Bullshark

DAG BFT Protocols Made Practical

Alberto Sonnino

Byzantine Fault Tolerance

Consensus on top of Narwhal

Goal of this project

Simple

- Zero-message overhead
- No view-change
- No common-coin

Performant

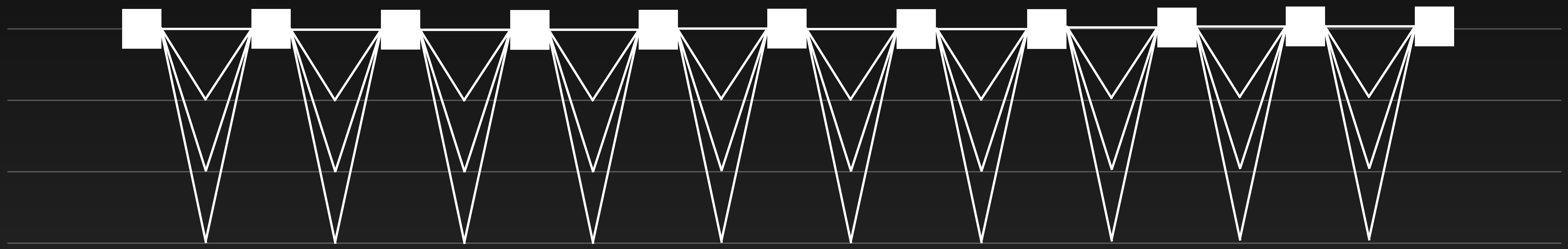
- Take advantage of Narwhal
- Exploit periods of synchrony

Current Designs

- Monolithic protocol sharing transaction data as part of the consensus
- Optimize overall message complexity of the consensus protocol
- Complex & Error-prone view-change protocol

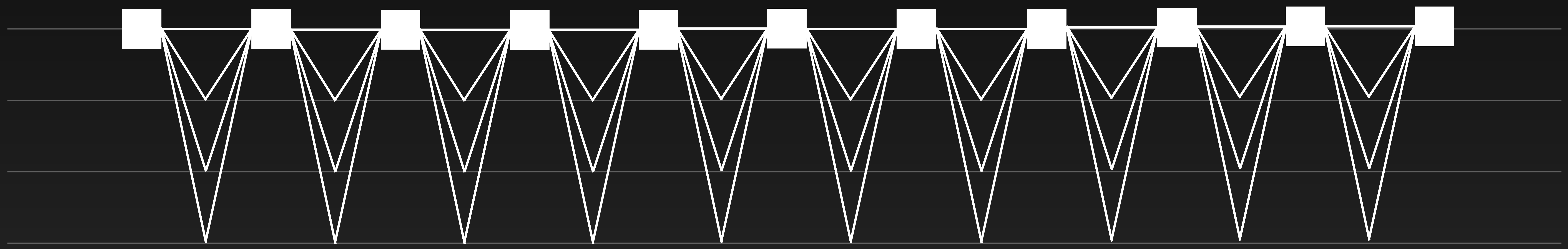
Current Designs

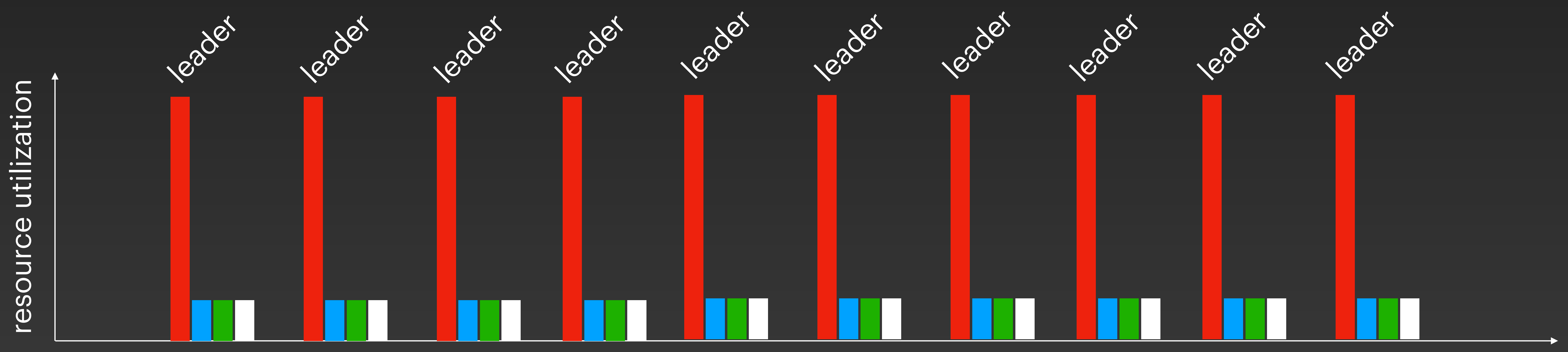
Typical leader-based protocols



Current Designs

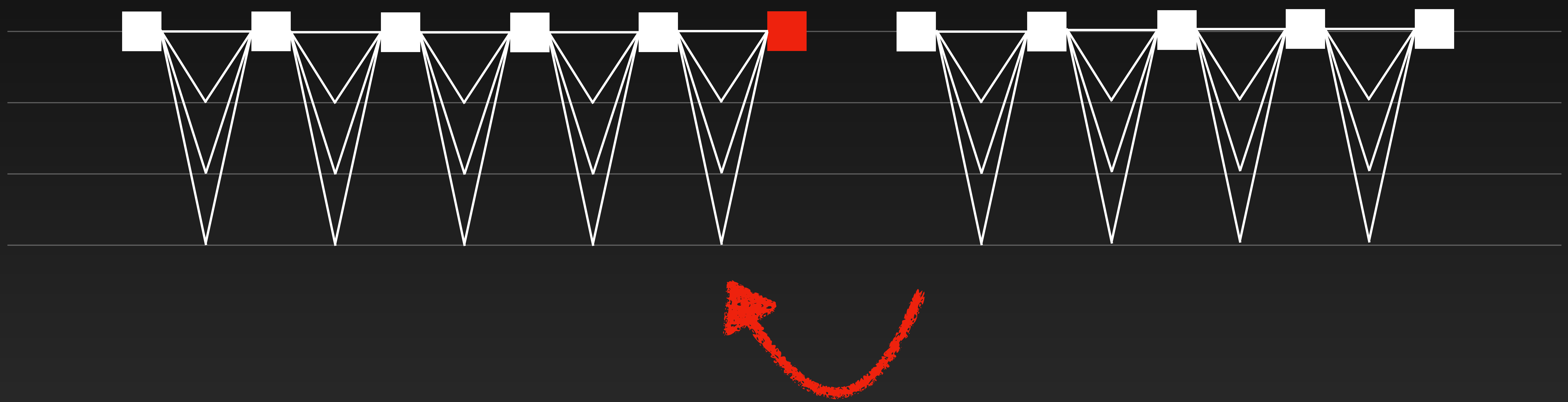
Typical leader-based protocols





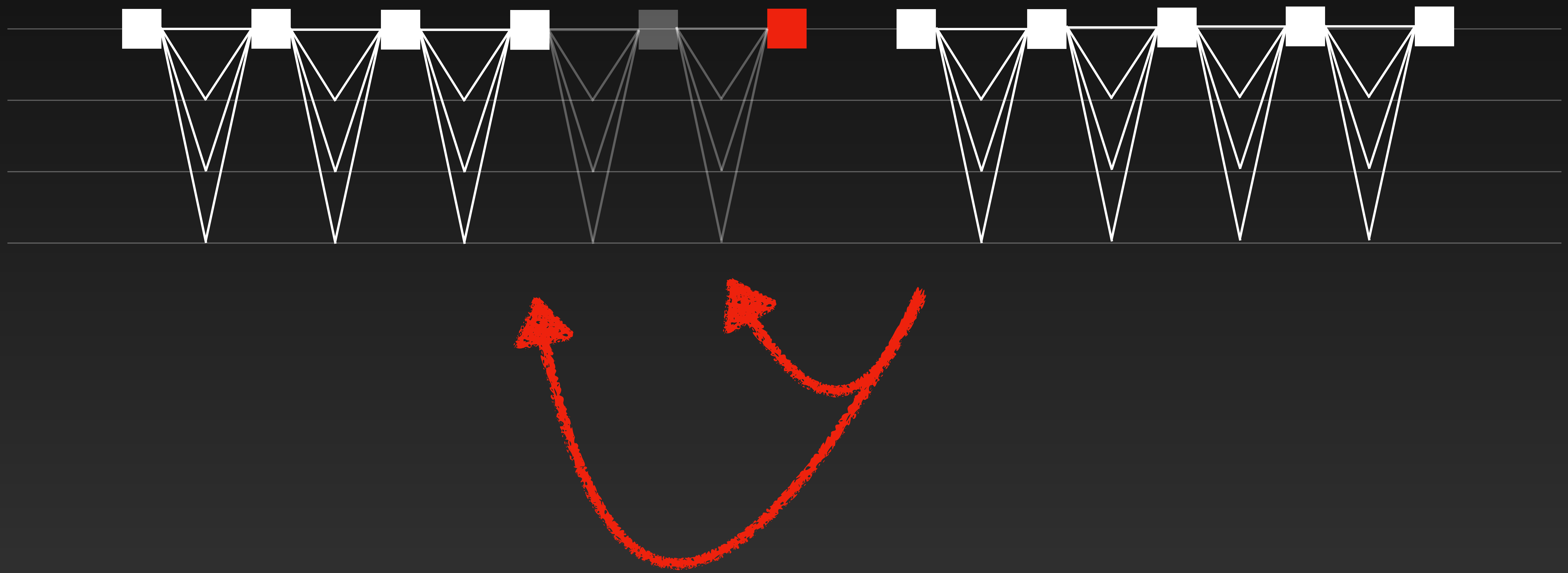
Current Designs

Typical leader-based protocols



Current Designs

Typical leader-based protocols



Narwhal

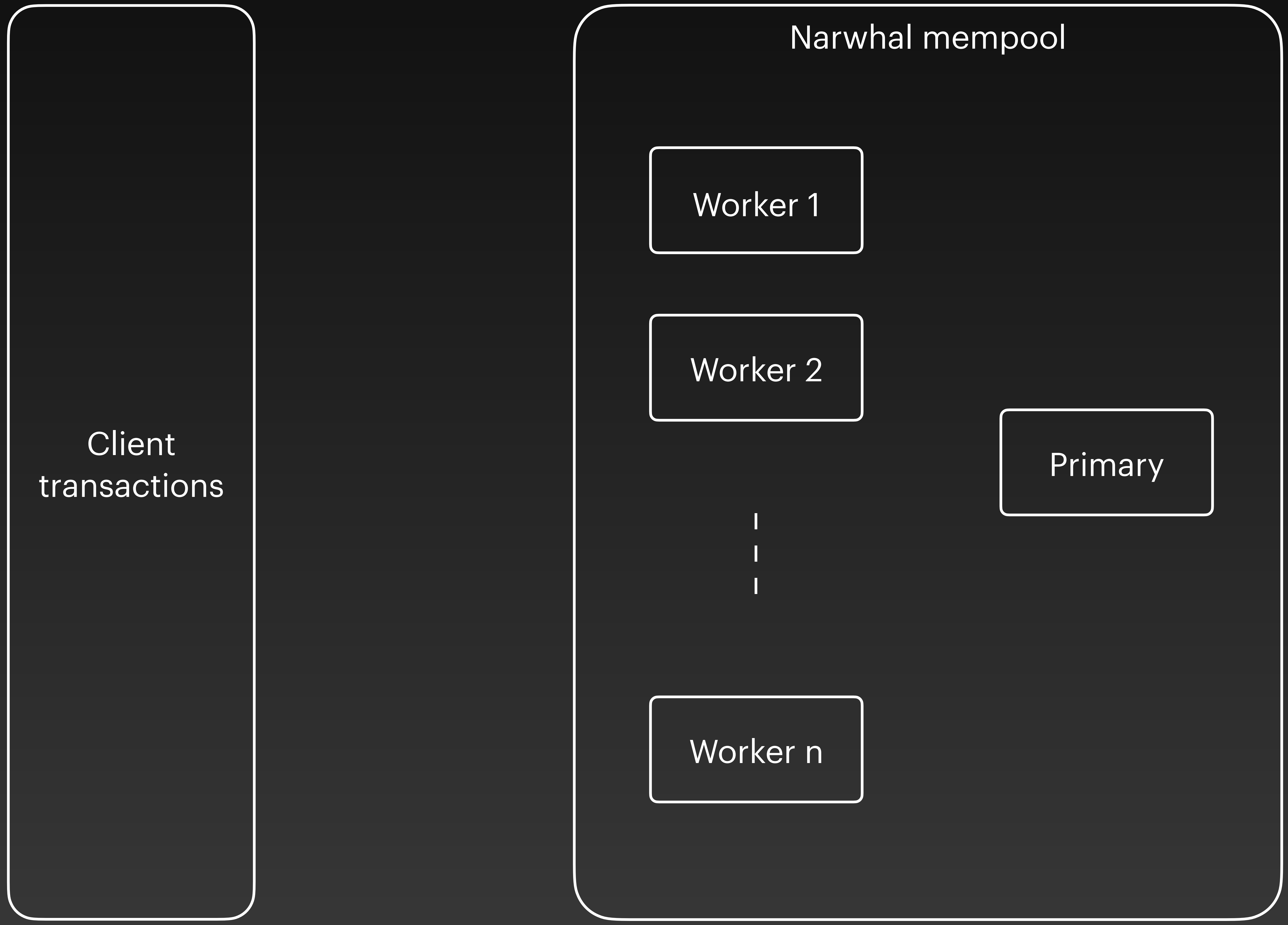
Dag-based mempool

The mempool is the key

Reaching consensus on metadata is cheap

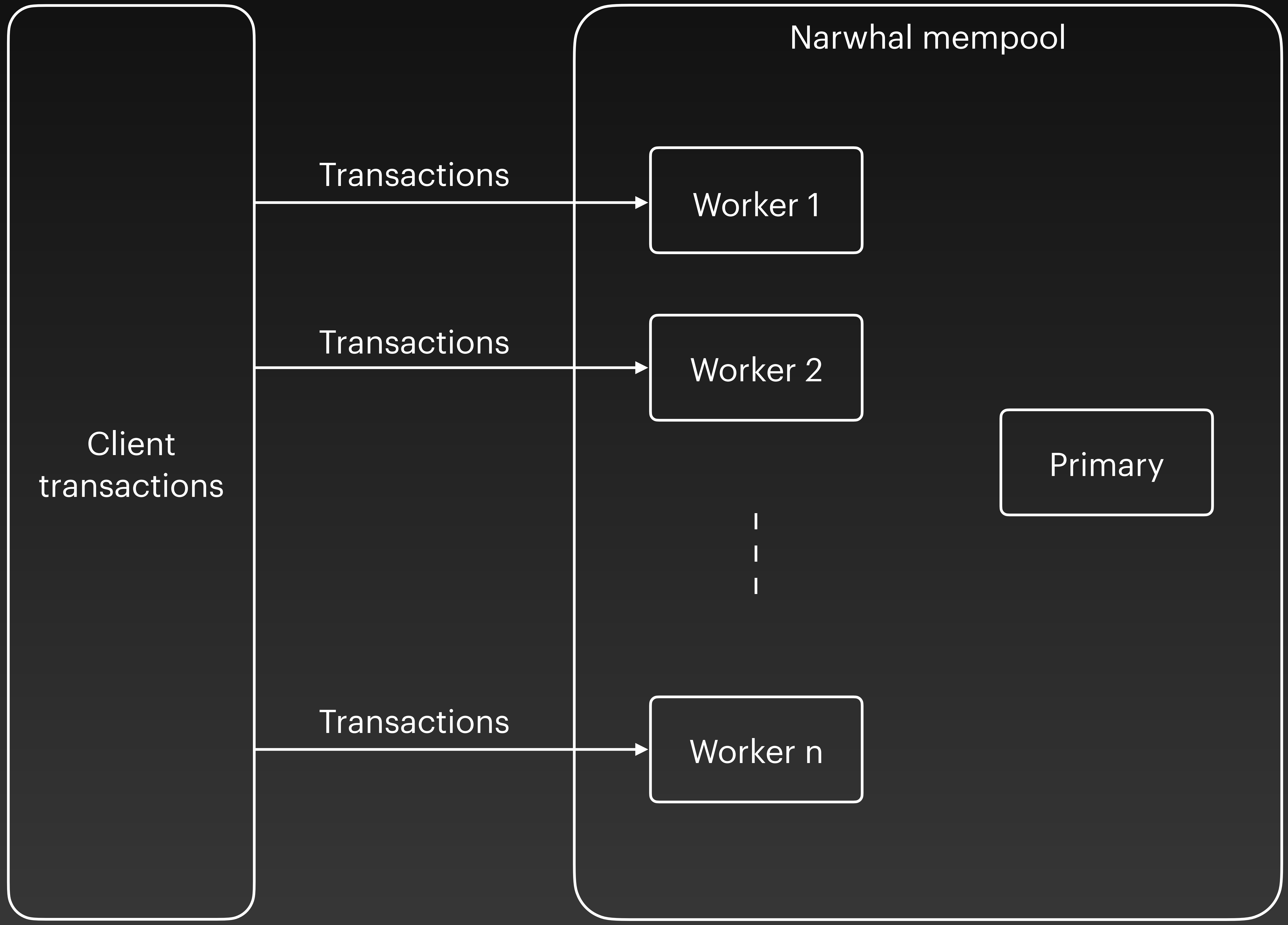
Narwhal

The workers and the primary



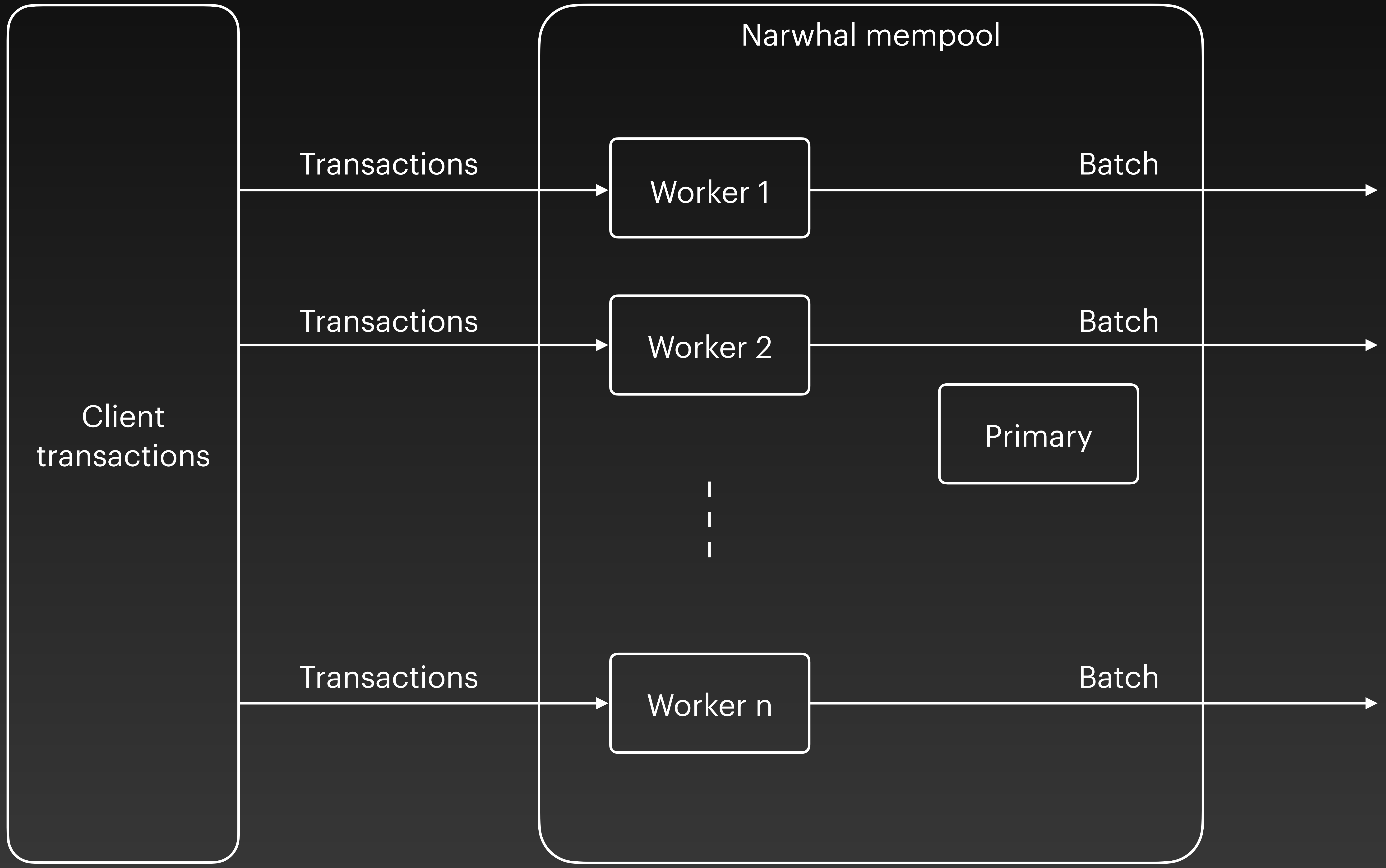
Narwhal

The workers and the primary



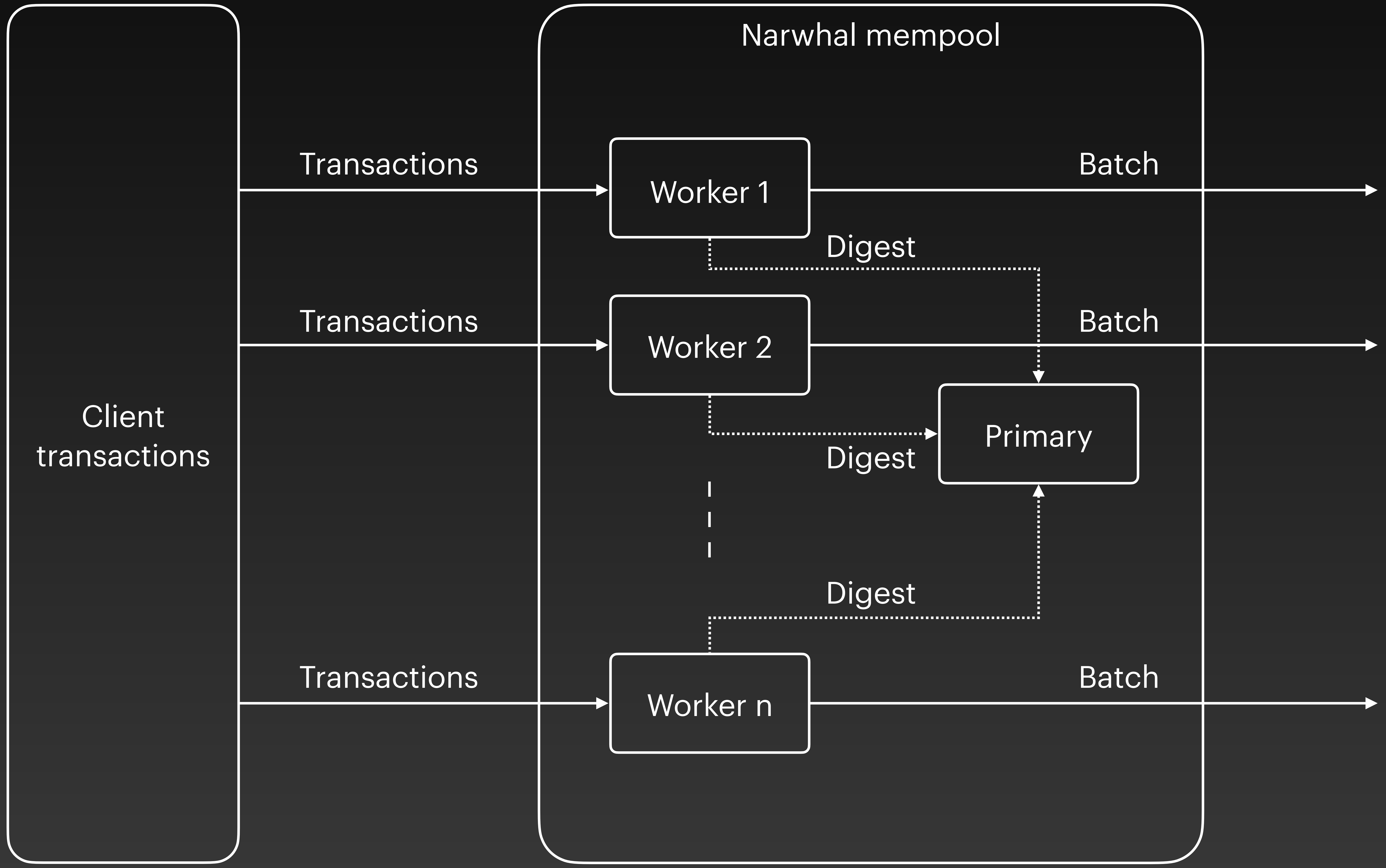
Narwhal

The workers and the primary



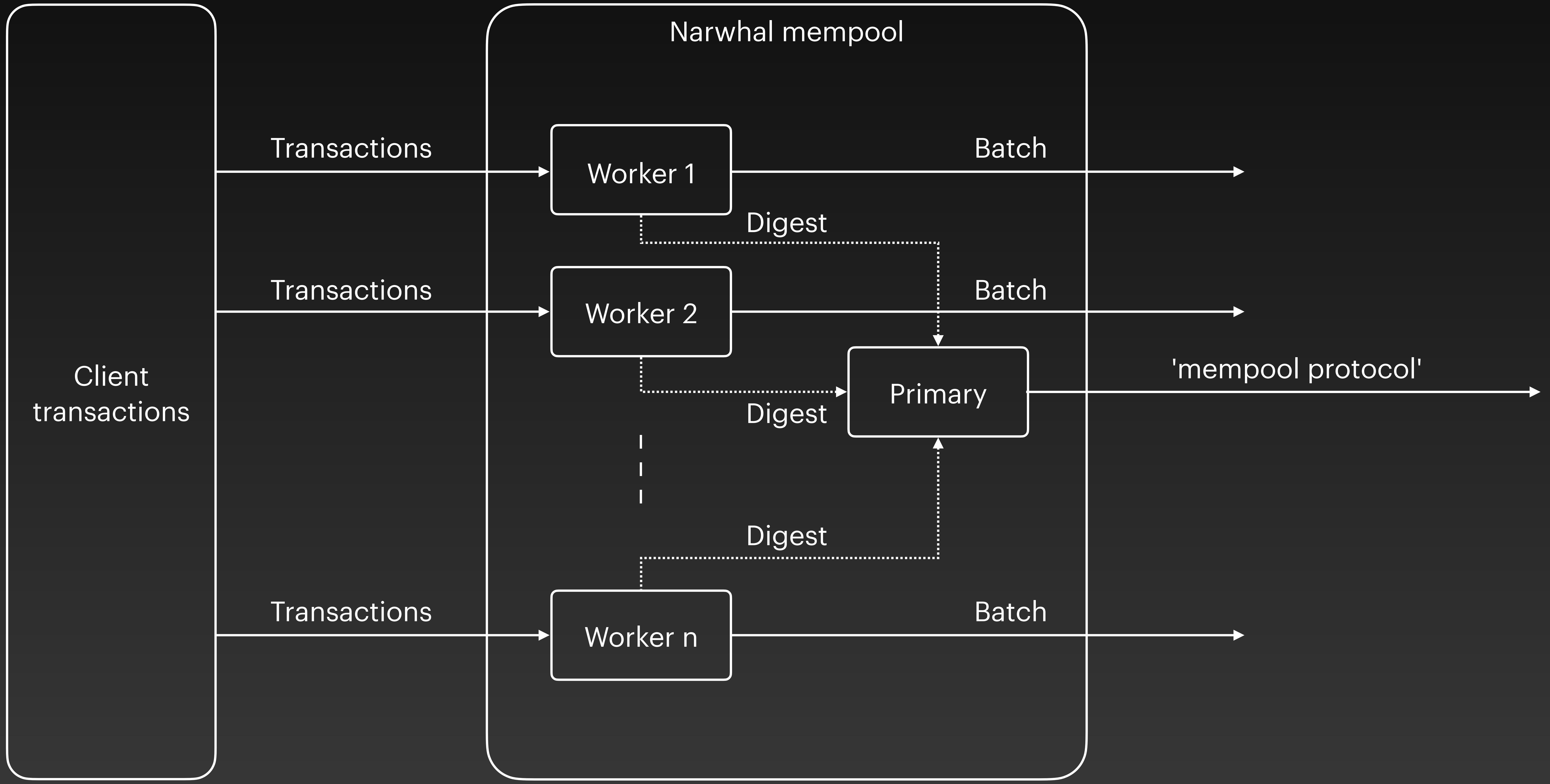
Narwhal

The workers and the primary



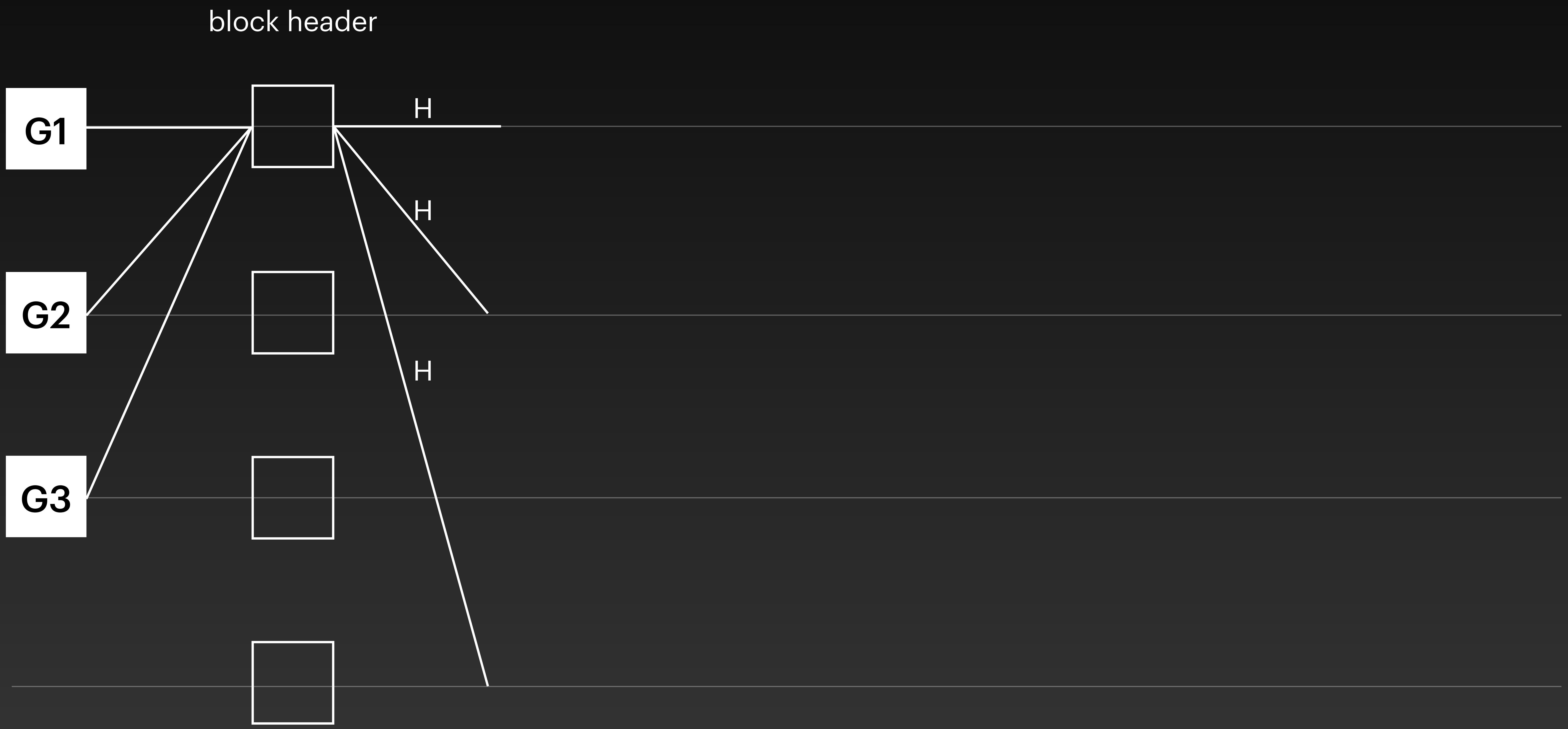
Narwhal

The workers and the primary



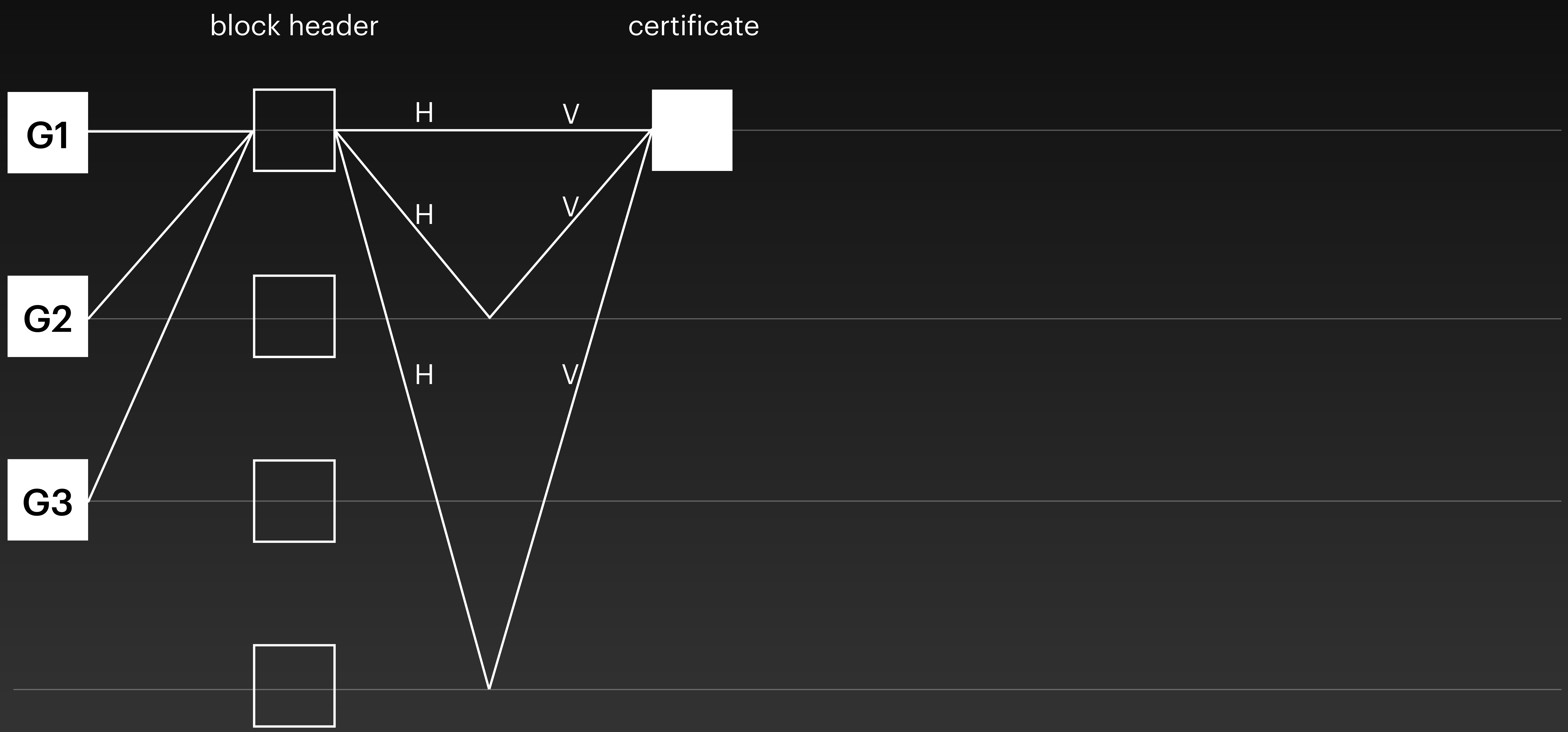
Narwhal

The primary machine

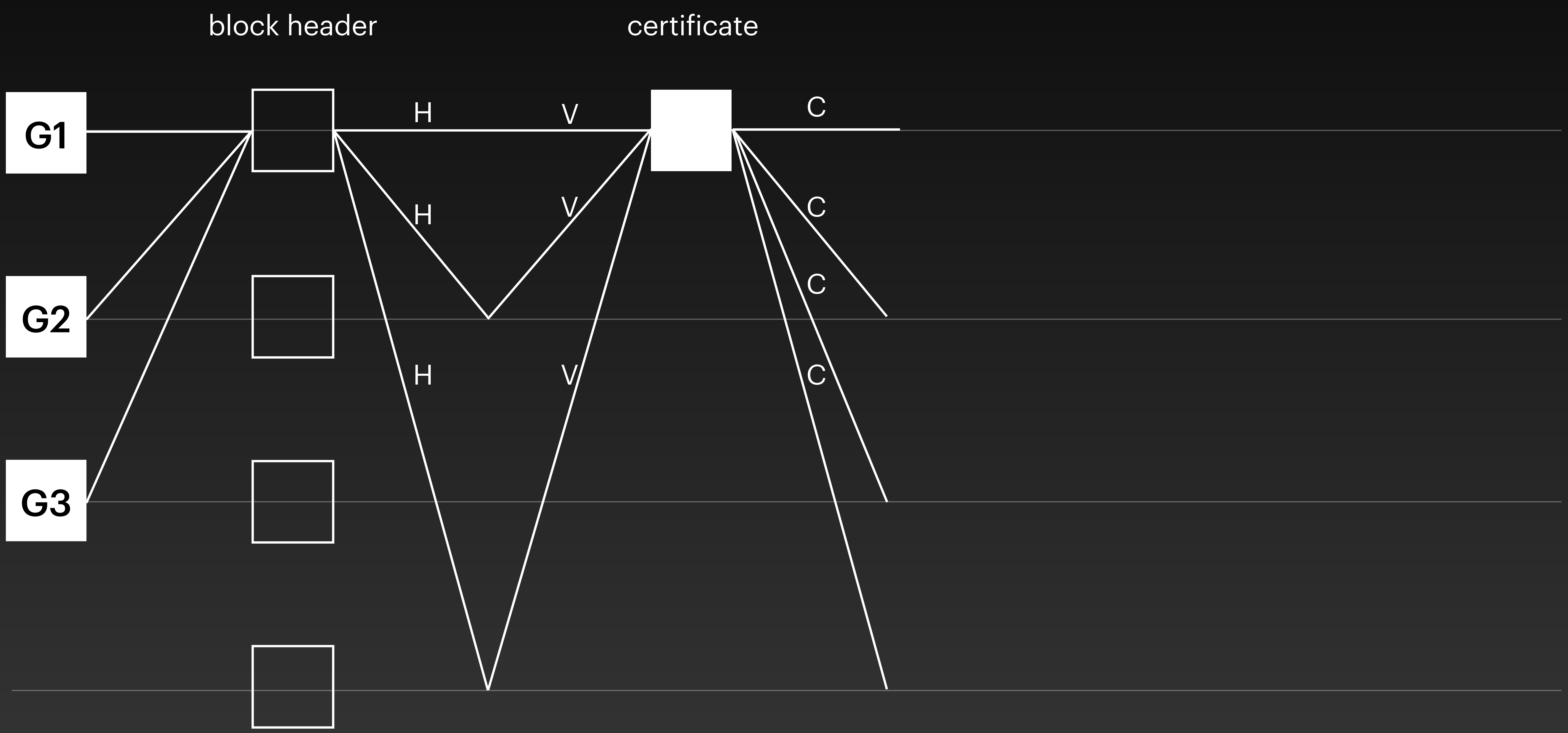


Narwhal

The primary machine

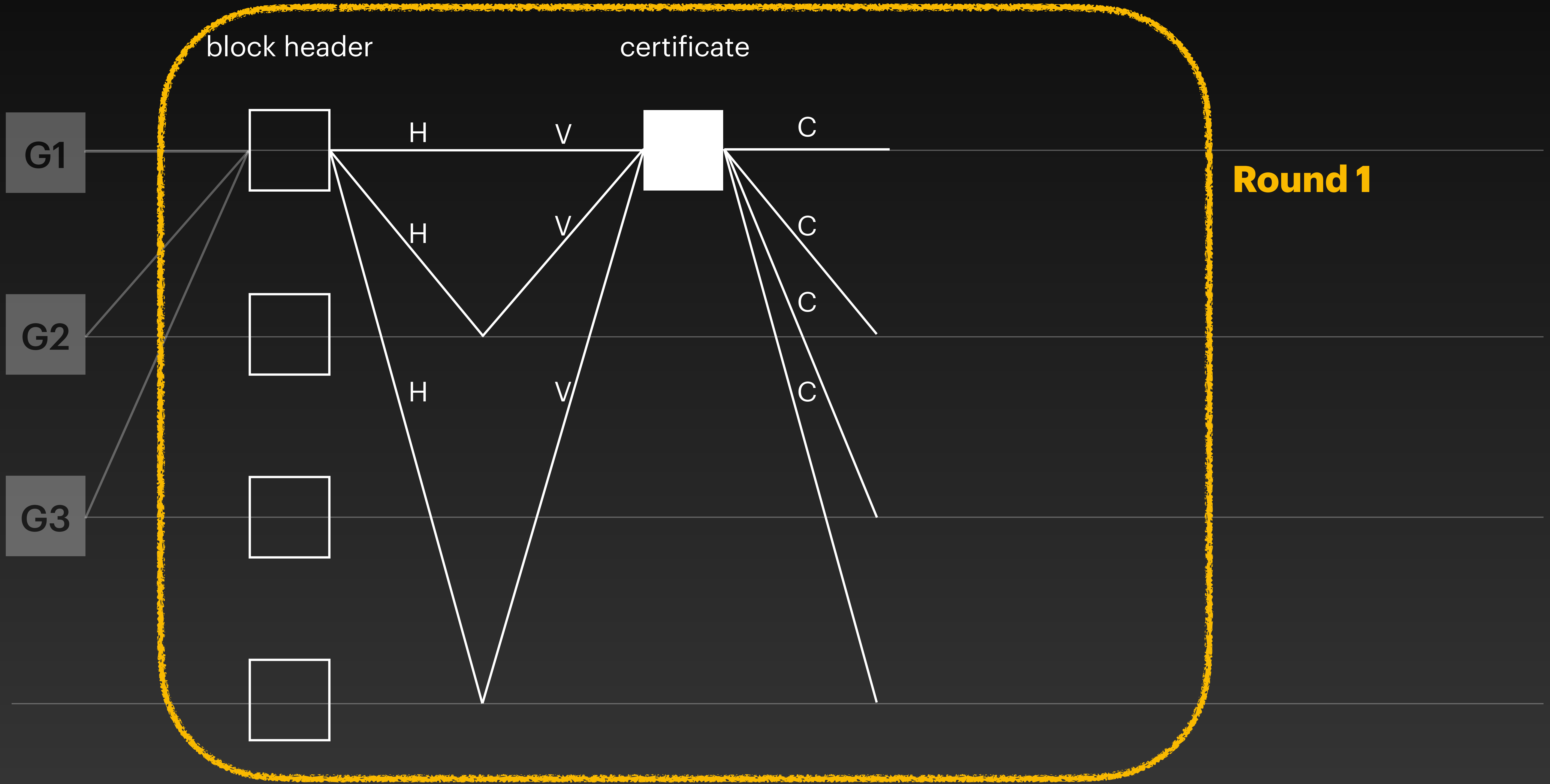


Narwhal
The primary machine



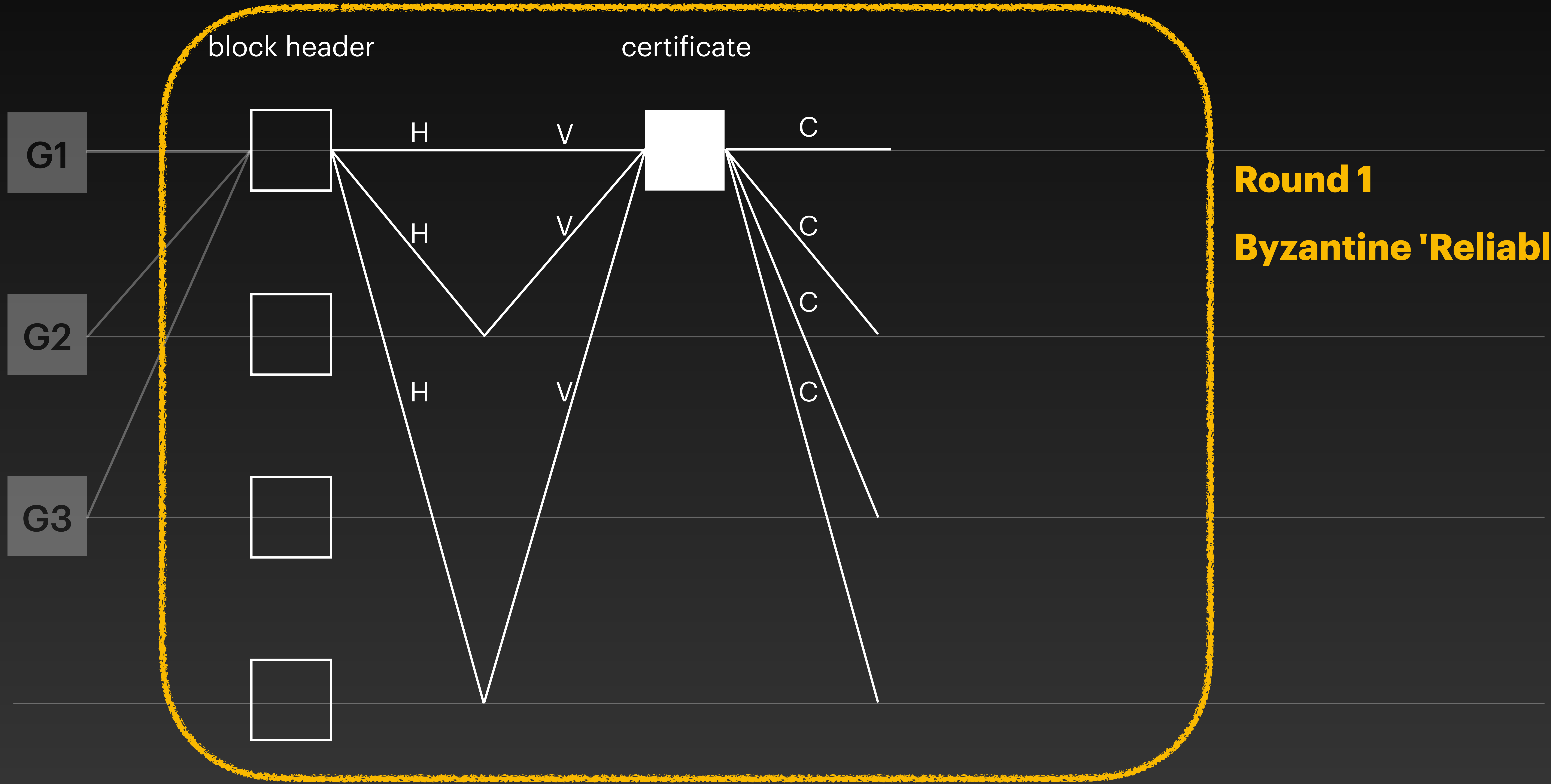
Narwhal

The primary machine



Narwhal

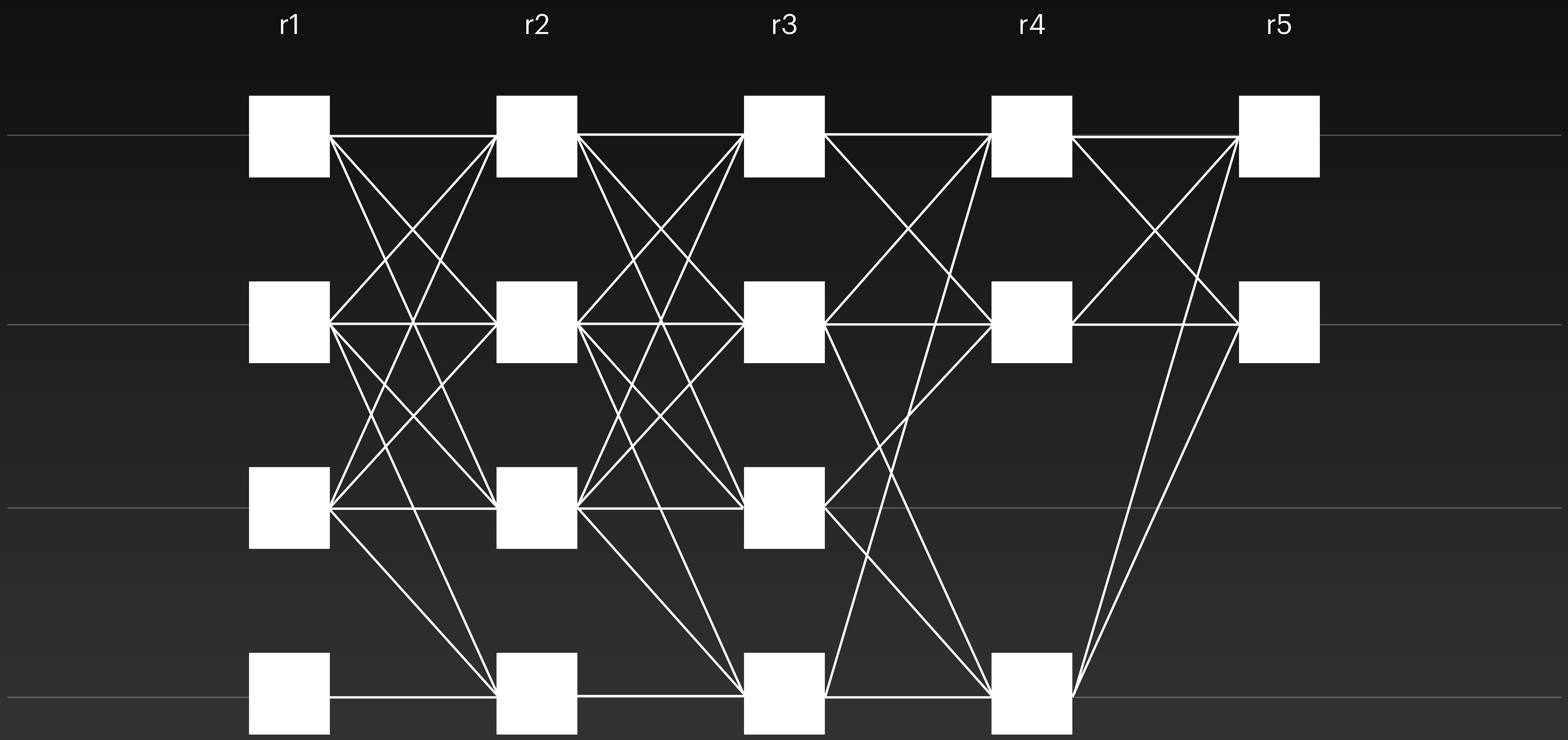
The primary machine



Round 1

Byzantine 'Reliable' Broadcast

Narwhal
The primary machine

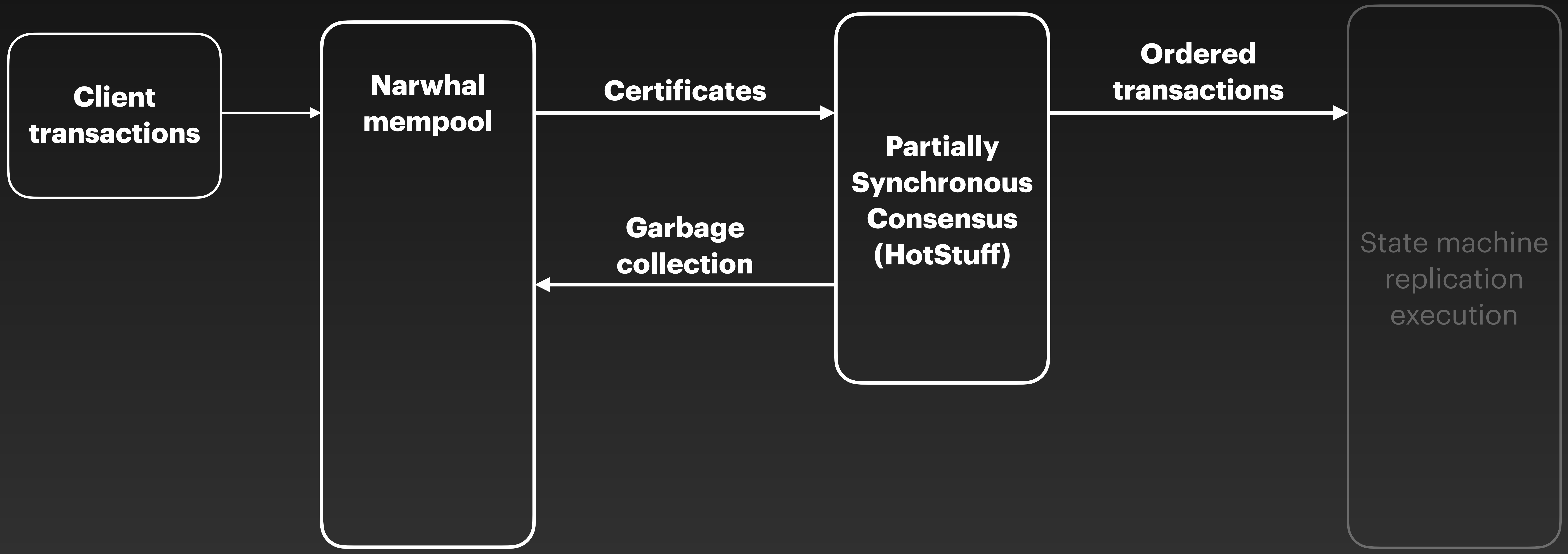


HotStuff on Steroids

Just by replacing the mempool

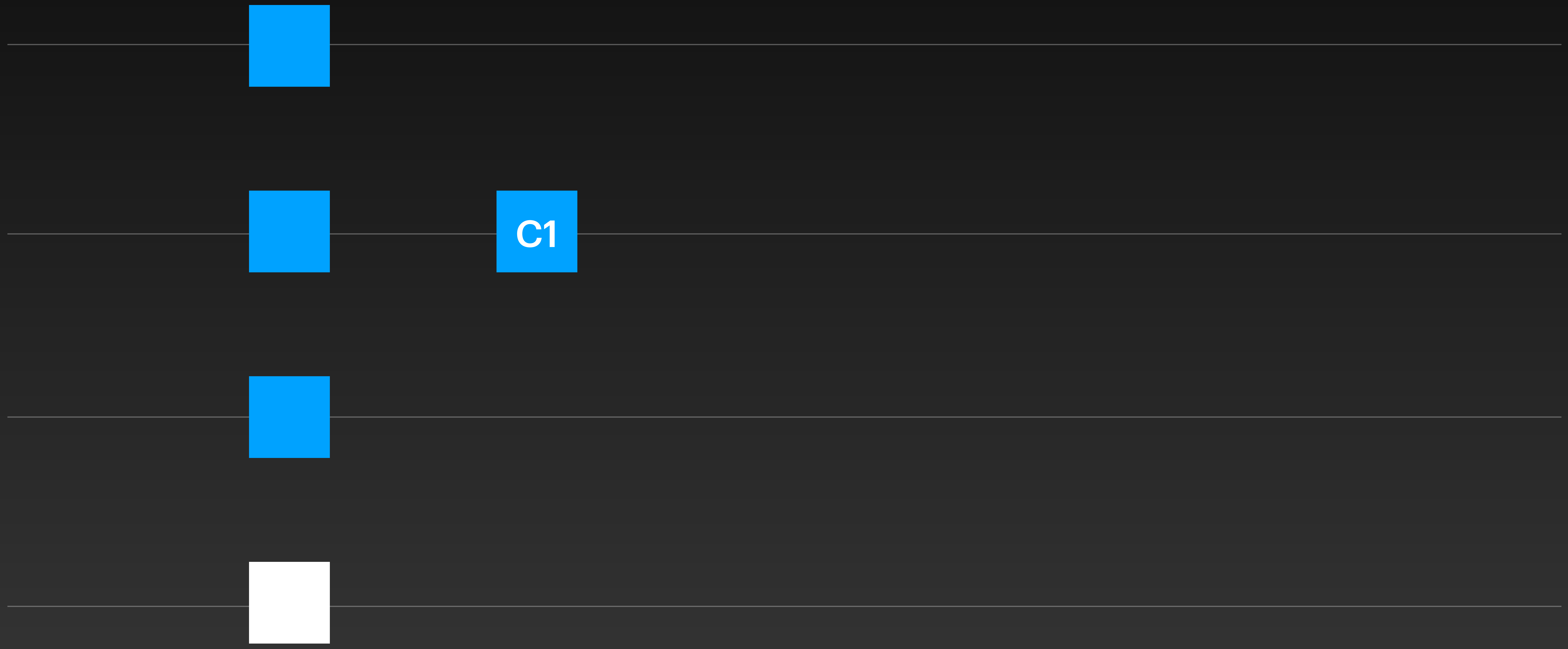
HotStuff on Narwhal

Overview



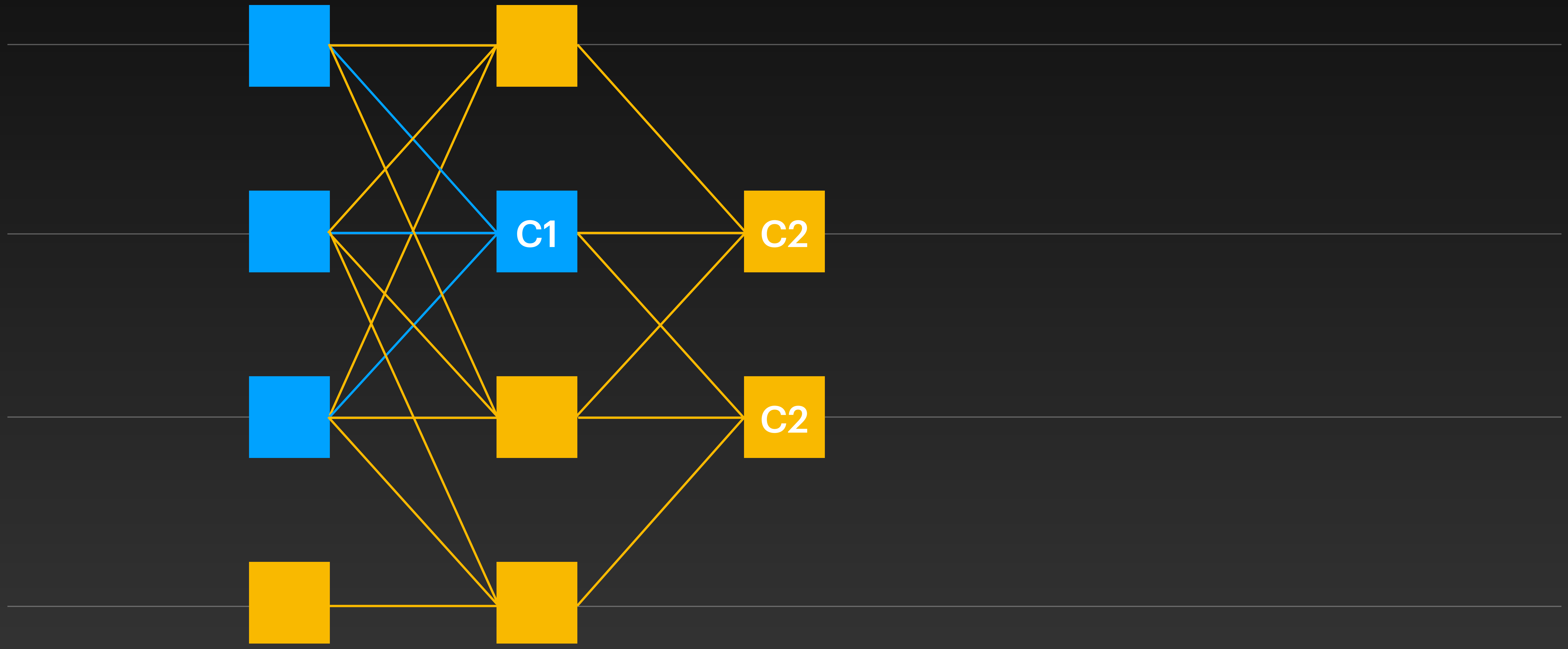
HotStuff on Narwhal

Enhanced commit rule



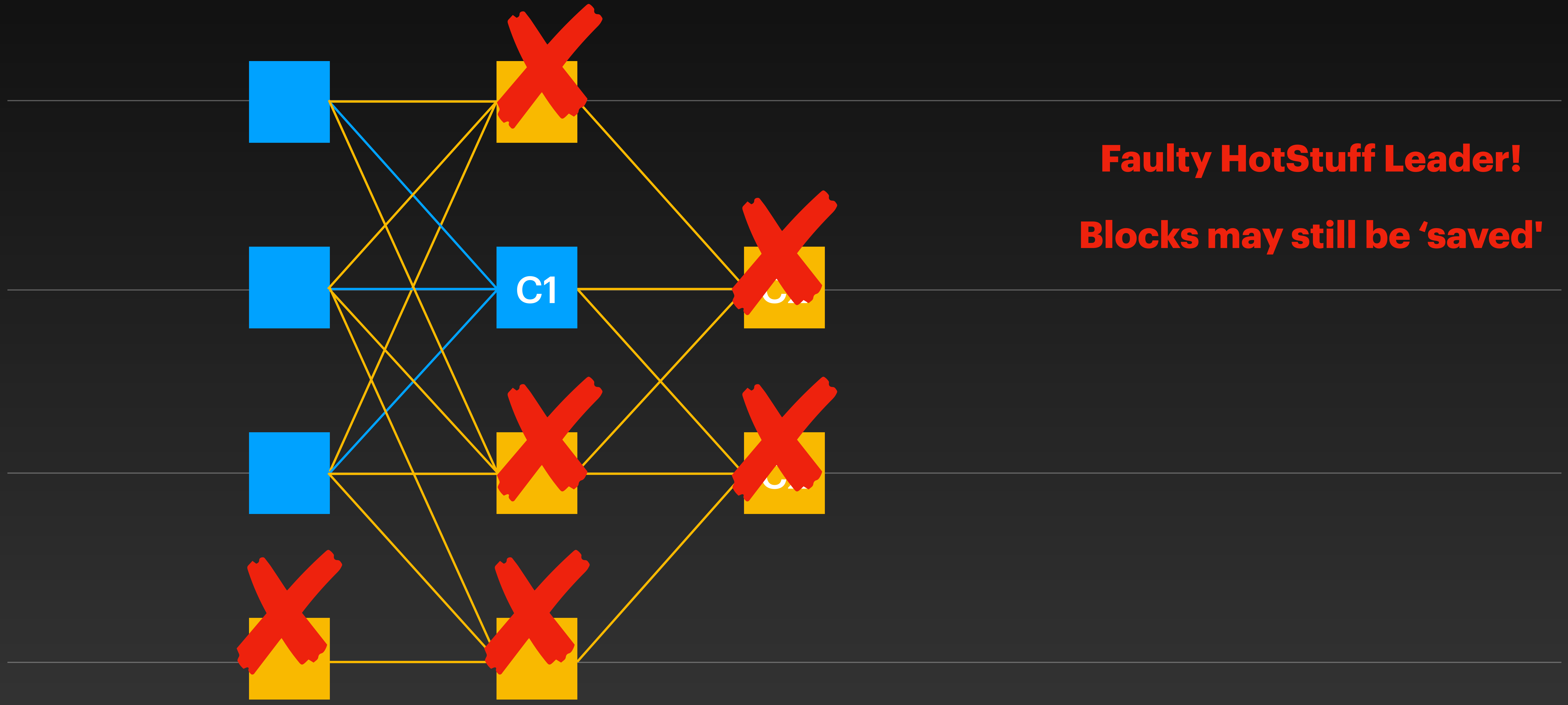
HotStuff on Narwhal

Enhanced commit rule



HotStuff on Narwhal

Enhanced commit rule

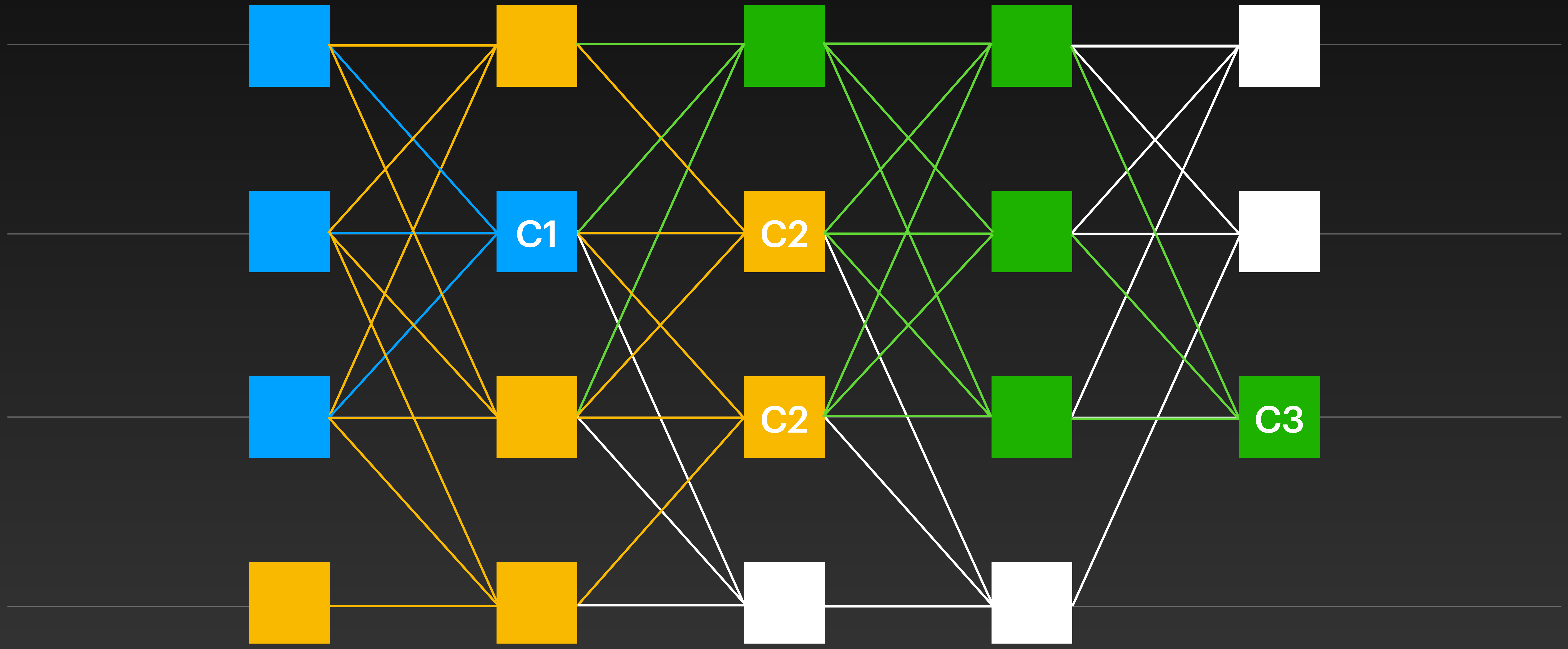


Faulty HotStuff Leader!

Blocks may still be ‘saved'

HotStuff on Narwhal

Enhanced commit rule

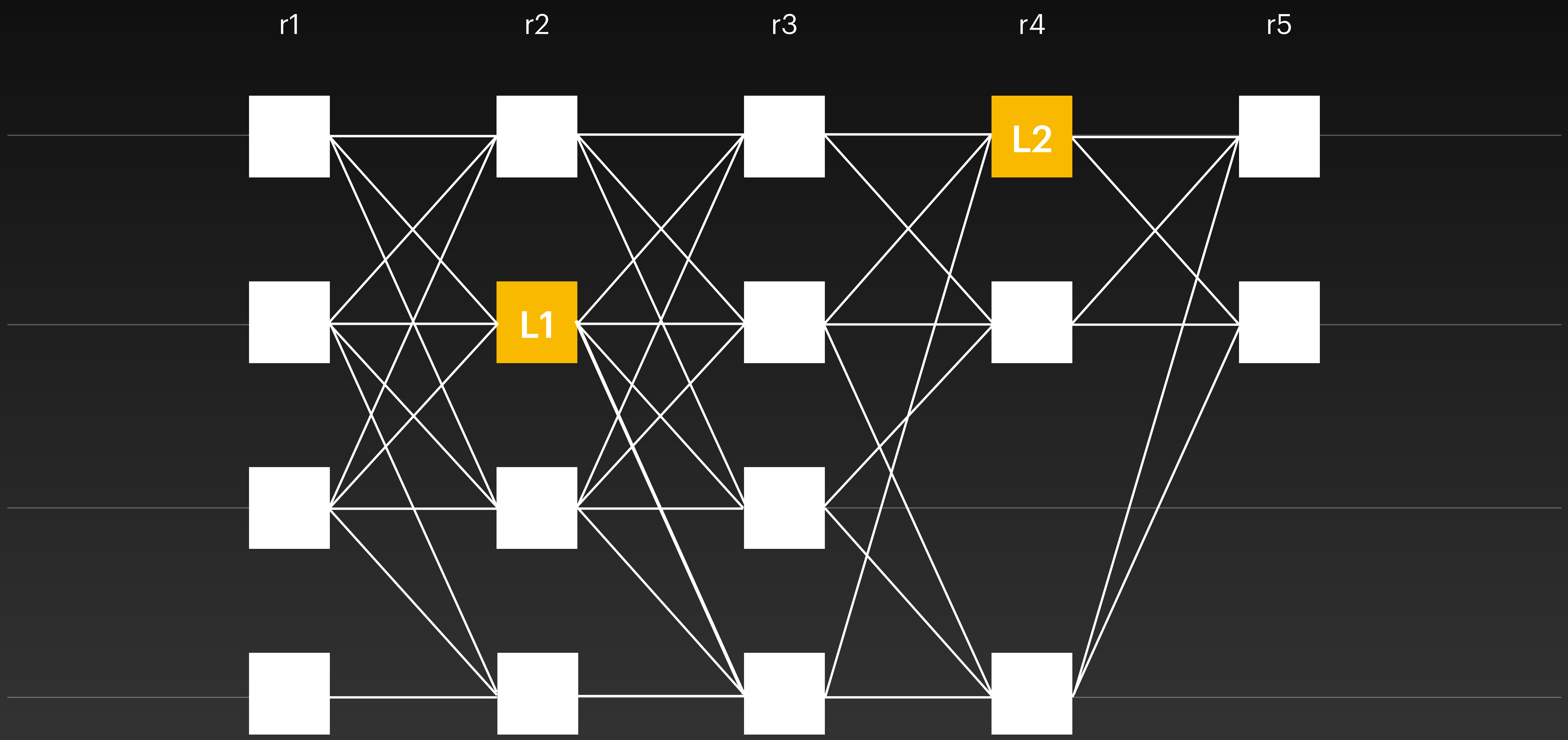


Modified Narwhal

Adapt Narwhal for partial-synchronous networks

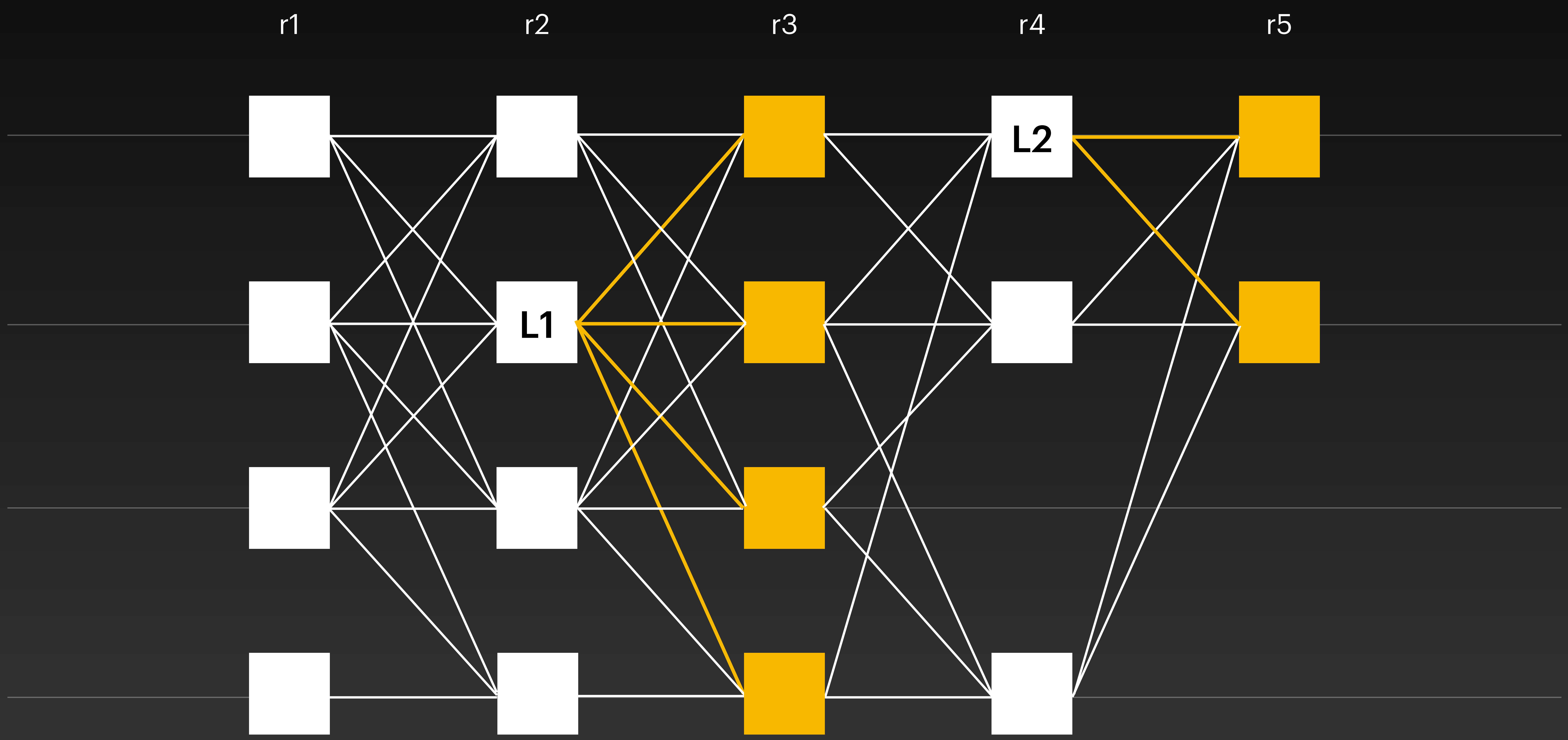
Modified Narwhal

Even rounds: wait for the leader (or to timeout)



Modified Narwhal

Odd rounds: wait for enough votes (or to timeout)



Bullshark

Zero-message partially-synchronous consensus

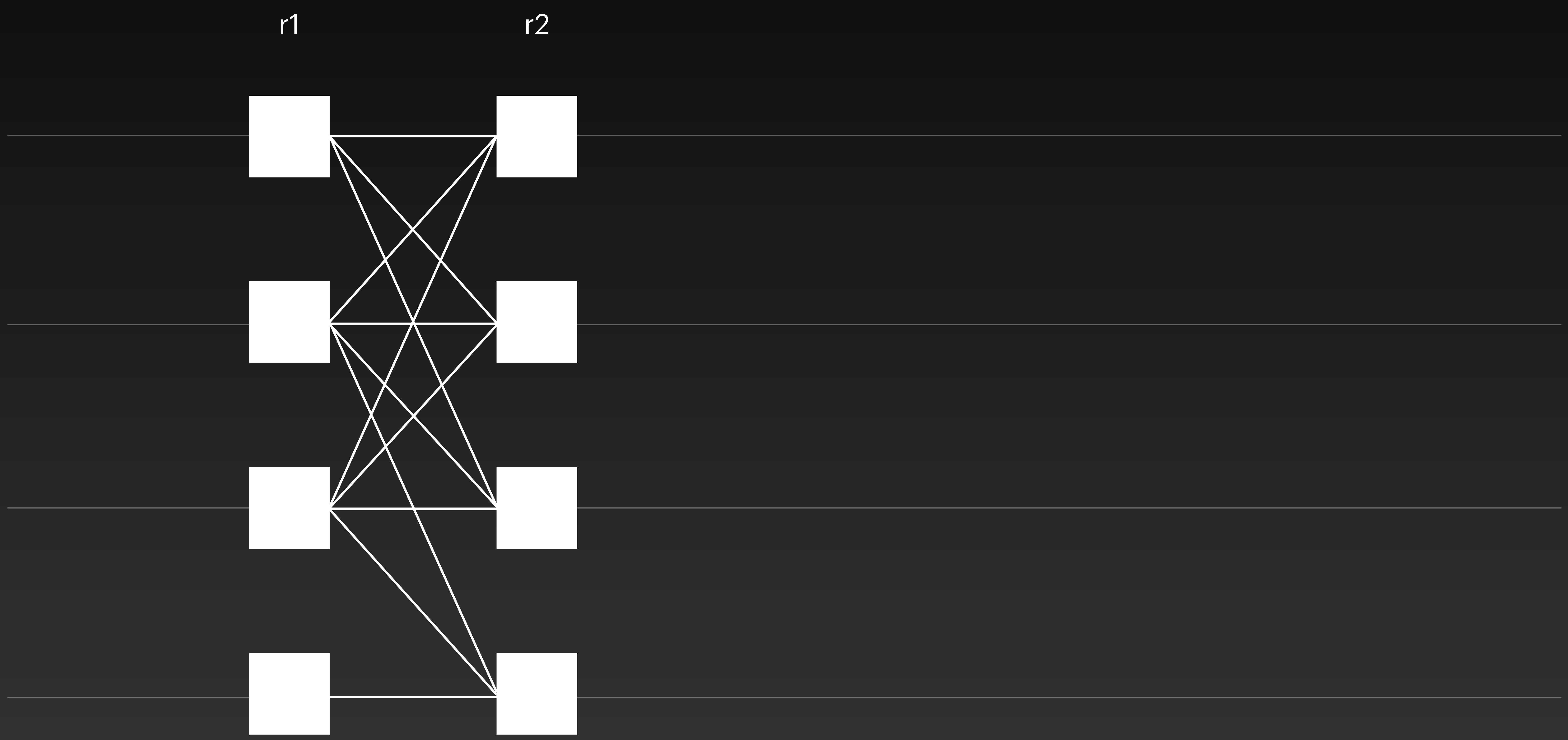
Bullshark

Zero-message partially-synchronous consensus

* without asynchronous fallback

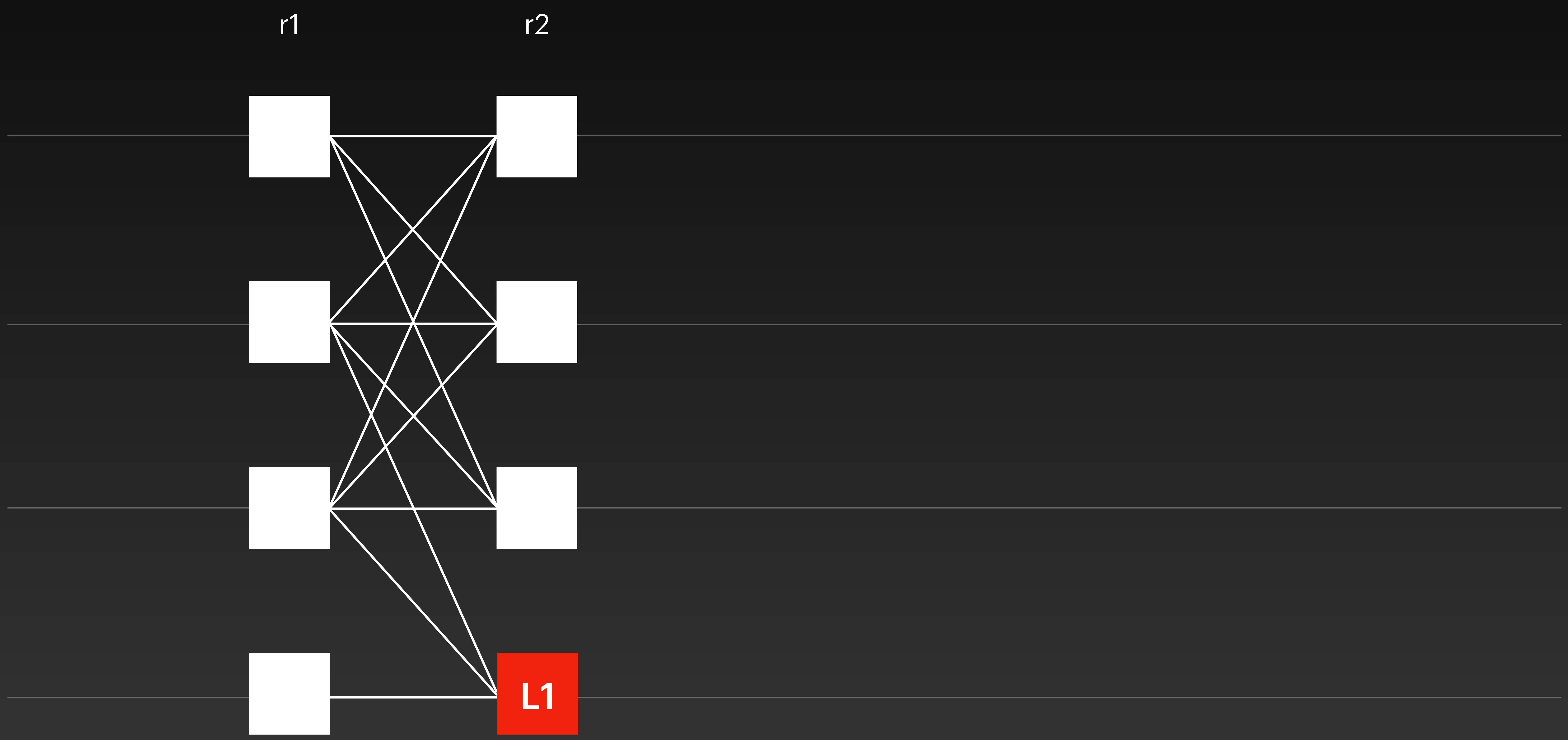
Bullshark

Just interpret the DAG



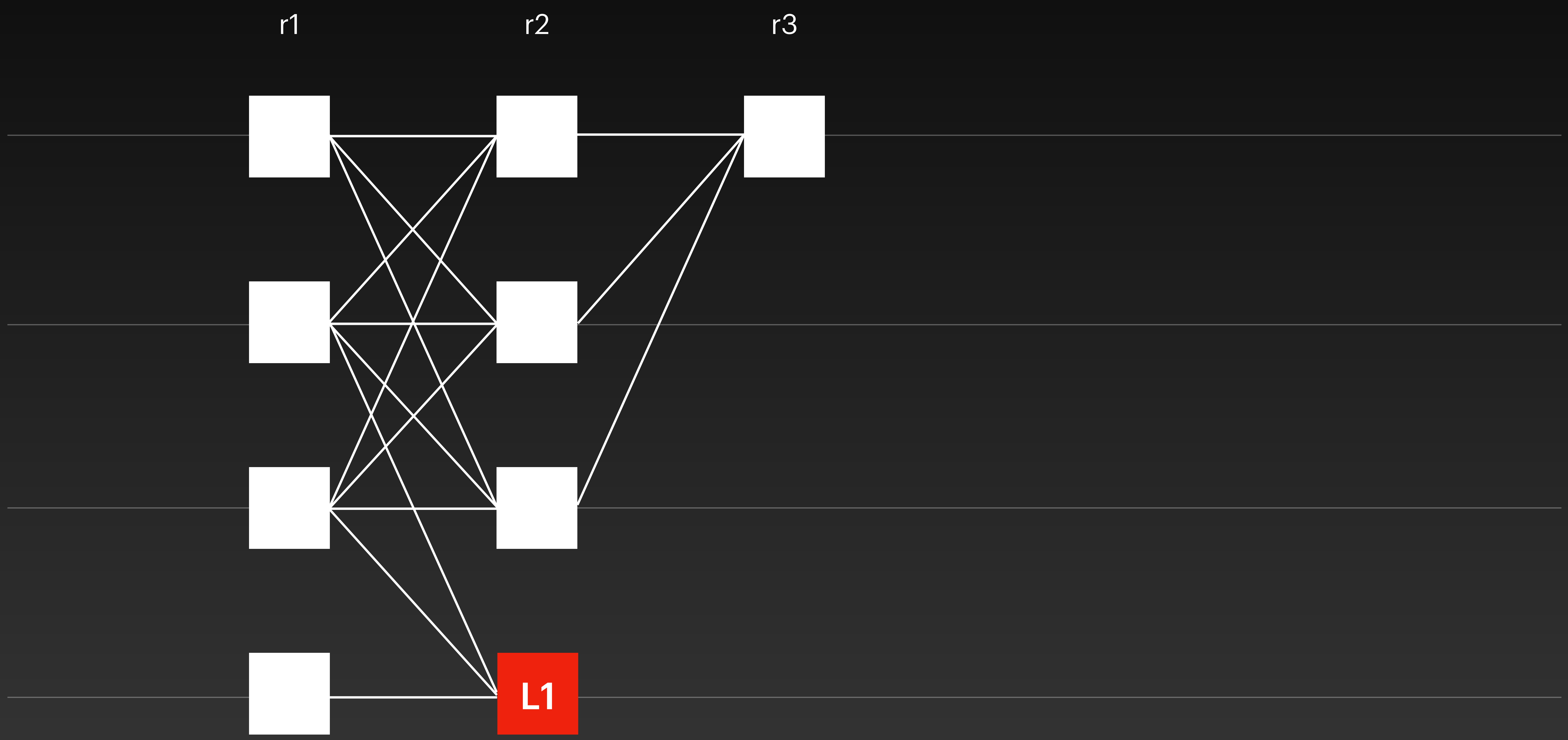
Bullshark

Deterministic leader every 2 rounds



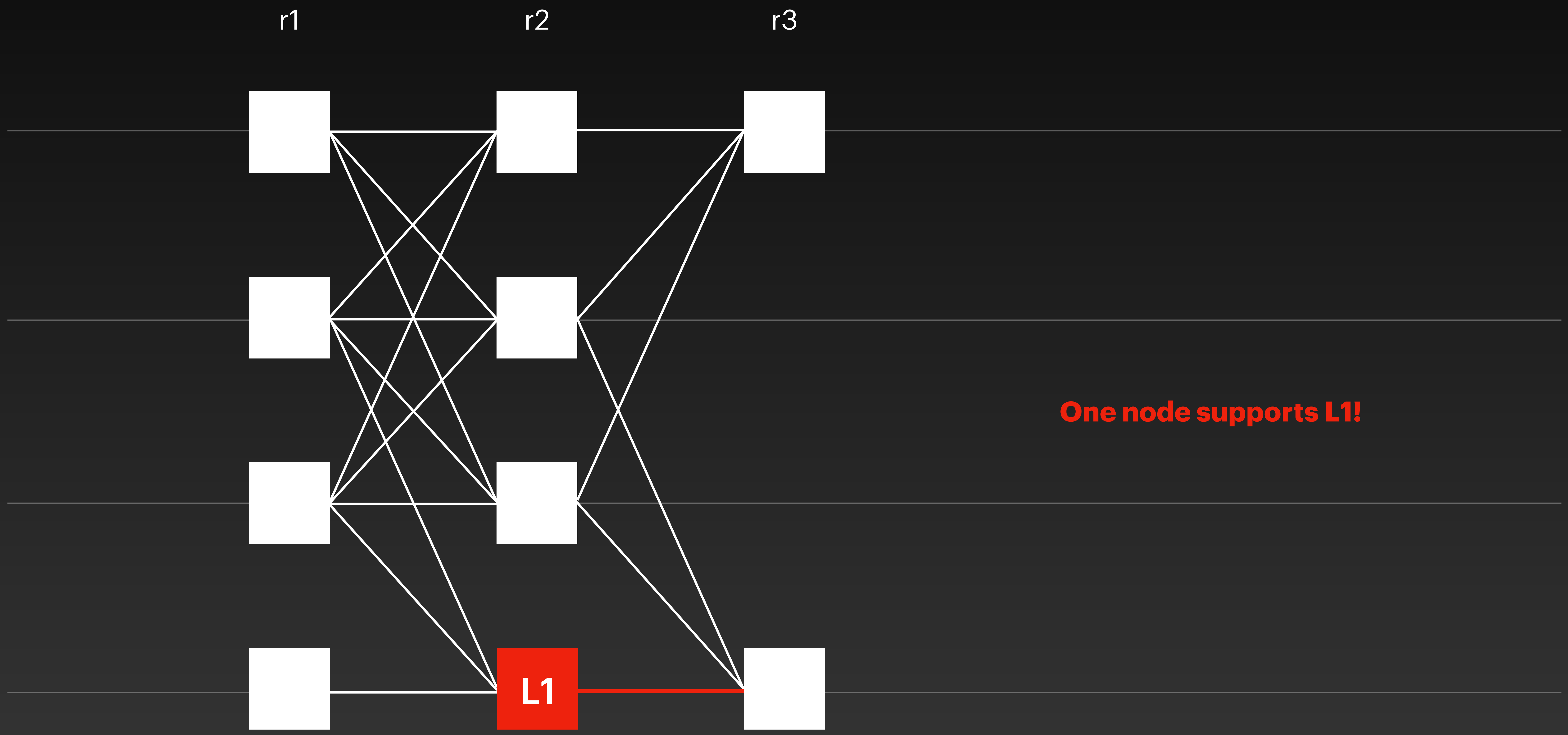
Bullshark

The leader needs f+1 links from round r



Bullshark

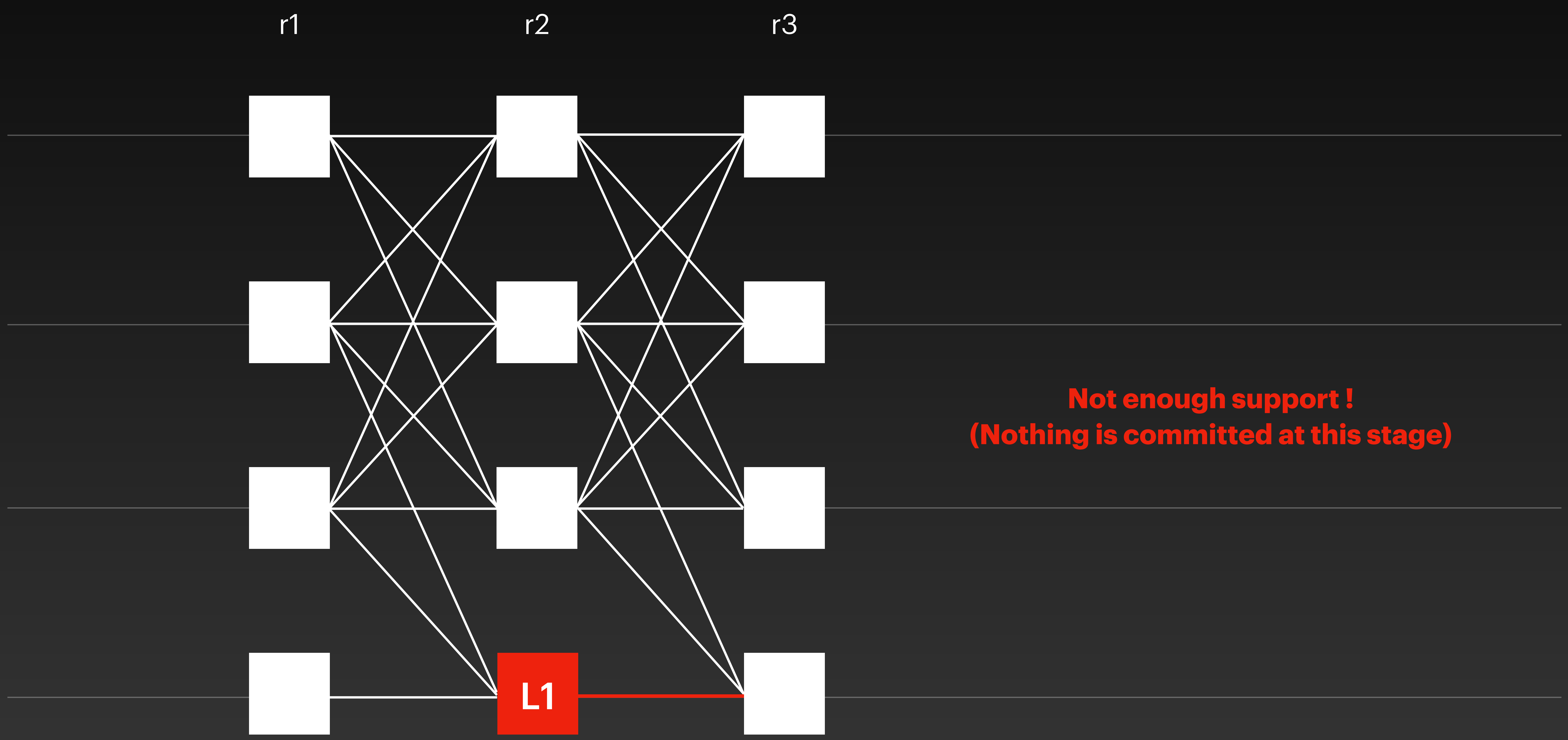
The leader needs f+1 links from round r



One node supports L1!

Bullshark

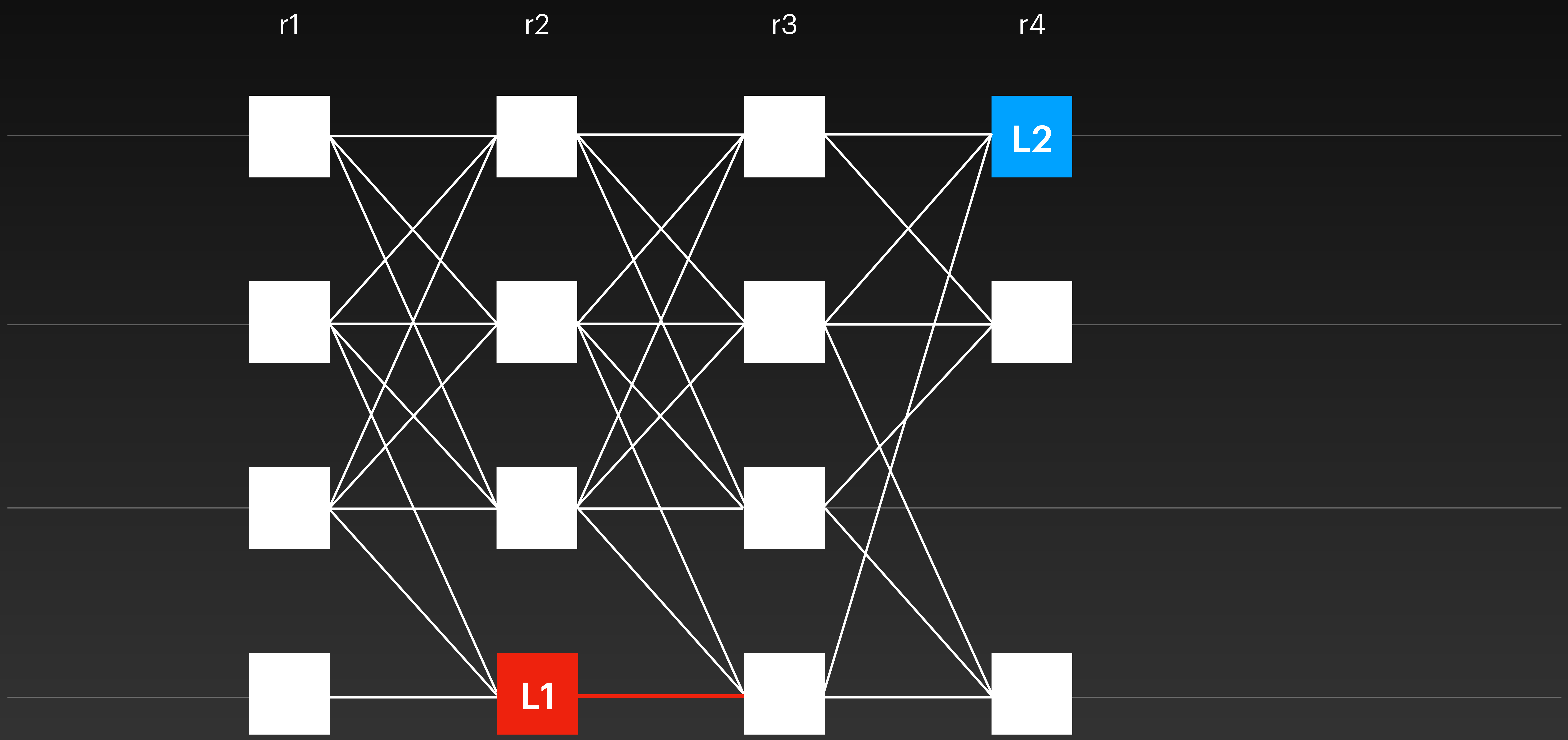
The leader needs f+1 links from round r



Not enough support !
(Nothing is committed at this stage)

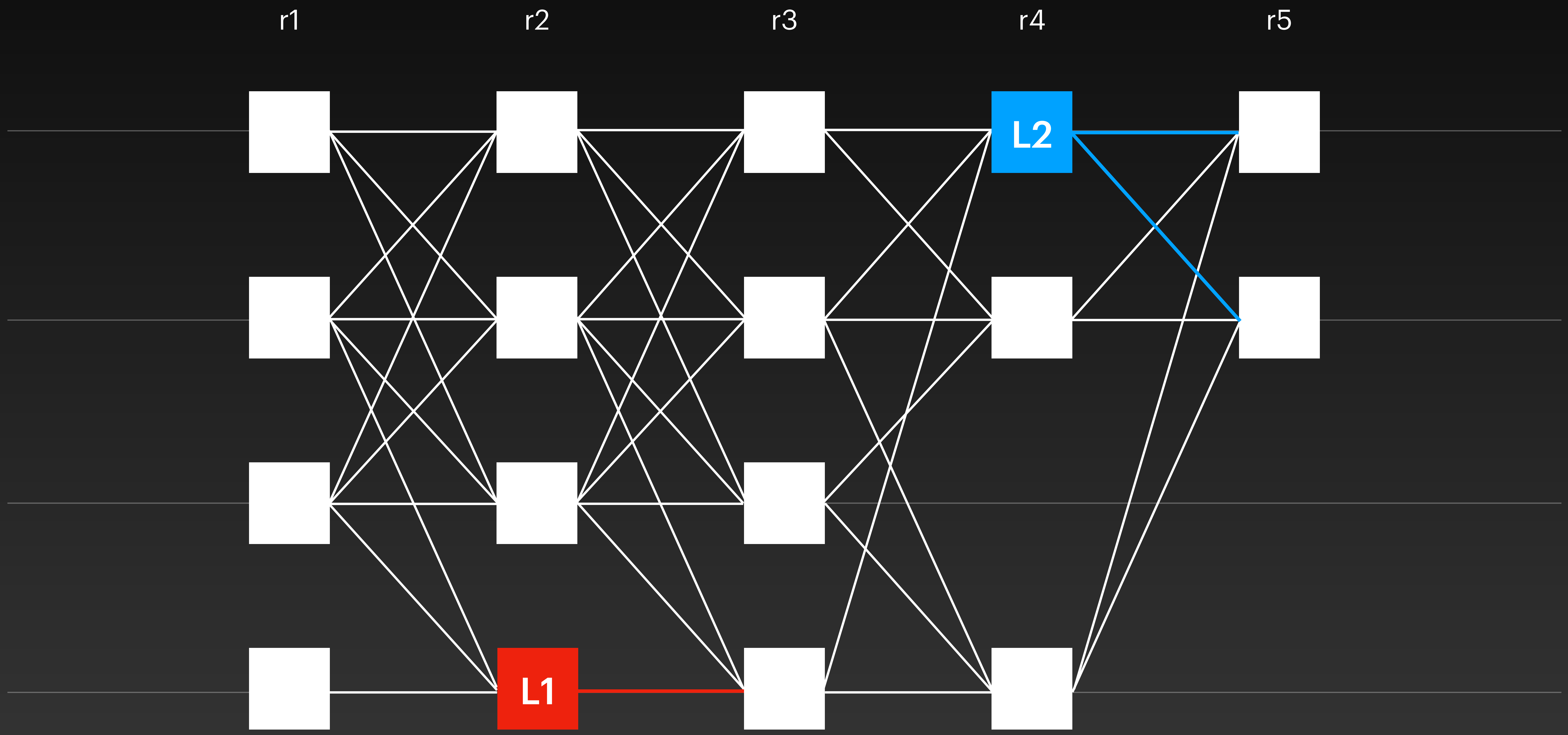
Bullshark

Elect the leader of r4



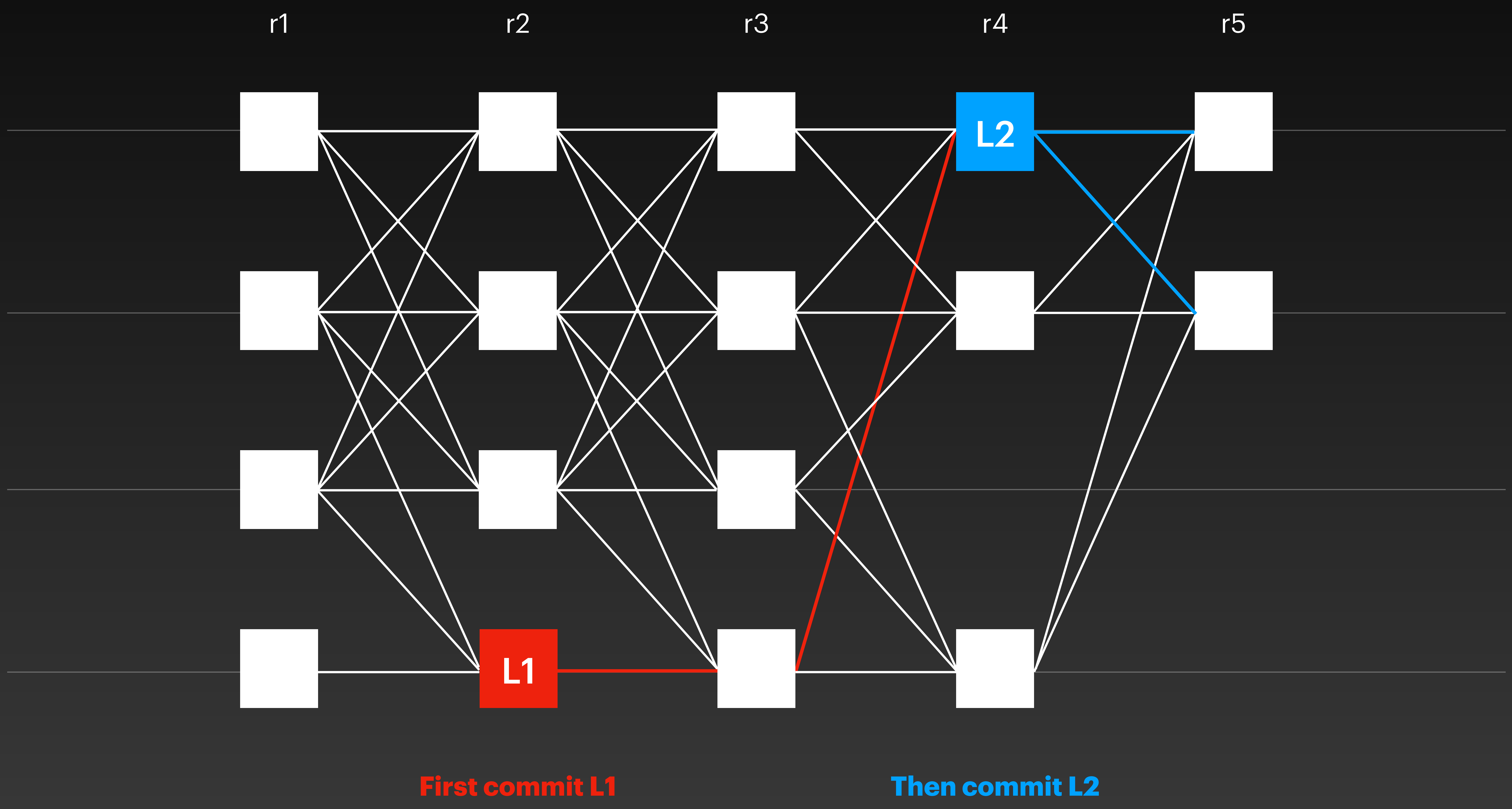
Bullshark

Leader L2 has enough support



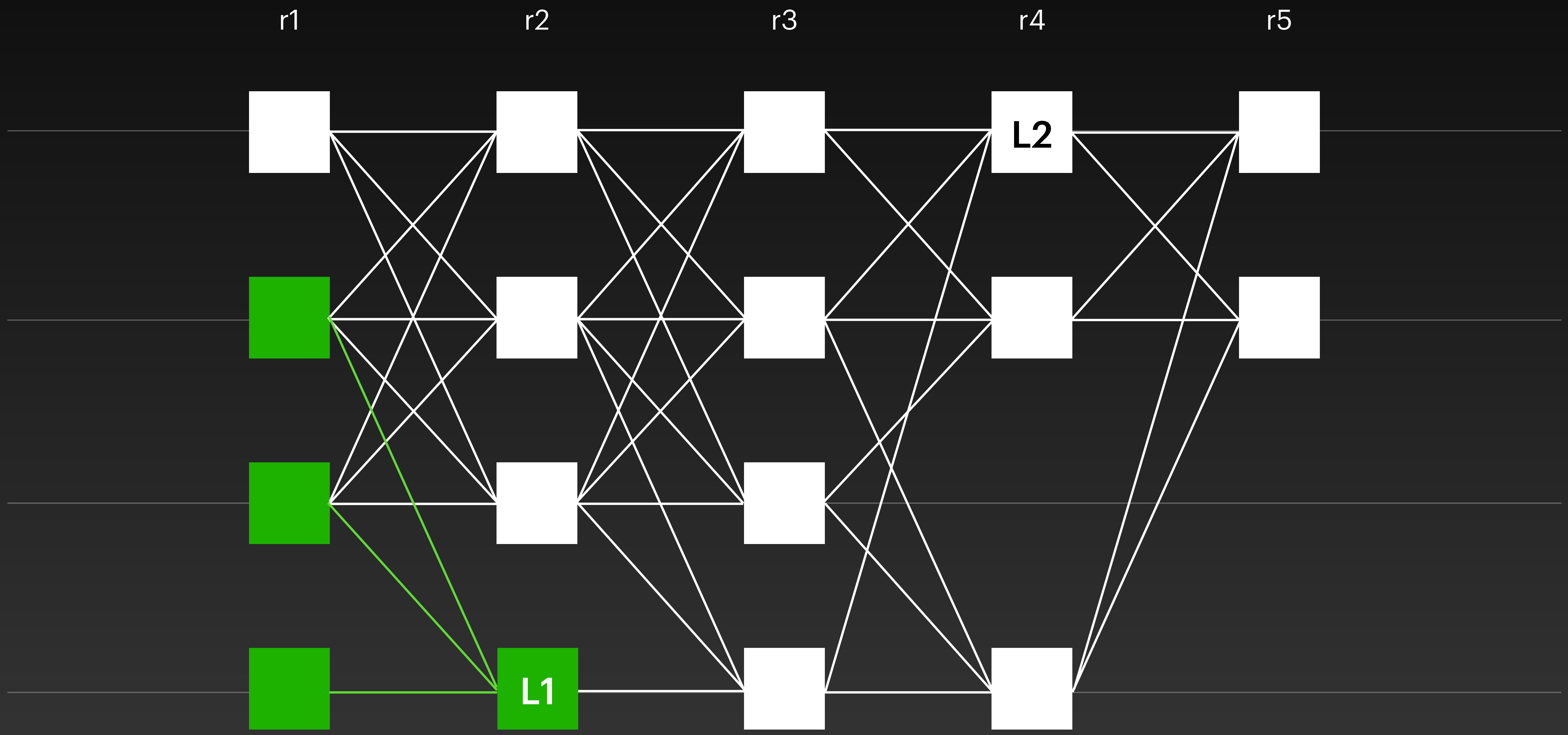
Bullshark

Leader L2 has links to leader L1



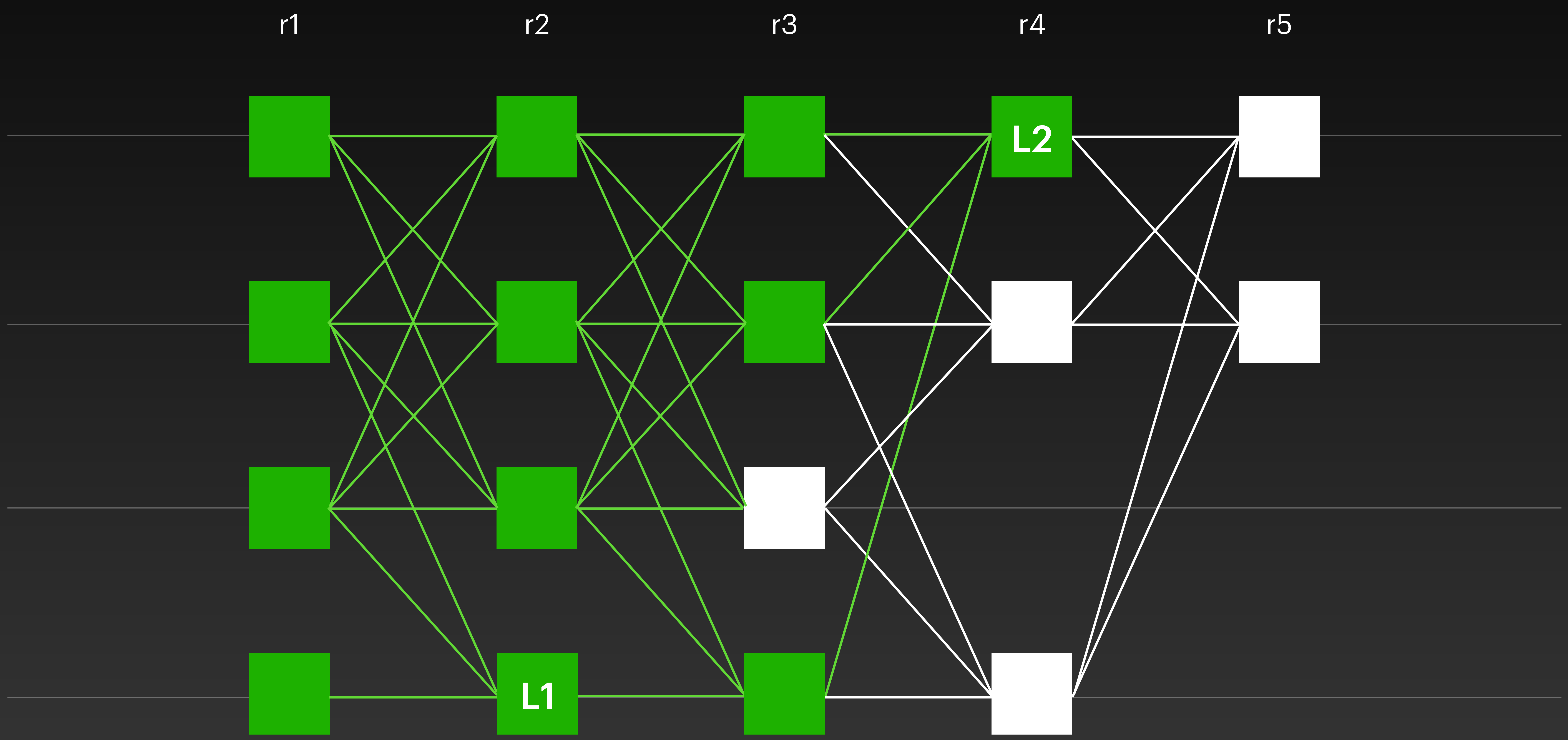
Bullshark

Commit all the sub-DAG of the leader



Bullshark

Commit all the sub-DAG of the leader



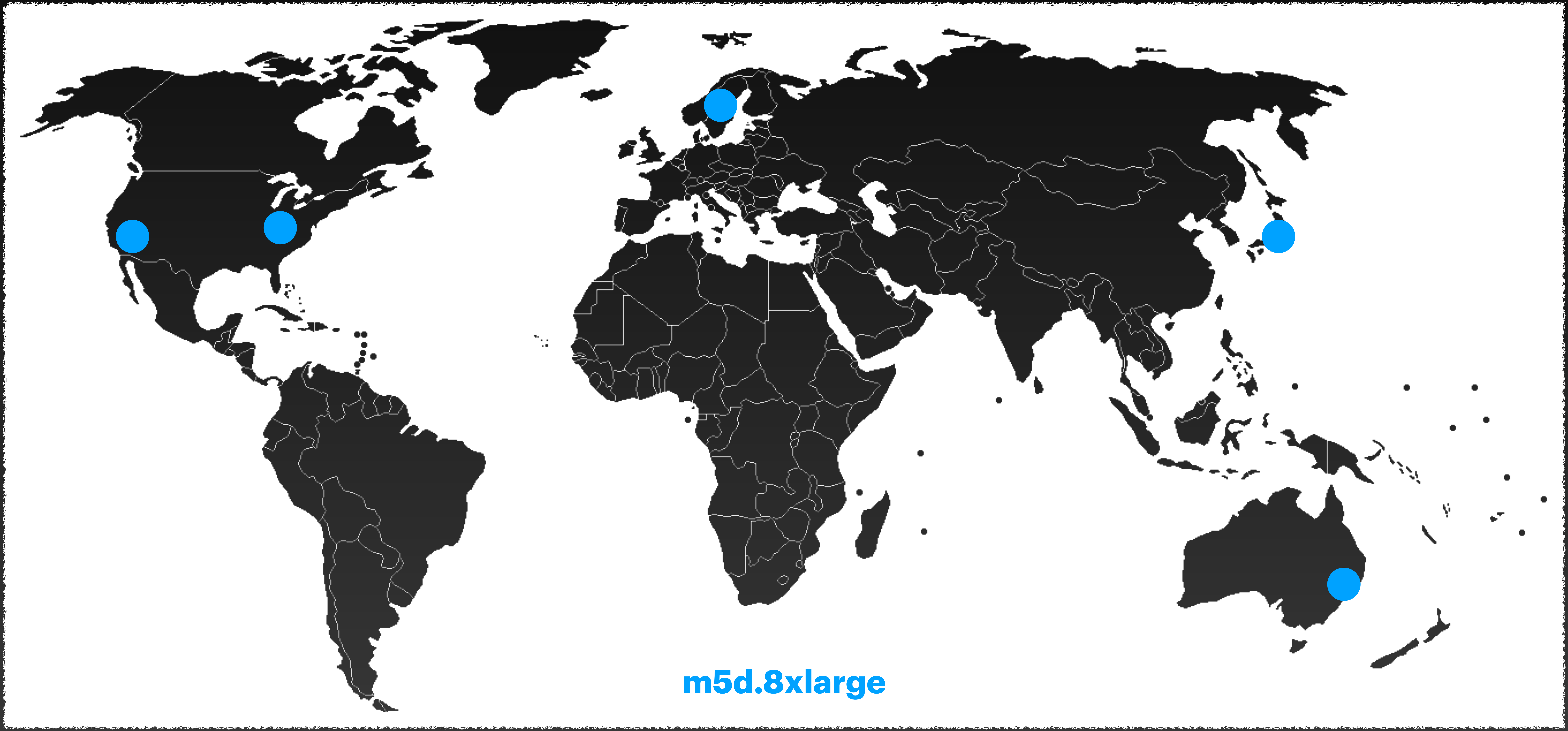
Implementation

- Written in Rust
- Networking: Tokio (TCP)
- Storage: RocksDB
- Cryptography: ed25519-dalek

https://github.com/asonnino/narwhal

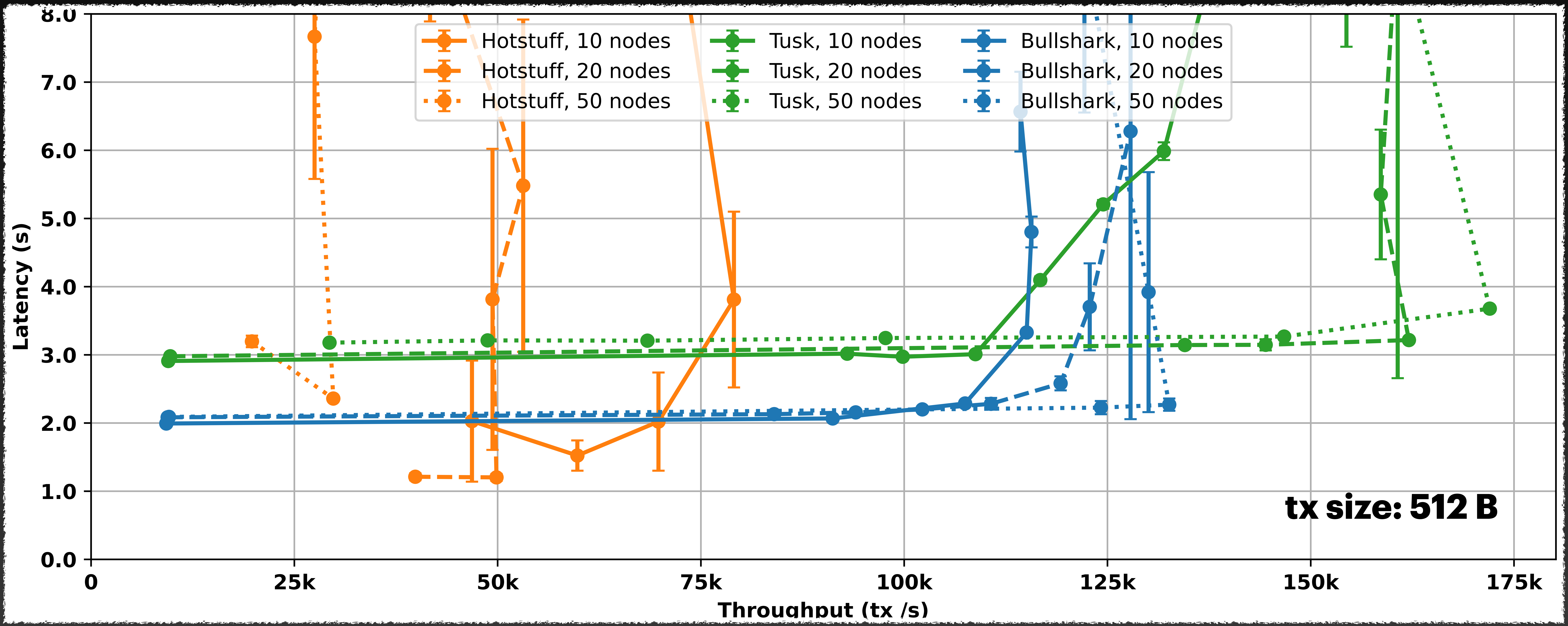
Evaluation

Experimental setup on AWS



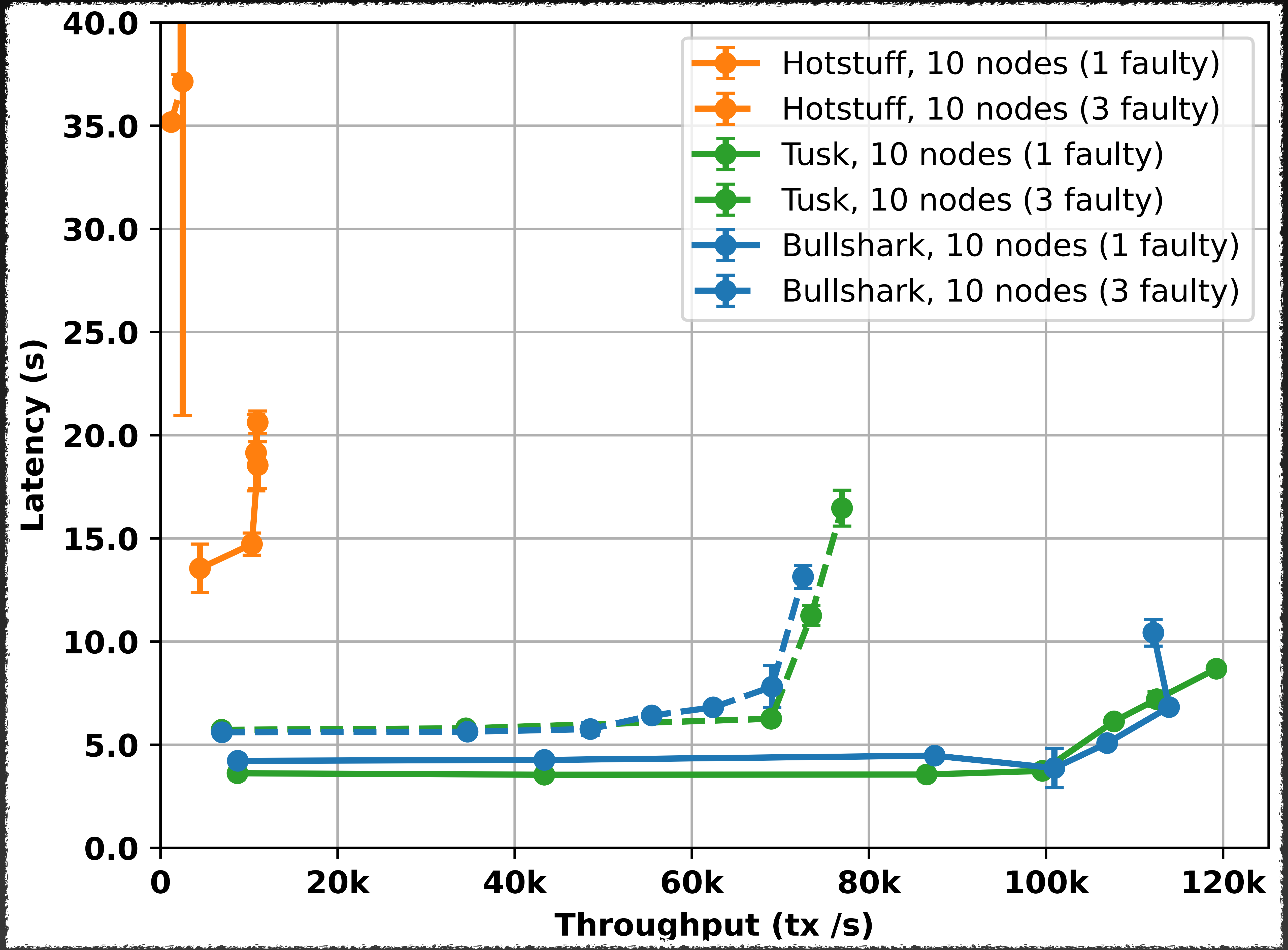
Evaluation

Throughput latency graph



Evaluation

Performance under faults



Summary

Bullshark

- Zero-message overhead, no view-change, no common-coin
- Disseminate data with Narwhal, exploits periods of synchrony

- **Paper:** https://sonnino.com/papers/bullshark.pdf
- **Code:** https://github.com/asonnino/narwhal

Engineering

Lessons Learned

Evaluation

Our biggest mistakes

- 😫 Add crash-recovery after-the-fact
- 😫 Add the synchroniser after-the-fact
- 😫 Add epoch changes after-the-fact
- 😫 Do not benchmark from day 1
- 😫 Start with fancy crypto
- 😫 Hide away serialisation
- 😫 Complex networked systems
- 😫 Isolate modules as in papers
- 😫 (Use grpc and magic network stack)

Evaluation

Our biggest mistakes

😫 Add crash-recovery after-the-fact

😫 Add the synchroniser after-the-fact

😫 Add epoch changes after-the-fact

- **What is the minimum state you need to persist across crash-recovery?**
- **The synchroniser will eventually be your bottleneck / source of instability**
- **Epoch changes are more complex than they look (sync new validators/ update configs from chain) — Advise: kill the node and reboot it.**

Evaluation

Our biggest mistakes

😫 Add crash-recovery after-the-fact

😫 Add the synchroniser after-the-fact

😫 Add epoch changes after-the-fact

😫 Do not benchmark from day 1

- **Many concurrency bugs found on WAN benchmarks under high load**
- **Spent months optimising blinding**

Evaluation

Our biggest mistakes

😫 Add crash-recovery after-the-fact

😫 Add the synchroniser after-the-fact

😫 Add epoch changes after-the-fact

😫 Do not benchmark from day 1

😫 Start with fancy crypto

😫 Hide away serialisation

- **Huge complexity; resulted redundant crypto operations**
- **Crypto serialisation was a bottleneck**

Evaluation

Our biggest mistakes

- 😫 Add crash-recovery after-the-fact
- 😫 Add the synchroniser after-the-fact
- 😫 Add epoch changes after-the-fact
- 😫 Do not benchmark from day 1
- 😫 Start with fancy crypto
- 😫 Hide away serialisation
- 😫 Complex networked systems
  - **Harder crash-recovery / should start with collocated workers**

Evaluation

Our biggest mistakes

- 😫 Add crash-recovery after-the-fact
- 😫 Add the synchroniser after-the-fact
- 😫 Add epoch changes after-the-fact
- 😫 Do not benchmark from day 1
- 😫 Start with fancy crypto
- 😫 Hide away serialisation
- 😫 Complex networked systems
- **Debugging / perf improvement nightmare**

- 😫 Isolate modules as in papers
- 😫 (Use grpc and magic network stack)

alberto@mystenlabs.com

Alberto Sonnino

EXTRA

Benchmark of BFT Systems

Evaluation

Typical mistakes

- 😫 Forgo persistent storage
- 😫 Do not sanitise messages
- 😫 Local/LAN benchmark + ping
- 😫 Many nodes on same machine
- 😫 Change parameters across runs
- 😫 Set transaction size to zero
- 😫 Preconfigure nodes with txs
- 😫 Send a single burst of transactions
- 😫 Benchmark for a few seconds
- 😫 Start timer in the batch maker
- 😫 Evaluate latency w/ only the first tx
- 😫 Separate latency and throughput
- 😫 Only benchmark happy path

Evaluation

Set the benchmark parameters

```
Faults: 0 node(s)

Committee size: 10 node(s)

Transaction size: 512 B
```

Evaluation

Set the benchmark parameters

Faults: 0 node(s)

Committee size: 10 node(s)

Transaction size: 512 B

Header size: 1,000 B

Max header delay: 200 ms

GC depth: 50 round(s)

Sync retry delay: 5,000 ms

Sync retry nodes: 3 node(s)

batch size: 500,000 B

Max batch delay: 200 ms

Evaluation

Typical mistakes

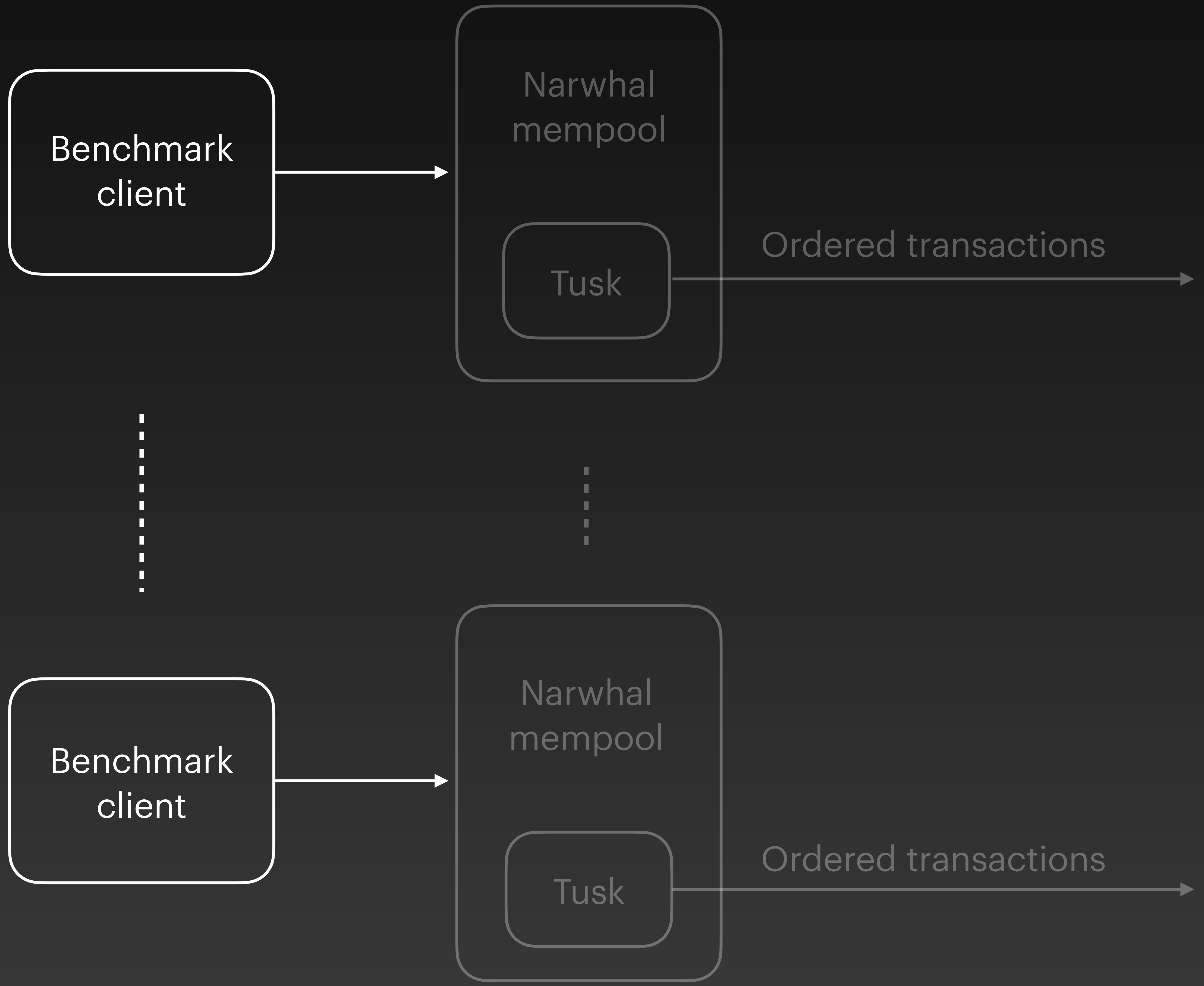
- 😫 Forgo persistent storage
- 😫 Do not sanitise messages
- 😫 Local/LAN benchmark + ping
- 😫 Many nodes on same machine
- 😫 Change parameters across runs
- 😫 Set transaction size to zero
- 😫 Preconfigure nodes with txs
- 😫 Send a single burst of transactions
- 😫 Benchmark for a few seconds
- 😫 Start timer in the batch maker
- 😫 Evaluate latency w/ only the first tx
- 😫 Separate latency and throughput
- 😫 Only benchmark happy path

Evaluation

Benchmark clients

Fixed input rate

For a long time (minutes)

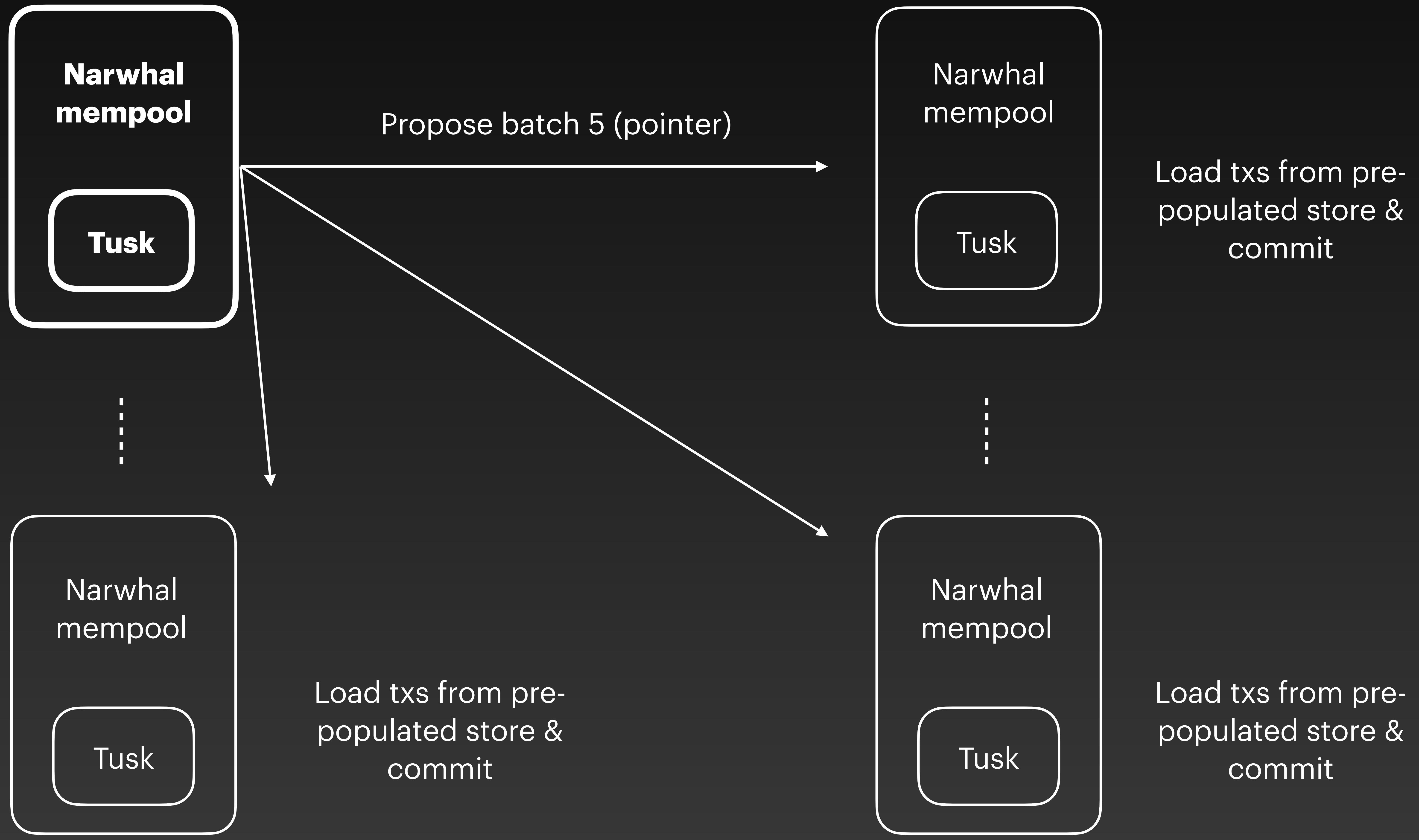


Evaluation

Typical mistakes

- 😫 Forgo persistent storage
- 😫 Do not sanitise messages
- 😫 Local/LAN benchmark + ping
- 😫 Many nodes on same machine
- 😫 Change parameters across runs
- 😫 Set transaction size to zero
- 😫 Preconfigure nodes with txs
- 😫 Send a single burst of transactions
- 😫 Benchmark for a few seconds
- 😫 Start timer in the batch maker
- 😫 Evaluate latency w/ only the first tx
- 😫 Separate latency and throughput
- 😫 Only benchmark happy path

Evaluation
Typical mistake



Evaluation

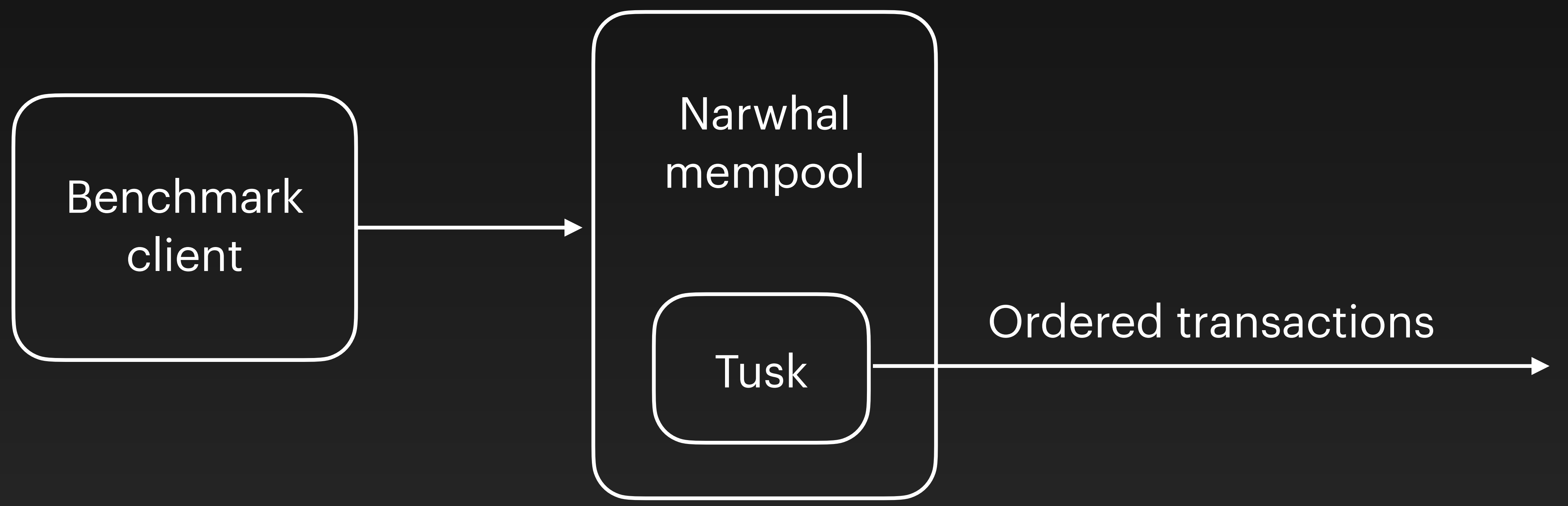
Typical mistakes

- 😫 Forgo persistent storage
- 😫 Do not sanitise messages
- 😫 Local/LAN benchmark + ping
- 😫 Many nodes on same machine
- 😫 Change parameters across runs
- 😫 Set transaction size to zero
- 😫 Preconfigure nodes with txs
- 😫 Send a single burst of transactions
- 😫 Benchmark for a few seconds
- 😫 Start timer in the batch maker
- 😫 Evaluate latency w/ only the first tx
- 😫 Separate latency and throughput
- 😫 Only benchmark happy path

Evaluation

Typical mistake

send 50k txs (once)

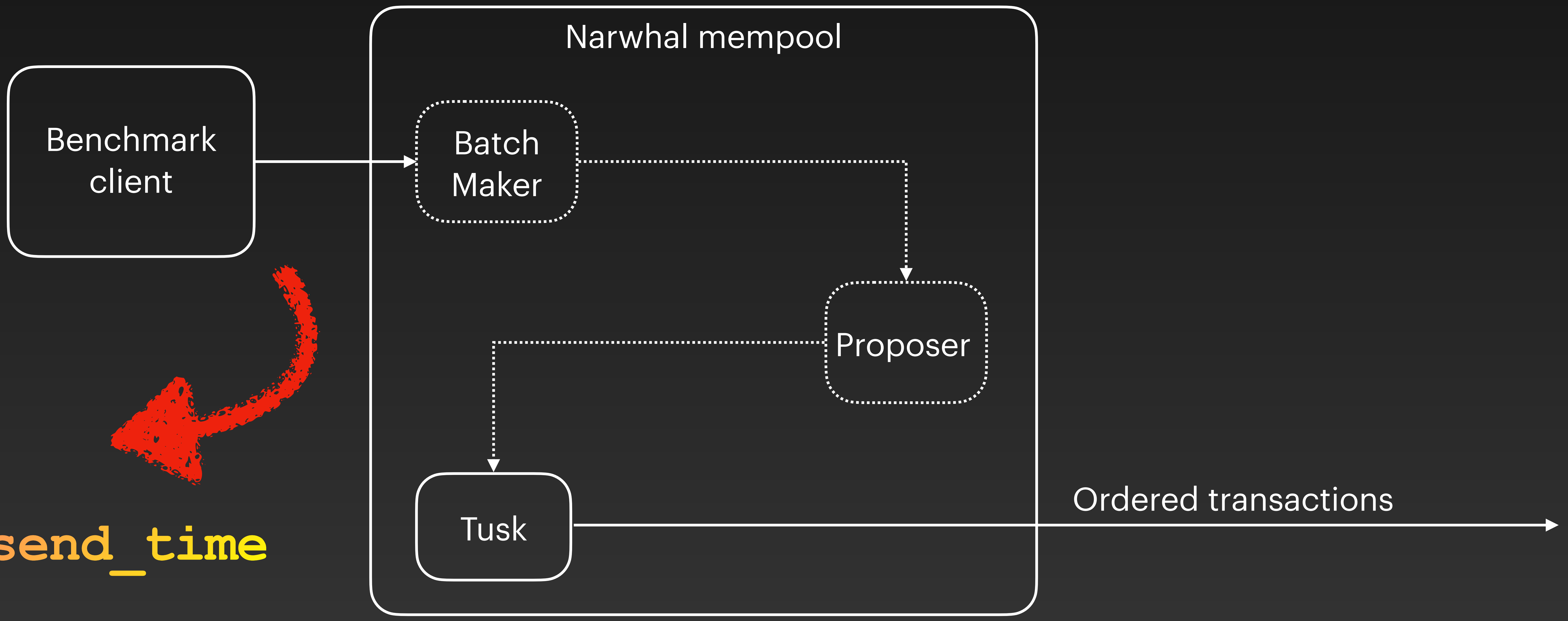


output after 400 ms

😫 TPS = 50k / 400ms = 125k tx/s 😫

Evaluation

Instrument the codebase



`bench_start_time`

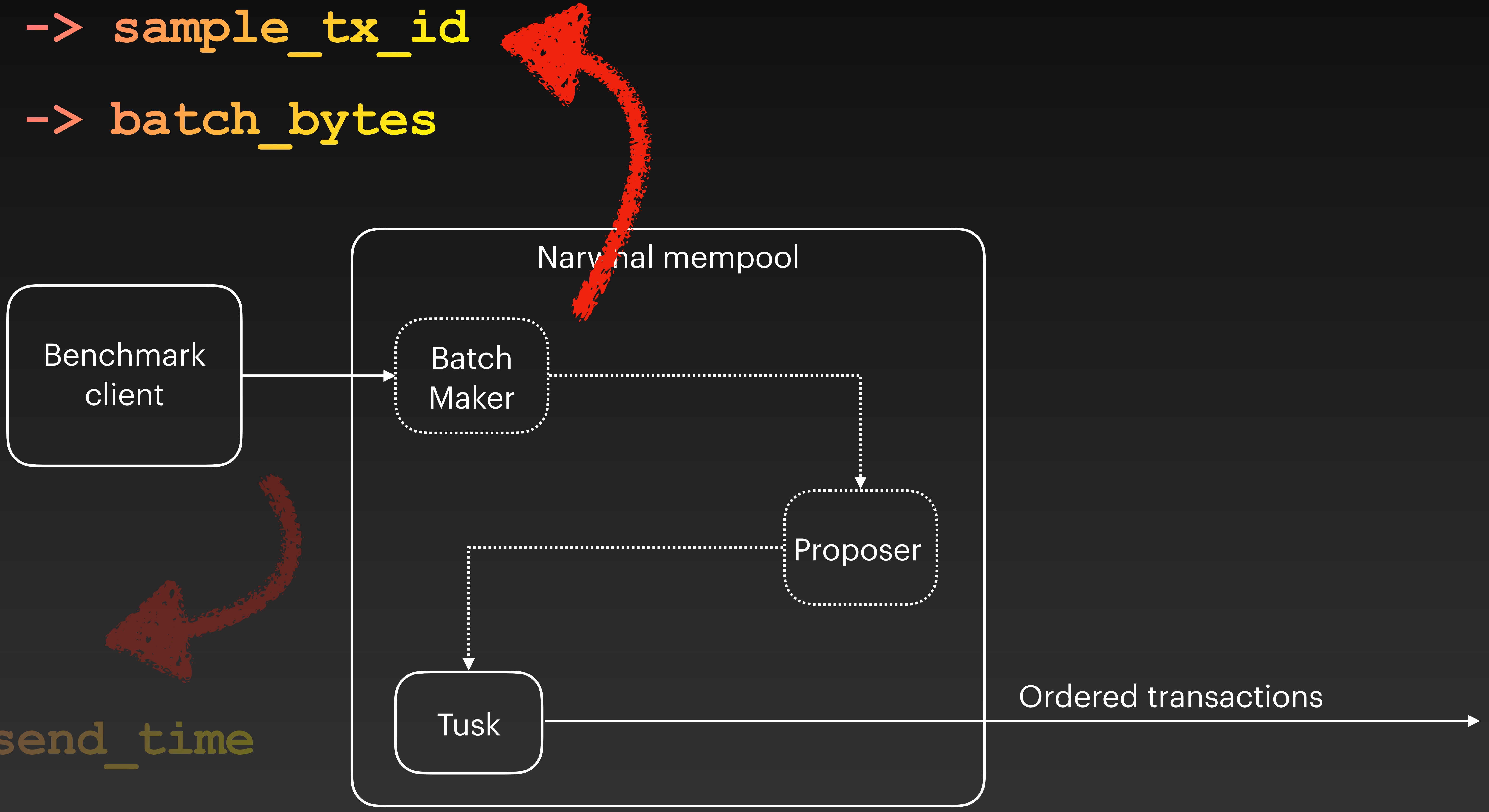
`sample_tx_id -> send_time`

Evaluation

Instrument the codebase

`batch_digest -> sample_tx_id`

`batch_digest -> batch_bytes`

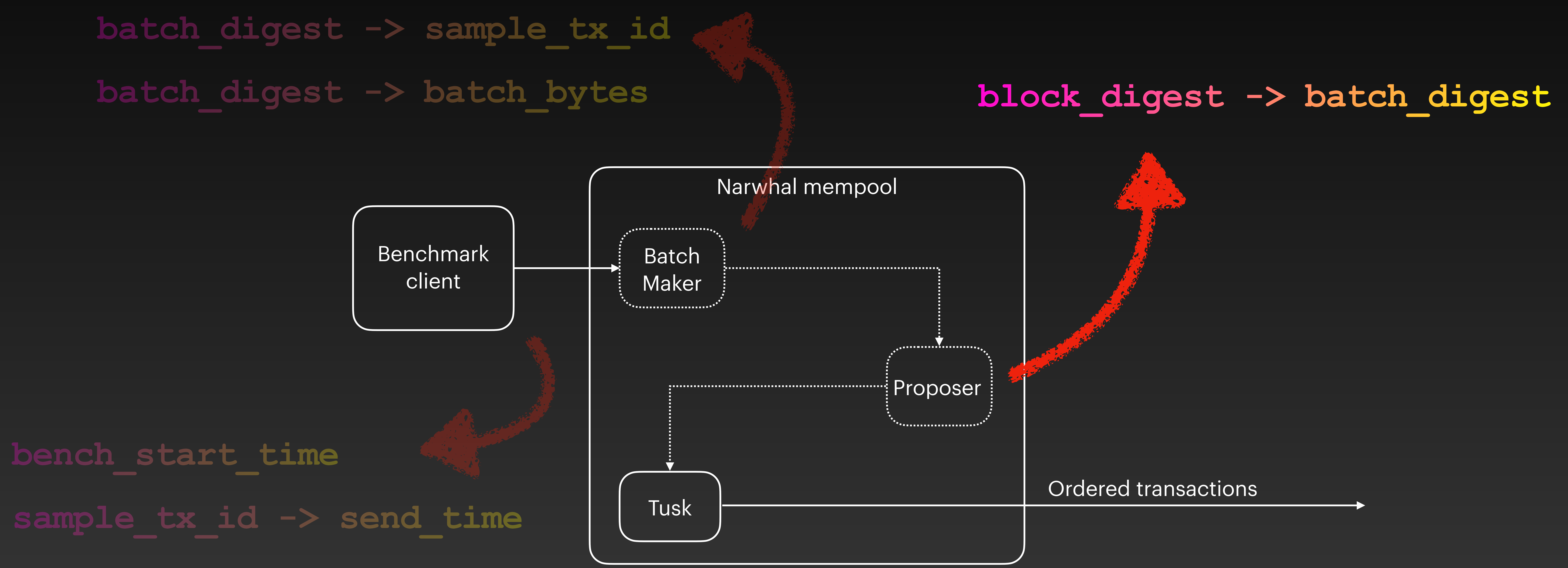


`bench_start_time`

`sample_tx_id -> send_time`

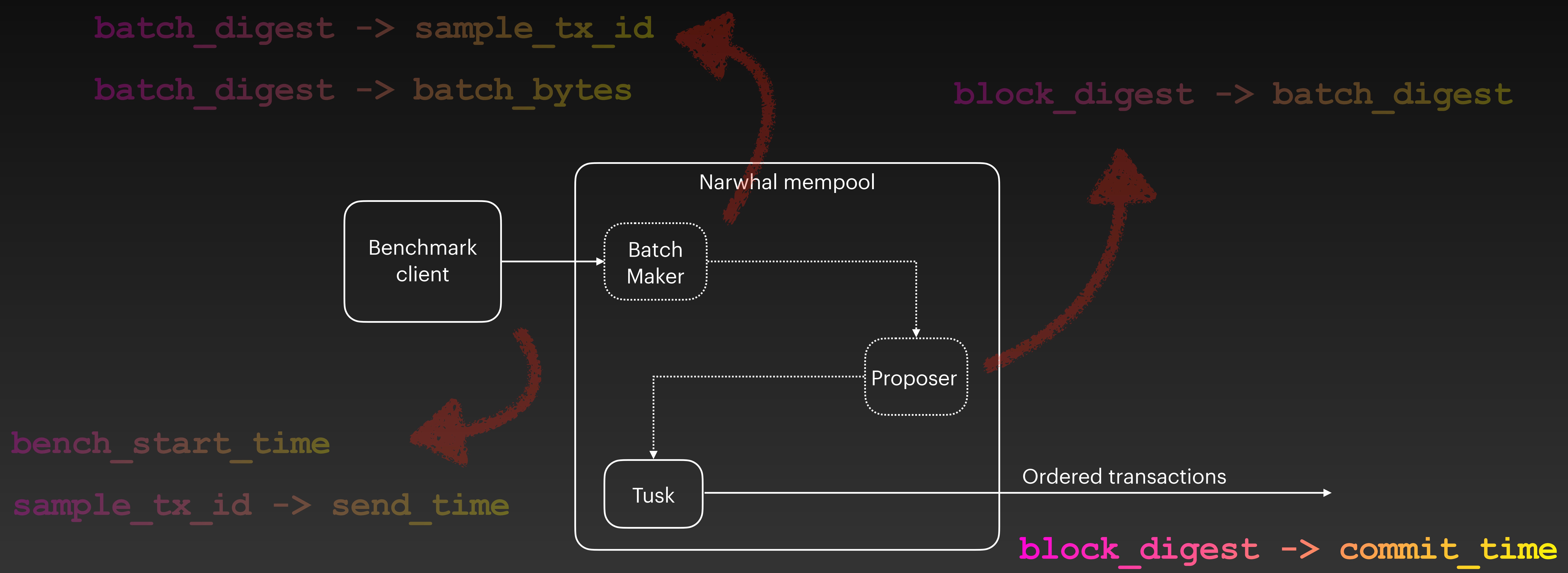
Evaluation

Instrument the codebase



Evaluation

Instrument the codebase

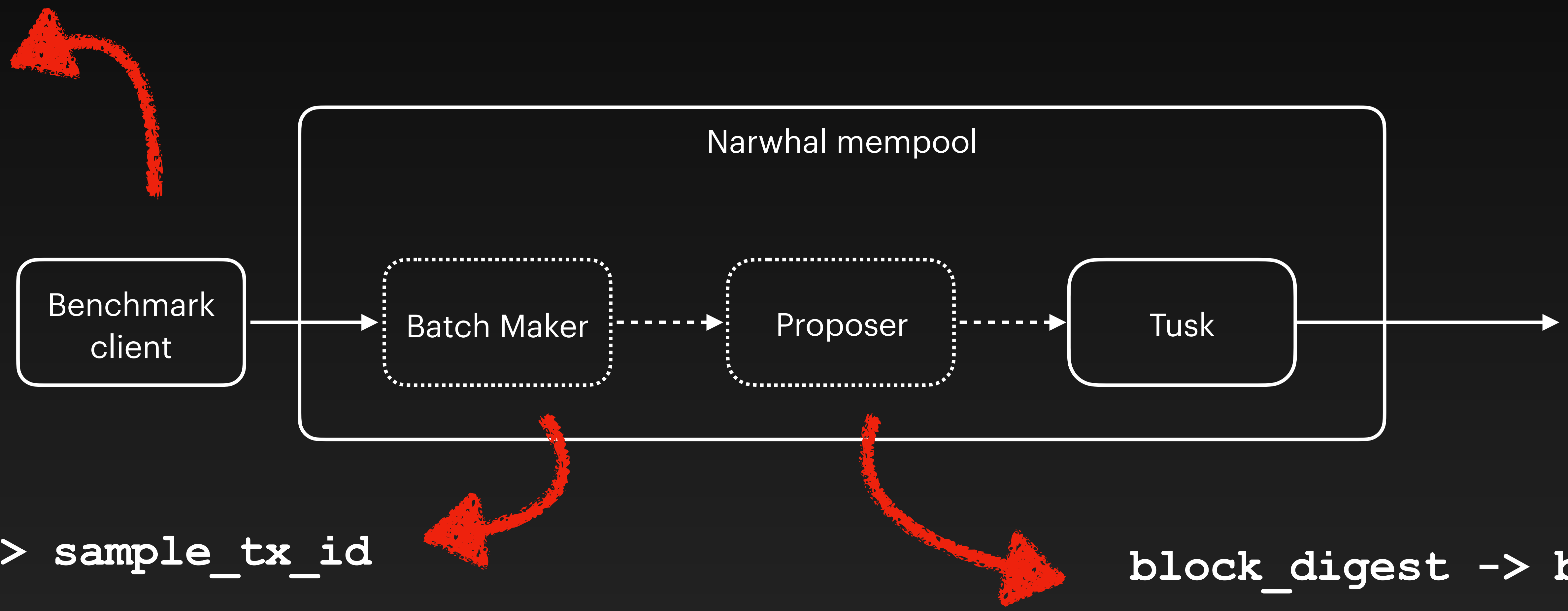


Evaluation

Compute throughput

bench_start_time

sample_tx_id -> send_time



block_digest -> commit_time

batch_digest -> sample_tx_id

batch_digest -> batch_bytes

block_digest -> batch_digest

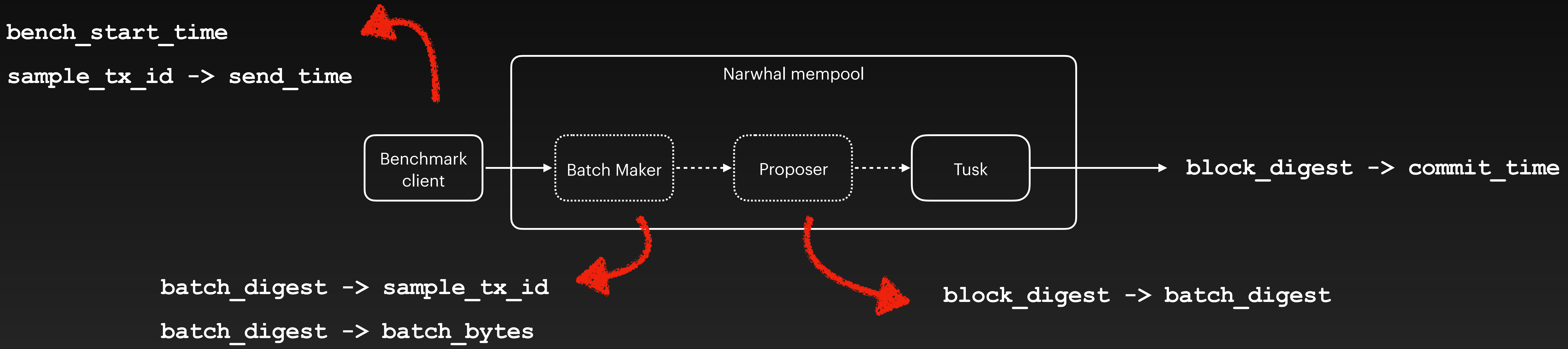
total_time = last_commit_time - bench_start_time

BPS = total_bytes / total_time

TPS = BPS / transaction_size

Evaluation

Compute latency




```
samples = commit_time - send_time

latency = average(samples)
```

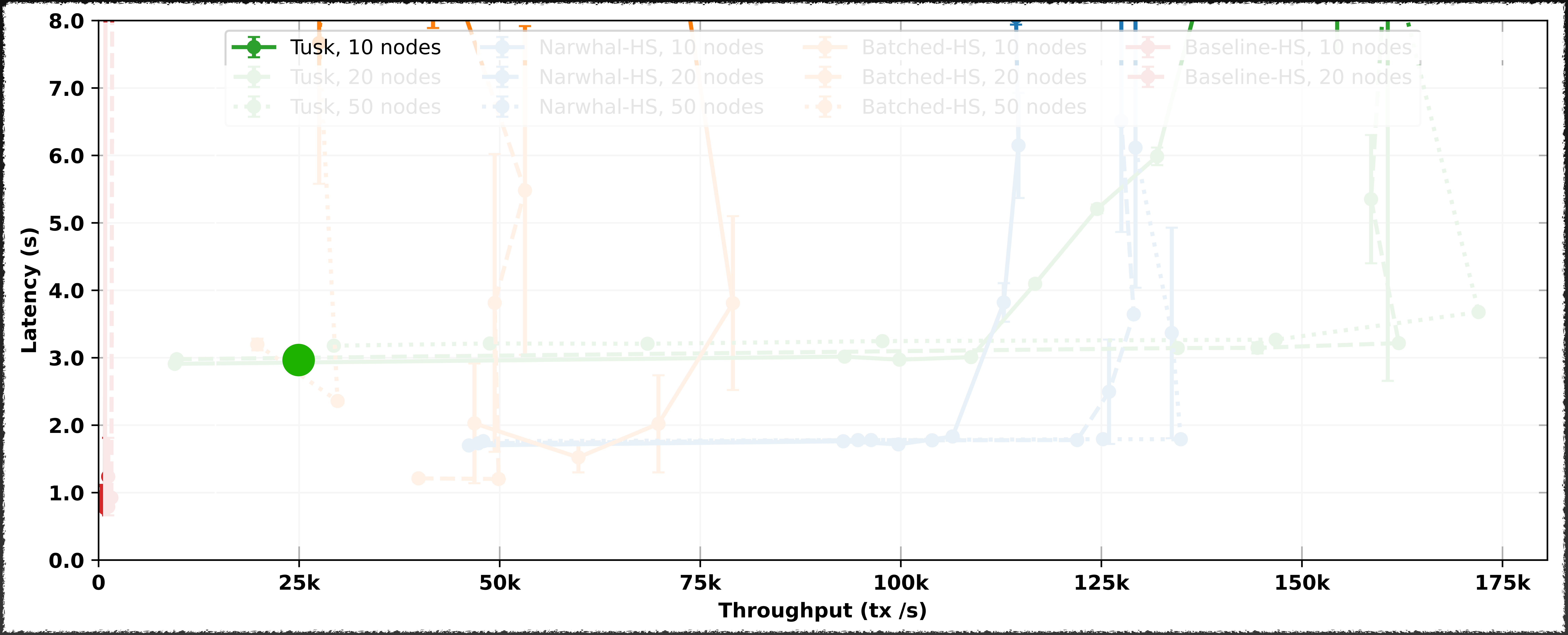
Evaluation

Typical mistakes

- 😫 Forgo persistent storage
- 😫 Do not sanitise messages
- 😫 Local/LAN benchmark + ping
- 😫 Many nodes on same machine
- 😫 Change parameters across runs
- 😫 Set transaction size to zero
- 😫 Preconfigure nodes with txs
- 😫 Send a single burst of transactions
- 😫 Benchmark for a few seconds
- 😫 Start timer in the batch maker
- 😫 Evaluate latency w/ only the first tx
- 😫 Separate latency and throughput
- 😫 Only benchmark happy path

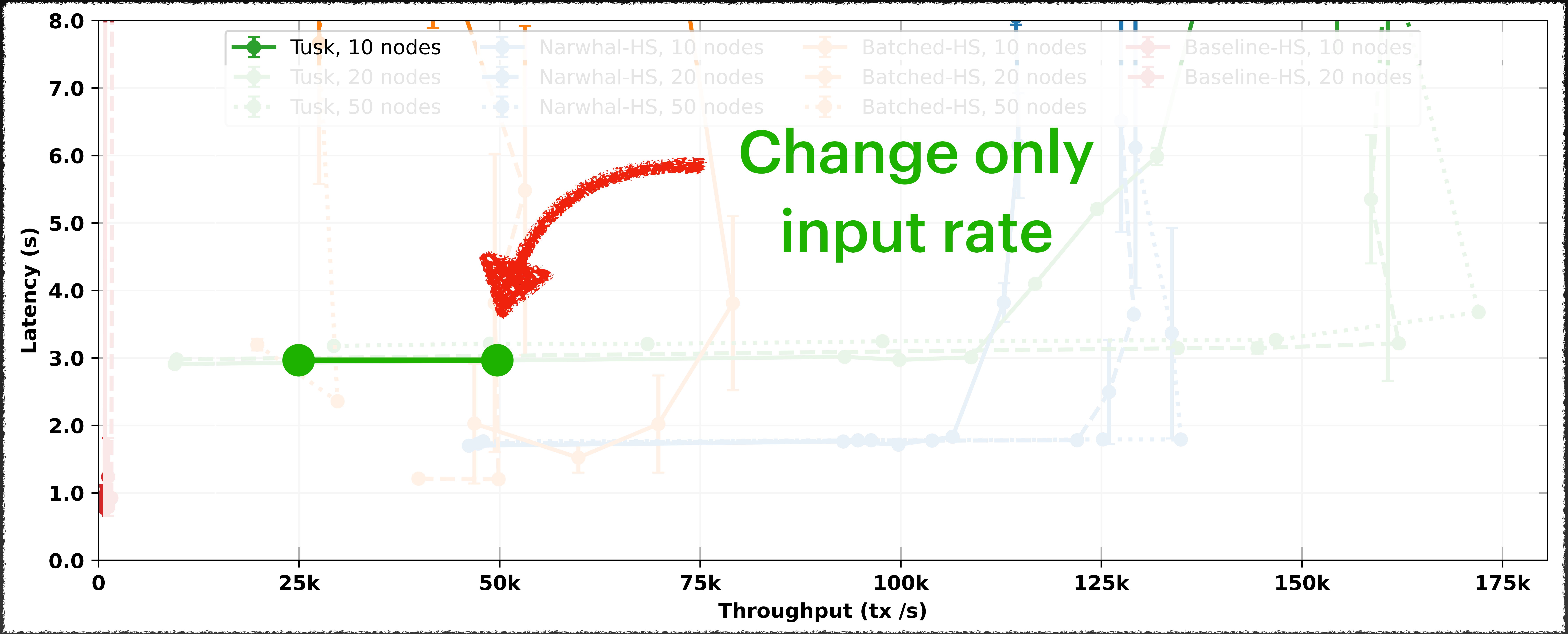
Evaluation

Throughput latency graph



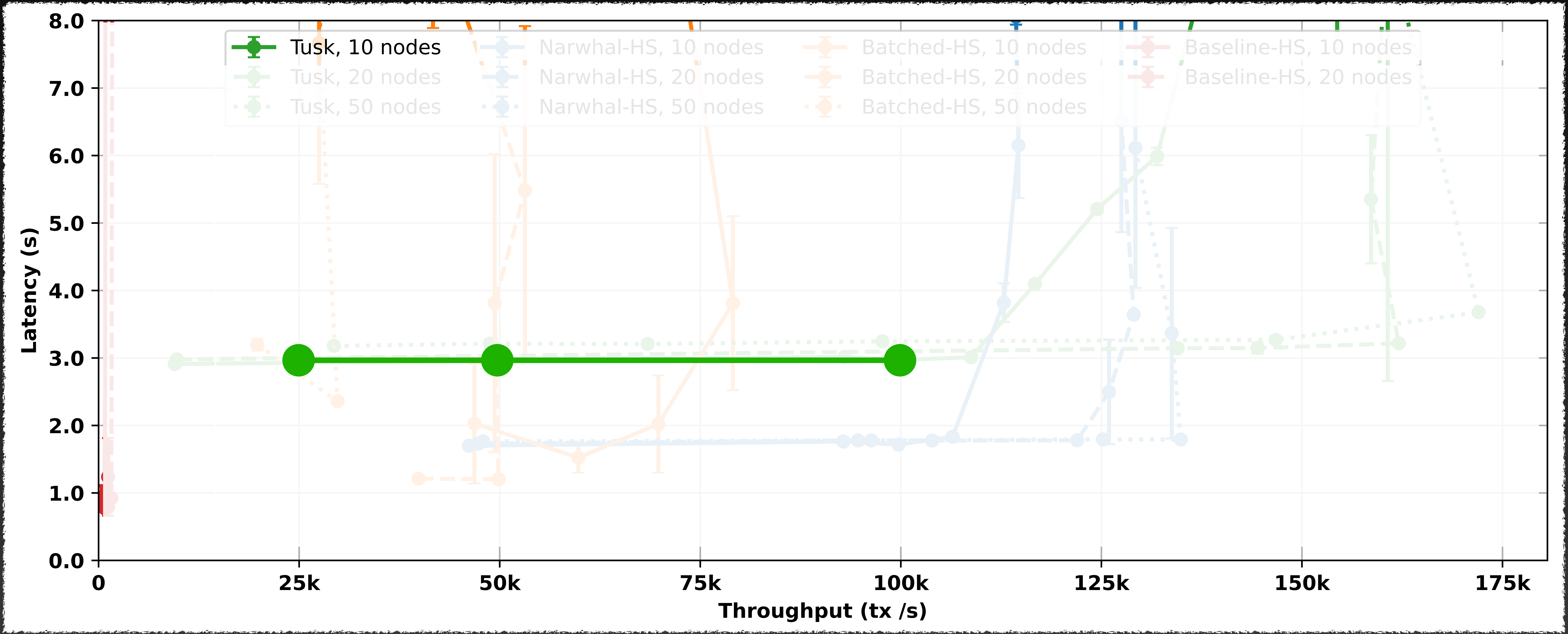
Evaluation

Throughput latency graph



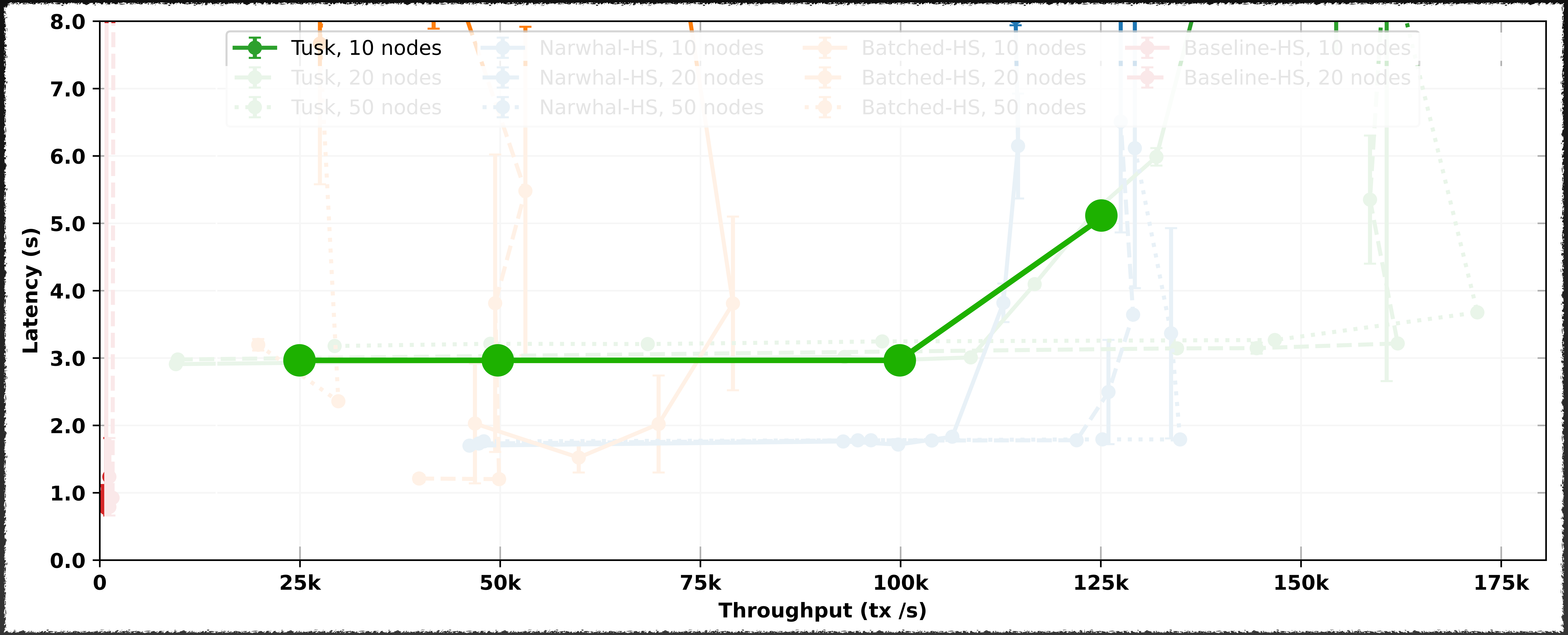
Evaluation

Throughput latency graph



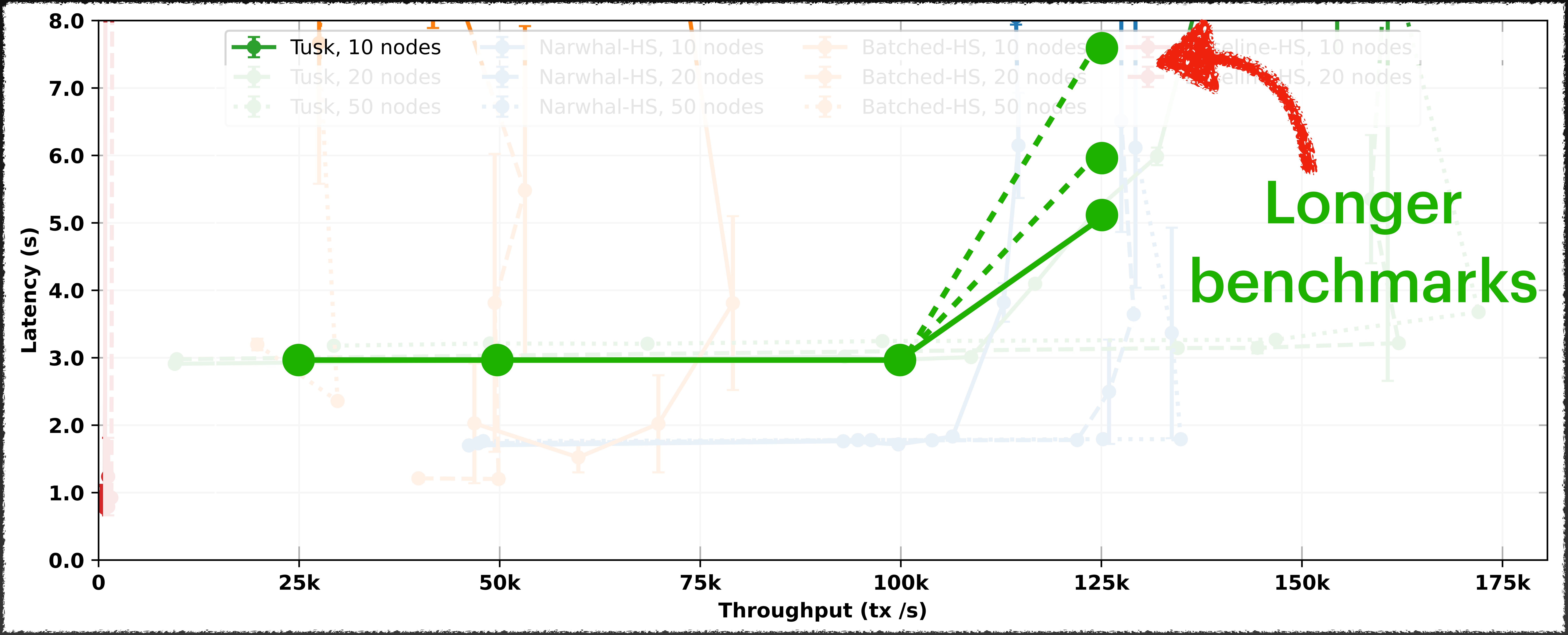
Evaluation

Throughput latency graph



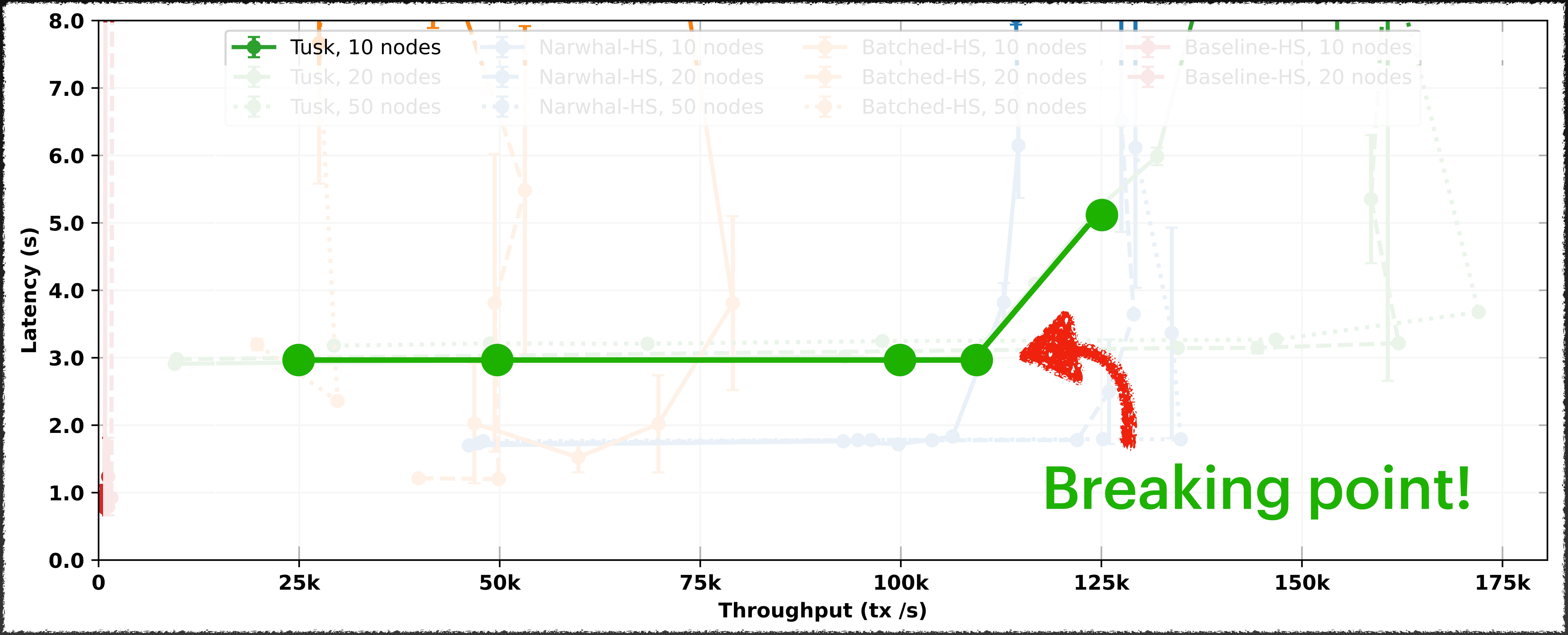
Evaluation

Throughput latency graph



Evaluation

Throughput latency graph



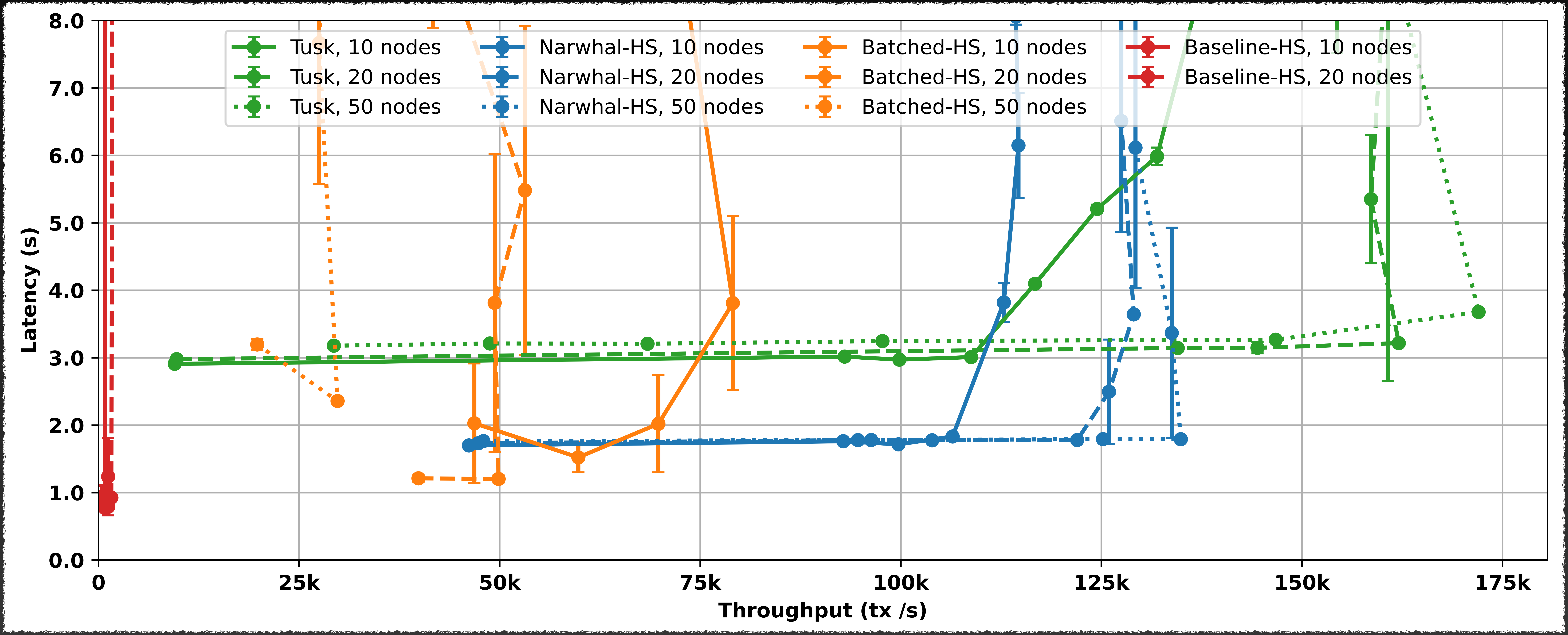
Evaluation

Typical mistakes

- 😫 Forgo persistent storage
- 😫 Do not sanitise messages
- 😫 Local/LAN benchmark + ping
- 😫 Many nodes on same machine
- 😫 Change parameters across runs
- 😫 Set transaction size to zero
- 😫 Preconfigure nodes with txs
- 😫 Send a single burst of transactions
- 😫 Benchmark for a few seconds
- 😫 Start timer in the batch maker
- 😫 Evaluate latency w/ only the first tx
- 😫 Separate latency and throughput
- 😫 Only benchmark happy path

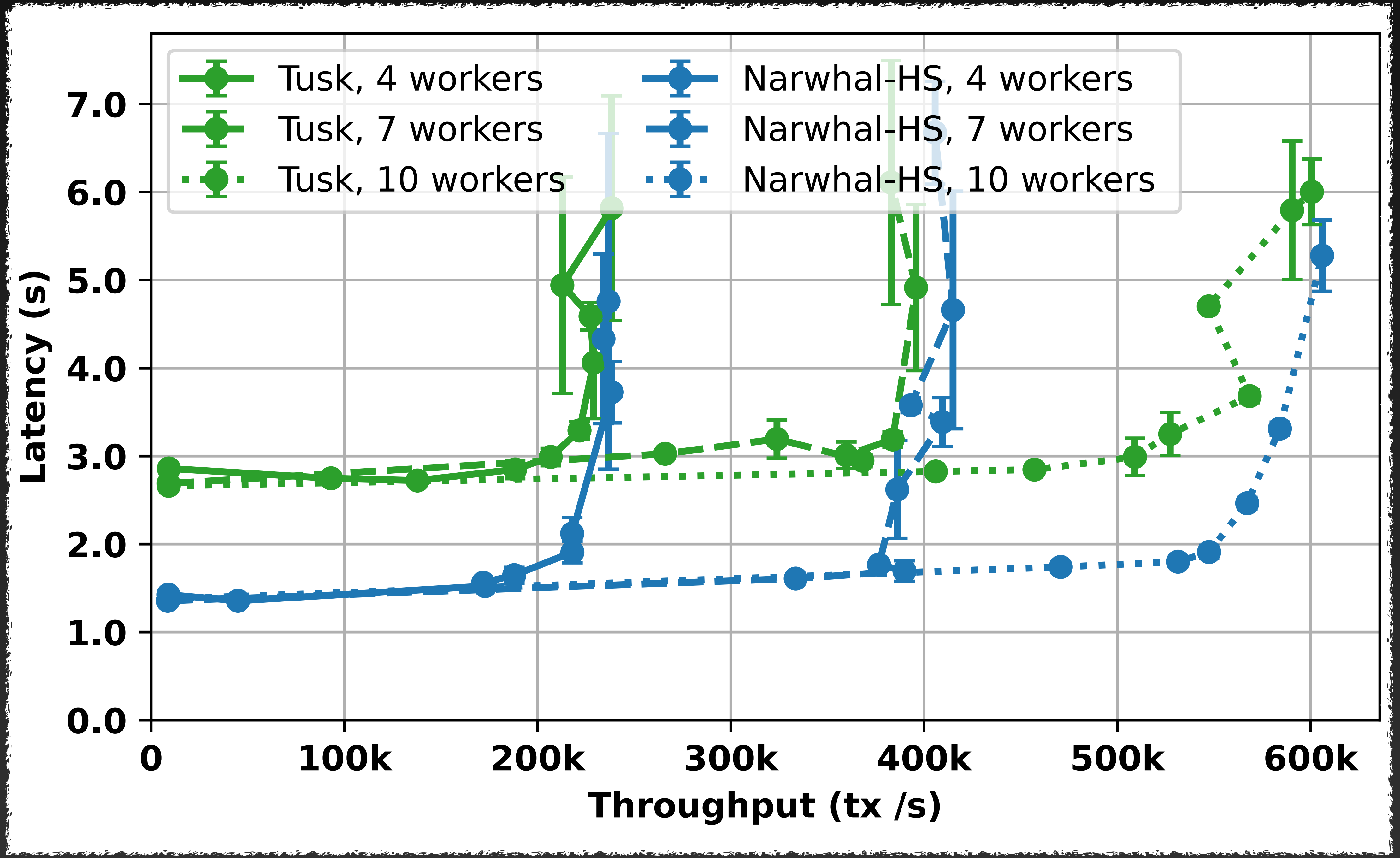
Evaluation

Throughput latency graph



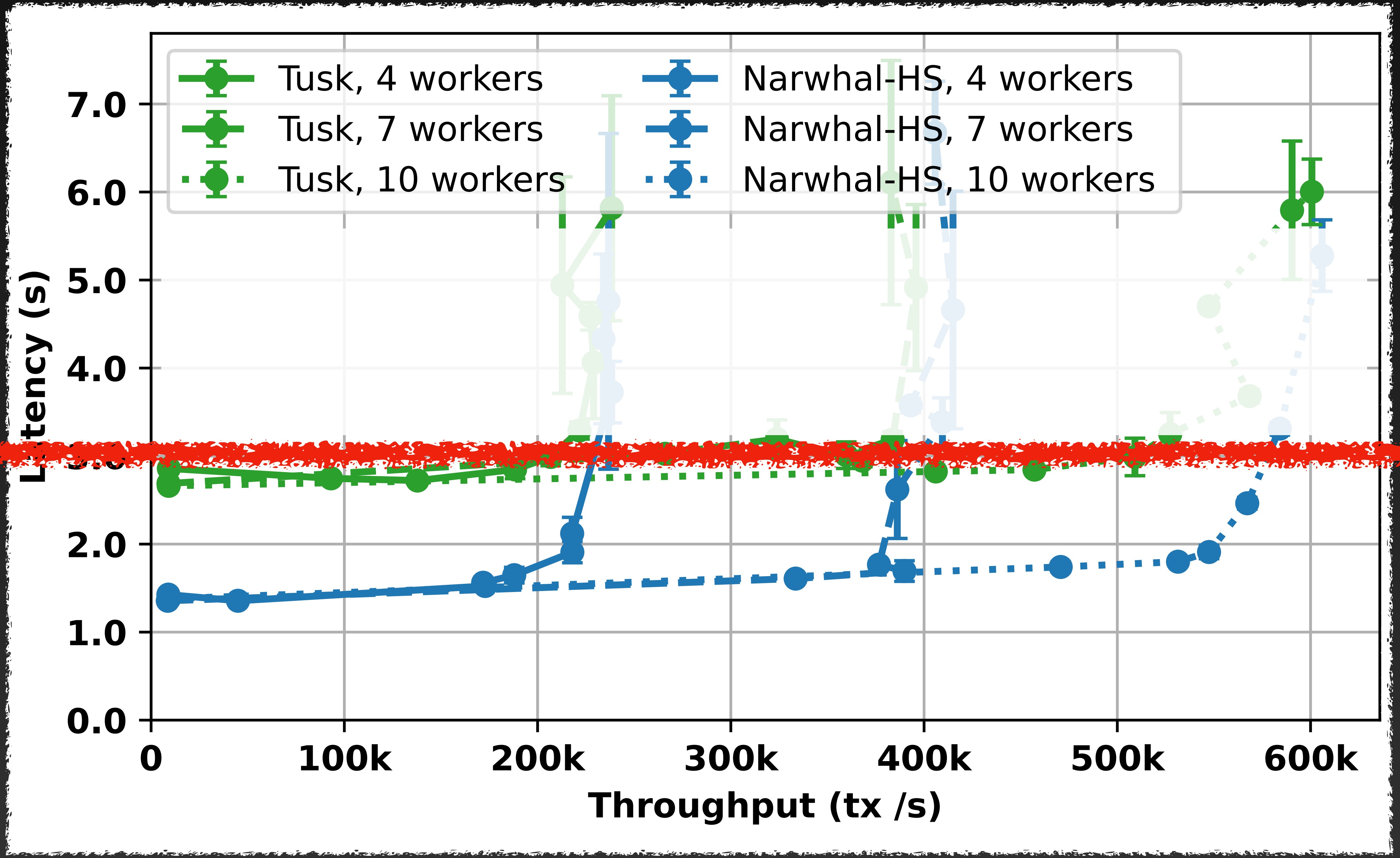
Evaluation

Scalability



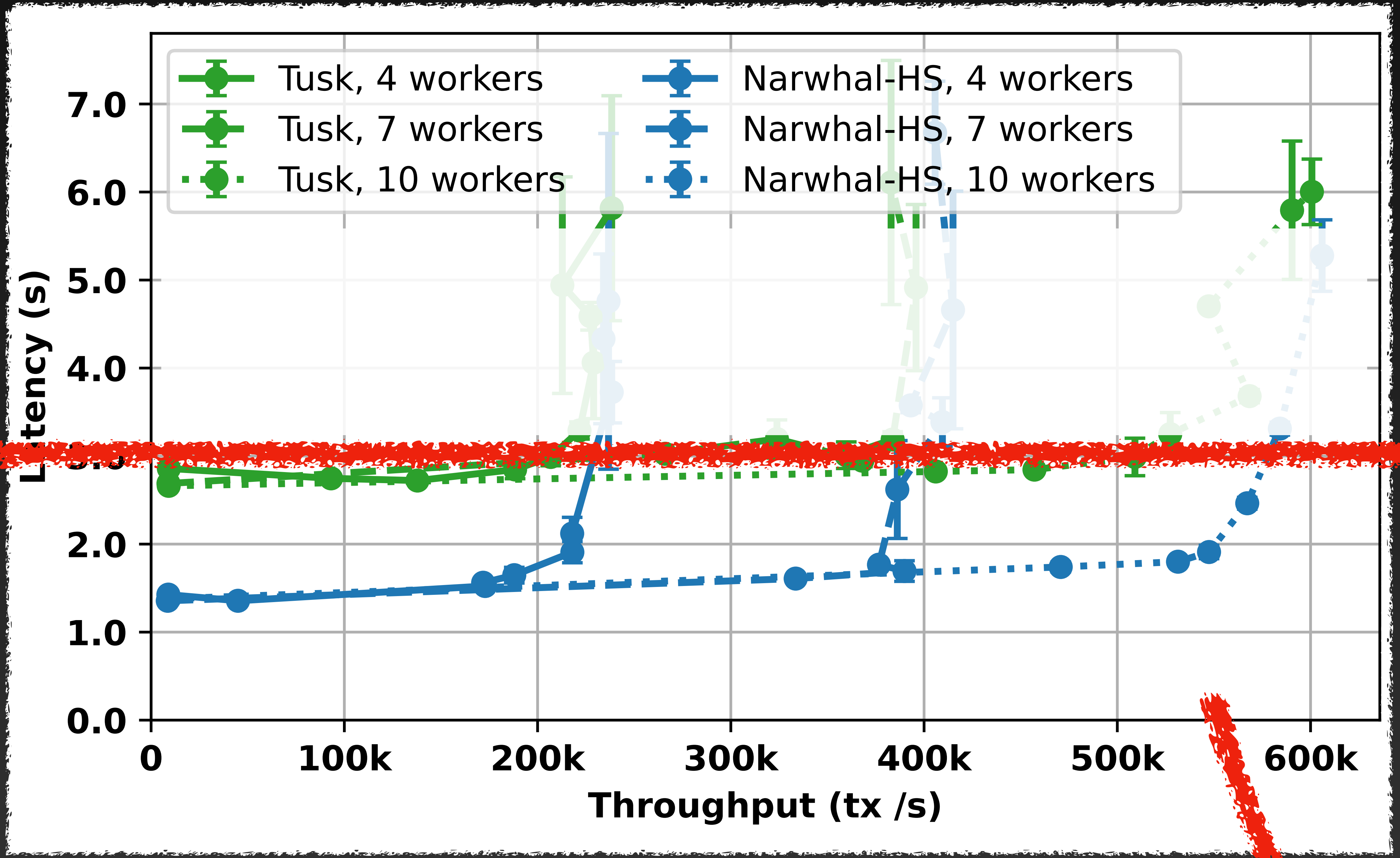
Evaluation

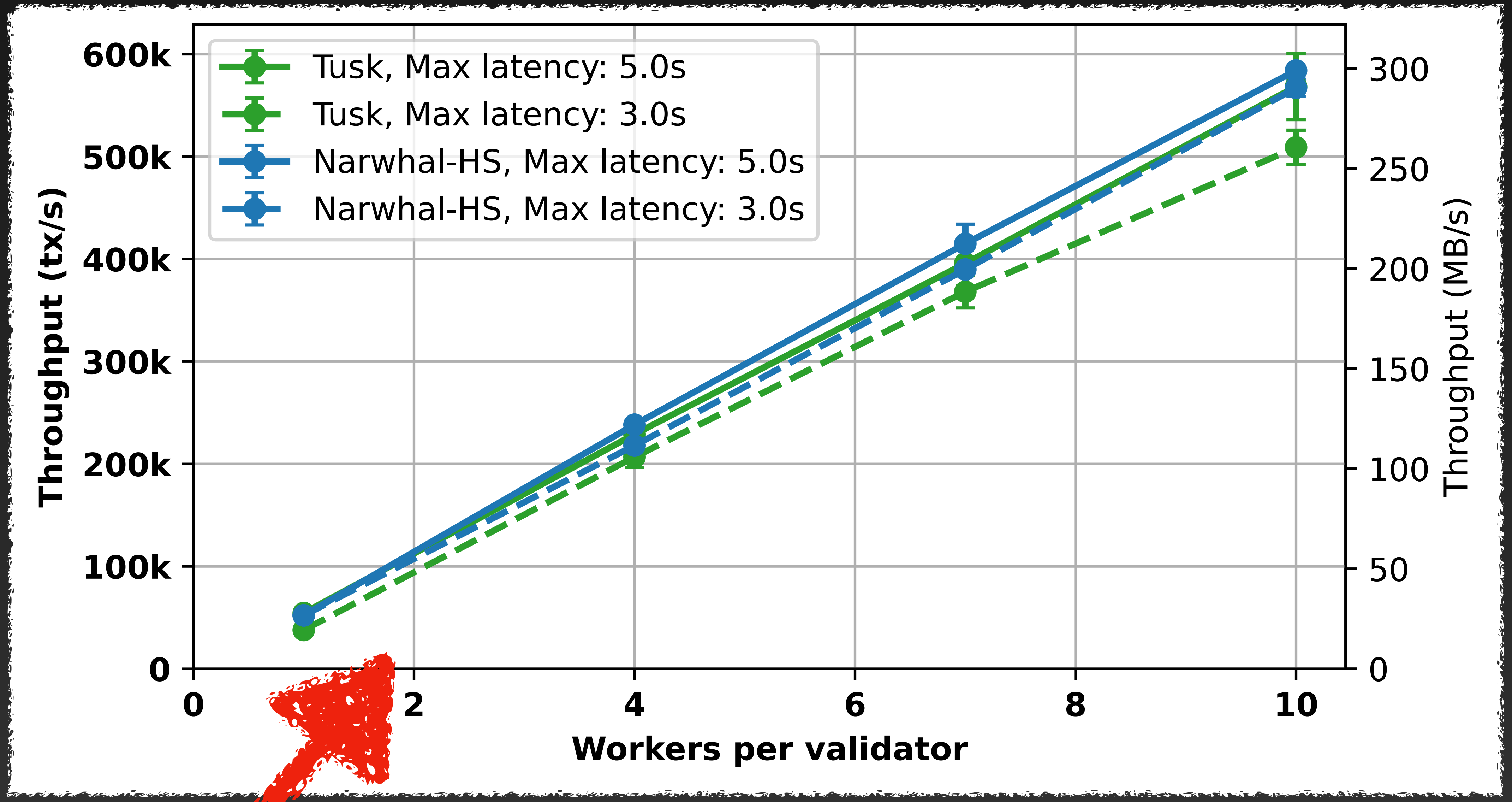
Scalability



Evaluation

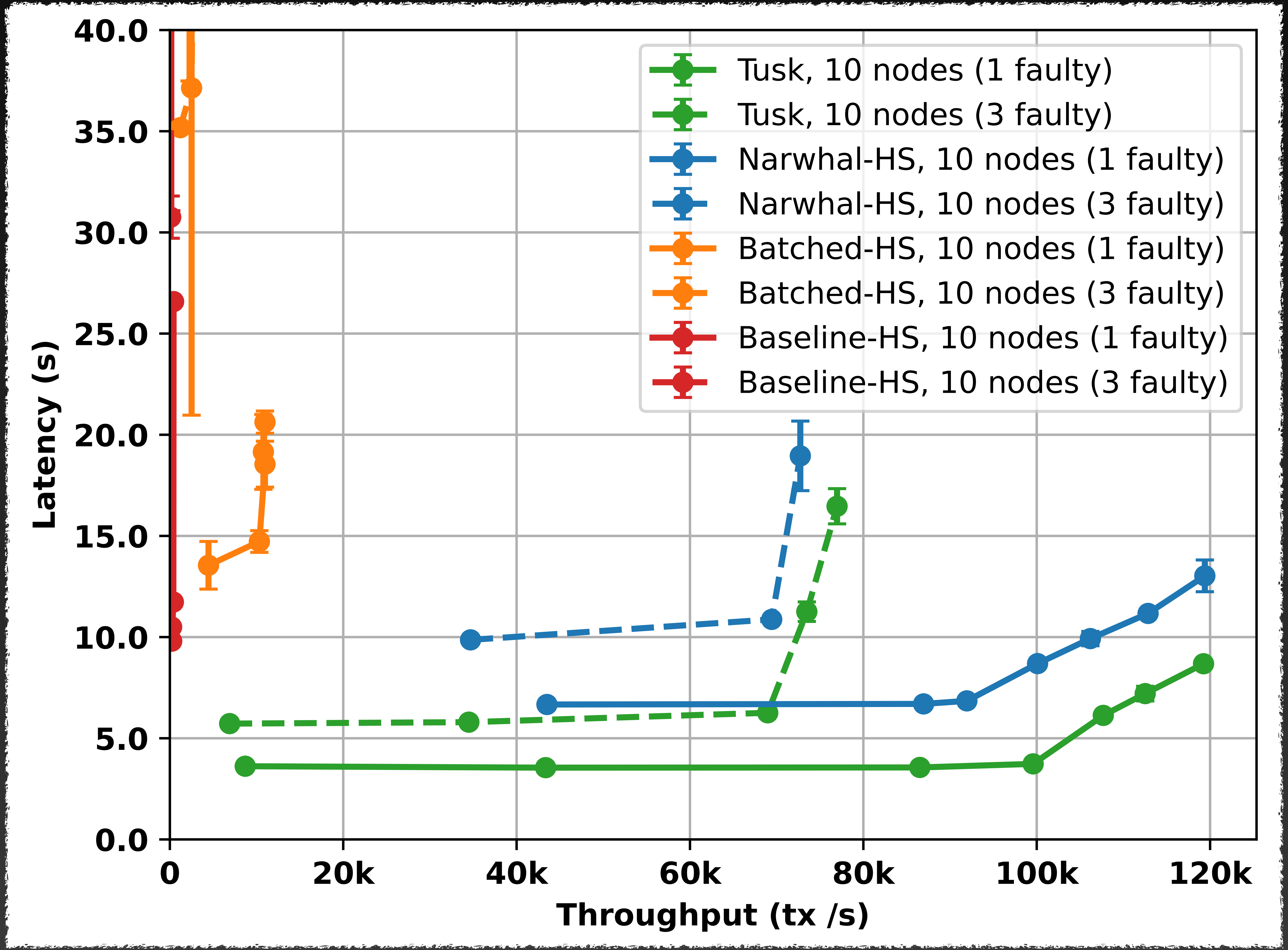
Scalability





Evaluation

Performance under faults



Evaluation

Typical mistakes

- 😫 Forgo persistent storage
- 😫 Do not sanitise messages
- 😫 Local/LAN benchmark + ping
- 😫 Many nodes on same machine
- 😫 Change parameters across runs
- 😫 Set transaction size to zero
- 😫 Preconfigure nodes with txs
- 😫 Send a single burst of transactions
- 😫 Benchmark for a few seconds
- 😫 Start timer in the batch maker
- 😫 Evaluate latency w/ only the first tx
- 😫 Separate latency and throughput
- 😫 Only benchmark happy path

Evaluation

Still many caveats

- Perfect load balance
- Transaction deduplication
- Synthetic load
- No Byzantine adversary
- No network adversary
- Only AWS network