Boosting State Machine Replication with Concurrent Execution

Eduardo Alchieri[1], Fernando Dotti[2], Parisa Marandi[3], Odorico Mendizabal[4] and Fernando Pedone[5]
[1]Departamento de Ciência da Computação, Universidade de Brasília, Brazil
[2]Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, Brazil
[3]Microsoft, UK
[4]Universidade Federal do Rio Grande, FURG, Brazil
[5]Universitá della Svizzera italiana (USI), Switzerland

***Abstract*—State machine replication is a fundamental technique to render services fault tolerant. One of the key assumptions of state machine replication is that replicas must execute operations deterministically. Deterministic execution often translates into sequential execution of requests at replicas. With the increasing demand for dependable services and widespread use of multi-core servers, several proposals for enabling concurrent execution in state machine replication have appeared in the literature. Invariably, these techniques exploit the fact that independent operations, those that do not share any common state or do not update shared state, can execute concurrently. Existing protocols differ in several important ways. In this paper, we survey this field of research and discuss the main aspects of the different protocols. Central aspects include conflict detection, representation and enforcing; tradeoffs involving existing architectures and level of allowed parallelism; workload-driven adaptation schemes; and implications of parallel state machine replication to recovery. Moreover, we discuss ongoing and future work directions for high-throughput state machine replication.**

I. Introduction

State machine replication (SMR) is a conceptually simple, yet effective approach to rendering systems fault-tolerant. The basic idea is that server replicas execute client requests deterministically and in the same order [20], [29]. Consequently, replicas transition through the same sequence of states and produce the same output. State machine replication can tolerate a configurable number of faulty replicas. Moreover, application programmers can focus on the inherent complexity of the application, while avoiding the difficulty of handling replica failures [9]. Not surprisingly, the approach has been successfully used in many contexts (e.g., [4], [10], [15]).

Modern multi-processor servers challenge the state machine replication model since deterministic execution of requests often leads to single-threaded replicas. To overcome this limitation, a number of techniques have been proposed to allow multi-threaded execution of requests in state machine replication (e.g., [1], [2], [11], [17], [18], [25]). Most existing techniques build on the observation that *independent* requests can execute concurrently while *conflicting* requests must be serialized and executed in the same order by the replicas—two requests conflict if they access common state and at least one of them updates the state, otherwise requests are independent.

Existing proposals differ on how dependency-based scheduling is performed to provide concurrent execution of independent requests and serialize conflicting requests. Based on existing proposals, there are three classes of protocols:

- *Late scheduling protocols:* Requests are scheduled for execution after they are ordered. This essentially means that requests are scheduled at the replicas. Besides the aforementioned requirement on conflicting requests, there are no further restrictions on scheduling.
- *Early scheduling protocols:* Part of the scheduling decisions are made before requests are ordered (e.g., the request must be executed by a subset of the existing worker threads). After requests are ordered, their scheduling at each replica must respect these restrictions (i.e., scheduling at the replicas determines which thread in the defined subset will execute the request).
- *Static scheduling protocols:* Scheduling decisions are made before requests are ordered for execution. Thus, there is no request scheduling at the replicas.

In this paper, we conjecture that there is an inherent tradeoff between the moment scheduling decisions are made and the efficiency of a particular approach. Intuitively, this seems to hold because postponing scheduling decisions allows to account for more information and maximize parallelism. We provide examples to convey our intuition and use different efficiency metrics to compare different techniques. Using the same metrics, we show that scheduling conflicting requests deterministically, a correctness requirement of state machine replication, leads to schedulers that are less efficient than non-deterministic schedulers.

Since late scheduling protocols are supposedly more efficient than their counterparts, one may wonder the raison-d'être of static and early scheduling techniques. To shed some light in the matter, we discuss scheduling techniques from different perspectives. For example, it turns out that tracking dependencies among requests in late scheduling may lead to scheduling bottlenecks. This is the case if requests are "light," that is, the computing effort to execute a request and to schedule the request are comparable. In addition to scheduling overhead, we also consider reconfiguration and recovery.

This paper makes the following contributions: (a) We place replication in the context of distributed applications (§II) and briefly cover the main approaches to concurrency in state

machine replication (§III). (b) We present in detail the main scheduling techniques proposed in the literature for parallel state machine replication (§IV). (c) We compare these techniques according to different criteria (§V). (d) We discuss proposals that are related to the covered techniques (§VI). (e) We discuss directions for future work (§VII).

II. Background

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

A. The role of replication in distributed applications

Large distributed systems are typically structured in tiers, a three-tier system being a prototypical case (see Figure 1). In these environments, users are at the top tier and submit requests to application servers, at the middle tier. Application servers execute the application logic and submit requests to the bottom tier. The bottom tier is responsible for handling the application state. In this context, SMR clients are the application servers in the middle tier and SMR replicas are the servers in the bottom tier. In this paper, we refer to clients and servers from the perspective of state machine replication.

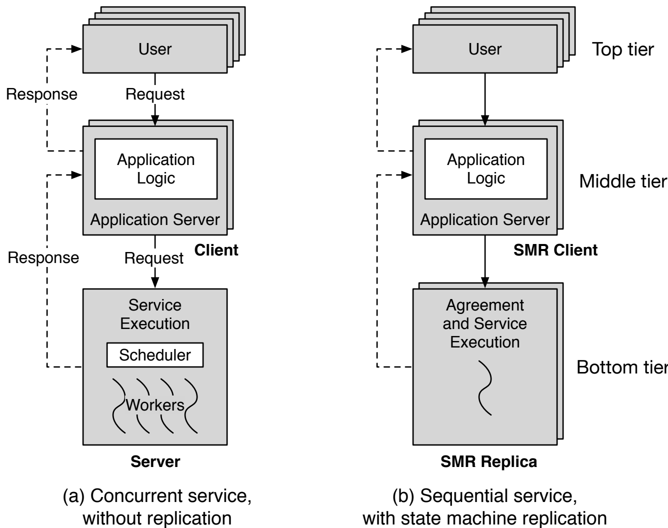

Fig. 1. Three-tier application (a) without replication and (b) with replication.

B. Agreement and execution in SMR

SMR renders a service fault-tolerant by replicating the server and coordinating the execution of client commands among the replicas [20], [29]. The service is defined by a state machine and consists of *state variables* that encode the state machine's state and a set of *commands* that change the state (i.e., the input). The execution of a command may (i) read state variables, (ii) modify state variables, and (iii) produce a response for the command (i.e., the output). Commands are *deterministic*: the changes to the state and the response of a command are a function of the state variables the command reads and the command itself.

SMR requires replicas to execute commands in the same order. Therefore, before commands are executed by the replicas, the replicas must agree on the execution order. These order requirements can be encapsulated by an atomic broadcast communication abstraction, defined by primitives $broadcast(m)$ and $deliver(m)$, where m is a message. Atomic broadcast ensures the following properties [8], [12][1]:

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Uniform Agreement*: If a process delivers a message m, then all correct processes eventually deliver m.
- *Uniform Integrity*: For any message m, every process delivers m at most once, and only if m was previously broadcast by a process.
- *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'.

C. Correctness

SMR provides clients with the abstraction of a highly available service while hiding the existence of multiple replicas. This last aspect is captured by *linearizability*, a consistency criterion [13]: a system is linearizable if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specification, and (ii) respects the real-time ordering of commands across all clients.

III. Overview of approaches to parallel state machine replication

In this section we review the basics of state machine replication and overview proposals that have introduced concurrency in state machine replication.

A. Classic SMR

State machine replication provides clients with the illusion of a non-replicated service, that is, replication is transparent to the clients. Clients broadcast commands to all replicas and wait for the response from one replica (see Figure 2 (a)). Before requests can be executed on the replicas they are ordered by the agreement layer. Since replicas execute commands deterministically and in the same order, every replica produces the same response after the execution of the same command. Differently from a non-replicated service, clients remain oblivious to failures, as the service is operational despite the failure of some of its replicas (i.e., up to f faulty replicas). In failure-free scenarios, however, a non-replicated service is often more efficient than a replicated service since in the replicated case requests reach the servers through the agreement layer and execution is single-threaded.

[1]Atomic broadcast needs additional synchrony assumptions to be implemented [6], [9]. These assumptions are not explicitly used by the protocols proposed in this paper.

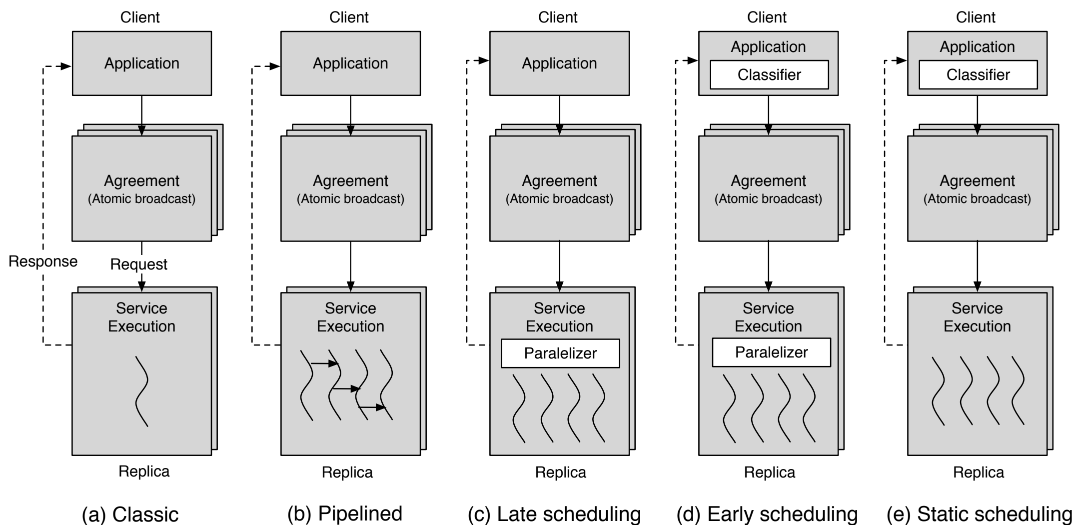

Fig. 2. Different techniques to implement state machine replication: (a) sequential execution; (b) pipelined execution; and concurrent execution with (c) late scheduling, (d) early scheduling, and (e) static scheduling.

B. Pipelined SMR

Having replicas execute commands sequentially by a single thread does not imply that the whole replica's logic must be single-threaded; multiple threads on a replica can cooperatively handle the requests. For example, one thread receives the requests, another executes the requests, and a third thread responds to the clients. In [28], the authors propose a pipelined architecture to exploit the processing power of multicore servers. The agreement layer (atomic broadcast) and the replicas are organized as a collection of modules connected through shared message queues where messages are totally ordered (see Figure 2 (b)). Although staging improves the throughput of state machine replication, there is always only one thread sequentially executing the commands.

C. Late scheduling

It has been observed that a replica can execute commands that access disjoint variables (independent commands) concurrently without violating consistency [29]. The notion of command interdependency is application-specific and must be provided by the application developer or automatically extracted from the service code. Recently, several replication models have exploited command dependencies to parallelize the execution on replicas. We discuss these techniques in this and in the next sections.

To benefit from command inter-dependencies to parallelize execution, some proposals add a parallelizer (i.e., deterministic scheduler) to the replicas [19]. The parallelizer delivers all the commands ordered through the agreement layer, examines command dependencies, and distributes them among a pool of worker threads for execution (see Figure 2 (c)). To distribute the commands among threads, besides considering dependencies, the scheduler can also balance the load among threads. Threads that are less occupied can be given more commands to execute if their execution does not conflict with the commands that are being executed by other threads.

D. Early scheduling

The parallelizer in late scheduling sequentially delivers and assigns commands for execution to the worker threads. To ensure that worker threads have balanced load and command dependencies are not violated, the parallelizer and the worker threads may share a graph data structure, where commands are vertices and command interdependencies are edges. In this scheme, the parallelizer adds vertices and edges to the graph, upon delivery, and the worker threads remove them from the graph, after execution. In some workloads, this shared data structure may introduce significant overhead. The idea of early scheduling [1], [2] is to make part of the scheduling decisions before commands reach the parallelizer, and thus reduce scheduling overhead (see Figure 2 (d)). For example, it can be agreed that commands involving an object x must be executed by a worker thread in a subset of all worker threads. When an operation on x is delivered, the parallelizer needs to coordinate with worker threads in the assigned subset, not with all worker threads.

E. Static scheduling

Static scheduling is a more extreme version of early scheduling, in which the worker thread to execute a command is decided when the command is broadcast for execution. P-SMR [23] is an example of static scheduling. P-SMR has no parallelizer or scheduler and worker threads on replicas concurrently deliver and execute multiple disjoint streams of ordered commands. To preserve correctness, commands in each stream must be independent from the commands in any other stream. To ensure independency among the concurrently delivered streams, unlike previous approaches in which command dependencies are determined at the replicas, in P-SMR command dependencies are determined by the clients, before commands are ordered. Commands in P-SMR are ordered by an atomic multicast library and clients multicast independent commands to different multicast groups. P-SMR implements

a fully parallel model in which independent commands are ordered, delivered, and executed in parallel. Dependent commands are ordered through dedicated multicast groups and executed sequentially (see Figure 2 (e)).

IV. Scheduling techniques in detail

In this section, we formalize the notion of conflict and detail the three scheduling approaches that exploit request independency to introduce concurrency in SMR.

A. The notion of conflict

State machine replication determines how service operations must be propagated to and executed by the replicas. Typically, (i) every correct replica must receive every operation; (ii) no two replicas can disagree on the order of received and executed operations; and (iii) operation execution must be deterministic: replicas must reach the same state and produce the same output upon executing the same sequence of operations. Even though executing operations in the same order and serially at replicas is sufficient to ensure consistency, it is not necessary.

Let R be the set of requests available in a service (i.e., all the requests that a client can issue). A request can be any deterministic computation involving objects that are part of the application state. We denote the sets of application objects that replicas read and write when executing a request r as r's *readset* and *writeset*, or $RS(r)$ and $WS(r)$, respectively. We define the notion of conflicting requests as follows.

Definition 1 (Request conflict)**.** The conflict relation $\#_R \subseteq R \times R$ among requests is defined as

$$(r_i, r_j) \in \#_R \text{ iff } \begin{pmatrix} RS(r_i) \cap WS(r_j) \neq \emptyset \vee \\ WS(r_i) \cap RS(r_j) \neq \emptyset \vee \\ WS(r_i) \cap WS(r_j) \neq \emptyset \end{pmatrix}$$

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*. Consequently, if two requests are independent (i.e., they do not share any objects or only read shared objects), then the requests can be executed concurrently at replicas (e.g., by different worker threads at each replica). Concurrent execution of requests raises the issue of how requests are scheduled for execution on worker threads. We distinguish between three categories of protocols.

B. Late scheduling

In this category of protocols, replicas deliver requests in total order and then a parallelizer at each replica assigns requests to worker threads for execution. The parallelizer must respect dependencies between requests. More precisely, if requests r_i and r_j conflict and r_i is delivered before r_j, then r_i must execute before r_j. If r_i and r_j are independent, there are no restrictions on their scheduling (e.g., the parallelizer can assign each request to a different worker thread).

CBASE [19] is a protocol in this category. The parallelizer at each replica delivers requests in total order and includes them in a dependency graph where vertices represent delivered but not yet executed requests and directed edges represent dependencies among them. Request r_i depends on r_j (i.e., $r_i \rightarrow r_j$ is an edge in the graph) if r_i is delivered after r_j, and they conflict. The dependency graph is shared with a pool of worker threads that choose requests for execution from the graph respecting their interdependencies: a request can be executed if it is not under execution and does not depend on any requests in the graph. After executing it, the worker thread removes the request from the graph and chooses another one.

With CBASE each node of the graph represents a single command submitted to the SMR. While it allows maximum concurrency to be achieved, because it represents the needed and sufficient information to start each command concurrently, each command has to be compared for conflict with each other one on graph insertion. For light commands, the cost of this operation may exceed command execution. In [25], the authors investigate the effect of batching on late scheduling. Instead of submitting a command at a time, clients broadcast batches of commands to the replicas. Dependencies are represented among batches and not commands. This technique reduces the overhead of dependency tracking but possibly serializes independent commands: If any two commands in different batches conflict, then the two batches conflict and are serialized according to their delivery order. Two independent commands included in these batches will also be serialized. To speed up batch conflict detection, in [25] batches carry a bitmap that encodes variables accessed by the commands in the batch. An intersection between two batch bitmaps means the batches conflict. This technique may include false positives since different variables may map to the same bits in the map. Although false positives reduce concurrency, they do not violate safety. In conflict-free scenarios this technique is shown to drastically increase throughput. As conflicts arise, throughput drops but still outperforming CBASE.

1) Replica execution model: The late scheduler adopts the following replica execution model:

i. All the replicas have $n+1$ threads: one scheduler thread and n worker threads.
ii. All threads access a common dependency graph to insert (scheduler) and to get/remove (workers) requests.
iii. The scheduler delivers requests in total order and inserts each request in the dependency graph, according to the conflict dependencies.
iv. Each worker selects a request from the graph if it is available for execution, marking the request as executing, and then removes it from the graph after execution.

Algorithms 1 and 2 represent the execution model for the scheduler and worker threads, respectively. Whenever a request is delivered by the atomic broadcast protocol, the scheduler (Algorithm 1) inserts it in the dependency graph. While doing that, the scheduler must have exclusive access to the graph. Moreover, developers must provide a function to indicate whether two requests conflict, following Definition 1. Each worker thread (Algorithm 2) gets one available request from the graph and marks it as in execution. After execution, the request is removed from the graph together with the edges indicating its dependencies by the worker thread. The workers

must have exclusive access to the graph both to select and to remove a request.

Algorithm 1 Late scheduler.

```
1: variables:
2:     graph ← ∅                                   // the dependency graph
3: on deliver(req):
4:     graph.insert(req);        // insert the request and its dependencies
```

Algorithm 2 Worker threads for late scheduling.

```
1: variables:
2:     graph ← ∅                                   // the dependency graph
3: while true do
4:     req ← graph.next();            // get the next available request
5:     exec(req)                                      // execute request
6:     graph.remove(req);     // remove the request and its dependencies
```

2) Liveness: From the total delivery order and the conflict relation, a directed acyclic graph emerges representing dependencies. The fact that the graph is a DAG ensures that, at any point in time, if the subset of independent commands is processed, necessarily other commands become free for execution. This is true for the initial state (no previous command) and inductively for the whole computation, ensuring liveness.

3) Safety: Since dependent commands are executed in the same order across replicas, whenever any two replicas have executed the same set of commands they have modified the state shared by any two commands in the same order and thus reach the same replica state.

C. Early scheduling

In the early scheduling approach [1], [2], clients tag a request with its class identifier, before it is ordered, to identify how it should be processed. A scheduler thread delivers requests in total order and assigns, based on the tag, each request to one or more threads, as detailed below. This association needs knowledge of the application. For example, one could establish that all requests that access object x are executed by thread t_0 and all requests that access object y are executed by thread t_1; requests that access both objects require threads t_0 and t_1 to coordinate so that only one thread executes the request. The advantage of early scheduling is that since scheduling decisions are simple at the replicas (e.g., there is no dependency graph), the scheduler is less likely to become a performance bottleneck.

In [1] the notion of request classes is introduced to denote concurrency and dependencies among requests, thus encoding application knowledge. Given a service with a set R of possible requests, a set C of classes is defined and, for each class of C: (i) a non-empty non-overlapping subset of requests from R is associated; and (ii) a subset of conflicting classes is associated. A third construction rule is that each request is associated to exactly one element of C. A conflict among classes happens when any two requests from those classes conflict, according to the conflict definition $\#_R$ above. Requests that belong to conflicting classes have to be serialized according to the total order induced by atomic broadcast while independent classes can be executed concurrently. Notice that a class may be self-conflicting, meaning that its requests have to be serialized.

1) Replica execution model: The early scheduler adopts the following replica execution model to support request classes:

i. All the replicas have $n+1$ threads: one scheduler thread and n worker threads.
ii. Each worker thread has a separate input queue and removes requests from the queue in FIFO order.
iii. The scheduler delivers requests in total order and dispatches each request r to one or more input queues:
   a. If scheduled to one worker only, r can be processed concurrently with other requests.
   b. If scheduled to more than one worker thread, then r depends on preceding requests assigned to these workers. Therefore, all workers involved in r must synchronize before one worker, among the ones involved, executes r.

2) Class to threads mapping: With this execution model, the following class-to-thread-mapping rules can be applied to ensure linearizable executions:

i. *Every class is associated with at least one worker thread,* to ensure that requests are eventually executed.
ii. *If a class is self-conflicting, it is sequential:* Each request is scheduled to all threads of the class and processed as described in the previous section.
iii. *If two classes conflict, at least one of them must be sequential.* The previous requirement may help decide which one.
iv. *For conflicting classes c_1, sequential, and c_2, concurrent, the set of workers associated to c_2 must be included in the set of workers associated to c_1.* This requirement ensures that requests in c_2 are serialized w.r.t. c_1's.
v. *For conflicting sequential classes c_1 and c_2, it suffices that c_1 and c_2 have at least one worker in common.* The common worker ensures that requests in the classes are serialized.

These rules result in several possible class-to-threads mappings. More concretely, a mapping is defined as follows.

Definition 2 (CtoT). $CtoT = C \rightarrow \{Seq, Cnc\} \times \mathcal{P}(T)$ where: C is the set of class names; $\{Seq, Cnc\}$ is the sequential or concurrent synchronization mode of a class; and $\mathcal{P}(T)$ the possible subsets of $T = \{t_0, .., t_{n-1}\}$, the n worker threads at a replica.

With a $CtoT$, Algorithms 3 and 4 present the execution model for the scheduler and worker threads, respectively. Whenever a request is delivered by the atomic broadcast protocol, the scheduler (Algorithm 3) assigns it to one or more worker threads. If a class is sequential, then all threads associated with the class receive the request to synchronize the execution (lines 4–6). Otherwise, requests are associated to a unique thread (line 7–8), following a round-robin policy (function $next$).

Algorithm 3 Early scheduler.

```
1: variables:
2:     queues[0, ..., n − 1] ← ∅                    // one queue per worker thread
3: on deliver(req):
4:     if req.class.smode = Seq then                 // if execution is sequential
5:         ∀t ∈ CtoT(req.classId)                    // for each conflicting thread
6:             queues[t].fifoPut(req)                // synchronize to exec req
7:     else                        // else assign req to one thread in round-robin
8:         queues[next(CtoT(req.classId))].fifoPut(req)
```

Algorithm 4 Worker threads for early scheduling.

```
1: variables:
2:     myId ← id ∈ {0, ..., n − 1}                   // thread id, out of n threads
3:     queue[myId] ← ∅               // the queue with requests for this thread
4:     barrier[C]                                    // one barrier per request class
5: while true do
6:     req ← queue.fifoGet()                         // wait until a request is available
7:     if req.class.smode = Seq then                 // sequential execution:
8:         if myId = min(CtoT(req.classId)) then     // smallest id:
9:             barrier[req.classId].await()          // wait for signal
10:            exec(req)                             // execute request
11:            barrier[req.classId].await()          // resume workers
12:        else
13:            barrier[req.classId].await()          // signal worker
14:            barrier[req.classId].await()          // wait execution
15:    else                                          // concurrent execution:
16:        exec(req)                                 // execute the request
```

Each worker thread (Algorithm 4) takes one request at a time from its queue in FIFO order (line 6) and then proceeds depending on the synchronization mode of the class. If the class is sequential, then the thread synchronizes with the other threads in the class using barriers before the request is executed (lines 8–14). In the case of a sequential class, only one thread executes the request. If the class is concurrent, then the thread simply executes the request (lines 15–16).

3) Safety: Since each request is associated to a class and classes have at least one thread, each request is enqueued to at least one thread. The thread queues are compatible with the total order since the scheduler delivers and assigns requests to threads in delivery order. Whenever two requests conflict, they belong to conflicting classes. The mapping enforces that these conflicting classes are either both sequential or at least one of them is sequential. In the first case threads of both classes process sequentially, and since they have at least one overlapping thread, this one thread synchronizes the threads from both classes. As the input queue of the threads is compatible with the total delivery order, so is the request processing. In the second case, all the threads of the concurrent class are also threads of the sequential one. Thus, whenever a request to the sequential class is issued, it is assigned also to all the threads of the concurrent class. Since this assignment is performed by the scheduler in delivery order, any concurrent requests previous (or subsequent) to the sequential one will be executed before (or after) preserving the total order. Moreover, it was showed in [1], [2] that if replicas use the same class definition, irrespective of the number of threads, a correct class-to-thread mapping will enforce the correct ordering of conflicting requests.

4) Liveness: At any point in time, the first element of the queue of a worker thread either is a unique request to that thread or it appears in different thread queues. In the first case, according to the execution model and mapping, the request is independent and can be executed. In the second case, all the threads with that request have to synchronize to execute it. Since all input queues are compatible with the delivery order, any request previous to this one appears before it in all needed threads and therefore can be processed before it, which guarantees that the request will eventually be the first in all queues where it appears and thus can be executed (i.e., the approach is deadlock-free).

D. Static scheduling

The main idea of static scheduling [23] is to remove scheduling decisions from the replicas. Consequently, there is no scheduler that could become a performance bottleneck at the replicas. Similar to the early scheduler, clients tag requests with the ids of the worker threads that will execute the requests. At the replica side, the atomic broadcast primitive uses the request tag to deliver requests directly to worker threads, without involving a scheduler. If two requests conflict, then they must be addressed to at least one common thread, which will impose a sequential execution on the requests. Independent requests can be addressed to different threads to be executed in parallel.

We call this technique static scheduling because clients directly choose the worker thread to execute their requests and this mapping does not change. This is differently from the previous two techniques, which implement dynamic scheduling. In early scheduling, clients identify the requests classes but the final mapping is done by the scheduler at the replicas.

1) Replica execution model: The static scheduler adopts the following replica execution model:

i. All the replicas have n worker threads.
ii. Clients tag requests with the ids of one or more worker threads and the atomic broadcast primitive delivers requests directly to the threads, according to the ids contained in the requests.
iii. If a request is delivered to one worker only, then the request can be processed concurrently with requests delivered in other worker threads.
iv. If two or more worker threads deliver the same request r, it is because r depends on preceding requests delivered by these workers. Therefore, all workers involved in r must synchronize before one worker, among the ones involved, executes r.

Algorithm 5 presents the execution model for the worker threads. Whenever a request is delivered by the atomic broadcast protocol, the worker thread identifies if it can execute the request without any synchronization, i.e., when the client tagged a request with only its identifier (lines 4-6). Otherwise, the thread must synchronize with the other threads involved in the conflict. For this purpose, threads communicate by signals and the thread with the lowest identifier executes the request while the other threads wait for the execution (lines 7-17).

Algorithm 5 Worker threads for static scheduling.

```
1:  variables:
2:     myId ← id ∈ {0,...,n−1}                    // thread id, out of n threads
3:  on deliver(req):
4:     ids ← tags(req)                              // get the threads ids
5:     if |ids| == 1 then                //request assigned to only this thread
6:        exec(req)                                       // execute request
7:     else                                  // conflict: more threads involved
8:        e ← min(ids)
9:        if e = myId then
10:          for j ∈ ids : j ≠ myId do
11:             wait for signal from j               // waits all threads to stop
12:          exec(req)                                   // execute request
13:          for j ∈ ids : j ≠ myId do
14:             signal j                                 // resumes all threads
15:       else
16:          signal e           // signals thread with lowest id to execute
17:          wait for signal from e                       // waits the execution
```

2) Liveness and Safety: Due to the similarities with the early scheduling strategy, liveness and safety can be analogously argued for static scheduling.

V. Analysis of scheduling techniques

This section briefly compares the various approaches to parallel state machine replication according to different criteria.

A. Scheduling efficiency

Now we discuss how different scheduling techniques compare using three criteria: average turnaround time, percentage of time processors are idle in a two-processor system, and makespan. The turnaround time is the interval from the time the request is delivered at a replica to the time it is executed; makespan is the total time needed to complete all requests.

We consider four requests: three read requests on different variables, $R(x)$, $R(y)$, and $R(z)$, and a write request $W(x, z)$, delivered in this order. Moreover, the requests have all been delivered at the replica when scheduling starts. The requests take 2, 4, 10 and 1 time units to be executed, respectively. The only conflict is between the write request and the read requests on x and z. Figure 3 shows the results for a concurrent nondeterministic scheduling, classic state machine replication, late scheduling, and two cases for early scheduling and static scheduling.

1) Determinism vs. non-determinism: In Figure 3, nonreplicated case, consider that requests arrived in the same order as they are delivered in the other techniques, that is, $R(x)$, $R(y)$, $R(z)$, and $W(x, z)$. If the service is not replicated, as long as conflicting requests are serialized, any possible order is correct, even if conflicting requests are reordered with respect to their arrival order. Reordering requests in this case does not harm linearizability since the reordered requests have not been answered yet; thus clients can not observe any realtime dependencies.

State machine replication, however, restricts concurrency to impose the same relative order of conflicting requests on replicas. Therefore, in this case, conflicting requests have to follow the total delivery order. In fact, this is the case for all scenarios of Figure 3 with replication, where $R(x)$ and $R(z)$ must precede $W(x, z)$.

We detail how the three metrics were computed for the nonreplicated case only, as the calculation for the other cases is similar. The turnaround time for requests $R(x)$, $R(y)$, $R(z)$, and $W(x, z)$ is, respectively, 1, 3, 7, and 10 time units, for an average of 5.25; this was the lowest average turnaround time for the techniques shown. In this execution, two processors were used during 10 time units each, for a total of 20 time units; out of this time, thread $T1$ was idle for 3 time units, resulting in 15% of idle time. Thread $T2$ determines the makespan of the execution since it is the last one to finish, at time 10.

2) Late scheduling: As already mentioned, the late scheduling technique detects pairwise dependencies among requests, representing them as a DAG. The DAG has enough information to optimize concurrency: for each request in the dependency graph, each other previous pending request that it depends on is identified. Since the unity of execution is a request and dependencies are pairwise among requests, any possible concurrency among execution units can be identified. Figure 3, late scheduling, shows one possible way to schedule requests that respects request dependencies.

3) Early scheduling: The early scheduling technique, as mentioned, allows to group requests in request classes to handle them with common restrictions. Whenever we group requests in coarse classes and not represent finer independence details, we may serialization of requests, which is safe but reduces performance. Moreover, even if all possible independence is represented by the class definition, for instance using one class per request and thus being equivalent to the granularity of late scheduling, the class to thread mapping is fixed and may also induce serialization while late scheduling could take any free thread from a pool to execute the next free request. Figure 3 shows two cases for early scheduling. Case 1 shows a situation where there is a concurrent class for read operation, associated to threads $T1$ and $T2$, and a sequential class for write operations. The class for writes conflicts with the read class, thus synchronizing all threads. Case 2 depicts the situation where reads and writes to x and z are handled in a sequential request class independent from accesses to y.

4) Static scheduling: Figure 3 also shows two cases for static scheduling. In Case 1, requests on x and y are assigned to thread $T1$, requests on z are assigned to thread $T2$, and requests on x and z are assigned to both threads, which must synchronize before the request is executed by one of the threads (in the example, $T1$). In Case 2, requests that access x and z are scheduled for execution at $T1$, and requests on y are scheduled for execution at $T2$.

B. Scheduling overhead

As concurrency may increasingly drop following the techniques presented above, so does the scheduling overhead at server side. The overhead of the scheduler at server side is important since it is a unique point through which all requests have to pass.

	Average turnaround time	%idle	Makespan
Concurrent No replication	**5.25** (1)	**15%** (1)	**10** (1)
Sequential	**11.75** (6)	**50%** (6)	**17** (6)
Late scheduling	**5.75** (2)	**29%** (3)	**12** (3)
Early scheduling Case 1	**7.25** (4)	**23%** (2)	**11** (2)
Early scheduling Case 2	**6.25** (3)	**15%** (1)	**10** (1)
Static scheduling Case 1	**7.75** (5)	**35%** (4)	**13** (4)
Static scheduling Case 2	**6.25** (3)	**15%** (1)	**10** (1)

Fig. 3. Executions of different scheduling techniques (for each metric, the relative position of each technique is shown between parenthesis).

Static scheduling imposes no scheduling overhead at server side while in early scheduling a choice is made to which thread to dispatch (in case of independent requests) or to which set of threads to dispatch (in case of conflicting requests). In any case, requests are enqueued for execution. Assuming that each worker thread has its queue, contention at these data structures is limited.

With late scheduling, a central structure (the DAG) is kept and the inclusion of a new request is $O(n)$, where n is the size of the graph. Also, the more elements in the graph, the more conflicts may arise. This may lead to contention for this structure while worker threads may try to access it to take free requests for processing.

C. *Reconfiguration*

Reconfiguration is the process of modify the multiprogramming level (MPL) of a parallel SMR, i.e., the number of worker threads [1]. Although the MPL does not significantly affect the performance of the late scheduler, since all needed synchronization is executed in the dependency graph, it impacts the performance of early [1] and static [23] schedulers because synchronization among the threads involved in a conflict is needed: a high MPL increases performance in workloads with predominantly non-conflicting requests, but negatively impacts performance if conflicting requests are the norm; conversely, a low MPL excels in workloads dominated by conflicting requests, but underperforms in the presence of non-conflicting requests. The main idea of reconfigurations is to keep the MPL according to the current workload.

Reconfiguring the MPL of a late scheduler is straightforward, as all synchronization is enforced by the graph. Thus, threads can be spawned and killed without coordination. In the early scheduler, it is necessary to synchronize the servers before executing a reconfiguration since the scheduler at the replicas executes the final mapping of requests to worker threads [1]. Finally, the viability of reconfiguration of a static scheduler is still an open problem: it is necessary to synchronize the clients since they assign requests directly to the worker threads.

D. *Recovery*

Parallel state machine replication renders recovery particularly challenging since throughput under normal execution (i.e., in the absence of failures) is expected to be high [23], [25]. Consequently, the log of commands that a recovering replica needs to apply to catch up with operational replicas may be large, which delays recovery. This situation renders the replicated system more vulnerable to failures since a recovering replica is only available once it can process new client commands.

Traditionally, recovering a crashed server boils down to fetching and installing a service checkpoint and retrieving and (re-)executing commands that are not included in the checkpoint. With standard recovery techniques, a recovering replica can only execute "new commands" after it has fetched and installed a checkpoint and retrieved and executed "old commands" (commands that were already ordered and perhaps even executed but are not included in the installed checkpoint).

Parallel state machine replication techniques exploit request interdependencies as already discussed. The same observation that independent commands can execute concurrently is also useful for designing efficient recovery protocols. In [24], the

authors present a recovery protocol that allows new commands to execute concurrently with recovery, before replicas are completely updated: a new command does not need to wait for an old command to be executed if the two are independent. A second optimization is inspired by the fact that a considerable amount of recovery time is due to state transfer and installation. Authors propose to divide a checkpoint into segments, and retrieve and install each segment only when it is needed for the execution of a command. Thus, a segment can be concurrently retrieved and installed with other segments. Checkpoint segments are handled on demand and possibly concurrently.

VI. Related work

In this section, we review other replication approaches related to the ones surveyed in this text. They fall in two categories: (i) optimistic scheduling and (ii) approaches that solve non-determinism during run-time (execution) using different strategies that enforce replicas to have the same trace of causally dependent commands.

Eve [18] proposes a significant depart from the previous techniques. Following an optimistic approach, in Eve replicas first execute the requests and then verify the correctness of the states through a verification stage, hence named as Execute-Verify (EV). Eve distinguishes one of the replicas as the primary to which clients send their requests. The primary replica organizes the requests into batches and assigns to each batch a unique sequence number. The primary then propagates the batched requests to the other replicas. All the replicas, including the primary, are equipped with a deterministic *mixer*. Using the application semantics, the mixer converts a batch of requests to a set of parallel batches such that all the requests in a parallel batch can be executed in parallel. Once the execution of a parallel batch terminates, replicas calculate a token based on their current state and send their token to the verification stage. The verification stage investigates the equality of the tokens. If the tokens are equal, replicas commit the requests and respond to the clients. Otherwise, replicas must roll back the execution and re-execute the requests in the order determined by the primary as it was batching the requests. The verification stage also adds to Eve the advantage of detecting concurrency bugs.

In [17], authors present Storyboard, an approach that supports deterministic execution in multi-threading environments. Their strategy uses a forecasting mechanism that, based on application-specific knowledge, heuristically predicts an ordered sequence of locks across replicas. Authors report forecasts to emerge from off-line executions to identify locking patterns of the application. 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. In this case, Storyboard blocks the execution of the command and repredicts the command's execution path. The *repredict* command runs a consensus protocol to determine a consistent point in the execution order across all replicas. The reprediction will contain at least the lock which the command currently seeks to acquire, but possibly also further locks. All replicas will proceed with the new forecast. This cycle of reprediction is repeated until the command completes.

CRANE [7] is a parallel SMR system that transparently replicates general multi-threaded programs. One important idea of CRANE is to combine the input determinism of Paxos and the execution determinism of deterministic multi-threading (DMT) [14], [26]. This technique maintains a logical time that advances deterministically on each thread's synchronization within a replica. Within each replica, CRANE intercepts POSIX socket and the Pthreads synchronization interface and implement deterministic versions of such synchronizing operations. To ensure total order delivery of synchronization commands across replicas, for each incoming socket call (e.g., `accept()` or `recv()`), CRANE runs a distributed consensus protocol, so that correct replicas see exactly the same sequence of calls. CRANE schedules synchronization commands using deterministic multi-threading (DMT) and assure that requests are processed in the same logical time across replicas. In terms of application design simplicity, CRANE [7] is a good choice. Its drawback is that multithreaded applications with intense synchronization incur higher overhead.

Rex [11] uses an execute-agree-follow strategy. A single server (primary) receives requests and processes them. The execution of each request has to be deterministic, the only source of non-determinism being the concurrent execution of requests. Fine-grained locks are assumed to access shared variables and synchronization events (e.g. lock, unlock) records causality among requests processing in a trace. 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. The Execute-agree-follow model proposed by Rex resembles the passive replication model more than the SMR.

VII. Final remarks

This paper surveyed techniques that boost SMR performance by parallelizing request execution. These techniques improve system performance, when compared to a classic SMR, and introduce tradeoffs. In the following, we briefly overview directions for future work.

1) Late scheduling concurrent graph: Regarding late scheduling, existing proposals rely on a DAG for dependency tracking and enforcing. This DAG is accessed concurrently by the scheduler and worker threads. Existing implementations use a single lock on the whole graph. It is natural to investigate concurrent structures to manipulate this graph with a finer grain locking strategy. Interestingly, we note that concurrent manipulation of graph data structures is a recent research topic [16], [21], [22], [27]. Among existing work, [27] discusses concurrency for a graph structure close to the one used in late scheduling, where both nodes and edges can be created and deleted and the graph is acyclic and directed. Nodes represent transactions and edges represent conflicts among transactions.

The graph is used to calculate serializable executions. Whenever a transaction is added, edges are included to represent conflicts and the graph is checked for cycles. In case of cycles, vertices and edges are removed to keep the graph acyclic.

2) Work stealing in early and static scheduling: In these scheduling techniques, requests are assigned to the queues of the worker threads for execution. Consequently, it is possible that some thread receives more requests than the others, depending on the mapping adopted in the system. This problem is more likely to happen in the static scheduler since the clients define the thread to execute its requests without any information about the system current load. Since requests could have different execution costs, this behavior affects also the early scheduler. Since threads may need to synchronize with overloaded threads, this phenomenon may impact the overall system performance.

Work stealing [3] is a technique that could be adapted to early and static schedulers in order to decrease the load at the overloaded threads. With this approach, one thread can steal the requests assigned to other threads. Consequently, all threads can present balanced load. The main challenge is that threads must steal requests and, at the same time, respect the interdependencies among the requests.

3) Dependency handling: While important scheduling techniques need information about a service's request conflicts, this information has to be declared by the application developer. The problem of identifying possible concurrency among requests is analogous to the problem of parallelizing sequential programs: parts of the program candidate to work in parallel have to be independent such that any relative order among them leads to consistent results. While there are already results towards parallelization of sequential programs, the automatic detection of independent program parts is not trivial, when possible at all. If such detection needs pointer analysis, for instance, this leads to an undecidable problem [5]. Nonetheless, the investigation of the applicability of existing techniques to parallel SMR as well as possible restrictions to the problem to allow automatic conflict detection are open questions.

Besides, although run-time structures and algorithms to detect and track dependencies are in use, they impose important overhead. This is specially the case for exact conflict detection techniques, such as used in late scheduling that impose pairwise comparison with all pending requests. Structures such as bloom-filters or bitmaps reduce overhead but may allow false-positives at rates that compromise performance gains. Designing structures and algorithms that efficiently handle dependencies is an open research problem.

Acknowledgments

This work is supported in part by CAPES (Brazil) and MCTIC/RNP (Brazil) through projects Scalable Dependability (PVE 88887.124751/2014-00) and P4Sec (grant number 002949), respectively.

References

[1] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone. Reconfiguring parallel state machine replication. In *SRDS*, 2017.

[2] E. Alchieri, F. Dotti, and F. Pedone. Early scheduling in parallel state machine replica. In *ACM SoCC*, 2018.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[4] M. Burrows. The chubby lock service for loosely coupled distributed systems. In *OSDI*, 2006.

[5] V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. *SIGPLAN Not.*, 38(1):115–125, Jan. 2003.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

[7] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang. Paxos made transparent. In *SOSP*, 2015.

[8] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, Dec. 2004.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[10] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

[11] Z. Guo, C. Hong, M. Yang, L. Zhou, L. Zhuang, and D. Zhou. Rex: Replication at the speed of multi-core. In *EuroSys*, 2014.

[12] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[13] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[14] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble. DDOS: taming nondeterminism in distributed systems. In *ACM SIGPLAN Notices*, volume 48, pages 499–508, 2013.

[15] J. D. J. C. Corbett and M. E. et al. Spanner: Google's globally distributed database. In *OSDI*, 2012.

[16] N. D. Kallimanis and E. Kanellou. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In *OPODIS*, 2015.

[17] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler. Storyboard: Optimistic deterministic multithreading. In *HotDep*, 2010.

[18] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: execute-verify replication for multi-core servers. In *OSDI*, 2012.

[19] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *DSN*, 2004.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[21] K. Lev-Ari, G. Chockler, and I. Keidar. A constructive approach for proving data structures' linearizability. In *DISC*, 2015.

[22] K. Lev-Ari, G. V. Chockler, and I. Keidar. On correctness of data structures under reads-write concurrency. In *DISC*, 2014.

[23] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *ICDCS*, 2014.

[24] O. M. Mendizabal, F. L. Dotti, and F. Pedone. High performance recovery for parallel state machine replication. In *ICDCS*, 2017.

[25] O. M. Mendizabal, R. T. S. Moura, F. L. Dotti, and F. Pedone. Efficient and deterministic scheduling for parallel state machine replication. In *IPDPS*, 2017.

[26] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[27] S. Peri, M. Sa, and N. Singhal. Maintaining acyclicity of concurrent graphs. *CoRR*, abs/1611.03947, 2016.

[28] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *ICDCS*, 2013.

[29] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.