System Implementation Strategies + Paxos

October 2025

Lindsey Kuper (@lindsey@recurse.social)
@lindsey

"Oh, you wanted to *increment a counter*?! Good luck with that!" -- the distributed systems literature

2:55 PM · Mar 9, 2015

358 Reposts **15** Quotes **714** Likes **23** Bookmarks

23

Overview

- Successful System Implementation Strategies
  - Understand the Concepts and Code Structure
  - Iterative Design Process
  - Modular Programming
  - Tips on Debugging
- Paxos

Understanding Concepts and Code Structure

Understand the Concept and Code Structure

- What is the conceptual system you want to build? **Concept**
  - Understand the concept and verify your knowledge with some examples
  - Rewrite the algorithm to some pseudocode, which can serve as the guide during actual programming
- How is the system physically built? **Build**
  - Read the skeleton code
  - Map the algorithms/concepts to the given code structure
  - Draw flow charts to understand the code flow
- How to use the system? **Usage**
  - Read the testing script to see how an external user will talk to our system and invoke its APIs to accomplish desired tasks

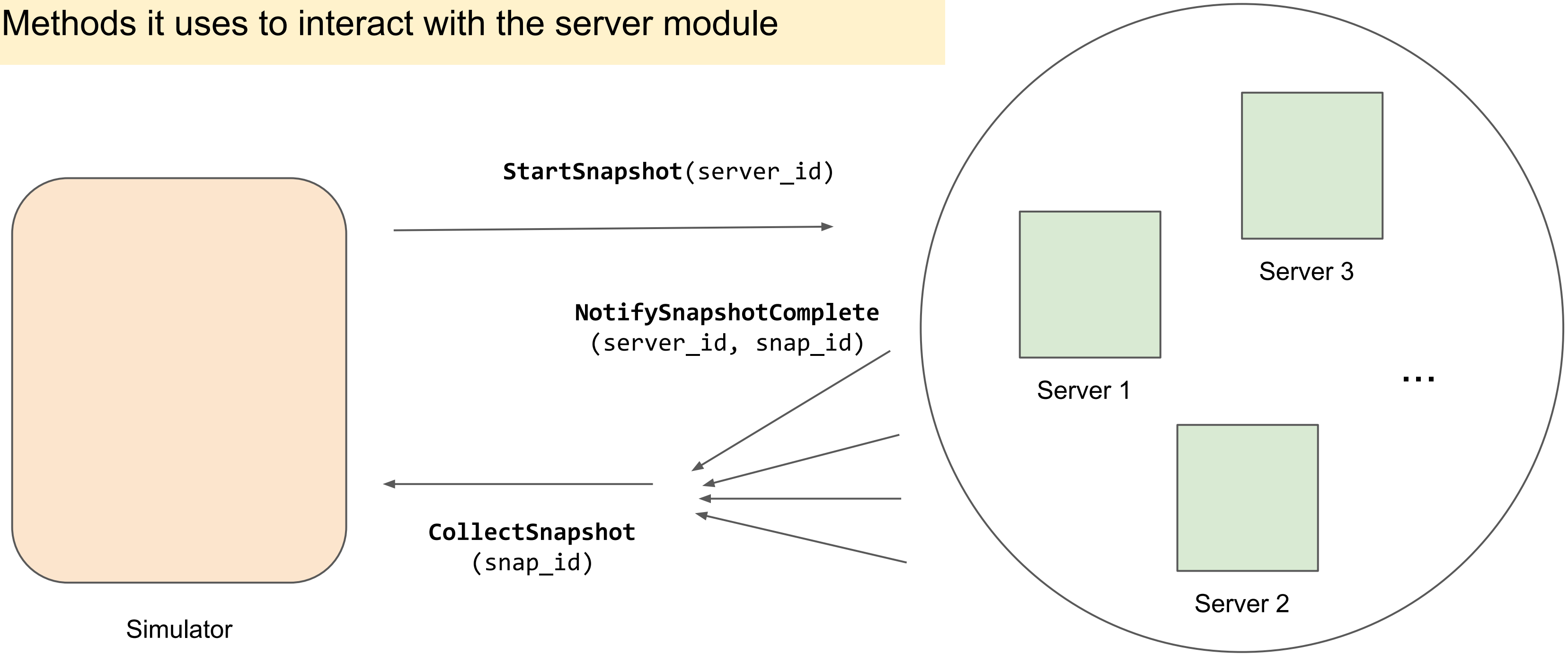
Understand Concept and Code Structure

- Fully comprehend the algorithm
- Spend time to map your understanding of the concept to the starter code
  - For both the system interface and individual modules, understand **what** data is transferred between and **how**
- **Charts** and **pseudocode** can help A LOT!

How is the System Physically Built?

Understand the simulator's implementation (see *simulator.go*)

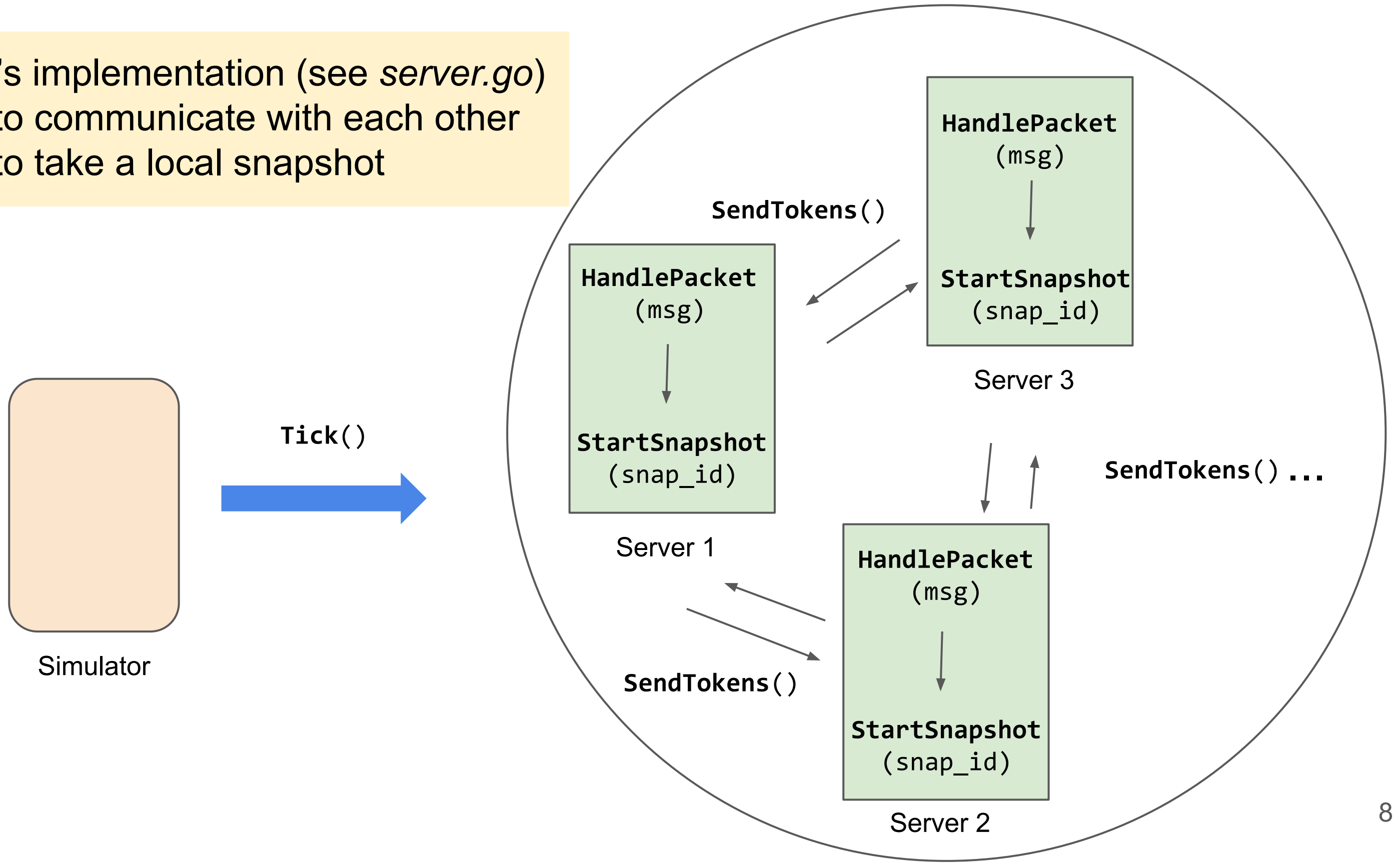
- The role of the simulator
- Methods it uses to interact with the server module



How is the System Physically Built?

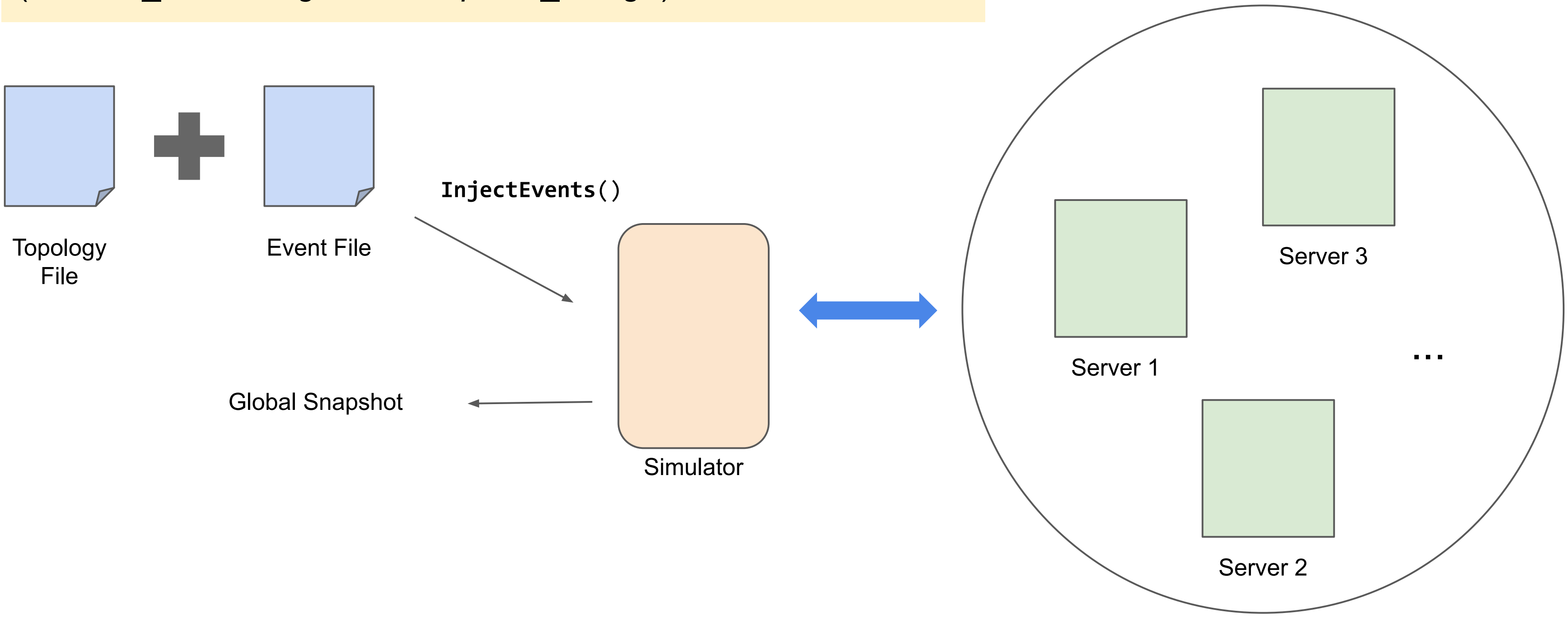
Understand the server's implementation (see *server.go*)

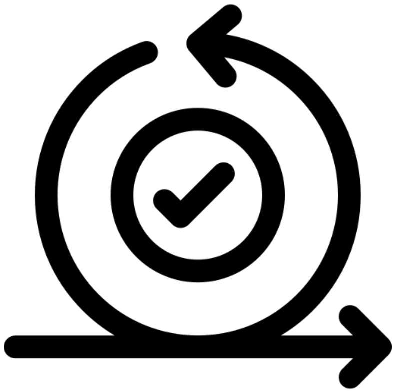
- Methods it uses to communicate with each other
- Methods it uses to take a local snapshot



How to Use the System?

Understand how the external environment talks to our system (see *test_common.go* and *snapshot_test.go*)





Iterative Design Process

Iterative Design Process

Common design methodology in product design, including software design

You will understand a little more about your design when you start implementing it.

- Start with the **base case** (aka simplest case)
  - Example: one global snapshot at a time for Assignment 2, distributed MapReduce without any failure for Assignment 1.3
- Test regularly: should pass test case for 2 nodes, then 3 nodes and …

- Add one more complexity at a time

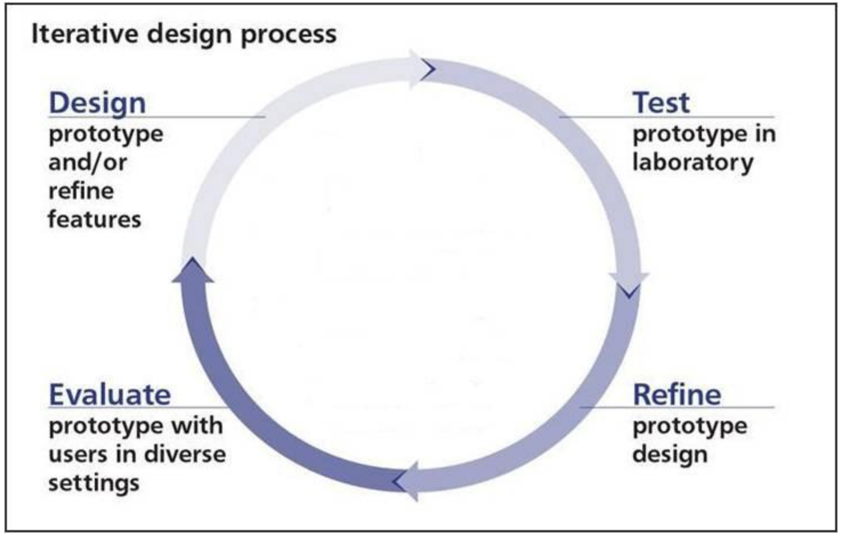
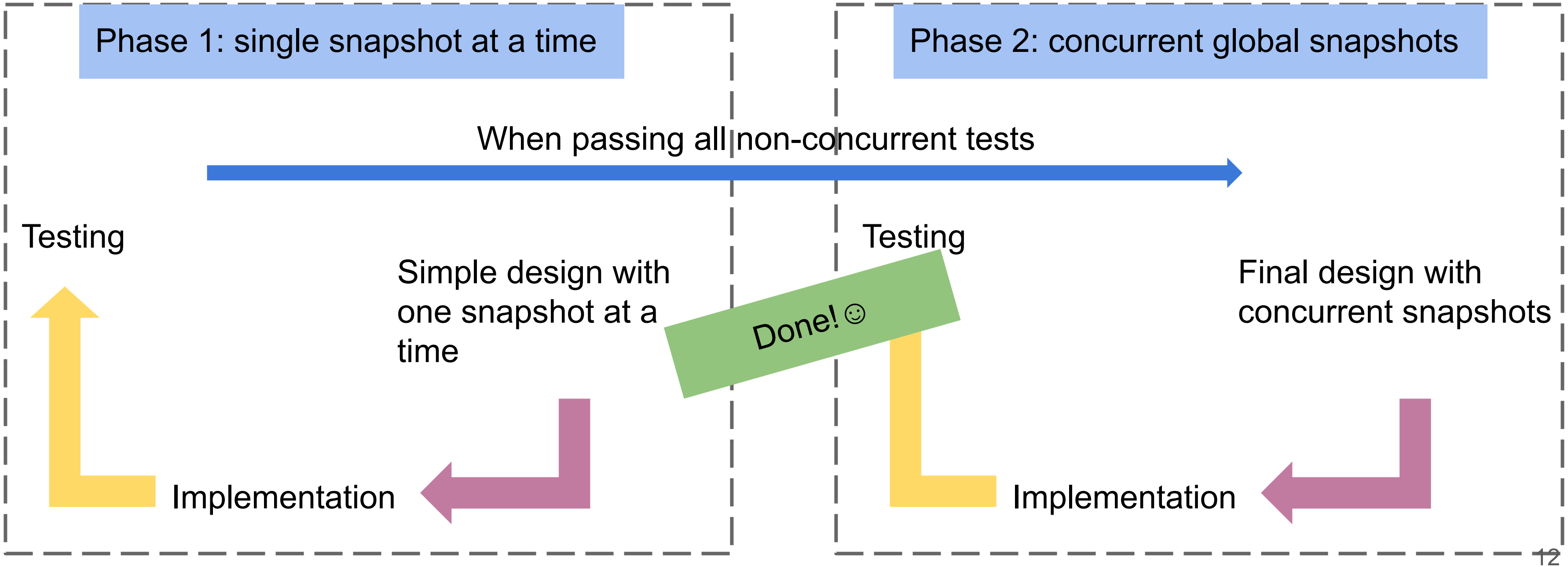
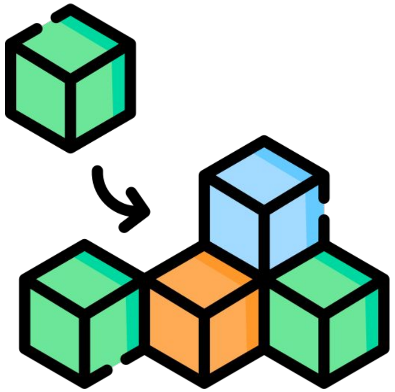


Image Source from the Internet

Iterative Design Process: Distributed Snapshot

Key Idea: Start Simple, then Build Up





Modular Programming

Modular Programming

Iterative design means <u>code change</u> every time when refining the design ☹️

Modular programming

- Decompose the system into several **independent modules/pieces**
- Use a set of simple yet flexible APIs for intra-module communication

Advantages of modular programming

- Makes it easier to reason about and debug each component of your system
- Requires **minimal change in the code**

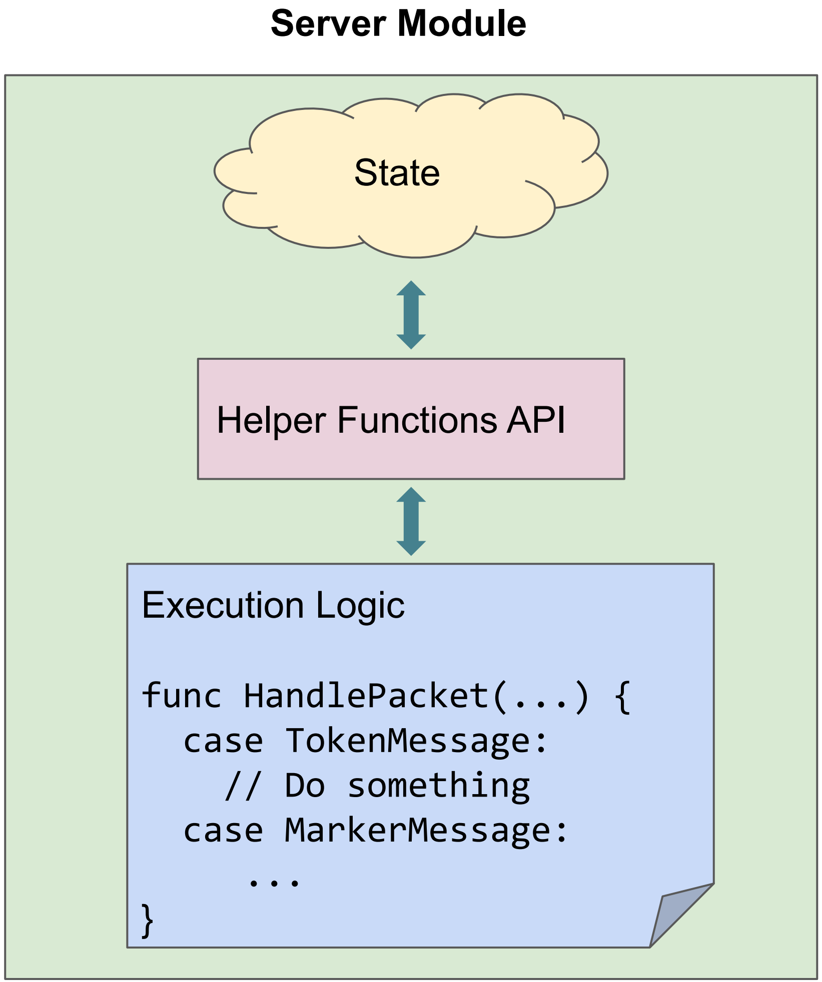
Modular Programming

Phase 1: single snapshot at a time

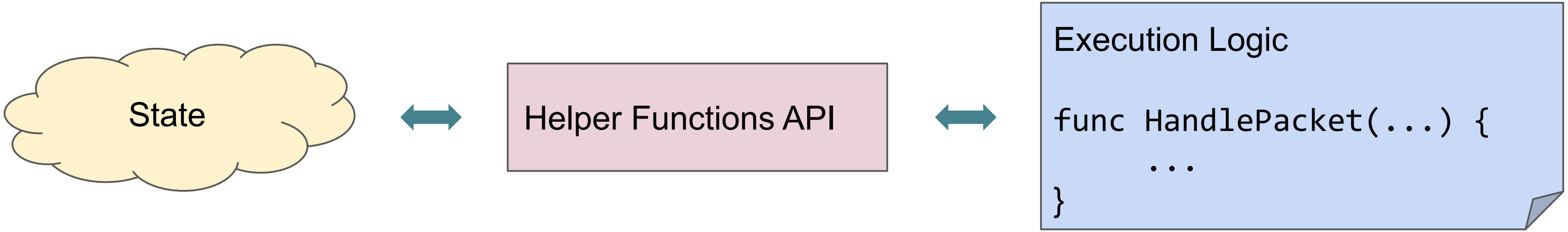
Divide our server module into 3 pieces:

- Server State
- Execution logic
- A layer of helper functions

Goal: write a flexible layer of helper functions



Modular Programming: Single Snapshot




```
// ID of the current snapshot
snapId: int (init to -1)

// State of the current snapshot
snapState: SnapshotState

// Track if each incoming
channel has seen a marker
message (default to false)
receivedMarker:
map(source channel, bool)
```

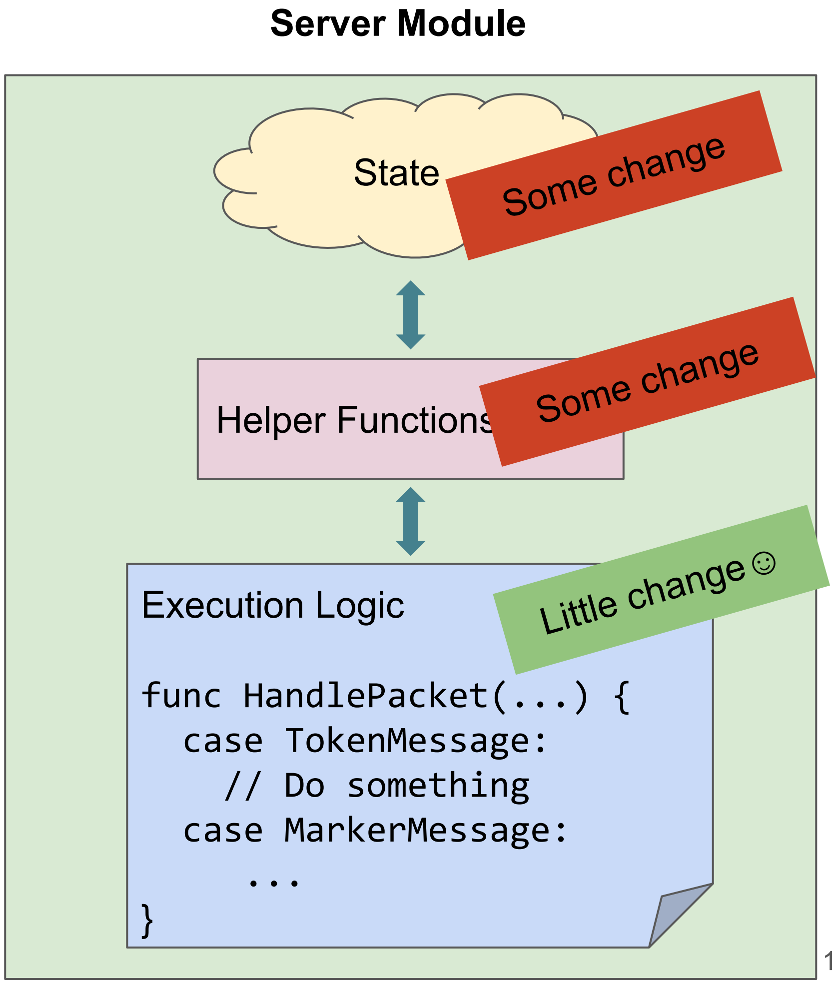
```
func updateSnapshot(src, msg) {
  snapMsg = SnapshotMessage(src, msg)
  snapState.messages.append(snapMsg)
}

func setReceivedMarker(src) {
  receivedMarker[src] = true
}

func firstMarkerMsg(snap_id) {
  return snapId != snap_id
}

Func receiveAllMarkers() {
  return receivedMarker.size ==
inboundLinks.size
}
```

```
func HandlePacket(src, msg) {
  ...
  case TokenMessage:
    updateSnapshot(src, msg)
    // Also, update server's local
state
  case MarkerMessage:
    snap_id = getSnapId(msg)
    if firstMarkerMsg(snap_id) {
      StartSnapshot(snap_id)
    } else {
      setReceivedMarker(src)
      if receiveAllMarkers() {
        // Notify simulator of the
completion
      }
}
```

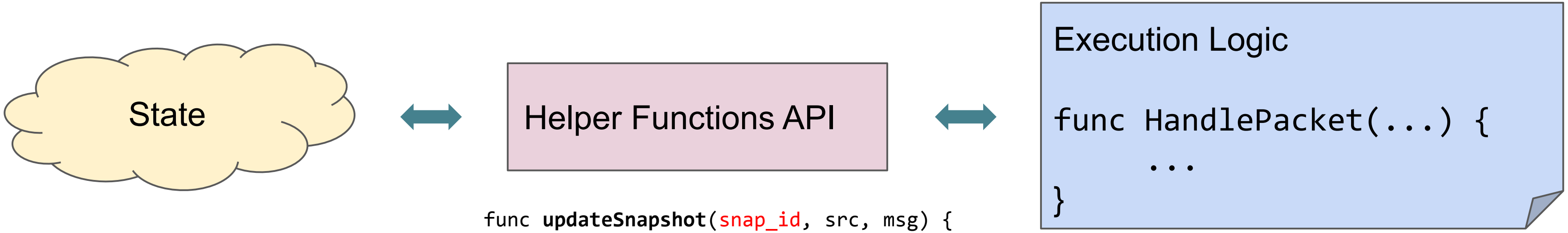
Modular Programming

Phase 2: concurrent snapshots

- Update the state variables and helper functions’ implementation
- Keep the API and execution logic unmodified (almost)



Modular Programming: Concurrent Snapshots




```
// States of concurrent snapshots
// map snapshot ID to its state
snapStates: map(int, SnapshotState)

// For each snapshot, track if each
incoming channel has seen a marker
message (default to false)
receivedMarker:
map(int, map(source channel, bool))
```

```
func updateSnapshot(snap_id, src, msg) {
  snapMsg = SnapshotMessage(src, msg)

snapStates[snap_id].messages.append(snapMsg)
}

func setReceivedMark(snap_id, src) {
  receivedMarker[snap_id][src] = true
}

func firstMarkerMsg(snap_id) {
  return (snap_id in snapStates.keys())
}

Func receiveAllMarkers(snap_id) {
  return receivedMarker[snap_id].size ==
inboundLinks.size }
```

```
func HandlePacket(src, msg) {
  ...
  case TokenMessage:
    for snap_id in snapStates.keys() {
      updateSnapshot(snap_id, src, msg)
    }
    // Also, update server's local state
  case MarkerMessage:
    snap_id = getSnapId(msg)
    if firstMarkerMsg(snap_id) {
      StartSnapshot(snap_id)
    } else {
      setReceivedMarker(snap_id, src)
      if receiveAllMarkers(snap_id) {
        // Notify simulator of the
completion
      }
}
```

1. Update state variables

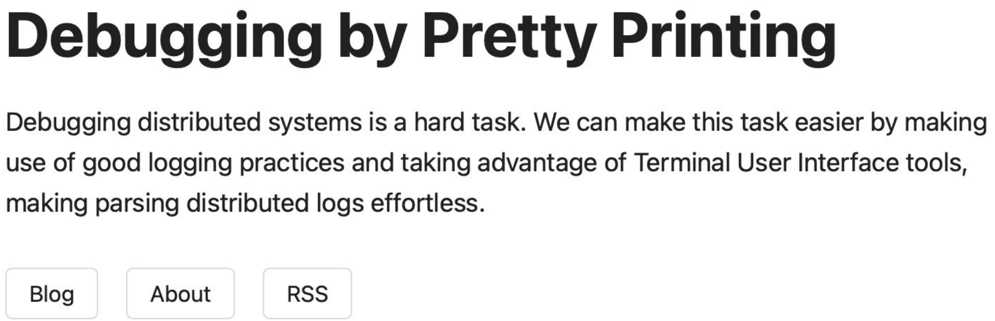
2. Update helper functions while keeping most of its API intact

3. Minimal change on execution logic

Tips for Debugging

Tips on Debugging

- **Start Early! (This is imperative for Assignment #4)**
- **Commit your code to Git often and early**, and every time when you pass a new test (enable comparative debugging later if necessary)
- Have proper naming for variables and add comments in your code
  - Easier for both you and others to read and debug your code
- Take advantage of Go Playground if you are not familiar with any Go specifics
- Print statements are your friend!
- Read this ASAP



Prints Are Your Friend ☺

- **Always verify** the behavior of your program! Sometimes, it may not align with your expectation because of some hidden bugs.
- Track execution using printing statements to understand the code flow
    - Especially helpful in the early development of your design when the code complexity is not too high
- Help catch errors in the early stage
- Example
    - In Assignment 2, we can print out the server state before and after `HandlePacket()` and `StartSnapshot()` that you implement after each tick of the simulator

Paxos

Paxos Made Simple

Leslie Lamport

01 Nov 2001

Abstract

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

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].

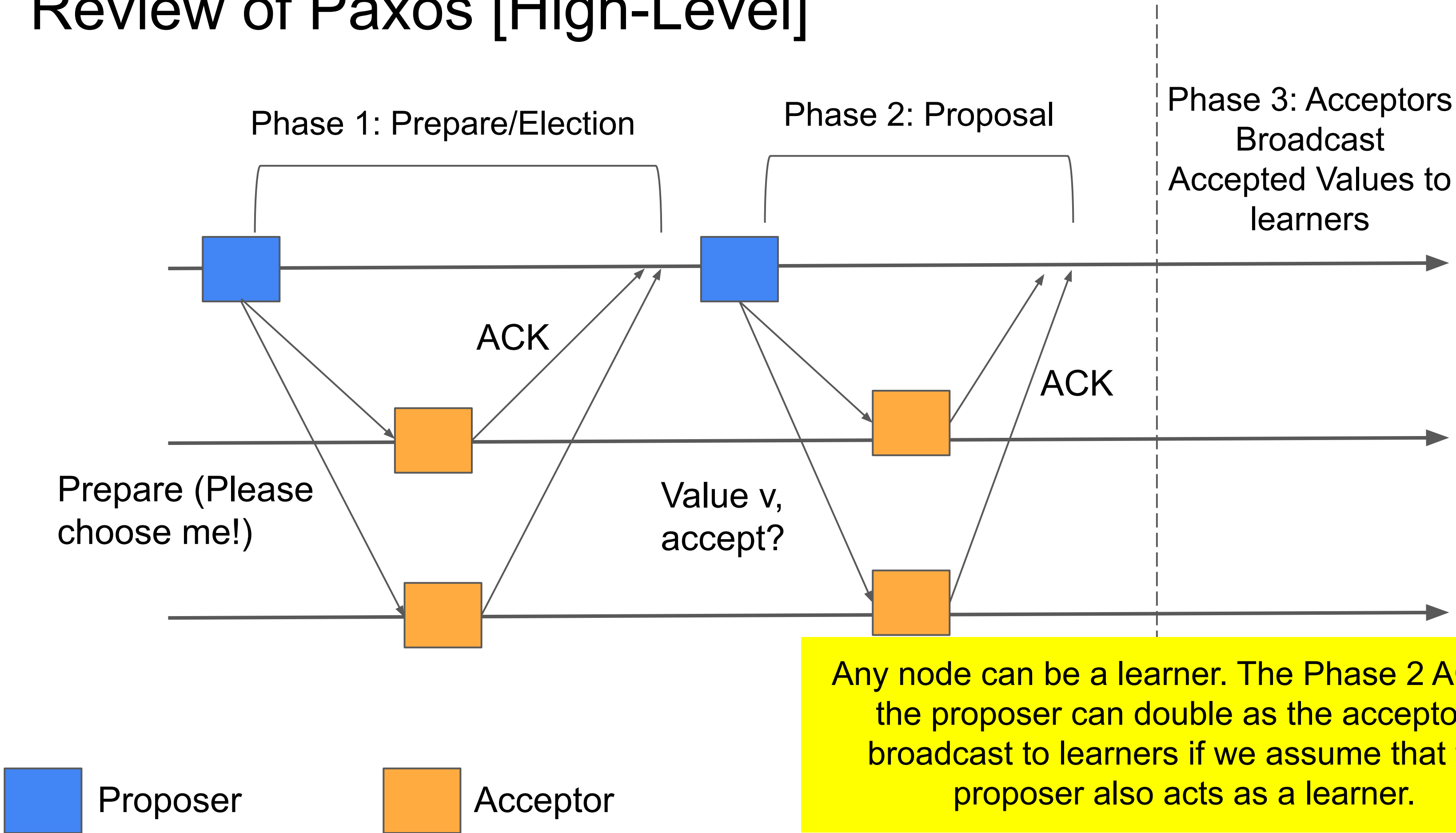
Paxos is all about consensus

Consensus

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

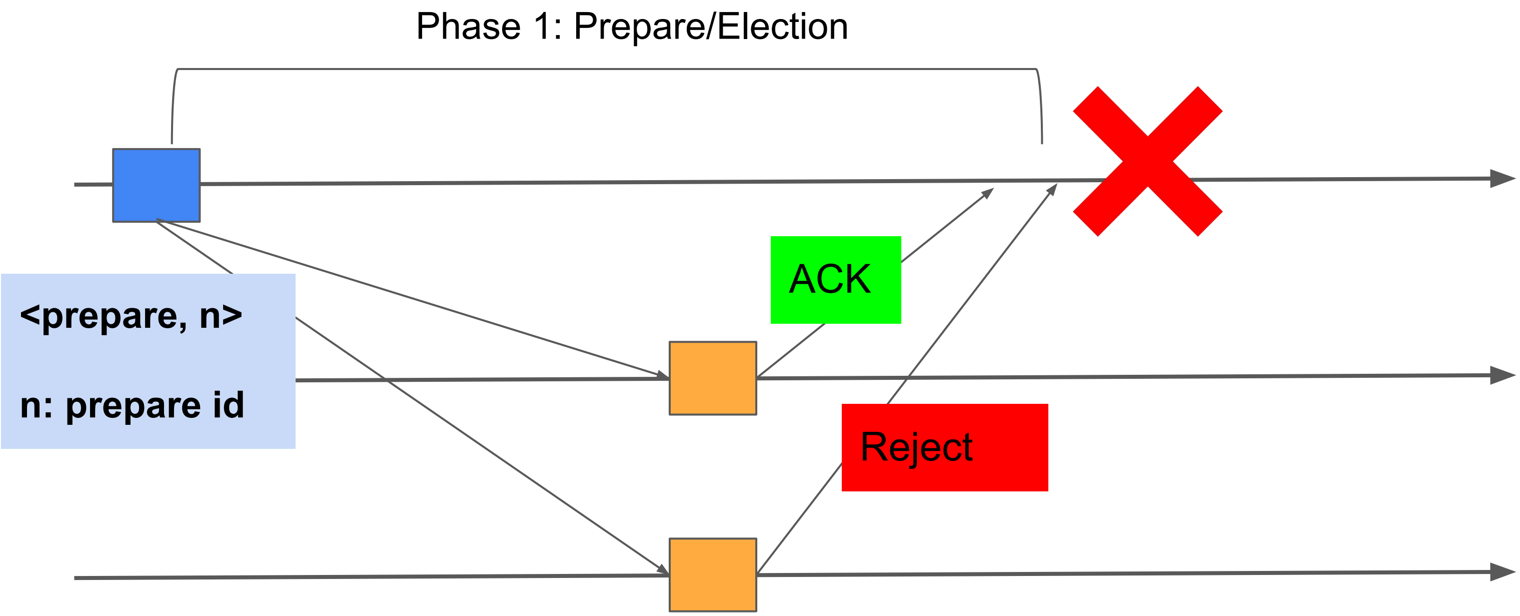
- **Termination: All non-faulty processes eventually decide on a value**
- **Agreement: All processes that decide do so on the same value**
- **Validity: Value decided must have proposed by some process**

Review of Paxos [High-Level]



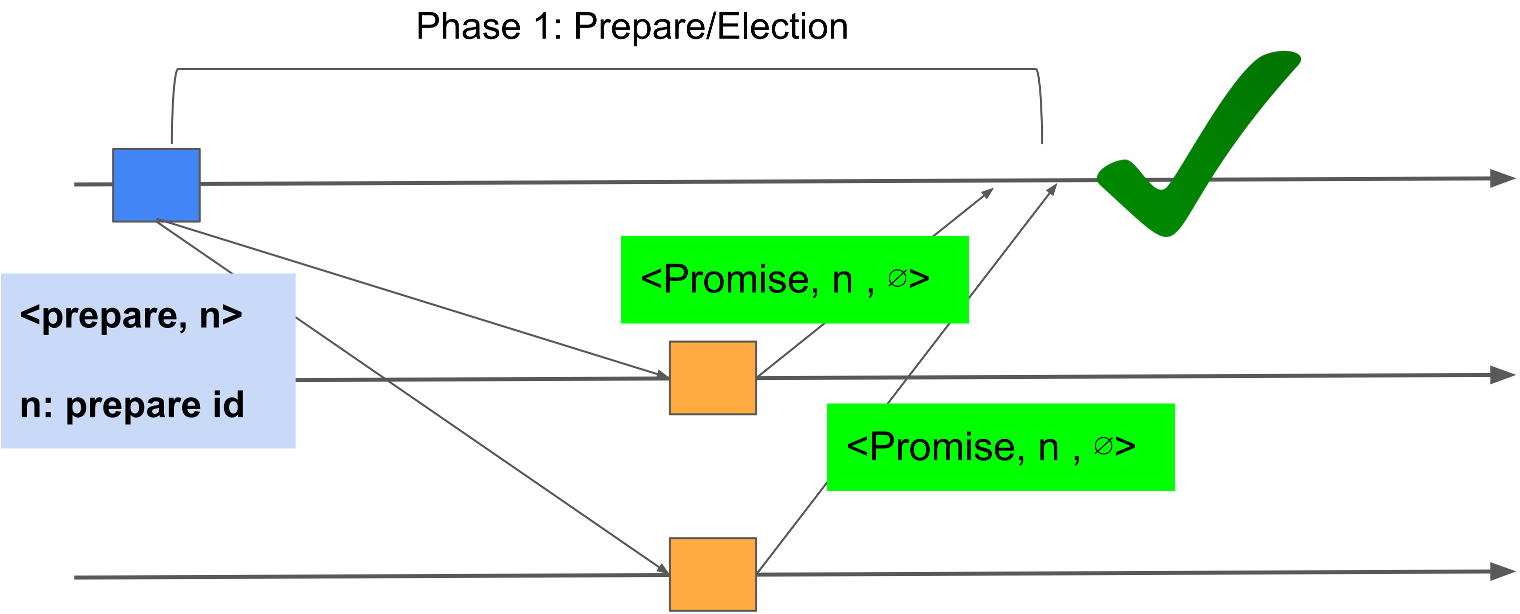
Any node can be a learner. The Phase 2 ACK to the proposer can double as the acceptors' broadcast to learners if we assume that the proposer also acts as a learner.

Review of Paxos [Detailed]



n < n_highest
This acceptor has seen a prepare message with a higher prepare_id

Review of Paxos [Detailed]



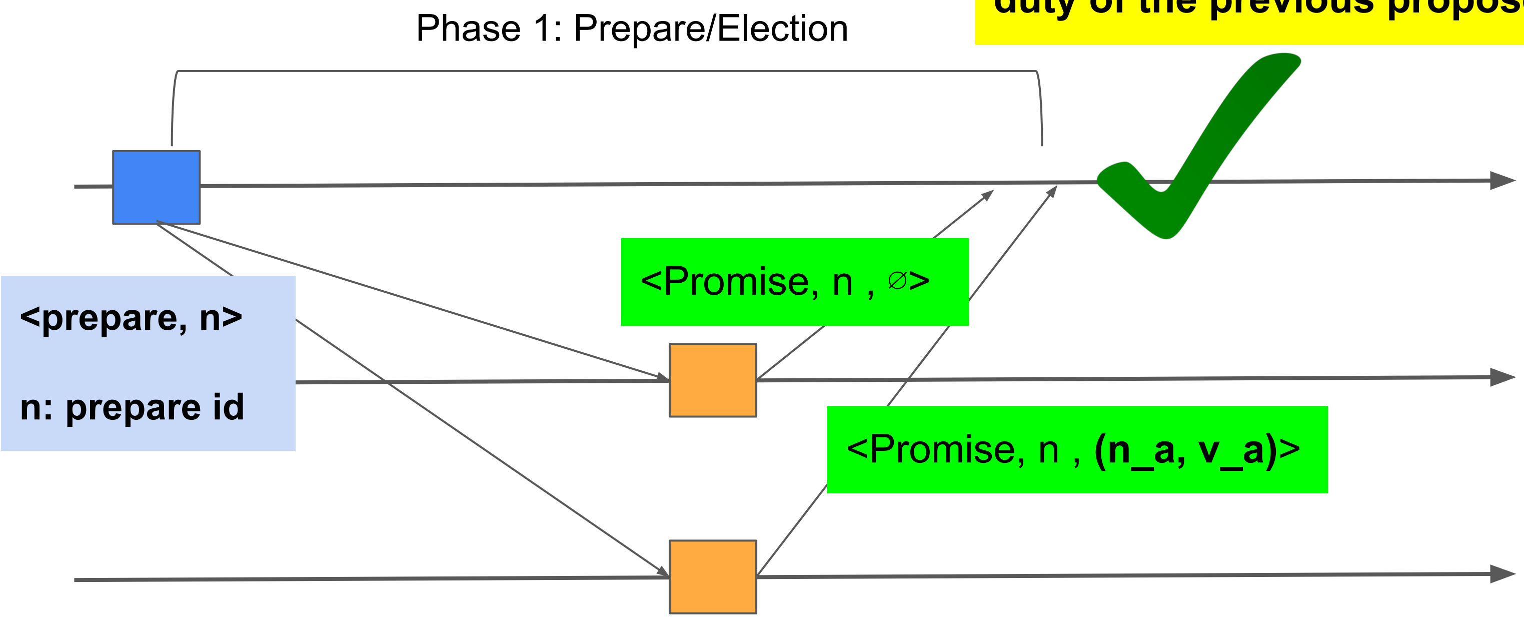
Both acceptors accept and update their **n_highest = n**
Assumption: None of them had accepted other proposals before

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**]



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 <accept, (n, v_a || v)> to acceptors

- **Acceptors:**
  - Upon receiving (n, v), if $n \geq n_h$,
    - Accept proposal and notify learner(s)

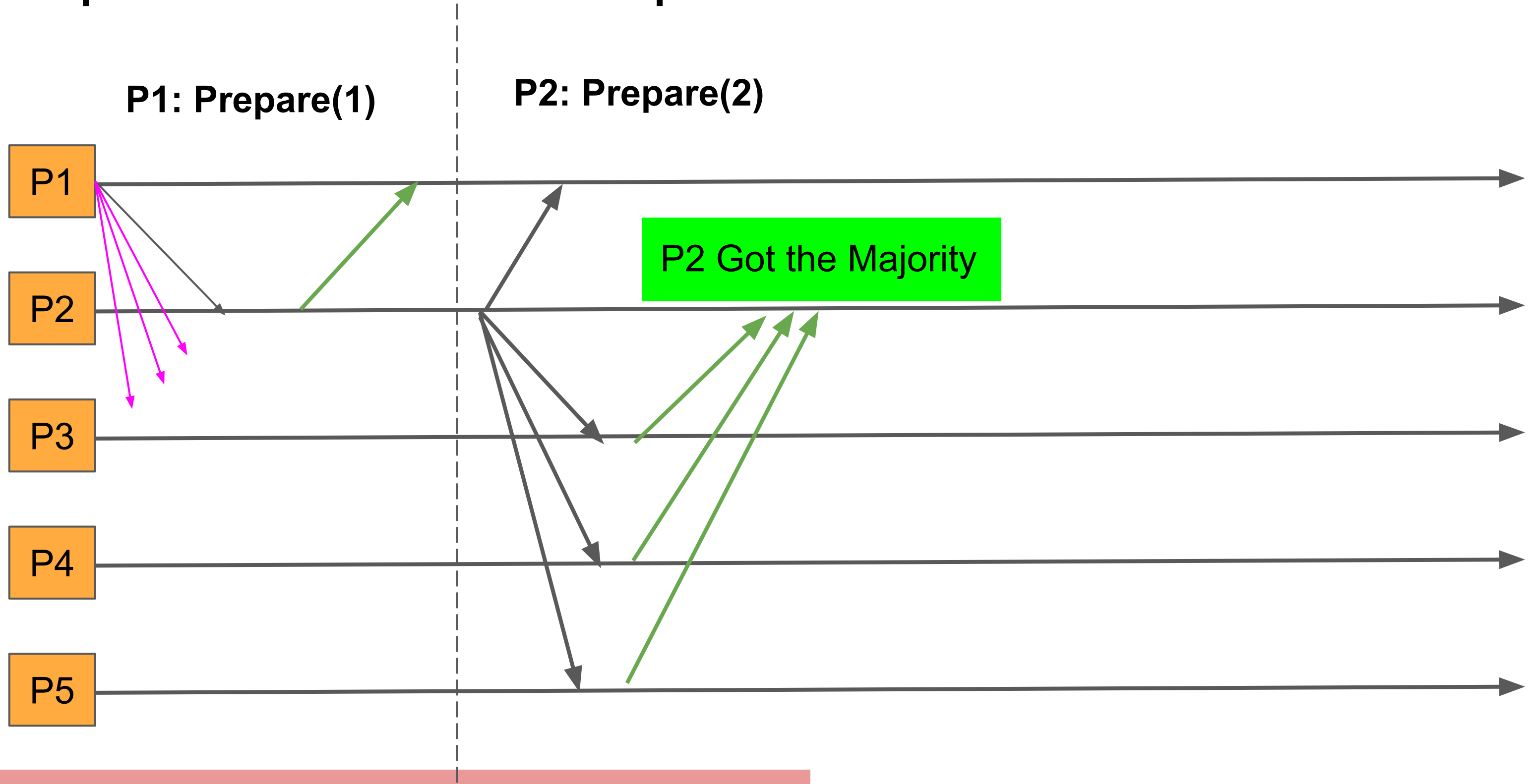
Let's practice

- 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.

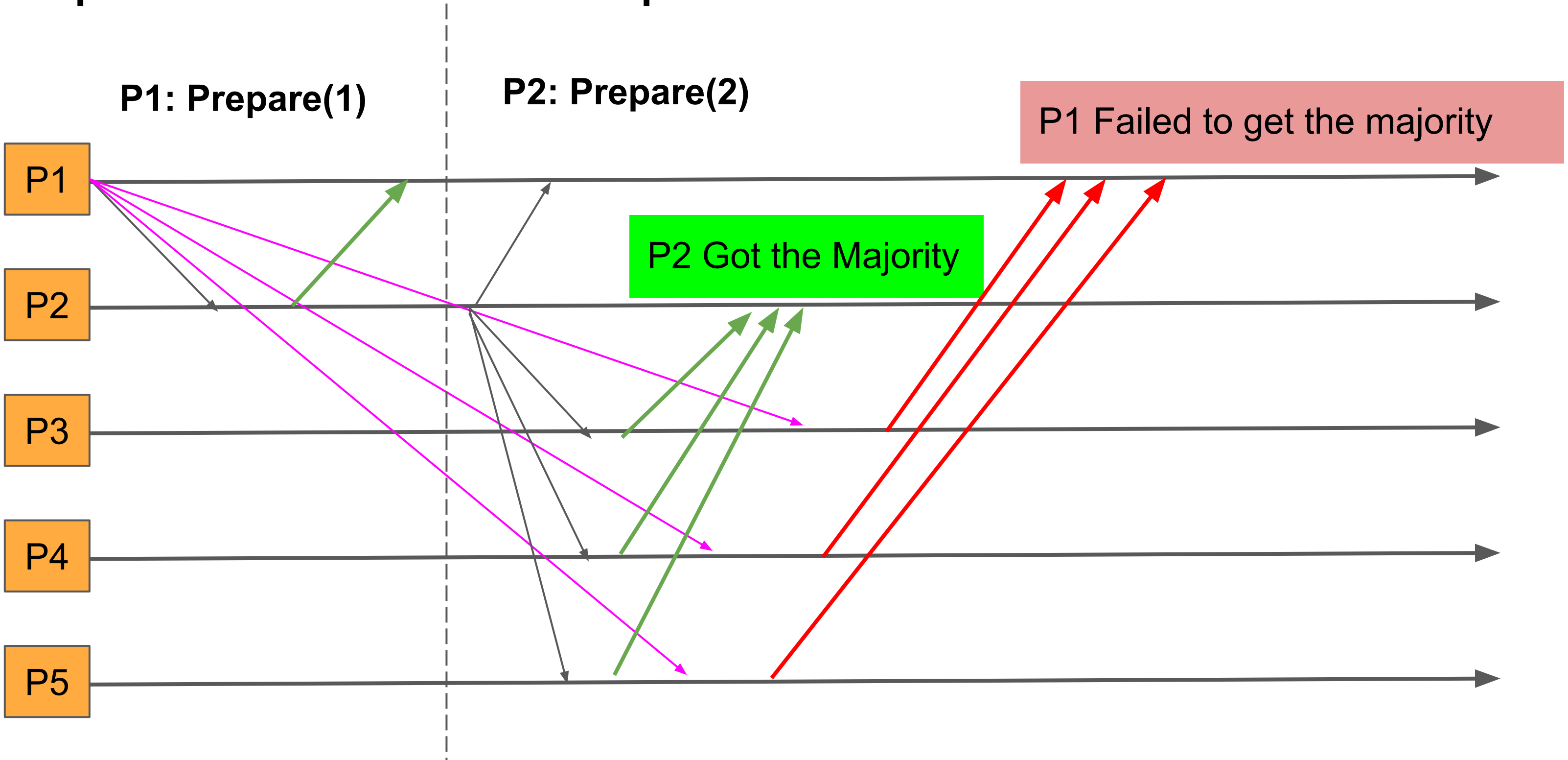
Let's practice: Failed Prepare



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

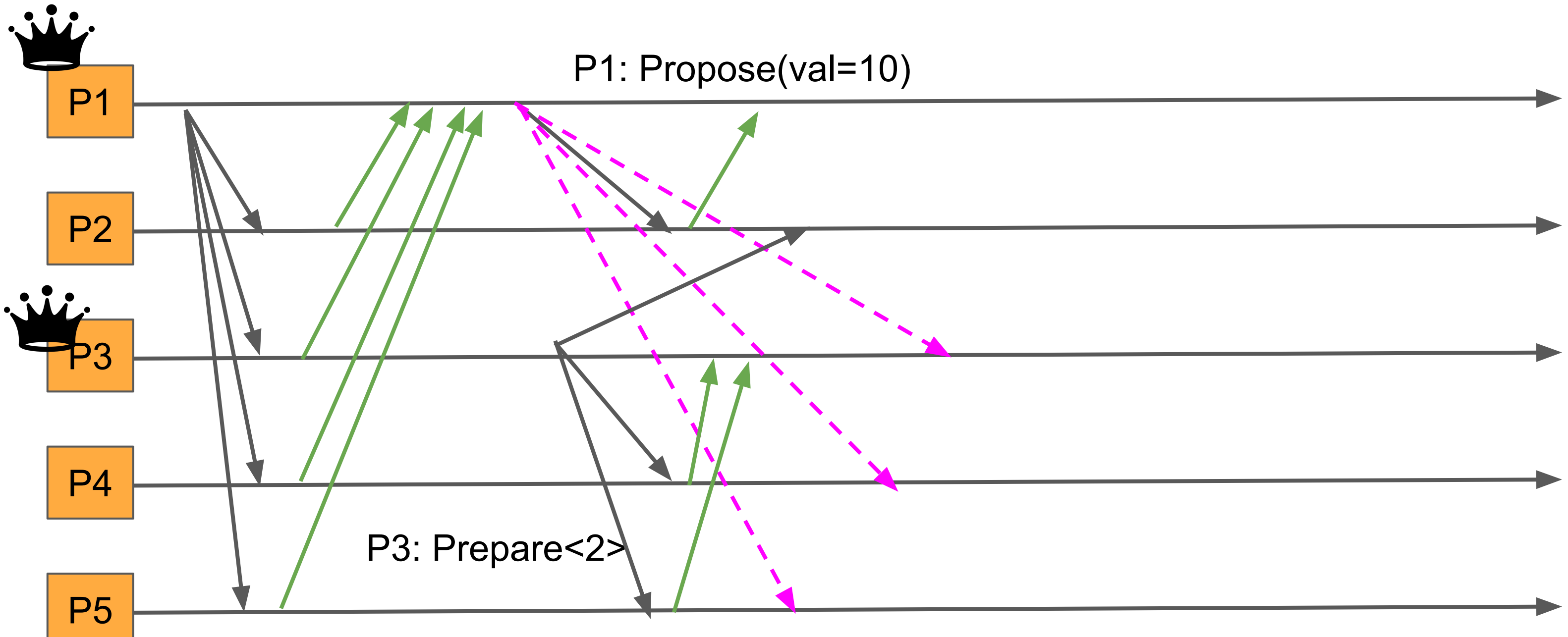


Pink Arrows are slow communications, still on-route

Let’s practice: Racing

Successful Prepare Phase; Now P1 can propose!

Let’s practice: Racing



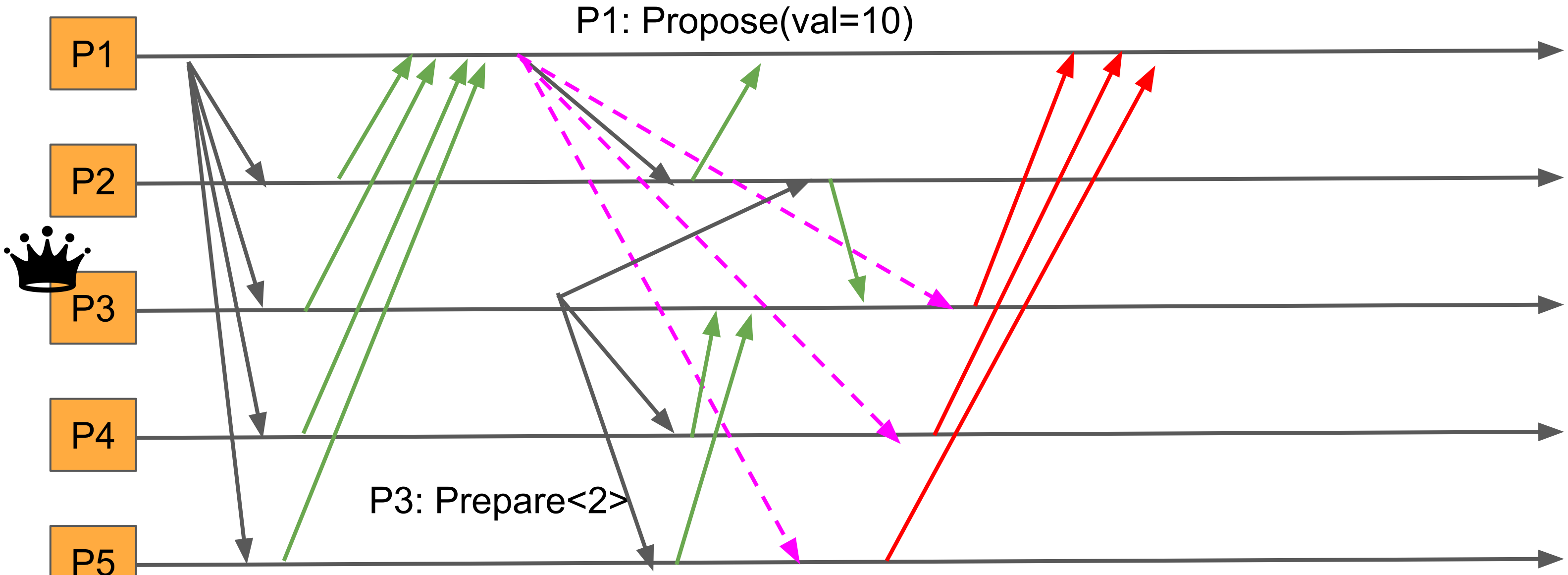
But before the proposal arrive at {P3, P4, P5}, P3 starts prepare with higher id

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

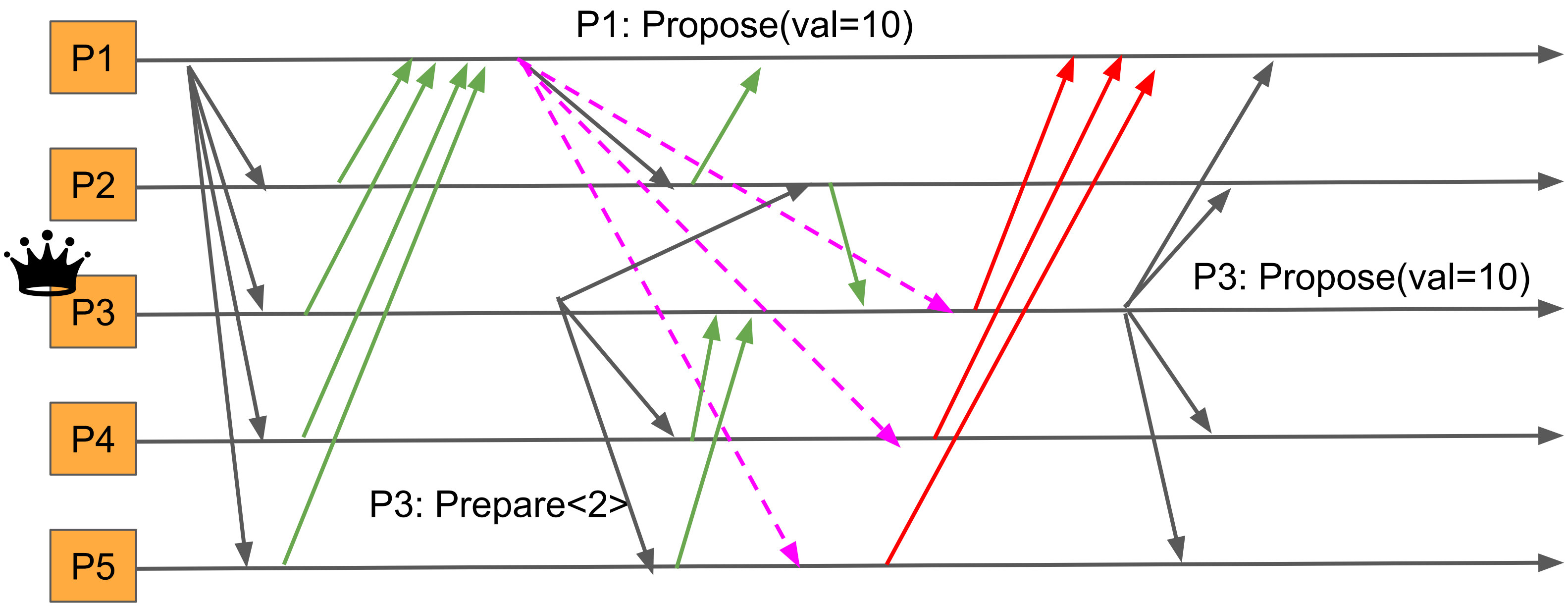
The old proposal value of P1 is now declined, because there was a higher new prepare.



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.

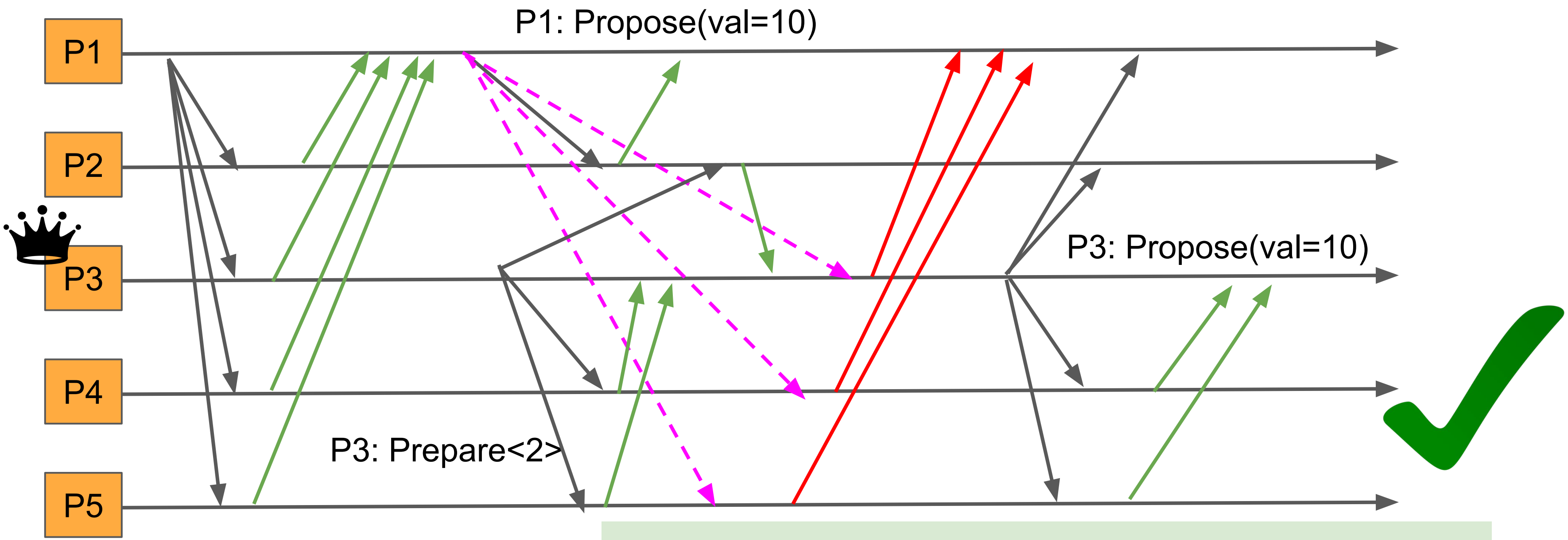
Let’s practice: Racing



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

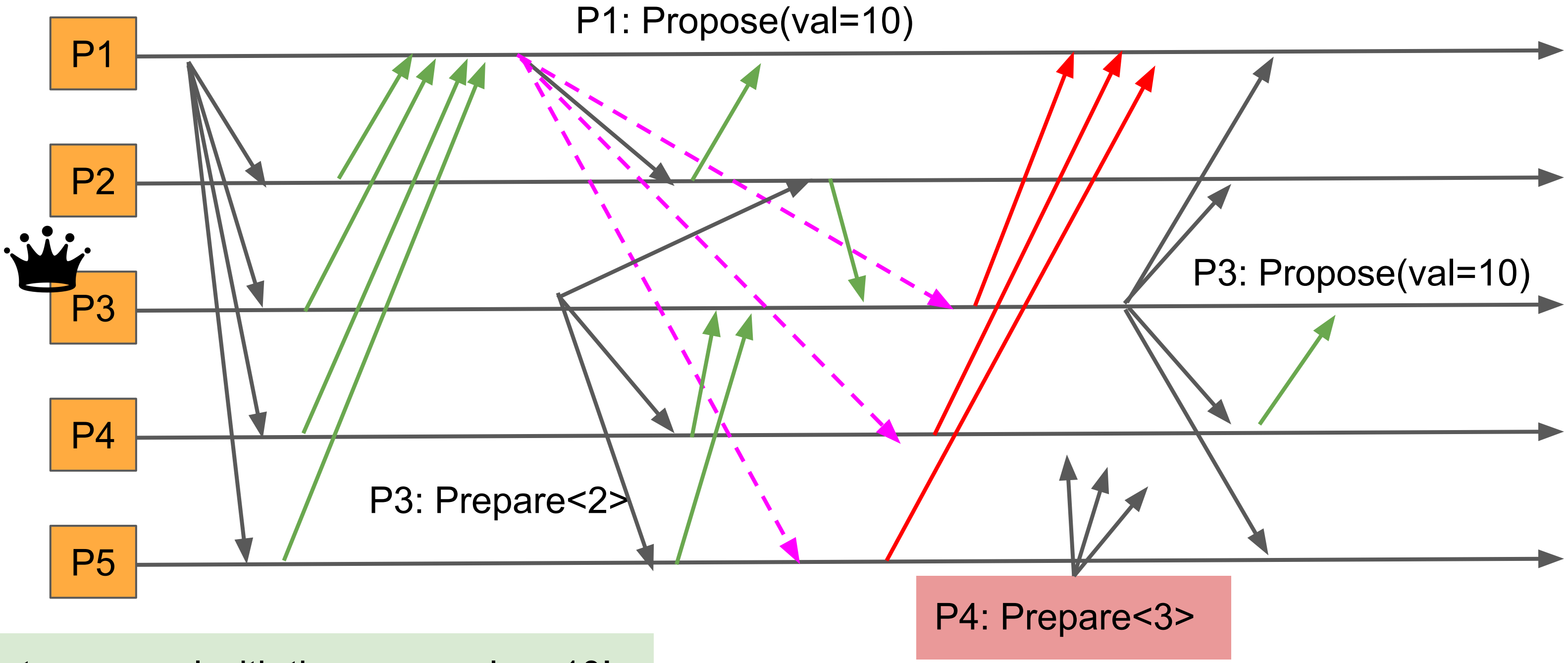


P3 starts proposal with the same value=10!

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

Let’s practice: Racing

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?



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!