Strengthening Atomic Multicast for Partitioned State Machine Replication

Leandro Pacheco
Università della Svizzera italiana (USI)
Switzerland

Fernando Dotti
Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul - Brazil

Fernando Pedone
Università della Svizzera italiana (USI)
Switzerland

ABSTRACT

Partitioned state machine replication is a technique that extends classical state machine replication with state partitioning (or sharding) to provide both fault tolerance and performance scalability. The crux of the technique is ordering client requests within a partition, among the replicas that implement the partition, and across partitions, involving all the replicas accessed by the request. To cope with the complexity of ordering requests, partitioned state machine replication can use atomic multicast, a communication abstraction. Atomic multicast provides the means for requests to be propagated reliably and consistently to one or more sets of groups of replicas, where each replica group implements one partition. The paper revisits atomic multicast from the perspective of partitioned state machine replication and makes the following contributions: First, we show that if one implements partitioned state machine replication using an atomic multicast with global total order, a strong order property, then replicas would need to further coordinate as part of the execution of requests to ensure correctness. Second, we introduce a stronger version of atomic multicast that accounts for real-time dependencies between requests. Our proposed atomic multicast can be used to order requests within and across partitions so that replicas do not need to further coordinate to ensure linearizability. Third, we extend a well-known implementation of atomic multicast to ensure the stronger order property.

ACM Reference Format:
Leandro Pacheco, Fernando Dotti, and Fernando Pedone. 2022. Strengthening Atomic Multicast for Partitioned State Machine Replication . In *Proceedings of 11th Latin-American Symposium on Dependable Computing (LADC 2022).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Designing strongly consistent applications that tolerate failures and scale performance is challenging. State machine replication and primary-backup replication, the two most fundamental techniques for fault tolerance [18], do not scale performance as replicas are added to the system. In state machine replication, client requests are executed by all the replicas in the same order. As long as execution is deterministic, replicas will transition through the same state changes and produce the same results. Since each replica executes every request, adding additional replicas will not result in any performance improvements. On the contrary, a larger number of replicas may lead to a degradation in performance. This happens because replicas need to coordinate to totally order requests, and the more replicas involved, the more messages need to be exchanged, reducing the number of requests that can be ordered in the system [17]. In primary-backup replication, the primary replica receives and executes all requests, and then sends state changes to the backup replicas. The backups simply apply the state changes. As in state machine replication, increasing the number of replicas may degrade performance as it increases communication between the primary and the backups.

State partitioning (also known as sharding) is an important technique to scale the performance of distributed applications [9]. The idea is to divide the application state into partitions and store each partition in a different set of servers. Therefore, requests that access different partitions can be executed in parallel. As long as the application state can be divided into an increasing number of partitions and requests access subsets of partitions uniformly, one can expect performance to improve when increasing the number of partitions. Handling requests that access data stored in multiple partitions is challenging as such requests require the involved servers to coordinate. Sharding can be combined with state machine replication to provide both performance scalability and fault tolerance. In Scalable State Machine Replication (S-SMR) [4], for example, each partition is implemented by a set of replicas which interact using state machine replication. Requests that access a single partition are ordered by the replicas in charge of the accessed partition. Requests that access multiple partitions must be consistently ordered by all the replicas involved in the request. In this paper, we refer to such an approach as *partitioned state machine replication.*

Ordering requests within and across partitions is complex (e.g., it may involve solving multiple instances of consensus [6], one consensus instance per partition). One way to cope with this complexity is to encapsulate the ordering of requests in a group communication protocol called atomic multicast—not to be confused with IP multicast, a network-level communication primitive. In atomic multicast, the replicas of a partition constitute a group, a unit of communication that can be addressed as a whole. The message sender defines the destination of a message by specifying the set of groups that must order and deliver the message. An atomic multicast algorithm determines how the destination replicas must coordinate to order messages within and across groups consistently.

This paper revisits the problem of atomic multicast from the perspective of strongly consistent replicated applications. More specifically, we relate the notions of strong consistency, in the form of linearizability [21], and atomic multicast in the context of partitioned state machine replication. Intuitively, linearizability states that clients of a sharded and replicated service must observe the execution of requests as if the service was implemented by a

LADC 2022, November 21–24, 2022, Fortaleza, CE, Brazil
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

single server. Linearizability captures real-time dependencies between requests, that is, if a request precedes another request in real time, then the execution of the first request should be reflected in the execution of the second request. Several specifications of atomic multicast exist in the literature [11]. In their seminal taxonomy, Hadzilacos and Toueg [20] have proposed a hierarchy of atomic multicast definitions, the strongest of which ensuring a property called global total order. In brief, global total order states that collectively replicas must order requests consistently. For example, if one replica orders message m_1 before message m_2 and another replica orders m_2 before m_3, then any replica that delivers m_1 and m_3 should order m_1 before m_3. We show in the paper that if one implements partitioned state machine replication using an atomic multicast with global total order, then replicas need to further coordinate as part of the execution of requests to respect the real-time dependencies required by linearizability. Indeed, existing approaches to partitioned state machine replication introduce an ad hoc coordination phase between servers during the execution of multi-shard requests (e.g., [4, 24]). The result presented in this paper suggests that to avoid this coordination, partitioned state machine replication needs a stronger notion of atomic multicast.

We introduce a stronger version of the strongest atomic multicast in [20], that accounts for real-time dependencies between requests. The proposed protocol can be used to order requests within and across partitions so that replicas do not need to further coordinate to ensure linearizability. It uses atomic global order, a property strictly stronger than global total order. We revisit a well-known atomic multicast algorithm, attributed to Skeen [5], and show that it does not satisfy atomic global order. Skeen's atomic multicast algorithm does not tolerate failures, but it serves as the basis for several protocols that have extended the algorithm's original ideas to cope with replica failures (e.g., [7, 8, 19, 32]). Our result holds for these extensions as well. We then show how Skeen's atomic multicast can be modified to guarantee atomic global order, and thus implement the proposed Atomic Multicast protocol.

This paper is organized as follows.

- Section 2 provides the background information needed to follow the rest of the paper, introducing basic assumptions and the definitions of atomic multicast, linearizability, state machine replication (SMR), and partitioned SMR.
- In Section 3, we prove that partitioned state machine replication with an atomic multicast primitive that ensures global total order requires additional replica coordination to implement linearizable applications. We propose an atomic multicast, based on atomic global order, and prove that when equipped with such an atomic multicast, replicas do not need this additional coordination.
- Section 4 shows that the well-known atomic multicast protocol proposed by Skeen does not ensure atomic global order and present modifications to the original protocol to guarantee the stronger property.
- Section 5 reviews several specifications and implementations of atomic multicast and a few systems that implement consistent order using ad hoc mechanisms.
- Section 6 concludes the paper with a few observations and directions for further research.

2 BACKGROUND

In this section, we present the system model and definitions used in the paper. We then characterize atomic broadcast and atomic multicast, two fundamental communication abstractions we build upon. We conclude the section with a description of state machine replication (SMR) and partitioned SMR.

2.1 System model and definitions

We assume a distributed system with a bounded set of processes, $\Pi = \{p_1, ..., p_n\}$. Processes can fail by crashing but never perform incorrect actions (i.e., no Byzantine failures). A process is *correct* if it does not crash, and *faulty* otherwise. We define $\Gamma = \{g_1, ..., g_m\}$ as the set of disjoint process groups in the system. We assume each group contains $2f + 1$ processes, where f is the maximum number of faulty processes per group. The assumption about disjoint groups has little practical implication since it does not prevent collocating processes that are members of different groups on the same machine. Yet, it is important since atomic multicast requires strong assumptions when groups intersect [19, 33].

Processes communicate by exchanging messages and do not have access to a shared memory or a global clock. Communication links do not create, corrupt, or duplicate messages, and guarantee that if a correct process p sends a message m to a correct process q, then q receives m. We assume the system is partially synchronous [13]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)*, and it is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown.

2.2 Atomic broadcast and multicast

Atomic broadcast and multicast are abstractions that offer processes strong communication guarantees. These abstractions can be used to render an application fault tolerant by means of replication (see Figure 1). Since atomic broadcast is a special case of atomic multicast, we present the more general properties of atomic multicast. Atomic multicast is defined by primitives multicast(m) and deliver(m), where m is a message addressed to a subset of groups in Γ. We represent the destination groups of message m as $m.dst$. By abuse of notation, we write $p \in m.dst$ instead of $\exists g \in \Gamma : g \in m.dst \wedge p \in g$.

Toueg and Hadzilacos [20] define three types of atomic multicast that differ by the strength of their guarantees. We consider the strongest type of atomic multicast, defined by the four properties presented next. (More precisely, we present the uniform version of the atomic multicast properties defined in [20].)

- *Validity:* If a correct process multicasts a message m, then some correct process in $m.dst$ delivers m.
- *Agreement:* If a process delivers a message m, then all correct processes in $m.dst$ eventually deliver m.
- *Integrity:* For any message m, every process p delivers m at most once, and only if $p \in m.dst$ and m was previously multicast.

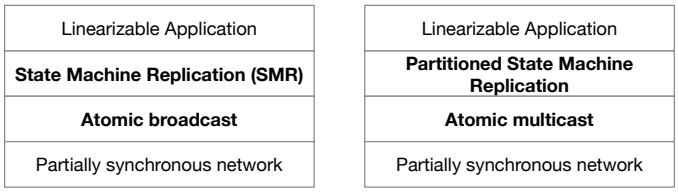

Figure 1: State machine replication.

- *Global total order:* Define relation $\prec$ on the set of messages processes deliver as follows: $m \prec m'$ iff there exists a process that delivers m before m'. The relation $\prec$ is acyclic.

Global total order avoids cycles in the delivery sequence of messages. For example, suppose there are three processes, p_x, p_y, and p_z, each one in a different group, and messages m_1, m_2 and m_3. Global total order prevents a situation where p_x delivers m_1 and then m_2 ($m_1 \prec m_2$), p_y delivers m_2 and then m_3 ($m_2 \prec m_3$), p_z delivers m_3 and then m_1 ($m_3 \prec m_1$).

But global total order by itself allows faulty processes to deliver undesired sequences of messages. Indeed, it allows "holes" to appear in the message delivery sequence of faulty processes. For example, consider an execution where messages m_1 and m_2 are multicast to group g. A process p in g delivers m_1 and then m_2, and a faulty process q in g delivers m_2, then fails, and never delivers m_1 (i.e., m_1 leaves a hole in the delivery sequence of q). This execution satisfies all atomic multicast properties above, but it is undesired because processes p and q may produce different results after executing a request in m_2. To prevent such executions, we also require atomic multicast to also satisfy prefix order [31].

- *Prefix order:* For any two messages m and m' and any two processes p and q such that $\{p, q\} \subseteq m.dst \cap m'.dst$, if p delivers m and q delivers m', then either p delivers m' before m or q delivers m before m'.

Atomic broadcast is a special case of atomic multicast where there is a single group in Γ.

2.3 State machine replication

State machine replication is a well-established approach to rendering applications fault-tolerant. Servers fully replicate the application state and execute the same requests in the same order. Consequently, every replica transitions through the same sequence of state changes and produces the same sequence of responses upon executing requests. Clients that interact with an application that tolerates failures by means of state machine replication observe the same application behavior as if the application was implemented by a single replica. This aspect of state machine replication is more formally captured by the property below.

- *Linearizability:* For any execution σ, there is a total order π on application requests that:
  (i) respects the semantics of the requests, as defined in their sequential specifications, and
  (ii) respects the real-time precedence of requests, where a request precedes another request in real time if the first request finishes before the second request starts [2, 21].

2.4 Partitioned state machine replication

State machine replication improves application availability but not performance, as a consequence of every replica executing all requests. Some approaches have proposed to partition the application state, also known as sharding, and implement each partition with an instance of SMR (e.g., [9]). If the partitioning is such that requests fall within one partition only (i.e., the objects read and written by the request belong to a single partition) and requests are evenly distributed among partitions, then we can scale performance with the number of partitions in the system. Moreover, since linearizability is a composable property [21], such a scheme will result in a linearizable application.

If the application state cannot be perfectly partitioned, as described above, then one must account for requests that involve multiple partitions (i.e., a request that reads and writes objects that belong to more than one partition). There are two aspects concerning multi-partition requests: how to consistently order requests that span multiple partitions, and how to execute them. Atomic multicast is a useful abstraction to propagate requests to partitions reliably and properly ordered.

Executing requests that involve multiple partitions is challenging since partitions may lack the state needed to execute the request. For example, assume a request r that swaps the contents of state variables x and y, which reside in partitions P_x and P_y, respectively. Replicas in P_x (resp. P_y) need the value of y (resp. x) in order to update x (resp. y). In S-SMR [4], after delivering a request, replicas exchange the state needed to execute the request. In the example above, replicas in P_x (resp. P_y) send to replicas in P_y (resp. P_x) variable x (resp. y). After exchanging the needed variables, replicas in all involved partitions execute r. Updates on variables not part of a partition are kept by the partition momentarily, during the execution of the request, and then discarded. In DynaStar [24], all variables read and written by a multi-partition request are moved to one of the partitions involved in the request. Only replicas in this partition execute the request.

The details of how to execute multi-partition requests are orthogonal to our contributions in this paper. Hence, for simplicity, we assume that multi-partition requests can be executed at each involved partition without exchange of data. A request finishes execution once the issuing client gets a reply from each involved partition. This simplified execution model does not allow a request to swap the contents of variables x and y if they reside in different partitions. But our model can still capture interesting applications. For illustrative purposes, hereafter, we consider a key-value store service that supports two types requests: inserts and range queries. Key-value pairs are partitioned in two partitions, P_0 being responsible for even-numbered keys and P_1 for odd-numbered keys. An insert $w(k, v)$ inserts the pair (k, v) and is a single-partition request, multicast only to the partition in charge of the key. A range query $r(k_{start}, k_{end})$ returns all previously inserted pairs for keys from k_{start} up to and including k_{end}. Range queries are multicast to both partitions, assuming ranges that span multiple keys.

3 ATOMIC GLOBAL ORDER

In this section, we argue that atomic multicast as the sole means of communication among replicas in partitioned state machine

replication is not enough to ensure linearizability. We then extend the atomic multicast properties to achieve linearizable executions and prove their correctness.

3.1 Atomic multicast alone is not enough

State machine replication can be easily implemented with atomic broadcast [18]. It suffices for client requests to be atomically broadcast to all replicas, which execute the requests following the order in which the requests are delivered. One would expect that partitioned state machine replication could be implemented using a similar approach, namely, by atomically multicasting requests to all the groups involved in the request. Upon delivering the request, a replica executes the request and sends the results to the client. The request finishes at the client after the client receives a response from at least one server in each group involved in the request. As we show next, however, this simple execution model and the multicast properties as described in Section 2 are not enough to ensure linearizability.

Consider an execution of the key-value store service described in Section 2.4, involving partitions P_0 and P_1, as depicted in Figure 2 (left). The two commands $w(0, v_0)$ and $w(1, v_1)$ insert key-value pairs for keys 0 and 1, respectively. Moreover, $w(0, v_0)$ precedes $w(1, v_1)$ in real-time, that is, $w(0, v_0)$ finishes at client b before $w(1, v_1)$ starts at client c. Now consider a concurrent range query $r(0, 1)$ that tries to read previously inserted pairs for keys 0 and 1. The range query accesses both partitions and is delivered and finishes at P_0 before $w(0, v_0)$ is delivered, and is delivered at P_1 after $w(1, v_1)$ ends. The prefix order property of atomic multicast is not violated since all processes that deliver the same messages, do it in the same order. The atomic multicast acyclic order property is not violated either: $w(0, v_0)$ and $w(1, v_1)$ are not directly related since they are delivered at different partitions, and thus, $r(0, 1) \prec w(0, v_0)$ at P_0 and $w(1, v_1) \prec r(0, 1)$ at P_1. Although the $\prec$ relation is acyclic, $w(1, v_1) \prec r(0, 1) \prec w(0, v_0)$, it does not account for the real time order in which $w(0, v_0)$ precedes $w(1, v_1)$. Even though $(0, v_0)$ is inserted before $(1, v_1)$ in the key-value store, the range query only returns the pair $(1, v_1)$, violating linearizability.

The problem above stems from the fact that atomic multicast properties do not capture real-time dependencies between requests. Consequently, a system that implements partitioned state machine replication with the atomic multicast properties described in Section 2 would need to introduce additional coordination across partitions to ensure linearizability. For example, in S-SMR[4], after multi-partition request $r(0, 1)$ is delivered and before it is executed by the replicas, partitions P_0 and P_1, involved in $r(0, 1)$, exchange "signal messages" to avoid the problem described above, as depicted in Figure 2 (right). Intuitively, if partitions P_0 and P_1 exchange messages during the execution of $r(0, 1)$, it is not possible for one partition to finish executing $r(0, 1)$ before the other starts. Thus, in between the execution of $r(0, 1)$ at the partitions involved, we cannot have other commands being executed at the same partitions.

3.2 Atomic Multicast for Partitioned SMR

The discussion in the previous section shows that differently than atomic broadcast in state machine replication, atomic multicast is not sufficient as a communication abstraction to ensure linearizability in partitioned state machine replication without additional coordination among replicas. We now strengthen atomic multicast so that replicas can execute the request after delivering it without further coordination.

Our strategy is to enlarge the scope of the ordering guarantees of atomic multicast. We achieve this by replacing the global total order property of atomic multicast with the following property, which accounts for the real time relation of messages.

- *Atomic global order*: Define relation $\prec$ on the set of messages processes deliver as follows: $m \prec m'$ iff (i) there exists a process that delivers m before m'; or (ii) m' is multicast after m is delivered at some destination, in real time. The relation $\prec$ is acyclic.

The atomic global order property introduces two aspects: it relates atomic multicast primitives in real time and it relates messages multicast to possibly disjoint destinations. Capturing these real-time dependencies is fundamental to linearizability. For example, the execution in Figure 2 (left) does not satisfy atomic multicast extended with atomic global order: (a) since $w(1, v_1)$ is multicast (by client c) after $r(0, 1)$ is delivered by the replica at partition P_0, it follows from atomic global order that $r(0, 1) \prec w(1, v_1)$; and (b) from the delivery order of requests at P_1, $w(1, v_1) \prec r(0, 1)$, which leads to a cycle and violates atomic multicast's global total order.

3.3 Proof of correctness

In the following, we show that atomic multicast extended with the atomic global order property ensures that partitioned state machine replication executions are linearizable.

Let σ be an execution of partitioned state machine replication where (a) a client starts a request by multicasting the request to all the partitions involved in the request (i.e., partitions containing data read and written as part of the request), (b) when the request is delivered by a replica, the replica immediately executes the request and responds to the client, and (c) the client considers the request as finished after it receives a response from at least one replica in each partition involved in the request.

Let π be a total order of requests in σ that respects $\prec$, the order induced on requests by atomic multicast extended with atomic global order.

To argue that π respects the semantics of requests, let C_i be the i-th request in π and p a process in partition x that executes C_i. We claim that when p executes C_i, all read operations issued by p as part of C_i result in values that reflect all requests that precede C_i and no value created by a request that succeeds C_i. This follows from (i) the fact that replicas execute requests sequentially, in the order in which they are delivered, and (ii) the assumption that when executing a multi-partition request, a replica does not read data stored at other partitions (see Section 2.4).

We now argue that π respects the real-time precedence of requests in σ. Assume that C_i ends at a client before C_j starts at a client. We must show that either (a) $C_i \prec C_j$; or (b) neither $C_i \prec C_j$ nor $C_j \prec C_i$. In case (a), C_i precedes C_j in π; in case (b), since C_i and C_j are not related, we can choose a total order π where C_i appears before C_j.

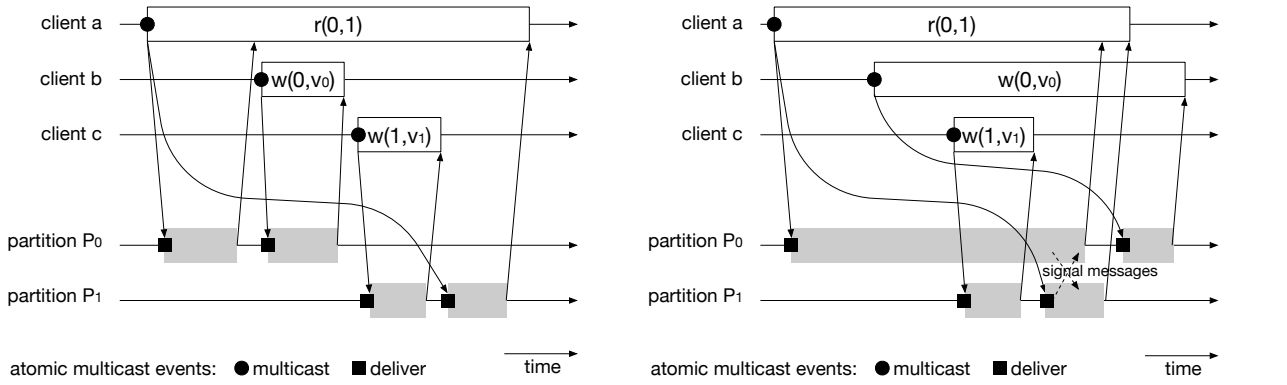

Figure 2: An execution that violates linearizability (left) and a linearizable execution from S-SMR [4] (right). For simplicity, we assume that each partition contains a single replica.

For a contradiction, assume that $C_j \prec C_i$. Since C_i ends in real time before C_j starts (from our initial assumption), the client issuing C_i has received a response from a replica p in each of the partitions involved in the execution of C_i. And from the algorithm, p has delivered C_i before executing it and responding to the client. We conclude that C_j is multicast after p delivers C_i. From the atomic order property of atomic multicast, we have that $C_i \prec C_j$, which leads to a cycle and violates the global total order property of atomic multicast, reaching a contradiction.

4 IMPLEMENTING ATOMIC GLOBAL ORDER

In this section, we present an early atomic multicast algorithm attributed to Skeen [5] and show that it does not guarantee the atomic global order property. We then extend this algorithm to ensure the property and argue about the correctness of the extended algorithm.

4.1 Skeen's atomic multicast

In Skeen's algorithm, each process assigns unique timestamps to multicast messages based on a logical clock [22]. The correctness of the algorithm stems from two basic properties: (i) processes in the destination of a multicast message first assign tentative timestamps to the message and eventually agree on the message's final timestamp; and (ii) processes deliver messages according to their final timestamp. These properties are implemented as follows. (We recall that Skeen's atomic multicast algorithm does not tolerate failures.)

(i) To multicast a message m to a set of processes, p sends m to the destinations. Upon receiving m, each destination updates its logical clock, assigns a local timestamp to m (a tuple of clock value and partition id for breaking ties), stores it, and sends its local timestamp for m to all destinations in $m.dst$. Upon receiving local timestamps from all destinations in $m.dst$, a process computes m's final timestamp ts as the maximum among all received local timestamps for m. The process then ensures its logical clock is higher than or equal to ts.

(ii) Messages are delivered respecting the order of their final timestamp. A process p delivers m when it can ascertain that m's final timestamp is smaller than the final timestamp of any messages p will deliver after m (intuitively, this holds because logical clocks are monotonically increasing).

Algorithm 1 Skeen's protocol at partition P_x. Additions to satisfy atomic global order in gray.

```
1:  clock ← 0                                   ▷ p's logical clock
2:  local[] ← ∅                                 ▷ map from message to local timestamp at p
3:  final[] ← ∅                                 ▷ map from message to decided final timestamp
4:  acked ← ∅                                   ▷ set of messages ACKed by all their destinations
5:  del ← ∅                                     ▷ set of delivered messages

6:  multicast(m):
7:      send ⟨START, m⟩ to m.dst

8:  when receive ⟨START, m⟩:
9:      clock ← clock + 1
10:     local[m] ← ⟨clock, P_x⟩
11:     send ⟨LOCAL-TS, m, local[m]⟩ to m.dst

12: when receive ⟨LOCAL-TS, m, ts⟩ from all partitions m.dst:
13:     final[m] ← maximum ts received for m
14:     clock ← max(clock, final[m])
15:     tryDeliver()
16:     send ⟨ACK, m⟩ to m.dst

17: when receive ⟨ACK, m⟩ from all partitions in m.dst:
18:     acked ← acked ∪ {m}
19:     tryDeliver()

20: tryDeliver():
21:     for each m ∈ final \ del : m ∈ acked in final[m] order
22:         if ∀m' ∈ local \ del :
23:                 (m' ∈ final ∧ final[m] < final[m']) ∨
24:                 (final[m] < local[m']) then
25:             del ← del ∪ {m}
26:             deliver(m)
```

The complete protocol is shown in Algorithm 1 (ignoring the extensions in gray). Figure 3 (left) shows an example execution of the algorithm, where two clients a and b multicast messages m and m' respectively, with $m.dst = \{P_x, P_y\}$ and $m'.dst = \{P_x, P_z\}$. Initially, message m is sent to its destinations and is assigned the local timestamps 1 (from P_x) and 5 (from P_y). As soon as P_y receives

the local timestamp from P_x, it knows the final timestamp of m is 5. P_y updates its logical clock to 5 and can immediately deliver m: any new messages will be assigned a higher local timestamp at P_y. P_x, on the other hand, cannot deliver m immediately after it receives the local timestamp from P_y: in the mean time it assigned a local timestamp of 2 to m', and must wait for the final timestamp of m' before it knows which of the two messages must be delivered first.

4.2 Extending Skeen's algorithm to ensure atomic global order

The execution in Figure 3 (left) demonstrates that Skeen's algorithm does not ensure atomic global order. Even though m' is multicast after m is delivered at partition P_y, in real-time, we have $m' \prec m$ at P_x.

We modify Skeen's algorithm to ensure atomic global order by including one extra property that needs to be satisfied: (iii) a process can only deliver a message with final timestamp ts once it knows that every destination in $m.dst$ will not assign a local timestamp smaller than or equal to ts. If (iii) is satisfied, once m is delivered with final timestamp ts by some process, no new message m' such that $m.dst \cap m'.dst \neq \emptyset$ can be assigned a final timestamp smaller than ts. The property is ensured by adding one extra message exchange between destinations. Once a process decides on the final timestamp of message m, after updating its clock if needed, it sends an acknowledgement to each other process in $m.dst$. A process can only deliver m once it receives an acknowledgement from every other process in $m.dst$. The additions to the protocol are shown in gray in Algorithm 1.

Figure 3 (right) shows a similar execution to the one in Figure 3 (left), but with the extended protocol. Partition P_y cannot deliver m at the moment it decides on the final timestamp: it must wait for P_x to acknowledge it. Since P_x timestamps m' before it acknowledges m, the delivery of m at P_y is delayed to a point after the multicast of m'. Thus, $m' \prec m$ does not create a cycle.

We now argue that the extended Skeen's protocol satisfies atomic global order. Let m and m' be two messages such that m' is multicast after m is delivered at some destination P. Furthermore, assume for a contradiction that there is some destination in common, P', that delivers m' before it delivers m. When m is delivered at P, from the algorithm, it must have received an ACK for m from P'. Thus, P' must know the final timestamp ts of m, and must have advanced its logical clock past ts. Since the $\langle \text{START}, m' \rangle$ message arrives at P' after that point, it follows that the local timestamp assigned to m' at P', and consequently its final timestamp ts', must be larger than ts. But since P' delivers m before m', we have that $ts < ts'$, a contradiction.

Section 5.2 discusses several fault-tolerant atomic multicast protocols that are derived from Skeen's protocol [8, 15, 16, 30, 32]. It is our understanding that the here proposed extension can easily be applied to these protocols as well.

5 RELATED WORK

We organize the related work from different perspectives. First we briefly comment on the existing atomic multicast properties and the here proposed atomic global order. We then examine existing atomic multicast protocols and discuss whether they satisfy the property or not. Atomic global order targets systems that aim at linearizability, so we follow with a discussion about the consistency level of current partitioned SMR and partitioned transactional systems.

5.1 Atomic multicast properties

The functionality and semantics of an atomic broadcast or multicast protocol is defined by a set of properties (i.e., validity, agreement, integrity and order) that establish relations on the occurrence of its primitives in any given run of the protocol. Different applications may have different requirements, and these properties can be made stronger or weaker depending on the need of the application. Many protocols have been proposed in the literature, relying on different assumptions and satisfying different sets of properties [11]. Atomic global order differs from previously proposed ordering properties in that (i) it relates delivery and multicast events and (ii) it relates them in real-time. As previously shown, atomic global order allows partitioned systems to provide linearizability without further coordination among processes.

5.2 Atomic multicast protocols

We focus on atomic multicast protocols based on destination agreement [11]. This is the class of protocols in which the order of messages results from agreement between the destination processes. We further divide these protocols in three classes: timestamp based, overlay based and deterministic merge based.

Timestamp based. In these algorithms, processes first agree on the assignment of message timestamps and then deliver messages in timestamp order. Every protocol discussed here is genuine, and employs a timestamping scheme similar to Skeen's, discussed in detail in Section 4.1.

The first protocol extending Skeen's protocol to be fault-tolerant by using consensus inside each destination group was presented in [15]. Each group acts as a process from Skeen's protocol, and relies on consensus to decide on timestamp proposals and advance the logical clock. Messages can be delivered in 6 communication steps. In [32], a similar protocol is proposed with optimizations to speed up delivery in specific circumstances.

In Scalatom [30], instead of using consensus inside each destination group to decide on timestamp proposals, a single consensus instance is executed among all destination processes. It can deliver messages in 6 communication steps. Scalatom also proposes and satisfies the additional property of *message size minimality*: protocol messages should have size proportional to the number of destination groups. We note that all algorithms discussed here also satisfy the property.

FastCast [8] proposes the use of stable leaders and an optimistic execution path that can deliver messages in 4 communication steps, in the best case.

Instead of using consensus as a black-box, White-Box Atomic Multicast [16] weaves Skeen's timestamping scheme and Paxos into a unified protocol. The protocol is primary-based, and can deliver messages in 3 communication steps at group leaders, in the best case. Each group leader is responsible for assigning the local timestamp for its group, and decides when a message is safe for delivery.

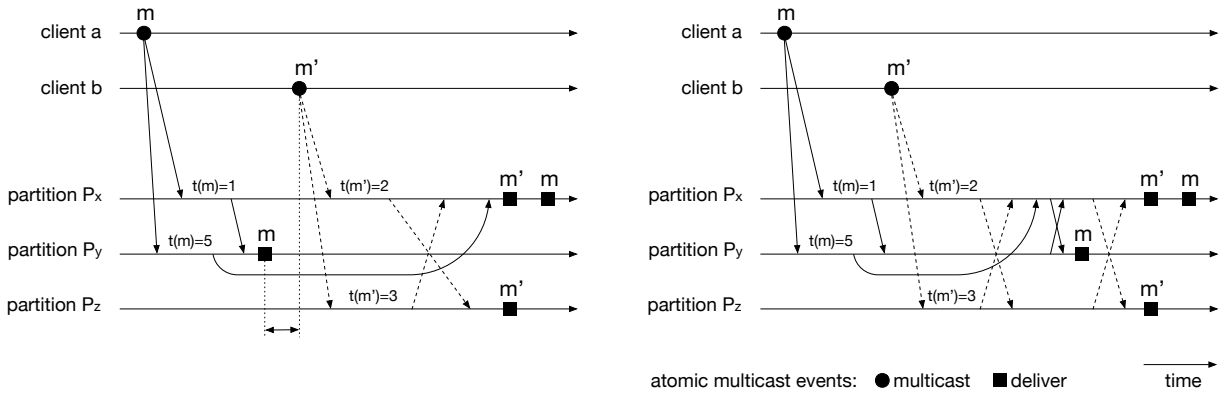

Figure 3: An execution showing that Skeen's atomic multicast violates atomic global order (left) and the extended algorithm that guarantees atomic global order (right).

RamCast [23] is a primary-based protocol that achieves high throughput and low latency through its use of Remote Direct Memory Access (RDMA). It combines ideas from Skeen and Protected Memory Paxos [1].

Overlay based. These algorithms rely on a predefined topology to propagate messages and to ensure atomic multicast properties.

In [12], a genuine atomic multicast protocol is proposed that uses a total order of groups as an overlay. A message m that needs to be multicast is initially sent to one group in $m.dst$. When the group receives m, consensus is used to order and deliver m inside the group, then m is forwarded to the next group in $m.dst$ (according to the total order of groups). A group that delivers m can only order the next message once it knows m is ordered in all of $m.dst$, after it receives an END message from the last group in $m.dst$.

Byzcast [7] is a byzantine fault-tolerant atomic multicast that arranges groups in a tree overlay. A message m enters the overlay in the lowest common ancestor of groups in $m.dst$. A group that receives m orders it and then propagates it down the tree, until all groups in $m.dst$ are reached. Byzcast may be considered partially genuine: delivering a message addressed to multiple groups may involve intermediary groups not part of the destination set.

Deterministic merge based. In these algorithms, each destination process applies a deterministic merge procedure to decide on the delivery order of received messages.

In [32], processes execute an unbounded sequence of rounds. Consensus is used inside each group to determine the set of messages proposed by the group in a given round. At the end of each round, processes gather messages from all groups and then deliver them in some deterministic order. The protocol is non-genuine.

Multi-Ring Paxos [26] builds on multiple instances of Ring Paxos [27], a ring-based consensus protocol. It provides guarantees akin to atomic multicast, but with a slightly different interface. Messages can only be sent to a single group, but receivers can subscribe to more than one group. Each group totally orders its messages and receivers use a deterministic merge to ensure a partial ordering of deliveries from the groups it subscribes to. Ridge [3] improves on Multi-Ring Paxos by reducing the latency inside each group and utilizing a timestamp-based merge procedure. Both protocols are non-genuine.

5.3 Existing algorithms and atomic global order

In the following, we discuss why, out of all surveyed protocols, only the round-based protocol described in [32] satisfies atomic global order.

Timestamp based. As shown in Section 4.1, Skeen's algorithm does not satisfy atomic global order. By the same argument, all of the algorithms that more closely match its execution (i.e., [8, 15, 32]) do not satisfy the property either.

In Scalatom [30], while a single instance of consensus is executed among groups in $m.dst$, a group's clock is only updated after it handles the decision for m's timestamp. It is possible for a group in $m.dst$ to propose a smaller timestamp even after some other group in $m.dst$ delivers m.

In the primary-based algorithms (i.e., [16, 23]), at the time a group leader delivers a message m with final timestamp ts, other group leaders in $m.dst$ may still propose local timestamps smaller than ts.

Overlay based. In the protocols relying on an overlay for partial order, messages may be ordered by groups in sequence, either following a total order [12] or down a tree [7]. Consider two messages m and m' and groups g and h such that $g \in m.dst$ and $h \in m.dst \cap m'.dst$, and g is earlier than h in the overlay. If m' is multicast after m is delivered by g, but before m its propagated to h, h may deliver m' before m, violating atomic global order.

Deterministic merge based. For Multi-Ring Paxos [26] and Ridge [3], consider the following case. Two messages m and m' are ordered by groups g and h respectively. A receiver subscribing only to g delivers m, and only then m' is multicast to h. The proposed merge procedures do not prevent another receiver, subscribing to both g and h, from delivering m' before m, violating atomic global order.

Out of all surveyed protocols, only the non-genuine, round-based protocol described in [32] satisfies atomic global order. If some process delivers a message m, it follows that the round to which m

belongs is closed in all groups. Any message multicast after that must belong to some later round, ensuring it is delivered after *m*.

5.4 Partitioned SMR

The quest for scalable SMR very often involves partitioning techniques, where handling multi-partition operations (MPOs) efficiently is one of the main challenges.

Scalable State Machine Replication (S-SMR) [4] is an approach that achieves scalable throughput and linearizability without constraining service commands or adding additional complexity to their implementation. S-SMR partitions the service state and relies on an atomic multicast primitive to consistently order commands within and across partitions. It is shown that simply ordering commands consistently across partitions is not enough to ensure linearizability in partitioned state machine replication. To ensure linearizability, S-SMR implements execution atomicity, a property that prevents invalid command interleavings. Partitions involved in the same operation signal each other such that all of them finish the operation before signaling the client (see also Figure 2).

In [25], the authors propose a genuine protocol based on Skeen's total order multicast [5] to order MPOs. The inter-partition coordination for MPOs is removed from the critical execution path of operations. This is achieved by postponing the execution of MPOs to a future time when their ordering has already been agreed across the partitions involved. For this, a new consensus interface and properties are proposed. Operations are executed in rounds. The proposal primitive allows to propose operations with the rounds intended to execute them. While single partition operations can be ordered quickly and execute soon, MPOs have to go through an inter-partition procedure. Scheduling MPOs for future rounds allows other operations to execute while the MPOs are being ordered. To ensure linearizability, when a process starts executing an MPO, it notifies all involved partitions. A process only replies to the client after having received this notification from all involved partitions. This solution is similar to the one presented in [4] and ensures that the reply externalized to the client is consistent with linearizability.

DynaStar [24] is a partitioned SMR solution that provides dynamic state partitioning to handle workloads with varying access patterns. Data can be moved between partitions, and a location oracle is used to monitor the workload and to re-calculate an optimized partitioning on demand. In DynaStar there is no multi-partition execution of commands. If a command accesses multiple partitions, all data is temporarily moved to a single partition that is then responsible for executing the command.

Tempo [14] is a leaderless partitioned SMR protocol that relies on a timestamping scheme similar to Skeen's. Each command has a coordinator replica that communicates with the destination replicas to replicate the command and to agree on its timestamp. To ensure linearizability, replicas exchange information about each timestamp they assign and will only execute a command once every command with a lower timestamp is known.

5.5 Transactional systems

As in SMR approaches, distributed transactional systems also rely on data partitioning for scalability. Instead of relying on an atomic multicast abstraction to partially order requests, these systems typically employ ad hoc protocols to coordinate multi-partition operations.

P-Store [34] offers a partially replicated key-value store for wide area networks with one-copy serializability semantics. Each transaction proceeds with optimistic reads, storing on its read-set the data and its version. Upon certification, needed for global and update transactions, it is checked whether the transaction observed a valid view of the database. The certification protocols proposed rely on genuine atomic multicast. Each site that executes the transaction has to certify it. If it cannot, then the whole transaction is aborted. To speed up the termination of transactions, a parallel certification protocol for independent transactions is also proposed.

Spanner [9] is a globally distributed database that shards data across many sets of Paxos state machines in datacenters spread all over the world. A Spanner instantiation is called a *universe*. A *universe* has *zones*, which are units of physical isolation. Data can be partitioned into zones and moved across them. A zone hosts up to several thousands of *spanservers*. Each spanserver belongs to a replication group, using Paxos, and implements a state machine on top of each *tablet* it hosts. Each spanserver supports between 100 and a 1000 tablets. If a transaction touches tablets within a Paxos group, the spanserver has ordering and locking mechanisms to ensure linearizability. In the case of a distributed transaction (i.e. involving several Paxos groups), there is a leader replica at each group implementing a *transaction manager* to cope with it. These managers coordinate to perform two-phase commit. Commits are assigned globally meaningful timestamps that reflect serialization order, which additionally satisfies external consistency and thus linearizability.

Granola [10] is an architecture to coordinate transactions, allowing to build reliable distributed storage applications and supporting serializability across all operations. The main roles involved are clients and repositories. Clients submit transactions to repositories. Repositories coordinate to ensure strong consistency. Each repository is implemented by a set of replicas using the state machine replication approach. Granola works with one-round transactions, i.e., transactions that do not allow interactions with the client. Transactions are assigned a timestamp at repositories, defining a position in the global serial order. Before committing, repositories ensure that they all agree on the timestamp. There are three types of transactions: (i) single-repository transactions, (ii) independent distributed transactions, and (iii) coordinated distributed transactions. Single repository transactions are similar to single-node storage processing. Independent distributed transactions, a feature introduced by Granola, are used when the distributed transactions can execute independently at repositories (e.g., distributed read, distributed update on replicated data). Using timestamps, they are ordered with respect to other transactions without locking or conflicts. If other transactions may lock items updated by an independent transaction, then a conflict is raised and the transaction has to be tried later by the client. Coordinated transactions require locking to support concurrency and require participants to agree whether to commit or abort the transaction.

Calvin [35] is another transaction coordination protocol developed in parallel with Granola. Calvin provides logical equivalence to a serial order of transactions, i.e., serializability. It is similar in

functionality to [10], but eliminates the need for a commit phase for distributed transactions. Calvin is organized in replicas, which are partitioned. Replicas have the same partitions and each partition runs three layers: (i) sequencer; (ii) scheduler; and (iii) storage. The (i) sequencer builds a global order of transactions being submitted across partitions, organizing them in batches per epoch (i.e., slices of 10 milliseconds) that are sent to the (ii) schedulers of the partitions involved. Calvin transactions declare the read/write items accessed such that the sequencer and the scheduler ensure that transactions follow a deterministic execution that eliminates concurrency conflicts. With this, transactions are executed to completion at the (iii) storage layer (i.e., any storage engine supporting a CRUD interface). A multi-partition transaction executes at all involved partitions: the needed values are read and then exchanged so that writes can be computed. If a node fails, it is assumed a replica node is available. Since node failures are dealt with and there are no concurrency conflicts due to sequencing and scheduling, transactions always commit, eliminating the need for distributed commit protocols. The absence of distributed commit protocols eliminates the need of locks. As a result, the contention footprint is considerably reduced, this being the key aspect to Calvin's scalability.

6 FINAL REMARKS

Partitioned state machine replication extends classic state machine replication (SMR) with the notion of state partitioning (or sharding). In both approaches, clients propagate requests to the replicas, which execute the requests sequentially in a consistent order. In the case of SMR, every request concerns all replicas, as each replica stores the full application state. In partitioned SMR, the application state is divided into partitions, and each request accesses data in one or more partitions. Clients must propagate requests to the partitions concerned by the data accessed in the request.

Many proposals that adopt the SMR model use an atomic broadcast primitive to order requests (e.g., [28, 29]). In the case of partitioned SMR, atomic multicast is more appropriate than atomic broadcast to propagate requests to replicas consistently (e.g., [34]) since propagating all requests to all replicas defeats the purpose of data partitioning. One can observe that as partitioned SMR generalizes classic SMR, atomic multicast generalizes atomic broadcast. Despite this analogy, there is an "asymmetry" in how the ordering abstractions are used in the replication approaches. In SMR, after delivering a request, replicas execute the request and reply to the client. No coordination between replicas is needed as part of the execution of a request. In partitioned SMR, as part of the execution of a multi-partition request, replicas in the involved partitions must coordinate to ensure linearizability. We show in the paper that this coordination is needed because atomic multicast with global total order does not capture real-time dependencies between requests. When equipped with atomic multicast that ensures atomic global order, replica coordination is not necessary.

Finally, we note that atomic global order is not minimal, in that it rules out executions that do ensure linearizability. For example, the execution depicted in Figure 2 (right) while linearizable is not allowed by atomic global order. To see why, notice that since request $r(0,1)$ is delivered at P_0 before $w(1,v_1)$ is multicast by client c, from atomic global order, $r(0,1) \prec w(1,v_1)$. Therefore, at P_1, $w(1,v_1)$ cannot be delivered before $r(0,1)$ (i.e., $w(1,v_1) \prec r(0,1)$) since this would create a cycle. One open question is whether one can come up with a property stronger than global total order (i.e., to prevent replica coordination) but weaker than atomic global order (i.e., to accept linearizable executions currently ruled out by atomic global order).

ACKNOWLEDGMENTS

Fernando Dotti is partially supported by National Council for Scientific and Technological Development, CNPq, Brazil; FAPERGS, RS, Brazil; and PUCRS-PrInt grant by CAPES, Brazil.

REFERENCES

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, and Igor Zablotchi. 2019. The Impact of RDMA on Agreement. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019).
[2] Hagit Attiya and Jennifer Welch. 2004. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons.
[3] Carlos Eduardo Bezerra, Daniel Cason, and Fernando Pedone. 2015. Ridge: high-throughput, low-latency atomic multicast. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 256–265.
[4] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. 2014. Scalable State-Machine Replication. In *DSN*. 331–342.
[5] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.* 5, 1 (1987), 47–76.
[6] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.
[7] Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. 2018. Byzantine fault-tolerant atomic multicast. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 39–50.
[8] Paulo Coelho, Nicolas Schiper, and Fernando Pedone. 2017. Fast Atomic Multicast. In *DSN*.
[9] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
[10] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 223–235. https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling
[11] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.* 36, 4 (2004).
[12] Carole Delporte-Gallet and Hugues Fauconnier. 2000. Fault-Tolerant Genuine Atomic Multicast to Multiple Groups.. In *OPODIS*. Citeseer, 107–122.
[13] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.
[14] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 178–193.
[15] Udo Fritzke, Philippe Ingels, Achour Mostéfaoui, and Michel Raynal. 1998. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 228–234.
[16] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. 2019. White-box atomic multicast. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 176–187.
[17] Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi, and Marko Vukolic. 2019. Can 100 Machines Agree? *CoRR* abs/1911.07966 (2019). arXiv:1911.07966 http://arxiv.org/abs/1911.07966
[18] Rachid Guerraoui and André Schiper. 1997. Software-Based Replication for Fault Tolerance. *Computer* 30, 4 (1997), 68–74.
[19] Rachid Guerraoui and André Schiper. 2001. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.* 254, 1-2 (2001), 297–316.
[20] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report. Ithaca, NY, USA.
[21] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
[22] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.
[23] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo R. Coelho, and Fernando Pedone. 2021. RamCast: RDMA-based atomic multicast. *Proceedings of the 22nd International Middleware Conference* (2021).

[24] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. 2019. DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication. In *ICDCS*.

[25] Zhongmiao Li, Peter Van Roy, and Paolo Romano. 2017. Enhancing throughput of partially replicated state machines via multi-partition operation scheduling. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*. 1–10. https://doi.org/10.1109/NCA.2017.8171364

[26] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2012. Multi-Ring Paxos. In *International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 1–12.

[27] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. 2010. Ring Paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN)*. IEEE, 527–536.

[28] Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. 2005. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst.* 23, 4 (nov 2005), 375–423.

[29] Fernando Pedone, Rachid Guerraoui, and André Schiper. 2003. The Database State Machine Approach. *Distributed Parallel Databases* 14, 1 (2003), 71–98.

[30] Luis Rodrigues, Rachid Guerraoui, and André Schiper. 1998. Scalable atomic multicast. In *International Conference on Computer Communications and Networks*. 840–847.

[31] Nicolas Schiper. 2009. *On Multicast Primitives in Large Networks and Partial Replication Protocols*. Ph. D. Dissertation. Università della Svizzera italiana.

[32] Nicolas Schiper and Fernando Pedone. 2008. On the inherent cost of atomic broadcast and multicast in wide area networks. In *International Conference on Distributed Computing and Networking (ICDCN)*. Springer, 147–157.

[33] Nicolas Schiper and Fernando Pedone. 2008. Solving Atomic Multicast When Groups Crash. In *OPODIS*, Theodore P. Baker, Alain Bui, and Sébastien Tixeuil (Eds.).

[34] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. 2010. P-Store: Genuine Partial Replication in Wide Area Networks. In *Symposium on Reliable Distributed Systems (SRDS)*.

[35] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*.