Enhancing Throughput of Partially Replicated State Machines via Multi-Partition Operation Scheduling

Zhongmiao Li[†⋆], Peter Van Roy[†] and Paolo Romano[⋆]
[†]Université catholique de Louvain [⋆]Instituto Superior Técnico, Lisboa & INESC-ID

***Abstract*—State-machine replication (SMR) is a fundamental technique to implement fault-tolerant services. Recently, various works have aimed at enhancing the scalability of SMR by exploiting partial replication techniques. By sharding the state machine across disjoint partitions, and replicating each partition over independent groups of processes, a Partially Replicated State Machine (PRSM) can process operations that involve a single partition by only requiring synchronization among the replicas of that partition — achieving higher scalability than SMR.**

Unfortunately, though, existing PRSM rely on inefficient mechanisms to coordinate the execution of multi-partition operations, which either impose global coordination across all nodes in the system or require inter-partition synchronization on the critical path of execution of operations. As such, performance and scalability of existing PRSM systems is severely hindered in the presence of even a small fraction of multi-partition operations.

This paper tackles this issue by presenting Genepi, a PRSM protocol that introduces a novel, highly efficient mechanism for regulating the execution of multi-partition operations.

We show via an experimental evaluation based on both synthetic benchmarks and TPC-C that Genepi can achieve up to 5.5× of throughput gain over existing PRSM systems, with only negligible latency overhead at low load.

I. Introduction

State-machine replication (SMR) is a well-known approach to implement fault-tolerant services. In a nutshell, SMR relies on consensus protocols [1] for replicas to agree, in a fault-tolerant way, on the set and common order of operations to be executed. Next, operations can be executed independently at each replica. As long as operations are deterministic (which is required by SMR), replicas will traverse the same sequence of states and thus reach the same final state. Nevertheless, a main limitation of classical SMR is that, it requires all replicas to host the full state and execute all operations that may alter the system's state. Consequently, the maximum rate at which update operations can be processed in SMR-based systems is limited by the computation capacity of a single replica, which is inherently non-scalable.

In order to tackle this scalability limitation, several recent works [2], [3], [4] have proposed to extend the basic SMR model to support *partial replication*, yielding what we call a Partially Replicated State Machine (PRSM). The idea is to shard the system's state across multiple partitions, and to replicate each partition over a different group of nodes. This allows to execute single partition operations (SPOs), i.e., operations that manipulate the state residing in a single partition, in an efficient and scalable way, namely by involving exclusively the replicas of that partition. Operations that span multiple partitions, or, more briefly, multi-partition operations (MPOs), though require additional synchronization: the replicas of all the target partitions of an MPO need in fact to establish a common agreement on how to order that MPO with respect to all the other operations that access these partitions. Unfortunately, the additional synchronization schemes required by current PRSM protocols [4], [2], [3] incur significant overhead, which can cripple throughput even in workloads that contain a very limited number of MPOs: as we will show, the peak throughput of a popular PRSM system [3] drops by about 80% in presence of even just 10% MPOs (§VI-E).

The reason underlying such a harsh performance impact is that existing PRSM schemes regulate the execution of MPOs either i) via non-scalable coordination mechanisms that involve all the nodes in the system (including nodes not involved by any MPO) [4], [2], whose overheads become unbearable in large scale systems, or ii) by imposing additional inter-partition synchronization along the critical path of MPOs' processing [3], thus inherently limiting the peak throughput achieved by each partition.

This paper proposes Genepi, a novel PRSM protocol that tackles the above mentioned limitations of existing systems by introducing an innovative consensus primitive. The design of Genepi is based on two key ideas:

- i) removing the inter-partition coordination required by MPOs from the critical path of execution of *any* operation in the system: in a nutshell, this is achieved by scheduling the execution of MPOs "in the future", i.e., postponing their execution by a duration comparable to the time necessary to execute, in background, an inter-partition protocol aimed at agreeing on a common execution order. While this latency is normally negligible in terms of user-perceived latency, the removal of any inter-partition synchronization from the critical path of execution of operations at each node has a profound (beneficial) impact on throughput.
- ii) maximizing the scalability of the MPO synchroniza-

This work is partially funded by the H2020 project 732505 LightKone, by FCT via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014 and by EACEA award 2012-0030. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

978-1-5386-1465-5/17/$31.00

tion mechanism: Genepi pursues this goal by introducing Scraper (Scalable consensus for partial replication), a novel, fault-tolerant distributed agreement scheme, inspired by Skeen's total order multicast algorithm [5]. Analogously to Skeen's algorithm, Scraper ensures a property that is crucial for scalability: a partition is involved in Scraper algorithm for an MPO only if it is actually involved by that partition — a property called minimality or also genuineness in the context of total order multicast algorithms [6].

We evaluate our protocol via both synthetic benchmarks and TPC-C. The experimental evaluation shows that Genepi can achieve up to 5.5× of throughput gain over existing PRSM systems, with negligible latency overhead at low load.

The remainder of the paper is organized as follows. §II reports the related work of Genepi. §III describes our system and data model. §IV and §V detail the Genepi protocol as well as the Scraper consensus abstraction. We describe the experimental evaluation of Genepi in §VI, and conclude the paper in §VII.

II. Related Work

a) State-machine replication: State-machine replication is a well-known technique to build fault-tolerant systems [7], [8], [9]. In its original definition, SMR assumes that an application, abstracted as a deterministic state machine, is fully replicated across all the nodes of the system. Replicas first agree on a common, total order of execution of operations. Next, operations are executed by all replicas according to the agreed order. Various work has been proposed to optimize the ordering phase or processing phase of SMR. SMR's ordering phase relies on consensus protocols, e.g., Paxos [1], another fundamental, long studied problem, for which a plethora of optimizations have been proposed in the literature. Some works [10], [11], [12] propose to only totally-order conflicting requests and avoid synchronization costs for commutative operations. Other works [13], [14] optimistically deliver and execute operations before their order is finalized, such that latency is reduced if the final order matches the optimistically guessed order. These optimizations tackle orthogonal problems, and could indeed be integrated with Genepi — which has in consensus/total order a key building block. Other works aimed at enhancing the efficiency with which replicas exploit local computational resource, e.g., multi-cores processors. In [8], the authors propose to organize replicas in multiple modules executing independent tasks, such as receiving and batching messages. Although operations are still sequentially executed by a single thread, this approach enhances throughput by parallelizing the execution of all other tasks. Instead, [9] exploits application semantics to allow executing commutative operations in parallel. These works also tackle distinct problems from those addressed by Genepi, and the mechanisms they propose could be integrated with it.

b) Partially-replicated state machine: Recently, several works [2], [3], [4] have explored the idea of building a PRSM. By partitioning the state of an application, these approaches allow SPOs to be ordered and executed only by the replicas of the target partition. As the number of replicas of each partition is typically much smaller than the total number of partitions in the system (a few copies vs hundreds or thousands of partitions), PRSM has a much higher scalability potential than plain SMR. Nonetheless, as already mentioned, existing PRSM systems incur severe overheads when processing operations that span multiple partitions (MPOs).

The PRSM system proposed by Marandi et al. [2], for instance, requires that every MPO is totally ordered and then broadcast to all the nodes in the system, independently of whether they are actually involved in the processing of MPOs. This global-scale coordination mechanisms are inherently non-scalable and, as such, are at odds with one of the key motivations at the basis of PRSM: overcome the scalability limitations of classic SMR.

Analogous considerations apply to Calvin [4]: in this case, the execution of MPOs is synchronized via a, which we call, scatter phase that requires all the partitions within each logical replica of the system to agree on the set of MPOs to include in the next batch of operations to be processed. In large scale systems, Calvin's scatter phase introduces significant overhead, forcing partitions to synchronize even though there exists no MPO that involves them. Further, the latency introduced by the scatter phase sits on the critical path of execution of the current batch of operations to be processed, which limits the rate at which globally operations can be processed in the system.

S-SMR [3] builds on a more scalable coordination primitive, atomic multicast [15], [5], to regulate the execution of both SPOs and MPOs. However, it imposes the exchange of signaling messages among all the partitions involved by an MPO m, blocking the execution of any operation serialized after m until the signaling phase for m is completed. Also in this case, the inter-partition synchronization lies on the critical path of execution of operations, limiting throughput. Further, the need for using of atomic multicast for both SPOs and MPOs imposes a non-negligible overhead with respect to the case of plain SMR: in fact, atomic multicast is known to be more expensive than consensus, and SMR can order all operations using solely consensus.

Unlike all the above systems, Genepi executes any MPO synchronization activity as a background process, fully overlapped with the processing of operations. Further, Genepi synchronizes MPOs via a highly scalable consensus protocol, which requires the participation exclusively of the partitions that are actually involved in the processing of MPOs. Finally, Genepi's design ensures that SPOs can be executed after having been ordered via a plain consensus protocol involving solely the replicas of their target partition, restricting the use of additional distributed coordination schemes only to MPOs.

c) Atomic multicast protocols: The MPO synchronization scheme employed by Genepi, which we called Scraper, is inspired by Skeen's total order multicast algorithm and to the subsequent, fault-tolerant versions that were proposed in the literature [15]. Similarly to Skeen's algorithm, Scraper aims to ensure that if two processes/partitions are involved

by the recipients of two messages/MPOs, say $m1, m2$, both processes/partitions deliver/execute them in the same order. Unlike Skeen's algorithm, though, Scraper allows processes to establish a lower bound on when a given MPO should be processed — a property that is key to support Genepi's idea of scheduling the execution of MPOs in the future, in order to overlap their synchronization and operation processing.

III. System and data model

We consider a distributed system consisting of a set of interconnected processes $\mathcal{S} = \{s_1, ..., s_n\}$. A process may fail by crashing, but it does not experience arbitrary faults, i.e., Byzantine faults do not occur. Set $\mathcal{S}$ is logically subdivided into P disjoint groups of processes, $\mathcal{S}_1$, ..., $\mathcal{S}_p$, where P is the number of data shards, or *partitions*, of the whole data set. All processes of a group replicate the same partition, and we assume partitions to be disjoint, namely, the intersection of data items in any pair of partitions is the empty set.

Clients interact with the system by submitting *operations*, along with any necessary input parameter. Each operation is a sequence of actions, which either reads or writes a data item or performs deterministic computations. We say that a partition is involved by an operation, if the operation reads or writes any data item belonging to that partition. If an operation only involves a single partition during its execution, it is called as a single-partition operation (SPO); respectively, an operation that involves more than one partition is named as an multi-partition operation (MPO). If an MPO's involved partitions need to exchange data to execute it, we name this type of MPOs as *dependent MPOs*; otherwise, we call it as *independent MPOs*. An example of the former is an MPO that reads two data items of two partitions, and sets the values of both to the sum of both; the latter can be an MPO that blindly updates data items of multiple partitions. As in typical PRSM systems [4], [3], we assume having an approximation about the set, possibly a superset, of the partitions that an operation will involve. Though, we do not assume knowing a-priori the exact data items to be accessed.

Finally, Genepi targets linearizability. A system is linearizable if there is a way to reorder the client operations in a sequence that (i) respects the semantics of the operations, as defined in their sequential specifications, and (ii) respects the real-time ordering of commands across all clients [16], i.e., if an operation completes before the invocation of another operation in real time, then the second operation should be ordered after the first one.

Communication primitives A process relies on the use of multi-instance consensus to reliably replicate states within its replication group. The consensus is defined by the primitives C-PROPOSE(v, r) and C-DECIDE(v, r), where v is an arbitrary value and r is an instance number. Consensus guarantees that (i) if a process decides v then some process proposed v *(uniform integrity)*; (ii) no two processes decide different values *(uniform agreement)*; and (iii) if one or more correct processes propose a value then eventually some value is decided by all correct processes *(termination)*. We assume that the system provides sufficient guarantees to implement consensus, e.g., an asynchronous system equipped with an eventually perfect failure detector oracle and in which a majority of process in each group is correct (does not crash).

One-to-one communication uses primitives SEND(p, m) and RECEIVE(m), where m is a message and p is the process m is addressed to. If sender and receiver are correct, then every message sent is eventually received. Further, a process can address a message to processes of a group, using reliable multicast primitives. Processes use R-MULTICAST(g, m) to reliably send a message m to all processes in group g, and message m is delivered at the destinations with R-DELIVER(m). Reliable multicast has the following properties: (i) if a correct process reliable-multicasts m to g, then every correct process of group g reliable-delivers m *(validity)*; (ii) If a correct process reliable-delivers m, then every correct process in its replication group reliable-delivers m *(agreement)*; (iii) for any message m, every process p in g reliable-delivers m at most once, and only if some process of its group has reliable-multicast m previously *(integrity)*.

IV. Genepi

Like in classic, i.e., fully replicated, SMR, Genepi operates according to an *order-then-execute* paradigm: Genepi operates in rounds, and, in each round, processes first execute an agreement protocol to agree on which operations shall be included in that round; then operations are actually executed.

However, since Genepi targets a partial replication model, it departs from the classic SMR approach in one fundamental aspect. In order to take full advantage of the scalability potential of partial replication, Genepi ensures a key property, which goes under the name of *genuineness* [17], [18], [19] (or *minimality* [6]) in the literature: Genepi involves a process in the ordering and execution phases of an operation only if that operation requires accessing any of the data partitions maintained by that process.

Genepi also departs from existing PRSM for what concerns the management of MPOs. As already discussed, in fact, coordinating the execution of MPOs is inherently more complex and expensive than for the case of SPOs and the mechanisms used by existing PRSM solutions to regulate the execution of MPO impose additional latency on the critical path of execution not only of MPOs but also of SPOs.

In order to avoid this issue, Genepi exploits a simple, but effective idea: scheduling the execution of SPOs in the present execution round, while postponing the execution of MPOs to future execution rounds. By moving the execution of MPOs to future execution rounds, Genepi trades off the latency perceived by MPOs, in order to enhance the global system throughput in two ways: i) by decoupling the execution of SPOs and MPOs, to ensure that the slower ordering phase incurred by MPOs does not hinder the execution of SPOs; ii) by overlapping the ordering phase of MPOs with the execution of other operations (both SPOs and previously received MPOs), to minimize the chance that a replica has to

halt processing operations as it is still waiting for completing the ordering phase of some MPOs.

Genepi pursues these goals by leveraging a new abstraction, which we called Scraper. Scraper exposes a consensus-like interface that allows replicas to specify lower bounds on the rounds in which MPOs and SPOs are to be scheduled and to reach agreement on which operations each round should ultimately include. In fact, the Scraper abstraction allows for greatly simplifying the design of Genepi, by decoupling, in a modular way, the problem of estimating the earliest round in which MPOs should be scheduled from the problem of how to design a scalable and fault-tolerant consensus protocol (whose complexity is encapsulated by Scraper).

In the following, we first detail the specification of the Scraper abstraction (§IV-A) and how it is used by Genepi to implement a scalable PRSM (§IV-B). Next, we present a protocol that implements the Scraper abstraction (§IV-C).

A. Scraper specification

Scraper provides a consensus-like interface designed to meet the needs of PRSMs and exposes two primitives:

- S-PROPOSE$(SPOs, r^{SPO}, MPOs, r^{MPO})$ allows processes to input a set of SPOs and a set of MPOs, and the corresponding rounds in which they propose to execute such operations.
- S-DECIDE$(OPs, round)$ notifies which set of operations, OPs, are to be included in a given $round$.

The properties guaranteed by Scraper are specified below:

- *Operation-Integrity*: if an operation op is decided, it must have been proposed. Also, a process p decides an operation op at most once and only if p is involved by op.
- *Round-Integrity*: a process decides for a round at most once, and only if some process has proposed for that round.
- *Uniform agreement*: no two processes decide the same operation in different rounds.
- *Monotonicity*: if an operation is proposed in round r, it can only be decided in a round r' s.t. $r' >= r$
- *Minimality*: if a correct process p sends or receives a message in order to decide on an operation op, then p is involved by op.
- *Round termination*: if a correct process proposes for a given round, it eventually decides for that round.
- *Operation termination*: if a correct process proposes an operation op, then op is eventually decided by all the correct processes involved by op.

We note that the above specification defines an abstraction that stands at a middle ground between consensus and atomic multicast.

On the one hand, the propose/decide interfaces exposed by Scraper resemble the ones offered by multi-instance consensus protocols (like Multi-paxos [20]); further, the *Uniform agreement*, *Round termination* and Operation termination properties embed typical safety and liveness properties of multi-instance consensus protocols.

On the other hand, analogously to atomic multicast, Scraper ensures that a process decides an operation only if that operation involves the manipulation of a data partition hosted by that process (*Integrity*). In fact, Scraper reinforces this property via the *Minimality* guarantee, which prevents processes not involved in an operation to participate at all, by exchanging any message, in the decision process for that operation. Scraper's minimality property is, in fact, an adaptation of an homonymous property defined by Guerraoui and Schiper [6] for atomic multicast, which is also sometimes referred to as genuineness in partial replication schemes, and is crucial to materialize the scalability potential of partially replicated systems.

Overall, differently from Consensus, Scraper's abstraction targets a partially replicated system and, as such, allows different processes (replicating different partitions) to decide different operations in a given round. Unlike Atomic multicast, though, it allows processes to exert control on the order with which different operations (i.e., SPOs and MPOs) should be decided — whereas Atomic multicast provides no means for an application to intentionally postpone the delivery of specific messages.

A final note for the *Monotonicity* property, which guarantees that operations are decided either in the round in which they were proposed, or in future rounds. This property allows processes that have already decided for round r, and that receive (e.g., due to system's asynchrony) proposals for including some operation op in a round $r' \leq r$, to include op in a round $r'' > r$. Such a flexibility is necessary to ensure that the decision on which operations to include in a round can be achieved without having necessarily heard from all the processes in the system — a crucial property not only for scalability purposes, but also for fault-tolerance.

B. Genepi's execution

Algorithm 1 reports the pseudo-code of the Genepi protocol. Processes periodically batch, each α time units, (e.g., typical α values are 5-10 msecs) the SPOs and MPOs received from local clients[1] and invoke the S-PROPOSE primitive to schedule the SPOs' execution in the current round (identified by $prop\_round$ in the pseudo-code) and MPO's execution in the future by δ rounds. The rationale underlying this mechanism is the following: reaching agreement on SPOs involves solely an intra-group synchronization, which is faster than the inter-group synchronization required by MPOs. Thus, on the one hand, scheduling SPOs for execution in the current round is optimal for latency; on the other hand, by scheduling MPOs for a "sufficiently-large" future round, Genepi can effectively remove from the critical path of execution the latency of the MPOs' synchronization: in fact, provided that the MPOs' synchronization takes less than $\delta \cdot \alpha$, the synchronization for the MPOs scheduled for round $r+\delta$ (which can run in parallel with processing of rounds $r, r+1, \ldots, r+\delta$) will be already

[1] If a process receives an operation that does not involve its local partition, it asynchronously sends this operation to any of its involved partition to be batched.

determined by the time round $r + \delta$ starts.We will further discuss how to choose a suitable value of δ in §V-A.

Next, processes wait for operations for the current round to be agreed upon via the S-DECIDE (line 13), and deterministically order them. This aims to ensure that MPOs in the same round will be executed by all involved partitions in the same order. We assume a single threaded-execution model[2], in which processes execute operations sequentially: if an action of an operation is a deterministic computation, a process directly executes it; if an action is an update, a process only applies the update if this action updates a local key; read actions, though, may require communication among partitions. For instance, an operation may read two data items in two partitions, and set the values of both to their sum, which requires cross-partition communication. To handle this case, when a partition of an MPO reads a local data item, it multicasts this data item to all other involved partitions; accordingly, if a partition reads a non-local data item, it waits until it receives the needed data item from some other partitions.

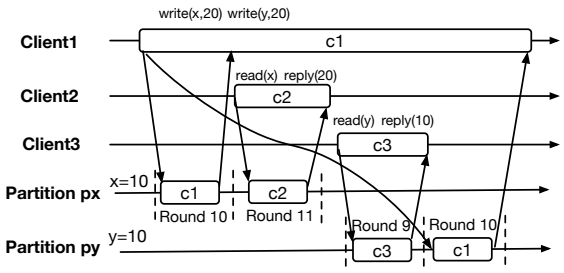
In order to ensure linearizability, Genepi relies on an additional mechanism, which aims at synchronizing the execution of independent operations, which, we recall, can be processed independently at any process. Fig. 1a illustrates the anomaly that would arise if independent operations were allowed to externalize a reply as soon as they had completed processing at a process. In this example, $c2$ completes before $c3$ starts, so $c2$ should be ordered before $c3$. However, $c2$ reads from $c1$ and $c3$ misses the update of $c1$.

Genepi tackles this problem via the following signaling mechanism: when a process starts executing an independent MPO it notifies all the involved partitions; when it completes processing an independent MPO, it does not immediately externalize the corresponding reply to the client, but waits until the operation has been signaled by all the involved partitions. Also, any operation op ordered after an independent MPO mpo can be immediately processed, but its reply must also be postponed until the reply of mpo has been externalized to its client. This is achieved by inserting all the replies in a queue ($reply\_queue$), and delivering a reply only when the corresponding operation is the first in queue and, if the operation is an independent MPO and it has already been signaled by every partition (lines 13-29).

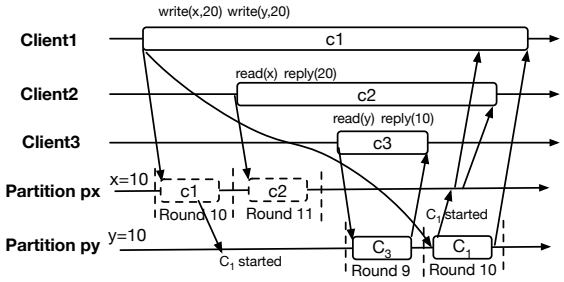
Referring to the example in Fig. 1a, this mechanism ensures that px notifies the clients of $c1$ and $c2$ only after having received the signal for $c1$ by py. This way, $c2$ and $c3$ overlap their execution in real-time, becoming concurrent operations and safeguarding linearizability.

We name this technique "delayed reply", as it delays the notification of replies to clients, without, however, blocking

[2]This is done to simplify presentation. Supporting multi-threaded execution in Genepi would require adopting a deterministic concurrency control scheme in order to ensure that processes serialize operations in an order compliant with the one established by Genepi. Several solutions exist in the literature, e.g., [21], [4], [14], which could be integrated with Genepi to relax this assumption.



(a) Linearizability violation: $c2$ reads from $c1$, $c3$ misses updates of $c1$, but $c3$ starts after $c2$ completes.



(b) Delayed reply: px executes $c1$ and $c2$, but only replies to clients when $c1$ has been started everywhere.

Fig. 1: Linearizability violation and delayed reply.

processing of operations — unlike the signaling scheme used in S-SMR [3] (see §II and §VI). We note that dependent MPOs do not need to explicitly send signal among partitions (line 15): partitions need to exchange data when executing a dependent MPO, which already achieves the effect of signaling.

C. Scraper

Algorithm 2 reports the pseudo-code of the proposed implementation of Scraper. To simplify presentation, the pseudo-code relies on the assumption (met by Genepi) that if a client invokes the S-PROPOSE primitive N times, it passes as input parameters for r^{SPO} and r^{MPO} the values [0,1,...,N] [δ,1+δ,...,N+δ], respectively.

Upon the invocation of S-PROPOSE at a process p, p executes an instance of an intra-group consensus aimed to i) exchange (and merge) the SPOs and MPOs locally input via S-PROPOSE at each process in the group, and ii) make sure that the process group agrees, in a fault-tolerant way, on the resulting set of merged operations. More in detail, when S-PROPOSE is invoked with r_l^{SPO} as input parameter, p uses a consensus instance that is uniquely identified by the tuple $\langle \circ, r_l^{SPO} \rangle$, where $\circ$ denotes the sequence of consensus instances used to replicate intra-group operations. Note that we are assuming that the consensus primitive, in this case, merges the sets of SPOs and MPOs proposed by each process and that, in case processes propose different values for r_l^{MPO} (the round in which to schedule their MPOs), consensus outputs the maximum value among all the proposals that it includes in its C-DECIDE event. We do not explicitly model this in the pseudo-code to simplify presentation.

Algorithm 1: Genepi execution

```
1  Algorithm variables
2  prop_round: the next round a leader should propose.
3  exec_round: the next round a process should execute.
4  reply_queue: a FIFO queue storing pending operations to be replied.
     Each entry is in the form of {operation, signals_to_receive}.
5  Initialization:
6       exec_round ← 0; prop_round ← 0
7       initialize reply_queue as an empty queue
8  // Batch operations and Scraper-propose them
9  upon new batch of ⟨SPOs,MPOs⟩ ready
10      S-PROPOSE(prop_round, SPOs, prop_round+δ, MPOs)
11      prop_round ← prop_round + 1
12 // Execute decided operations
13 upon S-DECIDE(round, decided_ops) and round = exec_round
14      for op ∈ sort(decided_ops)
15          if op.is_mpo() and op.is_independent()
16              for p ∈ op.partitions
17                  R-MULTICAST(p, {signal, op})
18          process(op)
19          reply_queue.append(op, op.partitions - 1)
20      exec_round ← exec_round + 1
21 upon deliver({signal, op})
22      //reduce by one the signal counter of op in reply_queue
23      reply_queue.getOp(op).signal()
24      while !reply_queue.empty()
25          {check_op, remain_signal} ← reply_queue.head()
26          if remain_signal = 0
27              reply(check_op)
28              reply_queue.dequeue()
29          else break
```

Next, p checks whether any of the operations that were locally input via S-PROPOSE were omitted in the set of operations decided for the corresponding consensus instance. This could happen, e.g., if p was suspected to have crashed and the consensus implementation timed out waiting for p's proposals. In this case, p must S-PROPOSE again the missing operations in some future round, in order to guarantee that these are eventually decided by Scraper. To this end, p determines (e.g., by broadcasting a query within the process group, which we omit in the pseudo-code for simplicity) what is the smallest consensus instance of type $\langle \mathtt{O}, c\rangle$ for which its group has not yet run consensus, and S-PROPOSE the missing SPOs/MPOs for round c/$c+\delta$, respectively.

At this point the group has already established which SPOs will be included in round r_l^{SPO}. However, it is still necessary to ensure that all the process groups involved in the locally received MPOs agree on the round in which these MPOs should be included. This responsibility is taken on by the group leader, which coordinates an inter-partition agreement algorithm that is similar in spirit to a fault-tolerant version of the Skeen's multicast algorithm presented in [6], [5] and works as follows:

1) The group leader sends a `request` message to the leaders of all the partitions involved by the MPOs gathered by its group. These messages are tagged with a unique identifier[3] and specifying the target scheduling round for its MPOs (r^{MPO}). Note that to ensure *minimality*, MPOs are only sent to their involved partitions (lines 17-19).

2) Group leaders wait for a small timeout value, β[4], to gather `request` messages from other partitions' leaders and store them in the $reqs$ set. Next, before replying to the received `request` messages, each leader replicates within its group, using consensus instance $\langle R, r_l^{SPO}\rangle$, the received requests along with r^{MPO}, namely the round in which its group had previously agreed, via consensus instance $\langle O, r_l^{SPO}\rangle$, to order its own MPOs. This step is done for fault-tolerance, as it ensures that if the leader is suspected (possibly falsely) to have crashed, any other process attempting to fail-over will deterministically behave like the original leader would in the following step (lines 20-24).

3) Next, leaders reply to the received `request` messages, sending back a `vote` message specifying as $vote\_round$ the maximum between the value proposed by the sender of the request message and r^{MPO}. As it will be clearer shortly, this is aimed to ensure that if a process has already triggered an S-DECIDE for a round r, it will necessarily specify a $vote\_round$ that is larger than r in any `vote` message it may ever send. For each `vote` message that a leader sends, it also adds a corresponding entry in a local queue, opQ. More in detail, in this queue it adds a tuple containing the unique request identifier, the corresponding $vote\_round$ and operations and a state value set to $prepared$ (`P`) (lines 25-28).

4) Each leader gathers `vote` messages from all the involved partitions and computes the maximum among all the received round values ($final\_round$, line 30): this will be the identifier of the round in which these MPOs will be eventually S-decided. Next, the coordinator notifies the involved partitions by sending back a `decision` message.

5) When a leader receives a `decision` message, it updates the round number of the corresponding entry in opQ and sets its state from pending (`P`) to decided (`D`, line 34).

6) The MPOs to be included in a given round r^D are determined by the leader of each group, when the following two conditions are met: i) the SPOs for the round r^D have already been determined (line 36) and ii) in the queue there are either no entries for round r^D (meaning that this round will not include any MPO, lines 38-42) or that all the entries for r^D are in a decided state (and in this case these entries will contain all the MPOs to be decided in round r^D by this process group, lines 43-45). Note that the first condition implies that the process group has already agreed, via consensus instance $\langle \mathtt{O}, r^D\rangle$, to reply to any future incoming `request` messages with a vote_round=$r^{MPO} > r^D$[5]. Hence, once this first condition is met, it is impossible that any process in the group will ever include in opQ any entry with a round smaller than or

[3]These are obtained via the `getID`(roundID) primitive, which ensures that any process in the same group will deterministically obtain the same unique identifier if it passes as input parameter the same rounID value.

[4]β should be chosen slightly larger than the inter-group communication latency, to allow the leaders of different partitions to mutually exchange their `request` messages.

[5]Recall that we are assuming that when applications invoke S-PROPOSE, they always request to schedule MPOs in a future round w.r.t. SPOs, i.e., $r_l^{MPO} > r_l^{SPO}$.

Algorithm 2: Scraper execution

```
1  Algorithm variables
2  r^D: the next round to decide
3  round_spos: a map containing SPOs for each round
4  opQ: a queue storing pending/decided (P/D) MPOs, with entries in the
     form of < id, round_number, operations, state >. Entries are in
     ascending order by their round_number.
5  Initialization:
6      round^D ←0
7      initialize opQ as an empty queue, round_spos as empty map
8  upon S-PROPOSE(SPOs_l, r_l^SPO, MPOs_l, r_l^MPO)
9      C-PROPOSE([SPOs_l, MPOs_l, r_l^MPO], ⟨O, r_l^SPO⟩)
10     wait C-DECIDE([SPOs, MPOs, r^MPO], ⟨O, r_l^SPO⟩)
11     if any proposed SPOs or MPOs is not included in C-Decide
12         discover what is smallest consensus instance, ⟨O, c⟩, that is
13         still undecided by the process group
14         S-PROPOSE([missedSPOs, c, missedMPOs, c + δ], c)
15     round_spos.put(r_l^SPO, SPOs)
16     if process is leader
17         for p ∈ MPOs.partitions()
18             OPs ← {ops∈MPOs∧p ∈MPOs.partitions()}
19             send(p.leader, [request, getID(r_l^SPO), r^MPO, OPs])
20         wait for a timeout β to gather incoming request messages
21         buffer the received request messages in the set reqs
22         // replicate the received requests within the group
23         C-PROPOSE([reqs], ⟨R, r_l^SPO⟩)
24         wait C-DECIDE([reqs'],⟨R, r_l^SPO⟩)
25         for m=[req_id, min_round, OPs] in reqs'
26             vote_round ← max(min_round, r^MPO)
27             opQ.add(red_id,vote_round, OPs, P)
28             send(m.sender(), [vote, req_id, vote_round])
29 upon receive [vote, req_id, round] from all involved partitions
30     final_round ← max. round received from any partition
31     for p ∈ all involved partitions
32         send(p.leader, [decision, req_id, final_round])
33 upon receive [decision, req_id, final_round]
34     opQ.updateById(req_id, final_round, D)
35 // SPOs for r^D have been already decided
36 upon round_spos.get(r^D)≠ ⊥∧ process p is leader
37     if opQ.first.round=r^D
38         // wait until all pending MPOs for r^D have been decided
39         wait (∀op∈ opQ.getByRound(r^D):op.state= D)
40         decided_mpos ← Union of all MPOs in opQ for round r^D
41         R-MULTICAST(p.partition, [mpos, r^D, decided_mpos])
42         opQ.removeAllEntriesByRound(r^D)
43     else // opQ.first.round>r^D
44         // No MPOs are to be included in this round
45         R-MULTICAST(p.partition, [mpos, r^D, ∅])
46     r^D ← r^D+1
47 upon R-DELIVER([mpos, round, MPOs]) ∧ round_spos.get(round)≠ ⊥
48     trigger S-DECIDE(i, round, round_spos.get(round) ∪ MPOs)
```

equal to r^D. As a consequence, as soon as the second condition becomes true, the set of decided entries for r^D in opQ can no longer vary, and can be safely delivered.

7) As a final step, when a leader determines the set of MPOs to be included in round r^D, it informs via a reliable multicast primitive all the members of its groups via a `mpos` message. This ensures that eventually all processes in the same group will trigger a S-DECIDE event for round r^D.

V. Discussion

This section discusses how far in the future should MPOs be scheduled to maximize throughput, without unnecessarily hindering latency (§V-A) and the fault-tolerance of Genepi.

A. Choosing δ

In §IV-B we anticipated that, in order to effectively remove the latency of the inter-group synchronization required by MPOs (l_{MPO}) from the critical path of execution of the system, the MPOs gathered at round r should be scheduled in a future round $r + \delta$ such that $\delta \cdot \alpha > l_{MPO}$ (recall that α is the batching period in Genepi). This condition, in fact, ensures that by the time round $r + \delta$ starts, the MPOs to be included in that round will have already been determined.

By analyzing the Scraper algorithm proposed in the previous section, we can now define more accurate indications on how δ should be tuned. Let us denote the average one-hop inter-group communication latency as l_i, the latency to run consensus within a group as l_c, and the latency of an intra-group reliable multicast as l_{rm}. The expected latency to complete the synchronization phase for MPOs of Algorithm 2, can then be expressed as:

$$l_{MPO} = 3 * l_i + 2 \cdot l_c + \beta + l_{rm}$$

which accounts for the latency of 3 inter-group communication steps (to send `request`, `vote` and `decision` messages), two intra-group consensus instances (Alg.2, lines 9-10 and 23-24), the timeout β (Alg. 2, line 20) and the final reliable broadcast.

In order to avoid imposing any delay to S-DECIDE for a given round r (Alg. 2, line 47), it is actually sufficient that the MPOs synchronization for round r ends by the time the SPO synchronization for round r is completed, which can be estimated as the time to execute an intra-group consensus (l_c). Keeping this into account we have: $l_{MPO} > \delta \cdot \alpha + l_c$.

Overall, an appropriate value for δ should satisfy:

$$(\delta - 1) * \alpha \leq 3 * l_i + \beta + l_c + l_{rm} \leq \delta * \alpha \qquad (1)$$

B. Fault-tolerance

The use of consensus in Genepi ensures that each group acts as a single, highly-available logical process. Indeed, to simplify reasoning on fault-tolerance, it is possible to see the two consensus instances $\langle \mathtt{O}, c\rangle$, $\langle \mathtt{R}, c\rangle$ used in each round c to replicate, respectively, batched operations and voted requests, as two *write-once registers* (wo-registers) [22]. As its name suggests, a wo-register can only be written once even if multiple processes concurrently try to write to it, and all subsequent writes return the first written value; reads to an empty wo-register returns the initial empty value, while reading a written wo-register returns the last written value.

In case the current leader is suspected to have failed, a new group leader can be elected to perform fail-over. The newly elected leader can recover the state of batched operations and voted requests via the corresponding wo-registers, and accordingly initiate ordering requests for undecided batched operations and/or reply to votes that have been replicated but not yet replied. This is safe even in case the original leader was actually correct, i.e., it was subject of a false failure

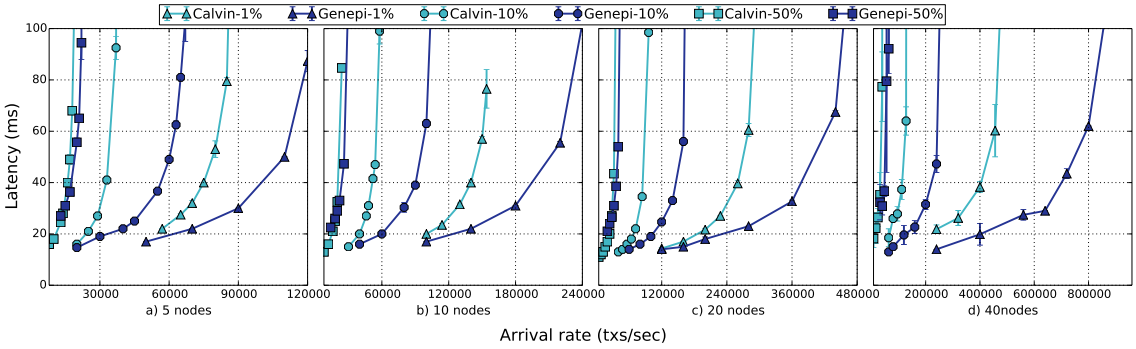


Fig. 2: Comparing Genepi and Calvin while varying the scale of the system and the MPOs (Micro benchmark).

suspicion. In fact, even in the presence of multiple concurrent leaders, the wo-register abstraction guarantees that they will necessarily synchronize their state before externalizing any output. The only issue that has to be accounted for, in such a scenario, is that multiple leaders may concurrently initiate the ordering requests of the same operations. However, as each ordering request is tagged by a deterministic unique id (§IV-C), duplicate requests can easily be filtered out, thus making the ordering process idempotent.

VI. Evaluation

This section reports the results of an experimental study aimed at quantifying the relative performance achievable by Genepi when compared to state of art PRSM protocols. We start by describing the implementation of the various protocols we evaluate, then we detail the experimental setup. Finally, we present evaluation results.

A. Implementation

In this study, we shall consider three protocols: Calvin [4], Genepi and a variant of Genepi, which we called Genepi-DE (*delayed execution*).

For Calvin we used the publicly available open source C++ implementation[6], which we used also as starting point to develop Genepi and Scraper. Genepi-DE allows us to compare the performance of Genepi's *delayed reply* signaling mechanism and of the signaling mechanism proposed in S-SMR [3], which we call *delayed execution*. As the name suggests, the S-SMR signaling scheme prevents new operations to be processed, while there is still some pending MPO — where Genepi's signaling mechanism only postpones the notification of replies to clients, but never prevents process from processing operations. The choice of implementing S-SMR's signaling scheme on top of Genepi allows us to compare the synchronization mechanisms of Genepi and S-SMR in a fairer, and more focused, way, since: i) Genepi, just like Calvin, is implemented in C++, whereas the only available implementation of S-SMR is in Java; ii) S-SMR's implementation relies on an implementation of atomic multicast (based on Multi-ring Paxos [23]) that, unlike Scraper, does not ensure the minimality/genuineness property. These implementation differences are quite substantial and, as such, would not allow to perform a fair, direct comparison between S-SMR and Genepi.

B. Experimental setup

Deployment and parameters Our experiment was conducted in the Grid'5000 testbed[7]. We used machines of the 'suno' cluster from the Sophia site. Each machine is equipped with two Intel Xeon E5520 CPU, each hosting 4 cores. Each machine has 32 GB of memory and has two Gigabit Ethernet cards. In the experiments we used 5, 10, 20 and 40 nodes. According to our measurement, the average round-trip time between nodes in the cluster is 0.4 ms. Due to hardware resource constraints, we do not enable replication in the experiment; instead, each node logically acts as a highly available partition, and for each replication request (due to C-PROPOSE invocation of consensus), we inject 3ms of delay to simulate the latency of consensus. We use 5ms as the duration to batch operations in Genepi (α) and 0.8 ms for the β timeout. We set $\delta = 2$, which is the recommended value obtained using Eq. 1 and assuming l_i=0.4, l_c=3 and $l_{rm} = 0.4$. This means, that MPOs are scheduled to be executed two rounds in the future of the current one.

Workloads The evaluation is based on two benchmarks: the TPC-C benchmark [24] and a micro benchmark. The TPC-C benchmark consists of five types of transactions (i.e. operations): *New-Order* (45%), *Payment* (43%), *Delivery* (4%), *Order-Status* (4%) and *Stock-Level* (4%). Of these five transactions, *New-Order*, *Payment* and *Delivery* are update transactions and the last two are read-only transactions. The *New-Order* transaction accesses a single partition with 90% probability and two partitions with 10% probability, and *Payment* accesses one partition with 85% probability and two partitions with 15% probability; the other three types of transactions always access a single partition. On average, 10% of transactions are distributed transactions, i.e., MPOs.

We also designed a micro benchmark that allows us to have precise control over different workload parameters. Unless otherwise specified, each operation reads and updates 10 keys. If an operation is an SPO, it accesses all 10 keys from a single partition; if an operation is an MPO, the number of keys to

[6]https://github.com/yaledb/calvin

[7]https://www.grid5000.fr/

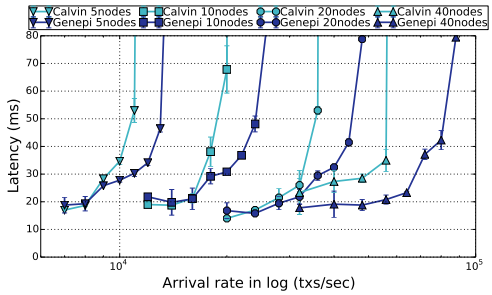


Fig. 3: Comparing Genepi and Calvin while varying the scale of the system (TPC-C).

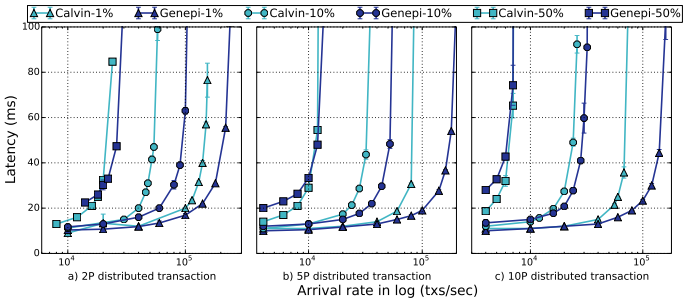


Fig. 4: Comparing Genepi and Calvin while varying the %MPOs and no. of involved partitions.

access per partition is calculated as $10/$*number of involved partitions*.

All the results represent the average of at least three runs.

C. Scalability

This experiment compares the scalability of Genepi and Calvin. We do not evaluate Genepi-DE in these experiments, since Genepi-DE differs from Genepi only in the handling of independent operations, and the workloads considered here do not include independent MPOs.

First, we consider the micro benchmark. We vary two parameters: the scale of the systems (5, 10, 20 and 40 nodes) and the percentage of MPOs (1%, 10% and 50%). The number of partitions an MPO accesses is fixed to two (we evaluate the impact of this parameter later, in Section VI-D). For all workloads in Fig. 2, Genepi is able to achieve higher peak throughput. Essentially, this is because Genepi orders MPOs using Scraper, which is able to (usually) totally overlap the ordering of MPOs with the processing of operations. Conversely, Calvin's ordering phase for MPOs lies on the critical path of operation processing, which can greatly hamper throughput. Note that this advantage is amplified when a system has larger scale, i.e. a larger number of nodes, as the duration of Calvin's ordering phase tends to have higher latency due to higher chance of incurring abnormally large message delays or stragglers. For workloads with 1% of MPOs in Fig. 2, Genepi achieves 38%, 47%, 57% and 83% throughput gains over Calvin, with 5, 10, 20 and 40 nodes in the system respectively. However, even though the throughput of Genepi is significantly higher than Calvin for workloads with 1% and 10% of MPOs, the improvement is not prominent for workloads with 50% of MPOs, where both systems achieve a relatively low throughput. In fact, recall that the execution of dependent MPOs requires involved partitions to exchange messages, i.e., to synchronize, which is dramatically slower than the execution of SPOs. As such, for the workload with 50% of dependent MPOs, the bottleneck in both systems is represented by the execution of MPOs.

Next, we evaluate both systems using the TPC-C benchmark. We vary the scale of the system and use the transaction mix described in §VI-B. Similar to the previous experiment, Fig. 3 shows that Genepi consistently achieves higher peak throughput than Calvin, and the relative improvement increases with the scale of the system: with 5, 10, 20 and 40 nodes, Genepi achieves 18%, 21%, 34% and 45% of throughput gain over Calvin. This experiments confirms that Genepi can deliver substantial throughput gains also in presence of complex, realistic workloads.

Finally, in these two experiments, we observe that, with Genepi, the average latency of MPOs is about 10ms (7ms~14ms) higher than SPOs: we argue that this is a small price to pay, which is hardly noticeable in typical scenarios (especially considering that the actual user-perceived latency includes, in practice, also the network latency between the client and the system, which is not considered here.), in order to enable much larger gains in terms of system's throughput.

D. Multi-partition operations

While the previous experiment has shown that Genepi achieves better performance than Calvin in workloads where MPOs access small number of partitions, in this experiment we are interested in assessing how the two protocols perform when MPOs access a larger number of partitions. We fix the scale of the system to ten nodes and vary the percentage of MPOs (1%, 10%, 50%) and the number of partitions an MPO involves (2, 5 and 10). Also, in this case, MPOs are all dependent.

Figure 4 shows that while Genepi provides higher throughput and lower latency than Calvin in most cases, it provides high latency when the workload has high percentage (50%) of MPOs and MPOs access large number of partitions (5 or 10). Essentially, because of these two characteristics, Scraper has to involve large number of nodes to order MPOs almost for each round, similar to Calvin which has to synchronize all nodes at each round. Furthermore, as Scraper requires more communication steps than Calvin to order messages, this makes the overall ordering latency of Scraper to be larger than Calvin. Nevertheless, we argue that the combination of these two cases are rare in practical applications, which (like TPC-C) normally strive to minimize the frequency of MPOs by optimizing the data partitioning scheme [25], [26], [27].

E. Signaling mechanism for independent MPOs

Finally, we evaluate the performance of Genepi and Genepi-DE when handling workloads with independent MPOs, which update keys (instead of first reading and then updating them, as in the previous experiments) from multiple partitions.

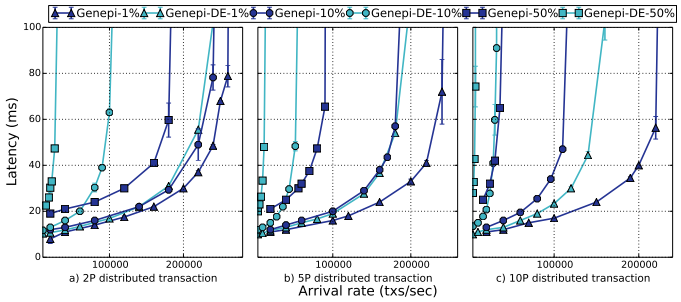


Fig. 5: Comparing Genepi and Genepi-DE.

To achieve linearizability, Genepi employs the delayed reply technique, while Genepi-DE employs S-SMR's signaling mechanism, which delays the execution of operations while waiting to gather signals for independent MPOs.

Figure 5 shows that the throughput of Genepi-DE drops dramatically even for a small proportion of independent MPOs, as we had anticipated in §I: when MPOs involve two partitions, a workload with 10% of MPOs achieves 50% less throughput than a workload with 1% of MPOs; this is even aggravated for workloads where MPOs involve ten partitions, where a workload with 10% MPOs achieves almost 80% less throughput than the workload with 1% MPOs. By delaying reply rather than delaying the execution of operations, the performance of Genepi is much more stable even with increasing number of MPOs: for the above two cases, the performance of Genepi only drops by 4% and 47%, respectively. This allows Genepi to achieve significant performance gains over Genepi-DE, namely up to 5.5× higher throughput for workloads with 50% of MPOs accessing two partitions.

VII. Conclusion

In this paper, we propose Genepi, a novel PRSM protocol that efficiently coordinates the execution of multi-partition operations. This is achieved by pursuing two key ideas: (i) removing the inter-partition coordination required by MPOs from the critical path of execution of *any* operation in the system, and (ii) maximizing the scalability of the MPO synchronization mechanism, by leveraging on a novel, fault-tolerant distributed agreement scheme, called Scraper.

Via an experimental study, based on both synthetic benchmarks and TPC-C, we show that Genepi achieves significant throughput gain over state of the art PRSM systems, at the cost of a negligible latency overhead for MPOs.

References

[1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[2] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 2011, pp. 454–465.

[3] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 331–342.

[4] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 1–12.

[5] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 3, pp. 272–314, 1991.

[6] R. Guerraoui and A. Schiper, "Total order multicast to multiple groups," in *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*. IEEE, 1997, pp. 578–585.

[7] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[8] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. Ieee, 2013, pp. 266–275.

[9] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 575–584.

[10] L. Lamport, "Generalized consensus and paxos," Tech. Rep.

[11] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing*, vol. 15, no. 2, pp. 97–107, 2002.

[12] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran, "Speeding up consensus by chasing fast decisions," in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, 2017, pp. 49–60. [Online]. Available: https://doi.org/10.1109/DSN.2017.35

[13] R. Palmieri, F. Quaglia, and P. Romano, "Osare: Opportunistic speculation in actively replicated transactional systems," in *Simposyum on Reliable Distributed Systems (SRDS)*. IEEE, 2011, pp. 59–64.

[14] S. Hirve, R. Palmieri, and B. Ravindran, "Archie: a speculative replicated transactional system," in *Proceedings of the 15th International Middleware Conference*. ACM, 2014, pp. 265–276.

[15] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theoretical Computer Science*, vol. 254, no. 1, pp. 297–316, 2001.

[16] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004, vol. 19.

[17] S. Peluso, P. Romano, and F. Quaglia, "Score: A scalable one-copy serializable partial replication protocol," in *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 2012, pp. 456–475.

[18] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. IEEE, 2012, pp. 455–465.

[19] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*. IEEE, 2010, pp. 214–224.

[20] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[21] R. Palmieri, F. Quaglia, and P. Romano, "Aggro: Boosting stm replication via aggressively optimistic transaction processing," in *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*. IEEE, 2010, pp. 20–27.

[22] S. Frolund and R. Guerraoui, "Implementing e-transactions with asynchronous replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 2, pp. 133–146, Feb 2001.

[23] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[24] "Tpc benchmark-w specification v. 1.8," http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, T. consortium.

[25] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-store: Fine-grained elastic partitioning for distributed transaction processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 245–256, 2014.

[26] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "Autoplacer: Scalable self-tuning data placement in distributed key-value stores," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 4, p. 19, 2015.

[27] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve, "Automated data partitioning for highly scalable and strongly consistent transactions," *IEEE transactions on parallel and distributed systems*, vol. 27, no. 1, pp. 106–118, 2016.