Exploiting Concurrency in Sharded Parallel State Machine Replication

Aldenio Burgos, Eduardo Alchieri, Fernando Dotti, and Fernando Pedone

Abstract—State machine replication (SMR) is a well-known approach to implementing fault-tolerant services, providing high availability and strong consistency. In classic SMR, commands are executed sequentially, in the same order by all replicas. To improve performance, two classes of protocols have been proposed to parallelize the execution of commands. Early scheduling protocols reduce scheduling overhead but introduce costly synchronization of worker threads; late scheduling protocols, instead, reduce the cost of thread synchronization but suffer from scheduling overhead. Depending on the characteristics of the workload, one class can outperform the other. We introduce a hybrid scheduling technique that builds on the existing protocols. An experimental evaluation has revealed that the hybrid approach not only inherits the advantages of each technique but also scales better than either one of them, improving the system performance by up to $3\times$ in a workload with conflicting commands.

Index Terms—Parallel State Machine Replication, Scheduling, Dependability.

1 Introduction

State machine replication (SMR) is a conceptually simple yet effective way to design services that must withstand failures [1], [2]. SMR provides clients with the abstraction of a highly available service while hiding the existence of multiple replicas, in what is known as strong consistency, or linearizability [3], [4]. In classic SMR, linearizability is achieved by having clients atomically broadcast commands to the replicas. Atomic broadcast is a communication abstraction that totally orders commands submitted by clients. Replicas deliver broadcast commands and execute them sequentially in delivery order (see Figure 1(a)).

Despite its widespread use, classic SMR makes poor use of multi-processor architectures since command execution is sequential (e.g., [5], [6]). A natural solution to improve the performance limitation of SMR would be to schedule commands to execute concurrently. Scheduling in state machine replication, however, differs from traditional scheduling techniques used in parallel systems in that SMR supports online services in a replicated environment. Guaranteeing strong consistency in the presence of replication introduces the need for determinism across replicas. Determinism in this context means that replicas must provide the same response upon executing the same command. It does not mean that replicas must make the same choices when scheduling commands for execution. Replicas must ensure determinism even though the queue of ordered commands at any two replicas may never be the same when a command is scheduled for execution, even though the complete sequence of ordered commands is the same at all replicas.

Several approaches have been proposed to accommodate concurrent execution in SMR (e.g., [7], [8], [9], [10], [11], [12], [13]). Most of these solutions are based on an early observation about SMR: although (multi-processor) concurrent command execution may result in non-determinism, independent commands (i.e., those that are neither directly nor indirectly dependent) can be executed concurrently without violating consistency [2]. Two commands are *independent* (or *non-conflicting*) if they either access different parts of the service state or only read state commonly accessed; conversely, two commands are *dependent* (or *conflicting*) if they access common state and at least one of the commands changes the shared state.

In this paper, we consider two categories of solutions to parallel state machine replication, and propose a novel approach that combines the advantages of existing techniques. Depending on how command interdependencies are identified and how commands are scheduled for execution, we can distinguish between late scheduling and early scheduling techniques.

With **late scheduling**, the scheduling of commands is handled entirely at the server side (see Figure 1(b)). Clients atomically broadcast commands for execution. A parallelizer at each replica delivers commands in total order, examines command dependencies, and includes delivered commands in a data structured shared with worker threads. The most common data structure to represent dependencies is a directed acyclic graph, or DAG (e.g., [8], [11], [13]). The DAG maintains a partial order among all pending commands, where vertices represent commands and directed edges represent dependencies. While dependent commands are ordered according to their delivery order, independent commands are not directly connected in the graph. Worker threads get commands ready for execution from the graph (i.e., vertices with no incoming edges) to be concurrently executed. When a worker thread completes the execution of a command, it removes the command from the graph (together with the edges to nodes that depend on it) and responds to the client that submitted the command. This approach poses the challenge that under high load (e.g., hundreds of thousands of commands per second) dependency

- *Aldenio Burgos and Eduardo Alchieri are with Universidade de Brasília, Brazil.*
- *Fernando Dotti is with PUCRS - Escola Politécnica, Brazil.*
- *Fernando Pedone is with Università della Svitzzera italiana, Switzerland.*

Manuscript submitted ... date.

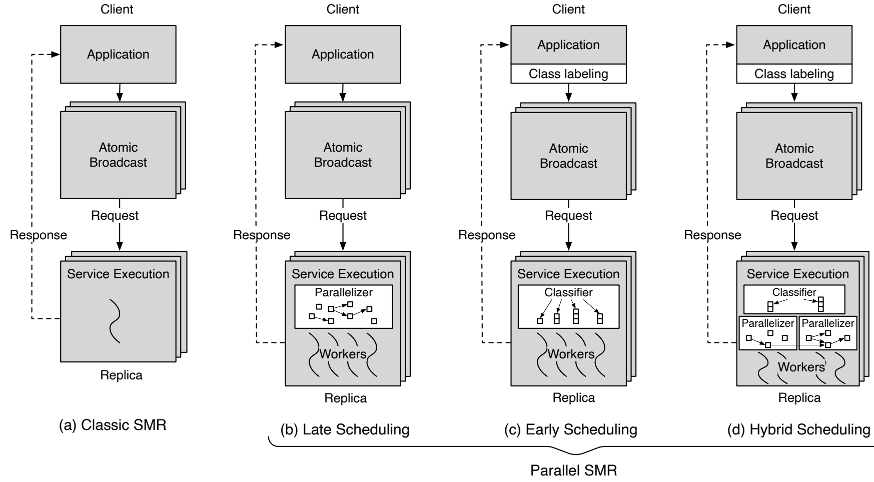


Fig. 1. Classic versus parallel state machine replication. Atomic broadcast and service are replicated in multiple servers (i.e., multiple boxes in the figure). Although it is possible to separate ordering (atomic broadcast) from execution (service), usually both are deployed in the replicas.

tracking may become itself the performance bottleneck. This aspect is tackled in [13] with a lock-free graph to handle dependencies.

In the **early scheduling** technique, requests are divided into classes and worker threads are preassigned to each class. Clients atomically broadcast commands, each command labelled with its class [12]. When a server delivers a command, a classifier at the server schedules the command to the worker threads previously assigned to the command's class (see Figure 1(c)). State partitioning or sharding is a natural way to divide requests into classes. For example, we could have one class per shard, to handle commands that access a single shard, and additional classes for commands that access multiple shards. This scheme results in relatively low scheduling overhead at the server since the scheduling of commands only requires a few simple operations, instead of more complex graph operations, as in late scheduling. However, additional synchronization is necessary at the worker threads to execute conflicting commands. Moreover, skewed workloads may cause unbalanced load among worker threads due to the fixed assignment of threads to classes [14].

We show in the paper that in the absence of conflicting commands (i.e., any two commands can execute concurrently) early scheduling largely outperforms late scheduling under different settings. This happens because the classifier can efficiently assign commands to threads, according to the classes-to-threads mapping, while the parallelizer becomes a bottleneck in late scheduling. Late scheduling, however, can handle conflicting commands more efficiently than early scheduling, even when the percentage of conflicting commands in the workload is low (i.e., 5%), since the cost of synchronizing threads in early scheduling is high. In this paper, we introduce a **hybrid scheduling** technique that builds on the advantages of late and early scheduling. The main idea is to shard the service state, then use a classifier to assign a command for execution to the corresponding shard, where a shard-specific parallelizer includes the command in a per-shard DAG. Worker threads remove the commands from their DAG and execute them. A central aspect is how to handle cross-shard commands without compromising the lock-freedom of the graph operations. In these cases, the related parallelizers insert the dependencies to the nodes in their respective DAGs and the command is inserted in any one of these DAGs. Notice that the scheduling of such a command may insert an edge that connects two DAGs (see Figure 1(d)). A performance evaluation shows that, in a sharded service, the hybrid scheduling performs similar to the early scheduling in a workload without conflicts and outperforms both late and early techniques by up to $3\times$ in a workload with conflicting commands.

This paper makes the following contributions:

- It revisits both late and early scheduling, discussing their main advantages and drawbacks.
- It proposes the hybrid scheduling approach, including detailed algorithms and correctness argument.
- It presents a detailed experimental evaluation, considering four different applications, to show the behavior of each scheduler in different settings. The experimental study showed that the hybrid approach is significantly more efficient than late and early scheduling.

The paper is organized as follows. Section 2 states the system model. Sections 3 and 4 detail the late and early scheduling techniques, respectively. Section 5 introduces the hybrid approach, combining both techniques. Section 6 reports on the performance evaluation. Section 7 survey related work and Section 8 concludes the paper.

2 System model and definitions

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

Processes communicate by message passing, using one-to-one or one-to-many communication. One-to-one communication uses primitives $send(m)$ and $receive(m)$, where m

is a message. If a sender sends a message enough times, a correct receiver will eventually receive the message. One-to-many communication uses atomic broadcast, defined by primitives $broadcast(m)$ and $deliver(m)$.[1] Atomic broadcast ensures the following properties [15], [16] [2]:

- *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'.

State machine replication provides *linearizability* [19], a form of strong consistency. An execution is linearizable if there is a way to total order client commands such that (a) it respects the semantics of the objects accessed by the commands, as expressed in their sequential specifications; and (b) it respects the real-time ordering of the commands in the execution. There exists a real-time order among two commands if one command finishes at a client before the other command starts at a client.

The scheduling algorithms we discuss in the paper exploit concurrency among commands. Let C be the set of possible commands and $\#_C \subseteq C \times C$ the conflict relation between commands. If $\{c_i, c_j\} \in \#_C$, then commands c_i and c_j conflict and replicas must serialize their execution; otherwise, replicas can execute c_i and c_j concurrently.

3 Late Scheduling

We generalize the requirements for parallel execution of commands using late scheduling with an abstract data type that keeps track of the order among conflicting commands [13]. We call this data structure *Conflict-Ordered Set (COS)*. This data structure is defined by three primitives with sequential specification as follows.

- $insert(c \in C)$ inserts command c in the data structure;
- $c \in C : get()$ returns c if and only if:
  - c is in the data structure,
  - no previous $get()$ has returned c, and
  - there is no c' in the data structure inserted before c such that $(c, c') \in \#_C$; and
- $remove(c \in C)$ removes c from the data structure.

Algorithm 1 details the behavior of the parallelizer and worker threads based on COS. There is a configurable set of worker threads T and a shared COS structure (lines 2 and 3). We use a semaphore to limit the number of commands in the COS data structure (line 4) and another to keep track of the number of commands that can be executed (line 5). When a command is delivered and there is enough space in the data structure, it is inserted in the COS (line 9). The insert primitive returns the number of commands ready to execute, so that the $ready$ semaphore can be updated (lines 9 and 10). A working thread waits for ready commands (line 13), requests a command with the get primitive, executes the command, and then $remove$s it, freeing space in the graph (line 17). During removal, new commands may become ready due to solved references (lines 16 and 18).

Algorithm 1 Late scheduling: parallelizer and workers

```
1:  Constants and data structures:
2:    T: set of working thread identifiers
3:    COS: the conflict-ordered set
4:    space ← new Semaphore(maxSize)                     {graph space}
5:    ready ← new Semaphore(0)                           {ready nodes}

6:  Parallelizer works as follows:
7:    onDeliver(req):                          {when a new request arrives}
8:      space.down()                           {wait for space available}
9:      rdy ← COS.insert(req)                  {insert it in the stucture}
10:     ready.up(rdy)                          {allow to retrieve ready nodes}

11: Each WorkingThread with t_id ∈ T executes as follows:
12:   while true do                            {infinite loop}
13:     ready.down()                           {wait free nodes to execute}
14:     c ← COS.get()                          {get a command c free to run}
15:     execute(c)                             {execute c and then}
16:     rdy ← COS.remove(c)                    {remove c from the structure}
17:     space.up()                             {allow to insert new nodes}
18:     ready.up(rdy)                          {allow to retrieve ready nodes}
```

3.1 Algorithm

In this section, we present a nonblocking COS implementation. Our algorithm uses native atomic types and an atomic compare-and-set operation. We also assume that $insert$s are called sequentially, according to the order defined by atomic broadcast (see Algorithm 1). This ensures that replicas handle conflicting commands consistently. While *insert* invokations are sequential (among themselves), *get* and *remove* invokations are concurrent with any operations.

The algorithm builds a directed acyclic graph to represent dependencies among commands. Each $Node$ of the graph (Algorithm 2, line 2) contains a command c, an atomic field with the command state, a list $depOn$ of references to nodes it depends on, a list $depMe$ of references to other nodes that depend on this node, and the nxt field, which represents the total order among commands. Each command transits through the following states, in the order shown:

- $\perp$ starting: the command is being inserted in the graph;
- wtg waiting: the command has already been inserted and depends on other commands to execute;
- rdy ready: all dependencies have been resolved and the command can be executed;
- exe executing: the command was taken for execution by some worker thread; and
- rmd removed: the node with the command has been logically removed from the COS data structure.

The strategy used to allow lock-free operations is to change the graph topology exclusively in the insert operation, while other operations may take commands for execution as well as logically remove nodes by appropriately marking their state only, without modifying any other field or the graph topology. The remove operation is logic in that it marks the node as removed while actual removal from the graph takes place during insertion of another node using

1. We use the terms send/receive for one-to-one communication and broadcast/deliver for one-to-many or group communication.

2. Atomic broadcast needs additional synchrony assumptions to be implemented [17], [18]. These assumptions are not explicitly used by the protocols proposed in this paper.

the helping technique, when one operation helps the other to accomplish its modifications.

The *insert* operation (Algorithm 2, line 13) assumes there is room in the graph to create a node. The new node is created with command c (Algorithm 3, line 1). Function $calculateDependencies$ (Algorithm 3, line 12) traverses nodes in N, from the oldest to the newest, and builds the needed dependencies. If any $n' \in N$ not logically removed conflicts with the new node n_n, then lines 18 and 19 insert edges to represent this dependency. During a physical remove (Algorithm 3, line 16) nodes logically removed have their outgoing edges deleted (Algorithm 3, lines 22 to 24). Then, the new node is included in the list N of nodes to become reachable in the graph, and its state is changed to waiting (Algorithm 2, lines 16 and 17). Finally, $testReady$ (Algorithm 3, line 7) checks for dependencies. If none are found, then the node becomes ready.

The *get* operation (Algorithm 2, line 19) returns a node in the graph ready to execute. When this operation is called, a ready node in the graph exists due to Algorithm 1, lines 10 and 18. The first node atomically tested to be equal to rdy (i.e., ready) is set to exe (i.e., executing) and returned. Since insertion follows delivery order, the node returned is the oldest ready to execute. Due to the atomicity of $compareAndSet$ (Algorithm 3, line 3) the node is returned by at most one get.

The *remove* operation (Algorithm 2, line 25) marks the node as logically removed by assigning rmd to its state. It checks if each of the dependent nodes became ready (lines 28 and 29). The edges from older (the one logically removed) to newer nodes ($depMe$) identify which nodes may become ready when this dependency is solved. The edges in the other direction ($depOn$) allow to evaluate if a node has all dependencies solved, that is, all nodes it depends on are logically removed (Algorithm 3, line 8). With this, the logical remove operation is able to signal new ready nodes only by checking the state of referred nodes and without any modifications to the graph structure, which is left to the $insert$ operation, while performing helped remove.

3.2 Correctness

The main argument for correctness is that the $insert$ operation performs all structural modifications to the graph, and its invocations are sequential (according to the atomic broadcast deliver order). Operations get and $remove$ do not change the topology of the graph, they logically mark nodes as taken for execution or as logically removed, using the $Node$'s atomic st attribute. As discussed, operation $insert$ is responsible for removing from the graph any logically removed nodes as soon as it traverses the graph to insert a new node. This rules out any possible topological inconsistency due to concurrency.

Graph traversals are performed only by $insert$ and get. These concurrent traversals are consistent despite the fact that the $insert$ operation may change the topology. Since both operations traverse nodes in the same order, and a possible modification on the reference to next node is atomic, be it due to the removal or insertion of a node, the get operation will either use the previous reference, or the new one. In either case due to the nature of the operation, the result is the same. Consider that get traverses a logically removed node. Since the node was already executed, get will not return it.

The $remove$ operation atomically marks a referenced node as executed, not traversing the graph. It also reads information from nodes depending on the one being logically deleted to check if any one became ready to execute. Since these operations may only atomically change the state of a node, they do not affect other operations. Also, since the nodes checked exist due to the existing references, by the

Algorithm 2 Lock-free COS: data types and operations

1: **Data types:**
2: $Node$: {
3: $c : Command,$
4: **atomic** $st : \{\bot, wtg, rdy, exe, rmd\}$ {*node may be starting, ...*}
5: {*...waiting, ready, executing, removed*}
6: $depOn$: set of $NodeRef$ {*nodes this one depends on*}
7: $depMe$: set of $NodeRef$ {*nodes that depend on this*}
8: $nxt : NodeRef$ {*next node in arrival order*}
9: }
10: $NodeRef$: **atomic reference** to $Node$

11: **Variables:**
12: $N : NodeRef$ {*nodes in COS*}

13: **insert**($c : Command$)**: int**
14: $n_n \leftarrow createNode(c)$ {*create new node structure*}
15: $n \leftarrow calculateDependencies(N, n_n)$ {*compute the dependencies*}
16: **if** $n = \bot$ **then** $N \leftarrow n_n$ **else** $n.nxt \leftarrow n_n$
17: $n_n.st \leftarrow wtg$ {*the node is now waiting...*}
18: **return** $testReady(n_n)$ {*...and tested if ready*}

19: **get**()**:** NodeRef {*assumes a ready node exists*}
20: $n \leftarrow N$ {*start searching for a ready node*}
21: **while** $n \neq \bot$ **do** {*consider each node, in arrival order*}
22: **if** $compareAndSet(n.st, rdy, exe)$ **then**
23: **return** n {*if node is ready, mark it and return*}
24: $n \leftarrow n.nxt$ {*go to next*}

25: **remove**($n : NodeRef$)**: int** {*assumes n with $n.st = exe$ exists*}
26: $n.st \leftarrow rmd$ {*logic removal*}
27: $rdys \leftarrow 0$ {*ready nodes counter*}
28: **for all** $n_i \in n.depMe$ **do** {*for all nodes depending on me*}
29: $rdys \leftarrow rdys + testReady(n_i)$ {*check if n_i is ready, count*}
30: **return** $rdys$ {*return number of ready nodes*}

Algorithm 3 Lock-free COS: auxiliary operations

1: **createNode**($c : Command$): NodeRef
2: **return reference to new** $Node\{c, \bot, \emptyset, \emptyset, \bot\}$

3: **compareAndSet**(a, b, c): boolean
4: **return atomic** { **if** $a = b$ **then** $a \leftarrow c; true$ **else** $false$ }

5: **conflict**($n_i, n_j : Node$): boolean
6: **return** $(n_i.c, n_j.c) \in \#_C$ {*is this pair in conflicts set*}

7: **testReady**($n : NodeRef$) $: 0, 1$
8: **if** $\{n_i \in n.depOn | n_i.st \neq rmd\} = \emptyset$ **then** {*n has no dependency*}
9: **if** $compareAndSet(n.st, wtg, rdy)$ **then** {*switch to rdy*}
10: **return** 1
11: **return** 0

12: **calculateDependencies**($N, n_n : NodeRef$) $: NodeRef$
13: $n' \leftarrow n \leftarrow N$ {*n' and n equal to N*}
14: **while** $n' \neq \bot$ **do** {*consider each node, in order*}
15: **if** $n'.st = rmd$ **then** {*n' logically removed*}
16: $helpedRemove(N, n', n)$ {*remove it*}
17: **else if** $conflict(n', n_n)$ **then** {*n' is valid, they conflict?*}
18: $n'.depMe \leftarrow n'.depMe \cup \{n_n\}$ {*n_n depends on n'*}
19: $n_n.depOn \leftarrow n_n.depOn \cup \{n'\}$ {*reference count in n_n*}
20: $n \leftarrow n'$; $n' \leftarrow n'.nxt$ {*consider the next*}
21: **return** n

22: **helpedRemove**($N, n', n : NodeRef$)
23: **for all** $n_i \in n'.depMe$ **do** {*for every node that depends on n'*}
24: $n_i.depOn \leftarrow n_i.depOn \setminus \{n'\}$ {*remove n' from dependencies*}
25: **if** $!compareAndSet(N, n, n'.nxt)$ **then** {*if not first node*}
26: **atomic** { $n.nxt \leftarrow n'.nxt$ } {*bypass n'*}

logically removed node and the assumption of a garbage collector, the operation succeeds.

Liveness follows from the fact that the graph is a DAG. Since commands can only depend on previous conflicting ones, according to a total order, dependencies will never build a cycle (i.e., the graph is acyclic). After a command executes, it will inductively remove dependencies to commands assuring that there is at least a next command that can be executed, or no command left.

4 EARLY SCHEDULING

The early scheduling approach classifies requests into *request classes* and defines dependencies between classes [12], [20]. Consider a service with a set R of requests, and let $C = \{C_1, C_2, ..., C_{nc}\}$ be a set of class descriptors, where nc is the number of classes. We define **request classes** as $\mathcal{R} = C \rightarrow \mathcal{P}(C) \times \mathcal{P}(R)$,[3] 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. Two classes conflict if one class contains at least one request that conflicts with a request in the other class. We restrict classes to be disjunct and non-empty.

4.1 From applications to request classes

Defining request classes for an application is related to the problem of sharding the application state. In its simplest form, the application state can be sharded into disjoint partitions according to some criteria (e.g., hash partitioning, range partitioning). More sophisticated sharding mechanisms may account for how data is accessed (e.g., grouping entries in the same shard if they are accessed together by the requests) [30]. For each shard, one can define a class of read-only requests (i.e., request that do not modify the shard state) and a class of update requests. Depending on the data accessed by requests and on the criterion used for sharding the data, one may need to define classes that involve multiple shards, either read-only or update. A read-only class does not conflict with any other read-only classes. An update class conflicts with itself (since it may contain conflicting requests) and with any other classes that contain requests that access the same shard.

For example, consider a key-value store application with an operation to write (put) a (key, value) pair and an operation to read (get) the value associated to a key. Each key is mapped to a single shard using hashing partitioning (i.e., hash_function(key) returns the key's shard). In a system with two shards, we could have one class to read and one class to write each shard, respectively, classes C_{R1} and C_{W1} for shard 1 and classes C_{R2} and C_{W2} for shard 2. The read classes contain get operations and the write classes contain put operations on the keys associated with the shard. A multi put operation that writes multiple entries would be part of a class C_{MW} that involves both shards. Class C_{MW} would conflict with all classes.

4.2 From request classes to threads

To ensure that requests in a class are executed, we map one or more threads to each class. To guarantee a correct execution (i.e., linearizable), we determine whether the execution of requests in the class must be sequential or concurrent, and whether classes must share worker threads. More precisely, the classes-to-threads mapping respects the following rules:

R1. *Every class is associated with at least one thread.* The threads mapped to the class execute requests that belong to the class.

R2. *If a class is self-conflicting (i.e., the class contains requests that conflict), it is sequential.* Even though a class is sequential, we may want to assign multiple threads to the class since this provides a means for classes to synchronize, as we show below. In this case, each request is scheduled to all threads of the class and processed as described below.

R3. *If two classes conflict, at least one of them must be sequential.* The previous requirement may help decide which one.

R4. *For conflicting classes c_1, sequential, and c_2, concurrent, the set of threads associated to c_2 must be included in the set of threads associated to c_1.* This requirement ensures that requests in c_2 are serialized w.r.t. requests in c_1.

R5. *For conflicting sequential classes c_1 and c_2, it suffices that c_1 and c_2 have at least one thread in common.* The common thread ensures that requests in the classes are serialized.

A classes-to-threads mapping is defined as $CtoT = C \rightarrow \{Seq, Conc\} \times \mathcal{P}(T)$, where C is the set of class descriptors; $\{Seq, Conc\}$ is the sequential or concurrent synchronization mode of a class; and $\mathcal{P}(T)$ is the possible subsets of the n threads $T = \{t_0, .., t_{n-1}\}$ at a replica. The previous rules result in several possible mappings and can be modeled as an optimization problem that searches the configuration that allows maximum concurrency in the execution [12]. The general idea is to observe the workload and maximize (resp., minimize) the number of threads in concurrent (resp., sequential) classes.

4.3 Algorithm

Clients label requests with their corresponding class. A replica has one classifier thread and n (worker) threads, each one with a separate input FIFO queue. The classifier delivers each request r in total order and schedules the request to one or more workers according to Algorithm 4: if r's class is sequential, then all threads mapped to the class receive the request to synchronize the execution (lines 16–17); if r's class is concurrent, the classifier assigns r to one thread among the ones mapped to the class (lines 18–19), following a round-robin policy (function $next$).

Each worker (Algorithm 5) takes one request at a time from its queue in FIFO order (line 6) and then proceeds depending on its class synchronization mode. If it is sequential, the worker synchronizes with other workers in the class using barriers before the request is executed (lines 11–18), and only one worker executes the request. If it is concurrent, then the worker simply executes the request (line 10).

4.4 Correctness

We argue by case analysis that any mapping that follows rules R1 to R5 generates linearizable executions. We start with classes without external conflicts.

(1) *Class c has no internal and no external conflicts.* Then any request $r \in c$ is independent of any other requests and can be

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

Algorithm 4 Early scheduling: classifier.

```
1:  Data types:
2:    T : set of thread identifiers
3:    C : set of class identifiers
4:    Req : {c : Command,               {a request has the command to execute}
5:           classId : C}               {and the identifier to the class it belongs}
6:    CtoT =                            {the class to threads mapping is...}
7:      ∀ classId ∈ C →                 {for each class}
8:        smode : {Seq, Conc},          {a synchronization mode}
9:        threads : P(T)                {a nonempty subset of threads}
10:   smode(classId) = CtoT(classId).smode
11:   threads(classId) = CtoT(classId).threads

12: Variables:
13:   queues[0, ..., n − 1] ← ∅         { one queue per thread}

14: Classifier works as follows:
15:   onDeliver(req):
16:     if smode(req.classId) = Seq then      { if execution is sequential}
17:       ∀t ∈ threads(req.classId)           { for each conflicting thread}
18:         queues[t].fifoPut(req)            { synchronize to exec req}
19:     else                                  { else assign it in round-robin}
20:       queues[next(threads(req.classId))].fifoPut(req)
21:                                           {assign req to one thread in round-robin}
```

Algorithm 5 Early scheduling: workers.

```
1:  Variables:
2:    queue[T]                {one empty request queue per thread in T}
3:    barrier[C]              {one initialized barrier per request class}

4:  Each WorkingThread with t_id ∈ T executes as follows:
5:    while true do                                 {infinite loop}
6:      req ← queue[t_id].fifoGet()                 {wait until a request is available}
7:      if smode(req.classId) = Seq then            {if execution is sequential}
8:        execWithBarrier(req, barrier[req.classId])  {use barrier}
9:      else                                        {concurrent execution}
10:       exec(req)                                 {execute the request}

11: execWithBarrier(req, barrier):
12:   if t_id = min(threads(req.classId)) then      {smallest id}
13:     barrier.await()                             {wait for signal}
14:     exec(req)                                   {execute request}
15:     barrier.await()                             {resume workers}
16:   else
17:     barrier.await()                             {signal worker}
18:     barrier.await()                             {wait execution}
```

dispatched to the input queue of any thread (assigned to c). According to R1, such a thread exists. The thread dequeues and executes without further synchronization (Algorithm 5, line 10).

(2) *Class c has internal conflicts but no external conflicts.* Then, by rule R2, c is sequential and any request $r \in c$ is enqueued to the input queues of all threads associated to c, according to the delivered order. These threads eventually dequeue r due to the same order in their queues and synchronize to execute it (Algorithm 5, line 8), i.e., requests belonging to c are executed respecting their delivery order.

Now we consider external conflicting classes.

(3) *Class c_1 has no internal conflicts, but conflicts with c_2.* By rule R3, one of the classes must be sequential. Assume c_2 is sequential. From R4 we have that the threads that implement c_1 are contained in the set of threads that implement c_2. It then follows that every request from c_1 is executed before or after any c_2's request, according to their delivery order. Notice that this holds even though requests in c_1 are concurrent.

(4) *Classes c_1 and c_2 have internal conflicts and conflict with each other.* Then c_1 and c_2 are both sequential. Both synchronize their threads to execute requests. According to restriction R5, these classes have at least one common thread t_x which suffices to impose that c_1 and c_2 execute their requests according to the delivery order.

5 A Hybrid Approach

The hybrid scheduling shards the service state and uses an instance of COS per shard, leading to a set of subgraphs (Figure 1(d)). Commands that access two or more shards interconnect the related subgraphs. In doing so, the hybrid scheduling removes the late scheduling bottleneck since there is one parallelizer per subgraph.

A sequential class is associated to each shard and mapped to a different parallelizer thread. (All insertions in a subgraph are sequential, and thus, there is no point in assigning multiple parallelizers per shard.) We also create sequential classes for all possible sets of shards accessed by a command and assign to these classes the parallelizers in the corresponding single-shard classes. These classes ensure proper synchronization of commands that access multiple shards. For example, if C_{S_1} and C_{S_2} are classes associated to shards S_1 and S_2, then each one will be mapped to a parallelizer thread t_{p_1} and t_{p_2}, respectively. Additionally, there is a class C_{S_1,S_2}, mapped to both threads t_{p_1} and t_{p_2}.

While in the early scheduler the classes-to-threads mapping affects performance, in the hybrid scheduler this mapping is not a concern since we use a special mapping instance, as previously described. Since we have just one parallelizer per single-shard class, it is not necessary to execute any additional synchronization and these parallelizers include requests in parallel in their subgraphs. Moreover, we optimized the multi-shard request scheduling by avoiding the use of barriers in their classes, and instead we use atomic counters to define when all parallelizers included the dependencies in their subgraphs.

Clients label requests with the class identifier, as in early scheduling. Every replica has one classifier and one parallelizer thread per shard, and a set of worker threads per shard. Each parallelizer has a separate input queue and removes requests from the queue in FIFO order. The classifier in a replica delivers requests in total order and dispatches each request r to one or more input queues:

- If r accesses only one shard S, r depends on preceding requests assigned to S and is inserted in the related queue.
- If r accesses two or more shards, one parallelizer inserts r in its subgraph and all parallelizers involved in the command insert dependencies in their subgraph to r.

Each worker thread selects a request available for execution from the related subgraph, marking it as executing, and then removes it from the subgraph after execution.

5.1 Algorithm

We now present hybrid scheduling in more detail. Algorithm 6 presents the structure types and variables used. In the following, we emphasize the main differences to the previous algorithms. C has the same previous definition of a set of request classes identifiers. We add a set of shard identifiers S and a mapping from classes to subsets of shards $CtoS$, similar to the classes-to-threads mapping of early scheduling. Each shard has an input queue and an associated lock-free COS extended from Algorithm 2 to handle cross-shard commands. There are counting semaphores

per shard to record the number of ready commands and free space for new commands. The extended COS node, now called ***HyNode***, adds two new attributes to the node used in the lock-free graph: s_{id} stores the identifier of the shard that will store the node; and rem_s stores the number of parallelizers that have not yet processed this node during insertion. Each HyNode also has two arrays of sets of nodes, one for nodes that depend on the inserted node and another for nodes the inserted node depends on. Each parallelizer accesses only its position in the arrays, avoiding race conditions on these sets. Variable $shards$ (line 22) represents all shards in a replica.

Algorithm 6 Hybrid scheduling: types and variables

1: **Data types:**
2: $Shard : \{$
3: $queue \leftarrow \emptyset,$ {*execution queue*}
4: $cos \leftarrow \langle N : \bot, R : \emptyset \rangle,$ {*extended lock-free COS - Alg. 10*}
5: $ready \leftarrow new\ Semaphore(0)$ {*shard ready nodes*}
6: $space \leftarrow new\ Semaphore(maxSize/|S|)$ {*shard space*}
7: $\}$
8: $Req : \{...\}$ {*as defined in Algorithm 4*}
9: $Node : \{...\}$ {*as defined in Algorithm 3*}
10: $HyNode\ extends\ Node : \{$ {*specialization of* $Node$}
11: $s_{id} : S,$ {*shard id*}
12: $rem_s : int$ {*remaining parallelizer counter*}
13: with $nxt : HyNodeRef$
14: with $depOn[]$: array of sets of $HyNodeRef$, one per shard
15: with $depMe[]$: array of sets of $HyNodeRef$, one per shard
16: $\}$
17: $HyNodeRef$ atomic reference to $HyNode$
18: **Variables:**
19: C {*as defined in Algorithm 4*}
20: S {*set of shard ids*}
21: $CtoS : C \rightarrow \mathcal{P}(S)$ {*a class maps to a subset of shards*}
22: $shards[S]$ {*one shard struct per shard id*}

Whenever a request is delivered, the classifier identifies the shards involved in the request (Algorithm 7, line 3) and chooses one of them to store the node (line 4). It then creates a hybrid node with these attributes and enqueues it in the input queue of all involved shards.

Algorithm 7 Per replica classifier

1: **Classifier works as follows:**
2: **onDeliver**$(req : Request)$:
3: $shards_{ids} \leftarrow CtoS(req.classId)$ {*involved shards identification*}
4: $ss_{id} \leftarrow selectShard(req.classId)$ {*responsible shard selection*}
5: $rem_s \leftarrow shards_{ids}.size$ {*number of parallelizer involved*}
6: $node \leftarrow createNode(req.c, ss_{id}, rem_s)$ {*COS HybridNode*}
7: **for all** $s \in shards_{ids}$ **do** {*for each shard involved*}
8: $shards[s].queue.fifoPut(node)$ {*put in shard's queue*}

Auxiliary functions:
9: **selectShard**$(class_{id} : C) : S$
10: **return** shard s chosen in round-robin among $CtoS(class_{id})$

11: **createNode**$(c : Command, s_{id} : int, rem_s : int)$: HyNodeRef
12: **return reference to new** $HyNode\{c, \bot, \emptyset, \emptyset, \bot, s_{id}, rem_s\}$

Next, the request is scheduled in each shard (Algorithm 8). A parallelizer per shard keeps reading in FIFO order the next node to schedule (line 3). If the shard is responsible for the node (line 4), it allocates space (line 5) to store the node and creates dependencies (line 6); otherwise, it only creates dependencies (line 8). Notice that the shard identifier is used in the insert operation to indicate in which subgraph the node and its dependencies must be included.

Algorithm 8 Per-shard parallelizer

1: **Parallelizer of shard** s_{id} **executes as follows:**
2: **while** true **do** {*infinite loop*}
3: $node \leftarrow shards[s_{id}].queue.fifoPull()$ {*next node to schedule*}
4: **if** $node.s_{id} = s_{id}$ **then** {*is this the responsible parallelizer?*}
5: $shards[s_{id}].space.down()$ {*grant space to insert...*}
6: $shards[s_{id}].cos.insert(node, s_{id}, true)$ {*...node and deps.*}
7: **else**
8: $shards[s_{id}].cos.insert(node, s_{id}, false)$ {*or deps. only*}

Before we delve into details of the extended lock-free COS operations for hybrid scheduling, we describe how worker threads execute commands (Algorithm 9). This algorithm adapts Algorithm 1 to associate each worker to a shard s_{id} to process requests. Whenever there is a node free of dependencies in that shard (line 3) it is retrieved (line 4), executed (line 5) and logically removed from its shard (line 6), releasing space (line 7).

Algorithm 9 Per-shard worker threads

1: **Each WorkingThread of shard** s_{id} **executes as follows:**
2: **while** true **do** {*infinite loop*}
3: $shards[s_{id}].ready.down()$ {*grant a ready node to work*}
4: $node \leftarrow shards[s_{id}].cos.get()$ {*get a node free to run*}
5: $execute(node.c)$ {*execute the command and then*}
6: $shards[s_{id}].cos.remove(node)$ {*mark node as removed*}
7: $shards[s_{id}].space.up()$ {*allow to insert new nodes*}

Algorithm 10 presents the extended COS implementation to deal with a sharded service. A node belongs to the responsible shard selected in Algorithm 7, and is related to all the other shards involved in its class. Nodes that belong to a shard are stored in N (line 2), and related nodes are stored in a set R (line 3). Each shard is responsible for allocating space, executing, and deleting nodes in N. While executing nodes in N, dependencies to other shards have to be respected. For each shard, related nodes represent nodes that are stored in other shards but may have dependencies with the shard nodes and thus must be checked during an insertion. The set R is only manipulated by the parallelizer related to the shard, thus also does not present race conditions.

The ***insert*** operation has as argument a flag that tells whether the node should be included in N (line 8) or in R (line 10). After the node is processed in all subgraphs it is involved in (line 11), the node changes its status to waiting (wtg), and is tested if ready (lines 12-13). The ***remove*** operation marks the node as removed (line 15), and evaluates if dependent nodes become executable. Finally, ***get*** is the same operation as in Algorithm 2. It returns the first ready node in the subgraph.

Functions $insertDeps$ and $insertRelatedDeps$ insert the dependencies related to nodes in N and R, respectively. In these functions, logically removed nodes are physically removed from the structures. Function $removeDeps$ removes the ingoing edges to a removed node. The $bind$ function is responsible for including a dependency between two nodes. Similar to the late scheduling, function $testReady$ verifies if a node is ready to execute.

5.2 Correctness

The hybrid scheduling combines the early with the late scheduling, and adds some refinements. Commands are delivered at each replica in total order (from atomic broadcast).

Algorithm 10 Extended lock-free COS

1: **Variables:**
2: $N : HyNodeRef,\ initially\ \bot$ {*List of HyNodes in the subgraph*}
3: $R : set\ of\ HyNode,\ initially\ \emptyset$ {*Set of related HyNodes*}

4: **insert**($n_n : HyNodeRef, s_{id} : int, insert_{node} : boolean$)
5: $insertRelatedDeps(n_n, s_{id})$ {*build dependencies with R*}
6: $n \leftarrow insertDeps(n_n, s_{id})$ {*build dependencies with N*}
7: **if** $insert_{node}$ **then** {*insert the node in this graph...*}
8: **if** $n = \bot$ **then** $N \leftarrow n_n$ **else** $n.nxt \leftarrow n_n$
9: **else** {*...or in the related nodes set*}
10: $R \leftarrow R \cup \{n_n\}$
11: **if** $decrementAndGet(n_n.rem_s) = 0$ **then** {*if this is the last...*}
12: $n_n.st \leftarrow wtg$ {*...parallelizer, the node is now waiting...*}
13: $testReady(n_n)$ {*...and tested if ready*}

14: **remove**($n :\ HyNodeRef$) {*assumes n exists and has* $n.st = exe$}
15: $n.st \leftarrow rmd$ {*logic removal*}
16: **for all** $s_{id} \in S$ **do** {*for all shards*}
17: **for all** $n_i \in n.depMe[s_{id}]$ **do** {*for all nodes depending on me*}
18: $testReady(n_i)$ {*check if* n_i *is ready*}

19: **get**() : $HyNodeRef$ {*as defined in Algorithm 2*}

Auxiliary functions:
20: **insertRelatedDeps**($n_n : NodeRef, s_{id} : int$)
21: **for all** $n \in R$ **do** {*consider each n node in R*}
22: **if** $n.st = rmd$ **then** {*n logically removed*}
23: $removeDeps(n, s_{id})$ {*remove its dependencies*}
24: $R \leftarrow R \setminus n$ {*remove n from R*}
25: **else if** $conflict(n_n, n)$ **then**
26: $bind(n_n, n, s_{id})$ {*conflict, insert a dependency*}

27: **insertDeps**($n_n : NodeRef, s_{id} : int$) : $NodeRef$
28: $n' \leftarrow n \leftarrow N$
29: **while** $n' \neq \bot$ **do** {*consider each node* n' *in N, in order*}
30: **if** $n'.st = rmd$ **then** {n' *logically removed*}
31: $removeDeps(n', s_{id})$ {*remove its dependencies*}
32: **if** $!compareAndSet(N, n, n'.nxt)$ **then** {*if not first node*}
33: **atomic** { $n.nxt \leftarrow n'.nxt$ } {*bypass it*}
34: **else if** $conflict(n_n, n')$ **then**
35: $bind(n_n, n', s_{id})$ {*conflict, insert a dependency*}
36: $n \leftarrow n';\ n' \leftarrow n'.nxt$ {*go to the next*}
37: **return** n {*return the last node in N*}

38: **removeDeps**($n : NodeRef, s_{id} : int$)
39: **for all** $n_i \in n.depMe[s_{id}]$ **do** {*consider each dependency*}
40: $n_i.depOn[s_{id}] \leftarrow n_i.depOn[s_{id}] \setminus \{n\}$ {*and remove it*}

41: **bind**($n_{new}, n_{old} : NodeRef, s_{id} : int$)
42: $n_{old}.depMe[s_{id}] \leftarrow n_{old}.depMe[s_{id}] \cup \{n_{new}\}$ {n_{new} *depends*}
43: $n_{new}.depOn[s_{id}] \leftarrow n_{new}.depOn[s_{id}] \cup \{n_{old}\}$ {*on* n_{old}}

44: **testReady**($n : NodeRef$)
45: **if** $\{n_i \in \{\exists s_{id} \in S : n_i \in n.depOn[s_{id}] \wedge n_i.st \neq rmd\}\} = \emptyset$ **then** {*n has no dependency*}
46: **if** $compareAndSet(n.st, wtg, rdy)$ **then** {*switch to rdy*}
47: $shards[n.s_{id}].ready.up()$ {*inform a new node ready in* s_{id}}

48: **decrementAndGet**($value : int$) : int
return atomic $\{value \leftarrow value - 1;\ value\}$

49: **compareAndSet**(a, b, c): boolean {*as defined in Algorithm 3*}

50: **conflict**($n_i, n_j : Node$): boolean {*as defined in Algorithm 3*}

The classifier sequentially evaluates the command classes and enqueues commands in the input queue of one or more parallelizers, according to the classes. By the discussed properties of early scheduling, we have that the delivery order is respected at all parallelizer queues.

Conflict classes represent any possible combinations of shards involved in cross-shard commands. A single-shard command belongs to a class that maps to the parallelizer representing that shard only, while a multi-shard command belongs to a class that maps to all involved parallelizers.

The parallelizer of a shard takes commands from its input queue, while preserving delivery order, and calculates conflicts against pending commands in the shard. This ensures that there will never be dependency cycles within or among shards—the last because the total order is preserved across all shard input queues.

Cross-shard commands are considered completely scheduled only when all parallelizers involved have calculated dependencies to pending commands in their respective shards. This is ensured by the number of parallelizers involved in the node (rem_s) which is atomically decremented whenever a parallelizer has completed its dependency analysis. Therefore, a node will never be considered for execution before all involved parallelizers have processed it and dependencies have been fully represented.

A command is considered for execution only if free from dependencies. This is ensured by get, which takes only ready commands, and by $testReady$, which switches commands from waiting to ready when dependencies are solved. Function $testReady$ is fired whenever a new node is inserted or a node is logically removed to respectively check if a new node is free from dependencies and if the removal of a node will free any other nodes to execute.

All arguments above show that dependent commands execute according to the total order, while independent commands from different or same shards execute concurrently. Due to acyclic dependencies, as discussed above, considering all shards, at all times it holds that there exists an oldest command that does not depend on any previous one and can execute. When this command executes, it is logically removed and dependencies solved. Consequently, it will inductively free dependencies to other commands, resulting in a dependency structure with the same property above, ensuring progress.

In addition, each command has a responsible shard. Workers associated to each shard will consider its commands for execution whenever they are ready. If more than one command is ready, then the oldest one is chosen, ensuring that ready commands at all shards eventually execute.

Regarding the lock-free concurrent access to the dependency graph, safety at each shard has the same arguments as in Section 3.2. Besides, during insertion of a cross-shard node, the involved shard parallelizers concurrently compute dependencies of the incoming node to different sets of nodes. This is done by adding node references to node attributes $depOn$ and $depMe$. These attributes are built with different subsets for each shard. Since each shard parallelizer updates a different set, the operation is both safe and free from synchronization among parallelizers.

When ready, a node is taken for execution (get) by one thread belonging to the shard responsible for the node. This thread executes and logically $remove$s the node. During a physical removal, all nodes on $depMe$ are visited to update their respective $depOn$ sets, by the parallelizer of each shard. When workers from different shards concurrently remove nodes that affect (release dependencies) a common node, the common node's $depOn$ set will be accessed concurrently. Again, due to the per shard dependency list at nodes, this is ensured to be safe and free from synchronization among workers from different shards.

6 Experiments

We experimentally evaluated the proposed scheduling protocols, aiming to show (1) the advantages and the draw-

backs of the late and early approaches (§6.4); (2) how hybrid scheduling circumvents these performance problems by leveraging the best characteristics of each of these techniques in a multi-sharded system (§6.5); (3) the performance of the protocols with skewed workloads (§6.6); and (4) the performance of the data structures alone without integration in a SMR framework (§6.7). We also analyzed the proposed protocols using different applications (§6.8).

6.1 Environment

We implemented all the scheduling techniques in BFT-SMART [21], an SMR library that can be configured to use protocols optimized to tolerate crash failures only or Byzantine failures. In all experiments, we configured BFT-SMART to tolerate crash failures. BFT-SMART was developed in Java and its atomic broadcast protocol executes a sequence of consensus instances, where each instance orders a batch of commands. To further improve the performance of BFT-SMART ordering protocol, we implemented interfaces to enable clients to send a batch of commands in the same message. The experimental environment was configured with 7 machines connected to a 1Gbps switched network. The machines were configured with the Ubuntu Linux 18.04 operating system and a 64-bit Java virtual machine version 10.0.2. BFT-SMART was configured with 3 replicas hosted in separate machines (Dell PowerEdge R815 nodes equipped with four 16-core AMD Opteron 6366HE processors running at 1.8 GHz and 128 GB of RAM) to tolerate up to 1 replica crash. Up to 800 clients were distributed uniformly across another 4 machines (HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of RAM). The experiments with the graph data structures alone without integration in a SMR were executed in one of the Dell PowerEdge machines.

6.2 Applications

We implemented four applications: a linked list, a Key-Value (KV) Store, an authenticated KV-Store, and a digital coin. The linked list was used to microbenchmark the system under different workloads, ranging the number of shards and conflicts, as well as the operations costs.

We implemented both single- and multi-sharded services based on a linked list. For a single-sharded service, we used a linked list application with operations to check whether an entry (i.e., an integer) is in the list ($contains$) and to include an entry in the list (add), representing a readers-and-writers service. Operation $contains(i)$ returns *true* if entry i is in the list, otherwise it returns *false*; operation $add(i)$ includes i in the list and returns *true* if i is not in the list, otherwise it returns *false*. In this context, $contains$ commands do not conflict with each other but conflict with add commands, which conflict with all commands. For a multi-sharded service, we used a set of linked lists (one per shard) application with single-shard and multi-shard operations to check whether an entry is in a subset of lists ($contains_S(i)$ executes $contains(i)$ in the lists associated with each shard $k \in S$) and to include an entry in a subset of lists ($add_S(i)$ executes $add(i)$ in the lists associated with each shard $k \in S$). In this case, $add_{S'}$ conflicts with each $contains_{S''}$ and with each $add_{S''}$ if $S' \cap S'' \neq \emptyset$.

Hereafter, we refer to operations that check whether an entry is in one or more lists and to operations that include an entry in one or more lists as *read* and *write* operations, respectively. Each list was initialized with $1k$ and $10k$ entries at each replica (ranging from 0 to *list size - 1*), representing operations with different execution costs. The integer parameter used in an entry in a read and write operations was a randomly chosen position in the list.

6.3 Optimizations and implementation

The first important optimization we introduced in the hybrid scheduling prototype was the moving of node creation from the classifier (Algorithm 7) to the parallelizers (Algorithm 8) in single-sharded commands. This is possible because there is only one shard involved and parallelizers do not share node information. We also avoid the atomic execution of line 11 in Algorithm 10 and simply execute lines 12 and 13 because the if statement will always return true in these cases. Another important remark about the hybrid scheduling is that we opted to assign workers per shard instead of global workers. This approach presented a better performance since fewer workers share common structures (mainly semaphores) and they already know the shard to lookup for a free command when unblocked.

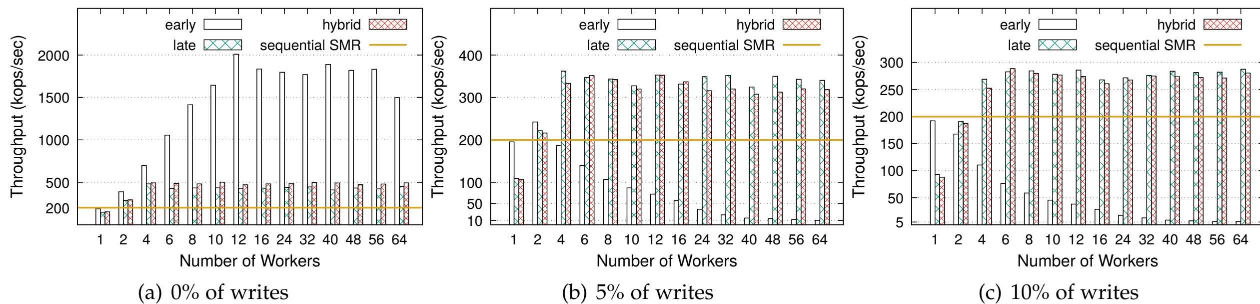
We used an efficient lock-free single-producer single-consumer implementation [22] for the FIFO queues used in the early and hybrid techniques. We also configured the maximum size of the dependency graph to 150 entries for the late and hybrid approaches. In the experiments, we measured the throughput of the system at the servers and the latency of each command at the clients. In the experiments with the data structures alone, we measured only the overall throughput obtained by the worker threads since it does not make sense to compute the latency in this case. A warm-up phase preceded each experiment.

6.4 Single-sharded systems

The first set of experiments considers reads and writes in a single shard. Figures 2 and 3 show the throughput presented by each scheduling approach for different execution costs and number of worker threads, considering three workloads: a workload with read operations only, a mixed workload composed of 5% of writes and 95% of reads, and a mixed workload composed of 10% of writes and 90% of reads. For the mixed workloads, read and write operations were uniformly distributed among clients.

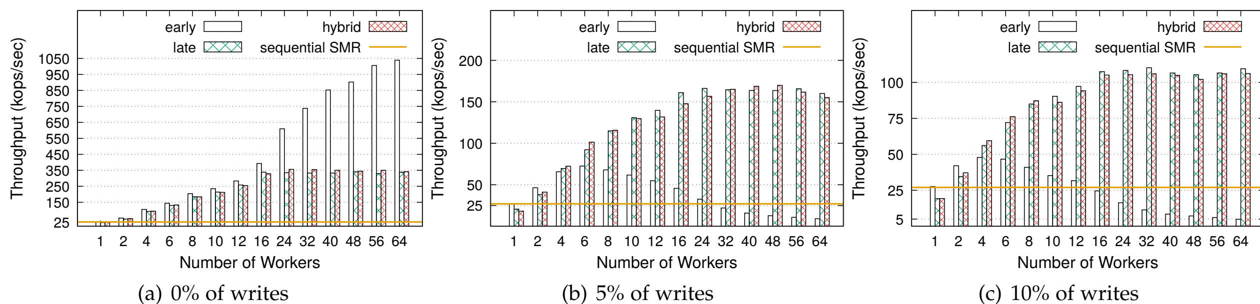
In general, early scheduling excels in workloads without conflicts, but the performance decreases abruptly when conflicts are present. This happens because the processing at the classifier is fast, but workers need to synchronize to execute conflicting commands. Moreover, increasing the number of workers decreases performance in the workloads with conflicts. In fact, the mapping of requests classes to worker threads is a complex optimization problem [12].

Late scheduling uses a lock-free graph to track command dependencies [13] and the parallelizer is responsible for maintaining the graph by inserting and removing commands (recall that workers only mark a command as logically removed). Although this approach better balances the workload among the workers, the parallelizer becomes



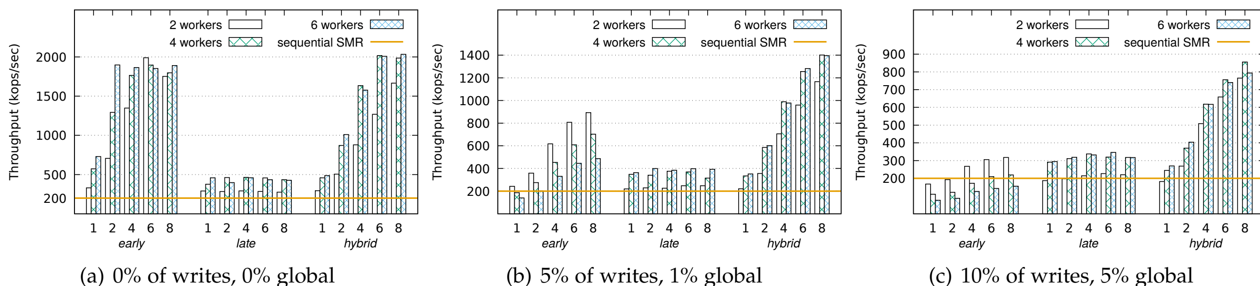
(a) 0% of writes (b) 5% of writes (c) 10% of writes

Fig. 2. Throughput for different percentage of writes and number of workers (list of 1k entries).



(a) 0% of writes (b) 5% of writes (c) 10% of writes

Fig. 3. Throughput for different percentage of writes and number of workers (list of 10k entries).



(a) 0% of writes, 0% global (b) 5% of writes, 1% global (c) 10% of writes, 5% global

Fig. 4. Throughput for different percentage of writes and global operations for systems configured with different number of shards and worker threads (list of 1k entries). For early and hybrid approaches, the number of workers is per shard, while in the late scheduling this number represent the total workers.

a bottleneck and performance does not increase with additional workers. Hybrid scheduling performs similar to late scheduling since for systems with only one shard both approaches work similarly.

6.5 Multi-sharded systems

Figure 4 presents the results for a multi-sharded system considering different number of shards and workers and three workloads: a workload with single-shard read operations only; a mixed workload composed of 5% of writes and 95% of reads, out of which 1% are multi-shard operations and 99% are single-shard operations; and a mixed workload composed of 10% of writes and 90% of reads, out of which 5% are multi-shard operations and 95% are single-shard operations. Single-sharded operations are uniformly distributed among shards and all operations are uniformly distributed among clients. Moreover, 10% of multi-shard operations involve all shards while the remaining ones are addressed to two shards randomly chosen.

Since hybrid scheduling uses one parallelizer per shard to insert (and remove) commands in (from) a subgraph, its performance scales with the number of shards. It also reaches the peak throughput of early scheduling in the workload with only read commands, when configured with 6 or more shards. Early scheduling does not scale with more than 2 workers per shard with conflicting commands and the performance of late scheduling is limited by the thread that maintains the graph. Notice that late scheduling was executed with fewer workers than the others since the parallelizer becomes a performance bottleneck with few workers.

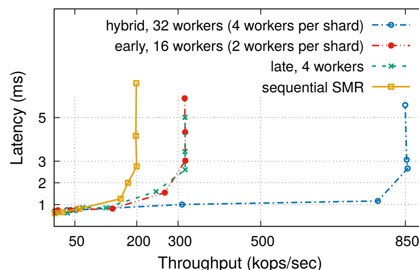


Fig. 5. Latency versus throughput for a system with 8 shards and a workload composed by 10% of writes and 5% of global operations (list of 1k entries).

Figure 5 shows the latency versus throughput results for the configurations that better performed with 8 shards and the workload with 5% global and 10% of writes (Figure 4(c)). In the approaches for parallel SMR, all commands have similar latency because they have similar execution costs and the synchronization of writes impacts the performance of reads ordered after a write. Obviously, the same behavior occurs in classic SMR. Consequently, Figure 5 presents the average latency considering all operations. It is possible to observe that all approaches presented similar latency until near system saturation and from this point on, latency increases abruptly. Since the same behavior occurs for the other configurations and workloads, we present only these cases. The same behavior is reported in previous works on SMR (e.g., [12], [13], [21]).

6.6 Skewed workloads

Figure 6 presents the throughput for a system with 4 shards considering the same workloads presented in the previous section (balanced) and also for cases in which one shard receives 50% of the single-sharded operations and the remaining ones are uniformly distributed across the other three shards (skewed). For each technique, we used the configuration that in general presented best performance for balanced workloads (Figure 4).

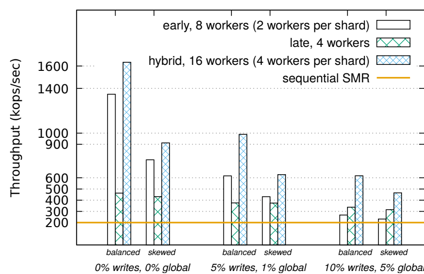


Fig. 6. Throughput for a system with 4 shards considering balanced and skewed workloads (list of 1k entries).

Late scheduling presented similar performance for skewed and balanced workloads since it does not distinguish between shards. The performance of early and hybrid scheduling decreases in skewed workloads since while some shard is overwhelmed, others have fewer work to process. However, hybrid scheduling still outperforming the others by a large margin.

6.7 Data structures performance

This section reports the performance for the data structures alone (i.e., without integration in a SMR). We consider one replica, where a thread loops over a list of pre-created requests (to spare creation times). We executed the experiments for the read only and no global operations workload since it is the one that most challenges SMR.

The results presented in Figure 7 report a similar performance for the techniques when integrated in the SMR framework (Figure 4(a)). This means that overall performance is limited by the synchronizations inside each replica as well as the time demanded to create objects (e.g., nodes to be inserted in the graph and requests from the serialized received commands to be delivered at upper layers).

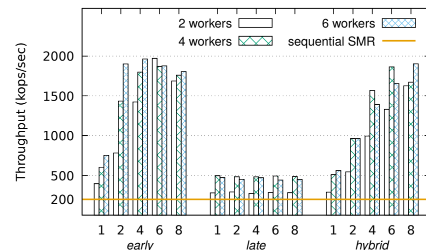


Fig. 7. Throughput for the workload composed of read only and no global operations for systems configured with different number of shards and worker threads (list of 1k entries). For early and hybrid approaches, the number of workers is per shard, while in the late scheduling this number represent the total workers.

6.8 Other applications

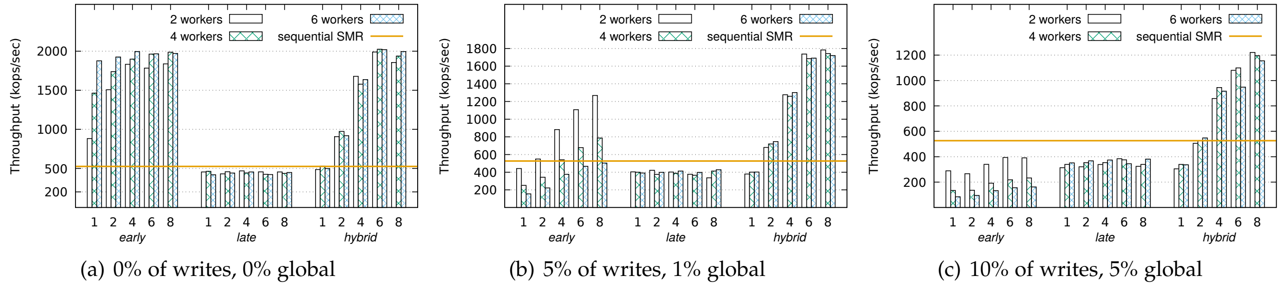
In this section, we comment on three other applications implemented to understand the performance of the proposed techniques: a KV-Store, an Authenticated KV-Store, and a Digital Coin.

KV-Store. We implemented a sharded KV-Store based on a set of tree maps (one per shard), with an operation to write (put) a (key, value) pair and another to read (get) a value associated to a key, where keys and values are integers. The concurrency model for this application is equal to the one for the linked lists. A KV-Store is particularly important because it is broadly used in many large online services (e.g., Twitter [23], Amazon [24] and Facebook [25]). For example, Twitter uses a KV-Store to store tweets that are usually written once and read multiple times. Such applications have a workload that contains mostly read operations and state that is easy to shard based on the keys values, as explained in Section 1.

Figure 8 presents the throughput of the KV-Store considering different configurations. The read and write operation percentages are distributed as described in Section 6.5. Although the performance results are similar to the ones reported for the linked list (Figure 4), it is interesting to notice that the operation cost for this application is small and, consequently, the sequential execution outperforms the late scheduler in all scenarios. In fact, it is cheaper to execute a put in a tree map than to include and remove a request in the dependency graph.

Authenticated KV-Store. Some KV-Stores provide authenticated access to increase security (e.g., [26], [27]). In this case, clients sign their requests and servers must verify the signatures to define if a client has permission to store or retrieve a value. Figure 9 presents performance values for our authenticated KV-Store. The scheduling overhead does not significantly impact performance since the operation costs in this case are high due to signature verification. In all setups, hybrid scheduling outperforms the other solutions.

Digital Coin. Finally, we implemented a digital coin application based on the UTXO (Unspent Transaction Output) model. This model is used by Bitcoin [28] to avoid double spending. In this context, recent work has shown that traditional SMR sequential execution can lead to poor performance [29]. We implemented operations (transactions) to issue a coin, to return an account balance, and to transfer coins among two user accounts. Similar to the KV-Store, the state of this application can be sharded based on the accounts



(a) 0% of writes, 0% global (b) 5% of writes, 1% global (c) 10% of writes, 5% global

Fig. 8. Throughput for different percentage of writes and global operations for a KV-Store systems configured with different number of shards and worker threads. For early and hybrid approaches, the number of workers is per shard, while for late scheduling we show the total number of workers.

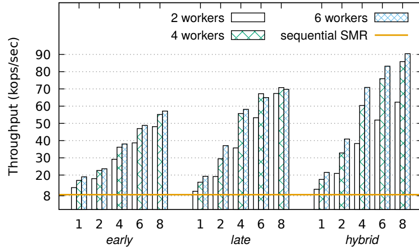


Fig. 9. Authenticated KV-Store throughput for the workload composed of 10% of writes and 5% global. For all approaches, the number of workers is per shard.

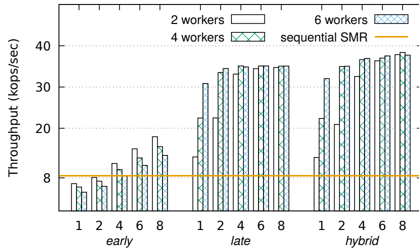


Fig. 10. Digital coin throughput for a workload composed by only transfer operations. For all approaches, the number of workers is per shard.

identifiers. In our concurrency model for this application, two operations conflict if they share a common user. For example, a transfer operation contains the sender and the receiver of some amount of coins and two transfers conflict if they share a common user (sender or receiver). However, all transfers for the same shard conflict in the early scheduler model, since it is not possible to infer that two different workers of a shard will execute in parallel only transfers without a common user. Bitcoin uses an even less restrictive concurrency model where two transactions conflict if they spend the same UTXO. Our model is based on the users because a per user list of valid UTXOs must be updated.

Figure 10 presents the throughput for a workload composed by transfer operations only. The users were uniformly distributed among the shards and randomly chosen, and 70% of the operations were between users in the same shard. Depending on the workload, even higher rates of single-shard operations can be obtained by moving accounts to the same shard based on affinity [30].[4] The transfer operations are costly because it is necessary to verify a signature and create/destroy UTXOs. Consequently, the late and hybrid approaches presented similar performance since their scheduling overheads are not significant. However, early scheduling performed poorly because it needs additional synchronization at the workers since all transfers to the same shard conflict.

4. Clustering coefficients of many current cryptocurrencies transaction graphs (e.g., 0.15 for Ethereum and Z-Cash; and 0.05 for Bitcoin) are much higher than coefficients for random graphs (0.0037) [31], [32].

7 Related work

This paper is at the intersection of two areas of research: concurrent graph structures and state machine replication.

7.1 Concurrent graph structures

In [33], a concurrent graph is proposed to compute serializable executions. Nodes and edges in the graph represent transactions and their conflicts, respectively. Whenever a transaction is added, edges are included to represent conflicts. In case of cycles, vertices and edges are removed to keep the graph acyclic. The graph is implemented as a linked list. Nodes and edges can be created individually and concurrently. The node list is ordered according to a key. The synchronization strategy is *lazy* [34], or optimistic: in a first step, the list is searched without locks. Once the right position to operate on the list is found, locks are acquired on needed nodes. Once locks are obtained, a validation is performed to check if the conditions during search are still valid for those nodes. If not, the operation is repeated. Upon deletion, the node is first marked as logically deleted, then locking and actual removal take place. Nodes and edges are manipulated independently. The strategy allows wait-free operations to traverse the graph to check if a node is in the list as well as to detect cycles.

In [35], the authors propose a concurrent graph, represented as an adjacency matrix. It contains a fixed vertex set and allows concurrent operations to insert, remove or modify weights of edges. A dynamic traversal is proposed to obtain a consistent view, i.e., the weights of all edges visited have co-existed at some point in time despite concurrent modifications. Operations are wait-free [36], achieved using a helping mechanism [37]. Operations concurrent to updates help updates to carry out edge modifications.

A concurrent, unbounded and directed graph is proposed in [38]. Addition, removal and lookup on the sets of vertices and edges are supported on a lock-free basis, while a reachability query is obstruction-free. It also uses a helping strategy to achieve lock-freedom.

Both our lock-free graph algorithm and the ones described above build on basic principles to allow concurrent access to a shared data structure. However, while our algorithm implements a COS, the ones above implement operations on a single node or a single edge. It is unclear how to implement COS using these approaches.

7.2 State machine replication

It has been early observed that independent commands can be executed concurrently in SMR [2]. Previous works have shown that many workloads are dominated by independent commands, which justifies strategies for concurrent execution (e.g., [8], [9], [10]). Existing proposals to parallelize the execution in state machine replication differ in the strategy and architecture to detect and handle conflicts. We can broadly classify related work in three groups.

7.2.1 Techniques based on application knowledge

A scheduler that serializes the execution of dependent commands and dispatches independent commands to be processed in parallel by a pool of worker threads is an example of technique that exploits application knowledge (i.e., in the form of dependent and independent commands) [8], [9], [12], [20]. This idea has also been explored in transactional systems, where information about data items accessed by transactions can be inferred a priori (e.g., [39], [40]). The schedulers studied in this paper use application knowledge.

7.2.2 Techniques oblivious to application knowledge

Rex [41], and CRANE [42] are techniques oblivious to application knowledge but resort to more complex runtime architectures to coordinate replicas and ensure consistent parallel execution. Instead of using a *consensus-execution* model as in SMR, Rex [41] uses an *execute-agree-follow* strategy. In Rex, a single primary server receives requests and processes them in parallel using different threads. While executing, the primary logs a trace of dependencies among requests based on the shared variables accessed (locked and unlocked) by each thread. Periodically it proposes a consistent cut of 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, following the causality on lock and unlock operations. While in SMR different consensus instances are independent, traces in Rex consensus instances are not since they have to satisfy the condition that one is a prefix of the other. Trace synchronization may incur in high network bandwidth consumption and performance overhead [42].

CRANE [42] solves non-determinism during run-time using several run-time mechanisms. The socket interface is augmented to perform agreement (using an underlying Paxos implementation) on the sequence of incoming calls across replicas. Thread synchronization uses deterministic multithreading (DMT) [43]. Additionally, CRANE introduces a *time bubbling* technique to enforce deterministic logical times for request bursts. However, the runtime overhead is non-negligible. Besides agreeing on each socket event, the DMT system incurs 12.7 % of overhead.

Deterministic schedulers have also been studied considering models more similar to the SMR model [44], [45]. These solutions provide a lock-level concurrency but replicas must execute a distributed consensus to define the order of lock requests. This strategy decreases the concurrency granularity (lock-level) but it imposes a considerable overhead since it needs to order lock requests.

Techniques in this class, including those using deterministic multithreading, tend to depart from the SMR model, which is based on input replication followed by independent execution at replicas. Instead, coordination among replicas is needed during the execution phase, using consensus, as reported. This penalizes performance, makes the design of the replicated service more complex and reduces the possible use of equivalent but different replica implementations at service level (e.g., for diversity).

7.2.3 Optimistic techniques

Finally, some techniques employ optimistic strategies to parallelize commands. In [7], [10] replicas execute commands in parallel as they arrive and then check for consistency after execution. If violated replicas coordinate to reexecute. In [46] application specific knowledge is used to predict 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 replicas have to establish a deterministic execution order in cooperation with other replicas, using a consensus protocol.

8 Conclusion

Parallel state machine replication techniques allow independent commands to be executed concurrently in a replica. To keep replicas consistent, each replica has to carefully handle and respect dependencies among commands. This is a non-trivial task since it requires dependency detection on a possibly high volume of commands. In this paper, we consider two classes of parallel SMR techniques and introduce a hybrid approach that leverages the advantages of the existing approaches. A detailed experimental evaluation showed that the hybrid approach outperforms the techniques it builds upon.

Acknowledgments

This work is partially supported by CAPES - Print - PUCRS and the Swiss Nacional Science Foundation (SNSF).

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 services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

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

[4] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.

[5] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006.

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *ATC*, vol. 8, 2010.

[7] 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.
[8] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, 2004.
[9] P. J. Marandi, C. E. B. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *ICDCS*, 2014.
[10] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *SRDS*, 2014.
[11] 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.
[12] E. Alchieri, F. Dotti, and F. Pedone, "Early scheduling in parallel state machine replica," in *ACM SoCC*, 2018.
[13] I. Escobar, E. Alchieri, F. Dotti, and F. Pedone, "Boosting concurrency in parallel state machine replication," in *Middleware*, 2019.
[14] E. Batista, E. Alchieri, F. Dotti, and F. Pedone, "Resource utilization analysis of early scheduling in parallel state machine replication," in *9th Latin-American Symposium on Dependable Computing*, 2019.
[15] 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.
[16] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, 2nd ed. Addison-Wesley, 1993.
[17] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, vol. 43, no. 2, pp. 225–267, 1996.
[18] 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.
[19] 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.
[20] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone, "Reconfiguring parallel state machine replication," in *SRDS*, 2017.
[21] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with bft-smart," in *DSN*, 2014.
[22] V. Maffione, G. Lettieri, and L. Rizzo, "Cache-aware design of general-purpose single-producer-single-consumer queues: Cache-aware single-producer-single-consumer queues," *Software: Practice and Experience*, vol. 49, 12 2018.
[23] T. Manhattan, "Manhattan, our real-time, multi-tenant distributed database for Twitter scale," sep 2021. [Online]. Available: https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-
[24] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
[25] S. Masti, "How we built a general purpose key value store for Facebook with ZippyDB," Aug. 2021. [Online]. Available: https://engineering.fb.com/2021/08/06/core-data/zippydb/
[26] y. GUO, X. Yuan, X. Wang, C. Wang, B. Li, and X. Jia, "Enabling encrypted rich queries in distributed key-value stores," *IEEE TPDS*, vol. 30, no. 6, pp. 1283–1297, 2019.
[27] L. Chen, W. Dai, M. Qiu, and N. Jiang, "A design for scalable and secure key-value stores," in *IEEE SmartCloud*, 2017.
[28] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://bitcoin.org/bitcoin.pdf
[29] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From byzantine replication to blockchain: Consensus is only the beginning," in *IEEE/IFIP DSN*, 2020.
[30] L. Hoang Le, E. Fynn, M. Eslahi-Kelorazi, R. Soulé, and F. Pedone, "Dynastar: Optimized dynamic partitioning for scalable state machine replication," in *IEEE ICDCS*, 2019, pp. 1453–1465.
[31] C. I. V. G. Kondor D, Posfai M, "Do the rich get richer? an empirical analysis of the bitcoin transaction network," *PLoS ONE*, vol. 9, no. 2, 2014.
[32] B. B. Motamed, A.P., "Quantitative analysis of cryptocurrencies transaction graph," *Applied Network Science*, vol. 4, no. 131, 2019.
[33] S. Peri, M. Sa, and N. Singhal, "Maintaining acyclicity of concurrent graphs," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1611.03947
[34] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
[35] N. D. Kallimanis and E. Kanellou, "Wait-Free Concurrent Graph Objects with Dynamic Traversals," in *OPODIS*, 2015.
[36] M. Moir and N. Shavit, "Concurrent data structures," in *Handbook of Data Structures and Applications.*, 2004.
[37] M. Herlihy, "A methodology for implementing highly concurrent data structures," *SIGPLAN Not.*, vol. 25, no. 3, pp. 197–206, 1990.
[38] B. Chatterjee, S. Peri, M. Sa, and N. Singhal, "A Simple and Practical Concurrent Non-blocking Unbounded Graph with Reachability Queries," *ArXiv e-prints*, Sep. 2018.
[39] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE TKDE*, vol. 15, no. 4, pp. 1018–1032, Jul. 2003.
[40] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *SIGMOD*, 2012.
[41] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: Replication at the speed of multi-core," in *EuroSys*, 2014.
[42] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos made transparent," in *25th Symposium on Operating Systems Principles*, 2015.
[43] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 97–108, 2009.
[44] G. Habiger, F. J. Hauck, J. Köstler, and H. P. Reiser, "Resource-efficient state-machine replication with multithreading and vertical scaling," in *European Dependable Computing Conference*, 2018.
[45] G. Habiger, F. J. Hauck, H. P. Reiser, and J. Köstler, "Self-optimising application-agnostic multithreading for replicated state machines," in *SRDS*, 2020.
[46] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: Optimistic deterministic multithreading." in *HotDep*, 2010.

Aldenio Burgos is a master student in Computer Science from University of Brasília, Brazil. His research interests include the theory and practice of dependable systems, more specifically the replication paradigm and blockchains.

Eduardo Alchieri received the Ph.D. degree from Federal University of Santa Catarina, Brazil, in 2011. He is currently a Professor at the Department of Computer Science of the University of Brasília, Brazil. His research interests include the theory and practice of secure and dependable distributed systems.

Fernando Dotti received the Ph.D. degree from Technical University Berlin, Germany, in 1997. He is currently a full professor with the School of Technology at Pontifical Catholic University of Rio Grande do Sul, Brazil. His research interests include the theory, modelling and analysis of distributed systems, including distributed algorithms and fault tolerance.

Fernando Pedone received the Ph.D. degree from EPFL in 1999. He is a Full Professor with the Faculty of Informatics, Università della Svizzera Italiana (USI), Switzerland. He has been also affiliated with Cornell University, as a Visiting Professor, EPFL, and Hewlett-Packard Laboratories (HP Labs). He has authored more than 100 scientific articles and seven patents. He is a Co-Editor of the book Replication: Theory and Practice. His research interests include the theory and practice of distributed systems and distributed data management systems.