2017 IEEE 37th International Conference on Distributed Computing Systems

High performance recovery for parallel state machine replication

Odorico M. Mendizabal* and Fernando Luís Dotti† and Fernando Pedone‡
*Universidade Federal do Rio Grande (FURG), Rio Grande, Brazil
†Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
‡University of Lugano (USI), Lugano, Switzerland

***Abstract*—State machine replication is a fundamental approach to high availability. Despite the vast literature on the topic, relatively few studies have considered the issues involved in recovering faulty replicas. Recovering a replica requires (a) retrieving and installing an up-to-date replica checkpoint, and (b) restoring and re-executing the log of commands not reflected in the checkpoint. Parallel techniques to state machine replication render recovery particularly challenging since throughput under normal execution (i.e., in the absence of failures) is very high. Consequently, the log of commands that need to be applied until the replica is available is typically large, which delays recovery. In this paper, we present two techniques to optimize recovery in parallel state machine replication. The first technique allows new commands to execute concurrently with the execution of logged commands, before replicas are completely updated. The second technique introduces on-demand state recovery, which allows segments of a checkpoint to be recovered concurrently.**

I. Introduction

Many internet services have strict availability and performance requirements. High availability requires tolerating component failures and can be achieved with replication. State machine replication (SMR) is a classical approach to managing replicated servers [1], [2]. In SMR replicas start in the same initial state and deterministically execute an identical sequence of client commands. Consequently, replicas traverse the same states and produce the same responses. To boost the performance of the service, one can deploy replicas in high-end servers (scale up). Modern servers, however, increase processing power by aggregating processors (or cores). Thus, to benefit from parallel architectures, replicas need to execute commands concurrently. Despite the fact that concurrent execution of commands seems at odds with SMR's requirement of deterministic execution, some approaches have revisited the classical SMR model to exploit parallelism (e.g., [3], [4], [5]).

Parallel state machine replication (PSMR) techniques are based on the observation that independent commands can execute concurrently while dependent commands must be serialized and executed in the same order by the replicas. Two commands are *dependent* if they access common state and at least one of the commands changes the state, and *independent* otherwise. Executing dependent commands concurrently may result in inconsistent states across replicas. Although the performance of PSMR depends on specifics of the technique and the workload mix of independent and dependent commands, studies report that parallel approaches to SMR result in large performance improvements (e.g., [3], [4], [5], [6]).

This paper considers recovery in PSMR, a topic that has received little attention in the literature. Although one could use recovery techniques designed for classical state machine replication (e.g., [7], [8], [9]), these are not appropriate for PSMR. To understand why, we briefly review the basics of recovery in SMR. During normal operation (i.e., in the absence of failures), replicas log the commands they execute and periodically checkpoint the application state. After replicas store the new checkpoint in stable storage, they can trim the log of commands, removing commands that are already reflected in the checkpoint. A recovering replica retrieves the most recent checkpoint and the log of "old commands", that is, commands that were already executed by the operational replicas but are not included in the retrieved checkpoint. The recovering replica can execute "new commands" after it has installed the checkpoint and executed the retrieved log of old commands.

Checkpoints play an ambivalent role in SMR. During normal operation, checkpoints hurt performance since they introduce overheads (e.g., execution stalls). Figure 1 quantifies checkpoint overhead in PSMR (details about the implementation and setup in Section VI). Frequent checkpoints result in bigger reductions in average throughput. This calls for sparse checkpoints, a strategy adopted by some systems (e.g., [8], [10]). Sparse checkpoints, however, result in large logs of old commands, which slows down recovery. Some techniques face this dilemma by trying to reduce the overhead of checkpoint creation (e.g., using copy-on-write). However, the frequency of checkpoints is also impacted by other practical concerns, such as checkpoint size and duration. This is because replicas typically perform checkpoints sequentially, only starting one checkpoint after the previous one has finished. Thus, checkpoint frequency is ultimately limited by how quickly a single checkpoint can be performed. Table I shows the measured log (in thousands of commands) during checkpoints with different sizes and durations. For example, it takes about 15 seconds to create and safely store a 512M-byte checkpoint, including serialization of the state. During this checkpoint, our PSMR prototype can execute nearly one million commands. Considering that the time between two checkpoints is at least the duration of the first checkpoint, the log sizes of Table I are minimum values.

Instead of attempting to increase the frequency of checkpoints to reduce the log of old commands (and thereby lower the downtime of a recovering replica), in this paper we rethink

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.193

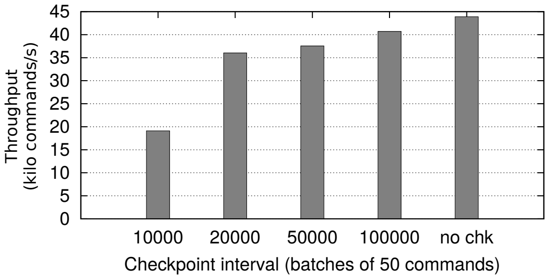


Fig. 1. Throughput versus checkpoint interval in PSMR.

Checkpoint size (MB)	Checkpoint duration (s)	Log size (in thousands of commands)
128	3.77	236
256	7.55	473
512	15.09	946
1024	30.19	1893
2048	60.38	3785

TABLE I
MINIMUM LOG SIZE (IN THOUSANDS OF COMMANDS) ACCORDING TO CHECKPOINT SIZE AND DURATION.

recovery from a more fundamental perspective. We discuss two techniques for high performance recovery in parallel state machine replication:

- *Speedy recovery of large logs.* Inspired by the mechanism introduced to handle dependencies among commands in PSMR (i.e., exploiting service semantics), we observe that a new command does not need to wait for an old command to be executed if the commands are independent. Speedy recovery allows concurrency of new and old commands if they are independent. Giving priority to independent new commands during recovery, we allow a recovering replica to answer new requests in reduced time. We propose techniques to speed up recovery while respecting dependencies among commands.
- *On-demand state recovery.* The second technique is based on the fact that a considerable amount of recovery time is due to state transfer and installation. We propose to divide a checkpoint into segments, and retrieve and install each segment only when it is needed for the execution of a command. We also allow segments to be concurrently retrieved and installed. Therefore, checkpoint segments are handled on demand and possibly concurrently. On-demand recovery is orthogonal to and could be combined with existing optimizations described in the literature, such as collaborative state transfer [7].

Speedy recovery and on-demand state recovery aim to minimize replica downtime. One may wonder why it is important to reduce the recovery time of a replica since non-faulty replicas can continue to serve client requests. It turns out, however, that the mean time to recover of a replica has an important impact on the reliability of a replicated system. For example, the mean time to fail of a two-replica system is computed as $MTTF^2/(2 \times MTTR)$, where $MTTF$ and $MTTR$ are the *mean time to fail* and the *mean time to recover* of a single replica, respectively [11]. By reducing the mean time to recover of a replica, one proportionally increases the mean time to fail of the compound without adding new replicas.

This paper makes the following contributions: (i) it discusses efficient techniques to reduce the recovery time of replicas in parallel state machine replication; (ii) it describes a full-fledged PSMR prototype that integrates speedy and on-demand state recovery; (iii) it experimentally assesses the performance of the proposed recovery techniques compares them to traditional recovery mechanisms under different scenarios.

The rest of the paper is organized as follows. Section II presents the system model. Section III recalls SMR, PSMR and recovery. Section IV introduces a protocol for efficiently handling command dependencies in PSMR. Section V discusses high performance recovery in PSMR. Section VI experimentally assesses the performance of the proposed protocols. Section VII surveys related work and Section VIII concludes the paper.

II. SYSTEM MODEL

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 behavior (e.g., no Byzantine failures). Process may crash and recover. A process is *correct* if it eventually remains up forever, or *faulty* otherwise. We assume f faulty servers, out of $n = 2f + 1$ 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 a sender sends a message enough times, a correct receiver will eventually receive the message. One-to-many communication is based on atomic broadcast, whose main primitives are *broadcast*(m) and *deliver*(i, m), where i refers to the consensus instance in which m was decided. This definition implicitly assumes that atomic broadcast is implemented with a sequence of consensus instances identified by natural numbers (e.g., [12], [13]). By introducing the consensus instance in the delivery event, a server can easily determine the messages it needs to retrieve upon recovering from a failure.

Atomic broadcast ensures that (i) if a process broadcasts message m and does not fail, then there is some i such that eventually every correct process delivers (i, m); and if a process delivers (i, m), then (ii) all correct processes deliver (i, m), (iii) no process delivers (i, m') for $m \neq m'$, and (iv) some process broadcast m. We implement atomic broadcast using Paxos [13]. Paxos requires additional synchronous assumptions but our protocols do not explicitly need these assumptions.

III. BACKGROUND

In this section, we review classical SMR, a parallel approach to SMR, and recovery in classical SMR.

A. Classical 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.

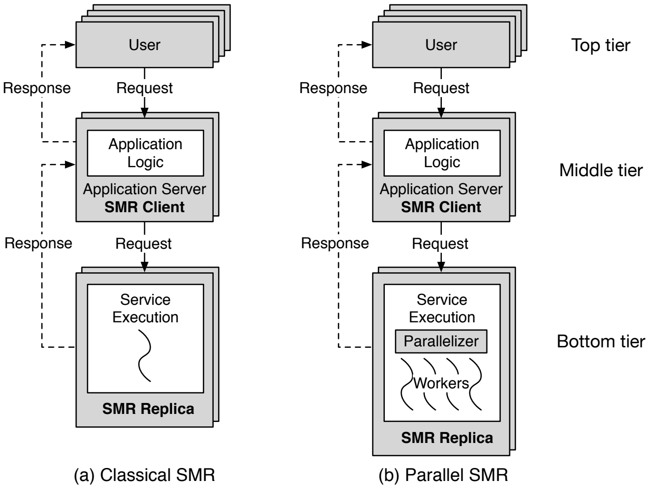
SMR provides clients with the abstraction of a highly available service while hiding the existence of multiple replicas. This last aspect is captured by *linearizability*, a consistency criterion [14]: a system is linearizable if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time ordering of commands across all clients [15]. In classical SMR, linearizability can be achieved by having clients atomically broadcast commands and replicas 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 clarification now is in order about the role of SMR clients and servers in the broader context of a distributed application. Large distributed systems are typically structured in tiers, a three-tier system being a prototypical case (see Figure 2). In these environments, users are at the top tier and submit requests to application servers, at the middle tier. Application servers execute the application logic and submit requests to the bottom tier. The bottom tier is responsible for handling the application state. In this context, SMR clients are the application servers in the middle tier and SMR replicas are the servers in the bottom tier. In this paper, we refer to clients and servers from the perspective of state machine replication.

B. Parallel state machine replication

Classical SMR makes poor use of multi-processor architectures since deterministic execution normally translates into (single-processor) sequential execution of commands. Although (multi-processor) concurrent command execution may result in non-determinism, it has been observed that "independent commands" can be executed concurrently without violating consistency [2]. A few approaches have been suggested in the literature to execute independent commands concurrently with the goal of improving performance (e.g., [3], [4], [5]). In this section, we describe CBASE, the approach proposed in [4], which we use as the motivation for the efficient recovery techniques proposed in this paper. We recall other approaches to parallel SMR in Section VII.

To parallelize the execution of independent commands, CBASE adds a deterministic scheduler, also known as *parallelizer*, to each replica (see Figure 2(b)). Clients atomically broadcast commands and the parallelizer at each replica delivers commands in total order, examines command dependencies, and distributes them among a pool of worker threads for execution. The parallelizer uses a dependency graph to maintain a partial order across all pending commands, where vertices represent commands and directed edges represent dependencies. A command is pending at a replica if it has been delivered at the replica but not yet executed. While dependent commands are ordered according to their delivery order, independent commands are not directly connected in the graph. Worker threads receive independent commands from the parallelizer (i.e., vertices with no incoming edges) to be concurrently executed. When a worker thread completes the execution of a command, it removes the command from the graph and responds to the client that submitted the command.



(a) Classical SMR

(b) Parallel SMR

Fig. 2. Classical and parallel state machine replication in a 3-tier application.

Figure 3(a) depicts an illustrative dependency graph with six commands, delivered in the order $a, b, ..., f$. Commands a, c and e are the next ones to be scheduled for execution and can execute concurrently. Commands a and b are dependent but a was delivered first; so, a must execute before b. Intuitively, fewer interdependencies between commands in the dependency graph favor concurrency. However, the cost of adding a new command in the dependency graph is proportional to the number of commands in the graph that are independent of the new command. For example, a new command g will be first compared to commands d and f; if g is independent of d, it will be compared to c and b, and so on. If g is independent of every command in the graph, it will be compared against all vertices. In Section IV we describe a more efficient way to handle command dependencies.

C. Recovery in state machine replication

The SMR approach has shown to be efficient and practical to develop applications that tolerate both crash [16], [17], [18], [19], [20], [21] and byzantine [8], [10], [22], [23], [24] failures. Despite the failure model adopted by existing solutions, most SMR implementations in the literature follow a similar recovery

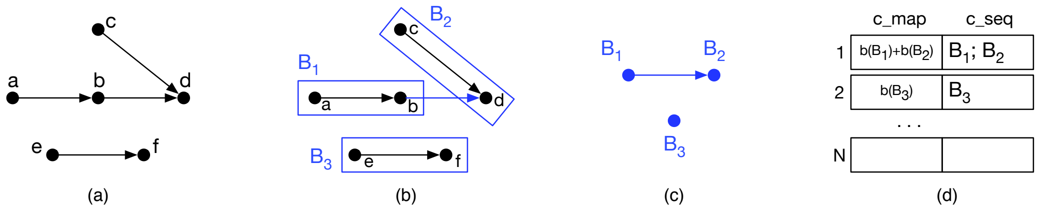


Fig. 3. Four representations of a dependency graph with six commands. (a) The original dependency graph, where edge $x \to y$ means that commands x and y are dependent and x was delivered before y. (b) The original graph grouped in batches of two commands. (c) The abridged dependency graph; notice that commands b and c are serialized in the abridged graph. (d) The stored dependency graph; batches B_1 and B_2 are assigned to worker thread t_1 and batch B_3 is assigned to worker thread t_2; $b(B)$ is a digest of the variables accessed by commands in B, used to track dependencies between command batches. The stored dependency graph preserves all dependencies in the original dependency graph.

approach: (i) Commands are logged in the order in which they are executed (as part of atomic broadcast) and a reply is only sent to the client after the corresponding command is logged and executed. (ii) Each replica periodically checkpoints its state to stable storage. Logged commands preceding the moment in which the checkpoint was taken are discarded from the log. Old checkpoints common to all replicas are also discarded. (iii) To recover from a failure, the recovering replica retrieves the latest checkpoint (i.e., state transfer) and the log with executed commands that are not reflected in the checkpoint from another replica or from remote storage. The recovering replica rebuilds its state by installing the checkpoint and replaying the log of retrieved commands not included in the last checkpoint. Client commands delivered during recovery are only processed after the recovery procedure is complete.

Regardless the specific recovery protocol in use, general optimizations on logging, checkpointing and state transferring can minimize recovery overhead, improving overall system performance. For instance, instead of logging single operations, some works log batches of operations [7], [8], [25], [22], [26] or perform logging of batches in parallel [7]. Creating a checkpoint after processing a certain number of commands, as proposed in most works in SMR (e.g., [8], [22], [27], [21], [26]) can degrade the performance of the service. This happens because commands are not processed while replicas save their state. If all replicas create checkpoints at approximately the same time, the necessary agreement quorum might not be available. Consequently, clients observe stalls in the execution during checkpoints. To overcome these problems, in [7] the authors force replicas to take checkpoints at different times. The idea is that there is always a quorum of replicas that can order and execute commands. In [9] the use of a helper process for taking checkpoints asynchronously is proposed. While the primary continuously processes requests and sends replies to the clients, the helper periodically takes checkpoints.

During a state transfer, at least one replica has to spend resources to send its own state to another replica. One way to avoid performance degradation is to ignore state transfer requests until the workload is low enough to process both the state transfer and regular messages [20]. Another way to minimize this overhead is through the reduction of the amount of information transferred. State can be efficiently represented by data structures based on hierarchical state partitions, as proposed in [8], incremental checkpoints [9], [28], or compression techniques. In [7], the authors present a collaborative state transfer protocol, so the burden imposed on replicas is evenly distributed among them.

IV. Dependency handling in PSMR

As already mentioned, PSMR techniques exploit service semantics to determine when commands are independent and can be executed concurrently [3], [4], [5]. Due to the high throughput of PSMR, the costs of identifying dependencies among incoming and pending commands, and then scheduling commands accordingly are non negligible. In this section, we introduce techniques to efficiently handle dependencies and schedule independent commands to execute concurrently. We combine three strategies:

- Batching: Instead of ordering one command at a time, batches of commands are handled during normal operation and recovery.
- Fast conflict detection: Each batch of commands contains a bitmap with a digest of the variables read and written by the commands in the batch. The bitmap is used for fast detection of conflicts among batches.
- Reduced dependency handling overhead: While tracking dependencies among batches of commands, the computation performed by the parallelizer is bounded by the number of worker threads at a replica and not the number of pending commands, as in [4].

Batched commands. Clients submit commands through a middle tier or proxy (see Figure 2), which groups commands from different clients and broadcasts the commands for execution as batches. When the proxy receives responses for all commands in a batch, it can submit a new batch of commands. There can be any number of client proxies, each one handling a group of clients. Batching increases throughput by reducing the overhead needed to handle commands (e.g., fewer system calls to deliver commands, fewer edges to store the graph, fewer comparisons to determine dependencies)[29].

Abridged conflict information. The way batche's bitmaps are encoded to satisfy the conflict detection property is application specific and can be achieved in different ways. In our prototype, we consider read and write commands in a database, where each operation includes the key of the entry read or written in the database. Therefore, we create bitmaps by hashing the key provided in the command; the hashed value corresponds to a bit set in the bitmap. Checking whether two batches contain dependent commands boils down to a bit-wise comparison of their bitmaps. While this approach is subject to false positives (i.e., it may detect a conflict when none exists), it is not prone to false negatives (i.e., it does not miss real conflicts).

The dependency graph. The paralellizer delivers batches in total order. Given the bitmaps of two batches, $b(B_i)$ and $b(B_j)$, the parallelizer determines whether there is (at least) a command in batch B_i that depends on some command in batch B_j by comparing their bitmaps. In such case they conflict and if B_i is delivered before B_j the conflict is represented by adding an edge from batch B_i to B_j in the dependency graph (see Figure 3(b) and (c)). This process inductively builds a directed acyclic graph since a new incoming batch may depend (insert edges) on elements of the set of already pending batches, and not the other way.

The stored dependency graph. Although the dependencies among batches assume the topology of a DAG, as discussed above, this structure is not actually stored. We introduce an optimization that allows new command batches to be added to the structure in $O(tb)$, where t is the number of worker threads and b the size of the bitmaps, instead of $O(g)$, where g is the size of the dependency graph. The goal of this optimization is to spare computation at the parallelizer, at the cost of limiting concurrency among commands. Hereafter, we refer to a batch of commands with at least one pending command as a pending batch of commands. Instead of comparing each incoming batch to all pending ones and storing the complete dependency information, the parallelizer considers the number of worker threads. The parallelizer and the worker threads share two data structures, each one with one entry per worker thread (see Figure 3 (d)). For each worker thread t_i, $c\_seq[i]$ contains the sequence of command batches the parallelizer assigned to t_i and $c\_map[i]$ contains a bitmap that encodes all commands in $c\_seq[i]$. When the parallelizer delivers a new batch B, it checks B's bitmap, $b(B)$, against each $c\_map[i]$ to determine which worker threads need to coordinate in order for commands in B to be executed. If no interdependencies are detected, the parallelizer chooses one worker thread t_i (e.g., the least loaded one), assigns B to t_i, by appending B to $c\_seq[i]$, and updates t_i's bitmap $b\_map[i]$. If there are interdependencies, the parallelizer determines all worker threads that have been scheduled commands on which B depends, adds B to the end of c_seq of such threads and updates their bitmaps.

The execution of commands. Each worker thread t_i executes commands following their order in $c\_seq[i]$, where commands in the same batch are handled in the order they appear in the batch. When a worker reaches a command batch B that requires coordination among workers, all involved workers coordinate so that a single worker executes the commands in B. After the thread executes all commands in B, it signals the other worker threads involved to proceed. When thread t_i is finished with B, t_i removes B from $c\_seq[i]$ and recomputes $c\_map[i]$ based on the batches currently in $c\_seq[i]$. As a consequence of this procedure, the parallelizer and the worker threads must access structures c_seq and c_map in mutual exclusion.

V. High performance recovery

In this section, we introduce two techniques to reduce recovery time: speedy recovery and on-demand state transfer. We conclude the section with a discussion about the implications of these techniques on system recovery.

A. *Speedy recovery of large logs*

Recovery typically faces the following tradeoff. In order to minimize the number of old commands a recovering replica needs to retrieve and execute, checkpoints must be frequent. Checkpoints, however, may degrade performance during normal execution; thus, for performance checkpoints should be infrequent. Even though some techniques strive to reduce the impact of checkpointing on normal execution (e.g., by using copy-on-write), checkpoint frequency is ultimately limited by how quickly a single checkpoint can be performed. Since concurrent checkpoints may overload the system and further complicate the design, checkpoints are typically configured to happen sequentially, that is, one checkpoint only starts after the previous one has finished.

In systems designed for high throughput, such as parallel state machine replication, a practical consequence of the recovery tradeoff mentioned above is that a recovering replica needs to handle a large sequence of old commands before it can execute new commands. This situation renders the replicated system more vulnerable to failures since a recovering replica is only available once it can process new client commands.

Instead of trying to reduce the sequence of old commands (e.g., by increasing checkpoint frequency), we approach the recovery tradeoff from a different perspective: we allow new commands to execute before old commands have been processed. In brief, our strategy is based on the observation that a new command does not need to wait for an old command to be recovered and executed if the two commands are independent.

In the rest of this section, we explain how we integrate this strategy in the parallel state machine replication scheme described in Section IV.

During normal operation, replicas create a *dependency log*, a data structure that contains bitmaps of command batches the replica executed since its last checkpoint. When a replica creates a new checkpoint, it trims its dependency log. To recover from a failure, a replica retrieves a recent checkpoint from stable storage, and the dependency log from an operational replica. Old commands not included in the restored checkpoint will be recovered by the replica using the delivery primitive of atomic broadcast. Since the dependency log contains a digest of the old commands (not the entire commands), it can be retrieved much more efficiently than the actual commands, which require executions of atomic broadcast to be recovered.

With the dependency log, the recovering replica can execute new commands before processing all old commands. The replica splits the delivery of commands into two flows: one flow with new commands and one flow with old commands. Old commands are scheduled for execution as during normal execution. A new command is scheduled for execution if it is independent of every old command that has not been scheduled yet. The replica uses the dependency log to check whether a new command is independent of pending old commands.

Algorithm 1 introduces the recovery of parallel state machine replication. When a replica starts, it first retrieves a checkpoint and the checkpoint identifier (line 12). The checkpoint identifier is the largest delivery instance of a command in the checkpoint. The replica then installs the checkpoint (line 13).

After the checkpoint is installed, the replica queries the atomic broadcast module to determine the latest instance of a delivery event (line 14), retrieves the dependency log with bitmaps of batches containing old commands, that is, commands in batches delivered before the latest instance (line 15), initializes variables i and k, which will keep track of delivery events for old and new commands, respectively (lines 16 and 17), and variables n_seq and n_map, which contain the sequence of new batches that cannot yet be executed and their bitmap representation (lines 18 and 19).

When the parallelizer delivers a batch B of old commands (line 22), it schedules B for execution and gets ready for the next batch (lines 23 and 24). If there are no more batches of old commands, the parallelizer schedules all batches of new commands that have been delivered but could not yet be scheduled because they contain commands that depended on old commands, either directly or indirectly (lines 25–27).

When the parallelizer delivers a batch B of new commands (line 28), it checks whether B's bitmap intersects with the bitmaps of batches with old commands that have not been scheduled yet (i.e., d_map structure) and with new commands that precede b (i.e., n_map structure). If there is no intersection, B is scheduled for execution (lines 29–30); otherwise, B is appended to n_seq to be executed later (line 32) and its related bitmap n_map is updated (line 33). Finally, the parallelizer is ready to deliver the next batch of new commands (line 34).

Algorithm 1

```
 1: procedure schedule($b : Batch$)
                                  {batch scheduling as discussed in Section 4}
 2: function ($chkp, int$) ← lastCheckpoint()
                       {returns checkpoint and the last instance number included}
 3: function $int$ ← currentDeliveryInstance()
                                        {returns current message instance number}
 4: function $bitmapSequence$ ← dependencyLog($i, j : int$)
                               {returns sequence of bitmaps of delivered batches i to j}
 5: Global strucutres:
 6:    $i, j, k$      {i..j range of missing instances; k is first new instance}
 7:    $n\_seq$               {sequence of batches with new commands}
 8:    $n\_map$                {bitmap associated with sequence above}
 9:    $d\_map$                 {bitmap sequence of missing batches}
10: Upon Restarting:
11: procedure ReInitialization()
12:    ($checkpoint, i$) ← lastCheckpoint()
13:    install $checkpoint$
14:    $j \leftarrow$ currentDeliveryInstance()
15:    $d\_map[i..j] \leftarrow$ dependencyLog($i, j$)
16:    $i \leftarrow i + 1$
17:    $k \leftarrow j + 1$
18:    $n\_seq \leftarrow \varnothing$
19:    $n\_map \leftarrow 0$
20: The parallelizer executes as follows:
21: when (deliver($i, B$) and $i \leq j$) or deliver($k, B$)
22:    if delivered ($i, B$) then                  {a batch of old commands}
23:       schedule($B$)                          {schedule batch for execution}
24:       $i \leftarrow i + 1$                        {set next old batch to be retrieved}
25:       if $i > j$ then                       {if done with batches of old commands}
26:          for each $B \in n\_seq$, in order do            {pending batches...}
27:             schedule($B$)                   {...of new commands are scheduled}
28:    else                                   {delivered a batch B of new commands}
29:       if $b(B) \cap (d\_map[i] \vee ... \vee d\_map[j] \vee n\_map) = \varnothing$ then
          {B does not depend on batches that precede B}
30:          schedule($B$)                       {schedule B for execution}
31:       else                                                 {B depends}
32:          $n\_seq \leftarrow n\_seq \oplus \langle B \rangle$        {add B to pending sequence}
33:          $n\_map \leftarrow n\_map \vee b(B)$            {update related bitmap}
34:       $k \leftarrow k + 1$                                  {set next new batch}
```

B. On-demand state recovery

Although processing new commands concurrently with old commands improves the availability of the system by bringing a recovering replica back up more quickly, old and new commands can only execute after the recovering replica has transferred and installed a checkpoint. On-demand state recovery addresses this shortcoming. The overall idea is to partition the service state into segments, and retrieve and install segments only when they are needed for the execution of a command (either a new or an old command). A checkpoint now is a collection of segments.

In order to implement on-demand state recovery, we have moved the logic involved in the retrieval and installation of a checkpoint to the worker threads, instead of performing it as the first action of a recovering replica. As soon as a recovering replica has retrieved the dependency log, it can schedule commands, as described in the previous section. Before a worker thread executes a command, it checks whether the needed segments are already installed or not. If a needed segment is not installed, the worker thread retrieves the segment from an operational replica (or remote storage), installs the segment, and then executes the command.

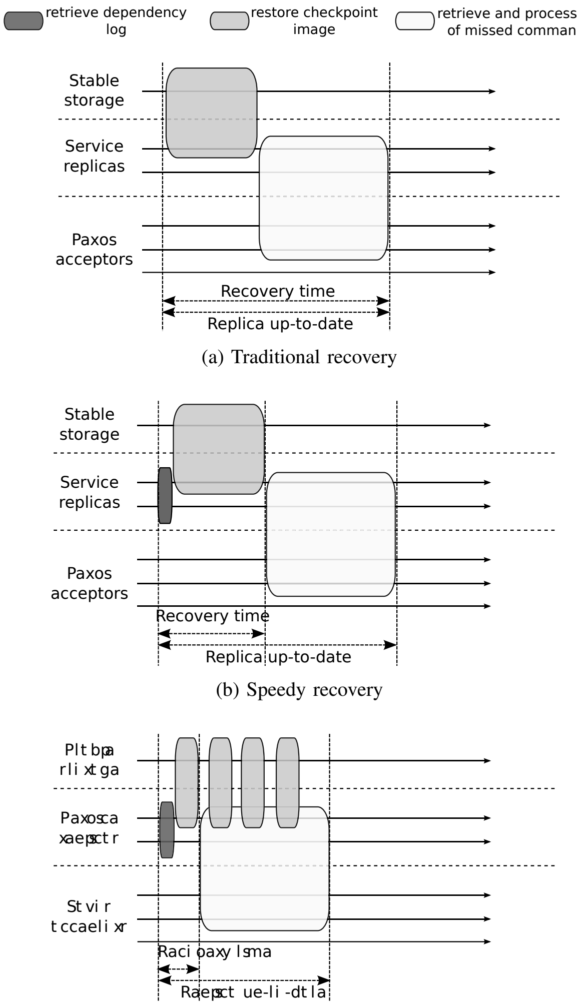
This scheme improves performance in two aspects. First, it defers the transferring and installation of segments to when they are needed. Second, it allows to parallelize these operations on different worker threads.

C. Implication of techniques on recovery

Differently from traditional recovery, we differentiate the instant in which the replica becomes available to process new commands (*recovery time*) from the instant in which the replica has finished processing all missing commands and is completely updated (*update time*).

Figure 4(a) depicts the steps involved in traditional recovery: a recovering replica retrieves and installs a checkpoint image from stable storage (e.g., local disk, NAS), and processes the log of missing commands obtained from the atomic broadcast module (e.g., Paxos). At this point the replica is recovered and updated, so it can start processing new commands.

With speedy recovery (Figure 4(b)), as soon as the dependency log is obtained from another replica, the recovering replica can start serving new commands while it recovers its state. When on-demand transfer is enabled (Figure 4(c)), the recovering replica can process new commands right after it retrieves a single checkpoint segment.



(a) Traditional recovery

(b) Speedy recovery

(c) Speedy recovery with on-demand state transfer

Fig. 4. Time to recover for each recovery technique.

VI. Evaluation

In this section, we describe our prototype, explain our assessment goals and methodology, describe the experiment environment and present the results of our performance study.

A. Implementation

In order to evaluate our recovery technique, we developed a key-value store service using the approach discussed in Section IV. In our prototype, we use 8-byte keys and 1024-byte values.

The service implements commands to create, read, update and remove keys from an in-memory database. The service periodically checkpoints its state and stores it in a remote file system. Atomic broadcast is implemented by Ring Paxos [30], a high-throughput atomic broadcast protocol.

Client commands are forwarded to a client proxy, which is responsible for batching and compressing those commands in a single request. To alleviate the burden on the parallelizer, the bitmaps for a batch are computed by the client proxy. Client proxies broadcast a request to the replicas and wait for the first reply from a replica for every command in the batch before broadcasting another batch. In the experiments we use update commands only since these are the most challenging ones from the perspective of state handling.

Upon receipt of a batch, a replica proceeds as discussed in Section IV. To execute commands, each worker thread decompresses and extracts commands from received batches before executing them. The dependency log discussed in Algorithm 1 is implemented as a list of consolidated bitmaps for groups of delivered batches. Thus, instead of keeping one bitmap for every single batch, several batches are grouped and associated to a bitmap that is computed as the union of the bitmaps of the batches in the group.

B. Goals and methodology

The high performance recovery techniques introduced in this paper aim to speed up recovery of large logs and reduce state transfer. Ultimately, both techniques are designed to increase a replica's availability. We wish to quantify the effects of these optimizations with emphasis on the following goals.

- **Recovery time.** A replica is recovered as soon as it can process new commands. We assess the time it takes for a replica to recover using the techniques introduced in the paper and compare them to classical recovery.
- **Throughput during recovery.** Speedy recovery allows new commands to be processed before recovery is finished, differently from classical recovery techniques. We determine the throughput of new commands during the restart of a replica.
- **Recovery breakdown.** We investigate the interplay of the steps involved in speedy recovery and on-demand state transfer and how each step contributes to recovery duration.
- **Impact on MTTF.** Since a recovering replica becomes available faster, the window of vulnerability for service outage is reduced. We quantify the gains by computing the mean time to fail (MTTF) of our proposed approach.

C. Environment and configuration

All experiments were executed on a cluster with two types of nodes: HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory; and Dell PowerEdge R815 nodes equipped with four 16-core AMD Opteron 6366HE processors running at 1,8 GHz and 128 GB of main memory. The HP nodes were connected to an HP ProCurve switch 2920–48G gigabit network switch, and the Dell nodes were connected to another, identical network switch. The switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 6.5 and had the Oracle Java SE Runtime Environment 8. Paxos' proposer, acceptors, and clients were deployed on HP nodes, while PSMR replicas were deployed on Dell nodes. Our prototype was set up to tolerate one failure, requiring three acceptors and two replicas.

Although we focus on recovery performance in this paper, we performed a series of experiments covering a range of scenarios helpful to identify the parameter space. Once we surveyed the performance of our prototype for several important parameters, we fix the system with 8 threads, operational load at 70% of peak throughput, batch size of 50 commands, checkpoint interval at 20K batches and investigate the behavior of recovery for 0% and 5% dependency probabilities.

The reason for choosing dependency probability of 0% is to understand the potential of the technique (i.e., it results in the best performance), while 5% is more than one should expect in a typical application, according to the literature. Moraru et al. [31] state that from the available evidence, dependency probabilities between 0% and 2% are the most realistic. For instance, in Chubby, for traces with 10 minutes of observation, fewer than 1% of all commands could possibly generate conflicts [16]. In Google's advertising back-end, F1, fewer than 0.3% of all operations may generate conflicts [19].

D. On efficient recovery in PSMR

Figure 5 shows the execution of a recovering replica combining speedy recovery with on-demand state transfer. We set up the space of generated keys in a way that checkpoint size is around 512M bytes and the dependency probability is 0%. The recovering replica takes about 7 seconds to recover, a period that corresponds to the time to download and install at least one checkpoint segment. Right after that, the replica can process new commands in that segment while downloading and installing other checkpoint segments on-demand.

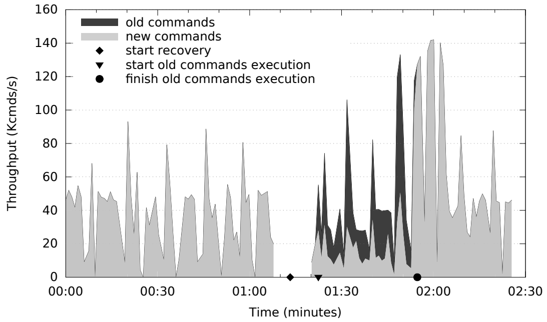


Fig. 5. Throughput using speedy recovery and on-demand state transfer with 0% of dependency probability.

Figure 6 shows a similar execution, but with dependency probability of 5%. The recovering replica also takes about 7 seconds to recover, but the throughput of new commands processed during recovery is lower. As expected, as the dependency probability increases, fewer new commands can be scheduled for execution in parallel with old commands. In the worst case, when all new commands conflict with commands in the dependency log, only old commands would be processed during recovery. This behavior resembles the classic recovery.

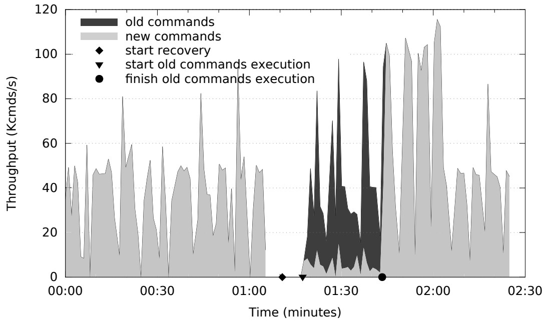


Fig. 6. Throughput using speedy recovery and on-demand state transfer with 5% of dependency probability.

In the previous experiments, once the first checkpoint segment is installed, the replica starts processing new and old commands assigned to that segment. The total throughput is given by the sum of the black and gray areas, representing the throughput of old and new commands, respectively.

Table II quantifies the recovery behavior for workloads with dependency probability of 0% and 5%. The time to download and install a checkpoint in both classical and speedy recovery techniques is similar. However, when using the on-demand state transfer technique, these times can be considerably reduced. For instance, checkpoint segment P3 takes less than half of the time taken by other techniques.

An important metric is the time to execute the first new command, since the replica is able to process new commands from that moment on. Obviously, in the classical approach, the replica processes the first new command after processing the whole log. Thus, the time to execute the last old command and the first new command are practically the same (approximately 33s in Table II). Speedy recovery can substantially reduce the time to execute the first new command. Our experiments demonstrated that speedy recovery is three times faster than the classical recovery in contention-free workloads. When on-demand state transfer is combined, recovery becomes more than 7 times faster.

Since our techniques aim at minimizing the unavailability of a replica, we investigate the time needed to transfer a checkpoint, to install the checkpoint, to process the log, and the moment when the first new command is serviced, denoting that the service is available for new requests. Figure 7 provides a graphical representation for the recovery cost breakdown.

Besides the dependency probability, the incoming rate of old commands also impacts the throughput of new commands during recovery. Higher rates increase the number of old commands competing with new ones to be processed. In the extreme case, the number of old commands delivered is so high that no new command is delivered while old commands are processed. This behavior is similar to the behavior of classic recovery, where old commands are processed first. By reducing the incoming rate of old commands, the system naturally increases the number of new commands delivered during recovery, increasing the chances of new commands being processed in parallel to old commands. In our experiments we set up the relearning of old commands to 200 batches every 0.2 seconds.

Finally, we compute the reduction in the mean time to fail ($MTTF$) of a service that uses our proposed approach. From [11], the $MTTF$ of a two-replica system is given by $MTTF_{single}^2/(2 \times MTTR_{single})$, where $MTTF_{single}$ and $MTTR_{single}$ are the mean time to fail and the mean time to recover of a single replica, respectively. Therefore, the reduction γ in $MTTF$ caused by *speedy* recovery, when compared to *classic* recovery, can be calculated as:

$$\gamma = \frac{MTTF_{classic}}{MTTF_{speedy}} = \frac{MTTF_{single}^2/(2 \times MTTR_{classic})}{MTTF_{single}^2/(2 \times MTTR_{speedy})}$$

which can be simplified as:

$$\gamma = \frac{MTTR_{speedy}}{MTTR_{classic}}$$

By comparing the "time of first new command" of classic recovery and speedy recovery (see Table II), we observe that $\gamma = 4.40/33.41 = 0.13$ and $\gamma = 2.89/32.37 = 0.09$, for 0% and 5% dependency probability, respectively.

VII. Related work

In this section, we review existing approaches to recovery in classical and parallel state machine replication. We conclude with a brief account of recovery in replicated database systems based on group communication.

A. Recovery in classical state machine replication

In Section III-C, we presented the basics of recovery in state machine replication. More advanced techniques have been proposed to improve the efficiency of logging, checkpointing and recovery in SMR. In [7] three techniques are proposed: parallel logging, sequential checkpointing and collaborative state transfer. The key ideas of parallel logging are to log groups operations instead of individual operations, and process operations in parallel with their storage. Grouping operations is conceptually similar to our batched commands. Taking advantage of replication in SMR, sequential checkpointing coordinates replicas such that they do not checkpoint their states at the same time to avoid hiccups during normal execution. While one replica is taking a checkpoint, other replicas continue to process requests. Instead of the traditional state transfer from one single replica, collaborative state transfer proposes that several replicas may send part of their checkpointed state to a recovering replica. These ideas are orthogonal to the ones we propose and could be used in parallel state machine replication. Some works have also discussed how replica recovery can be integrated with group communication primitives [30] and how to minimize the effects of a recovering replica on normal execution [32].

B. Recovery in parallel state machine replication

Although the recovery techniques described in the previous section could be used in parallel approaches to state machine replication, some proposals leverage specifics of the protocol to perform checkpoints and recovery efficiently. None of these proposals allow concurrent execution of old and new commands, neither implement on-demand checkpoint transferring.

We have already described CBASE's normal operation in Section III-C. Checkpointing is briefly discussed in [4] and recovery is not mentioned. To ensure that all replicas build the same sequence of checkpoints, a synchronization primitive executed at the replicas, but invoked by the agreement layer, is used to select a sequence number for checkpoints. Each replica blocks the execution of all the requests delivered after this sequence number until the checkpoint is completed.

In Eve [3] replicas first execute commands and then verify the equality of their states through a verification stage. Before execution, a primary replica groups client commands into batches and transmits the batched commands to all replicas. Then, replicas speculatively execute batched commands in parallel. After the execution of a batch, the verification stage checks the validity of replica's state, as defined by the common state reached by a majority of replicas. If too many replicas diverge, replicas roll back to the last verified state and re-execute the commands sequentially and deterministically. Checkpointing in Eve seems straightforward, but is not discussed in [3].

In [5], the authors propose a variation of parallel state machine replication, where the execution and the delivery of commands occur in parallel. Instead of using a single sequence of consensus executions to order commands, the approach uses multiple consensus sequences. Independent commands proposed in different sequences of consensus are executed concurrently. Dependent commands are proposed either in the same sequence or in a sequence shared among all threads; in both cases, dependent commands are executed in the same order across replicas. Checkpointing solutions for the protocol in [5] are proposed in [33]. One solution forces replicas to converge to a common state before checkpointing, similarly to [4]. In the other solution, replicas take checkpoints independently, reducing the overhead involved in synchronizing threads during a checkpoint. This solution is not applicable to the protocol proposed in this paper.

In Rex [34], a single server receives requests and processes them in parallel. While executing, the server logs on a trace dependencies among requests based on the shared variables accessed (locked) by each request. The server periodically proposes the trace for agreement to the pool of replicas. Together it also periodically proposes cuts in the computation. The other replicas receive the traces and replay the execution respecting the partial order of commands. If a cut is provided, a secondary replica, when achieving that cut, creates a snapshot of the state and propagates it to all other replicas. Recovery is performed with the installation of a recent snapshot followed by the

	Dependency probability = 0%						Dependency probability = 5%					
	Classic	Speedy	Speedy + on-demand				Classic	Speedy	Speedy + on-demand			
	recovery	recovery	P1	P2	P3	P4	recovery	recovery	P1	P2	P3	P4
Checkpoint download (s)	5.17	5.24	5.44	4.16	2.91	5.23	4.99	5.63	4.27	1.72	2.24	3.98
Checkpoint installation (s)	4.57	4.60	1.08	1.10	1.49	1.07	4.74	4.68	1.07	1.05	1.56	1.11
Log processing (s)	22.25	35.53	35.36				22.65	40.19	33.21			
Time of last old command	33.41	45.37	39.85				32.37	50.50	38.30			
Time of first new command	33.41	10.43	4.40				32.37	11.38	2.89			
Recovery speedup	**1**	**3.2**	**7.6**				**1**	**2.8**	**11.2**			
Throughput (normal execution)	34742	34705	36000				34446	34138	36100			
New command throughput (recovery)	0	26242	19702				0	11930	9546			

TABLE II
RECOVERY TECHNIQUES COMPARISON.

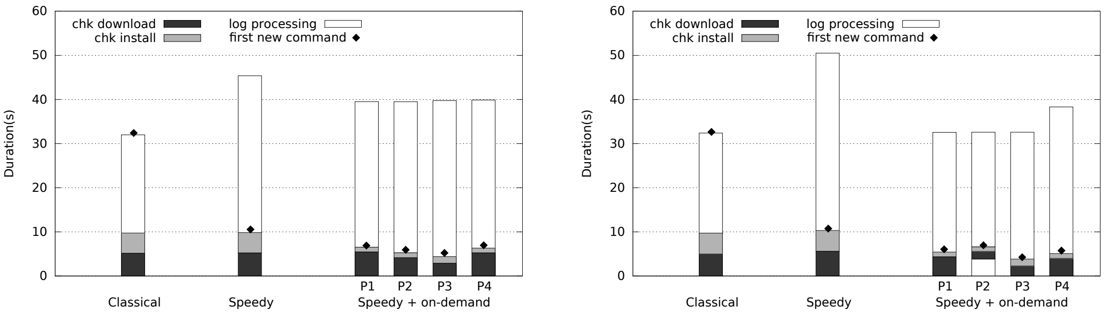


Fig. 7. Time taken during recovery for workloads with dependency probability 0% (left) and 5% (right).

replay of the logged commands according to their dependencies. During recovery of a replica, the throughput experienced is around 20% of normal operation and takes around 25 seconds to complete. Strictly, Rex does not implement state machine replication since only one replica executes commands, while the others follow its execution.

C. Recovery in transactional systems

The problem of efficiently recovering a failed replica has been largely considered in the context of database systems. Replication protocols based on group communication are the closest to our approach in that transactions are ordered before they can be committed. Some of these protocols explicitly address recovery. In [35] the authors discuss how to recover a crashed replica (or start a new one) without stopping transaction processing. The recovered replica only accepts new transactions once recovery has finished. In [36] crashed replicas can recover in parallel and at the same time several active replicas can serve them the needed data. The protocol in [37] proposes an adaptive approach which allows a recovering replica to catch up with operational replicas by transferring either the recent values of data items or the sequence of missed updates. In these works, transactions can only be processed once old transactions have been recovered. One exception is the approach of [38] in which new transactions can be executed before recovery has completed. The solution proposed in [38] builds a data structure that resembles the dependency graph, which is inappropriate in environments subject to very large performance.

VIII. CONCLUSION

Current advances in parallel state machine replication allow independent commands to be executed concurrently in a replica. To keep replicas consistent, each replica has to carefully handle and respect dependencies among commands. This is a non-trivial task since it requires dependency detection on a possibly high volume of commands. In this paper, we have proposed high performance recovery in PSMR, a set of coordinated techniques to reduce a crashed replica's unavailability period. Speedy recovery is a technique that naturally benefits from command dependencies, allowing new commands to be processed concurrently with old commands, if they are independent. On-demand state recovery enables to recover segment of the state when they are needed instead of recovering the whole state at once. Both techniques have proved to considerably reduce recovery time, when compared to traditional recovery in SMR. Integrated to our recovery strategy, we have proposed mechanisms to efficiently represent and calculate dependency among commands, complemented by an efficient scheduling mechanism that considers both dependencies and resource availability (number of working threads) and thus computes dependencies only when useful to parallelize commands execution.

ACKNOWLEDGMENT

We would like to thank Alysson Bessani for his comments and valuable advices on an earlier version of this paper. This work is partially supported by CAPES Brasil, PVE Project 88887.124751/2014-00.

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, "All about 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] P. J. Marandi, C. E. B. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *ICDCS*, 2014.

[6] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *SRDS*, 2014.

[7] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *ATC*, 2013.

[8] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *SOSP*, 2009.

[10] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "Cheapbft: resource-efficient byzantine fault tolerance," in *Eurosys*, 2012.

[11] J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

[13] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

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

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

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

[17] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *SOSP*, 2011.

[18] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live-an engineering perspective (2006 invited talk)," in *PODC*, 2007.

[19] 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.

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

[21] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 243–254, 2011.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 45–58.

[23] A. Bessani, J. a. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with bft-smart," in *DSN*, 2014.

[24] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 4, p. 12, 2015.

[25] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults." in *NSDI*, vol. 9, 2009, pp. 153–168.

[26] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis *et al.*, "Zeno: Eventually consistent byzantine-fault tolerance." in *NSDI*, 2009.

[27] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[28] M. Castro, R. Rodrigues, and B. Liskov, "Base: Using abstraction to improve fault tolerance," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 236–269, 2003.

[29] R. Friedman and R. Van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *HPDC*, 1997.

[30] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with atomic multicast," in *Middleware*, 2014.

[31] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *SOSP*, 2013.

[32] S. Benz, L. Pacheco, and F. Pedone, "Stretching multi-ring paxos," in *ACM SAC*, 2015.

[33] O. Mendizabal, P. Jalili Marandi, F. L. Dotti, and F. Pedone, "Checkpointing in parallel state-machine replication," in *OPODIS*, 2014.

[34] Z. Guo, C. Hong, M. Yang, L. Zhou, L. Zhuang, and D. Zhou, "Rex: Replication at the speed of multi-core," in *Eurosys*, 2014.

[35] B. Kemme, A. Bartoli, and O. Babaoglu, "Online reconfiguration in replicated databases based on group communication," in *DSN*, 2001.

[36] R. Jimenez-Peris, M. Patino-Martinez, and G. Alonso, "Non-intrusive, parallel recovery of replicated data," in *SRDS*, 2002.

[37] W. Liang and B. Kemme, "Online recovery in cluster databases," in *EDBT*, 2008.

[38] L. Camargos, F. Pedone, A. Pilchin, and M. Wieloch, "On-demand recovery in middleware storage systems," in *SRDS*, 2010.