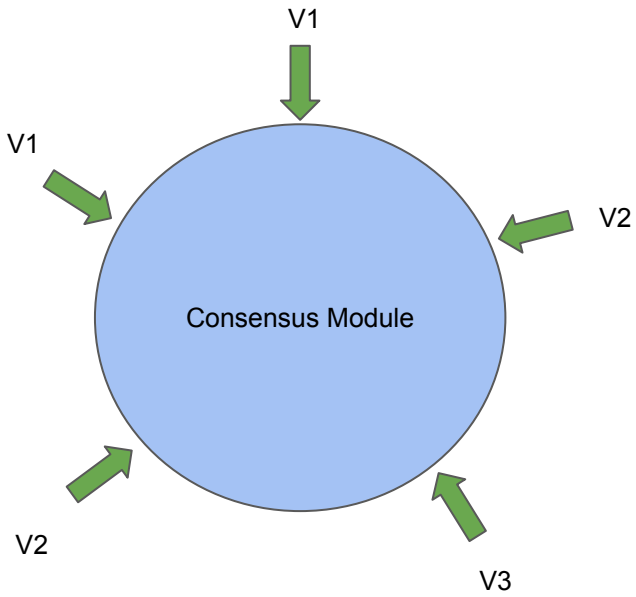
Paxos Made Simple

Leslie Lamport
ACM SIGACT News (Distributed Computing Column)

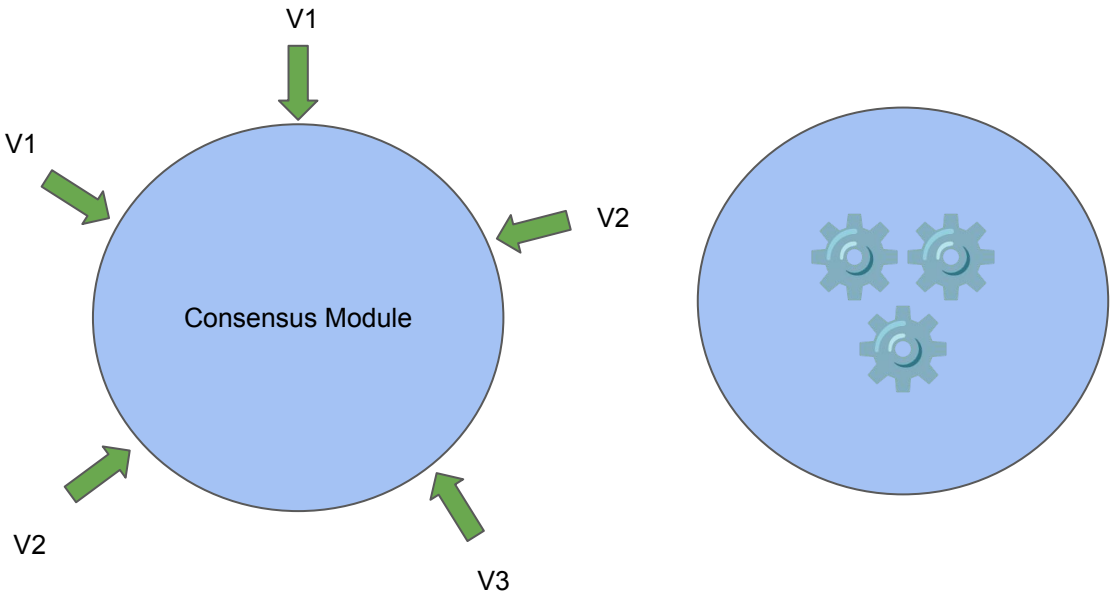
A Presentation for CS 6410
Suraaj Kanniwadi Sureshkannan

“The Paxos algorithm, when presented in plain English, is very simple”

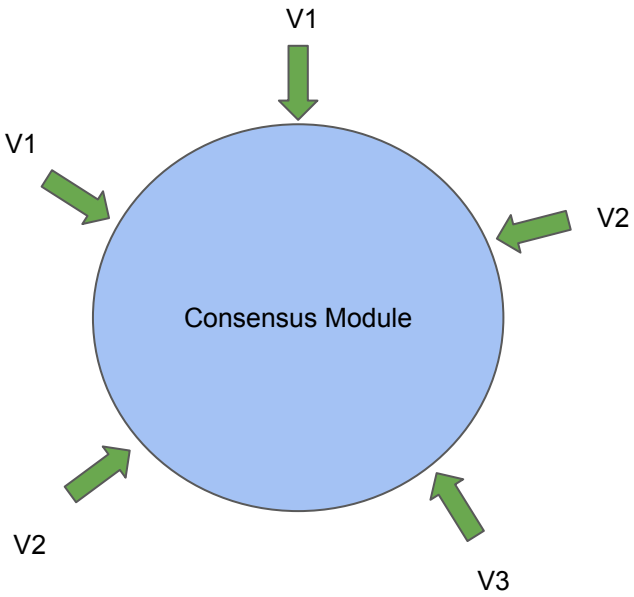
The Problem of Consensus

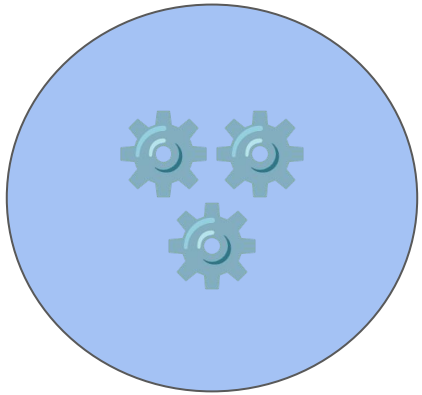


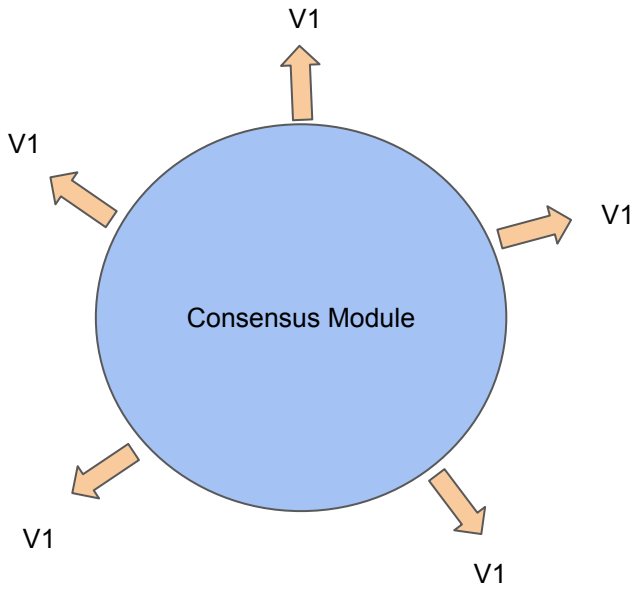
The Problem of Consensus



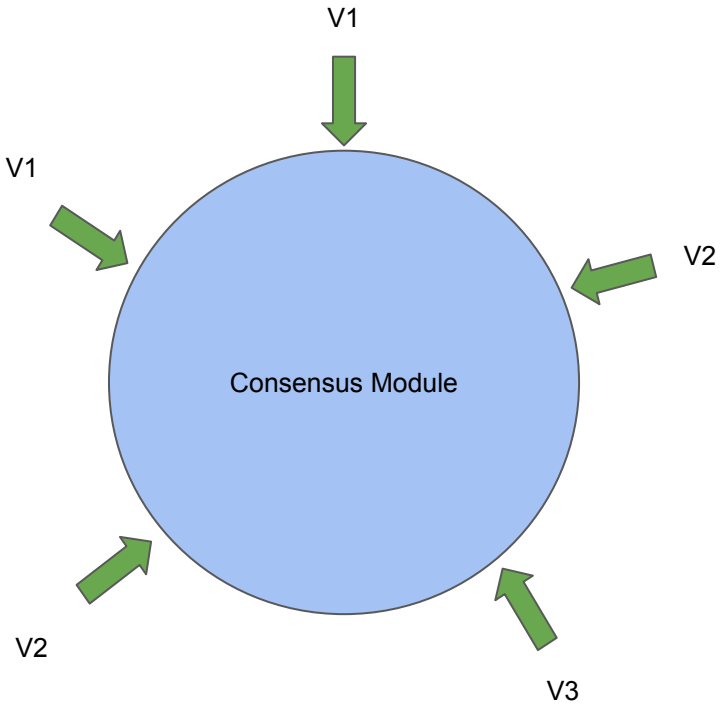
The Problem of Consensus



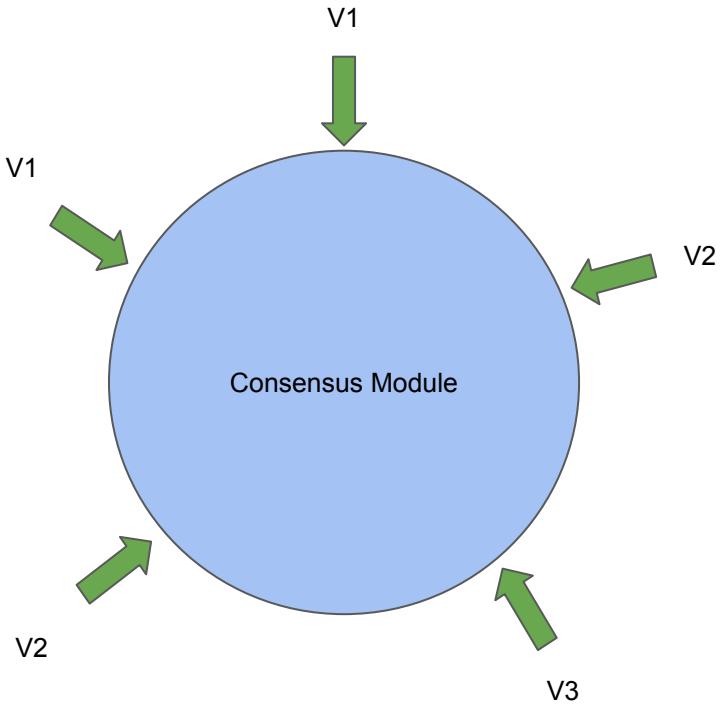


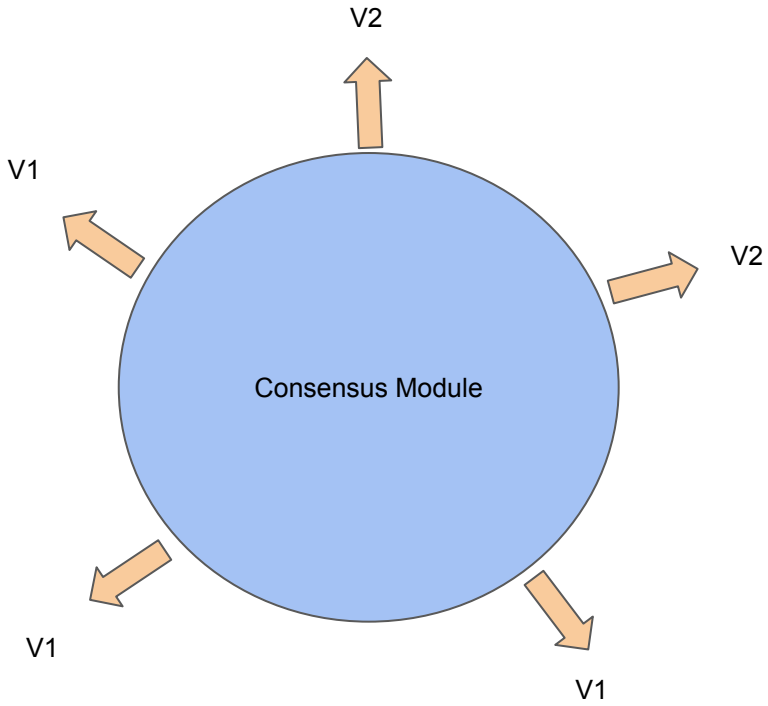


What we want in Consensus - 1

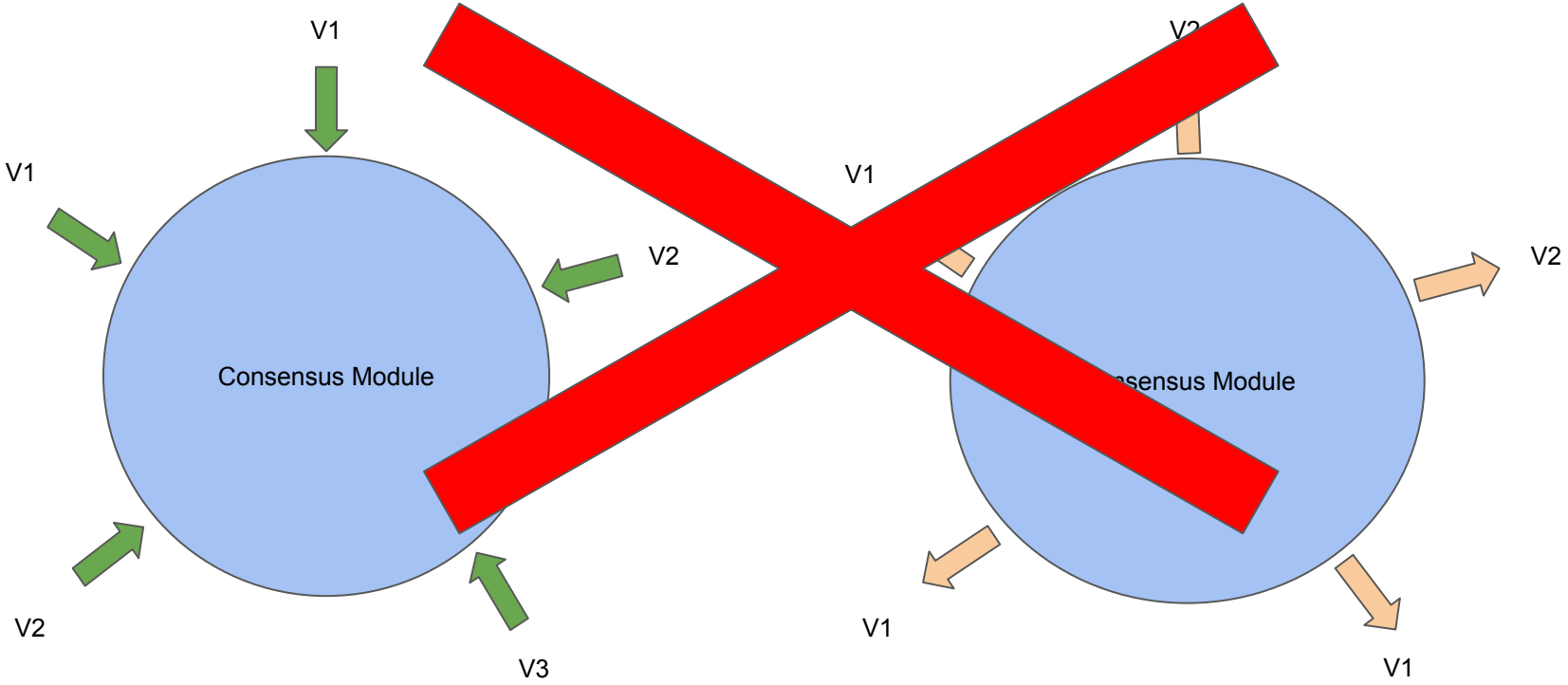


What we want in Consensus - 1

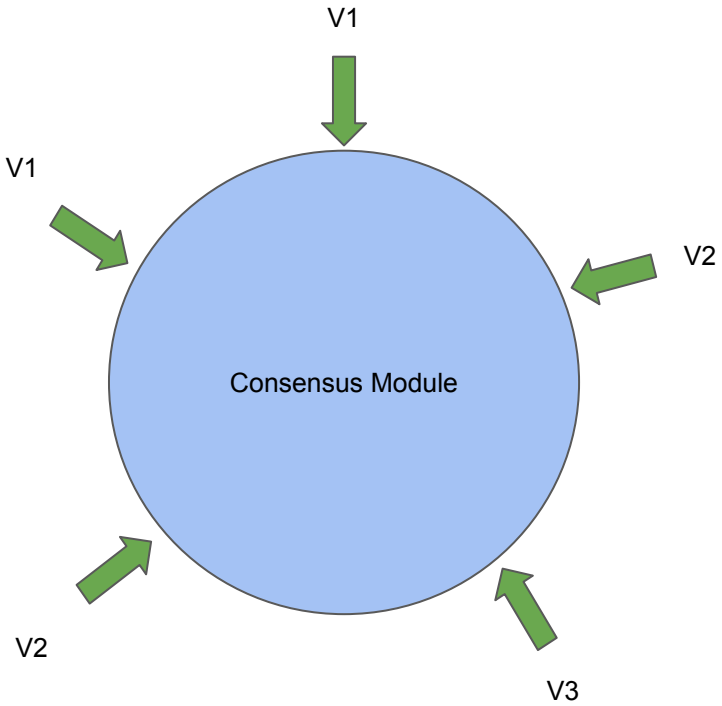


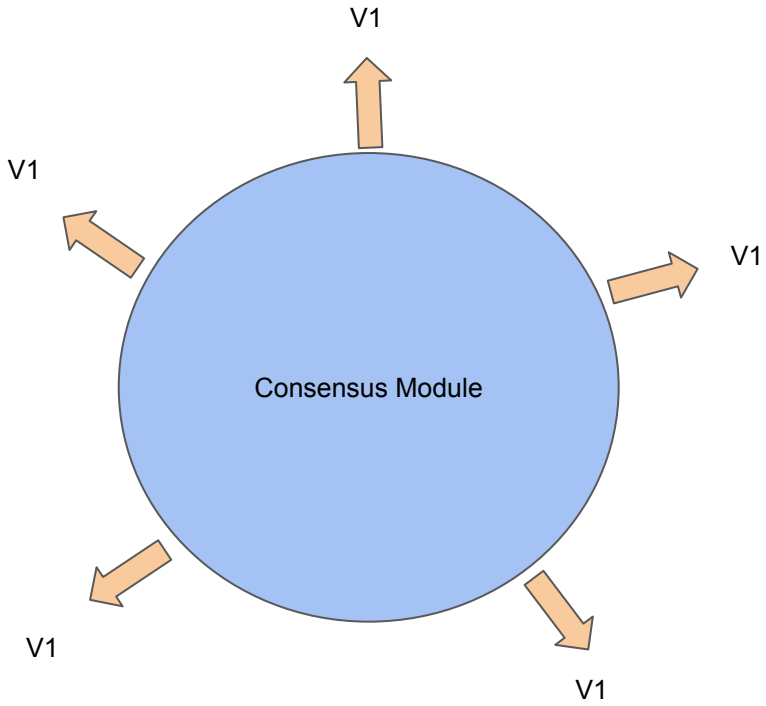


What we want in Consensus - 1

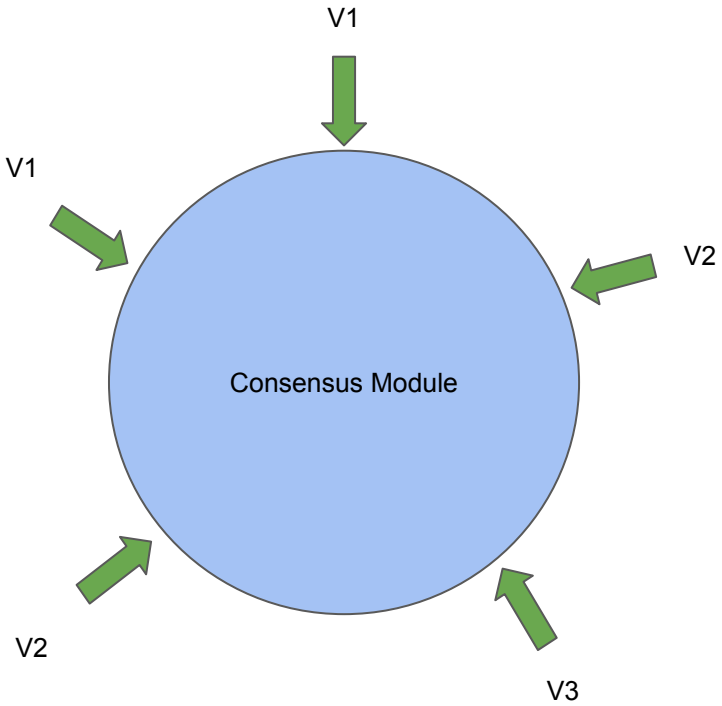


What we want in Consensus - 1 **(Agreement)**

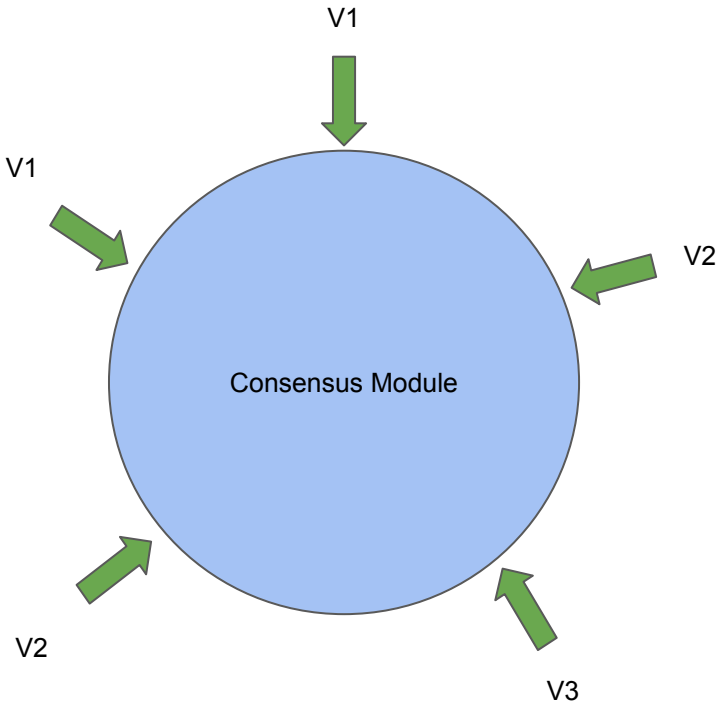


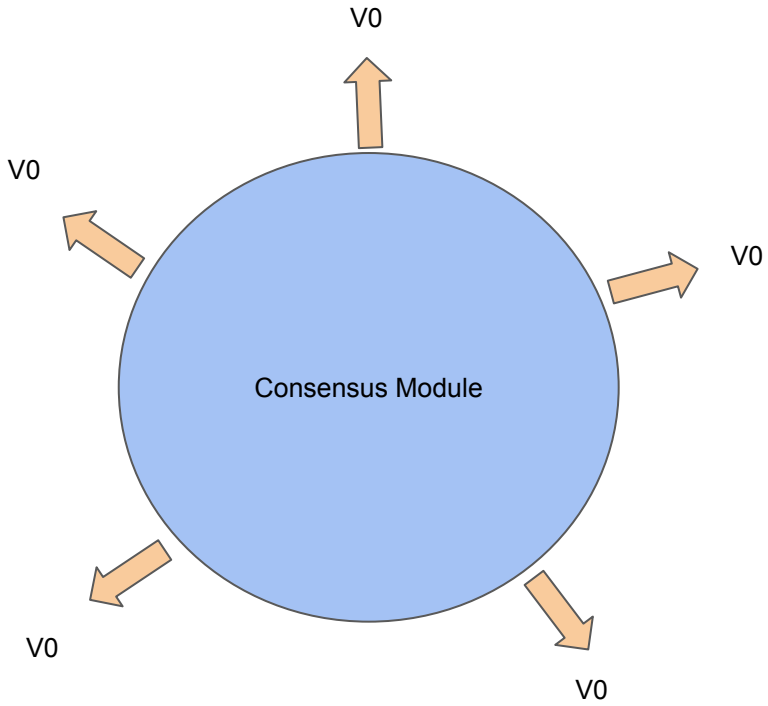


What we want in Consensus - 2

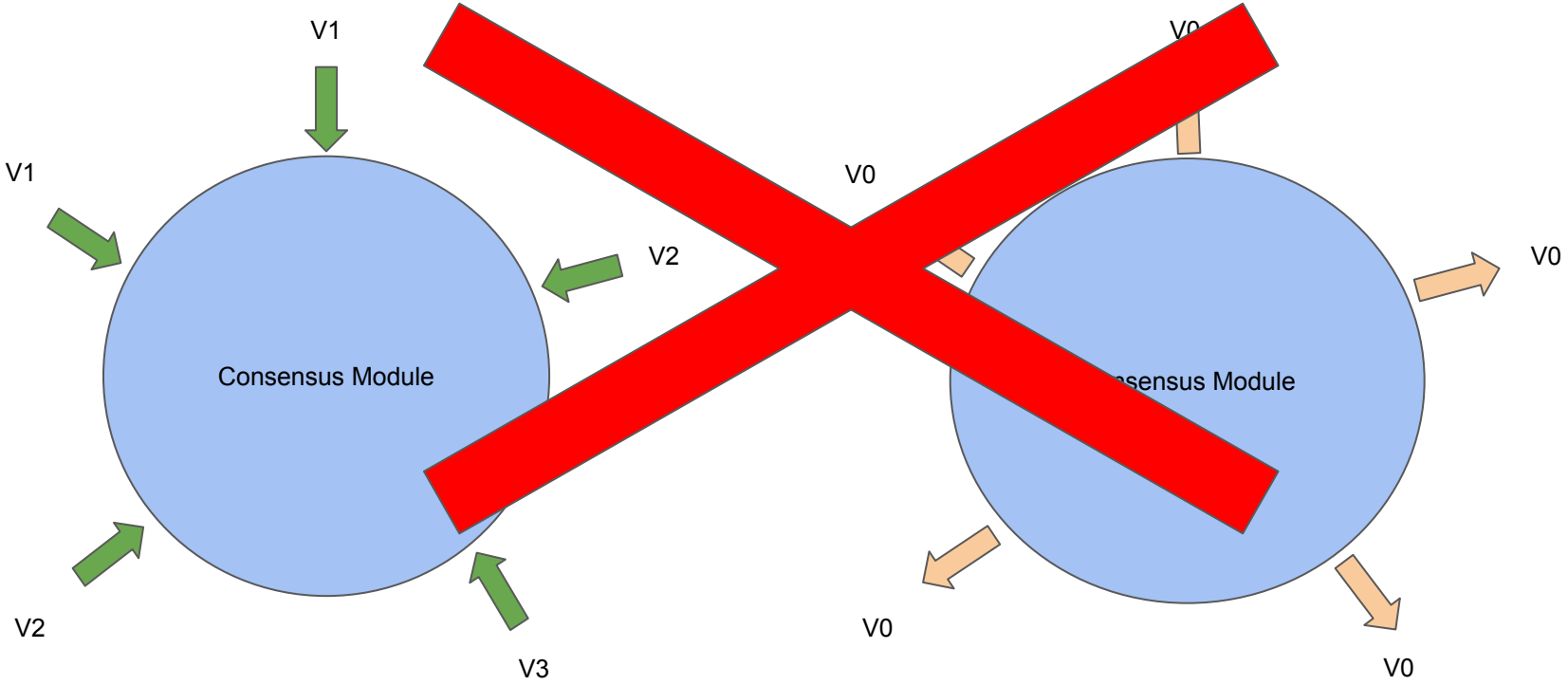


What we want in Consensus - 2

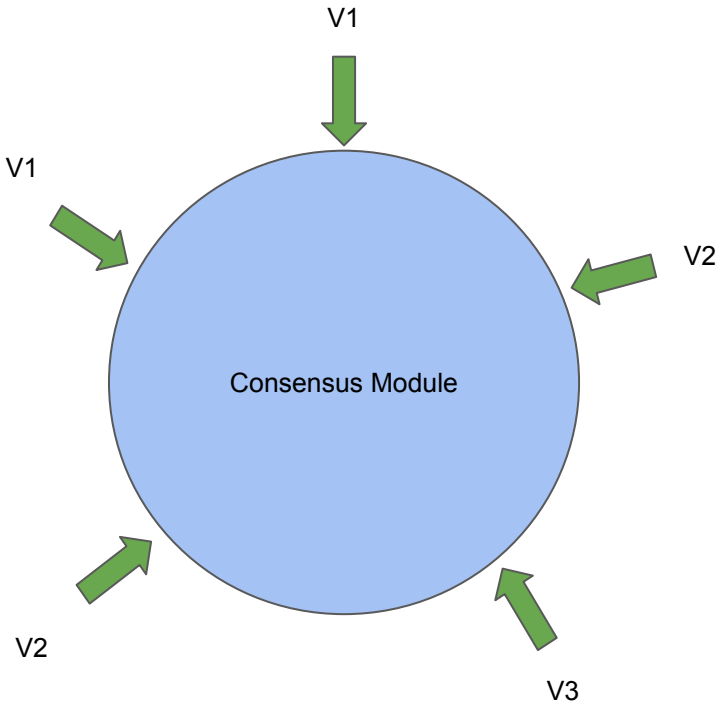


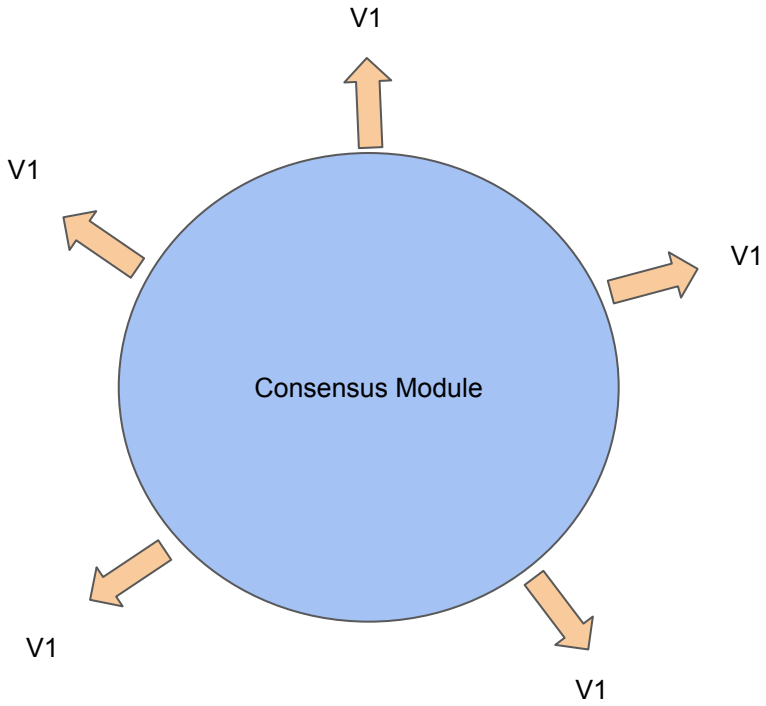


What we want in Consensus - 2

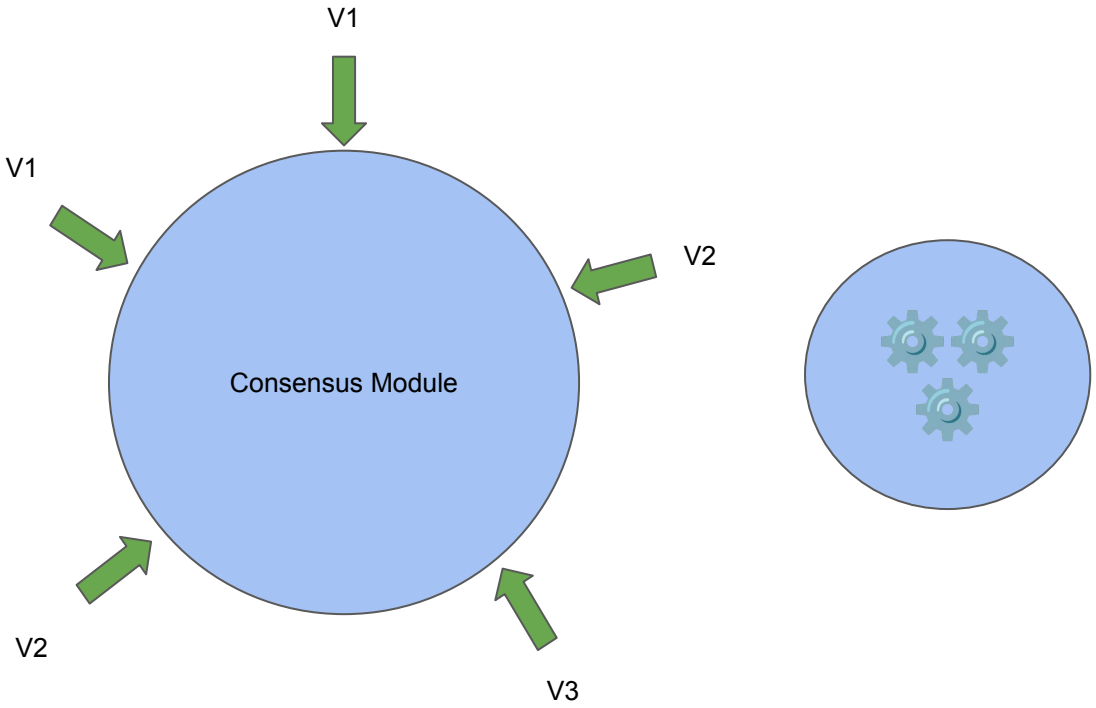


What we want in Consensus - 2 **(Validity)**

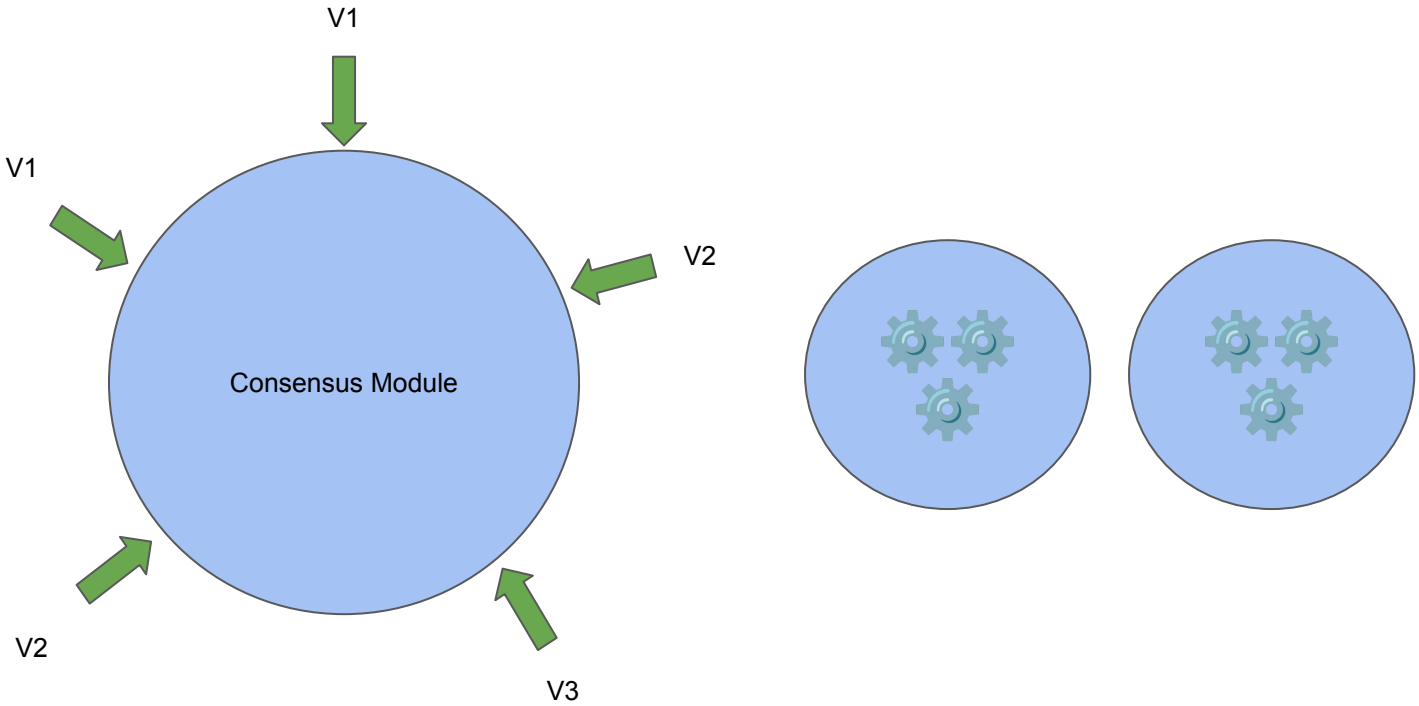




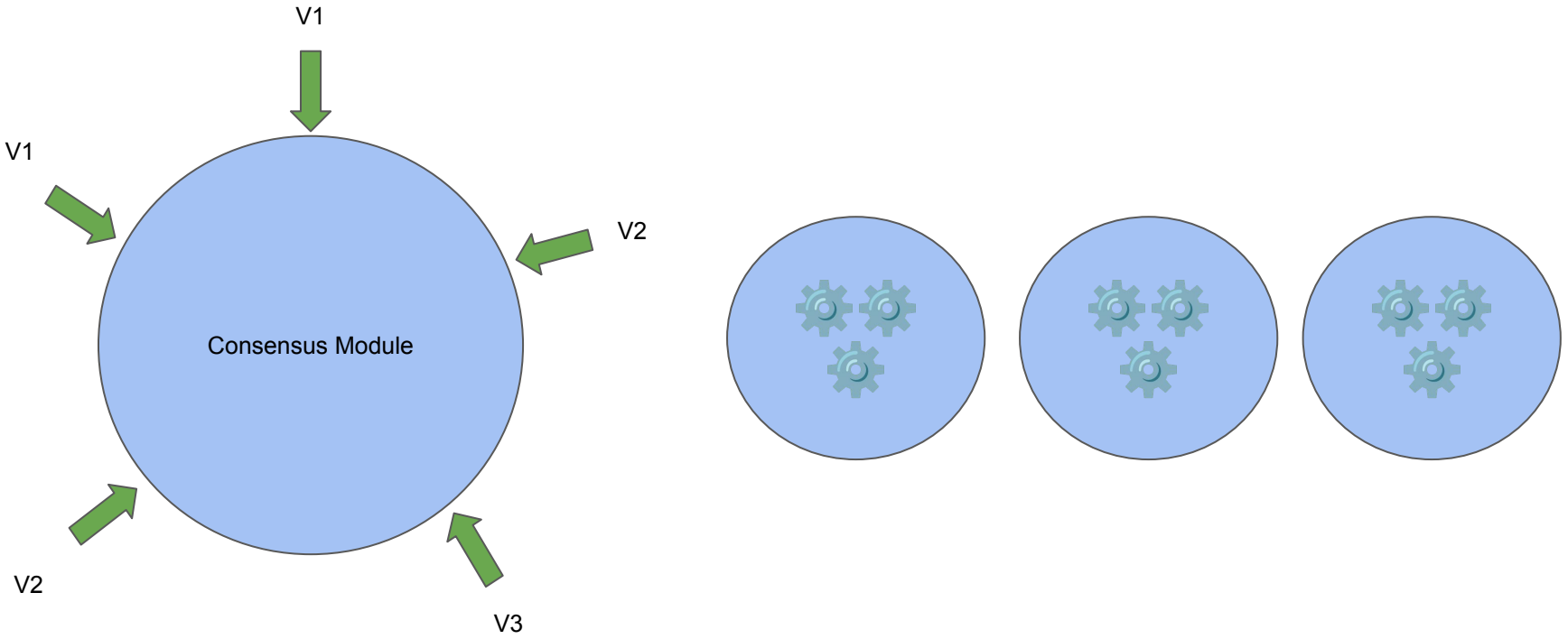
What we want in Consensus - 3



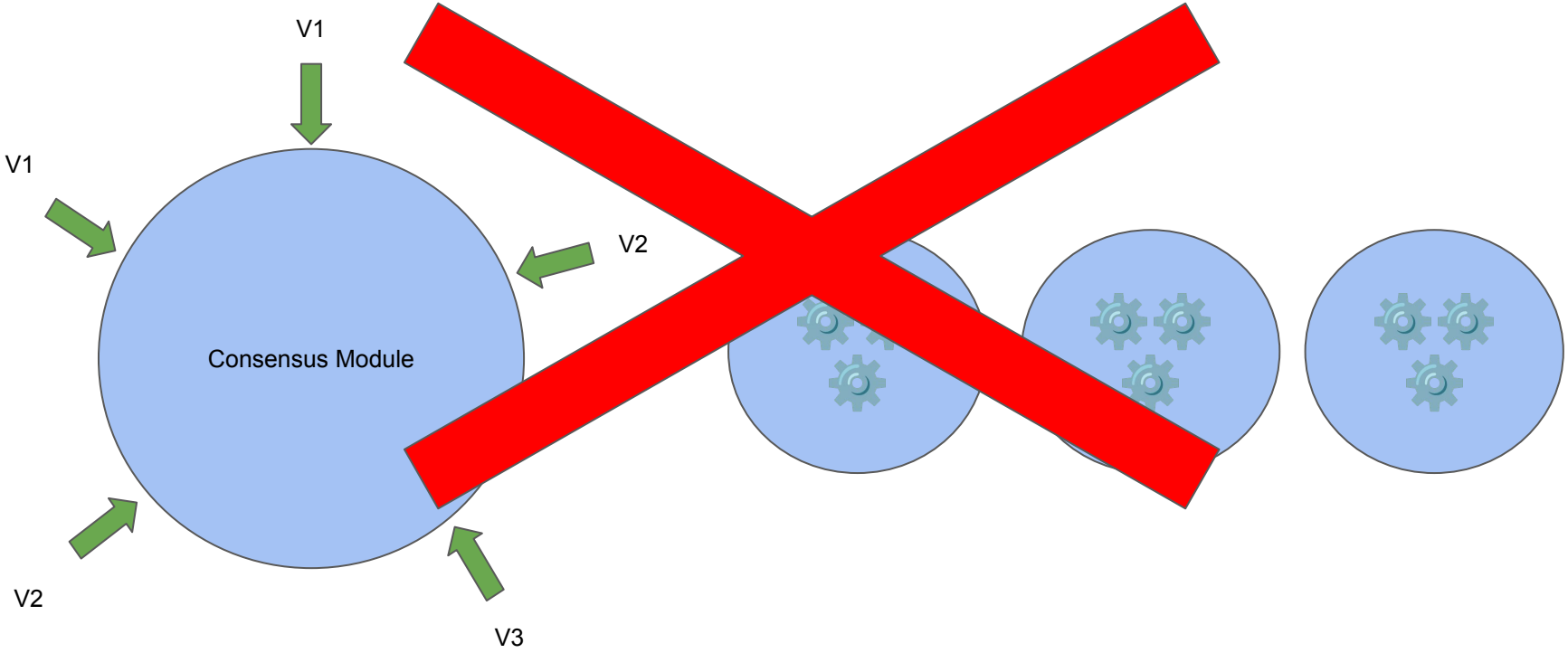
What we want in Consensus - 3



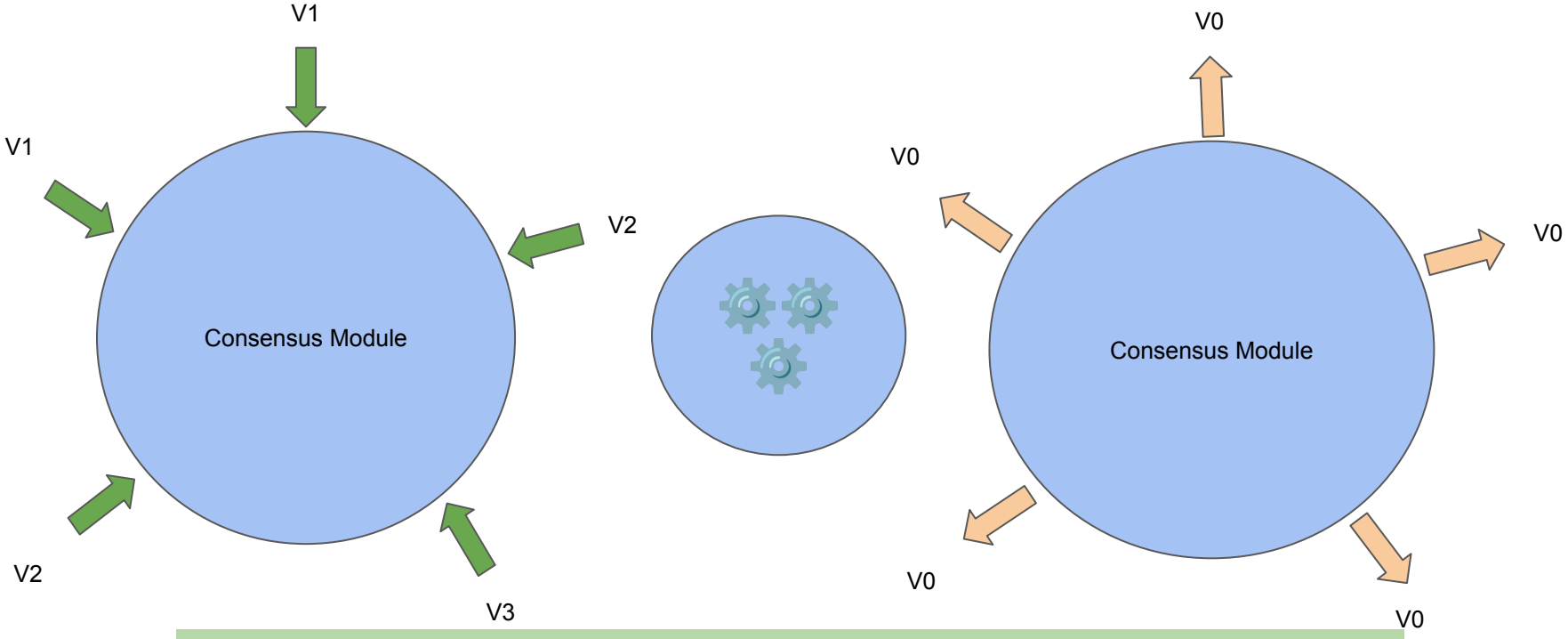
What we want in Consensus - 3



What we want in Consensus - 3

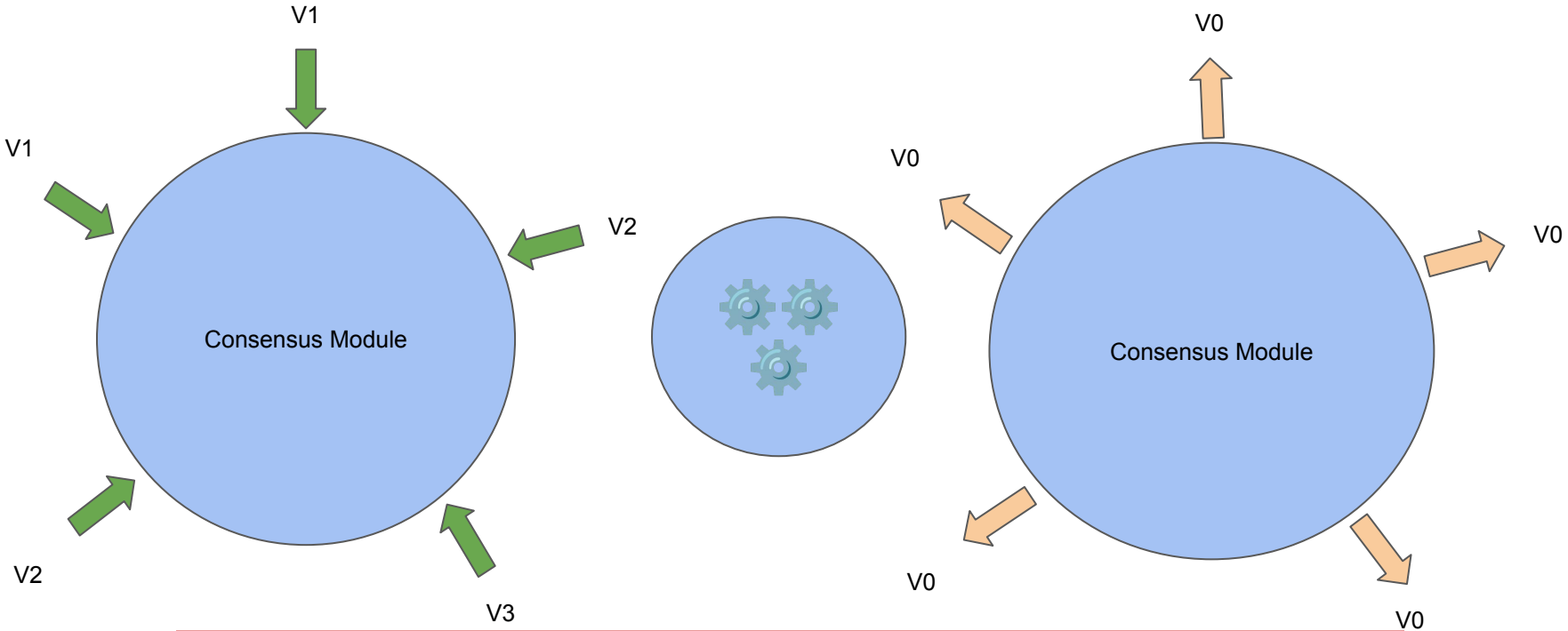


What we want in Consensus - 3 **(Termination)**



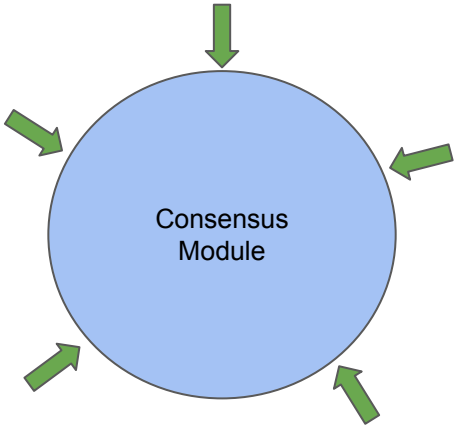
Note: All the outputs need not be ‘sent out’ at the same time

What we want in Consensus - 3 **(Termination)**

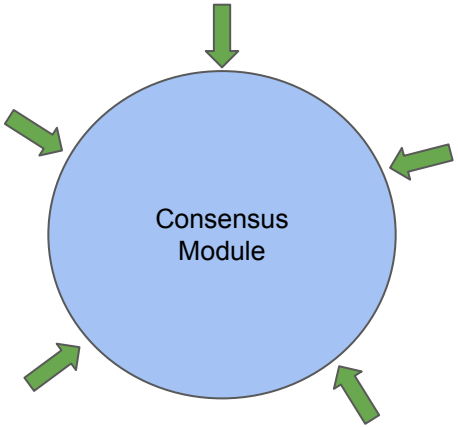


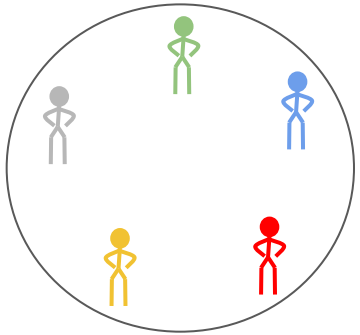
Not Possible in Asynchronous, Crash Failure Model of Distributed Systems

An Implementation of Consensus

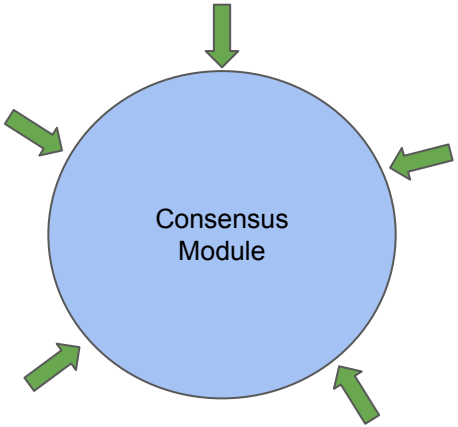


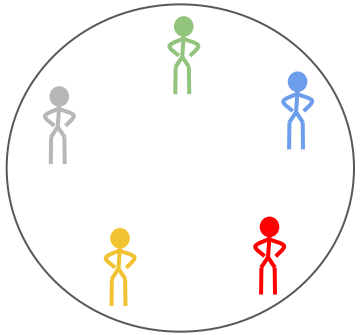
An Implementation of Consensus





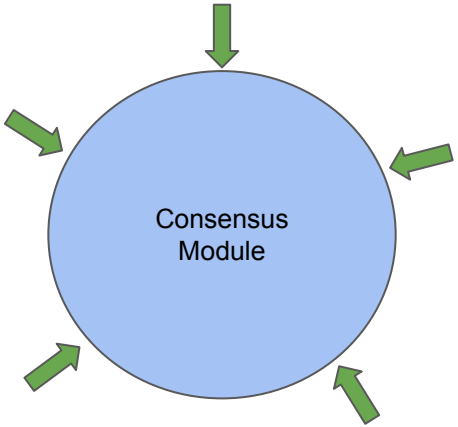
An Implementation of Consensus

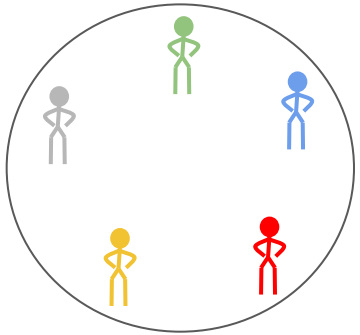






An Implementation of Consensus

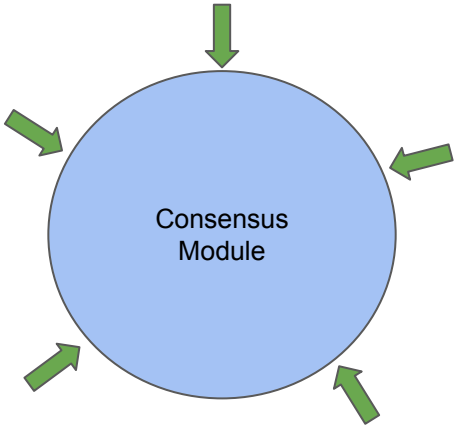


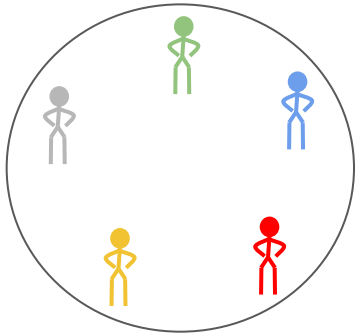






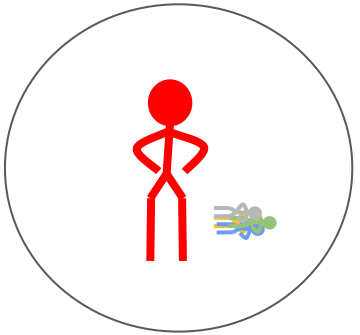
An Implementation of Consensus



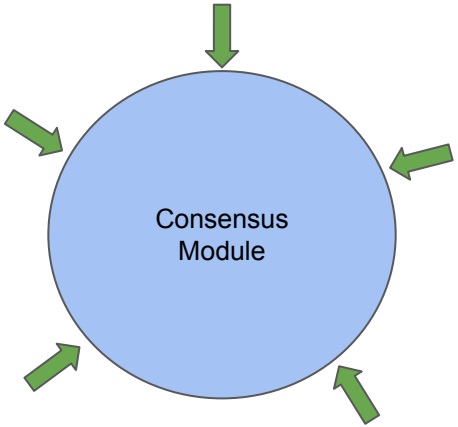


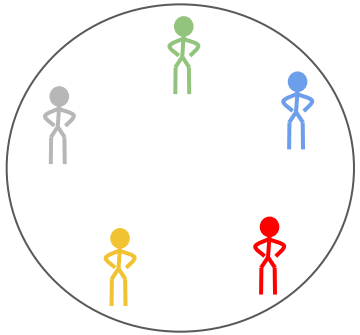






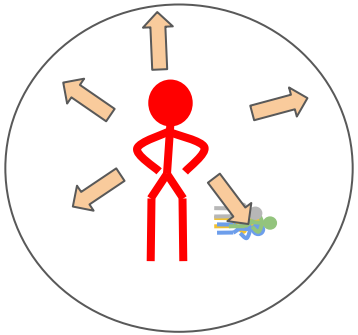
An Implementation of Consensus

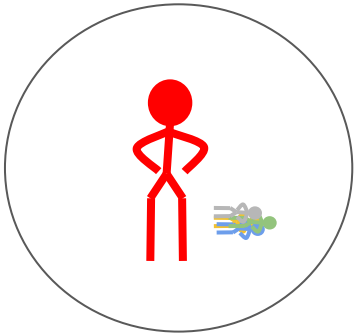




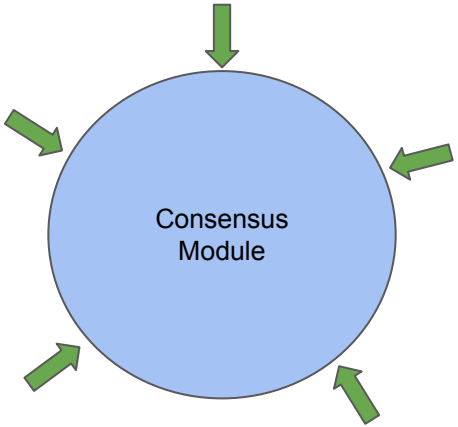


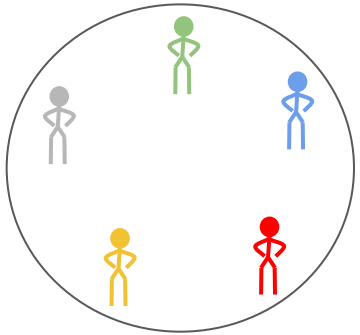






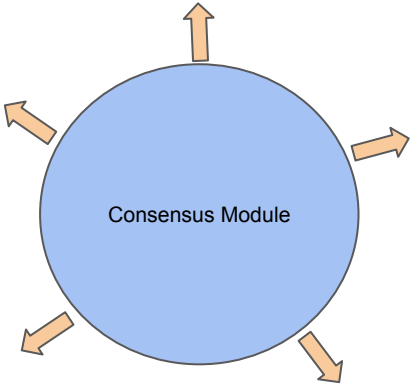
An Implementation of Consensus

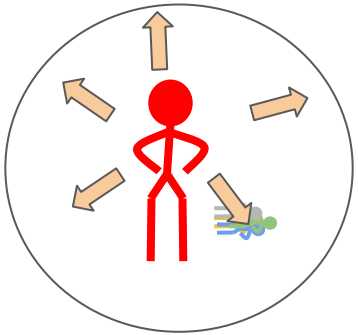


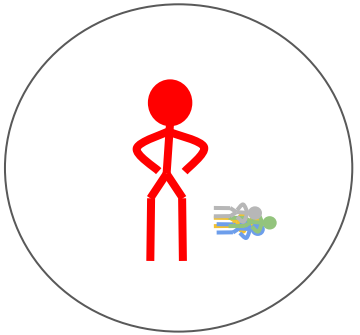








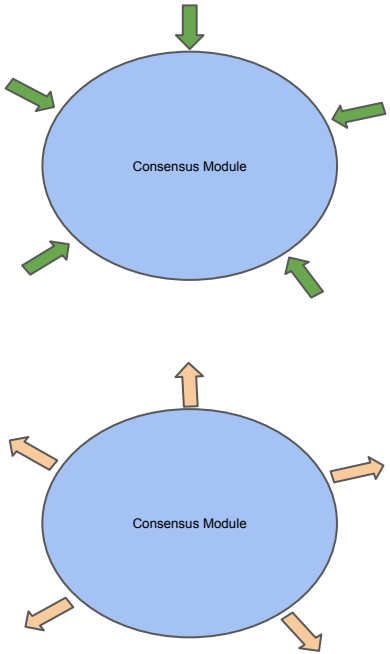




Consensus treatment in the paper

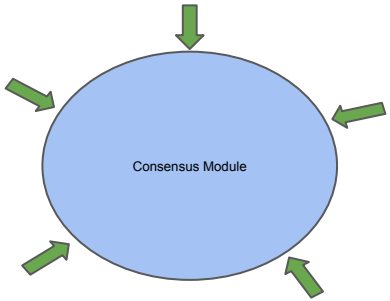
Consensus treatment in the paper

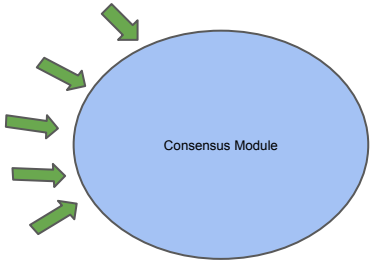
Separation into proposers and listeners **(More General)**

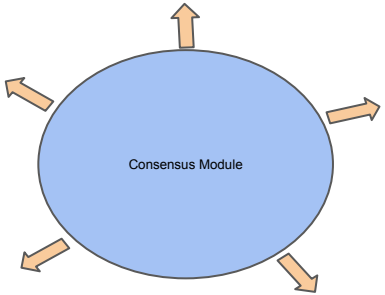


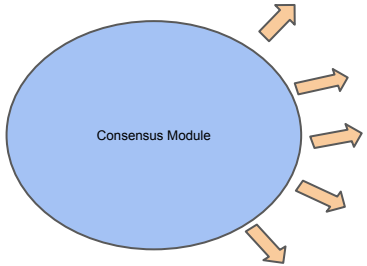
Consensus treatment in the paper

Separation into proposers and listeners **(More General)**









Consensus treatment in the paper

Consensus treatment in the paper

Propose - Learn → Propose - Choose - Learn

Consensus treatment in the paper

Propose - Learn → Propose - Choose - Learn

Requirements modified slightly. Most notably:

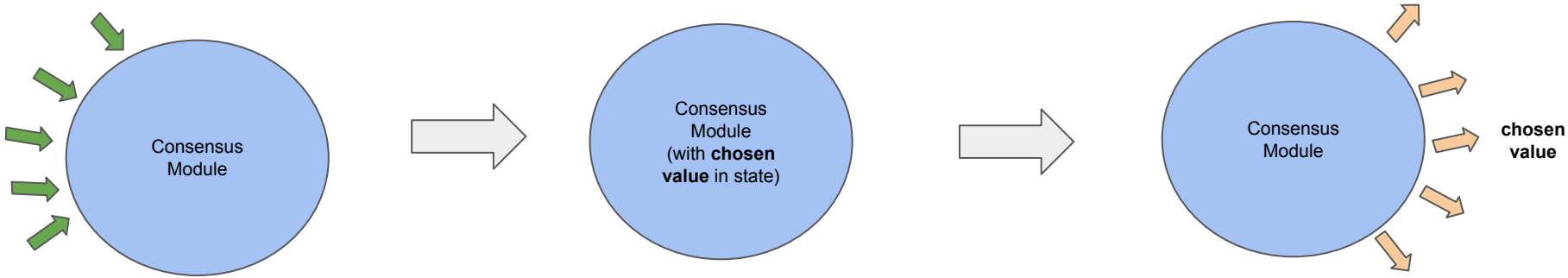
- **Only a single, valid value is chosen (maybe multiple times)**, which will then be the output
- Note than ‘the chosen value’ can thought of as a state function

Consensus treatment in the paper

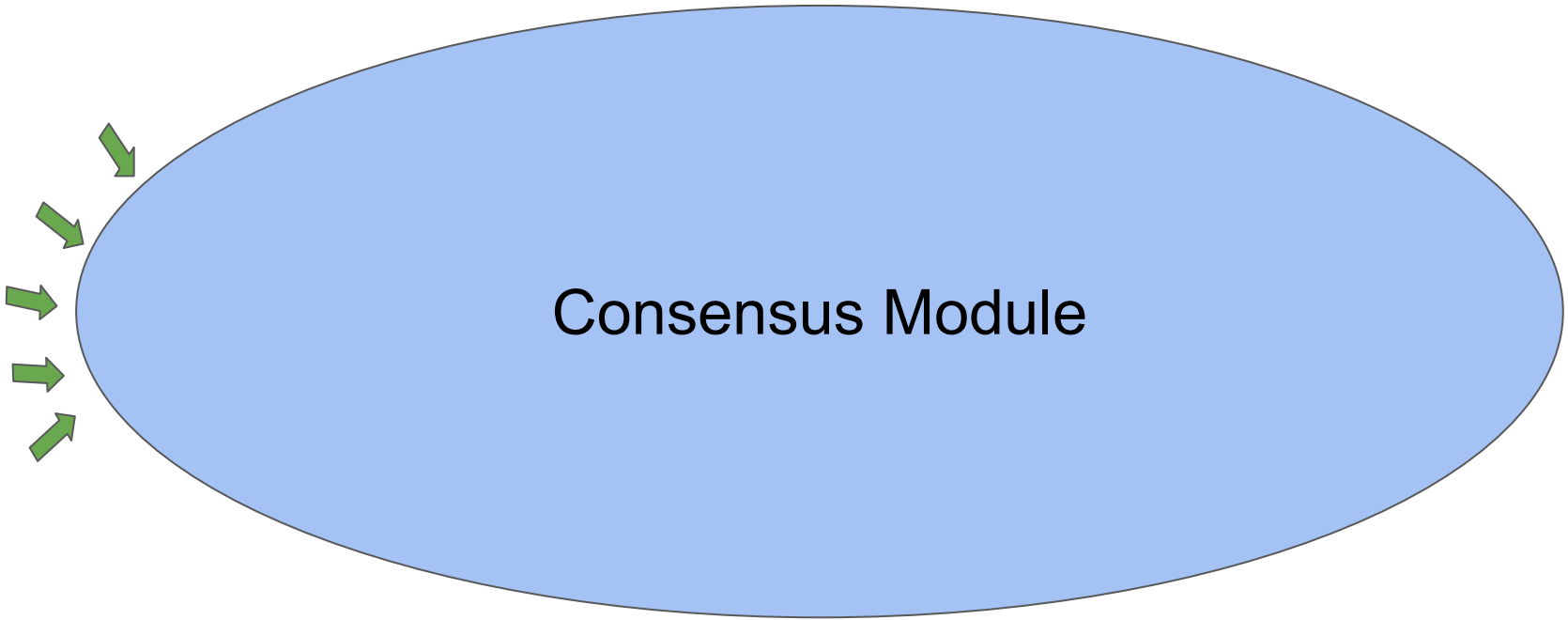
Propose - Learn → Propose - Choose - Learn

Requirements modified slightly. Most notably:

- **Only a single, valid value is chosen (maybe multiple times)**, which will then be the output
- Note than ‘the chosen value’ can thought of as a state function



Let’s design our system!



Let’s design our system!

Let’s design our system!

- Note that we take a bunch of values as input. Thus, let’s have a **set of values**, initialized from the external input.

Let’s design our system!

- Note that we take a bunch of values as input. Thus, let’s have a **set of values**, initialized from the external input.
- Recall, we have to do **Propose - Choose - Learn**

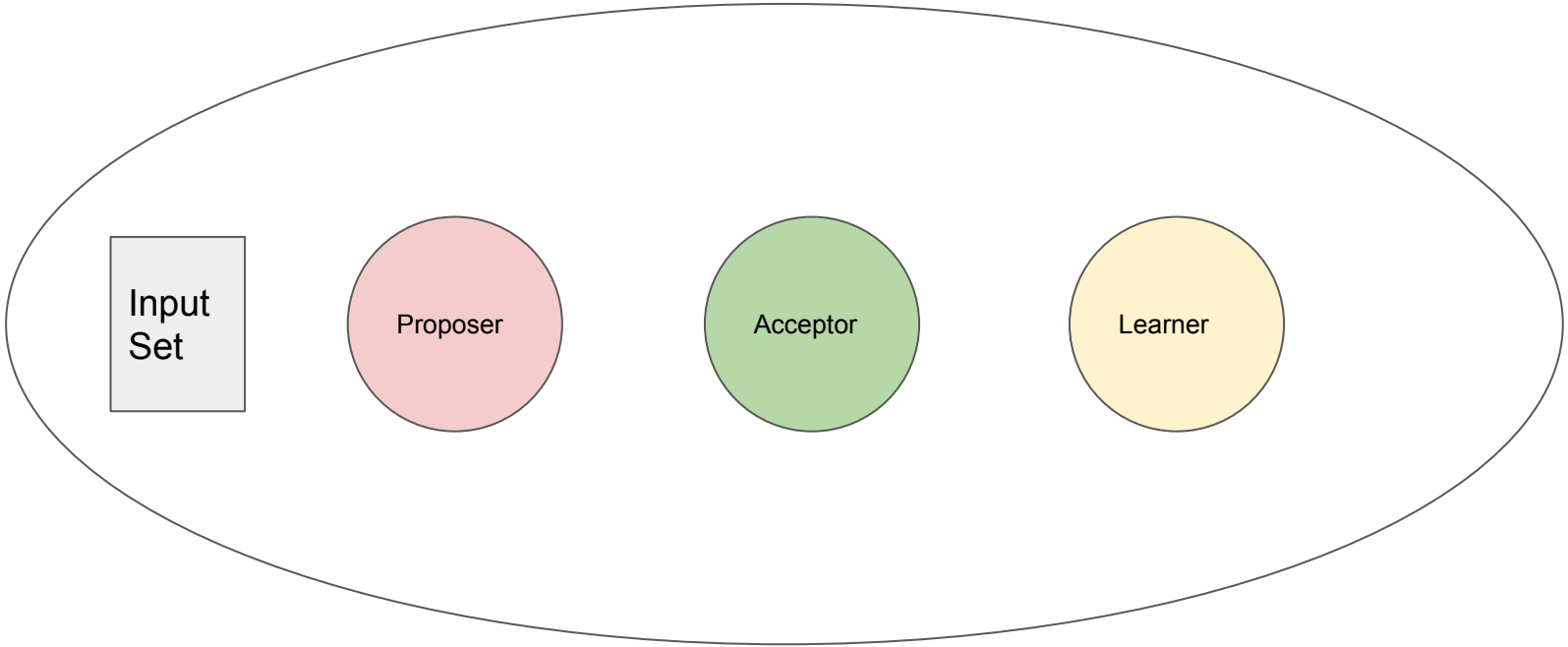
Let’s design our system!

- Note that we take a bunch of values as input. Thus, let’s have a **set of values**, initialized from the external input.
- Recall, we have to do **Propose - Choose - Learn**
- Let’s have entities in our system that do all of those!
  - **Proposer** - “Picks up” values of our **set of input values** and “proposes them”
  - **Acceptor** - Listens to proposals and “chooses” **THE ONE**
  - **Learner** - Learns **THE ONE** (somehow) and communicates it with everyone who needs the output

Let’s design our system!

- Note that we take a bunch of values as input. Thus, let’s have a **set of values**, initialized from the external input.
- Recall, we have to do **Propose - Choose - Learn**
- Let’s have entities in our system that do all of those!
  - **Proposer** - “Picks up” values of our **set of input values** and “proposes them”
  - **Acceptor** - Listens to proposals and “chooses” **THE ONE**
  - **Learner** - Learns **THE ONE** (somehow) and communicates it with everyone who needs the output
- Super simple implementation :)
- Note: This is an asynchronous, message-passing system

Let’s design our system!



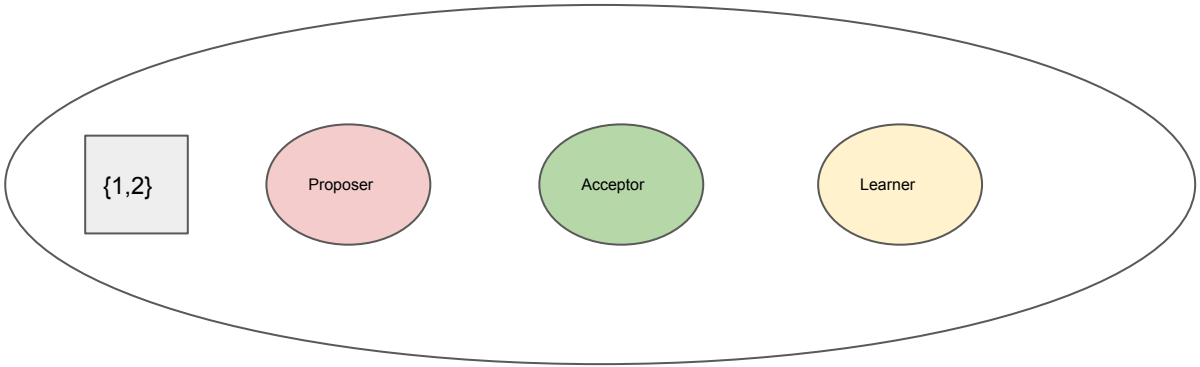
Implementing this

This model is simply an abstraction. In the real world, this may be implemented as follows:

- Input set: A bunch of client computers sending requests
- Proposer, Acceptor, Learner: Processes running on any machine in the world

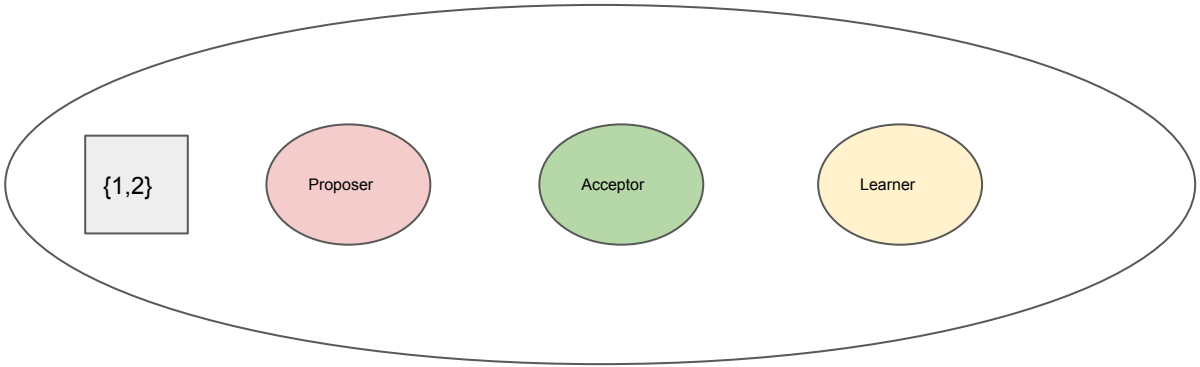
Note: These can be running on the same machine as well

Let’s design our system! An illustration



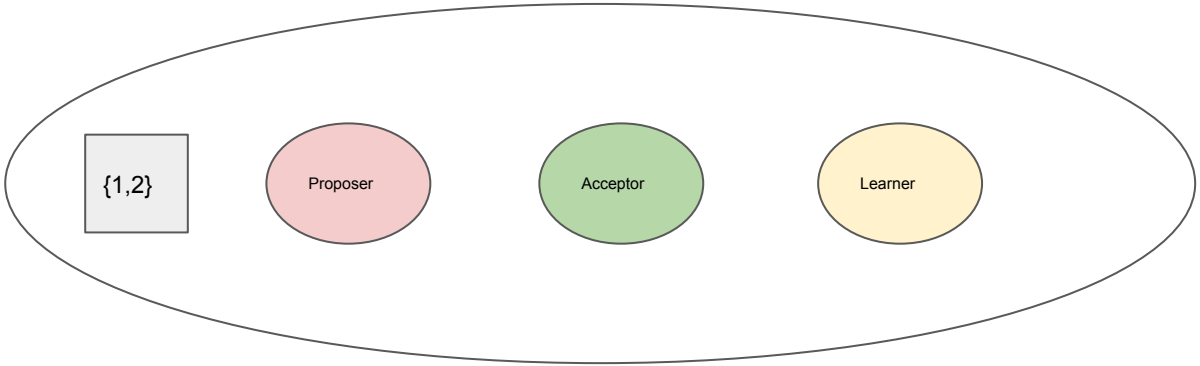
1. Proposer proposes 1 to acceptor

Let’s design our system! An illustration



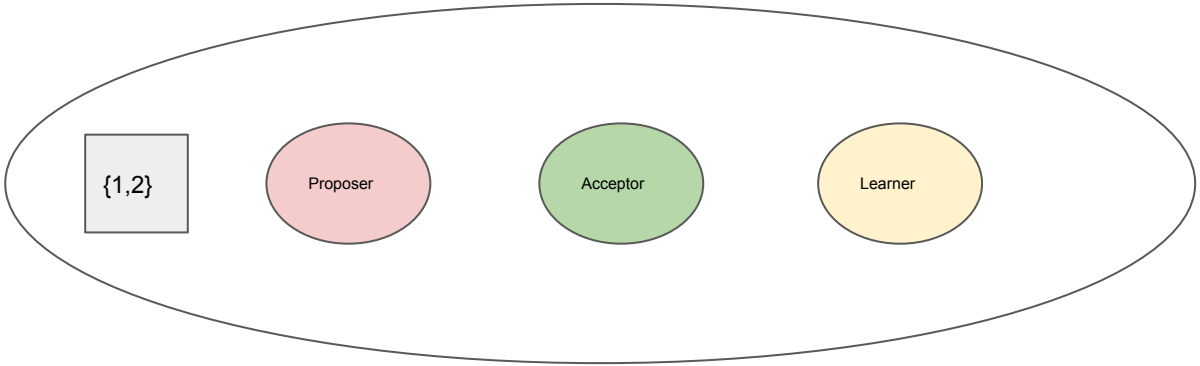
1. Proposer proposes 1 to acceptor
2. Proposer also decides to propose 2 to acceptor

Let’s design our system! An illustration



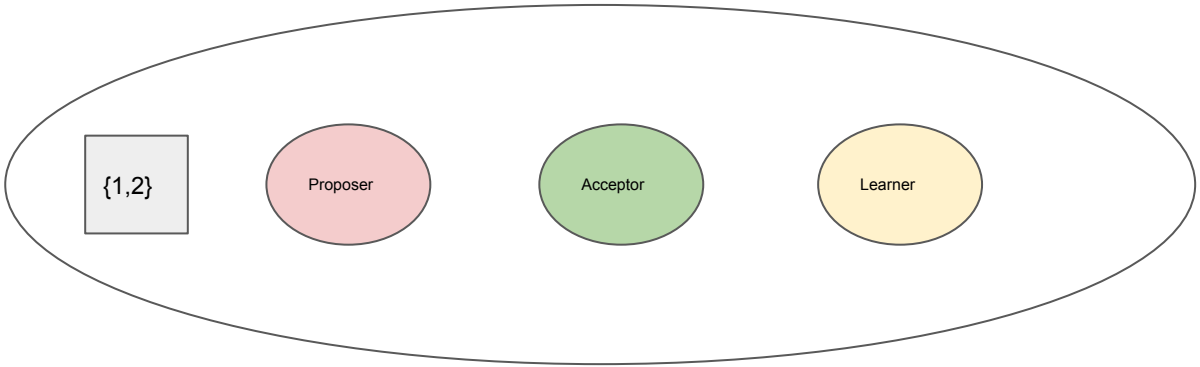
1. Proposer proposes 1 to acceptor
2. Proposer also decides to propose 2 to acceptor
3. Now the acceptor has received 1 as well as 2 from the proposer.
    - **Choosing algorithm**: Just store the first one received - this is THE ONE

Let’s design our system! An illustration



1. Proposer proposes 1 to acceptor
2. Proposer also decides to propose 2 to acceptor
3. Now the acceptor has received 1 as well as 2 from the proposer.
    - **Choosing algorithm**: Just store the first one received - this is THE ONE
4. Meanwhile, our learner keeps bugging the Proposer and Acceptor to let it know **THE ONE**. Once the Acceptor chooses either 1 or 0, it lets the learner know

Let’s design our system! An illustration



1. Proposer proposes 1 to acceptor
2. Proposer also decides to propose 2 to acceptor
3. Now the acceptor has received 1 as well as 2 from the proposer.
    - **Choosing algorithm**: Just store the first one received - this is THE ONE
4. Meanwhile, our learner keeps bugging the Proposer and Acceptor to let it know **THE ONE**. Once the Acceptor chooses either 1 or 0, it lets the learner know
5. The learner then sends the output

The End

Thank you. We have successfully implemented consensus!

Any questions!

Failures! Crashes!

Failures! Crashes!

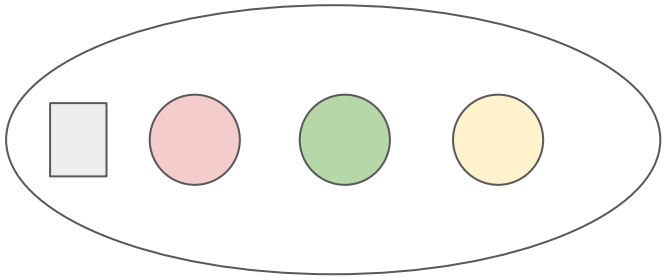
- Previous implementation **perfectly okay** if machines don’t crash!

Failures! Crashes!

- Previous implementation **perfectly okay** if machines don’t crash!
- However, it cannot handle it if even one of the following happens
  - Proposer crashes
  - Acceptor crashes
  - Learner crashes

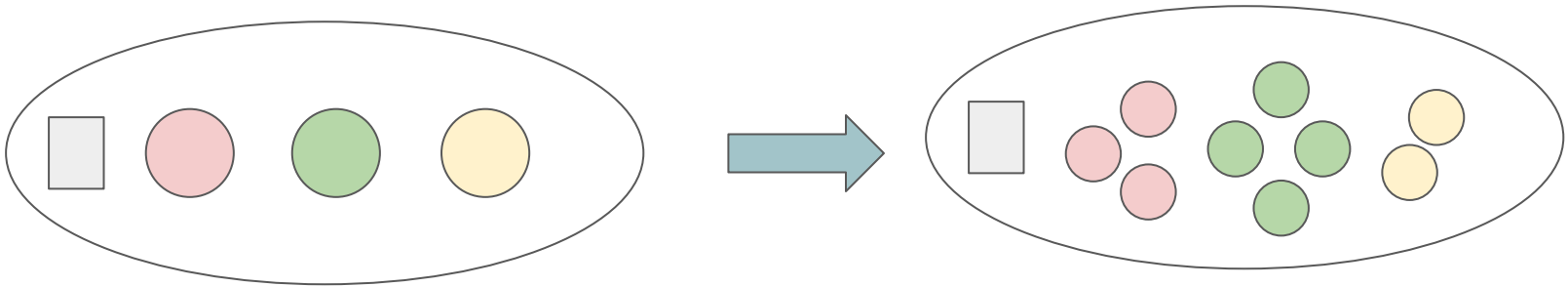
Failures! Crashes!

- Previous implementation **perfectly okay** if machines don’t crash!
- However, it cannot handle it if even one of the following happens
  - Proposer crashes
  - Acceptor crashes
  - Learner crashes



Failures! Crashes!

- Previous implementation **perfectly okay** if machines don’t crash!
- However, it cannot handle it if even one of the following happens
  - Proposer crashes
  - Acceptor crashes
  - Learner crashes



Side Note: What it means to crash (TLA$^+$ Style)

Typically, a process implementation is has the following form:

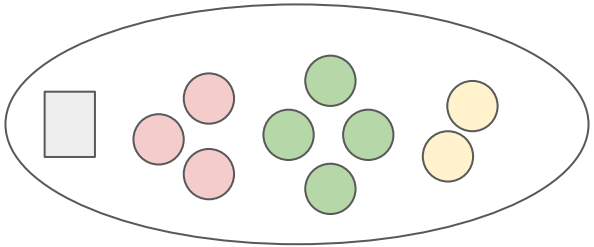
$$\text{Init} \land \Box[\text{Next}]_{\text{vars}} \land \text{Liveness}$$

Where:

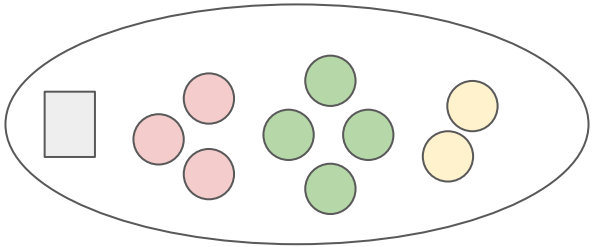
- **Init**: Constraints initial states of our process
- **Next:** Constraints possible executions **(But does not enforce them)**
- **Liveness:** Enforces that “Next” happens

Thus, if we can model a process using $\boldsymbol{Init \land \Box[Next]_{vars} \land Liveness}$, we can model a “crashable version” as $\boldsymbol{Init \land \Box[Next]_{vars}}$

Let’s design our Fault-Tolerant system!

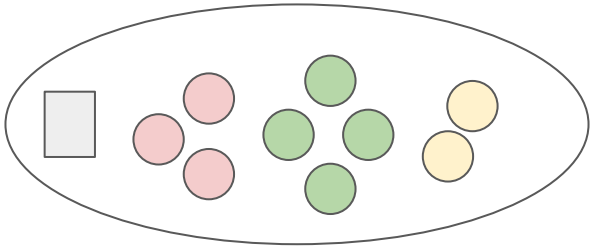


Let’s design our Fault-Tolerant system!



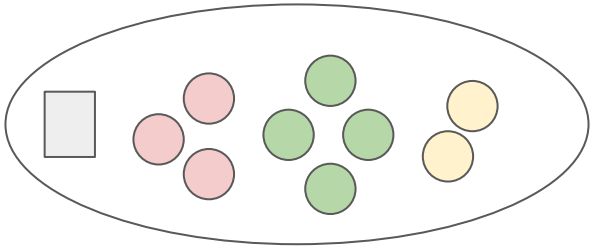
- Recall we have to implement **Propose-Choose-Learn.**

Let’s design our Fault-Tolerant system!



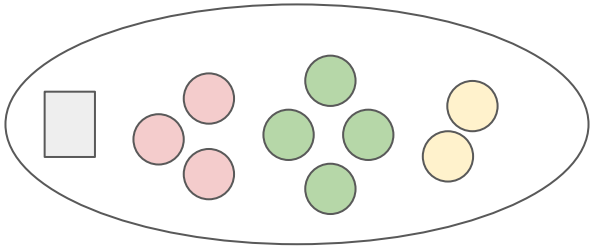
- Recall we have to implement **Propose-Choose-Learn.**
- The challenge here is getting to **Choose** (Here is where all the interesting things happen)

Let’s design our Fault-Tolerant system!



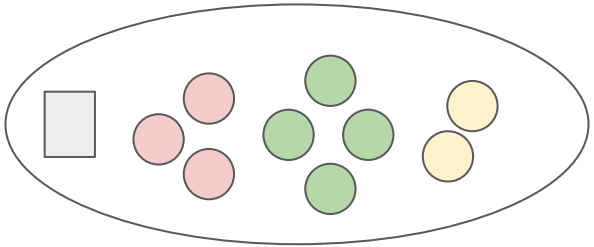
- Recall we have to implement **Propose-Choose-Learn.**
- The challenge here is getting to **Choose** (Here is where all the interesting things happen)
- Once we have a single value chosen for the rest of time, learning it is simple

Let’s design our Fault-Tolerant system!



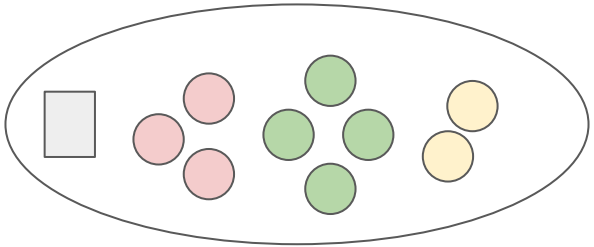
- Recall we have to implement **Propose-Choose-Learn.**
- The challenge here is getting to **Choose** (Here is where all the interesting things happen)
- Once we have a single value chosen for the rest of time, learning it is simple
- So, this talk will ignore listeners – Our goal is for the system to just **choose**

Let’s design our Fault-Tolerant system!

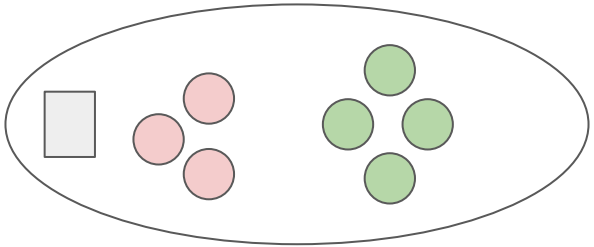


- Recall we have to implement **Propose-Choose-Learn.**
- The challenge here is getting to **Choose** (Here is where all the interesting things happen)
- Once we have a single value chosen for the rest of time, learning it is simple
- So, this talk will ignore listeners – Our goal is for the system to just **choose**.
- In Robbert’s protocol – The system has ‘chosen’ == it is univalent

Let’s design our Fault-Tolerant system!



- Recall we have to implement **Propose-Choose-Learn.**
- The challenge here is getting to **Choose** (Here is where all the interesting things happen)
- Once we have a single value chosen for the rest of time, learning it is simple
- So, this talk will ignore listeners – Our goal is for the system to just **choose**.
- In Robbert’s protocol – The system has ‘chosen’ => it is univalent



What it means to Choose

What it means to Choose

- Let’s have **THE ONE** (Our chosen value) as a state function

What it means to Choose

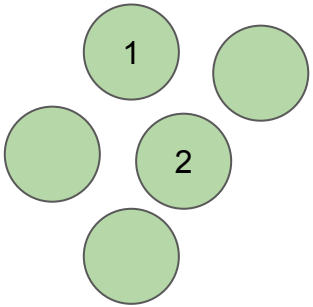
- Let’s have **THE ONE** (Our chosen value) as a state function
- If we freeze our entire system state and inspect everything in our system, we should know if the chosen value exists, and if it does, what it is

What it means to Choose

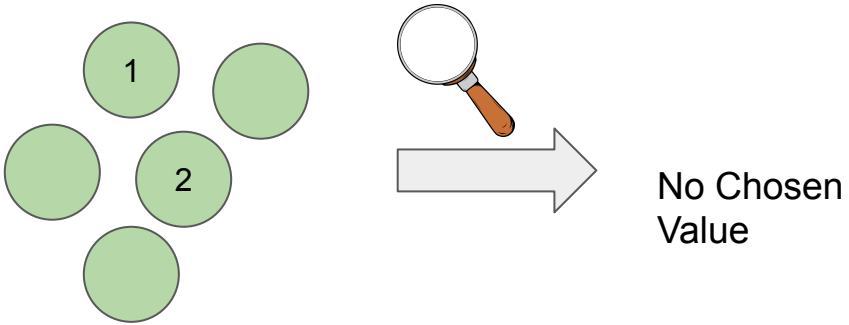
- Let’s have **THE ONE** (Our chosen value) as a state function
- If we freeze our entire system state and inspect everything in our system, we should know if the chosen value exists, and if it does, what it is
- Let’s have a simple way to determine this:
  - As part of each acceptor’s state, we (may) have a ‘value accepted’
  - If a majority of acceptors have the same ‘value accepted’, it is **THE ONE**

What it means to Choose - An Illustration

What it means to Choose - An Illustration

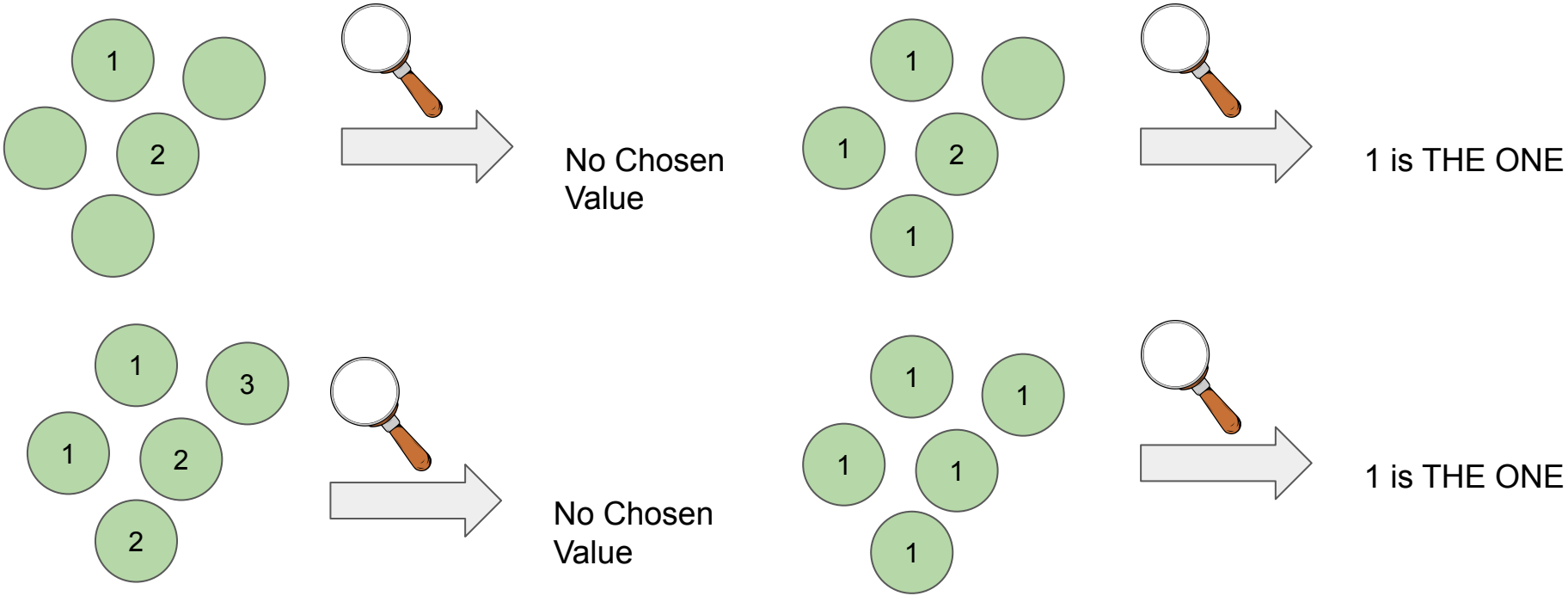


What it means to Choose - An Illustration



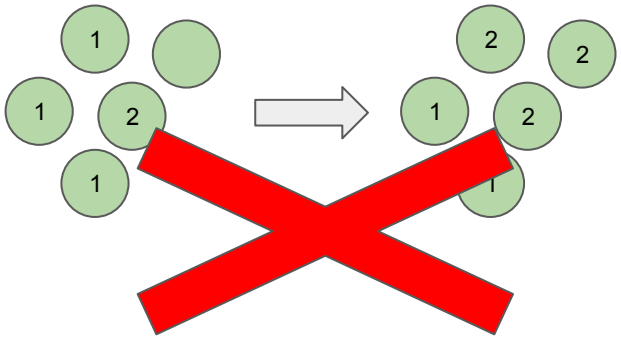
No Chosen Value

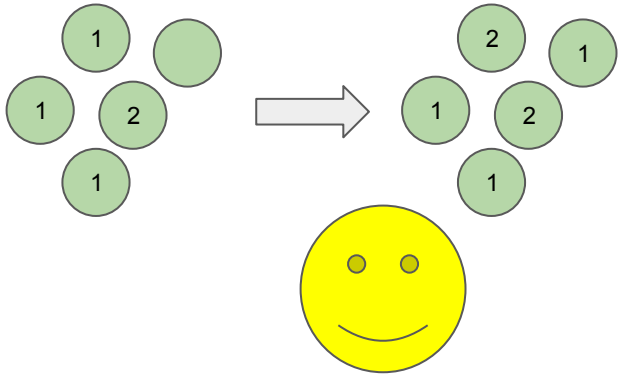
What it means to Choose - An Illustration



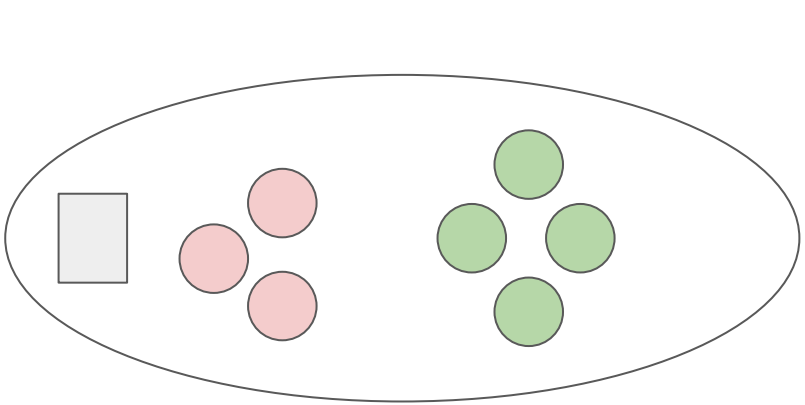
Some Implications

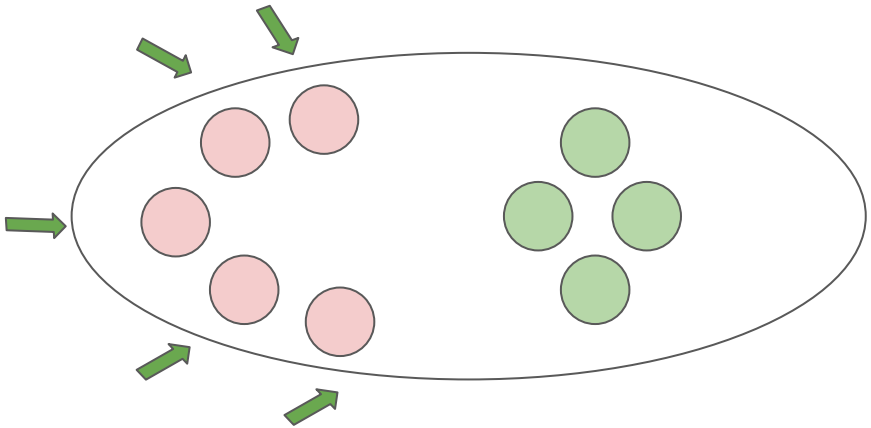
- Once a value is chosen, **no other value can be chosen** later
- In other words:
  - Once we have a majority of acceptors with *v* as the accepted value, and in the future, if some (other) majority of acceptors have *v'* as the accepted value, then *v* = *v'*





Side Note: The better way to think





Somehow, the first time I read the paper, the latter was the mental picture I got, where each proposer is ‘responsible’ for one input. However, the former is a better mental picture to keep:

- It stresses the fact that each agent is **working together**. Proposers just want one value to get accepted, and there is no notion of **their value.**
- This is especially true in the case of the full Paxos State Machine algorithm, where we have a **leader (distinguished proposer)**
- It is also easier to think about failure models – If we have n proposers, as many as n-1 proposers can crash.

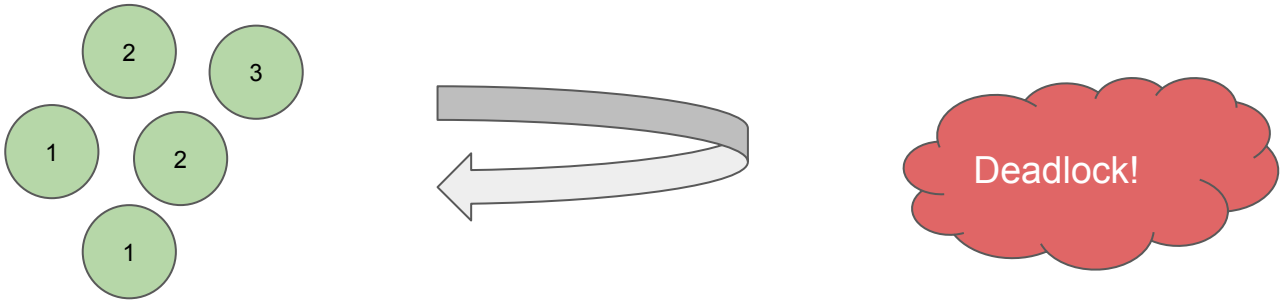
Simple Implementation Attempts

Simple Implementation Attempts

1. Every acceptor accepts **only** the first proposal it ever receives

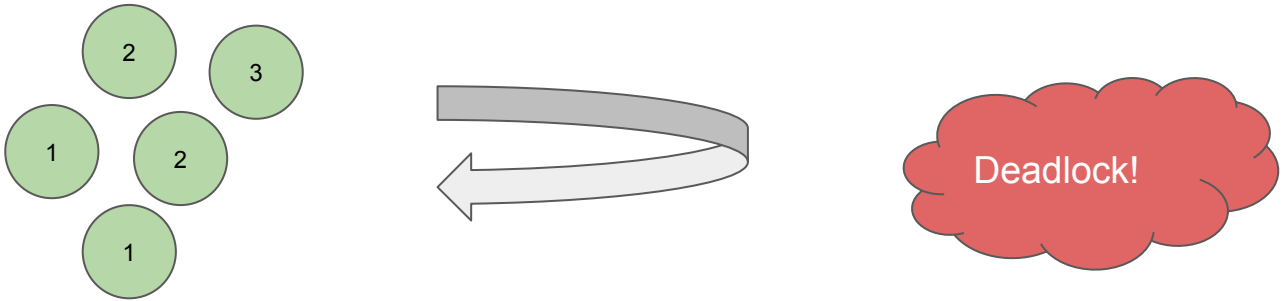
Simple Implementation Attempts

1. Every acceptor accepts **only** the first proposal it ever receives



Simple Implementation Attempts

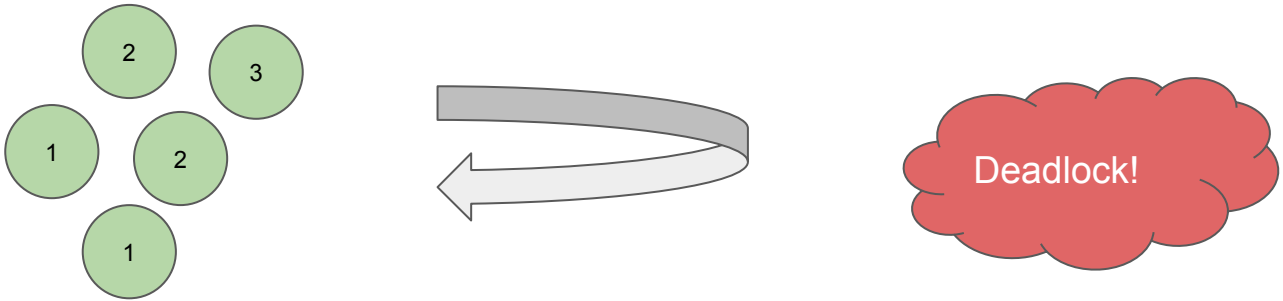
1. Every acceptor accepts **only** the first proposal it ever receives



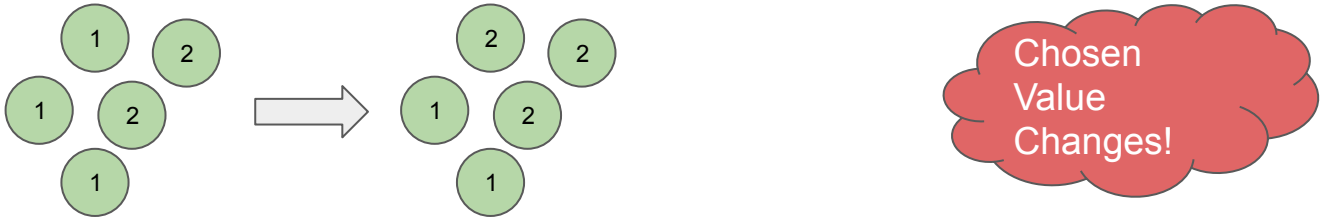
2. Every acceptor accepts **every** proposal it receives

Simple Implementation Attempts

1. Every acceptor accepts **only** the first proposal it ever receives



2. Every acceptor accepts **every** proposal it receives



A Two Phase Approach?

A Two Phase Approach?

When every acceptor accepts **every** proposal it receives, there may be a point when there is a chosen value, which may change if a new value is proposed and chosen.

A Two Phase Approach?

When every acceptor accepts **every** proposal it receives, there may be a point when there is a chosen value, which may change if a new value is proposed and chosen.

- However, let’s see if the proposer can somehow make a **safe guess** for this value, **by asking listeners first**, and then propose this guessed value.

A Two Phase Approach?

When every acceptor accepts **every** proposal it receives, there may be a point when there is a chosen value, which may change if a new value is proposed and chosen.

- However, let's see if the proposer can somehow make a **safe guess** for this value, **by asking listeners first**, and then propose this guessed value.
- When a Proposer asks Listeners, it may get a bunch of values that may have been accepted. We'd have to somehow choose one. **What is a safe guess?**

A Two Phase Approach?

When every acceptor accepts **every** proposal it receives, there may be a point when there is a chosen value, which may change if a new value is proposed and chosen.

- However, let's see if the proposer can somehow make a **safe guess** for this value, **by asking listeners first**, and then propose this guessed value.
- When a Proposer asks Listeners, it may get a bunch of values that may have been accepted. We'd have to somehow choose one. **What is a safe guess?**
- **Insight:** We'd have to compare proposals. Let us maintain a global order for proposals

Ordering Proposals

Ordering Proposals

- Every single proposal issued (obviously by a proposer) has a **unique** “timestamp” (just a number - a logical clock) along with a value

Ordering Proposals

- Every single proposal issued (obviously by a proposer) has a **unique** “timestamp” (just a number - a logical clock) along with a value
- Acceptors have a single “accepted proposal” as part of state (not just “accepted value”)



Ordering Proposals

- Every single proposal issued (obviously by a proposer) has a **unique** “timestamp” (just a number - a logical clock) along with a value
- Acceptors have a single “accepted proposal” as part of state (not just “accepted value”)



Unique Timestamps? – Even 2 proposers cannot send out proposals with the same number. **What? How can we achieve this?**

Ordering Proposals

- Every single proposal issued (obviously by a proposer) has a **unique** “timestamp” (just a number - a logical clock) along with a value
- Acceptors have a single “accepted proposal” as part of state (not just “accepted value”)



Unique Timestamps? – Even 2 proposers cannot send out proposals with the same number. **What? How can we achieve this?**

Simply partition $\mathbb{N}$ (fairly)!

“Choosing” Proposals

“Choosing” Proposals

- Now, we also modify what it means to ‘choose’, slightly

“Choosing” Proposals

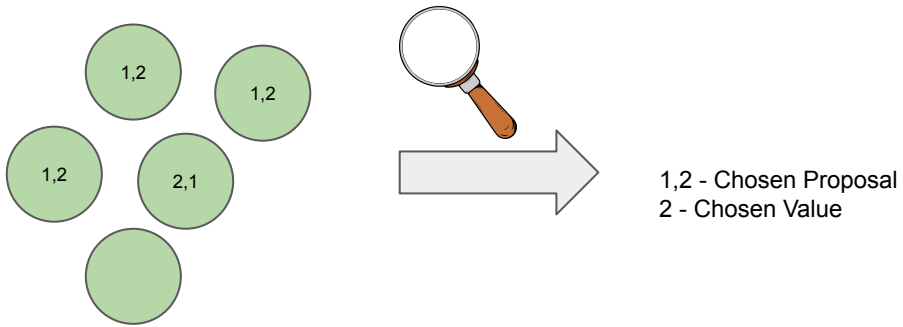
- Now, we also modify what it means to ‘choose’, slightly
- Proposal [# , Val] is chosen when it is the “accepted proposal” of a majority of acceptors

“Choosing” Proposals

- Now, we also modify what it means to ‘choose’, slightly
- Proposal [# , Val] is chosen when it is the “accepted proposal” of a majority of acceptors
- **Choosing a value** - When a proposal is chosen, its value is THE ONE

“Choosing” Proposals

- Now, we also modify what it means to ‘choose’, slightly
- Proposal [# , Val] is chosen when it is the “accepted proposal” of a majority of acceptors
- **Choosing a value** - When a proposal is chosen, its value is THE ONE



“Ask and Propose”

Let’s design this for proposers.

“Ask and Propose”

Let’s design this for proposers.

We keep the following motto: *Proposers work together. If a proposal is chosen, every ‘greater’ (larger numbered) proposal proposed will have THE ONE as the value*.

“Ask and Propose”

Let’s design this for proposers.

We keep the following motto: *Proposers work together. If a proposal is chosen, every ‘greater’ (larger numbered) proposal proposed will have THE ONE as the value*.

An illustration:

- Ask listeners and wait for a majority response
- Suppose we get { [2,a] , [1,b] , [3, c] , nil , nil } as the response, **what is the value we propose**? **There is only one that upholds our motto.**

“Ask and Propose”

Let’s design this for proposers.

We keep the following motto: *Proposers work together. If a proposal is chosen, every ‘greater’ (larger numbered) proposal proposed will have THE ONE as the value*.

An illustration:

- Ask listeners and wait for a majority response
- Suppose we get { [2,a] , [1,b] , [3, c] , nil , nil } as the response, **what is the value we propose**? **There is only one that upholds our motto.**
- [3,c] is **the guess**, keeping our motto in mind
- As a patriotic sender, we uphold the motto by sending [n,c] to as many acceptors as we can (where n > 3)

“Ask and Propose” - The End?

“Ask and Propose” - The End?

It’s been fun so far. However, let’s see what happens when 2 proposers do this Ask - Propose cycle.

“Ask and Propose” - The End?

It’s been fun so far. However, let’s see what happens when 2 proposers do this Ask - Propose cycle.

1. Proposer 1 asks and waits for majority response – gets { nil }
    a. Now, any value is as good as any other to be THE ONE
    b. Proposer decides to propose 1
    c. Proposer 1 patriotically sends [100, 1] to as many acceptors as possible

“Ask and Propose” - The End?

It’s been fun so far. However, let’s see what happens when 2 proposers do this Ask - Propose cycle.

1. Proposer 1 asks and waits for majority response – gets { nil }
    a. Now, any value is as good as any other to be THE ONE
    b. Proposer decides to propose 1
    c. Proposer 1 patriotically sends [100, 1] to as many acceptors as possible
2. Proposer 2 asks and waits for majority response – gets { nil }
    a. …
    b. Proposer 2 patriotically sends [80, 2] to as many acceptors as possible

“Ask and Propose” - The End?

It’s been fun so far. However, let’s see what happens when 2 proposers do this Ask - Propose cycle.

1. Proposer 1 asks and waits for majority response – gets { nil }
   a. Now, any value is as good as any other to be THE ONE
   b. Proposer decides to propose 1
   c. Proposer 1 patriotically sends [100, 1] to as many acceptors as possible
2. Proposer 2 asks and waits for majority response – gets { nil }
   a. …
   b. Proposer 2 patriotically sends [80, 2] to as many acceptors as possible
3. Here is how the acceptors state may change

{ nil , nil , nil } → … → { [80,2] , [80,2] , [100,1] } → … → { [80,2] , [100,1] , 100,1] }

“Ask and Propose” - The End?

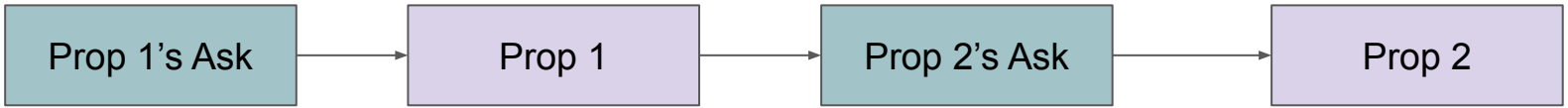
It’s been fun so far. However, let’s see what happens when 2 proposers do this Ask - Propose cycle.

1. Proposer 1 asks and waits for majority response – gets { nil }
   a. Now, as good as any other to be THE ONE
   b. Proposer se 1
   c. Proposer 1 patri 00, 1] to rs as possible
2. Proposer 2 asks and w nse – gets { nil }
   a. …
   b. Proposer 2 patriotic tors as possible
3. Here is how th state may cha

{ nil , nil , nil } → { [80,2] , [80,2] , [100,1] } → … 2] , [100,1] , 100,1] }

The Problem

- Ideally, we would like a situation as follows:
  - Ask Completes
  - Its Propose Completes
  - Another Ask Completes
  - Its Propose Completes …..
- What's happening is something like this:
  - Ask, Ask, Propose, Propose
- How can we solve this?
- **Let's have a global order on everything**
- When an "Ask" completes, "competing past Proposes" should not be allowed to complete

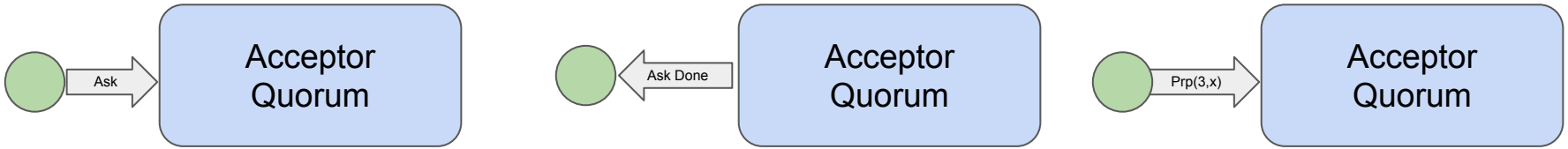


The Solution

- We have to order “Asks” as well – Simply include the proposal number we will use as well when we send an “Ask”.

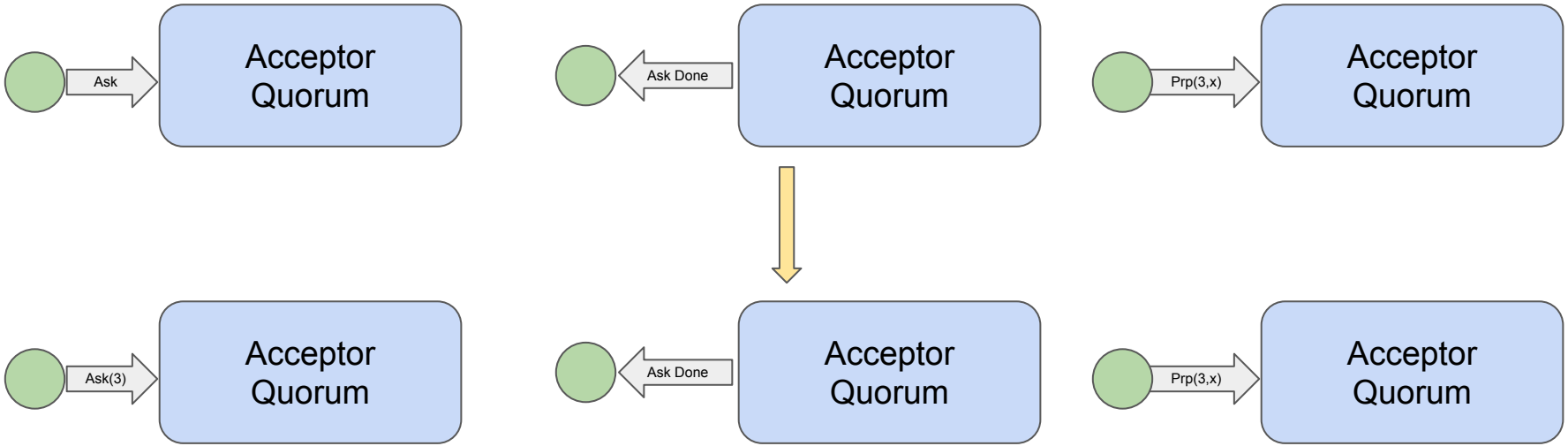
The Solution

- We have to order “Asks” as well – Simply include the proposal number we will use as well when we send an “Ask”.



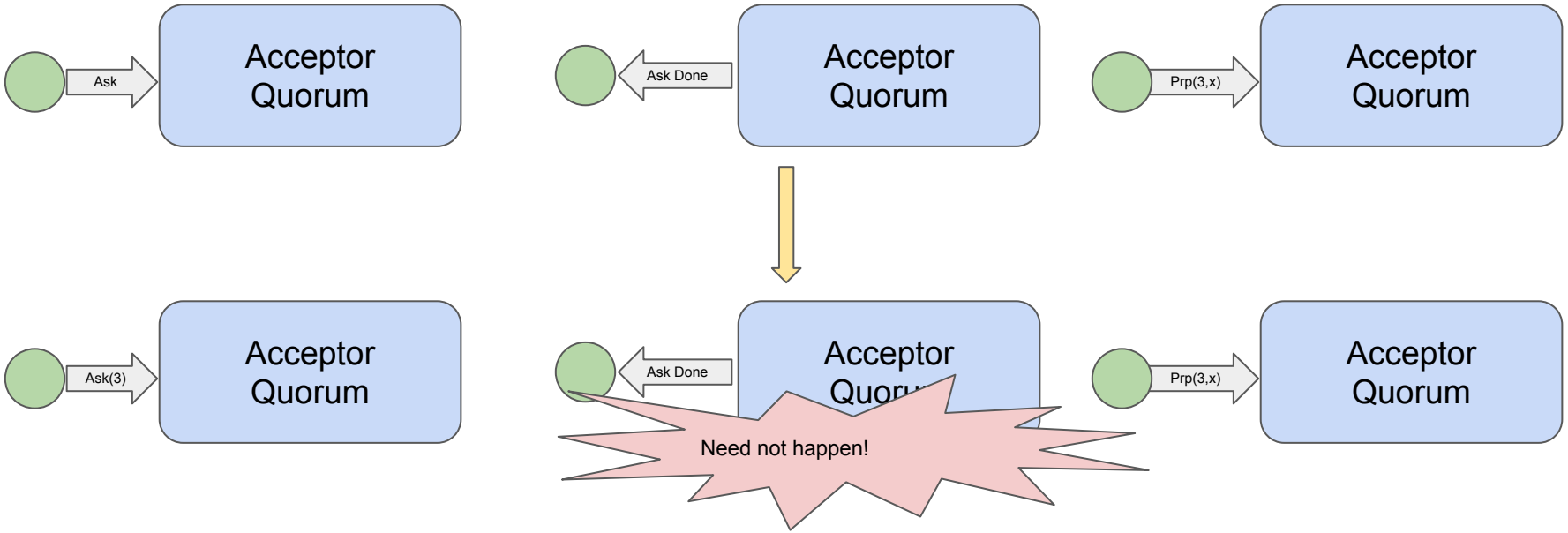
The Solution

- We have to order “Asks” as well – Simply include the proposal number we will use as well when we send an “Ask”.



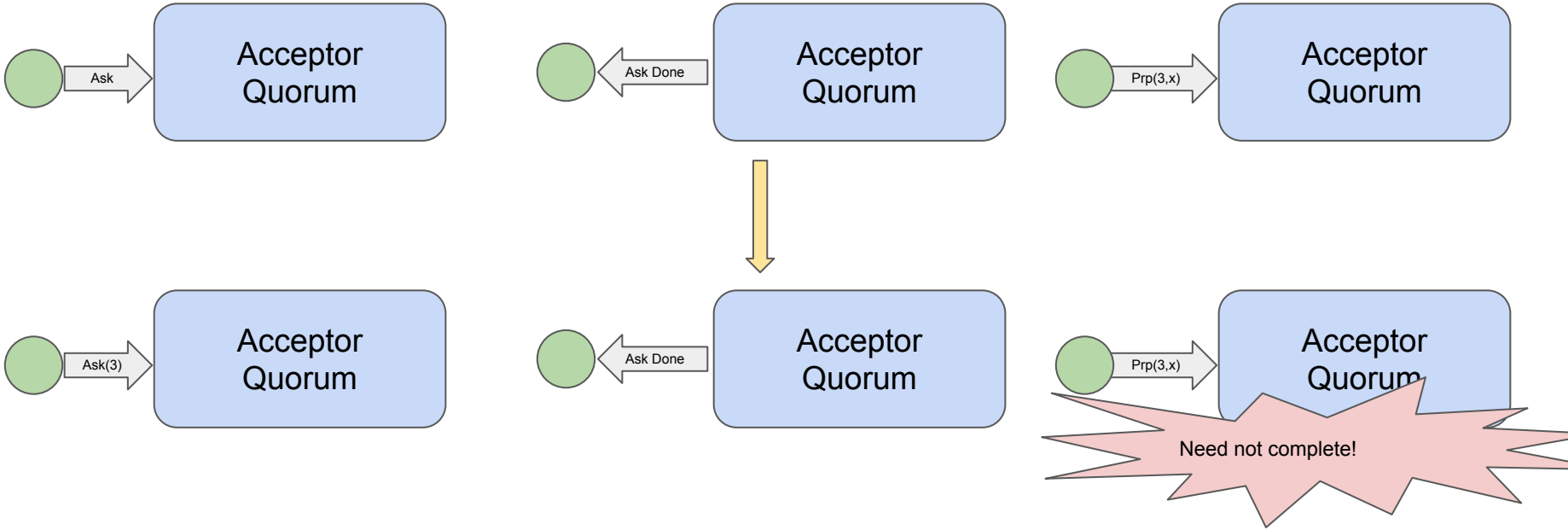
The Solution

- We have to order “Asks” as well – Simply include the proposal number we will use as well when we send an “Ask”.



The Solution

- We have to order “Asks” as well – Simply include the proposal number we will use as well when we send an “Ask”.



The Protocol

Acceptors

The Protocol

Acceptors

- Each acceptor, in addition to storing an accepted proposal, stores a “newest number seen”

The Protocol

Acceptors

- Each acceptor, in addition to storing an accepted proposal, stores a “newest number seen”
- A request (Ask / Propose) with a lower number should not be allowed to complete → Acceptors need not entertain these requests

The Protocol

Acceptors

- Each acceptor, in addition to storing an accepted proposal, stores a “newest number seen”
- A request (Ask / Propose) with a lower number should not be allowed to complete → Acceptors need not entertain these requests
- Any new request increases this “newest number seen”. In addition:
  - For Ask, we have to send the response
  - For Propose, we have to store the proposal

The Protocol

Acceptors

- Each acceptor, in addition to storing an accepted proposal, stores a “newest number seen”
- A request (Ask / Propose) with a lower number should not be allowed to complete → Acceptors need not entertain these requests
- Any new request increases this “newest number seen”. In addition:
  - For Ask, we have to send the response
  - For Propose, we have to store the proposal

Proposers

The Protocol

Acceptors

- Each acceptor, in addition to storing an accepted proposal, stores a “newest number seen”
- A request (Ask / Propose) with a lower number should not be allowed to complete → Acceptors need not entertain these requests
- Any new request increases this “newest number seen”. In addition:
  - For Ask, we have to send the response
  - For Propose, we have to store the proposal

Proposers

- Send an “Ask(n)” to as many acceptors as possible.

The Protocol

Acceptors

- Each acceptor, in addition to storing an accepted proposal, stores a “newest number seen”
- A request (Ask / Propose) with a lower number should not be allowed to complete → Acceptors need not entertain these requests
- Any new request increases this “newest number seen”. In addition:
  - For Ask, we have to send the response
  - For Propose, we have to store the proposal

Proposers

- Send an “Ask(n)” to as many acceptors as possible.
- If we get a majority response, follow the Proposer motto and propose a **safe value**

The Protocol

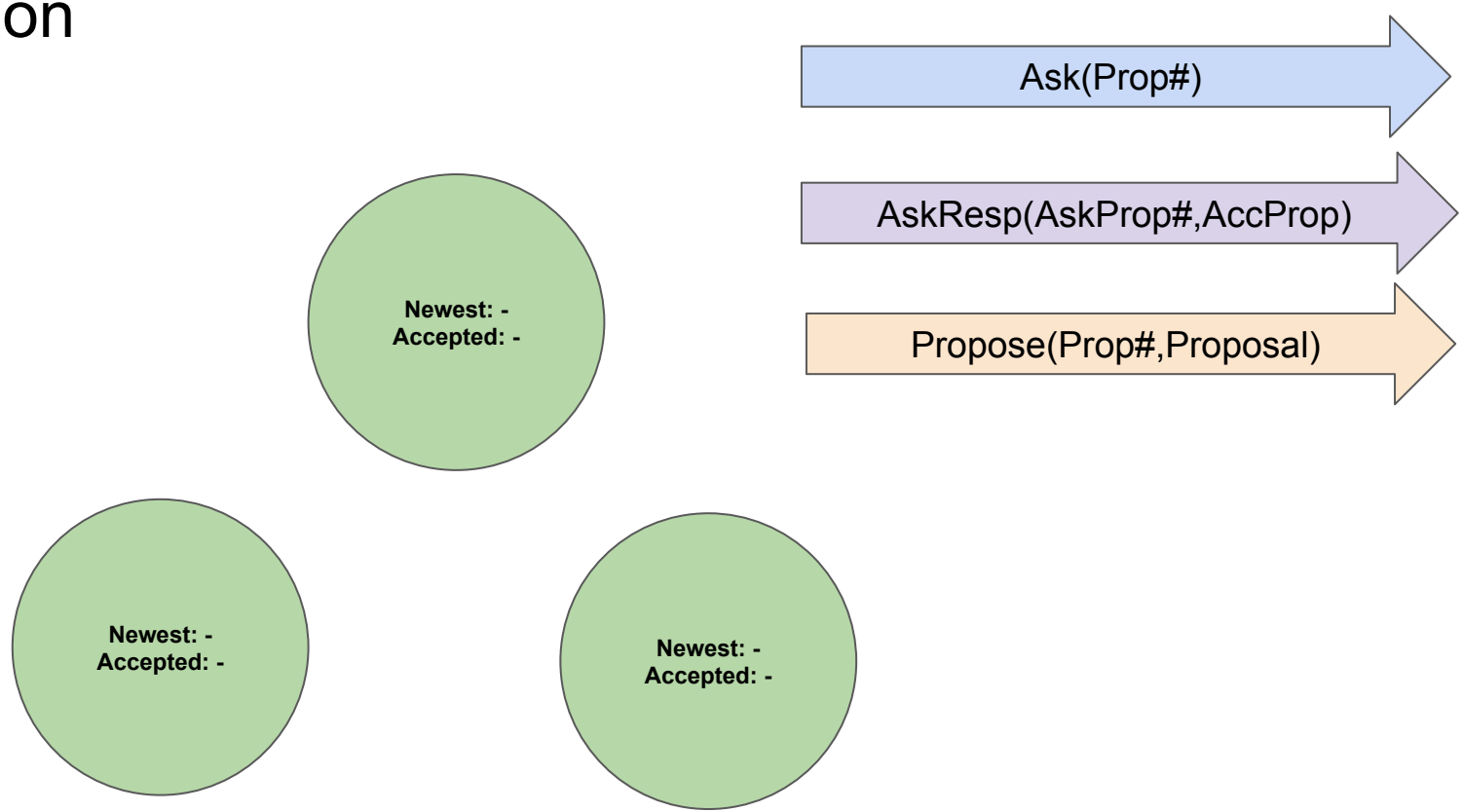
Acceptors

- Each acceptor, in addition to storing an accepted proposal, stores a “newest number seen”
- A request (Ask / Propose) with a lower number should not be allowed to complete → Acceptors need not entertain these requests
- Any new request increases this “newest number seen”. In addition:
  - For Ask, we have to send the response
  - For Propose, we have to store the proposal

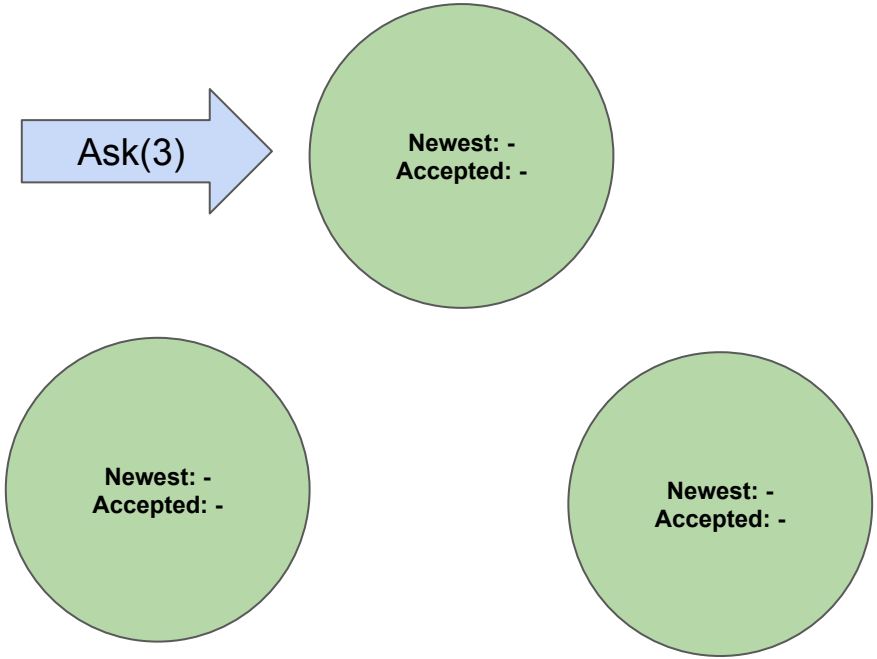
Proposers

- Send an “Ask(n)” to as many acceptors as possible.
- If we get a majority response, follow the Proposer motto and propose a **safe value**
- If we don’t get a majority response, no harm done – We can just try to “Ask” with a bigger proposal number

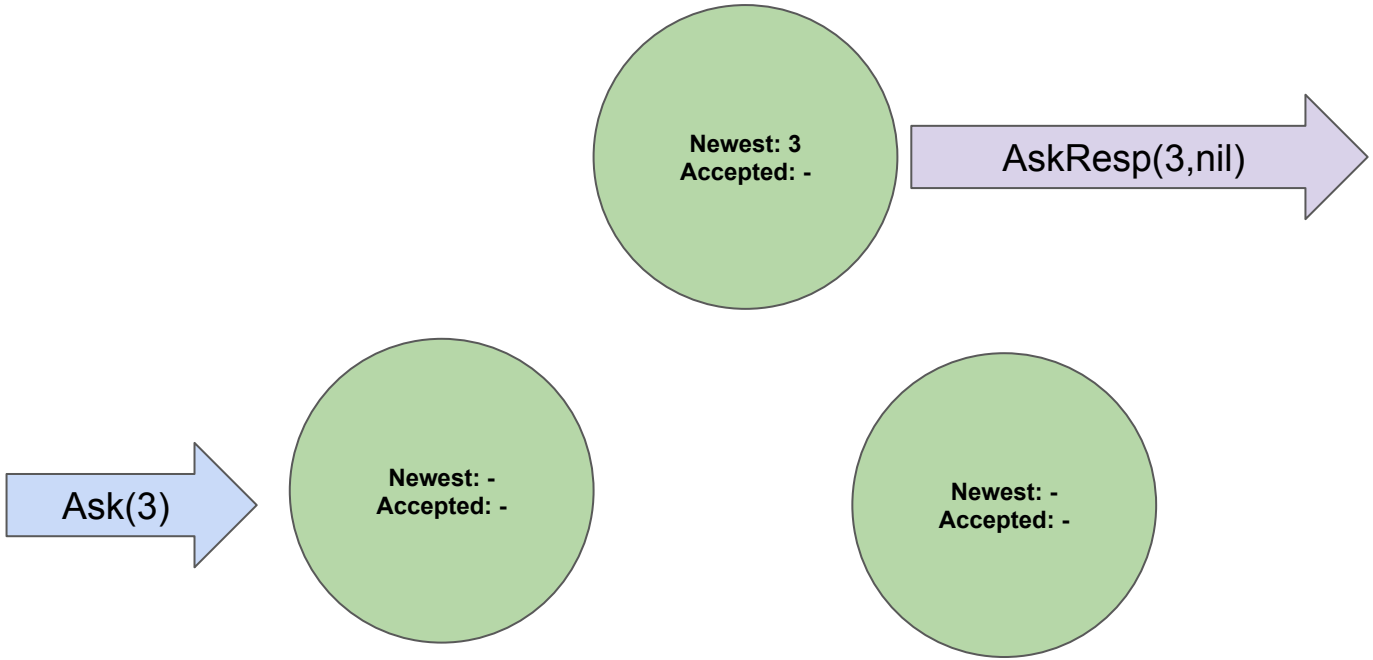
An Illustration



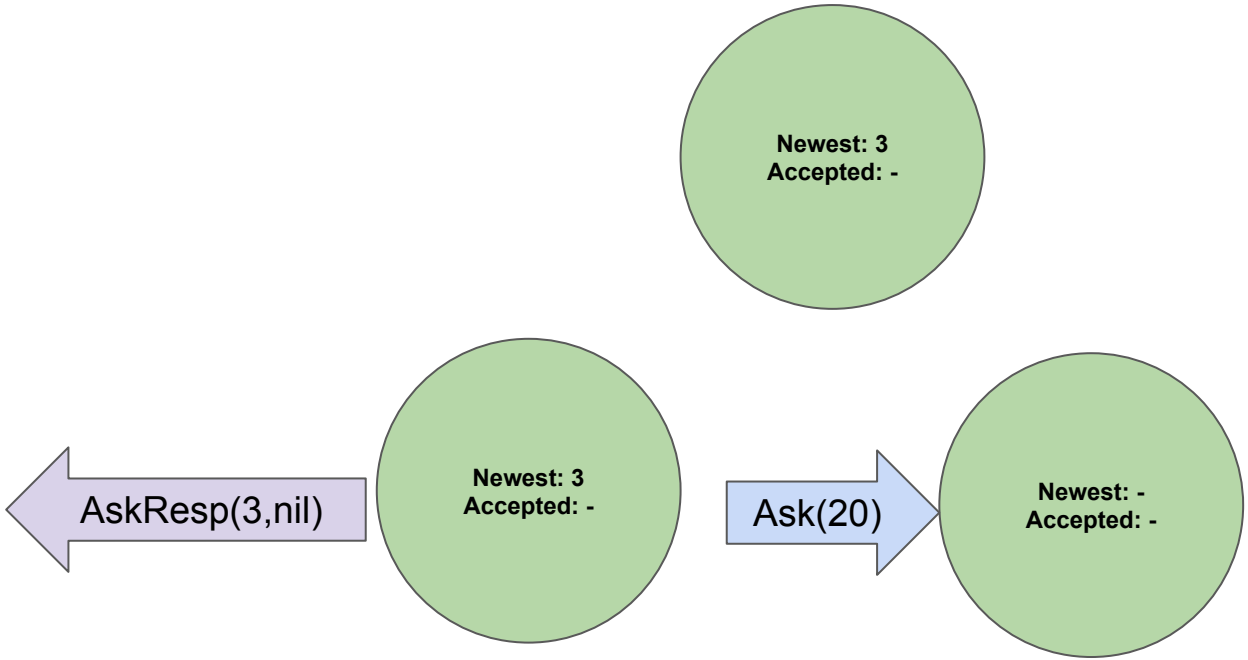
An Illustration



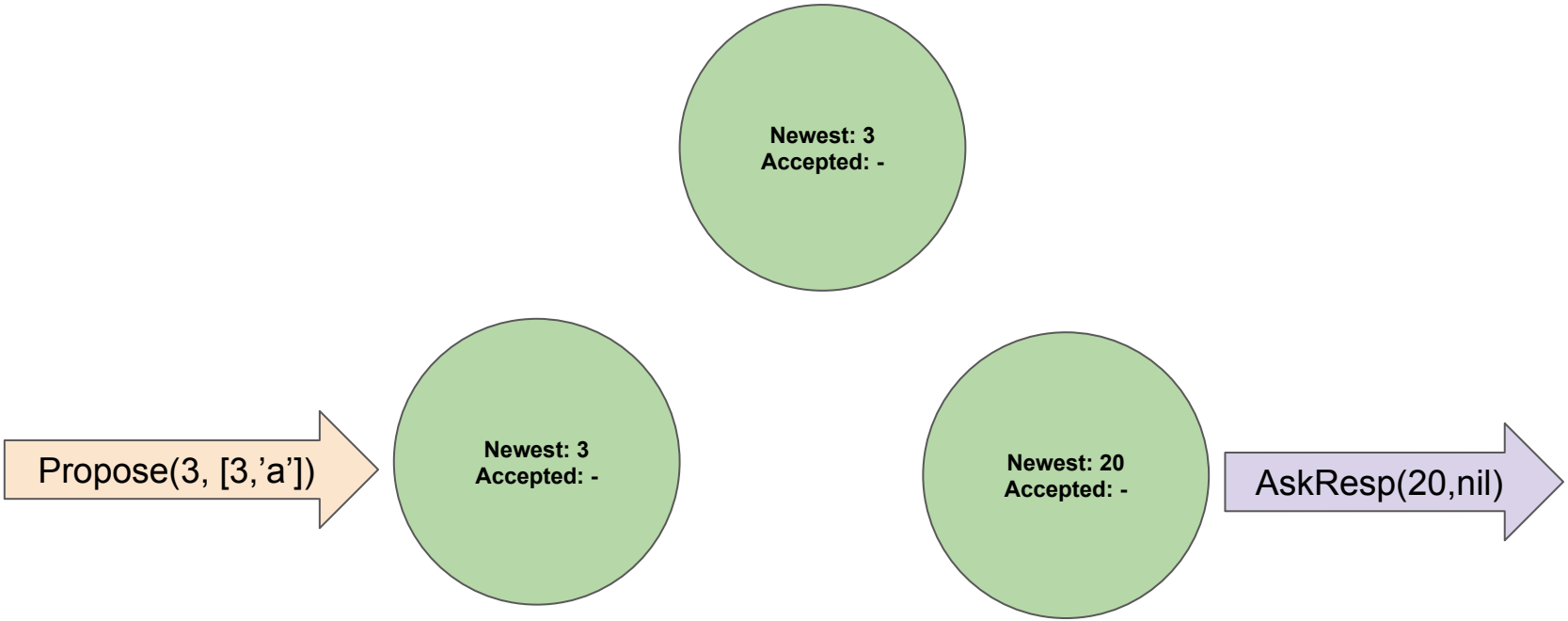
An Illustration



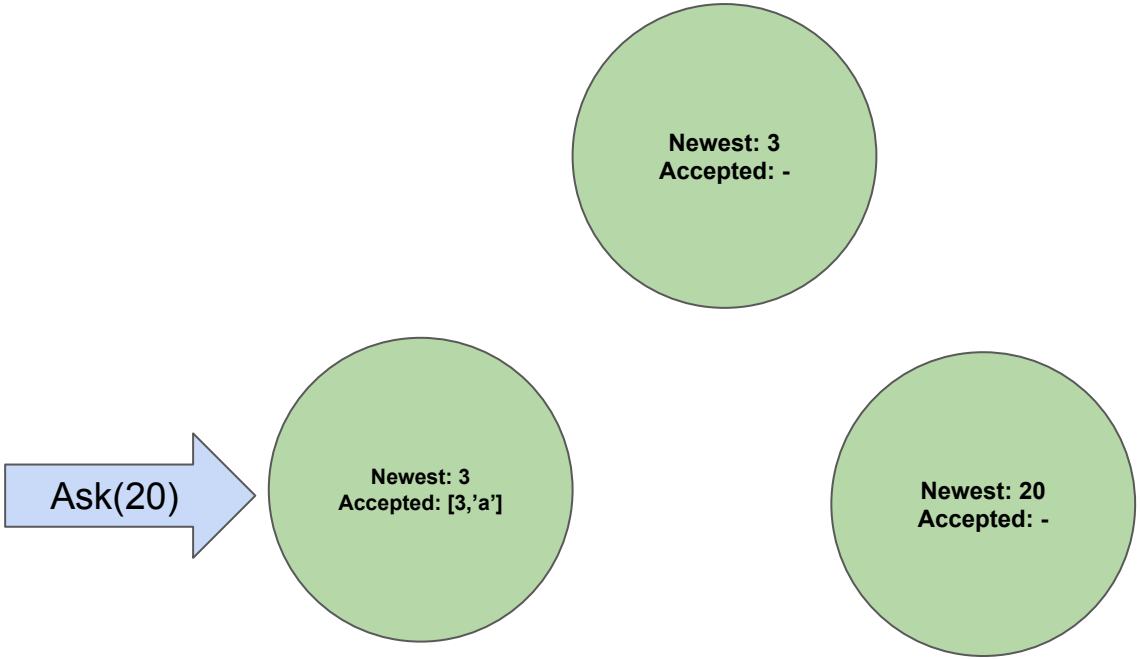
An Illustration



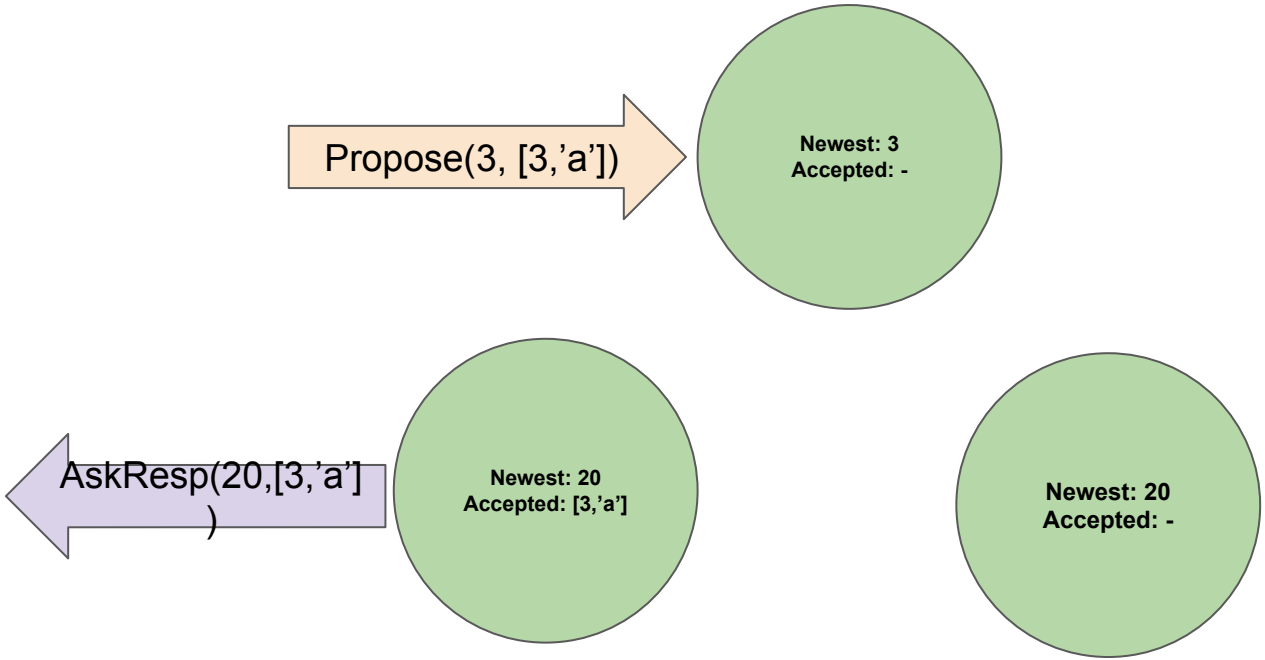
An Illustration



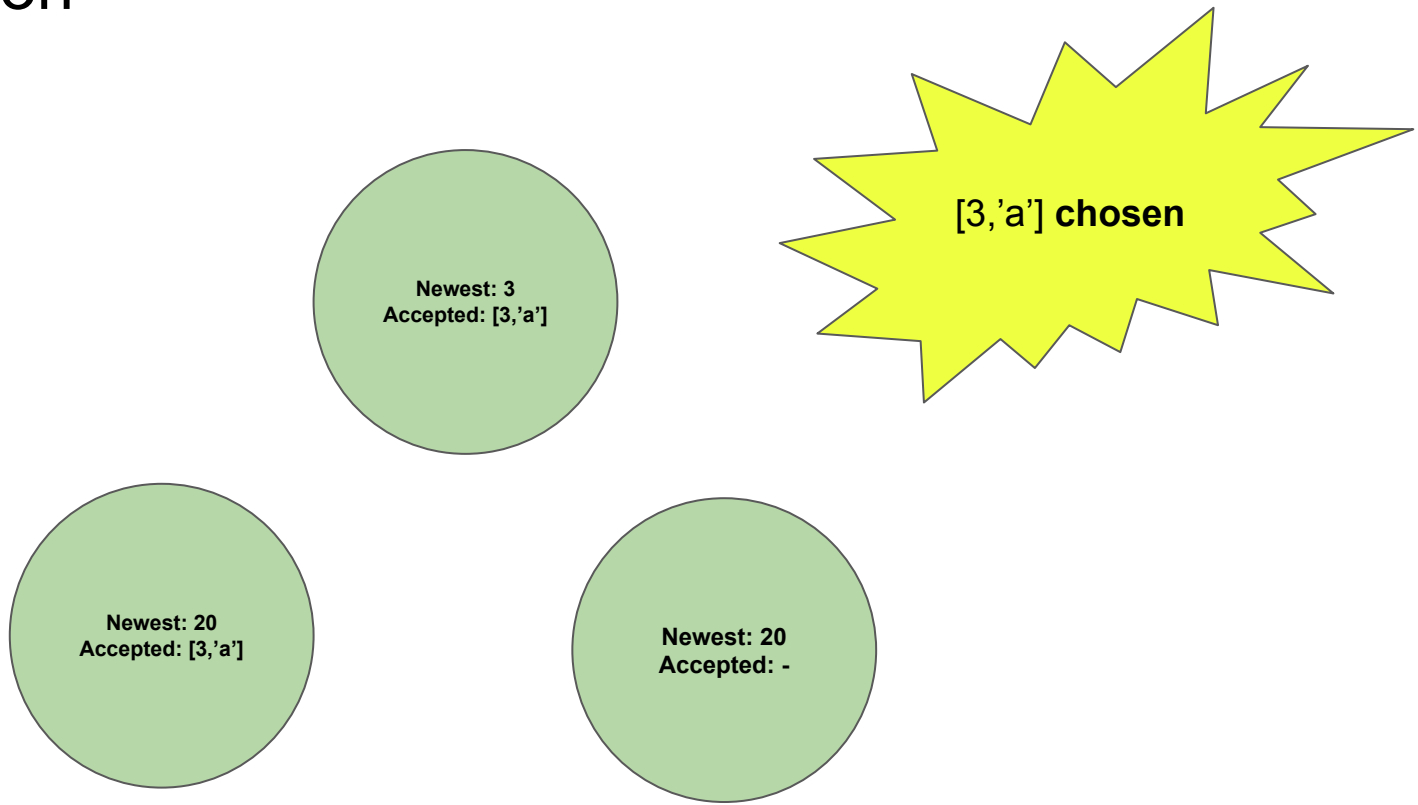
An Illustration



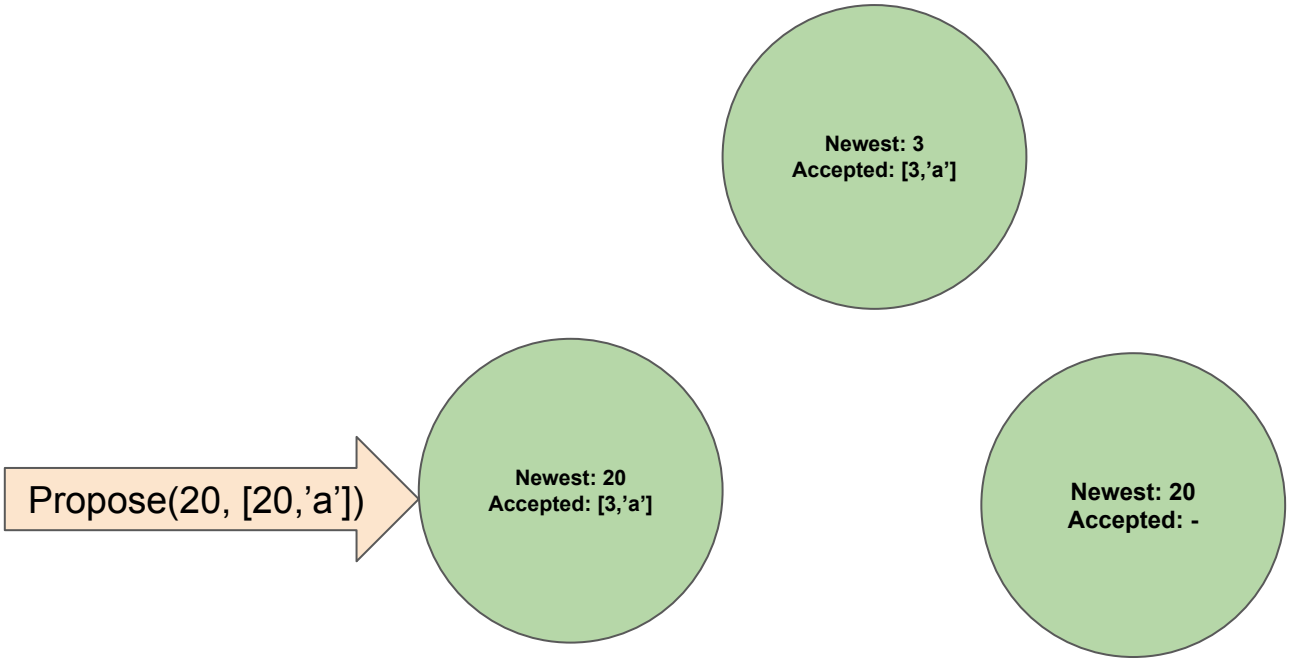
An Illustration



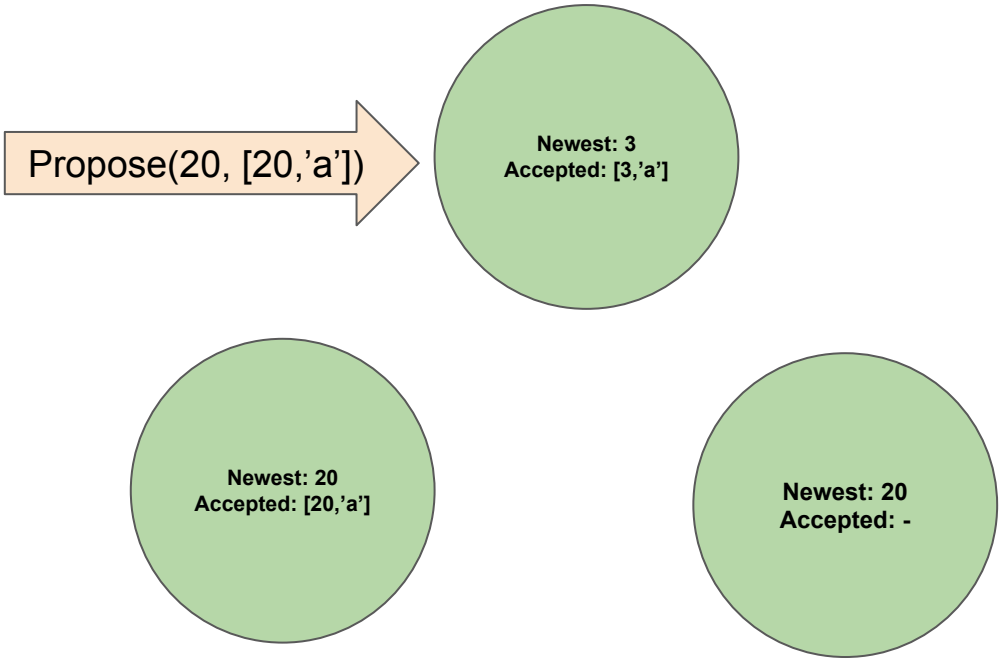
An Illustration



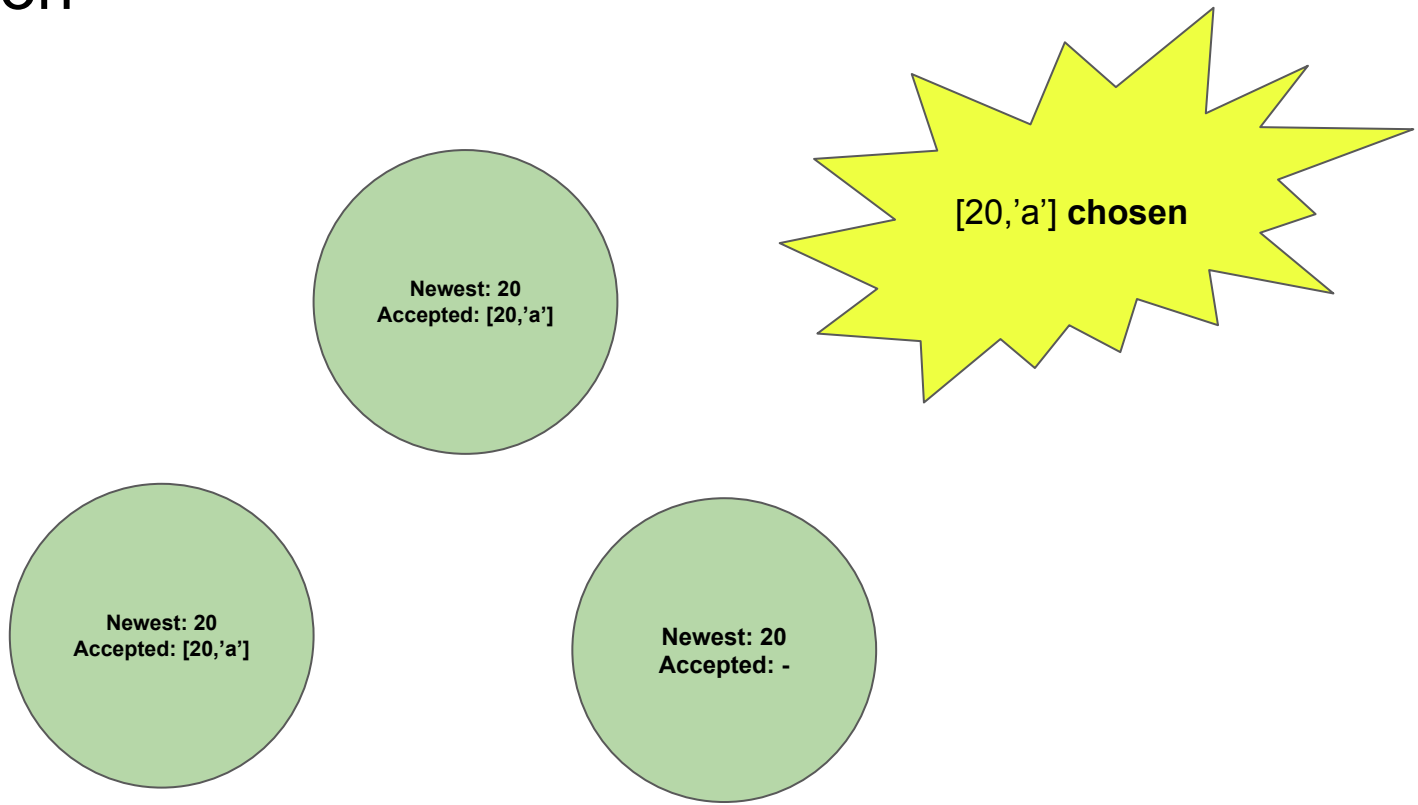
An Illustration



An Illustration



An Illustration



Some Notes

- In the paper,
  - *Ask* is called *Prepare*
  - *Propose* is called *Accept*

Some Notes

- In the paper,
  - *Ask* is called *Prepare*
  - *Propose* is called *Accept*
- Termination is not guaranteed
  - We can be stuck in a Prepare(1), Prepare(2), Prepare(3) …. execution with no value being chosen

Some Notes

- In the paper,
  - *Ask* is called *Prepare*
  - *Propose* is called *Accept*
- Termination is not guaranteed
  - We can be stuck in a Prepare(1), Prepare(2), Prepare(3) …. execution with no value being chosen
- What about *learning?*

Some Notes

- In the paper,
  - *Ask* is called *Prepare*
  - *Propose* is called *Accept*
- Termination is not guaranteed
  - We can be stuck in a Prepare(1), Prepare(2), Prepare(3) …. execution with no value being chosen
- What about *learning?*
- There are several ways to do the learning step:
  - Learners can keep bugging acceptors, and get a majority of their responses to see the chosen value
  - Learners can also ask a proposer to propose. The proposer can easily learn the chosen value by examining responses

Maybe next time?

- Full Paxos!
- A complete state machine implementation
- **Idea:** Have each node run several instances of single paxos
  - One instance per each entry of the replicated log
  - We try to keep some proposal number as ‘distinguished’, which will be the only one sending *Accept* messages to each instance.
  - We wait a little longer for distinguished proposals to complete (maybe by ignoring newer *Prepare* messages)
  - Is super efficient
- Hopefully, this talk gives you a good foundation to embark on that journey!

Discussion

- How many failures can we tolerate?
  - Note that we have multiple proposers, acceptors, and listeners
  - Each of these can fail

Discussion

- How many failures can we tolerate?
  - Note that we have multiple proposers, acceptors, and listeners
  - Each of these can fail
- Recall RVR’s consensus protocol. Why is this protocol so different?
  - What is the ‘state’ in that protocol?