Reconfiguring Parallel State Machine Replication

Eduardo Alchieri*, Fernando Dotti†, Odorico M. Mendizabal‡ and Fernando Pedone§
*Departamento de Ciência da Computação – Universidade de Brasília, Brazil
†Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul, Brazil
‡Centro de Ciências Computacionais – Universidade Federal do Rio Grande, Brazil
§Universitá della Svizzera italiana (USI), Switzerland

***Abstract*—State Machine Replication (SMR) is a well-known technique to implement fault-tolerant systems. In SMR, servers are replicated and client requests are deterministically executed in the same order by all replicas. To improve performance in multi-processor systems, some approaches have proposed to parallelize the execution of non-conflicting requests. Such approaches perform remarkably well in workloads dominated by non-conflicting requests. Conflicting requests introduce expensive synchronization and result in considerable performance loss. Current approaches to parallel SMR define the degree of parallelism statically. However, it is often difficult to predict the best degree of parallelism for a workload and workloads experience variations that change their best degree of parallelism. This paper proposes a protocol to reconfigure the degree of parallelism in parallel SMR on-the-fly. Experiments show the gains due to reconfiguration and shed some light on the behavior of parallel and reconfigurable SMR.**

I. Introduction

State Machine Replication (SMR) is a well-known technique to implement fault-tolerant systems [1], [2]. The idea is to replicate servers and execute client requests deterministically and in the same order at all replicas. SMR's execution model provides strong consistency (i.e., linearizability [3]) but limits performance since in order to ensure determinism, replicas usually execute requests sequentially. This performance limitation is particularly important if replicas are equipped with multiple processors.

To improve the performance of state machine replication, some approaches have proposed to parallelize the execution of non-conflicting requests (e.g., [4]–[7]). Two requests are *non-conflicting* if they access different objects or only read shared objects, otherwise they are *conflicting*. The idea behind parallel SMR approaches is to optimize the execution of non-conflicting requests: while conflicting requests must be executed sequentially, by a single thread at each replica, non-conflicting requests can be executed in parallel at replicas, by multiple threads.

Transitioning from parallel to sequential execution requires expensive synchronization among threads (e.g., typically a barrier), whose cost usually increases with the number of threads involved. This situation creates a performance tradeoff that involves the *degree of parallelism* of a replica (i.e., its number of execution threads). A high degree of parallelism increases performance in workloads with predominantly non-conflicting requests, but negatively impacts performance if conflicting requests are the norm, due to the need for additional synchronization. Conversely, a low degree of parallelism excels in workloads dominated by conflicting requests, but underperforms in the presence of non-conflicting requests.

This performance dilemma is exacerbated by the fact that existing approaches to parallel SMR define the degree of parallelism statically, at system startup. Therefore, although parallel SMR is promising, its performance hinges on the "right" degree of parallelism, which is difficult to configure a priori. In addition, many workloads undergo variations during the execution [8], which affect their ideal degree of parallelism. Motivated by these limitations, in this paper we propose protocols to reconfigure parallel SMR on-the-fly. In brief, the idea is to monitor the execution and reconfigure the degree of parallelism based on the workload. The challenge is to achieve replica reconfiguration without violating SMR's strong consistency guarantees.

More precisely, the paper makes the following contributions:

- We propose a way to express concurrency of a service by means of *classes of requests*. Requests in a class may conflict with other requests in the class or with requests in other classes. Modeling concurrency with classes of requests simplifies the scheduling of requests.
- We present a parallel SMR protocol that uses the notion of classes of requests.
- We show that one can deploy a hybrid replicated system in which replicas are configured with different degrees of parallelism, in an attempt to optimize for workloads with different characteristics.
- We present a protocol that supports reconfiguration in the degree of parallelism, making it possible for replicas to adapt to the workload.
- We experimentally evaluate the proposed systems using two applications, a tuple space and a linked list. The main conclusions of our evaluation are twofold: (a) the hybrid system cannot sustain high performance, an inherent limitation, not an artifact of our implementation; and (b) the reconfigurable system outperforms static configurations.

The remainder of this paper is organized as follows. Section II defines the system model. Section III discusses the related work. Section IV presents our approach to expressing concurrency through classes of requests. Section V provides protocols for parallel, hybrid and reconfigurable SMR. Section VI reports on an experimental evaluation. Finally, Section VII concludes the paper.

II. System model and consistency

We assume a distributed system composed of interconnected processes that communicate by exchanging messages. There is an unbounded set of clients and a bounded set of service replicas. The system is asynchronous: there is no bound on message delays and on relative process speeds. We assume the crash failure model and exclude malicious and arbitrary behavior. A process is *correct* if it does not fail, or *faulty* otherwise. There are f faulty replicas, out of $n = 2f + 1$.

Processes have access to an atomic broadcast communication abstraction, defined by primitives $broadcast(m)$ and $deliver(m)$, where m is a message. Atomic broadcast ensures the following properties [9], [10][1]: (i) *Validity*: If a correct process broadcasts a message m, then it eventually delivers m. (ii) *Uniform Agreement*: If a process delivers a message m, then all correct processes eventually deliver m. (iii) *Uniform Integrity*: For any message m, every process delivers m at most once, and only if m was previously broadcast by a process. (iv) *Uniform Total Order*: If both processes p and q deliver messages m and m', then p delivers m before m', if and only if q delivers m before m'.

Our consistency criterion is *linearizability*. An execution is linearizable if it satisfies the following requirements [3]:

1) It respects the real-time ordering of requests across all clients. There exists a real-time order among any two requests if one request finishes at a client before the other request starts at a client.
2) It respects the semantics of the requests as defined in their sequential execution.

III. Related Work

The performance of state machine replication is bounded by the rate at which replicas can order and execute requests. Many works in the literature have proposed protocols to order messages efficiently (e.g., [13]–[19]). Less attention, however, has been paid to increasing the rate of requests that replicas can execute. In fact, in light of the requirement that command execution must be deterministic, most systems based on state machine replication execute requests sequentially. It has been early observed, however, that requests do not have to be executed in the same order if they do not content for the same data items [2]. As a result, some proposals have exploited the semantics of applications to introduce concurrency in the execution of requests (e.g., [4]–[6], [8], [20], [21]).

In [4], the authors present CBASE, a parallel SMR where replicas are augmented with a deterministic scheduler. Based on application semantics, the scheduler serializes the execution of conflicting requests according to the delivery order and dispatches non-conflicting requests to be processed in parallel by a pool of worker threads. Consistency is ensured since requests are delivered in the same order at replicas and the scheduler adopts deterministic policies to establish a partial order for request processing.

[1]Atomic broadcast needs additional assumptions about the system in order to be implemented [11], [12]. Hereafter, we simply assume the existence of an atomic broadcast oracle.

In Rex [22], a single server receives requests and processes them in parallel. While executing, the server logs a trace of dependencies among requests based on the shared variables locked by each request. The server periodically proposes the trace for agreement to the pool of replicas. The other replicas receive the traces and replay the execution respecting the partial order of commands.

CRANE [23] implements a parallel SMR by deploying deterministic versions of POSIX socket and the Pthreads synchronization interfaces. To this aim, CRANE combines the input determinism of Paxos and the execution determinism of deterministic multi-threading (DMT) [24].

In [5], the authors propose a parallel SMR approach that uses multiple multicast groups to partially order commands across replicas, where each group leads to a different stream of commands delivered at each replica. In this approach, non-conflicting commands are not delivered by a single component (e.g., the scheduler) and then scheduled for parallel execution. Instead, non-conflicting commands can be directly delivered by multiple worker threads by mapping command streams to multiple sockets. Therefore, the overhead associated with a parallelizer mechanism is minimized by this approach.

Speculative strategies are implemented by Eve [25] and Storyboard [26]. In Eve, before execution, a primary replica groups client commands into batches and transmits the batched commands to all replicas. Then, replicas speculatively execute batched commands in parallel. After the execution of a batch, the verification stage checks the validity of replica's state. If too many replicas diverge, replicas roll back to the last verified state and re-execute the commands sequentially. In Storyboard [26], a forecasting mechanism is used. Based on application-specific knowledge, Storyboard predicts the same ordered sequence of locks across replicas. While forecasts are correct, commands can be executed in parallel. If the forecast made by the predictor does not match the execution path of a command, then the replica has to establish a deterministic execution order in cooperation with other replicas.

Analogously to [4], our approach implements a scheduler that dispatches commands to worker threads: non-conflicting commands are executed in parallel while conflicting commands are serialized. In addition, our approach is independent of the ordering protocol, which can order batches of requests in a single instance of consensus [27]–[29], or use several instances of consensus in parallel, in the same manner as P-SMR [5]. Differently from all previous approaches to parallel state machine replication, we propose to reconfigure the number of threads at run time. By changing the number of worker threads on-the-fly, our approach is able to adapt to variations in the workload.

Other forms of reconfiguring a state machine replication have been proposed in the literature. For example, [30], [31] discuss techniques to dynamically change the set of replicas while [8], [21] propose protocols to reconfigure partitions and the location of objects in a scalable state machine replication [20], respectively. These techniques are orthogonal to the ones presented in this paper.

IV. Parallelism in state machine replication

Parallel SMR exploits the fact that strong consistency does not require all requests to be executed in the same order at all replicas. In this section, we formalize this idea and provide examples of applications that can exploit parallelism.

A. *Conflicts, classes and correctness*

To argue about the correctness of our parallel SMR algorithms, we introduce next some basic definitions.

Definition 1 (Request conflict)**.** Let r_i and r_j be application requests in R, and Rr_i and Rr_j, and Wr_i and Wr_j the requests' readset and writeset, respectively. The conflict relation $\#_R \subseteq R \times R$ among requests is given by: $(r_i, r_j) \in \#_R$ if $(Wr_i \cap Wr_j \neq \emptyset) \vee (Wr_i \cap Rr_j \neq \emptyset) \vee (Rr_i \cap Wr_j \neq \emptyset)$. Requests r_i and r_j *conflict* if $(r_i, r_j) \in \#_R$. We refer to pairs of requests not in $\#_R$ as *non-conflicting* or *independent*.

Definition 2 (Request sequence)**.** A *request sequence* is a pair $(R, <_R)$ where R is a set of requests and $<_R \subseteq R \times R$ is an irreflexive total order. We assume that $(R, <_R)$ represents the requests and their total delivery order resulting from atomic broadcast.

Definition 3 (Request dependency relation)**.** Given a request sequence $(R, <_R)$ and the conflict relation $\#_R \subseteq R \times R$ among requests, the derived request dependency relation $\prec_R$ is the transitive closure of $<_R \cap \#_R$. This means that for any two conflicting requests in R, their relation specified in $<_R$ is in $\prec_R$ while independent requests are not related. The transitive closure is needed to relate any two requests whose order is induced indirectly.

Based on the definitions presented above, we now develop a strategy to argue about the correctness of our parallel state machine protocols.

Definition 4 (Execution)**.** An *execution* of $\prec_R$ is any total order that is compatible with $\prec_R$.

Proposition 1. *Any execution of $\prec_R$ respects linearizability.*
Proof: Regarding the first strong consistency condition (Section II), whether sequential or parallel SMR, i.e., execution respectively according to $<_R$ or $\prec_R$, this holds if a request is answered only after executed. If r_i is answered before r_j is issued, it implies that r_i certainly executed before r_j which did not exist when r_i was decided to execute. The real-time ordering of requests is respected. If both r_i and r_j are submitted and not responded, they are concurrent. In such case the ordering protocol decides a uniform delivery order to be followed.

Regarding the second condition (Section II), it has to be shown that the parallel execution (i.e., any execution respecting $\prec_R$) is equivalent to a sequential one both from the clients perspective and from the possible intermediate SMR states.

Due to the determinism of SMR requests, the resulting state and output of a request are completely determined by the input parameters and state. When two requests are independent, according to Def. 1 they will read the same SMR state contents irrespective of the order of their execution. Also, their write sets are disjoint. Therefore, both requests will compute the same output to the clients and state updates irrespective of the relative orders of these requests. This means that the semantics of such requests are the same as in their sequential execution. Also, after both requests have executed, the SMR reaches the same state irrespective of their order. When two requests are dependent, then $\prec_R$ is followed. Since $\prec_R \subseteq <_R$, the total order of conflicting requests is respected. □

The above definition of conflict is minimal in the sense that it represents the exact conflicts among requests. We now introduce the notion of *classes of requests*, which group requests and their interdependencies. In the following sections, we show how one can use request classes to schedule requests in a simple manner. Each class has a descriptor and conflict information. Requests belonging to conflicting classes have to be serialized according to the ordering relation; requests belonging to non-conflicting classes can be executed concurrently. Thus, requests in a class that conflicts with itself must be serialized.

Definition 5 (Request classes)**.** Let $R = \{r_1, .., r_{nr}\}$ be the set of request descriptors, where nr is the number of requests; and $C = \{c_1, .., c_{nc}\}$ be the set of class descriptors, where nc is the number of request classes. We define request classes as $RC = C \to \mathcal{P}(C) \times \mathcal{P}(R)$,[2] that is, any class in C may conflict with any subset of classes in C, and is associated to a subset of requests in R.

A request class set RC is correct with respect to a request conflict $\#_R$ if every conflict in $\#_R$ is captured in RC, as formally defined next.

Definition 6 (Correct request classes)**.** Given a request conflict relation $\#_R$, request classes RC is correct with respect to $\#_R$ if for all $(r_i, r_j) \in \#_R$, it follows that $\exists C_i \to (CC_i, CR_i) \in RC$ and $\exists C_j \to (CC_j, CR_j) \in RC$ such that: (a) $r_i \in CR_i$, (b) $r_j \in CR_j$, (c) $C_j \in CC_i$, and (d) $C_i \in CC_j$.

B. *Applications and request classes*

We illustrate the use of classes of requests with three applications. In Section VI we assess these applications experimentally with parallel SMR protocols.

1) Simple Tuple Space: Consider a tuple space with four types of requests: $out(t)$ to insert tuple t in the space, $rdp(\bar{t})$ to read a tuple that matches template $\bar{t}$ from the space; $inp(\bar{t})$ to remove a tuple that matches $\bar{t}$ from the space; and $cas(\bar{t}, t)$ that inserts t and returns $true$ if there is no tuple that matches $\bar{t}$ on the space, or otherwise returns $false$.

We define: $C = \{c_r, c_w\}$, $R = \{rdp, out, inp, cas\}$, and $RC = \{(c_r \to (\{c_w\}, \{rdp\})), (c_w \to (\{c_r, c_w\}, \{out, inp, cas\}))$. This states a concurrency model where rdp requests do not conflict with each other while out, inp and cas requests conflict with each other and with rdp.

[2]We denote the power set of set S as $\mathcal{P}(S)$.

2) Extended Tuple Space: Now we consider that a tuple space may store tuples with different number of fields and model concurrency according to this information.

We define $C = \{c_i, c_{i,j} | 1 \leq i, j \leq n\}$ where n is the maximum number of fields that tuples or templates can have, $R = \{rdp_i, out_i, inp_i, cas_{i,j} | 1 \leq i, j \leq n\}$, and $RC = \{(c_i \rightarrow (\{c_i, c_{i,j}, c_{j,i}\}, \{out_i, inp_i, rdp_i\})) | 1 \leq i, j \leq n\} \cup \{(c_{i,j} \rightarrow (\{c_i, c_j, c_{i,k}, c_{k,i}, c_{j,k}, c_{k,j}\}, \{cas_{i,j}\})) | 1 \leq i, j, k \leq n\}$, where subscripts i and j in out_i, inp_i, rdp_i, and $cas_{i,j}$ mean the number of fields in the tuple or template. With this concurrency model, out, inp, and rdp requests to tuples with different number of fields are concurrent, but conflict otherwise; and a cas request and any other request conflict if they refer to a tuple with the same number of fields.

3) Linked List: As another example consider a linked list used to store and retrieve integers through the following requests: $add(i)$, which includes i in the list and returns *true* if i is not in the list, otherwise returns *false*; $remove(i)$, which removes i from the list and returns *true* if i is in the list, otherwise returns *false*; $get(p)$, which returns the element at position p; and $contains(i)$, which returns *true* if i is in the list, otherwise returns *false*.

We specify a concurrency model where $C = \{c_r,\ c_w\}$, $R = \{get, contains, add, remove\}$, and $RC = \{(c_r \rightarrow (\{c_w\}, \{get, contains\})), (c_w \rightarrow (\{c_r, c_w\}, \{add, remove\}))\}$. With this concurrency model, get and $contains$ requests do not conflict with each other but conflict with add and $remove$ requests, which conflict with all requests.

V. Parallel and Reconfigurable SMR

This section presents our proposal for a parallel and reconfigurable SMR. We first show how concurrency expressed using request classes is mapped to worker threads and then present different approaches to parallelism in state machine replication.

A. From request classes to threads

In parallel SMR, performance is impacted by the workload and the number of worker threads. The workload is defined by the application semantics and usage profile, which may induce a higher or lower number of conflicting commands. The number of worker threads is a consequence of replica configuration, that is, a high number of worker threads increases the parallelism in predominantly non-conflicting workloads. To benefit from the potential parallelism of an application, both aspects should be considered while tracking dependencies.

In our approach, interdependencies among requests are captured by *request classes*. Once interdependencies among classes are defined, parallelism is expressed by mapping request classes to worker threads. Note that the request classes definition is problem-oriented and architecture-independent, while the association of request classes to worker threads is particular to the architecture (e.g., number of processors or cores in a replica).

We represent the mapping of request classes to threads with function $CtoT$ (see Def. 7 below). The $CtoT$ mapping must respect all interdependencies between classes. Moreover, a condition for consistency is that if two classes conflict, then they must be assigned to at least one thread in common. Although in our prototype the application developer must provide the $CtoT$, we believe one could automate the creation of $CtoT$ for a given request class.

Definition 7 (Classes to threads, CtoT)**.** If $T = \{t_0, .., t_{n-1}\}$ is the set of replica thread identifiers, where n is the number of threads, and C is the set of class names as in Def. 5, then $CtoT = C \rightarrow \mathcal{P}(T)$.

In the Simple Tuple Space and Linked List, we can define the mapping as $CtoT(c_r) = \emptyset$ (i.e., requests can be concurrently executed by any threads) and $CtoT(c_w) = \{t_0, .., t_{n-1}\}$ (i.e., all threads must synchronize before a request is executed).

In the Extended Tuple Space, considering tuples with at most n fields, we can define $CtoT(c_i) = CtoT(c_{i,j} | i = j) = \{t_{i-1}\}$ (i.e., a single thread executes a request without synchronization) and $CtoT(c_{i,j} | i \neq j) = \{t_{i-1}, t_{j-1}\}$ (i.e., two threads must synchronize for the execution).

B. A protocol for parallel SMR

Whenever a request is delivered by the atomic broadcast protocol, the scheduler (Algorithm 1) assigns it to one or more worker threads, according to the request class. Each thread executes its requests (Algorithm 2), synchronizing the execution of conflicting requests to preserve consistency.

Scheduler. The communication between the scheduler and worker threads takes place through synchronized queues. Requests that could be concurrently executed are associated to a unique thread in a round-robin policy ($putAny$), otherwise all threads involved receive the request to synchronize the execution ($putOnConflicting$).

Algorithm 1 Scheduler.

```
variables: Variables used by the scheduler.
    // used definitions: C (classes), CtoT (class to conflicting threads)
    $n$                                   // the number of worker threads
    $queues[0, ..., n-1] \leftarrow \emptyset$            // queue per thread (see Alg. 2)
    $nextThread \leftarrow 0$        // thread to exec next non-conflicting request
    $barriers = barrier[C]$                  // one barrier for each class
auxiliary functions:
 putAny(req)
  1) $queues[nextThread].fifoPut(req)$        // assigns req to a thread...
  2) $nextThread \leftarrow (nextThread + 1)\%n$        //...in round-robin policy
 putOnConflicting(req)
  3) $\forall t \in CtoT(req.classId)$                  // all conflicting threads ...
  4)     $queues[t].fifoPut(req)$                // ... synchronize to exec req
on initialization:
  5) $\forall c \in C$, $barriers(c) \leftarrow$ new barrier for $|CtoT(c)|$ threads
on deliver(req):
  6) if $CtoT(req.classId) = \emptyset$ then                     // no conflicts
  7)     $putAny(req)$
  8) else                                                    // a conflict
  9)     $putOnConflicting(req)$
 10) end if
```

Worker Threads. Upon receiving a request that could be concurrently executed, the thread simply executes it. Otherwise, the thread uses the corresponding barriers to synchronize with the involved threads ($execWithBarrier$).

Algorithm 2 Worker Threads.

```
variables: Variables used by each worker thread.
    // used definitions: C (classes), CtoT (class to conflicting threads)
    $n$                          // the number of worker threads
    $myId \leftarrow id \in \{0, ..., n-1\}$   // thread $id$, $n$ is the number of threads
    $queue$          // the synchronized queue with requests for this thread
    $barriers = barrier[C]$       // one barrier for each class (see Alg. 1)
auxiliary functions:
execWithBarrier(req,barrier, threadIds)
 1) if $myId = min(threadIds)$ then            // thread with smallest id ...
 2)     $barrier.await()$               // ... waits for other threads to stop ...
 3)     $exec(req)$                            // ... executes the request ...
 4)     $barrier.await()$                    // ... and resumes other threads
 5) else
 6)     $barrier.await()$                // thread signalizes ok to execute ...
 7)     $barrier.await()$                        // ... and waits for the execution
 8) end if
on thread run:
 9) while true do
10)     $req \leftarrow queue.fifoGet()$        // awaits until a request is available
11)     if $CtoT(req.classId) = \emptyset$ then                    // no conflict
12)         $exec(req)$          // executes request and replies to client
13)     else              // conflict: synchronizes with involved threads
14)         $execWithBarrier$
                    $(req, barriers(req.classId), CtoT(req.classId))$
15)     end if
16) end while
```

1) Why it works: Here we argue that the above described execution model respects $\prec_R$ as discussed in Section IV.

Definition 8 (Replica)**.** Given a request sequence $(R, <_R)$, a *replica Rp* is a finite set of worker threads that execute this request sequence, that is $Rp = \{WT_i | 0 \leq i \leq n-1, n \in \mathbb{N}, WT_i = (R_i, <_{R_i})$ is also a request sequence$\}$ such that (see Algorithm 1):

i) All requests in R are enqueued at some worker thread: $\bigcup_{0 \leq i \leq n-1} R_i = R$; which is granted since every delivered request is assigned either to one thread (no conflict case); or to a subset of conflicting threads.

ii) The queues of the worker threads are compatible with the request sequence order: $<_{R_i} = <_R |_{R_i}$; $<_R$ is represented by the total delivery order. Since *on deliver* is performed in the delivery order, the scheduler appends requests to threads in the correct sequence. The total order of each $<_{R_i}$ is compatible with $<_R$.

iii) A request is in multiple threads if and only if it depends on items on those threads: $r \in R_i$ and $r \in R_j$ with $i \neq j \Leftrightarrow (\exists r_i \in R_i, r_j \in R_j, r_i \prec_R r$ and $r_j \prec_R r)$. This is ensured by *on deliver*. It identifies conflicts with all involved threads and enqueues the request to all of them.

Considering that each replica has the same conflict classes, requests will be enqueued to the same threads and thus induce the same dependencies. Note that since $<_{R_i}$ is an image of $<_R$, the dependency relation on requests of R_i, $\prec_{R_i}$, must be $\prec_R$ restricted to the items in R_i.

Definition 9 (Replica run)**.** Given a Rp with n worker threads WT over a request sequence $(R, <_R)$, a complete *run π of replica Rp* is a list defined inductively as follows ($\bullet$ denotes the inclusion of an element in front of a list):

i) π is the empty list if all worker threads are empty;

ii) $\pi = r \bullet \pi'$ if

a) r is the first request of a set of m worker threads of Rp and r does not appear in any other worker thread.

b) π' is the replica run of the replica R' obtained by removing r from all worker threads in which it appeared. This behavior is implemented in Algorithm 2. In case r appears in only one queue the thread executes it. Otherwise every involved worker thread eventually dequeues r and enters a barrier with other threads in $classId$. Only one thread executes r and all other threads wait for it to complete.

Given the formalization above of a replica run, now we have to discuss that such a run is an execution of $\prec_R$ which has been argued to be linearizable.

Proposition 2. *Replica run is an execution of $\prec_R$.*

Proof: The run π of a replica is:

i) the empty list: the empty execution satisfies $\prec_R$;

ii) $\pi = r \bullet \pi'$, in which case we have to show that the choice of executing r respects $\prec_R$. r is the first request of a set of m worker threads of Rp and r does not appear in any other worker thread. In this case r conflicted with requests of threads in m, according to Def. 8.iii. For each thread WT_i in m, r respects $\prec_{R_i}$, as discussed. Since $\bigcup_{0 \leq i \leq n-1} R_i = R$ (Def. 8.i) and since a request r is in multiple threads iff it depends in items on those threads (Def. 8.iii), each dependency relation $\prec_{R_i}$ is compatible with $\prec_R$ and all dependencies of r on other requests in $(R, \prec_R)$ are accordingly mapped to $\prec_{R_i} \in m$. This ensures that the execution $r \bullet \pi'$ respects $\prec_R$. The case where r appears in a single thread is a particular case where m has only one thread. Either r conflicted with prior requests for the thread or r is independent. The first case is correct due to the Fifo execution. In the second case, independent requests are assigned to some R_i, having a valid order since any order would suffice.

Since a replica run is an execution of $\prec_R$, according to Proposition 1 a replica run is safe. □

C. Replicas with different configurations

In an attempt to optimize for workloads with predominance for both conflicting and non-conflicting requests, replicas could be configured with different degrees of parallelism, ranging from sequential to a predefined maximum threshold. The idea is that sequential replicas execute faster conflicting requests while parallel replicas provide high performance for non-conflicting requests. Consequently, the system will present good perfomance in both scenarios.

The modified scheduler is presented in Algorithm 3 while the new algorithm for worker threads is shown in Algorithm 4. The main difference from the previous protocol is that we must handle the case of replicas that have fewer threads than the target number of threads in the $CtoT$ mapping. These requests are mapped to all threads, which must synchronize their execution to ensure linearizability.

1) Why it works: The general argument is that Proposition 2 is independent of the number of threads. In more detail, Algorithms 3 and 4 only differ from Algorithms 1 and 2, respectively, in that they verify if the set of conflicting threads returned by function $CtoT$ is supported by the replica's configuration. If it is, Algorithms 3 and 4 behave as Algorithms 1 and 2. Otherwise, Algorithms 3 and 4 synchronize all threads to execute the request. This preserves safety since this enforces the total delivery order in such cases, which is always correct. All executions will preserve the total order of conflicting requests, which satisfies $\prec_R$.

Algorithm 3 Scheduler for different configurations.

```
variables:                                        // Same as in Alg. 1, adding:
   $bAll$                         // new: barrier to synchronize all threads
auxiliary functions:                              // Same as in Alg. 1, adding:
 putAll(req)
 1)  $\forall q \in \{0, ..., n-1\}$                              // all threads ...
 2)      $queues[q].fifoPut(req)$                   // ... synchronize to exec req
on initialization:                                  Same as in Alg. 1, adding:
   $bAll \leftarrow$ new barrier for $n$ threads
on deliver(req):
 3)  if $CtoT(req.classId) = \emptyset$ then                           // no conflict
 4)      $putAny(req)$                                       // Same as in Alg. 1
 5)  else                                                           // conflicts
 6)      if $\forall tId \in CtoT(req.classId), tId < n$ then  // a valid set of threads
 7)          $putOnConflicting(req)$                         // Same as in Alg. 1
 8)      else
 9)          $putAll(req)$                                         // New behavior
10)      end if
11)  end if
```

Algorithm 4 Worker Threads for different configurations.

```
variables:                                           // Same as Alg. 2, adding:
   $bAll$              // new: barrier to synchronize all threads (see Alg. 3)
auxiliary functions:                                         // Same as Alg. 2
on thread run:
 1)  while true do
 2)      $req \leftarrow queue.fifoGet()$                              // Same as Alg. 2
 3)      if $CtoT(req.classId) = \emptyset$ then
 4)          $exec(req)$
 5)      else                                                       // conflict
 6)          if $\forall tId \in CtoT(req.classId), tId < n$ then
 7)                         // a valid set of threads, do the same as in Alg. 2
 8)              $execWithBarrier$
                          $(req, barriers(req.classId), CtoT(req.classId))$
 9)          else                                // otherwise: new behavior
10)              $execWithBarrier(req, bAll, \{0, ..., n-1\})$
11)          end if
12)      end if
13)  end while
```

D. Reconfiguration

In the protocols discussed so far, the number of worker threads of a replica (i.e., the replica's degree of parallelism) is defined statically, at system startup. The degree of parallelism of a replica is important and directly affects the performance of the system [5]. A large number of threads tends to increase performance in workloads with many non-conflicting requests, but performance decreases with the increase in the number of conflicting requests. A small number of threads hurts performance in workloads with few conflicting requests.

Since the workload may change during the execution, and given the constraints described above, setting an ideal number of worker threads a priori is a difficult task, although fundamental to achieving good performance. We now present a reconfiguration protocol that changes the number of threads during the execution, adapting to the current workload.

The main idea of the protocol is to divide threads into active and inactive. A thread is active if it is able to execute requests, otherwise it is considered inactive. Only the active threads will participate in parallel SMR execution and thus impact system performance. The reconfiguration takes place by activating and deactivating threads. At system startup, in addition to specifying the initial number of active threads, the system administrator must also set the minimum and maximum threads that can be active at the same time and provide a policy with the rules to be followed for activating and deactivating threads.

Algorithm 5 Reconfigurable Scheduler.

```
variables:       // Same as in Alg. 3, with following additions/redefinitions:
   $minThreads$     // new: the minimum number of active worker threads
   $maxThreads$     // new: the maximum number of active worker threads
   $bMax$                // new: barrier for $maxThreads$ number of threads
   $n$          // redef: not all threads but number of threads in use (active)
   $n'$                                // new: next number of active threads
   $queues[0, ..., maxThreads-1] \leftarrow \emptyset$     // redef: queues to all threads
auxiliary functions:                   // Same as in Alg. 3, overwriting to n':
 putAny(req)
 1)  $queues[nextThread].fifoPut(req)$ // assigns req to an active thread
 2)  $nextThread \leftarrow (nextThread + 1)\%$ n'          //in roud-robin policy
 putAll(req) =
 3)  $\forall q \in \{0, ...,$n'$-1\}$                            // all active threads ...
 4)      $queues[q].fifoPut(req)$                  // ... synchronize to exec req
checkForReconfiguration(req)
 5)  $num \leftarrow reconfigPolicy(req, minThreads, n', maxThreads)$
 6)  if $num \neq 0 \wedge (minThreads \leq n' + num \leq maxThreads)$ then
 7)      $n' \leftarrow n' + num$
 8)      $nextThread \leftarrow 0$
 9)      $\forall q \in \{0, ..., maxThreads-1\}$
10)          $queues[q].put($RECONFIG(n')$)$                  // all threads have it
11)  end if
on initialization:                                 // Same as in Alg. 3, adding:
   $bMax \leftarrow$ new barrier for $maxThreads$ number of threads
   $n = n' \leftarrow$ number of active threads
on deliver(req):                  // lines 3 to 11 of Alg. 3 changing $n$ by $n'$
12)  if $CtoT(req.classId) = \emptyset$ then                           // no conflict
13)      $putAny(req)$
14)  else                                                           // conflicts
15)      if $\forall tId \in CtoT(req.classId), tId < n'$ then
16)          $putOnConflicting(req)$
17)      else
18)          $putAll(req)$
19)      end if
20)  end if
21)  $checkForReconfiguration(req)$
```

Scheduler. The scheduler (Algorithm 5) performs similarly to the previously described schedulers. However, it adds behavior to reconfigure the number of active threads, described in $checkForReconfiguration$. First, the reconfiguration policy is used to verify the need for reconfiguration, which should return the number of threads to activate (a positive number) or deactivate (a negative number). Activated/deactivated threads will always be the ones with the highest identifiers. If it is necessary to reconfigure the system, a special request

(RECONFIG) is added in the queue of all threads (even the inactive ones, which also participate in the reconfiguration).

Worker Threads. The additional behavior in the worker threads (Algorithm 6) is that they need to execute RECONFIG requests to update the number and barrier for all active threads.

Algorithm 6 Worker Threads for reconfigurable replicas.

```
variables:      // Same as in Alg. 4, with following additions/redefinitions::
    $maxThreads$    // new: the maximum number of active worker threads
    $myId \leftarrow id \in \{0, ..., maxThreads - 1\}$ // redef: to $maxThreads$
    $bMax$  // new: barrier to synchronize $maxThreads$ number of threads
    $n$           // redef: not all threads but number of threads in use (active)
    $bAll$ // redef: barrier to synchronize not all threads but all active threads
auxiliary functions:                              // Same as Alg. 4, adding:
 exec(RECONFIG($newN$)) = // add: definition of $exec$ for reconfiguration
 1) $bAll \leftarrow$ new barrier for $newN$ threads        // redefine barrier with the
                                   // $newN$ calculated in $checkForReconfiguration$
 2) $n \leftarrow newN$          // switch worker threads to work with next value
on thread run:
 3) while true do
 4)     $req \leftarrow queue.fifoGet()$
 5)     if $req \neq$ RECONFIG then                        // Same as Alg. 4
 6)         lines 3 to 12 of Alg. 4
 7)     else                              // new behavior - reconfiguration
 8)         $execWithBarrier(req, bMax, \{0, ..., maxThreads - 1\})$
 9)     end if
10) end while
```

1) Reconfiguration Policies: The definition of the new system configuration and when to adopt it follows a reconfiguration policy which is specified by the user. The policy considers the request being scheduled, the current number of active threads, in addition to the minimum and maximum number of active threads for the replica. Replicas could use different policies since they can run with different configurations.

Algorithm 7 Reconfiguration Policy.

```
variables: Variables and sets used.
    $counter \leftarrow 0$           // counter for the number of received requests
    $conflict \leftarrow 0$          // counter for the number of conflict requests
    $period \leftarrow 2000$     // number of requests to check for reconfigurations
int reconfigPolicy(request, minThreads, currentThreads, maxThreads)
 1) $counter \leftarrow counter + 1$
 2) if $\mid CtoT(request.classId) \mid \neq 0$ then
 3)     $conflict \leftarrow conflict + 1$
 4) end if
 5) if $counter = period$ then
 6)     $p \leftarrow conflict * 100/period$; $conflict \leftarrow 0$; $counter \leftarrow 0$
 7)     if $(p \leq 20) \wedge ((currentThreads + 1) \leq maxThreads)$ then
 8)         return 1
 9)     else if $(p > 20) \wedge$
                        $((currentThreads - 1) \geq minThreads)$ then
10)         return -1
11)     end if
12) end if
13) return 0
```

Algorithm 7 presents an example policy, which states that for every 2000 requests the percentage of conflicting requests is calculated. If the workload has an amount of up to 20% of conflicting requests, an additional thread is activated, until the maximum number of threads is reached. Otherwise, a thread is deactivated until the minimum number of threads is reached. With a workload of over 20% conflicting requests, a lot of synchronization is needed and generally better performance is obtained with a sequential execution [5].

2) Why it works: To keep the execution safe it has to be shown that the replica run satisfies $\prec_R$ even if the replica is reconfigured to a different number of working threads. As already argued in Section V-C1, replicas with different number of threads subject to the same request sequence $(R, <_R)$ generate the same results (satisfy $\prec_R$). Since the number of threads is not important to keep safety, correctly switching the number of active threads will generate safe executions. We substantiate this claim by showing that the computation before and after the reconfiguration is generated correctly.

When the scheduler decides to change the number of threads (i.e., update n'), a $reconfig(n')$ request is appended to all $(maxThreads)$ threads. At this moment it is the last request of all queues and all threads synchronize to execute it. With this we have a point of reference in the concurrent execution of worker threads. Now we argue that requests before this point of reference are processed correctly. Any request processed before $reconfig$ is enqueued is assumed correct by previous argumentation (algorithms without reconfiguration) since the logic is the same. Any request prior to an enqueued $reconfig$ was enqueued considers the previous value of n. Since the barrier $bAll$ is only modified when $reconfig$ is executed (and not when it is enqueued), $bAll$ accords to the value of n previous to reconfiguration and thus involves the correct number of active threads before $reconfig$. By previous argumentation, the prefix before $reconfig$ is correct.

Now we argue about the correct processing after $reconfig$. $reconfig$ is executed in mutual exclusion. It changes the value of $bAll$ and n according to the new number of threads informed in the request $reconfig(n')$. According to the scheduler, any request enqueued after $reconfig$ considered the new number of active threads n'. When $reconfig$ takes place it updates $bAll$ and n (which is used in worker threads) to the new value n'. Therefore after the point of reference provided by $reconfig$, the number of active threads considered n accord to the barrier $bAll$ and to n' (i.e., the number of active threads considered by the scheduler after the respective $reconfig(n')$ was issued). This establishes a correct execution after $reconfig$. Since n' is only used at the scheduler and the new n for worker threads is informed in the $reconfig(n')$ message, one can observe that several enqueued $reconfig$ messages with different number of active threads are possible and the service requests enqueued between any two $reconfigs$ are processed with the correct values of n and $bAll$ at worker threads. It is also clear that the $reconfig$ process does not lose, insert, or change the order of requests in the queues.

VI. Experimental evaluation

In order to assess the performance of parallel state machine replication, we implemented the proposed protocols in the BFT-SMaRt environment [27] and conducted experiments in Emulab [32]. BFT-SMaRt was developed in Java and its atomic broadcast protocol executes a sequence of consensus instances, where each instance orders a batch of requests.

Experimental Setup: The Emulab environment was configured with 5 d710 machines (2.4 GHz 64-bit Intel Quad Core

Xeon E5530 with 2 CPU threads per core, 12GB of RAM and 1 Gbps network cards) and a 1Gbps switched network. The software installed on the machines was Ubuntu 14 64-bit and a 64-bit Java virtual machine version 1.8.0_121. For all experiments, BFT-SMART was configured with three replicas hosted in separate machines to tolerate up to one replica crash, while 90 clients were distributed uniformly across another two machines. We evaluated the raw throughput of the system at the servers and the latency perceived at the clients over the course of 300 seconds.

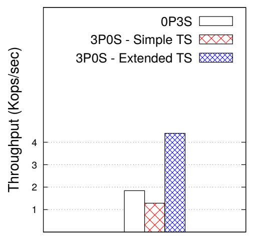
Goals: Our main goal is to analyze the differences among the previously discussed approaches and show how reconfigurations could increase the performance. The first set of experiments shows the expressiveness of the classes of requests proposed to model concurrency (Section VI-A). The second set of experiments depicts the behavior of the system under different replica configurations (Section VI-B). Finally, the last set of experiments shows how reconfigurations could improve performance (Section VI-C).

Notation: We use the notation `xPyS` to represent the configuration with `x` parallel and `y` sequential replicas. Except for the execution with reconfiguration, parallel replicas were configured with 10 working threads.

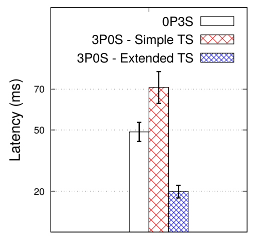
A. Modeling concurrency with classes of requests

To show how concurrency modeling impacts performance, we implemented a tuple space and analyzed the two models defined in Section IV-B using the thread mappings ($CtoT$) presented at Section V-A. The tuple space was initialized with $100k$ tuples, with the number of fields uniformly distributed from 1 to 10 at each replica. As for the workload, each client submits one of the possible requests at a time, rdp, inp, out and cas, and all request types appear in equal proportion in the workload. The number of fields for a tuple or template in each operation was uniformly distributed from 1 to 10.

Figure 1 presents the results for sequential and parallel executions for both concurrency models. The Simple Tuple Space (TS) concurrency model allows reduced concurrency, resulting in a workload with 75% of conflicting requests. This significantly impacts performance, which is worse than in a sequential execution (`0P3S`) due to the need for additional synchronization.



(a) Throughput



(b) Latency

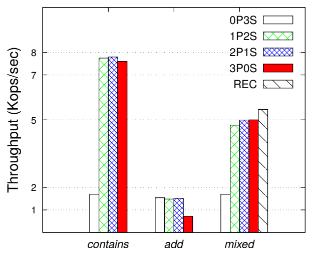
Fig. 1. Sequential vs. parallel executions for two concurrency models.

The Extended Tuple Space (TS) concurrency model, however, results in performance that is more than twice the performance of the sequential execution. In this model, out, rdp and inp requests handling tuples or templates with the same number of fields are mapped and sequentially executed by a single thread, but are executed concurrently with other requests for tuples or templates with different number of fields. cas requests may be mapped to one or two threads, depending on the request parameters. This model improves performance since it only needs to synchronize groups of two threads in cas executions. No other expensive synchronization among threads is necessary. These results show that classes of requests can precisely capture the concurrency of an application.

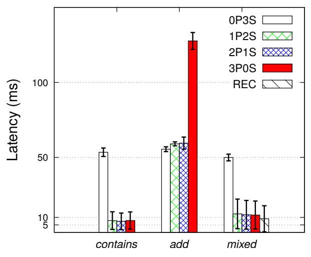
B. Replicas with different degrees of parallelism

We use a linked list implementation, the concurrency model defined at Section IV-B, and the thread mapping ($CtoT$) presented at Section V-A to analyze the performance of deployments in which replicas are statically configured with different degrees of parallelism. The list was initialized with $100k$ entries at each replica and we used add and $contains$ to represent operations of classes c_w and c_r, respectively. The parameter for these operations was always the last element in the list. Consequently, operations in an execution have similar cost, leading to more stable measurements.

Figure 2 presents the results for different replica configurations under workloads composed of $contains$ requests only, add requests only, and mixed requests, where clients cycle through 60-second periods during which one request type is submitted only, starting with $contains$. Recall that while $contains$ requests do not conflict with each other, any two add requests conflict. The throughput presented is the average obtained throughout the execution. For workloads with only $contains$ or add requests, there was little variation in throughput and latency during the execution, while for the mixed workload throughput and latency fluctuate over time (more details about the mixed workload in the next section). For the mixed workload, we also show the results for the reconfiguration technique (REC).



(a) Throughput



(b) Latency

Fig. 2. Different replica configurations.

The system with different replica configurations (`1P2S` and `2P1S`) presented better performance for both $contains$ or add workloads, while parallel (`3P0S`) and sequential (`0P3S`)

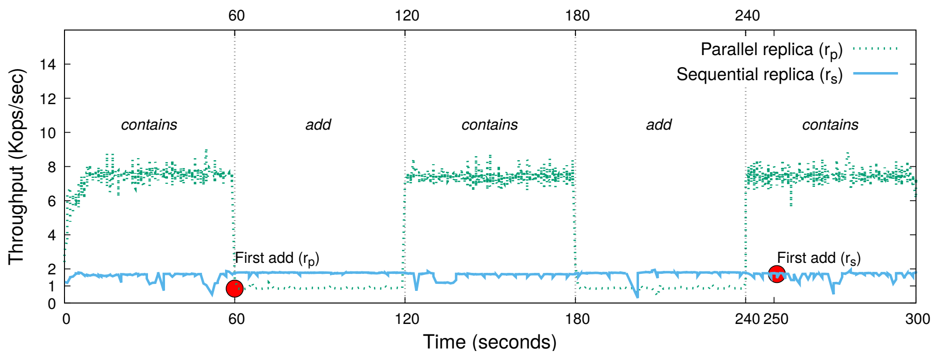
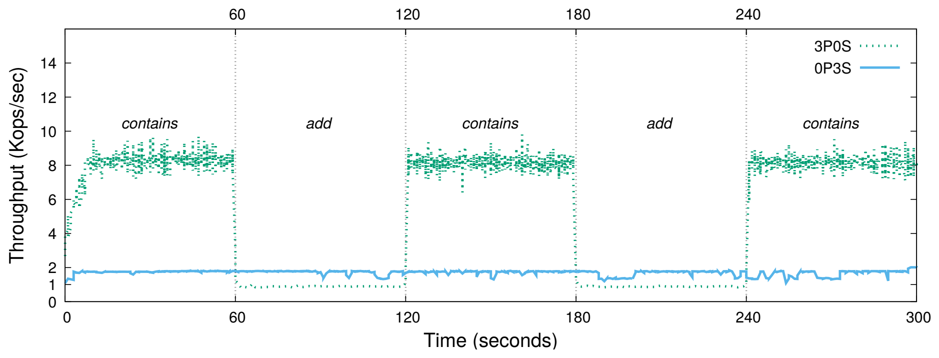
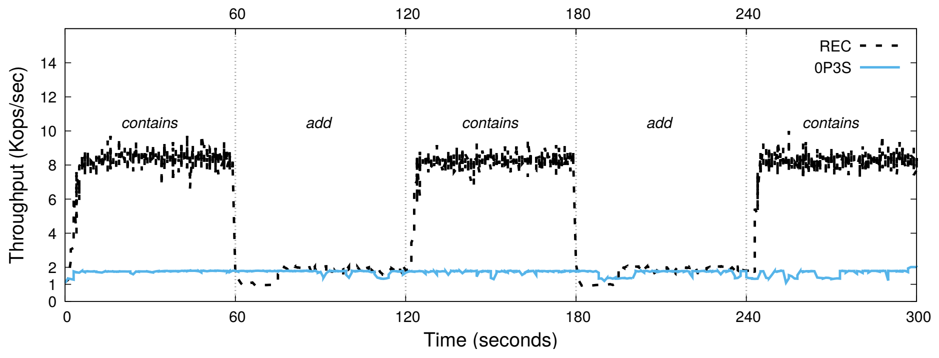


Fig. 3. Parallel vs. sequential replica throughput for the mixed workload.



(a) Sequential vs. Parallel



(b) Sequential vs. Reconfigurable

Fig. 4. Throughput for the mixed workload and reconfigurations.

configurations outperform each other in the presence of non-conflicting and conflicting requests, respectively. This means that, unlike a traditional P-SMR [5], [6], hybrid approaches that mix replicas with different degrees of parallelism provide good performance for both workloads. Unfortunately, however, the hybrid approach has two serious drawbacks: *(i)* the resources of the replicas that lag behind could exhaust rapidly, making the replicas crash or drop requests; and *(ii)* it does not provide the best performance for mixed workloads.

Figure 3 depicts the behavior of a parallel and a sequential replica of configuration `1P2S` for the mixed workload. The figure shows the sequential replica became substantially delayed while executing $contains$ requests and only executed the first add request at time 250 while the parallel replica executed the first add request at time 60 seconds. From second 60 to 250, the sequential replica stored approximately $650k$ operations in its buffer, the number of operations executed by the parallel replica in this period. For workloads with enough non-conflicting requests, this situation would eventually lead to a buffer overflow. Ironically, due to the delay, the sequential replica does not help improve performance when requests conflict and the system will run at the pace of the parallel replica, even though the parallel replica is not the best choice for conflicting requests. We conclude that one cannot optimize performance of state machine replication by mixing replicas with different degrees of parallelism.

C. Reconfiguration in action

We used the same linked list implementation, configurations and the mixed workload of the previous experiments to show how reconfiguration can boost performance. Figure 4 presents the throughput for sequential (`0P3S`), parallel (`3P0S`) and reconfigurable (`REC`) executions. The hybrid systems (`1P2S` and `2P1S`) presented a behavior similar to the parallel execution since the performance was determinated by parallel replicas (see previous discussion about Figure 3). In the reconfigurable execution, the system started with only 1 active thread and used the policy of Algorithm 7 to activate/deactivate threads, ranging from 1 to 10 active threads. Figure 2 (mixed columns) shows the average throughput and latency perceived by clients.

Reconfiguration improves system performance and, at the same time, saves resources since threads are deactivated when they are unnecessary and their presence may negatively impact performance. For example, at time 60 seconds when clients start to invoke only dependent requests, performance drops to approximately 0.8 Kops/sec and the system starts to deactivate threads. After approximately 10 seconds, it remains with only one active thread and the throughput becomes similar to a sequential execution (approximately 1.8 Kops/sec). Notice that the policy will define the time to react after a workload change.

VII. Conclusions

This paper reports on our efforts to increase the performance of parallel state machine replication by dynamically adapting the degree of parallelism of replicas. Dynamic reconfiguration is important since when it comes to setting the ideal degree of parallelism for a workload, there is no "one-size-fits-all" solution. Configurations with few threads perform well with conflicting operations, but cannot reach high performance in the absence of conflicts; conversely, configurations with many threads maximize performance in the absence of conflicts, but introduce too much overhead when handling conflicting requests. Moreover, we show that combining replicas with different degrees of parallelism introduces important drawbacks.

Acknowledgments

This work is supported in part by CAPES (Brazil) and CNPq (Brazil) through projects Scalable Dependability (PVE 88887.124751/2014-00) and FREESTORE (Universal 457272/2014-7), respectively.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[2] F. B. Schneider, "Implementing fault-tolerant service using the state machine aproach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[3] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programing Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[4] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *IEEE/IFIP Int. Conference on Dependable Systems and Networks*, 2004.

[5] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *IEEE International Conference on Distributed Computing Systems*, 2014.

[6] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *IEEE Int. Symposium on Reliable Distributed Systems*, 2014.

[7] M. Zbierski, "Parallel byzantine fault tolerance," in *Soft Computing in Computer and Information Science*, ser. Advances in Intelligent Systems and Computing, A. Wilinski, I. E. Fray, and J. Pejas, Eds. Springer Publishing, 2015, vol. 342, pp. 321–333.

[8] L. H. Le, C. E. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[9] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, S. Mullender, Ed. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 97–145.

[10] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[12] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[13] M. Castro and B. Liskov, "Practical byzantine fault-tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[14] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *ACM Symposium on Operating Systems Principles*, 2005.

[15] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2.

[16] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010.

[17] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ-Replication: A hybrid quorum protocol for byzantine fault tolerance," in *Symposium on Operating Systems Design and Implementation*, 2006.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems*, vol. 27, no. 4, pp. 7:1–7:39, Dec. 2009.

[19] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *European Conference on Computer Systems*, 2010.

[20] C. E. Bezerra, F. Pedone, and R. V. Renesse, "Scalable state-machine replication," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[21] D. da Silva Boger, J. da Silva Fraga, and E. Alchieri, "Reconfigurable scalable state machine replication," in *Latin-American Symposium on Dependable Computing*, 2016.

[22] Z. Guo, C. Hong, M. Yang, L. Zhou, L. Zhuang, and D. Zhou, "Rex: Replication at the speed of multi-core," in *European Conference on Computer Systems*, 2014.

[23] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos made transparent," in *ACM Symposium on Operating Systems Principles*, 2015.

[24] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 97–108, 2009.

[25] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: execute-verify replication for multi-core servers," in *Symposium on Operating Systems Design and Implementation*, 2012.

[26] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: Optimistic deterministic multithreading." in *Workshop on Hot Topics in System Dependability*, 2010.

[27] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[28] R. Friedman and R. van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *ACM Symposium on High-Performance Parallel and Distributed Computing*, 1997.

[29] O. M. Mendizabal, R. T. S. Moura, F. L. Dotti, and F. Pedone, "Efficient and deterministic scheduling for parallel state machine replication," in *IEEE International Parallel & Distributed Processing Symposium*, 2017.

[30] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, pp. 63–73, Mar. 2010.

[31] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 103–115, 2006.

[32] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Symposium on Operating Systems Design and Implementation*, 2002.