State Machine Replication Scalability Made Simple

Authors: Chrysoula Stathakopoulou, Matej Pavlovic, Marko Vukolic
Team: Matthew Scates, Pavel Frolikov, Malik Smith, Josh Pike

UCDAVIS
COMPUTER SCIENCE

Overview

State Machine Replication:

- Distributed network of machines/nodes
- Machines/Nodes replicate some append only data-structure and coordinate on the state of this data-structure

ISS - Insanely Scalable State Machine Replication

- Solution to State Machine Replication (SMR)
    - Multiplexed instances of single-leader protocols that solve Total Order Broadcast (TOB)
- Efficient and Modular

What is State Machine Replication?

- Clients can make requests to change the state and will get a response back from the network

- Total Order Broadcast (TOB): messages are delivered to every node in the same order

Properties of Total Ordered Broadcast

Agreement	Liveness
Totality	Integrity

SMR Properties

Nodes assign a unique sequence number sn such that these properties hold:

- **Integrity**: If a correct node delivers (sn, r), where $r.id.c$ is a correct client's identity, then client c broadcast r.
- **Agreement**: If two correct nodes deliver, respectively, (sn, r) and (sn, r'), then $r = r'$.
- **Totality**: If a correct node delivers request (sn, r), then every correct node eventually delivers (sn, r).
- **Liveness**: If a correct client broadcasts request r, then some correct node eventually delivers (sn, r).

r = (o, id) is a client request, where o is the payload and id is a unique identifier

Where does their solution fit in?

Single Leader TOB

- PBFT
- Paxos
- Raft
- HotStuff

- Single Leader Bottleneck

Parallel-Leader TOB

- RCC
- BFT-Mencius
- HoneyBadgerBFT

- Duplicate Requests

Parallel TOB w/o Duplication

- MIR-BFT
- FnF

- Made to parallelize specific TOB protocols

What does Insanely Scalable SMR (ISS) do?

- Duplication prevention
  - Resilient request partitioning mechanism from MIR-BFT
- Parallel leader protocol w/o epoch primary
  - I.e. Parallel instances of TOB are independent
  - Avoid difficult complications when this primary crashes or is byzantine
- Simple and modular design
  - General enough to accommodate most leader-driven ordering protocols (BFT or CFT) and make them scale.
- Assumes partially synchronous network model

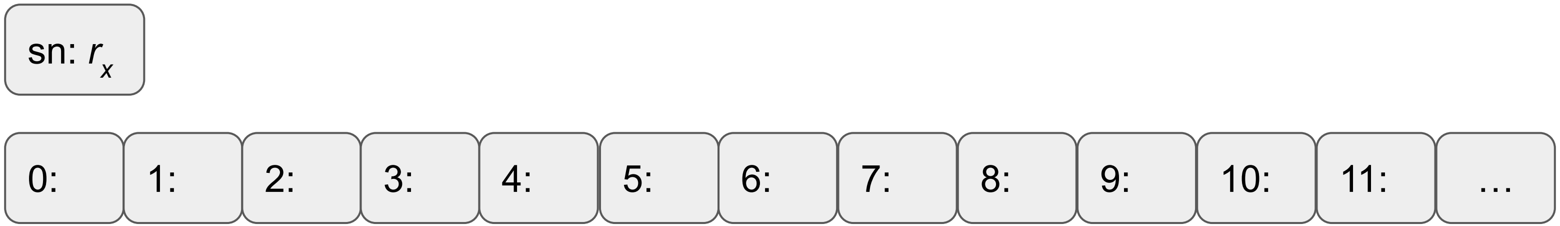
High-Level: Log

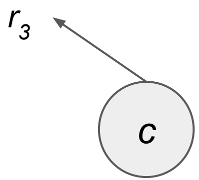
- Contiguous, ordered log of (“batches of”) requests *r*
- Sequence Number *sn* represents offset from 0^{th} position of log

sn: r_x

0:	1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	…

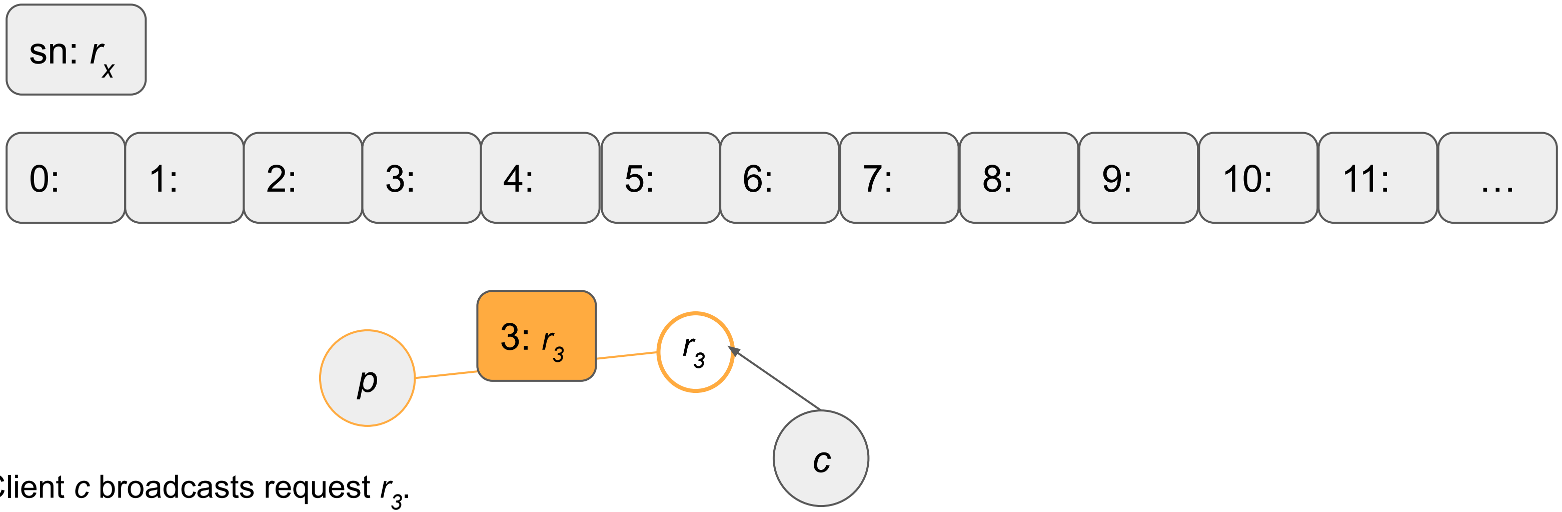
High-Level: Log





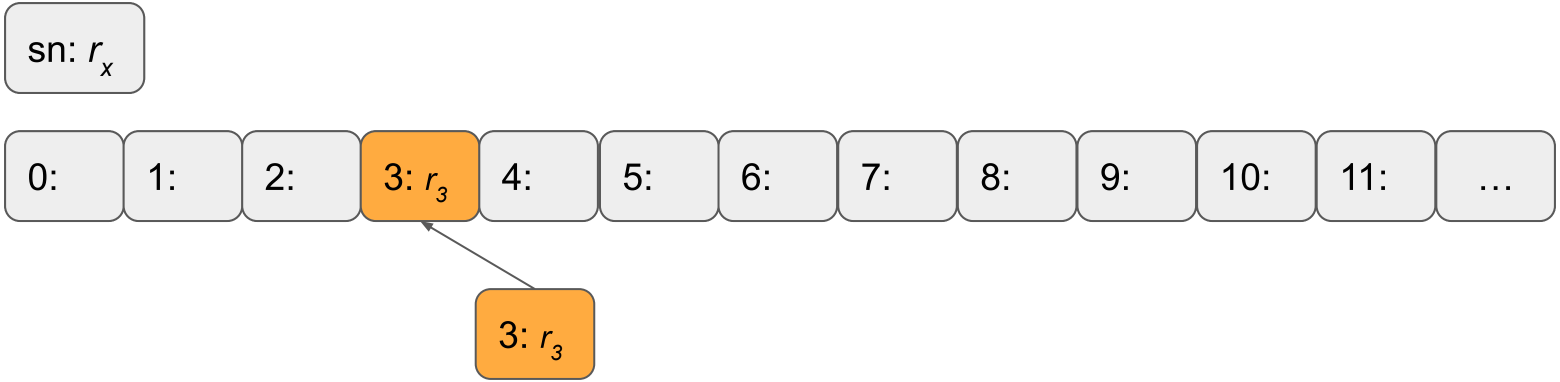
1. Client c broadcasts request r_3.

High-Level: Log



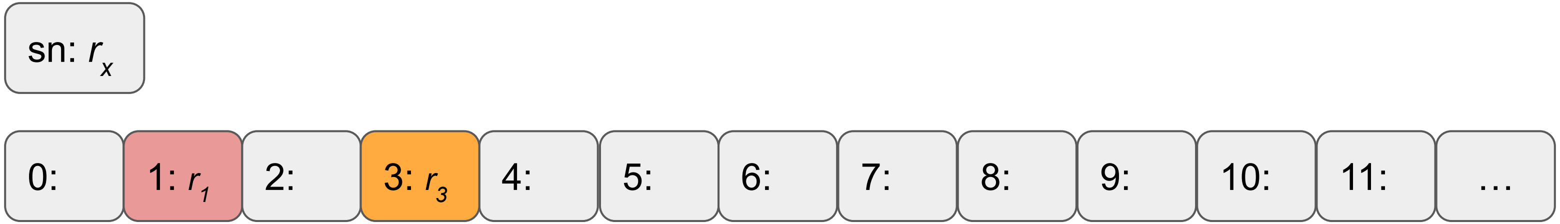
1. Client c broadcasts request r_3.
2. Leader Node p assigns sequence number 3 to request.

High-Level: Log



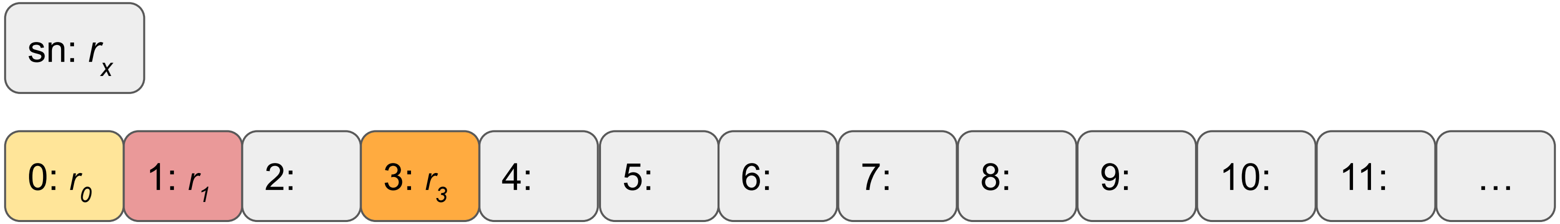
1. Client *c* <u>broadcasts</u> request r_3.
2. Leader Node *p* assigns sequence number 3 to request.
3. Request r_3 <u>committed</u> with the assigned sequence number 3 and added to the log.

High-Level: Log



1. Client *c* broadcasts request r_3.
2. Leader Node *p* assigns sequence number 3 to request.
3. Request r_3 <u>committed</u> with the assigned sequence number 3 and added to the log.
4. All positions preceding request's log position filled.

High-Level: Log



1. Client c broadcasts request r_3.
2. Leader Node p assigns sequence number 3 to request.
3. Request r_3 <u>committed</u> with the assigned sequence number 3 and added to the log.
4. All positions preceding request's log position filled.

High-Level: Log

sn: r_x

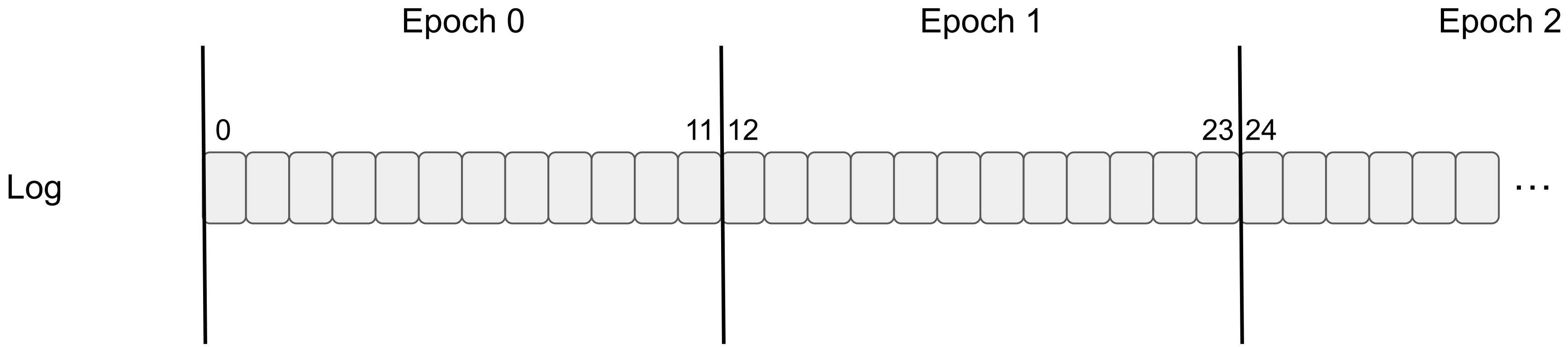
0: r_0	1: r_1	2: r_2	3: r_3	4:	5:	6:	7:	8:	9:	10:	11:	…

Preceding spots filled!!

1. Client *c* broadcasts request r_3.
2. Leader Node *p* assigns sequence number 3 to request.
3. Request r_3 committed with the assigned sequence number 3 and added to the log.
4. All positions preceding request's log position filled.
5. Request r_3 is ***delivered*** and can be executed.

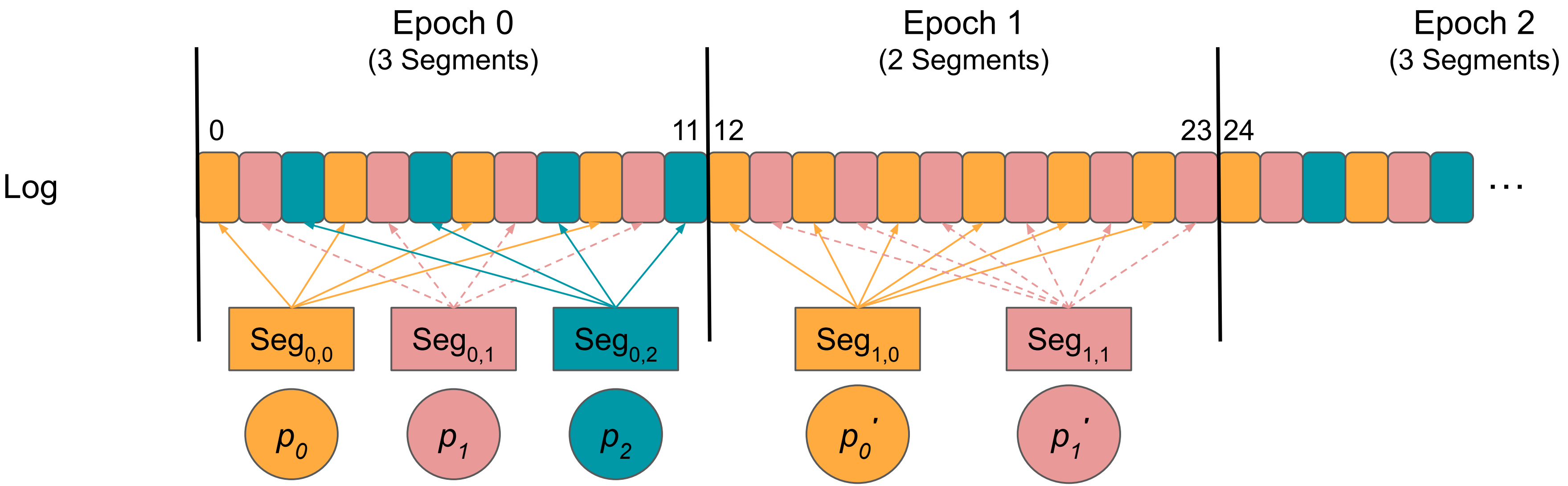
Epochs

- Finite partitions of the log
- Epochs processed sequentially



Segments

- Epochs are further partitioned into subsets of sequence numbers
- Each segment corresponds to <u>one</u> instance of Sequenced Broadcast
- ISS assigns a leader node *p* to each segment



Sequenced (Total Order) Broadcast ***SB***

- Variant of Byzantine Total Order Broadcast (TOB)
- Each segment corresponds to <u>one</u> instance of SB
- Segment's Leader = SB's Leader
- Batches of requests broadcast to SB instance Batches of responses delivered back

<u>**Key Differences to TOB**</u>:

- Instantiated with an explicit set of sequence numbers and allowed messages
- Instantiated with Failure Detector D of the class $\Diamond$*S(bz)*
    - Detects <u>quiet nodes</u>
- Correct nodes deliver messages from the allowed set of messages and $\perp$
- SB terminates for all sequence numbers.

Sequenced Broadcast ***SB*** - Failure Detector

A failure detector of the ◊*S(bz)* class guarantees:

Strong Completeness:

There is a time after which every quiet node is permanently suspected by every correct node.

Eventual Weak Accuracy:

There is a time after which some correct node is never suspected by any correct node.

Sequenced Broadcast: Properties

<u>Integrity:</u> If a correct node delivers (*sn*, *msg*) with the message not being bottom and the sender is correct then the sender broadcast (*sn*, *msg*)

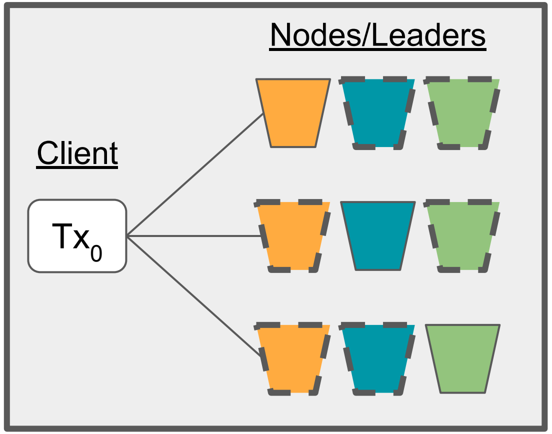
<u>Agreement:</u> If two correct nodes deliver a message with the same sequence number then their messages must be the same

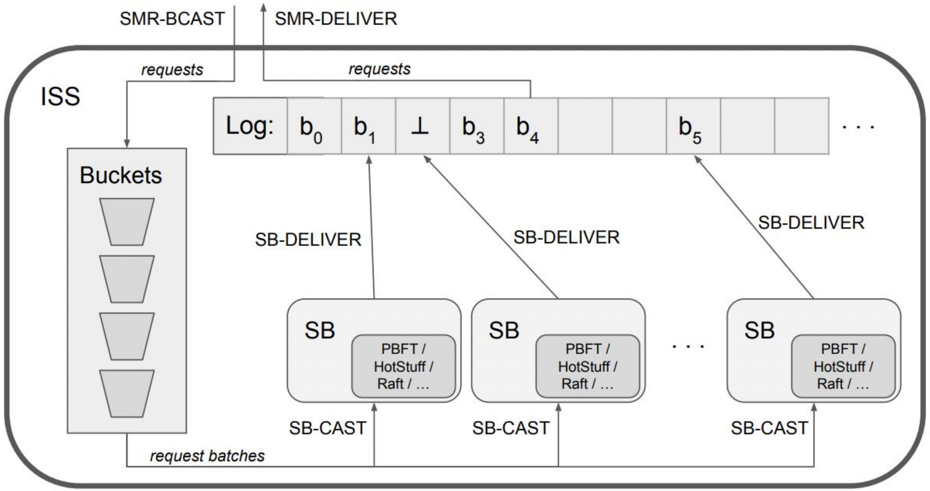
<u>Termination:</u> The SB instance eventually delivers a message for every sequence number in the Segment (the message or bottom)

<u>Eventual Progress:</u> If a correct node delivers bottom for some sequence number, this node suspected a sender after the SB instance was initialized

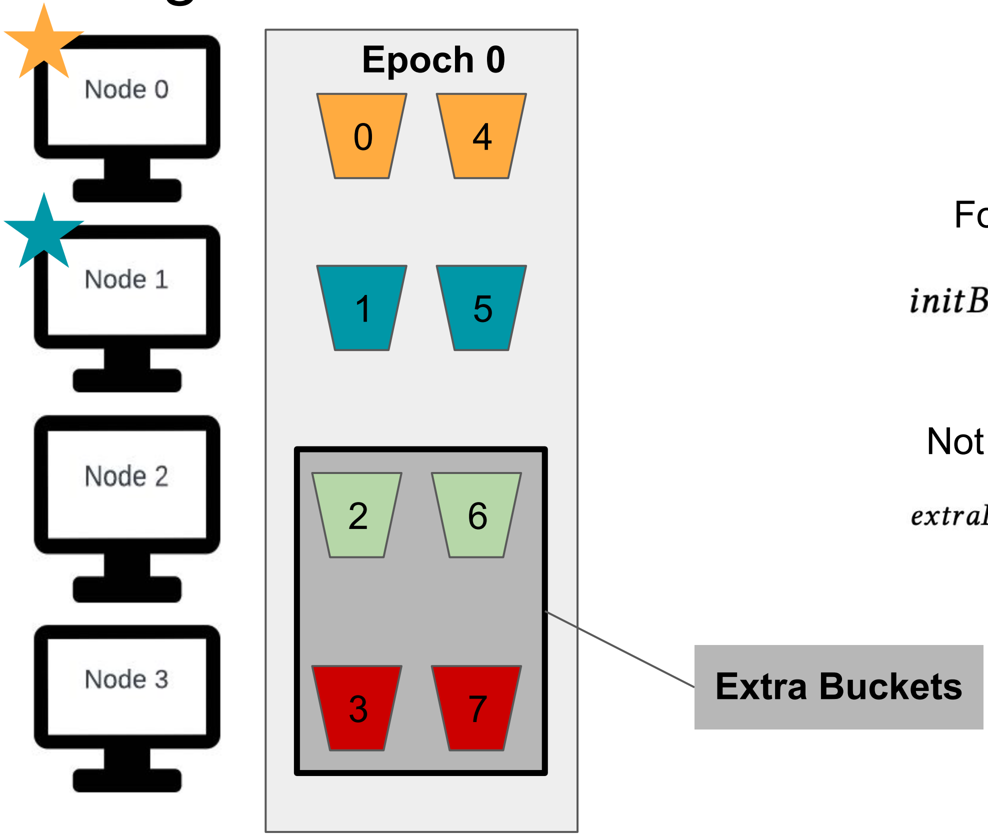
Protocol: Request batches

- When a client sends a request, ISS adds it to its corresponding bucket
- A leader can only propose a batch of request from its assigned buckets
- Before proposing a batch a leader:
    - Waits for bucket to have enough request
    - Waits until timeout occurs





Initializing The Buckets



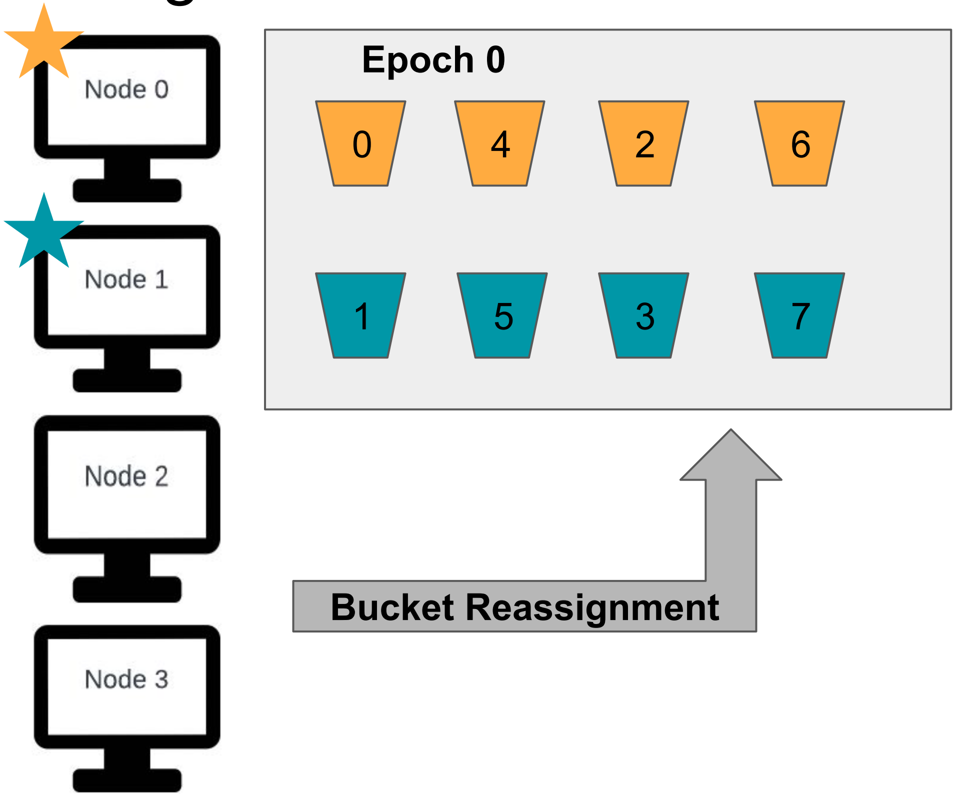
For epoch e, from nodes 0≤i≤n.

$$initBuckets(e, i) = \{b \in \mathcal{B} \mid (b + e) \equiv i \bmod n\}$$

Not all buckets are assigned to a leader

$$extraBuckets(e) = \{b \in \mathcal{B} \mid \exists i : i \notin Leaders(e) \wedge b \in initBuckets(e, i)\}$$

Initializing The Buckets: Redistribution

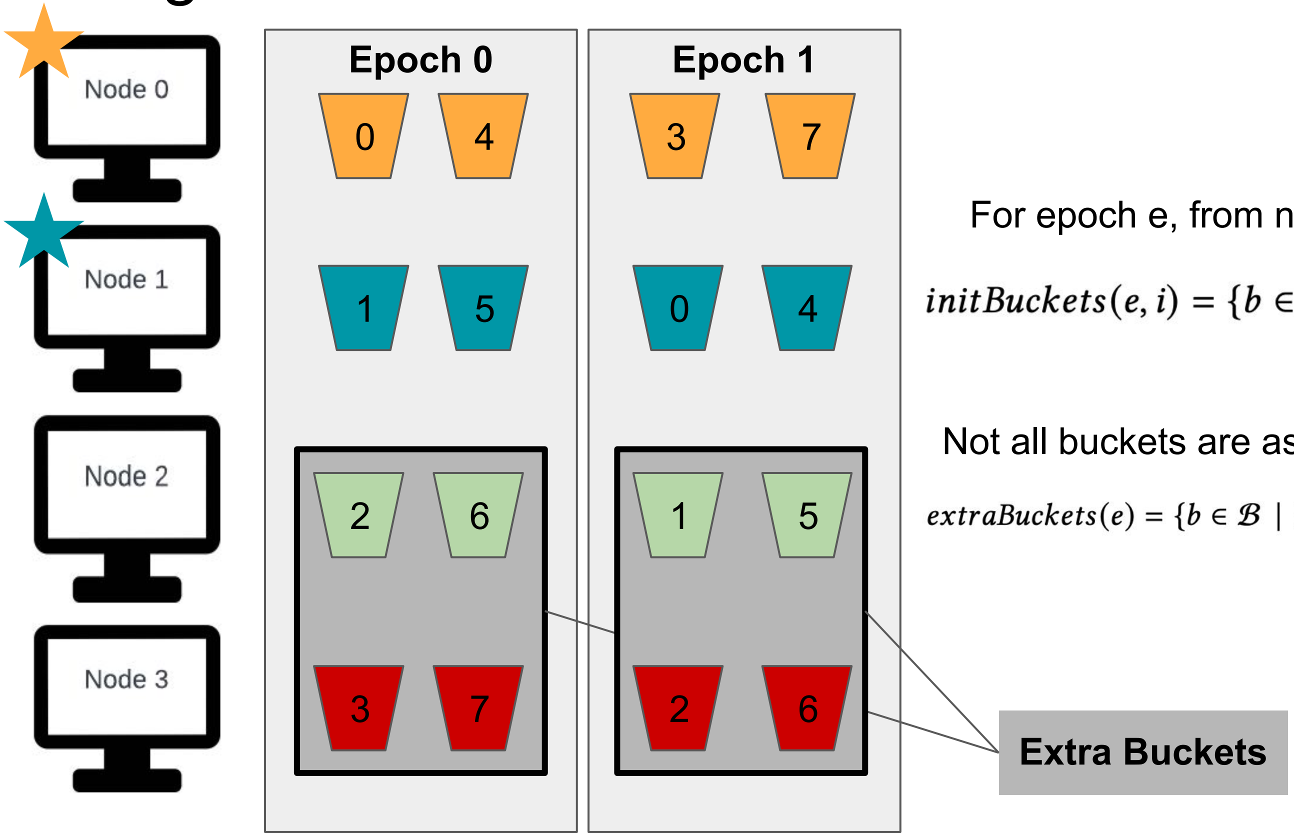


Here, we redistribute the buckets.

Where Leaders(e) is the set of leaders on epoch e, and l(e,k) is the k-th leader on lexicographic order for epoch e

$$
\begin{aligned}
Buckets(e, l(e,k)) = \quad & initBuckets(e, l(e,k)) \cup \\
& \{b \in extraBuckets(e) \mid \\
& (b + e) \equiv k \bmod |Leaders(e)|\}
\end{aligned}
$$

Initializing The Buckets



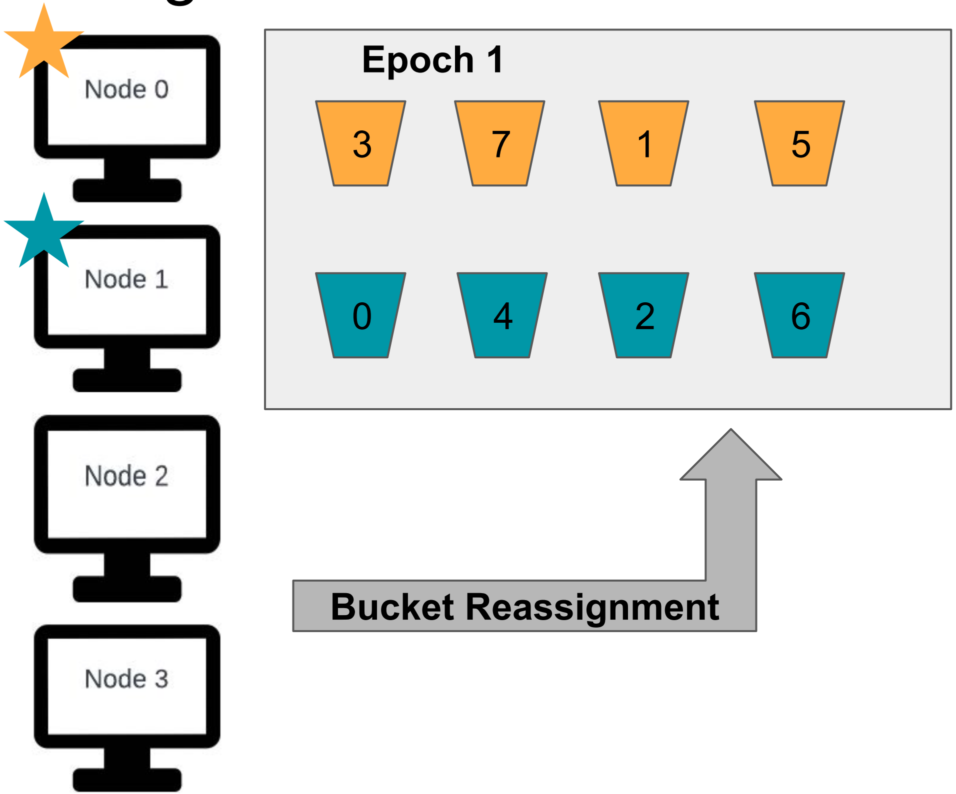
For epoch e, from nodes 0≤i≤n.

$$initBuckets(e, i) = \{b \in \mathcal{B} \mid (b + e) \equiv i \bmod n\}$$

Not all buckets are assigned to a leader

$$\begin{aligned} extraBuckets(e) = \{b \in \mathcal{B} \mid \exists i : i \notin Leaders(e) \\ \land\, b \in initBuckets(e, i)\} \end{aligned}$$

Initializing The Buckets: Redistribution



Redistribute the buckets again…

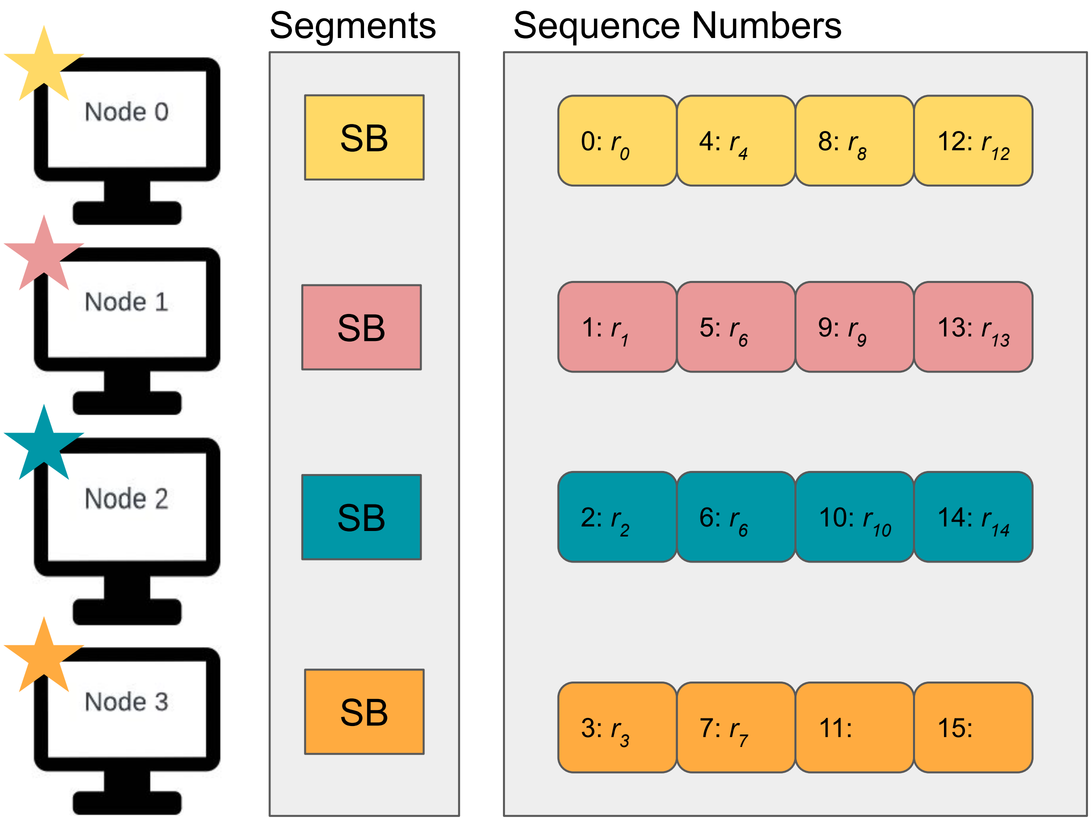
Where Leaders(e) is the set of leaders on epoch e, and l(e,k) is the k-th leader on lexicographic order for epoch e

$$\begin{aligned} Buckets(e, l(e,k)) = \quad & initBuckets(e, l(e,k)) \cup \\ & \{b \in extraBuckets(e) \mid \\ & (b + e) \equiv k \bmod |Leaders(e)|\} \end{aligned}$$

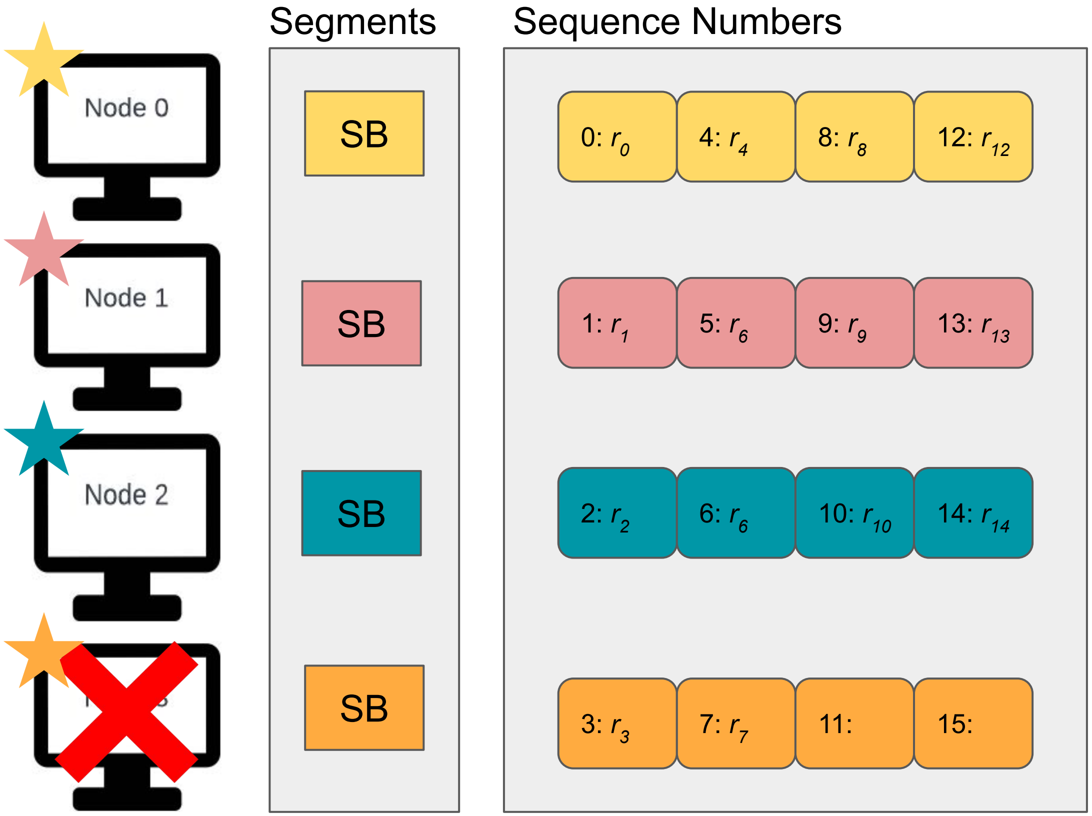
Node Crashes

- Once detected, the leader selection policy removes the faulty node from the leaderset
- Faults to Consider:
  - Epoch-start failure
  - Epoch-end failure (worst case)
  - Byzantine Stragglers
- A temporary node becomes leader of a crashed segment
  - Temporary leader can only assign a nil value (⊥)

Node Crashes

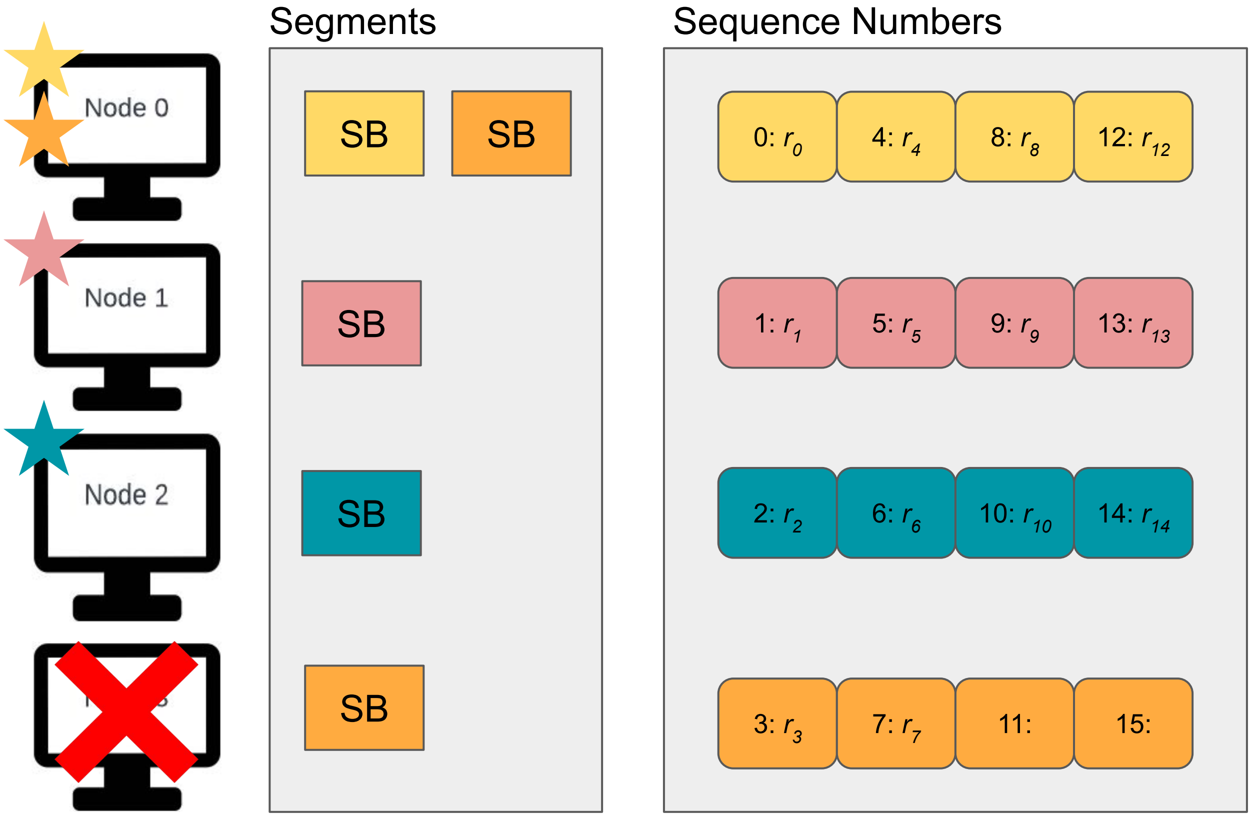


Node Crashes



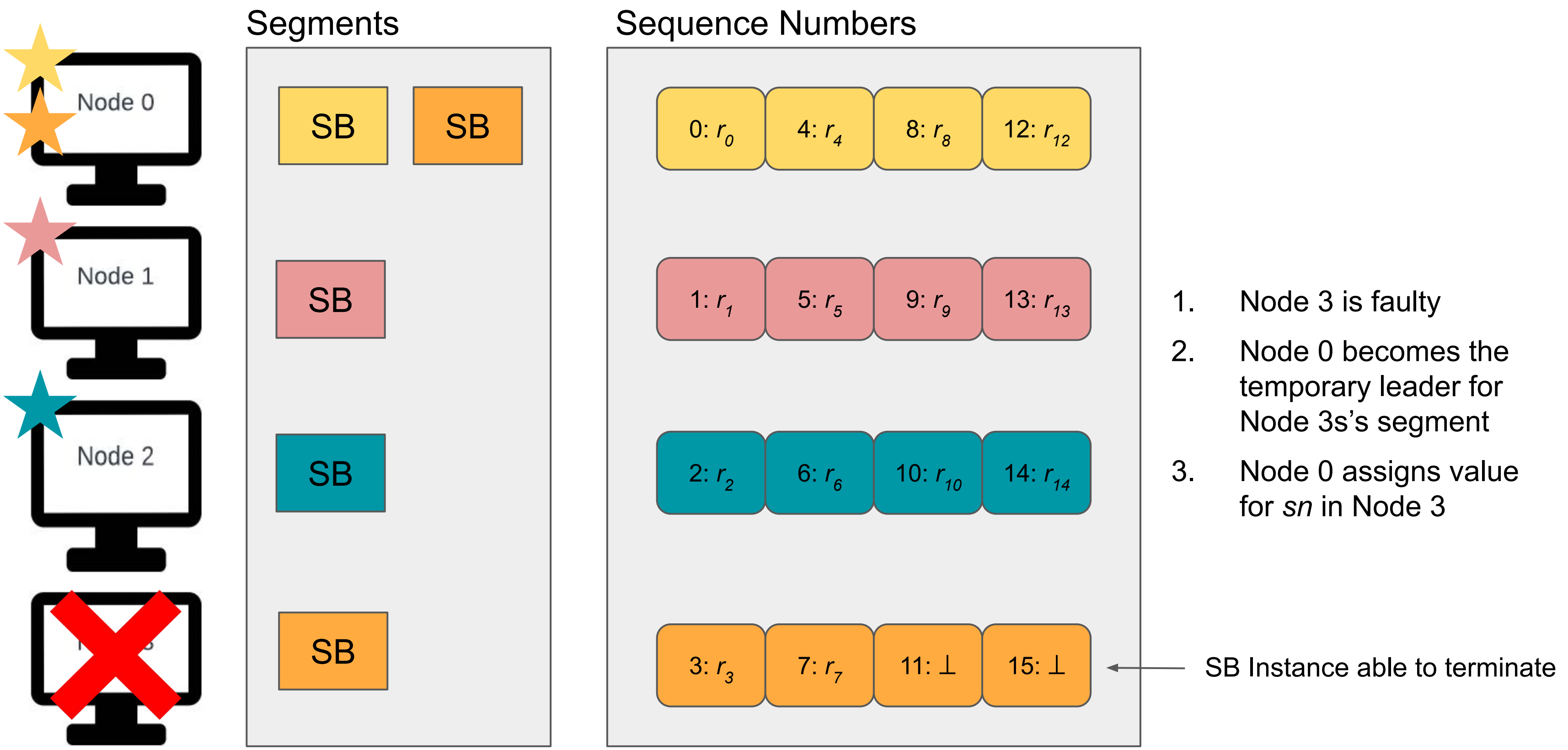
1. Node 3 is faulty

Node Crashes



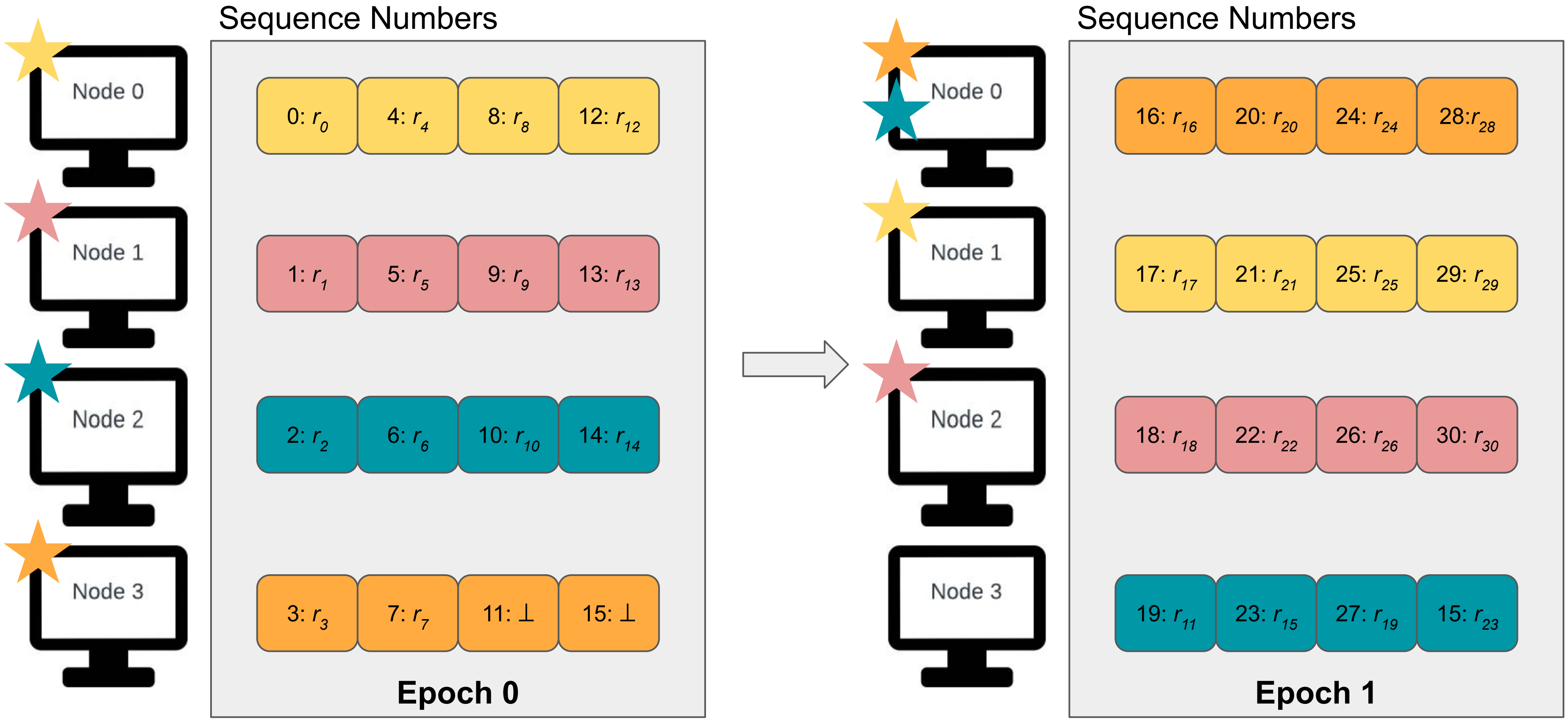
1. Node 3 is faulty
2. Node 0 becomes the temporary leader for Node 3s's segment

Node Crashes



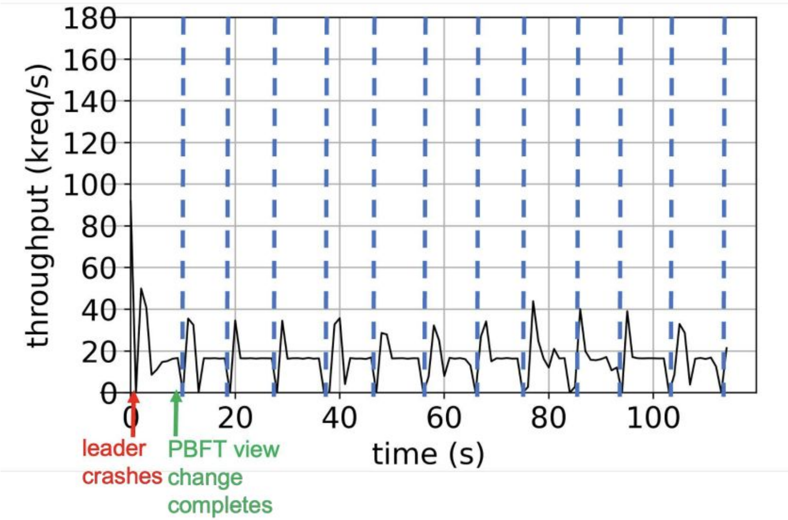
1. Node 3 is faulty
2. Node 0 becomes the temporary leader for Node 3s's segment
3. Node 0 assigns value for *sn* in Node 3

Node Crashes



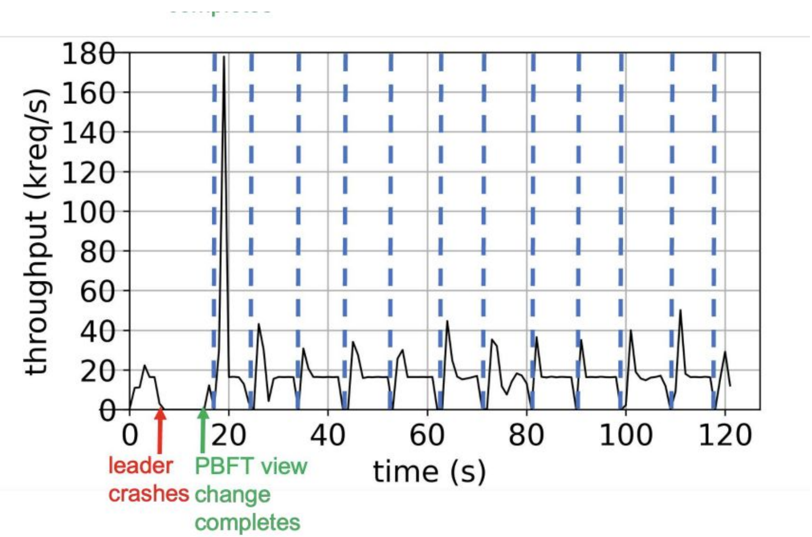
Epoch-start Failure

- Occurs at the beginning of an epoch
- Worst-case scenario for the number of proposed sequence numbers in an epoch



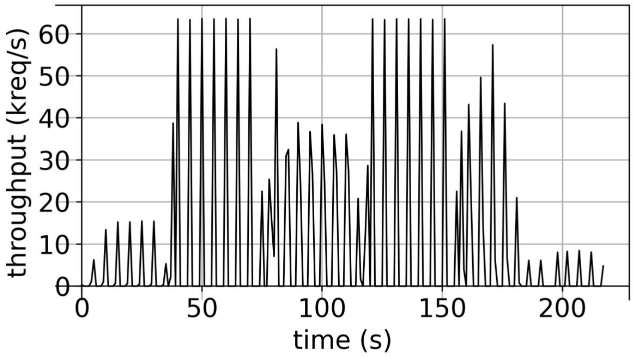
Epoch-end Failure

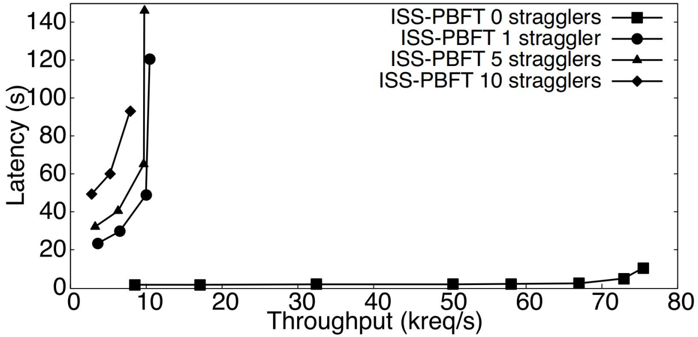
- Occurs at the end of an epoch
- Epoch change is further delayed
- Causes an increase in the latency



Byzantine Straggler

- Attempt to delay to proposed request for as long as possible, without being detected
- Straggler avoids proposing request
  - Harm latency and throughput
  - Creates “holes” in the log





Implementation

- Programmed in the Go language
  - Used gRPC for communication with TLS protocol
- Raft, Chained HotStuff, and PBFT can be used for request batches.
- PBFT was used, with some changes
  - No times outs of single requests
  - Makes sure to commit some batch before a timeout, then resets the timer.
  - View change prevented by proposing an empty batch when there is no incoming requests
- Raft with changes
  - Fix a leader to skip election phase
- The leader selection policy used is the BFT-Mencius policy from “Bounded Delay in Byzantine-Tolerant State Machine Replication” by Milosevic et al.

Experiment

- Used a Wide-Area Network which spans 16 data centers all around the world.
- Used virtual machines with 32 x 2.0 GHz VCPUs and 32GB RAM running Ubuntu Linux 20.04
- 500 bytes per request
- Fixed batched size

	PBFT	**HotStuff**	**Raft**
Initial lederset size	$\lvert\mathcal{N}\rvert$	$\lvert\mathcal{N}\rvert$	$\lvert\mathcal{N}\rvert$
Max batch size	2048	4096	4096
Batch rate	32 b/s	not applicable	32 b/s
Min batch timeout	0 s	1 s	0 s
Max batch timeout	4 s	0	4 s
Min epoch length	256	256	256
Min segment size	2	16	16
Epoch change timeout	10 s	10 s	[10,20) s
Buckets per leader	16	16	16
Client signatures	256-bit ECDSA	256-bit ECDSA	none

Table 1: ISS configuration parameters used in evaluation

Table 1 from Stathakopoulou's et al.'s paper

Results: Scalability

- Performance Improvement for a single leader from 1 to 128 nodes:
  - 37x improvement for PBFT
  - 56x improvement for Chained HotStuff
  - 55x for Raft

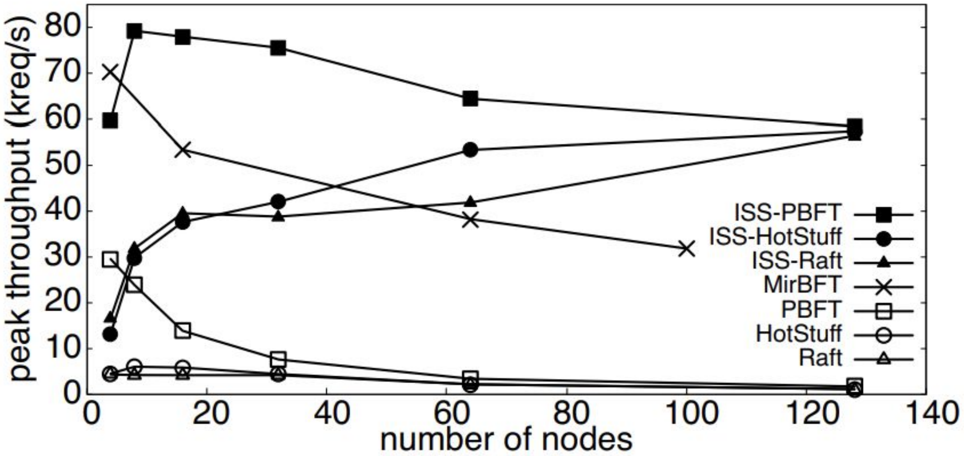


Figure 5 from Stathakopoulou et al.'s paper

Results: Throughput v Latency for PBFT

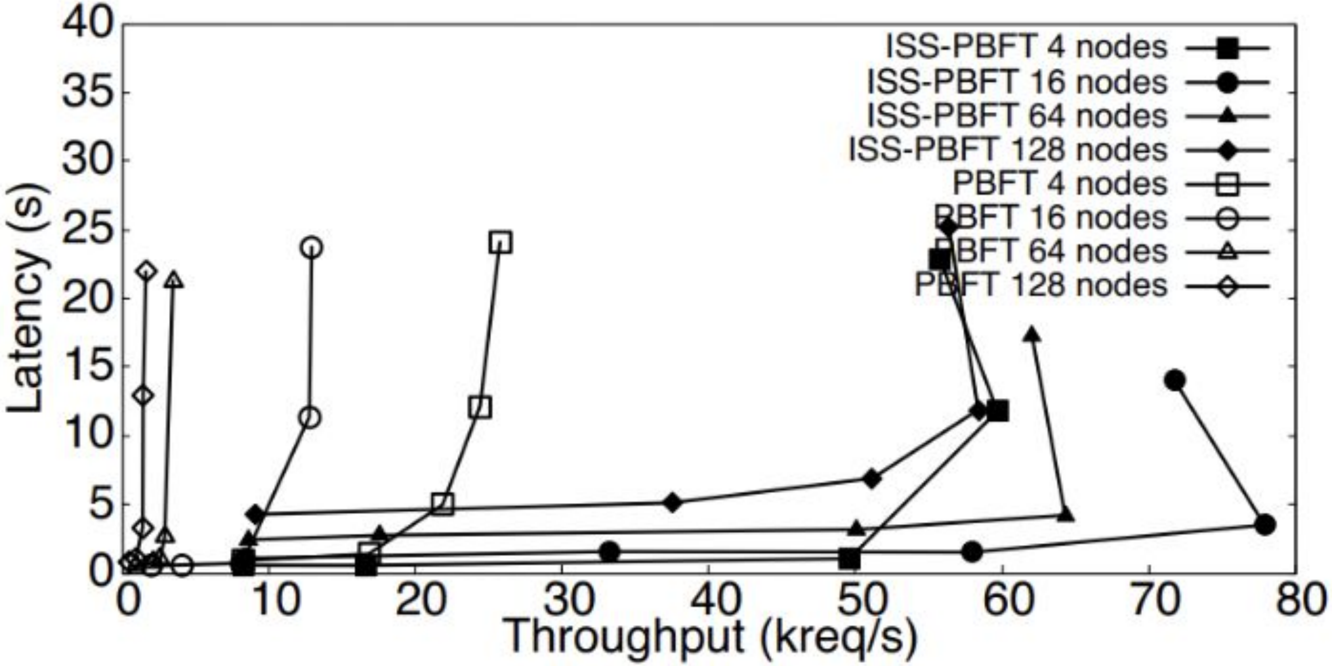


Figure 6 from Stathakopoulou et al.'s paper.

Results: Throughput v Latency for Raft

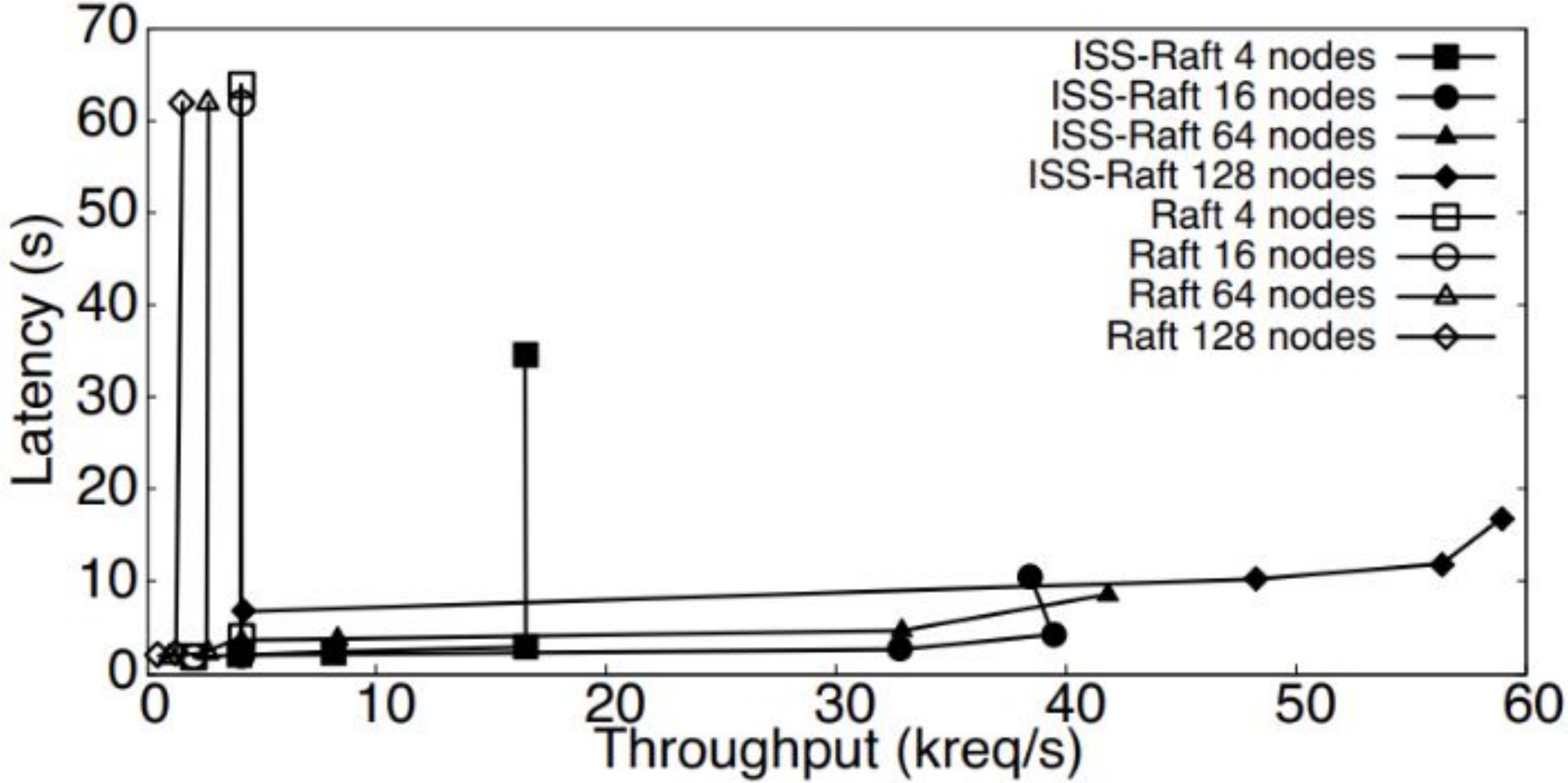


Figure 6 from Stathakopoulou et al.'s paper.

Results: Throughput for HotStuff-Chained

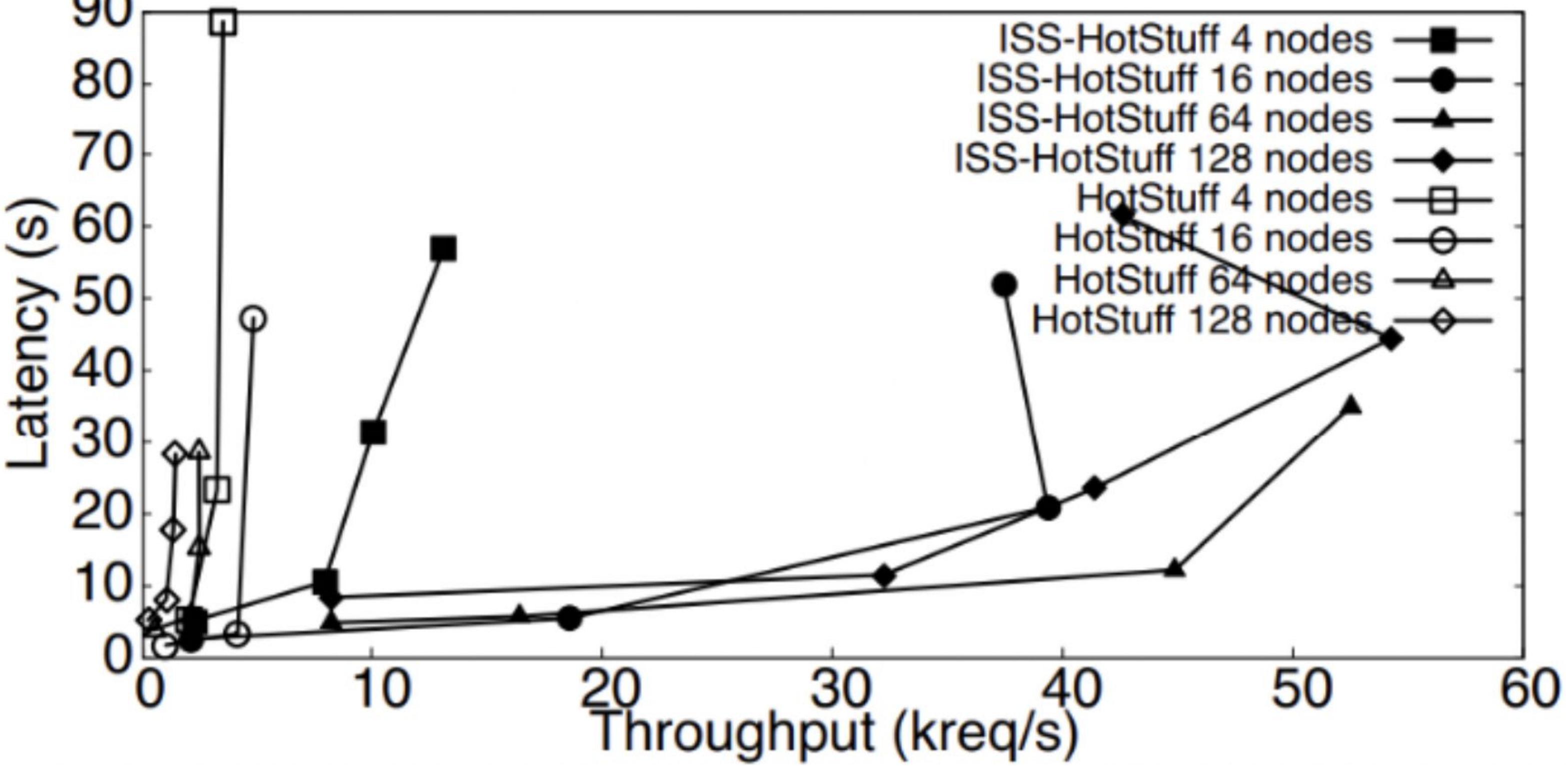
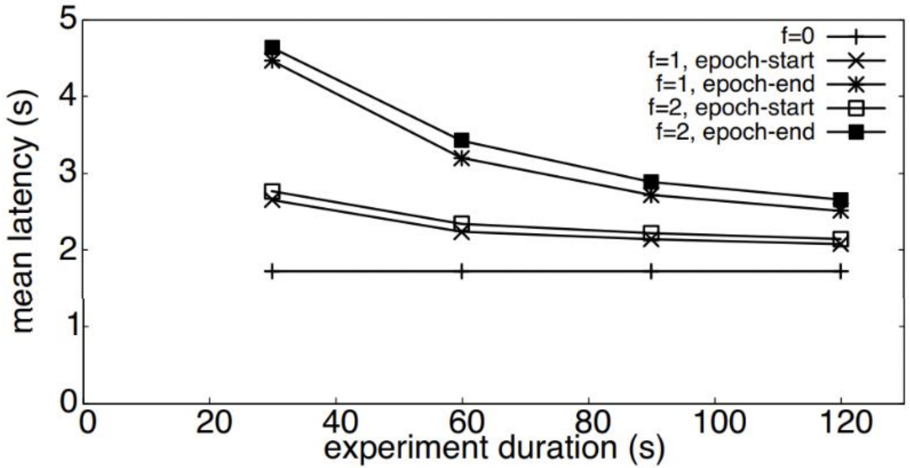
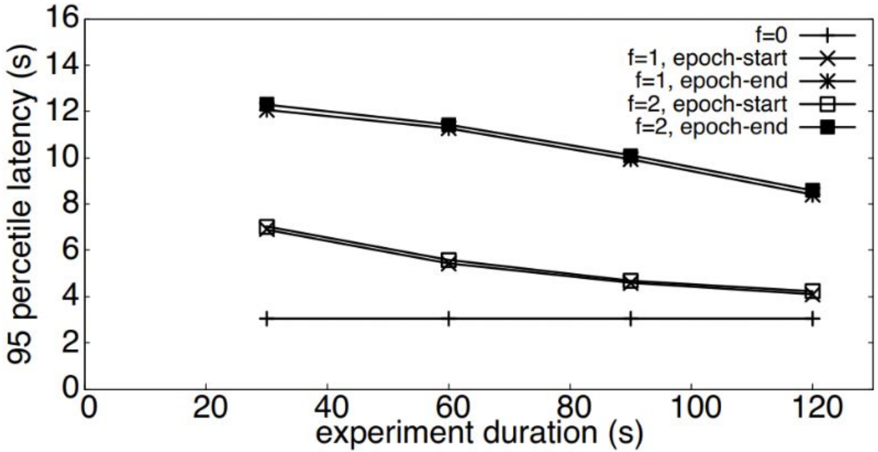


Figure 6 from Stathakopoulou et al.'s paper.

Results: Crash Faults

- The f represents number of crash faults
- Latency converges to the level of fault-free execution due to removing faulty leaders





References

Our paper extended: https://arxiv.org/pdf/2203.05681.pdf

Our paper: https://vukolic.com/eurosys22-final269.pdf

Video:
https://www.youtube.com/watch?v=zhu4b88wLKE&ab_channel=ProtocolLabs

Appendix

Sequenced Broadcast: Properties (Math Version)

Integrity: If a correct node sb-delivers (sn, m) with $m \neq \perp$ and σ is correct then σ sb-cast (sn, m).

Agreement: If two correct nodes sb-deliver, respectively, (sn, m) and (sn, m'), then $m = m'$.

Termination: If p is correct, then p eventually sb-delivers a message for every sequence number in S, i.e., $\forall sn \in S : \exists m \in M \cup \{\perp\}$ such that p sb-delivers (sn, m).

Eventual Progress: If some correct node sb-delivers $(sn, \perp)$ for some $sn \in S$, then some correct node p suspected σ after sb is initialized at p.

SMR: Properties (Math Version)

Nodes assign a unique sequence number such that these properties hold:

- **Integrity**: If a correct node delivers (sn, r), where $r.id.c$ is a correct client's identity, then client c broadcast r.
- **Agreement**: If two correct nodes deliver, respectively, (sn, r) and (sn, r'), then $r = r'$.
- **Totality**: If a correct node delivers request (sn, r), then every correct node eventually delivers (sn, r).
- **Liveness**: If a correct client broadcasts request r, then some correct node eventually delivers (sn, r).

r = (o, id) is a client request, where o is the payload and id is a unique identifier

On RCC Duplication

multiple leaders. We do not compare, however, to other multi-leader protocols that do not prevent request duplication (e.g., Hashgraph [23], Red Belly [13], RCC [20], OMADA [16], BFT-Mencius [27]). The codebase of these protocols is un-

Helpful Links

[TOB]

[Types of broadcast]

[Safety/Liveness in SMR]

[RCC Video]