Parallel State Machine Replication from Generalized Consensus

Tarcisio Ceolin Junior*†, Fernando Dotti* and Fernando Pedone‡
*Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul - Brazil
†Universidade Federal de Santa Maria - Brazil
‡Università della Svizzera italiana - Switzerland

***Abstract*—State machine replication (SMR) is a established approach to building fault-tolerant services. In search for high SMR throughput, approaches that exploit semantic information in the ordering and execution of commands have emerged. Generalized consensus and parallel state machine replication are two representative examples, respectively. Although both approaches have been proved effective in isolation, no study in the literature has considered their integration. In this paper, we investigate the integration of generalized consensus and parallel SMR. We derive algorithms to parallelize the execution of commands based on the ordering of commands provided by consensus. As a prototype, we extended Egalitarian Paxos and conducted many experiments varying conflict rates, command computational costs, and number of cores at replicas. Compared to Egalitarian Paxos, the integrated approach (a) results in important throughput gains, as command independency and computational cost increase, and (b) converges to the same performance with high conflict rates or reduced number of cores.**

***Index Terms*—State Machine Replication, Generalized Consensus, Distributed Algorithms**

I. Introduction

State machine replication (SMR) is a conceptually simple, yet effective approach to rendering systems fault tolerant. The basic idea is that server replicas execute client requests deterministically and sequentially, in the same order [1], [2]. Consequently, replicas transition through the same sequence of states and produce the same output. State machine replication enables application programmers to focus on the inherent complexity of the application, while avoiding the difficulty of handling replica failures [3]. Not surprisingly, the approach has been successfully used in many contexts (e.g., [4]–[6]).

Different strategies have been proposed to enhance SMR's throughput. A large category of solutions introduce concurrency in the ordering and execution of commands by taking advantage of application semantics information. More precisely, two commands conflict if they access common state and at least one of them updates the state; otherwise the commands are independent. When commands are independent, their order is irrelevant and expedite decisions can be taken to order and execute commands.

Leaderless consensus protocols exploit application semantics to optimize command ordering. Since conflicts may arise if different replicas propose commands concurrently, protocols such as Generalized Paxos [7], Generic Broadcast [8] and Egalitarian Paxos [9] use application semantics information to decide on the order of conflicting commands—we call these generalized consensus protocols.

Regarding command execution, a number of techniques have been proposed to overcome the limitation of sequential execution and allow parallel execution of non-conflicting commands. This is specially important considering modern multi-processor architectures. The main challenge is to ensure the same deterministic replica behavior out of parallel command execution. In [10], [11] replicas execute optimistically and then agree on the results, maybe having to re-execute some commands. In [12] one replica executes, logs dependencies among commands, and then uses consensus to replicate its trace of dependencies such that other replicas follow the same trace. In several approaches [13]–[17], consensus totally orders commands and then replicas identify conflicts to introduce concurrency in the execution of independent commands.

Although both the ordering and execution of commands exploit common aspects, namely, application semantics, there are no studies considering the integration of the approaches. In this paper, we investigate if and how far the partial order resulting from generalized consensus can be beneficial for parallel command execution at replicas. Our contributions are: (i) we devise algorithms for a parallel command execution that use conflict information from generalized consensus; (ii) we implement a prototype using Egalitarian Paxos (ePaxos) as consensus protocol; (iii) we assess performance implications under several parameters, such as conflict rates, command execution costs and number of available cores.

The experimental evaluation revealed that the integrated approach results in important performance gains, as command independency and computational cost increase. The integrated approach and ePaxos have similar performance under high conflict rates or reduced number of cores. The results corroborate our argument for integration, since no significant overhead is introduced in conflict detection by parallel execution.

The paper is organized as follows. Section II presents main assumptions and background on consensus. Section III details our proposal of using generalized consensus dependency information to the parallel execution of SMR commands. Section IV details our system prototype, the evaluation results. Sections V and VI survey related work and conclude the paper.

II. Background

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

To implement a total order, we assume servers use a consensus protocol. Consensus is defined by primitives $propose(v)$ and $decide(v)$, the first primitive is used to propose values and the second primitive to decide on a proposed value., Consensus ensures the properties of Termination, Validity, Integrity, and Agreement [18]. Since we are interested in a sequence of decisions, we define primitive $decide(i, v)$ where i is the consensus instance number, a natural number that associates an increasing order, without gaps, to decisions. Each decision has a unique number across replicas and enjoys the consensus properties above. Consensus requires additional synchronous assumptions but our protocols do not explicitly need these assumptions [3].

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

A. State Machine Replication

State Machine Replication (SMR) 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 the response of a command are a function of the state variables the command reads and the command itself.

SMR requires replicas to execute commands in the same order. Therefore, before commands are executed by the replicas, the execution order must be agreed. This is achieved using consensus. Whenever a command is issued to an SMR replica, it is proposed in consensus. SMR is implemented having replicas execute commands according to the consensus order, as shown in Algorithm 1.

State machine replication provides strong consistency. Clients work with the illusion of a non-replicated service, that is, replication is transparent. Differently from a non-replicated service, clients remain oblivious to failures, as the service is operational despite the failure of some of its replicas (i.e., up to f faulty replicas).

Algorithm 1 SMR Replica Execution

```
1: constants and data structures
2:     $i : 0$                                   {next expected decision}
3: Replica works as follows:
4: upon decide($i, c$)                           {consensus sequence}
5:     $executeAndReplyToClient(c)$
6:     $i \leftarrow i + 1$
```

B. Generalized Consensus

The Paxos [20] consensus protocol is widely used in different SMR deployments. Paxos and derived protocols are based on a distinguished process or leader to coordinate consensus. The use of a single process for this task may limit throughput and therefore alternatives have emerged to allow more than one process to coordinate consensus. Lamport's Fast Paxos [21] introduces the possibility that a coordinator may delegate the right to other proposers to directly address acceptors, and acceptors to accept proposals from other proposers. This both reduces one message delay and has the potential to increase throughput by avoiding a single point through which all proposals should flow. The drawback of this design is the possibility of proposal collisions.

Generalized Paxos [7] extends Fast Paxos to cope with this aspect. It generalizes the state-machine approach to allow agreement on a partially ordered set of consensus instances. Lamport proposes generalized consensus to be applied in different situations by defining suitable structures, called c-structs, for each case. C-structs define if consensus instances conflict or not. If not, their relative order is not important and in case of collision they can be delivered in two message delays, in any order, at replicas.

The case of interest here is consensus for command histories. A command history is a partial order of commands that relates conflicting commands only. In the following, we define the notion of conflict.

Definition 1 (Commands, read and write sets, conflict)**.** Let C be the set of commands available in a service (i.e., all the commands that a client can issue). A command can be any deterministic computation involving objects that are part of the application state. We denote the sets of application objects that replicas read and write when executing a command c as c's *readset* and *writeset*, or $RS(c)$ and $WS(c)$, respectively. The conflict relation $\#_C \subseteq C \times C$ among commands is:

$$(c_i, c_j) \in \#_C \text{ iff } \begin{pmatrix} RS(c_i) \cap WS(c_j) \neq \emptyset \vee \\ WS(c_i) \cap RS(c_j) \neq \emptyset \vee \\ WS(c_i) \cap WS(c_j) \neq \emptyset \end{pmatrix}$$

Commands c_i and c_j *conflict* (*depend* or *interfere*) if $(c_i, c_j) \in \#_C$. Pairs of commands not in $\#_C$ are *non-conflicting* (*independent* or *non-interferring*).

Generic broadcast [8] also delivers a partial order of commands, being equivalent to Lamport's generalized consensus for command histories [7]. Likewise, command semantics is modeled with a conflict relation and a broadcast algorithm that works with any conflict relation.

Generalized Paxos (for command histories) and Generic Broadcast deliver sequences of commands, which are compatible with the partial order built from their dependencies. SMR replicas that build on these protocols would behave as in Algorithm 1. Replicas may observe different but equivalent orders that commute non-conflicting commands. Executing commands according to the observed order will not compromise linearizability.

C. Egalitarian Paxos

Egalitarian Paxos (ePaxos) [9] is another protocol that uses command semantics to avoid collisions during consensus. Different replicas concurrently coordinate consensus for different commands. If commands do not conflict, then they are delivered independently. ePaxos however is organized differently at the service interface, allowing one to access command dependency information gathered during consensus.

a) Overview of EPaxos consensus phase: A command in ePaxos is a consensus instance. Clients send commands to replicas, which then propose instances concurrently. A replica takes care of coordinating consensus (i.e., it acts as command leader) for the instances it proposes. During coordination, the presence or absence of conflicts with pending instances in other replicas are identified and accordingly registered in a set of conflicts for each instance. More concretely, this takes place at Phase 1 when a command leader sends an instance to other replicas, which evaluate the proposed instance against their locally registered instances to identify a set of conflicting ones. The command leader gathers the conflict sets from different replicas. If all have the same set, possibly empty (i.a., no interference), replicas have the same view and the fast path succeeds. If not, then the command leader is in charge of synchronizing the complete conflict set across replicas using the slow path.

A replica keeps a set of instances that evolve through consensus, set I in Algorithm 2, line 2. An instance information is composed of: (i) the command; (ii) the set of other instances it conflicts with; (iii) a sequence number to be used in case of solving cycles; (iv) its state, which can be, in order: PreAccepted, Accepted, Committed, and Executed (see lines 3 to 6). An instance is in state Committed when its information is complete and replicated across replicas. From the perspective of consensus, a committed instance is delivered.

b) Overview of EPaxos execution phase: Committed instances are registered in the instances set I and build a partial order of conflicting commands. The ePaxos execution phase builds a total order compatible with this partial order and sequentially executes according to this order. Algorithm 2 summarizes the execution phase. The execution algorithm periodically visits the instances set to detect a committed (see line 27). Whenever an instance is *Committed*, the dependency graph of that instance is recursively built (lines 11 to 17). All instances in the resulting graph are committed (line 19). Figure 1 (a) depicts a possible dependency graph for node 1, where instances are depicted as circles with sequence numbers, and edges are dependencies. If instances were independent, their

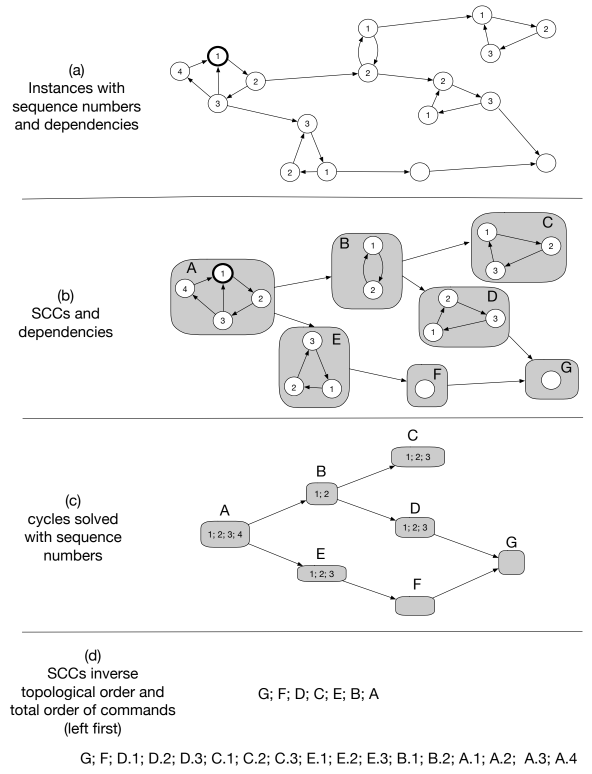

(a) Instances with sequence numbers and dependencies
(b) SCCs and dependencies
A
B
C
D
E
F
G
(c) cycles solved with sequence numbers
1; 2; 3; 4
1; 2
1; 2; 3
1; 2; 3
1; 2; 3
(d) SCCs inverse topological order and total order of commands (left first)
G; F; D; C; E; B; A
G; F; D.1; D.2; D.3; C.1; C.2; C.3; E.1; E.2; E.3; B.1; B.2; A.1; A.2; A.3; A.4

Fig. 1: Instances and strongly connected components (SCCs).

respective dependency graphs would return a single instance each. When instances conflict, they will be connected in the dependency graph. If they conflict and collide during consensus, then strongly connected components are configured.

Figure 1 (b) shows strongly connected components (SCCs) in gray boxes. The SCCs in the dependency graph are identified, resulting in a list of SCCs (see line 18), which can be computed using Tarjan's algorithm [22]. We assume SCCs are returned in reverse topological order (i.e., deepest elements first), as mentioned in lines 18 and 21. Figure 1 (d) illustrates an inverse topological order. Within each SCC, instances are ordered according to the sequence number provided during ePaxos consensus (e.g., see Figure 1 (c)), as in line 19, and then executed in that order, if not yet executed (lines 23 to 25). With this, the same total execution order is followed at each replica.

III. Parallel SMR from Generalized Consensus

Generalized consensus protocols avoid consensus collisions using semantic conflict information. This same information can be used to exploit intra-replica parallelism for independent commands and thus enhance the replica execution throughput in workloads dominated by independent commands. In this section, we delve into this aspect.

Figure 2 compares classic SMR and generalized consensus-based approaches, cases (a) and (b), to our proposal, case (c). We leverage generalized consensus to reduce latency and improve throughput, by exploiting consensus conflict identification to schedule concurrent instances for parallel

Algorithm 2 ePaxos Execution Phase

```
1: data structures
2:   $I : \{c \times deps \times seq \times state \mid$
3:       $c \in C,$                                          {the command}
4:       $deps \in I,$      {other instances from which this one depends}
5:       $seq \in \mathbb{N},$                          {to solve cycles if needed}
6:       $state \in \{PreAccepted, Accepted, Committed, Executed\}$
7:       $\}$
8:   $G : (N, E)|$                                      {dependency graph}
9:       $N \subseteq I,$                          {nodes are consensus instances}
10:      $E \in N \times N$                           {edges are dependencies}
11: procedure G : buildDepGrapg(i,(N,E))
12:   for all $j \in i.deps | j.state \neq Executed$ do    {$i$ depends on $j$}
13:     wait until $j.state = Committed$
14:     $N \leftarrow N \cup \{j\}$
15:     $E \leftarrow E \cup \{(i, j)\}$
16:     $(N, E) \leftarrow buildDepGraph(j, (N, E))$
17:   return $(N, E)$          {a graph where all instances are committed}
18: procedure sccList : findSCCs((N,E))
          {sccList is a list of SCCs in G=(N,E) in reverse topological order}
                          {reverse order is olders first in sccList}
19: procedure instList : sort((N,E))
        {instList is a list with all instances $i \in N$ in i.seq increasing order}
                     {this solves cycles deterministically across replicas}
20: procedure execList(sccList)
21:   for all $scc \in sccList$, in reverse topological order do
22:     $instList \leftarrow sort(scc)$                          {solves cycles}
23:     for all $inst \in instList$, in order do
24:       $execute(inst.c)$
25:       $inst.status \leftarrow Executed$
26: Replica's execution works as follows:
27: upon $i = [r, dep, seq, Committed] \in I$      {committed instance $i$}
28:   $dg \leftarrow buildDepGraph(i, (\{i\}, \emptyset))$    {graph starts with $i$ only}
29:   $sccList \leftarrow findSCCs(dg)$
30:   $execList(sccList)$
```

execution. Parallel approaches to SMR identify independent commands and process them concurrently, thereby enhancing the replicas execution throughput (see §V for a survey on existing approaches). Using conflict information provided by generalized consensus spares the cost of identifying and representing conflicts among pending instances at replicas.

A. Dependency-based instance scheduling

Figure 1 (a) shows committed instances and their dependencies, while Figure 1 (b) depicts their corresponding identified SCCs. Any two SCCs not directly or transitively linked by the directed dependency edges could execute concurrently, such as SCCs C, D and F in Figure 1 (c).

Using ePaxos as consensus protocol, we propose the concurrent execution of committed instances. More precisely, we introduce Algorithm 3, which is derived from Algorithm 2 (see Section II-C0b). Algorithm 3 shows in cyan (or gray) non-modified parts from Algorithm 2 and in black the modifications to concurrently execute SCCs whenever possible.

Like in Algorithm 2, in Algorithm 3 periodically and sequentially instances are checked if committed (line 39). The

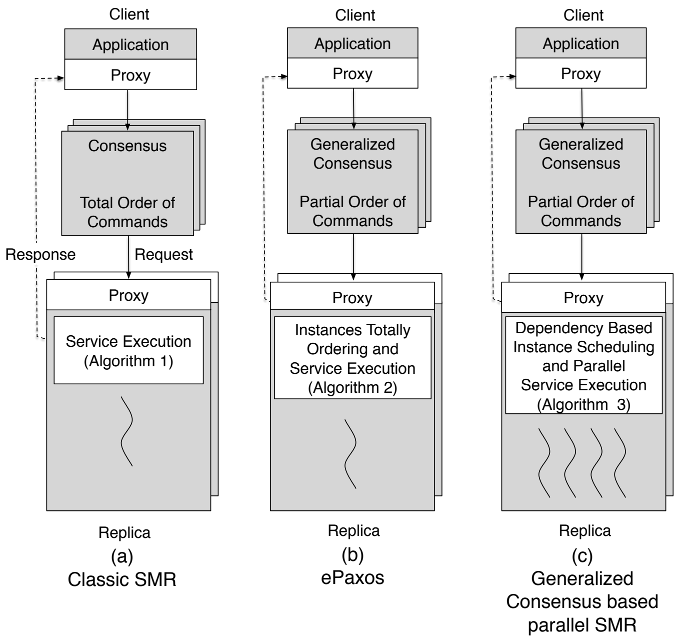

Client
Application
Proxy
Consensus
Total Order of Commands
Response
Request
Proxy
Service Execution (Algorithm 1)
Replica
(a) Classic SMR
Client
Application
Proxy
Generalized Consensus
Partial Order of Commands
Proxy
Instances Totally Ordering and Service Execution (Algorithm 2)
Replica
(b) ePaxos
Client
Application
Proxy
Generalized Consensus
Partial Order of Commands
Proxy
Dependency Based Instance Scheduling and Parallel Service Execution (Algorithm 3)
Replica
(c) Generalized Consensus based parallel SMR

Fig. 2: SMR and Parallel SMR architectures.

dependency graph starting at an instance is built (lines 16 to 23), and SCCs are identified (line 41). Algorithm 3 thus preserves the sequential visit to the instances set to build the dependency graph.

Differently from Algorithm 2, however, SCCs identified are launched for concurrent execution (line 37). Now we have the sequential visit to find committed instances in parallel with execution of commands. To avoid an instance to be taken twice for execution, it is marked as $Executing$ when included in a dependency graph (line 17). $Executing$ is a new instance state defined for this purpose (line 7). This step was not needed in the original algorithm since instances would switch directly from $Committed$ to $Executed$, assuredly executing exactly once because SCC execution and the visit to instance sets was sequential.

To limit the population of dynamically created worker threads to a maximum number of threads (constant nWT in line 13), we use a counting semaphore (line 14). The semaphore is initialized with the maximum number of worker threads and is decremented whenever a thread is created (line 36) and incremented when the worker thread finishes (line 33).

Like in Algorithm 2, $findSCCs$ is used with the dependency graph (line 41) to generate a list of SCCs. Each SCC in this list is then launched for concurrent execution (line 37). While SCCs can execute concurrently if dependencies allow, the internal instances of each SCC have a total order (line 27). For each instance belonging to an SCC, in the total order, its dependencies with respect to instances belonging to other SCCs have to be resolved (lines 29 and 30), and then the instance can be executed.

During execution of each SCC, dependencies for each instance are enforced in line 29. It states that to execute

Algorithm 3 Parallel ePaxos Execution Phase

```
1:  data structures
2:      $I$ : { $c \times deps \times seq \times state$ |
3:          $c \in C$,
4:          $deps \in I$,
5:          $seq \in \mathbb{N}$,
6:          $state \in$ {PreAccepted, Accepted, Committed,
7:                  Executing          {new added state to instance}
8:                  Executed}
9:      }
10:     $G : (N, E)|$                             {dependency graph}
11:         $N \subseteq I$,                      {nodes are consensus instances}
12:         $E \in N \times N$                    {edges are dependencies}
13:     $nWT$ : maximum number of worker threads
14:     $avWT$ : countingSemaphore($nWT$)
15:                     {credit of worker threads that can be launched}
16: procedure G : buildDepGrapg(i,(N,E))
17:     $i.state \leftarrow Executing$            {take for execution}
18:     for all $j \in i.deps | j.state \notin \{Executing, Executed\}$ do
19:         wait until $j.state = Committed$
20:         $N \leftarrow N \cup \{j\}$
21:         $E \leftarrow E \cup \{(i, j)\}$
22:         $(N, E) \leftarrow buildDepGraph(j, (N, E))$
23:     return $(N, E)$             {all instances are marked $Executing$}
24: procedure sccList : findSCCs((N,E))
                            {sccList is the list of SCCs in G=(N,E)}
25: procedure instList : sort((N,E))
        {instList is a list with all instances $i \in N$ in i.seq increasing order}
                    {this solves cycles deterministically across replicas}
26: procedure concExec(scc) dynamically created thread
27:     $instList \leftarrow sort(scc)$
28:     for all $i \in instList$, in order do    {for each instance in order}
29:         for all $j \in i.deps \setminus instList$ do  {dependencies to other...}
30:             wait $j.state = Executed$       {...scc's should be resolved}
31:         $execute(i.c)$
32:         $i.state \leftarrow Executed$
33:     $avWT.up()$                             {increments or unblocks}
34: procedure concExecList(sccList)
35:     for all $scc \in sccList$ do
36:         $avWT.down()$                       {decrements or blocks if 0}
37:         start concurrent thread to $concExec(scc)$
38: Replica's execution works as follows:
39: upon $i = [r, dep, seq, Committed] \in I$      {committed instance $i$}
40:     $dg \leftarrow buildDepGraph(i, (\{i\}, \emptyset))$   {graph starts with $i$ only}
41:     $sccList \leftarrow findSCCs(dg)$
42:     $concExecList(sccList)$
```

an instance, all instances it depends on from other SCCs (therefore excluding $instList$, which is the nodes of the scc) have to be resolved.

B. Correctness

We now argue that Algorithm 3 ensures that conflicting commands are executed in the same order across replicas. We recall that ePaxos ensures that all replicas have the same instance dependency and sequence number information.

Algorithm 3 preserves the periodic sequential recursive visit to the instances set to build the dependency graph. The recursion stops when: no dependencies are found, or the instance being visited has already been executed or taken for execution. Since instances cannot depend on future instances, the set of dependencies considered while building a dependency graph is finite. Moreover, eventually all instances (that one specific instance depends) either are $Committed$ and can be taken for execution or were executed and are resolved. Therefore, instances are progressively taken for execution whenever ePaxos commits them.

Regarding execution, each SCC can be executed as soon as its dependencies with respect to other SCCs are solved. SCCs have a topological order, i.e., by definition there are no cycles among SCCs. Therefore there is no deadlock. Within an SCC, instances are totally ordered by sequential number, which disambiguates cycles homogeneously across replicas. Then, it suffices to follow the order within the SCC to ensure that for each instance, the dependencies to other SCCs are respected. This enforces the same order among SCCs since all replicas have the same conflict information for each instance.

IV. Experiments

In this section, we discuss topics related to PePaxos, our parallel ePaxos prototype, and present the results of our experimental evaluation.

A. Implementation of PePaxos

The proposed concurrent scheduling algorithm was developed and integrated into Egalitarian Paxos. Our prototype is based on the ePaxos implementation,[1] written in the Go programming language version 1.13.1 and publicly available.[2] Each SCC is launched in a separate goRoutine for concurrent execution, following Algorithm 3, line 37. Lines 29 and 30 are implemented with a busy-wait strategy to check if the specific instances in other SCCs were $Executed$. The counting semaphore is implemented by a channel with the size of the number of worker threads, initially full, where down operations are reads (a read removes one item) and up operations are writes of items from/to the channel.

B. Application

When evaluating the performance of semantic-aware protocols, one important aspect to consider is the rate of non-conflicting commands in the workload. For example, according to [23], in 10-minute traces, Chubby experienced less than 1% of commands that could possibly generate conflicts; [24] reports that fewer than 0.3% of all operations in Google's advertising back-end (F1) may generate conflict; and according to [9], conflict probabilities between 0% and 2% are the most realistic.

Another relevant aspect to consider is the execution cost of a command. The execution cost of a command depends essentially on the nature of the application. Parallel SMR approaches have considered a range of applications, from

[1]https://github.com/efficient/epaxos
[2]https://github.com/tarcisiocjr/pepaxos

social networks [25], to key-value stores [9], [26], and file systems [26].

Instead of considering one specific application, we evaluate PePaxos with a linked list whose command costs and conflict rate can be naturally configurable. Different list sizes allow us to easily configure and evaluate different command execution costs. The list has the following operations:

- $contains(int)$: checks whether an entry (i.e., an integer) is in the list; it returns *true* if entry i is in the list, otherwise it returns *false*;
- $add(int)$ and $remove(int)$: add or remove an entry; they return true, respectively false, if the item is not in the list and false, respectively true, otherwise.

Hereafter, we refer to operations that check whether an entry is in the list and to operations that add/remove an entry in the list as *read* and *write* operations, respectively. In the concurrency model for this application, read commands do not conflict with each other but conflict with write commands, which conflict with all commands. Write operations block the whole list. We use conflict probability as the probability of write operations. The integer parameter used in a read operation is randomly chosen. To keep the execution cost stable and experiments more controllable, we fixed the list population to the desired values (1, 10k, 100k and 1M) during the entire experiment. Otherwise we would have to run the experiments to reach a steady population and present the results according to it. Therefore, for the experiments we start with a populated list and write operations just replace elements, keeping the population.

C. Methodology and environment

All experiments were run in a local-area network (LAN). ePaxos and PePaxos were configured with three replica nodes, to tolerate up to one crash. Between 3 and 3000 clients were distributed uniformly across 10 nodes. Nodes were hosted in separate machines, each machine with four 16-core AMD Opteron 6366HE processors running at 1.8 GHz, 128 GB of RAM, SATA SDD disks, and 1Gbps ethernet card. Client nodes are equipped with a four-core AMD Opteron 2212 processor at 2.0GHz, 4GB of memory and 1Gbps ethernet card. The machines were configured with Ubuntu Linux 18.04 64bits operating system. The RTT between nodes is around 0.1ms.

The state of each replica is kept in main memory. Replicas reply to the client only after executing the command. Our experiments use 16-byte messages. We use batching to increase the throughput, every 5ms (or 1000 commands) each proposer batches all requests in its queue.

In our prototype, we varied the concurrency level, allowing from 1 to 64 parallel goRoutines to execute (parameter nWT, line 13 in Algorithm 3). Experiments with list size 1, $10k$, $100k$ and $1M$ entries were conducted, representing operations with different execution costs. The conflict probability is varied in 0, 1, 2, 25 and 100%, meaning the share of write operations. ePaxos (and consequently PePaxos) serializes two batches of commands if they both contain conflicting commands. In our experiments, batch sizes decrease as execution costs increase; with large execution costs, a batch contains approximately one command. We run a warm-up phase of 30 sec and collect throughput of the system and the latency of each command at the clients for the next 60 sec.

We measured maximum throughput, and latency and throughput for the highest power point. The *highest power* is the point where the ratio of throughput divided by latency is at its maximum. It indicates the inflection point where the system reaches its peak throughput before latencies start to increase due to queueing effects. This point can be regarded as a possible working situation before system saturation.

D. Results

In Figures 3, 4, 5 and 6 we present throughput and latency achieved by PePaxos, respectively, for applications with list population of 1, 10K, 100K and 1M elements. In each figure the results are for configurations in the cartesian product of conflict probabilities and maximum number of threads. Graphs on the left side of the figures (a) depict the maximum throughput. Graphs on the right side of the figures (b) show throughput and latency for the highest power point achieved in each configuration. They do not correspond to values for the same workload. In Figure 7, we show latency and throughput for the same workload for list sizes 10 and 100k.

Very light cost operations

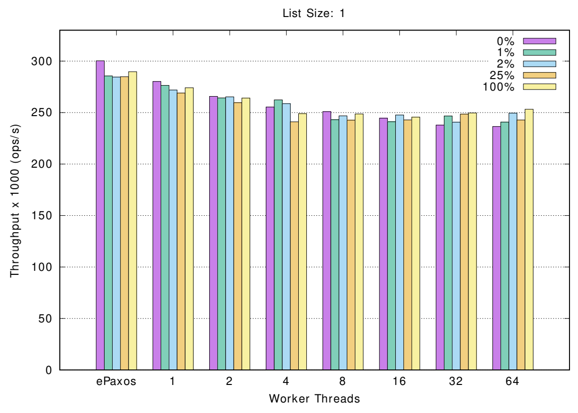
Figure 3 depicts the results for executions with very light cost operations. At this side of the spectrum of command execution times, parallel execution does not make up for the overhead of scheduling multiple threads. In Figure 3 (a), we observe that ePaxos throughput is slightly better than PePaxos with one thread: $\sim$290K ops/sec vs. $\sim$275K ops/sec. We observe in Figure 3 (a) that as we add threads, throughput is gradually impacted. Also, we observe that lower conflict rates do not lead to better throughput.

Recall that the algorithm has a sequential part, to identify SCCs to be executed and then launch them for parallel execution. The observations above let us conclude that the sequential part of the algorithm becomes a bottleneck for these very light command execution times. The overhead of creating threads per command is higher than executing the commands, the sequential execution is faster in this case and therefore conflicts do not affect throughput.

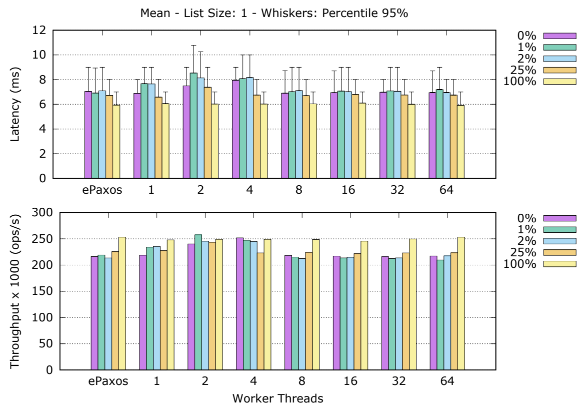
In Figure 3 (b) we observe latency and throughput associated to the highest power point for each of the configurations. All latencies fall in a narrow range, as well as the throughput. This effect is expected: as discussed, since the addition of threads does not help in this case, the behavior approaches the sequential one.

Moderate cost operations

Figure 4 shows results when the application handles a population of 10K elements. Here we observe that it is worth using the technique proposed. From Figure 4 (a) we observe

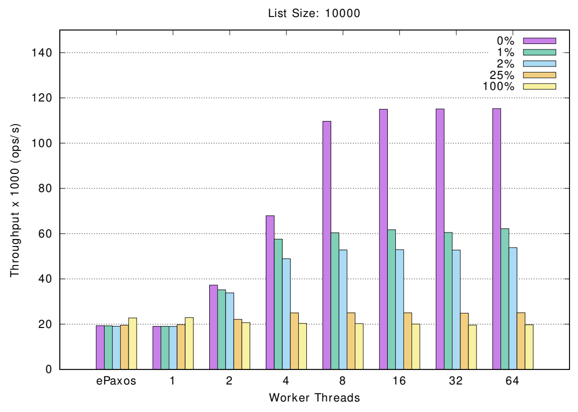


(a) Maximum throughput

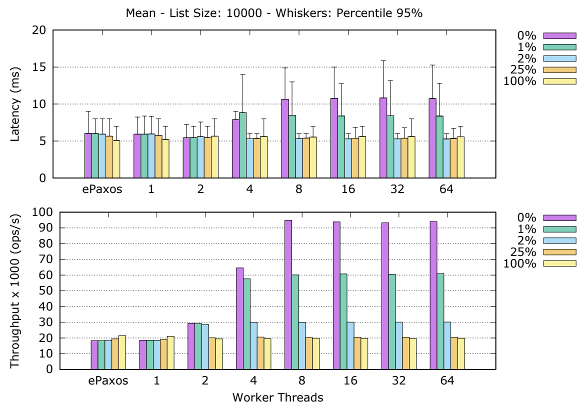


(b) Highest power point: latency and throughput

Fig. 3: Throughput and latency for varying number of threads and conflicts, for very light cost operations (list size 1).



(a) Maximum throughput



(b) Highest power point: latency and throughput

Fig. 4: Throughput and latency for varying number of threads and conflicts, for moderate cost operations (list size 10K).

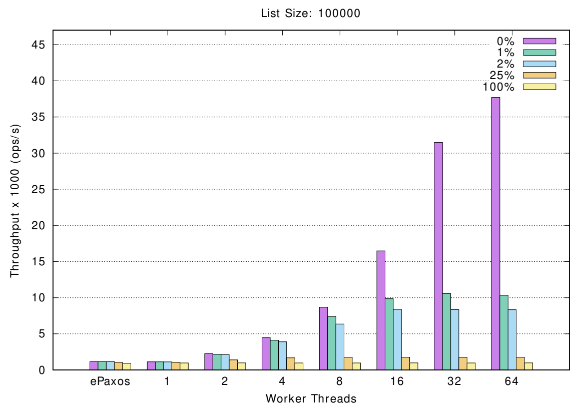
that throughput increases as we add up to 8 worker threads. For 0%, 1%, 2% and 25% conflict, PePaxos with 8 threads performs respectively $\sim 5\times$, $\sim 3\times$, $\sim 2.7\times$ and $\sim 1.2\times$ faster than ePaxos. PePaxos with 1 thread and 100% conflicts performs as well as ePaxos, but slightly loses performance with more threads due to additional overhead with a sequential workload. After 8 threads we observe the same throughput values, indicating that the sequential part of the algorithm prevents throughput from further scaling with the number of threads.

In Figure 4 (b) we have the highest power points for each of the configurations. For 0% conflict we notice an increase in latency as the number of threads increase. This is because for each number of threads the highest power ratio chosen had increased throughput, generated by different workloads.

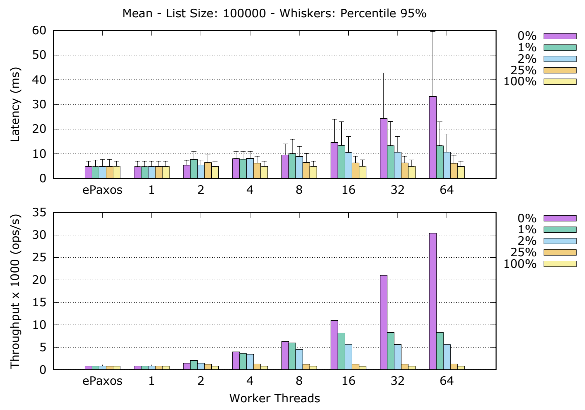
Heavy cost operations

Figure 5 shows results when the application handles a population of 100K elements, i.e. $10\times$ the moderate costs population. Accordingly, in this scenario ePaxos shows a throughput loss from 20K ops/sec to ~2K ops/sec and an increase in latency from ~5 to ~50 ms if compared to the moderate case. From Figure 5 (a), again PePaxos with 1 thread shows throughput and latency results compatible with ePaxos. With up to 64 threads PePaxos scales throughput. For 0%, 1%, 2% and 25% conflict, PePaxos with 64 threads performs respectively $\sim 18\times$, $\sim 9\times$, $\sim 7.2\times$ and $\sim 1.5\times$ faster than ePaxos. With 100% conflicts PePaxos performs as well as ePaxos, for any number of threads.

The highest power points, Figure 5 (b), show that throughput scales with the number of threads also for the points before

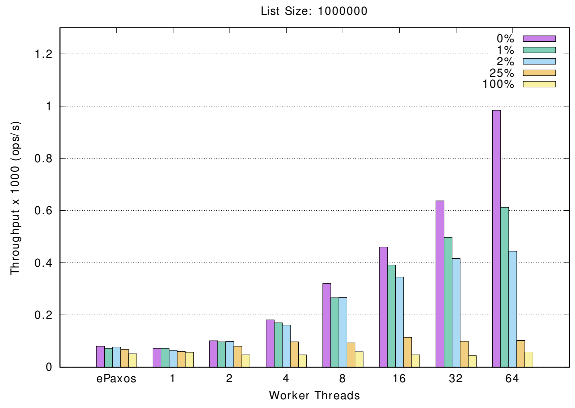


(a) Maximum throughput

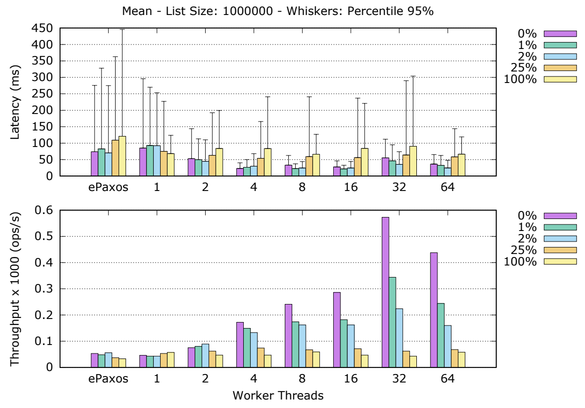


(b) Highest power point: latency and throughput

Fig. 5: Throughput and latency for varying number of threads and conflicts, for heavy cost operations (list size 100K).



(a) Maximum throughput



(b) Highest power point: latency and throughput

Fig. 6: Throughput and latency for varying number of threads and conflicts, for very heavy cost operations (list size 1M).

saturation, in each configuration. For these points, in general latencies follow the throughput, being associated to the population of commands being handled at a replica by the sequential part of the algorithm. In the interval from 16 to 64 threads, for 0% conflicts, where considerable throughput gains are observed, this latency behavior is quite pronounced.

Very heavy cost operations

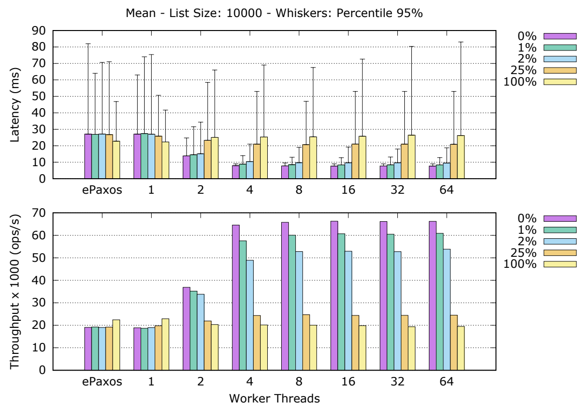
Figure 6 shows results when the application handles a population of 1M elements. In Figure 6 (a) we observe the same throughput behavior as in Figure 5 (a), however in a different interval due to the list size 10× higher which implies in increased command execution times.

Differently from Figure 5 (b), however, in Figure 6 (b) we observe a different behavior of latencies for the highest power points. Here latencies generally decrease as throughput raises. This reveals that the augmented command execution times play a more important role compared to the sequential part of the algorithm.

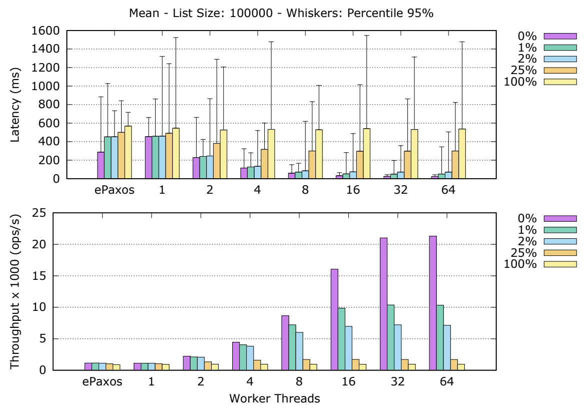
Results for the same workload

In the experiments reported before, workloads varied for each bar since we selected either the maximum throughput or the highest power point achieved for each configuration. Now we fix the workload and observe the behavior with different configurations. The results are depicted in Figure 7.

As we add threads to the experiment, for the same conflict rate we have increasing throughput and decreasing latency. For the same number of threads configuration, and for increasing conflict rates, we have decreasing throughput and increasing latency. These observations generalize to different list sizes, but here we show the 10K and 100K configurations, with 500



(a) Latency and throughput for list size 10K



(b) Latency and throughput for list size 100K

Fig. 7: Throughput and latency with the same workload, list sizes 10K and 100K.

clients in each case. This behavior is followed, with variations, for different workloads (number of clients).

V. Related Work

As it has been early observed [2], independent commands can be executed concurrently in SMR. Previous works have shown that many workloads are dominated by independent commands, which justifies strategies for parallel command execution (e.g., [25]–[30]). We organize the main approaches to parallel SMR into four classes, surveyed next.

One approach, which we call *(i) late scheduling* deals with these aspects exclusively at the replica side. A total order of commands is delivered at replicas which then, instead of sequentially executing them, detect conflicts among pending commands to schedule independent ones in parallel. In CBASE [13], replicas are augmented with a deterministic scheduler to accomplish this. Decided commands are inserted in a directed acyclic dependency graph. During command (node) insertion, dependencies with previously inserted commands are detected and included in the graph, as directed edges. Commands without dependencies are processed by a pool of threads. The execution of a command leads to the exclusion of its node from the graph, which removes dependencies. When multiple cores (threads) are used, the dependency graph introduces contention. Therefore in [14] the authors propose faster mechanisms to detect dependencies, however at the price of false positives, introducing a trade-off between scheduling overhead and concurrency level. The same problem is tackled in [17] from a different perspective. Here, algorithms and structures for a lock-free dependency graph are proposed, considerably reducing contention.

In [16], a technique called *(ii) early scheduling* is proposed to avoid contention at a synchronizing data structure (e.g., DAG). The idea is that scheduling decisions at replicas should be as expedite as possible. Therefore, clients classify commands into classes. Replicas have an a priori calculated mapping from classes to threads, derived from a definition of conflicts among command classes and expected workload per class. At replicas, the class information is used to dispatch the respective command to the input queue of one or more threads, according to the mapping already decided. The execution model and the thread mapping ensure the sequential execution of conflicting commands. This technique can lead to increased throughput, but performance can be penalized if the workload deviates from the expected when elaborating the threads to classes mapping. P-SMR [26] also avoids a central parallelizer or scheduler. This is achieved by mapping commands to different multicast groups at clients. Non-conflicting commands are propagated through different multicast groups that partially order commands across replicas. Commands are delivered by multiple worker threads according to the multicast group. This approach imposes a choice of destination group at the client side, based on command information which is application specific. Non-conflicting commands can be sent to distinct groups, while conflicting ones are sent to the same group(s). At the replica side, each worker thread is associated to a multicast group and processes commands as they arrive.

Instead of dealing with conflicts prior to command execution, as previously reported, *(iii) optimistic* techniques introduce an a posteriori approach. In Eve [10], replicas optimistically execute batched commands in parallel, as they arrive, and then check after execution if consistency is violated through agreement among replicas. In case of a consistency violation, replicas roll-back and re-execute the commands sequentially. While roll-back is expected to be rare, it impacts performance. In Storyboard [11], a forecasting mechanism predicts the same ordered sequence of locks across replicas. When forecasts are correct, commands can be executed in parallel. Otherwise, replicas stop command processing and use agreement to recompute the command's execution path. In [30], P-SMR [26] is extended with optimistic execution to increase concurrency among commands. Instead of conservatively assuming that two commands conflict when not enough information is available at the clients, it is optimistically assumed that commands do not conflict. If a conflict happens (detected upon execution), the involved commands must be re-executed in conflict mode.

Finally, some techniques employ *(iv) runtime coordination* mechanisms to ensure deterministic execution at replicas. Rex [12] uses an *execute-agree-follow* strategy. A single server, called primary, receives requests and processes them in parallel in 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. Then, it periodically 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. Trace synchronization may result in high network bandwidth consumption and performance overhead [31]. CRANE [31] uses another strategy to solve non-determinism during command execution. 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) [32]. Additionally, CRANE introduces a time bubbling technique to enforce deterministic logical times for request bursts. The runtime overhead is non-negligible. Besides agreeing on each socket event, the DMT system incurs 12.7 % of overhead.

Among the four surveyed classes above, our architecture is close to *late scheduling* approaches since scheduling decisions are all at the server side, before execution. Even though there has been considerable effort on exploring parallelism in state machine replication, no approach to date has used generalized consensus conflict information to favor concurrent command processing at replicas, the main contribution of this paper.

VI. Conclusion

This paper proposes an approach that benefits from generalized consensus to schedule parallel execution of independent SMR commands. This is a natural approach since the same conflict information from consensus is used during execution. Contrasting to architectures for parallel SMR that impose a total order with typical consensus, and then compute command dependencies for execution, the proposed approach favors both ordering and execution in an integrated fashion. A detailed performance evaluation leads us to conclude that using dependency information from consensus to favor the parallel execution of SMR commands is not only feasible, but also results in important performance gains.

Acknowledgements

We thank the reviewers for the constructive suggestions. This work is supported in part by the Swiss Government Excellence Scholarships, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and HP Brasil Indústria e Comércio de Equipamentos Eletrônicos LTDA using incentives of Brazilian Informatics Law (Law nº 8.2.48 of 1991).

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.
[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990.
[3] 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, 1985.
[4] M. Burrows, "The chubby lock service for loosely coupled distributed systems," in *OSDI*, 2006.
[5] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in Scatter," in *SOSP*, 2011.
[6] J. D. J. C. Corbett and M. E. et al, "Spanner: Google's globally distributed database," in *OSDI*, 2012.
[7] L. Lamport, "Generalized consensus and paxos," *Technical Report MSR-TR-2005-33, Microsoft Research*, 2005.
[8] F. Pedone and A. Schiper, "Handling Message Semantics with Generic Broadcast Protocols," *Distrib. Comput.*, vol. 15, no. 2, 2002.
[9] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is More Consensus in Egalitarian Parliaments," in *SOSP*. ACM, 2013.
[10] 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.
[11] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: Optimistic deterministic multithreading," in *HotDep*. USENIX Association, 2010.
[12] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: Replication at the speed of multi-core," in *EuroSys*. ACM, 2014.
[13] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*. IEEE, 2004.
[14] O. Mendizabal, R. de Moura, F. Dotti, and F. Pedone, "Efficient and deterministic scheduling for parallel state machine replication," in *IPDPS*. IEEE, 2017.
[15] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone, "Reconfiguring parallel state machine replication," in *SRDS*, 2017.
[16] E. Alchieri, F. Dotti, and F. Pedone, "Early scheduling in parallel state machine replica," in *ACM SoCC*, 2018.
[17] I. A. Escobar, E. Alchieri, F. L. Dotti, and F. Pedone, "Boosting concurrency in parallel state machine replication," in *Middleware*. ACM, 2019.
[18] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, vol. 43, no. 2, 1996.
[19] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, 1990.
[20] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, 1998.
[21] ——, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, 2006.
[22] R. Tarjan, "Depth-first search and linear graph algorithms," in *Annual Symposium on Switching and Automata Theory*. IEEE, 1971.
[23] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006.
[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally-distributed database," in *OSDI*, 2012.
[25] L. H. Le, C. E. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *DSN*, 2016.
[26] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state machine replication for parallelism," in *ICDCS*, 2014.
[27] C. E. Bezerra, F. Pedone, and R. V. Renesse, "Scalable state-machine replication," in *DSN*, 2014.
[28] D. da Silva Boger, J. da Silva Fraga, and E. Alchieri, "Reconfigurable scalable state machine replication," in *LADC*, 2016.
[29] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, 2004.
[30] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *IEEE SRDS*, 2014.
[31] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos made transparent," in *SOSP*, 2015.
[32] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 97–108, 2009.