Recovery Algorithms for Paxos-based State Machine Replication

Jan Kończak, Paweł T. Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. *Member, IEEE,*

Abstract—In this article, we propose and evaluate three different state recovery algorithms aimed for Paxos—one of the most popular distributed agreement protocols. Paxos is commonly used to maintain consistency among state machine replicas despite of failures of processes. The first algorithm, that we call FullSS, originates from the original Paxos and requires that the system frequently uses stable storage during regular (non-faulty) execution. The other two state recovery algorithms, ViewSS and EpochSS, scarcely requite access to stable storage, and the recovering process must do much less work to restore its lost state, and to catch up on the current state of the system. We thoroughly analyze and compare the behavior of the three algorithms during state recovery and also during regular, non-faulty system execution, under various workloads (e.g., causing the network or CPU saturation). The experimental results show that by using ViewSS and EpochSS, we can significantly improve process recovery with respect to the original Paxos, if only it can be assumed that at any time a majority of replicas are up running (excluding those replicas that are just recovering). Moreover, these algorithms do not impact the performance of Paxos during regular (non-faulty) operation. However, FullSS is the only choice out of the three, if the system must tolerate catastrophic failures.

Index Terms—Distributed algorithms, Paxos, state machine replication, fault-tolerance.

1 Introduction

Replication is an important enabling technology for increasing service availability and performance. At the core of this approach are *distributed agreement* protocols that are used for maintaining consistency among state machine replicas. Paxos [1] is by far the most known protocol of this sort. It has been used in many commercial systems, e.g., in Chubby [2], for implementing distributed locks, and in Spanner [3], for implementing distributed transactions. However, so far some important practical problems of using Paxos were not researched sufficiently deeply. One of such problems is state recovery after failures. To guarantee service high availability, a crashed replica must recover state and become up-to-date as quickly as possible. Therefore, efficient recovery algorithms are required. In this paper, we study a range of such algorithms that, we claim, are suitable for the Paxos protocol.

In [1], describing Paxos, and related papers (e.g. [4]), some protocol data must be written to stable storage, so that a crashed replica is able to recover its state by reading the storage content at the start up. In [5], a snapshot-based recovery algorithm was discussed, but, as above, it frequently uses stable storage, which severely impacts system performance. To make the accesses faster, instead of disks, *solid state drives (SSD)* could be used, but the best alternative in terms of performance would be to minimize the use of *any* stable storage devices. In [6], the authors discuss "stable-storage free recovery", but in this approach the processes that recovered after crashes stop sending any new Paxos messages, which, as the authors admit, limits potential applications (see discussion in Section 2).

According to Brewer's CAP conjecture [7], it is impossible for a distributed system to simultaneously provide consistency, availability, and partition tolerance. Paxos is typically used in systems that guarantee consistency and availability. To guarantee the latter property, the majority of replicas must remain operational at any time. With such an assumption, we can propose efficient recovery algorithms, in which processes scarcely use stable storage, as they can contact other replicas during recovery to update their state. The question arises what are the minimal data that must be written to stable storage and how often they should be written, in order to be able to recover a process that crashed?

We study three different recovery algorithms for Paxos: FullSS, ViewSS, and EpochSS. FullSS follows the original paper on Paxos—the system frequently uses stable storage during regular (non-faulty) execution. ViewSS was adopted from viewstamped replication [8]. EpochSS is similar to ViewSS, but requires less access to stable storage. Given the majority of replicas is always up, it can support crash recovery with no use of stable storage during regular execution, which aids performance. Stable storage is only accessed by a process on its start up and recovery. To our best knowledge, EpochSS did not appear elsewhere in the literature on Paxos. All algorithms can deal with *stray messages*, i.e., messages that were sent but not delivered before a crash.

Our work is the first comprehensive experimental assessment and comparison of recovery techniques for Paxos. The obtained results show that a suitable support of recovery in Paxos can have negligible influence on system performance, both during normal (non-faulty) operation as well as during recovery. For this, we developed *JPaxos* [9]—a state machine

- *J. Kończak, P. T. Wojciechowski, and T. Żurkowski were with the Institute of Computing Science, Poznań University of Technology, 61-051 Poznań, Poland. E-mail: {Jan.Konczak, Pawel.T.Wojciechowski}@cs.put.edu.pl, Tomasz.Zurkowski@gmail.com*
- *N. Santos and A. Schiper were with School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland. E-mail: {Nuno.Santos, Andre.Schiper}@epfl.ch*

replication framework based on a highly optimized variant of Paxos, equipped with the three recovery algorithms and an efficient snapshotting mechanism.

The structure of the paper is as follows. After discussing related work (Sec. 2), we define a system model (Sec. 3) and the Paxos protocol (Sec. 4), giving details of the protocol that help to explain recovery. Then, we present three recovery algorithms for Paxos (Sec. 5). Next, we analyze the recovery process (Sec. 6), and present the results of the experimental evaluation (Sec. 7). We conclude in Sec. 8.

2 Related Work

In this section we describe related work which is closely related to ours.

Recovery in Paxos

De Prisco, Lampson, and Lynch [10] use a timed I/O automaton model to formally analyze Paxos. As in [1], they assume that, whenever required, the state is recorded to stable storage.

Boichat *et al.* [6] describe a simple method of recovery of a replica after crash, which requires Paxos to write data to stable storage, once per each decision. This is equivalent to FullSS. They also describe another recovery method, called *Winter*, that does not use stable storage at all, but it requires that the majority of processes never crash. Upon recovery, a replica broadcasts a message indicating that it recovered its state and it votes no more for any decision. This means that for three replicas, one is allowed to crash and recover its state (possibly many times) while the other two must never crash or the system becomes unavailable forever. Boichat *et al* assess that their Winter method is "not really useful for a practical system". In our work, we take weaker and much more practical assumptions about the system and only limit the number of processes that may crash at the same time (in case of ViewSS and EpochSS).

Kirsch and Amir [11] show the Paxos protocol with a simple state recovery method that uses stable storage at key places of the protocol. Contrary to us, no snapshots are recorded in order to decrease the memory and processing time required by recovery. They evaluated the performance of Paxos, considering no disk writes, as well as synchronous / asynchronous disk writes. The results show a large negative impact of stable storage (disks) on the system throughput. Therefore, in our work our focus was on reducing the use of disks to minimum while retaining practical assumptions about the system.

Rao, Shekita, and Tata [5] use Paxos to build a data store system with support of state recovery based on stable storage. To speed up recovery, a catch-up method with snapshots is used, as in our system. However, unlike us, at the end of the catch up phase, the leader momentarily blocks new writes to ensure that another process has fully caught up the current state. The authors compare the performance of the system with a *hard disk drive (HDD)* and a high-end *solid state drive (SSD)*. Unfortunately, they do not show the results with no recovery support, so it is hard to estimate the overhead of the recovery algorithms.

Many authors describing some prototype and industry-strength implementations of Paxos (see e.g., [2], [3], [4], [12], [13], [14]) and popularizing the Paxos algorithm (see e.g., [15], [16]) do not even mention, or only give some vague idea, about support for a crash-recovery model of failure. Moreover, some other authors use the term "recovery" in a completely different context. E.g., in [17], [18], "recovery" means, in fact, restoring system availability after the crash of a leader by electing a new leader. To sum up, despite a lot of interest in the Paxos protocol, little progress was made regarding support for state recovery after crashes. However, some state recovery methods have been developed for other protocols, which are similar to Paxos. Below we discuss the most relevant examples.

Recovery in Non-Paxos Consensus Protocols

Viewstamped replication [8], [19] is an efficient state machine replication protocol, similar in operation to Paxos. Oki and Liskov [8] describe an efficient state recovery method in VR, that that does not demand frequent accesses to stable storage. The algorithm requires little data to be stored permanently, and the writes occur sporadically. The VR algorithm also gives a clear description of how the state of late replicas is updated. Liskov and Cowling [19] propose a state recovery method that does not use stable storage at all, but extends the system assumptions, by putting restrictions on system asynchrony and clock behaviour. However, the authors do not explain how a replica joining the system learns whether it should recover from previous state, or start execution for the first time. In our case, we use epoch or view numbers for this, which have to be stored in stable storage to survive crashes.

Aguilera, Chen, and Toueg [20] proposed failure detectors aimed for the crash-recovery model, and determined under what conditions stable storage is necessary in order to solve consensus in this model. Based on the failure detectors, they proposed two consensus algorithms: one requires stable storage and the other does not. They show that stable storage is not needed to recover if and only if always-up replicas outnumber unstable or eventually-down replicas (as classified by their failure detectors). They also showed that if there are no replicas which are always-up, then recovery without frequent accesses to stable storage is not possible. Since assuming that some processes are always up is impractical, in our work we circumvent this impossibility result by restricting the number of simultaneous failures.

Ongaro and Ousterhout [21] described Raft, a consensus algorithm for managing a replicated log. The Raft algorithm uses a catch-up method that is similar to ours, but in order to support recovery all key data of the algorithm must be written to stable storage. Thus, their approach to state recovery does not bring significantly new ideas.

Junqueira, Reed and Serafini [18] described Zab, an atomic broadcast protocol for primary-backup systems that offers some support for recovery. However, the authors, in fact, use the term "recovery" for a correct initialization of a new leader once the previous one crashed. No information was given on how a crashed replica recovers its state.

Michael *et al.* [22] propose a new crash model called *Diskless Crash-Recovery* (DCR) and an algorithm for maintaining fault-tolerant shared objects. DCR is a crash-recovery model without stable storage, complemented by an oracle that generates locally unique identifiers and tells the process upon startup whether it has been started for the first time ever. No algorithm of the oracle is given, and there is no algorithm, which we are aware of, that provides such an oracle in asynchronous system without the use of stable storage. Similarly to our work, in DCR a majority of processes must be up at any time to enable liveness. Both ViewSS and EpochSS can operate with no stable storage if an oracle such as required by DCR were available (see Section 5.5 for explanation).

In [22], the authors also introduce the concept of *crash consistent quorums*, which assumes DCR and consists of primitives that update and read the state, with only one guarantee that if an update completes, than a read must observe it. The authors show how to build shared objects using the crash consistent quorum concept, providing sample pseudocodes for an atomic register and *virtual stable storage (VSS)*. By writing every received message to VSS, any protocol in the crash-stop model can be converted to a protocol in the crash-recovery model. The authors argue that this allows for a straightforward migration of Paxos to the DCR model. However, we expect the resulting system, even if thoroughly optimized, to be very inefficient. At least one write to VSS per command would be necessary, and writes to VSS take as much time as issuing a command in Paxos. With identical assumptions our EpochSS needs no writes to storage and gives the same guarantees.

In [22], the authors point out an error in the recovery protocol described in our early technical report on JPaxos [9], which has since then been corrected.

Optimization of State Recovery

In the state machine replication, to recover a replica from a crash, the recovery protocol must both ensure that the process does not violate any guarantees (safety and other), and that the state machine is able to process new requests, by updating its state to a sufficiently recent one. The latter is typically achieved by log and state transfer. In our paper, we focus on the algorithmic aspects of state recovery in the Paxos algorithm, abstracting from any concrete applications. Note that for some specific workloads the recovery time is dominated by the log and state transfer. Moreover, there are workloads for which creating periodically a state snapshot severely impacts the performance of a replica. Some authors, e.g., [23], [24], have recently proposed efficient solutions for preparing state snapshots and for transferring the log and state. The solutions that they propose are largely orthogonal to our work, and they can be deployed independently of the recovery algorithms proposed in our paper. Below we discuss this work in detail.

Bessani *et al.* [23] proposed a solution to optimize a system that supports recovery by writing all vital data to stable storage. They propose three techniques: sequential checkpointing, collaborative state transfer, and parallel logging. The first two techniques optimize creating state snapshots, and enable state transfer in a model with Byzantine faults, so they are orthogonal to the recovery algorithm. Parallel logging attempts to postpone and to batch synchronous writes in order to reduce their number and alleviate their latency by splitting writes into the invocation and completion actions, and using the time in-between these actions for regular processing. Parallel logging reduces the performance penalty of synchronous writes, achieving the system throughput close to the system in the crash-stop model, under workloads with 1kB and 4kB commands. We observe similar system throughput in case of the FullSS recovery algorithm with commands of 1kB (see Section 7.1). However, while performance gain achievable by parallel logging highly depends on system workload, the ViewSS and EpochSS algorithms, which we propose, retain system performance regardless of workload.

Mendizabal, Dotti and Pedone [24] focus on system recovery in the *Parallel SMR (PSMR)*. PSMR is a variant of the state machine replication, where dependencies among commands are known a priori, so any two commands known to be independent need neither to be delivered, nor executed in order. To support system recovery, all vital data must be written to stable storage. The novel idea is to let the recovering replica execute new commands before the state of the replica gets fully updated. This is possible as long as the new commands are independent of the missing ones. They also divide the state snapshot into segments that can be installed independently, thus to execute a new command the recovering process needs to fetch only the segments that contain all dependencies of the command. The solutions presented in [24] are tailored for PSMR, and none of them can be used to improve system recovery in the classical state machine replication approach.

Reconfiguration and Recovery

Reconfiguration [25], [26], [27], [28] can also be used for retaining availability despite crashes. *Reconfiguration* is an action that changes the set of processes P (we assume P to be constant). It can be used to dynamically select the crash resilience level by changing the number of replicas, as well as to remove a crashed replica from P and add a newly started replica to P. In the latter case, reconfiguration is initiated upon suspecting a crash of a process, either by some replica or by an external component. This is problematic, especially in unstable periods, as false suspicions can lead to removal of correct replicas from P. When a dynamic set of processes is used, the clients must be supplied with a mechanism for locating replicas that are currently active. Typically, Paxos with the reconfiguration support requires a majority of replicas (of the current configuration) to stay alive, similarly to ViewSS and EpochSS.

The classical approach to reconfiguration in SMR is to use a dedicated SMR command [25], [26]. Recently some efforts were made to support reconfiguration without the need for consensus [27], [28]. The *Replacement* algorithm [27] is especially relevant here, as it is dedicated to use reconfiguration for handling failures. Unlike typical reconfiguration, replacement does not support changing the size of P—it only allows to replace a (suspected to crash) replica with a new one. Thus, a replica and its subsequent replacements can retain the replica identifier, but each replacement has a new replica *version*. To support these versions in Paxos, the authors extend Paxos to Version Paxos by adding versions to all messages, sending a vector of versions in the Prepare and Propose messages (see Section 4.1), and verifying the versions in majority checks. Version Paxos is 3% slower than Paxos (although the authors speculate that this slowdown can be reduced). In contrast, ViewSS and

EpochSS have no impact on Paxos performance. While the impact of a false suspicion on performance is reduced in [27] compared to the classical reconfiguration, it still requires initializing the state of the replaced replica.

3 System Model

We consider an asynchronous distributed system in which processes located at nodes communicate by exchanging messages. We make no assumptions on the time it takes to deliver a message, or on relative process speeds. Every two processes are connected through a fairloss communication link that can fail by dropping messages.[1] Processes can fail by crashing and may subsequently recover. We assume the existence of a failure detector of class $\diamond\mathcal{W}$ [29]. We assume no Byzantine failures. When a process crashes it loses all of its state, but it may use local *stable storage* to save (and retrieve upon recovery) some part of its state.

We use p, q to denote processes. P is a set of all processes. Q is a majority set of processes in P. Let $F(t)$ be the set of processes in P that are not functioning at time t. We say a process p is *up at time* t if $p \notin F(t)$ and p is *down at time* t if $p \in F(t)$. We say that process p *crashes* at time t if p is up at time $t-1$ and down at time t. When all processes are down at time t, there is a *catastrophic failure*. Process p is *recovered* at time t ($t \geq 1$) if p was down at time $t-i$ and up at time t (for some $i > 0$). When a process crashes, it loses all its volatile state, but when it is restarted, it should learn (as part of the recovery algorithm) that it has to commence steps to recover. A *recovering process* is the one that executes a recovery algorithm, but is not recovered yet.

We define a *crash resilience level*, denoted R, as the upper bound on the number of replicas that can crash simultaneously without impeding recovery. E.g., R is equal to n for FullSS and $\lfloor \frac{n-1}{2} \rfloor$ for ViewSS and EpochSS, where n is the total number of all replicas.

A *state machine* [30] consists of a set of states, a set of commands, a set of responses, and a functor that assigns a response/state pair to each command/state pair. A state machine executes a command by changing its state and producing a response, with the command and the machine's current state determining its new state and its response. A *distributed computing system* consists of several processes (*replicas*) that are connected by a network. The processes are replicated and synchronized by having every process independently simulate the execution of the same state machine. The state machine is tailored to the particular application, and is implemented by a general algorithm for simulating an arbitrary state machine, such as Paxos, which handles the problems of synchronization and fault tolerance.

4 The Paxos Protocol

4.1 Overview of Paxos

In this section, we summarize the original *Paxos* in [1] and [31], to aid in understanding of the recovery algorithms. Each state machine command is chosen through a series of numbered *ballots*, where each ballot is a referendum on a single command. One of the processes is designated as a *leader*; it sends ballots with the commands proposed by clients to the other processes (called *followers*). In each ballot, a process has the choice of either voting for the proposed command or not voting. A process does not vote if it has already voted in a higher ballot. Obviously, a crashed process does not vote, too. In order for a ballot to succeed and a command to be *issued for execution* (or *issued*, in short), a *majority set* of the processes must vote for it. Otherwise, another ballot has to be conducted. Thus, a single command can be voted in several ballots. Note that the majority sets voting on any two ballots will have at least one process in common. So, any command which has been issued will appear in the store of at least one process of any majority set participating in a subsequent ballot.

The protocol allows a leader to conduct any number of ballots concurrently by running a separate instance of the protocol for each command number. The protocol instances are numbered in turn, and a command issued within instance i has the number i. The issued commands must be executed by each state machine according to their numbers. When a new leader is chosen, messages are exchanged between the new leader and the other processes in the system to ensure that each of the processes has all of the commands that the other processes have. As part of this procedure, any command for which one of the processes has previously voted but does not have a command number is broadcast as a proposed command in a new ballot.

We use the following notation: b, b' are ballot numbers (partitioned among the processes: $b = (k, p)$, where k is an integer and p denotes a process); i, j are command numbers; c, d are commands; d_i is the ith command; $v_i = (q, b, d_i)$ is a *vote* cast by process q for command d_i in ballot number b.

The following variables are used[2]: $lastTried[p]$ is the number of the last ballot that p tried to begin, or -∞ if there was none, $nextBal[q]$ is the number of the last ballot in which q agreed to participate, or -∞ if it has never agreed to participate in a ballot, $prevBal[q, i]$ is the number of the last ballot in which q voted for command number i, or -∞ if it never voted, $prevDec[q, i]$ is the command for number i for which q last voted, or $\perp$ if q never voted.

The following messages are used: Prepare(b,j) is a message sent by a leader with a new ballot number b for command number j; PrepareOK(b, D, V, X) is a message sent in ballot b to the leader by a process q with q's knowledge on commands: *decided* (D), *voted* (V), and *missing* (X); Propose(i, b, d) is a message sent by a leader to begin a ballot; Voted(i, b) is a message sent by a process to indicate its vote; and Success(i, d) is a message indicating that command d has been issued for number i.

Below are the protocol steps (executed atomically):

Phase 1: Ballot initialization

1. A leader p chooses a new ballot number b greater than $lastTried[p]$, sets $lastTried[p]$ to b, and sends a Prepare(b, j) message to all processes including itself, where j is the smallest number for which p does not know a command.

2. Upon receipt of Prepare(b, j) from p with $b \geq nextBal[q]$, process q sets $nextBal[q]$ to b and sends PrepareOK(b, D, V, X) to p, where D is the set of pairs (i, d_i) with $i \geq j$ s.t. q knew that d_i was a command

1. For brevity, we present recovery algorithms assuming reliable links which can be implemented on top of such unreliable links.

2. In [1], their values are kept with commands in stable storage.

issued with number i; V is the set of pairs (i, v_i) s.t.: (i) $i \geq j$, (ii) q does not know a command with number i, (iii) q has voted in a ballot for command number i, and (iv) $v_i = (q, prevBal[q, i], prevDec[q, i])$ is the most recent vote cast by q for number i; X is the set of numbers $< j$ for which q does not know the corresponding command.

3. After receiving PrepareOK(b, D, V, X) from every process in some majority set, where $b = lastTried[p]$, the leader p adds the commands from each set D to its list of issued commands. The leader also sends the other processes any commands it knows but they do not according to X in their PrepareOK messages.

For every V and every vote v_i s.t. $(i, v_i) \in V$, the leader p executes ballot b to broadcast a command d for number i, as follows: a) p chooses some majority set Q of processes from among those from which it has received the PrepareOK messages for b. If any of those processes have voted in any ballot for command number i, then d must equal the command in the latest ballot for which such a vote was cast, else d can equal any command. b) p executes 3′ with Q. If there is a gap between numbers of commands decided or voted for, then the leader process p tries to issue "no-op" commands for these numbers.

Phase 2: Voting

3′. Leader p *proposes* d: p sends a Propose(i, b, d) message to every process in Q, where $b = lastTried[p]$. [3]

4. Upon receipt of Propose(i, b, d) from leader p with $b = nextBal[q]$, process q casts its vote in ballot number b for the command d, sets $prevBal[q, i]$ to b and $prevDec[q, i]$ to d, and sends Voted(i, b) back to p.

5. When leader p has received Voted(i, b) from every process in Q (the quorum for ballot number b), where $b = lastTried[p]$, then it considers d (the command of that ballot) to be successfully broadcast (or *decided*) and sends a Success(i, d) message to every process.

6. Upon receiving a Success(i, d) message, q *issues* d_i.

A single leader is selected for all command numbers. To decide a command, it chooses the lowest command number i that it is still free to choose, and executes Phase 2. If the leader is suspected to have crashed, a new leader is elected. It then executes Phase 1. If the suspicion was wrong there can be more than one leader until stale leaders cease to execute. [4] A new leader p may not know the current ballot number. If q received Prepare(b, j) or Propose(i, b, d) from p with $b < nextBal[q]$, then it sends $nextBal[q]$ to p. Then, p initiates a new ballot with a larger ballot number.

4.2 Our JPaxos Protocol

To conduct an experimental evaluation, we developed JPaxos [9]—a SMR tool based on Paxos, equipped with a choice of our recovery algorithms and the catch-up protocol (described in Section 4.3). The source code of JPaxos in Java is publicly available [33]. In JPaxos, to issue consecutively numbered commands for execution, successive instances of Paxos are launched, where several consecutive instances are run in parallel (see [32]). Paxos consists of two phases: ballot initialization and voting. The first phase, which essentially establishes a new leader, is conducted for all instances at once. The second phase, which is an attempt to agree upon a command proposed by the leader, is run separately for any instance i. In the first phase, the Prepare and PrepareOK messages are used. In the second phase, the Propose and Accept messages are used, and collectively called *votes*. The Prepare, PrepareOK, and Propose messages correspond to analogous messages in Paxos in Section 4.1, but some arguments are eluded. Moreover, the Accept message corresponds to Voted, but Voted is sent to the leader in [1], while in JPaxos Accept is broadcast, hence the Success message is no longer needed.

Algorithm 1 The simplified pseudocode of JPaxos

```
 1: Initialization:
 2:    procId                  {a unique, non-zero process identifier}
 3:    prevBal[i] ← (0,0)
 4:    prevDec[i] ← ⊥
 5:    currBal ← (0,0)
 6:    isLeader ← false
 7: procedure ProposeCommand(i, d)
                              enabled when isLeader and prevDec[i] = ⊥
 8:    prevBal[i] ← currBal
 9:    prevDec[i] ← d
10:    send(𝒫, Propose⟨i, currBal, d⟩)
11: procedure BecomeLeader()
12:    isLeader ← false
13:    instanceList ← { i | no command issued for i }
14:    currBal ← (k, procId) s.t. (k, procId) > currBal          {k ∈ Int.}
15:    send(𝒫, Prepare⟨currBal, instanceList⟩)
16: upon Prepare⟨$bal_q$, instanceList⟩ from q s.t. $bal_q$ ≥ currBal
17:    if q ≠ procId then isLeader ← false
18:    currBal ← $bal_q$
19:    prepInst ← ø
20:    for all i ∈ instanceList s.t. prevDec[i] ≠ ⊥ do
21:       prepInst ← prepInst ∪ { (i, prevBal[i], prevDec[i]) }
22:    send(q, PrepareOK⟨currBal, prepInst⟩)
23: upon PrepareOK⟨$bal_q$, $prepInst_q$⟩ from 𝒬 s.t.
                              $bal_q$ = currBal and not isLeader
24:    for all (i,_,_) ∈ $prepInst_q$ s.t. $prepInst_q$ delivered do
25:       prevBal[i] ← max({ $b_i$ | (i, $b_i$, _) ∈ $prepInst_q$ })
26:       prevDec[i] ← d s.t. (i, prevBal[i], d) ∈ $prepInst_q$
27:       send(𝒫, Propose⟨i, currBal, prevDec[i]⟩)
28:    isLeader ← true
29: upon Propose⟨i, $bal_q$, d⟩ from q s.t. $bal_q$ = currBal
30:    prevBal[i] ← $bal_q$
31:    prevDec[i] ← d
32:    send(𝒫, Accept⟨i, $bal_q$⟩)
33: upon Accept⟨i, $bal_q$⟩ from 𝒬 s.t. all i are equal and currBal = $bal_q$
                                                    and prevBal[i] = $bal_q$
34:    if no command issued for i then issue prevDec[i] in instance i
```

In Algorithm 1, we show the pseudocode of the JPaxos protocol. For simplicity, we omitted some optimizations. We use the following notation: send(p, m) sends m to process p, send$(\mathcal{P}, m)$ broadcasts m to processes in $\mathcal{P}$, and max(S) returns the largest element in set S. *prevBal*[*i*] is the number of the last ballot in which the process voted for command number i, or $(0, 0)$ if it never voted. *prevDec*[*i*] is the command for number i for which the process last voted, or $\perp$ if it never voted. *currBal* is the most recent ballot number known to the process (*currBal* replaces *lastTried* and *nextBal* from Section 4.1). *isLeader* is true if and only if the process is eligible to propose new commands. Proposing "no-ops" (in place of gaps) by a newly selected leader is not depicted.

In JPaxos only the leader can propose commands. If a

3. This step can be combined with step 5 of the previous command. Also, a single Propose message can be used for a batch of commands (as in [32]).

4. If suspicions were arisen constantly, Paxos could never decide. To ensure progress, it is sufficient to use the $\diamond\mathcal{W}$ failure detector [29].

process suspects that the current leader crashed, it starts the ballot initialization phase to become a new leader. The clients are allowed to send commands to any replica. If a replica other than the leader (a follower) receives a command from a client, then it forwards the commands to the leader. The leader in JPaxos *batches* commands, that is instead of proposing a single command, a list of commands is proposed. This retains high throughput with small commands [32].

Each replica remembers a ballot number. When a process p starts the ballot initialization phase (line 11), it chooses a ballot number b greater than its current ballot number (*currBal*) and sends a Prepare message to all (line 15). The Prepare message carries b and the list of all instances for which no commands were issued by p (line 13). A process q that receives a Prepare for a ballot number b greater or equal to its current ballot number (line 16), first sets its *currBal* to b and then sends a PrepareOK back to p. In PrepareOK, q sends the most recent vote (if any) for each instance indicated by Prepare (line 21). (In Section 4.1, PrepareOK carried also data that were used by the leader to update the state of the followers; this is replaced by the catch-up protocol in JPaxos.) When p receives the PrepareOK messages from a majority of processes (including itself, line 23), then p becomes a leader. p must now propose a command for any instance i such that p did not issue a command in i, but some process sent a vote for a command in i (line 27). In these instances, p must propose, in each i, the command which was most recently voted for in i.

To propose a command in instance i, the leader p sends to all a Propose message with p's current ballot number, instance number i, and the proposed command c (line 10). A process q that receives Propose for a ballot number equal to q's current ballot number, records that it cast a vote for c in instance i and ballot b (lines 30-31), and sends an Accept message to all. The Accept message contains only i and b. Any process that receives Propose from the leader and votes (a vote is either Accept or Propose) from a majority, all for instance i in ballot b (line 33), issues the command c in instance i. In Section 4.1, Voted is sent only to the leader, and, after receiving the majority of votes, the leader sends a Success message with i and c (and without b) to all. JPaxos broadcasts the Accept, and since in a given instance i and ballot b only a single command c can be proposed, then gathering both Propose and a majority of votes supersedes the Success message.

4.3 The Catch-Up Protocol

In Paxos, all non-faulty processes must eventually issue the same sequence of commands. To get any missing commands, after waiting a sufficiently long time, a stale process can propose itself as a new leader, and execute Phase 1. But frequent leader changes are inefficient and thus should be avoided. Therefore, in JPaxos we implemented the *catch-up protocol*, which is used by a stale replica to learn about any missing commands from other replicas (see also, e.g. [5], [11], [34]). The protocol allows to simplify the Paxos algorithm described in Section 4.1, and is also used by our recovery algorithms. A similar idea appeared in [1] as *long ledgers*, but no algorithmic details were given. Below we briefly describe the catch-up protocol.

Algorithm 2 The catch-up protocol
```
 1: issued($i$) returns true iff some command issued for $i$
 2: $maxInst \leftarrow 0$
 3: $lastSnap$ = [
      $state \leftarrow$ state machine and Paxos protocol state
      $i \leftarrow \max(\{ i \mid \forall j \leq i$, some $d_j$ issued in $state \})$
    ]
 4: procedure updateShapshot($newSnap$)
 5:   $lastSnap \leftarrow newSnap$
 6: procedure startCatchUp($highestInst$, $p$)
 7:   $missing \leftarrow \{ i \mid$ not issued($i$) and $i \leq highestInst \}$
 8:   $maxInst \leftarrow \max(\{maxInst, highestInst\})$
 9:   send($p$, CatchUpQuery$\langle missing \rangle$)
10: upon CatchUpQuery$\langle missing_q \rangle$ from $q$
11:   if $\exists\, i \in missing_q : i \leq lastSnap.i$ then
12:     send($q$, CatchUpSnapshot$\langle lastSnap \rangle$)
13:     $missing_q \leftarrow missing_q \setminus \{ i \mid i \leq lastSnap.i \}$
14:   $log \leftarrow \varnothing$
15:   for all $i \in missing_q$ s.t. issued($i$) do
16:     $log \leftarrow log \cup \{ (i, prevDec[i]) \}$
17:   send($q$, CatchUpResponse$\langle currBal, log \rangle$)
18: upon CatchUpSnapshot$\langle lastSnap_q \rangle$ from $q$
                        s.t. $\exists\, i : ( i \leq lastSnap_q.i$ and not issued($i$) )
19:   restore state from $lastSnap_q.state$
20:   $lastSnap \leftarrow lastSnap_q$
21:   if $lastSnap_q.i \geq maxInst$ then CatchUp completed
22: upon CatchUpResponse$\langle currBal_q, log_q \rangle$ from $q$
23:   $currBal \leftarrow \max(\{currBal, currBal_q\})$
24:   for all ($i$, $d$) s.t. ($i$, $d$) $\in log_q$ and not issued($i$) do
25:     $prevBal[i] \leftarrow currBal_q$
26:     $prevDec[i] \leftarrow d$
27:     issue $d$ for number $i$
28:   if $\forall i \leq maxInst$, issued($i$) then
29:     CatchUp completed
30:   else
31:     $(\_, p) \leftarrow currBal$                    {$p$ is a leader}
32:     startCatchUp($maxInst$, $p$)
```

In JPaxos, a process queries an arbitrary process for the missing commands if it gets a message for a much higher command number than expected or it does not receive any messages for some time (and a leader is not suspected to be faulty). Also, it has to execute all missing commands (in the order of their numbers) before it is allowed to execute any new commands. Since there can be many missing commands (e.g. due to a network split), every process periodically creates a *snapshot* of its local state in the main memory. Then, a stale process receives a snapshot, a *log* that includes the issued commands whose effects are not yet included in the current snapshot, and the ballot number. All these data can then be used by a stale replica to restore the current state on its own node. By using a snapshot, the state recreation is efficient and the log is bounded.

In JPaxos, the catch-up protocol is also used to support recovery. In case of ViewSS and EpochSS, the protocol is the only mean used by a recovering process to get to know all that occurred before a crash, which is necessary for the process to become *correct*, i.e., to keep promises defined in the next section. In all our recovery algorithms, the catch-up protocol is also used by a recovering process to get to know all that occurred until now (to issue all the commands decided so far, so that any upcoming commands can be issued). In Section 6, we discuss how the catch-up protocol impacts the system performance.

In Algorithm 2, we present the pseudocode of the catch-up protocol that is used by the JPaxos protocol and the recovery algorithms. To start catch-up a process calls start-

CatchUp(*highestInst*, p), passing as arguments *highestInst* and ID of a process p to which the CatchUpQuery request will be sent (line 9). *highestInst* is the latest instance number that the process had learned either from the RecoveryAck, Propose, Accept, or Prepare messages, or *highestInst* is piggybacked in the messages of the JPaxos failure detector. The RecoveryAck message will be explained in Section 5. The CatchUpQuery request is initially sent to any process p that is *not* a Paxos leader. In response, p simply returns requested data, i.e., a snapshot of state, a log of commands, and the ballot number (lines 10–17). A follower is queried, as the leader is the most busy process in Paxos, and thus assigning an extra task to it may decrease the system performance. If the snapshot and the log received from the follower do not represent a complete state up to *highestInst* (line 28), the leader is queried (line 32).

During normal system operation, procedure updateShapshot is called periodically to record the current snapshot of state, passing a fresh snapshot *newSnap* as the argument. Creating snapshots not only allows to prevent the log of commands from growing, but it also speeds up the catch-up process.

The actual implementation of the catch-up protocol depends on the concrete recovery algorithm, as follows. If FullSS is in use, *lastSnap* must be written in stable storage, and the effects of lines 23, 25 and 26 must also be stored in stable storage. If ViewSS is in use, only the new ballot number in line 23 must be written to stable storage. In case of EpochSS, the catch-up protocol does not write to stable storage at all.

5 Recovery Algorithms

Upon startup a process must check if it has never run before or it recovers from a crash. If it is the first run of the process, then the process just begins the normal operation and takes part in Paxos. Otherwise, the process must enter a *recovery phase*, during which it executes the recovery algorithm. During the recovery phase, a recovering process cannot actively take part in Paxos, i.e., the recovering process is neither allowed to vote (i.e., to reply to Propose messages) nor to take part in the leader election (i.e., to reply to Prepare messages) until it has updated its state, but it can deliver and process these messages.

By referring to Algorithm 1, to ensure safety of Paxos, processes make two promises: (1) when a process p sends PrepareOK in ballot b (line 22), then p promises not to reply to messages in ballots older than b, and (2) when a process p sends Accept for ith command d in ballot b (line 32), it promises to reply to any forthcoming Prepare messages for command number i with command d and ballot number b.

A recovering process must also not break these promises. One solution is to never forget them by writing each promise in stable storage (as in FullSS). Another solution relies on the knowledge obtained from other processes, which requires much less stable storage accesses. However, querying another process for the promises made before a crash is not sufficient since the process may not know all of them—some messages carrying promises made before a crash may still be in transit. We call them *stray messages*.

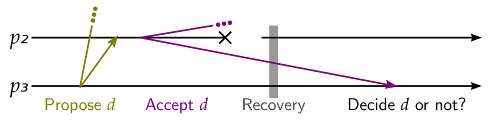

Fig. 1. Safety violation caused by a stray message Accept.

In Figure 1, we show an example of a potential safety violation caused by a stray message. A process p_2 broadcasts a message Accept and crashes. After p_2 recovers, a process p_3 delivers a stray message Accept. If p_3 does not drop it, nor p_2 learns during recovery that it might have sent it, then p_2 and p_1 (not shown) can decide differently than p_3.

To prevent this undesired behavior, the ViewSS and EpochSS recovery algorithms include a mechanism to either discard all stray messages (ViewSS), or make them harmless (EpochSS). The mechanisms are explained, respectively, in the beginning of Section 5.2 and in Section 5.3.

5.1 The FullSS Algorithm

The *Full Stable Storage (FullSS)* algorithm follows the idea in [1], i.e., all processes executing the Paxos protocol synchronously write all critical data to stable storage. A leader writes the new ballot number before sending Prepare (in line 15 of Algorithm 1). Followers write the new ballot number before sending PrepareOK (in line 22). A process writes the command it voted for and the current ballot number before sending Accept (in line 32). Note that recording less data would not allow the system to recover after a catastrophic failure. On recovery after a crash, the recovering process retrieves data from stable storage, restores its state and the log, and joins Paxos. Other processes are not involved in recovery.

If it can be assumed that at any time a majority of replicas are up (excluding those that are just recovering), then other algorithms can be proposed to recover the system. Two such algorithms are described in the following sections. They do not tolerate catastrophic failures but increase the overall system performance.

5.2 The ViewSS Algorithm

In Paxos, a ballot number changes when a new leader is elected. The *View Stable Storage (ViewSS)* algorithm, presented in Algorithm 3, enforces incrementing a ballot number also during a recovery phase, which will allow processes to discard any stray messages, as they will carry an older ballot number (from before the crash). For this to work, all messages are labelled with the current ballot number (which is already required by Paxos) and a process must write a ballot number to stable storage before it can modify it, so that after the process has crashed and restarted it can learn the ballot number from before the crash. To guarantee progress, at any time a majority of processes must be up and not recovering.

The ViewSS algorithm involves no changes to the original Paxos, just extends it with a recovery phase, and requires that the new ballot number is (synchronously) written to stable storage on leader change (lines 14′ and 18″). If a leader does not crash and the network is stable, the system

Algorithm 3 View-based recovery

Notation: Line XX replaces line XX in Algorithm 1.
Lines XX′, XX″, … are inserted after line XX in Algorithm 1.

```
        ...
14′ : write $currBal$ to stable storage
        ...
18  : if $bal_q \neq currBal$ then
18′ :     $currBal \leftarrow bal_q$
18″ :     write $currBal$ to stable storage
        ...
35  : upon start
36  :     read $currBal$ from stable storage
37  :     if $currBal \neq (0,0)$ then
38  :         send($\mathcal{P} \setminus procId$, Recovery$\langle currBal \rangle$)
39  :         wait for RecoveryAck$\langle bal_q, bal_p, highestInst \rangle$ from $\mathcal{Q}$ s.t.
                  $bal_p = currBal$ and $(i, q_l) = bal_q$ delivered from $q_l$
                  where $(i, q_l) = \max(\{bal_q \mid bal_q \text{ delivered}\})$
40  :         $currBal \leftarrow \max(\{bal_q \mid bal_q \text{ delivered}\})$
41  :         catch up to max({$highestInst$ | $highestInst$ delivered})
42  : upon Recovery$\langle bal_q \rangle$ from $q$
43  :     if $bal_q \geq currBal$ then
44  :         call procedure BecomeLeader()
45  :         wait until either $isLeader$ or $currBal$ changes
46  :     $highestInst \leftarrow \max(\{\, i \mid prevDec[i] \neq \bot \,\})$
47  :     send($q$, RecoveryAck$\langle currBal, bal_q, highestInst \rangle$)
```

Algorithm 4 Epoch-based recovery

```
        ...
6′  : $epoch \leftarrow [0, 0, \ldots, 0]$                    {of size $|\mathcal{P}|$}
        ...
22  : send($q$, PrepareOK$\langle currBal, epoch, prepInst \rangle$)
23  : upon PrepareOK$\langle bal_q, epoch_q, prepInst_q \rangle$ from $\mathcal{Q}$ s.t.
          $bal_q = currBal$ and not $isLeader$
          and $epoch_q[q] = \max(\{epoch[q]\} \cup epoch_{\mathcal{Q}}[q])$ where
          $epoch_{\mathcal{Q}}[p] = \{\, epoch_q[p] \mid epoch_q \text{ delivered from } q \in \mathcal{Q} \,\}$
23′ :     for all $p \in \mathcal{P}$ do $epoch[p] \leftarrow \max(\{epoch[p]\} \cup epoch_{\mathcal{Q}}[p])$
        ...
35  : upon start
36  :     read $epoch[procId]$ from stable storage
37  :     $epoch[procId] \leftarrow epoch[procId] + 1$
38  :     write $epoch[procId]$ to stable storage
39  :     if $epoch[procId] \neq 1$ then
40  :         send($\mathcal{P} \setminus procId$, Recovery$\langle epoch[procId] \rangle$)
41  :         wait for RecoveryAck$\langle bal_q, epoch_q, highestInst \rangle$ from $\mathcal{Q}$ s.t.
                  $epoch_q[procId] = epoch[procId]$
                  and $epoch_q[q] = \max(epoch_{\mathcal{Q}}[q])$
                  and $(i, q_l) = bal_q$ delivered from $q_l$
                  where $(i, q_l) = \max(\{bal_q \mid bal_q \text{ delivered}\})$ and
                  $epoch_{\mathcal{Q}}[p] = \{\, epoch_q[p] \mid epoch_q \text{ delivered from } q \in \mathcal{Q} \,\}$
42  :         $currBal \leftarrow \max(\{bal_q \mid bal_q \text{ delivered}\})$
43  :         for all $p \in \mathcal{P}$ do $epoch[p] \leftarrow \max(epoch_{\mathcal{Q}}[p])$
44  :         catch up to max({$highestInst$ | $highestInst$ delivered})
45  : upon Recovery$\langle epoch_q \rangle$ from $q$ s.t. $epoch_q \geq epoch[q]$
46  :     $epoch[q] \leftarrow epoch_q$
47  :     if $currBal = (\_, q)$ then
48  :         call procedure BecomeLeader()
49  :         wait until either $isLeader$ or $currBal$ changes
50  :     $highestInst \leftarrow \max(\{\, i \mid prevDec[i] \neq \bot \,\})$
51  :     send($q$, RecoveryAck$\langle currBal, epoch, highestInst \rangle$)
```

performance is identical as in the crash-stop model. In case of frequent leader changes, the system performance will decrease compared to the crash-stop model. But the impact of a (single) synchronous write to stable storage is negligible, as the leader change itself drastically degrades performance.

The recovery phase starts with a recovering process reading the latest ballot number written to stable storage before crash (line 36). Then, the recovering process sends the Recovery message to all processes and waits for the RecoveryAck messages from the majority. Regardless of the number of responses, the process must also wait for a reply from the leader of the largest ballot number seen in the RecoveryAck messages delivered so far (line 39). This is crucial, since that leader may be the only process that is aware of voting in which the recovering process sent a stray message prior to crashing. If a replica holds a ballot number that is not greater than the ballot number it has received in the Recovery message, then it initiates a ballot change before responding. Once the recovering process gathers replies, it uses the catch-up protocol to update its state, including all issued commands, up to the command that has the highest number recorded in RecoveryAcks (line 41). Then, the process is recovered and can join the Paxos protocol.

In ViewSS, the Recovery message consists of the process identifier and the ballot number read from the stable storage. The ballot number in the stable storage is incremented, among others, upon gathering RecoveryAck from a majority of processes. It is possible that a process starts a recovery procedure r with ballot number b in its stable storage, then crashes before completing r and starts a recovery procedure r', again with b in its stable storage. In such case, the process sends identical Recovery messages in r and r', and thus it cannot tell apart RecoveryAck for requests in r and r'. While it may seem erroneous, such case does not violate correctness. It is possible *iff* the process did not finish the recovery procedure r, and so was not able to send any Paxos protocol messages. ViewSS requires only that the current ballot number after the recovery completes is greater than any ballot number in which the recovering process could have sent a Paxos protocol message, and this requirement still holds. On the contrary, in EpochSS, upon each recovery the Recovery message is unique, as it contains the epoch number which is incremented upon each recovery.

5.3 The EpochSS Algorithm

The *Epoch Stable Storage (EpochSS)* algorithm requires that every process stores in stable storage an *epoch number* (initially equal 0), incremented every time the process restarts. So, the number tells how many times the process started recovering. Processes piggyback their epoch numbers onto the PrepareOK, Recovery and RecoveryAck messages, which makes it possible to recognize and ignore any stray messages sent in Phase 1. To make any stray messages of Phase 2 harmless, the recovering process first has to learn the highest command number it could have known prior to the crash, then to wait until it learns all commands up to this number.

Contrary to ViewSS, the EpochSS protocol requires only one synchronous write to stable storage per fault-free run of a process. Moreover, there are no redundant view changes. Like ViewSS, it tolerates at most a minority of processes to be down or recovering at the same time. If a majority of replicas simultaneously crash or are recovering, the system becomes unable to process client requests and unable to recover a crashed replica.

We present the pseudocode of EpochSS in Algorithm 4, where *epoch* is a vector of epoch numbers known by a process. A recovering process first broadcasts the Recovery message to all replicas (line 40). It then waits for replies from the majority, including a reply from the leader (line

41). From the leader's RecoveryAck, the recovering process learns the highest command number i processed by the system. In order to make harmless any stray messages of Phase 2, the recovering process uses the catch-up protocol to fetch the snapshot and all missing commands (as indicated by i) from other replicas (line 44). When all the necessary data are transferred, the process can join Paxos.

Phase 2 of Paxos (voting), where each process spends most of the life, is unaltered. However, contrary to ViewSS, Phase 1 (ballot initialization) requires some changes to be used with EpochSS. Firstly, the PrepareOK message must also carry $epoch_p$ (line 22). Secondly, a process initiating a new ballot (i.e., a new leader) must reject any stray PrepareOK message m once it learns (based on the epoch numbers it got from the Recovery and PrepareOK messages) that m was sent before recovery of its sender (line 23). From the performance point of view, the changes brought by EpochSS to Paxos have a minor impact, affecting only the leader election by slightly increasing the size of the messages exchanged at the leader election.

5.4 Stray Recovery Messages

Before (or after) a recovering replica fully recovers it may crash again. So, a recovering process may receive recovery messages originated from the previous recovery rounds. Moreover, if recovery messages were retransmitted, some of them can be delivered before crash, while some other can be delivered after crash. All that messages, called *stray recovery messages*, are undesirable, so they should be detected and ignored. For this purpose, all critical messages, including Recovery and RecoveryAck, carry a number, which is a ballot number in the ViewSS algorithm, and an epoch number (or a vector of epoch numbers) in EpochSS. This number is guaranteed to be unique per recovery round of each process, with an exception that we described in Section 5.2.

5.5 Stable-Storage-Free Recovery

The goal of any recovery algorithm is to put the recovering replica into a state of readiness for operation. Of course, this only occurs if the replica had indeed crashed, otherwise the replica is a fresh process. Therefore, a replica must discover if it ever run before (i.e., it was involved in rounds of Paxos), or it is a fresh process. In the former case, the recovery algorithm helps to overcome amnesia of a recovering replica and deals with any stray messages ("from the past"), while in the latter case no special action is necessary. If we assume that stable storage is available, then this problem can be easily solved, as we showed before, i.e., a process can write to stable storage any critical data that is necessary to "refresh memory", and on recovery, it can read the data and to learn its current status this way.

The question arises if process recovery can be provided without resorting to stable storage at all? Essentially, any such solutions can be boiled down to creating a fresh, unique ID by a replica on every boot-up, which is then piggybacked on every critical message, where a fresh process is assigned a "null" ID which indicates that the process does not recover. Based on this ID, other processes can make a right action and help the process to recover its state, if necessary. If generating such an ID were possible, we could modify our algorithms, for instance EpochSS, so that it does not require stable storage. EpochSS ensures that in every majority of processes, at least one process at any time knows the epoch number for any process that is up and is not recovering. Thus, a replica before starting the recovery could query a majority about its epoch number, using a fresh ID to tell apart responses for its query. Then, it could start the normal EpochSS recovery algorithm. So, if generating such an ID were possible, process recovery could be provided without resorting to stable storage at all. However, we have to reject these solutions on the grounds of assumptions that we made in this paper, as explained below.

We assumed that the network can arbitrary delay and duplicate the messages, so it cannot be a source of unique IDs. A globally unique ID could be created using a generator of random numbers, which is, however, unsatisfactory, as it gives only a probabilistic guarantee of uniqueness. Another solution is to use a system clock, as in Viewstamped Replication [19]. However, in Paxos and our system model, we assume an asynchronous system, and no guarantees on clock synchronization. In particular, there is no guarantee that after replica boots up the clock advances with respect to the past.

6 Recovery Process

So far we described how a replica recovers, but left some issues open. To discuss the details of the recovery process, we present its flow, divided into logical steps. Next, we describe each step and discuss a couple of optimizations. Finally, we explain the impact on overall system performance.

In our analysis of recovery, we must include state update. While in FullSS a replica is considered as recovered even before it starts learning decisions it missed while being down, it becomes fully usable no sooner than after its state is up to date. In ViewSS and EpochSS, the state update is already part of the recovery process, as to become operational, a recovering process must learn values it may have known before the crash. As there is no way to determine when exactly the crash occurred, the recovering replica learns everything up to the moment when it started the recovery phase, which brings the replica up to date.

6.1 Steps of the Recovery Process

We can split the recovery process into the following steps which are common for our recovery algorithms. They simplify the analysis and comparison of the algorithms. Steps 1-3 are strictly related to the recovery algorithms, while Steps 4-6 describe the state update:

1) *Accessing stable storage* – a recovering replica reads data from stable storage,
2) *Algorithm-specific actions* – any local actions that are specific for a given recovery algorithm,
3) *Message exchange* – replicas exchange any recovery messages (if necessary),
4) *Asking for the current state* – the recovering replica asks a peer for a state transfer,
5) *Transferring a state* – a peer replica sends a snapshot and subsequent log entries,

6) *Applying the transferred state* – the recovering replica restores its state on basis of received data.

Unless a replica is run for the first time, at start-up it executes the above steps in order to recover. In Step 1, FullSS reads the most recent local snapshot and the log of all Accept and PrepareOK messages, whereas ViewSS and EpochSS only read a single value, respectively, the ballot and epoch number. In Step 2, FullSS updates the replica on basis of the read data, eventually reaching the state it had at the moment of a crash. For FullSS, this step completes the recovery phase, letting a recovering replica to fully participate in Paxos. However, unless other replicas made no progress from the crash, the replica still needs to update itself before it can deliver upcoming commands. In the same step, a replica using EpochSS just synchronously writes a new epoch number to a disk, while ViewSS performs no actions. In Step 3 (absent in FullSS), the replica broadcasts a Recovery message and waits for the RecoveryAck messages from a majority set of processes. The responses acknowledge receiving Recovery and also allow the recovering replica to recognize a command number of the most recent voting (see Alg. 3 line 41 and Alg. 4 line 44). This information is used to determine what state this replica must reach in order to finish the recovery process. In ViewSS, in case if the leader has not changed since the crash, the Recovery message also triggers a ballot change.

While Steps 1-3 differ between the recovery algorithms, the remaining steps are identical for all of them. They aim at restoring the state of the recovering replica using the catch-up protocol. In Step 4, a recovering replica p sends a query to any other replica. The query indicates what should be sent in response. While ViewSS and EpochSS learn what state p is missing in Step 3, FullSS must wait for any message of the Paxos protocol, in order to know whether the catch-up is required, and if so then which commands p does not know. When a replica q receives the query, it replies with the most recent snapshot. As the snapshot may represent a stale state, q also sends any commands that are following the snapshot creation (Step 5). It may occur that no snapshot has yet been created, or that the snapshot represents the current state. Then, only a log, or only a snapshot is sent. As soon as a process receives a snapshot, it can use it to restore the state (Step 6). Afterwards, all subsequent commands received from q can be issued. Unlike other steps, Steps 5 and 6 can be executed in parallel, as restoring the state on basis of a snapshot can proceed while the missing decisions are still in transit.

While local steps always succeed at first try, steps involving peers (3, 4 and 5) may fail due to process or network failures. The failing step is repeated until it succeeds. Step 3 may fail due to a timeout on message delivery or a race condition with ballot change (if a recovering replica gets a message that was sent by a process before it became the current leader). Steps 4 and 5 may fail if a target replica is down or if the messages are not delivered in a timely manner. It may also happen that the target replica is outdated, thus contacting other replica is needed.

6.2 Performance Issues of Replica Recovery

In Section 5, we discussed the overhead that each recovery algorithm induces on every process during normal (non-faulty) system execution. To summarize: FullSS brings a major performance penalty in a voting phase, due to synchronous writes to stable storage; ViewSS slows down each ballot change by demanding a single synchronous write (of a single ballot number); EpochSS enlarges the ballot change messages with a vector of epoch numbers (one per each replica). In this section, we analyze and compare system performance during a recovery phase.

Despite many similarities, each algorithm has its own characteristics. The main differences are as follows: FullSS requires reading the state from stable storage and restoring it, and updating it with decisions missed during down time. ViewSS must broadcast a message and gather responses, possibly causing a ballot change thereby, and fetch a state of a peer replica. EpochSS has the same complexity as ViewSS, however, it never forces a redundant ballot change. To expose the differences, below we present the border cases in terms of the recovery cost.

If no decisions were taken yet, all algorithms skip updating state, which gives the shortest possible recovery—FullSS recovers as soon as the replica starts, while other algorithms just need to exchange messages with peers.

When no decisions were taken during downtime, FullSS is a clear winner (under an assumption that the stable storage is faster than the network). While all algorithms must recreate state, ViewSS and EpochSS also need to fetch it from a peer via the network. Thus, with infrequent decisions, FullSS may be preferred. Also, if few decisions were taken since crash, replicas are likely to still have them in their logs. Therefore, FullSS has to fetch substantially less data than other algorithms, while the amount of local work required to bring the state back is roughly the same.

However, if multiple decisions were taken since crash, FullSS becomes inferior, as the state restored by FullSS from stable storage is out of date. So, just after a costly state recreation the replica has to update its state again, but this time using the same protocol as in ViewSS and EpochSS. Thus, in total, state update will be more resource consuming than in EpochSS and ViewSS.

In the statistically most probable case, the crashed process is not a leader. Note also that the crash of a follower does not induce a leader change. Then, the difference in the behavior of ViewSS and EpochSS is obvious. In EpochSS, a recovering process only exchanges messages related to state update, while in ViewSS a new ballot is forced, and the replicas respond to Recovery with RecoveryAck after the new leader has been elected. Thus, typically a system using EpochSS is expected to recover faster than with ViewSS.

6.3 Recovery Bottlenecks

The recovery algorithms use three resources: stable storage, processor, and the network, each of which can limit the system performance. Below we characterize possible bottlenecks of the recovery process, and explain under what circumstances a particular resource can noticeably limit the system performance. Note however that applications may range from systems with tiny snapshots that are very fast to restore, up to systems where the size of a snapshot is very large, so the time necessary to recover from it can be enormous. Moreover, the size of a command to be decided, the frequency of snapshot creation, and the time it takes to execute the commands, also depend on an application.

Stable Storage: In modern computers the mass storage devices are rarely a bottleneck. However, in Step 1 of the recovery process, FullSS may need to read a large amount of data from stable storage. In case the read speed is low, loading the data to physical memory might be an issue. An alternative to reading from stable storage is fetching the data from peers, thus in case when storage limits performance, this task can be offloaded onto the network.

Processing power: During recovery the system needs to restore its state from a snapshot and execute a possibly large number of commands. While the Paxos algorithm uses almost no processor time in this step, the system may need to perform some resource-consuming computations in order to restore the state and execute the commands. This can severely impact the time of the recovery process. If this is the limiting factor of recovery, all that can be done on the level of recovery algorithms and state update is to minimize the amount of any redundant data that are fetched. In FullSS the state may need to be restored twice (in Step 2 and 6).

Network Bandwidth: During recovery a large volume of data may be transferred from peers, most of it during the state update in Step 5. ViewSS and EpochSS also exchange recovery messages with all replicas in Step 3, but this boils down to a single best-effort broadcast and its acknowledgment, so is unlikely to impact the recovery when compared to the cost of the state update. With a moderate network throughput transferring the data may easily limit the performance of recovery. The amount of these data cannot be decreased, since the desired result of recovery is to bring the replica back to the operational state.

6.4 The Impact of Recovery on Other Replicas

Recovery of a crashed replica introduces an overhead on other (non-faulty) replicas. In this section, we discuss the impact of this overhead on the overall system performance.

The peer replicas are contacted in Step 3 of the recovery process. In EpochSS this has a negligible impact on performance, as it boils down to receiving a single message and transmitting an acknowledgment. However, in ViewSS, this step may force a ballot change (after receiving Recovery), so it may stop the whole system until a new leader is elected. This results in a clear performance penalty. On average, the duration of the ballot change is similar to the time of deciding a few commands by Paxos, plus the time of a synchronous write to stable storage. The synchronous write may take even as much time as deciding a few commands. Thus, recovery with ViewSS may stop the system for the time interval equal to deciding at least several commands.

There is a single corner case in EpochSS when a recovering process forces a ballot change. The following conditions must be met for that: 1) the current leader p crashes, 2) no process starts the ballot change, 3) p starts recovery. This is possible if the leader crashes and restarts so fast that no process noticed the crash. EpochSS cannot avoid this particular ballot change, since p could have sent a Propose message before crashing, and this message must be rejected by processes which learned that p crashed and recovered. Notice, that in this case the system is unavailable until a leader is operational. Selecting a new leader takes less time then recovery, hence we see no point in striving to avoid this ballot change. Both in EpochSS and ViewSS when the leader crashes, a single ballot change happens as soon as the crash is detected. Upon a follower crash, in EpochSS no ballot change happens, and in ViewSS a ballot change happens at recovery.

In Step 5, a single peer replica performs a state transfer to the recovering replica. While selecting the most recent snapshot and subsequent requests is not resource demanding, transferring them consumes a significant amount of bandwidth. So, the peer can slow down noticeably. How big is the impact of such slowdown on the overall system performance depends on additional factors. For example, with three replicas (one of which is recovering), slowing down the peer causes a global slowdown. In a system with five replicas, one of which is recovering and one is slowed down, the remaining three replicas are enough to form a majority with unaffected performance, thus the system should retain its speed. Note, that the replica sending its state to the recovering one only uses the outgoing link, so it will not become outdated. Also, if the network is capable of transferring the state update alongside with protocol messages with no performance penalty, the system is not affected by Step 5.

In a system using ViewSS or EpochSS, simultaneous crash failures exhibit the same behavior as simultaneous crashes in Paxos in the crash-stop model. In terms of performance this means that: 1) the system stays available, as long as at most f replicas crashed, 2) the clients connecting to the system spread among less replicas, so the load on each replica raises, 3) the leader replica broadcasts messages to a smaller number of replicas, so the average bandwidth from the leader to each replica raises (assuming that broadcast is implemented as a series of unicasts). In the saturated network workloads, this results in increased throughput, while in the saturated CPU workloads, this causes decreased throughput (see Section 7.3.2).

A recovery process of one replica is independent from a recovery process of another replicas. In EpochSS and ViewSS each replica separately has to gather the RecoveryAck message from a majority of replicas that are up. In ViewSS, a crash and subsequent recovery of a follower forces a ballot change. If multiple followers crashed, and then recovered simultaneously, a single ballot change suffices. If multiple replicas are simultaneously updating state, the impact on system performance in JPaxos is expected to be equal to the sum of impacts of single state updates.

7 Experimental Evaluation

In this section we present results of an experimental evaluation of the recovery algorithms using JPaxos [9]—our implementation of Paxos, equipped with the batching and pipelining optimizations for a higher performance. A variant of JPaxos with no recovery support was tested in [32].

JPaxos is a state machine replication tool. For our tests we used *EchoService*, a very simple replicated service: for a)

	crash stop	FullSS	ViewSS	EpochSS
Saturated net.	38 087	1 871	38 200	38 209
Saturated CPU	158 288	11 164	157 317	159 192

b)

	crash stop	FullSS	ViewSS	EpochSS
Saturated net.	38 130	36 768	38 015	38 329
Saturated CPU	156 787	103 323	155 685	157 228

TABLE 1
The number of requests per second with: a) HDD and b) RAM disk.

every client request a, the service sends a back to the client. Moreover, randomly, but on average once per ten thousand of requests, a snapshot of a minimal size (one byte) is created on each node in order to limit the growth of the log. The service introduces minimal overhead on the system, thus enables evaluation of Paxos and the recovery process alone.

We run tests in a cluster of identical machines, equipped with Intel® Xeon® X3230 (4×2.66 GHz, 8 MB L2 cache), 1Gbps LAN (running at wire speed), OpenJDK 1.7.0_40, openSUSE 12.3. In almost all tests we simulated stable storage using a `tmpfs` RAM disk, unless we stated otherwise. Then we used HDD ST3250620NS (SATA II, 7200rpm, 16MB cache, avg seek/write/latency 8.5/10/4.16 ms).

Depending on the workload and available resources, either network or CPU is limiting the system performance. Typically, in every Paxos implementation huge requests saturate the available bandwidth, while numerous tiny requests use up all available processor time long before any other resource runs out. Therefore, we carried out tests in two setups: 1) requests are large enough to saturate the network bandwidth (the network is *saturated* or is a *bottleneck*), and 2) requests are small enough so that the system runs with maximum CPU utilization (CPU is *saturated* or is a *bottleneck*). In our cluster the borderline between the request size saturating the network and CPU lies at around 320B, thus we chose 1024B for saturating the network and 128B for saturating the CPU. For setups 1 and 2, a proper number of concurrent requests was selected (2.1k and 6k, respectively), so that increasing the number does not increase the system throughput, and the latency is kept low enough to prevent re-sending requests due to a timeout.

We evaluated each recovery algorithm using the same scenarios, considering several cases specific for a particular algorithm. FullSS does not require catch-up after recovery if no commands were decided since a crash. So, we examined two cases: 1) stable storage is out-of-date, and 2) stable storage is up-to-date. ViewSS enforces ballot change in order to invalidate any stray messages: If a process receives the Recovery message with a ballot number that is greater than or equal to the ballot number held by the process, the ballot change must be enforced. So, we considered two cases when evaluating ViewSS: 1) *leader crash*, and 2) *follower crash*, indicating what was the rôle of the recovering replica prior to a crash. In the first case, ballot change occurred prior to recovery, while in the latter case no ballot change occurred since crash, so the Recovery messages induce a ballot change during recovery. In contrary, EpochSS has no special cases to consider per evaluation scenario.

7.1 Evaluation of No-Crash Operation

In Table 1a, we show the system *throughput* (the total number of requests per second) when an HDD was used as stable storage. In this case, FullSS is much slower than ViewSS and EpochSS.

While in our cluster an HDD was able to perform only at most 30 `sync()` operations per second, a SSD had a much shorter access time, and cutting edge SDDs can break the boundary of one million IOPS. So, in other experiments we simply used a RAM disk that simulates an ideal bus-speed stable storage device. A RAM disk significantly improves the throughput of EchoService with FullSS (see Table 1b).

If the network is a bottleneck, EchoService with FullSS is only 3.5% slower than with ViewSS or EpochSS (both equipped with a RAM disk) or EchoService with no recovery support (denoted "crash stop"), but contrary to them it is able to recover from catastrophic failures. So, it can be useful in some cases. When the CPU is a bottleneck, the benefits of FullSS are less clear—it is 34% slower. This can be attributed to a higher demand on processing power, which stems from the need to invoke kernel functions per each voting. In both scenarios, the performance of ViewSS- and EpochSS-based EchoService is indistinguishable from "crash stop".

7.2 Evaluation of Recovery Operation

In this experiment, we show how long it takes for a replica to recover. The sooner a replica recovers, the sooner resilience to failure is restored. To evaluate the recovery process, we measured the time of the following events:

1) a recovering replica broadcast the Recovery message,
2) the last Recovery was delivered and the corresponding RecoveryAck was sent,
3) the recovering replica got all RecoveryAck messages from a majority set,
4) the recovering replica started the catch-up protocol,
5) the recovering replica received a snapshot,
6) the recovering replica got all missing commands,
7) the recovery is finished.

In Figure 2a-c, we present the evaluation results of the system recovery for three scenarios: a) the system is idle (it does not process any requests during the recovery phase), b) the system is saturated with small requests, which causes the processor to become the bottleneck, and c) the network is saturated with large requests. We use the following symbols: *Fo* and *Fu* is FullSS with stable storage contents, respectively, out of date and up to date, *Vf* and *Vl* is ViewSS, respectively, with follower crash and with leader crash, and *E* is EpochSS.

Little difference can be seen between the idle system and the system with saturated CPU (see Figure 2a-b). The system with the saturated network was recovering longer than the other systems (see Figure 2c). This indicates that a network is the main bottleneck for the recovery process, and the recovery phase does not use excessively processor time on other nodes except the one that is recovering. The amount of usage of CPU by the recovering replica depends mainly on the replicated service.

Below we discuss the evaluation results. First, we analyze the recovery of the FullSS-based system. Next, we examine the recovery phase using ViewSS and EpochSS, and the catch-up protocol. Finally, we discuss the overhead introduced by slow stable storage, request processing, and snapshot processing.

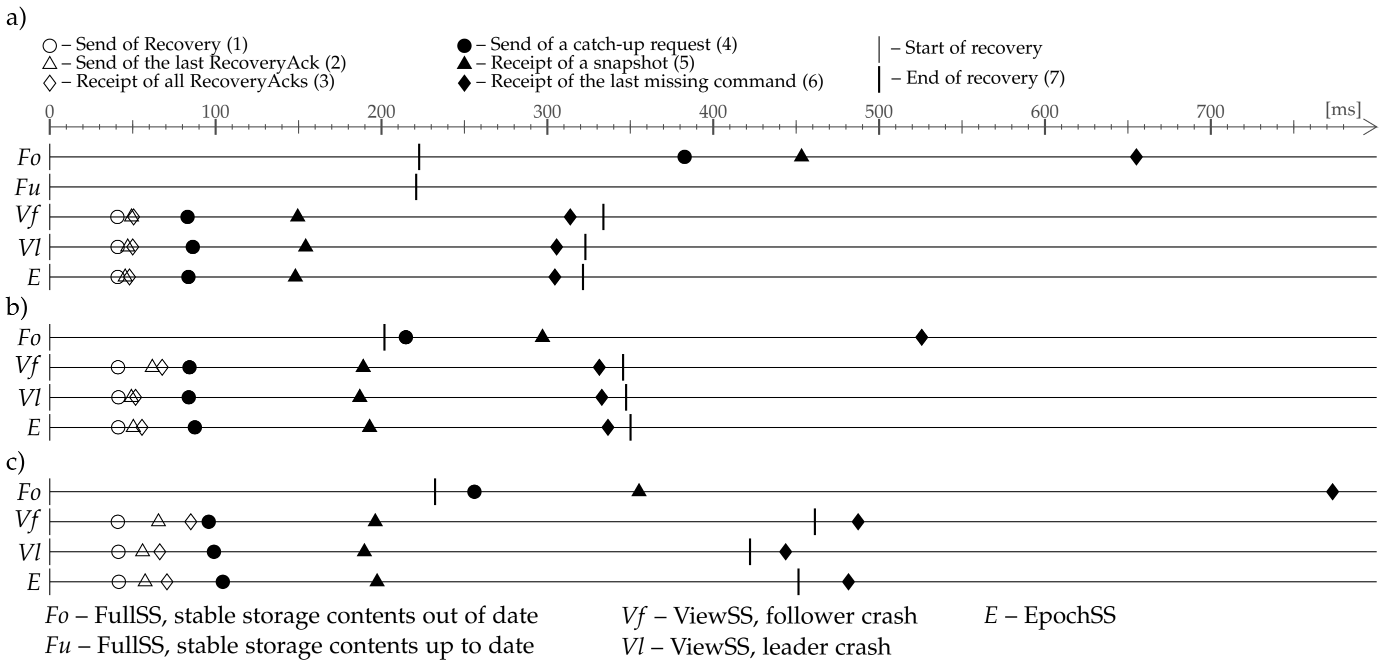

Fig. 2. Time needed to recover a crashed replica in case of: a) the idle system, b) CPU saturation, and c) network saturation.

7.2.1 FullSS

In FullSS, recovery starts from reading a snapshot and logs from stable storage, followed by recreating the state of a recovering replica. This completes the recovery phase (7) (see Figure 2, where we use numbers 1–7 to identify a given action). Even though the replica is able to recover by its own, it must run the catch-up protocol afterward to become up-to-date (if the logs were outdated). In JPaxos, catch-up is triggered either by the first Paxos message, or by a heartbeat message of the failure detector. In the idle system, we tested two scenarios: 1) stable storage was up-to-date, and 2) some requests were processed when the recovering replica was down. In Figure 2a, case Fo, catch-up is triggered, and it occurs later than in any other test in Figure 2 (on average 160 ms after finishing recovery, and 380 ms after replica started). As no commands are voted, catch-up was triggered by the failure detector (the heartbeat messages are sent periodically when no client requests are issued).

A restarted replica can serve new requests only when it has finished recovery (7) *and* it is up to date (6). In all our tests, the FullSS-based system finished recovery faster than other systems. However, it becomes up to date faster than other systems only in one case—when the whole system did not receive any requests since crash. In other cases, FullSS was the last one to restore full functionality of a recovering replica. Also, the catch-up protocol takes more time in the FullSS-based system than in any other system (15% if the system is idle, 20% if the CPU is saturated, and even 40% more time if the network is saturated).

7.2.2 ViewSS and EpochSS

A recovering replica using ViewSS and EpochSS must contact other replicas. The time instant (1) when it initializes itself and sends a Recovery message is the same for all our tests. The time instant (2) when all other replicas respond with a RecoveryAck message varies across tests. In the idle system, the elapsed time between (1) and (2) is negligible. The same is when CPU is saturated, except for the case of the crash of a follower in ViewSS (discussed later). Gathering the RecoveryAck messages by the recovering replica (3) takes approximately the same amount of time as for the Recovery message. As expected, the exchange of recovery messages takes more time when the network is saturated. However, in most of our tests, the elapsed time between time instants (1) and (3) is just a fraction of time which is necessary to recover a process (2% to 6% of the total recovery time, except for a few cases discussed below).

ViewSS enforces a ballot change when a follower crashes, which impacts results in Figures 2b and 2c. It takes more time (compared to other results in Figure 2) before replicas can process the Recovery messages (2) and when Recovery-Acks are delivered to the recovering process (3). However, the overall system performance does not seem to be affected, contrary to the expectations.

7.2.3 Catch-Up

No matter which recovery algorithm is used the system executes the catch-up protocol. While the FullSS-based system only uses it to update state, ViewSS and EpochSS require it for correctness. The protocol is initiated by sending a catch-up request (4). In ViewSS and EpochSS, this occurs as soon as possible, while in FullSS—once the recovering process notices that it is late. In response to the catch-up request, a recovering replica receives from another replica a snapshot (5) and/or the missing commands (6). Completing catch-up in case of FullSS takes noticeably more time than in other systems, since all data must be written to stable storage. As expected, in case of ViewSS and EpochSS it takes the least time to complete the catch-up protocol when the system is idle. When the network is saturated, a process is able to recover a short while before the catch-up protocol finished.

This may occur if some commands were decided between sending RecoveryAck and sending a snapshot.

7.2.4 Hard Disk Drive

Figures 2a-c present the results obtained for the EchoService system equipped with a RAM disk, as a substitute for high-speed stable storage devices, such as SSDs. We repeated all tests for an HDD. FullSS reads a large amount of data from disk. So, not surprisingly, the time required to complete catch-up by our system with HDD is twice longer when the system is idle and over four times longer when the CPU or network is saturated, compared to the same system using a RAM disk. In effect, it takes the recovering replica about 3200 ms to be able to process any further requests when the network is saturated, while other algorithms require less than 600 ms.

In case of ViewSS, if a leader crashed, there is no difference. However, if a follower crashed, the Recovery message is received by replicas (2) after approximately 120 ms from being sent by the recovering replica. The time required to exchange recovery messages (2–3) is at least twice longer than when using the RAM disk. The main cause of delay are synchronous writes which are performed on ballot change by every replica. So the delay affects all replicas.

In case of EpochSS, the start-up time (1) increases by 33 ms (nearly twice longer as before), since the recovering replica must synchronously write a new epoch number to stable storage. Processing the subsequent events (2-7) takes the same amount of time as in the system with a RAM disk (just they appear later in time). This is because other replicas do not write anything to a disk.

7.2.5 Snapshot and Log

Our EchoService benchmark was configured in such a way that the time of recreation of state from a snapshot and the time of executing (on average) 5000 requests from a log were negligible. This is because our main goal was to measure the recovery time imposed purely by the recovery algorithms. Thus, the results can be seen as an estimation of the lower bound for actual services, where these times can be longer. Moreover, in EchoService the size of snapshots was just one byte, while a real service may produce much larger snapshots, which means that also the time required to transfer a snapshot between replicas can be longer than in our tests. Thus the time to bring a crashed service back to operation can be much longer than the maximal time of recovery (around 0.6 s) that we measured in our system using ViewSS and EpochSS.

7.3 Crash-Recovery vs. System Throughput

When the number of replicas in a distributed system changes due to a crash or a recovery, the system performance changes as well. In this section, we examine how the performance of Paxos changes upon a crash, as well as upon recovery, when the recovering replica interacts with other replicas. In case of our lightweight EchoService, the system performance comprises the performance of Paxos and the catch-up protocol. In order to measure the throughput of an individual replica, our service reports every 100 ms the total number of processed requests. This sampling rate has negligible impact on the rest of the system.

In our experiments, we run 3 replicas and a number of clients. At 3sec from the start of our benchmark, the clients begin sending requests. At 12.5sec after system stabilizes and some requests were executed, one replica crashes. At 18sec the replica is up again (and starts recovery). In Figures 3, 4, and 7, we depict the throughput from 10sec to 22sec of our benchmark run.

7.3.1 The Impact of a Crash on System Performance

In Figures 3a-d, we present the throughput of a non-faulty replica in the EpochSS-based system, where either the network is a bottleneck (a-b), or CPU is a bottleneck (c-d), and a crashed replica is a follower (a, c) or a leader (b, d).

As expected, the crash of a follower does not cause service downtime, while the crash of a leader makes the system unable to take new decisions until the crashed leader becomes suspected and a new leader is chosen. JPaxos was configured so that a follower suspects a leader process to have crashed if it has not received any message from the leader for 1000 ms. Then, the follower starts a new ballot to select a new leader. Between the crash and the election of a new leader the service is unavailable.

After a self-adjustable timeout, the clients that do not receive responses to their requests (e.g. due to a replica crash or a network partition), reconnect to other replicas and issue their requests again. Some of these replicas pass the client requests to the old leader that has crashed but is not yet suspected by them. Therefore, the clients must reconnect and issue their requests again. Since they reconnect after non-uniform timeouts, it takes a while before the system reaches its maximum throughput, which is depicted in Figure 4. In the network saturation scenario (a), the throughput rises rapidly, as a moderate number of requests in this scenario quickly saturates the network. On the contrary, in the CPU saturation scenario (b), a vast number of clients must reconnect before the throughput reaches its limit, hence it takes more time to reach the maximum performance.

In Figure 5, we show an interesting asymmetry between the performance of two replicas p and q that remained up after the leader crashed. The explanation is as follows. In some executions, the leader managed to send the Propose message for a command i to follower p, but crashed before sending it to follower q. In such case, p broadcasts Accept and issues command i (as it gathered the majority of votes, i.e., its own vote and an implicit vote of the leader). However, q cannot issue a command for i yet, as it only has got the vote cast by p. Let us assume that p becomes a new leader. Then, once any new commands were decided, p can issue them to the service, while q cannot, as it first has to learn the missing ith command, which in JPaxos occurs only through the use of the catch-up protocol. As soon as q learns command number i, it can issue it to the service, together with any other commands decided by the new leader. The impact of catch-up can be seen as a peak in performance of the follower q (see Figure 5b). The results shown in the figures are an average of a few hundred measurements. So, the throughput of q is a composition of the executions in which q did not miss any command (so its performance was stable p's), and the executions in which q first missed

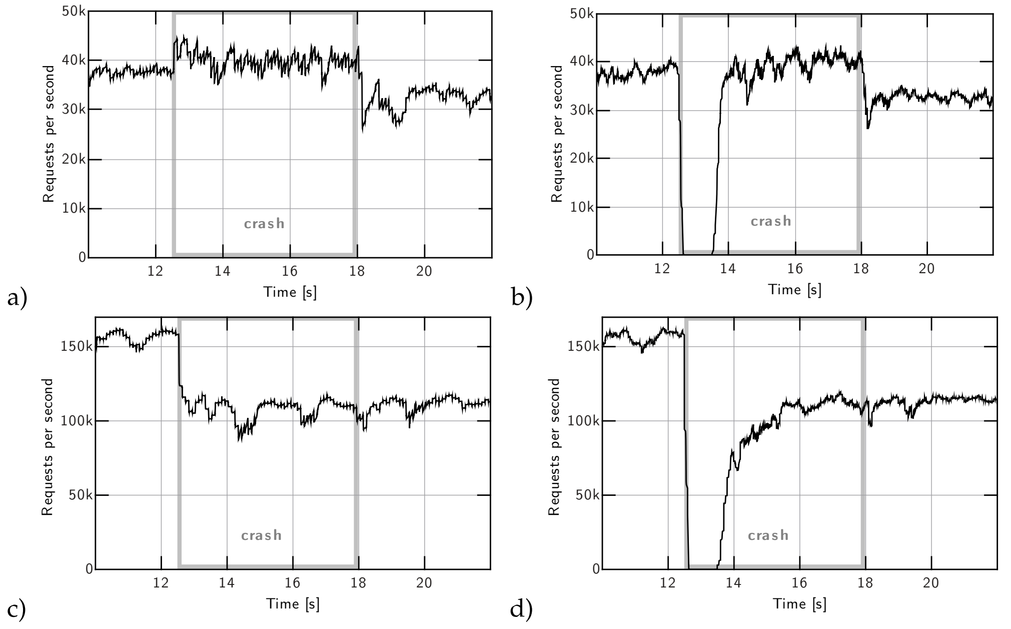

Fig. 3. Throughput of a non-faulty replica in EpochSS: a,c) the crash of a follower, b,d) the crash of a leader, a-b) saturated network, and c-d) saturated CPU.

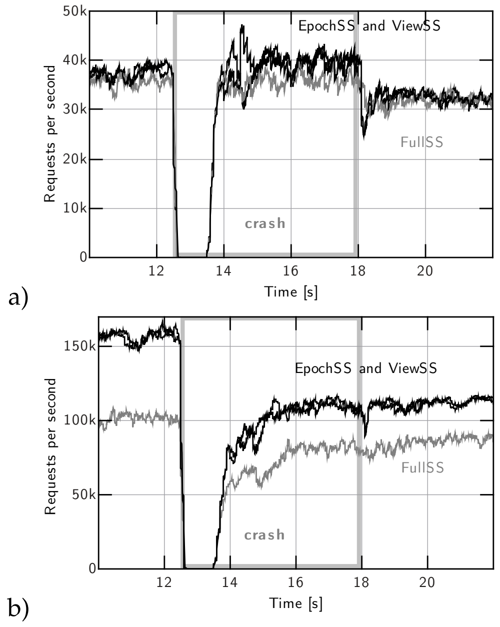

Fig. 4. The crash of a leader: a) saturated network, b) saturated CPU.

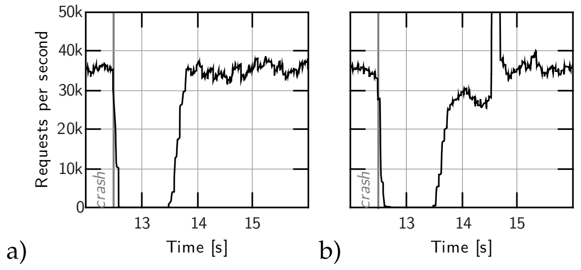

Fig. 5. Asymmetry of throughput: a) a new leader p, b) a follower q.

commands, so the throughput of q was zero, and later issued a large batch of commands, so the peak in performance.

7.3.2 Throughput During Stable Period

We analyzed the system throughput in the following *stable periods* – i.e., when no crash nor recovery impacted the system performance:

A – just before crash (10–12s),
B – just before recovery begins (16–18s),
C – after the recovery, when the system is stable (20–22s).

Firstly, let us compare the performance of a non-faulty replica (a leader or a follower) in periods A and B, see Figure 3. This comparison shows how the system performance changes when a replica crashes and stays down. If the network is the bottleneck (3a-b), the throughput of the measured replica slightly rises in period B, since there is more available bandwidth for the communication between the leader and the follower and the clients, as no data are transferred to the crashed replica. If the CPU is the bottleneck (3c-d), the performance drops in period B compared to A, since more clients connect to the remaining (already overloaded) replicas, thus increasing the amount of work on each of them. In general, if only two replicas are up the performance of the non-faulty replica is less stable (especially when the network is saturated).

In all cases, in the time period C (after the recovery process completed) the system performance is lower than in the time period A (before the crash). This is because JPaxos currently does not support load balancing. In effect, the clients that reconnected to a new replica are not redirected back to the old replica once it has recovered after a crash. In every case, load balancing is not tightly coupled with recovery, so the performance drop does not expose any drawback of the recovery algorithms.

Note also that if the network is saturated, the system performance in the time period C (after recovery completed)

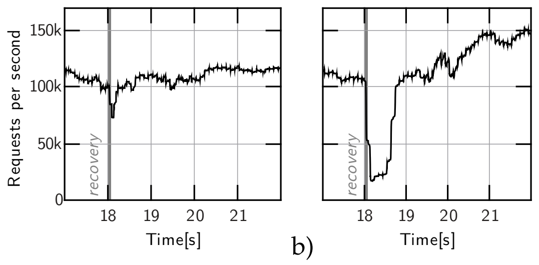

Fig. 6. Throughput in ViewSS-based system with: a) RAM disk, b) HDD.

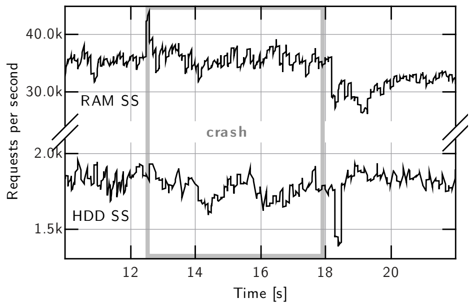

Fig. 7. Throughput in FullSS-based system: RAM disk vs. HDD.

is lower than in the time period *B* (just before recovery begins), as messages must also be sent to the recovered replica, but the network bandwidth is already used up (see Figures 4a-b). Whenever the CPU got saturated, the system performance in the periods *B* and *C* remains unchanged, as no additional processing is required.

Although JPaxos currently does not support load balancing, the system performance is slowly restored, as long as new clients join the system or old clients reconnect. This behavior can be observed in Figure 6, where we show the performance of ViewSS-based system using a RAM disk and HDD. When we tested ViewSS with an HDD, the leader change enforced by the recovering replica took more time than in the system configuration using a RAM disk. In effect, the system became unavailable for a moment, so the clients started reconnecting, thus restoring the balance.

7.3.3 Throughput During Recovery

At recovery, the system performance drops, which is especially visible if the network is saturated. The drop occurs during the catch-up phase (see Fig. 2b-c for the offset and duration of catch-up). It slightly differs for different algorithms (see Fig. 4). While ViewSS and EpochSS have a noticeable performance drop, in FullSS the drop is smaller in value, but is spread over a larger time span (as is the catch-up).

When JPaxos uses an ideal stable storage emulated by a RAM disk, the differences between the recovery algorithms are small, but with HDD they become distinct, as follows. First of all, in FullSS the catch-up takes more time, but the impact of recovery on other replicas is visible only as a short drop of performance at the beginning of the catch-up from the 18.3s till 18.6s (see Figure 7). With ViewSS, the follower crash does not affect the system performance, but in case of the leader crash the impact of the enforced view change is clearly visible. As seen in Figure 6b, the processing stops for a short period (from 18sec till 18.7sec), needed to perform the view change and restart the Paxos protocol in the new view. The EpochSS results remain identical regardless of what medium was used as stable storage.

7.4 Discussion of System Workload Impact

Below, we discuss the impact of the system workload on our results.

In Section 7.1, we examined performance overhead of supporting the system recovery during crash-free periods. In case of ViewSS and EpochSS, the results are independent of the system workload. In case of FullSS, the performance penalty raises when the processing power is the factor limiting performance.

In Section 7.2, we examined the duration of the recovery process. The time it takes to recover the system highly depends on the workload. In FullSS, the system recovery time depends solely on the application characteristic, as the recovery only consists of updating the system state. In ViewSS and EpochSS, Steps 1-3 of the recovery process are independent of the system workload, yet the duration of Steps 4-6 (i.e., of the state update) depends on the application. If the application needs to transfer a large amount of data to update the system state, or, if restoring a state from snapshot lasts long, or, if executing requests takes much time, then obviously the state update dominates the recovery process. Therefore, in order to differentiate between the recovery algorithms, in our experimental evaluation we chose the system workload with negligible times of command execution and state restoration. Also, the size of data transferred during state update was relatively small, and, on average, equal to five thousands requests.

In Section 7.3, we examined how crash and recovery affect the system throughput. These results are independent of the system workload, with one exception. In Section 7.3.3, we show how transferring data to the process that is being recovered affects the system performance. If state update takes more time, due to the application characteristic, then the observed slowdown will last longer.

8 Conclusions

We analyzed and compared three recovery algorithms for Paxos-based state machine replication: FullSS, which essentially renders the original presentation of Paxos in [1], and ViewSS and EpochSS which minimize the use of stable storage during normal (non-faulty) system operation.

Not surprisingly, restoring lost state is the main factor of the total time of recovery (even for our simple EchoService application), no matter which algorithm was used. However, FullSS was significantly worse in our comparison, as the lost state is first read from stable storage (which survives crashes) and later is updated by data received from peers, while the ViewSS and EpochSS algorithms restore lost state exclusively from other replicas through catch-up. Operations of ViewSS and EpochSS (not related to catch-up) took at most 100 ms during recovery when a RAM disk was used, and—in case of EpochSS only—just 33 ms longer for HDD.

An advantage of FullSS is that it allows the system to recover also when less than a majority of processes remains up at the same time. However, the performance penalty during normal (non-faulty) system operation is very large due to frequent synchronous writes to stable storage. When we used a RAM disk (as a substitute for a high-end SSD), performance of Paxos improved but it was still visibly worse compared to Paxos equipped with the other two recovery algorithms (kernel calls of synchronous writes suffice to considerably slow down processing). Also, recovery of a crashed process took more time in FullSS.

ViewSS and EpochSS do not impact the performance of Paxos during normal operation (the former requires just one synchronous write to stable storage in the reign of a leader vs. the latter that slightly increases the sizes of ballot change and recovery messages) and introduce a negligible slowdown during process recovery. However, ViewSS may sometimes slow down the system during process recovery (when a leader change is forced), giving no pros in return. Therefore, EpochSS seems to be favorable, unless the system must tolerate more than $\lfloor \frac{n-1}{2} \rfloor$ crashes at once, in which case FullSS is the only choice out of the three.

In a system using EpochSS or ViewSS, when a crashed process recovers, some other process is slowed down, as it has to transfer data to the recovering process. This may slightly slow down the whole system. Moreover, in case of ViewSS, it is likely that a new leader has to be elected, which effectively prevents the system from deciding commands for a while. In our test, if the CPU was saturated, the performance drop was not larger than 5% and lasted a fraction of a second, whereas when the network was saturated, the drop reached at most 15% and lasted until the recovery finished.

Obviously, the crash of a leader suspends the system until the crash is detected and a new leader is elected. Moreover, the clients may also be blocked, waiting on a timeout after which they resend the requests. A highly optimized system should implement a load balancing mechanism to distribute the requests evenly among all replicas that survived crashes and that have recovered and re-joined the system.

In the future work, we would like to investigate recovery algorithms for the future computer architectures, equipped with a non-volatile random-access memory (NVRAM), built with the use of emerging non-volatile technologies (e.g. 3D XPoint [35]). Note that the recovery algorithms presented in this paper will still be applicable (e.g., to catch-up the state after restart, and to recover the state kept in CPU registers and volatile part of RAM or lost due to hardware failure), but NVRAM opens space for improvement.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their valuable comments which helped to improve the manuscript. This work was supported by the Polish National Science Centre (grant No. DEC-2012/06/M/ST6/00463); and within the TEAM programme by the Foundation for Polish Science, co-financed by the European Union under the European Regional Development Fund (grant No. POIR.04.04.00-00-5C5B/17-00).

REFERENCES

[1] L. Lamport, "The Part-time Parliament," *ACM Trans. Comput. Syst.*, 1998.
[2] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in *Proc. of OSDI '06*, 2006.
[3] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-distributed Database," in *Proc. of OSDI '12*, 2012.
[4] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live: An Engineering Perspective," in *Proc. of PODC '07*, 2007.
[5] J. Rao, E. J. Shekita, and S. Tata, "Using Paxos to build a scalable, consistent, and highly available datastore," in *Proc. of VLDB Endowment '11*, 2011.
[6] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, "Deconstructing Paxos," *SIGACT News*, vol. 34, no. 1, pp. 47–67, Mar. 2003.
[7] A. Fox and E. A. Brewer, "Harvest, yield and scalable tolerant systems," in *Proc. of HotOS-VII: the 7th Workshop on Hot Topics in Operating Systems*, Mar. 1999.
[8] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," in *Proc. of PODC '88*, 1988.
[9] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, "JPaxos: State machine replication based on the Paxos protocol," EPFL, Tech. Rep. EPFL-IC-TR-167765, July 2011.
[10] R. D. Prisco, B. W. Lampson, and N. A. Lynch, "Revisiting the Paxos algorithm," *Theor. Comput. Sci.*, vol. 243, no. 1-2, pp. 35–91, 2000.
[11] J. Kirsch and Y. Amir, "Paxos for System Builders: An Overview," in *Proc. of LADIS '08*, 2008.
[12] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Proc. of ICDCN '10*, 2010.
[13] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos Replicated State Machines As the Basis of a High-performance Data Store," in *Proc. of NSDI '11*, 2011.
[14] P. J. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *Proc. of DSN '12*, 2012.
[15] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Comput. Surv.*, 2015.
[16] B. Lampson, "The ABCD's of Paxos," in *Proc. of PODC '01*, 2001.
[17] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, 2006.
[18] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance Broadcast for Primary-backup Systems," in *Proc. of DSN '11*, 2011.
[19] B. Liskov and J. Cowling, "Viewstamped Replication Revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.
[20] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," in *Proc. of DISC '98*, 1998.
[21] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. of USENIX ATC '14*, 2014.
[22] E. Michael, D. R. K. Ports, N. K. Sharma, and A. Szekeres, "Recovering shared objects without stable storage," in *Proc. of DISC '17*, 2017.
[23] A. N. Bessani, M. Santos, J. Felix, N. F. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proc. of USENIX Annual Technical Conference '13*, 2013.
[24] O. M. Mendizabal, F. L. Dotti, and F. Pedone, "High performance recovery for parallel state machine replication," in *Proc. of ICDCS '17*, 2017.
[25] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010.
[26] ——, "Vertical paxos and primary-backup replication," in *Proc. of PODC '09*, 2009.
[27] L. Jehl, T. E. Lea, and H. Meling, "Replacement: Decentralized failure handling for replicated state machines," in *Proc. of SRDS '15*, 2015.
[28] L. Jehl and H. Meling, "Asynchronous reconfiguration for paxos state machines," in *Proc. of ICDCN '14*, 2014.
[29] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM*, vol. 43, no. 4, pp. 685–722, 1996.
[30] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM (CACM)*, vol. 21, no. 7, pp. 558–565, 1978.

[31] L. B. Lamport, "Fault-tolerant system and method for implementing a distributed state machine," Nov. 1993, US Patent no US 5261085 A.
[32] N. Santos and A. Schiper, "Tuning Paxos for high-throughput with batching and pipelining," in *Proc. of ICDCN '12*, 2012.
[33] "JPaxos – Java library and runtime system." [Online]. Available: https://github.com/JPaxos/JPaxos
[34] L. Rodrigues and M. Raynal, "Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication," *IEEE Trans. on Knowl. and Data Eng.*, vol. 15, no. 5, pp. 1206–1217, 2003.
[35] "Intel's 3D XPoint," https://software.intel.com/en-us/articles/3d-xpoint-technology-products.

Jan Kończak is currently pursuing a PhD degree and working as a research assistant in the Institute of Computing Science, Poznań University of Technology, Poland, where he also received BS and MS degrees in Computer Science, in 2011 and 2012 respectively. His research interests include fault tolerant distributed algorithms, transactional memory, state machine replication and group communication systems.

Paweł T. Wojciechowski received the Habilitation degree from Poznań University of Technology, Poland, in 2008, and the PhD degree in computer science from the University of Cambridge, in 2000. He was a postdoctoral researcher in the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, from 2001 to 2005. He is currently an associate professor in the Institute of Computing Science of Poznań University of Technology. He has led many research projects and coauthored dozens of papers. His research interests span topics in concurrency, distributed computing, and programming languages.

Nuno Santos is a Software Engineer at Raw Labs, a startup developing tools for data exploration, analysis and mining. In 2012, he received a PhD in Computer Science from the EPFL, Switzerland, on the topic of analysis, implementation and evaluation of Paxos-like consensus algorithms. He received a BSc in Mathematics and a MSc in Computer Science from the University of Coimbra, Portugal, in 2000 and 2003, respectively. From 2003 to 2006 he worked for one year as a software engineer at Wit-Software, Portugal, and for two years at the European Organization for Nuclear Research (CERN), Switzerland.

Tomasz Żurkowski is currently a Software Engineer at Google, working on core search infrastructure. He received BS and MS degrees in Computer Science from Poznań University of Technology, Poland, in 2011 and 2012 respectively.

André Schiper graduated in physics from the ETHZ in Zurich in 1973 and received the PhD degree in computer science from the EPFL in 1980. He has been a professor of computer science at EPFL since 1985 (retiring in 2014) leading the Distributed Systems Laboratory. His research interests are in the areas of dependable distributed systems, middleware support for dependable systems, replication techniques (including for database systems), group communication, distributed transactions, and MANETs. He was a member of the editorial boards of Springer-Verlag's Distributed Computing (2003-2014), the IEEE Transactions on Dependable and Secure Computing (2004-2008), and Inderscience's International Journal of Security and Networks (2005-). He is a member of the IEEE.