Decentralized Systems Engineering

CS-438 – Fall 2024

DEDIS

Pierluca Borsò-Tan and Bryan Ford

EPFL

Credits: D. Ongaro, J. Ousterhout, L. Alvisi, A. Ghodsi, D. Mazières, L.Q. Torres, et al.

So far...

- Decentralized communication
- Unstructured & structured search
- Data storage

Let’s pick up where we left off ...

Conflict-Free Replicated Data Types (CRDTs)

Various types:

- Values
- Counters
- Sets
- Lists
- Log-based
- Text

Two main categories:

- Operation-based – commutative replicated data types (CmRDTs)
- State-based – convergent replicated data types (CvRDTs)

→ Theoretically equivalent

State-based CRDT – Formalism

Let U be the set of update operations, and V the set of values.

A state-based CRDT is a 5-tuple (S, s^0, q, u, m), where:

- S is the set of states;
- $s^0 \in S$ is the initial state;

- $q : S \to V$ is the **query** function
- $u : S \times U \to S$ is the **update** function
- $m : S \times S \to S$ is the **merge** function

G-Counter CRDT

Specifications:

- Grow-only counter,
  replicated across N machines

- Add(x)
  updates our local counter

- Query() returns the value

- Merge(other_state)
  merge's other's state

```
class GCounter(object):
  def __init__(self, i, n):
    self.i = i # server id
    self.n = n # number of servers
    self.xs = [0] * n

  def add(self, x):
    assert x >= 0
    self.xs[self.i] += x

  def query(self):
    return sum(self.xs)

  def merge(self, other):
    zipped = zip(self.xs, other.xs)
    self.xs = [max(x, y) for (x, y) in zipped]
```

G-Counter CRDT

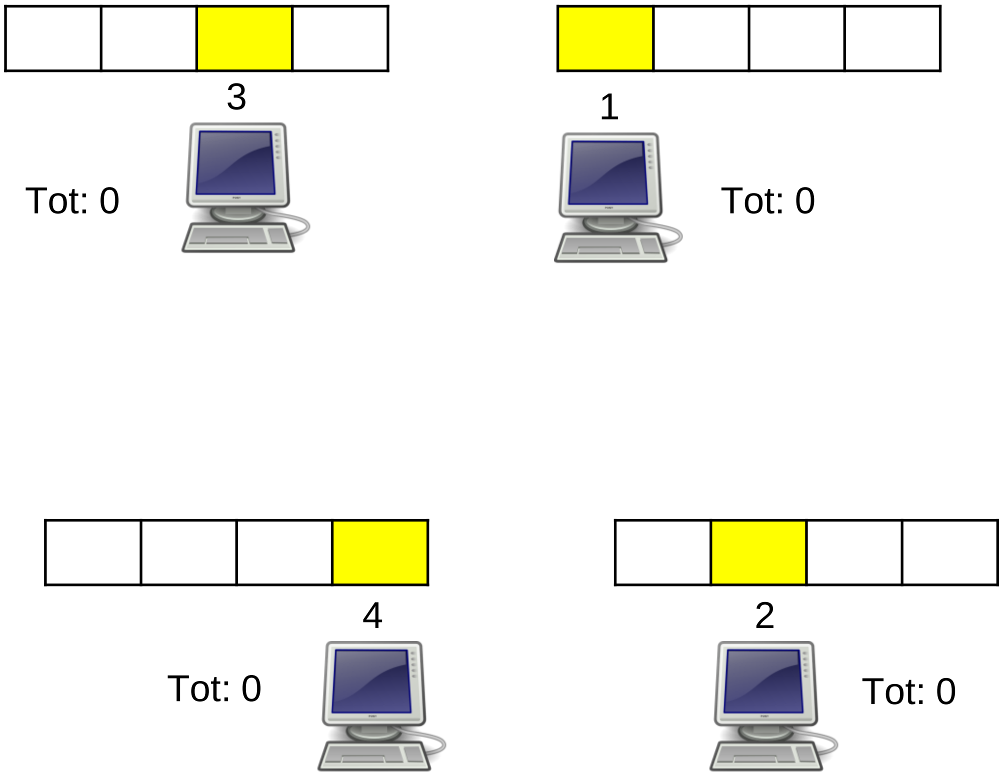
Specifications:

- Grow-only counter,
  replicated across N machines

- Add(x)
  updates our local counter

- Query() returns the value

- Merge(other_state)
  merge's other's state



G-Counter CRDT

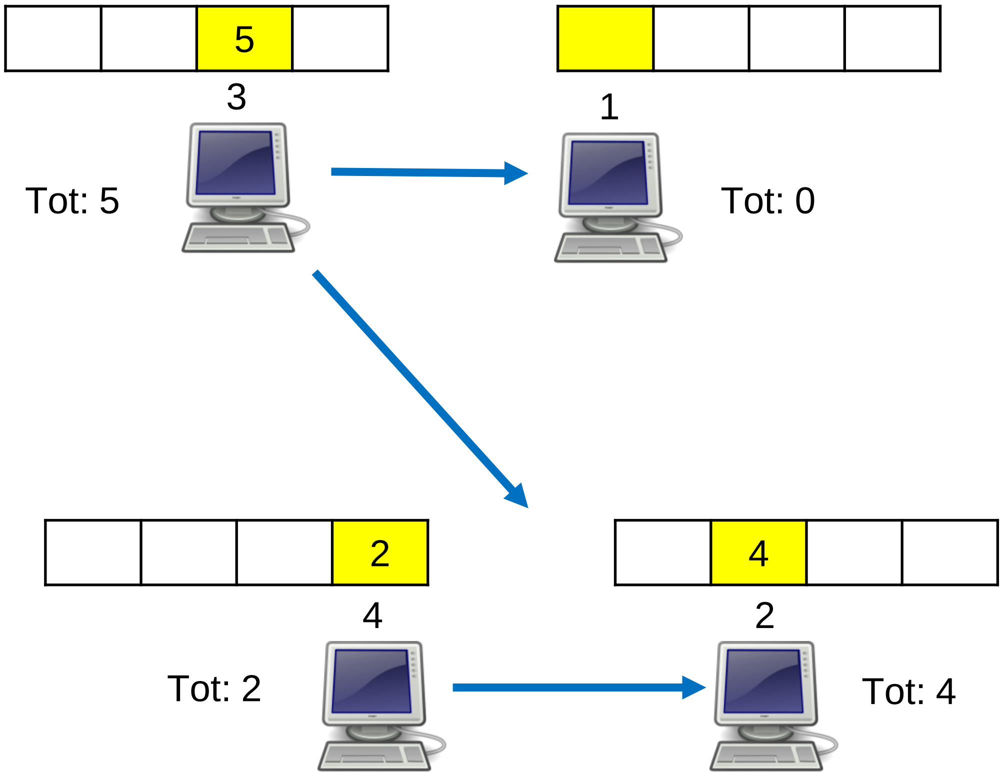
Specifications:

- Grow-only counter,
  replicated across N machines

- Add(x)
  updates our local counter

- Query() returns the value

- Merge(other_state)
  merge’s other’s state



G-Counter CRDT

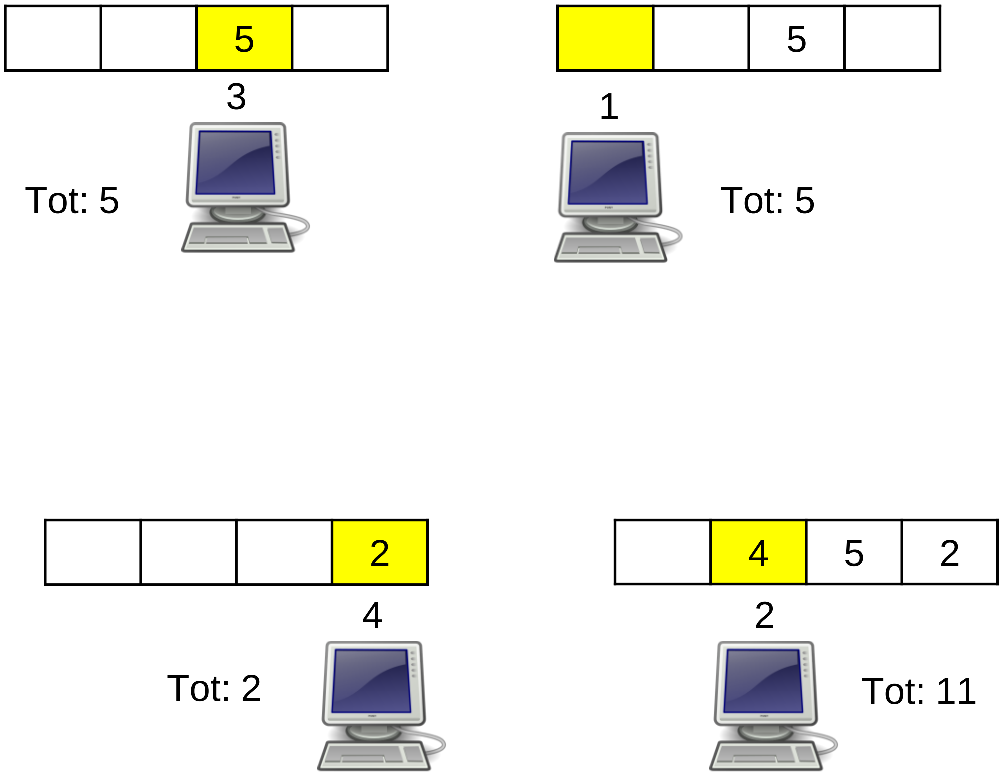
Specifications:

- Grow-only counter,
  replicated across N machines

- Add(x)
  updates our local counter

- Query() returns the value

- Merge(other_state)
  merge's other's state



G-Counter CRDT

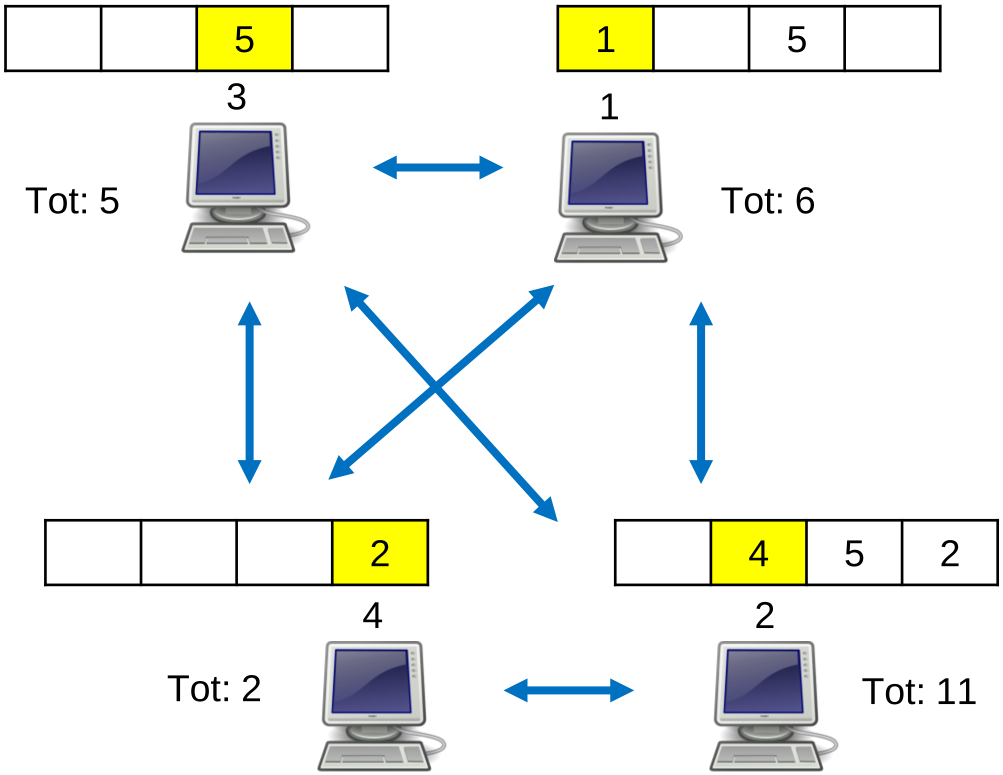
Specifications:

- Grow-only counter,
  replicated across N machines

- Add(x)
  updates our local counter

- Query() returns the value

- Merge(other_state)
  merge's other's state

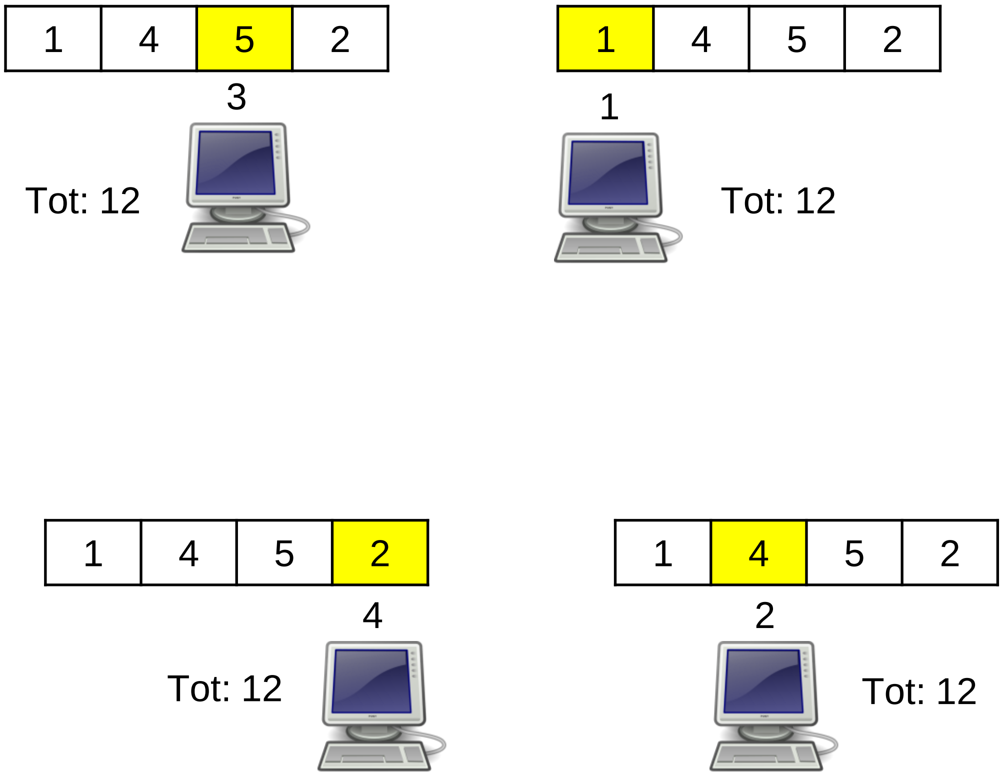


G-Counter CRDT

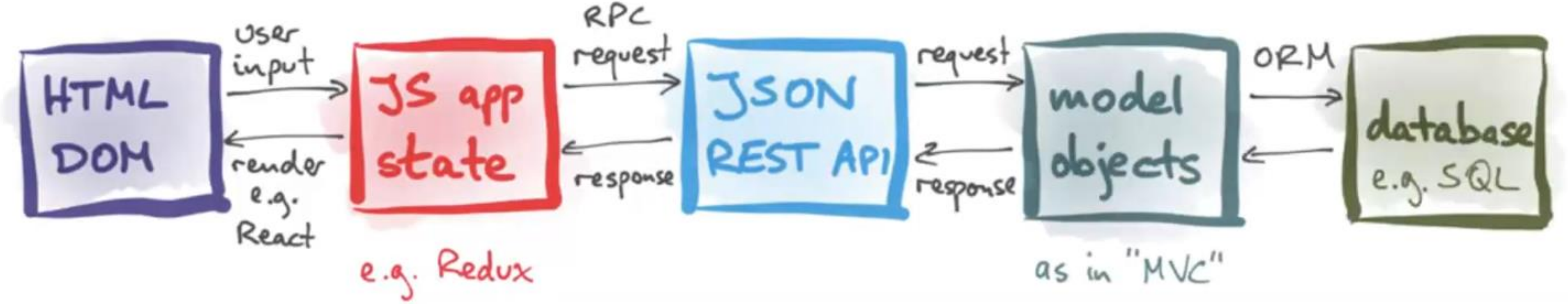
History, as seen locally:

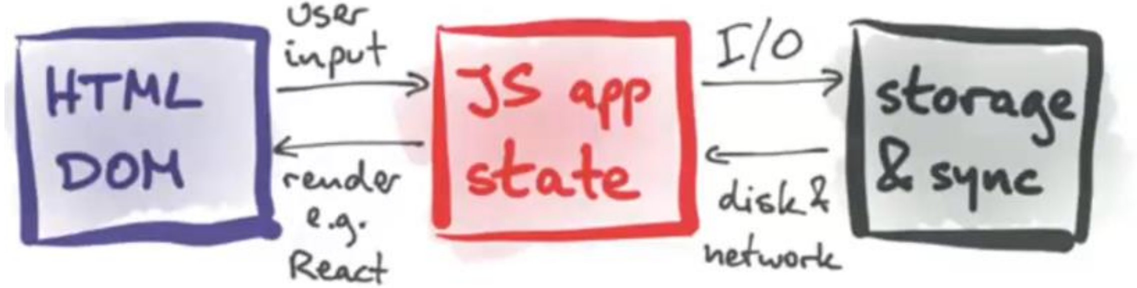
Node 1: 0 → 5 → 6 → 12
Node 2: 0 → 4 → 11 → 12
Node 3: 0 → 5 → 12
Node 4: 0 → 2 → 12

… eventually consistent !



Local-First Software – simpler backends





Strong Consistency ?

- What if we wanted a shared history of the “state” ?

    Google Docs approach:

    → centralize

    → use time stamps

    → does not ensure consistency

- How could we stay distributed (or even decentralized) and be consistent ?
- How could we build the *same, incremental* history of the state ?

    Today’s lecture: Replication and consensus !

Replication and Consensus

Paxos

(Homework 3)

Consistent Data Replication

You know of:

- Redundant Array of Independent Disks (RAID)
- Centralized, distributed databases (Master/slave replication)

Our goal, decentralization:

- No privileged “master”
- Replicated & **consistent** data

→ Hard problem, requires **consensus**

Consensus

- Consensus is agreeing on **one** result
- Once a **majority** agrees on a proposal, that is consensus
- The consensus is **eventually** known by everyone
- Involved parties want to agree on **any** result, not just their own

  ... in the presence of failures

- Types

  Permissioned (today) – known nodes

  Permissionless (week 9 & 10) – anyone 

Single-value Consensus (formally)

We want all nodes ("processes") to agree on a single value

- **Agreement / Safety**

every *correct* process must agree on the same value

- **Termination / Liveness**

eventually, every *correct* process decides some value

- **Integrity / Validity** (weak / strong / ...)

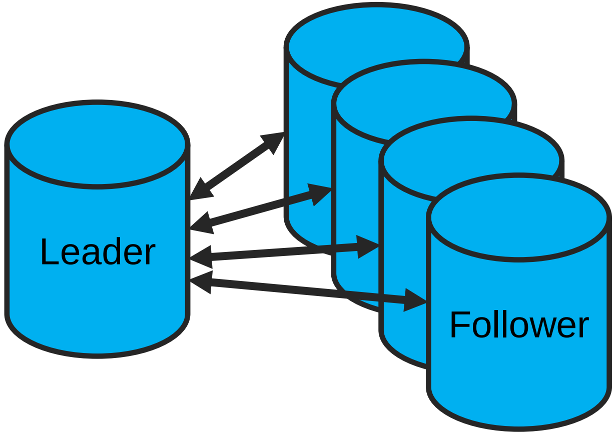
If all *correct* processes proposed value X, then *correct* processes must decide X

If a *correct* process decides X, then X must have been proposed by correct process

- f processes can fail → failure model?

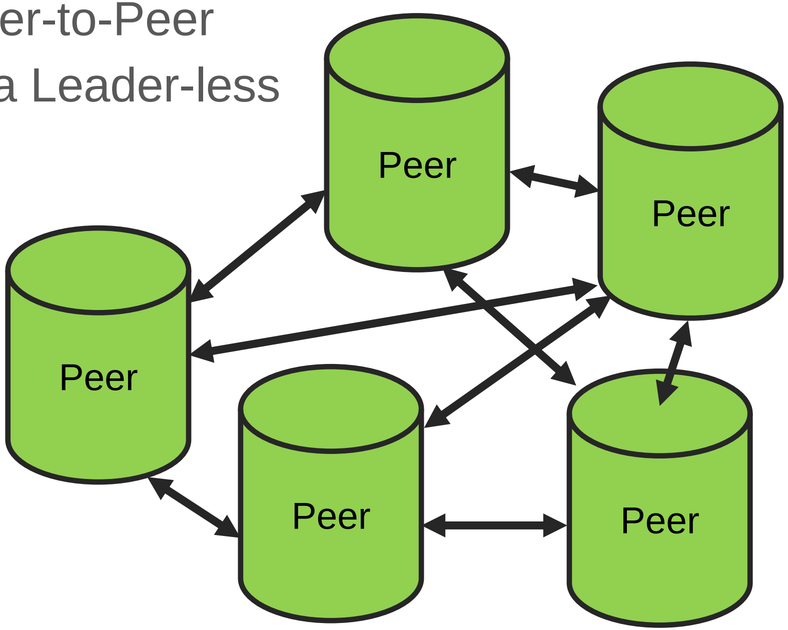
Types of (permissioned) consensus

- Leader-based



- Electing/rejecting leader is tricky, and requires consensus
- “Following” is easy & efficient

- Peer-to-Peer
  aka Leader-less



- Consensus is needed continuously
- No “extra” work when node fails

Building a consensus...

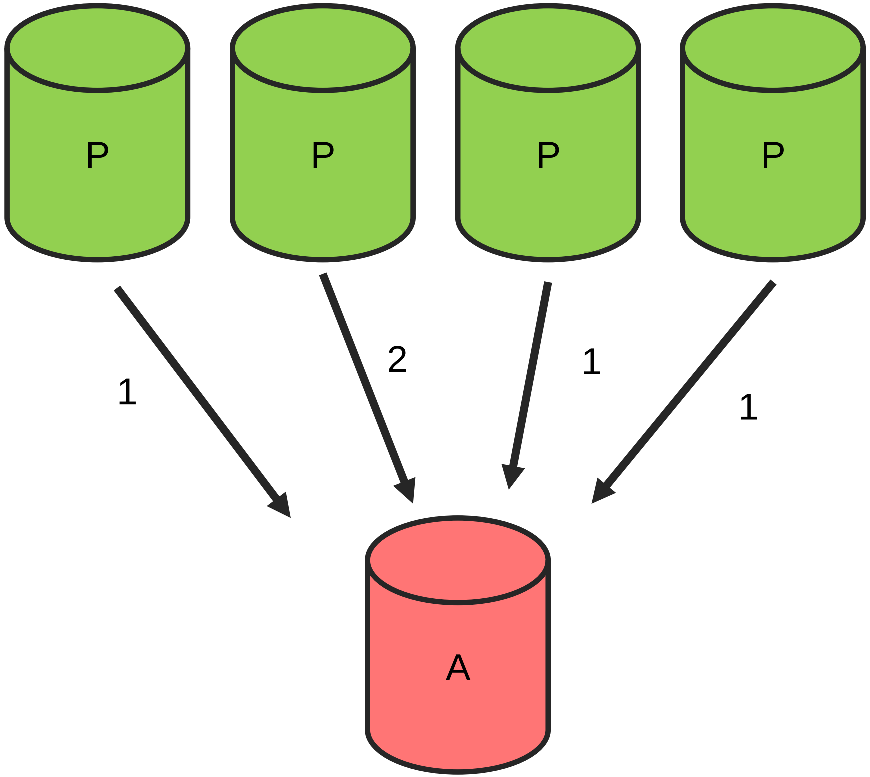
Easy (and wrong) !

- All “proposers” node vote
- One acceptor choses the value

What if the acceptor crashes

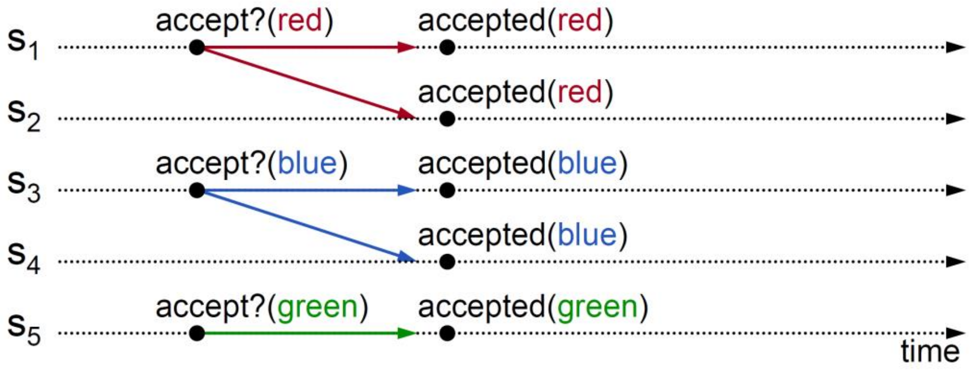
... before choosing ?

... after choosing ?



Building a better consensus...

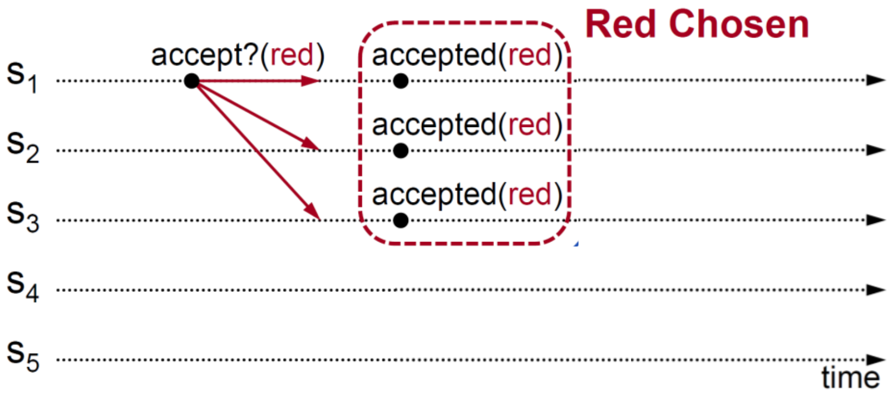
- All “proposers” node vote
- Multiple “acceptors” node
- Value is chosen if accepted by majority



Easy (and still wrong) : split votes !

Building a better consensus ?

- Same as before
- Now, “acceptors” nodes accept **every** value they receive
- Value is chosen if accepted by majority



- We need a **two-phase** protocol !

Paxos

- A family of distributed algorithms for consensus

Three roles:

- Proposers: put forth values to be chosen
- Acceptors: respond to proposers, reach consensus
- Learners: learn the agreed upon value

- Nodes can take any (or even all) roles
- Nodes must know how many acceptors make up a majority
- Nodes must be persistent: they can’t walk back on choices

Paxos phases : intuition

- Prepare phase

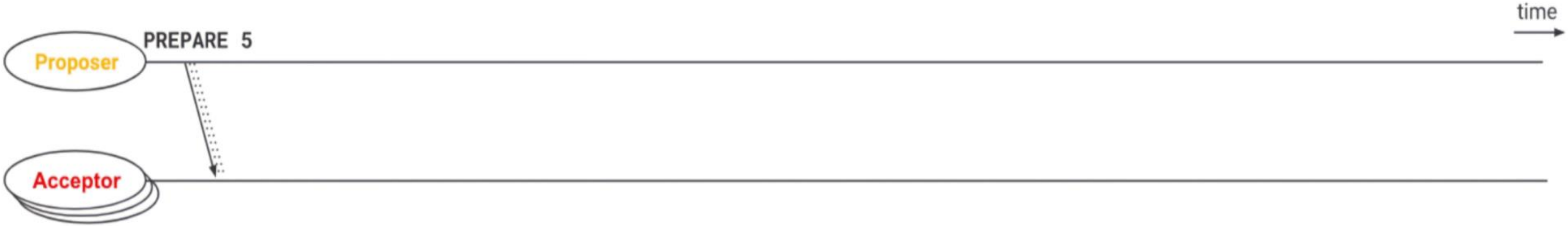
    Proposer: “Will you consider a value I propose ?”
    Each acceptor: “Okay” / “Nope...”

If a majority is obtained:

- Accept phase

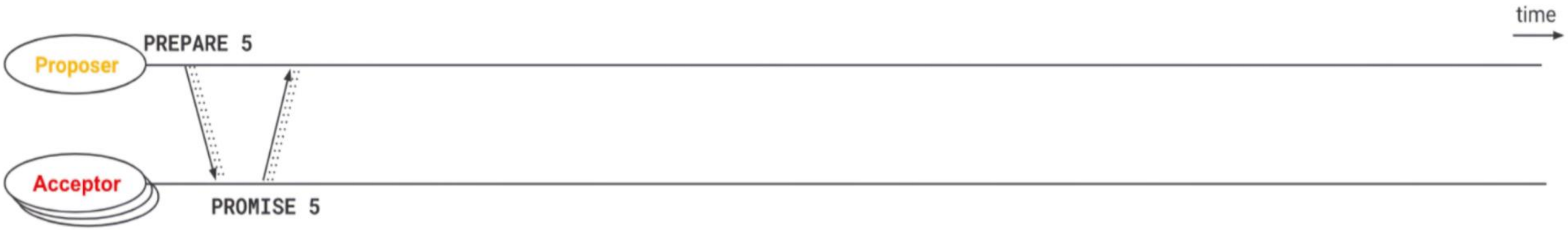
    Proposer: “Here’s my proposed value: X”
    Acceptors: “Okay for X !” / “Nope!”

The Paxos Algorithm



⇨ Proposer wants to propose a certain value:
It sends **PREPARE** **<u>IDp</u>** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

The Paxos Algorithm

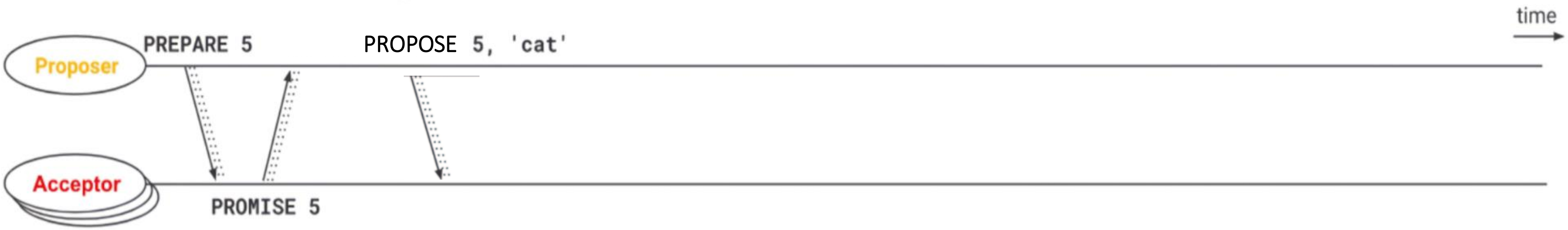


⇨ Proposer wants to propose a certain value:
It sends **PREPARE IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
(?) Reply with **PROMISE IDp**.

If a majority of acceptors promise, no ID<IDp can make it through.

The Paxos Algorithm



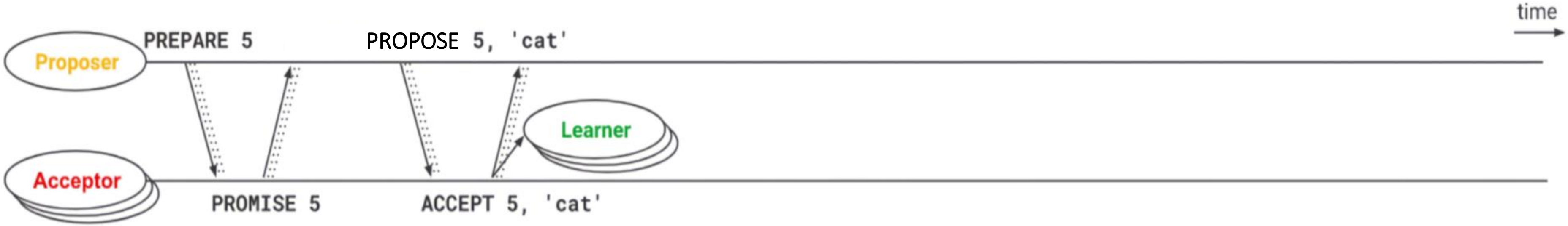
⇨ Proposer wants to propose a certain value:
It sends **PREPARE IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
(?) Reply with **PROMISE IDp**.

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **IDp, VALUE** to a majority (or all) of **Acceptors**.
(?) It picks any value it wants.

If a majority of acceptors promise, no ID<IDp can make it through.

The Paxos Algorithm



⇨ **Proposer** wants to propose a certain value:
It sends **PREPARE** <u>**IDp**</u> to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. **Proposer** 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
(?) Reply with **PROMISE** <u>**IDp**</u>.

⇨ **Proposer** gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE <u>IDp</u>, <u>VALUE</u> to a majority (or all) of **Acceptors**.
(?) It picks any value it wants.

⇨ **Acceptor** receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT** <u>**IDp**</u>, <u>**value**</u>. Also send it to all **Learners**.

If a majority of acceptors promise, no ID<IDp can make it through.

The Paxos Algorithm

⇨ Proposer wants to propose a certain value:
It sends **PREPARE** **IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
(?) Reply with **PROMISE** **IDp**.

If a majority of acceptors promise, no ID<IDp can make it through.

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **IDp, VALUE** to a majority (or all) of **Acceptors**.
(?) It picks any value it wants.

⇨ **Acceptor** receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT** **IDp, value**. Also send it to all **Learners**.
If a majority of acceptors accept IDp, value, consensus is reached.
Consensus is and will always be on value (not necessarily IDp).

The Paxos Algorithm

⇨ Proposer wants to propose a certain value:
It sends **PREPARE IDp** to a majority (or all) of Acceptors.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ Acceptor receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
(?) Reply with **PROMISE IDp**.

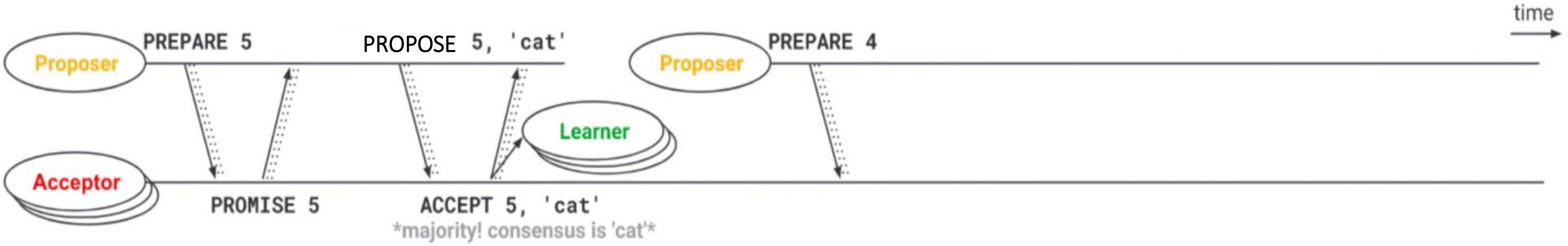
If a majority of acceptors promise, no ID<IDp can make it through.

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE IDp, VALUE to a majority (or all) of Acceptors.
(?) It picks any value it wants.

⇨ Acceptor receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT IDp, value**. Also send it to all Learners.
If a majority of acceptors accept IDp, value, consensus is reached.
Consensus is and will always be on value (not necessarily IDp).

⇨ Proposer or Learner get **ACCEPT** messages for IDp, value:
If a proposer/learner gets majority of accept for a specific IDp,
they know that consensus has been reached on value (not IDp).

The Paxos Algorithm



⇨ Proposer wants to propose a certain value:
It sends **PREPARE IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
(?) Reply with **PROMISE IDp**.

1 If a majority of acceptors promise, no ID<IDp can make it through.

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **IDp, VALUE** to a majority (or all) of **Acceptors**.
(?) It picks any value it wants.

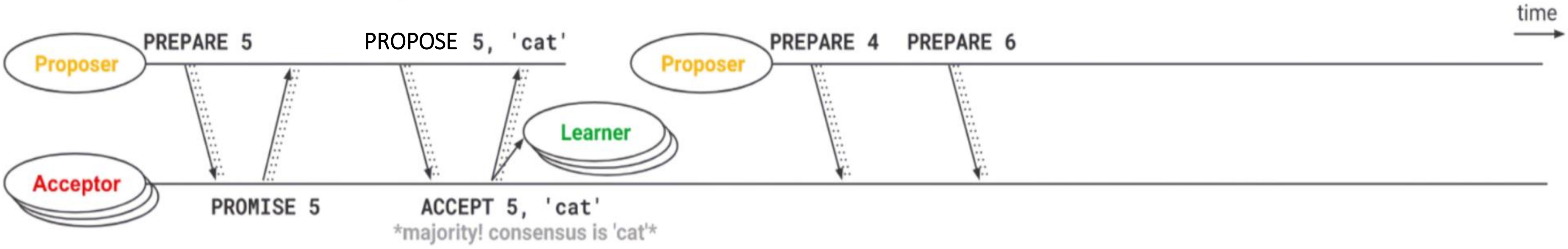
⇨ **Acceptor** receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT IDp, value**. Also send it to all **Learners**.

2 If a majority of acceptors accept IDp, value, consensus is reached. Consensus is and will always be on value (not necessarily IDp).

⇨ Proposer or **Learner** get **ACCEPT** messages for IDp, value:

3 If a proposer/learner gets majority of accept for a specific IDp, they know that consensus has been reached on value (not IDp).

The Paxos Algorithm



⇨ Proposer wants to propose a certain value:
It sends **PREPARE** **IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE** **IDp** **accepted** **IDa**, **value**.
No -> Reply with **PROMISE** **IDp**.

1 If a majority of acceptors promise, no ID<IDp can make it through.

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **IDp**, **VALUE** to a majority (or all) of **Acceptors**.
(?) It picks any value it wants.

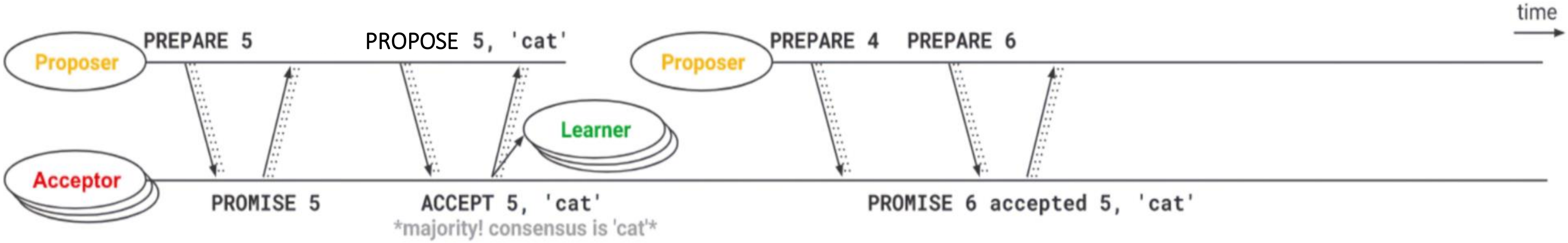
⇨ **Acceptor** receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT** **IDp**, **value**. Also send it to all **Learners**.

2 If a majority of acceptors accept IDp, value, consensus is reached.
Consensus is and will always be on value (not necessarily IDp).

⇨ Proposer or **Learner** get **ACCEPT** messages for IDp, value:

3 If a proposer/learner gets majority of accept for a specific IDp, they know that consensus has been reached on value (not IDp).

The Paxos Algorithm



⇨ Proposer wants to propose a certain value:
It sends **PREPARE IDp** to a majority (or all) of Acceptors.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ Acceptor receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE IDp accepted IDa, value**.
No -> Reply with **PROMISE IDp**.

1 **If a majority of acceptors promise, no ID<IDp can make it through.**

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **IDp, VALUE** to a majority (or all) of Acceptors.
Has it got any already accepted value from promises?
Yes - > It picks the value with the highest **IDa** that it got.
No -> It picks any value it wants.

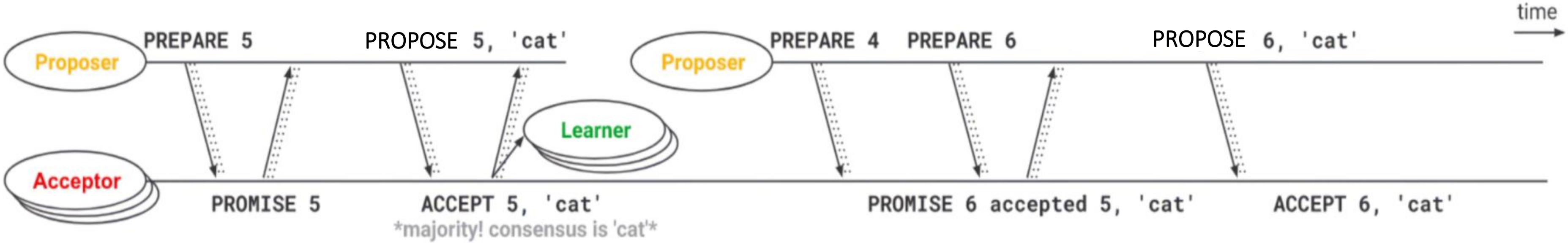
⇨ Acceptor receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT IDp, value**. Also send it to all Learners.

2 **If a majority of acceptors accept IDp, value, consensus is reached. Consensus is and will always be on value (not necessarily IDp).**

⇨ Proposer or Learner get **ACCEPT** messages for IDp, value:

3 **If a proposer/learner gets majority of accept for a specific IDp, they know that consensus has been reached on value (not IDp).**

The Paxos Algorithm



⇨ Proposer wants to propose a certain value:
It sends **PREPARE** **IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE** **IDp** **accepted** **IDa**, **value**.
No -> Reply with **PROMISE** **IDp**.

1 If a majority of acceptors promise, no ID<IDp can make it through.

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **IDp**, **VALUE** to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes - > It picks the value with the highest **IDa** that it got.
No -> It picks any value it wants.

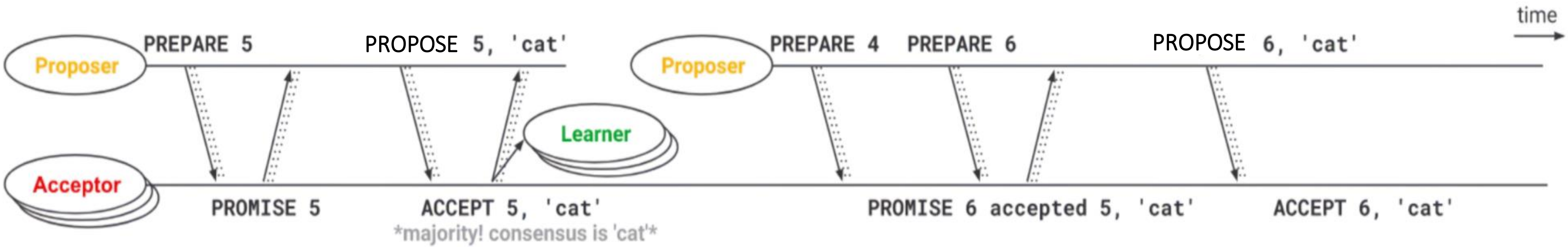
⇨ **Acceptor** receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT** **IDp**, **value**. Also send it to all **Learners**.

2 If a majority of acceptors accept IDp, value, consensus is reached. Consensus is and will always be on value (not necessarily IDp).

⇨ Proposer or Learner get **ACCEPT** messages for IDp, value:

3 If a proposer/learner gets majority of accept for a specific IDp, they know that consensus has been reached on value (not IDp).

The Paxos Algorithm



⇨ **Proposer** wants to propose a certain value:
It sends **PREPARE <u>IDp</u>** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. **Proposer** 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE <u>IDp</u> accepted <u>IDa</u>, <u>value</u>**.
No -> Reply with **PROMISE <u>IDp</u>**.

1 **If a majority of acceptors promise, no ID<IDp can make it through.**

⇨ **Proposer** gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **<u>IDp</u>, <u>VALUE</u>** to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes - > It picks the value with the highest **IDa** that it got.
No -> It picks any value it wants.

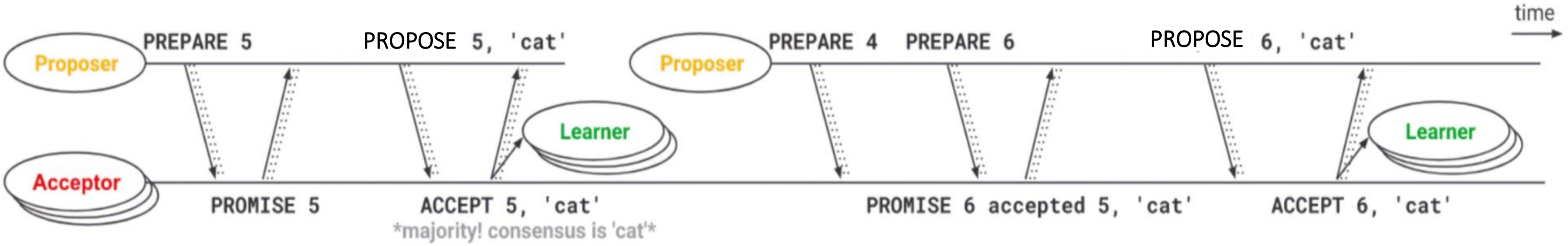
⇨ **Acceptor** receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT <u>IDp</u>, <u>value</u>**. Also send it to all **Learners**.

2 **If a majority of acceptors accept <u>IDp</u>, <u>value</u>, consensus is reached.**
Consensus is and will always be on <u>value</u> (not necessarily IDp).

⇨ **Proposer** or **Learner** get **ACCEPT** messages for IDp, value:

3 **If a proposer/learner gets majority of accept for a specific <u>IDp</u>,**
they know that consensus has been reached on <u>value</u> (not IDp).

The Paxos Algorithm



⇨ Proposer wants to propose a certain value:
It sends **PREPARE <u>IDp</u>** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. Proposer 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE <u>IDp</u> accepted <u>IDa</u>, <u>value</u>**.
No -> Reply with **PROMISE <u>IDp</u>**.

1 **If a majority of acceptors promise, no ID<IDp can make it through.**

⇨ Proposer gets majority of **PROMISE** messages for a specific IDp:
It sends PROPOSE **<u>IDp</u>, <u>VALUE</u>** to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes - > It picks the value with the highest **IDa** that it got.
No -> It picks any value it wants.

⇨ **Acceptor** receives an PROPOSE message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT <u>IDp</u>, <u>value</u>**. Also send it to all **Learners**.

2 **If a majority of acceptors accept <u>IDp</u>, <u>value</u>, consensus is reached.**
Consensus is and will always be on <u>value</u> (not necessarily IDp).

⇨ Proposer or **Learner** get **ACCEPT** messages for IDp, value:

3 **If a proposer/learner gets majority of accept for a specific <u>IDp</u>, they know that consensus has been reached on <u>value</u> (not IDp).**

Paxos Challenges

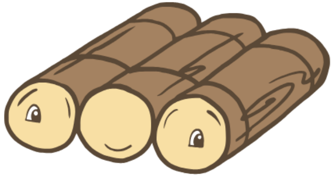
- Contention

- Non crash-stop behaviour
  - Asynchrony
  - Byzantine faults

- (In)efficiency of simple Paxos
  - Introducing a leader
  - Protocol “Quick wins”

- Choosing multiple, subsequent values (e.g. Multi-Paxos)

The Raft Consensus Algorithm



- Designed to be easy to understand
- Functionally equivalent to Paxos
- Easier to implement (claim)
- Widely used in the industry

    MongoDB, CockroachDB

    Etcd, Neo4j, RabbitMQ

    ...