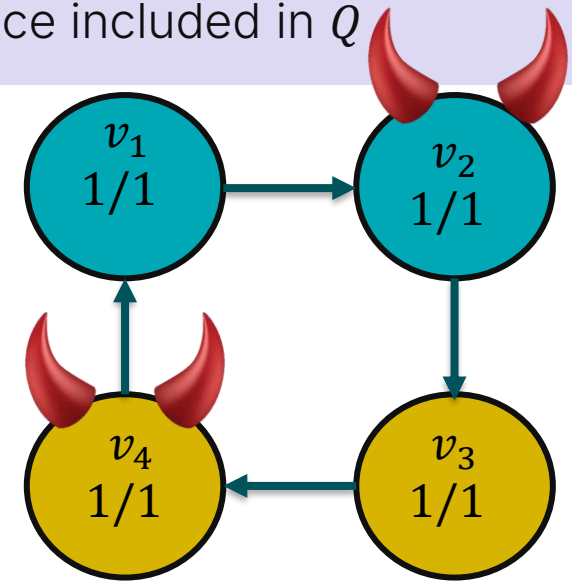
# “Possible quorums” account for worst-case Byzantine behavior

Definition: $Q$ is a possible quorum when

- $Q$ contains a well-behaved member
- Every ***well-behaved*** member of $Q$ has a slice included in $Q$

If $v_2$ and $v_4$ are faulty then

- $\{v_1, v_2\}$ is a possible quorum
  - ✓ $v_1$ has a slice included in $\{v_1, v_2\}$
  - ✓ $v_2$ is faulty
- $\{v_3, v_4\}$ is a possible quorum
  - ✓ $v_3$ has a slice included in $\{v_3, v_4\}$
  - ✓ $v_4$ is faulty



# We can now analyze the system in terms of possible quorums

Assuming that

- Every two ***possible quorums*** have a well-behaved member in common
- Every validator has a well-behaved quorum

The voting algorithm ensures Unanimity, Agreement, and Validity

*What about Totality?*

# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ✓ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- ✓ Quorums in Federated Byzantine Agreement systems
- ➢ Reliable Voting in FBA systems
  - ✓ A simple, safe straw-man protocol
  - ✓ Quorum intersection is not that simple (possible quorums)
  - ➢ Consensus clusters
  - Obtaining Totality: the Federated Voting protocol
- Total-Order Broadcast by porting Simplex to FBA

# In an open system, we cannot assume possible quorums will suitably intersect

Validators are free to choose their agreement requirements

- Some might choose to agree with the “wrong” validators
- Too many validators get hacked
- Some validators might genuinely not want to agree with others and form their own community

Does the system devolve into chaos?

# Consensus Clusters

# Consensus Clusters

Fix a faulty set $\mathcal{F}$

# Consensus Clusters

Fix a faulty set $\mathcal{F}$

Definition: **Intertwined**
A set of well-behaved validators is intertwined when every two of their possible quorums have a well-behaved member in common

# Consensus Clusters

Fix a faulty set $\mathcal{F}$

Definition: **Intertwined**
A set of well-behaved validators is intertwined when every two of their possible quorums have a well-behaved member in common

Definition: **Consensus cluster** (**cluster** for short)
A set of validators $\mathbb{C}$ is a cluster when all its members are well-behaved and

- $\mathbb{C}$ is intertwined
- $\mathbb{C}$ is a quorum

# Consensus Clusters

Fix a faulty set $\mathcal{F}$

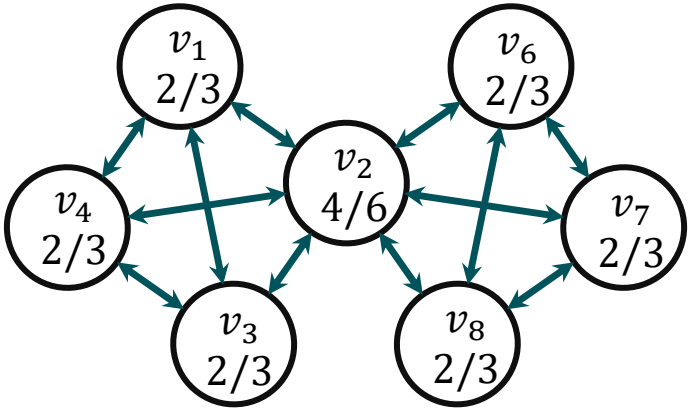
Definition: **Intertwined**
A set of well-behaved validators is intertwined when every two of their possible quorums have a well-behaved member in common

Definition: **Consensus cluster** (**cluster** for short)
A set of validators $\mathbf{C}$ is a cluster when all its members are well-behaved and

- $\mathbf{C}$ is intertwined
- $\mathbf{C}$ is a quorum

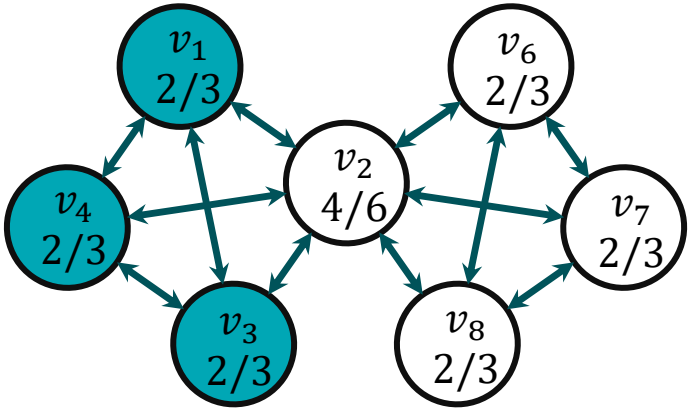
Notation: we will use capital $\mathbf{C}$ for clusters ($\mathbf{C}$, $\mathbf{C}$', $C_1$)

# Example



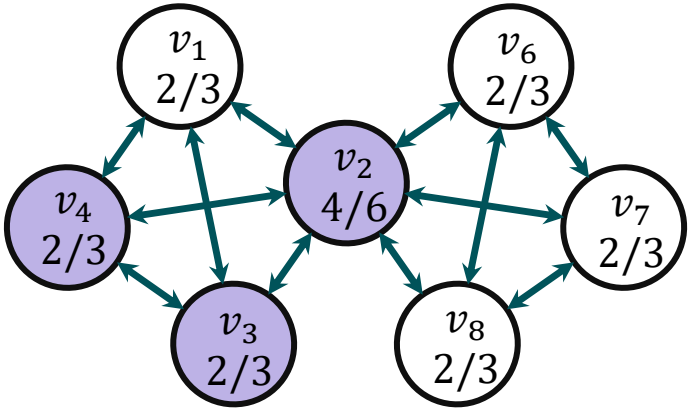
# Example

- $\{v_1, v_3, v_4\}$ is a cluster



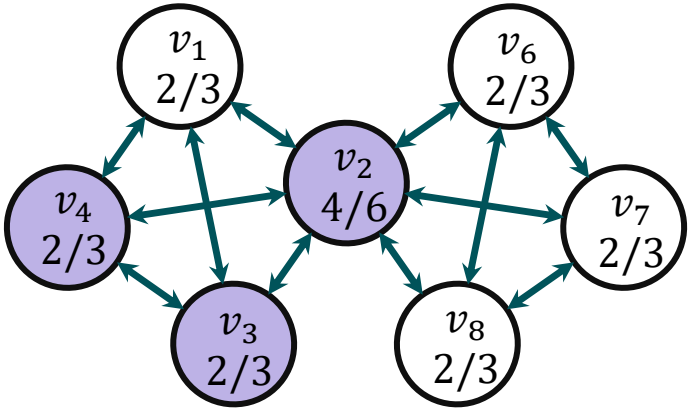
# Example

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_2, v_3, v_4\}$ is not a cluster



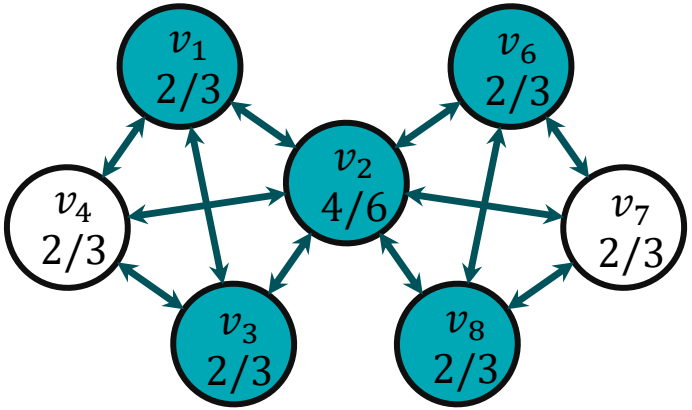
# Example

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_2, v_3, v_4\}$ is not a cluster
  - It is not a quorum



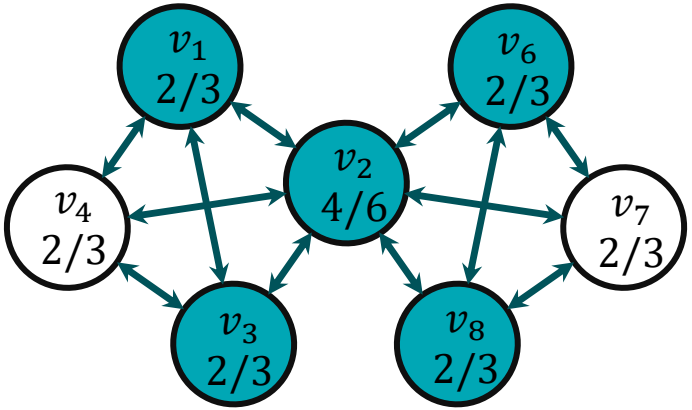
# Example

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_2, v_3, v_4\}$ is not a cluster
  - It is not a quorum
- $\{v_1, v_2, v_3, v_6, v_8\}$ is not a cluster



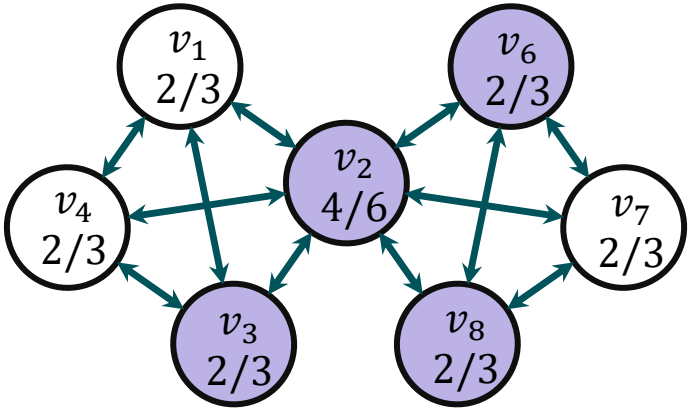
# Example

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_2, v_3, v_4\}$ is not a cluster
  - It is not a quorum
- $\{v_1, v_2, v_3, v_6, v_8\}$ is not a cluster
  - It is not intertwined



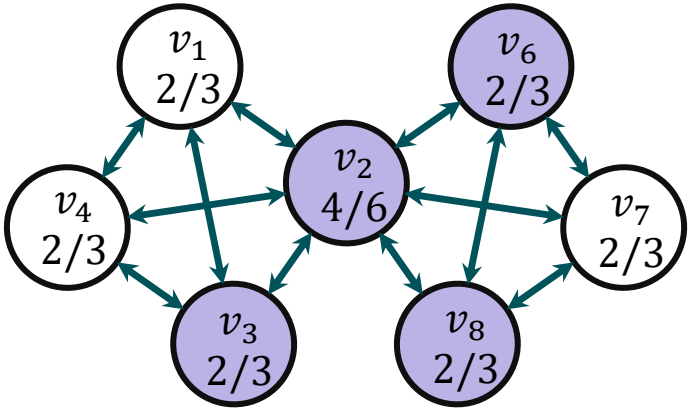
# Example

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_2, v_3, v_4\}$ is not a cluster
  - It is not a quorum
- $\{v_1, v_2, v_3, v_6, v_8\}$ is not a cluster
  - It is not intertwined
- $\{v_2, v_3, v_6, v_8\}$ is not a cluster



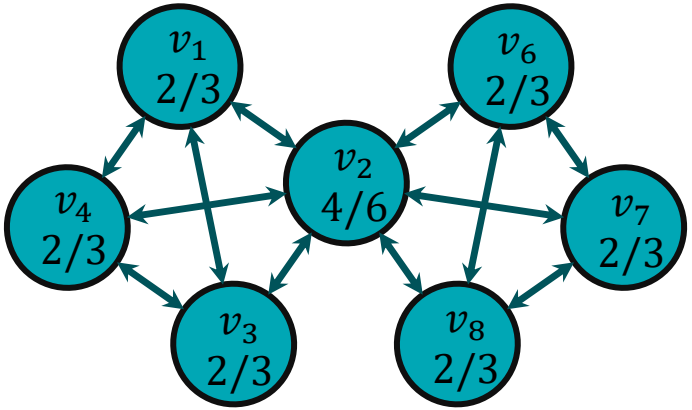
# Example

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_2, v_3, v_4\}$ is not a cluster
  - It is not a quorum
- $\{v_1, v_2, v_3, v_6, v_8\}$ is not a cluster
  - It is not intertwined
- $\{v_2, v_3, v_6, v_8\}$ is not a cluster
  - It is not a quorum

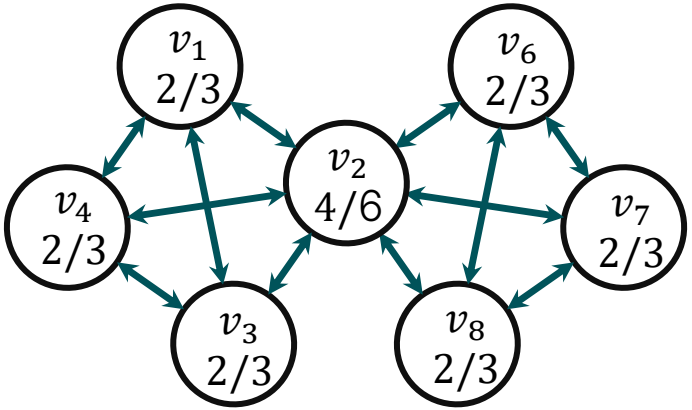


# Example

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_2, v_3, v_4\}$ is not a cluster
  - It is not a quorum
- $\{v_1, v_2, v_3, v_6, v_8\}$ is not a cluster
  - It is not intertwined
- $\{v_2, v_3, v_6, v_8\}$ is not a cluster
  - It is not a quorum
- The whole system is a cluster

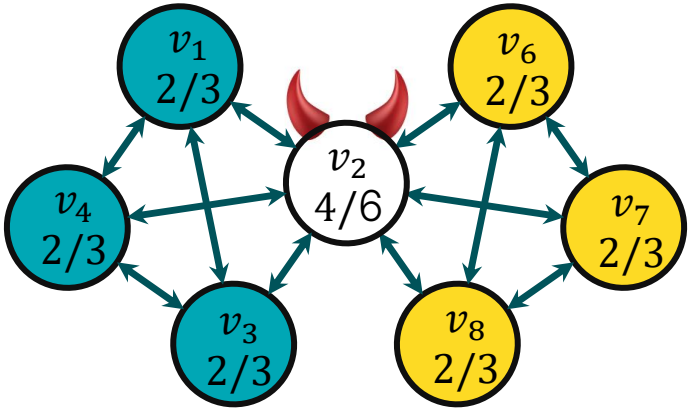


# Example



# Example

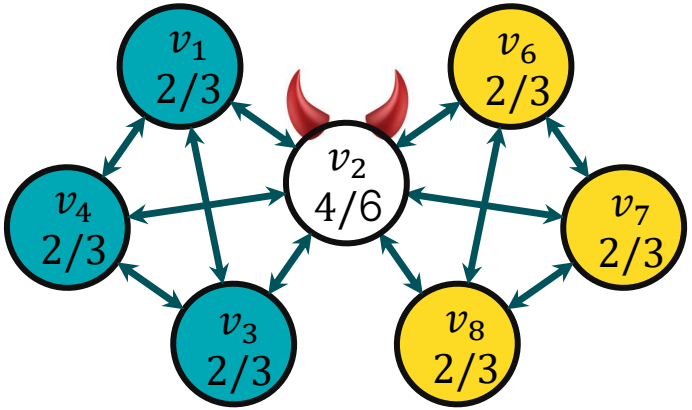
Suppose $\mathcal{F} = \{v_2\}$



# Example

Suppose $\mathcal{F} = \{v_2\}$

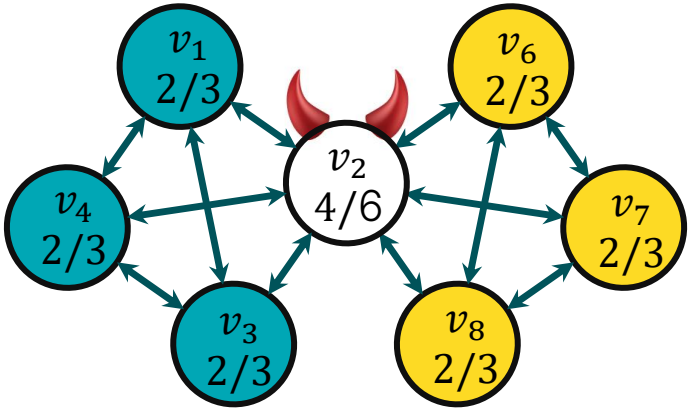
- $\{v_1, v_3, v_4\}$ is a cluster



# Example

Suppose $\mathcal{F} = \{v_2\}$

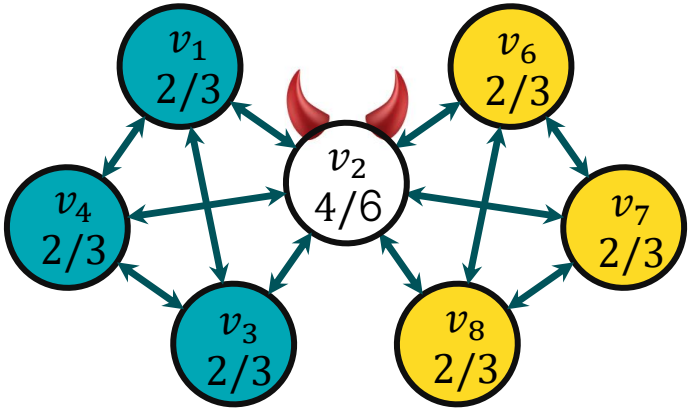
- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_6, v_7, v_8\}$ is a cluster



# Example

Suppose $\mathcal{F} = \{v_2\}$

- $\{v_1, v_3, v_4\}$ is a cluster
- $\{v_6, v_7, v_8\}$ is a cluster
- Those are the only two clusters



# The system partitions itself into disjoint maximal clusters
(and a set of confused validators)

# The system partitions itself into disjoint maximal clusters
## (and a set of confused validators)

Notation: Write $V_1 \not\between V_2$ when $V_1$ and $V_2$ intersect

# The system partitions itself into disjoint maximal clusters
## (and a set of confused validators)

Notation: Write $V_1 \between V_2$ when $V_1$ and $V_2$ intersect

*Lemma*: If $C \between C'$ then $C \cup C'$ is intertwined

Proof: It suffices to show that $\forall Q, Q'.\ Q \between C$ and $C' \between Q'$ implies $Q \between Q'$

# The system partitions itself into disjoint maximal clusters
## (and a set of confused validators)

Notation: Write $V_1 \between V_2$ when $V_1$ and $V_2$ intersect

*Lemma*: If $C \between C'$ then $C \cup C'$ is intertwined

Proof: It suffices to show that $\forall Q, Q'.\ Q \between C$ and $C' \between Q'$ implies $Q \between Q'$

$Q \between C$ $\quad$ $C \between C'$

# The system partitions itself into disjoint maximal clusters
## (and a set of confused validators)

Notation: Write $V_1 \not\emptyset V_2$ when $V_1$ and $V_2$ intersect

*Lemma*: If $C \not\emptyset C'$ then $C \cup C'$ is intertwined

Proof: It suffices to show that $\forall Q, Q'.\ Q \not\emptyset C$ and $C' \not\emptyset Q'$ implies $Q \not\emptyset Q'$

$C$ is intertwined
$Q$ and $C'$ are quorums

$$\frac{Q \not\emptyset C \qquad C \not\emptyset C'}{Q \not\emptyset C'}$$

# The system partitions itself into disjoint maximal clusters
## (and a set of confused validators)

Notation: Write $V_1 \wr V_2$ when $V_1$ and $V_2$ intersect

*Lemma*: If $C \wr C'$ then $C \cup C'$ is intertwined

Proof: It suffices to show that $\forall Q, Q'.\ Q \wr C$ and $C' \wr Q'$ implies $Q \wr Q'$

$C$ is intertwined
$Q$ and $C'$ are quorums

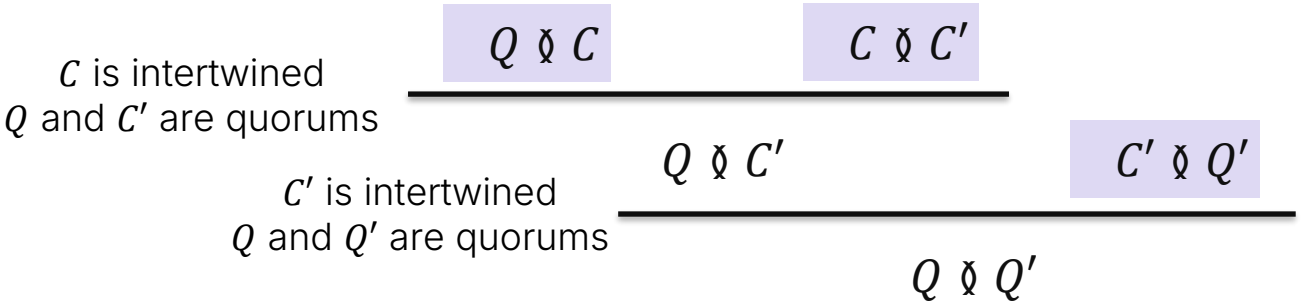
$$\frac{Q \wr C \qquad C \wr C'}{Q \wr C'} \qquad C' \wr Q'$$

# The system partitions itself into disjoint maximal clusters
(and a set of confused validators)

Notation: Write $V_1 \between V_2$ when $V_1$ and $V_2$ intersect

*Lemma*: If $C \between C'$ then $C \cup C'$ is intertwined

Proof: It suffices to show that $\forall Q, Q'.\ Q \between C$ and $C' \between Q'$ implies $Q \between Q'$



# The system partitions itself into disjoint maximal clusters
(and a set of confused validators)

# The system partitions itself into disjoint maximal clusters
(and a set of confused validators)

*Corollary*

If $C \between C'$ then $C \cup C'$ is a cluster

# The system partitions itself into disjoint maximal clusters
(and a set of confused validators)

*Corollary*

If $C \between C'$ then $C \cup C'$ is a cluster

*Proof*

- $C \cup C'$ is a quorum because each is a quorum
- $C \cup C'$ is intertwined by the previous lemma

# The system partitions itself into disjoint maximal clusters
(and a set of confused validators)

*Corollary*
If $C \between C'$ then $C \cup C'$ is a cluster

*Proof*
- $C \cup C'$ is a quorum because each is a quorum
- $C \cup C'$ is intertwined by the previous lemma

*Theorem*
Maximal consensus clusters are disjoint

# The system partitions itself into disjoint maximal clusters
(and a set of confused validators)

*Corollary*
If $C \between C'$ then $C \cup C'$ is a cluster

*Proof*
- $C \cup C'$ is a quorum because each is a quorum
- $C \cup C'$ is intertwined by the previous lemma

*Theorem*
Maximal consensus clusters are disjoint

*Proof*
Immediate from the corollary

# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ✓ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- ✓ Quorums in Federated Byzantine Agreement systems
- ➢ Reliable Voting in FBA systems
  - ✓ A simple straw-man protocol
  - ✓ Quorum intersection is not that simple (possible quorums)
  - ✓ Consensus clusters
  - ➢ Obtaining Totality: the Federated Voting protocol
- Total-Order Broadcast by porting Simplex to FBA

# New goal: Reliable Voting *for clusters*

New Specification:

For every value $x$ and every cluster $C$:

***Unanimity:*** If all members of $C$ vote for $x$, then they all eventually confirm $x$

***Validity:*** If a member of $C$ confirms $x$, then a member of $C$ voted for $x$

***Agreement:*** No two members of $C$ confirm different values

***Totality:*** If a member of $C$ confirms $x$, then all members of $C$ eventually confirm $x$

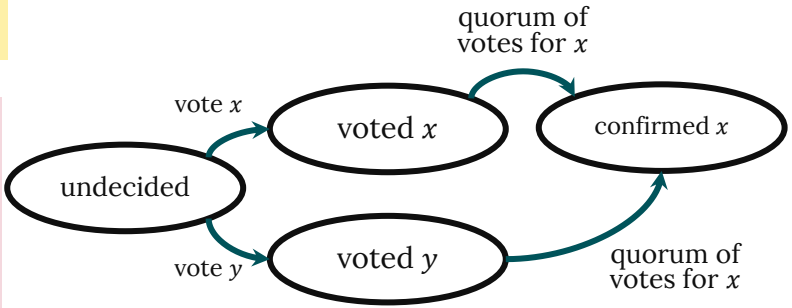
***Note***: We do not know in advance which set will be clusters

# Does the voting algorithm implement Reliable Voting in the FBA model?

Validity, Agreement, and Unanimity hold

Problem: Totality

Because quorums are subjective, a quorum that convinces $v_1$ may not convince $v_2$



# To obtain Totality, we introduce a new “accept” phase

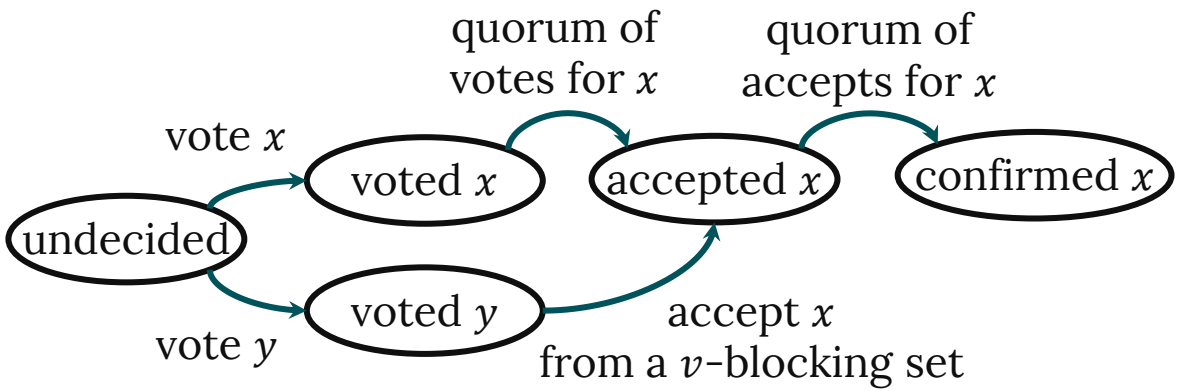
# To obtain Totality, we introduce a new “accept” phase

Definition: a set of validators $V$ is a $v$-*blocking* set when every slice of $v$ intersects $V$ (we also say $v$ is blocked by $V$)

# To obtain Totality, we introduce a new “accept” phase

Definition: a set of validators $V$ is a $v$-*blocking* set when every slice of $v$ intersects $V$ (we also say $v$ is blocked by $V$)

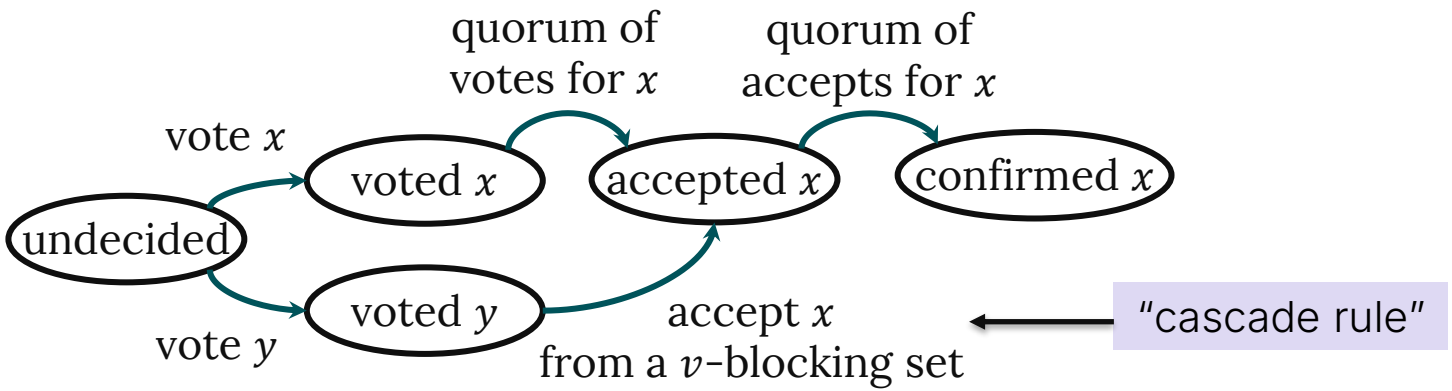
**The Federated Voting algorithm:**



# To obtain Totality, we introduce a new “accept” phase

Definition: a set of validators $V$ is a $v$-*blocking* set when every slice of $v$ intersects $V$ (we also say $v$ is blocked by $V$)

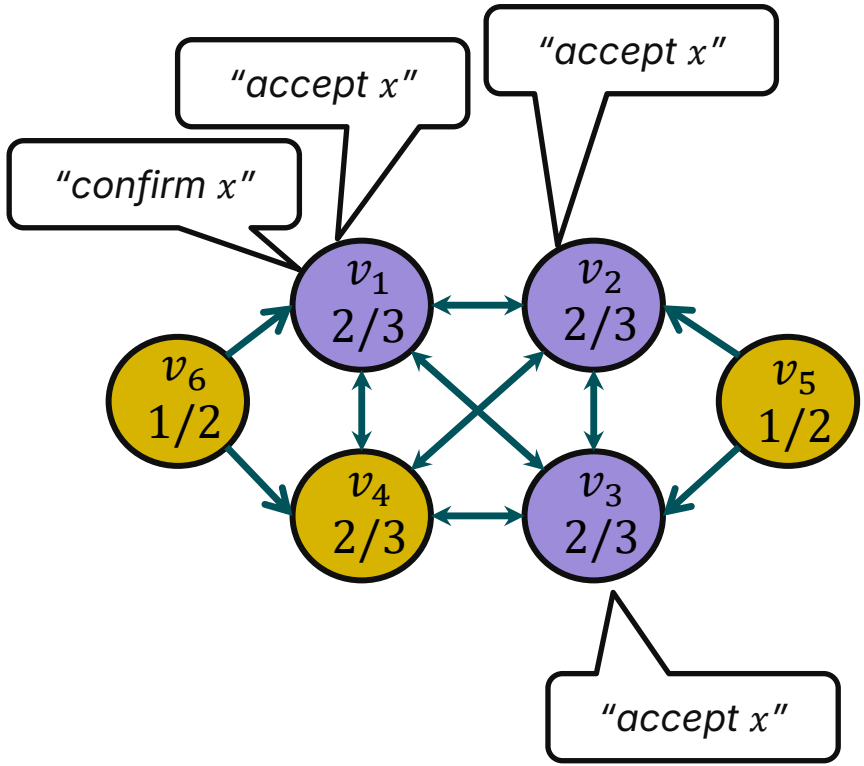
**The Federated Voting algorithm:**



# If a cluster member confirms, then "accept" and "confirm" cascade to the entire cluster

Say $v_2$ confirmed $x$ because $v_1, v_2, v_3$ accepted $x$, but $v_5, v_6, v_4$ voted for $y$

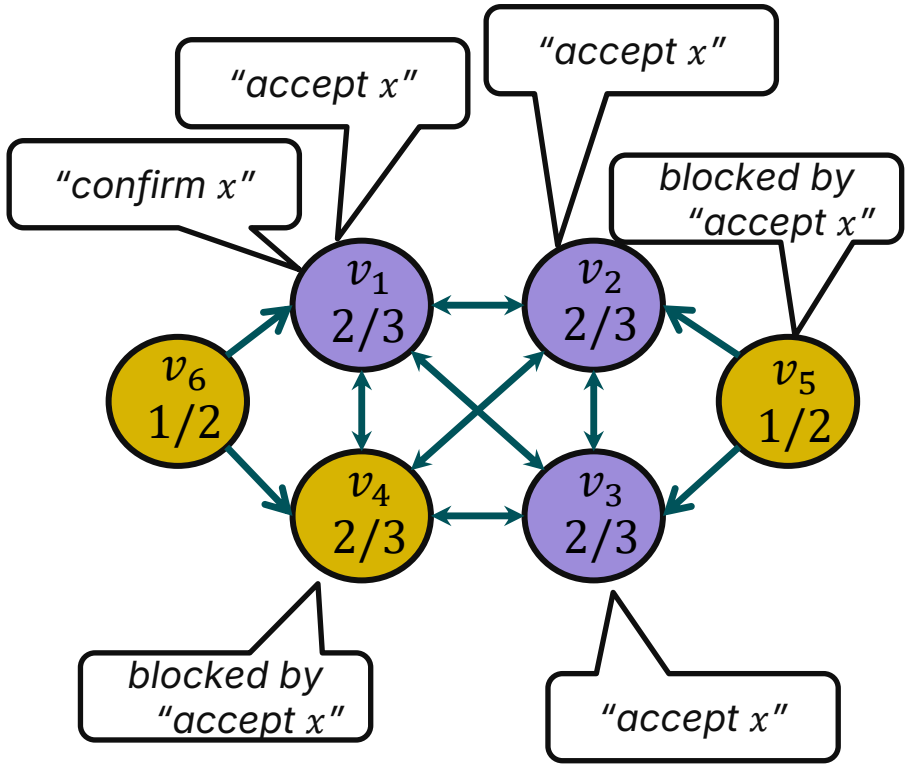
1. $v_4$ and $v_5$ are blocked by accept $x$, but not $v_6$
2. $v_4$ and $v_5$ accept $x$
3. $v_6$ is blocked by accept $x$
4. $v_6$ accepts $x$



# If a cluster member confirms, then "accept" and "confirm" cascade to the entire cluster

Say $v_2$ confirmed $x$ because $v_1, v_2, v_3$ accepted $x$, but $v_5, v_6, v_4$ voted for $y$

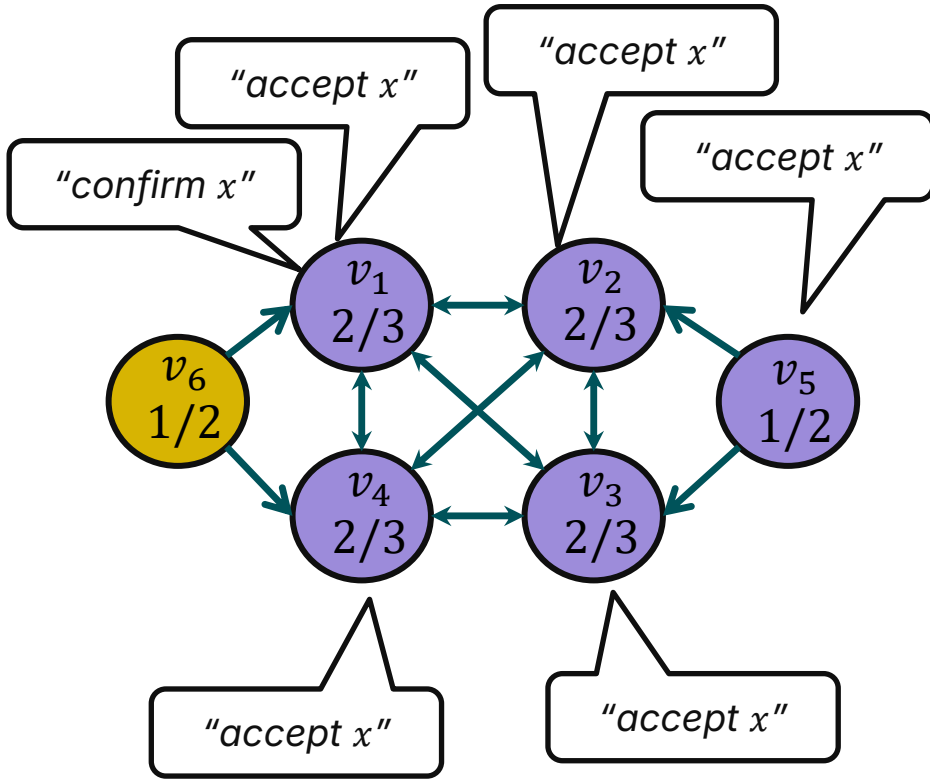
1. $v_4$ and $v_5$ are blocked by accept $x$, but not $v_6$
2. $v_4$ and $v_5$ accept $x$
3. $v_6$ is blocked by accept $x$
4. $v_6$ accepts $x$



# If a cluster member confirms, then "accept" and "confirm" cascade to the entire cluster

Say $v_2$ confirmed $x$ because $v_1, v_2, v_3$ accepted $x$, but $v_5, v_6, v_4$ voted for $y$

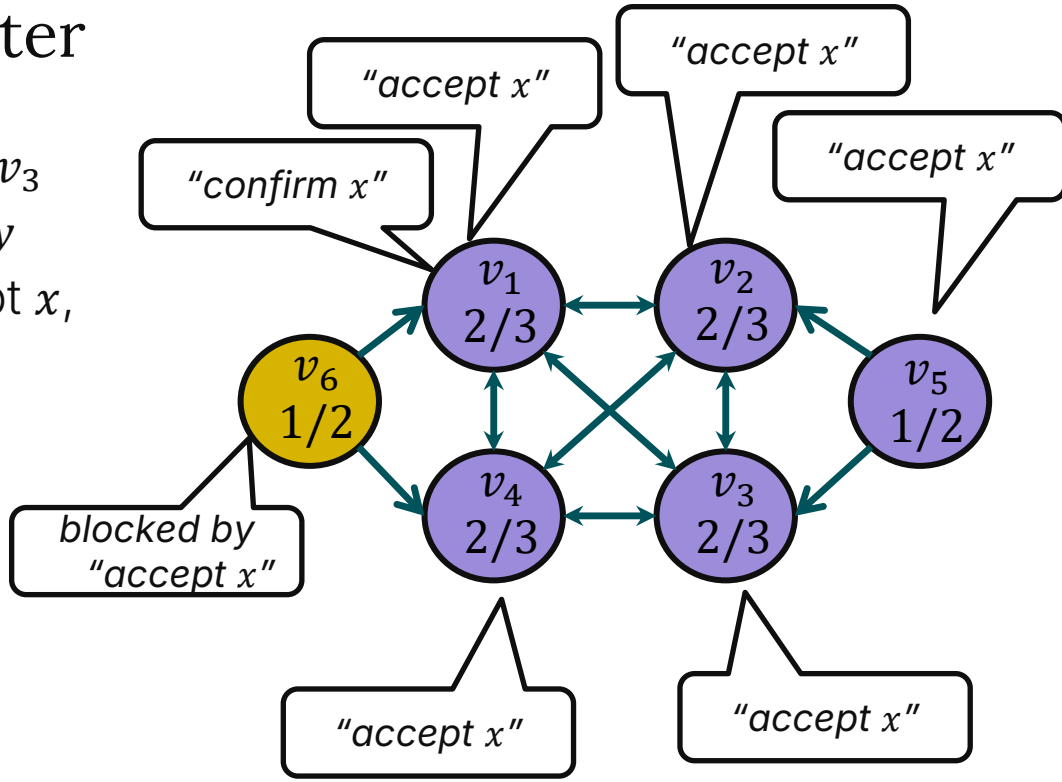
1. $v_4$ and $v_5$ are blocked by accept $x$, but not $v_6$
2. $v_4$ and $v_5$ accept $x$
3. $v_6$ is blocked by accept $x$
4. $v_6$ accepts $x$



# If a cluster member confirms, then “accept” and “confirm” cascade to the entire cluster

Say $v_2$ confirmed $x$ because $v_1, v_2, v_3$ accepted $x$, but $v_5, v_6, v_4$ voted for $y$

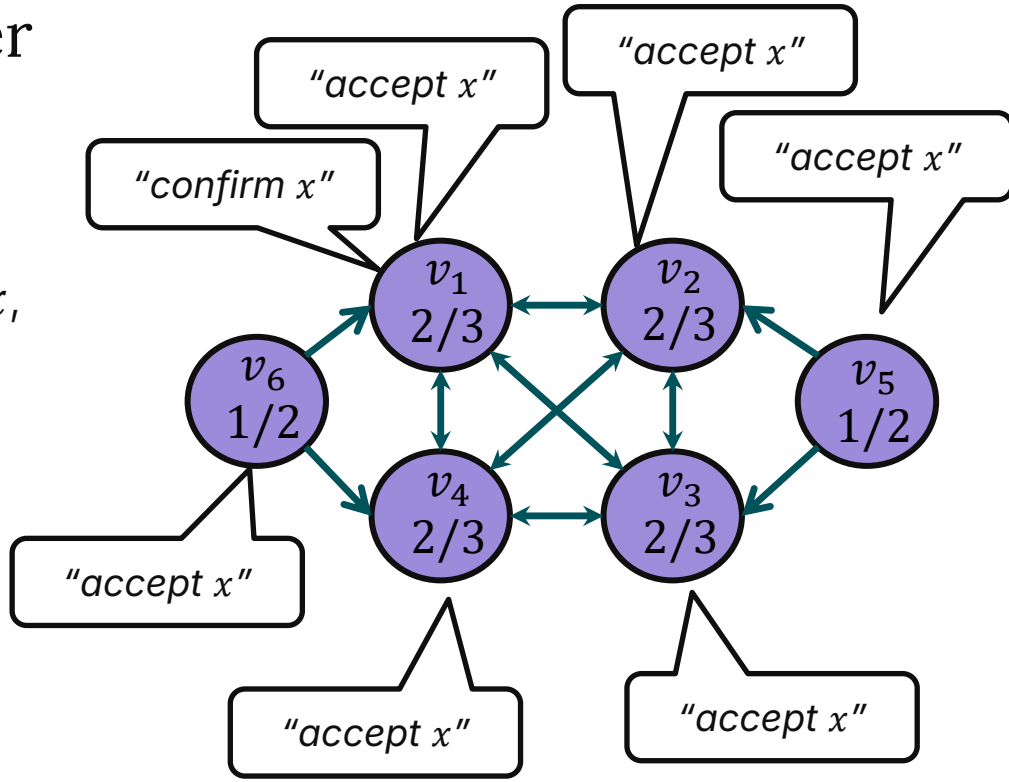
1. $v_4$ and $v_5$ are blocked by accept $x$, but not $v_6$
2. $v_4$ and $v_5$ accept $x$
3. $v_6$ is blocked by accept $x$
4. $v_6$ accepts $x$



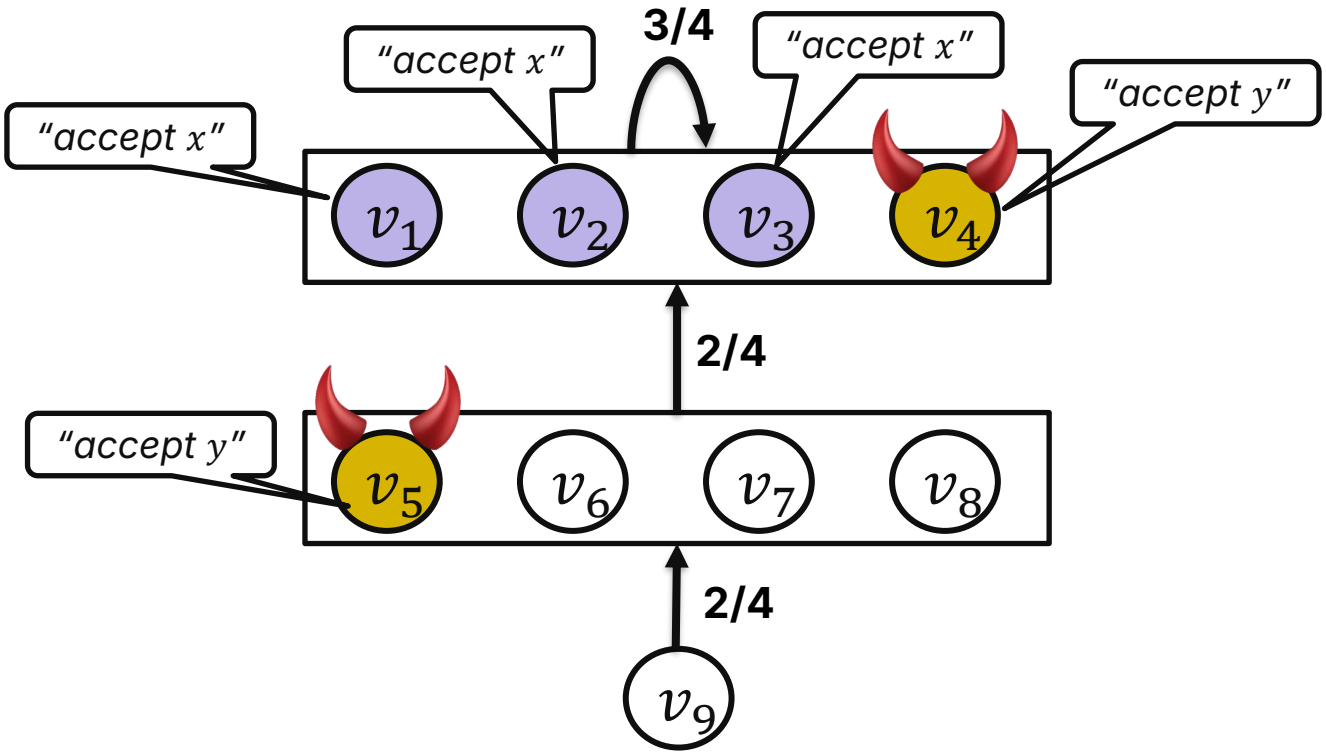
# If a cluster member confirms, then "accept" and "confirm" cascade to the entire cluster

Say $v_2$ confirmed $x$ because $v_1, v_2, v_3$ accepted $x$, but $v_5, v_6, v_4$ voted for $y$

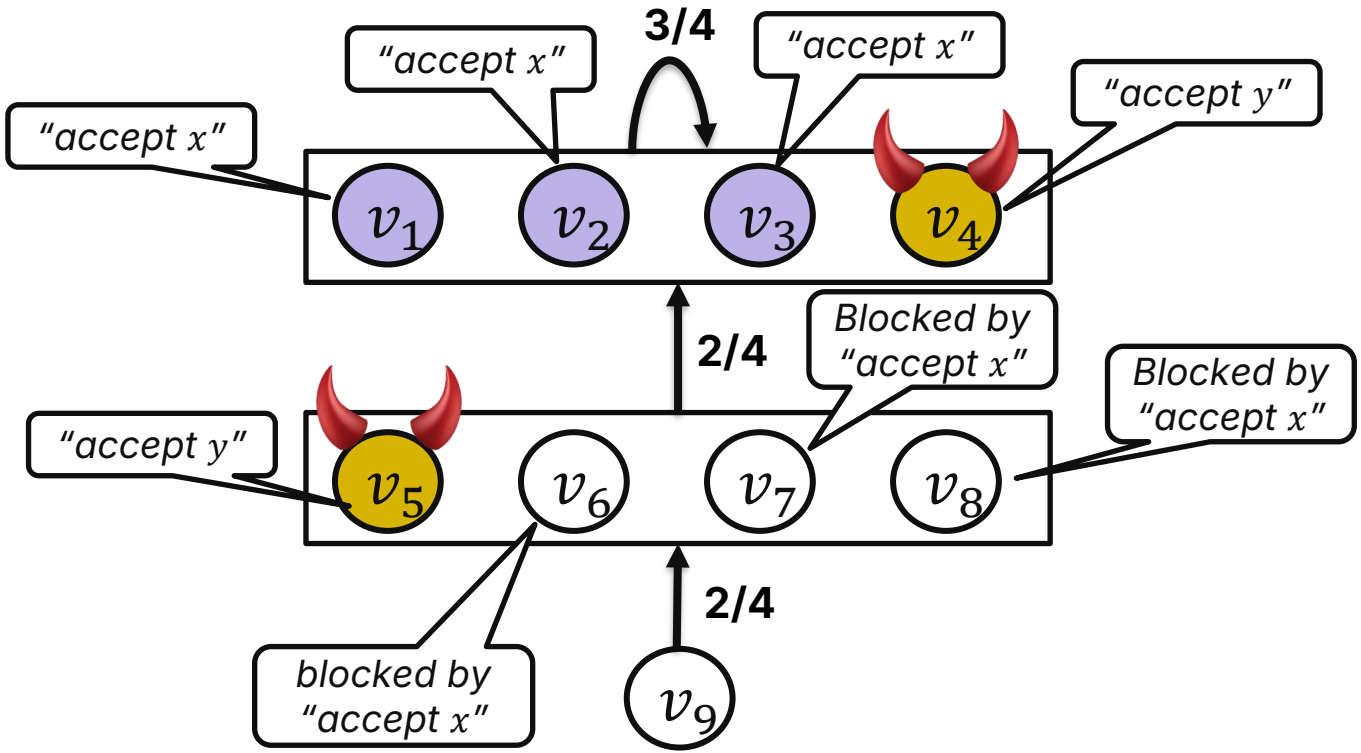
1. $v_4$ and $v_5$ are blocked by accept $x$, but not $v_6$
2. $v_4$ and $v_5$ accept $x$
3. $v_6$ is blocked by accept $x$
4. $v_6$ accepts $x$



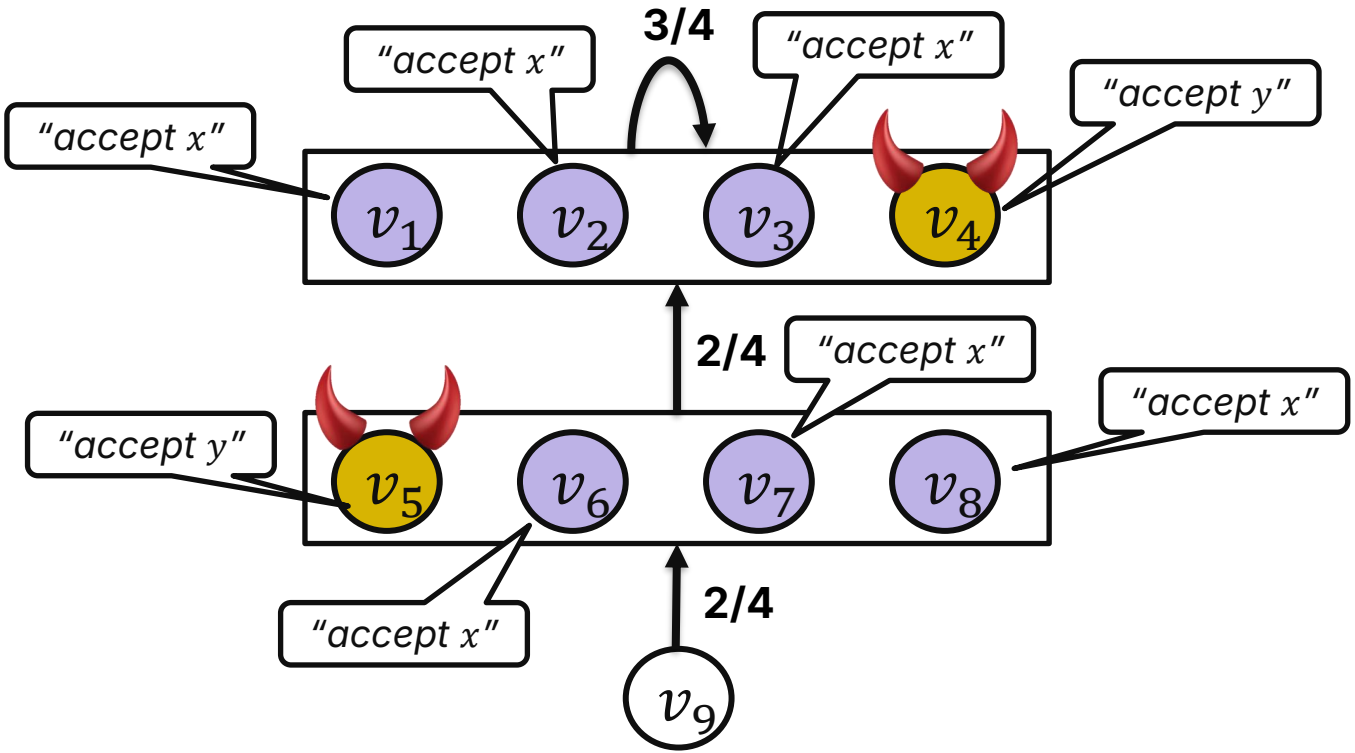
# If a cluster member confirms, then “accept” and “confirm” cascade to the entire cluster



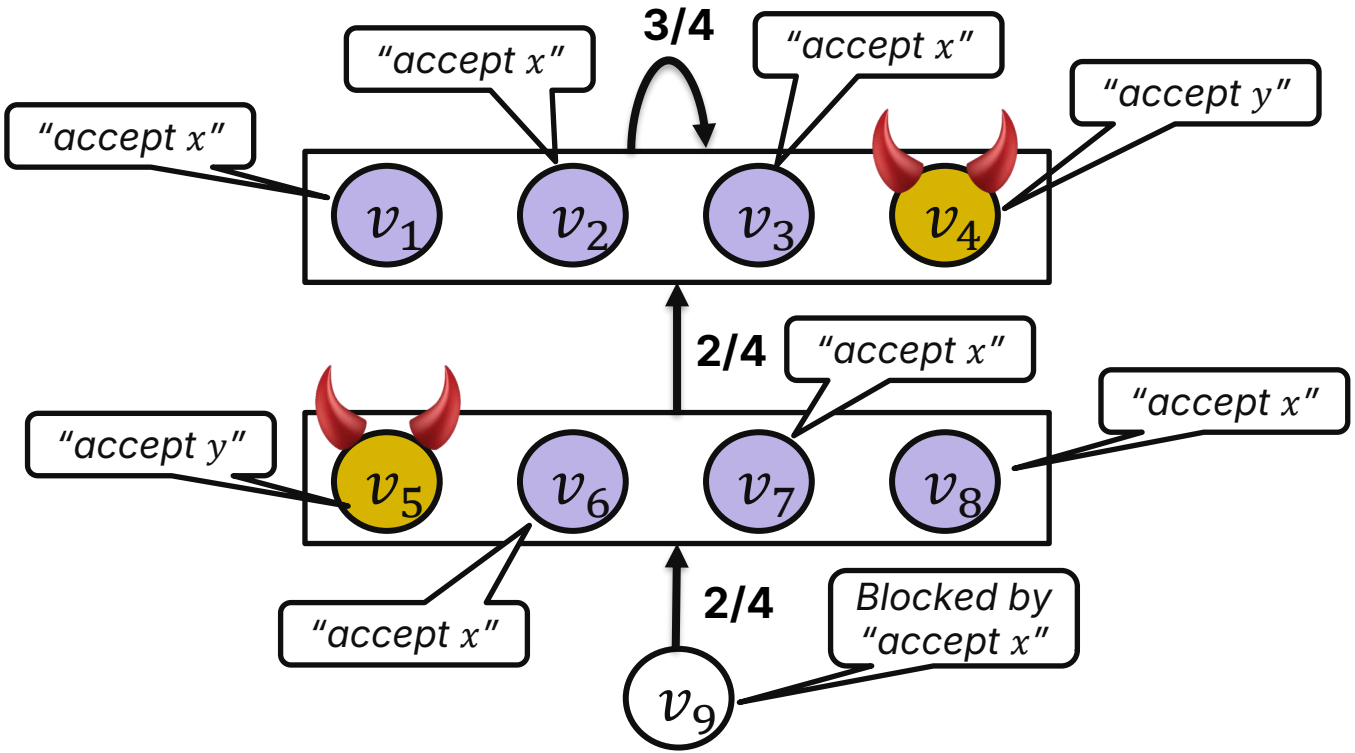
# If a cluster member confirms, then “accept” and “confirm” cascade to the entire cluster



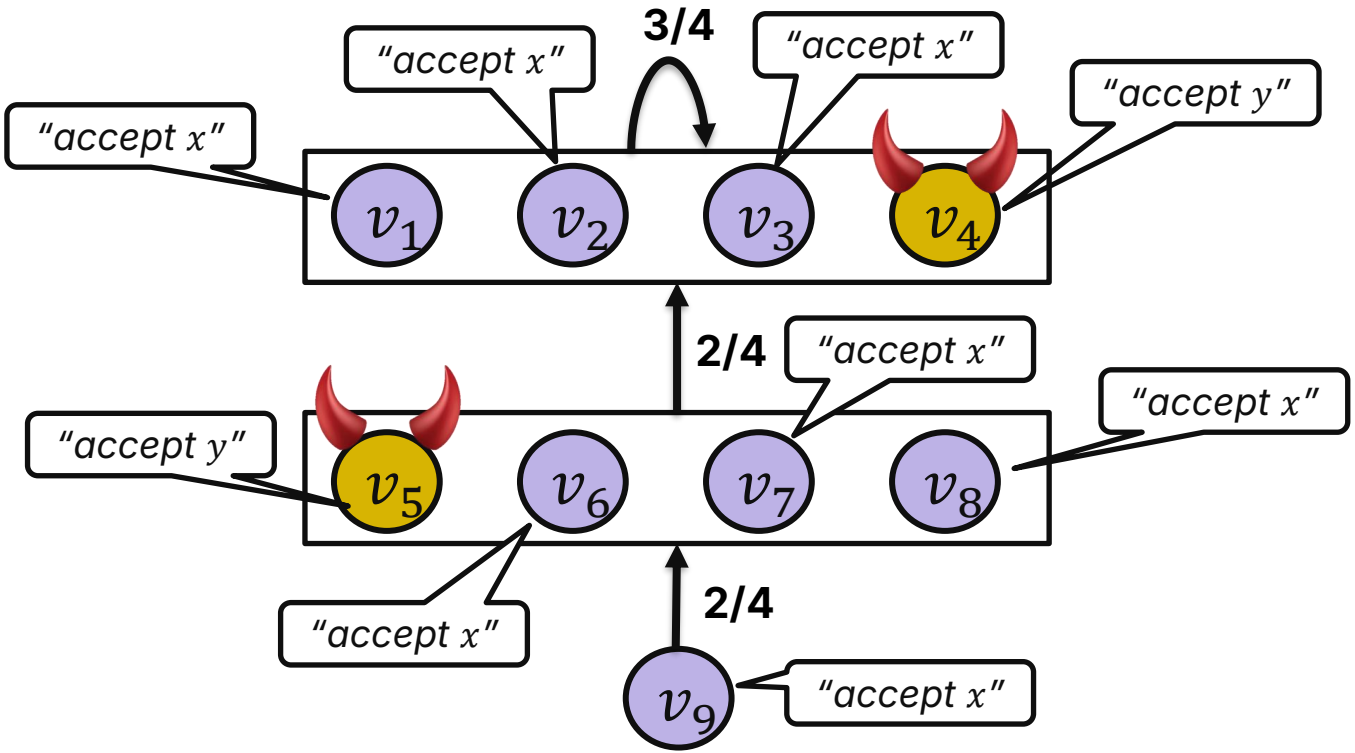
# If a cluster member confirms, then “accept” and “confirm” cascade to the entire cluster



# If a cluster member confirms, then “accept” and “confirm” cascade to the entire cluster



# If a cluster member confirms, then “accept” and “confirm” cascade to the entire cluster



# The cascade rule preserves Agreement and ensures Totality

It suffices to show:

1. **For Agreement**: If a validator $v \in C$ accepts $x$, then a member of $C$ accepted $x$ through the quorum rule
2. **For Totality**: If a quorum $Q$ of a validator $v \in C$ accepts $x$, then "accept $x$" cascades to all members of $C$

# Notion of “possibly cascades”

Definition: a set $U$ of validators **possibly cascades** to $v$:

- If $v \in U$, then $U$ possibly cascades to $v$
- If every slice of $v$ contains a validator $v'$ such that $U$ possibly cascades to $v'$ or $v'$ is faulty, then $U$ possibly cascades to $v$

Intuition: formalizes “accept $x$” cascading assuming faulty nodes help

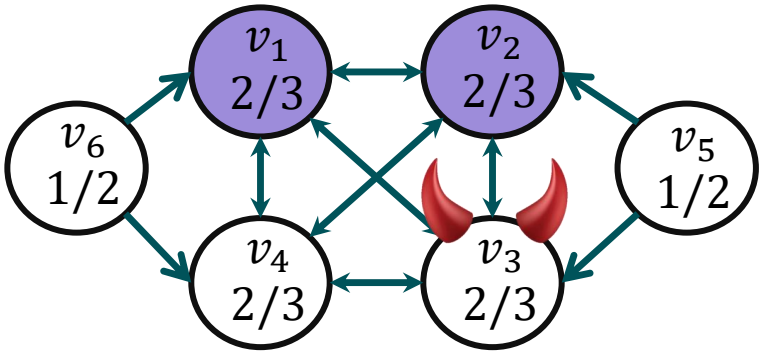
# Notion of “possibly cascades”

Definition: a set $U$ of validators **possibly cascades** to $v$:

- If $v \in U$, then $U$ possibly cascades to $v$
- If every slice of $v$ contains a validator $v'$ such that $U$ possibly cascades to $v'$ or $v'$ is faulty, then $U$ possibly cascades to $v$

Intuition: formalizes “accept $x$” cascading assuming faulty nodes help

$U = \{v_1, v_2\}$ possibly cascades to $v_6$ and $v_5$ because: if $U \cup \boldsymbol{\mathcal{F}} = \{\boldsymbol{v_1, v_2, v_3}\}$ accepts $x$, the “accept $x$” cascades to the whole system



# Notion of “surely cascades”

Definition: a set $U$ of well-behaved validators **surely cascades** to $v$:

- If $v \in U$, then $U$ surely cascades to $v$
- If $v$ is well-behaved and every slice of $v$ contains a well-behaved validator $v'$ such that $U$ surely cascades to $v'$, then $U$ cascades to $v$

Intuition: formalizes “accept $x$” cascading assuming faulty nodes do ***not*** help

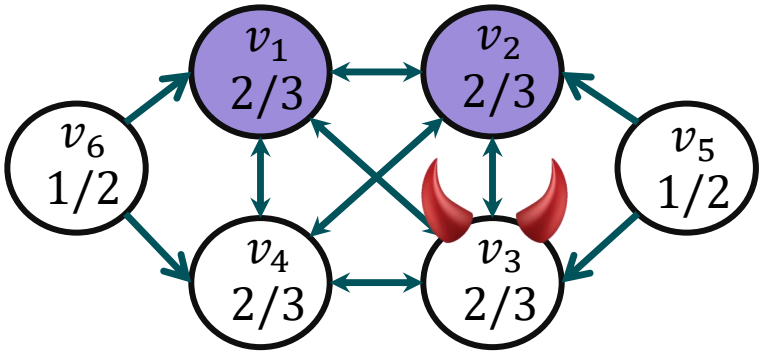
# Notion of “surely cascades”

Definition: a set $U$ of well-behaved validators **surely cascades** to $v$:

- If $v \in U$, then $U$ surely cascades to $v$
- If $v$ is well-behaved and every slice of $v$ contains a well-behaved validator $v'$ such that $U$ surely cascades to $v'$, then $U$ cascades to $v$

Intuition: formalizes “accept $x$” cascading assuming faulty nodes do ***not*** help

$U = \{v_1, v_2\}$ does not surely cascade to $v_6$ and $v_5$: if $\boldsymbol{\mathcal{F} = \{v_3\}}$ remains silent, no cascade occurs



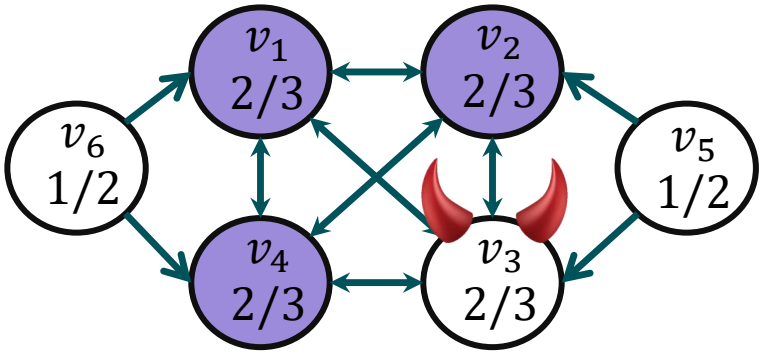
# Notion of “surely cascades”

Definition: a set $U$ of well-behaved validators **surely cascades** to $v$:

- If $v \in U$, then $U$ surely cascades to $v$
- If $v$ is well-behaved and every slice of $v$ contains a well-behaved validator $v'$ such that $U$ surely cascades to $v'$, then $U$ cascades to $v$

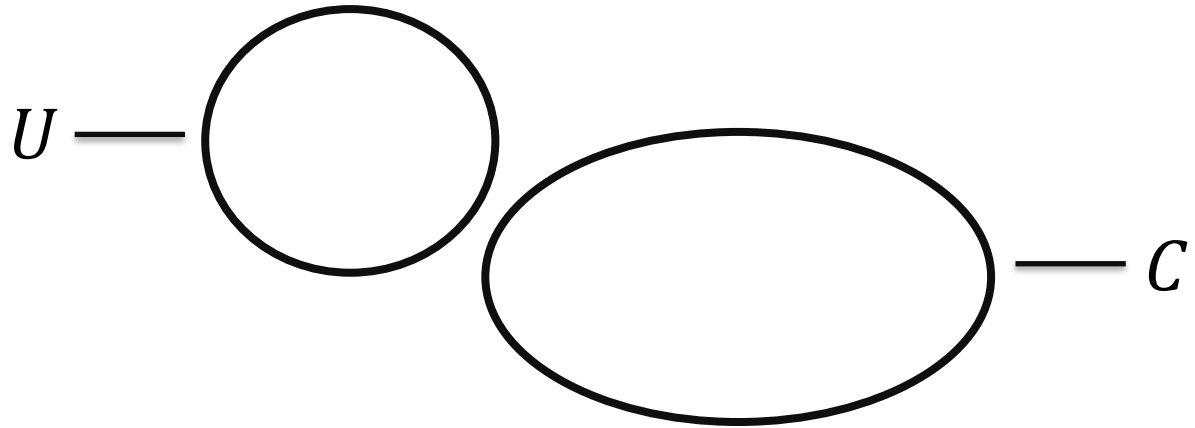
Intuition: formalizes “accept $x$” cascading assuming faulty nodes do ***not*** help

$U = \{v_1, v_2, v_4\}$ surely cascade to $v_6$ but not $v_5$

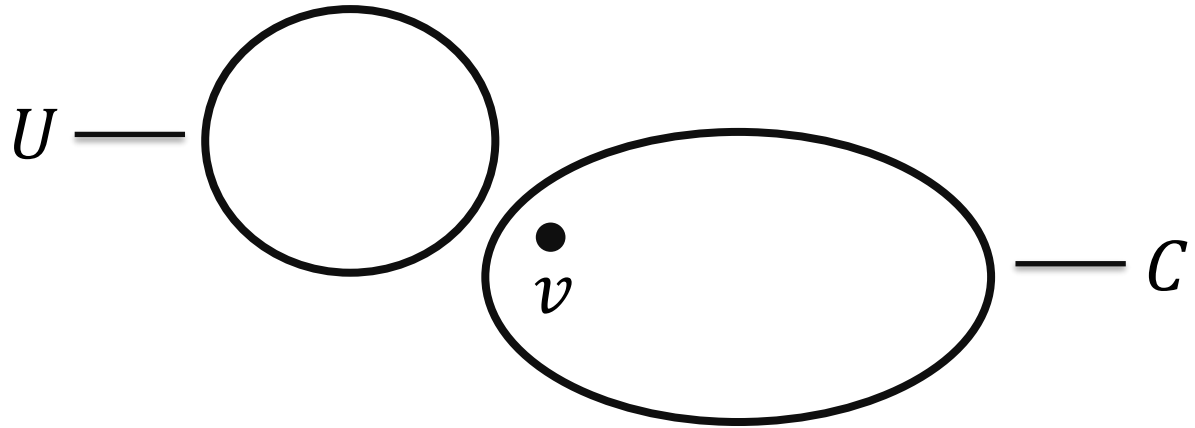


# The cascade rule preserves agreement

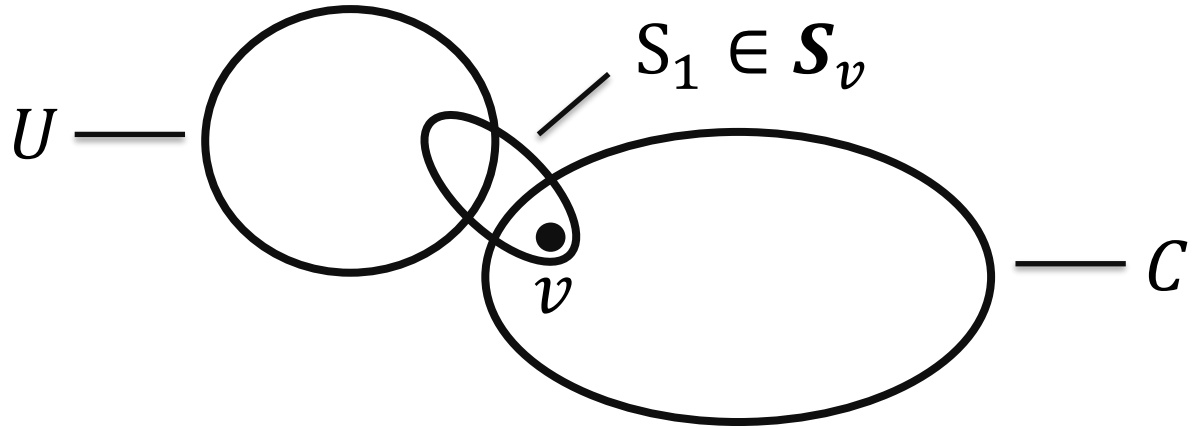
# The cascade rule preserves agreement



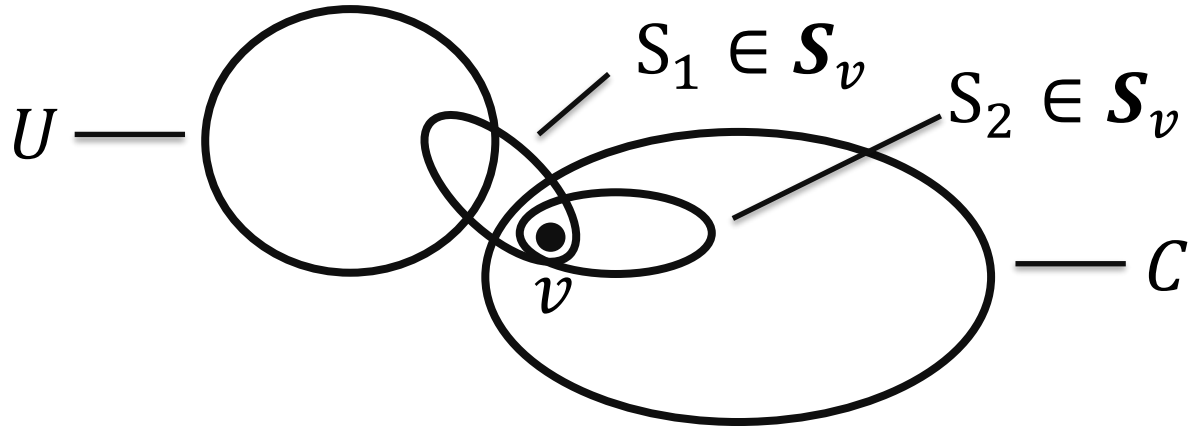
# The cascade rule preserves agreement



# The cascade rule preserves agreement

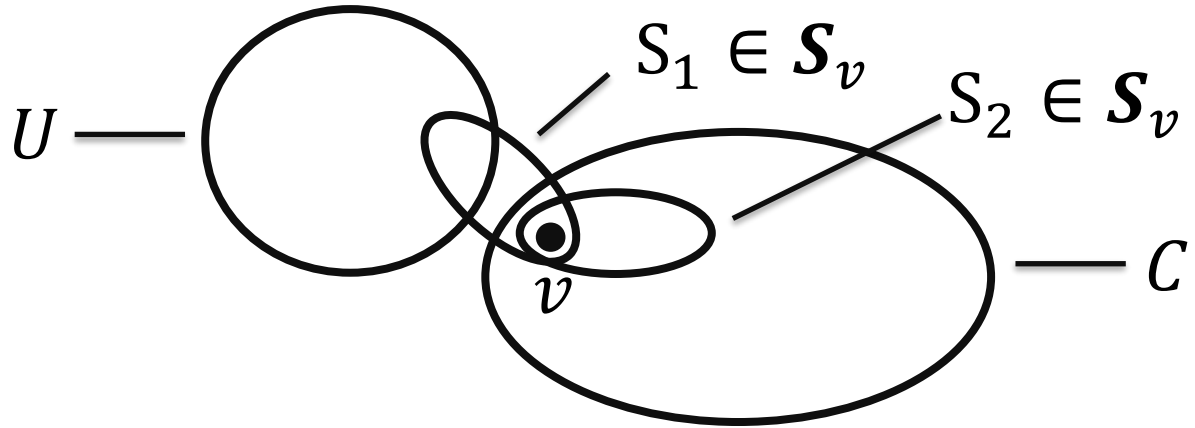


# The cascade rule preserves agreement



# The cascade rule preserves agreement

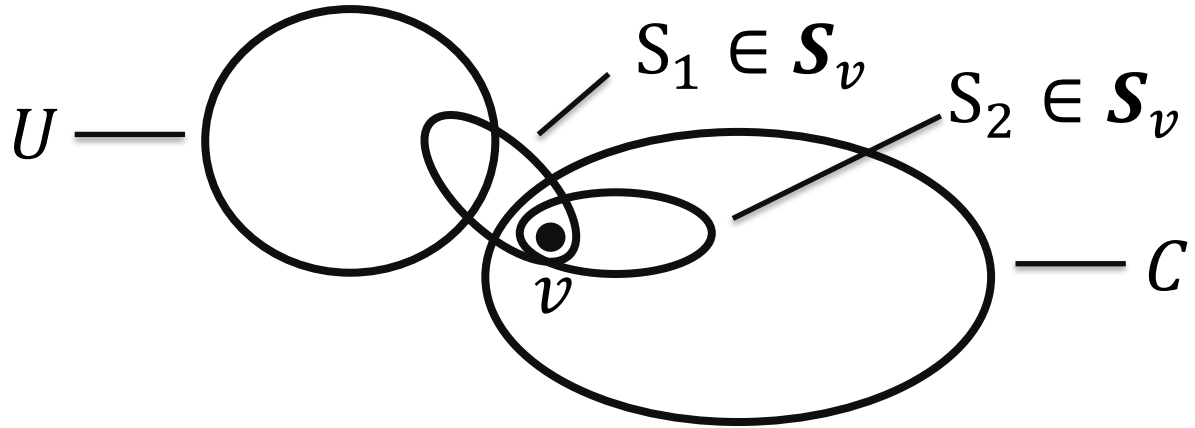
Lemma: if $v \in C$ and $U$ possibly cascades to $v$, then $C \not\wr U$



# The cascade rule preserves agreement

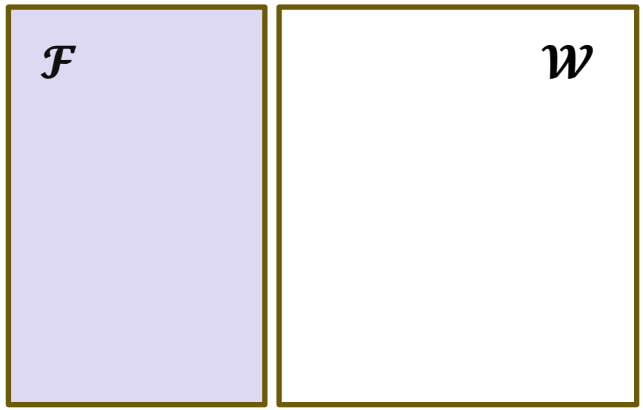
Lemma: if $v \in C$ and $U$ possibly cascades to $v$, then $C \between U$

Proof: $C$ is a well-behaved quorum, so every member of $C$ has a well-behaved slice inside $C$. Thus, the cascade cannot start from outside $C$.



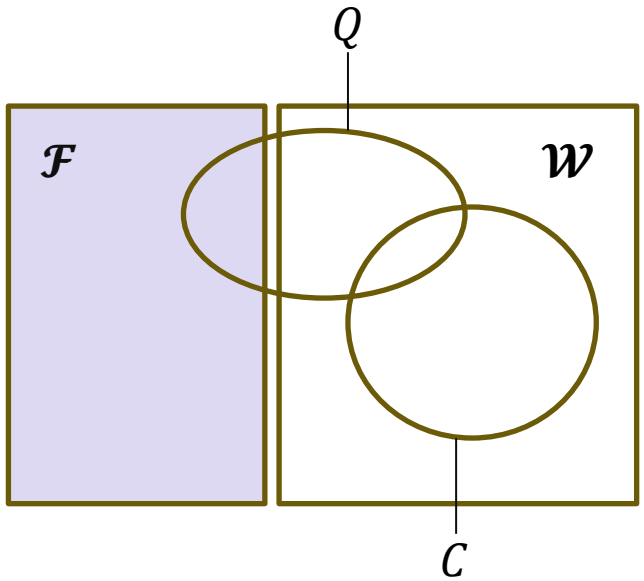
# Totality: The Cascade Theorem

# Totality: The Cascade Theorem



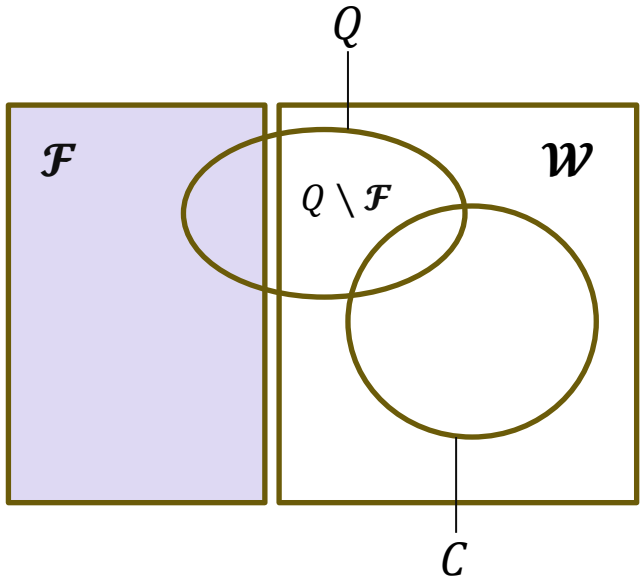
# Totality: The Cascade Theorem

Consider $Q \not\downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)



# Totality: The Cascade Theorem

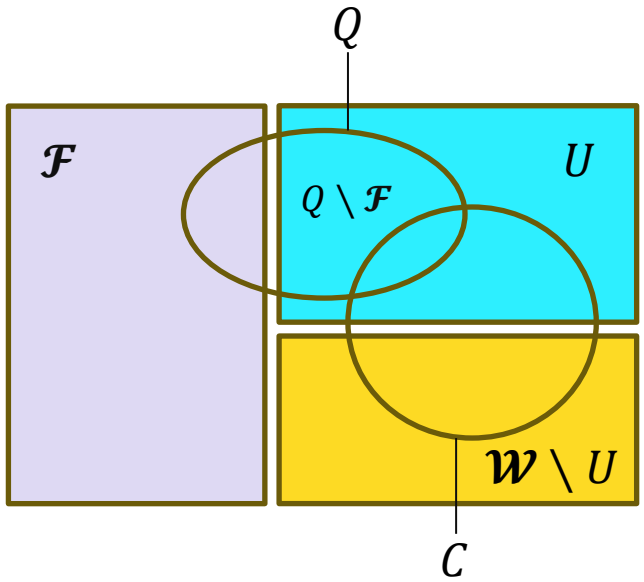
Consider $Q \not\downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)
Then $Q \setminus \mathcal{F}$ surely cascades to every member of $C$



# Totality: The Cascade Theorem

Consider $Q \not\downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)
Then $Q \setminus \mathcal{F}$ surely cascades to every member of $C$

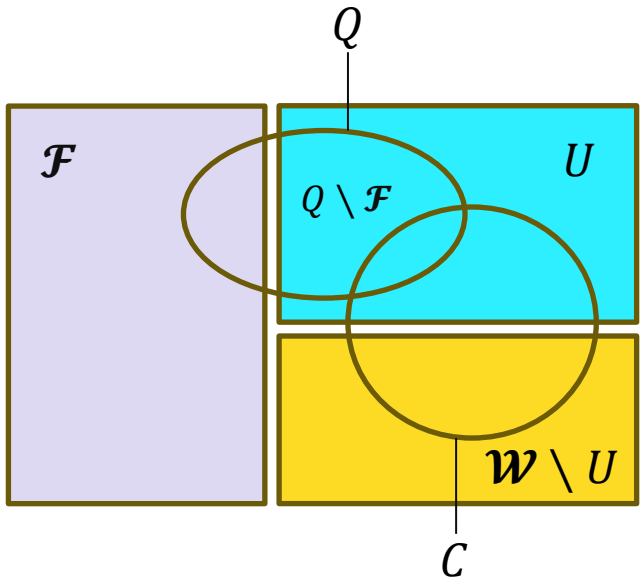
1. Assume toward a contradiction that $Q \setminus \mathcal{F}$ surely cascades only to $U$ such that $C \not\subseteq U$



# Totality: The Cascade Theorem

Consider $Q \not\downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)
Then $Q \setminus \mathcal{F}$ surely cascades to every member of $C$

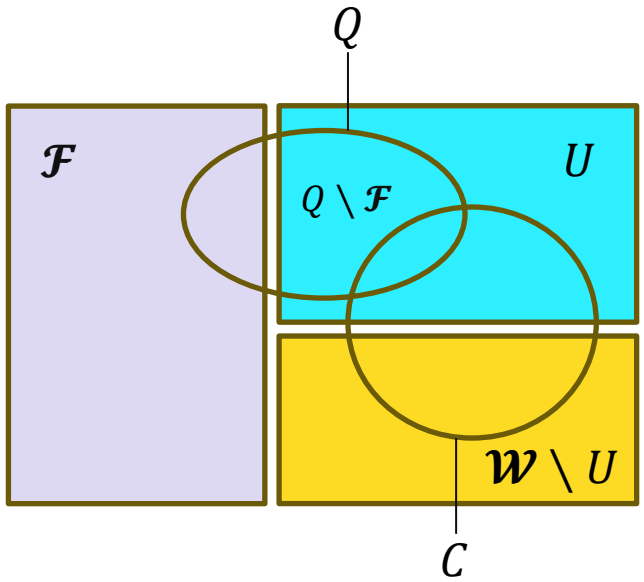
1. Assume toward a contradiction that $Q \setminus \mathcal{F}$ surely cascades only to $U$ such that $C \not\subseteq U$
2. So, every member of $\mathcal{W} \setminus U$ have a slice disjoint from $U$



# Totality: The Cascade Theorem

Consider $Q \not\downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)
Then $Q \setminus \mathcal{F}$ surely cascades to every member of $C$

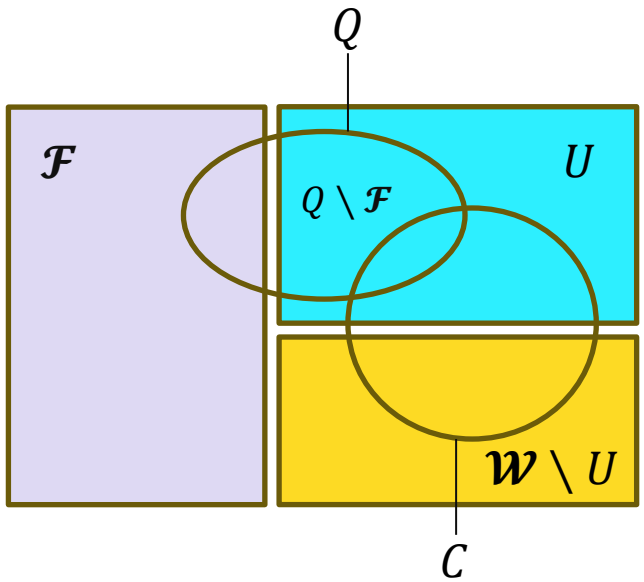
1. Assume toward a contradiction that $Q \setminus \mathcal{F}$ surely cascades only to $U$ such that $C \not\subseteq U$
2. So, every member of $\mathcal{W} \setminus U$ have a slice disjoint from $U$
3. So, $\mathcal{F} \cup \mathcal{W} \setminus U$ is a possible quorum



# Totality: The Cascade Theorem

Consider $Q \not\downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)
Then $Q \setminus \mathcal{F}$ surely cascades to every member of $C$

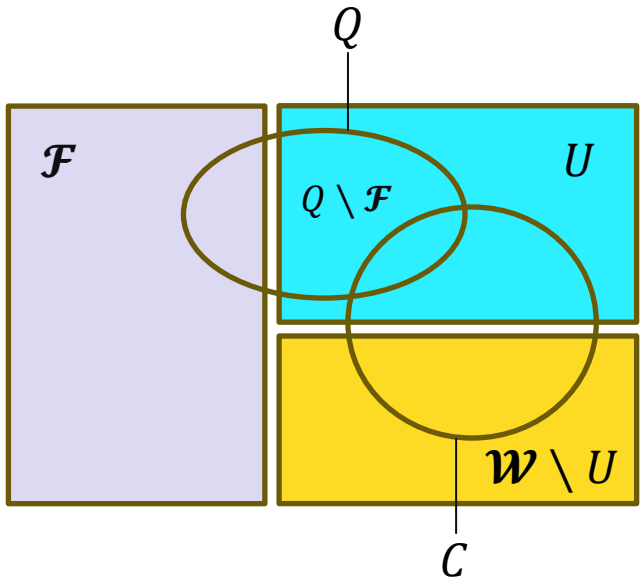
1. Assume toward a contradiction that $Q \setminus \mathcal{F}$ surely cascades only to $U$ such that $C \not\subseteq U$
2. So, every member of $\mathcal{W} \setminus U$ have a slice disjoint from $U$
3. So, $\mathcal{F} \cup \mathcal{W} \setminus U$ is a possible quorum
4. But $Q \cap \mathcal{F} \cup Q \setminus \mathcal{F}$ is a possible quorum



# Totality: The Cascade Theorem

Consider $Q \not\Downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)
Then $Q \setminus \mathcal{F}$ surely cascades to every member of $C$

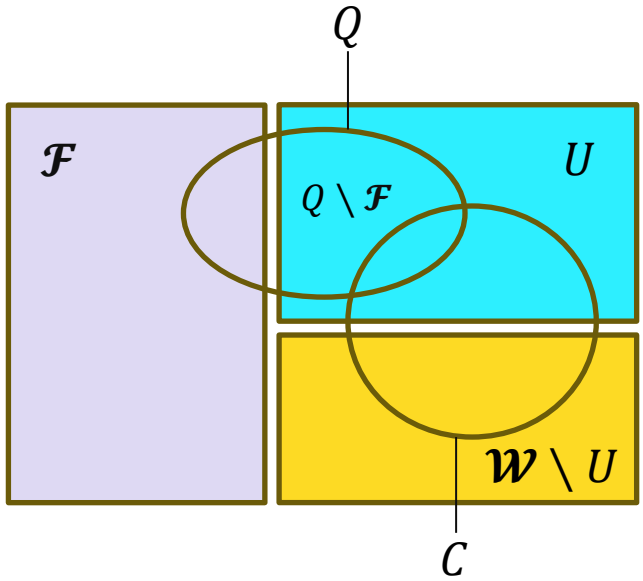
1. Assume toward a contradiction that $Q \setminus \mathcal{F}$ surely cascades only to $U$ such that $C \not\subseteq U$
2. So, every member of $\mathcal{W} \setminus U$ have a slice disjoint from $U$
3. So, $\mathcal{F} \cup \mathcal{W} \setminus U$ is a possible quorum
4. But $Q \cap \mathcal{F} \cup Q \setminus \mathcal{F}$ is a possible quorum
5. And $\mathcal{W} \setminus U$ and $Q \setminus \mathcal{F}$ are disjoint



# Totality: The Cascade Theorem

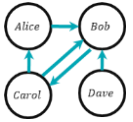
Consider $Q \not\downarrow C$ (say a member of $C$ confirmed after receiving accepts from $Q$)
Then $Q \setminus \mathcal{F}$ surely cascades to every member of $C$

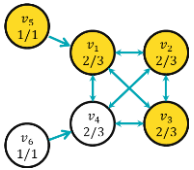
1. Assume toward a contradiction that $Q \setminus \mathcal{F}$ surely cascades only to $U$ such that $C \not\subseteq U$
2. So, every member of $\mathcal{W} \setminus U$ have a slice disjoint from $U$
3. So, $\mathcal{F} \cup \mathcal{W} \setminus U$ is a possible quorum
4. But $Q \cap \mathcal{F} \cup Q \setminus \mathcal{F}$ is a possible quorum
5. And $\mathcal{W} \setminus U$ and $Q \setminus \mathcal{F}$ are disjoint
6. Thus, $C$ is not intertwined. Contradiction!

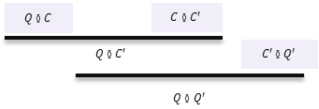


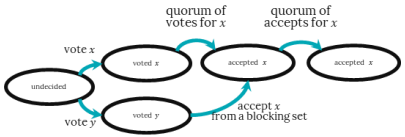
# Recap: Reliable Voting in FBA

1. Validators declare local agreement requirements
2. Subjective quorums emerge as sets that satisfy the requirements of all their members
3. Byzantine validators can “cut” the quorums of well-behaved validators
4. The system partitions itself into disjoint clusters
5. The Federated Voting algorithm solves Reliable Voting for consensus clusters
    - Without initial knowledge of who participates
    - Without knowing which sets are clusters



# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ✓ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- ✓ Quorums in Federated Byzantine Agreement systems
- ✓ Reliable Voting in FBA systems
  - ✓ A simple, safe straw-man protocol (attaching slices to vote messages)
  - ✓ Quorum intersection is not that simple (possible quorums)
  - ✓ Consensus clusters (maximal disjoint clusters are disjoint)
  - ✓ Obtaining Totality: the Federated Voting protocol
- ➢ Total-Order Broadcast by porting Simplex to FBA

# Total-Order Broadcast in FBA with the Simplex algorithm

Simplex is a popular and simple blockchain consensus algorithm

- Chan and Pass, ***Simplex consensus: A simple and fast consensus protocol***, TCC 2023
- Shoup, ***Sing a song of Simplex***, DISC 2024

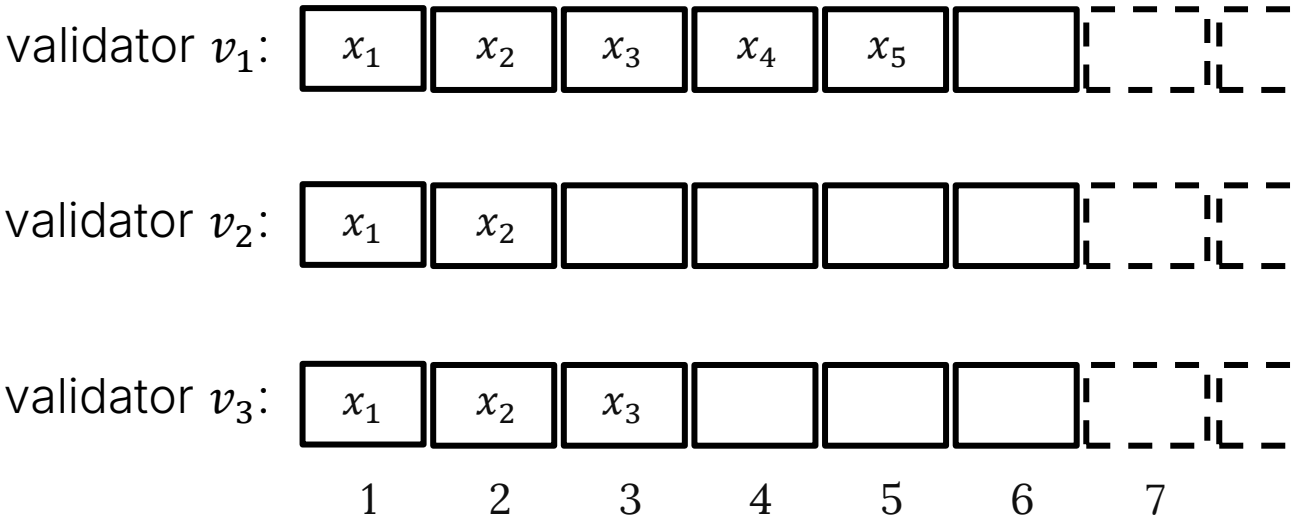
We will now see how we can phrase Simplex in terms of two building blocks: Reliable Voting and leader-election

This will make it easy to port Simplex to FBA

Note: for historical reasons, the Stellar network uses the Stellar Consensus Protocol (SCP), but Simplex is simpler to explain

# Remember Total-Order Broadcast

Each validator has a local copy of a shared log of values $x_1$, $x_2$, $x_3$, etc. and we must guarantee Validity, Agreement, and Progress



# First attempt: agree on which value to assign to each successive slot of the log using Reliable Voting

For slot $s$ from $0$ to $\infty$:

1. Arm a slot timer
2. Wait for a proposal from the pre-assigned slot leader (e.g., from validator number $(s\%N) + 1$, if we know $N$)
3. Vote for the leader's proposal using Reliable Voting
4. Upon confirming $x$ or timing out, go to the next slot

# First attempt: agree on which value to assign to each successive slot of the log using Reliable Voting

For slot $s$ from $0$ to $\infty$:

1. Arm a slot timer
2. Wait for a proposal from the pre-assigned slot leader (e.g., from validator number $(s\%N) + 1$, if we know $N$)
3. Vote for the leader's proposal using Reliable Voting
4. Upon confirming $x$ or timing out, go to the next slot

If a value is confirmed in a slot, we eventually have agreement on it

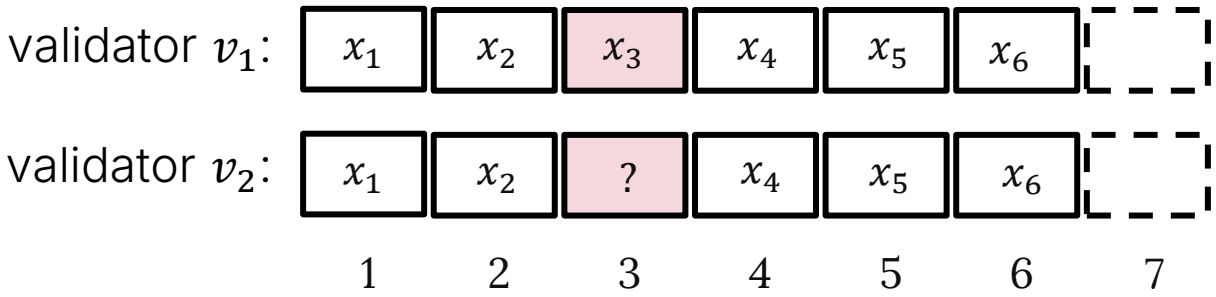
# First attempt: agree on which value to assign to each successive slot of the log using Reliable Voting

For slot $s$ from $0$ to $\infty$:

1. Arm a slot timer
2. Wait for a proposal from the pre-assigned slot leader (e.g., from validator number $(s\%N)+1$, if we know $N$)
3. Vote for the leader's proposal using Reliable Voting
4. Upon confirming $x$ or timing out, go to the next slot

If a value is confirmed in a slot, we eventually have agreement on it

Problem: no agreement on whether a slot timed out or not



# The simplex solution

Each slot $s$, arm a timer and then vote twice:

1. On a pair $(x, s')$, proposed by the slot leader, where $s'$ is the slot's parent
2. On whether the first vote completed timely (before the timer) or late (timed out)

# The simplex solution

Each slot $s$, arm a timer and then vote twice:

1. On a pair $(x, s')$, proposed by the slot leader, where $s'$ is the slot's parent
2. On whether the first vote completed timely (before the timer) or late (timed out)

Voting rule: vote for $(x, s')$ only if:

- all slots between $s'$ and $s$ are confirmed late
- slot s′ has a confirmed value and parent

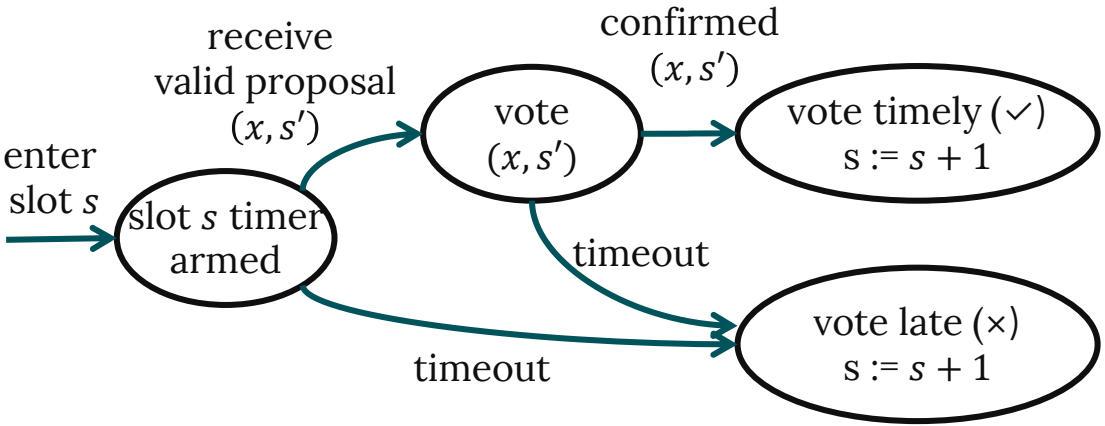
# The simplex solution

Each slot $s$, arm a timer and then vote twice:

1. On a pair $(x, s')$, proposed by the slot leader, where $s'$ is the slot's parent
2. On whether the first vote completed timely (before the timer) or late (timed out)

Voting rule: vote for $(x, s')$ only if:

- all slots between $s'$ and $s$ are confirmed late
- slot s′ has a confirmed value and parent




slot $s$

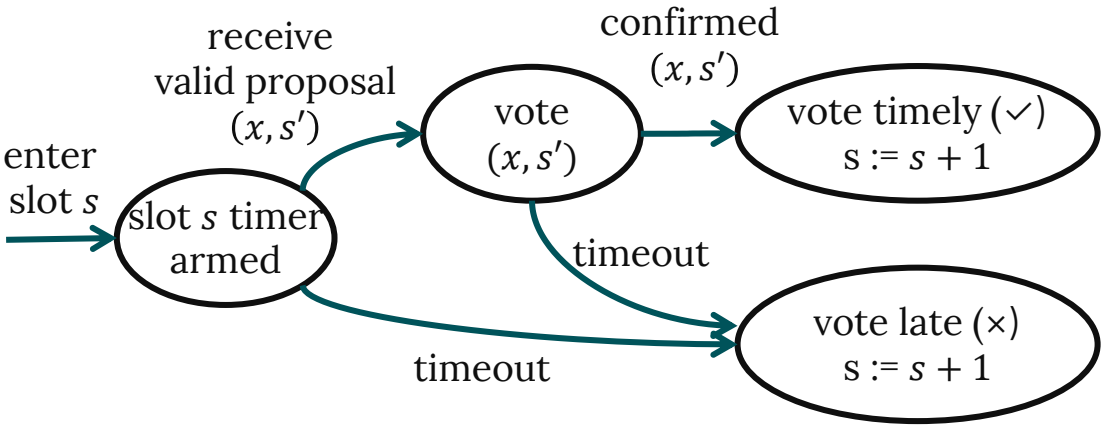
# The simplex solution

Each slot $s$, arm a timer and then vote twice:

1. On a pair $(x, s')$, proposed by the slot leader, where $s'$ is the slot's parent
2. On whether the first vote completed timely (before the timer) or late (timed out)

Voting rule: vote for $(x, s')$ only if:

- all slots between $s'$ and $s$ are confirmed late
- slot s′ has a confirmed value and parent




slot $s$

4 possible slot states:

$(x, s')$ ✓

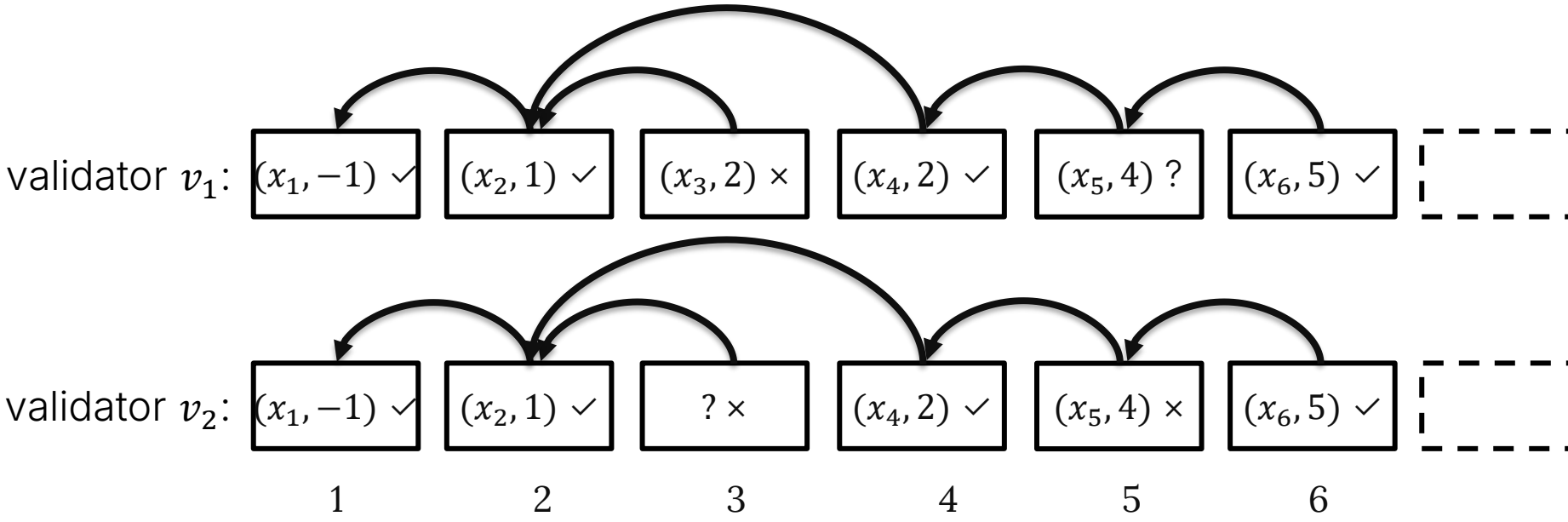
$(x, s')$ ?

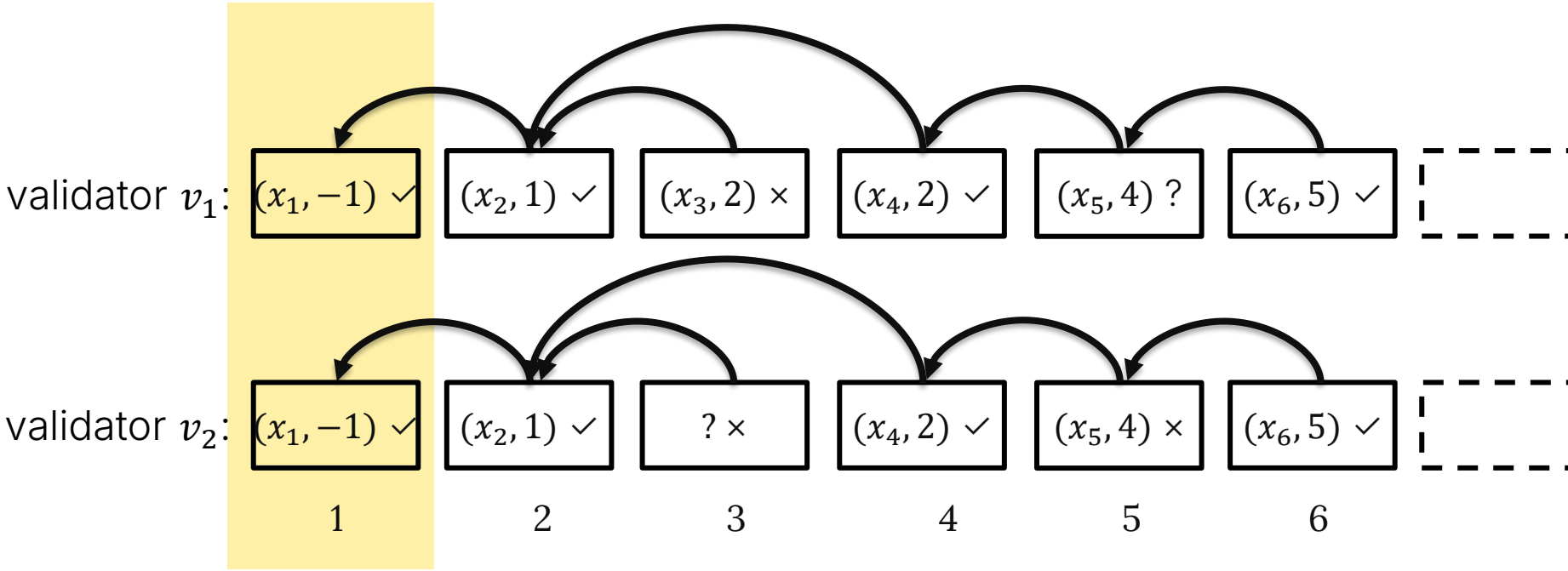
$(x, s')$ ×

? ×

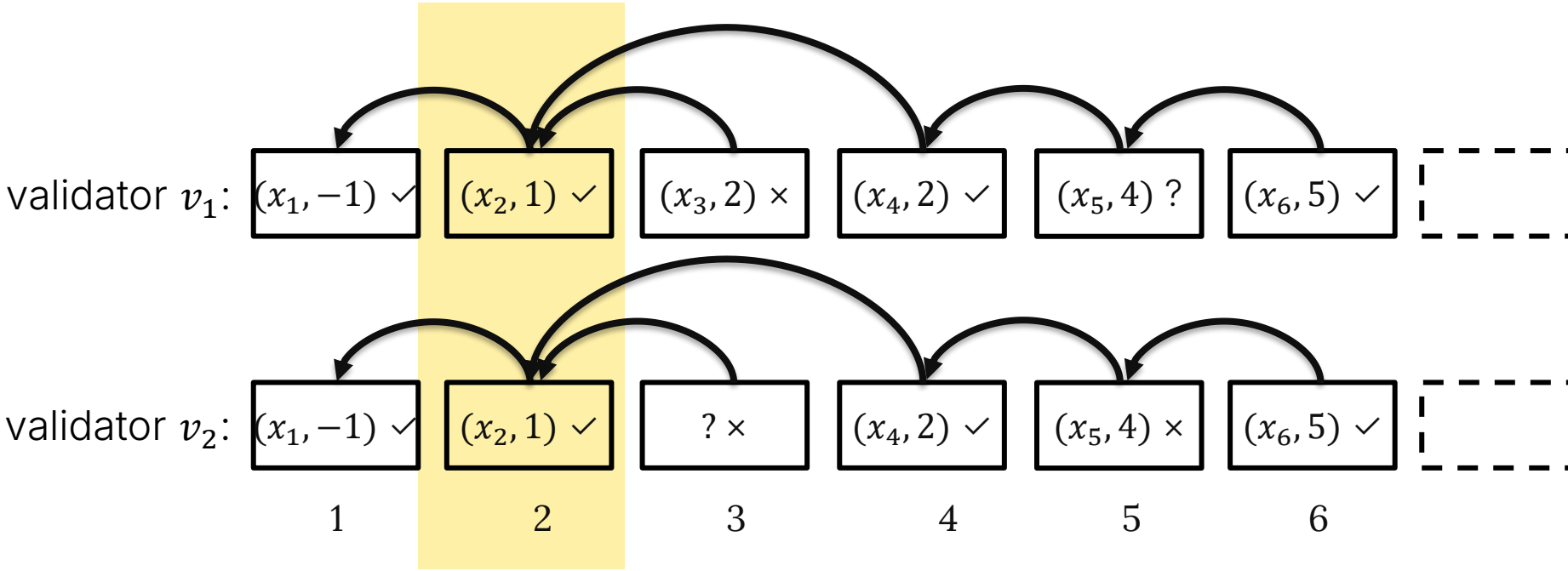
Impossible slot state:

? ✓

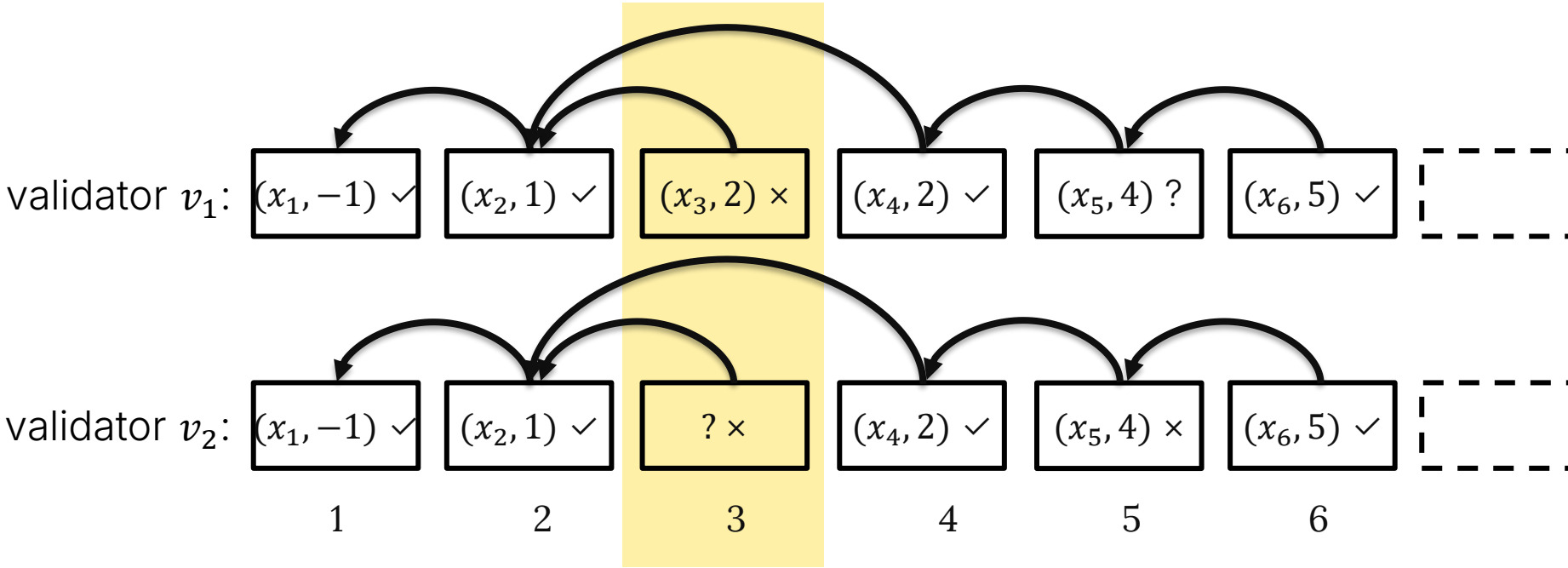

validator $v_1$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
$(x_3, 2)$ ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ?
$(x_6, 5)$ ✓
validator $v_2$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
? ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ×
$(x_6, 5)$ ✓
1
2
3
4
5
6


validator $v_1$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
$(x_3, 2)$ ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ?
$(x_6, 5)$ ✓
validator $v_2$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
? ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ×
$(x_6, 5)$ ✓
1
2
3
4
5
6

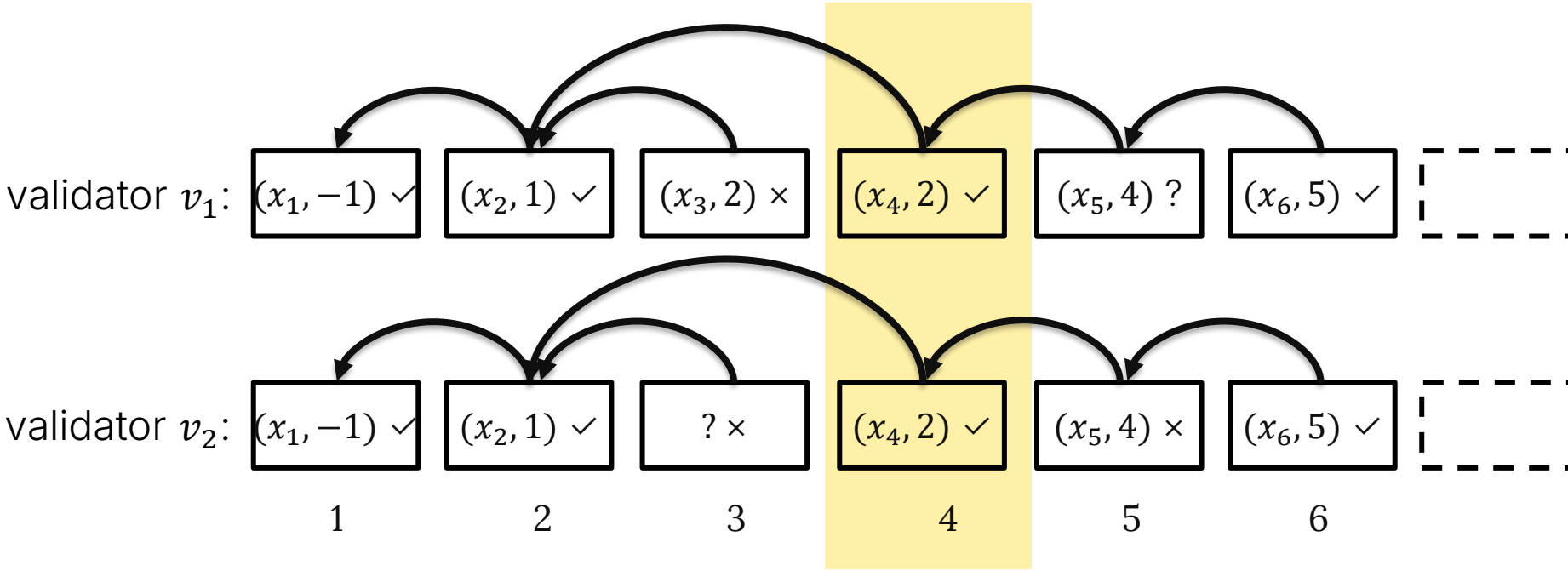
log: $x_1$


validator $v_1$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
$(x_3, 2)$ ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ?
$(x_6, 5)$ ✓
validator $v_2$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
? ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ×
$(x_6, 5)$ ✓
1
2
3
4
5
6

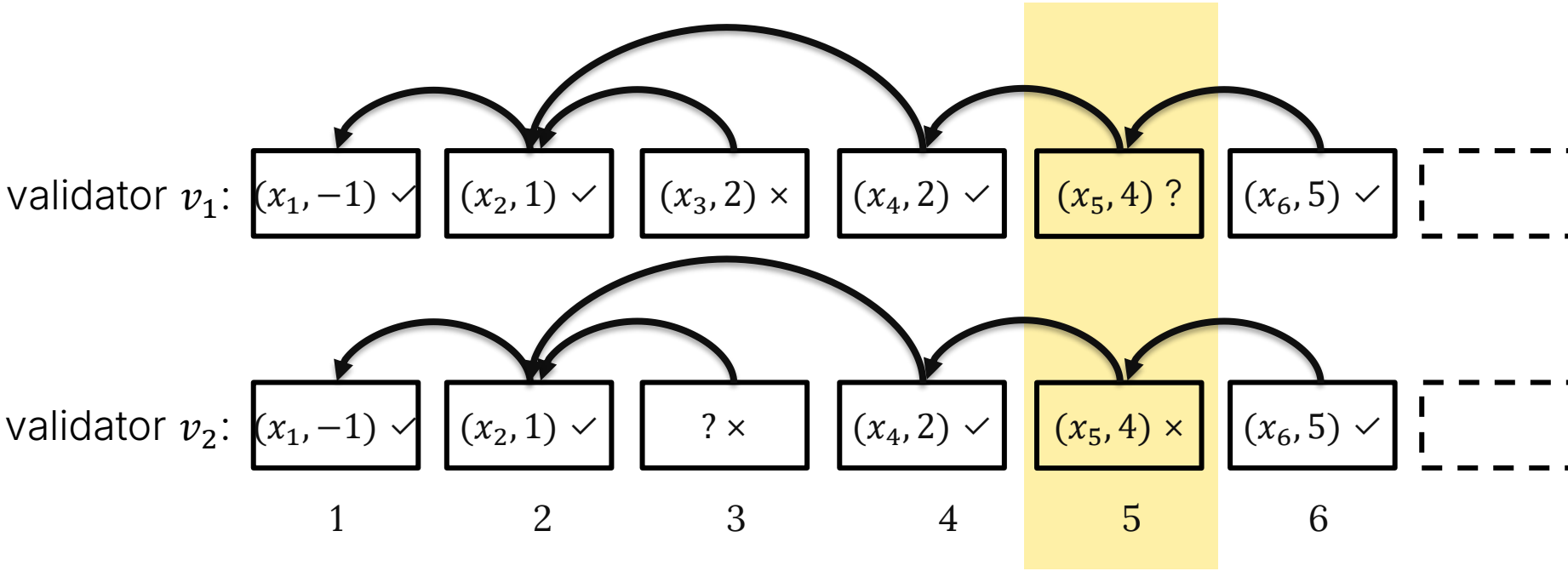
log: $x_1, x_2$


validator $v_1$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
$(x_3, 2)$ ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ?
$(x_6, 5)$ ✓
validator $v_2$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
? ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ×
$(x_6, 5)$ ✓
1
2
3
4
5
6

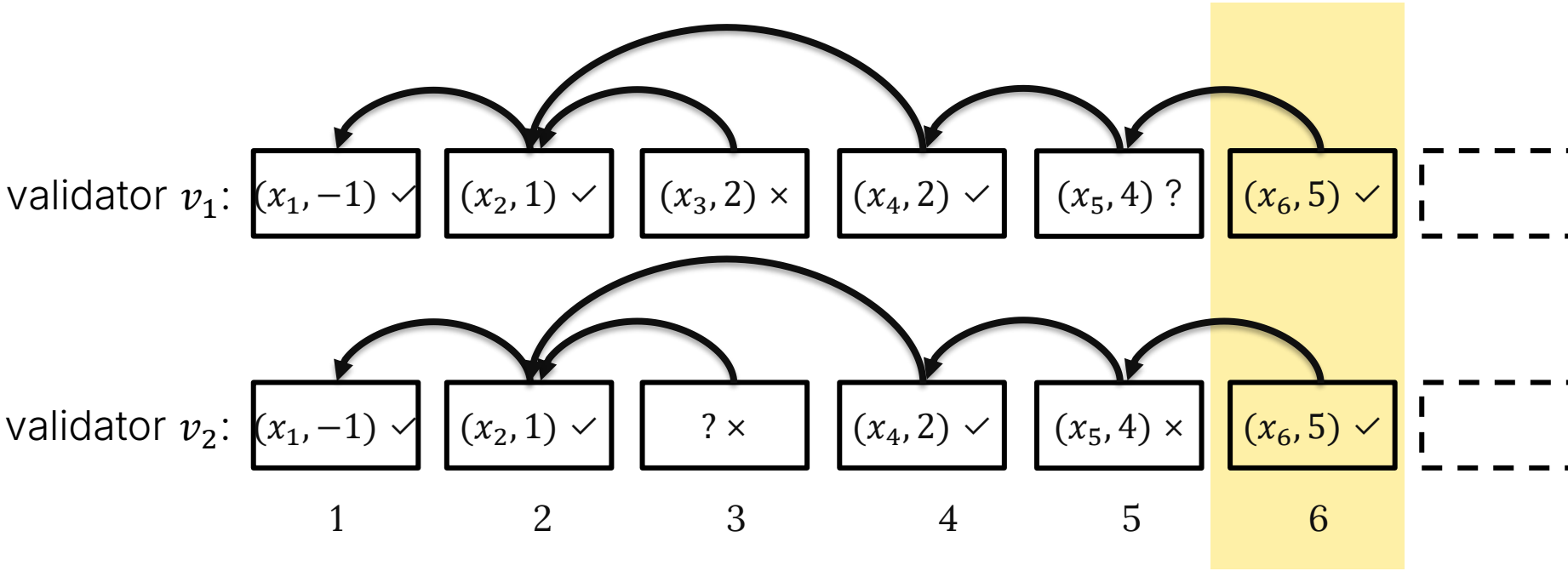
log: $x_1, x_2$


validator $v_1$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
$(x_3, 2)$ ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ?
$(x_6, 5)$ ✓
validator $v_2$:
$(x_1, -1)$ ✓
$(x_2, 1)$ ✓
? ×
$(x_4, 2)$ ✓
$(x_5, 4)$ ×
$(x_6, 5)$ ✓
1
2
3
4
5
6

log: $x_1, x_2, x_4$

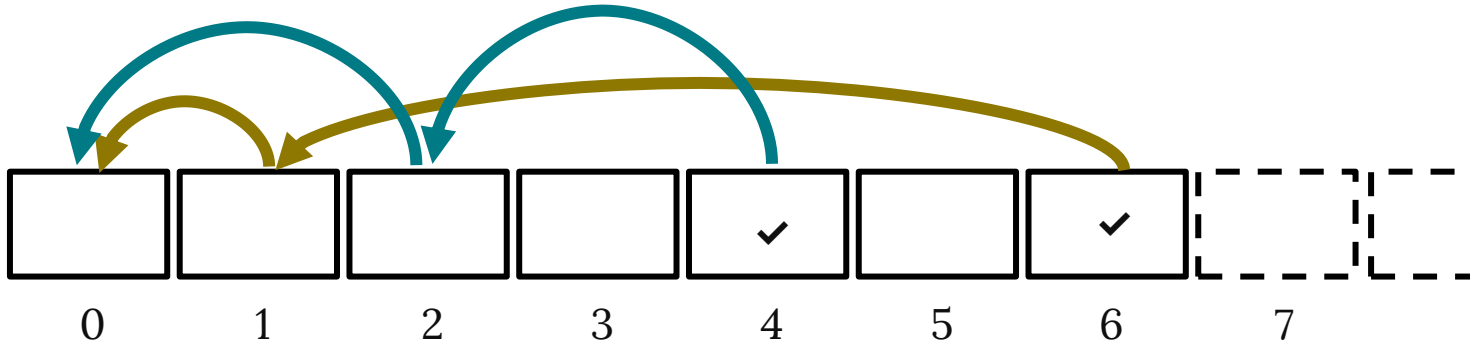



log: $x_1, x_2, x_4, x_5$



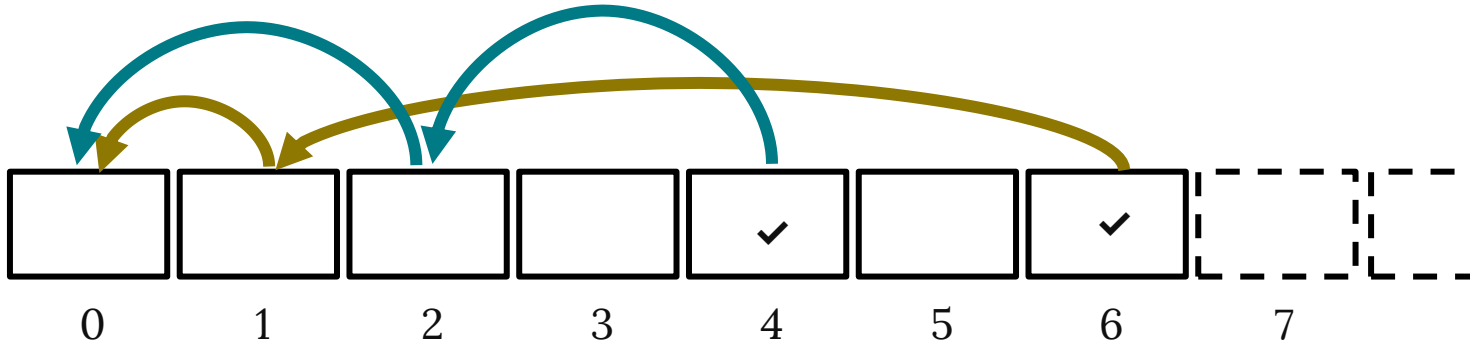

log: $x_1, x_2, x_4, x_5, x_6$

# Agreement is guaranteed



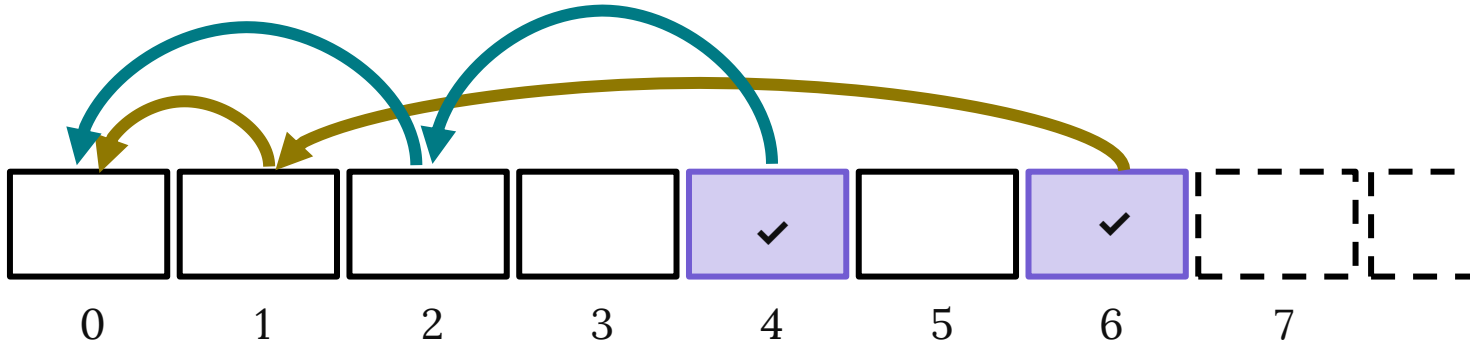
# Agreement is guaranteed

If we have a fork, consider the two confirmed-timely slots *s* and *s'* at the tips.



# Agreement is guaranteed

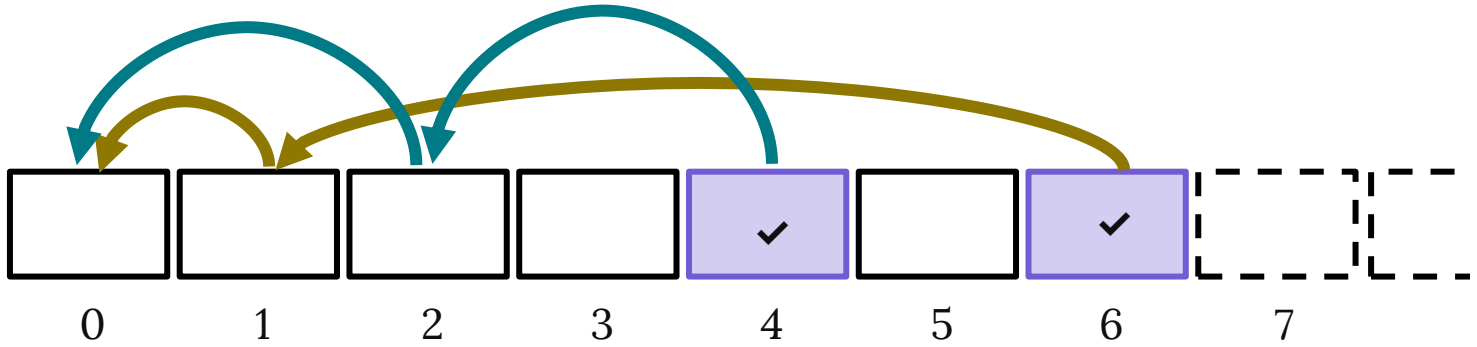
If we have a fork, consider the two confirmed-timely slots $s$ and $s'$ at the tips.



# Agreement is guaranteed

If we have a fork, consider the two confirmed-timely slots *s* and *s'* at the tips.

The branch of the higher tip must jump over the lower tip

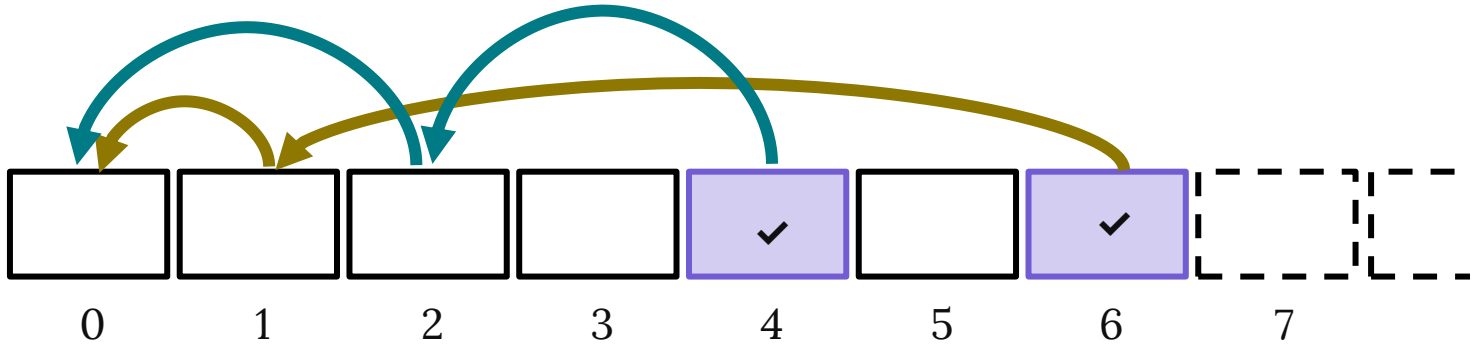


# Agreement is guaranteed

If we have a fork, consider the two confirmed-timely slots *s* and *s'* at the tips.

The branch of the higher tip must jump over the lower tip

This contradicts the fact that slots between a slot and its parent must be confirmed late



Simplex is never stuck because, each slot, at least one of the two votes succeeds

# Simplex is never stuck because, each slot, at least one of the two votes succeeds

**Lemma**:
At least one of the two votes terminates

# Simplex is never stuck because, each slot, at least one of the two votes succeeds

**Lemma**:
At least one of the two votes terminates

**Proof**:

Suppose that the first vote gets stuck

Then all well-behaved validators time out and vote for “skip” in the second vote

By Unanimity of Reliable Voting, they all confirm the slot skipped

# In periods of synchrony, slots commit

If validators' timers are large enough compared to the network delay and the slot leader is well-behaved, then all well-behaved validators commit

# In periods of synchrony, slots commit

If validators' timers are large enough compared to the network delay and the slot leader is well-behaved, then all well-behaved validators commit

- Under asynchrony, we can keep starting new slots (we won't get stuck), until...

# In periods of synchrony, slots commit

If validators' timers are large enough compared to the network delay and the slot leader is well-behaved, then all well-behaved validators commit

- Under asynchrony, we can keep starting new slots (we won't get stuck), until...
- Under synchrony, new slots that have a well-behaved leader commit

# Simplex in FBA

We have seen that simplex relies on two building blocks:

- Reliable Voting
- Assigning leaders to slots

✓ We have already seen how to implement Reliable Voting

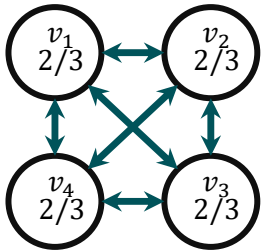
▪ We need a way to assign leaders to slots!

# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$

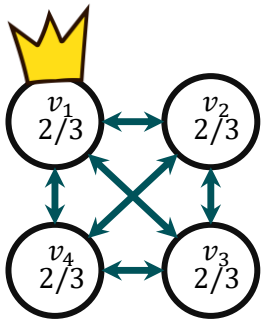
# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$



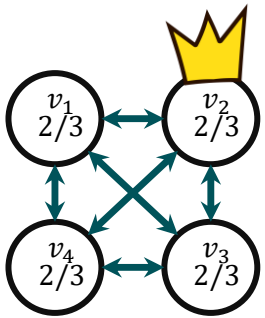
# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$



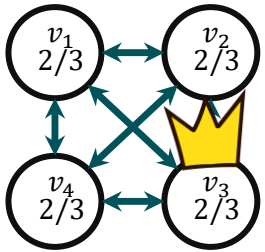
# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$



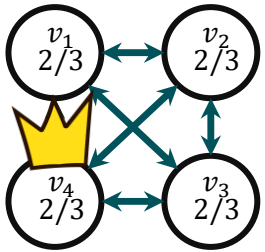
# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$



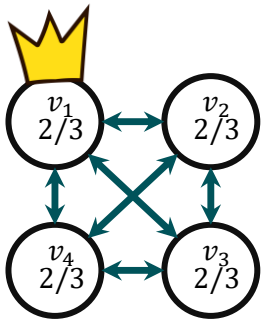
# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$



# Leader election in BFT closed systems

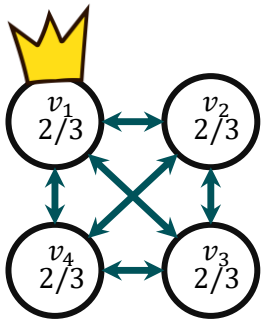
If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$



# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$

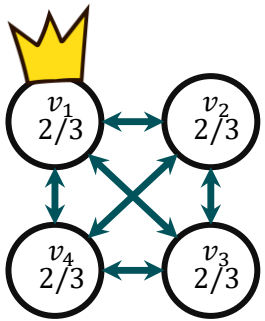
Ensures fair rotation but blindly elects crashed validators



# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$

Ensures fair rotation but blindly elects crashed validators



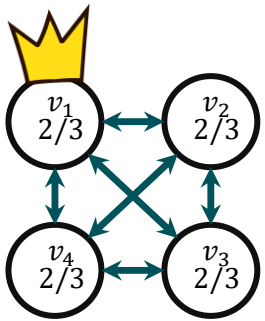

Another solution:

- Derive a pseudorandom ranking from the slot number (e.g., hash validator ID + slot number)
- Track the set $K$ of known-online validators (those we heard from recently)
- Pick the highest-ranked validator among $\mathrm{K} \cup \{self\}$

# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$

Ensures fair rotation but blindly elects crashed validators




Another solution:

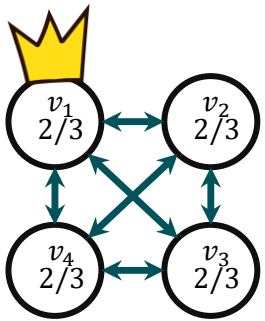
- Derive a pseudorandom ranking from the slot number (e.g., hash validator ID + slot number)
- Track the set $K$ of known-online validators (those we heard from recently)
- Pick the highest-ranked validator among $\mathrm{K} \cup \{self\}$

In synchronous, crash-only periods, this avoids electing crashed validators, guarantees agreement on the slot leader, and achieves fair rotation

# Leader election in BFT closed systems

If validators know $N$, we can say validator number $(s\%N) + 1$ is leader of slot $s$

Ensures fair rotation but blindly elects crashed validators




Another solution:

- Derive a pseudorandom ranking from the slot number (e.g., hash validator ID + slot number)
- Track the set $K$ of known-online validators (those we heard from recently)
- Pick the highest-ranked validator among $\mathrm{K} \cup \{self\}$

*Problem: Does not work in FBA because attackers can pollute $K$ with as many validators as they want (often called a Sybil attack)*

# Assigning leaders to slots in FBA

We want to assign a leader to each slot $s$ such that:

# Assigning leaders to slots in FBA

We want to assign a leader to each slot $s$ such that:

- The leader role rotates fairly

# Assigning leaders to slots in FBA

We want to assign a leader to each slot $s$ such that:

- The leader role rotates fairly
- If a validator has crashed, it is not assigned as leader of new slots

# Assigning leaders to slots in FBA

We want to assign a leader to each slot $s$ such that:

- The leader role rotates fairly
- If a validator has crashed, it is not assigned as leader of new slots
- Disagreement on the leader is okay if it is unlikely under normal conditions (validators disagreeing on the slot leader can prevent committing, but is not a safety issue)

# Assigning leaders to slots in FBA

Idea: Just adapt the BFT solution

- Derive a pseudorandom ranking from the slot number (e.g., hash validator ID + slot number)
- Track the set $K$ of known-online *members of the validator's own quorum slices*
- Pick the highest-ranked validator among $K \cup \{self\}$

# Assigning leaders to slots in FBA

Idea: Just adapt the BFT solution

- Derive a pseudorandom ranking from the slot number (e.g., hash validator ID + slot number)
- Track the set $K$ of known-online *members of the validator's own quorum slices*
- Pick the highest-ranked validator among $K \cup \{self\}$

Problem

If validators have significantly different sets of quorum slices, agreement is unlikely because their $K$s will be significantly different

# Assigning leaders to slots: first improvement

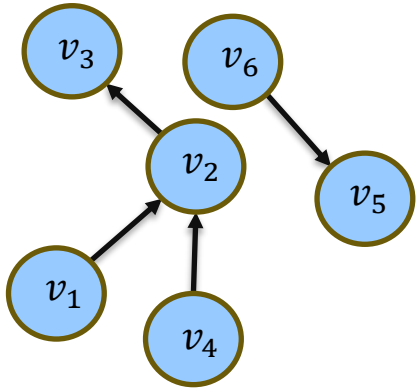
Interpret each validator's initial leader choice (call it a candidate) as an edge in a directed graph

- This forms a set of disjoint, rooted trees
- Take the root as leader of its tree

# Assigning leaders to slots: first improvement

Interpret each validator's initial leader choice (call it a candidate) as an edge in a directed graph

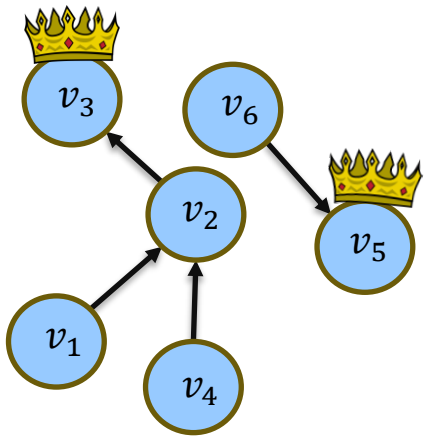
- This forms a set of disjoint, rooted trees
- Take the root as leader of its tree



# Assigning leaders to slots: first improvement

Interpret each validator's initial leader choice (call it a candidate) as an edge in a directed graph

- This forms a set of disjoint, rooted trees
- Take the root as leader of its tree

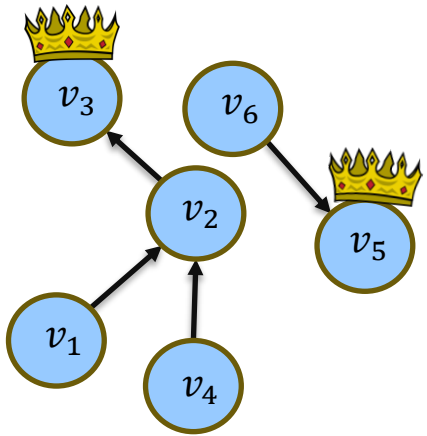


# Assigning leaders to slots: first improvement

Interpret each validator's initial leader choice (call it a candidate) as an edge in a directed graph

- This forms a set of disjoint, rooted trees
- Take the root as leader of its tree

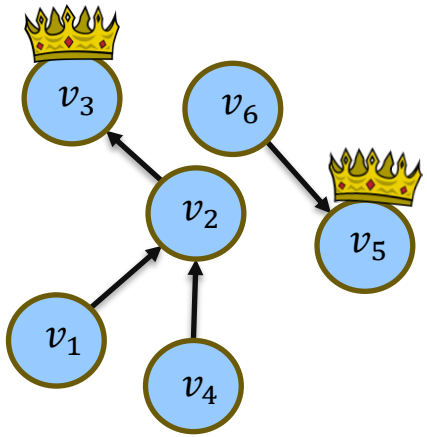
This reduces the number of leaders, but disagreement is still too likely



# Assigning leaders to slots: second improvement

Use Federated Voting to reach agreement on the leader

✓ If a quorum of a cluster initially agrees, then the full cluster will confirm the leader



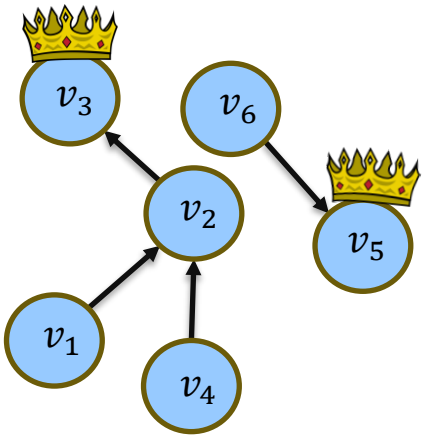
# Assigning leaders to slots: second improvement

Use Federated Voting to reach agreement on the leader

✓ If a quorum of a cluster initially agrees, then the full cluster will confirm the leader

Problem

- Federated Voting can get stuck
- Then could we just repeat?
- Same problem as with the TOB non-solution: there is no agreement on whether a vote got stuck or not



# Assigning leaders to slots: final solution

Observation: initial leader votes are determined pseudo-randomly using a common seed, so the result of  Federated Voting is deterministic and can be simulated locally “in the head”

# Assigning leaders to slots: final solution

Observation: initial leader votes are determined pseudo-randomly using a common seed, so the result of Federated Voting is deterministic and can be simulated locally “in the head”

Leader-assignment Algorithm

# Assigning leaders to slots: final solution

Observation: initial leader votes are determined pseudo-randomly using a common seed, so the result of Federated Voting is deterministic and can be simulated locally “in the head”

Leader-assignment Algorithm

- Simulate Federated Voting on the leader “in the head”, using local knowledge of others’ slices

# Assigning leaders to slots: final solution

Observation: initial leader votes are determined pseudo-randomly using a common seed, so the result of Federated Voting is deterministic and can be simulated locally “in the head”

Leader-assignment Algorithm

- Simulate Federated Voting on the leader “in the head”, using local knowledge of others’ slices
- Repeat with a fresh pseudorandom ranking until “self” confirms a leader in the simulation

# Assigning leaders to slots: final solution

Observation: initial leader votes are determined pseudo-randomly using a common seed, so the result of Federated Voting is deterministic and can be simulated locally “in the head”

Leader-assignment Algorithm

- Simulate Federated Voting on the leader “in the head”, using local knowledge of others’ slices
- Repeat with a fresh pseudorandom ranking until “self” confirms a leader in the simulation
- This guarantees each cluster eventually agrees on a leader, if local views of others’ slices are accurate

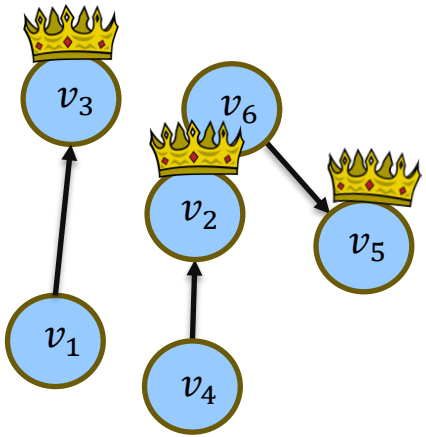
# Assigning leaders to slots: final solution

Observation: initial leader votes are determined pseudo-randomly using a common seed, so the result of Federated Voting is deterministic and can be simulated locally "in the head"

Leader-assignment Algorithm

- Simulate Federated Voting on the leader "in the head", using local knowledge of others' slices
- Repeat with a fresh pseudorandom ranking until "self" confirms a leader in the simulation
- This guarantees each cluster eventually agrees on a leader, if local views of others' slices are accurate

say any 4 is a quorum




round 1: no leader confirmed

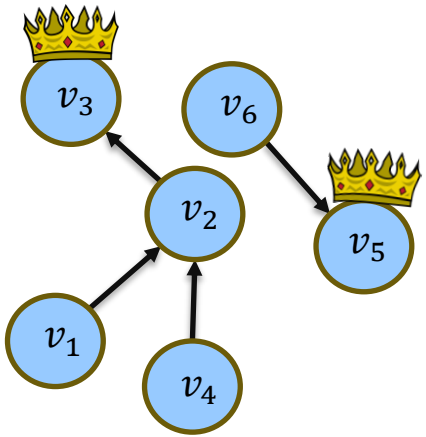
# Assigning leaders to slots: final solution

Observation: initial leader votes are determined pseudo-randomly using a common seed, so the result of Federated Voting is deterministic and can be simulated locally "in the head"

Leader-assignment Algorithm

- Simulate Federated Voting on the leader "in the head", using local knowledge of others' slices
- Repeat with a fresh pseudorandom ranking until "self" confirms a leader in the simulation
- This guarantees each cluster eventually agrees on a leader, if local views of others' slices are accurate

say any 4 is a quorum




round 2: leader confirmed

# Roadmap

- ✓ Intro: the problem of issuer-enforced finality
- ✓ Background: Partial Synchrony, Total-Order Broadcast, BFT quorum systems, Reliable Voting
- ✓ Quorums in Federated Byzantine Agreement systems
- ✓ Reliable Voting in FBA systems
  - ✓ A simple, safe straw-man protocol (attaching slices to vote messages)
  - ✓ Quorum intersection is not that simple (possible quorums)
  - ✓ Consensus clusters (maximal clusters are disjoint)
  - ✓ Obtaining Totality: the Federated Voting protocol (and the cascade theorem)
- ✓ Total-Order Broadcast by porting Simplex to FBA
  - ✓ Simplex in terms of Reliable Voting
  - ✓ Leader election in FBA systems

# References


1. David Mazières, *The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus.* 2015.
2. Lokhava et al., *Fast and secure global payments with Stellar.* SOSP 2019.
3. Losa, Gafni, and Mazières, *Stellar Consensus by Instantiation.* DISC 2019.
4. Florian et al., *The Sum of Its Parts: Analysis of Federated Byzantine Agreement Systems.* Distributed Computing, 2022.
5. Chan and Pass. *Simplex Consensus: A Simple and Fast Consensus Protocol.* TCC 2023.
6. Shoup, *Sing a song of Simplex*, DISC 2024.
7. Dwork, Lynch, and Stockmeyer. *Consensus in the Presence of Partial Synchrony*. Journal of the ACM, 1988.
8. Malkhi and Reiter. *Byzantine quorum systems.* Distributed Computing, 1998.

# The simplified slice generator

# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

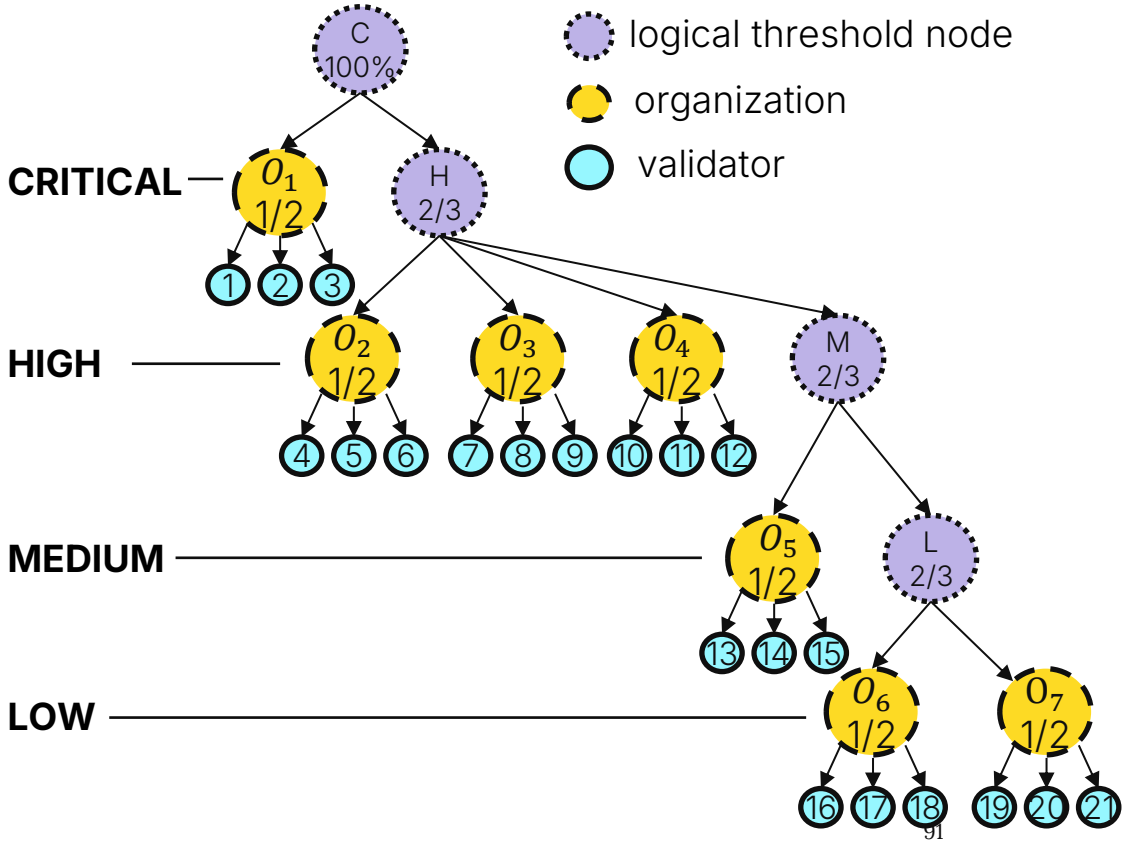
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



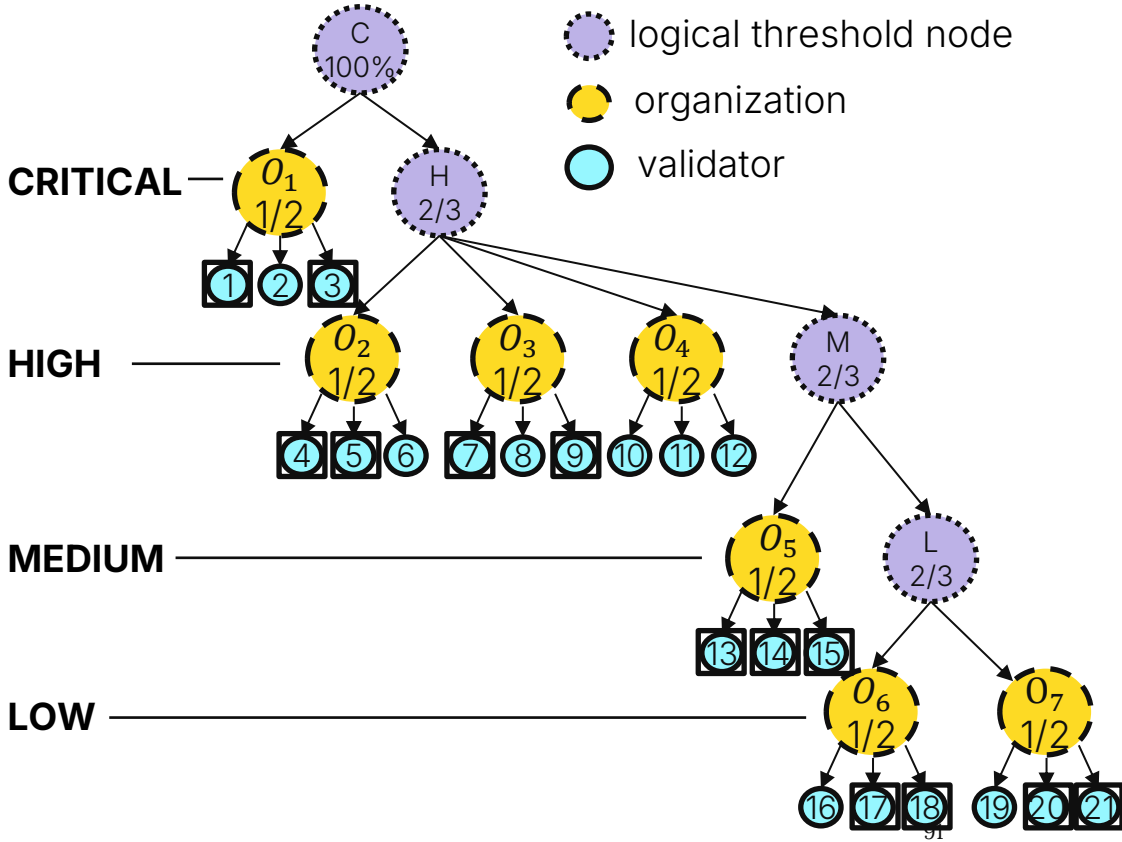
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



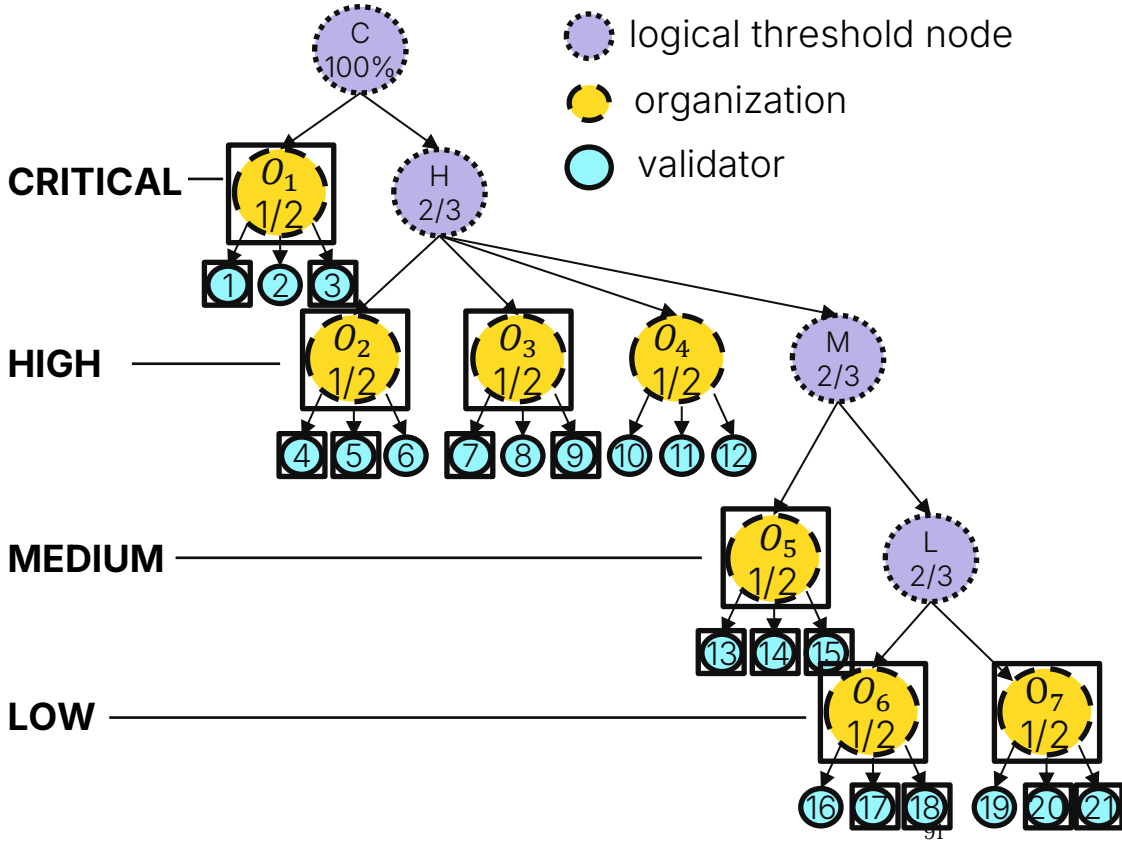
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



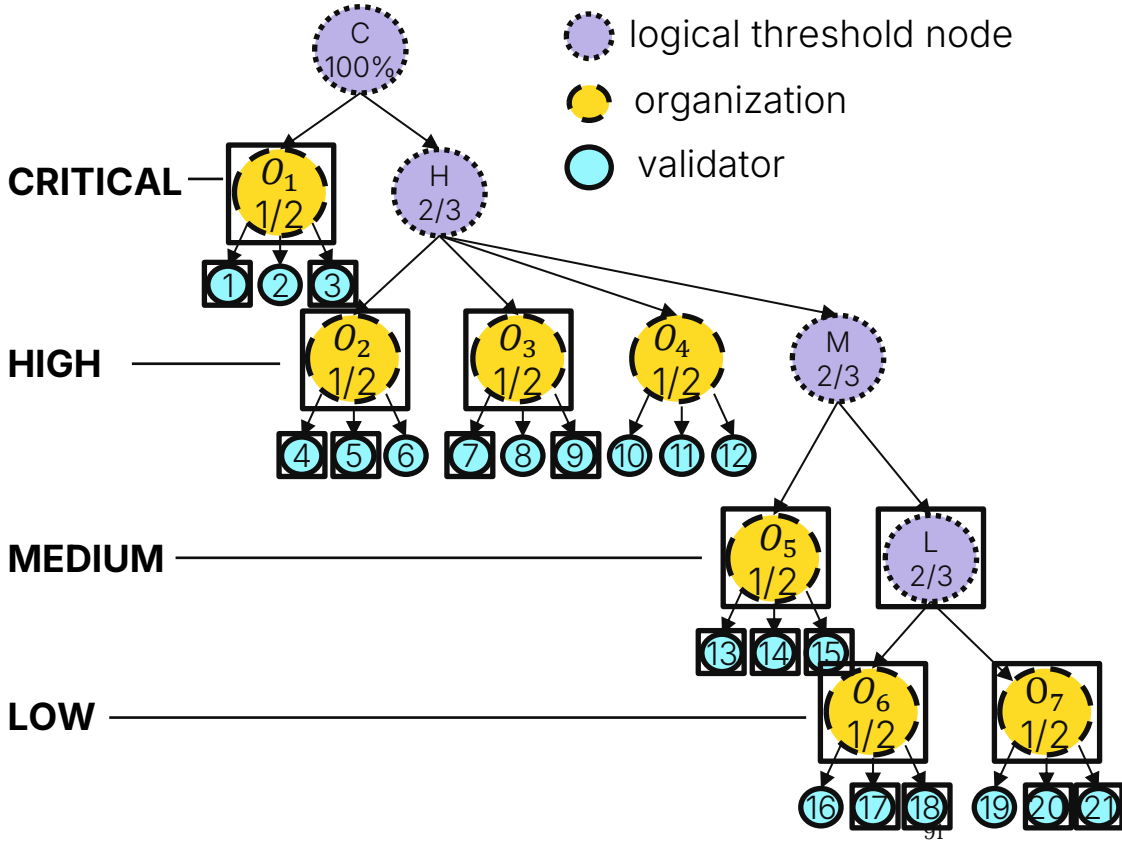
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



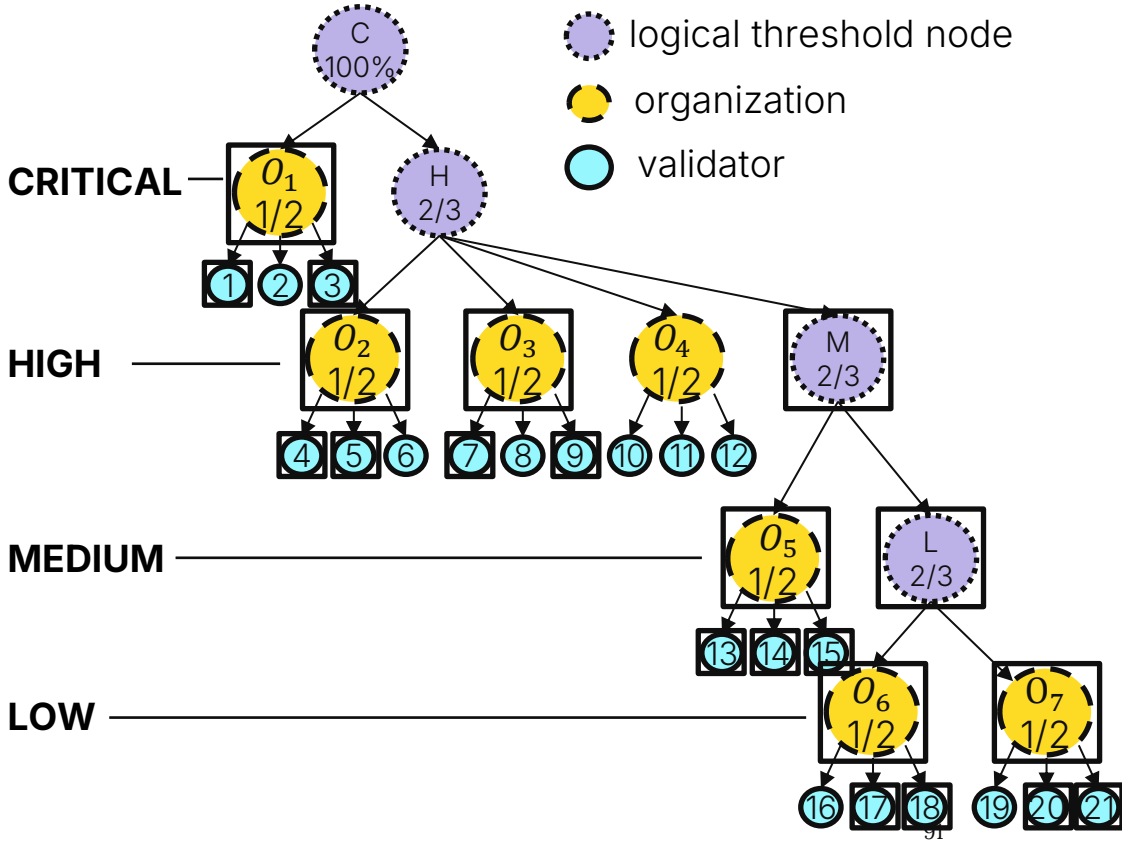
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



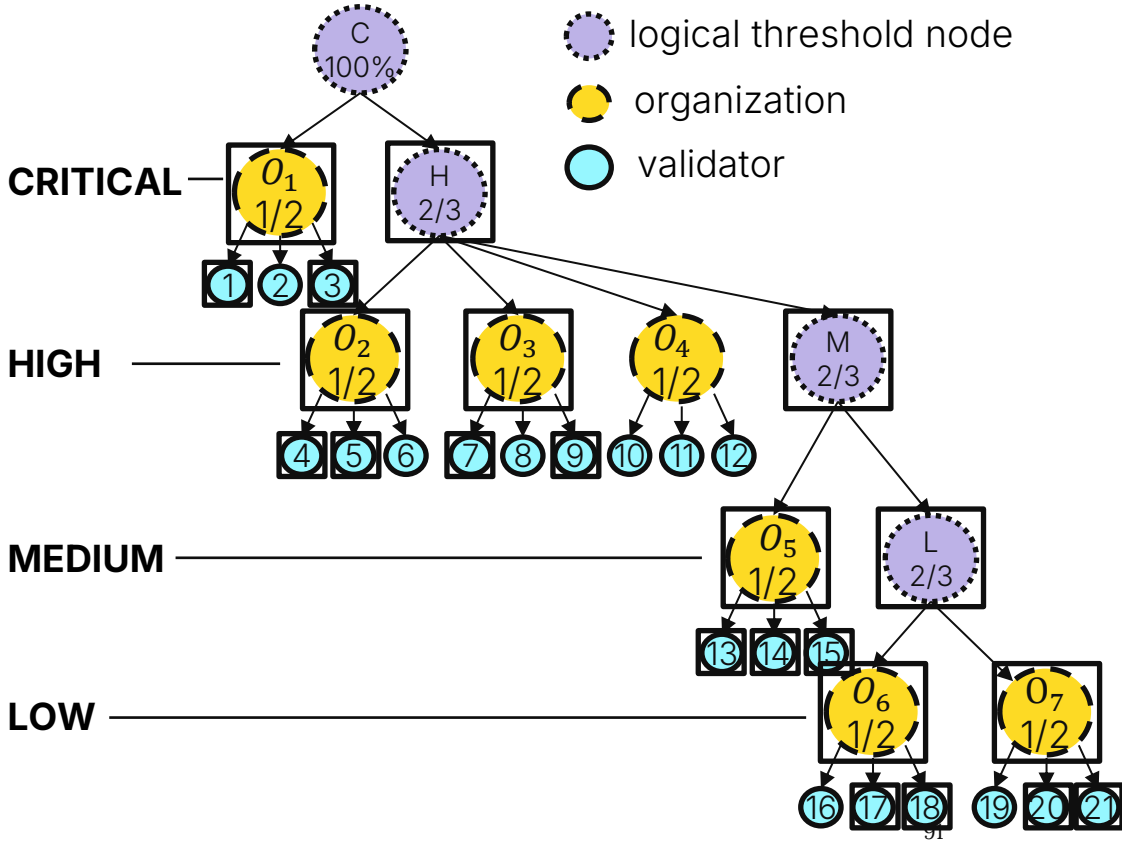
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



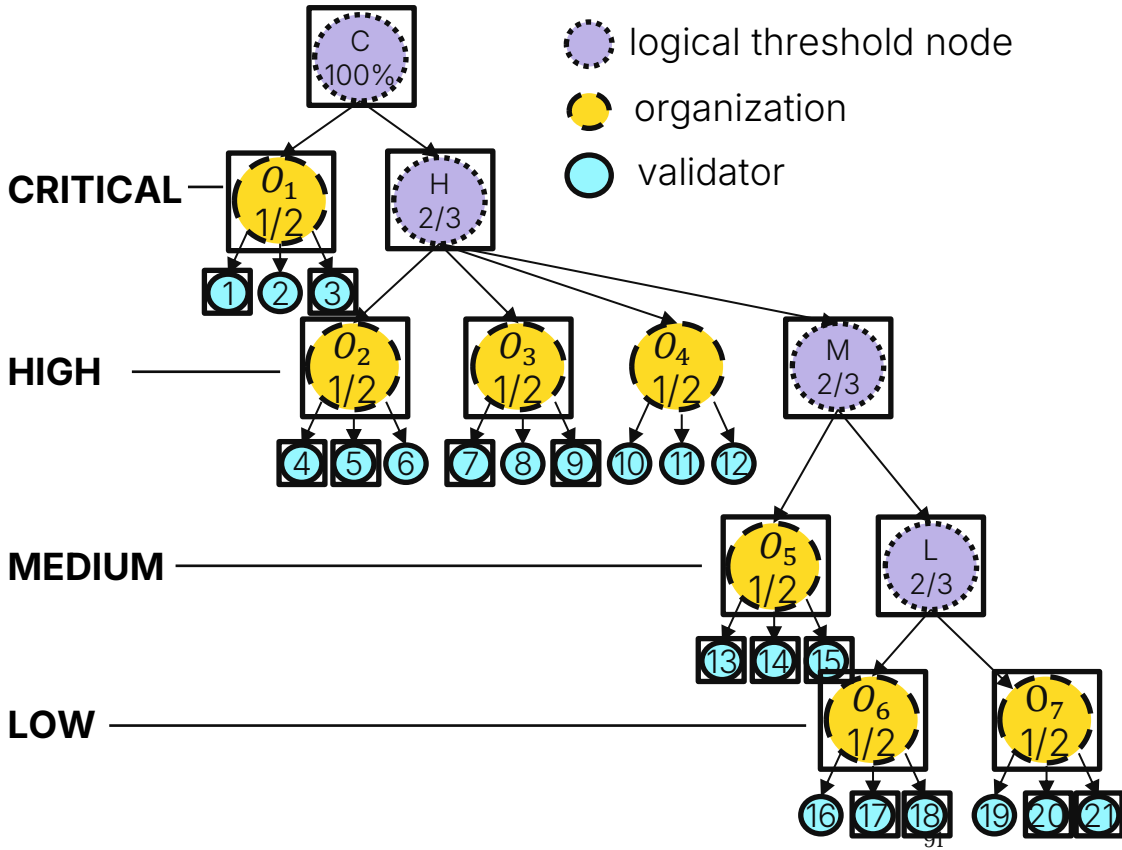
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



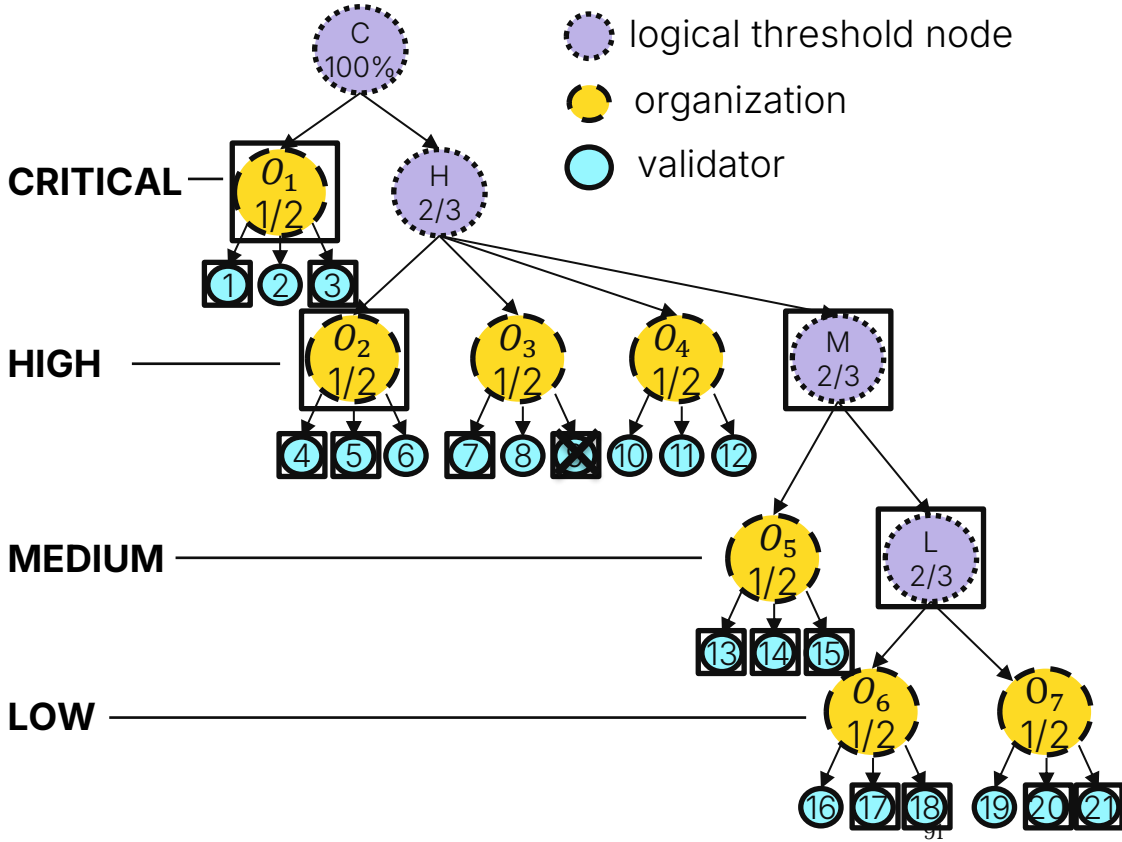
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



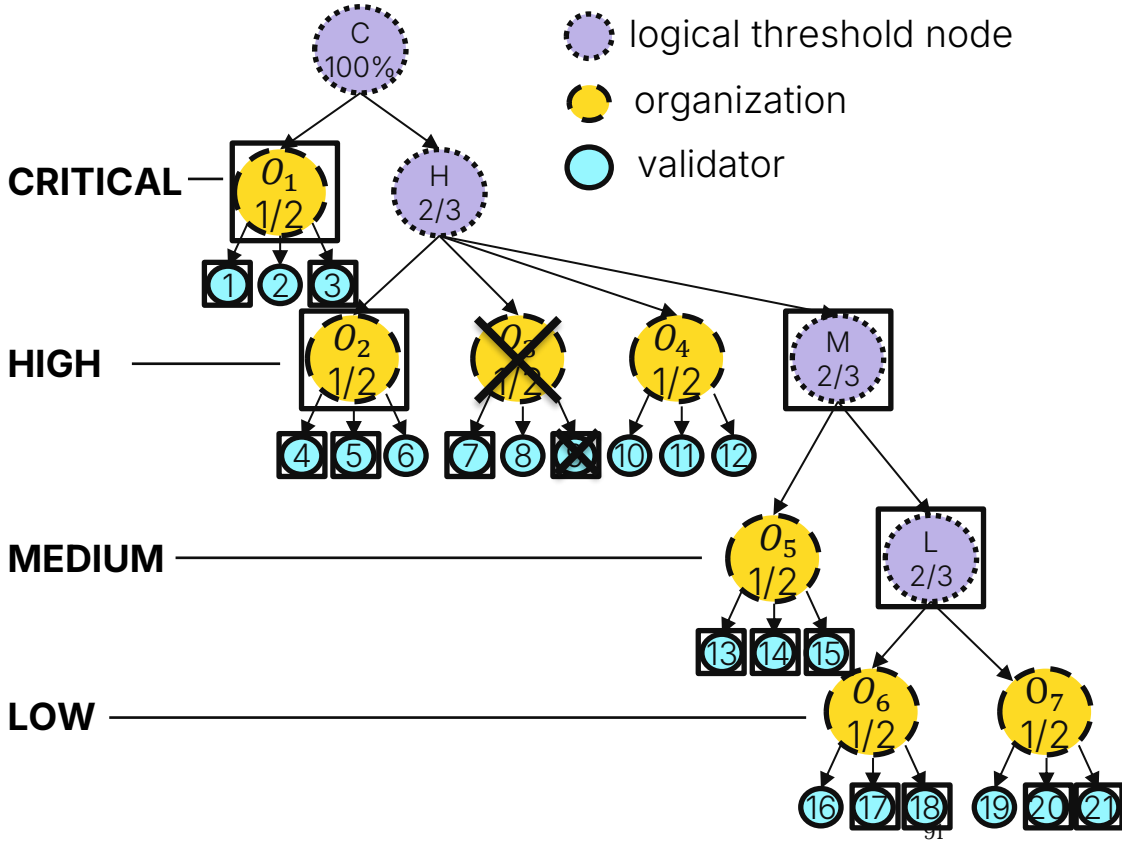
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



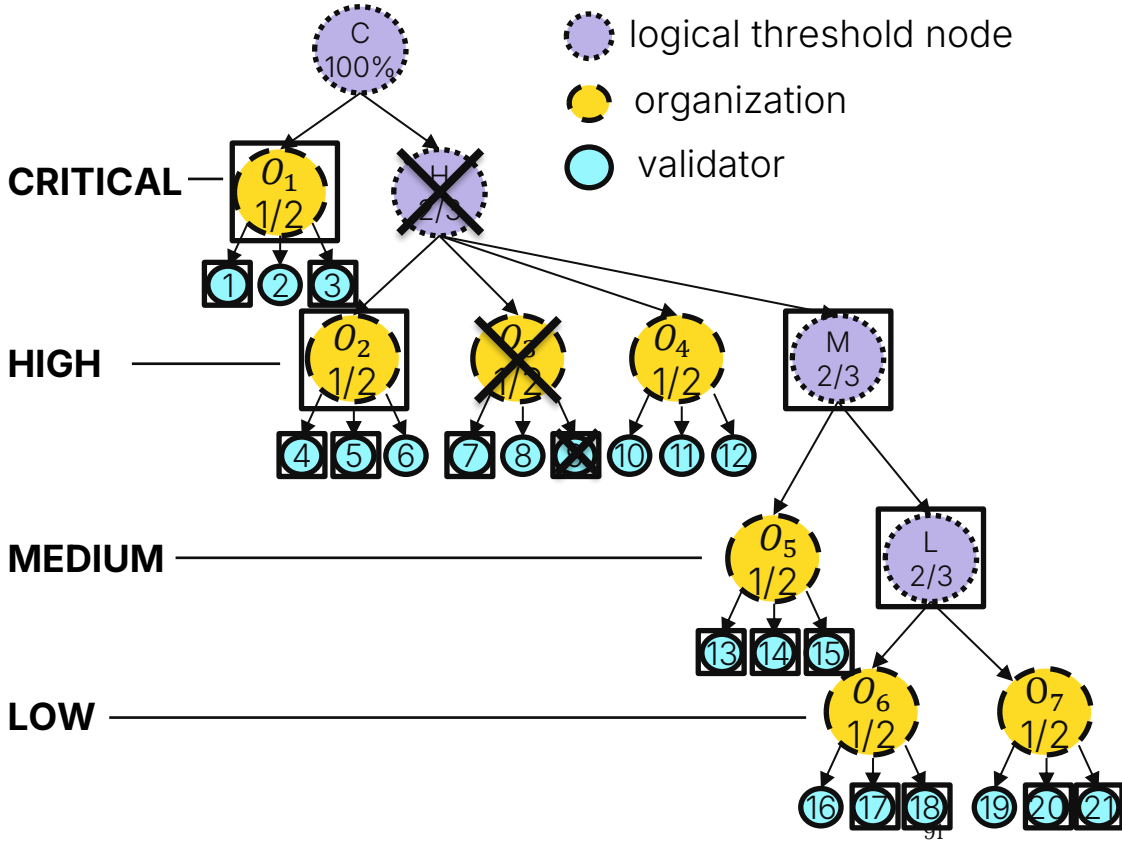
# The simplified slice generator

Organizations run multiple validators (typically 3)

A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



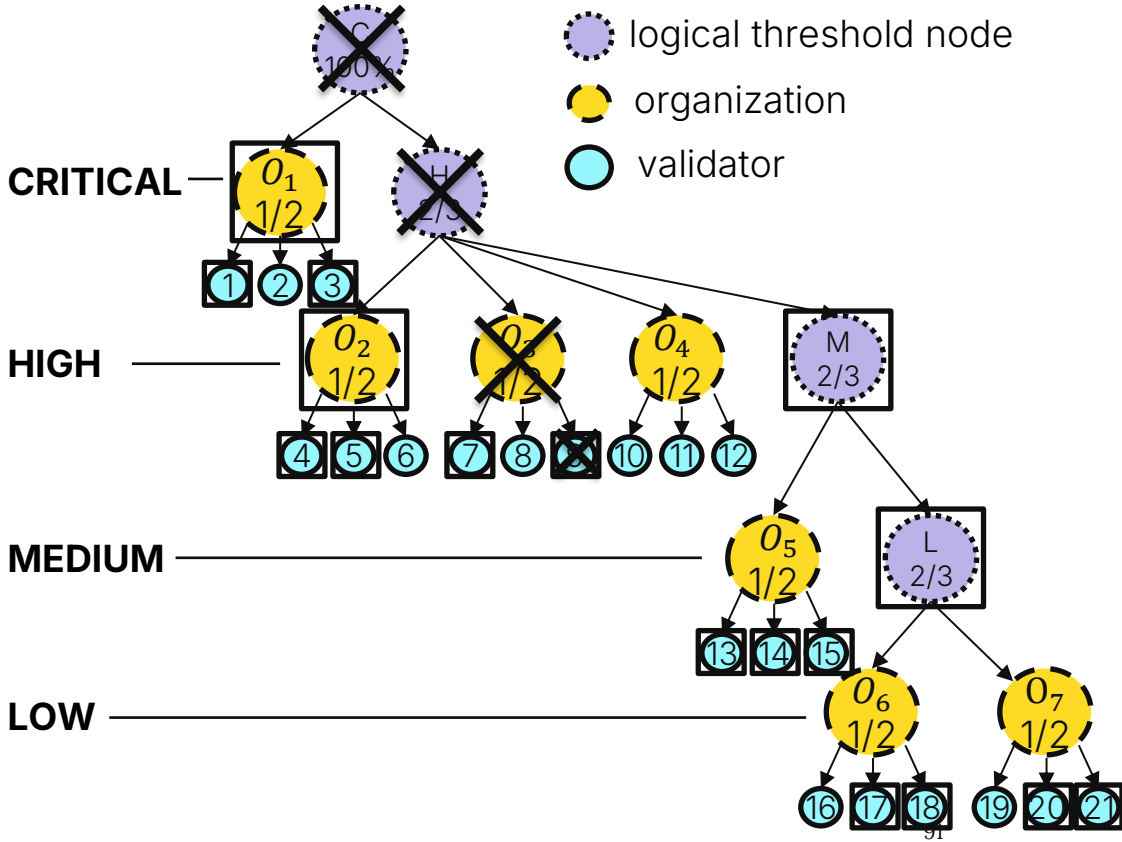
# The simplified slice generator

Organizations run multiple validators (typically 3)

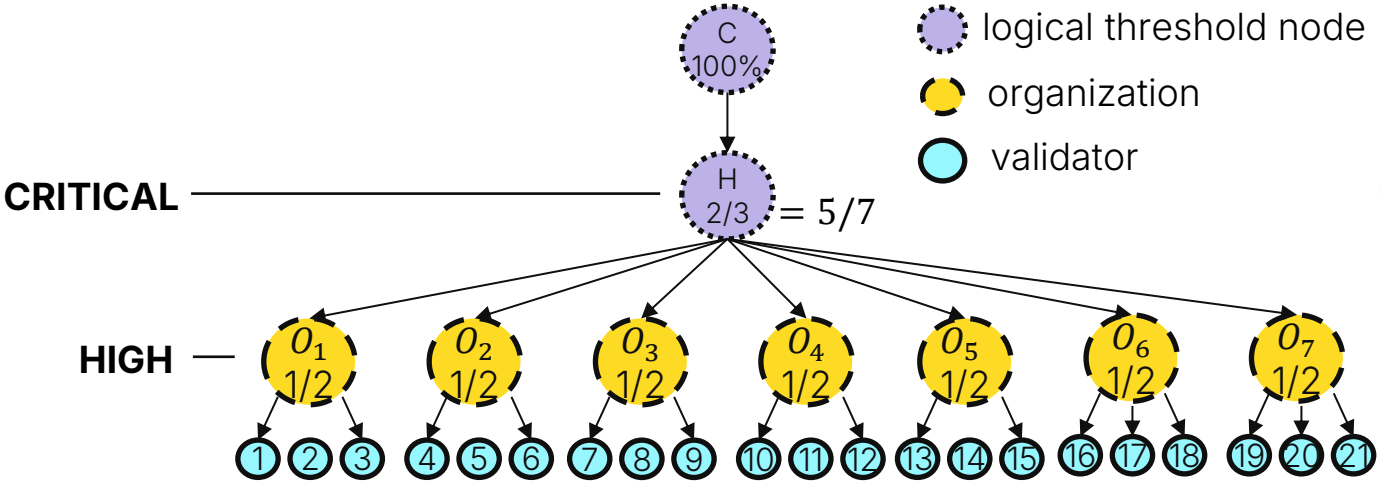
A validator assigns a level to each organization: CRITICAL, HIGH, MEDIUM, or LOW

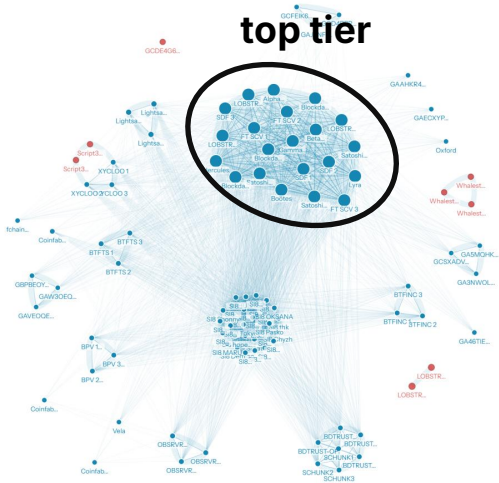
The slice generator creates a tree of thresholds as on the figure

The inter-organization threshold is set to ½ and the per-level threshold is set to 2/3



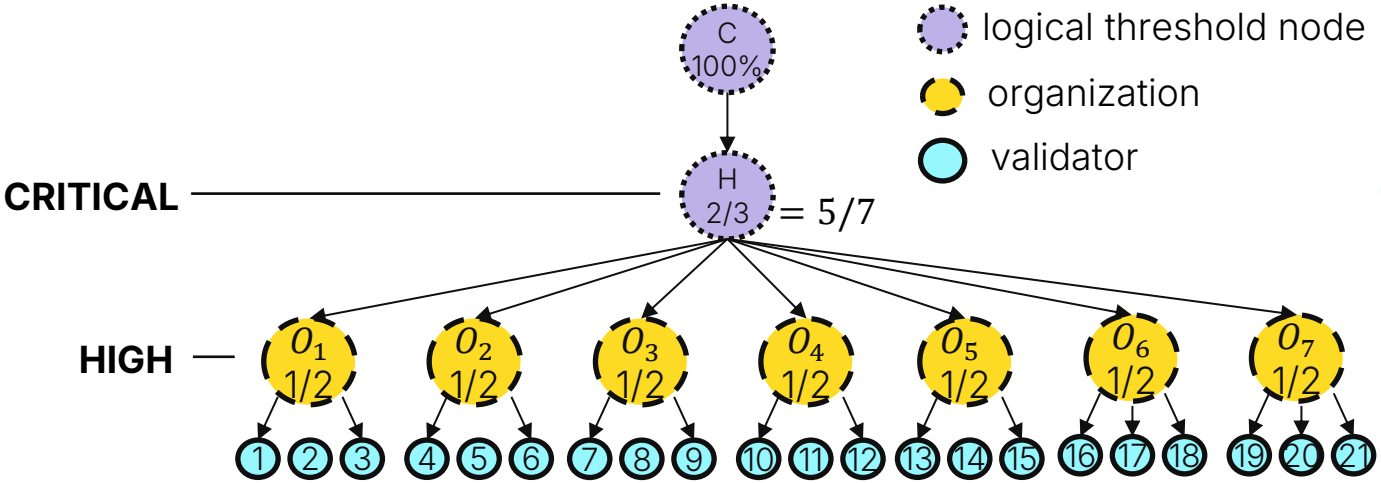
# Current top-tier configuration

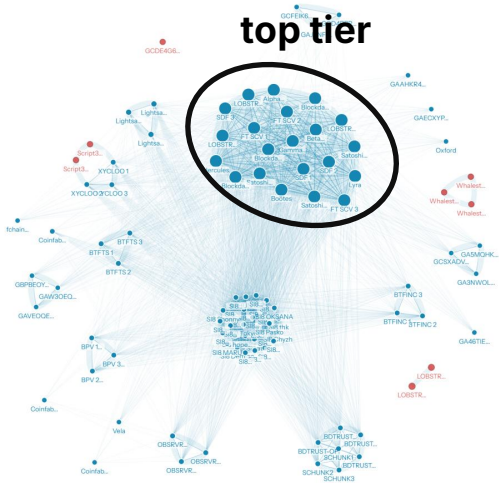



7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

# Current top-tier configuration

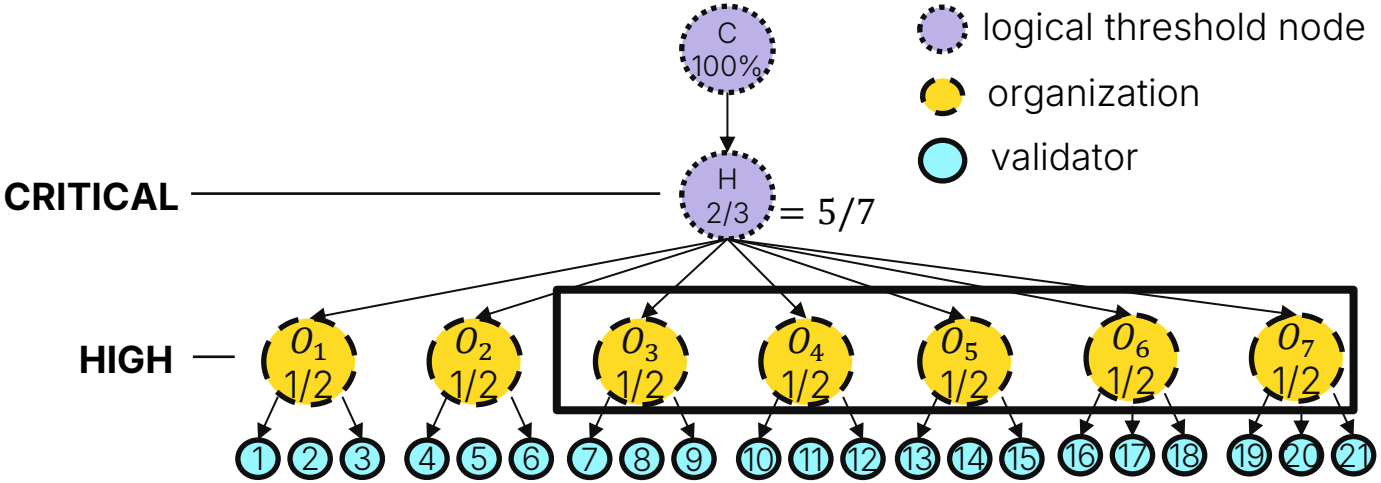


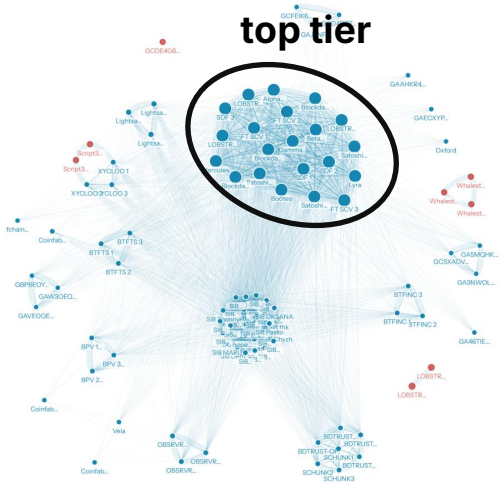




7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

Minimal splitting set:

# Current top-tier configuration

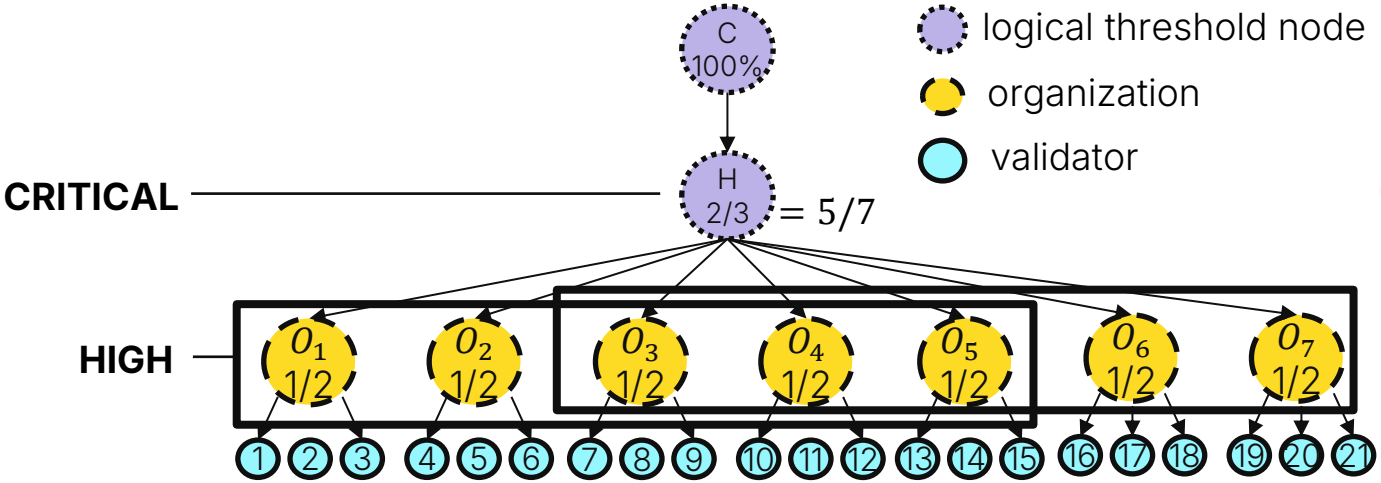


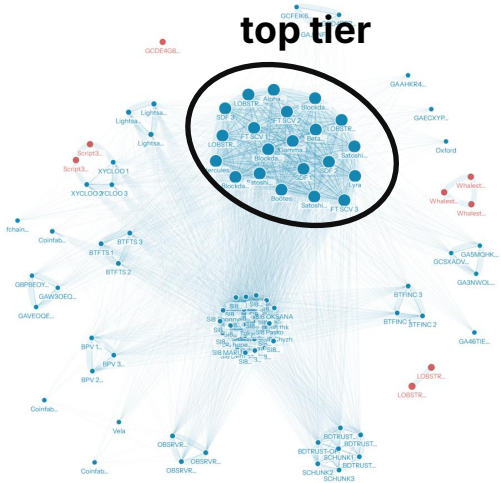




7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

Minimal splitting set:

# Current top-tier configuration

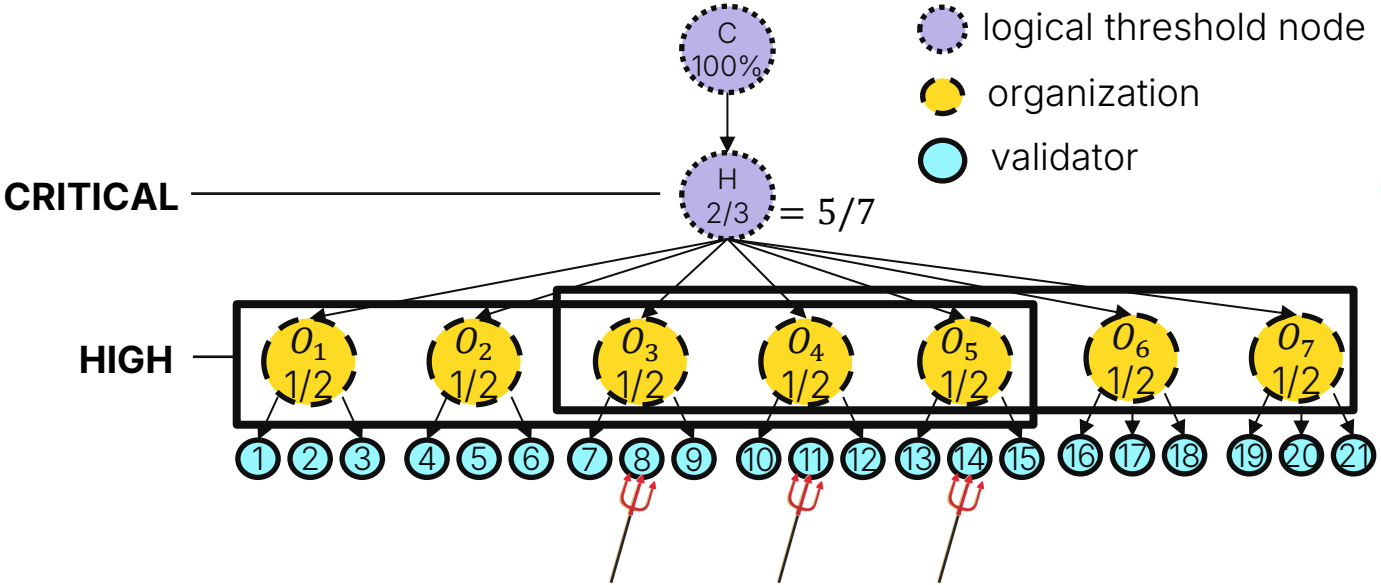


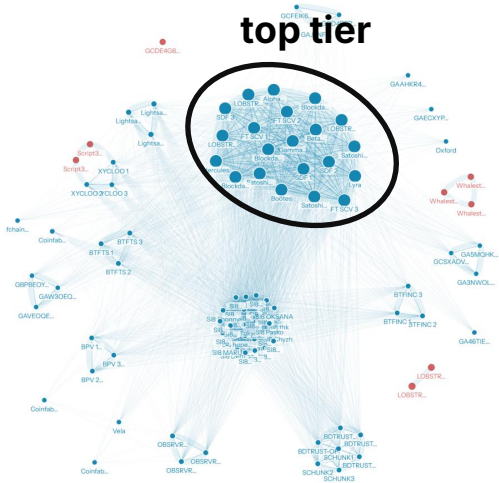




7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

Minimal splitting set:

# Current top-tier configuration

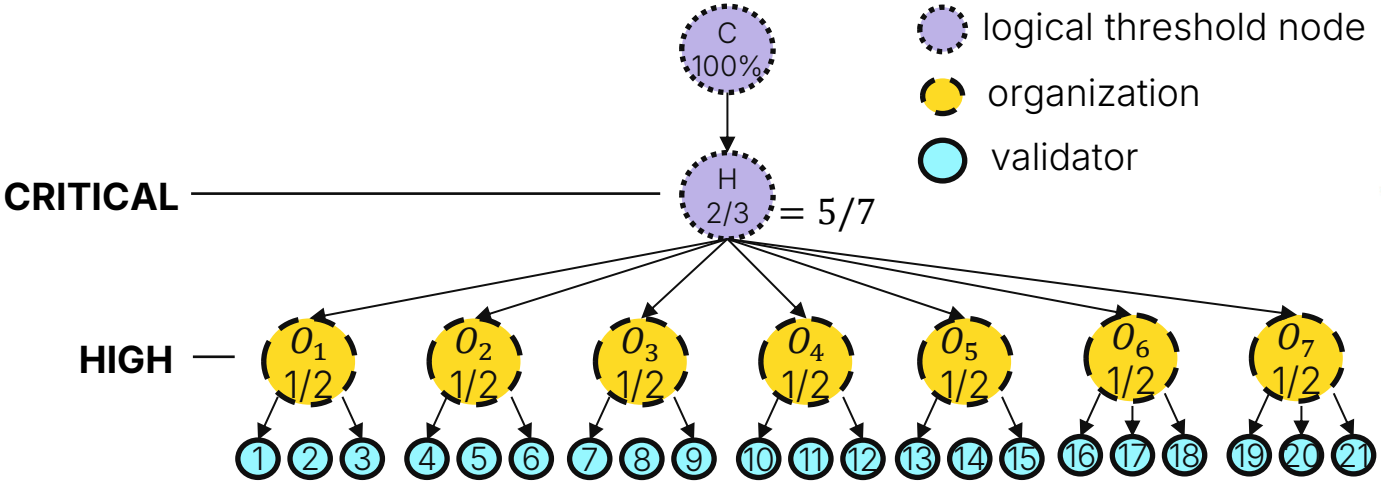


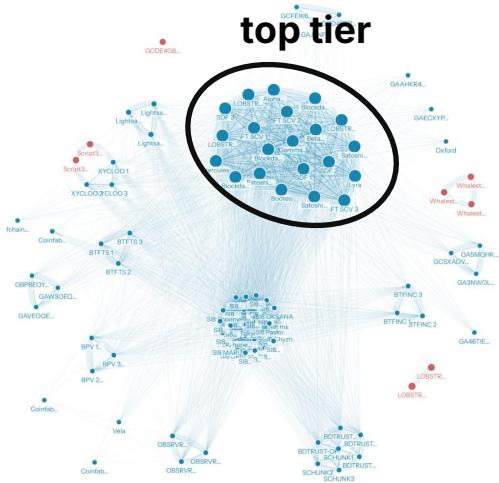




7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

Minimal splitting set:
3 validators

# Current top-tier configuration

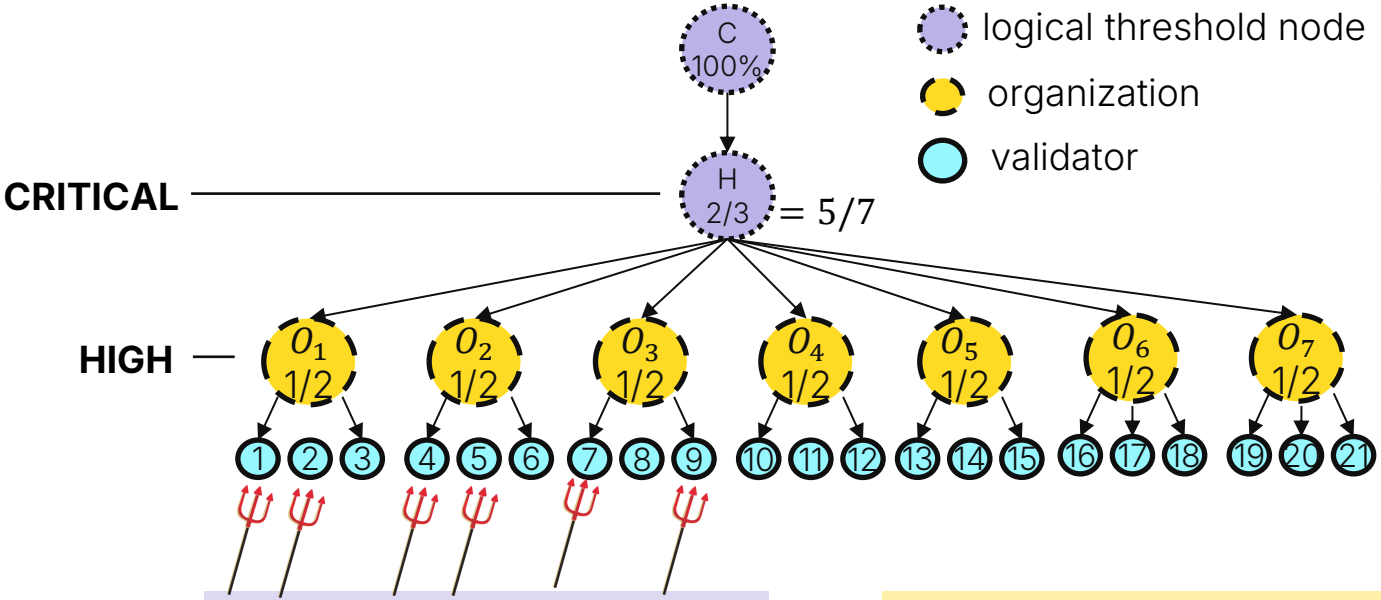


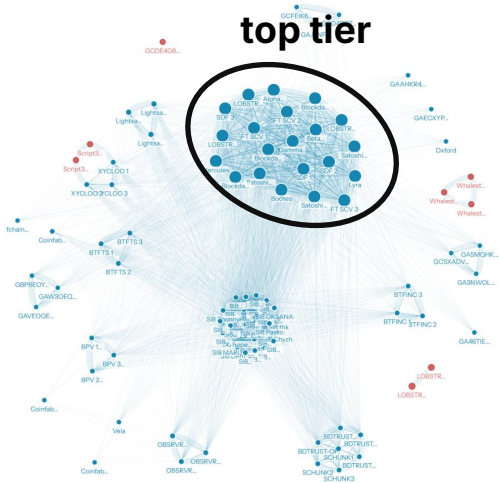




7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

Minimal splitting set:
3 validators
minimal quorum-blocking set:

# Current top-tier configuration

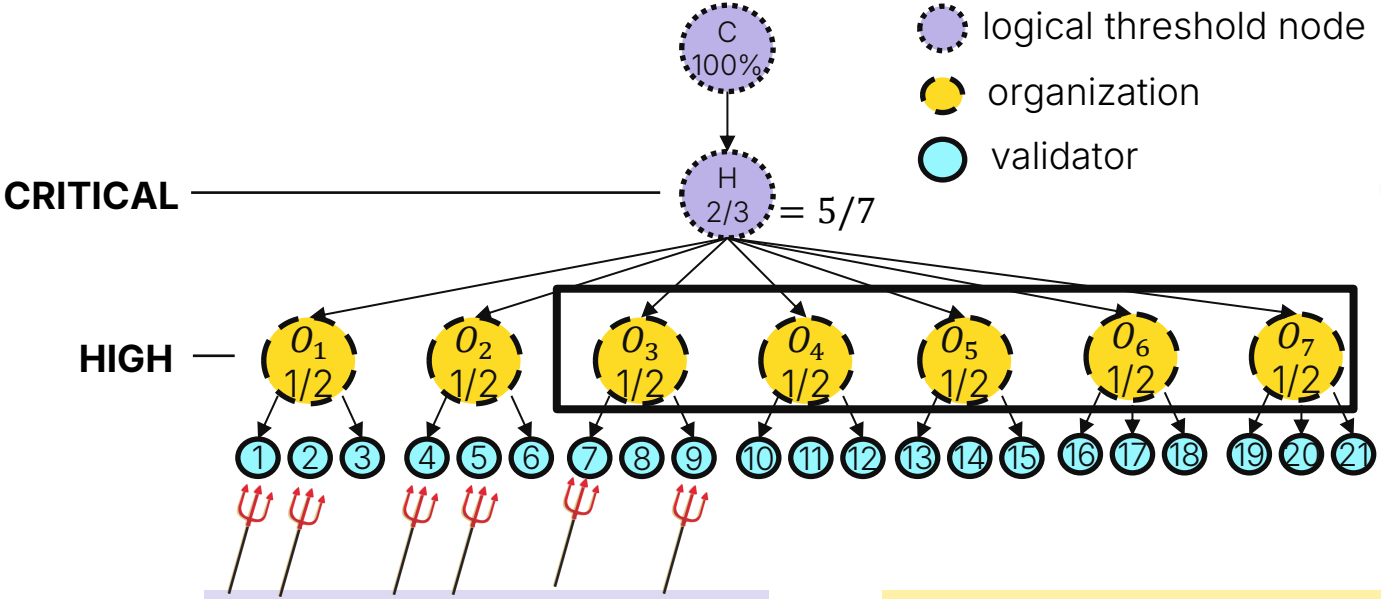


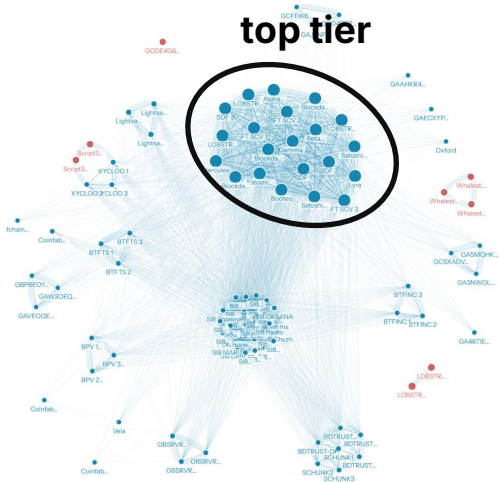




7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

Minimal splitting set:
3 validators
minimal quorum-blocking set:
6 validators

# Current top-tier configuration




7 top-tier organizations
each setup as level HIGH
All have the same slices
⇒ quorum = slice

Minimal splitting set:
3 validators
minimal quorum-blocking set:
6 validators

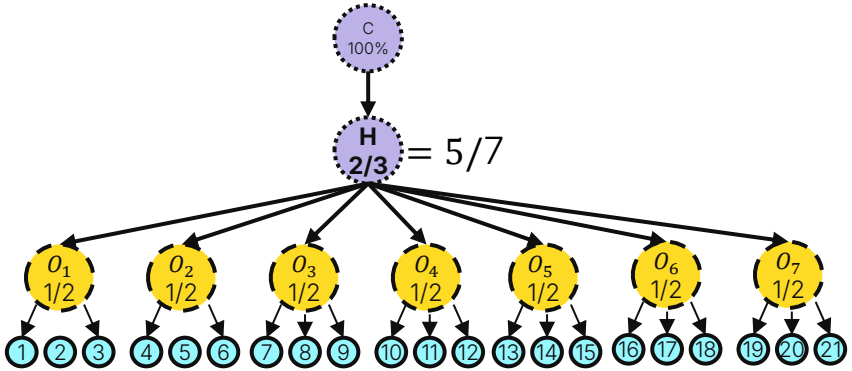
# Puzzle

logical threshold node

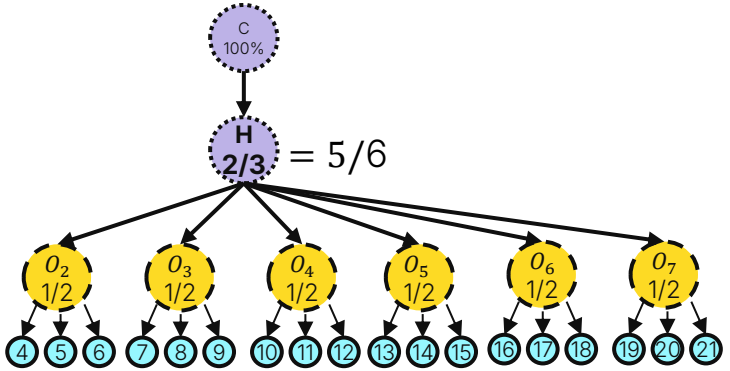
organization

validator

validators 1 to 20 use:




validators 21 uses:

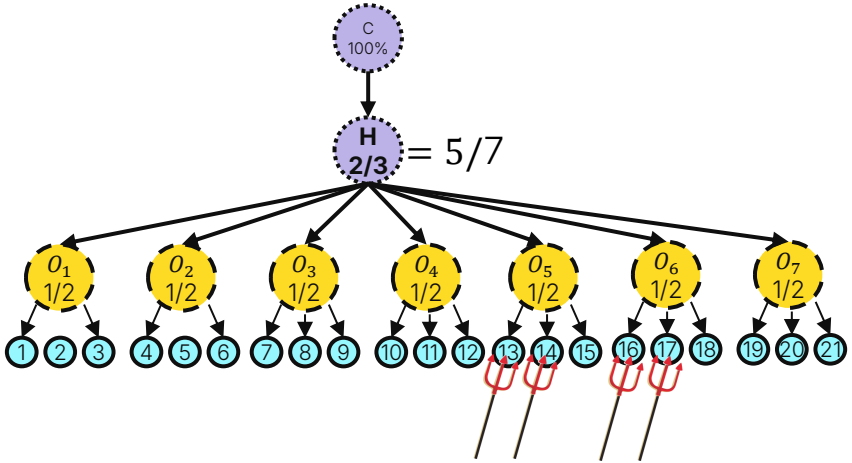



What is the size of a minimal blocking set?

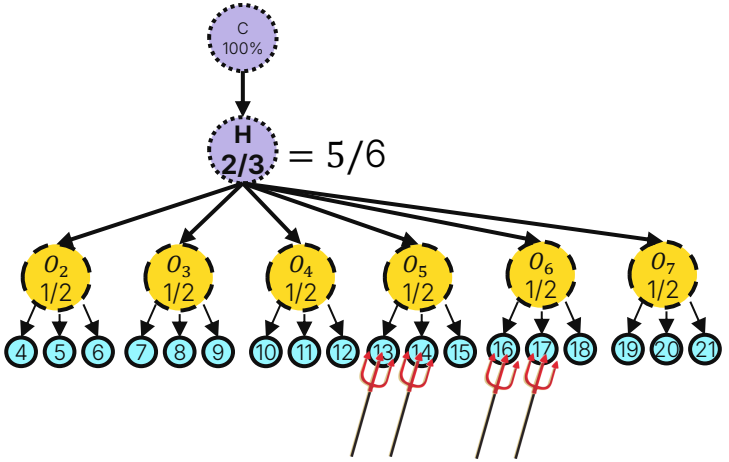
# Puzzle

logical threshold node
organization
validator

validators 1 to 20 use:




validators 21 uses:




What is the size of a minimal blocking set?

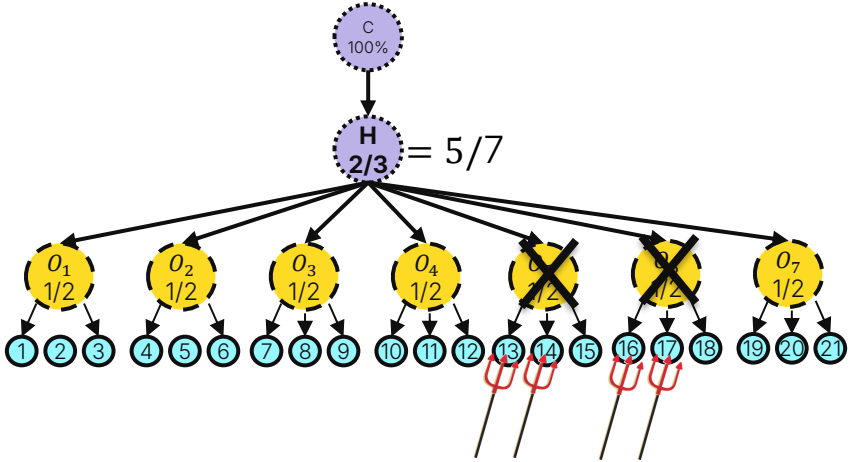
# Puzzle

logical threshold node

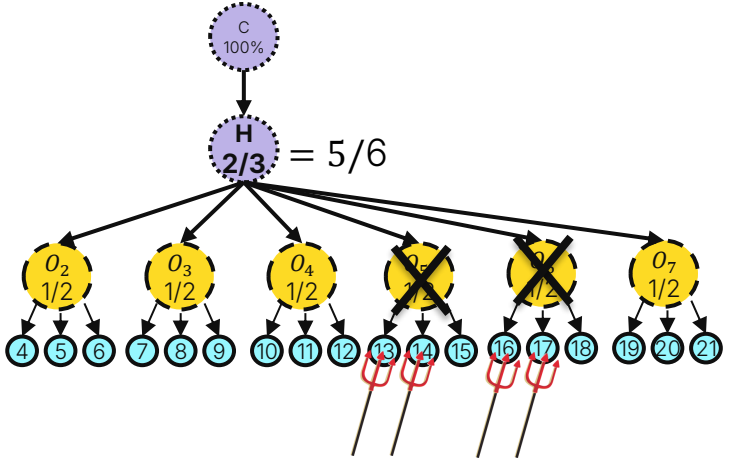
organization

validator

validators 1 to 20 use:




validators 21 uses:

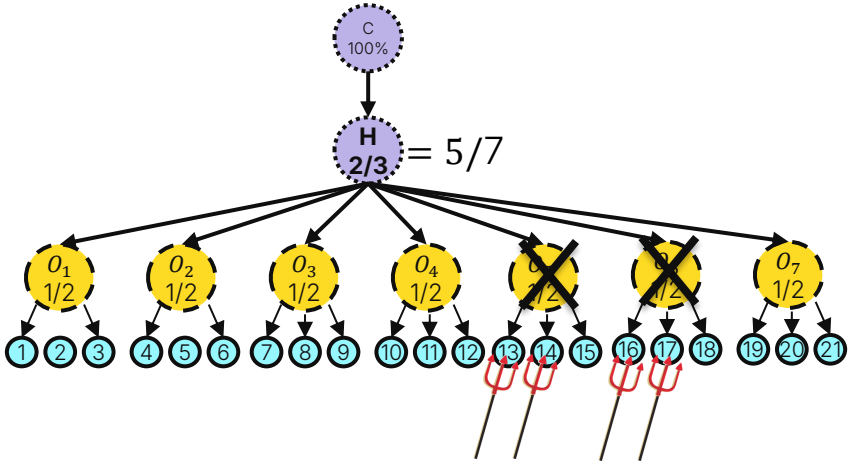



What is the size of a minimal blocking set?

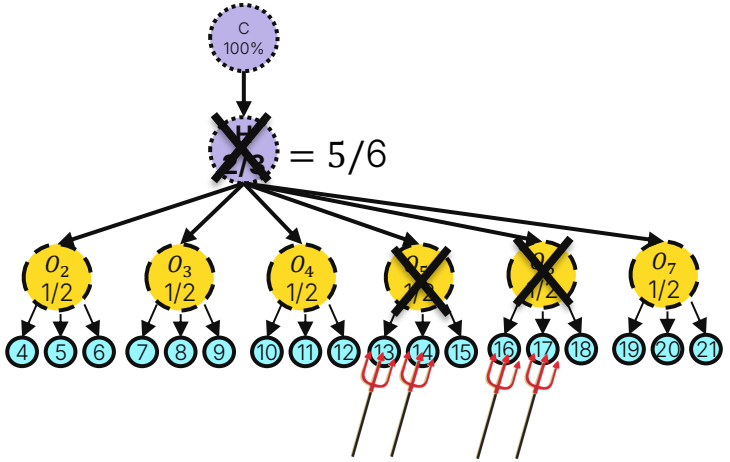
# Puzzle

logical threshold node
organization
validator

validators 1 to 20 use:




validators 21 uses:

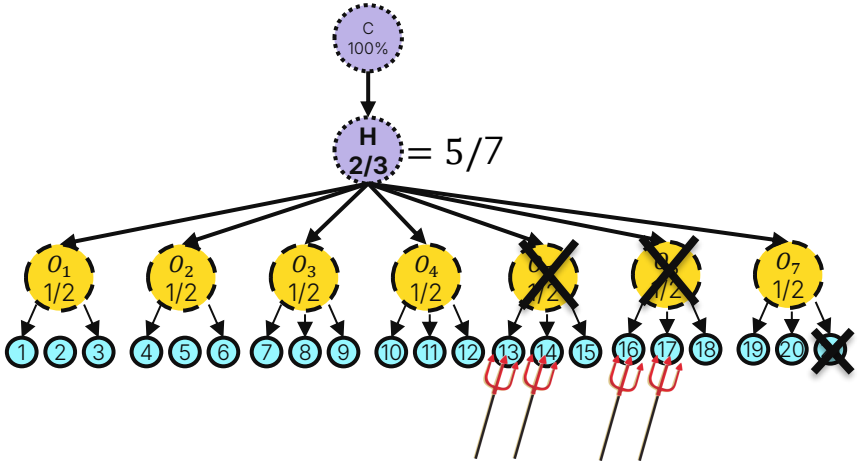



What is the size of a minimal blocking set?

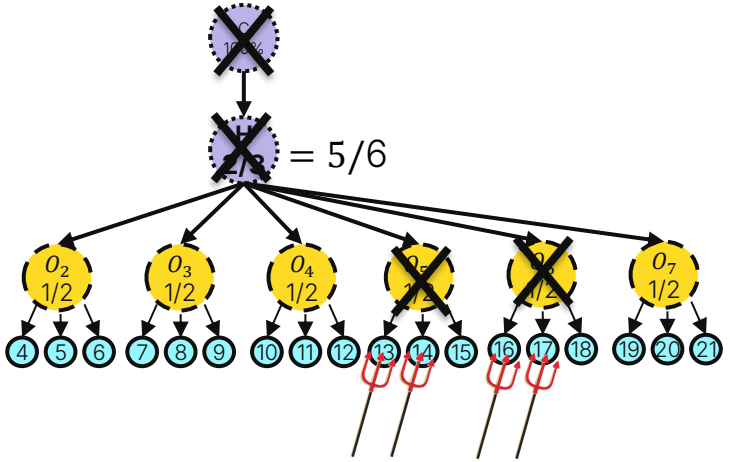
# Puzzle

logical threshold node
organization
validator

validators 1 to 20 use:




validators ~~21~~ uses:

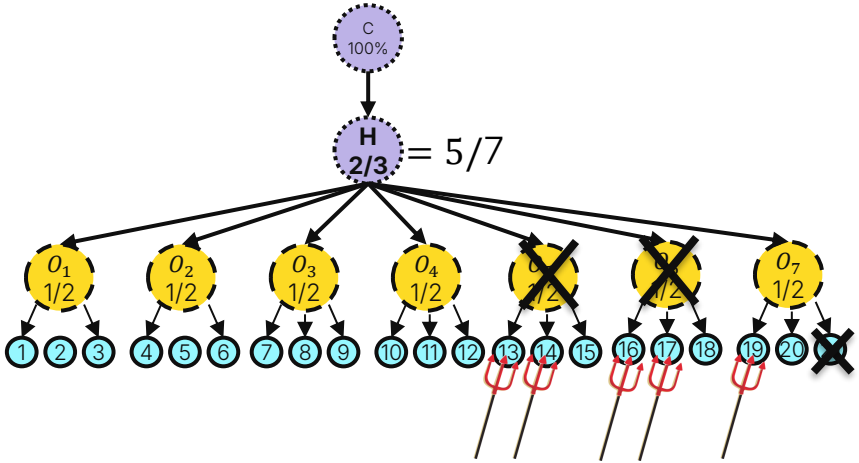



What is the size of a minimal blocking set?

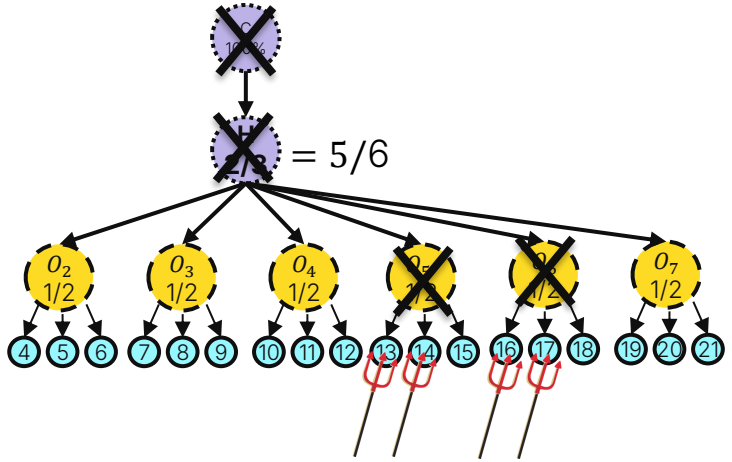
# Puzzle

logical threshold node
organization
validator

validators 1 to 20 use:

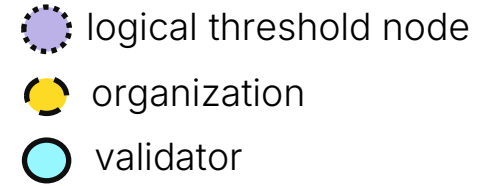



validators ~~21~~ uses:

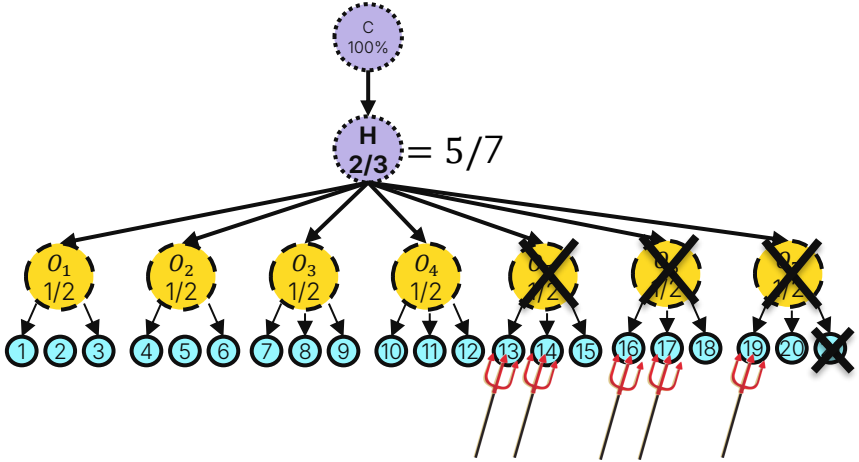



What is the size of a minimal blocking set?

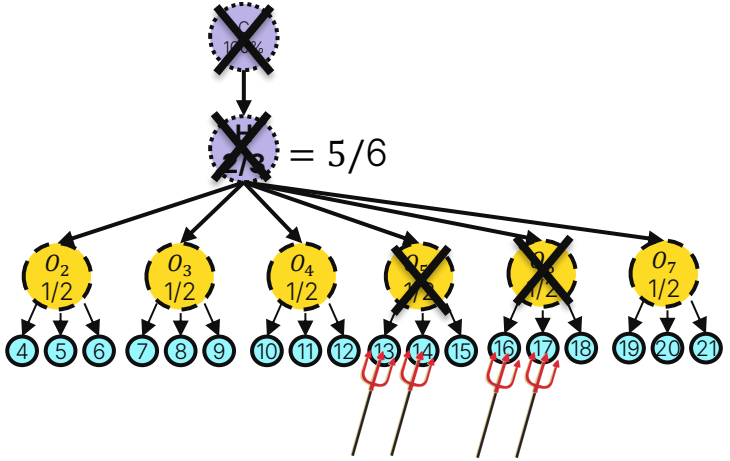
# Puzzle




validators 1 to 20 use:




validators 21 uses:

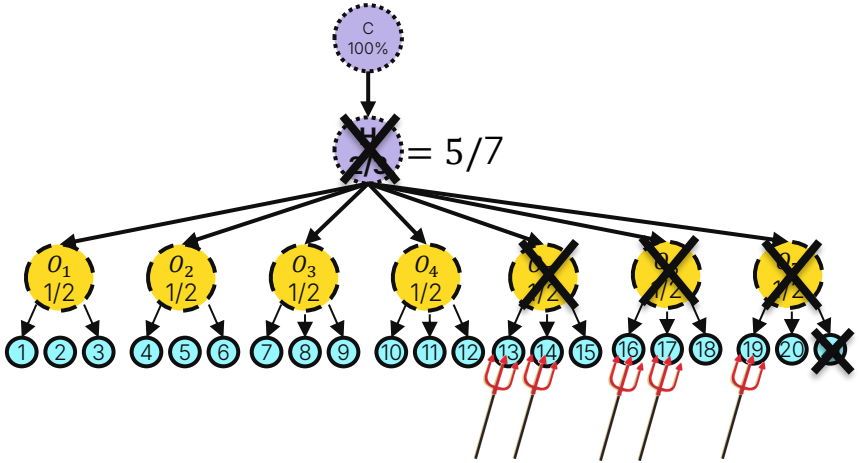



What is the size of a minimal blocking set?

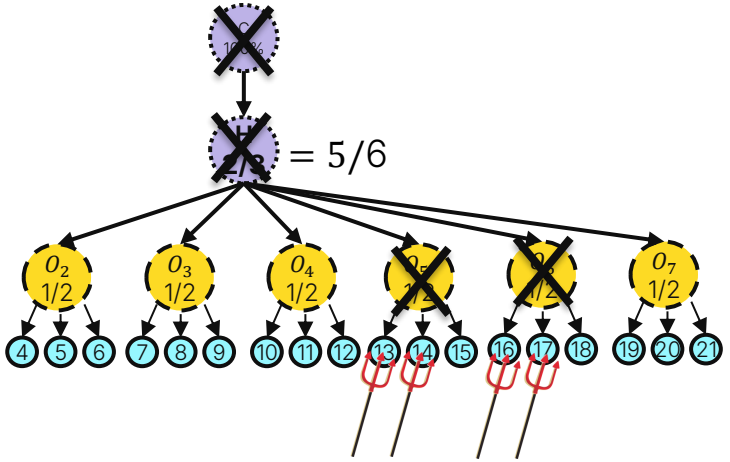
# Puzzle

logical threshold node
organization
validator

validators 1 to 20 use:




validators 21 uses:

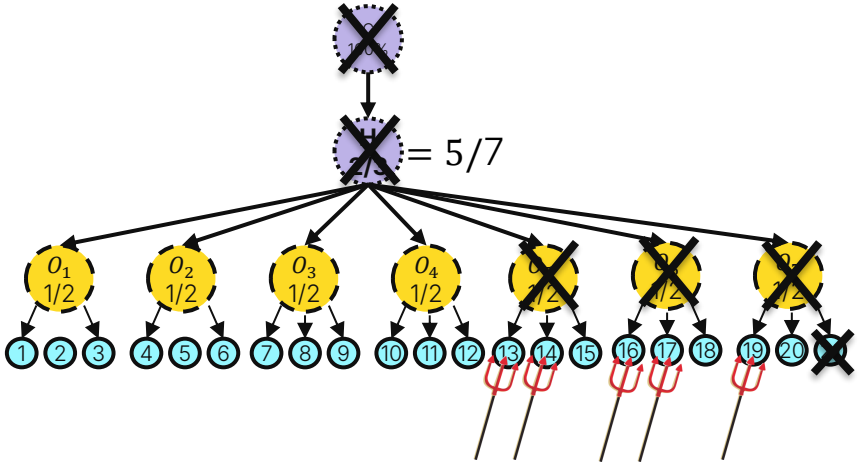



What is the size of a minimal blocking set?

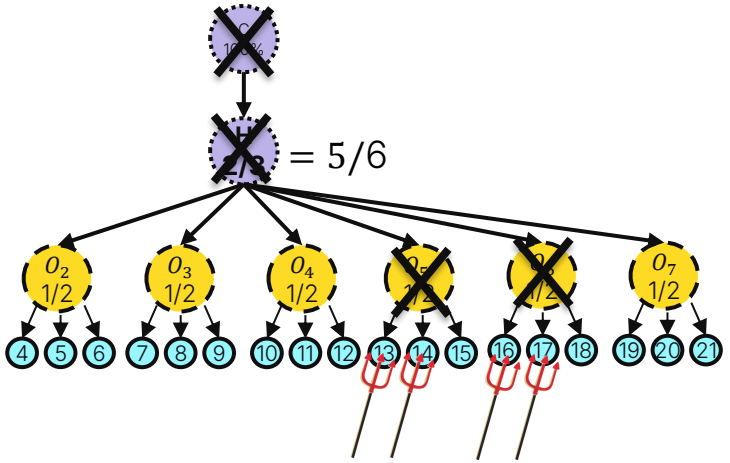
# Puzzle

logical threshold node
organization
validator

validators 1 to 20 use:




validators 21 uses:

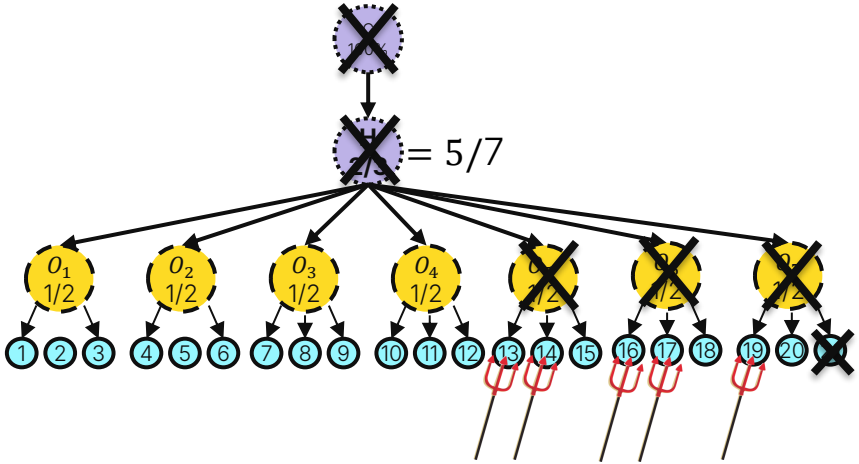



What is the size of a minimal blocking set?

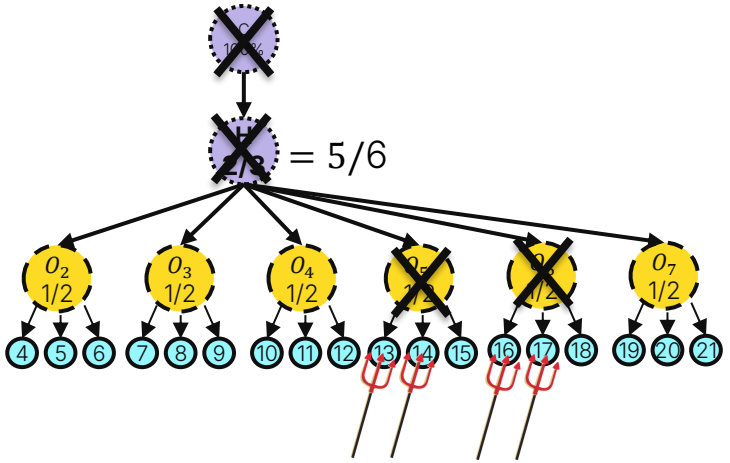
# Puzzle

logical threshold node
organization
validator

validators 1 to 20 use:




validators 21 uses:




What is the size of a minimal blocking set?

5