Rethinking State-Machine Replication for Parallelism

Parisa Jalili Marandi, Carlos Eduardo Bezerra, Fernando Pedone
University of Lugano, Switzerland

***Abstract*—State-machine replication, a fundamental approach to designing fault-tolerant services, requires commands to be executed in the same order by all replicas. Moreover, command execution must be deterministic: each replica must produce the same output upon executing the same sequence of commands. These requirements usually result in single-threaded replicas, which hinders service performance. This paper introduces Parallel State-Machine Replication (P-SMR), a new approach to parallelism in state-machine replication. P-SMR scales better than previous proposals since no component plays a centralizing role in the execution of independent commands—those that can be executed concurrently, as defined by the service. The paper introduces P-SMR, describes a "commodified architecture" to implement it, and compares its performance to other proposals using a key-value store and a networked file system.**

I. Introduction

State-machine replication (SMR) is a fundamental approach to designing fault-tolerant services [1], [2]. By replicating the servers, the service remains available for clients even if some replicas fail. Once servers are replicated, consistency among the replicas must be ensured. State-machine replication achieves strong consistency by regulating how client commands are propagated to and executed by the replicas: (i) every nonfaulty replica must receive every command; (ii) replicas must agree on the order of received and executed commands; and (iii) the execution of commands must be deterministic (i.e., a command's changes to the state and results depend only on the replica's state and on the command itself).

State-machine replication is a technique to improve the availability of a service, not its performance. In fact, in some cases a single-server implementation of a service will likely outperform its replicated counterpart since the single server can benefit from concurrency while the replicated servers will be typically sequential. Executing commands sequentially is a serious performance limitation in modern processors, which are essentially parallel (i.e., equipped with multiple processing units, interconnected through multiple network interfaces). Nevertheless, rendering state-machine replication parallel is challenging.

In state-machine replication, upon executing the same sequence of commands replicas evolve through the same sequence of states and produce the same responses. It has been observed, however, that replicas do not need to pass through the same state sequence to produce the same responses [2]. This is the case of commands that access disjoint variables: if commands do not contend for shared variables, replicas can execute them in parallel (i.e., concurrently). Some previous works have build on this observation to introduce parallelism in state-machine replication [3], [4]. In brief (we provide more details in subsequent sections), in these systems the key idea is to deliver commands across replicas in total order but allow the commands to execute concurrently when possible.

In this paper, we seek solutions that not only allow parallelism in state-machine replication but also scale with the hardware resources available at processors (e.g., processing units, network interfaces). The last requirement has two important consequences. First, we should avoid solutions that rely on a single component or on a fixed set of components, an obvious potential performance bottleneck. Second, we must accommodate parallelism in the execution of commands and in the ordering of these commands. Failing to address these issues will result in limited performance improvements as more services run at "main-memory speed" and are deployed on processors equipped with an ever-increasing number of processing units.

Parallel State-Machine Replication (P-SMR), the approach we present in this paper, fulfills the requirements above: it introduces parallelism in the execution of commands and in the protocol used to order these commands; additionally, in the most common cases performance scales with replicas' hardware resources. Similarly to previous proposals, P-SMR exploits service semantics to determine when commands can execute concurrently and when serial execution is needed. This is captured by the notion of dependency between commands: two commands are deemed dependent if they cannot execute concurrently. P-SMR is optimized for independent commands, when concurrency is possible. Services whose state is mostly read (e.g., name services) or can be partitioned so that most commands fall in one partition or another but rarely in both (e.g., file systems) can benefit from P-SMR.

This paper makes the following contributions: First, it introduces a novel approach to parallel state-machine replication (P-SMR) that scales performance with the number of processing units in a replica when commands are independent. Second, it describes a "commodified architecture" for state-machine replication and shows how replicated services built based on this architecture can seamlessly use P-SMR. Third, it shows how P-SMR can be used to boost the performance of two highly available services, a key-value store and a networked file system. Fourth, it details a prototype of P-SMR, assesses its performance, and compares it to other state-machine replication approaches.

The remainder of the paper is structured as follows. Section II describes our system model and assumptions. In Section III, we present an architecture for state-machine replication and motivate the need for P-SMR. In Section IV, we introduce Parallel State-Machine Replication. Section V illustrates how P-SMR can be used with two services. Section VI discusses the implementation of our highly available services using different strategies. Section VII contains a performance evaluation of these systems. Section VIII surveys related work and Section IX concludes the paper.

II. System model and assumptions

We assume a distributed system composed of interconnected processes. There is an unbounded set $C = \{c_1, c_2, ...\}$ of *client* processes and a bounded set $S = \{s_1, s_2, ..., s_n\}$ of *server* 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 malicious and arbitrary process behavior (e.g., no Byzantine failures). Processes are either *correct*, if they never fail, or *faulty*, otherwise. We assume f faulty servers, out of $n = f + 1$ servers. Processes communicate by message passing, using either one-to-one or one-to-many communication. One-to-one communication is through primitives send(m) and receive(m), where m is a message. If sender and receiver are correct, then every message sent is eventually received. One-to-many communication is based on atomic multicast. Atomic multicast is defined by the primitives *multicast*(γ, m) and *deliver*(m), where $\gamma \subseteq S$ is a group of destinations. Let relation $<$ be defined such that $m < m'$ iff there is a process that delivers m before m'. Atomic multicast ensures that (i) if a server delivers m, then all correct servers in γ deliver *m (agreement)*; and (ii) relation $<$ is acyclic *(order)*. The order property implies that if s and r deliver messages m and m', then they deliver them in the same order.

Atomic multicast is typically available to applications as a library (see Figure 1) and implemented using one-to-one communication and additional system assumptions [5], [6] (see Section VI). The replication protocols we consider in the paper use atomic multicast as a “black box" and do not explicitly use these assumptions.

III. Background and motivation

State-machine replication renders a service fault-tolerant by replicating the server and coordinating the execution of client commands among the replicas [1], [2]. 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 response of a command are a function of the state variables the command reads and the command itself.

State-machine replication 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: 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 specifications, and (ii) respects the real-time ordering of commands across all clients [7]. In traditional state-machine replication (SMR), linearizability is achieved by having each replica execute commands sequentially in the same order. Since commands are deterministic, replicas will produce the same state changes and response after the execution of the same command.

A “commodified architecture" for state-machine replication can be organized as follows (see Figure 1). Clients and servers interact in a way similar to remote procedure invocations [8]. Clients access the service by invoking service commands (with the appropriate parameters). Client proxies intercept client invocations, turn them into requests that include a command identifier and the marshaled parameters, and multicast the requests to the replicas. Requests are delivered by the server proxies, which re-assemble invocations and issue them against the local server replica. Each server executes one command at a time. Similarly to remote procedure calls, the client and client proxy (respectively, server and server proxy) can be implemented as a single process, sharing a common address space. The command's response follows the reverse path to the client using one-to-one communication. Even though the client proxy may receive the response for a command from multiple servers, all responses are the same and the proxy returns only one response to the client.

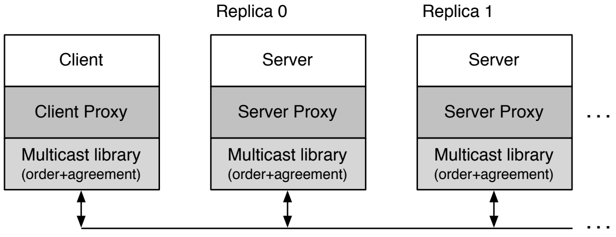

Fig. 1. A simple state-machine replication architecture.

It has been observed that replicas can execute “independent commands" concurrently without violating consistency [2]. Two commands are *independent* if they either access different variables or only read variables commonly accessed; conversely, two commands are *dependent* if they access one common variable v and at least one of the commands changes the value of v. For example, two read commands are independent, while a read and an update command on the same variable are dependent. A few approaches have been suggested in the literature to execute independent commands concurrently with the goal of improving performance [3], [4]. Although these approaches differ in how they achieve concurrency, they totally order commands and each replica delivers commands sequentially as a single stream. Since the execution of independent commands is concurrent but delivery (and possibly scheduling) is sequential, hereafter, we refer to these approaches as *semi-parallel state-machine replication* (sP-SMR).

This paper proposes to parallelize both the execution and the delivery of commands, an approach we refer to as *parallel state-machine replication* (P-SMR) (see Table I). P-SMR uses multiple multicast groups to partially order commands across replicas, where each group leads to a different stream of commands delivered at the replica. P-SMR improves on traditional state-machine replication by allowing independent commands to execute concurrently. It has two advantages over semi-parallel state-machine replication. First, since a replica can handle multiple streams of commands, multicast can be implemented more efficiently. This happens because each command stream is independent of the other, and can use different threads within a node or even involve different sets of nodes per stream. For example, our multicast library uses one Paxos [6] instance per stream, and each stream can have a different set of acceptor nodes (see Section VI-A). Moreover, command streams can be mapped to multiple network interfaces, commonly available in modern servers. Hence, communication at a server is not limited by the bandwidth of a single network interface, but by the aggregate bandwidth of the server's interfaces. Second, commands are delivered by the worker threads directly, instead of having a single thread that first delivers the commands and then assigns them to the workers, thereby reducing overhead.

Command...	SMR	sP-SMR	P-SMR
...delivery	sequential	sequential	parallel
...execution	sequential	parallel	parallel

TABLE I. DEGREES OF PARALLELISM IN STATE-MACHINE REPLICATION.

IV. PARALLEL STATE-MACHINE REPLICATION

In this section, we present P-SMR's design principles (Section IV-A), consider architectural issues (Section IV-B), detail the protocol (Section IV-C), consider its advantages and limitations (Section IV-D), and show that it is linearizable and deadlock-free (Section IV-E).

A. Design rationale

P-SMR's design is guided by two principles:

- *Optimize performance for the common case.* P-SMR targets workloads dominated by independent commands. This is the case of services subject mostly to read commands (e.g., name services) or whose state can be partitioned so that most commands access variables in one partition and rarely in multiple partitions (e.g., file systems).
- *Keep replication transparent.* In the architecture presented in Section III, replication is transparent for clients: details related to communicating with multiple replicas are hidden from the clients and handled by the client proxies and the multicast library. Similarly to SMR, P-SMR should not expose replication details to the client application.

B. Client and server organization

P-SMR builds on the architecture depicted in Figure 1. Client and server proxies are created based on (a) the *signature* of each service command, including the command's identifier and a description of the command's input and output parameters with their types; and (b) the *command dependencies (C-Dep)*, specifying which commands depend on each other. Therefore, in addition to providing the server's code, the service designer must also provide the command signatures and the C-Dep. Although the C-Dep can be automatically generated from the signatures and the server's code, in our prototype C-Deps were created manually.

At the client side, command signatures are used by the client proxy to create a request from the client invocations and return a response to the client. At the server side, command signatures are used by the server proxy to turn delivered requests into local server invocations and assemble the response of commands. In both cases, the process is analogous to the one used in the commodified state-machine replication architecture presented in Section III.

C-Dep is used to automatically compute the *Command-to-Groups (C-G)* function, used by the client proxy to determine the multicast groups a request must be multicast to and by the server proxy to coordinate the local execution of dependent commands—more details in the next section. Similarly to SMR, a client application in P-SMR will be oblivious to replication. Moreover, since coordination among worker threads, in the case of dependent commands, is handled by the server proxy, a server designed for state-machine replication will work unchanged in P-SMR.

C. Protocol design

P-SMR takes as input the command dependencies (C-Dep) of a service and the desired multiprogramming level (MPL) at the replicas to define how independent commands can be executed concurrently and dependent commands are synchronized. The multiprogramming level is a parameter of the system that defines the number of worker threads at the servers. It can be set, for example, based on the number of processing units (i.e., cores) at the servers. In a configuration where MPL is set to k, we identify worker threads as $t_1, \ldots, t_k$. P-SMR organizes threads in k multicast groups such that the i-th thread of each replica, t_i, belongs to group g_i.

Basic principle. A client proxy executes command C by multicasting a request with C to a set of destination groups, computed by the C-G function (see Algorithm 1). Worker threads at the server proxy deliver commands and invoke their execution against the local server. The execution of a worker thread alternates between two modes: The thread is in *parallel mode* when it delivers a command multicast to a single group and in *synchronous mode* when it delivers a command multicast to multiple groups. In parallel mode, upon delivering C, thread t_i executes C, sends C's response to the client and waits for the next command. In synchronous mode, threads that deliver C, hereafter identified as τ_C, synchronize using barriers: threads in τ_C send a signal to one designated thread $t_i \in \tau_C$ (signal (a) in Figure 2) and wait for a signal from t_i; after t_i receives the signals it executes C, sends C's response to the client, and signals threads in τ_C to continue with the next command (signal (b) in Figure 2). Note that two commands in P-SMR will be ordered consistently across replicas if they are multicast to the same group or they are dependent.

Defining command dependencies. In our prototype, C-Dep can encode two levels of dependency information: commands that depend on each other regardless their parameters (i.e., regardless the objects the command accesses) and commands that depend on each other according to their parameters (e.g., two updates on the same object). C-Dep includes all such interdependencies; if no entry exists in C-Dep asserting the dependency of two commands, they are independent. For example, a straightforward C-Dep can simply distinguish commands that read the service's state, those that can be executed concurrently, from commands that modify the service's state, those that must be synchronized. In Section V we show how our scheme can represent more complex interdependencies using two general services, a key-value store and a networked file system.

Mapping commands to destinations. The client proxy determines the destination groups of a command using a Command-to-Group (C-G) function that maps the command id and its

Algorithm 1: Parallel State-Machine Replication (P-SMR)

```
A client proxy c executes a call to command C with
    identifier cid and input and output parameters as follows:
  $\gamma \leftarrow C\text{-}G(cid, input)$          {$\gamma$ is the set of groups involved in C}
  multicast($\gamma$, [$cid, input$])
  wait for first response
  $output \leftarrow$ response
  return
Thread $t_i$ at a server proxy executes a command as follows:
  upon deliver([$cid, input$]), multicast by $c$
    $\gamma \leftarrow C\text{-}G(cid, input)$
    if $\gamma$ is a singleton then
      // Thread $t_i$ is in parallel mode
      execute $cid$ with $input$ parameters
      send response to $c$
    else
      // Thread $t_i$ is in synchronous mode
      $e \leftarrow min\{j : g_j \in \gamma\}$          {pick a thread deterministically}
      if $i = e$ then
        for each $j \neq i$ such that $g_j \in \gamma$
          wait for signal from $t_j$
        execute $cid$ with $input$ parameters
        send response to $c$
        for each $j \neq i$ such that $g_j \in \gamma$
          signal $t_j$          {let thread $t_j$ resume its execution}
      else
        signal $t_e$
        wait for signal from $t_e$          {thread $t_i$ pauses its execution}
```

input parameters to a set of multicast groups. C-G is part of the client proxy; it is created based on the multiprogramming level and the service's C-Dep. Computing C-G requires solving an optimization problem that seeks to maximize concurrency among independent commands while keeping dependent commands synchronized. Concurrent execution of independent commands is achieved by assigning the commands to different groups; proper synchronization of dependent commands amounts to assigning at least one common group to any two dependent commands.

The amount of concurrency in a service depends on the interdependencies among the service's commands. A C-Dep that tightly captures interdependencies will likely result in more concurrency at the replicas. For example, consider a service with `get_state(in: int x, out: char[] v)` and `set_state(in: int x, char[]v)` commands, where `x` is an object identifier and `v` an object value. A simple C-Dep would state that `set_state` depends on any other command, regardless the object accessed. Defining such a C-Dep requires inspecting commands `get_state` and `set_state` and concluding that the first reads the service's state and the second modifies the service's state. A C-G for this C-Dep could assign a `get_state` command to a single group and a `set_state` command to all groups, as shown next, where `get_state` is assigned to a random group between 1 and k (recall that k is the multiprogramming level):

```
function $C\text{-}G(cid)$
  switch $(cid)$
    case get_state: return(random($1..k$))
    case set_state: return($ALL\_GROUPS$)
```

A more complex C-Dep identifies that `set_state` depends only on other commands on the same object x. In this case, a C-G could assign commands on the same object to the same group and commands on different objets to different groups:

```
function $C\text{-}G(cid, x)$
  return($(x \bmod k) + 1$)
```

The result is that commands assigned to different groups can execute concurrently, even if they both modify the state of objects. Moreover, additional information, if available, can be used when computing the C-G function. For example, objects that are commonly accessed could be assigned to different groups, allowing increased concurrency.

D. Advantages and limitations

In the following, we compare P-SMR to different approaches to state-machine replication according to three aspects: performance, transparency, and load balancing.

Performance. While P-SMR and sP-SMR improve the performance of state-machine replication by allowing independent commands to execute concurrently, P-SMR has two advantages with respect to sP-SMR. First, P-SMR offloads scheduling decisions from the replicas, avoiding a bottleneck-prone scheduler, which must deliver a single stream of commands and assign them to the worker threads for execution. This is important in time-critical services running at main-memory speed and in modern servers equipped with an ever-increasing number of processing units. Second, since replicas in P-SMR can handle multiple parallel streams of commands, they can make better use of local hardware resources (e.g., command streams can be distributed among multiple network interfaces) and allow efficient multicast implementations, with different sets of nodes responsible for ordering different streams of commands (see Section VI-A for more details).

Transparency. Both P-SMR and sP-SMR require more information about a service than state-machine replication. In both P-SMR and sP-SMR, commands that depend on each other must be identified (the C-Dep structure), although some implementations of sP-SMR [3] can cope with mistakes in C-Dep (see Section VIII). State-machine replication does not need C-Dep since commands are executed sequentially. In both P-SMR and sP-SMR, however, the client application is oblivious to these details, which must be identified by the service designer or automatically inferred from the server's code. In P-SMR, the client proxy uses the C-G function, derived from C-Dep, to identify the set of groups each command must be multicast to; in sP-SMR, the scheduler uses C-Dep to schedule independent commands concurrently and dependent commands sequentially. Additionally, in P-SMR client and server proxies must agree on the multiprogramming level used at the servers.

Load balancing. In P-SMR commands are assigned to working threads *statically* using the C-G function, which is computed based on the desired multiprogramming level and the command dependencies; in sP-SMR, commands are assigned to working threads *dynamically* by the scheduler. Consequently, load balancing in P-SMR is more limited than in sP-SMR. For example, in the service described in Section IV-C, if heavily accessed objects are assigned to the same group, then one worker thread will end up executing more commands than the other worker threads. If heavily accessed objects are known

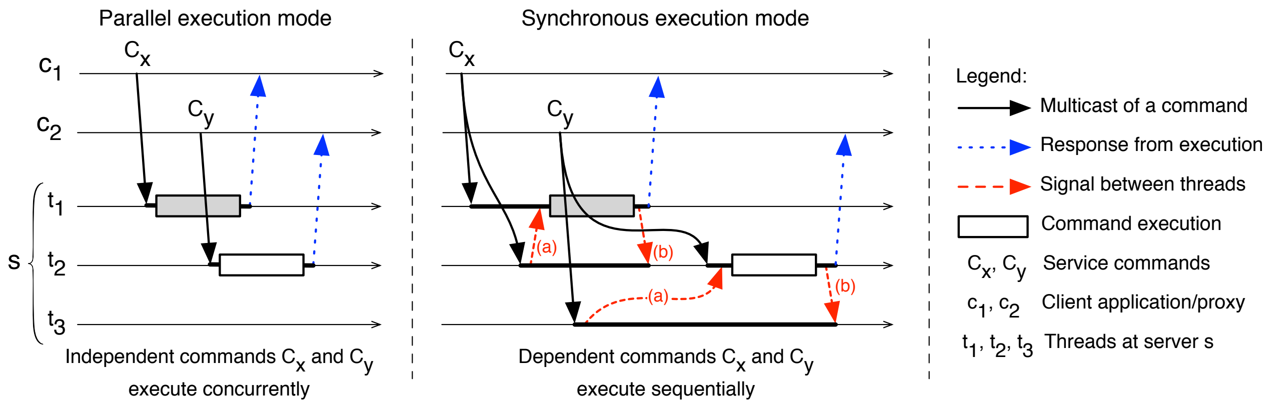
Parallel execution mode
Synchronous execution mode
C_x
C_y
c1
c2
t1
t2
t3
s
(a)
(b)
(a)
(b)
Independent commands Cx and Cy execute concurrently
Dependent commands Cx and Cy execute sequentially
Legend:
Multicast of a command
Response from execution
Signal between threads
Command execution
Cx, Cy Service commands
c1, c2 Client application/proxy
t1, t2, t3 Threads at server s

Fig. 2. Two execution modes in P-SMR, parallel (left) and synchronous (middle). For clarity, we show the execution of clients c_1 and c_2 against a single server replica s with three worker threads, t_1,t_2 and t_3. Signals (a) and (b) are explained in Section IV-C.

in advance, this information can be used when computing the C-G function so that such objects are assigned to distinct groups. Accommodating dynamic changes in access patterns, however, would require recomputing C-G. In Section VII we assess the performance of P-SMR under skewed workloads and compare the results to other approaches.

E. Correctness

We show that P-SMR is linearizable and deadlock-free.

P-SMR is linearizable. From the definition of linearizability (see Section III), we show that there is a permutation π of commands in $\mathcal{E}$ that respects (i) the real-time ordering of commands across all clients, and (ii) the semantics of the commands. Let C_x and C_y be two commands in $\mathcal{E}$ submitted by clients c_x and c_y, respectively.

There are two cases to consider.

Case (a): C_x and C_y are independent. Thus, either C_x and C_y access disjoint sets of variables or only read variables commonly accessed. Consequently, the execution of one command does not affect the execution of the other and they can be placed in any relative order in π. We arrange C_x and C_y in π so that their relative order respects their real-time dependencies, if any.

Case (b): C_x and C_y are dependent. Assume C_x and C_y are multicast to groups in γ_x and γ_y, respectively. From the fact that C_x and C_y depend on each other, $\gamma_{xy} = \gamma_x \cap \gamma_y \neq \emptyset$. In every correct server s, C_x (resp. C_y) is delivered by all threads in groups in γ_x (resp. γ_y) and executed by one thread, say t_x (resp. t_y). From the order property of atomic multicast, every thread in groups in γ_{xy} delivers C_x and C_y in the same relative order. Without lack of generality, assume C_x is delivered before C_y.

We first claim that t_x executes C_x before t_y executes C_y and the execution satisfies the sequential semantics of the commands. To see why, notice that t_x only executes C_x after t_i delivers C_x and every other thread in groups in γ_x delivers C_x and signal t_i. Every thread $t \neq t_i$ in a group in γ_x then waits until t_i executes C_i to proceed with the next command. Thus, t_y will only receive a signal from threads in groups in γ_{xy} and execute C_y after t_x has executed C_x. Consequently, the two commands execute in sequence, which satisfies their semantics.

We now claim that the delivery order satisfies any real-time constraints among C_x and C_y. Without lack of generality, assume C_x finishes before C_y starts, that is, C_x precedes C_y in real time. Thus, before C_y is multicast by a client, C_x has completed (i.e,. its client has received C_x's response). The claim follows from the fact that before C_x is executed, it must be multicast, and thus C_x is delivered before C_y.

From the claims above, we can arrange C_x and C_y in π according to their delivery order so that the execution of each command satisfies its semantics.

P-SMR is deadlock-free. For a contradiction, assume a deadlock where thread x_1 waits for $x_2,\ldots,x_l$ waits for x_1. Let $p(x)$ (resp. $n(x)$) be the thread that precedes (resp. succeeds) x in the deadlock chain. Thread x waits for $n(x)$ if (1) there is a command $C_{x,n(x)}$ multicast to groups that contain x and $n(x)$; (2) x delivered $C_{x,n(x)}$; and (3) x needs a signal from $n(x)$ (a) before x executes $C_{x,n(x)}$ or (b) after $n(x)$ executes $C_{x,n(x)}$.

We now claim that x delivers $C_{x,n(x)}$ before $C_{p(x),x}$, that is, $C_{x,n(x)} < C_{p(x),x}$. To see why, assume x delivers $C_{p(x),x}$ before $C_{x,n(x)}$. From the algorithm, when x delivers $C_{x,n(x)}$ it has (a) sent a signal to $p(x)$, if $p(x)$ was to execute $C_{p(x),x}$ or (b) received a signal from $p(x)$, if x was to execute $C_{p(x),x}$. In both cases, $p(x)$ cannot wait for x, as assumed in our deadlock chain.

From the claim above, $C_{x_l,x_1} < C_{x_{l-1},x_l} < \ldots < C_{x_l,x_1}$, which contradicts the atomic multicast order property.

V. Services

In this section, we show how highly available services replicated using state-machine replication can use P-SMR. We consider two services, a key-value store and a networked file system.

A. Key-value store

The key-value store implements commands to read and modify an in-memory database, as presented below. An insert includes key k and value v in the database and possibly returns an error code (e.g., out of memory). A delete removes k from the database or returns an error code if the entry does not exist. A read returns the value of k and an update replaces the current value of k with v. In both cases, an error code is returned if the key does not exist.

- `insert(in:int k, char[] v, out:int err)`
- `delete(in:int k, out:int err)`
- `read(in:int k, out:char[] v, int err)`
- `update(in:int k, char[] v, out:int err)`

The main key-value store's data structure is a B$^+$-tree. While a read does not result in any changes in the tree, an update changes a single entry, the one corresponding to the provided key (if present). Inserts and deletes may modify multiple entries, depending on the structure of the tree when the command is executed (i.e., requiring partitioning and joining tree cells). Therefore, we establish the following dependencies between commands: inserts and deletes depend on all commands; an update on key k depends on other updates on k, on reads on k, and on inserts and deletes.

B. Networked File System

The Networked File System (NetFS) implements a subset of all FUSE[1] calls (commands), enough to manipulate files and directories, as described next. Each command takes a set of parameters as input, including a complete file path name, and returns a sequence of bytes or possibly an error code. For simplicity, NetFS does not support soft and hard links.

Some file system calls change the structure of the file system tree (i.e., what files and directories each directory contains). Besides, each file descriptor seen by a client when opening a file is mapped to a local file descriptor at each NetFS server. Such mapping is done with a hash table accessed by all threads, which must then synchronize. Therefore, the following NetFS calls depend on all calls: `create`, `mknod`, `mkdir`, `unlink`, `rmdir`, `open`, `utimens`, `release`, `opendir`, `releasedir`. Calls to `access`, `lstat`, `read`, `write` and `readdir` depend on all calls mentioned above and on each other if they use the same file path.

VI. Implementation

In this section, we describe the implementation and configuration of P-SMR and the other approaches assessed in the evaluation.

A. Atomic multicast

The multicast library implements the abstraction of groups by composing multiple parallel instances of Paxos; each multicast group is mapped to one or more Paxos instances. A message can be addressed to a single group only, not to multiple groups. In our P-SMR prototype, each thread t_i belongs to two groups: one group, g_i, to which no other thread in the server belongs, and one group g_{all}, to which every thread in each server belongs. Threads deliver messages from multiple streams and use a deterministic merge mechanism to ensure ordered delivery [9]. This is enough to implement both C-G functions presented in Section IV-C. Commands multicast to a group are batched by the group's coordinator (i.e., the coordinator in the corresponding Paxos instance) and order is established on batches of commands. Each batch has a maximum size of 8 Kbytes. The system was configured so that each Paxos instance uses 3 acceptors and can tolerate the failure of one acceptor.

B. Key-value store

The key-value store implements a B$^+$-tree where each entry has an 8-byte integer key, used as the tree index, and an 8-byte value. The servers implement all commands described in Section V-A. In order to generate enough load to reach maximum performance, each client maintains a window of outstanding requests that can contain up to 50 commands. The tree is initialized with 10 million keys on each replica and unless specified otherwise, clients select the keys uniformly.

In addition to P-SMR, we implemented a semi-parallel state-machine replication approach (sP-SMR), traditional state-machine replication (SMR), and a non-replicated architecture with a single multi-threaded server directly connected to the clients (no-rep). In no-rep and sP-SMR a scheduler at the server is responsible for scheduling incoming commands for execution at worker threads. We also compare the approaches above to Berkeley DB version 5.3 (BDB), deployed as a client-server architecture. We configured BDB to use the in-memory B-tree access method with transactions disabled and multithreading and locking enabled. Differently from P-SMR, sP-SMR and no-rep, BDB uses locks to synchronize the concurrent execution of commands. As a result, there is no scheduler interposed between clients and server threads: each server thread receives requests through a separate socket, executes them, and responds to clients. Except for the no-rep and BDB experiments, in which there is only one replica, the key-value store is fully replicated on two servers.

C. Networked File System

After NetFS is mounted at a client node, client calls are intercepted by FUSE and redirected to a local file system proxy, which multicasts them as requests to remote servers. This design differs from the architecture presented in Section III in which each client has its own proxy. In NetFS, all clients at a node share the same client proxy. Since file system calls are intercepted by FUSE, client applications do not need to be linked with the client proxy to use NetFS. At the server, incoming requests are received by the server proxy and executed against a local in-memory file system.

In P-SMR we created eight path ranges, each one assigned to a separate thread at the server, which corresponds to the number of cores available in each server node. The file system proxy at a client uses atomic multicast to submit requests to servers, according to the command and file path. Nine multicast groups are used, eight of them for per-path requests, and one for serialized requests. In addition to P-SMR, we also implemented SMR and sP-SMR. sP-SMR uses eight worker threads and also relies on atomic multicast to order commands. A single scheduler thread delivers all requests and, if they are independent, enqueues them for execution by one of the

[1] http://fuse.sourceforge.net/

workers. In the case of a request requiring sequential execution, the scheduler waits for the worker threads to finish their ongoing work and then assigns the request to one worker thread. In all cases, a request is compressed by the client and uncompressed by the worker thread that executes the request, which after executing the command compresses the response and sends it back to the client. All implementations use lz4 compression algorithm.[2]

VII. Evaluation

In the following, we describe the experimental setup (Section VII-A) and detail our findings (Sections VII-B–VII-G).

A. Experimental setup

We ran all the tests on a cluster with two types of nodes: (a) HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory, and (b) Dell SC1435 nodes equipped with two dual-core AMD Opteron processors running at 2.0 GHz and 4 GB of main memory. The HP nodes are connected to an HP ProCurve Switch 2910al-48G gigabit network switch, and the Dell nodes are connected to an HP ProCurve 2900-48G gigabit network switch. Each node is equipped with two network interfaces. The switches are interconnected via a 20 Gbps link. The nodes ran CentOS Linux 6.2 64-bit with kernel 2.6.32. Clients were deployed on the Dell nodes; Paxos's acceptors and servers were deployed on the HP nodes.

B. Performance of independent commands

Benchmark setup: In this experiment we evaluate the key-value store with a workload composed of read commands only. The values we report correspond to the peak throughput of each technique and are obtained with 8 threads for P-SMR, 2 threads for sP-SMR and no-rep, 1 thread for SMR, and 6 threads for BDB. In the case of norep and sP-SMR the number of threads excludes the scheduler.

Results: The throughput of P-SMR is about 3.15 and 2.75 times higher than SMR and sP-SMR, respectively (Figure 3). The scheduler in sP-SMR and no-rep becomes CPU-bound and caps performance. The throughput of SMR is limited by what a single thread can achieve, whereas no-rep is multithreaded and achieves higher throughput. The throughput of no-rep is slightly higher than sP-SMR as no-rep does not rely on atomic multicast. BDB has the lowest throughput due to high overhead with locking, reflected in the CPU usage. Latency of P-SMR is the highest at peak throughput. Although not shown in the figure, under similar throughput P-SMR has latency comparable to the other techniques. no-rep's latency is slightly higher than SMR due to the overhead of the scheduler. Latency of sP-SMR is affected by both the overhead of ordering and scheduling and is higher than the latency of no-rep and SMR.

C. Performance of dependent commands

Benchmark setup: In this experiment we determine the maximum throughput of the key-value store service when commands are inserts and deletes. The values are obtained with 4 threads for BDB and with 1 thread for all the other techniques. In case of norep and sP-SMR the number of threads excludes the scheduler. These are the configurations that correspond to the peak throughput of each technique; for the performance of dependent commands under different number of threads we refer the reader to Section VII-D.

Results: SMR is not subject to synchronization overhead which allows it to reach the highest throughput (Figure 4). Moreover, throughput in SMR remains constant at about 842K cps, both with independent and dependent commands; in BDB the throughput decreases from 140K cps to 105 K cps. P-SMR's latency is higher than SMR's and sP-SMR's. The long tail in the CDF graphs suggest that P-SMR's latency is subject to more variation than SMR's and sP-SMR's.

D. Scalability

Benchmark setup: We measure the maximum throughput of the key-value store service while the number of threads changes from one to eight when commands are independent and then dependent. In sP-SMR, the number of threads reflects the worker threads excluding the scheduler.

Results: With independent commands only, the throughput of all the techniques, except for BDB, compare equally with one thread (Figure 5). As threads are added, the throughput of all the techniques, except for P-SMR, decreases. For sP-SMR and no-rep this happens due to scheduling overhead at the scheduler. P-SMR has better scalability than the other techniques (see bottom left graph). With dependent-only commands, in all the approaches, except for BDB, throughput decreases with the number of worker threads due to the overhead of synchronization. The throughput of BDB increases up to 4 threads and then it decreases due to locking overhead.

E. Performance of mixed workloads

Benchmark setup: We measure the maximum throughput of the key-value store service with workloads composed of inserts, deletes, and reads. The x-axis shows the percentage of dependent commands (inserts and deletes) with respect to all the commands in the workload. P-SMR uses 8 workers in this experiment. We compare the performance of P-SMR to SMR, the only approach that is not subject to synchronization overhead and therefore has the highest performance under dependent commands. The average latency corresponds to the maximum throughput.

Results: SMR's throughput remains constant with the workload mix (Figure 6). This is expected since most of the cost to execute read, insert and delete operations is related to traversing the tree (statistics gathering starts after the tree is initialized; thus, few inserts and deletes involve changes in multiple levels of the tree). P-SMR's throughput is above SMR's up to about 10% of dependent commands. The reduction in performance is due to synchronization overhead. P-SMR's latency decreases as the percentage of dependent commands increases. The decrease in latency corresponds to a reduction in throughput.

F. Performance of skewed workloads

Benchmark setup: The workload is composed of 50% updates and 50% reads against the key-value store. We evaluate

[2] http://code.google.com/p/lz4/

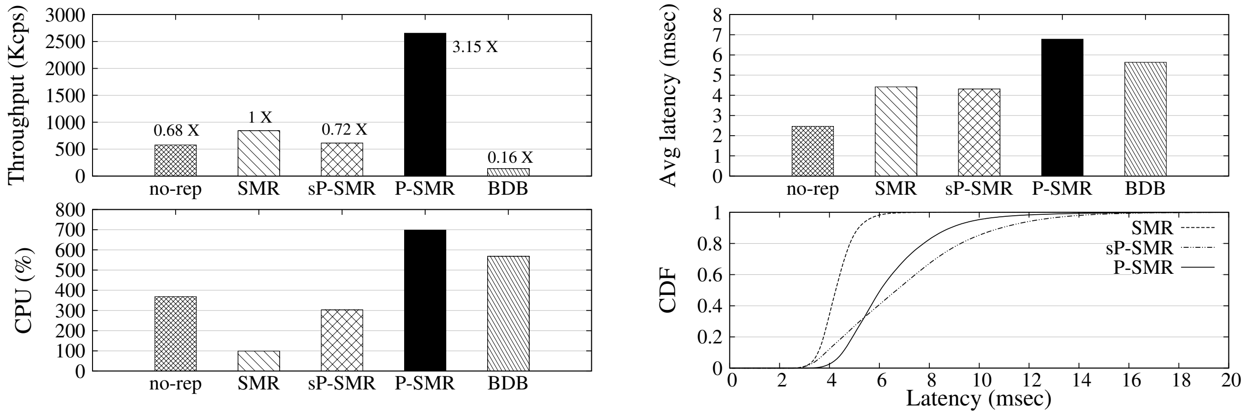

Fig. 3. Performance of independent commands; throughput in Kilo commands executed per second (Kcps) (top-left); CPU usage (bottom-left); average latency in milli seconds (top-right); CDF of latency (bottom-right).

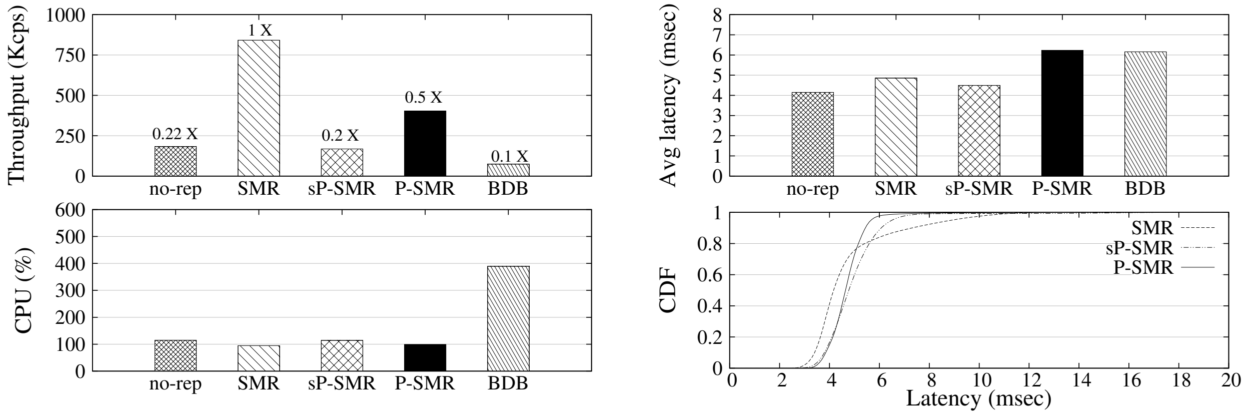

Fig. 4. Performance of dependent commands; throughput in Kilo commands executed per second (Kcps) (top-left); CPU usage (bottom-left); average latency in milliseconds (top-right); CDF of latency (bottom-right).

the scalability of each approach with a uniform key selection distribution and a Zipfian distribution (with exponent value of one). In skewed distributions, communication is expected to be unevenly balanced across multicast groups. In sP-SMR, the number of threads reflects the number of worker threads excluding the scheduler. Besides absolute values for the maximum throughput we also show the normalized throughput of an individual thread. Perfect scalability means that the throughput of each thread remains constant as worker threads are added.

Results: With a uniform selection of keys, commands are evenly distributed across groups and P-SMR's throughput increases up to the capacity of each available core (Figure 7). With a Zipfian distribution, however, P-SMR's throughput is bounded by the most-loaded multicast group (point with 8 threads). sP-SMR is not bounded by a single multicast group as is P-SMR, but by the load the scheduler can handle until it becomes CPU-bound. Increasing the number of worker threads after two threads has a negative impact on sP-SMR's performance since the scheduler spends more time synchronizing with worker threads. Also notice that with 1 and 2 threads the throughput of sP-SMR with a uniform workload is lower than its throughput with a Zipfian distribution. In the Zipfian distribution some keys are accessed more often than the others, and thus, there are higher chances that these keys are cached at the processor. According to the normalized per thread throughput, P-SMR scales better with the number of cores than sP-SMR under both uniform and Zipfian distributions.

G. NetFS performance

Benchmark setup: We have performed two separate experiments, one with read commands only and one with write commands only. All calls to read and write the same file are dependent, while reading and writing different files can be done in parallel. A read request has a small input parameter (i.e., the data to be read) and a large response (i.e., bytes read). A write request contains the buffer to be written as input and a small response. Each request reads (or writes) 1024 bytes from (to) a file.

Results: SMR reaches peak throughput of approximately 100 Kcps (reads) and 110 Kcps (writes), whereas sP-SMR caps throughput at approximately 116 Kcps for both reads and writes, an improvement of about 1.2x and 1.1x, respectively (Figure 8). The small improvement is explained by the use of CPU, where the scheduler becomes a bottleneck before fully using the remaining cores. In P-SMR, read and write commands reach peak throughput of 309 Kcps and 327 Kcps, respectively, an improvement of 3.1x and 3x. Both P-SMR and

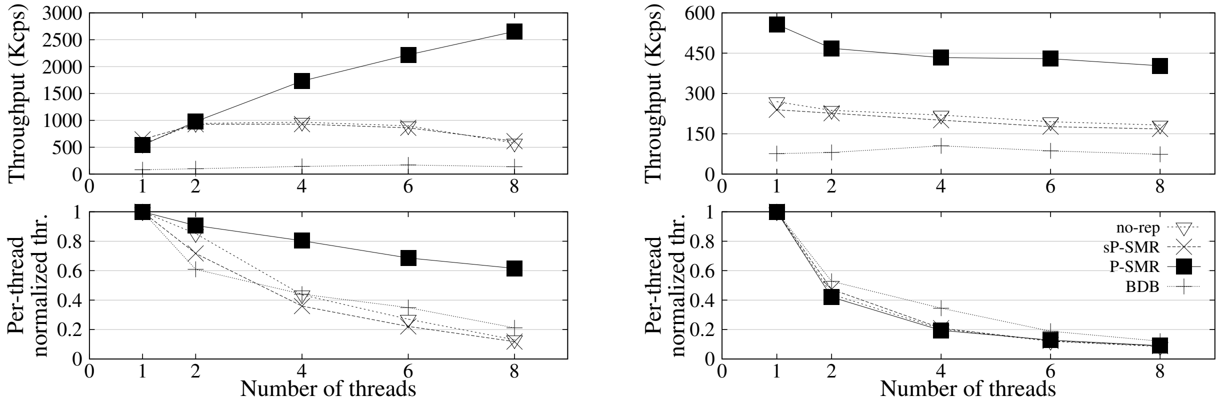

Fig. 5. The effect of the number of threads on the performance independent commands (left) and dependent commands (right); maximum throughput in Kilo commands executed per second (Kcps) (top graphs); normalized per-thread throughput (bottom graphs).

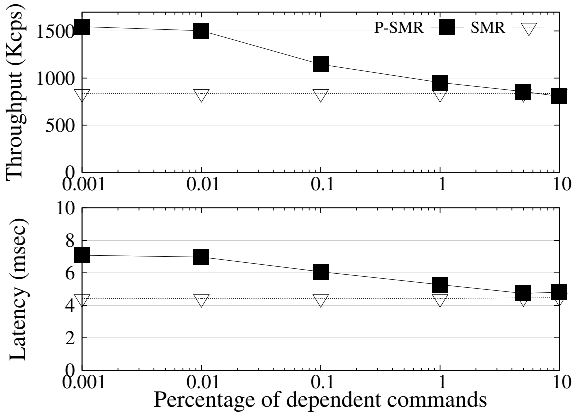

Fig. 6. Performance of mixed workloads (both independent and dependent commands); throughput measured in Kilo commands executed per second (Kcps) (top); the average latency measured in milliseconds (bottom); x-axis is in log scale.

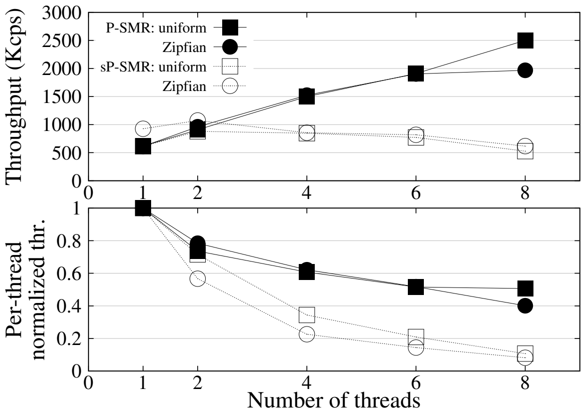

Fig. 7. The effect of the number of threads on the performance of a skewed workload; maximum throughput in Kilo commands executed per second (Kcps) (top); normalized per-thread throughput (bottom).

sP-SMR outperform SMR. Servers had to decompress requests and compress replies. As compression with lz4 takes longer than decompression, read requests took longer to execute than write requests. This is the reason for the latency difference between reads and writes.

VIII. Related work

General-purpose approaches. Allowing multiple threads to execute commands concurrently may result in state and output inconsistencies if dependent commands are scheduled differently in two or more replicas. In [10], [11], [12], [13] the authors propose different approaches to enforcing deterministic multithreaded execution of commands. These solutions impose performance overheads and may require re-development of the service using new abstractions. Another solution is to allow one of the multithreaded replicas to execute commands non-deterministically and log the execution path, which will be later replayed by the rest of the replicas. Logging and replaying have been mainly developed for debugging and security rather than fault tolerance [14], [15], [16], [17], [18], [19], [20]. These approaches typically have high overhead due to logging and may suffer from inaccurate replay, leading to differences among original and secondary copies.

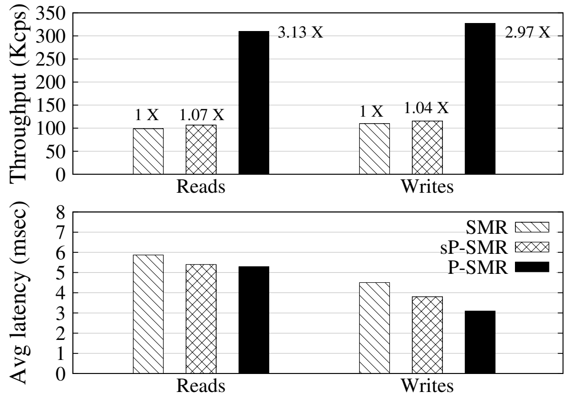

Fig. 8. Performance of read and write commands in NetFS; maximum throughput in Kilo commands executed per second (Kcps) (top); average latency in milli seconds (bottom).

State-machine replication-specific approaches. Having replicas execute commands sequentially by a single thread does not imply that the whole replica's logic must be single-threaded. In [21], the authors propose a staged architecture to exploit the processing power of multi-core servers. A replica is organized as a collection of modules connected through shared message queues. Although staging improves the throughput of state-machine replication, there is always only one thread sequentially executing the commands. In [4], one thread at each replica delivers all commands in order and schedules them for execution against worker threads (i.e., sP-SMR model). The scheduling of commands is deterministic and guarantees that only independent commands are executed concurrently; dependent commands are serialized and executed according to the order established upon delivery. Eve [3] is another instance in the sP-SMR category in which replicas agree on the internal state and output after the execution of each command rather than on their execution path. Eve employes an *execute-verify model* by relying on speculative execution. On each replica, multiple threads can execute commands concurrently. After commands are executed, replicas verify whether they resulted in the same state changes and output. In case of inconsistencies replicas have to rollback the commands and sequentially re-execute them in a unique agreed-upon order. Since replicas verify whether commands resulted in the same state changes and output, Eve can tolerate mistakes when determining command dependencies. Although P-SMR also exploits service semantics to introduce multithreaded execution in state-machine replication, it does not rely on a single thread to deliver and schedule commands [4] and to verify and possibly rollback the execution of commands [3].

Using semantics to improve performance. Other works have proposed the use of application semantics to improve the performance of state-machine replication (e.g., [22], [23], [24]). These are based on the assumption that if two commands commute (e.g., incrementing a counter), then different replicas can execute them in different order and still reach the same final state. These works aim at reducing the delay to deliver a command by avoiding an expensive ordering protocol when possible.

IX. Conclusions

State-machine replication is a fundamental approach to designing highly available services. Hence, it comes as no surprise that a number of approaches have been proposed to allow multithreaded state-machine replication. Some of these approaches take advantage of application semantics to execute independent commands concurrently and serialize dependent commands. Parallel State-Machine Replication also uses application semantics, but differently from previous proposals, it does not rely on a single component to deliver and execute commands. We assessed P-SMR experimentally under several conditions and found that it outperforms classical state-machine replication by a factor of more than 3 and other approaches by a factor of more than 2.

Acknowledgements

We wish to thank Nicolas Schiper, Robbert van Renesse, and the anonymous reviewers for their help and suggestions to improve the paper. This work was supported in part by the Zeno Karl Schindler Foundation and the Swiss National Science Foundation under grant number 146404.

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. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, “Eve: Execute-verify replication for multi-core servers,” in *OSDI*, 2012.

[4] R. Kotla and M. Dahlin, “High throughput byzantine fault tolerance,” in *DSN*, 2004.

[5] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[6] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.

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

[8] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, 1984.

[9] P. J. Marandi, M. Primi, and F. Pedone, “Multi-Ring Paxos,” in *DSN*, 2012.

[10] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, “Efficient system-enforced deterministic parallelism,” in *OSDI*, 2010.

[11] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, “Deterministic process groups in dos,” in *OSDI*, 2010.

[12] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, “DMP: deterministic shared memory multiprocessing,” in *ASPLOS*, 2009.

[13] A. Thomson and D. J. Abadi, “The case for determinism in database systems,” *Proc. VLDB Endow.*, vol. 3, pp. 70–80, Sept. 2010.

[14] G. Altekar and I. Stoica, “ODR: output-deterministic replay for multi-core debugging,” in *SOSP*, 2009.

[15] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, “Execution replay of multiprocessor virtual machines,” in *VEE*, 2008.

[16] P. Montesinos, L. Ceze, and J. Torrellas, “Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently,” in *ISCA*, 2008.

[17] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, “PRES: probabilistic replay with execution sketching on multiprocessors,” in *SOSP*, 2009.

[18] M. Ronsse and K. De Bosschere, “Recplay: a fully integrated practical record/replay system,” *ACM Trans. Comput. Syst.*, vol. 17, pp. 133–152, May 1999.

[19] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, “DoublePlay: parallelizing sequential logging and replay,” *SIGPLAN Not.*, vol. 47, pp. 15–26, Mar. 2011.

[20] M. Xu, R. Bodik, and M. D. Hill, “A “flight data recorder" for enabling full-system multiprocessor deterministic replay,” in *ISCA*, 2003.

[21] N. Santos and A. Schiper, “Achieving high-throughput state machine replication in multi-core systems,” tech. rep., EPFL, 2011.

[22] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, “Thrifty generic broadcast,” in *DISC*, 2000.

[23] F. Pedone and A. Schiper, “Generic broadcast,” in *DISC*, 1999.

[24] L. Lamport, “Generalized consensus and paxos,” Tech. Rep. MSR-TR-2005-33, Microsoft Research (MSR), Mar. 2005.