BullShark

Presenters: Gia-Phong Nguyen, Harshil Bhandari, Ziyang Chen

Authors: Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, Lefteris Kokoris-Kogias

Intro

PBFT Review

DAG Overview

DAG-Rider Overview

BullShark

Eventually Synchronous BullShark

Garbage Collection

Evaluation

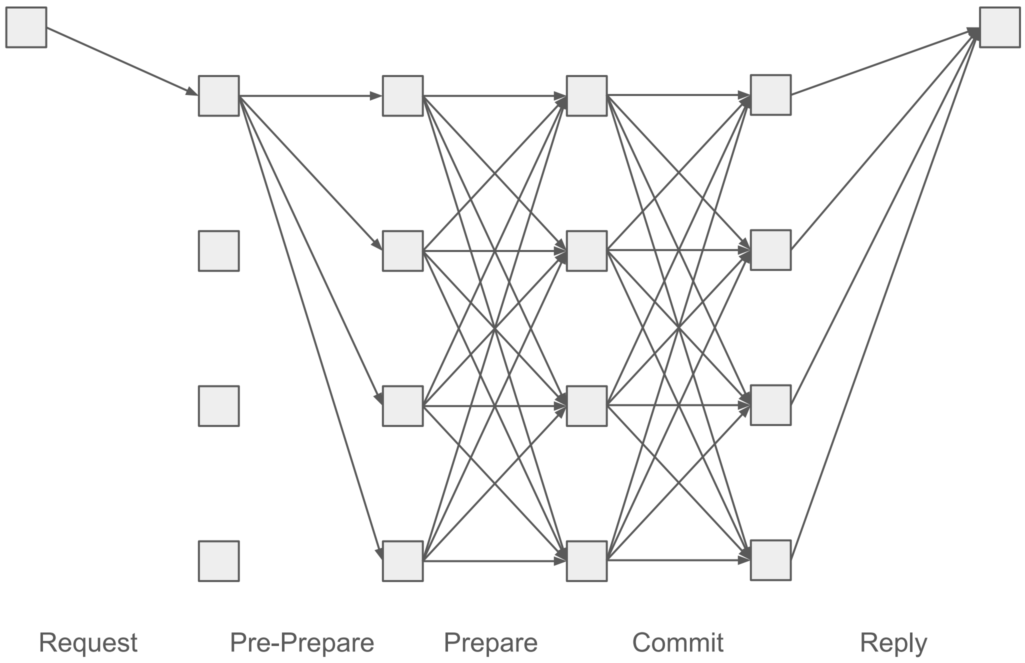
PBFT Review

Pre-prepare phase: The leader sends a pre-prepare message to all the other replicas containing a sequence number.

Prepare Phase: Nodes propose a value and broadcast it to others. Each node collects a set of messages from other nodes, confirming the proposed value.

Commit Phase: Nodes broadcast a commit message once they receive enough prepare messages. When a node collects enough commit messages, it commits the proposed value.

View Change: In case of node failure or Byzantine behavior, a view change is initiated. The system switches to a new view, and a new primary is chosen to continue the consensus process.



PBFT shortcomings

Latency and Communication Complexity

- Several all to all communications for one transaction

View Change

- Expensive synchronization

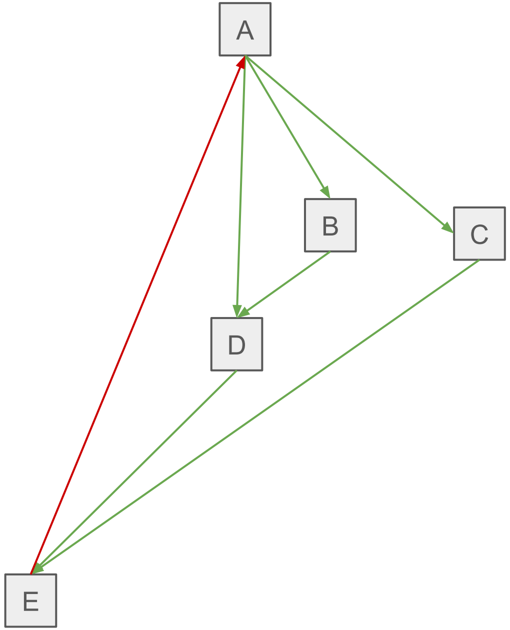
What is a DAG

A directed acyclic graph (DAG) is a conceptual representation of a series of nodes

The order of the nodes is depicted by a graph, each node may represent an activity

Each edge is directed and represents the flow from one node to another

Rule: the flow goes in a specific direction and it contains no cycles

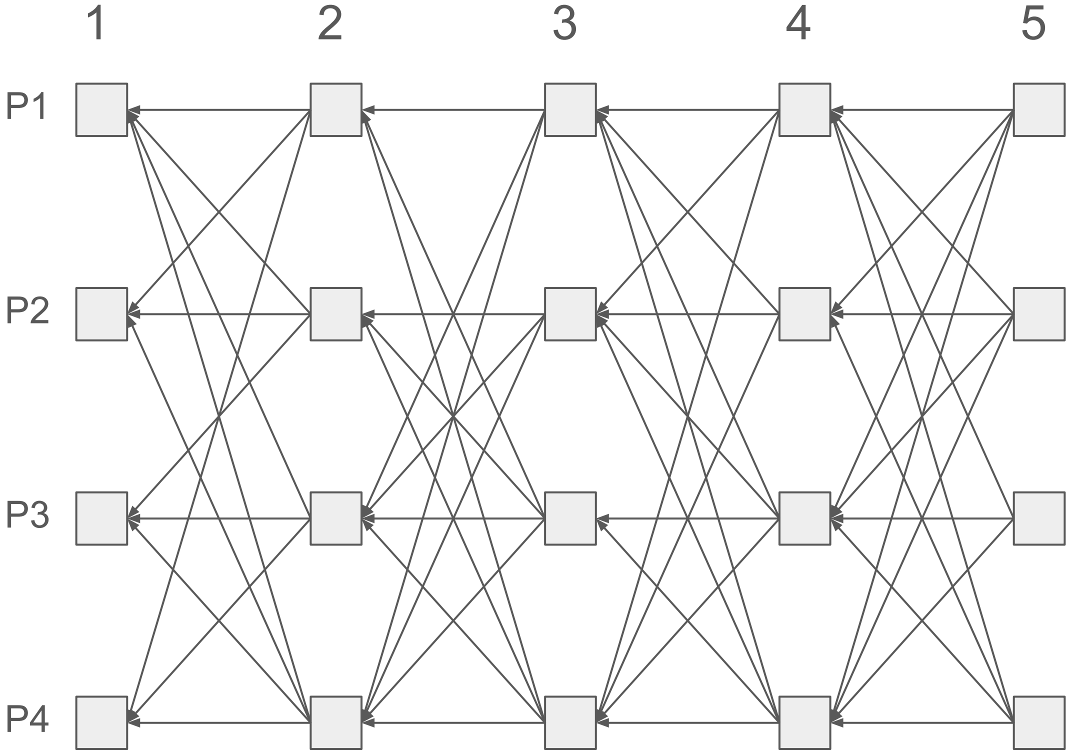


What is DAG-BFT?

Node: a party at a specific time with a block of transactions

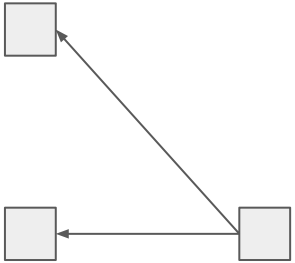
Edges: communication of of that block to another party

Adding vertices with edges = new transaction after receiving some other blocks

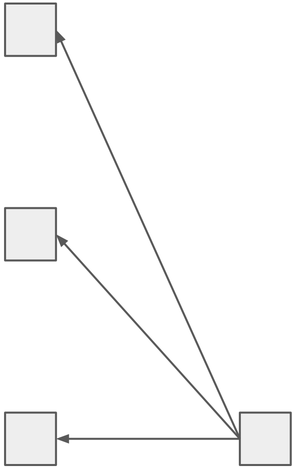


DAG-BFT - Vertex Creation

Need 2f + 1 edges to previous round to create new vertex



Not Valid

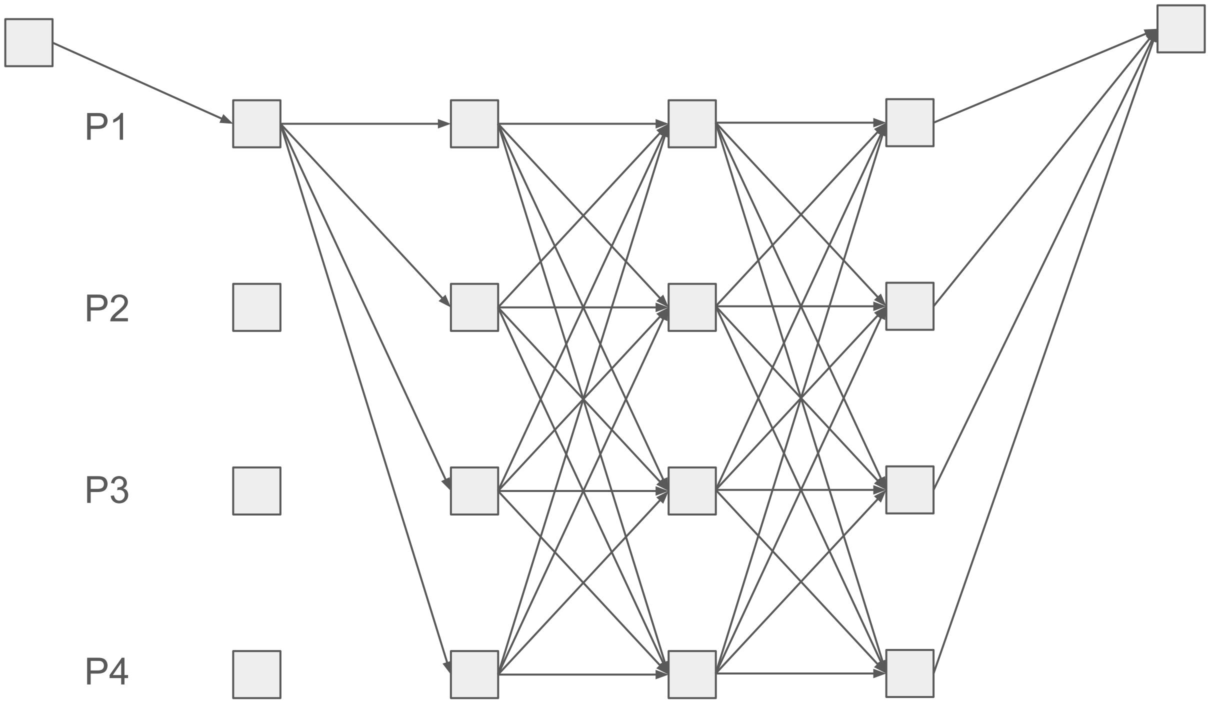


Valid

Why DAG-BFT?

PBFT: one request per round

View change protocol is expensive



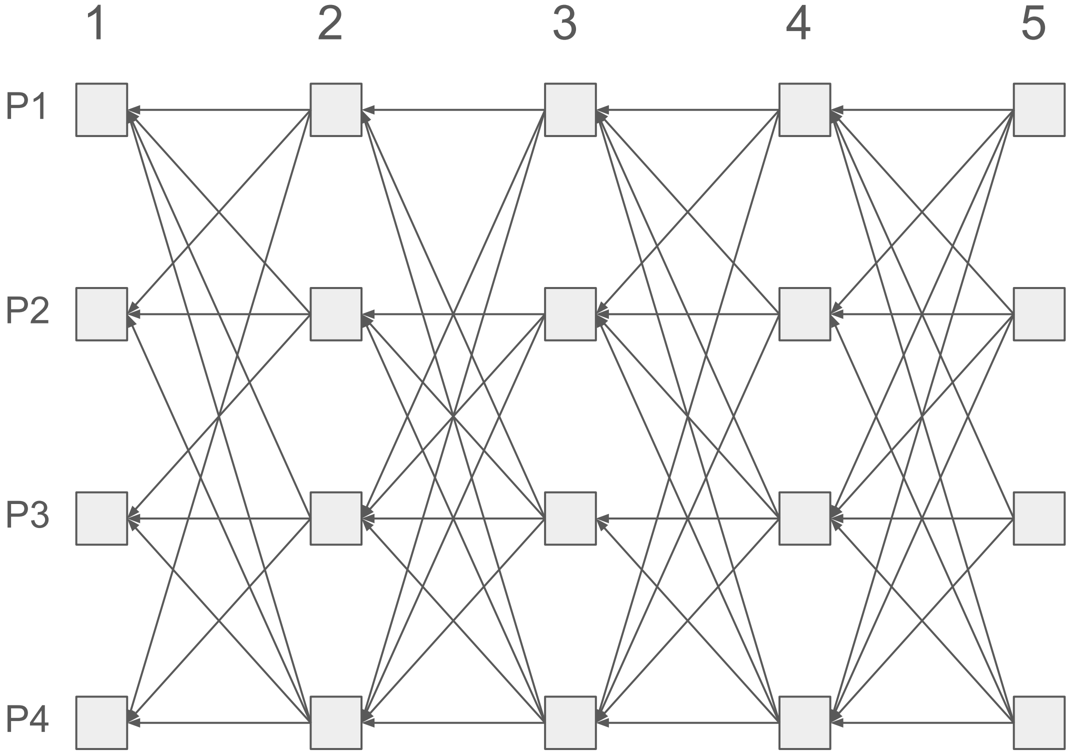
Why DAG-BFT?

Commit several transactions at once

Don’t need to be a primary to propose new transactions

Separation of communication and consensus logic

- No need for view change protocol



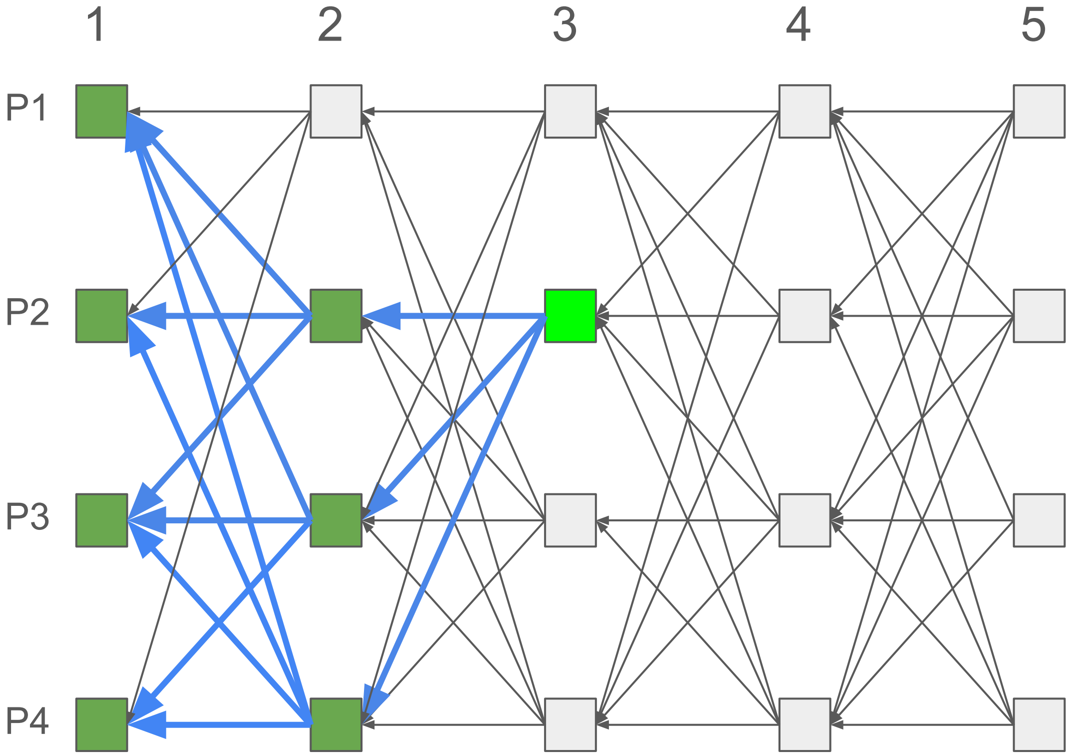
Challenges to DAG

Validity

Reliability

Non-Equivocation

Completeness



Reliable Broadcast

Reliable broadcast is an important building block of many asynchronous protocols

Agreement:

If some non-faulty party outputs a value then eventually all non-faulty parties will output the same value.

Validity:

If the leader is non-faulty then eventually all non-faulty parties will output the leader’s input

DAG-Rider overview

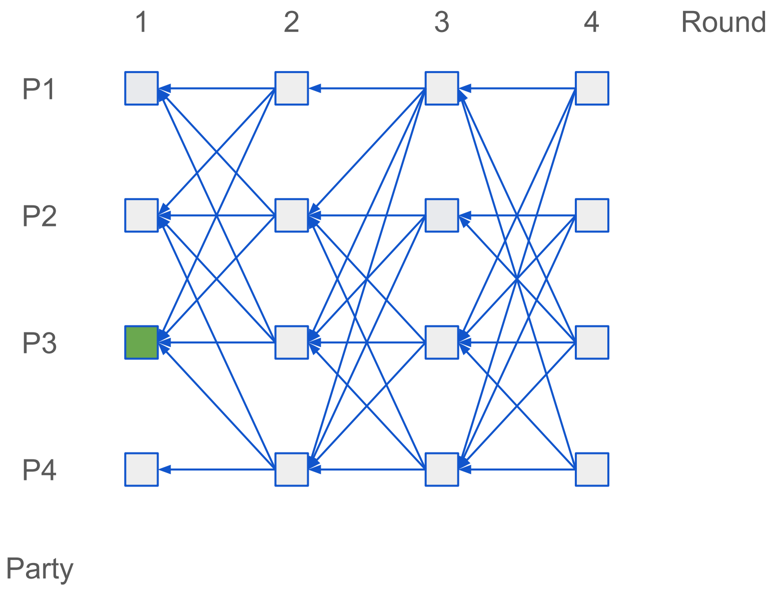
Uses DAG to abstract the communication layer among parties

Each vertex in the DAG represents a message disseminated via reliable broadcast, and it contains the references (edges of the DAG) to previously broadcast vertices.

Each honest party maintains a local copy, might observe different views of the DAG.

Utilizes reliable broadcast to prevent equivocation, and to guarantee that all honest parties eventually deliver the same messages

- The views of the DAG eventually converge



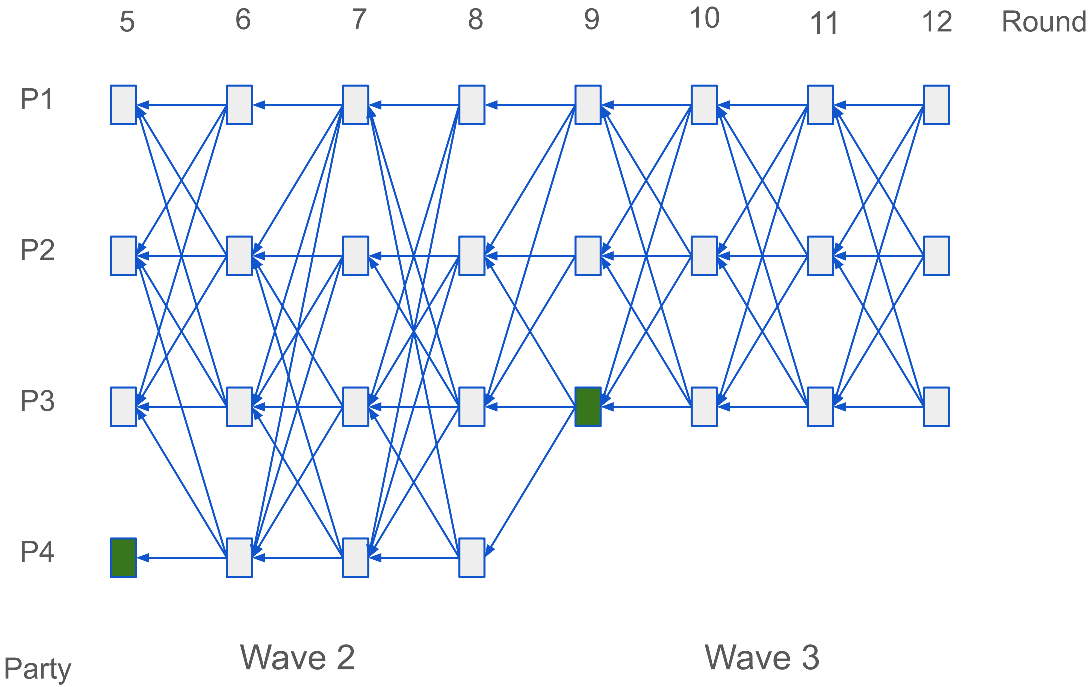
DAG-Rider overview

Two types of edges: Strong edges, weak edges

Reliable broadcast

Does not waste any of the messages, all proposed values by correct processes are eventually ordered.

Structured into a wave-by wave approach, each wave consist of 4 consecutive rounds, try to commit a randomly chosen leader vertex every wave.



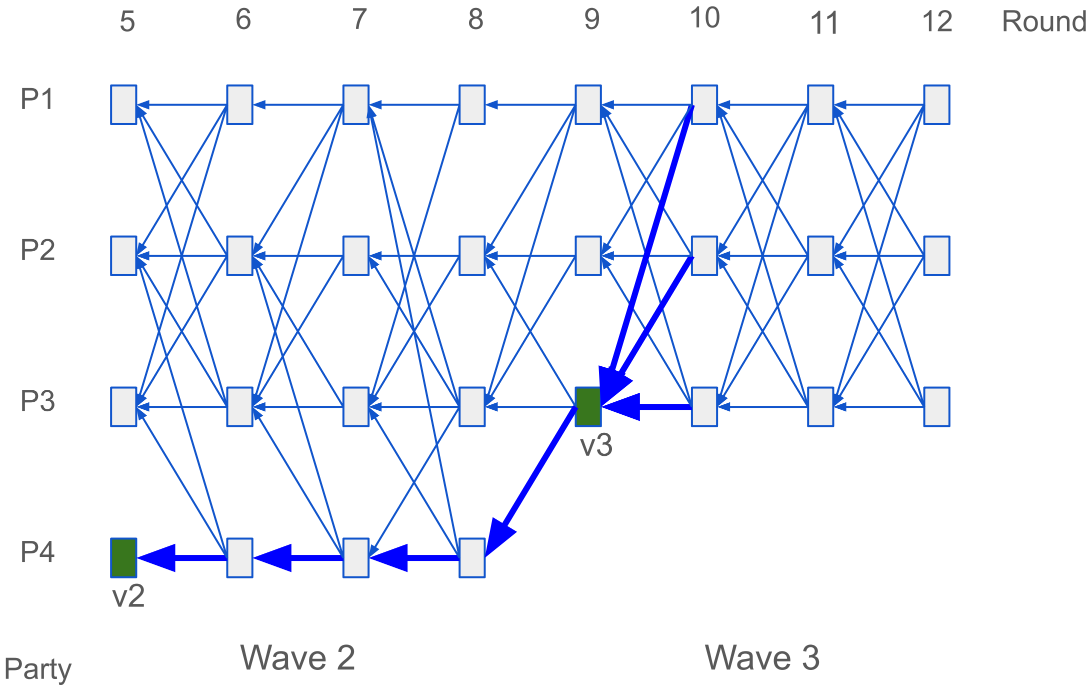
DAG-Rider overview

Leader selection:

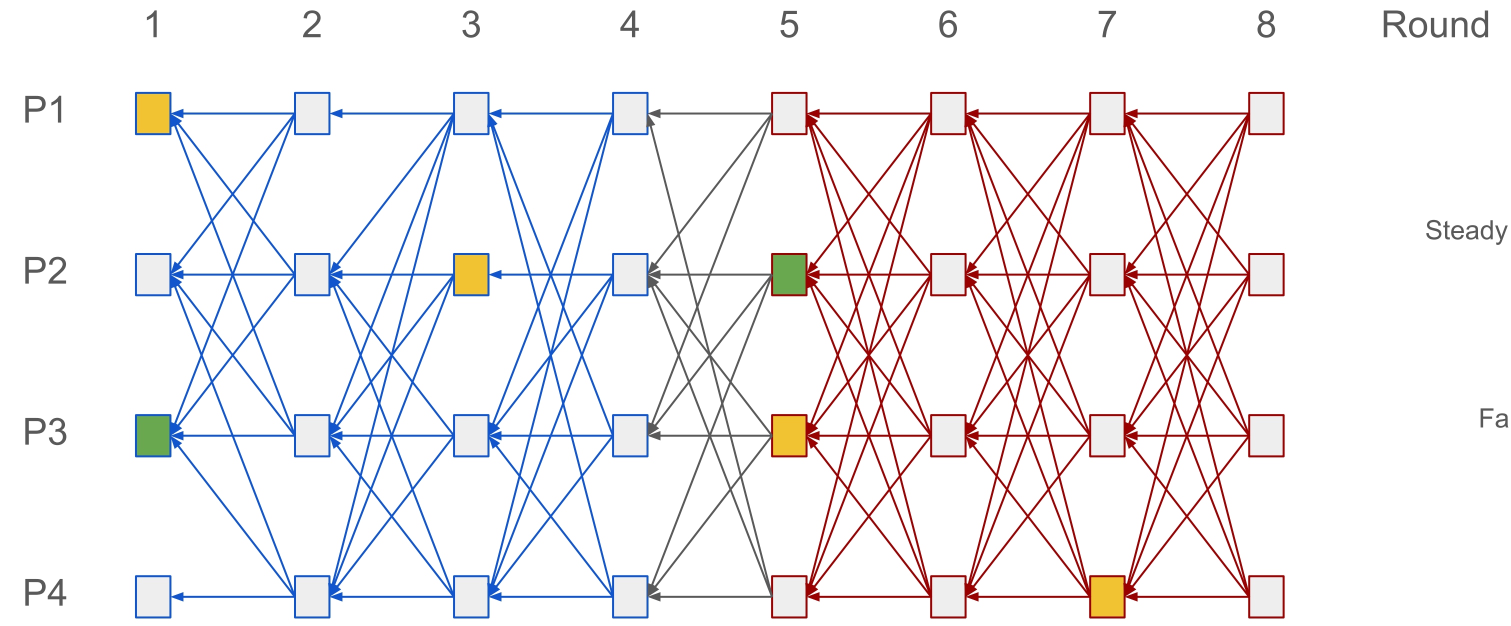
- Global perfect coin
  - Agreement
  - Unpredictability
- Elect the leader retrospectively

Shortcomings:

- Inefficient in the common-case
- Assume some impractical assumptions, such as unbounded memory



BullShark - Overview



Party

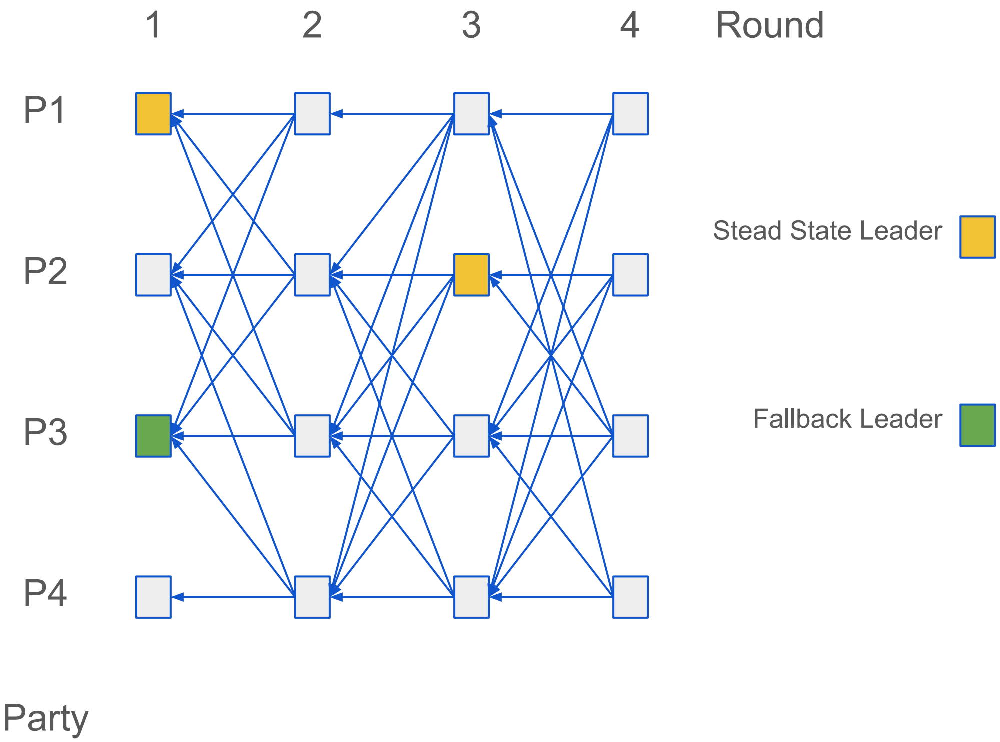
Wave 1
(steady state)

Wave 2
(fallback)

Steady State Leader 🟨

Fallback Leader 🟩

BullShark - Leaders



Three leaders:

- 2 Steady State
- 1 Fallback

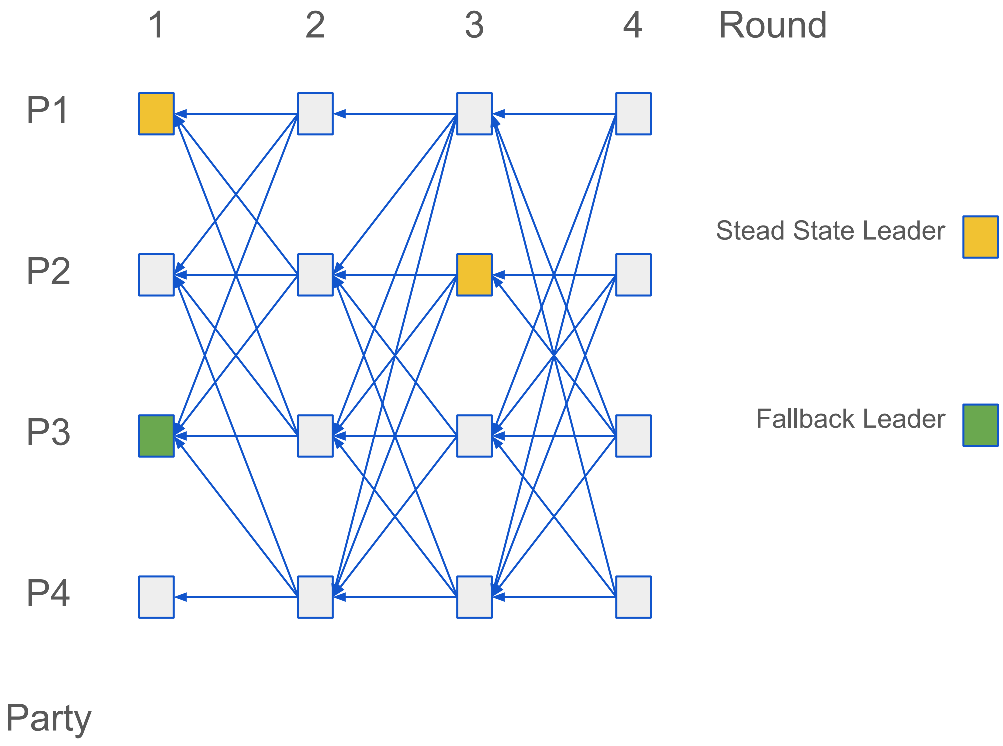
Steady State:

- Synchronous

Fallback:

- Asynchronous
- Similar to DAG Rider

BullShark - Voting Types



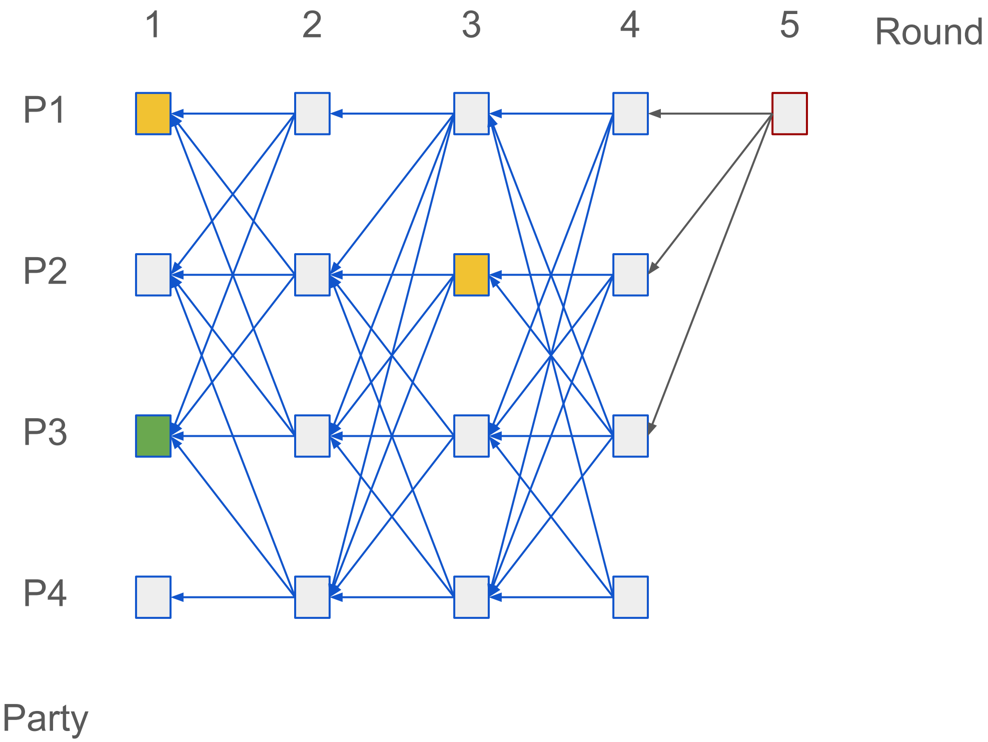
Voting types of a party are determined for the whole wave

Decided at beginning of the wave

Voting type of a party is based off of result of last wave

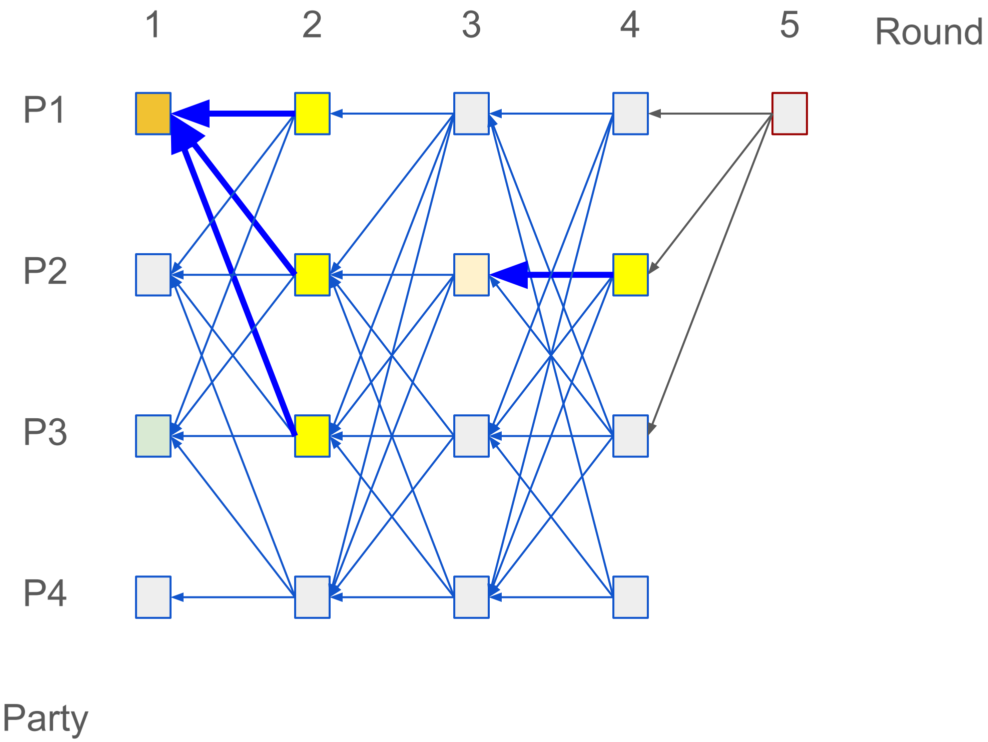
- Any unsuccessful commit in last wave -> fallback
- Else: steady state

BullShark - Votes



We resolve votes for any given in the wave in the first round of the next wave

BullShark - Committing

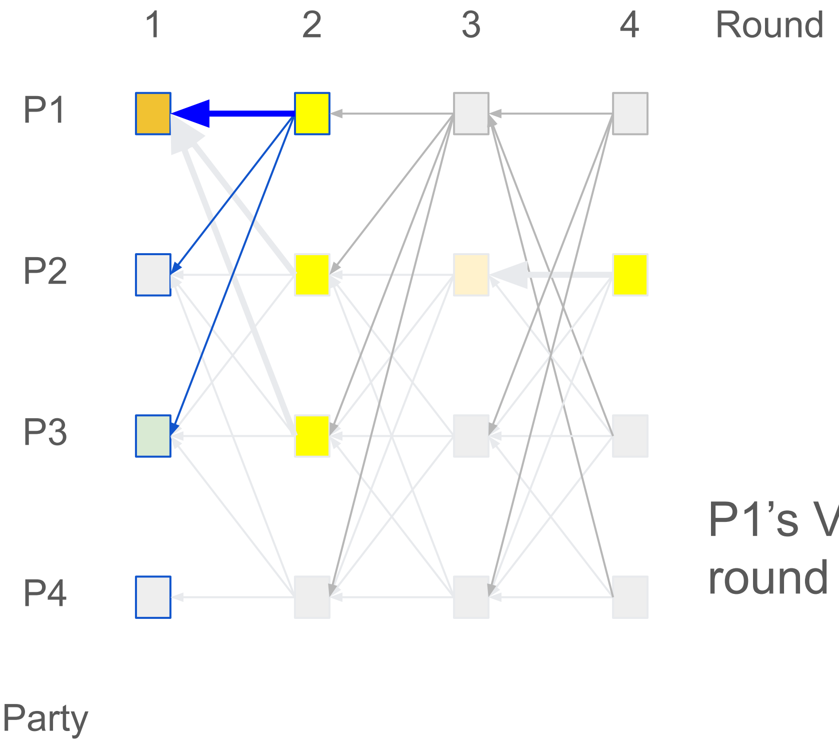


Need 2f + 1 to commit leader

- Only one type of leader can be committed

- For any given party, they will see f + 1 votes on the leader’s proposal, and know which voting type occurred

BullShark - Committing

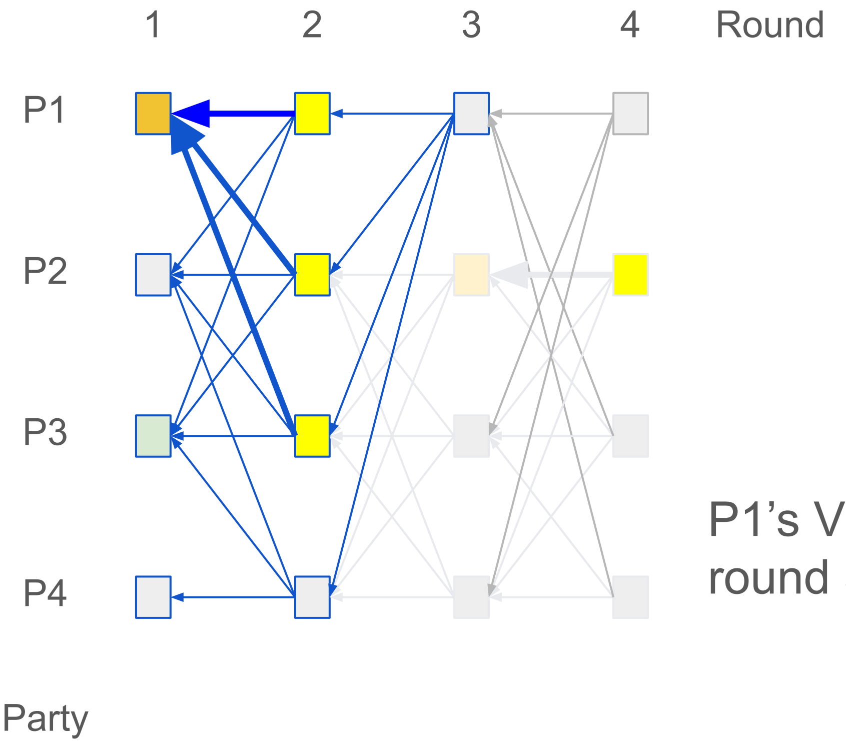


Parties don’t immediately know that a commit occurred

P1’s View on round 2

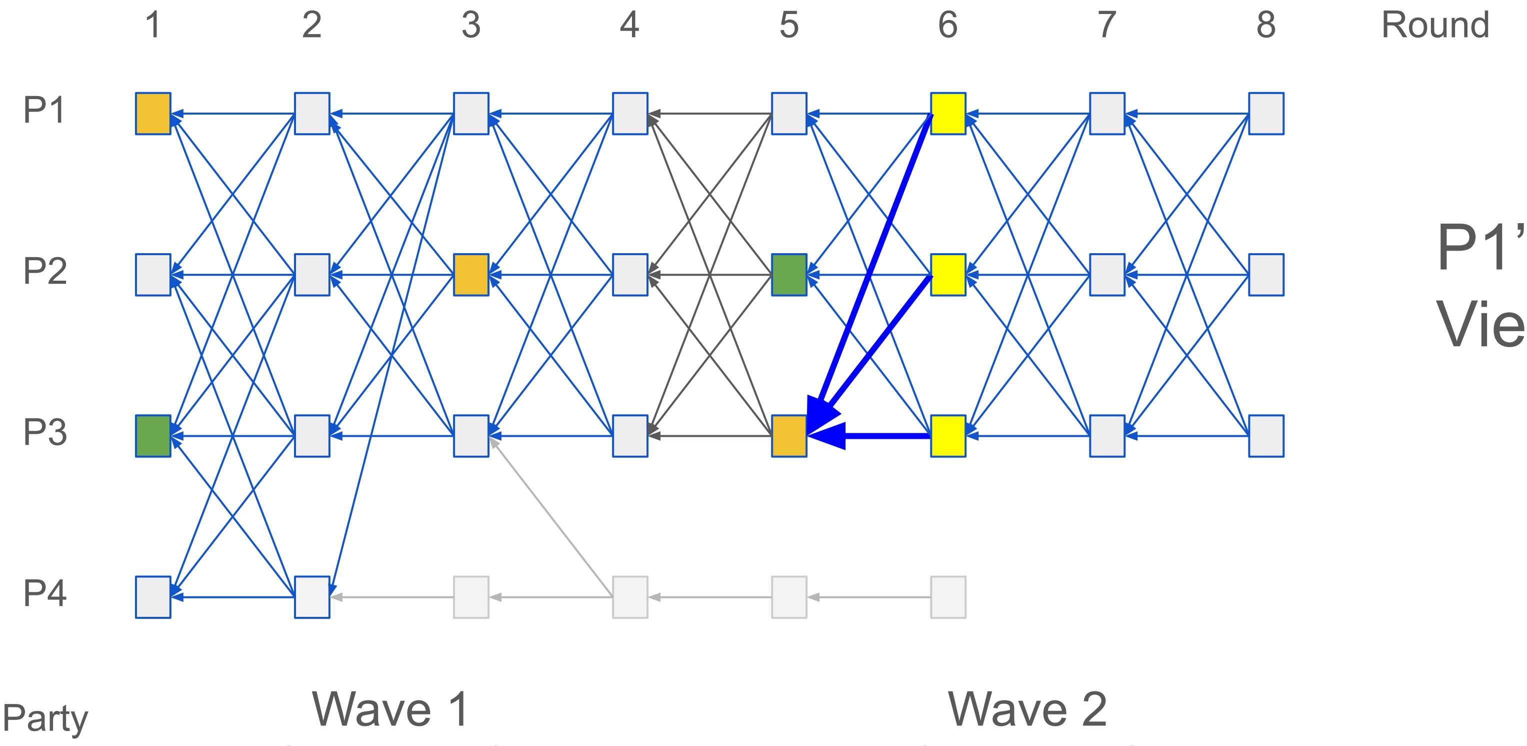
BullShark - Committing

But they will find out eventually



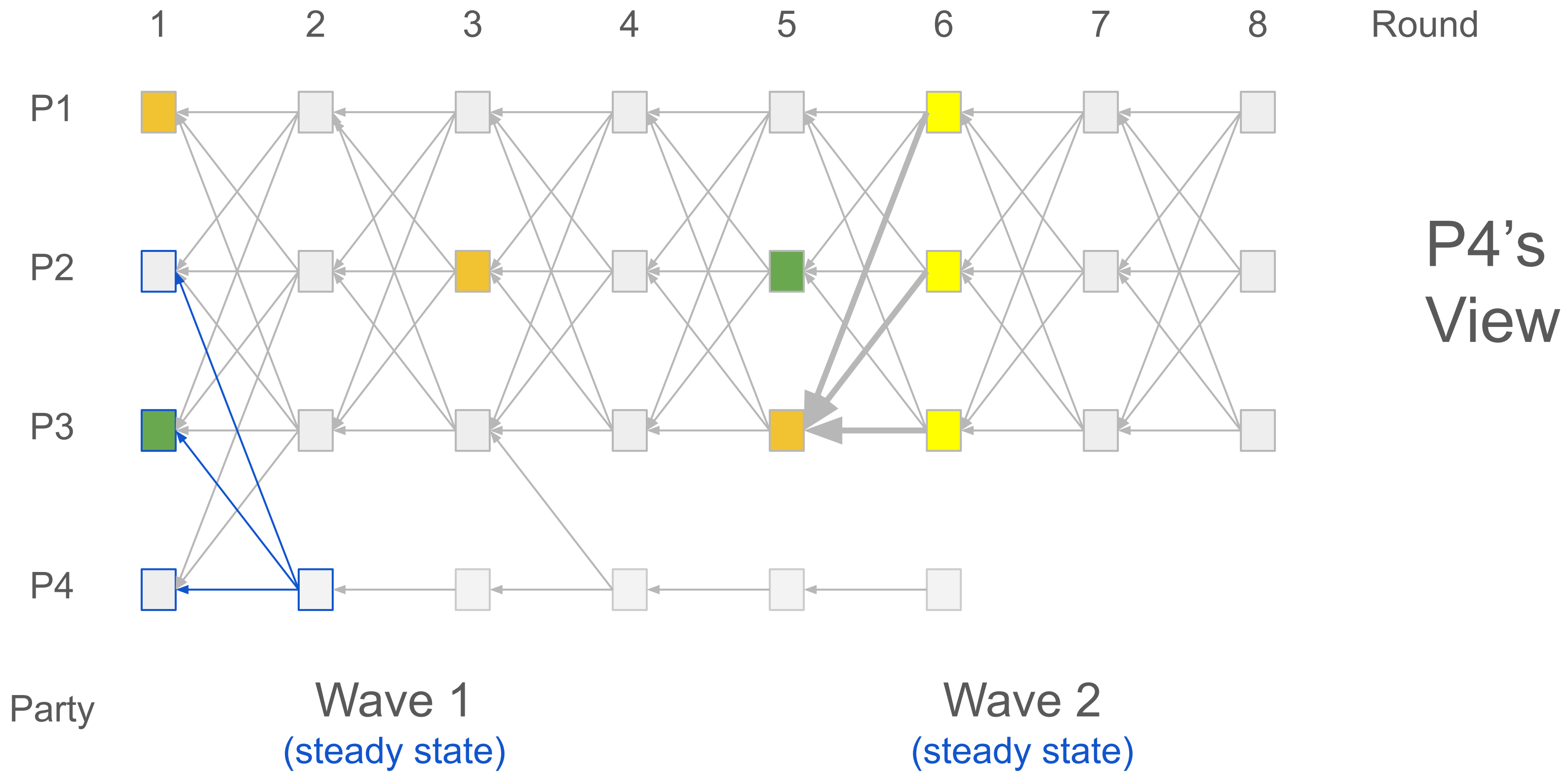
P1's View on round 3

BullShark - Committing Cont.

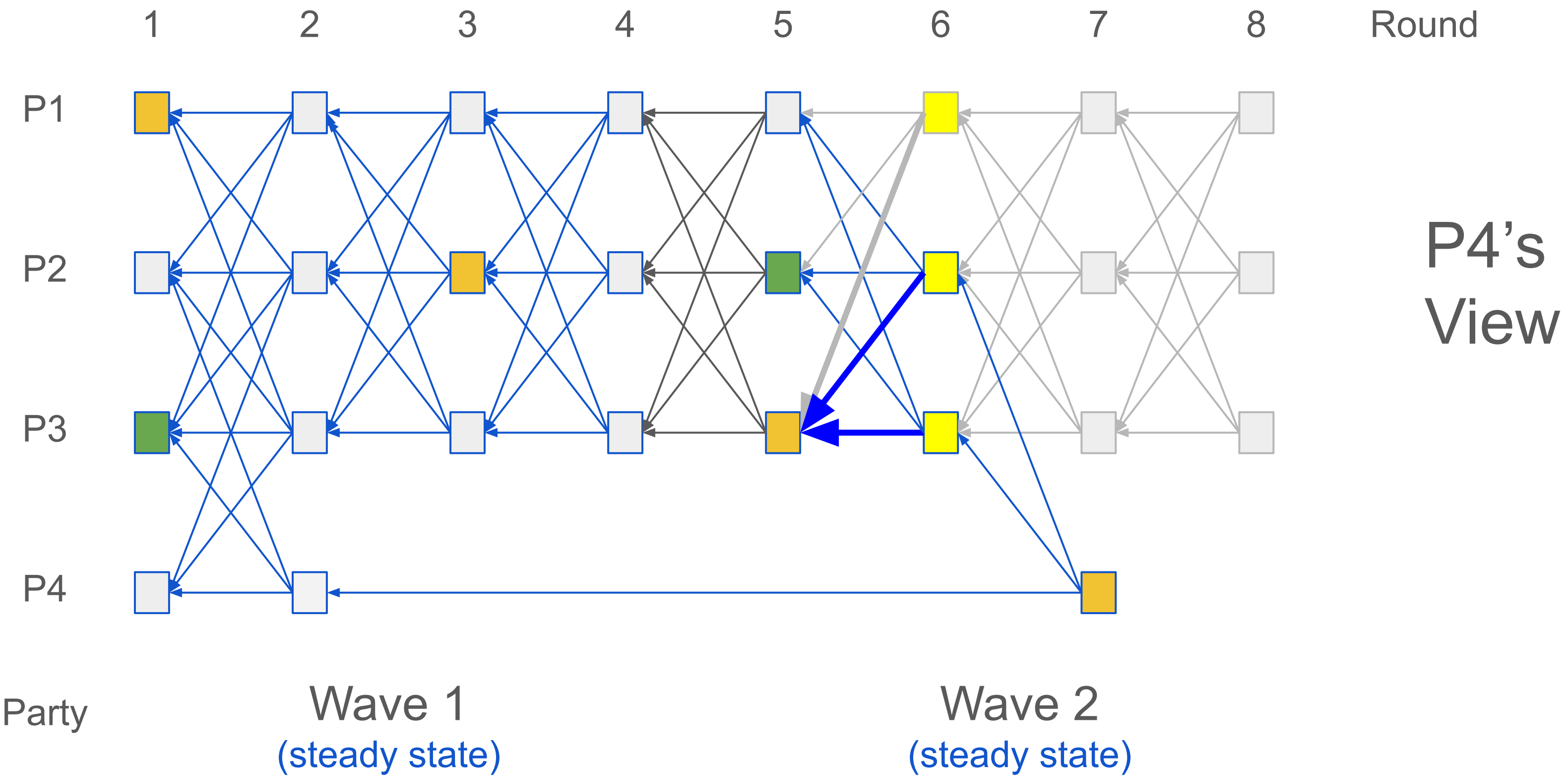


P1’s View

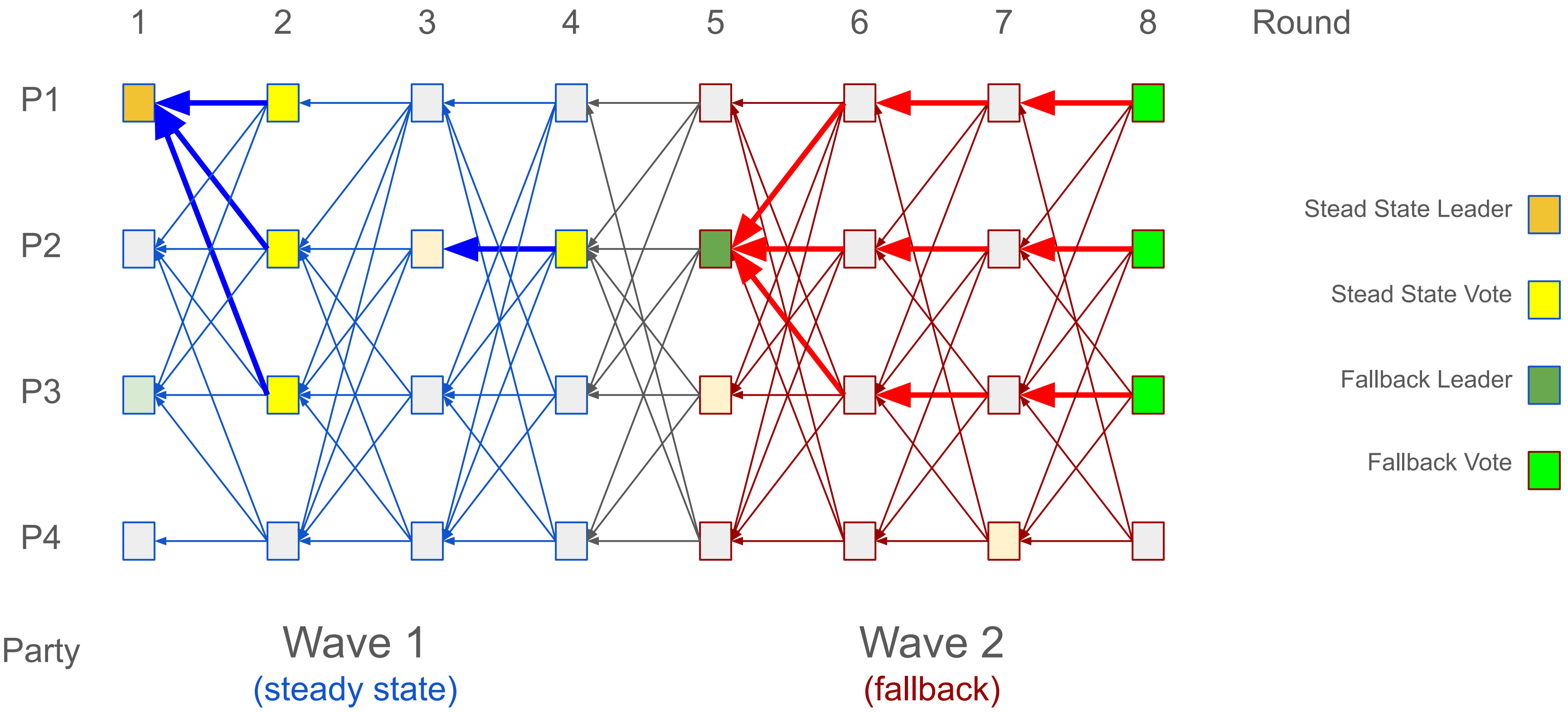
BullShark - Committing Cont.



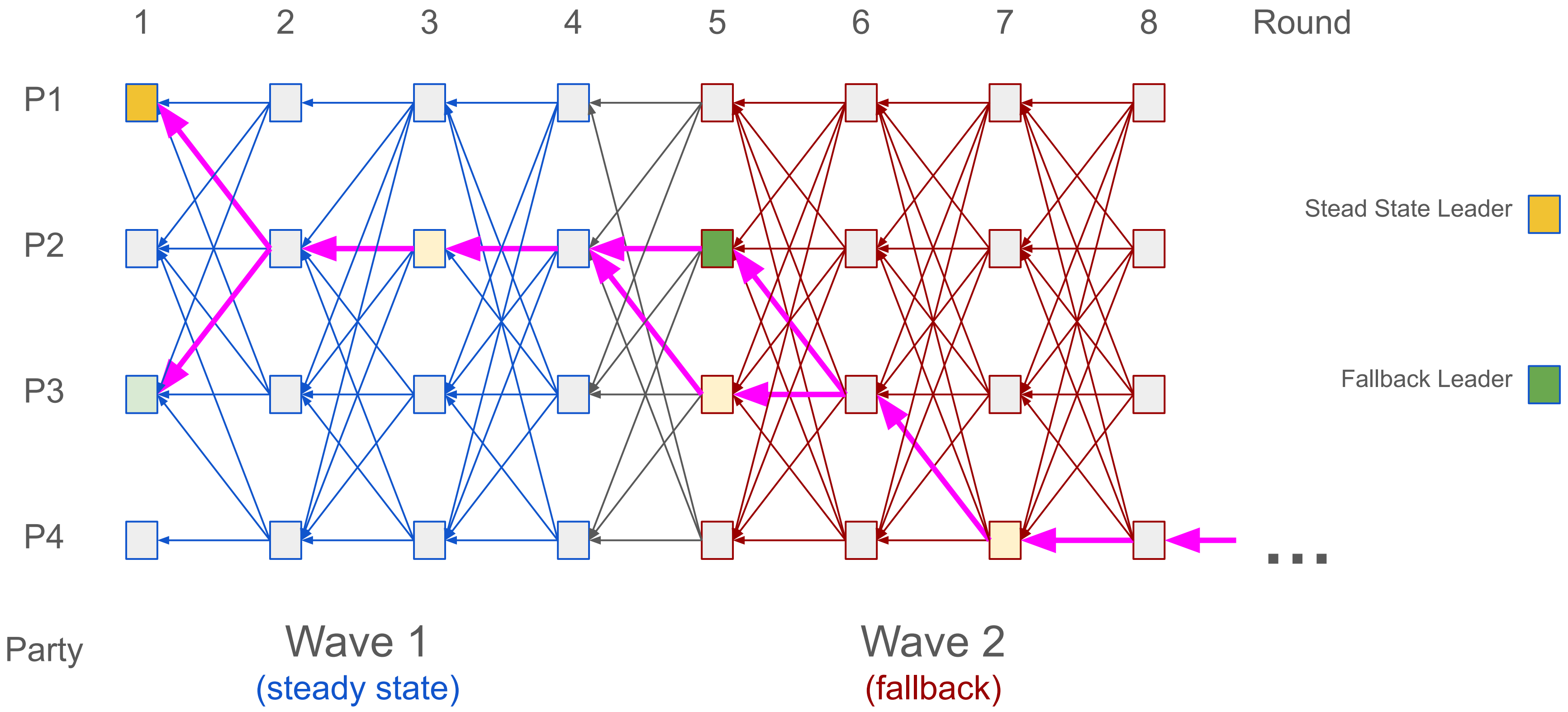
BullShark - Committing Cont.



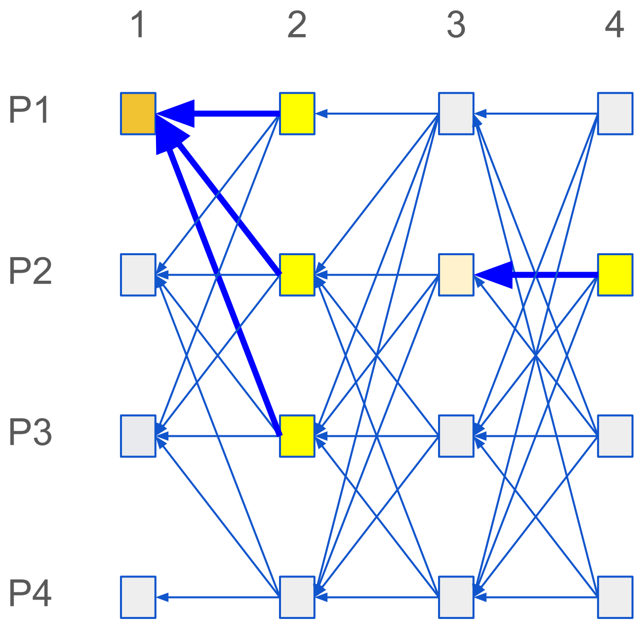
BullShark - Voting and Committing



BullShark - Ordering



Eventually Synchronous BullShark



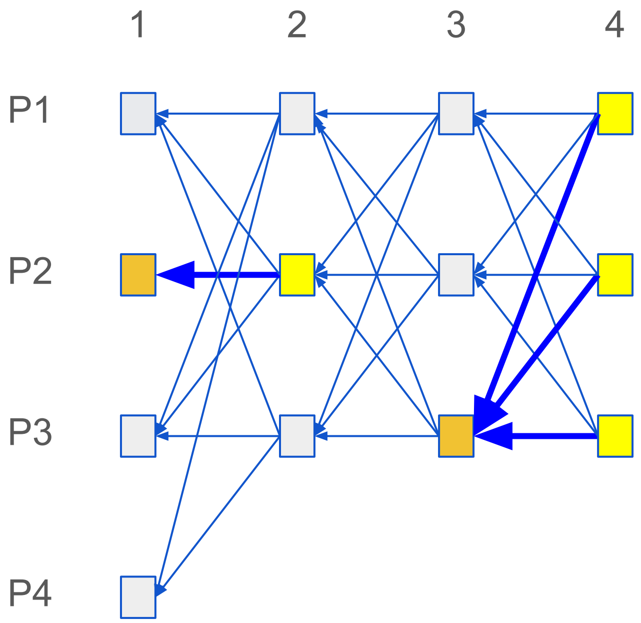
Almost the same as BullShark but with no fallback leaders

Just commit over and over again rather use a fallback

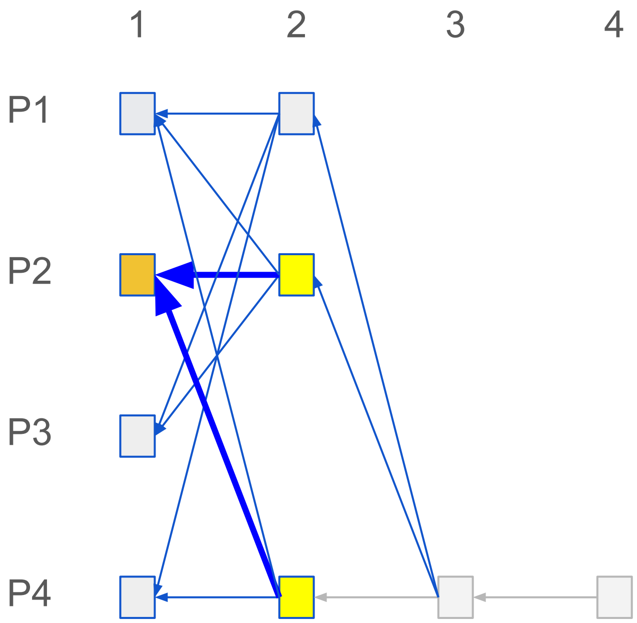
Only need f + 1 vote for commit

Votes are for previous round

Eventually Synchronous BullShark - Committing and Ordering



Wave 1 (P1 view)
(steady state)



Wave 1 (P4 view)
(steady state)

Garbage Collection

Deletion of older vertices

Why do we need garbage collection?

- We need to destroy older vertices since we don’t have infinite memory
    - But we also need to be fair in how we destroy nodes
    - And we can’t destroy nodes that we still need

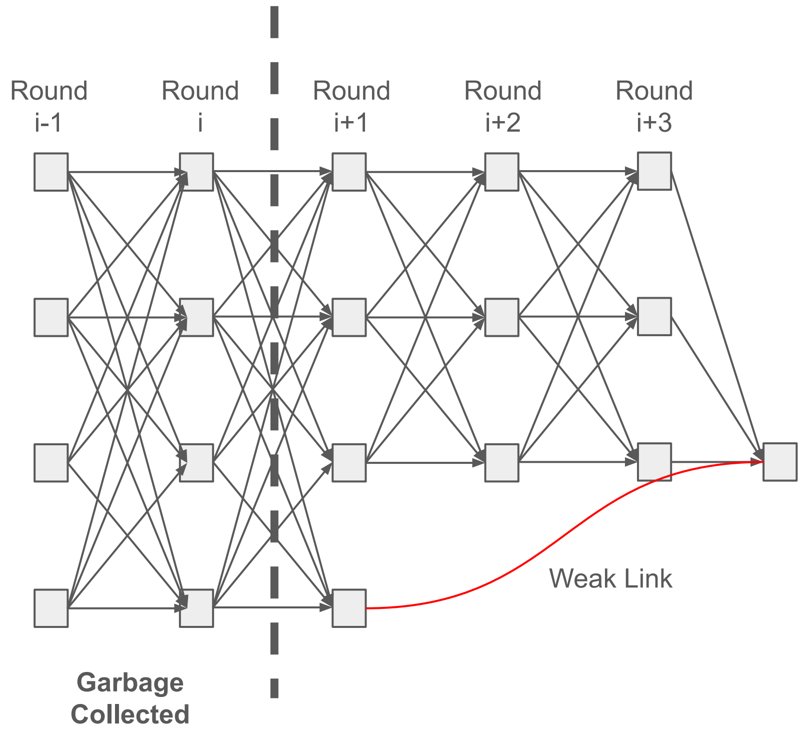
Garbage Collection

Timestamp Assignment: BullShark assigns timestamps to vertices in the Directed Acyclic Graph (DAG)

Garbage Collection Round: BullShark designates a specific round as *GCRound* where a threshold is established for adding new information

Threshold: This is set based on timestamp differences between rounds

Synchronization: Garbage collection is synchronized with a predefined delta in time



Evaluation

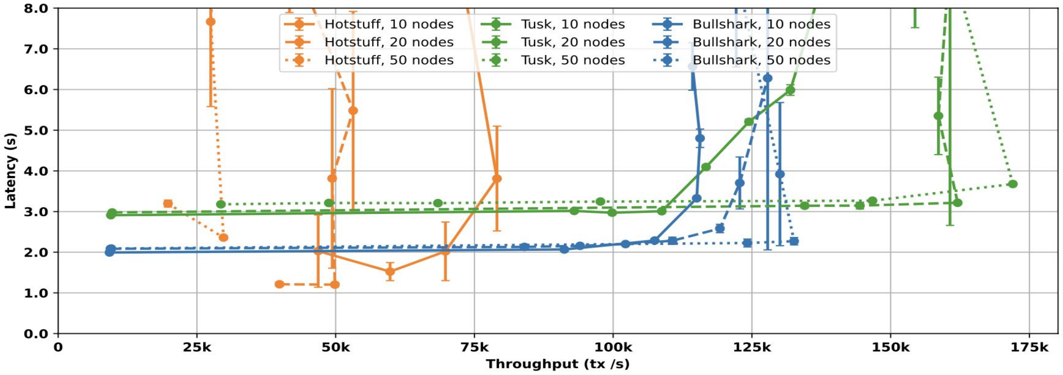
Performance Metrics: Throughput and Latency

Throughput:

- **HotStuff:** Peaks at 70,000 tx/s (10 parties), lowers to 50,000 tx/s (20 parties), and drops further to around 30,000 tx/s (50 parties).
- **Tusk:** Exhibits significantly higher throughput, peaking at 110,000 tx/s (10 parties) and reaching around 160,000 tx/s for larger committees (20 and 50 parties).
- **BullShark:** Strikes a balance, achieving throughput of 110,000 tx/s (10 parties) and 130,000 tx/s (50 parties), over 2x higher than HotStuff.

Latency:

- **HotStuff:** Low latency, approximately 2 seconds.
- **Tusk:** Requires 4 DAG rounds, resulting in higher latency.
- **BullShark:** Achieves low latency at around 2 seconds, comparable to HotStuff and 33% lower than Tusk.



References

"All You Need is DAG"

https://arxiv.org/pdf/2102.08325.pdf

"Bullshark: DAG BFT Protocols Made Practical"

https://arxiv.org/pdf/2201.05677.pdf

"Bullshark: The Partially Synchronous Version"

https://sonnino.com/papers/bullshark-simple.pdf

https://decentralizedthoughts.github.io/2022-06-28-DAG-meets-BFT/

https://blog.chain.link/bft-on-a-dag/

"Practical Byzantine Fault Tolerance"

https://pmg.csail.mit.edu/papers/osdi99.pdf