State Machine Replication Scalability Made Simple

Anonymous Author(s)
Submission Id: #269

Abstract

Consensus, state-machine replication (SMR) and total order broadcast (TOB) protocols are notorious for being poorly scalable with the number of participating nodes. Despite the recent race in reducing overall message complexity of leader-driven SMR/TOB protocols, scalability remains poor and the throughput is typically inversely proportional to the number of nodes. We present ISS, a generic construction to turn leader-driven protocols into scalable multi-leader ones. Towards our scalable SMR construction we use a novel primitive called Sequenced (Total Order) Broadcast (SB) which we wrap around PBFT, HotStuff and Raft leader-driven protocols to make them scale. Our construction is general enough to accommodate most leader-driven ordering protocols (BFT or CFT) and make them scale. Our implementation of ISS reaches a throughput of 58 kreq/s at a scale of 128 nodes using PBFT, 57 kreq/s using HotStuff and 56 kreq/s using Raft.

1 Introduction

A lot of research effort has been recently dedicated to scaling consensus and total-order broadcast (TOB) protocols, fundamental primitives in distributed computing. By scaling we mean maintaining high throughput and low latency despite a growing number of nodes (replicas) n. Driven by the needs of blockchain systems, particular focus lies on Byzantine fault-tolerant (BFT) protocols in the eventually synchronous (deterministic protocols) or fully asynchronous (randomized protocols) model.

In this model, the classical Dolev/Reischuk (DR) [11] lower bound requires $\Omega(n^2)$ worst case message complexity, which was a focal complexity metric of many subsequent protocols including recent HotStuff [29]. However, we claim message complexity to be a rather poor scalability metric, demonstrated by the fact that HotStuff and other leader-driven protocols scale inversely proportionally to the number of nodes, despite some of them matching the DR lower bound. This is because in leader-driven protocols, the leader has at least $O(n)$ bits to send, even in the common case, yielding n^{-1} throughput scalability.

A recent effort to overcome the single leader bottleneck by allowing multiple parallel leaders (Mir-BFT [28]) in the classical PBFT protocol [8] demonstrates high scalability in practice. Despite certain advantages of PBFT, e.g., it is highly parallelizable and designed to not require signatures on protocol messages, among the many (existing and future) TOB solutions, there is no one that fits all use cases. HotStuff [29] is the first protocol with linear message complexity both in common case and in leader replacement, making it suitable for highly asynchronous or faulty networks. On the other hand, other protocols, e.g. Aliph/Chain [16], have optimal throughput when failures are not expected to occur often. Finally, crash fault-tolerant (CFT) protocols such as Raft [26] and Paxos [22] tolerate a larger number of (benign) failures than BFT protocols for the same number of nodes.

Our work takes the Mir-BFT effort one step further, introducing ISS (Insanely Scalable SMR), the first modular framework to make leader-driven TOB protocols scale. Notably, and unlike previous efforts [28][4], ISS achieves scalability without requiring a primary node to periodically decide on the protocol configuration. ISS achieves this by introducing a novel abstraction, Sequenced Broadcast (SB), which requires each instance of an ordering protocol to *terminate* after delivering a finite number of messages. This in turn allows nodes in ISS to decide on the configuration independently and deterministically, without requiring additional communication and without relying on a single primary node.

ISS implements total order by multiplexing multiple instances of SB which operate concurrently on a *partition* of the domain of client requests. We carefully select the partition to maintain safety and liveness, as well as to prevent redundant data duplication, which has been shown to be detrimental to performance [28]. This is qualitatively better than related modular efforts [17, 25] which do not provide careful partitioning and load balancing and hence cannot achieve the same scalability.

ISS implements TOB by maintaining a *contiguous* log of (batches of) client requests at each node. Each position in the log corresponds to a unique sequence number and ISS agrees on the assignment of a unique request batch to each sequence number. Our goal is to introduce as much parallelism as possible in assigning batches to sequence numbers while avoiding *request duplication*, i.e., assigning the same request to more than one sequence number. To this end, ISS subdivides the log into non-overlapping *segments*. Each segment, representing a subset of the log's sequence numbers, corresponds to an independent instance of SB that has its own leader and executes concurrently with other SB instances.

To prevent the leaders of two different segments from concurrently proposing the same request, and thus, wasting resources, while also preventing malicious leaders from censoring (i.e., not proposing) certain requests, we adopt and generalize the rotating bucketization of the request space introduced by MirBFT [28]. ISS assigns a different *bucket*—subset of pending client requests—to each segment. No bucket is assigned to more than one segment at a time and each request maps (through a hash function) to exactly one bucket. ISS

periodically changes the bucket assignment, such that each bucket is guaranteed to eventually be assigned to a segment with a correct leader.

To maintain the invariant of one bucket being assigned to one segment, all buckets need to be re-assigned at the same time. ISS therefore uses finite segments that it groups into *epochs*. An epoch is a union of multiple segments that forms a contiguous sub-sequence of the log. After all log positions within an epoch have been assigned request batches, and thus, no requests are "in-flight", ISS advances to the next epoch, meaning that it starts processing a new set of segments forming the next portion of the log.

We implement and deploy ISS on a wide area network (WAN) spanning 16 different locations spread around the world, demonstrating ISS' performance using two different BFT protocols (PBFT[8] and Hotstuff[29]) and one CFT protocol (Raft[26]). On 128 nodes we see a 37x improvement in throughput for PBFT, 56x for HotStuff and 55x for Raft.

The rest of this paper is organized as follows. Section 2 presents the theoretical foundation of our work. It models the systems we study in this work (Section 2.1), introduces the SB abstraction (Section 2.2) and describes how we multiplex SB instances with ISS (Sections 2.3 and 2.4). Sections 3 and 4 respectively describe the details of ISS and its implementation, including three different leader-driven protocols. In Section 5 we prove that multiplexing SB instances with ISS implements SMR and in Section 6 we evaluate the performance of ISS. In Section 7 we discuss related work. We conclude in Section 8.

2 Theoretical Foundations

2.1 System Model

We assume a set $\mathcal{N}$ of *node* processes with $|\mathcal{N}| = n$. At most f of the nodes in $\mathcal{N}$ can fail. We further assume a set C of *client* processes of arbitrary size, any of which can be faulty. Each process is identified by their public key, provided a public key infrastructure. Unless mentioned otherwise, we assume Byzantine, i.e., arbitrary, faults. Therefore, we require $n \geq 3f + 1$. We further assume that nodes in $\mathcal{N}$ are computationally bounded and cannot subvert standard cryptographic primitives.

Processes communicate through authenticated point-to-point channels. We assume an eventually synchronous network [12] such that the communication between any pair of correct processes is asynchronous before an unknown time GST, when the communication becomes synchronous.

Nodes in $\mathcal{N}$ implement a state machine replication (SMR) service to clients in C. To broadcast request $r \in \{0,1\}^*$, a client invokes SMR-BCAST(r).

Nodes assign a unique sequence number sn to r and eventually output SMR-DELIVER(sn, r) such that the following properties hold:

SMR1 Authenticity: If a correct node delivers (sn, r) then some client in C broadcast r.

SMR2 Agreement: If two correct nodes deliver, respectively, (sn, r) and (sn, r'), then $r = r'$.

SMR3 Totality: If a correct node delivers request (sn, r), then every correct node eventually delivers (sn, r).

SMR4 Liveness: If a correct client broadcasts request r, then some correct node eventually delivers (sn, r).

2.2 Sequenced Broadcast (SB)

Sequenced Broadcast (SB) is a variant of Byzantine total order broadcast [6] with explicit sequence numbers and an explicit set of allowed messages. Let M be a set of messages and $S \subseteq \mathbb{N}$ a set of sequence numbers. Only one designated sender node $\sigma \in \mathcal{N}$ can broadcast messages by invoking SB-BCAST(sn, m) with $(sn, m) \in S \times M$.

SB-DELIVER(sn, m) is triggered at a correct node p when p delivers message m with sequence number sn. In certain cases, p is allowed to deliver a special *nil* value $m = \perp \notin M$. Eventually, however, (intuitively, when the system starts behaving synchronously – after GST), p will be guaranteed to deliver non-nil messages $m \neq \perp$.

An instance of $SB(\sigma, M, S)$ has the following properties:

SB1 Integrity: If a correct node delivers (sn, m), then $sn \in S$ and $m \in M \cup \{\perp\}$. Moreover, if $m \neq \perp$ and σ is correct then σ broadcast (sn, m).

SB2 Agreement: If two correct nodes deliver, respectively, (sn, m) and (sn, m'), then $m = m'$.

SB3 Termination: If p is correct, then p eventually delivers a message for every sequence number in S, i.e., $\forall sn \in S : \exists m \in M \cup \{\perp\}$ such that p delivers (sn, m).

SB4 Eventual Progress: Eventually, the following will hold: If σ is correct and σ broadcasts (sn, m) and σ does not broadcast (sn, m') with $m \neq m'$ [1], then some correct node delivers (sn', m), for some $sn' \in S$.

The key difference in comparison to TOB is that SB is invoked for a well-defined set of sequence numbers S and messages M, such that each sequence number is eventually mapped to a message in $M \cup \{\perp\}$. The $\perp$ value guarantees that, upon invocation of $SB(\sigma, M, S)$, if the dedicated sender σ does not broadcast anything or in case σ is not correct, SB3 (Termination) can still be satisfied.

Sequenced Broadcast can be seen as a restricted form of consensus, and thus can easily be obtained from existing TOB and consensus protocols. To implement SB, a leader-driven consensus protocol (like PBFT [8] or HotStuff [29]) only needs to respect the following: For each sequence number in S and for a set of messages M: (1) Let the initial leader be the dedicated sender σ; (2) Enforce that no other leader except for σ can propose a "fresh" value different from $\perp$; (3) Enforce that correct nodes either deliver a message from M or $\perp$; (4)

[1] Progress is only required if σ does not broadcast conflicting messages.

In an infinite execution, σ becomes leader infinitely many times (e.g., by a round-robin leader assignment) or forever.

2.3 Multiplexing Instances of SB with ISS

ISS multiplexes instances of SB to implement SMR. Each node maintains a log of ordered messages. Each position in the log corresponds to a *sequence number* signifying the offset from the start of the log. The log is partitioned in subsets of sequence numbers called *segments*. Each segment corresponds to one instance of SB. ISS nodes obtain requests from clients and, after mapping them to a log position using an instance of SB, deliver them together with the assigned sequence number.

ISS proceeds in *epochs* identified by monotonically increasing integer *epoch numbers*. Each epoch e is associated with a set of segments. The union of those segments forms a set $Sn(e)$ of consecutive sequence numbers. Epoch 0 (the first epoch) starts with sequence number 0. The mapping of sequence numbers to epochs is a function known to all nodes with the only requirements that it is monotonically increasing and that there are no gaps between epochs. More formally, $max(Sn(e)) + 1 = min(Sn(e+1))$. Epoch length can be arbitrary, as long as it is finite. For simplicity, we use a fixed, constant epoch length.

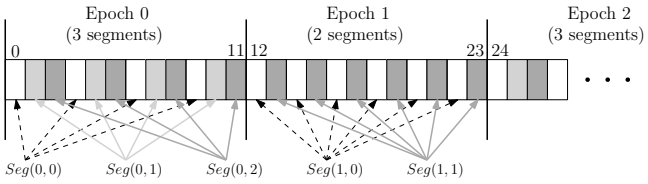

Figure 1. Log partitioned in epochs and segments. In this particular example, each epoch is 12 sequence numbers long. The first epoch has 3 segments while the second epoch only has 2. I.e., $|Leaders(0)| = 3$, $|Leaders(1)| = 2$, $max(Sn(1)) = 23$ and $max(Seg(0,1)) = 10$.

Epochs are processed sequentially, i.e., ISS first agrees on the assignment of request batches to all sequence numbers in $Sn(e)$ before starting to agree on the assignment of request batches to sequence numbers in $Sn(e+1)$.

Within an epoch, however, ISS processes segments in parallel. Multiple leaders, selected according to a *leader selection policy*, concurrently propose batches of requests for different sequence numbers in $Sn(e)$. To this end, ISS assigns a different *leader* node to each segment in epoch e. The numbers of leaders and segments in each epoch always match. We denote by $Seg(e,i)$ the subset of $Sn(e)$ for which node i is the leader. This means that node i is responsible for proposing request batches to sequence numbers in $Seg(e,i)$. No node other than i can propose batches for sequence numbers in $Seg(e,i)$. Let $Leaders(e)$ be nodes that are leaders in epoch e. We associate sequence numbers with segments in a round-robin way[2], namely, for $0 \leq i < |Leaders(e)|$,

$$Seg(e,i) \subseteq Sn(e) = \{sn \in Sn(e) | sn \equiv i \bmod |Leaders(e)|\}$$

An example with $|Leaders(0)| = 3$ and $|Leaders(1)| = 2$ is illustrated in Figure 1.

In order not to waste resources on duplicate requests, we require that a request cannot be part of two batches assigned to two different sequence numbers. We enforce this at three levels: (1) within a segment, (2) across segments in the same epoch, and (3) across epochs.

Within a segment, we rely on the fact that the correct leader will propose (and a correct node will accept) only batches with disjoint sets of requests for each sequence number within a segment.

Across segments, we partition the set of all possible requests into *buckets* using a hash function and enforce that only requests from different buckets can be used for different segments within an epoch. We denote by $\mathcal{B}$ the set of all possible buckets. To each segment, we assign a subset of $\mathcal{B}$, such that each bucket is assigned to exactly one segment in each epoch. We denote by $Buckets(e,i) \subseteq \mathcal{B}$ the set of buckets assigned to leader i in epoch e.[3]

Across epochs, nodes synchronize their logs on committed requests (see also Section 3.3) such that a committed request will not be committed again.

In summary, a segment of epoch e with leader i is defined by the tuple $(e, i, Seg(e,i), Buckets(e,i))$.

For a set of buckets $B \subseteq \mathcal{B}$, we denote with $batches(B)$ the set of all possible batches consisting of valid (we define request validity precisely later in Section 3.6) requests that map to some bucket in B. For each segment $(e, i, Seg(e,i), Buckets(e,i))$, we use an instance $SB(i, batches(Buckets(e,i)), Seg(e,i))$ of Sequenced Broadcast. We say that leader i *proposes* a batch $b \in batches(Buckets(e,i))$ for sequence number $sn \in Seg(e,i)$ if i executes SB-BCAST(sn, b) at the corresponding instance of SB. A batch b *commits* with sequence number sn (and is added to the log at the corresponding position) at node n when the corresponding instance of SB triggers SB-DELIVER(sn, b) at node n.

During epoch e, all nodes that are leaders in e simultaneously propose batches for sequence numbers in their corresponding segments. ISS multiplexes all segments into the single common log as shown in Figure 1. Each node thus executes $|Leaders(e)|$ SB instances simultaneously, while being a leader for at most one of them.

Epoch e ends and epoch $(e+1)$ starts when all sequence numbers in $Sn(e)$ have been committed. Nodes keep the old

[2]Any assignment of sequence numbers to segments is possible and leads to a correct algorithm. We choose the round-robin way because it is the least likely to create "gaps" in the log and thus minimizes request latency.

[3]We sloppily say that we assign a bucket to a leader i when assigning a bucket to a segment for which i is the leader.

instances active until all corresponding sequence numbers become part of a stable checkpoint. This is necessary for ensuring totality (even for slow nodes which might not have actively taken part in the agreement).

2.4 Assigning Buckets to Segments

ISS partitions the request hash space into buckets which it assigns to leaders/segments and rotates the bucket assignment across epochs. At any point in time, a leader can assign sequence numbers only to requests from its assigned buckets. This approach has first been used in Mir-BFT [28] to counter request duplication and censoring attacks.

The assignment of buckets to leaders is fixed leaderset an epoch and changes at each epoch transition. To ensure liveness, each bucket must repeatedly be assigned to a correct leader. To this end, ISS rotates the bucket assignment on every epoch transition as follows (illustrated in Figure 2).

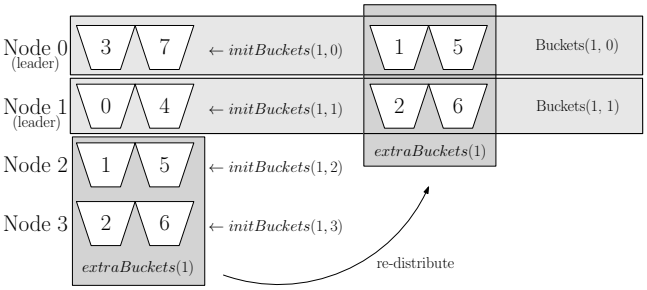

Figure 2. Example assignment of 8 buckets to 2 leaders in a system with 4 nodes in epoch 1.

For epoch e, we start by assigning an initial set of buckets to *each node* (leader or not) in a round-robin way. Let $initBuckets(e, i) \subseteq \mathcal{B}$ be the set of buckets initially assigned to each node i, $0 \leq i < n$ in epoch e.

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

However, not all nodes belong to $Leaders(e)$. Let $extraBuckets(e)$ be the set of buckets initially assigned to non-leaders.

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

We must re-distribute those extra buckets to the leaders of epoch e. We do this also in a round robin way. Let $l(e, k)$, $0 \leq k < |Leaders(e)|$ be the k-th leader (in lexicographic order) in epoch e. The $Buckets(e, l(e, k))$ of the k-th leader in e are thus defined as follows.

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

An example bucket assignment is illustrated in Figure 2.

With this approach, all buckets are assigned to leaders and every node is eventually assigned every bucket at least through the initial bucket assignment. ISS ensures liveness as long as, in an infinite execution, it occurs infinitely many times that a correct node is leader for at least n consecutive epochs. This condition can be easily fulfilled, as we describe in Section 3.4.

3 ISS Algorithm details

In this section we present the ISS algorithm details. The main algorithm that produces a totally ordered log is described in Algorithm 1 and auxiliary functions and epoch initialization functions are presented in Algorithms 2 and 3, respectively.

As described in the previous sections, ISS proceeds in epochs, each epoch multiplexing multiple segments into a final totally ordered log. We start with epoch number 0 (line 7) and an empty log (line 10). All buckets are initially empty (line 14). Whenever a client submits a new request (line 19), ISS adds the request to the corresponding bucket (line 20).

Algorithm 1 Main ISS algorithm for node p

```
1:  Parameters:
2:      numBuckets
3:      batchTimeout
4:      epochLength
5:
6:  upon init() do
7:      currentEpoch ← 0
8:      segments ← ∅
9:      for sn ∈ ℕ do
10:         log[sn] ← none
11:         proposed[sn] ← none
12:     end for
13:     for 0 ≤ b < numBuckets ∈ do
14:         buckets[b] ← empty bucket
15:     end for
16:     initEpoch(0)
17:     runEpoch(0)
18:
19: upon receiving request do
20:     buckets[hash(request)].add(request)
21:
22: function runEpoch(e) :
23:     for sn ∈ seqNrs(e) with segOf(sn).leader = p do
24:         propose(sn)
25:     end for
26:
27: upon SB.SB-DELIVER((sn, batch)) do
28:     log[sn] ← batch
29:     if batch ≠ ⊥ then
30:         for req ∈ batch do
31:             buckets[hash(req)].remove(req)
32:         end for
33:     else if proposed[sn] ≠ none then
34:         resurrectRequests(proposed[sn])
35:     end if
36:
37: upon ∀sn ∈ seqNrs(currentEpoch) : log[sn] ≠ none do
38:     initEpoch(currentEpoch + 1)
39:     runEpoch(currentEpoch + 1)
40:
```

3.1 Epoch initialization

At the start of each epoch e, ISS: (1) Calculates $Leaders(e)$, the set of nodes that will act as leaders in e, based on the

Algorithm 2 Auxiliary functions

```
41: function segOf(sn) :
42:     return seg ∈ segments : sn ∈ seg.seqNrs
43:
44: function seqNrs(e) :
45:     return {i ∈ ℕ|
46:             e · epochLength ≤ i < e · (epochLength + 1)}
47:
48: function propose(sn) :
49:     wait until      segments[sn].batchReady()
50:                  or batchTimeout
51:
52:     batch ← segOf(sn).cutBatch()
53:     segOf(sn).SB.SB-BCAST(sn, batch)
54:     proposed[sn] ← batch
55:
56: function resurrectRequests(batch) :
57:     for req ∈ batch do
58:         buckets[hash(req)].add(req)
59:     end for
60:
```

used leader selection policy (line 64). (2) For each node l in $Leaders(e)$, creates a new segment with leader l (lines 66 and 67). (3) Assigns all sequence numbers $Sn(e)$ of epoch e in a round-robin way to all created segments (line 69). (4) Assigns buckets to the created segments as described in Section 2.4 (line 70). (5) Creates an instance of SB for each created segment (line 72).

3.2 Ordering request batches

ISS orders requests in batches, a common technique which allows requests to be handled in parallel, amortizes the processing cost of protocol messages, and, therefore, improves throughput. During an epoch, every node l that is the leader of a segment s proposes request batches for sequence numbers assigned to s (line 24). l does so by broadcasting the batches using the instance of SB associated with s. Every node then inserts the delivered (sequence number, batch) pairs at the corresponding positions in its copy of the log (line 28). We say the node *commits* the batch with the corresponding sequence number since, once inserted to the log, the assignment of a sequence number to a batch is final.

Proposing Batches. To propose a request batch for sequence number sn, l first constructs the batch using requests in the buckets assigned to s. To implement efficient request batching while preserving low latency, l waits until at least one of the following conditions is fulfilled: (1) the buckets assigned to s contain enough requests (more than a predefined $batchSize$) (line 50), or (2) a predefined time elapses since the last proposal (line 51). Under low load, this condition sets an upper bound on the latency of requests waiting to be proposed, even if the batch is filling slowly.

l then constructs a $batch$ using up to $batchSize$ requests from the buckets assigned to s (line 52), removes those requests from their buckets, and proposes the batch by invoking SB-BCAST$(sn, batch)$ on the SB instance associated with s (line 53).

Every leader also keeps track of the values it proposed for each sequence number (line 54). This, as we explain later, is important in the case of asynchrony.

Assembling the Final Log. Whenever any instance of SB (belonging to any segment) delivers a value associated with a sequence number (line 27) at some node n, n inserts the delivered value at position sn of the log (line 28). If a request batch $\neq \perp$ has been delivered (line 29), n removes the contained requests from their corresponding buckets (line 31) to avoid proposing them again in a later epoch.

Note that buckets are local data structures at each node, and thus each node manages its buckets locally. Nodes add to their local buckets all requests they obtain from clients, but only propose batches constructed from buckets assigned to their segments. It is thus normal that a node n delivers batches containing requests mapping to other buckets than those n uses for proposing. Therefore, to avoid request duplication across epochs, each node must remove all delivered batches from its local buckets.

If the special value $\perp$ has been delivered by SB and, at the same time, n itself had been the leader proposing a batch for sn (line 33), n "resurrects" all requests in the batch it had proposed (line 34) by returning them to their corresponding buckets (line 58). This scenario can appear in case of asynchrony / partitions, where a correct leader is suspected as faulty after having proposed a batch. Such a leader must return the unsuccessfully proposed requests in their buckets and, if batches with those requests are not committed by other leaders in the meantime, retry proposing them in a later epoch where it is again leader of a segment with those buckets.

A node considers the ordering of a request finished when it is part of a committed batch with an assigned sequence number sn and the log contains an entry for each sequence number $sn' \leq sn$.

Each request is delivered with a unique sequence number sn_r denoting the total order of the request. sn_r is derived from the sequence number of the batch in which the request is delivered and the position of the request in the batch. Let $\mathcal{S}_{sn}$ be the number of requests in a batch delivered with sequence number sn and let r be the k^{th} request in this batch. For each such request r, ISS outputs SMR-DELIVER(r, sn_r) where:

$$sn_r = k + \sum_{i=0}^{sn-1} \mathcal{S}_i \quad (1)$$

3.3 Advancing Epochs

ISS advances from epoch e to epoch $e+1$ when the log contains an entry for each sequence number in $Sn(e)$ (line 37). This will eventually happen for each epoch at each correct node due to SB *Termination*. Only then the node starts processing messages related to epoch $e+1$ and starts proposing batches for sequence numbers in $Sn(e+1)$ (lines 38 and 39).

Requiring a node to have committed all batches in epoch e before proposing batches for $e+1$ prevents request duplication

across epochs. When a node transitions from e to $e+1$, no requests are "in flight"—each request has either been already committed in e or has not yet been proposed in $e+1$.

3.4 Selecting Epoch Leaders

In order to guarantee that each request r submitted by a correct client is ordered (liveness), we must ensure that, eventually, there will be a segment in which r is committed. As implied by the specification of SB, this can only be guaranteed if a correct leader proposes a batch containing r. Even then, SB only provides eventual progress, potentially requiring a batch with r to be proposed multiple times by a correct leader before it is committed. The choice of epoch leaders is thus crucial.

ISS selects leaders according to a *leader selection policy*, a function known to all nodes that, at the end of each epoch e, determines the set of leaders for epoch $(e+1)$.

In order to guarantee liveness of the system, the leader selection policy must ensure, for each bucket b, that, in an infinite execution, b will be assigned infinitely many times to a segment with a correct leader. Different leader selection policies pose different trade-offs with respect to performance. For simplicity, we adopt the policy of BFT-Mencius [25] because it trivially ensures liveness by keeping always enough correct nods in the leaderset.

3.5 Checkpointing and state transfer

ISS implements a simple checkpointing protocol. Every node i, in each epoch e, when the log contains an entry for each sequence number in $Sn(e)$, broadcasts a signed message $\langle \text{CHECKPOINT}, max(Sn(e)), D(e), \sigma_i \rangle$, where $D(e)$ is the Merkle tree root of the digests of all the batches in the log with sequence numbers in $Sn(e)$. Upon acquiring a strong quorum of $(2f+1)$ CHECKPOINT messages, node i creates a stable checkpoint $\langle \text{STABLE-CHECKPOINT}, max(Sn(e)), \pi(e) \rangle$, where $\pi(e)$ is the set of $2f+1$ signatures. At this point i can garbage collect all segments of epoch e.

When a node i has fallen behind, for example when the node starts receiving messages for a future epoch, the node performs a state transfer, i.e., i fetches the missing log entries along with their corresponding stable checkpoint which proves the integrity of the data.

ISS checkpointing is orthogonal to any checkpointing and state transfer mechanism pertaining the SB implementation because SB instances must terminate independently.

3.6 Request handling

We denote with $\langle \text{REQUEST}, req_{id}, p, \sigma_{pk} \rangle$ a client request message, where p the payload, req_{id} the request identifier and σ_{pk} a cryptographic signature with client's public key pk. The identifier req_{id} is a tuple $(client_{id}, req_{sn})$, where req_{sn} is a monotonically increasing *per client* request number and $client_{id}$ a unique client identifier. Our implementation associates an integer $client_{id}$ to the client's public key pk. The signature is calculated over req_{id} and payload to guarantee integrity and authenticity. Similarly to Mir-BFT [28], clients can submit multiple requests in parallel within a client watermark window which ISS advances at the end of each epoch.

Each correct node, upon receiving a valid request, adds the request, based on its identifier, to the corresponding bucket. A request is considered *valid* if: (1) it has a valid signature (2) the public key corresponds to a client in the client set C of the system, and (3) is within the client watermarks. Buckets are idempotent, i.e., each correct node adds a request to the corresponding bucket exactly once. Moreover, buckets are implemented as FIFO queues to guarantee liveness with the oldest request always being proposed first.

Requests are uniformly distributed to buckets using modulo as a hash function. With $|\mathcal{B}|$ denoting the total number of buckets and $||$ denoting concatenation, each request r maps to a bucket b: $b = r.client_{id} || r.req_{sn} \bmod |\mathcal{B}|$

Importantly, we exclude the payload of the request from the bucket mapping function to prevent malicious clients from biasing the uniform distribution. In a permissioned system the client cannot assume different identities and may only bias the outcome of the hash function by choosing the request sequence number. However, we limit the available sequence numbers for each client, and therefore their ability to bias the request distribution to buckets, with the client watermarking mechanism.

Algorithm 3 Epoch initialization

```
function initEpoch(e) :
    currentEpoch ← e

    leaders ← leader_selection_policy(e)
    for 0 ≤ l < |leaders| ∈ do
        seg ← new segment
        seg.leader ← leaders[l]
        seg.seqNrs ← {sn ∈ seqNrs(e) |
                      sn ≡ l mod |leaders|}
        seg.buckets ← Buckets(e, leaders[l])
        seg.SB ← Initialize new instance of
            SB(leaders[l], batches(seg.buckets), seg.seqNrs, equals)

        segments ← segments ∪ {seg}
    end for

function Buckets(e, i) :
    for n ∈ N do
        initBuckets[n] ← {b < numBuckets |
                          (e + b) ≡ n mod |N|}
    end for
    leaders ← epLeaders_POLICY(e)
    extraBuckets ← {b : ∃n |
                    n ∉ leaders ∧ b ∈ initBuckets[n]}
    return initBuckets[i] ∪ {b ∈ extraBuckets |
                             (e + b) ≡ (index of i in leaders) mod |leaders|}
```

4 ISS Implementation

We implement ISS in Go, using gRPC for communication with TLS on nodes with two network interfaces; one for client-node and one for node-to-node communication.

Our implementation is highly concurrent: multiple threads are handling incoming client requests, verifying request signatures, sending / receiving messages, and executing various sub-protocols such as checkpointing and fetching missing protocol state . Each SB instance also executes in its own thread. A separate thread orchestrates all of the above.

In the rest of this section, we discuss our implementation of the ordering subprotocols (4.1), the interaction between ISS and its clients (4.2), and crucial technical aspects for achieving robustness and high performance (4.3).

4.1 SB implementation

In this section we discuss (1) how we implement SB with different leader-driven consensus protocols and (2) adaptations in the leader-driven protocols critical for ISS performance.

All protocol implementations adhere to the following common design principles:

1.The first leader is the segment leader and all other nodes of the system participate as followers.

2. After a leader-change, any new leader (including the segment leader if it becomes leader again), can only propose $\perp$ values for any sequence number not initially proposed by the segment leader.[4]

3. A follower accepts a proposal only if (a) all requests in the batch are valid according to Section 3.6, (b) all requests belong to the buckets of the segment, and (c) either the segment leader broadcast the proposal, or the proposal is $\perp$.

Our implementation currently supports two Byzantine fault-tolerant total order broadcast protocols (PBFT and HotStuff) and a crash fault-tolerant one (Raft).

4.1.1 PBFT.

We follow the PBFT protocol as described by Castro and Liskov [8], with a few adaptations.

Our implementation need not deal with timeouts at the granularity of single requests, as PBFT does. To prevent censoring attacks (and thus ensure liveness), a PBFT replica initiates a view change if any request has not been committed for too long. Since ISS prevents censoring attacks by bucket re-assignment, it is sufficient for us to make sure to commit *some batch* before a timeout fires and reset this timer when committing any batch. In absence of incoming requests, the primary periodically proposes an empty batch to prevent a view change. Moreover, we implement view-change with signatures according to Castro and Liskov [7] for simplicity.

4.1.2 HotStuff.

We implement chained HotStuff according to Yin *et. al* [29] with BLS [5] threshold signatures using DEDIS library for Go [1].

In our implementation each batch corresponds to a HotStuff command, and each segment sequence number corresponds to a HotStuff view. Each segment is implemented as a new HotStuff instance checkpoint to a root certificate QC_0. To ensure that all sequence numbers can be delivered, i.e., to ensure that we can always "flush" the pipeline of chained HotStuff, we extend the segment with 3 dummy sequence numbers corresponding to dummy empty batches which are not added to the ISS log. Figure 3 demonstrates an example of a segment with 3 sequence numbers.

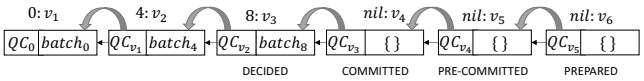

Figure 3. Chained HotStuff execution for a segment with 3 sequence numbers $\{0, 4, 8\}$. When view v_6 is prepared, v_3 is decided and $batch_8$ can be added to the log.

4.1.3 Raft.

Briefly, in Raft nodes set a random timer within a configurable range, which they reset every time they receive a message from the elected leader. If the timer fires, the node advances to a new term (similar to PBFT view) and enters an election phase as a candidate leader. An elected leader periodically sends append-entry requests for new values, possibly also empty, as a heartbeat. The leader collects responses according to which it might resend to the followers any value they declare as missing.

We implement Raft according to [26] with minor adaptations. We fix the first leader to be the leader of the segment skipping the election phase. The leader periodically sends append-entry requests containing batches until the end of the segment. The leader continues to send empty append-entry requests until the end of the epoch to guarantee that enough nodes have added all the batches of the segment to their log.

4.2 Interaction with Clients

Clients submit requests to ISS by sending request messages to ISS nodes. To guarantee Liveness (Section 2.1, SMR4) a client must ensure that at least one correct leader eventually receives and broadcasts the request. A trivial solution is to send the request to all nodes. However, ISS implements an optimistic a *leader detection* mechanism to (1) help the clients find the correct leader for each request faster and (2) better load balance request processing among the nodes.

At any point in time, the bucket a particular request belongs to is assigned to a single segment with a unique leader. Therefore, the client only needs to send its request to the node currently serving as a leader for the corresponding bucket.

When a node outputs delivers a request r, as described in Section 3.2, it sends a response message to the client that submitted r. When the client obtains a quorum of responses, it considers the request delivered.

ISS keeps the clients updated about the assignment of buckets to leaders. At each epoch transition, all ISS nodes send a message with the assignment for the next epoch to all clients. A client accepts such a message once it receives it from a quorum of ISS nodes. The client submits all future requests

[4] Enforcing $\perp$ values is necessary for SB to be implementable. Otherwise both SB *Integrity* and SB *Termination* cannot be satisfied at the same time.

to the appropriate node. Moreover, it resubmits all requests submitted in the past that have not yet been delivered. This guarantees that all correct nodes will eventually receive the request, which ensures liveness.

To make sure that, in most cases, a leader already has a request when it is that leader's turn to propose it, the client sends its request to two additional nodes that it projects to be assigned the corresponding bucket in the next two epochs.

4.3 Important Technical Aspects

We now describe several aspects of our implementation that are fundamental for ensuring ISS' robust performance.

4.3.1 Rate-limiting proposals in PBFT. PBFT's ability send proposals in parallel is instrumental for achieving high throughput. However, as soon as a load spike or a temporary glitch in network connectivity occurs (as it regularly does on the used cloud platform), the leader can end up trying to send too many batches in parallel. Due to limited aggregate bandwidth, *all* those batches will take longer to transfer, triggering view change timeouts at the replicas.

We address this issue by setting a hard limit on the rate of sending batches "on the wire", allowing (the most part of) a batch to be transmitted before the transfer of the next batch starts. This measure limits peak throughput but is effective at protecting against unnecessary view changes.

4.3.2 Concurrency Handling. A naive approach to handling requests, where each client connection is served by a thread that, in a loop, receives a request, locks the corresponding bucket, adds the request, and unlocks the bucket, is detrimental to performance. We attribute this to cache contention on the bucket locks.

Still, access to a bucket does have to be synchronized, as adding (on reception) and removing (on commit) requests must happen atomically. At the same time, an efficient lock-free implementation of a non-trivial data structure such as a bucket could be a research topic on its own.

We thus dedicate a constant, limited number of threads (as many as there are physical CPU cores) only to adding requests to buckets, such that each bucket is only accessed by one thread, removing most of the contention. The network-handling threads pass the received requests to the corresponding bucket-adding threads using a lock-free data structure optimized for this purpose (a Go channel in our case).

4.3.3 Deployment, Profiling, and Analysis. ISS comes with tools for automating the deployment of hundreds of experiments across hundreds of geo-distributed nodes on the cloud and analyzing their outputs. They profile (using `pprof`) the execution at each node, pinpointing lines of code that cause stalling or high CPU load. For example, the above-mentioned cache contention was pointed to by the profiler. They also plot various metrics over time, such as the size of proposed batches, commit rate, or CPU load. Automatic exploration of the multi-dimensional parameter space proved essential for understanding the inner workings of the system.

Using hundreds of cloud machines with hourly billing also incurs significant cost. Automatically commissioning cloud machines only for the time strictly necessary to run our experiments and releasing those resources as soon as possible most likely saved thousands of dollars.

5 Correctness

We prove that multiplexing SB instances with ISS implements an SMR service to a set of clients, as defined in Section 2.1.

SMR1 Authenticity: If a correct node delivers (sn, r) then some client in C broadcast r.

Proof. A correct node only delivers request r if it is inserted in the log as part of a committed batch b (see Section 3.2), such that SB-DELIVER(sn_b, b) is output by an $SB(\sigma, M, S)$ instance, where S is a segment of the log, σ the leader of the segment and M the set of batches in the buckets of the segment. SB1 (Integrity) guarantees that $m \in M \cup \perp$ and, since b is a batch of requests and therefore $b \neq \perp$, then $b \in M$.

A correct leader only invokes *sbcast* with valid requests. A validity condition is that every request of the batch has valid signature for a public key of a client in C (see Section 3.6). Therefore, batches in M only contain requests from some client in C and, thus, r is broadcast by some client in C. □

SMR2 Agreement: If two correct nodes deliver, respectively, (sn, r) and (sn, r'), then $r = r'$.

Proof. Let us assume by contradiction that there exist requests r_1, r_2 delivered with the same sequence number sn such that $r_1 \neq r_2$. We examine two mutually exclusive cases:
1. r, r' are in the same batch b committed with sn_b.
2. r is in batch b committed with sn_b and r' is in a different batch b' committed with $sn_{b'}$.

In case (1) let r_1 be the k_1^{th} request in b_1 and r_2 the k_2^{th} request in b_1, where $k_1 \neq k_2$ since they denote positions and each request has a different position in a batch. Then, from Equation (1), $sn = k_1 + \sum_{i=0}^{sn_b - 1} \mathcal{S}_i$ and also $sn = k_2 + \sum_{i=0}^{sn_b - 1} \mathcal{S}_i$, a contradiction.

In case (2), for r, r' to have the same sequence number, it follows that $sn_b = sn'_b$. Since b and b' are delivered with the same sequence number, they belong to the same segment S and, thus, also in the same set M for which and instance $SB(\sigma, M, S)$ was invoked. From SB2 it follows that $b = b'$, a contradiction. □

Lemma 5.1. *If a correct node instantiates $SB(\sigma, M, S)$ then every correct node eventually instantiates $SB(\sigma, M, S)$.*

Proof. We prove by induction on the epoch e that any two correct nodes i and j eventually instantiate SB with the same state σ, M, S. In epoch 0 all nodes instantiate SB with the same state as dictated by the deterministic initialization of ISS. We

assume that in epoch e nodes i and j instantiate SB with the same state. Then, by SB *Termination* both i and j finish epoch e advance to epoch $e+1$. Moreover, by ISS Algorithm both i and j advance from e to $e+1$ based on the same log state (by *SB-Agreement*). Thus, both i and j instantiate SB with the same state, i.e., sender σ, message set M and segments S. □

SMR3 Totality: If a correct node delivers request (sn, sr), then every correct node eventually delivers (sn, sr).

Proof. Let us assume that some correct node i delivers r with sequence number sn. There exist two mutually exclusive contradicting cases:
1. There is some correct node j that delivers $r' \neq r$ for sn.
2. There is some correct node j that does not deliver any request for sn.

Case (1) is trivially contradicted by SMR3 property (Agreement). Let us now examine case (2). From Equation (1) request r, delivered by i, uniquely corresponds to some batch b with sequence number sn_b in the log of i which i has committed with sn_b. Therefore, there exists an instance $SB(\sigma, M, S)$ which outputs SB-DELIVER(sn_b, b) at node i. By Lemma 5.1, node j also eventually invokes $SB(\sigma, M, S)$. Then property SB 3 (Termination) guarantees that, for each sequence number S, and therefore for also sn_l, j delivers a message $m \in M \cup \perp$. Moreover property SB 2 (Agreement) guarantees that $m = b$. It follows that j delivers r for sn, a contradiction. □

Lemma 5.2. *If a correct client broadcasts request r, then every correct node eventually receives r and puts it in the respective bucket.*

Proof. A correct node either receives the request directly from the client or in a proposed batch. Let b be the batch which contains r. If b is committed before r sends the request to all nodes, all correct nodes have received r as part of b and have added r in the respective bucket. Otherwise, after GST, every correct node will eventually receive r by the correct client periodically resubmitting r until it is delivered. □

Lemma 5.3. *If a correct client broadcasts request r, then some correct segment leader eventually invokes* SB-BCAST(sn, b) *with a batch b which includes r.*

Proof. Let us assume r maps to bucket B. By Lemma 5.2 all correct nodes eventually add r in B. Since B is implemented as a FIFO queue (Section 3.6) and the leader selection policy guarantees a correct node i becomes leader of B infinitely often (Section 3.4), then i eventually invokes SB-BCAST(sn, b) with a batch b that includes r. □

SMR4 Liveness: If a correct client broadcasts request r, then some correct node eventually delivers (sn, r).

Proof. By property SB 4 (Eventual Progress), if a correct node σ broadcasts request r in a batch b with some sequence number sn_b, such that σ does not broadcast any other batch with sn_b, then some correct node will be eventually output SB-DELIVER(sn_b, b), and, therefore, r will be delivered with some sequence number sn. ISS dictates that the node which broadcasts b is the leader of a segment and a correct leader only broadcasts one request per sequence number. By Lemma 5.3 such a correct node exists and, therefore, (sn, r) will be eventually delivered by some correct node. □

6 Evaluation

Our implementation is modular, allowing to easily switch between different protocols implementing SB. We use 3 well-known protocols for ordering requests: PBFT [8] (BFT), HotStuff [29] (BFT), and Raft [26] (CFT). We evaluate the impact ISS has on these protocols by comparing its performance to their respective original single-leader versions. We also compare ISS to Mir-BFT [28] which has also multiple leaders. We do not compare, however, to other multi-leader protocols which do not prevent request duplication (e.g., Hashgraph [21], Red Belly [10], RCC [18], OMADA [13], BFT-Mencius [25]). Mir-BFT evaluation demonstrates that the performance of this family of protocols deteriorates as the number of nodes increases in presence of duplicate requests. Moreover, the codebase of these protocols is unavailable or unmaintained. We first evaluate performance in absence of failures. Then, we study how ISS behaves when failures occur.

6.1 Experimental Setup

	PBFT	HotStuff	Raft
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

We perform our evaluation in a WAN which spans in 16 datacenters across Europe, America, Australia, and Asia on IBM cloud. All processes run on a dedicated virtual machine with 32 x 2.0 GHz VCPUs and 32GB RAM running Ubuntu Linux 20.04. All machines are equipped with two network interfaces, public and private, rate limited for repeatability to 1 Gbps; one for request submission and one node-to-node communication respectively. Clients submit requests with 500 byte payload, the average Bitcoin transaction size [2]. We use 16 client machines, also uniformly distributed across all datacenters, each running 16 clients in parallel which communicate independently with the nodes using TLS. We evaluate throughput, i.e., the number of requests the system delivers per second, and end-to-end latency, i.e., the latency from the moment a client submits a request until the client receives $f+1$ responses.

Table 1 summarizes the set of parameters of our evaluation.

6.2 Failure-Free Performance

Figure 4 shows the overall throughput scalability of PBFT, HotStuff, and Raft, with and without ISS, as well as that of Mir-BFT.

We evaluate the scalability of ISS with up to 128 nodes, uniformly distributed across all 16 datacenters. Mir-BFT is evaluated on the same set of datacenters on machines with the same specifications. For a meaningful, apples-to-apples, comparison, we disabled Mir-BFT optimizations (signature verification sharding and light total order broadcast). Such optimizations could be implemented on top of ISS yielding even better performance. However, this goes beyond the scope of this work. For all protocols we run experiments with increasing client request submission rate until the throughput is saturated.

In Figure 4 we report the highest measured throughput before saturation. We observe that ISS dramatically improves the performance of the single leader protocols as the number of nodes grows. 37x, 56x and 55x improvement for PBFT, HotStuff, and Raft respectively on 128 nodes. This improvement is due to overcoming the single leader bandwidth bottleneck. Moreover, as the number of nodes grows, ISS-PBFT outperforms Mir-BFT. While in theory, in a fault-free execution we would expect the two protocols to perform the same, we attribute this improvement to the more careful concurrency handling in ISS implementation.

ISS-PBFT maintains more than 58 kreq/s on 128 nodes. Its performance though, drops compared to smaller configurations. We attribute this to the increasing number of messages each node processes, which, with a fixed batch rate (Table 1), grow linearly with the number of nodes. We further observe that throughput increases for Raft and HotStuff ISS implementations with the number of nodes, approaching that of ISS-PBFT. HotStuff instance is latency-bound. This is because sending a proposal requires first assembling a quorum certificate which depends on the previous proposal. However, running multiple independent protocol instances helps improve the overall throughput. Raft, on the other hand, suffers from the redundant re-proposals; if the followers do not respond fast enough the leader resends the proposal. There is a trade-off between increasing the batch timeout to prevent premature re-proposals and end-to-end latency. With more nodes the batch timeout increases and re-proposals are reduced.

In Figure 5 we observe that ISS latency grows with the number of nodes. This is due to our choice of a fixed batch rate in order to reduce message complexity and sustain high throughput with increasing number of nodes.

6.3 ISS Under Faults

In this section we fix PBFT as the protocol multiplexed with ISS and study its performance under crash faults and Byzantine stragglers in a WAN of 32 nodes. The PBFT view change timeout is set at 10 seconds.

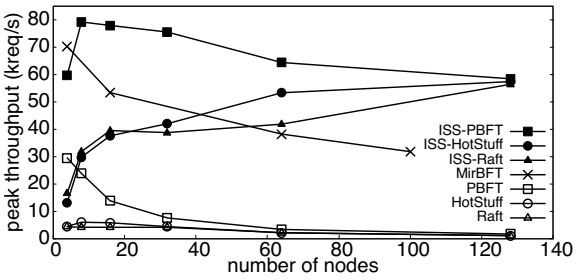

Figure 4. Scalability of single leader protocols, their ISS counterpart, and MirBFT.

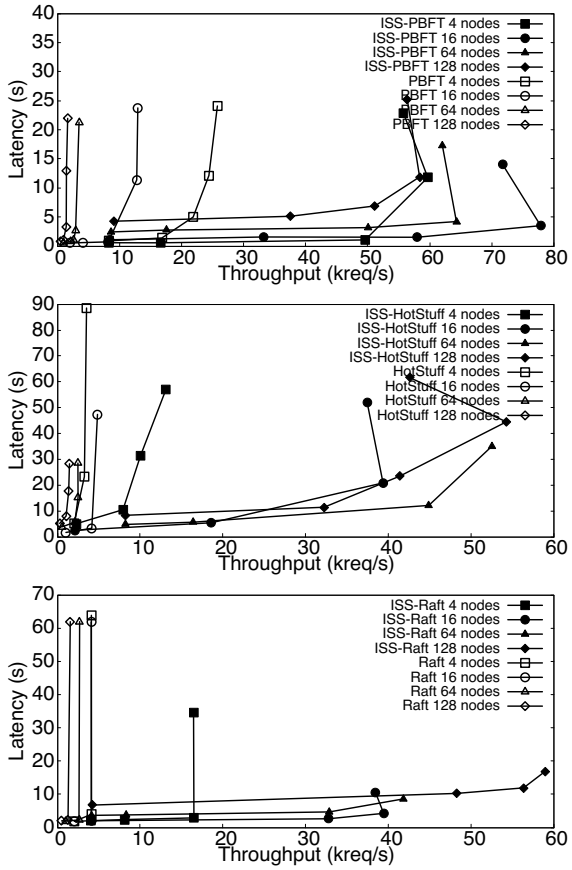

Figure 5. Latency over throughput for increasing load for (a) ISS-PBFT, (b) ISS-HotStuff, and (c) ISS-Raft

6.3.1 Crash Faults. We study two edge cases of faults: (a) one or more leaders crash at the beginning of the first epoch of the execution and (b) one or more leaders crash before sending the proposal for the last sequence number they lead in the first epoch of the execution. The *epoch-start* crash fault is a worst-case scenario for the number of proposed sequence numbers in an epoch. The *epoch-end* crash fault is a worst-case scenario for the duration of the epoch; all nodes need to wait for the fault to be detected at the end of the epoch.

Figure 6 captures the decreasing impact of the crash faults with our leader selection policy as the experiment duration increases. We see that latency converges to that of a fault-free execution as we increase the duration of the experiment. This is due to our leader selection policy removing the faulty node from the leaderset once detected. Note that, regardless of their number, epoch-end failures have a stronger impact on latency (as they delay requests in all buckets) than epoch-start failures (affecting only the faulty nodes' buckets).

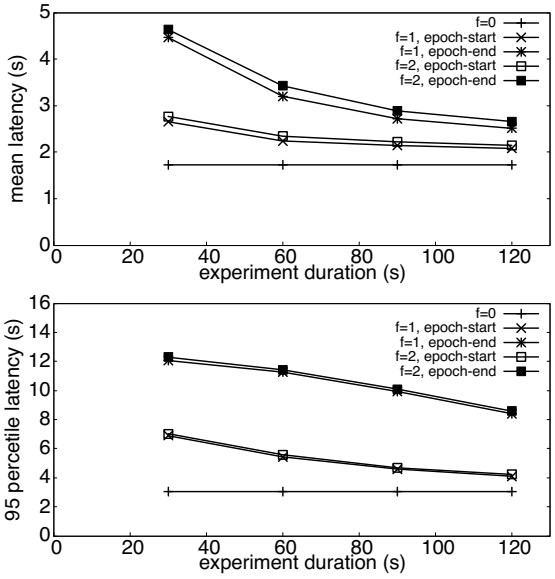

Figure 6. Impact of crash faults on mean (a) and tail (b) end-to-end latency for increasing experiment duration.

Figure 7 shows throughput over time. The short drops to 0 in throughput correspond to the epoch change. We see that an epoch-start fault does not delay the epoch change, as the timer detecting the faulty leader of one segment runs in parallel with other leaders agreeing on requests in their segments. On the other hand, the epoch-end fault delays the epoch change. However, ISS quickly recovers by ordering more than 170k req/s at the beginning of the second epoch (see the spike in Figure 7(b)).

We compare ISS performance under crash faults to MirBFT. In Figure 8 we study run MirBFT on 32 nodes with a single epoch-start crash fault. MirBFT stops processing any message during the epoch change, unlike ISS where segments make progress independently. This results to any crash fault having an impact similar to that of the epoch-end fault for ISS. Moreover, MirBFT relies on an epoch primary for liveness. Every time the crashed node becomes epoch primary it causes an ungraceful epoch change. In Figure 8 this happens around $t = 600$. The phenomenon repeats periodically. Finally, ISS crash fault recovery is more lightweight, since it concerns only the batches of a single segment.

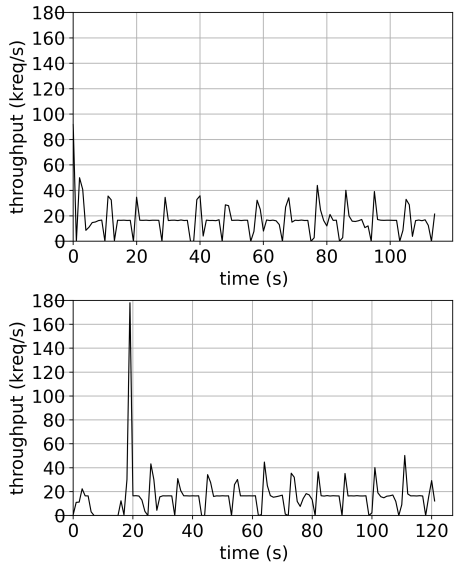

Figure 7. Throughput running average (over 1s intervals) over time with one crash fault at the beginning (a) and at the end (b) of the first epoch.

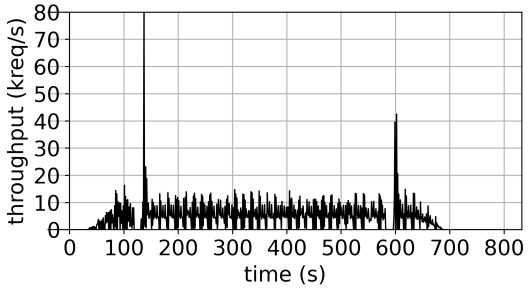

Figure 8. MirBFT throughput running average (over 1s intervals) over time with one epoch-start fault. Epoch change timeout is at 10s and epoch duration is 256 blocks.

6.3.2 Byzantine Stragglers. A Byzantine straggler delays proposals as much as possible without being suspected as faulty and does not add requests in its proposals to harm latency and throughput. We evaluate latency and throughput with $f = 1$ up to the maximum tolerated number of $f = 10$ stragglers. In our evaluation the straggler sends out an empty proposal every 0.5x epoch change timeout.

Figure 9 shows the impact of an increasing number of stragglers. ISS with PBFT reaches from 15% of its maximum throughput with one straggler to 10% of its maximum throughput with 10 stragglers. This, though, translates to maintaining more than 11.4 and 7.9 kreq/s on 32 nodes. Mean latency before saturation increases from $14x$ with one up to $29x$ with 10 stragglers.

Figure 10 shows how throughput is affected over time with total submission rate of 16.4kreqs/s. The performance degradation is due to the "holes" in the log temporarily created by

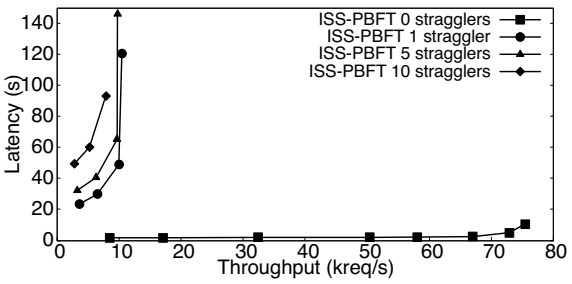

Figure 9. Latency over throughput for an increasing number of stragglers.

the stragglers. Request delivery progresses as fast as the slowest straggler, hence the spikes in the graph. This is inherent to any SMR protocol [3] until the straggler is removed from the leaderset. Straggler resistance in ISS depends on the underlying SB implementation. A more sophisticated leader selection policy implementation could dynamically detect and remove stragglers from the leaderset. ISS facilitates such dynamic detection by comparing the progress of SB instances, which is promising future work.

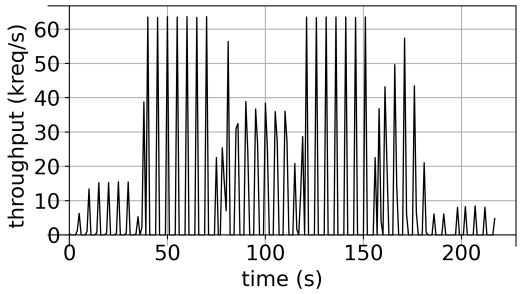

Figure 10. Throughput running average (over 1s intervals) over time with one Byzantine straggler.

7 Related Work

Consensus under Byzantine faults was first made practical by Castro and Liskov [7][8] who introduced PBFT, a semi-permanent leader-driven protocol. The quadratic message complexity of PBFT across all replicas triggered vigorous research towards protocols with linear message complexity. Ramasamy and Cachin [27] replace reliable broadcast in the common case (fault-free execution) with echo broadcast achieving common case message complexity $O(n)$ per delivered payload. Echo broadcast is also exploited in [20][15] to achieve linear common case message complexity. Only recently, HotStuff [29], introduced a 4th communication round to the 3 message rounds of reliable broadcast, to achieve linear message complexity also for the recovery phase (view-change) of the protocol. Regardless the improvement of message complexity, all aforementioned protocols have a single leader, persistent or not, limiting throughput scalability.

Mencius[24] introduced multiple parallel leaders, running instances of Paxos [23], to achieve throughput scalability and low latency in WAN under crash fault assumptions. BFT-Mencius [25] was the first work to introduce parallel leaders under byzantine faults. BFT-Mencius introduced the Abortable Timely Announced Broadcast communication primitive to guarantee bounded delay after GST. However, BFT-Mencius, partitions requests among instances by assigning deterministically clients to replicas, which cannot guarantee load balancing. Moreover, this opens a surface to duplication performance attacks, since malicious clients and replicas can abuse the suggested denial of service mitigation mechanism.

Guerraoui *et al.* [16] also introduced an abstraction which allows BFT instances to abort. The paper uses the abstraction to compose sequentially different BFT protocols, which allows a system to choose the optimal protocol according to network conditions. Our work, on the other hand, composes TOB instances in parallel to achieve throughput scalability.

Mir-BFT [28] is the multi-leader protocol which eliminates request duplication *ante* broadcast, effectively preventing duplication attacks. Later FnF [4] suggests improved leaderset policies for throughput scalability under performance attacks. FnF adopts MirBFT's request space partitioning mechanism for duplication prevention. Similarly, Dandelion[19] leverages the same mechanism to combine Algorand[14] instances. However, Mir-BFT and FnF multiplex PBFT and SBFT [15] instances, respectively, leveraging a single replica in the role of epoch primary. ISS not only eliminates the need for an epoch primary but also provides a modular framework to multiplex any single leader protocol that can implement SB.

Parallel to this work, several works attempt multiplexing BFT instances to achieve high throughput (Redbelly[9], RCC[18], Omada[13]). However, similarly to BFT-Mencius, clients are assigned to primaries, and, after a timeout, a client can change primary to guarantee liveness, allowing again duplication attacks.

8 Conclusion

In this work we introduce ISS, a general construction for efficiently multiplexing instances of leader-based TOB protocols and increasing their throughput. ISS leverages request space partitioning to prevent duplicate requests similarly to MirBFT but rotates the partition assignment without the need of a replica to act as a primary, even in case of faults. To achieve this, we introduced a Sequenced Broadcast, a novel abstraction that generalizes leader-based TOB protocols and which allows periodically terminating and synchronizing their otherwise independent instances. Our evaluation shows that our careful engineering in ISS implementation along with the multi-leader paradigm indeed results in scalable performance for three single leader protocols (PBFT[8], HotStuff[29], and Raft [26]), outperforming their original designs by an order of magnitude at scale.

References

[1] Cosmos: A network of distributed ledgers. https://github.com/dedis/kyber. Accessed: 30.05.2021.

[2] Bitcoin visuals: Transaction sizes. https://bitcoinvisuals.com/chain-tx-size, 2019.

[3] Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State machine replication is more expensive than consensus. Technical report, 2018.

[4] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, and Roger Wattenhofer. Fnf-bft: Exploring performance limits of BFT protocols. *CoRR*, abs/2009.02235, 2020.

[5] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[6] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer-Verlag New York Inc, 2010.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance. *Operating Systems Review*, 33:173–186, 1998.

[8] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[9] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[10] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly blockchain. *CoRR*, abs/1812.11747, 2018.

[11] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 1985.

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[13] Michael Eischer and Tobias Distler. Scalable byzantine fault-tolerant state-machine replication on heterogeneous servers. *Computing*, 101(2):97–118, 2019.

[14] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[15] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 568–580, 2019.

[16] Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. The Next 700 BFT Protocols. In *Proceedings of the ACM European conference on Computer systems (EuroSys)*, 2010.

[17] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Scaling blockchain databases through parallel resilient consensus paradigm. *CoRR*, abs/1911.00837, 2019.

[18] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: resilient concurrent consensus for high-throughput secure transaction processing. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1392–1403. IEEE, 2021.

[19] Kadir Korkmaz, Joachim Bruneau-Queyreix, Sonia Ben Mokthar, and Laurent Réveillère. Dandelion: multiplexing byzantine agreements to unlock blockchain performance. *arXiv preprint arXiv:2104.15063*, 2021.

[20] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2007.

[21] L. Baird. The Swirlds Hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf, 2016.

[22] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[23] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), December 2001.

[24] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[25] Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in byzantine-tolerant state machine replication. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS*, 2013.

[26] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–319, 2014.

[27] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2005.

[28] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. Mir-bft: High-throughput bft for blockchains. *arXiv preprint arXiv:1906.05552*, 2019.

[29] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC*, 2019.