

Paxos

October 2026

Abstract

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

Paxos Made Simple

Leslie Lamport

01 Nov 2001

1 Introduction

The Paxos algorithm for implementing a fault-tolerant distributed system has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers [5]. In fact, it is among the simplest and most obvious of distributed algorithms. At its heart is a consensus algorithm—the “synod” algorithm of [5]. The next section shows that this consensus algorithm follows almost unavoidably from the properties we want it to satisfy. The last section explains the complete Paxos algorithm, which is obtained by the straightforward application of consensus to the state machine approach for building a distributed system—an approach that should be well-known, since it is the subject of what is probably the most often-cited article on the theory of distributed systems [4].

Paxos is all about consensus

Consensus

Given a set of processors, each with an initial value:

- **Termination:** All non-faulty processes eventually decide on a value
- **Agreement:** All processes that decide do so on the same value
- **Validity:** Value decided must have proposed by some process

Wrong Solutions to Consensus

- 1 leader that accepts proposals

Wrong Solutions to Consensus

- **1 leader that accepts proposals**
 - **Can't progress when the leader dies!**

Wrong Solutions to Consensus

- **1 leader that accepts proposals**
 - Can't progress when the leader dies!
- **Everyone can accept proposals, wait for a majority**

Wrong Solutions to Consensus

- **1 leader that accepts proposals**
 - Can't progress when the leader dies!
- **Everyone can accept proposals, wait for a majority**
 - Fails to progress when there's lots of proposals!

Invariants

- **All proposals are have a unique identifier**
 - Strictly increasing integer + proposer ID like 1a for machine a
- **A proposal (i, v) has a value equal to the proposal with the largest identifier < i**
 - Avoids the second problem of conflicting values

Pseudocode

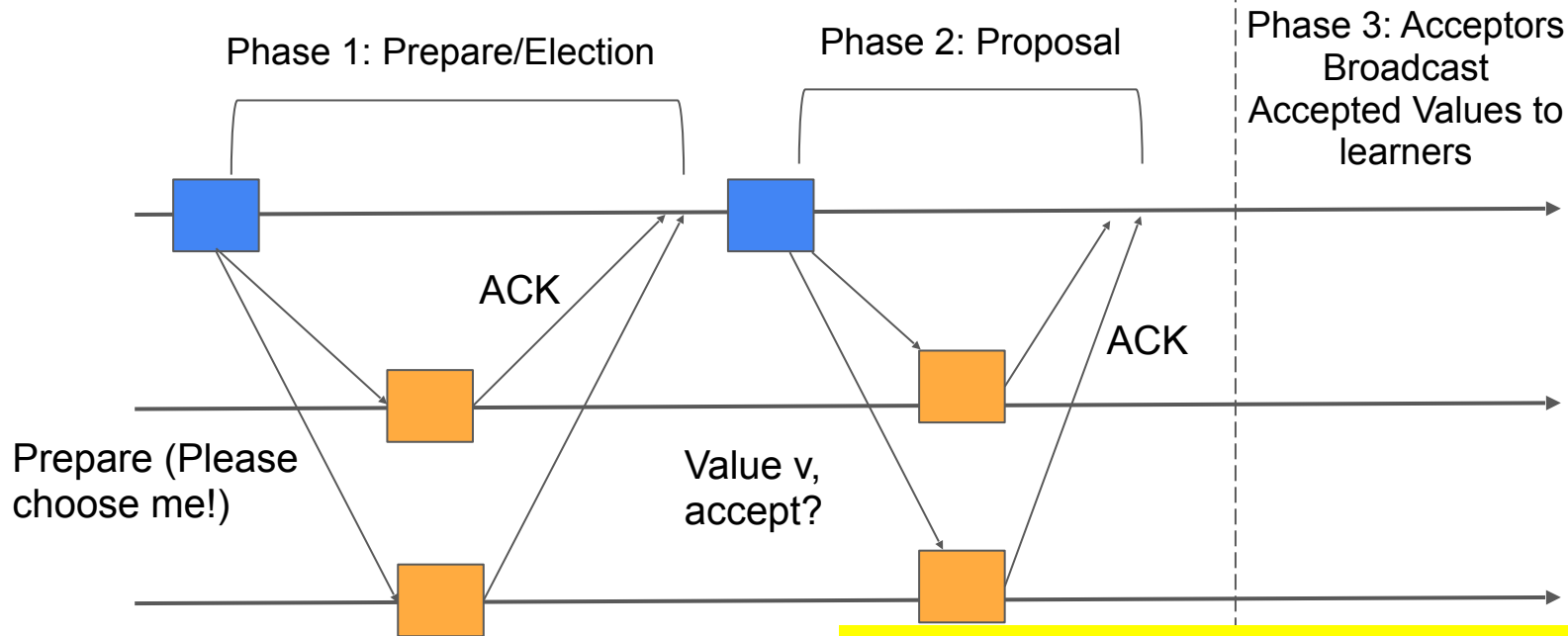
PROPOSER:

- p1** $i \leftarrow$ a fresh id, larger than any seen
send `prepare(i)` to all acceptors
- p2** wait for `promise(i, accepted)` from a majority
- p3** if a promise carries an accepted (i', v') :
 $v \leftarrow$ the v' with the largest i'
 else $v \leftarrow v_0$
- p4** send `accept(i, v)` to all acceptors
- p5** wait for `accepted(i)` from a majority
 $\rightarrow (i, v)$ is chosen

ACCEPTOR:

- a1** on `prepare(i)`: if $i \geq$ promised:
 promised $\leftarrow i$; reply `promise(i, accepted)`
 else reply `reject`
- a2** on `accept(i, v)`: if $i \geq$ promised:
 promised $\leftarrow i$; accepted $\leftarrow (i, v)$
 reply `accepted(i)`
 else reply `reject`

Review of Paxos [High-Level]



Proposer

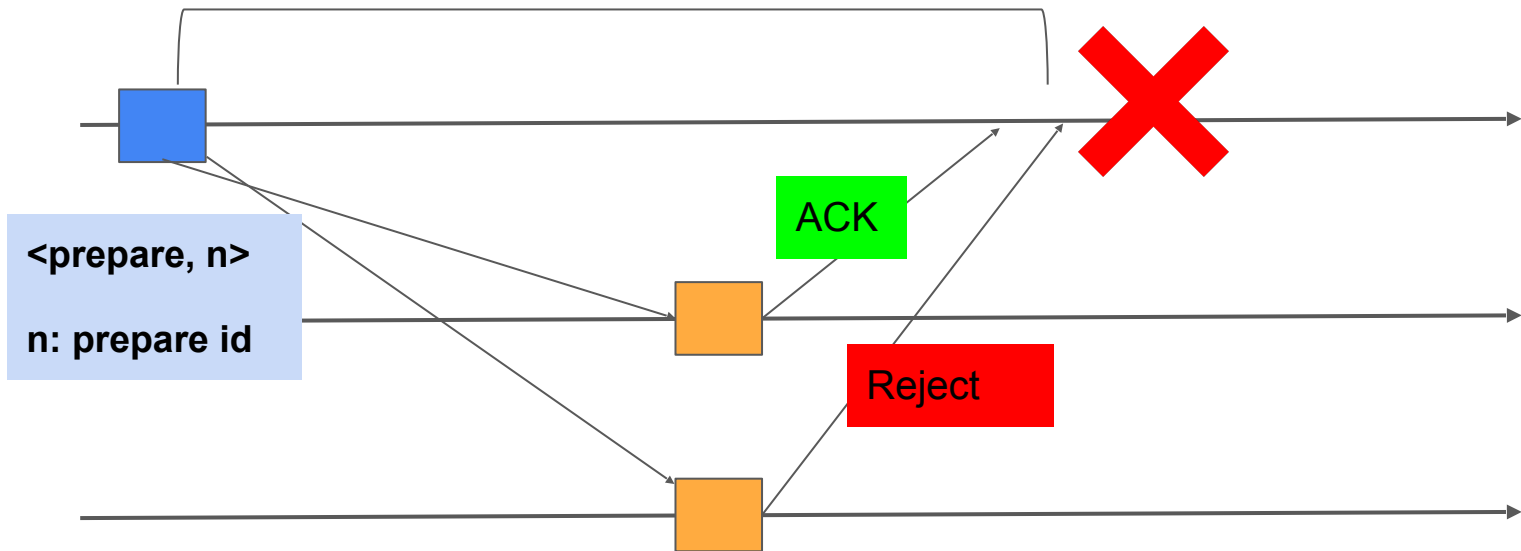


Acceptor

Any node can be a learner. The Phase 2 ACK to the proposer can double as the acceptors' broadcast to learners if we assume that the proposer also acts as a learner.

Review of Paxos [Detailed]

Phase 1: Prepare/Election



$n < n_{\text{highest}}$

This acceptor has seen a prepare message with a higher prepare_id



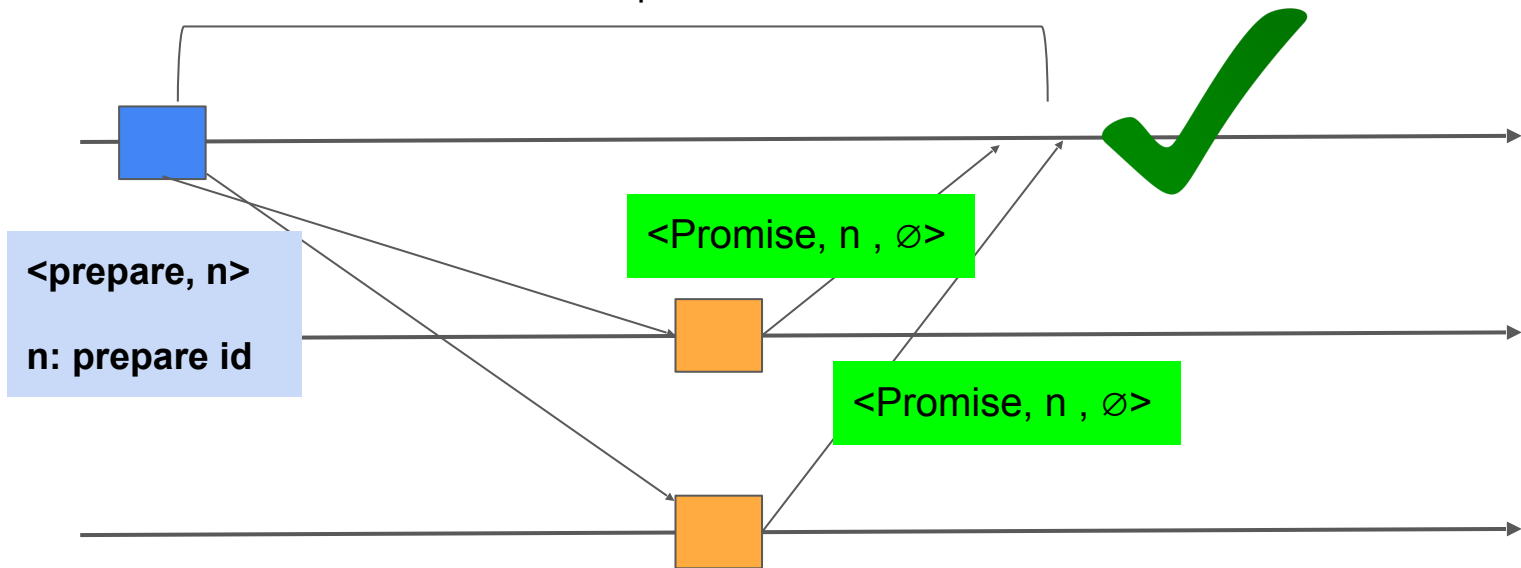
Proposer



Acceptor

Review of Paxos [Detailed]

Phase 1: Prepare/Election



Both acceptors accept and update their **n_highest = n**
Assumption: None of them had accepted other proposals before



Proposer

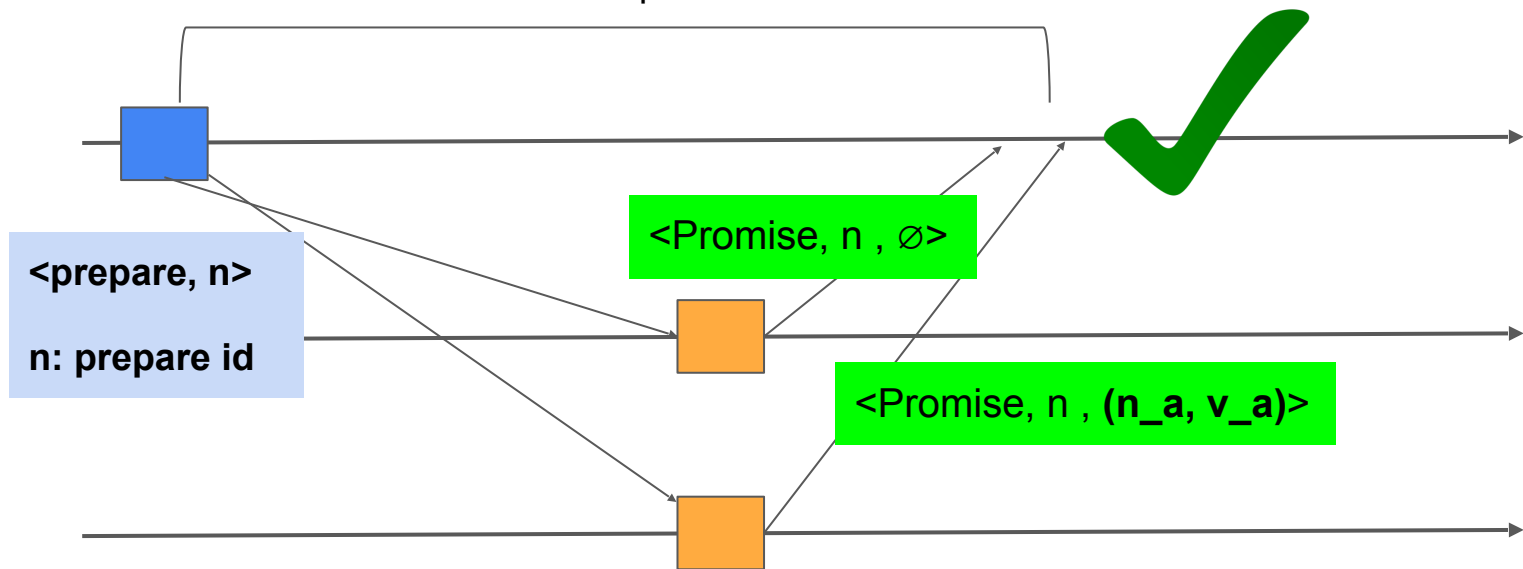


Acceptor

Review of Paxos [Detailed]

So here the proposer has to retry with the v_a , instead of its own proposed value [**Do not race, just complete the duty of the previous proposer**]

Phase 1: Prepare/Election



Both acceptors accept and update their **n_highest = n**
Assumption: One of them accepted an older proposal before [the proposal made by n_a prepare round]



Proposer



Acceptor

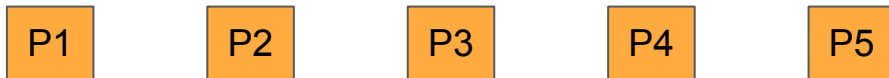
Review of Paxos [Detailed]

Paxos Phase 2

- **Proposer:**
 - If receive promise from **majority** of acceptors,
 - Determine v_a returned with highest n_a , if exists
 - Send $\langle \text{accept}, (n, v_a \parallel v) \rangle$ to acceptors
- **Acceptors:**
 - Upon receiving (n, v) , if $n \geq n_h$,
 - Accept proposal and notify learner(s)

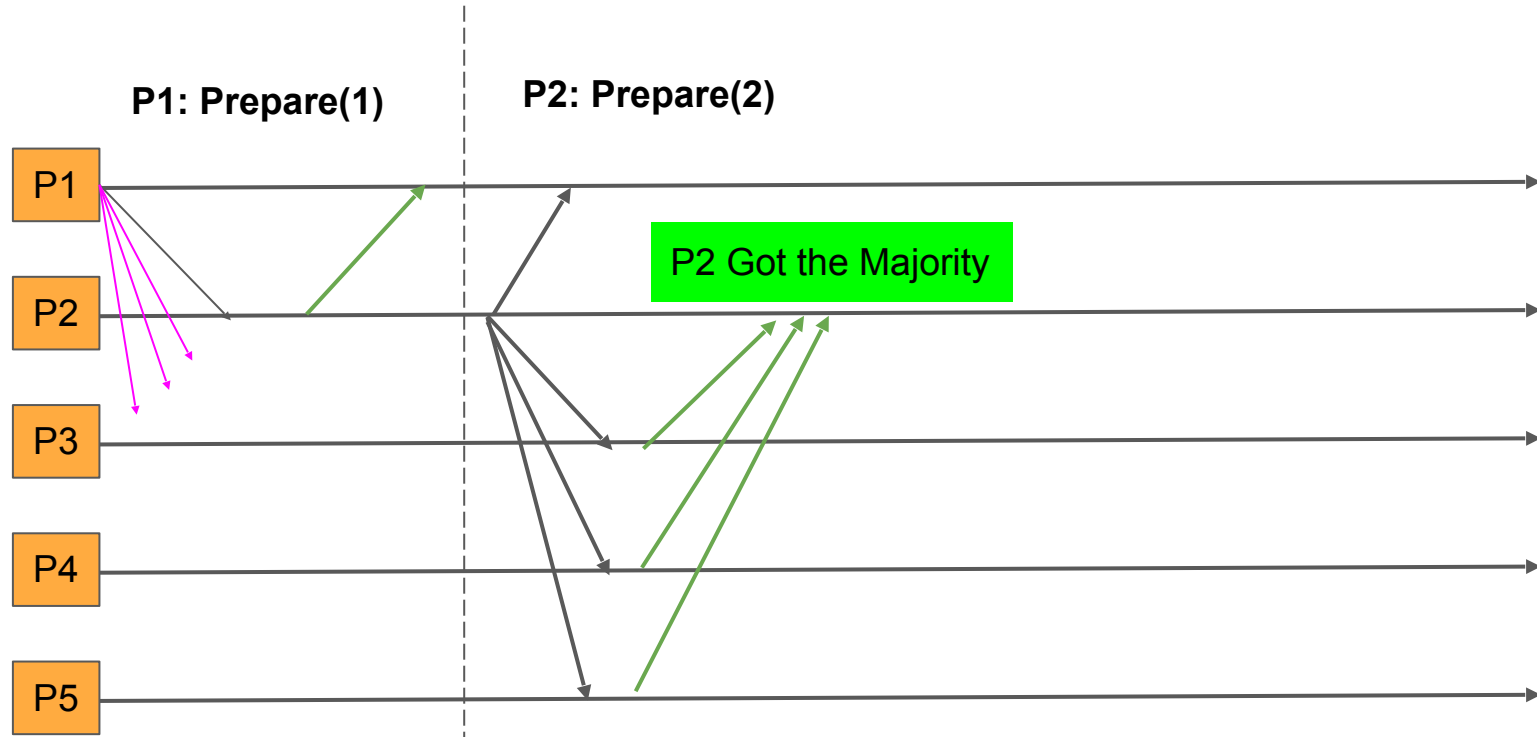
Let's practice

- In this example, we have 5 processes. **All of them are learners.**
- All of them might become proposers and acceptors at different times.



Each prepare/proposal needs at least $(5+1)/2 = 3$ ACK [including itself]
Green Arrows are ACK and Red ones are REJECT.

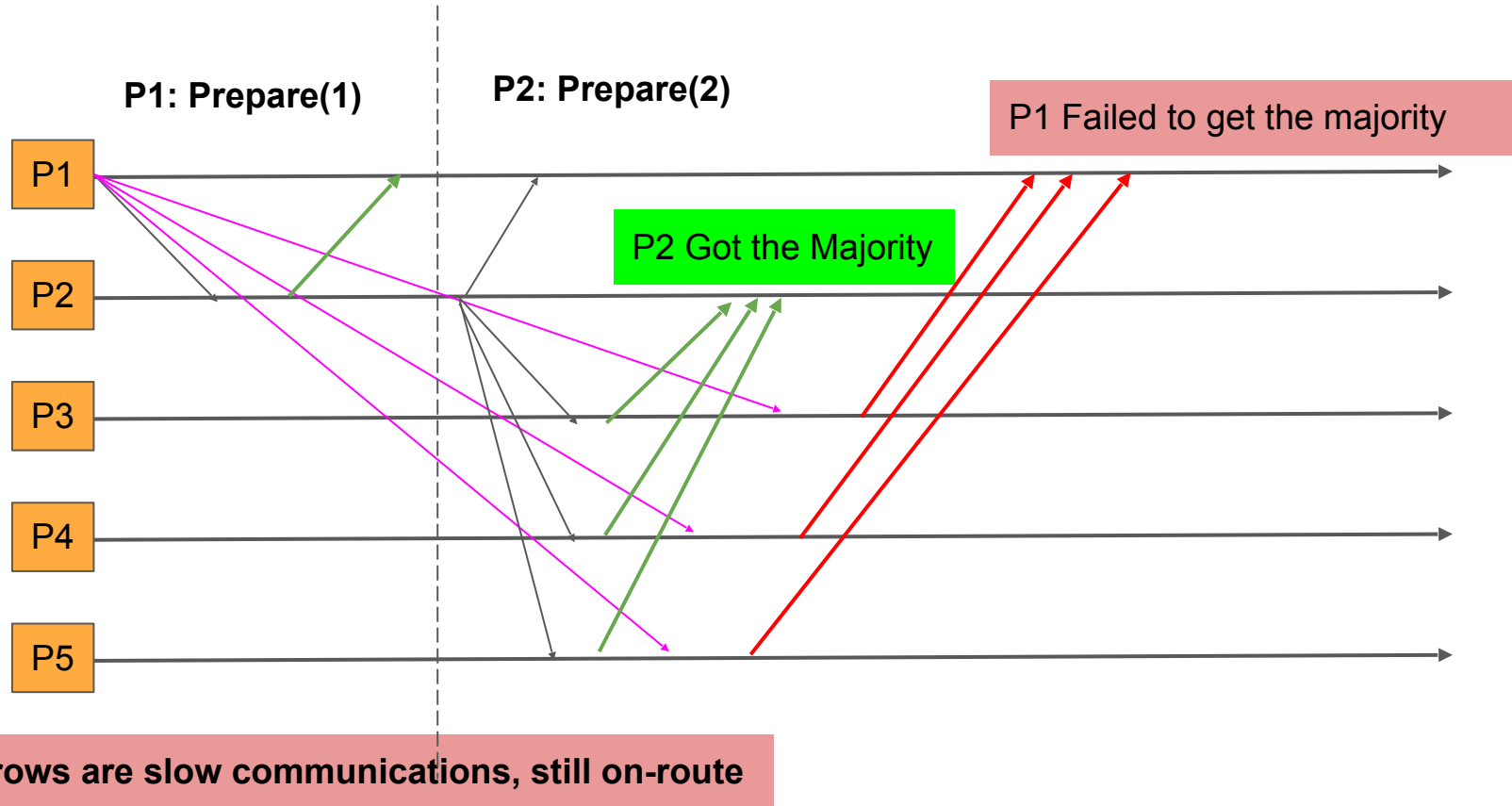
Let's practice: Failed Prepare



Pink Arrows are slow communications, still on-route

Each prepare/proposal needs at least $(5+1)/2 = 3$ ACK [including itself]
Green Arrows are ACK and Red ones are REJECT.

Let's practice: Failed Prepare

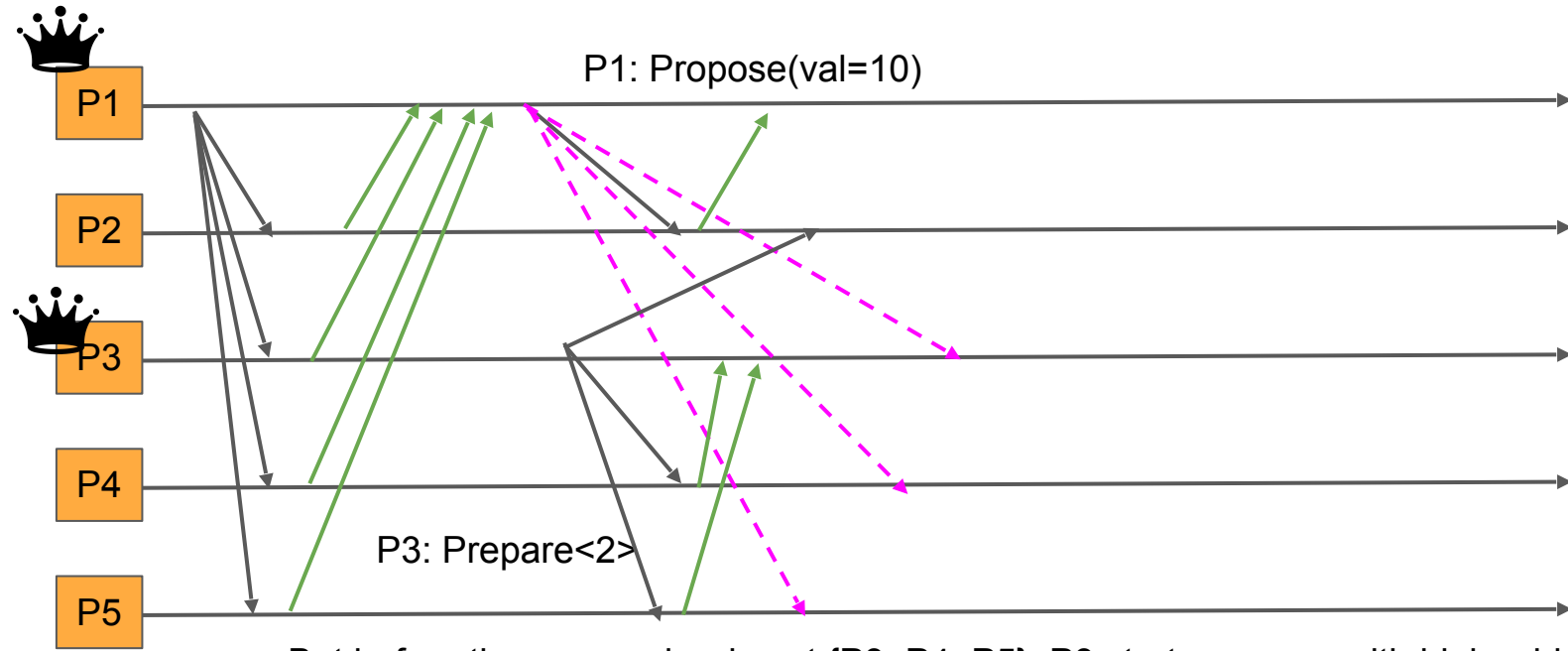


Let's practice: Racing



Successful Prepare Phase; Now P1 can propose!

Let's practice: Racing



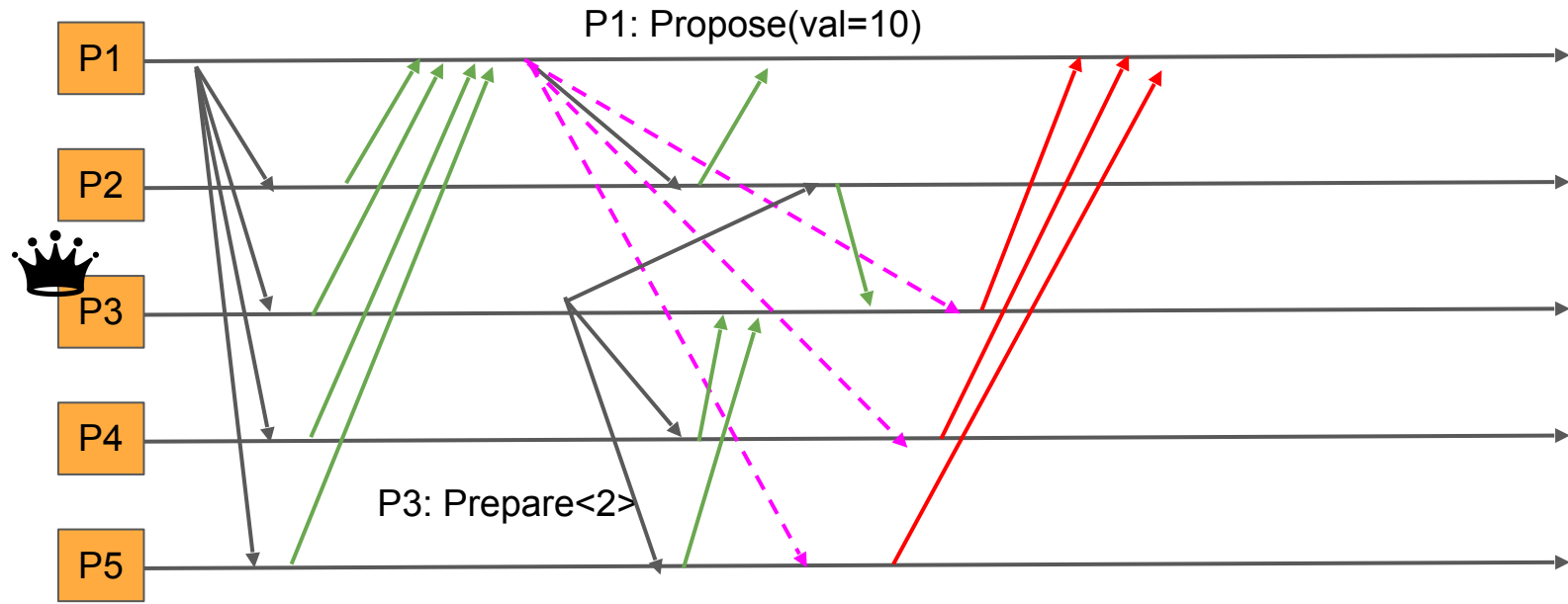
But before the proposal arrive at {P3, P4, P5}, P3 starts prepare with higher id

And {P4, P5} accepts the new prepare! So P3 wins the prepare too.

Some of the late arrows that will arrive in future and get rejected are not shown for readability.

Let's practice: Racing

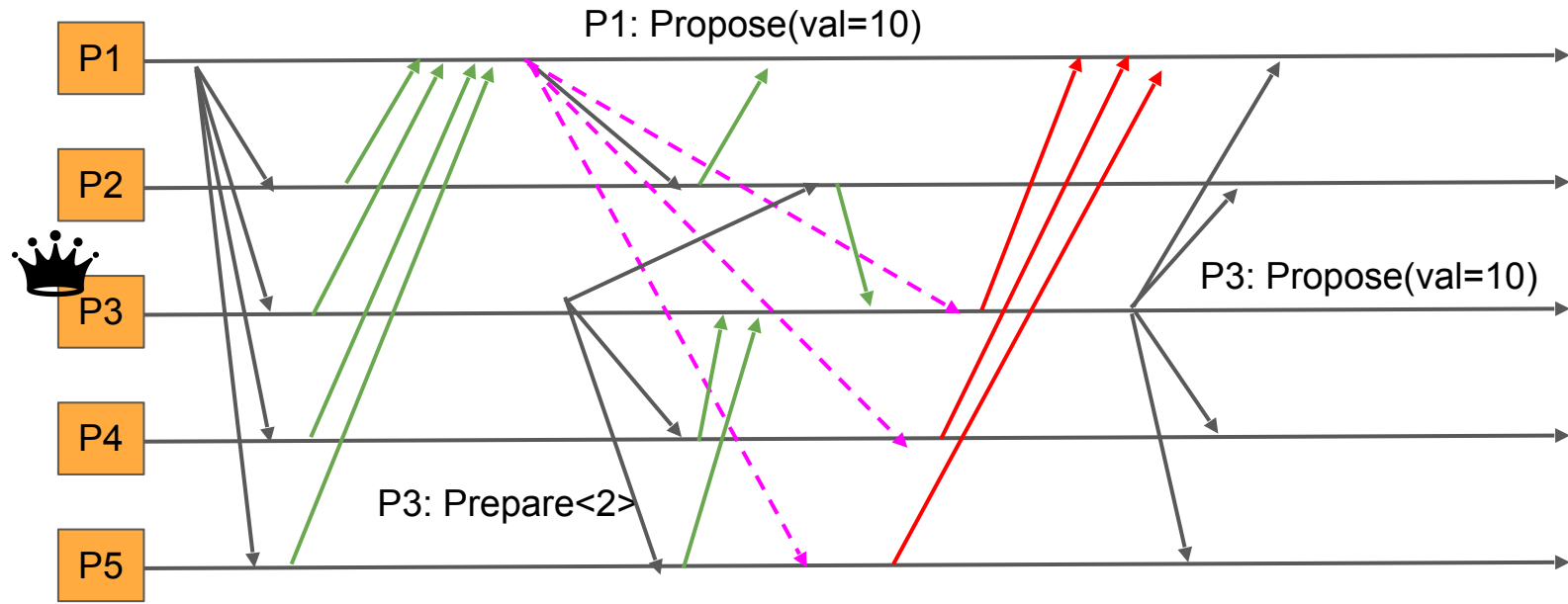
The old proposal value of P1 is now declined, because there was a higher new prepare.



P2 also accepts the prepare from P3, but sends the previously accepted value 10 to P3

Some of the late arrows that will arrive in future and get rejected are now shown for readability.

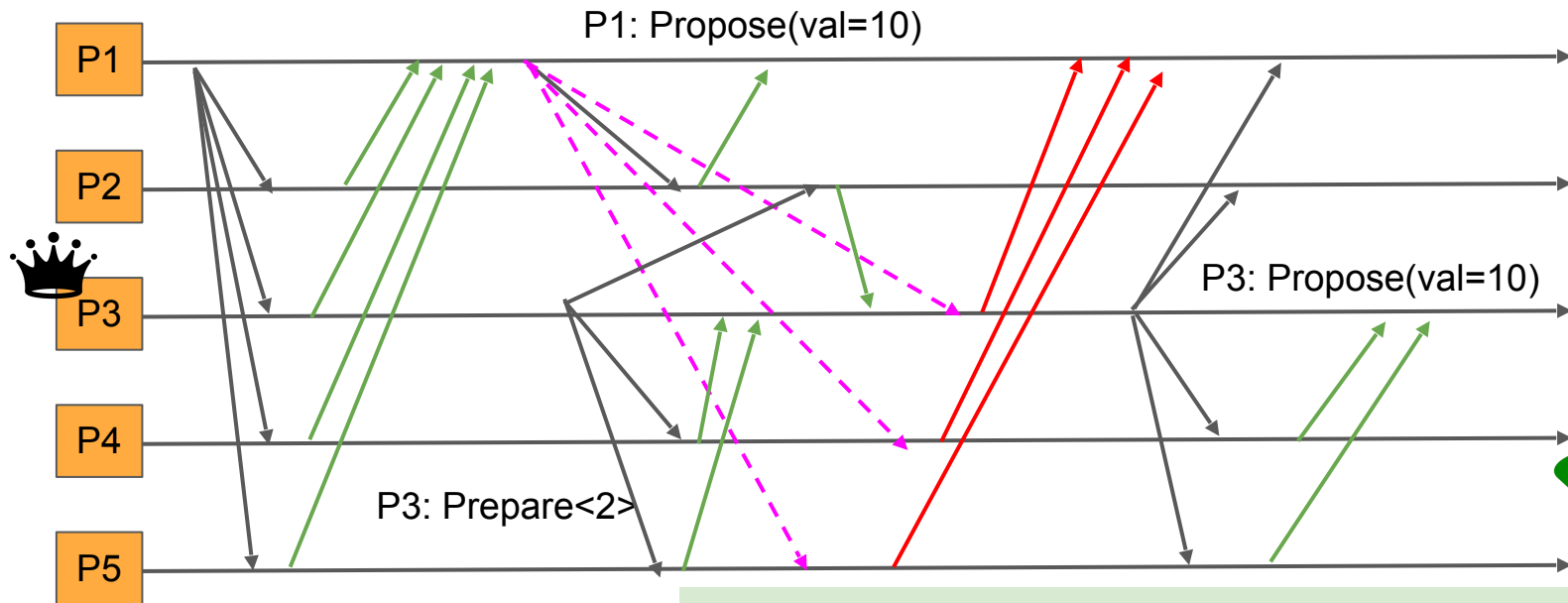
Let's practice: Racing



P3 starts proposal with the same value=10!

Some of the late arrows that will arrive in future and get rejected are now shown for readability.

Let's practice: Racing

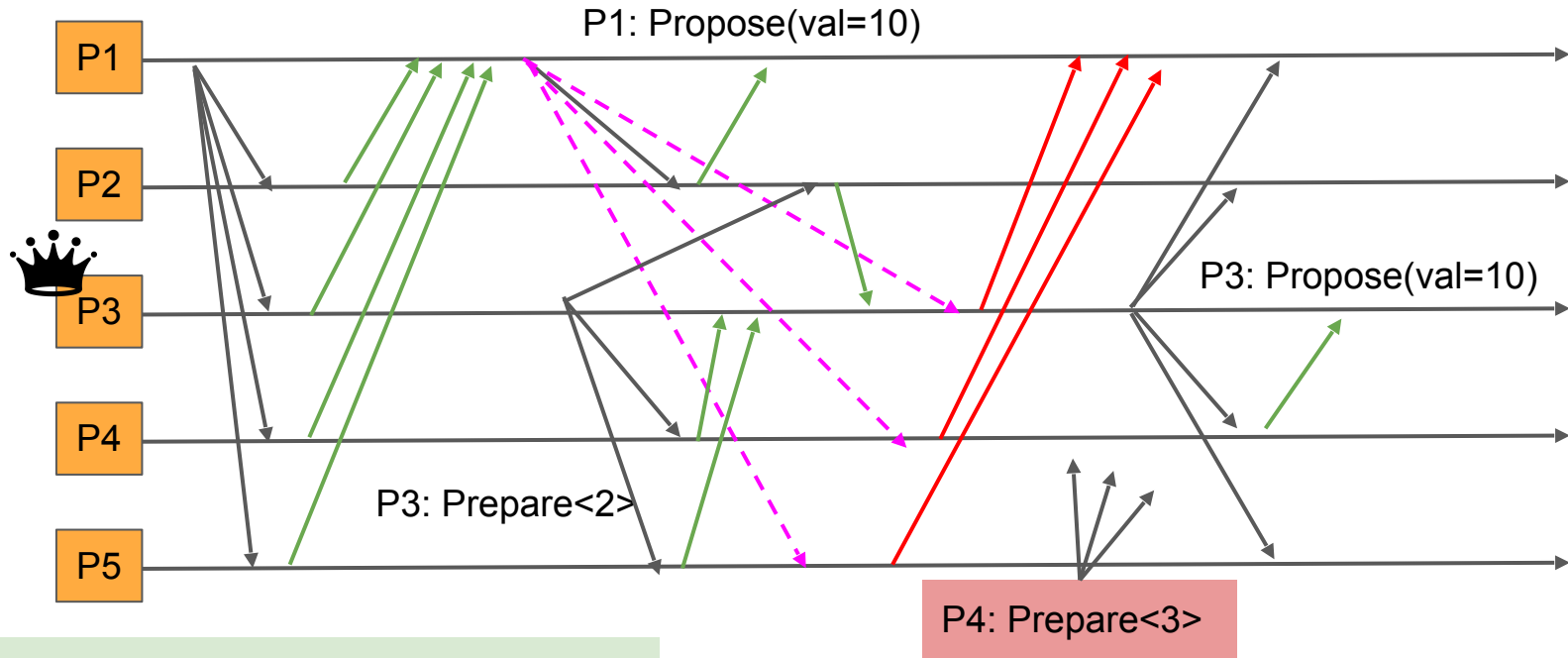


P3 starts proposal with the same value=10!

If at least 2 other processes ACK, we're all good! 10 is decided and P3's proposal is complete

But what if (similar to P1's proposal), some process send `<prepare>` with a higher id before at least 2 processes ACK to P3's proposal?

Let's practice: Racing



P3 starts proposal with the same value=10!

This can repeat indefinitely: before any proposal round completes, another process

- starts a new one,
- wins the majority in the prepare phase, and
- causes previous proposals to be rejected.
- The new leader then tries to finish the proposal (with the old value), but is interrupted again — and the cycle continues.

Takeaway

What we saw was safe!

Whenever an accept happens it respects previously accepted values.

But liveness failed

- Because two proposers continually **preempt** each other
- Accept phases never gather a majority of accepts.
- Higher proposal numbers keep **invalidating earlier attempts**, so *no progress* is made until one proposer stops or the network schedules deliver differently.
- In practice people inject artificial delays between processes so they don't race concurrently and there is some delay!