Jan Kończak

Recovery algorithms in State Machine Replication with volatile and non-volatile main memory

Doctoral Dissertation

Thesis Advisor:
Paweł T. Wojciechowski, Ph. D., Dr. Habil., Assoc. Prof.

Poznań, June 2022

List of publications

Jan Z. Kończak, Nuno Santos, Tomasz Żurkowski, Paweł T. Wojciechowski, and André Schiper. JPaxos: State Machine Replication based on the Paxos protocol. Technical Report EPFL-IC-TR-167765, EPFL, July 2011

Jan Z. Kończak, Paweł T. Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. Recovery algorithms for Paxos-based State Machine Replication. *IEEE Transactions on Dependable and Secure Computing*, 18(2):623–640, March-April 2021 (published as early access in July 2019)

Jan Z. Kończak and Paweł T. Wojciechowski. Failure recovery from Persistent Memory in Paxos-based State Machine Replication. In *40th International Symposium on Reliable Distributed Systems, SRDS '21*, September 2021

Outside of the main scope of the thesis, the author participated in the following work:

Bartosz Bosak, Jan Kończak, Krzysztof Kurowski, Mariusz Mamoński, and Tomasz Piontek. Highly integrated environment for parallel application development using QosCosGrid middleware. In *Building a national distributed e-Infrastructure - PL-Grid - scientific and technical achievements*, volume 7136 of *Lecture Notes in Computer Science*, pages 182–190. Springer, 2012

Paweł T. Wojciechowski and Jan Kończak. A formal model of crash recovery in distributed Software Transactional Memory. In *Proceedings of the 4th Workshop on the Theory of Transactional Memory, WTTM '12*, July 2012

Jan Kończak, Paweł T. Wojciechowski, and Rachid Guerraoui. Ensuring irrevocability in wait-free Transactional Memory. In *11th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT '16*, March 2016

Jan Z. Kończak, Pawel T. Wojciechowski, and Rachid Guerraoui. Operation-level wait-free Transactional Memory with support for irrevocable operations. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3570–3583, 2017

Abstract

State Machine Replication (SMR) is a well-established method to provide high availability and strong consistency when failures must be taken into account. In SMR a deterministic application called state machine is run on multiple machines, and each state machine is provided with the same sequence of commands, what makes the state machines indistinguishable by the clients that submit commands. The hard part of building a SMR system is to order commands, and for this usually a consensus algorithm is used. This thesis targets specifically Paxos, which is the most popular protocol of such kind today. The goal of SMR systems is to provide resilience for failures, so among others resilience for machine crashes. This is usually paired with ability to recover the crashed machines. Providing such ability to a consensus algorithm requires persisting the progress on storage that survives crashes, for upon recovery a replica must restore the (lost on crash) processing state, and this must be possible also in case all replicas crash. However, the writes to storage must be synchronous and the performance of SSDs that are most commonly used as the storage is limited, so supporting recovery decreases the performance of SMR systems.

This thesis explores two ways of supporting crash recovery in Paxos-based SMR systems that allow for performance closer to the performance of a system without recovery support than the existing solutions. The first way is to assume that at any point in time some majority is up. Then, the state of processing that a replica must restore at recovery is preserved among the replicas that are up. Two algorithms presented in the thesis, ViewSS and EpochSS, take advantage of it and recover by first informing other replicas about the recovery and inquiring them about current system progress, and then fetching the required data from them. For correctness, in these algorithms the recovering replica must learn which state it has to restore, and other replicas must handle correctly messages delivered after recovery, but sent by the recovering replica before it crashed.

The other way of providing recovery makes no assumptions on number of simultaneous crashes, and takes advantage of only recently marketed persistent memory (pmem). Whereas ViewSS and EpochSS extend the Paxos algorithm, the solutions that use pmem, mPaxos and mPaxosSM, are complete SMR systems that base on Paxos, and store the critical data in non-volatile main memory. The first, mPaxos, requires the state machine to fulfill standard requirements: it must execute requests deterministically, create snapshots regularly and be able to recover from a snapshot. The second, mPaxosSM, has different requirements: the state machine must obviously execute requests deterministically, but instead of making and recovering from snapshots, it has to use pmem to continuously persist its state, and be able to stop processing on demand and resume operation from pmem contents fetched from another replica. mPaxos and mPaxosSM allow for faster recovery, and keep data in the persistent main memory rather than duplicate the data in the main memory and in a storage device, like the competing methods of supporting recovery do. Moreover, mPaxosSM gets rid of the need to create snapshots periodically, what improves the overall performance. All proposed methods of supporting recovery are compared against each other and the well-known method that writes all progress to storage, and for each method the flow of recovery and impact on the system performance is discussed.

Finally, prototype implementations of SMR systems using ViewSS, EpochSS, mPaxos and mPaxosSM are compared experimentally against a system that does not support recovery (no-crash) and a system that supports recovery by writing the vital data to storage that survives crashes (FullSS). First, no-crash, FullSS, ViewSS and EpochSS are evaluated on hardware that is not equipped with pmem, then FullSS, EpochSS, mPaxos and mPaxosSM are evaluated on hardware with

pmem. The evaluation shows that pmem-based systems recover faster than the other, but, especially in mPaxosSM, a recovered replica takes a longer time to catch up the progress it missed while being down. During fault-free period ViewSS and EpochSS retain the throughput of no-crash system, and mPaxos is either better than or on par to FullSS that uses pmem as storage, depending on the workload. Throughput of mPaxosSM, unlike the other systems that are limited by network bandwidth, is limited by pmem write bandwidth that is surprisingly low when the writes need to be performed from a single thread. However, mPaxosSM still outperforms other systems whenever creating snapshots periodically incurs sufficiently high performance drop, what is experimentally shown in the thesis for an example application. Typically, upon recovery of one replica, some other replica must send to the recovering replica progress made by the system at least from the crash to the recovery. The evaluation shows that with some exceptions this has an acceptably low impact on overall system performance.

Contents

Chapter 1

Introduction

An average person considers as a fundamental truth that the services in the internet are always ready to answer one's request. When the websites of mainstream search engines or social media stop responding, many people immediately assume that their internet service provider is to blame, for it is unthinkable that a website such as this is down. At the same time it is clear for everybody that software and hardware crashes. But software developers, thanks to ongoing research effort, got very good at masking the crashes. Services are run on multiple machines spread across the world, so that even the failures that take down all the computing resources in a large area are not able to disrupt availability. Generally speaking, it is the true image of how things are going. But there are the details: the details that decide how good it works. And what happens in the rare corner cases, when things go bad.

Computer services can stay available despite crashes because they are distributed systems: multiple machines connected with the network. As long as a sufficient part of such system is reachable, one expects it to answer. Obviously, a request is sent to one of the machines. But, will each machine answer identically? The intuition says that it should. This, however, is usually not true. Machines need to synchronize to provide a consistent answer, and this turns out to be expensive. Not too expensive to afford, but simply most of the tasks the computer systems do can live without strong consistency. For instance, no one will notice that different people see reviews in a different order in an online shop. Buy one will notice that one bought the last item in stock and suddenly got a message that the item has been sold to somebody else, who contacted another machine in the same distributed system running the shop. Surprisingly, this is still considered by many business models fine as long as a compensation will satisfy the unfortunate buyer.

However, there are cases where different machines of the same distributed system must provide consistent answers, or at least providing inconsistent answer is too inconvenient. Continuing the online shop example, when the parcel is dispatched, a distributed system must not assign the same slot for two parcels in a parcel locker – even if handling each parcel is processed by a different machine of the distributed system. While an honest person would probably report this upon noticing the problem, solving it would be overly bothering. People usually notice abnormal cases in the physical world, but the computers follow blindly the instructions upon processing digital resources. If a distributed system, due to the lack of consistency, assigns exclusive access to a resource to two programs, the programs will just assume the exclusive access and continue without noticing that they cause havoc by operating on the same resource concurrently.

Highly available and fault tolerant strongly consistent distributed systems are surprisingly hard to create from the theoretic point of view. To tell that a system stands up to the expectations one has first to define formally both the expectations (system properties) and the world in which the system is run (system model). It has been proven that in the most general model, in the asynchronous system, strong consistency is impossible if machines can crash. Therefore, the partially synchronous system model was created – one that is as permissive to failures as the asynchronous, but requires periods of operation uninterrupted by failures. With such assumptions, there exist algorithms that can be used to build distributed systems that act equivalently to a centralized system (strong consistency) and progress as long as at least half of the machines are operational.

A very popular way of building strongly consistent distributed systems is the State Machine Replication (SMR). It operates on a simple principle: all machines in the distributed system handle the same requests in identical order. In other words, a centralized system is duplicated on multiple machines. Obviously, this does not give performance benefit over a centralized system. But not this is the goal of SMR – it allows continuous operation despite crashes. The main challenge of building a SMR system is to order the requests equally on all machines. This task is usually pushed on to a consensus algorithm, among which the Paxos algorithms gained a large popularity. Paxos, or rather its MultiPaxos variant, can be fed with client requests and it assigns system-wide consecutive numbers for the requests. The commands contained in the client requests are then executed by the replicated application, which in SMR is called State Machine, for it has to fulfill certain requirements – be deterministic and answer a command only on basis of the command and all past commands.

There is a hard performance limit for SMR systems: such system cannot process more client requests than the state machine is capable of handling. Some SMR systems are very close to this limit. However, there are known issues that slow down SMR systems, and most of them have to do with providing fault tolerance. Whenever one machine is separated from the network for a longer while, it has to learn what it missed upon restoring connection to other replicas. It is not always possible to just sent it all the commands that it missed, for the number of such commands grows unbounded with the duration of the network split. Hence, a snapshot of the state of the state machine is sent, but to have it, the SMR system must regularly receive such snapshot from the state machine. And creating it impacts the performance. The throughput of the state machine is one possible limit of performance, the other is the consensus protocol. Paxos is usually faster than the state machine if, however, it does not support recovery. Ability to continue work despite crashes is one thing, and ability to recover the crashed machines is another. The former is a fundamental property of Paxos, the latter is an option. The ordinary way of recovering a crashed replica is to restore from a storage device that survives crashes (such as an SSD) the state of processing at the moment of crash. This, however, requires writing down the state during normal operation of the replica, and doing so negatively impacts the performance of algorithms such as Paxos.

This thesis is devoted to lowering the cost of supporting crash recovery as well as evaluating the proposed solutions. Four ways of lowering the cost of supporting recovery are proposed: ViewSS, EpochSS, mPaxos and mPaxosSM.

In SMR, each replica does the same work. So, it is possible to recover one replica by copying the state of another. This is what ViewSS and EpochSS do. However, one has to recover the state either equal or more recent than that at the moment of crash, so before transferring state from a running replica to the recovering one, the latter has to learn how recent state it needs to recover. This can be achieved by asking a majority of replicas, and doing it correctly is one of the key challenges of ViewSS and EpochSS. The other is dealing with messages that were sent by the recovering replica before crash, but are not yet delivered at recovery – for when the recovering replica queries other replicas on what it needs to recover, the messages that possibly affect the answer are not yet known to the queried replicas. This method of providing recovery has almost no impact on performance of the SMR system, but it has one drawback: at recovery a majority of replicas has to be queried, and if half of the replicas or more are down, the recovery is no longer possible.

It is not possible to support recovery of replicas if more than a majority is crashed at a time if the replicas do not write down what they do to the stable storage, for the obvious reason that the replicas in the crashed majority might have been the only that knew about some ordered command. To support any number of simultaneous crashes and work around the latency problem of mass storage devices, one would have to find a more efficient way to store the required data. Recently a new class of memory called Persistent Memory (pmem) emerged. mPaxos and mPaxosSM attempt to leverage pmem to reduce the cost of recovery support that affects failure- and recovery-free operation, and additionally reduce the duration of recovery. mPaxosSM goes a step further and changes the assumptions the state machine has to fulfill. Namely, it is required to persist its state across crashes in pmem. Doing so allows to get rid of the need to create snapshots periodically by the state machine, and so to further improve overall performance.

Together with the ideas of how to reduce the cost of supporting recovery, a complete Paxos-based SMR framework is presented in this thesis. Such framework is implemented in JPaxos, together with ViewSS, EpochSS, mPaxos and mPaxosSM (the last being actually a fork rather than part of JPaxos). Having a prototype of each idea as a variant of the same system allows for a fair experimental comparison, and so all the systems were evaluated to test how well they perform in failure-free periods, as well as at crashes and recoveries. The evaluation encompasses also the process of catching up a recovered replica with the rest of the system, for the systems (apart from those based on ViewSS and EpochSS) recover the state a replica had at crash, and so the recovered replica must catch up. The tests were run in a range of workloads – the replicated state machine was either a key-value data store or a simple echo service (returning back to the client the data it got), the tests were repeated for different sizes of request and snapshots, and, where it had a rationale, different number of replicas in the system. Following the evaluation, conclusions and tips on choosing the right method of supporting recovery for a given workload are provided.

Chapter 2

System model and definitions

2.1 Basic definitions

In this thesis I consider a *distributed system* to be a number of *nodes* interconnected by network *links* such that the nodes are working together to reach one or more common goals.

A node is a piece of computing equipment that is capable of running one or more *processes*. The node (and so a process) can communicate with other nodes by sending *messages* via the links. *Machine* is a synonym for node. A process is an instance of a computer program that realizes a specified algorithm.

A node can be either up or down, with the common sense meaning of up and down referring to computer equipment. A *crash* of a node is an event when it becomes down. A node can start at any time by changing state from down to up[1]. A process on a node can be either up, or down, or recovering. At time $t = -\infty$ the process is down, then at some time it *starts for the first time* (this is called *initial start* or a *fresh start*). A process can *crash* and become down. When a node crashes, all processes running on the node crash. A process alone can also crash (e.g., due to a software bug or exceeding resource limits). A process that is down and has previously crashed can be *restarted*. Such process must then run a *recovery* procedure and until it is finished, the process is *recovering*. Once the process recovers, it becomes up again. This is presented in Fig. 2.1.

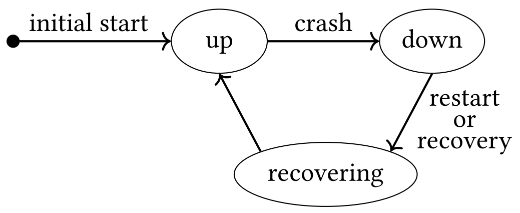


Figure 2.1: Possible process states and transitions

As mentioned before, the nodes are connected by links. In real computer systems the machines are typically connected to a backbone network using a single link of a limited bandwidth. *Uplink* is used to refer to the traffic on this link in direction from the node to the backbone (that is, messages sent by this node to other nodes), and *downlink* refers the traffic on this link in the opposite direction. Links, uplinks and downlinks can be characterised by their *bandwidth*, that is the number of bits per second that can be sent via the link. Bandwidth includes all protocol overheads.

A message can be sent to a single destination. This is called *unicast*. A message can also be *broadcast* – that is sent to all nodes.

2.2 Failure types

The components of the system can fail in a variety of ways.

Links can fail by:

- Delaying messages. The time period for which each message is delayed is finite, and no bound may be known for the delay.
  Delaying messages can cause reordering of the messages, i.e., when node n_1 sends messages m_1 and m_2 (in that order), then n_2 can receive m_2 before m_1. A broadcast message has usually individual delay for each destination.

[1]Typically after some external intervention, such as getting powered on (again), or manual or automatic restart of a hanged operating system or hardware.

- Losing messages. A message that was broadcast can be lost for some destinations while arrive at other destinations.
- Duplicating messages. A message sent once can arrive multiple times at its destination.

In real networks all these failures can affect network packets. Transport protocols may (as e.g., TCP) or may not (as e.g., UDP) provide stricter guarantees.

In this thesis message integrity problems are not considered – it is assumed that messages always arrive unmodified. This is usually guaranteed in real networks (unless explicit malicious actions to alter the messages are considered).

Typically all link failures must be considered in a distributed system, for it is not feasible to detect such failures reliably in real systems.

Process failures are typically divided into non-byzantine and byzantine. The latter are failures where a process does not act according to the algorithm it is supposed to. Such failures are not considered in this thesis. Two models of non-byzantine failures of processes are common: *crash-stop* and *crash-recovery*. In the former, once a process crashed, it never starts again. In the latter, a process that crashed can start again. Processes store the data in *volatile* storage and *stable* (also called *persistent* and *non-volatile*) storage. When a process crashes, it loses all data that was stored in volatile storage as well as its processing state. Only the data that has been stored persistently remains available at recovery.

2.3 Asynchronous, synchronous, and partially synchronous system

To argue about distributed systems, especially on its progress, it is often needed to define time constraints on possible message delays and relative processing speed of the processes. A *synchronous* system has an upper limit on message delay and relative processing speed. *Asynchronous* system is missing those limits.

In real computer systems there are time periods when the system acts as a synchronous one. However, in real systems there also occur time periods of asynchrony. Some problems, including those vital for the thesis, are insolvable in asynchronous systems and possible in synchronous systems. To allow for practical solutions to such problems, there is a need for system models that allow for building algorithms solving the problems when system acts synchronously and do not violate correctness when the system acts asynchronously. *Partially synchronous* systems were proposed to this end [DLS88]. In particular, a partially synchronous system can be defined as an asynchronous system up to an unknown time t, and synchronous system afterwards. While in theory such system remains synchronous forever, in practice it is sufficient for the system to remain synchronous for duration long enough to terminate a disputed distributed algorithm. Such system model is also referred as *eventually synchronous*.

2.4 Failure detector

A process in a distributed system may have access to a *failure detector* that answers whether other processes are up or down. The failure detector does not have to answer correctly. Notions of accuracy and completeness restrict how wrong the answers can be [CHT96]. For the problems discussed here a failure detector called *eventually weak* suffices: such a failure detector guarantees that eventually every process that is down will be recognized as down by all, and at least one process that is up will eventually be considered as up by all. Equivalent failure detectors are often called *omega* (or Ω) failure detectors [ADGFT03].

2.5 Correctness

To assess if a distributed algorithm works fine a correctness criterion has to be defined. Two popular properties are suitable for this thesis: strong consistency and linearizability. *Strong consistency* compares a distributed system with a reference centralized system (i.e. a single processor system). If any outcome of the distributed system is also possible in the centralized system, then the distributed system is strongly consistent. *Linearizability* extends that by the requirement on preserving the real time order between the systems [AW94].

2.6 Progress and termination

Technically, a distributed system that just does nothing at all, and so does nothing to solve the problem it is supposed to solve, is correct (strongly consistent and linearizable). Therefore it is required from any algorithm to satisfy some progress property. Usually progress is defined by a guarantee that each operation of the system will (at least eventually) terminate. The exact notion of progress (and operation) is problem-dependant. It is worth noting that in a distributed system even if some process reaches termination it still keeps on running, for other processes in the system may expect to get an answer to a message from the process in order to reach termination as well. Usually a process cannot detect whether all other processes either terminated or remain down indefinitely, since partially synchronous system as well as asynchronous system with a eventually accurate failure detector can detect failures only eventually.

2.7 Consensus

Consensus is a fundamental problem of distributed computing of making multiple processes agree upon something. With an algorithm that reaches such agreement it is easy to build sequentially consistent distributed systems. However when failures are allowed consensus is surprisingly complex.

2.7.1 Definition

Assume a set of N processes $P = \{p, q, ...\}$. The processes can be either *correct* or *faulty*[2]. A correct process is a process that eventually stays up (i.e., for a correct process p there is a time t such that at t the process p is up and p does not crash after t). Each process can *decide* a value exactly once. Before a process decides, it may *propose* a value to be decided.

Consensus is reached if the following conditions are met:

- $[T]$ every correct process eventually decides,
- $[V]$ a process can decide v iff some process proposed v,
- $[A]$ if any process p decided v_p and any process q decided v_q, then $v_p = v_q$.

[2]Some articles call these processes *good* and *bad*.

T stands for termination (forces processes to decide), V for validity (forces processes not to decide a predefined value), and A for agreement (forces processes to agree).

2.7.2 Impossibility result & its workarounds

Consensus is impossible in an asynchronous system where one process can crash, even when the links are reliable (i.e., each message is always delivered once) [FLP85].

To create an algorithm that solves consensus some of these conditions must be weakened. Getting rid of process crashes is not practical, hence the asynchrony of the system must be restricted. One can either assume availability of a failure detector ([CHT96]) in an asynchronous system or replace the asynchrony with partial synchrony ([DLS88]) – both these approaches allow creating algorithm that solve consensus. In real systems consensus algorithms always guarantee correctness and tolerate failures, while termination is only guaranteed under extra conditions.

2.8 Total Order Broadcast

Total Order Broadcast (TOB), also known as *atomic broadcast*, is another fundamental problem in distributed systems. Sending a message via TOB is called *TOB-casting* the message. All TOB-cast messages must be delivered in the same order by all processes. Receiving the message is called TOB-deliver.

Formally uniform TOB is described by the following conditions:

- [T] If a correct process sends a message, the message will be eventually delivered
- [V] A process can deliver a message if it was sent by some process
- [A] If some process delivers a message, then all correct processes deliver the message.
- [O] If some process delivers message m_1 before m_2, then no other process delivers m_2 before m_1

T stands for termination, V for validity, A for agreement and O for total order.

TOB is equivalent to consensus. Reaching consensus using TOB is trivial – the first delivered message becomes the decision. Inversely one can build TOB using consensus is as follows: consecutive consensus algorithm instances are run and each process maintains a logical clock. A process that wants to TOB-cast m increments its logical clock and reads it to l, then reliably broadcasts (l, m). Upon receiving such broadcast each process updates its local clock and adds (l, m) to list of messages to be ordered. Processes propose in the lowest not yet decided consensus instance one of the messages with lowest l. The values decided in subsequent consensus instances are TOB-delivered.

Typically implementations of TOB explicitly use multiple instances of consensus (and skip the partial ordering imposed by reliably broadcasting the message with a logical clock, that in theory is just needed to prevent starvation of some message and thus guarantees termination).

2.9 State machine

State machine (SM) is a formalism that embodies computer programs following the ubiquitous request-response design pattern – SM covers programs that work by answering requests one after another in a deterministic fashion. Deterministic means here that the response depends only on previous requests, and not, for instance, on time or randomness.

To define a state machine, one needs a set of *states* $S = \{s, ...\}$, *commands* $C = \{c, ...\}$ and *results* $R = \{r, ...\}$. Then the state machine is defined by its initial state s_0 and transition function $\delta(s_i, c) \longrightarrow (s_{i+1}, r)$. The state machine always starts from the initial state and can *execute* a command c to produce a result r and change its state from s_i to s_{i+1}. A state s encoded as a transferable array of bytes is referred to as a *snapshot* (or a *checkpoint*) of the state machine.

In the wild, many services (e.g., web services) are in fact state machines, even if this is not explicitly stated by their authors. For instance, multiple traditional database software systems either are or can be configured to fulfill SM requirements.

2.10 State Machine Replication

State Machine Replication (SMR) binds together state machine and Total Order Broadcast to provide availability of the state machine despite failures. In SMR the same state machine is run on multiple nodes, called *replicas*. SM commands are ordered using Total Order Broadcast, and then executed, in order, on each SM. All SM start from the same initial state, therefore each replica after the same sequence of commands has the same state (hence the name replica).

A process that wants to execute a command is called a *client*. To have its command c executed, the client must send a *request* containing c using Total Order Broadcast. Since all replicas that eventually receive the command c contained in the request execute it and produce (the same) result, it is usually implementation-defined which replica(s) send a *reply* containing the result back to the client.

A SMR system consists of N replicas denoted $p, q, ...$. Typically each replica is assigned an individual number called *identifier* (*id*), in range $[1, N]$, to tell the replicas apart. In this thesis the set of replicas is predefined and does not change (the replica group is static). Alternative approach is discussed in Chapter 4.

2.11 Quorum and majority

Assume that the system consists of a set P of N processes, $P = \{p_1, p_2, ..., p_n\}$. For such set P one can construct a set of *quorums*, that is a nonempty set of processes sets $\mathcal{Q} = \{Q_1, Q_2, Q_3, ...\}$ such that every two quorums (that is, elements of $\mathcal{Q}$) have at least one common process ($\forall_{Q_i \in \mathcal{Q}} \forall_{Q_j \in \mathcal{Q}} Q_i \cap Q_j \neq \emptyset$). The set of quorums $\mathcal{Q}$ can be arbitrary as long as the conditions hold.

Quorums are used to store permanently information in distributed system in the following way: when a process p wants to make some information durable, then it sends it to some quorum Q. Once p gets acknowledgements from all processes in Q, any process q can query any quorum Q' for such an information and is guaranteed to learn it, for there is at least one process that is in both quorums (by definition $Q \cap Q' \neq \emptyset$).

For most purposes the best choice of the set of quorums is a set of all *majorities*, that is a set of more than half of the processes. One can define such set of quorums either as $\forall_{Q \in \mathcal{Q}} |Q| > \frac{|P|}{2}$ or $\forall_{Q \in \mathcal{Q}} |Q| = \left\lfloor \frac{|P|}{2} + 1 \right\rfloor$. The latter is usually preferred, as a process typically awaits messages form any quorum, so no quorum of more than

$\left\lfloor \frac{|P|}{2} + 1 \right\rfloor$ is going to be used in practice. In this thesis only such quorums are used. Use of variously sized quorums for certain benefits has been proposed [HMS16].

2.12 Notation in pseudocodes

For clarity, the notation used in pseudocodes is explained below.

$\perp$	represents an empty (aka null, blank, unset) value
$a = 1$	tests if a equals 1
$a \leftarrow 1$	sets a to 1
Msg$\langle a, b\rangle$	message Msg carrying two informations – a and b
(a, b)	a tuple of a, b
$\{a, b\}$	a set containing a, b
$\{i \mid i > 5\}$	a set containing all i that satisfy $i > 5$
$\lvert P\rvert$	number of elements in the set P
$max(S)$	the largest element in set S
$S \leftarrow \{(a, b)\}$	S is assigned a set containing a tuple
$b : (\_, b) \in S$	b such that there is a tuple in S which has b on second position notice that _ stands for wildcard
a mod b	modulo operator (reminder of division of a by b)
$S \leftarrow S \cup \{a\}$	adds element a to set S
$S \leftarrow S \setminus \{a\}$	removes element a from set S
send(Msg, p)	send message Msg to process p
send(Msg, P)	send message Msg to all processes in set P

The operator $t < t'$ on tuples $t = (e_1, e_2, e_3, ...)$, $t' = (e'_1, e'_2, e'_3, ...)$ returns true iff $e_i < e'_i \wedge \forall_{j<i}\, e_j = e'_j$.

Chapter 3

Paxos-based State Machine Replication

While the word Paxos became a highly ambiguous term in distributed systems, it usually refers to a consensus algorithm presented by Leslie Lamport in a technical report published in 1989 [Lam89], that was submitted to ACM TCS in 1990 and published eight years later [Lam98]. The article (and TR) called *Part-time parliament* has been written in an unusual form, mixing computer science with entertaining storytelling that somehow repelled reviewers and still confuses some readers accustomed to a well-established structure and wording of computer science articles. Much confusion about Paxos stems from the fact that [Lam98] contains textual description of several versions of a consensus algorithm, ranging from simple versions (preliminary, basic, complete) to a practical version (agreeing upon a series of decisions) with multiple optimisations. Only the basic version has a formal (and proven) pseudocode, while the word Paxos is usually used to refer to the practical version.

Paxos has properties and complexity identical to some consensus algorithms predating it (such as Viewstamped Replication [OL88]), but gained much more popularity than others.

3.1 The Basic Protocol – SinglePaxos

To present Paxos algorithm, first the exact algorithm of "Basic Protocol" from [Lam98] is presented (with pseudocode notation adjusted to be more suitable for this thesis) and explained. Before Paxos is presented in detail, a brief introduction to its basic terms is given, as follows: in Paxos the value to agree upon by a set of processes P is chosen through a series of numbered *ballots*, where each ballot is a referendum on one value selected by the process that started the ballot. Such process is called *leader*. To start a ballot b, a process p sends a NextBallot message and awaits LastVote messages from a quorum. Once the quorum replies, p becomes the leader of b, selects a value v to vote upon and broadcasts it in a BeginBallot message. Processes *vote* for v in b by sending a Voted message to p. When p receives Voted from a quorum, then consensus has been reached, and p broadcasts a Success message with v. If p crashes, other processes must start a new ballot (until some process eventually succeeds).

3.1.1 The algorithm

Algorithm 1 SinglePaxos (the Basic Protocol)

```
Each process $p$ (with number *id*) keeps in stable storage the following data:
1:  $outcome \leftarrow \bot$          once agreed upon, the value is written here
2:  $nextBal \leftarrow -\infty$         highest acknowledged ballot number
3:  $lastTried \leftarrow -\infty$       ballot number that $p$ attempted to begin
4:  $prevBal \leftarrow -\infty$         ] last voted value and ballot in which $p$ cast a vote
5:  $prevDec \leftarrow \bot$          ]
Process $p$ keeps in volatile storage the following data:
6:  $status \leftarrow idle$          whether $p$ is *trying* to start a ballot, or *polling* for the answers
                                to ballot it started, or neither (*idle*)
7:  $prevVotes \leftarrow \emptyset$        when *trying*, answers to start ballot request are stored here
8:  $quorum \leftarrow \emptyset$           when *polling*, the set of processes that answered start ballot
                                request (and also a quorum of processes)
9:  $voters \leftarrow \emptyset$           when *polling*, the set of processes that voted in this ballot
10: $decree \leftarrow \bot$           when *polling*, the value proposed in this ballot by $p$
```

```
11: **procedure** TryNewBallot
12:     $\mathit{lastTried} \longleftarrow b$ **s.t.** $b > \mathit{lastTried}$ **and** $b \bmod |P| = \mathit{id}$
13:     $\mathit{status} \longleftarrow \mathit{trying}$
14:     $\mathit{prevVotes} \longleftarrow \emptyset$
15:     send(NextBallot$\langle \mathit{lastTried} \rangle$, $P$)
16: **upon** NextBallot$\langle b \rangle$ from $q$ **s.t.** $b \geq \mathit{nextBal}$
17:     $\mathit{nextBal} \longleftarrow b$
18:     send(LastVote$\langle b, \mathit{prevBal}, \mathit{prevDec} \rangle$, $q$)
19: **upon** LastVote$\langle b_q, b_v, v \rangle$ from $q$ **s.t.** $b_q = \mathit{lastTried}$ **and** $\mathit{status} = \mathit{trying}$
20:     $\mathit{prevVotes} \longleftarrow \mathit{prevVotes} \cup \{(q, b_v, v)\}$
21: **upon** $\mathit{status} = \mathit{trying}$ **and** $Q \subseteq \{q \mid (q, \_, \_) \in \mathit{prevVotes}\}$ **where** $Q$ is a quorum
22:     $\mathit{status} \longleftarrow \mathit{polling}$
23:     $\mathit{quorum} \longleftarrow Q$
24:     $\mathit{voters} \longleftarrow \emptyset$
25:     $\mathit{decree} \longleftarrow v$ **s.t.** $\begin{cases} b_{max} = -\infty : v \text{ is a new value} \\ b_{max} \neq -\infty : (\_, b_{max}, v) \in \mathit{prevVotes} \end{cases}$
            **where** $(\_, b_{max}, \_) \in \mathit{prevVotes}$ **and** $\forall (\_, b, \_) \in \mathit{prevVotes} : b_{max} \geq b$
26:     send(BeginBallot$\langle \mathit{lastTried}, \mathit{decree} \rangle$, $Q$)
27: **upon** BeginBallot$\langle b, v \rangle$ from $q$ **and** $b = \mathit{nextBal}$ **and** $\mathit{nextBal} > \mathit{prevBal}$
28:     $\mathit{prevBal} \longleftarrow b$
29:     $\mathit{prevDec} \longleftarrow v$
30:     send(Voted$\langle b \rangle$, $q$)
31: **upon** Voted$\langle b \rangle$ from $q$ **and** $b = \mathit{lastTried}$ **and** $\mathit{status} = \mathit{polling}$
32:     $\mathit{voters} \longleftarrow \mathit{voters} \cup \{q\}$
33: **upon** $\mathit{quorum} \subseteq \mathit{voters}$ **and** $\mathit{status} = \mathit{polling}$ **and** $\mathit{outcome} = \bot$
34:     $\mathit{outcome} \longleftarrow \mathit{decree}$
35:     send(Success$\langle \mathit{outcome} \rangle$, $P$)
36: **upon** Success$\langle v \rangle$ from $q$ **and** $\mathit{outcome} = \bot$
37:     $\mathit{outcome} \longleftarrow v$
```

3.1.2 The algorithm explained

Paxos starts when some process, say p, calls the TryNewBallot procedure (line 11). The procedure can be also started by p once it assumes that the process which was responsible for the most recent ballot has crashed.

As the name suggests, the TryNewBallot procedure attempts to start a new ballot, and for this purpose p must first select a *ballot number*, say b. When some process took part a ballot number b', then it will never take part in any ballot that has a lower number than b' (line 27). Therefore it makes sense to select b that is greater than *nextBal* and *lastTried* (the pseudocode, following [Lam98], checks only *lastTried*; if b is lesser then *nextBal*, then the attempt to start a new ballot is very likely to fail). Ballot numbers must be divided among processes, so that no two processes select the same number. It suffices to say that a process p can start ballot number b such that b modulo number of processes yields p's id.

With b selected, p broadcasts it in a BeginBallot message. Upon receiving such message with b larger than its *nextBal* (line 16) a process q records not to take part in any ballot that has number lesser than b and sends back to p a message LastVote (with

b to identify it as an answer to the specific BeginBallot). If q voted in any previous ballot b_v, LastVote contains also the number b_v and the value voted in ballot b_v.

Process p awaits LastVote messages from any quorum. With LastVote from the quorum, p can select the value to be voted for in ballot b (line 25). Notice that thanks to nonempty intersection of any two quorums, it is guaranteed that if all processes in some quorum voted in some ballot $b' < b$, then p will learn that ballot b' existed as well as p will know what value v was voted. Since p cannot tell whether a quorum voted for the value or not, p must propose the value v in its ballot. It is possible that processes informed p about several past ballots; in such case p must select the value voted for in the most recent ballot. If no process indicates a past ballot, then p selects its own proposal v. This guarantees that if a quorum voted for some value v in ballot b', then any subsequent ballot will vote for v as well, and so v will be ultimately agreed upon by all.

Once p selects the value v, p sends it (together with b) to the processes that answered the BeginBallot (*quorum*, see line 23). The processes, unless crashed or accepted a higher ballot number in the meantime, record b and v as the last vote and send Voted back to p.

When p receives Voted from all processes in *quorum*, then it learns that v is deemed to be voted for in any subsequent ballot, so it has been agreed upon. So, p records that v has been decided and broadcasts a message Success with v to all. Any process that receives the Success message decides v as well.

3.1.3 Phases

In the Paxos algorithm, usually two *phases* are recognized: *ballot initialization* phase and *voting* phase (or simply *first* and *second* phase). The first begins when a process p starts the TryNewBallot procedure, and ends once LastVote messages from a majority is received by p. This phase essentially is a *leader election* merged with querying a quorum for the most recent voting. The second phase starts by broadcasting by the leader p a BeginBallot message and ends either when consensus is reached or when another process attempts to become leader. So the second phase stands for the leader proposing a value and the processes voting for it, as well as disseminating the voting results (decision).

3.1.4 Helpful deviations from the Basic Algorithm

This thesis uses several changes in this algorithm. Quorums are replaced by simple majority. BeginBallot is broadcast, instead of being sent to a quorum. Consequently, when a process p receives a BeginBallot with ballot b from q before NextBallot from q, then it assumes that q is the leader of b (having already gathered enough LastVote messages from a majority not including p), updates its own *nextBal* to b and votes for whatever q proposed.

Another change used in this thesis is that Voted is broadcast rather then sent to the leader only. This way any process keeps track whether a ballot succeeded, and the Success messages are not needed anymore. The reason for such change is as follows: from CPU point of view the leader has less work to do, and from network point of view the leader needs to send less data. In practice the leader is the most busy process, and in the typical full mesh (all to all communication) topology is the process that needs to send most data, so the change is beneficial.

Another common deviation from the algorithm above is that when a process receives NextBallot and it already decided, then it sends back Success rather than LastVote. However, this thesis deals with such cases differently (see Section 3.2.3).

3.2 MultiPaxos

Distributed systems usually need to agree upon an (unbounded) series of values. But the Basic Protocol presented earlier agrees upon a single value and terminates. It is therefore commonly referenced to as *SinglePaxos*. A simple but inefficient way of getting a series of values is to start a new *instance* of consensus algorithm such as the SinglePaxos as soon as the previous terminates. But it makes much more sense to create an algorithm basing on SinglePaxos and explicitly design it to agree upon consecutively numbered values. Algorithms of this kind are referenced to as *MultiPaxos*. In the first Lamport paper on Paxos [Lam98] such an algorithm, together with some non-trivial optimisations, is presented as "Multi-Decree Parliament".

This thesis presents another version of MultiPaxos that is used to develop recovery algorithms. The version leaves out recovery and reflects crash-stop version of JPaxos [JPa22]. The algorithm redefines and renames messages: Prepare and PrepareOK are used instead of NextBallot and LastVote, moreover Propose and Accept replace BeginBallot and Voted. The intended use case of the algorithm is to order commands for state machine, therefore in place of values the algorithm decided *commands*. Also, *outcome* is not used explicitly; when a process learns that a command is agreed upon, the process *issues* it, and it is assumed that a process can check whether it already issued a command in instance *i*. In the pseudocode, P stands for the set of all processes, while Q stands for any quorum (a set of process such that $|Q| > |P|/2$).

The algorithm, as the basic one, 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 this algorithm the leader can propose commands, and other replicas, called *followers*, accept the commands. If a follower suspects that the current leader has crashed, it starts the ballot initialization phase to become a new leader. The clients are allowed to send commands to any replica. If a follower receives a command from a client, then it forwards the command to the leader.

The MultiPaxos algorithm pseudocode is presented in Algorithm 2. It works as follows. 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 the PrepareOK, q sends the most recent vote (if any) for each instance indicated by the Prepare (line 21). When p receives PrepareOK messages from a majority of processes (including itself, line 23), then p becomes a leader. It 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. It may happen that p did not issue instance i and j such that $i < j$, and in the PrepareOK messages there was a vote for j, but no vote for i. In such case

Algorithm 2 The simplified pseudocode of JPaxos

```
1:  **Initialization:**
2:      $procId$                                  {a unique, non-zero process identifier}
3:      $prevBal[i] \longleftarrow (0,0)$                                          {for all $i$}
4:      $prevDec[i] \longleftarrow \perp$                                            {for all $i$}
5:      $currBal \longleftarrow (0,0)$
6:      $isLeader \longleftarrow$ false
7:  **procedure** ProposeCommand($i$, $d$) **enabled when** $isLeader$ **and** $prevDec[i] = \perp$
8:      $prevBal[i] \longleftarrow currBal$
9:      $prevDec[i] \longleftarrow d$
10: send($P$, Propose$\langle i, currBal, d\rangle$)
11: **procedure** BecomeLeader()
12:     $isLeader \longleftarrow$ false
13:     $instanceList \longleftarrow \{ i \mid$ no command issued for $i \}$
14:     $currBal \longleftarrow (k, procId)$ **s.t.** $(k, procId) > currBal$                 {$k \in$ Int.}
15:     send($P$, Prepare$\langle currBal, instanceList\rangle$)
16: **upon** Prepare$\langle bal_q, instanceList\rangle$ from $q$ **s.t.** $bal_q \geq currBal$
17:     **if** $q \neq procId$ **then** $isLeader \longleftarrow$ false
18:     $currBal \longleftarrow bal_q$
19:     $prepInst \longleftarrow \emptyset$
20:     **for all** $i \in instanceList$ **s.t.** $prevDec[i] \neq \perp$ **do**
21:         $prepInst \longleftarrow prepInst \cup \{ (i, prevBal[i], prevDec[i]) \}$
22:     send($q$, PrepareOK$\langle currBal, prepInst\rangle$)
23: **upon** PrepareOK$\langle bal_q, prepInst_q\rangle$ from $Q$ **s.t.** $bal_q = currBal$ **and not** $isLeader$
24:     **for all** $(i,\_,\_) \in prepInst_q$ **s.t.** $prepInst_q$ delivered **do**
25:         $prevBal[i] \longleftarrow \max(\{ b_i \mid (i, b_i, \_) \in prepInst_q \})$
26:         $prevDec[i] \longleftarrow d$ **s.t.** $(i, prevBal[i], d) \in prepInst_q$
27:         send($P$, Propose$\langle i, currBal, prevDec[i]\rangle$)
28:     $isLeader \longleftarrow$ true
29: **upon** Propose$\langle i, bal_q, d\rangle$ from $q$ **s.t.** $bal_q = currBal$
30:     $prevBal[i] \longleftarrow bal_q$
31:     $prevDec[i] \longleftarrow d$
32:     send($P$, Accept$\langle i, bal_q\rangle$)
33: **upon** Accept$\langle i, bal_q\rangle$ 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 $i$
```

p must propose a command not only for j, but also for i, as MultiPaxos is supposed to produce a series of consecutively numbered commands, so a gap would prevent command decided in instance j from being executed. While p could propose any value in i, for simplicity it always proposes an empty operation (*no-op*).

To propose a command in instance i, the leader p sends a Propose message to all processes, 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 from a majority (a vote is either Accept or Propose), all for instance i in ballot b (line 33), issues the command c in instance i.

3.2.1 Number of concurrently voted instances

The algorithm allows the leader to propose commands in any instance i, and so to conduct voting in arbitrary number of instances concurrently. In practice, this is usually restricted so that leader can propose only commands in instances that have the number not greater than a tunable *window size* than the instance with least number that is not yet decided. This is akin to the TCP sliding window protocol.

3.2.2 Batching commands

It is often beneficial for performance to vote in one instance for a list of commands instead of a single command. Such optimisation is called *batching*, and a list of commands is called a *batch*. This retains high throughput with small commands [SS12].

3.2.3 The Catch-Up Protocol

In SinglePaxos, when a process p has not yet decided and it starts considering the leader faulty, then p attempts to become a new leader. This might happen either for the old leader was indeed faulty, or because of network failures and asynchrony. Obviously, learning the value is required for termination of the algorithm. In MultiPaxos, 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 could propose itself as a new leader by executing Phase 1. But frequent leader changes are inefficient and thus should be avoided. Therefore, in JPaxos the *catch-up protocol* is implemented, which is used by a stale replica to learn about any missing commands from other replicas by querying the replicas for decisions. Such procedure is commonly used (e.g., [RST11, KA08, RR03]), and an idea of this kind appeared in Section 3.3.2 of [Lam98], but no algorithmic details were given.

The need for snapshots

In MultiPaxos all processes must issue the same unbounded sequence of commands. When failures happen, this raises the following problem. When a process p is separated from others (e.g., due to a network split), then it does not learn the currently decided commands. Once the connectivity is restored, p has to learn the missing decisions. Notice that other processes cannot tell if p crashed or is correct, but, for now, is separated from other processes. So the number of decisions other replicas need to remember grows without a limit. Obviously, it is not viable to store in memory an unbounded number of decisions whenever some process, such as p, is unresponsive. Practical use cases of Paxos, such as SMR, provide a solution to this. SMR replicates a state machine (SM), and instead of executing all the missing commands, the state of the SM on p can be updated to reflect a state after executing the missing commands. For this purpose, in SMR systems such as JPaxos every process periodically creates a *snapshot* of its local state in the main memory. Then, when a process such as p needs to catch up, it receives the snapshot and only those decided commands that are not yet executed in the state represented by the snapshot. By creating snapshots on a regular basis, the number of such commands is bounded. However, creating a snapshot is an additional cost that often noticeably affects performance.

Algorithm 3 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
        $i \leftarrow \max(\{\, i \mid \forall\, j \le i$, some $d_j$ issued in $state \,\})$
      ]
4:  **procedure** updateSnapshot($newSnap$)
5:      $lastSnap \leftarrow newSnap$
6:  **procedure** startCatchUp($highestInst$, $p$)
7:      $missing \leftarrow \{\, i \mid$ not issued($i$) **and** $i \le highestInst \,\}$
8:      $maxInst \leftarrow \max(\{maxInst, highestInst\})$
9:      send($p$, CatchUpQuery$\langle missing \rangle$)
10: **upon** CatchUpQuery$\langle missing_q \rangle$ from $q$
11:     **if** $\exists\, i \in missing_q : i \le lastSnap.i$ **then**
12:         send($q$, CatchUpSnapshot$\langle lastSnap \rangle$)
13:         $missing_q \leftarrow missing_q \setminus \{\, i \mid i \le lastSnap.i \,\}$
14:     $log \leftarrow \emptyset$
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.** $\exists\, i$ : ( $i \le lastSnap_q.i$ **and** not issued($i$) )
19:     restore state from $lastSnap_q.state$
20:     $lastSnap \leftarrow lastSnap_q$
21:     **if** $lastSnap_q.i \ge maxInst$ **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** $\forall i \le maxInst$, issued($i$) **then**
29:         CatchUp completed
30:     **else**
31:         $(\_, p) \leftarrow currBal$                                {$p$ is a leader}
32:         startCatchUp($maxInst$, $p$)
```

The procedure in details

The catch-up procedure starts whenever a process learns that it is missing some commands that other processes already decided. There are a couple of ways a process can learn that: either it gets a message for a higher instance number than expected, or it does not receive for some time any messages in an instance that it did not decide yet (and the leader is not suspected to be faulty). To start the catch-up procedure a process calls startCatchUp(*highestInst*, p), passing as arguments *highestInst* and id of a process p to which the CatchUpQuery request will be sent (line 9). The argument *highestInst* is the latest instance number that the process learned either from the RecoveryAck, Propose, Accept, or Prepare messages, or from the JPaxos failure detector – *highestInst* is piggybacked on the heart-beat messages. (The RecoveryAck message will be explained in Chapter 6.) 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 (non-leader) replica 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, the updateSnapshot procedure 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.

3.3 The SMR framework

The vast majority of papers on State Machine Replication (SMR) using Paxos focus on the Paxos protocol itself and provide vague general description of the SMR part or discuss only selected details of putting together Paxos and the state machine. For the purpose of describing recovery process as a whole, here a full and detailed description of *Paxos-based SMR framework* is presented. The framework reflects JPaxos [JPa22] and consists of two main parts: a *Paxos thread*, which is responsible for agreeing operations in consecutive instances, and a *Replica thread*, which receives new operations from clients, passes the agreed operations for execution by the state machine, returns the results to the clients, and manages snapshots created by the state machine. While the parts are called *threads* for simplicity, the name only signifies logically distinct sets of responsibilities, and does not relate to system threads or number of thereof. For instance, in JPaxos the Replica thread is split among multiple, well-synchronized system threads in order to enable efficiency. Interestingly, in JPaxos the Paxos algorithm with common optimisations is sufficiently performant with one system thread for logic and one system thread receiving and dispatching messages per each link to another replica. Figure 3.1 depicts data of the Paxos and the Replica thread as well as logical steps of ordering and executing a command. The data and the steps are discussed below.

3.3.1 Paxos thread data

In the SMR framework, Paxos stores the following data (explained below): ballot number, proposer state, log and snapshot. *Ballot number* is the largest ballot number ever seen by the process, and it is predominantly used to reject messages from past ballots and telling which process is the leader. *Proposer state* is used iff the process either is, or tries to become the leader, and is used to tell apart those two states. Additionally, when a process is trying to become a new leader, it stores PrepareOK messages that arrived so far. *Log* (called also *the Paxos Log* all over the thesis) is a sequence of entries for consecutive Paxos instances, which is updated by the Paxos thread on a regular basis, where each entry contains the following data:

- a state – tells whether the replica got the Propose and whether the instance is decided,
- the number of the last ballot in which the replica cast a vote,
- the last value (batch of operations) seen by the replica,

- a list of processes L_{Accept} from which an Accept message was delivered, together with the ballot number of the Accept messages (records in L_{Accept} with a lower ballot number are erased).

Snapshot is the latest state of state machine extended with Paxos metadata. Exact snapshot contents is discussed in Section 3.3.4.

3.3.2 Replica thread data

The replica thread interfaces with clients, state machine and the Paxos thread. A design choice in JPaxos is that when a client submits some command to a replica and no response is returned within a timeout, the client submits the same command again to another replica. Moreover, JPaxos guarantees that each command will be executed at most once regardless how many times it was submitted. Therefore, unless the client crashes, a command is executed exactly once, what is typically perceived as a valued trait. To detect that a command has been already submitted or executed, upon client's first request ever a replica assigns the client a globally unique identifier (*client id*), and the client then numbers its operations. Consequently, each replica has to keep track of last executed command number for each client, and does so by maintaining an associative array called *reply map*. The map, for each client id, stores the last executed command number from this client and the state machine response generated at executing the command.

The replica thread is informed by Paxos thread upon each new decision and puts the numbers of decided instances in a *decided instances* set. *Next instance id* is maintained to know which instance must be executed next.

Commands passed to the state machine are also numbered, and so the replica thread stores a *command sequential number* as well. Notice that MultiPaxos requires deciding a no-op in some cases, and a no-op has no command sequence number. Hence, one must never assume that command sequential number are same as instance number. More prominently, when batching is in use, a single instance contains multiple commands.

In JPaxos, the state machine passes a snapshot to the Replica thread alongside the last command number, say l, executed in the state represented by the snapshot. (l can be any number greater than the command number of previous snapshot.) The replica must then tell which instance number corresponds to the command l, and for this purpose the replica thread maintains a list of pairs of instance number and first command number executed in this instance. (It is possible to avoid the list by passing information about instance numbers to the state machine and requiring it to tag the snapshot with the instance number as well.)

3.3.3 Operation of the SMR framework

Here, a brief description of how the framework operates is given, by tracing a command from being sent in a client request to sending a reply back to the client. This is depicted as steps $\boxed{1}$÷$\boxed{11}$ in Figure 3.1.

When a client has never contacted the system before, it has to request an id first. For this, it contacts any replica and is assigned a client id consisting of the replica id and a locally unique number assigned by the replica[1].

[1]In crash-recovery variants of JPaxos the client id is composed of the replica id, the number of times the replica started and a consecutive client number counted from the current replica start.

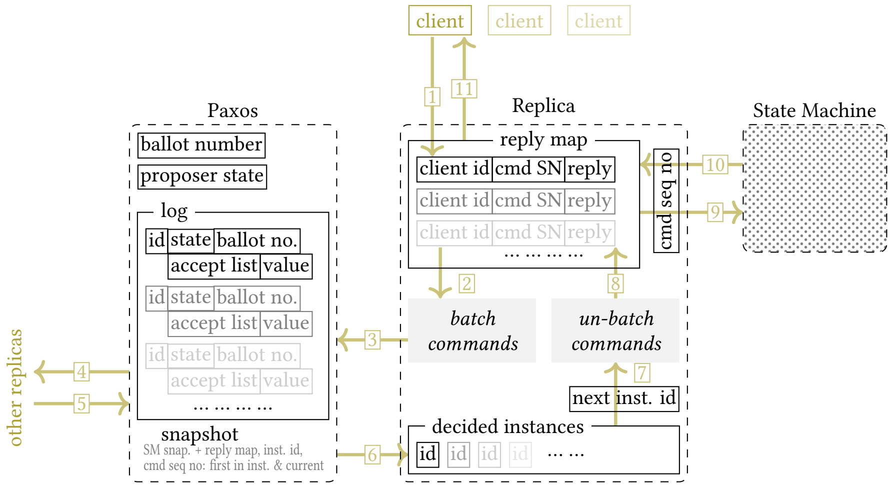


Figure 3.1: Paxos framework: data structures and steps of handling a command

$\boxed{1}$ To execute a command c, the client sends its id, c, and a sequential number (*sn*) in a request to any replica r. The pair (id, sn) is globally unique identifier of the command c. r checks if it has executed c before by consulting the reply map. In case the client sent the request to another replica after a timeout and the system already executed the command, a reply is sent back to client immediately. Otherwise, c is added to a list of commands awaiting for being ordered. $\boxed{2}$ Next, commands are packed in batches. It is good for throughput when network protocol headers and metadata of Paxos messages are negligible compared to the size of the batch. However, one has to be cautious not to collect the commands for a batch for too long in order to keep latency in bounds. In JPaxos a configurable and constant batch size is assumed. If there are no undergoing ballots, a new batch is created as soon as a single command arrives. Else, commands for a batch are collected either until a (configurable) timeout counted from the arrival of the first command in batch expires, or the size of the commands exceeds the batch size. (This is basically reusing the Nagle's algorithm and was positively tested to improve overall throughput.) The timeout for collecting commands in a batch is disrespected whenever certain (tunable) number of instances is already decided but not yet executed by the state machine.

$\boxed{3}$ Once a batch of commands is ready, it is passed to Paxos thread. If the current replica is a follower, it sends the batch to the leader. The leader replica proposes the batches as values in consecutive Paxos instances, respecting a sliding window as described in Section 3.2.1. $\boxed{4}$ $\boxed{5}$ The batches are then agreed upon in the instances.

$\boxed{6}$ When a replica agrees upon a value (a batch of client commands) in an instance i it informs the replica thread, which adds the number i to the list of decided instances. $\boxed{7}$ As soon as the state machine finishes executing commands decided in an instance $i-1$, the replica thread attempts to remove instance i from the list of decided instances, and once it succeeds, $\boxed{8}$ it unpacks the batch of commands for execution. Commands ready to be executed are first checked against the reply map. If the command has already been executed, it is discarded. $\boxed{9}$ Else, it is passed to the state machine, together with command sequential number, for execution. $\boxed{10}$ Once the state machine returns the response, the reply map is accordingly updated and the command sequential number is incremented. $\boxed{11}$ If the client is connected to the replica, then a reply containing the state machine response is sent to the client. So, only one replica, the one that the client is connected to, sends the reply to the client. In case of link or machine failures, the client can connect to any replica to receive the reply (from the reply map).

3.3.4 The contents of a snapshot

A snapshot represents the state of the state machine and the Replica thread data. When the state machine produces a snapshot of its state, it passes the snapshot to the Replica thread alongside with the number l of the last operation whose result is recorded in the state. Then, the Paxos framework extends the snapshot with the following metadata: l, the number i of the instance in which this operation was decided, the number k of the first operation in the batch of operations in that instance, and the reply map. Thus, the system can take snapshots after executing any operation.

Once a snapshot has been created, instances from the log that are lesser than i can be safely erased, as well as any old snapshots. In JPaxos however only instances earlier than the previous snapshot are erased from the log. Replicas typically query each other for the missing state in two cases: either after a message delay/loss, or after a netsplit/crash. In the latter, when a replica is missing lots of data, sending a snapshot is right. But in the former a replica is missing very little information and sending a snapshot is excessive. Therefore, for an efficient catch-up it is worthwhile to keep also a number of recent instances even if a more recent snapshot is available.

When a replica has to recover its state from a snapshot, it replaces the reply map with the one from snapshot, updates next instance number to be executed and next command sequential number accordingly. Then it resumes execution from the instance i by omitting first $l - k + 1$ operations in i, as executed. (It is possible also to omit recording k and rely on the reply map to prevent the first $l - k + 1$ from being executed and thus also assigned a sequence number.)

Chapter 4

State of the art

There have been numerous articles related to Paxos consensus algorithm and its use to build a SMR system. Lamport's original description of Paxos, a consensus algorithm suitable for SMR replication, presented in [Lam89, Lam98] was followed by attempts to explain it more clearly (e.g., [Lam01a], [Lam96], [Lam01b]), or to fill in any missing details and extend the algorithm to provide a better foundation for implementation (e.g., [KA08], [vRA15], [BBH+11]). Also, many optimisations were proposed (e.g., [Lam06], [MJM08], [HMS16]), including a prominent group of ideas that leverage the semantics of SMR operations to boost the performance (e.g., [Lam05], [CSP08], [MPP12, MBP14], [MAK13], [BPvR14, LFE+19], [EBR+20]). In addition, similar protocols have been proposed. Paxos has a close resemblance to Viewstamped Replication [OL88, LC12] that was published a year before Paxos and was developed independently of Paxos. With the motivation of creating an algorithm that produces a result equivalent to (multi-)Paxos, and having as the primary goal understandability, Raft [OO14] has been proposed. [vRSS15] and [WZM+19] formalize how Paxos, Viewstamped Replication, ZAB [HKJR10] and Raft relate.

The rise of demands for automated solutions to cluster management, failover, and sharding finally led to the adoption of consensus (including Paxos) in practical systems, with prominent examples of Chubby [Bur06], ZooKeeper [HKJR10], Spinnaker [RST11] and Spanner [CDE+12]). In [Bur06], the authors describe the design of Google's Chubby, a lock service which uses Paxos. Paxos, as originally stated, is a page of pseudocode, while inside of Chubby the implementation grew up to several thousand lines of C++. Similar code size estimate is mentioned in (weakly documented) use of Paxos in Microsoft's data center management solution [Isa07]. In [CGR07], the authors document the evolution of the Paxos algorithm from theory into practice while developing Chubby. A mechanism of durable logs and snapshots serialized on disk is briefly described. As it can be seen from these papers, making a centralized consensus system production-ready can come at the cost of adding optimizations and recovery mechanisms.

Recovery in Paxos

The paper introducing Paxos [Lam89, Lam98] presents a protocol that already supports crash-recovery: the legislators (processes) write the most important notes in a ledger they carry at all times (stable storage), while other notes are written on a slip of paper (DRAM), and in case they leave the parliamentary chamber (crash), the contents of the ledger allows them to continue work upon returning to the chamber (recovering). The support of recovery by writing the vital data to stable storage is called *FullSS* in this thesis.

De Prisco, Lampson, and Lynch [PLL00] use a timed I/O automaton model to formally analyze Paxos. As in [Lam98], they assume that, whenever required, the state is recorded to stable storage.

Boichat *et al.* [BDFG03] 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 processes belonging to some majority 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". ViewSS and EpochSS take weaker and more practical assumptions about the system, and limit only the number of processes that may crash at the same time.

Kirsch and Amir [KA08] show the Paxos protocol with a simple state recovery method that uses stable storage at key places of the protocol. Apparently, 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.

Rao, Shekita, and Tata [RST11] use Paxos to build Spinnaker – a data store system with support of state recovery based on stable storage. They explain how the system deals with a leader failure as well as how a crashed machine is recovered (and they use the word recovery to describe both of these actions). The recovery of a crashed machine proceeds in two phases: local recovery (from a persistent log) and catch up. If the replica has lost all its data because of a disk failure, then it moves directly to the catch up phase[1], which is similar to the one presented in Section 3.2.3, but the leader is contacted. At the end of the catch up phase, the leader momentarily blocks new writes to ensure that the recovering process has fully caught up the current state. (In the framework presented in the thesis, the busy leader is not disturbed at first place and the system is not blocked during catch up.) 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., [Bur06, CGR07, MPSP10, BBH$^+$11, MPP12, CDE$^+$12]) and popularizing the Paxos algorithm (see e.g., [vRA15, Lam01b]) do not even mention, or only give some vague idea, about support for a crash-recovery model of failures. Moreover, some other authors use the term recovery in a completely different context. E.g., in [Lam06, JRS11], 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, surprisingly little progress was made regarding support for state recovery after crashes. However, some state recovery methods have been developed for protocols similar to Paxos.

Recovery in non-Paxos consensus protocols

Viewstamped Replication (VR) [OL88, LC12] is an efficient state machine replication protocol, similar in operation to Paxos. Oki and Liskov [OL88] describe an efficient state recovery method in VR 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 [LC12] 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. With stable storage, lack of any data in the stable storage naturally indicates the first start ever.

Aguilera, Chen, and Toueg [ACT98] proposed failure detectors aimed for the crash-recovery model, and determined under what conditions stable storage is nec-

[1]Regrettably, the article does not comment on if and how the correctness is ensured in such case.

essary 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, ViewSS and EpochSS circumvent this impossibility result by restricting the number of simultaneous failures.

Ongaro and Ousterhout [OO14] described Raft, a consensus algorithm for managing a replicated log. Raft allows candidates for leader to be elected only if they have the most up-to-date logs. This prevents the need for transferring data from follower to leader upon election. The Raft algorithm uses a catch-up method that is similar to presented in this thesis, 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 [JRS11] described ZAB, an atomic broadcast protocol for primary-backup systems that offers some support for crash recovery. ZAB was developed as a part of Apache ZooKeeper. The authors use the term recovery also for a correct initialization of a new leader once the previous one crashed, and describe it in detail. However, no information is given on how a crashed replica recovers its state (apart from telling that it does).

Michael *et al.* [MPSS17] 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 I am aware of, that provides such an oracle in asynchronous system without the use of stable storage. Similarly to EpochSS and ViewSS, 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.
In [MPSS17], 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, then 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, I 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 EpochSS needs no writes to storage and gives the same guarantees.

Optimization of state recovery

In the State Machine Replication, to recover a replica after a crash, one must both ensure that the process does not violate any guarantees (such as safety), 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 (see Section 3.2.3).

While the size of commands and the process of creating snapshot is application-specific, selecting the recent snapshot and the Paxos log entries and sending them to the recovering replica is the responsibility of the framework. For many workloads the recovery time is dominated by the log and state transfer. Creating periodically a state snapshot impacts the performance of a replica [MDP16], and in some applications the impact can be severe. Some authors, e.g., [BSF$^+$13], [MDP17], proposed ways to enhance efficiency of preparing state snapshots and transferring the log and state. The solutions that they propose can be deployed independently of the recovery algorithms proposed in this thesis and can reduce the time it takes a replica to catch up with other replicas.

Bessani *et al.* [BSF$^+$13] 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, respectively, reduce the impact of creating state snapshots on performance of the system, and enhance state transfer in a model with Byzantine faults. 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 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. FullSS evaluation presented in this thesis shows similar results with sufficiently large batch sizes for 1kB commands (see Section 7.2.4 and 7.3.4). While throughput gains achievable by parallel logging depend on the system workload, the ViewSS and EpochSS algorithms retain system throughput and latency regardless of workload.

Mendizabal, Dotti and Pedone [MDP17] 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, in [MDP17] all vital data are written to stable storage. The novel idea (with respect to recovery) 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. The state snapshot is divided 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.

Reconfiguration and recovery

Reconfiguration [LMZ10, LMZ09, LC12, JLM15, JM14] is a procedure that changes the set of processes P in a Paxos-based SMR system. (In this thesis, P is assumed to be constant.) It can be used to dynamically increase or decrease the number of replicas (which impacts the crash resilience level), but can also be used to replace a crashed replica by a fresh replica. In the latter case, reconfiguration is initiated upon suspecting a crash of a replica, either by some other replica, or by an external component. This method has some drawbacks. First, false suspicions (frequent in unstable periods) can lead to removal of replicas from P that are, in fact, correct. Second, as the system consists of a dynamic set of replicas, 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 [LMZ10, LMZ09]. Recently some efforts were made to support reconfiguration without the need for consensus [JLM15, JM14]. The Replacement algorithm [JLM15] is especially relevant here, as it is dedicated to use reconfiguration for handling replica 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 3.2), and verifying the versions in majority checks. Version Paxos is 3% slower than Paxos (although the authors speculate that this slowdown can be reduced). While the impact of a false suspicion on performance is reduced in [JLM15] compared to the classical reconfiguration, it still requires initializing the state of the replaced replica.

Persistent memory

As soon as Intel Optane DC Persistent Memory Modules were released, a number of papers that assess their performance was written, e.g., [IYZ$^+$19, HT20, PIL$^+$19, MDSG20, GKL20, YKH$^+$20]. There is also a growing number of papers that propose better software support for pmem, for instance by adding awareness of pmem to various programming languages [WZL$^+$18, GVVS20, HS21] or lowering cost of managing persistent memory allocations [CWB$^+$20, BAM19]. However, software support for pmem has been researched long before the first prototype hardware was available. [BBCR22] is a survey of software systems and libraries designed for pmem.

Since the first prototypes of persistent memory products made it to market, one can observe a large number of new, pmem-optimized (non-replicated) key-value stores that leverage pmem's byte addressability and low latency, yielding systems that outperform traditional block-based solutions (see, e.g., [HPM$^+$18], [BHC$^+$13], [CJ15], [DHK$^+$15], [XJXS17]). In [HPM$^+$18], the authors proposed a key-value store that leverages both the byte-addressability of pmem and the lower latency of DRAM, using a technique called cross-referencing logs to keep most pmem updates off the critical path. In [XJXS17], the authors proposed a persistent hash index residing in pmem for fast searching and a B+-Tree index residing in DRAM for fast updating and supporting range scan. On top of the hybrid index, they built a key value store.

Little work has been done so far on the use of persistent memory for consensus protocols. In [DHL$^+$18], the authors propose actually consensus for enhancing pmem: an approach to providing fault-tolerance for storage class memory (SCM; SCM encompasses pmem) by replicating its contents with consensus. They argue that the emerging SCM devices are likely to fail after brief use, and replicating the contents of the memory will mitigate the threat of data loss and thus solve a critical challenge for adopting this new technology. They use a generalisation of ADB protocol [ABD95] which ensures linearizable read-write access to memory and they run it within programmable network switches.

In [RD17], the authors propose a Raft-based replicated log plug-in for key-value data stores. It uses a small amount of pmem instead of normal mass storage devices in the performance-critical code path that appends data to the log. In parallel to the operation of Raft, the data is drained from pmem into a log placed on a standard SSD. This allows for reducing the performance penalty caused by latency of mass storage device. Essentially, they use ordinary Raft, but the most recent stable storage writes are cached in pmem for efficiency.

In [BFAP16], the authors use a catchy title to bring together consensus and non-volatile memory, however they briefly discus selection of papers on distributed consensus, and selection of papers on non-volatile memory, rather than papers that bridge the two matters.

Chapter 5

Persistent Memory

People always want better, faster and cheaper hardware. In reality some trade-offs are unavoidable. In early computers the predominant form of main, random-access memory – the magnetic core memory – was non-volatile thanks to the physical phenomenon it was based on. The core memory was then replaced by static and dynamic RAM that are faster, denser and cheaper, but no longer non-volatile. Thus, for the last 40 years, which is more than the half of Turing-complete hardware history, the standard computer had the main memory that was random-access, fast, directly addressable and volatile, and various data storage devices that are non-volatile but slower, of larger capacity and usually require copying the data to main memory before accessing it. Operating systems and the practice of creating computer programs adopted to the volatile, fast and limited in capacity DRAM and the persistent and slow mass storage. The research attempts to enhance the existing technologies as well as propose new, better ones. Mass storage device throughput and capacity increase and latency drops, but the performance numbers are still far off from what is possible with DRAM. Raising DRAM capacity further currently encounters an obstacle: raising density by scaling down single cells is not possible, and workarounds are being worked on. One of the research goals in memory development is to create memory that would be as good as DRAM, but would not lose its contents on power loss. For years multiple novel technologies were proposed, but until recently either they did not scale in capacity, or the memory deteriorated after too few accesses, or the latency was too high to compete with DRAM.

In 2019 the first product that challenged DRAM made it to the market. Intel and Micron introduced Intel® Optane™ DC Persistent Memory Modules that use a phase-change memory technology called 3D XPoint. These modules are installed in DDR5 slots just like ordinary DRAM modules, but require hardware support from the motherboard and the CPU to function. Alongside with coining the hardware, Intel prepared the software ecosystem for the persistent memory by actively acting towards standardizing the access to pmem, adding support to major operating systems and developing a set of user-level libraries. Since limited performance of storage is an obstacle on a way to efficient SMR that supports recovery, this thesis assesses how pmem can help. In this chapter *persistent memory* (*pmem*) refers to the Intel hardware and its properties. The future of non-volatile main memory technology is yet unclear, but one should hope that in the future the persistent memory will be no worse than it is today, and so that the gains from using pmem will only rise.

5.1 Pmem properties and configuration

The following key characteristic of the persistent memory are discussed: persistency, direct access, byte addressability, latency, throughput and capacity.

5.1.1 Persistency

Pmem can be configured in a variety of ways: some offer persistency, some do not. First, hardware can be configured to put pmem behind DRAM, and use DRAM as a cache of hottest pmem entries. This is called 2-level-memory, and advertises to the operating system a continuous region of volatile memory of the capacity of pmem. This setup is not interesting for this thesis and is intended to seamlessly use pmem to increase main memory capacity with minimal performance loss. The other setup is called 1-level-memory, and exposes the DRAM and pmem as separate regions of memory. The setups are depicted in Figure 5.1.

The next tier of settings is selecting persistency and interleaving. The persistent memory modules (*PMM*) can be either non-interleaved or interleaved. Non-interleaved setting lets the operating system access each PMM on a separate address range, and interleaved setting exposes multiple PMM modules as a continuous address range, directing writes to adjoining addresses to alternating PMMs in a manner similar to striping. Current generations of pmem have almost no benefit of using either mode in terms of latency, and advertise the interleaved mode as allowing for a higher throughput. Pmem physical address space can be divided into *regions*. The regions can have one of two modes: App Direct and Memory Mode. For the former it is possible to select interleaving, the latter is always interleaved. The Memory Mode simulates volatile memory by hardware encrypting all accesses to the pmem with a key that is generated on upon machine start and lost on powering down or restarting the machine. It is intended to extend the capacity of main memory and is of no interest to the matter discussed in this thesis.

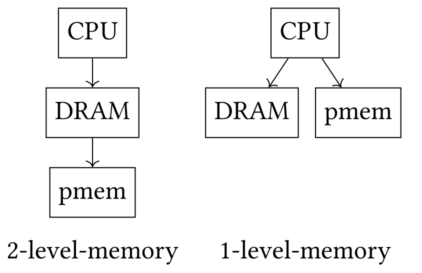


Figure 5.1: 1-LM / 2-LM configurations

Configuring pmem regions in the App Direct mode is the only way to get persistence. Such region is advertised to the operating system as persistent. However, it is crucial to understand when the data written to such addresses becomes persistent. Two *power fail protected domains* have been defined: ADR and eADR ((Enhanced) Asynchronous DRAM Refresh). The Figure 5.2 graphically represents which data survive power loss.

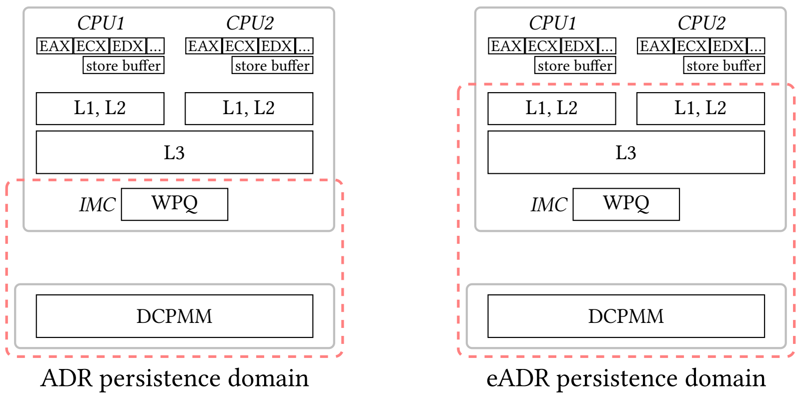


Figure 5.2: Persistence domains

When a store instruction is issued, the data is first written from a register to the store buffer. Then it is automatically written to CPU caches. There is no time limit how long the write will be present only in caches. From the caches a cache line containing the write is then pushed to the write pending queue (WPQ) of the memory controller (which nowadays resides in the CPU package, hence it is called integrated memory controller – IMC). The memory controller sends data via DDR5 bus to PMM (using a DDR-T protocol, which is an extension of standard DDR5 protocol). With

ADR hardware only the data that is in the WPQ or PMM is guaranteed to persist across power failures. This has severe consequences on programming applications that use pmem. The alternative, eADR, guarantees that also the contents of the CPU caches will end up in the PMM upon a power failure, while the writes that have already been issued but are only in the store buffers can be lost. So eADR basically makes the persistency guarantees be the same as cache coherence guarantees. Sadly, as late as July 2021 Intel sales specialist indicated that he is not aware of any platform that supports eADR, so this thesis deliberates on ADR platforms. On ADR platforms the programmer can ensure that a write has been pushed at least to the WPQ by calling a SFENCE instruction followed by writing back or flushing the cache line containing the write. Alongside with pmem support, Intel added a CLWB (cache line write back) instruction for this purpose. (However, while the CPUs of the first generation that supports pmem recognize this instruction, they flush the cache line instead of writing it back and leaving the CPU cache intact. The CLWB instruction in later CPU generations works as expected.)

5.1.2 Byte addressability and direct access

Just like in ordinary DRAM, every single byte can be addressed in pmem. This is different from typical storage devices were blocks of a fixed size are addressed. And, in the same way as in DRAM, an access to a specific byte fetches (transparently to the software) a full cache line containing the byte to CPU caches. The pmem addresses are within normal address space, thus allowing for *direct access* (*DAX*) – ordinary store and load machine instructions can read from and write to pmem. Again, this is the way normal main memory works, and unlike storage devices where, before a data item can be accessed by a machine instruction, first a block of data must be copied from the storage device to the main memory, then the block must be written back to the storage device. Pmem requires no form of paging – to access it from a user-level program it is sufficient that the operating system sets up page tables to translate virtual addresses to physical addresses within the pmem address range.

5.1.3 Latency

Latency of memory accesses is one of the most important parameter that told apart main memory and storage devices. To read cold data, DRAM access time is in order of 100ns, bleeding edge SSDs is in order of 10µs, and an ordinary SSD is in order of 100µs. So, storage devices have latency two orders of magnitude worse than DRAM. Pmem latency is 2÷3 times as high as DRAM. This figure is true both for reads and writes. With good locality or a smart prefetcher this latency difference between pmem and DRAM can be hidden from the end user. However, the latency is still going to be visible upon persisting writes – when one issues a store followed by a store fence and a cache line write back machine instructions, then the CPU stalls until either the write ends up in an empty place in WPQ (what takes about 100ns) or until an earlier write from WPQ is flushed to PMM and the current write in enqueued in WPQ (what takes up to 300ns). The former latency is encountered with infrequent writes, the latter otherwise. Obviously a single SFENCE and one CLWB per each changed cache line can flush multiple writes. The latency numbers presented in this thesis characterize memory on the same NUMA node (memory attached to the same socket as the core). The pmem latency has been evaluated at least in [IYZ+19, HT20, PIL+19, GKL20, YKH+20] and confirmed on hardware used for evaluation.

5.1.4 Bandwidth

Characterising bandwidth (or throughput) of pmem turns out to be quite complex task. The manufacturer's product brief gives six figures characterising the throughput for each model, and even this is not enough to guess the throughput a given workload is going to achieve.

First quirk is that the DDR-T extension of the DDR5 protocol which is used to communicate with PMM sends four cache lines as a single transfer. So, performing small random accesses may be 4 times slower than large random accesses. Thus Intel gives bandwidth for 64B (one cache line) and 256B (four cache lines) random reads/writes. For instance, advertised read bandwidth of 512GB module of Optane PMM 200 series is 1.4 GB/s for the former and 5.3 GB/s for the latter access size. Read and write bandwidth are not symmetric – write bandwidth is substantially lower; for the module mentioned above the write throughputs are advertised respectively as 0.47 GB/s and 1.89 GB/s.

Transferring four cache lines at a time has also another effect: blind writes of a full cache line must be performed as a read-modify-write operation in hardware. So, blind writes smaller than aligned 256B cause a read from PMM. Mixed read/write workload turns out not to be a simple interpolation between read and write throughputs [IYZ+19].

Unsurprisingly, when multiple PMM are configured to work in interleaved mode, the throughput can in favorable circumstances scale linearly with PMM count.

There is also a bitter catch: an upper throughput limit per single physical thread. The advertised throughput, that indeed can be approached in a multithreaded application, is unobtainable by a single threaded application. In applications that need to serialize writes, such as a state machine in SMR replication or even a database [Fed20], the write throughput is limited to about 0.7GB/s. This is less than the typical throughput of PCI-Express storage devices or a 10Gbps network that are ubiquitous in data centers. The write throughput on the nodes used in evaluation (6 interleaved PMM per socket; for hardware details, see Section 7.3.1) is presented in Figure 5.3.

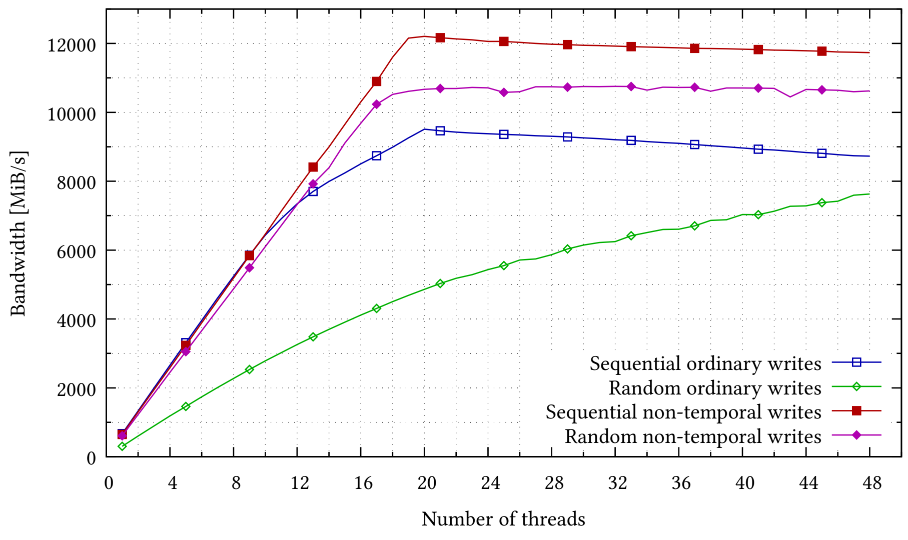


Figure 5.3: Write bandwidth of 256B blocks on 6 interleaved PMMs for 1÷48 threads. (FIO 3.23 with libpmem ioengine.)

5.1.5 Capacity

Currently three capacities of pmem modules are available: 128GB, 256GB and 512GB. This is beyond capacity of a typical DRAM module, and with 6 memory channels per socket one can get up to 3TB of PMEM (officially PMM must be paired with a DRAM module on the same channel). For the matters discussed in this thesis, this means that one does not have to care about pmem memory consumption.

5.2 Programming considerations, libraries, and tools

5.2.1 Managing pmem space by operating system

The Storage Networking Industry Association (SNIA) proposed a standard for implementing pmem support in operating systems so that the OS can manage permissions and allocation sizes and the end user applications can take full advantage of pmem.

Once an application exits, the persistent memory it allocated should not be not freed (like ordinary main memory allocations), but should stays allocated, so that it can be reattached to the the application when it is run again. Also, the user should be able to remove pmem of applications on demand, for instance when he decides never to run the application again. To facilitate this, SNIA proposed to use a file system extended by pmem support. Currently at least the ext4, xfs and NTFS file systems have such support. Then, an application can request persistent memory by allocating a file of specified size, and the operating system checks permissions and quotas by standard means. The user can, provided he has permissions, remove the file to free memory at any time.

The file systems supporting pmem are required to provide the applications with direct access to the underlying memory of a file. When an application calls the standard POSIX `mmap()` library function with right flags, the operating system (OS) is required to set up the virtual memory of the process so that the physical address range of the contents of the file becomes a part of the process virtual memory. Then, every access within the mapped region is translated by memory management unit (MMU) to pmem. Since `mmap()` does not automatically create page table entries (aka MMU mappings) but sets up the OS to do it on demand, most libraries facilitating pmem access bundle together `mmap()` with access to each memory page of the mapped region so that OS creates the page table entries. Once this is done, the OS is no longer anyhow involved in accessing the pmem by the application[1].

The standard routines for accessing files are still required to work, so one can choose between `mmap()` + direct access and standard file API. While standard file API can also benefit from DAX features, it involves the operating system by each access, adding thereby unnecessary overhead.

Another option that was proposed (but seems not to gain much popularity despite being implemented) is to expose pmem as a character device, and `mmap()` the device from the application. This allows only a single application to use the device, and is in this aspect inferior compared to the file system approach, while giving almost no benefit over it.

The Figure 5.4 presents the software architecture of accessing pmem in the way describe above as well as in a variety of ways that are uninteresting for this thesis for they do not provide DAX.

[1]For this reason certain features, such as reflink (COW), are not available in DAX file system.

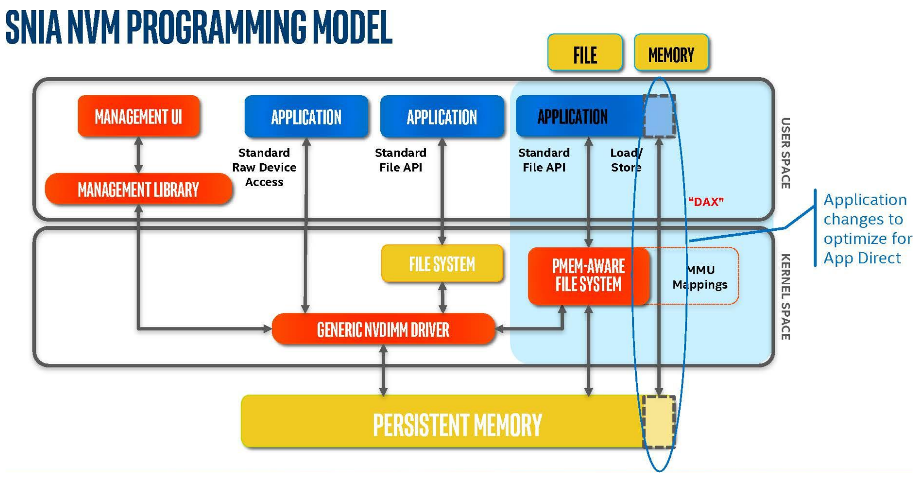


Image source: Intel pmem resources

Figure 5.4: SNIA NVM Programming Model

5.2.2 Inherent challenges of pmem programming

Persistent memory pools

In programming persistent memory it is common to refer to a continuous, logically indivisible piece of the pmem as a memory *pool*. It can refer to a single file, or a number of files that are assembled in virtual memory one after another, or a whole device. For programming it is crucial that the offset between any two data items within the same pool always stays the same.

Address space randomisation

Current operating systems, in order to hinder attacks that require known memory addresses, use position-independent code (PIC) and address space layout randomization (ASLR). The former requires code to run correctly regardless of the address in memory where the code was loaded. The latter is a technique to randomise start addresses of base program, all loaded libraries, heap, stack, etc. so that potential attacker cannot tell at which address a given function or data item is located in the current execution. While the `mmap()` function allows the user to specify a preferred base address of the mapping, it is a hint rather than a strict requirement – one may never assume that e.g., third party libraries used by the same application do not use addresses overlapping with the requested address range. So, one has to assume that at every start of the application the same pmem pool will start at a different virtual address. This makes it impossible to store raw pointers in pmem, both to other objects in pmem as well as to functions. This breaks most object systems that need to store pointers alongside object (e.g., a virtual method table or a pointer to the type description). So far, no programming language allows storing objects with virtual functions in pmem. There are several ways to record a pointer in pmem to other object in pmem. First, if the object is in the same pool, it is sufficient to store the offset to this object from the pointer address. To dereference such pointer one has first to add its value (the offset) to its address. A downside of such pointer is that it cannot be copied or moved within memory without recalculating its value. Another common way of pointing to objects in pmem is to add some form of identifier to the pools and record the identifier and offset from the pool start as a pointer to an object. This has higher cost of dereferencing, for one has to look up the base address of the pool by the identifier and add to it the offset. If a single pool is used, one can store the offset only, however such solution scales badly. Obviously, pointers in DRAM to pmem can be raw pointers, for their lifetime does not extend beyond one execution.

Root objects

Some objects need to be located on known offsets from the beginning of pmem allocated to the application, so that the newly started application knows where they are. Such objects are called *root objects*. If the application dynamically creates new objects, they need to be reachable from these root objects so that after restarting the application the dynamically created object can be found.

Memory allocation within pmem and memory leaks

Since creating or changing the size of persistent memory pool has a high overhead, the applications typically request a large space and then use a userspace memory allocator. Apart from what normal memory allocators need to do, those that can be used for pmem have to satisfy one crucial property: pointer to the allocated memory must never be lost, regardless of power failures that can cause some writes to pmem to be lost – the writes that are not yet written back from CPU caches. For instance, when the programmer wants to add a node to a linked queue, the pointer of the tail has to be changed from null to the newly allocated memory, and the memory allocator must consider the memory allocated. One may never let the allocator consider the block allocated while not persisting the pointer – this would lead to memory leak. In pmem, an undetected memory leak stays there even after restarting the application, so it has much more devastating effect than ordinary memory leaks. Obviously, also it is not acceptable to see after a power failure and application restart a non-null pointer while the memory allocator considers the memory free.

Two approaches have been proposed: the more popular is to record all metadata of the memory allocation operation in a journal, then mark the recorded data as valid, then execute the allocation, and finally erase the journal entry. Then, at restart the application has to check the journal for entries that are valid and re-execute them. The other approach is to allocate memory, persist as little of the the allocator state as possible, and hand over the allocated pointer to the programmer, allowing the crash to interrupt before the pointer is persisted anywhere in pmem. Then, at start of the application one has to scan recursively all objects and rebuild the list of free and allocated memory. This has much lower overhead of each allocation, but needs a way to scan for possible pointers and takes time at application start [CWB+20].

Consistency of data after power failure

Currently, on a power failure contents of all registers and the stack[2] is lost. The application at restart has no way to tell when the crash happened, and it could happen in the middle of logically indivisible computation. For instance, in the classical bank transfer example the money could have been subtracted from one account, but not yet added to the other. Moreover, pmem persistency guarantees on ADR hardware

[2]Stack contains e.g., return addresses that are no longer valid on restart.

say only that the data which was written back from the CPU caches to the pmem is there after the power failure. The user can explicitly write back the data, but the hardware can also write back the data at any point of time – for instance when a cache line needs to be evicted to make space for new lines. Software has no control of this – it is the hardware job to (pre)fetch needed lines and flush those considered as no longer necessary. So, in the classical bank transfer example when the money was subtracted from one account and then added to the other, and if those two accounts do not share a cache line, it is possible that after a power failure only the second account holds the new value, for the CPU flushed line containing it, but did not make it to flush the first.

So, the software has to deal with two intersecting problems: the power failure may leave the pmem inconsistent due to interrupting code in arbitrary place, and the power failure may leave the pmem inconsistent because only some of the updates have been flushed from CPU caches down to the PMMs. A generic way of dealing with such problems is to use transactions – that is, to write a log of operations, then persist the log, and when the application state is logically consistent again, commit the transaction by first marking in the log that the transaction commits, and then committing it. Then, upon restarting the application after a failure, one has to check if any transaction was in progress, and depending on the transaction state either roll it back, or commit it again. While transactions of this kind may seem a clean, elegant and mature solutions, they have inherent performance overhead and the programmer must explicitly begin and commit the transaction as well as annotate all writes that need to be logged by the transaction.

5.2.3 Tools and libraries

Here, a brief list of pmem-dedicated tools is given. This list is not complete and does not contain general-purpose tools that can also help in utilising pmem, but it outlines what sort of tools are currently available.

Administration The Optane PMMs can be configured using Intel's `ipmctl` tool to create pmem regions and do basic health checks. The regions can be further configured using vendor-agnostic `ndctl` tool, which also can display some information about PMMs.

Programming Currently predominant set of libraries that simplify pmem programming is the Persistent Memory Development Kit (PMDK), an open-source framework maintained mainly by Intel that encompasses a broad range of pmem use cases. It facilitates using pmem both as volatile and non-volatile memory, or accessing pmem via RDMA. PMDK provides abstractions on various levels: from low level operations such as `pmem_memcpy` to combine `memcpy` and persisting the data in an efficient way, up to high level components such as a ready to use key-value datastore. Most libraries included in PMDK build upon the low-level *libpmem* library that simplifies pmem memory mapping as well as moving around memory and persisting writes. The largest, and used in the systems evaluated in this thesis, library of PMDK is *libpmemobj* that provides convenient management of memory pools, a memory allocator, routines for creating atomic (but not isolated) transactions, and a couple of gadgets. It also offers a user space tool `pmempool` for inspecting the memory pool, including among others allocator metadata. Both libpmem and libpmemobj

are in C. For C++ the PMDK has *libpmemobj-cpp* that contains convenient bindings to libpmemobj and a growing number of data containers (lists, vectors, trees).
Other libraries exist, but offer no functionality beyond of what PMDK offers. For instance, the most mature pmem library for Java (that in fact builds upon PMDK) – Low-Level Persistence Library (LLPL) – only recently added support for a container as complex as a linked list, and POD data structures need to be laid out by hand by writing primitives on manually calculated offsets from the beginning of a structure.

Testing `pmemcheck` and `pmreorder` attempt to, respectively, learn pmem access pattern of an application (by instrumenting the application), and check if some order of these accesses does not violate user-provided invariants.

Performance monitoring A tool `pcm-memory` from the Intel's Processor Counter Monitor (PCM) toolset can be used to get information about currently used bandwidth of each PMM, including the transfers issued by e.g., hardware prefetcher that are otherwise invisible for the software.

Chapter 6

Recovery in Paxos

To discuss recovery, a good starting point is to consider the case when all replicas crash and the system needs to recover. This situation can tell apart data that must be recovered from the data that can be lost with no effect on correctness. The former data must be persisted on stable storage. Actually, in Paxos a crash of all replicas is no different with respect to recovery than a crash of any quorum. A circumstance when at least a quorum is down at the same time is called a *catastrophic failure*.

The data which must survive crashes must be persisted in the stable storage before a network message depending on the data is sent. Else the correctness could be violated if a crash occurred after sending the message, but before persisting the data. However, this means that the writes must be performed in *synchronous* manner, that is a write must be followed by an operation that flushes the CPU caches, memory pages, etc., down to the stable storage medium (an example of such operation is the `fsync()` POSIX function). For conventional storage media, such as solid-state drive (SSD), this requires asking the operating system to synchronize the data, and takes a long time compared to the duration of other local actions which Paxos needs to do. In some Paxos workloads the latency of synchronous writes is the factor limiting performance even with the currently top performant SSDs. While the data is needed only on recovery, it must be written during normal failure-free operation. So, if one requires the ability to recover crashed replicas, one shall expect lower performance.

It is well studied which data must survive crashes in Paxos (e.g., in [KA08]). Essentially, a replica must remember the most recent leader it accepted and the most recent vote it cast (in every instance). Correctness of Paxos directly depends on these data. Moreover, a practical Paxos framework requires storing persistently the snapshot (the same snapshot that is used to limit the size of the log; see Section 3.2.3).

While ability to recover is often considered a must in practical systems, the lower performance is unwelcome. Therefore, the system designers often seek for assumptions that allow the system to retain top performance while still offer recovery. The most common assumption of this kind is that catastrophic failures never happen. This is in line with aim of highly available systems – when the system has to be always available, one must ensure that at any time a majority of replicas is ready to answer client requests. Much is being done to be as close to this goal as possible: replicas are run in different locations (georeplication) for independence in hardware and network failures, and updating and maintaining software is done nonsimultaneously to prevent software faults from taking down multiple nodes at the same time.

When catastrophic failures are assumed never to occur, recovery algorithms that use quorum-bases system principles to replace stable storage can be designed. One of the goals of this thesis is to show, discuss and evaluate such algorithms. Another goal of this thesis is to assess how persistent memory can be leveraged to raise throughput of the Paxos framework that supports recovery when catastrophic failures must be taken under account. The main performance focus of this thesis is the throughput. Wherever a trade-off between latency and throughput was encountered, it was resolved in favour of the latter. Since the discussed algorithms and implementations used for evaluation were designed to optimise throughput, it would be unfair to comment and compare latency.

An approach orthogonal to recovery that can be used to deal with crash failures is to change the set of replicas in runtime. In Paxos, this is referred to as reconfiguration. While it does not provide recovery per se, it can be used to remove a replica that is suspected to be down as well as add a new replica so that the size of the group remains unchanged. Reconfiguration does not support catastrophic crashes. Pros and cons of using reconfiguration to deal with crashes are briefly discussed in Chapter 4.

6.1 SinglePaxos versus MultiPaxos framework recovery

As described in Sections 3.1 and 3.2 the term Paxos, while usually names the algorithm that agree upon consecutive values, may refer as well to a single-decision consensus algorithm. Recovery differs in SinglePaxos and MultiPaxos, especially when the latter is part of a larger framework. Upon recovery of a replica r in SMR framework usually the instances that were voted for when r crashed are, at the start of recovery, already decided, executed, included in a snapshot and erased from log of other replicas. In general, unless a majority of replicas crash or an extended period of network asynchrony occurs, it is sane to expect that at recovery all instances that the recovering replica has seen before crash are already decided. Therefore, some recovery procedures for Paxos framework focus on efficient catch-up, obviously guaranteeing that correctness is intact in the rare case when some instance that was being voted before crash is still being voted upon recovery. Recovery in SinglePaxos on the other hand usually focuses on resuming the voting (in the only instance). The difference is especially visible in the recovery algorithms that assume no catastrophic crashes – in MultiPaxos it is then acceptable to wait until some instance gets decided before recovery can progress. This thesis, as its title says, deals with recovery algorithms for a complete Paxos-based SMR framework.

6.2 Recovery algorithms supporting catastrophic failures

In this section first the well-known algorithm that writes the data required to recover to stable storage is presented. Then, approaches tailored explicitly for the persistent memory are proposed.

6.2.1 Full Stable Storage

In Lamport's Paxos [Lam98], to support recovery, a replica writes a part of the Paxos algorithm's data to the stable storage: it records at least every ballot number sent in LastVote message and every voted value and ballot number sent in a Voted message (see Algorithm 1; in JPaxos, the equivalent messages are PrepareOK, Propose and Accept). Notice that the data stored persistently in the stable storage, for fast access, is still written to main memory, so the data is effectively duplicated in non-volatile and volatile memory. When a replica crashes and subsequently recovers, it reads from the stable storage the data and reconstructs on its basis the state of Paxos. The replica must not answer any messages before it reads and applies all data from the stable storage (but the replica can still update its state on basis of incoming messages). After recovery, the state of the replica guarantees that both consistency and progress guarantees hold, however the replica may know less than it did at crash – for instance, it no longer knows the decisions, for it does not record anything upon Success messages. The aforementioned data is the minimal amount of information that must be written synchronously in order to provide correctness. Lamport proposes several ways of enhancing recovery. First, Success messages can be written asynchronously to the stable storage. These asynchronous writes add no latency and ideally need to store only the instance number and hence do not affect performance, and in case of recovery the state is restored more accurately. [Lam98] also proposes to copy the state from a peer upon recovery in case the system advanced a lot since the crash of the replica (so basically to perform what is called here the catch-up). In this thesis the above recovery algorithm is called *Full Stable Storage* (FullSS in short).

In SMR replication, when snapshots are used, the replica should write snapshots to the stable storage, and once a snapshot is written, it can erase data of the instances that are fully executed in the snapshot. At recovery, after reading a snapshot, framework-specific information from the snapshot is used to coordinate the state of the framework with the State Machine. In particular, the command numbers allow to recognize the moment when the snapshot was created.

A system that uses FullSS to build a Paxos-based SMR framework with recovery support is called in this thesis JPaxos+SS. As one should expect from the name, it is implemented in JPaxos, and works as follows. A leader writes synchronously to stable storage the new ballot number before sending Prepare (in line 15 of Algorithm 2, see Section 3.2) and writes synchronously to stable storage the command it proposes and the current ballot number before sending Propose (line 10). Followers write synchronously to stable storage the new ballot number before sending PrepareOK (in line 22), and write synchronously to stable storage the command they vote 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. The process also writes asynchronously every decision, by writing its instance number only. If the process did not send an Propose or Accept for the decided value before (because, for instance, it learned the decision from catch-up), then it writes first the data it usually writes upon sending an Accept. 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. After recovery, to catch up with other replicas, the recovered replica launches the catch-up protocol. As in case of non-crashed, lagging replicas, this occurs when the replica realizes that it lags behind, by comparing its last instance number (the recovered replica read it from its stable storage) with the instance number i, carried by either the Propose, Accept or the heartbeat message of the failure detector that it first receives. If i is higher than expected, then the replica starts the catch-up procedure.

6.2.2 Persistent-memory aware algorithms

The FullSS recovery algorithm has one major drawback: it negatively affects system performance due to limited performance of stable storage media. This is visible in most workloads [KA08, RST11], and while there are optimisations that reduce the overhead [BSF+13], the performance impact of disk writes cannot be completely mitigated. FullSS works by explicitly copying certain data items from the main memory (which is volatile) to the stable storage. Persistent Memory (pmem), that only recently became available as a retail product, makes a fundamental change in computer hardware: the main memory is no longer deemed to be volatile. Recommended hardware configurations have both DRAM and pmem modules, hence a part of the main memory is still volatile, but each computer program can request a non-volatile main memory area. Having such possibility, there is no need of copying data from main memory to stable storage anymore, for the main memory (or part of it) is the stable storage. Therefore, the proposed pmem-aware systems use a different approach to take the best of pmem. The data items are partitioned into a group that has to survive crashes and is thus stored in the persistent main memory, and into a group that can be lost upon a crash and is stored in volatile main memory. Such division is on one hand motivated performance-wise (pmem is still not as good as DRAM), on the other it is a recommended programming pattern [Sca20].

Pmem-based algorithms presented in this thesis have one more design goal: they eliminate whatsoever recovery of the Paxos protocol – the part of main memory that is stored in pmem suffices to resume execution immediately. This has a positive impact on SMR behavior. While in FullSS-based systems after crash and restart a replica must first recover the state from stable storage before it can reply to Paxos messages, in the pmem-aware systems shown in this thesis a replica can immediately participate in the Paxos protocol upon start. It is worth noting that such design choice is expected to have little impact on performance.

Two Paxos-based SMR systems that take advantage of pmem are presented in this thesis: mPaxos and mPaxosSM. They differ in the extent of pmem use: the former uses pmem only to persist Paxos protocol data (including the last state machine snapshot), while the latter requires the state machine itself to keep its own critical state in persistent memory. Consequently, upon restart mPaxos has to recreate the state of the state machine (which uses only DRAM) from the snapshot stored in pmem, while mPaxosSM does not do it, as the state machine itself uses persistent memory. More importantly, because the state machine in mPaxosSM must be ready to resume operation at restart with no loss of progress basing solely on data stored in pmem, this very property allows to treat the data in pmem as an always fresh snapshot, and so in mPaxosSM the state machine no longer has to create a snapshot periodically.

mPaxos

The design of mPaxos follows closely Paxos-based State Machine Replication framework as described in Section 3.3. mPaxos supports failure recovery, but unlike JPaxos+SS, no data is copied to the stable storage – instead, part of the main memory is persistent. The following vital data are maintained up to date in persistent memory by the Paxos thread:

- the current ballot number, and the state of a proposer (inactive, executing Phase 1, or elected as a new leader).
- the log, with the content as described in Section 3.3.1 and summarized below,
- snapshots of the state machine's state, extended with data that the Replica thread maintains (explained below),
- a bounded list of log entries that predate the last snapshot (to optimize the catch up protocol),

In mPaxos prototype implementation, some auxiliary data are maintained as well:

- the number of the lowest and highest instance in the log,
- the lowest undecided instance number in the log,
- an epoch number, incremented on each replica startup and used solely to assign client IDs[1].

A Paxos log entry for each instance contains: the number of the last ballot in which a replica sent an Accept or a Propose, the number of the last ballot in which a replica got an Accept (if an Accept arrives before a Propose and the replica voted in an earlier ballot, then both numbers are needed), the current state (whether a fresh Propose was received, whether the instance is decided), a list of processes from which a Propose or an Accept was delivered, and the last command seen by the replica.

The Paxos (producer) and Replica (consumer) threads share in pmem a producer-consumer queue (list) of instance numbers which are decided, but not yet executed.

[1]In mPaxos implementation the client ID consists of replica ID, epoch number and a consecutive number, obtained using fetch-and-add on a volatile counter. Many other solutions are possible.

Decided instances are added to this queue on a regular basis (irrespective of snapshots' creation). When a new snapshot is created, all instances having all commands executed before the snapshot was created are removed from the queue. On every snapshot creation, the Replica thread updates in pmem the following variables that are stored both in DRAM and pmem:

- the number of the next instance,
- the next command to be executed by the state machine,
- the reply map[2].

This way, after crash, the Replica thread has in pmem the values of these variables in a state that is consistent with the state machine snapshot.

Persistent memory speeds up starting the system after crash. There is no need to read a snapshot and logs from stables storage, and there is no need to recreate the state of Paxos, as this state is already in the memory ready for use. But pmem is slower than DRAM, so the choice which data to store in pmem was made considering also high system performance during fault-free runs. While Paxos immediately returns to its state from the moment of crash, the state machine is restored from a snapshot that usually reflects an earlier state. Therefore in mPaxos the state machine has to execute (on the restored state) the commands that were executed after the snapshot but before the crash.

mPaxosSM

The alternative system that uses pmem is mPaxosSM. Comparing to mPaxos, it takes a different approach with respect to the state machine. Since pmem has matured into a fairly complete programming model, and suitable libraries to provide state machine developers with the benefits of persistent memory are available, so one can expect the developers to be able to design the state machine in such way that they maintain the state necessary for crash recovery in pmem. For instance, in a key-value map (that is used in the evaluation in Section 7.3), this state is just the content of the map.

A pmem-aware state machine is required to resume operation and guarantee the properties it offers after being restarted, regardless of the moment of the crash and the abruptly disrupted writes to the persistent memory. The SMR framework can take advantage of this requirement, and knowing that the state machine can be restarted from contents of the pmem at any point of time, use the persistent memory of the state machine as an ever fresh snapshot of its state. Thus, in mPaxosSM, a state machine does not create periodically snapshots. The thread executing the Paxos protocol stores the same data in pmem as it was in mPaxos, except that there are no snapshots. The log is truncated at will, when the Paxos framework decides to do it.

However, a replica is still obliged to create a snapshot on demand, when it is asked for it by a lagging replica that is executing the catch-up protocol. For this, the Paxos framework simply requests the state machine to return the name of the file storing critical data in pmem[3] and to lock computation, so that it can read data that is not being modified concurrently. Once the snapshot is created, the computation is released. The lagging replica executes the reversed procedure to install the catch-up snapshot back to pmem. To support the mechanism SMR's API was amended.

[2]A reply (the state machine response associated with the client id and request sn) is first written to a map in volatile memory, and upon snapshot is moved to a map in the persistent memory. A lookup in the reply map first attempts to retrieve the value from volatile memory then persistent memory.

[3]To use pmem, a program must `mmap()` a file created in a pmem-aware file system (see Section 5.2.1).

The Replica thread modifies on a regular basis in pmem:

- the number of the next instance to be executed by the state machine,
- the number of the next command to be executed by the state machine,
- a queue with the numbers of instances decided, but not yet executed,
- the reply map,
- a list of files created from the snapshot received during catch-up, and
- a flag signaling if the catch-up snapshot is being installed.

The two last items on the list play a role when the catch-up procedure was interrupted by a power failure while a snapshot was being installed – in such case the replica must continue installing it upon starting anew.

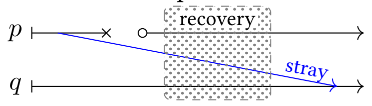
mPaxosSM affects the design of state machines – the programmers do not need to create snapshots. Instead, they need to use pmem. The effort to write code that efficiently creates a snapshot is comparable to the effort needed to persist the state of a typical state machine in pmem. In return, mPaxosSM offers the instant time of system recovery after crash.

There is one pitfall regarding recovery. If a replica crashed while executing some command, mPaxosSM does not know after recovery if the last command was executed completely or not. Therefore it has to pass the command again for execution to the state machine with the associated number. The state machine knows the numbers of the executed commands, so it can recognize that the command is repeated and has to decide whether to execute it or not. Note that the state machine programmer is responsible to get it right, as the decision depends on the semantics of commands. For example, in case of the KV-Map, reexecuting the same write (or the same read) right after the first execution is not harmful and the programmer may not to care about this issue at all, but a command "increment a counter" could not be repeated.

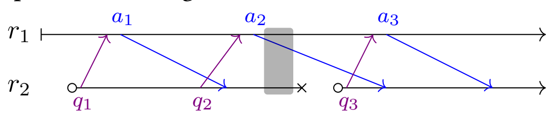
6.3 Recovery algorithms not supporting catastrophic failures

This Section presents two recovery algorithms for Paxos (and Paxos-based SMR), called ViewSS and EpochSS, that assume no catastrophic crashes. That is, at any point of time some quorum must be up. If this ever fails, the presented algorithms stop – no process can recovery anymore. Correctness holds regardless whether the assumption holds (the system is never going to decide inconsistently), however some Paxos decisions may be lost forever. With no catastrophic failures assumption, a process r can recover by querying peer replicas for the processing state and informing them about the recovery. In such case, the processes do not need to write to stable storage every value they voted for. This allows the system to support recovery and to retain the performance of a system that does not support recovery. Compared to systems that support catastrophic failures the I/O latency bottleneck is eliminated.

Designing such algorithms has some pitfalls. The major problem to deal with are stray messages. A *stray message* is a message sent by a process p before it crashed and delivered to the destination after the process recovered:



The process p cannot learn that it sent the stray message by asking other replicas, for the message is still in transit at recovery. So the recovery algorithms must ensure that correctness is retained when the other replicas receive a stray message. The recovering process itself can get a stray message that is an answer to a query for the state of other processes. Consider the following case where r_2 starts with a recovery, crashes and attempts to recover again:



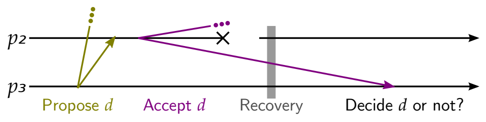
The replica r_2 (before crash) sent query q_2 (same as q_1) after suspecting that q_1 or its answer a_1 is lost. Once it got a_1 (same as a_2) the system made some progress (depicted as a gray box). Then, after crash and recovery, r_2 has to recognize that a_2 is not an answer to its query q_3, but an answer to a query sent before crash. Failing to do so may lead to accepting outdated information – a_2 does not include progress that was made after r_2 got a_1 and before r_2 crashed.

Upon startup a process must check if it has never run before or it has been restarted after a crash. For this purpose, JPaxos+SS simply checks if the stable storage is nonempty, and mPaxos and mPaxosSM attempt to reattach the pool of persistent memory created on first run. The algorithms that recover its state from peers also need a way to tell if they should commence recovery. Obtaining this knowledge from other nodes in the system is not possible, for the assumptions regarding network allow stray messages as well as message duplication, and so the information that this is a fresh start could arrive multiple times. Algorithms presented in this thesis use the stable storage to store a single number, and the presence of this data tells a process that it has to recover.

By referring to MultiPaxos algorithm (Section 3.2, Algorithm 2), the processes make two promises that must be kept in order to ensure safety of Paxos:

1. when a process p sends[4] PrepareOK in ballot b (line 22), then p promises not to reply to messages in ballots older than b,
2. when a process p sends Accept for command d in instance i and ballot b (line 32), it promises to reply to any forthcoming Prepare message for instance i with command d and ballot number b.

The role of the recovery algorithm is to restore the state of the replica so that those promises hold. A process can learn almost all of the promises it made by querying other replicas. However, some promises made before a crash are still confined in stray messages. Consider a potential safety violation caused by a stray message:



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 recovery algorithms include a mechanism to either discard all stray messages (ViewSS), or wait until the system progresses sufficiently so that the message is harmless (EpochSS).

Notice that until the promises are learned (and so recovery finishes), the recovering process p cannot actively take part in Paxos, i.e., it is neither allowed to vote (reply to Propose) nor to take part in the leader election (reply to Prepare). However, p can deliver and process messages, for instance it can, after receiving in an instance i a Propose and Accept messages from a quorum, decide a command in instance i.

[4]In the algorithm pseudocode processes send messages (such as Prepare) also to self and answer themselves. In implementation the promises are also made by code replacing the response to self.

Algorithm 4 View-based recovery

Notation: Line `XX` replaces line `XX` in Algorithm 2.
Lines `XX'`, `XX''`, ... are inserted after line `XX` in Algorithm 2.

```
      ...
14' : write $currBal$ to stable storage
      ...
18  : if $bal_q \neq currBal$ then
18' :   $currBal \longleftarrow bal_q$
18'':   write $currBal$ to stable storage
      ...
35  : upon start
36  :   read $currBal$ from stable storage
37  :   if $currBal \neq (0,0)$ then
38  :     send($P \setminus procId$, Recovery$\langle currBal \rangle$)
39  :     wait for RecoveryAck$\langle bal_q, bal_p, highestInst \rangle$ from $Q$ s.t.
                          $bal_p = currBal$ and $(i, q_l) = bal_q$ delivered from $q_l$
                            where $(i, q_l) = \max(\{bal_q \mid bal_q$ delivered$\})$
40  :     $currBal \longleftarrow \max(\{bal_q \mid bal_q$ delivered$\})$
41  :     catch up to $\max(\{highestInst \mid highestInst$ 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  :   $highestInst \longleftarrow \max(\{ i \mid prevDec[i] \neq \perp \})$
47  :   send($q$, RecoveryAck$\langle currBal, bal_q, highestInst \rangle$)
```

6.3.1 The ViewSS algorithm

In Paxos, a ballot number changes when a new leader is elected. The *View Stable Storage (ViewSS)* algorithm, presented in Algorithm 4, enforces incrementing a ballot number also during a recovery phase, which will allow processes to discard any stray messages, as they 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 each new ballot number to stable storage, so that after the process crashed and has been 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 the new ballot number to be 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 performance is the same 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 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 the crash occurred (line 36). Then, the recovering process sends the Recovery message to all processes and waits for the RecoveryAck messages from a 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 a 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 instance that has the highest number recorded in RecoveryAcks (line 41). Once the state is updated, the process finishes recovering and joins the Paxos protocol. Notice that while ViewSS fetches decided instances (by means of catch-up) and so it has to wait until the instances are decided, then for instances not yet decided it would instead suffice for correctness to learn the value voted for in the new ballot. This approach is not used as typically MultiPaxos is robust enough to decide the instances even before the catch-up query is delivered.

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 if and only if 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.

ViewSS involves absolutely no changes in the voting phase of Paxos, so the performance in failure-free periods shall be identical to performance of crash-stop Paxos. Upon changing the leader a synchronous write is required, so the leader election takes longer. Unlike all other algorithms presented in this thesis, in some cases ViewSS requires a leader election when a replica recovers, so a new leader is elected even though the current one is fine. It is possible that the same replica will stay the leader, but it still has to conduct a leader election and increase the ballot number.

6.3.2 The EpochSS algorithm

The *Epoch Stable Storage (EpochSS)* algorithm requires that every process stores in stable storage an *epoch number* (initially set to 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 ballot number 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 stops progressing.

Algorithm 5 Epoch-based recovery

```
      ...
 6':  $epoch \leftarrow [0, 0, \ldots, 0]$                                        {of size $|P|$}
      ...
22 :  send($q$, PrepareOK⟨$currBal$, $epoch$, $prepInst$⟩)
23 :  upon PrepareOK⟨$bal_q$, $epoch_q$, $prepInst_q$⟩ from $Q$ s.t.
                                              $bal_q = currBal$ and not $isLeader$
                      and $epoch_q[q] = \max(\{epoch[q]\} \cup epoch_Q[q])$ where
                        $epoch_Q[p] = \{ epoch_q[p] \mid epoch_q$ delivered from $q \in Q \}$
23':    for all $p \in P$ do $epoch[p] \leftarrow \max(\{epoch[p]\} \cup epoch_Q[p])$
      ...
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($P \setminus procId$, Recovery⟨$epoch[procId]$⟩)
41 :      wait for RecoveryAck⟨$bal_q$, $epoch_q$, $highestInst$⟩ from $Q$ s.t.
                                                $epoch_q[procId] = epoch[procId]$
                                                and $epoch_q[q] = \max(epoch_Q[q])$
                                                and $(i, q_l) = bal_q$ delivered from $q_l$
                                   where $(i, q_l) = \max(\{bal_q \mid bal_q$ delivered$\})$ and
                        $epoch_Q[p] = \{ epoch_q[p] \mid epoch_q$ delivered from $q \in Q \}$
42 :      $currBal \leftarrow \max(\{bal_q \mid bal_q$ delivered$\})$
43 :      for all $p \in P$ do $epoch[p] \leftarrow \max(epoch_Q[p])$
44 :      catch up to $\max(\{highestInst \mid highestInst$ delivered$\})$
45 :  upon Recovery⟨$epoch_q$⟩ 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⟨$currBal$, $epoch$, $highestInst$⟩)
```

The pseudocode of EpochSS is presented in Algorithm 5, 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. First, 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.

6.4 Comparing the recovery algorithms

To compare the recovery process of SMR systems that use the methods for supporting recovery presented in Section 6.2 and 6.3, the flow of the recovery process is presented here, divided into logical steps. Each step is described and a couple of optimizations are discussed. Finally, the impact of supporting recovery on the overall system performance is explained.

In the analysis of recovery, the state update must also be included. While in FullSS, mPaxos and mPaxosSM 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 – only after catching up with the rest of the system the replica can execute new client requests. 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.

6.4.1 Steps of the recovery process

The recovery process can be split into the following steps which are common for all recovery methods. They simplify the analysis and comparison of the methods. 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 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 the received data.

Unless a replica is run for the first time, at start 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. Pmem-based recovery algorithms reattach their persistent memory (i.e., `mmap()` a file from a pmem-aware filesystem and perform an access to each memory page to create page table mappings between virtual and physical addresses). Once this is done, mPaxos and mPaxosSM can answer Paxos messages. In Step 2, FullSS updates the replica on basis of the read data. For FullSS, this step completes the recovery phase, and the replica can answer Paxos messages. In the same step, mPaxos updates the state of the state machine by passing it the snapshot, a replica using EpochSS just synchronously writes a new epoch number to a disk, while mPaxosSM and ViewSS perform no actions. While a replica using FullSS, mPaxos or mPaxosSM recovered, its state reflects the state it had at the moment of a crash, so unless other replicas made no progress from the crash, the replica still needs to update itself before it can deliver upcoming commands. In Step 3 (executed only by ViewSS and EpochSS), 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 Algorithm 4, line 41, and Algorithm 5, 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 systems using ViewSS and EpochSS learn what state p is missing in Step 3, other systems 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 the replica 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. Notice that in mPaxosSM the snapshot also may not include all decided instances despite being the snapshot of current state of q – this occurs when q has already decided but not yet executed some instances. As soon as a process receives a snapshot, it can use the snapshot 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.

6.4.2 Performance issues of replica recovery

In Sections 6.2 and 6.3, the overhead that each recovery method induces on every process during a normal (fault-free) period of system execution was discussed. To summarize: FullSS brings a major performance penalty in the voting phase, due to synchronous writes to stable storage; mPaxos and mPaxosSM use for some data items pmem that is slightly slower than DRAM; ViewSS slows down each ballot change by demanding a single synchronous write (of the new ballot number); EpochSS enlarges the ballot change messages with a vector of epoch numbers (one per each replica). In this section, performance of the systems during a recovery phase is analyzed.

Despite many similarities, each algorithm has its own characteristics. The main differences are as follows: FullSS requires reading the state from stable storage, restoring it and updating the state machine. mPaxos must only update the state machine, and mPaxosSM does not need to do a thing. ViewSS must broadcast a message and gather responses, possibly causing a ballot change thereby. EpochSS has the same complexity as ViewSS, but it never forces a redundant ballot change. Regardless of the recovery method, the recovering replica likely needs to catch up. FullSS, mPaxos and mPaxosSM have to fetch from peers only the decisions that were taken during downtime, and EpochSS and ViewSS must fetch all from peers. To expose the differences, the border cases in terms of the recovery cost are presented below.

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

When no decisions were taken during downtime, mPaxosSM is up and up to date immediately. All other algorithms must at least update the state machine and execute all instances not included in the snapshot. Data required to this end is already present in memory upon restart in mPaxos. FullSS, ViewSS and EpochSS must first recover the data structures (the log and a recent snapshot). FullSS reads it from the stable storage, while ViewSS and EpochSS fetch the data from a peer via the network. Fetching the data via network takes more time than reading it from storage, assuming that the stable storage is faster than the network.

Also, if few decisions were taken since the crash occurred, replicas are likely to still have them in their logs. Therefore, FullSS, mPaxos and mPaxosSM have to fetch from peers substantially less data than other algorithms, while the amount of local work required to bring the state back is roughly the same.

When multiple decisions were taken since the crash occurred, the state restored by systems tolerating catastrophic failures is out of date. This is meaningless for performance of recovering and updating mPaxosSM and has little impact on mPaxos as well, for catch-up can start immediately in these systems. But in a FullSS-based system catch-up can start only after restoring the state from the stable storage. Thus, state update in FullSS will be more resource consuming than in other systems.

The catch-up procedure is identical for all systems, but its cost may not be equal, as they use different storage media for the data. ViewSS and EpochSS store data only in DRAM, so they will always perform fastest. mPaxos and mPaxosSM store some data structures in pmem, that is slightly more expensive than DRAM. With sufficiently fast network the throughput of pmem can limit the catch-up speed. The same applies to FullSS that has to write some of the data it fetches via catch-up to stable storage (and wait for the data to be written).

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.

6.4.3 Recovery bottlenecks

The recovery algorithms use three resources: stable storage, processor, and the network, each of which can limit the system performance. Below possible bottlenecks of the recovery process are characterized, and an explanation is provided under what circumstances a particular resource can noticeably limit the system performance. Note however that state machines 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 the state machine.

Stable storage In modern computers 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 bandwidth 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. However, the replica still needs to parse the data on the stable storage to learn what data needs to be fetched.

During catch-up the stable storage (or pmem) may also affect the robustness of recovery – the newly learned decisions have to be persisted. Again, if the write throughput or latency of the storage medium limit the performance more than the network, FullSS, mPaxos and mPaxosSM performance can be limited by storage.

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 state machine 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. FullSS and mPaxos may need to restore the state 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.

6.4.4 The impact of recovery on other replicas

Recovery of a crashed replica imposes an overhead on other (non-faulty) replicas. The impact of this overhead on the overall system performance is discussed below.

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 the Recovery message), 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 a time interval at least as long as that required to execute 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 than recovery, hence there is 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, say r, performs a state transfer to the recovering replica. Selecting the most recent snapshot and subsequent requests is not resource demanding in all algorithms with an exception of mPaxosSM. mPaxosSM uses the pmem area of the state machine as its snapshot, but for the data to be consistent the state machine must make no changes to the data while it is being read by the framework. Hence, r must stop the state machine for the time when the snapshot is created. Regardless of the recovery method, transferring the snapshot and decisions 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 7.2.6).

A recovery process of one replica is independent from any recovery processes of other 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 is expected to be equal to the sum of impacts of single state updates.

Chapter 7

Evaluation

To characterise the performance of systems described in Chapter 6 and tell which method of supporting recovery is best fitted for a given workload all the systems were evaluated in several scenarios. First, the throughput when no failures occur was assessed. Then, the robustness of recovery and catch-up procedure was evaluated. Additionally, the impact of crash on system throughput was recorded. Due to timeline of research progress and hardware availability, the evaluation is split into two waves: at first FullSS, ViewSS and EpochSS were tested on older hardware, then mPaxos and mPaxosSM (alongside FullSS and EpochSS as reference) were tested on a more recent hardware.

7.1 Evaluated systems and benchmarks

7.1.1 JPaxos, mPaxos and mPaxosSM

FullSS, ViewSS, EpochSS and mPaxos systems are implemented in JPaxos [JPa22] – a Paxos-based State Machine Replication middleware coauthored by me. JPaxos is equipped with optimizations such as the batching and pipelining for a higher performance. A variant of JPaxos with no recovery support was evaluated in [SS12].

mPaxosSM is implemented as a fork of JPaxos, since mPaxosSM changes the state machine API. The state machine no longer has to do snapshots periodically, but instead it must, on demand, pause processing and provide name of the files that represent persistent memory of the state machine.

JPaxos is implemented in Java, but Java has insufficient support for pmem yet. Thus, all routines accessing pmem are written in C++ and *Java Native Interface (JNI)* is used to call the routines from Java. The C++ code uses libpmemobj-cpp, which is one of the libraries included in Persistent Memory Development Kit (PMDK; see Section 5.2.3). PMDK transactions are extensively used to ensure that the data in pmem can at any point of time be brought to a consistent state (which is done by the PMDK library upon restarting the application after a failure), e.g., data received in a message are written to pmem as a transaction. Obviously, also the standard read-write locks are used to protect objects and prevent the changes made by one thread from becoming visible to other threads until the changes are complete – the PMDK transactions aim only for atomicity, and are indifferent to isolation.

Using JNI has some drawbacks: it prevents Java's just-in-time compiler (JIT compiler) from certain optimisations (for instance inlining), moreover to pass data such as strings and byte arrays one has to copy them from C++ unmanaged memory to Java garbage collected memory. For instance, if one wants to read from memory mapped pmem in C++ and needs to pass the data to Java, one has to allocate new Java memory buffer (or array) and copy the data to the buffer. To reduce the number of JNI calls in mPaxos and mPaxosSM, the following data are cached: ballot number, the lowest undecided instance number, the proposer state, the number of the highest instance in the log, and the numbers of the next instance and command to be executed by the state machine, and the epoch number (that is used solely to assign client ids in mPaxos and mPaxosSM).

Obviously, the mPaxos and mPaxosSM systems use Direct Access (DAX) mechanism to access persistent memory, so they can access and flush data to pmem without the need to involve the operating system, as the memory is accessed with no caches in DRAM (unlike traditional block devices). So, a program can execute a CPU store instruction followed by the `SFENCE` and `CLWB` instructions to persist data, rather than executing a `write` syscall followed by a `fsync` syscall.

7.1.2 System tuning

Performance of a system depends on numerous factors. Some of them are generally workload or hardware related, but other factors are parameters that decide how well the hardware is used and how good the system is suited for a particular workload. Such parameters are called tunables and should be adjusted so that in a specific workload on a specific hardware the system yields best possible performance. The results of evaluation are meaningful only if the evaluated system is fully tuned.

The following tunable parameters were selected for each hardware configuration, system, request size and scenario:

- the number of clients (also referred to as the client count),
- MultiPaxos window size (number of concurrently voted Paxos instances),
- preferred size of a batch of commands,
- timeout of waiting for the commands to form a batch,
- number of threads communicating with clients,
- limit on number of instances that are already decided but are not yet executed,
- frequency of snapshot creation,
- failure detector parameters.

The parameters were adjusted so that increasing (or decreasing) them no longer increased throughput of the system. While mathematical models of selecting right values for the tunable parameters exist (e.g., [SS12]), the settings recommended by these models did not yield the optimal performance for some parameters, hence the values were validated and adjusted experimentally. The Figure 7.1 offers a glimpse on selecting the right values – microbenchmarks with a wide range of tunable values were run and optimal settings were identified.

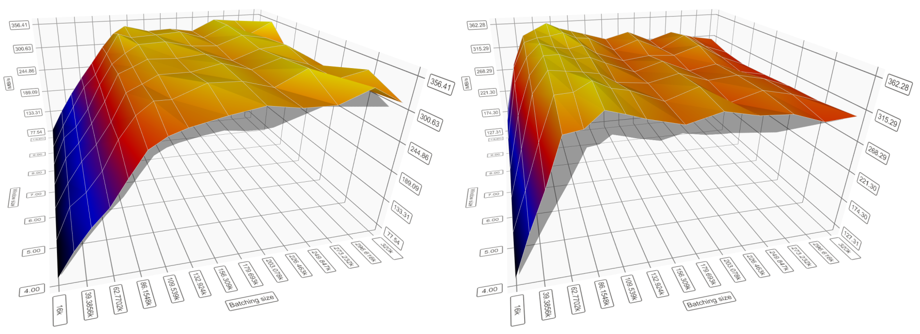


Figure 7.1: Selecting preferred size of a batch of commands and MultiPaxos window size for FullSS (on the left) and mPaxos (on the right) for 2.5kB requests.

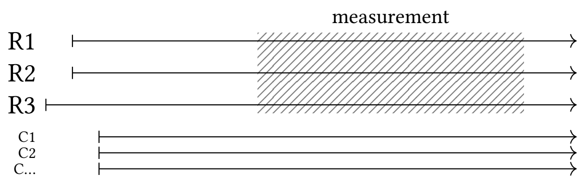
7.1.3 Benchmark scenarios

The evaluation was conducted by repeating a couple of scenarios with different client request sizes and for each of the evaluated systems.

Usually each scenario was repeated multiple times to validate repeatability of the results and average away minor performance swings. The number of runs that were needed to get a clean result depended on what was measured, and ranged from three (to get an average throughput in failure-free case) to about two hundred[1] (to plot throughput in time at crash and recovery).

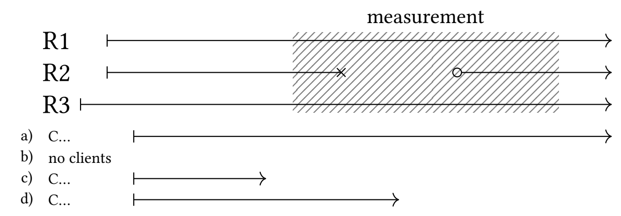
[1]In a few of the 200 runs some other replica than the one intended won the leader election, and to get meaningful plots one has obviously to average over identical runs, hence these few runs were subtracted.

Stable period A scenario also referred to as failure-free is the basic scenario for testing the throughput of a system. It is very simple: three or five replicas are run at the beginning of the test. Then clients start and each client commences sending an endless stream of requests. In the period starting from, for instance, 15th second to 45th second, the throughput is measured:



The time diagram above pictures the scenario. One of the replicas is intentionally started earlier. This is a repeating pattern in all scenarios, for in JPaxos in such case this replica is predominantly selected as the leader, and other scenarios that include failures intentionally crash either the leader or a follower. The throughput is measured after a warm-up for several reasons: in Java it takes a while to optimize the code in runtime, JPaxos dynamically adjusts some parameters (e.g., timeouts, snapshot size estimate), and the state machine may need some time to reach stable performance (e.g., when a KV-Map is used, it starts from empty map and clients need to fill it with data before stable performance is reached).

Crash and recovery The other scenario involves crash and subsequent recovery of a replica:



Replicas are started as usual, and R3, or R5 if the system consists of 5 replicas, is assumed to win the leader election (if another replica becomes the leader, then the run is discarded). Then, either R2 or the leader is crashed by killing the process. The former is called *follower crash*, while the latter is called *leader crash*. The clients are started right after the replicas and send requests either uninterrupted, or only before crash, or before crash and before recovery, or even not at all (see the a)-d) cases in the diagram). During the run each replica records every 100 ms its current throughput as well as the time of recovery and catch-up related events.

7.2 ViewSS and EpochSS

7.2.1 Hardware used for evaluation

The tests were run 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 stable storage was simulated using a `tmpfs` RAM disk, unless stated otherwise; then ST3250620NS HDD was used (SATA II, 7200rpm, 16MB cache, avg seek/write/latency 8.5/10/4.16 ms).

7.2.2 State machine used for evaluation

For the evaluation an *EchoService* state machine was used. It operates trivially: for every command a it responds with a. Moreover, randomly, but on average once per ten thousand of requests, a snapshot of a minimal size (one byte) is created 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.

Out of scope of this thesis are the tests that were run for assessing correctness of the implementation of the systems – these used the EchoService that additionally recorded a SHA-512 hash of a combined with the previous hash, and produced a snapshot that was the previous hash.

7.2.3 Client request and response sizes

In a Paxos-based SMR system that offers no recovery support, depending on the workload and available resources, either network or CPU is limiting the system performance. Typically, in every implementation huge requests saturate the available bandwidth, while numerous tiny requests use up all available processor time long before any other resource runs out. Thus, the tests were carried out 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 (the CPU is *saturated* or is a *bottleneck*).

In the evaluation environment the borderline between the request size saturating the network and CPU lies at around 320B, as depicted in Figure 7.2. Thus 1024B was chosen as saturating the network and 128B as saturating the CPU.

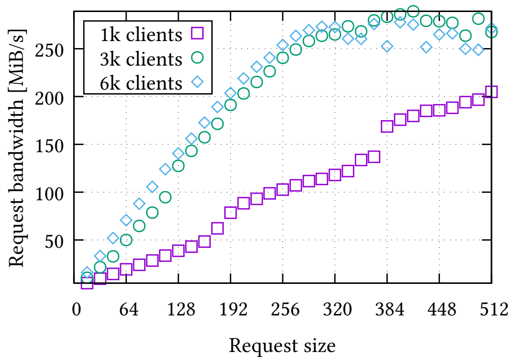


Figure 7.2: Throughput microbenchmark

For setups 1 and 2, a proper number of concurrent requests (that is the same as the number of clients) was selected (respectively 2.1k and 6k), so that increasing the number does not increase the system throughput, and the latency is kept low enough.

The recovery algorithms were evaluated using scenarios described in Section 7.1.3, 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, two cases were examined: 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, two cases are considered when evaluating ViewSS: 1) *leader crash*, and 2) *follower crash*, indicating what was the role 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 the crash, so the Recovery messages induce a ballot change during recovery. EpochSS has no special cases to consider per evaluation scenario.

7.2.4 Evaluation of no-crash operation

Table 7.1a shows the system throughput as 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.

a)

Request size	crash stop	FullSS	ViewSS	EpochSS
1024B (saturated net.)	38 087	1 871	38 200	38 209
128B (saturated CPU)	158 288	11 164	157 317	159 192

b)

Request size	crash stop	FullSS	ViewSS	EpochSS
1024B (saturated net.)	38 130	36 768	38 015	38 329
128B (saturated CPU)	156 787	103 323	155 685	157 228

Table 7.1: The number of requests per second with: a) HDD and b) RAM disk.

While the HDD was able to perform only at most 30 `fsync()` operations per second, SSDs have much shorter access time, and cutting edge SDDs can break the boundary of one million IOPS. So, in other experiments a RAM disk was used to simulate an ideal bus-speed stable storage device. A RAM disk significantly improves the throughput of EchoService with FullSS (see Table 7.1b).

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, in some workloads the performance loss from supporting failure of a majority of replicas might be acceptable. However, when the CPU is a bottleneck, FullSS is 34% slower than competitors. 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”.

7.2.5 Evaluation of Recovery Operation

This experiment presents how much time a replica needs to recover. The sooner the crashed replica recovers its state, the sooner the distributed system will have the level of fault-tolerance and availability that it had before the crash occurred. To evaluate the recovery process, the time of the following events was measured:

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 RecoveryAck messages from a quorum,
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.

Figure 7.3a-c presents the evaluation results in three cases: 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. The following symbols are used: *Fo* and *Fu* is FullSS with stable storage contents, respectively, out of date and up to date, *Vf* and *Vl* is ViewSS, respectively, with crash of a follower and a leader, and *E* is EpochSS.

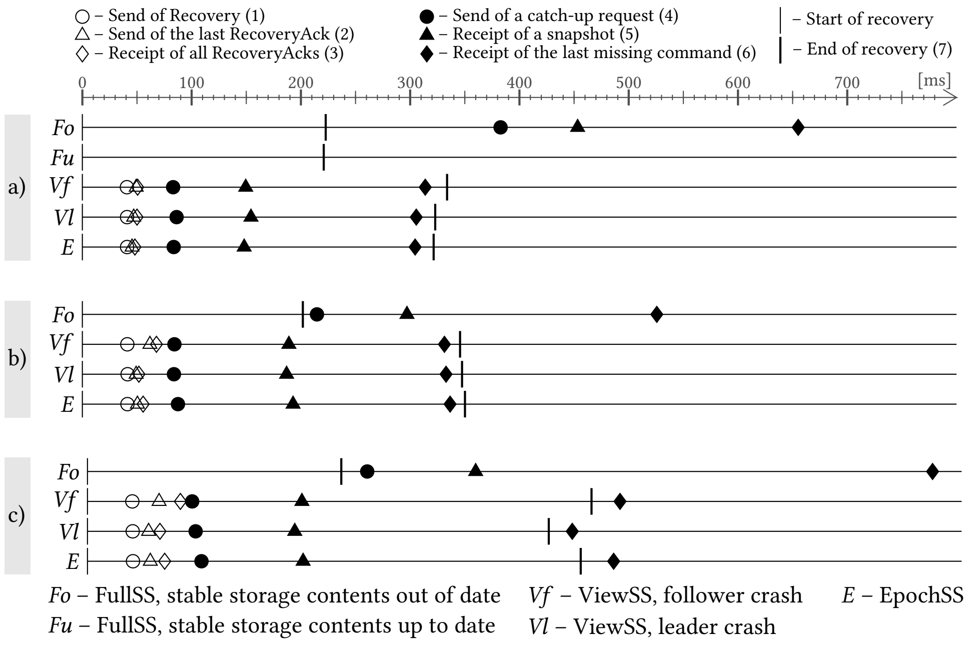


Figure 7.3: The time needed to recover a crashed replica in case of: a) the idle system, b) CPU saturation, and c) network saturation.

Little difference can be seen between the idle system and the system with saturated CPU (see Figure 7.3a-b). (However, keep in mind that in this experiment the state machine used almost no CPU to process requests). The system with the saturated network was recovering longer than the other systems (see Figure 7.3c). 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 CPU usage by the recovering replica depends mainly on the replicated service.

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 7.3, where numbers 1÷7 are used to identify given actions). 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, the catch-up is triggered either by the first Paxos message, or by a heartbeat message of the failure detector. In the idle system, two scenarios were tested: 1) stable storage was up-to-date, and 2) some requests were processed when the recovering replica was down. In Figure 7.3a, case Fo, catch-up is triggered, and it occurs later than in any other test in Figure 7.3 (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 this experiment, 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 the crash occurred. 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).

ViewSS and EpochSS

A replica that uses ViewSS and EpochSS must contact other replicas on recovery. The time instant (1) when it initializes itself and sends a Recovery message is the same for all 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 the 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 the tests, the elapsed time between time instants (1) and (3) is just a fraction of time which is needed to recover a process (2% to 6% of the total recovery time, except for a few cases discussed below).

ViewSS enforces a ballot change upon recovery of a crashed follower (unless the ballot already changed since the crash), which impacts results in Figures 7.3b and 7.3c. It takes more time (compared to other results in Figure 7.3) before replicas can process the Recovery messages (2) and when RecoveryAcks are delivered to the recovering process (3). However, the overall system performance does not seem to be affected, contrary to the expectations.

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 the catch-up 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 this occurs 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 has finished. This may occur if some commands were decided between sending RecoveryAck and sending a snapshot.

Hard disk drive

Figures 7.3a-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. The tests were also repeated with an HDD. FullSS, on one hand, reads a large amount of data from disk, on the other, must persist all data learned by the means of catch-up. So, not surprisingly, the time required to complete the catch-up with an HDD is twice longer when the system is idle and over four times longer when the CPU or the 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 has 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 the 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.

Snapshot and log

The EchoService benchmark was configured in such a way that the time of recreating the state from a snapshot and the time of executing (on average) 5000 requests from a log were negligible. This is because the main goal of this experiment 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 produces much larger snapshots, which means that also the time required to transfer a snapshot between replicas is longer than in the 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 was measured in the system using ViewSS and EpochSS. (In Section 7.3.5 the tests are repeated for a state machine that produces 200MB snapshot.)

7.2.6 Impact on 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. So, during the tests every 100 ms the total number of processed requests was recorded. With this data one can show 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 the lightweight EchoService, the system performance comprises the performance of Paxos and the catch-up protocol. The 100 ms sampling rate has negligible impact on the rest of the system and was intentionally not in sync with crash or recovery time point, and averaging over multiple runs allows for sufficiently stable results.

In the experiments, the system consisted of 3 replicas and a number of clients. At 3sec from the start of the benchmark, the clients begin sending requests. At 12.5sec (after the system stabilizes and some requests have been executed) one replica crashes. At 18sec the replica is started again (and starts recovery). Figures 7.4, 7.5, and 7.8 depict the throughput from 10sec to 22sec of the benchmark run.

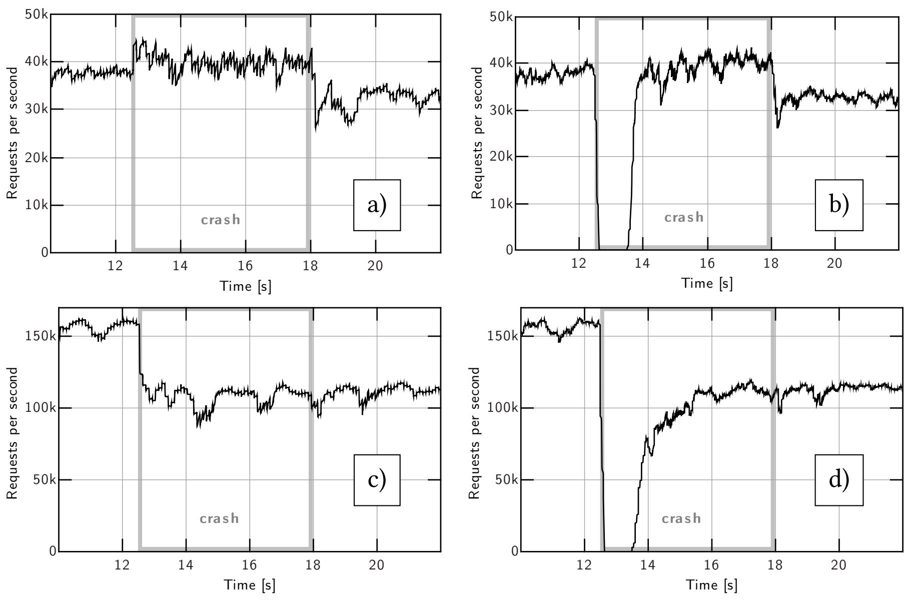


Figure 7.4: Throughput of a non-faulty replica in EpochSS: a,c) the crash of a follower, b,d) the crash of a leader, a,b) saturated network, and c,d) saturated CPU.

The impact of a crash on system performance

In Figures 7.4a-d present the throughput of the EpochSS-based system from the client perspective, by showing throughput of a non-faulty replica. It covers four cases: either the leader or a follower was crashed and the throughput was limited either by the network (1kB request size) or by the CPU (128B request size).

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 repli-

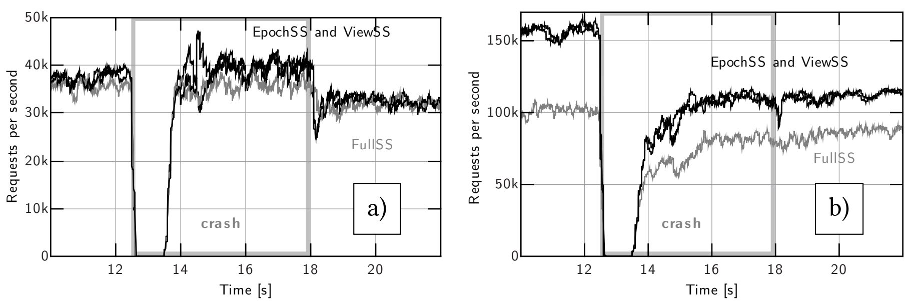


Figure 7.5: The crash of a leader: a) saturated network, b) saturated CPU.

cas 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. 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. These results are identical regardless of the method used to support recovery, what can be told by looking at Figure 7.5.

In Figure 7.6, an interesting asymmetry is shown 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. Now, 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 7.6b). 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), and the executions in which q first missed commands, so the throughput of q was zero, and later issued a large batch of commands, so the peak in performance.

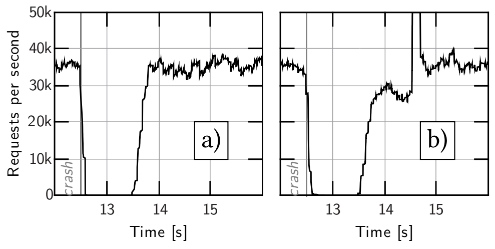


Figure 7.6: Asymmetry of throughput: a) a new leader p, b) a follower q.

Throughput during stable period

To tell if the performance of a system after a crash and recovery is restored, the system throughput in the following *stable periods* was analyzed (a stable period is understood as a time period when neither crash nor recovery takes place and the system performance is at a stable level):

A – just before crash (10÷12s),

B – just before recovery begins (16÷18s),

C – after the recovery, when the system is stable again (20÷22s).

First, the performance of a non-faulty replica (a leader or a follower) in periods A and B is compared (see Figure 7.4). This comparison shows how the system performance changes when one replica crashes and stays down. Notice that in this period the remaining replicas have to send answers to clients that were previously handled by the crashed replica. If the network is the bottleneck (7.4a-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 (7.4c-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 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) 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 7.4a-b). Whenever the CPU is saturated, the system performance in the periods *B* and *C* remains unchanged, as no additional processing is required.

While Figure 7.4 presents results for EpochSS, the results for other systems show the same trends, what is pictured in Figure 7.5.

Although JPaxos currently does not support load balancing, throughput after recovery is slowly restored as long as new clients join the system or old clients reconnect. This behavior can be observed in Figure 7.7 that shows the performance of a ViewSS-based system using a RAM disk and a HDD. When ViewSS was tested with a 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.

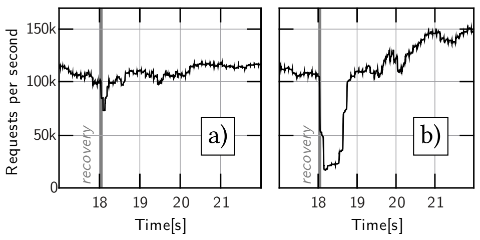


Figure 7.7: Throughput in ViewSS-based system with: a) RAM disk, b) HDD.

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 Figure 7.3b-c for the offset and duration of catch-up). It slightly differs for different algorithms (see Figure 7.5). 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 an 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 7.8). In ViewSS if the leader crashed, then the recovery does not affect the system throughput, but in case the follower crashed the impact of the view change that is enforced on the recovery is clearly visible. As seen in Figure 7.7b, 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.

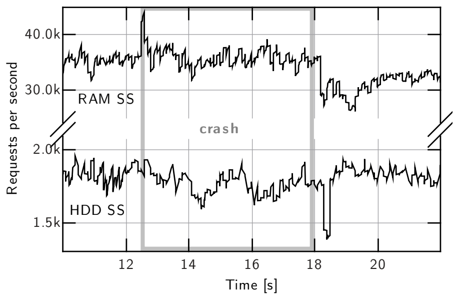


Figure 7.8: Throughput in FullSS-based system: RAM disk vs. HDD.

7.2.7 State machine impact on performance

The EchoService state machine used in this part of evaluation had little impact on the performance of the system. Obviously, if the state machine uses CPU or disk bandwidth, then it will impact the overall throughput of the system. This raises the question if by supporting recovery this impact will raise or stay unchanged.

In Section 7.2.4, the performance overhead of supporting the system recovery during crash-free periods was examined. In case of ViewSS and EpochSS, the results are independent of the system workload. In case of FullSS, the performance penalty may raise when the processing power or storage bandwidth is the factor limiting performance.

In Section 7.2.5, the duration of the recovery process was examined. The time it takes to recover the system highly depends on the workload. In FullSS, the system recovery time depends solely on characteristics of the application, 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 the experimental evaluation the system workload with negligible times of command execution and state restoration was selected. Also, the size of data transferred during state update was relatively small, and, on average, equal to five thousands requests.

Section 7.2.6 examined how crash and recovery affect the system throughput. These results are independent of the system workload, with one exception. 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.

7.3 mPaxos and mPaxosSM

Having obtained pmem-equipped hardware, the evaluation of the Paxos-based SMR systems supporting recovery was possible. To this end, mPaxos and mPaxosSM were tested and compared against FullSS and EpochSS. Additionally several changes in the way evaluation was conducted were introduced: the state machine has been changed to one simulating real work and clients were forced to reconnect to other replica every 1000 requests.

7.3.1 Hardware used for evaluation

All replicas in the evaluation were run on identical machines. While each machine had two identical NUMA nodes, the software was restricted to use only one. A single NUMA node was equipped with Intel® Xeon® Gold 6252N (24×2.3GHz, 35.75MB cache) and six interleaved Intel® Optane™ DCPMM modules, 128GB each. The machines were equipped with 1TB Intel® DC P4510 NVMe SSDs, and were connected to a 10Gbps Ethernet switch (running at wire speed). The machines were running OpenSUSE Tumbleweed with Linux 5.8, libmemobj-cpp 1.10, and openjdk 14.

Each replica was run on a separate machine. In tests with 3 replicas in the system, the clients sending requests were run on two machines with same configuration as the replica machines. In tests with 5 replicas in the system, the clients sending requests were run on one machines with same configuration as the replica machines and two other machines (that together provided similar performance as one replica machine).

7.3.2 State machine used for evaluation

For this second part of the evaluation all tests use a KV-Map state machine. The KV-Map is a key–value store supporting PUT and GET commands. The GET command takes a key (an arbitrary byte array) as a parameter and returns the last value put into the map or a `null` value if no value has been associated with the key. The PUT command takes a key and a value (also an arbitrary byte array) and returns the previous value that was put into the map under the key. The snapshot of the KV-Map contains the most recent value for each key, so its size is the sum of sizes of all keys and values. The size of a snapshot was selected in each test by choosing the range of the keys used by clients. For simplicity, the keys were binary-encoded 4-byte integers.

7.3.3 Memory backends used in evaluation

To tell how mPaxos and mPaxosSM compares against FullSS and EpochSS it suffices to compare the performance figures. But to attribute the differences in performance to a specific reason one has to test the possible causes of speedup or slowdown one after another. For this purpose tests for mPaxos and mPaxosSM systems were repeated with several memory backends. A *memory backend* (or a *backend*, in short), is understood in this thesis as a set of routines that access data in main memory. These routines store the data either in DRAM (*@RAM*), or in pmem emulated in DRAM (*@emulp*), or in pmem (*@emulp*). To emulate pmem by DRAM one can administratively, at booting OS, mark a range of physical DRAM addresses as being persistent memory. Thus, from boot the operating system treats an address range within DRAM just as if it were an address range within a persistent memory module. Such memory uses pmem software routines and has DRAM performance (and, obviously, is volatile).

Table 7.2: Systems and memory backends

System@backend	Snapshots	Paxos	State machine	Crash-Recovery
JPaxos+SS	periodic	Java, data *flushed* to stable storage (e.g. disks)		catastrophic failures supported
JPaxos+epochs	periodic	Java	Java	catastrophic failures not supported
mPaxos@RAM	periodic	JNI / STL	Java	crash-stop, no recovery
mPaxos@pmem	periodic	JNI / PMDK	Java	catastrophic failures supported
mPaxos@emulp	same as mPaxos@pmem, but pmem is emulated in DRAM			
mPaxosSM@RAM	on demand	JNI / STL	JNI / STL	crash-stop, no recovery
mPaxosSM@pmem	on demand	JNI / PMDK	JNI / PMDK	catastrophic failures supported
mPaxosSM@emulp	same as mPaxosSM@pmem, but pmem is emulated in DRAM			

Table 7.2 presents all evaluated systems (with all backends). JPaxos+SS and JPaxos+epochs are the original JPaxos implementations with FullSS and EpochSS recovery algorithms, respectively. JPaxos+epochs stores all data in DRAM using Java Collections, while JPaxos+SS additionally writes synchronously to files. For a fair comparison with other systems, in the experiments JPaxos+SS writes to files stored in pmem, although results for SSDs are also presented where appropriate.

Due to insufficient support for pmem in Java (see Section 5.2.3) mPaxos and mPaxosSM implement pmem access in C++ and bind Java and C++ using JNI (see Section 7.1.1). To estimate the overhead of JNI, JPaxos+epochs that uses standard Java Collections is compared with mPaxos@RAM that (like mPaxosSM@RAM) uses JNI and C++ standard data structures (aka STL). To tell if using JNI and moving away from periodic snapshots harmed the performance, JPaxos+epochs was compared against mPaxosSM@RAM. To estimate the overhead of PMDK and pmem, mPaxos and mPaxosSM were compared for three backends: @RAM, @emulp, and @pmem. Comparing @RAM against @emulp tells how much performance difference should be attributed to software, and comparing @emulp against @pmem – hardware.

7.3.4 Impact on system throughput

To measure the system throughput the KV-Map state machine was run on three replicas and 400 clients were sending requests – either a 8kB `put` (50% requests), or a 9B `get` (50% requests). So the average request size was 4kB. The size of the requests was selected so that increasing it further does not increase significantly the throughput in JPaxos. The systems were tested with two snapshot sizes: 10MB and 100MB, which corresponds, respectively, to 1k and 12.5k unique keys in the KV-Map.

Two metrics are presented: *system throughput* (or *throughput*, in short) in *requests per second (RPS)* and *leader uplink use*, i.e., the use of leader's outgoing network bandwidth, in *bits per second (bps)*. The leader's outgoing network link is the most busy one in the system. When the network limits the performance, then leader uplink use approaches 10Gbps. The link use was calculated using network interface statistics and is expressed it in bits per second, divided by 10^6 to get Mbps. The experimental evaluation results are summarized in Table 7.3 and discussed below.

Throughput with periodic snapshots

First the performance of systems configured to create small snapshots (10MB) is discussed. JPaxos+epochs uses 9.6Gbps of leader uplink, executes 106k RPS, and the network bandwidth limits the performance. mPaxos@RAM uses 9.3Gbps of leader uplink and executes 100k RPS, so is 5% slower than JPaxos+epochs. This is the cost of

Table 7.3: System throughput with 4kB requests

System and backend or SS (SSD, pmem)	**Snapshot size**	**Leader uplink [Mbps]**	**Throughput [RPS]**
JPaxos+epochs	10MB	9 685.44	106 278.43
mPaxos@RAM		9 338.53	100 653.97
mPaxos@emulp		9 221.89	99 721.62
mPaxos@pmem		8 928.79	96 038.44
JPaxos+SS (pmem)		8 977.74	97 904.78
JPaxos+SS (SSD)		8 072.67	83 265.09
JPaxos+epochs	100MB	8 851.13	93 294.19
mPaxos@RAM		7 254.97	72 936.65
mPaxos@emulp		6 936.74	68 666.71
mPaxos@pmem		6 357.79	64 492.50
JPaxos+SS (pmem)		7 440.99	76 019.19
JPaxos+SS (SSD)		6 231.96	62 100.08
mPaxosSM@RAM	–	9 804.99	106 049.49
mPaxosSM@emulp		7 235.30	79 077.20
mPaxosSM@pmem		5 514.80	60 395.49

using JNI – the only difference in failure-free operation between JPaxos+epochs and mPaxos@RAM is that the former stores the data items described in Section 6.2.2 using standard Java data structures, while the latter uses equivalent standard data structures in C++. The performance of the data structures in C++ is no worse than in Java. However, JNI (the mechanism responsible for calling C++ code from Java) brings the observed performance penalty. This could be eliminated if the systems were fully implemented in a language with efficient native pmem support (e.g., C++). Especially handling a snapshot causes moving lots of data between Java and the shared library (C++), and the memory allocations and deallocations in C++ happen in critical path, while in Java there is no need to copy the data within main memory and at least freeing memory is done concurrently. Handling each new application snapshot in mPaxos@RAM momentarily halts the system, and for the rest of the time the network limits throughput. In mPaxos@pmem the use of leader uplink is 8.9Gbps and 96k requests are processed per second. This is 10% less RPS than JPaxos+epochs and 5% less than mPaxos@RAM. mPaxos@pmem, akin to mPaxos@RAM, halts the system while a snapshot is processed. The 5% drop in performance can be attributed to the latency of pmem. In this test, the network, not the performance of pmem, was the main cause of RPS limit.

With larger snapshots (100MB) the system throughput decreases for all systems/ backends. JPaxos+epochs executes 12% less requests, while mPaxos@RAM and mPaxos@pmem, respectively, 28% and 33%. The magnitude of the throughput drop is high for the systems using JNI (mPaxos@RAM and mPaxos@pmem), and moderate for JPaxos+epochs. Hence, JNI is a major cause for the RPS drop, but the impact of limited pmem performance is also visible.

The results show that mPaxos@pmem with small snapshots is slower (by 10%) than JPaxos+epochs, but the latter system restricts the number of simultaneous failures, so it is less practical. mPaxos@pmem is also slightly slower (by 2%) than JPaxos+SS (pmem) that uses persistent memory as stable storage. However, when one deducts the cost caused by JNI (which is estimated to be around 5% by comparing mPaxos@RAM with JPaxos+epochs), mPaxos@pmem is expected to be slightly faster than JPaxos+SS (pmem), both of which tolerate catastrophic failures, and about 20% faster than JPaxos+SS (SSD) that uses SSDs as stable storage. On the other hand, mPaxos@pmem with larger snapshots is 31% slower than JPaxos+epochs, and is also slower (by 15%) than JPaxos+SS (pmem). As before, this cost includes JNI overhead, which for large snapshots is around 22%, when comparing mPaxos@RAM with JPaxos+epochs. So, one should expect that mPaxos@pmem will be no worse than JPaxos+SS (pmem) provided one gets rid of JNI. On the other hand, the current implementation of mPaxos@pmem (with JNI) outperforms for large snapshots JPaxos+SS that writes to SSDs by 4%.

Throughput with on-demand snapshots

As a baseline to evaluate mPaxosSM's throughput mPaxosSM@RAM was selected. mPaxosSM@RAM uses 9.8Gbps of the leader uplink and processes 106k client requests per second. This saturates the leader uplink, and is on par with JPaxos+epochs. So, redesigning the system to the new state machine API and truncating the log at will did not incur any performance problems.

With mPaxosSM@emulp the leader uses only 7.2Gbps of its outgoing link and the throughput drops to 79k RPS – 26% less than mPaxosSM@RAM, even though mPaxosSM@emulp still writes to DRAM (emulated pmem) rather than real pmem. The only difference between mPaxosSM@emulp and mPaxosSM@RAM is that the former uses the libpmemobj-cpp library of PMDK and writes to memory using PMDK transactions, while the latter uses standard C++ data structures. On the other hand, mPaxosSM@pmem uses 5.5Gbps of leader uplink and processes 60k RPS, so it is 44% slower than mPaxosSM@RAM and 24% slower than mPaxosSM@emulp. Note that mPaxosSM@emulp and mPaxosSM@pmem share the same code; the only difference is that the former writes to pmem emulated in DRAM, while the latter writes to real pmem. Apparently the use of PMDK is responsible for about half of the total performance drop on the system throughput, while the difference in performance between DRAM and pmem accounts for the second half.

It is hard to overcome the performance drop caused by pmem being less performant than DRAM. The main cause of this hardware-related drop is the limited write bandwidth from a single thread to pmem. A `put` request in mPaxosSM must to be persisted (in a single thread that executes the requests) in the KV-Map and in the responses map (so in total twice), and a response to a `get` request must be persisted in the responses map (so in total once). Thus, with 50% `get` and 50% `put`, on average a request needs 1.5 times the size of a `put` worth of data to be written to pmem. That amounts to $60\text{kRPS}\cdot 8\text{kB}\cdot 1.5 \approx 700$ MB/s, what is close to the bandwidth limit for writes from a single thread (see Section 5.1.4).

Profiling mPaxosSM

To tell the exact reason of the throughput drop in mPaxosSM caused by software support for pmem – the one seen when comparing mPaxosSM@emulp against mPaxosSM@RAM – VisualVM profiler and Linux's `perf` profiler were used. The profilers found out that the performance of the mPaxosSM backends using PMDK is limited by a thread that constantly uses 100% CPU. This thread is responsible for unbatching decided instances and passing commands one by one to the state machine for execution and storing the replies in the reply map. These operations must be executed sequentially (cannot be parallelized).

The thread limiting performance was profiled with the `perf` tool both as part of a normally running mPaxosSM@pmem system as well as run in isolation (with mock input). The preliminary version of mPaxosSM extensively used PMDK's transactions and memory allocator, and as a result in 67% of the samples, the profiler hit either PMDK's transaction upkeep, or PMDK's memory allocations. Most of the time was spent in copying data accessed from within a transaction into the undo log (`pmemobj_tx_xadd_range_direct`, 26.7% of CPU time), and in committing the transaction (`pmemobj_tx_commit`, 21.1% of CPU time). Allocating memory (`pmem::obj::make_persistent<>`) took 8.8%, and freeing it (`pmem::obj::delete_persistent<>`) took 3.9% of CPU time. The results show a huge cost of creating transaction logs and memory allocation. The transactions are there to guarantee that a crash at any point of time leaves the persistent memory in a state that can be recovered to a consistent one, so to some extent this cost is unavoidable.

To reduce the cost of the transactions and allocations a custom list of free memory blocks was created. When a large memory block (such as a command) in mPaxosSM is no longer used, it is added to a list of blocks to reuse (instead of being freed). When a new memory block is needed and there is a block of corresponding size in the list, it is taken from the list rather then allocated from PMDK. In applications as specific as mPaxosSM and KV-Map such list can be simple enough to be noticeably faster than the general-purpose PMDK allocator. Another nontrivial change to raise performance was done in KV-Map upon replacing an old value for a given key with a new value: instead of overwriting the old value, a new memory block is taken from the list (or allocated if none in the list suits) and the new value is written to it, and finally the pointer to the value is updated (and the old block ends up in the list of blocks to reuse). This may seem counter-intuitive, however in pmem overwriting the old value requires to copy it to the transaction log first, so one has to write both old and new value to pmem upon overwriting, and upon using a new memory block one can write to it without checkpointing the old value. Importantly, this gain is visible only if the list of blocks to reuse is in use, else the cost of PMDK memory allocator exceeds the gain. After these changes, overall throughput increased by 20%, and `perf` shows that committing transaction takes 7.5% of the Replica thread time, copying data accessed from within a transaction into the undo log takes 21.0% of the time, and operations on the custom free block list takes 11.7% of the time. For details, see the flame graphs embedded within this PDF file.

Throughput for different request sizes

So far, all results were presented for an average request size of 4kB. Here, the throughput for request of average sizes in range from 128B to 10kB is discussed. The results are obtained by running KV-Map in each evaluated system with three replicas with different request sizes, window sizes (3÷10) and batching sizes (32kB÷320kB). Each setup was measured three times. For every system and request size an average of 10 highest RPS figures was calculated. To compare the throughput for different request sizes, it is presented in MBytes/s, calculated as RPS multiplied by the average request size. The size of a snapshot is in order of 10MB. Figure 7.9 presents the vital part of the evaluation results (up to 6kB).

Overall, JPaxos+epochs and mPaxosSM@RAM are on top with almost the same throughput. For small requests (< 1kB) all systems with all backends are limited by the CPU, and apart from mPaxosSM@emulp and mPaxosSM@pmem have a similar throughput. The latter two use more CPU for PMDK and pmem, so they perform worse. For moderate requests (1kB÷3.5kB) mPaxos (with any backend) outperforms JPaxos+SS (pmem). With sufficiently large requests the throughput of all systems apart from mPaxosSM@emulp and mPaxosSM@pmem is limited by the

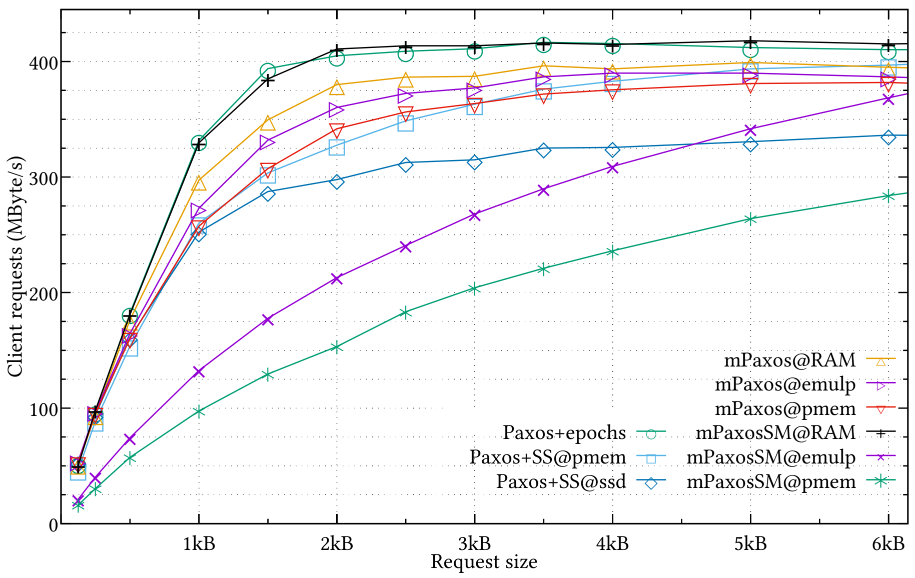


Figure 7.9: System throughput for different request sizes.

available network bandwidth and (except for mPaxosSM@RAM) by the momentarily halts when periodic snapshots are created. When the requests are 2.5kB or larger, mPaxos@RAM, mPaxos@emulp and mPaxos@pmem are, respectively, 5%, 7% and 10% slower than JPaxos+epochs. With requests beyond 8kB JPaxos+SS approaches the performance of JPaxos+epochs by 1% when it uses pmem, but it is 20% slower if it uses SSDs. However, JPaxos+SS (with SSDs or pmem as SS) requires four times as many commands to be batched in each instance to reach top performance when compared to any other system. Each snapshot contains 1000 key-value pairs and last response for each client, so while the size of the snapshot was in order of 10MB, it differed across runs with different request sizes and client counts. Hence with larger requests there is a slight drop in performance for all systems that need to create snapshots periodically. The throughput of mPaxosSM@emulp raises, with rising request size, to eventually match performance of mPaxosSM@RAM with 9kB requests. mPaxosSM@pmem throughput also raises, but only until the full write bandwidth of pmem from Replica thread is used up.

Impact of size and frequency of creating snapshots

As indicated in the previous sections, creating snapshots periodically impacts the performance. First, the state machine must create the snapshot, second, the Framework must add metadata to the state machine snapshot, truncate Paxos log and (in FullSS and mPaxos) store the snapshot persistently. The size of a snapshot is application-specific. Obviously larger snapshots take more time to handle. Making snapshots infrequently is usually not an option – the Paxos log can be truncated only up to the last snapshot, so with throughput as high as 400MB/s, after 10 seconds the size of the log alone would amount as much as 4GB, what is considered a value too large for a middleware component in many applications. Therefore, in this section the impact on throughput of snapshot size and the frequency of creating it is tested.

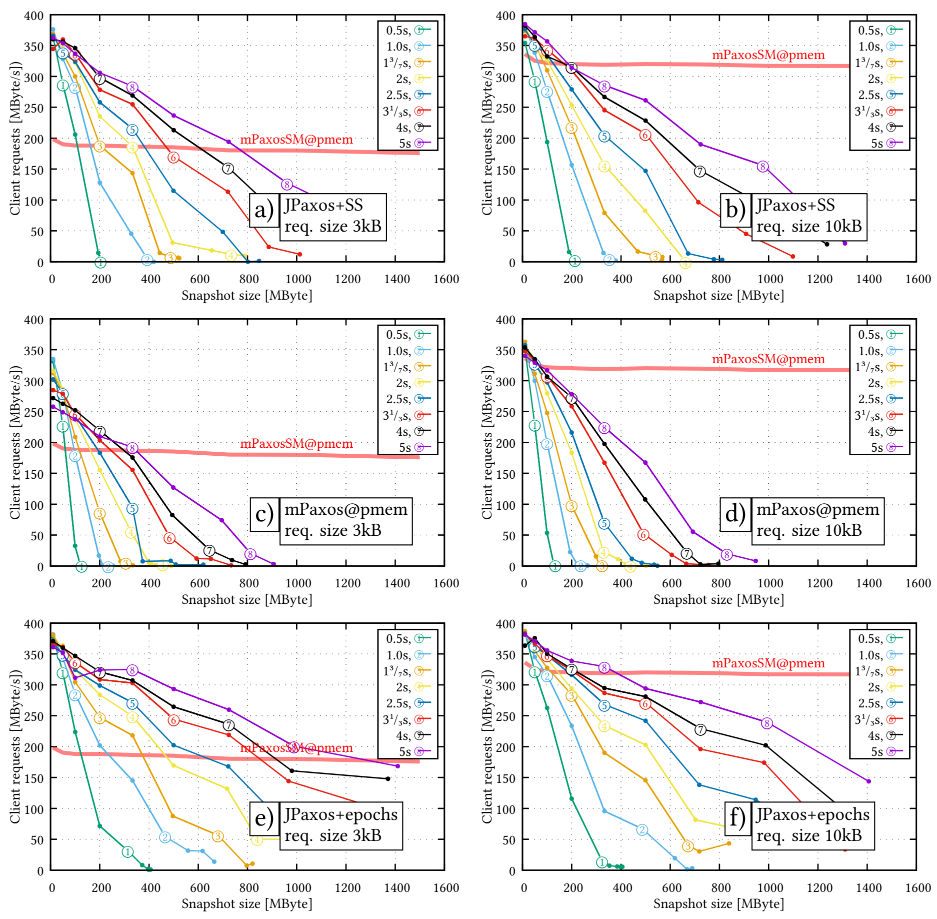


Figure 7.10: Throughput of JPaxos+SS, mPaxos@pmem and JPaxos+epochs for 3kB and 10kB average request size with different size of snapshot and snapshot creation frequency. Corresponding mPaxosSM@pmem throughput depicted for reference.

In the benchmark three KV-Map replicas were run and an appropriate number of clients were sending continuous stream of requests: either average size of 3kB (50% of 6kB `put` and 50% of 9B `get`), or of 10kB (50% of 20kB `put` and 50% of 9B `get`). The throughput of a 20s long period was analyzed. Each single test point had a constant period between commencing snapshot creation of either 0.5s, 1s, $1\frac{3}{7}$s, 2s, 2.5s, $3\frac{1}{3}$s, 4s or 5s. If creating a snapshot took more than the interval, than a new snapshot was created immediately after the old. Creating the snapshot on each two replicas was coordinated to overlap as little as possible (cf. Sequential Checkpointing in [BSF+13]) – a replica *id* created snapshot when the clock time t and the interval i satisfied $t \bmod i = \mathit{id}/3{\cdot}i$. The number of keys in the KV-Map was selected to form snapshots of either 10MB, 50MB, 100MB, 200MB, 333MB, 500MB, 725MB, 1000MB or 1500MB in size. This size would be reached if all keys were used by clients (clients choose a key for each request randomly), what not always was the case, hence for each test the real snapshot size was recorded. The results for JPaxos+SS, JPaxos+epochs and mPaxos@pmem are presented in Figure 7.10, together with the throughput of mPaxosSM@pmem for the same number of keys in KV-Map as a reference.

The results are far from surprising: the larger the snapshot or the more frequently created snapshot, the lower throughput of the system. In some combinations (e.g., snapshots over 400MB initiated every 0.5s) the system throughput dropped to zero, for creating that large snapshot took more time than the 0.5s. JPaxos+epochs (7.10e,f) decreased its throughput relatively less with rising snapshot size/frequency that the other systems, for it does not have to persist the snapshot. For the latter, mPaxos@pmem (7.10c,d) needs more time than JPaxos+SS (7.10a,b), hence FullSS is losing less throughput with rising snapshot size/frequency than mPaxos. The trends do not vary with request sizes, and as expected, for the larger requests sizes the throughput was slightly better in all cases.

The interesting part of the results of this benchmark is the comparison with mPaxosSM@pmem. mPaxosSM does not create snapshots periodically. Hence, its results are almost disaffected by the number of keys in KV-Map (apart a small raise in throughput when the number of keys is low, what might be attributed to fitting the data in CPU caches or coalescing pmem writes in WPQ). For every request size there is a boundary of snapshot size and its creation frequency when mPaxosSM@pmem outperforms each system. For 3kB requests (7.10a,c,d) with moderate snapshots (≤500MB) both JPaxos+SS and JPaxos+epochs can be configured to be faster than mPaxosSM@pmem while the Paxos log size is still less than 1GB. However, if one wants to limit the log size to 1GB, then with 3kB requests and snapshots over 500MB all systems have lower throughput than mPaxosSM@pmem. Keep in mind that apart from the log one must fit in memory also the snapshot and the state machine (that requires at least as much memory as the snapshot), so for throughput *th*, snapshot interval i and snapshot size s the memory footprint is at least $th{\cdot}i{+}2{\cdot}s$. For 10kB requests, mPaxosSM@pmem outperforms in the tests mPaxos@pmem for snapshots ≥100MB, JPaxos+SS for snapshots ≥200MB, and JPaxos+epochs for snapshots ≥400MB. So, this benchmark shows that mPaxosSM is better than competitors if the snapshots are large enough. Obviously, one might just do snapshots less frequently to boost performance, but that leads to huge main memory usage and, last but not least, increased recovery time, for upon recovery all requests following the last snapshot need to be executed.

7.3.5 Restarting after crash

Restarting after crash in mPaxos

Recovery and catch-up of a crashed replica in FullSS, ViewSS and EpochSS is evaluated in Section 7.2.5. The procedure of restarting a replica in mPaxos@pmem differs from JPaxos+SS only in the respect that the latter must additionally read the logs from stable storage. The performance of catching up is to some extent affected by the need to store persistently the snapshot and newly learned decisions. The tests from Section 7.2.5 are repeated here, but this time with JPaxos+epochs, JPaxos+SS and mPaxos@pmem, and with KV-Map state machine rather than EchoService. Essentially, the tests measure how long it takes a replica to recover and catch up. Three replicas are started and 1k clients continuously send requests (3kB on average). The periodically created snapshot has 200MB. A replica is crashed (let denote it R_r), and restarted after 20s. The results are presented in Figure 7.11 and discussed below.

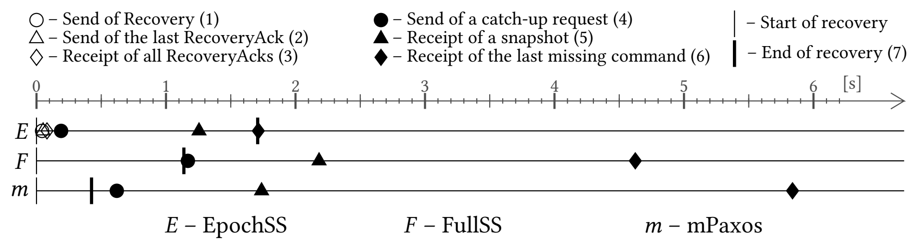


Figure 7.11: The time a replica needs to recover and catch-up while the system constantly serves 3kB client requests.

With JPaxos+epochs, R_r starts in 0.04s and needs another 0.04s to exchange the recovery messages. Then it executes the catch-up protocol, which takes 1.6s. In JPaxos+epochs, R_r can send any Paxos messages only once the catch-up succeeds, that is 1.7s from the restart. In JPaxos+SS, R_r can reply to the Paxos messages once it restores its state from stable storage, which takes less time – 1.14s. But the recovered state is outdated, so JPaxos+SS starts catch-up once it learns that other replicas advanced while R_r was down. When R_r runs with mPaxos@pmem, it can send any Paxos messages immediately after restart. Starting R_r, which includes recovering the application from the snapshot (read from pmem), takes as little as 0.42s. Obviously, R_r must also catch up with other replicas to update its state. Catching up takes least time in JPaxos+epochs – 1.5s, while JPaxos+SS requires 3.5s and mPaxos@pmem 5.2s. This difference in the catch-up duration is mainly related to the cost of persisting received snapshot and decisions – JPaxos+SS and mPaxos@pmem need to write the data to, respectively, stable storage (which is pmem in the tests) and pmem. However, JPaxos+SS writes to pmem using standard file API that has higher latencies and involves kernel in the I/O, allowing thereby to surpass the single-thread write bandwidth limit inherent to pmem.

Restarting after crash in mPaxosSM

A replica with mPaxosSM needs to do even less upon being restarted than one with mPaxos. There is no need to restore the state machine from snapshot, for it is required to persist its state. However, at catch-up, the peer replica that is queried for the state has to create a snapshot. Apart from this difference, the code executing the catch-up procedure is identical in mPaxosSM@pmem and mPaxos@pmem. Notice that the catch-up procedure ends once all missing decisions from a peer are fetched, not when the commands that are in the decisions have been executed. In this test the duration of restarting a replica, catching up with the decisions and catching up with execution is evaluated. Also, the impact of restart on the entire system is discussed.

This experiment was repeated for three and five KV-Map replicas. All replicas were started, and 1k clients were sending non-stop either `get` (50%) or 6kB `put` (50%) requests. In the 30th second of the test one of the replicas was crashed (R_r), and in the 50th second it was restarted. The size of the state machine's pmem was set to 512MB[2]. Typically a client sends requests repeatedly to the same replica as long as it

[2]One reason to use a larger size than the snapshot's size used in the previous subsection, is that the results are more pronounced, and the other is that the size of pmem is the memory capacity of a state machine, while a snapshot prepared by it contains only the vital, and possibly compacted, data.

receives replies on time. Otherwise, the client connects to another replica and stays connected. To tell when R_r catches up with execution and so is ready to answer clients on time, the clients are forced to change replicas every 1k requests. Once the clients connected to R_r get their reply in a timely manner, R_r is considered as fully functional.

Figure 7.12 shows how the network outgoing link usage changes for all replicas when a follower crashes and restarts in a system with three replicas. It takes about 0.3s to start a process, that is, start Java VM, call the code initializing mPaxosSM@pmem, set up pmem memory mappings, establish network connections, etc. More precisely, it takes 0.23s from the start of R_r to establish connections to other replicas (from now on the replica can reply to Paxos messages), and after another 0.07s a catch-up query is sent to a replica R_f. In response, R_f generates a snapshot in 0.38s. For this time the processing on R_f stops. Transmitting the snapshot (518MB) from R_f to R_r takes 0.85s. Then, R_r spends 0.55s on writing the snapshot to pmem, and another 0.23s restarting the state machine on the newly written pmem. This concludes the catch-up protocol. Altogether, it takes 2.3s to start R_r anew and apply a snapshot received from a peer. The total number of replicas does not impact the results.

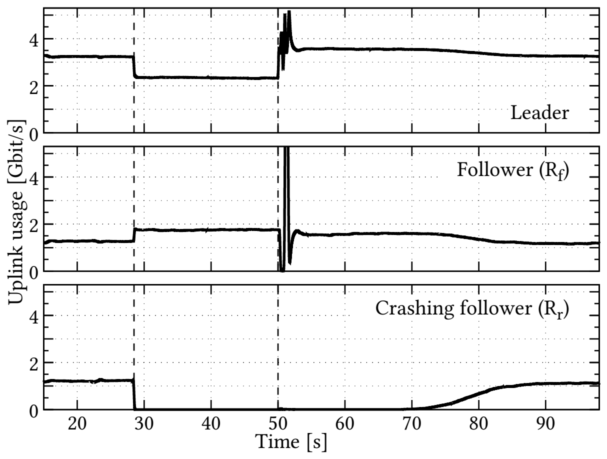


Figure 7.12: Outgoing link usage while a follower crashes and later restarts in mPaxosSM@pmem with three replicas.

Figure 7.13 presents how the system throughput changes for a system with three and five replicas when the leader crashes and restarts. The system throughput drops slightly at crash, as with R_r down each replica must handle more clients. Once the recovering R_r executes the catch-up, the system with three replicas suffers a severe throughput drop, as R_r is catching up and R_f is preparing a snapshot, so only one replica is operational. With five replicas the impact of the catch-up protocol on the overall throughput is moderate – R_r is catching up and R_f creates a snapshot, and three remaining replicas (which is a majority) are taking new decisions and replying to the clients non-stop. Once the catch-up concludes, the

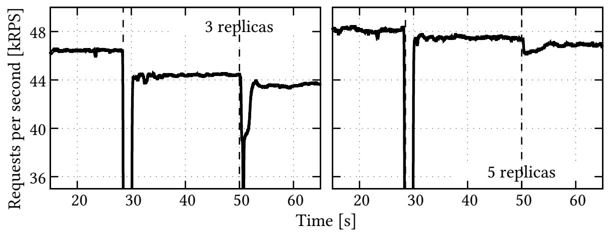


Figure 7.13: Throughput upon leader crash and restart in mPaxosSM@pmem of a replica that is not R_f and R_r.

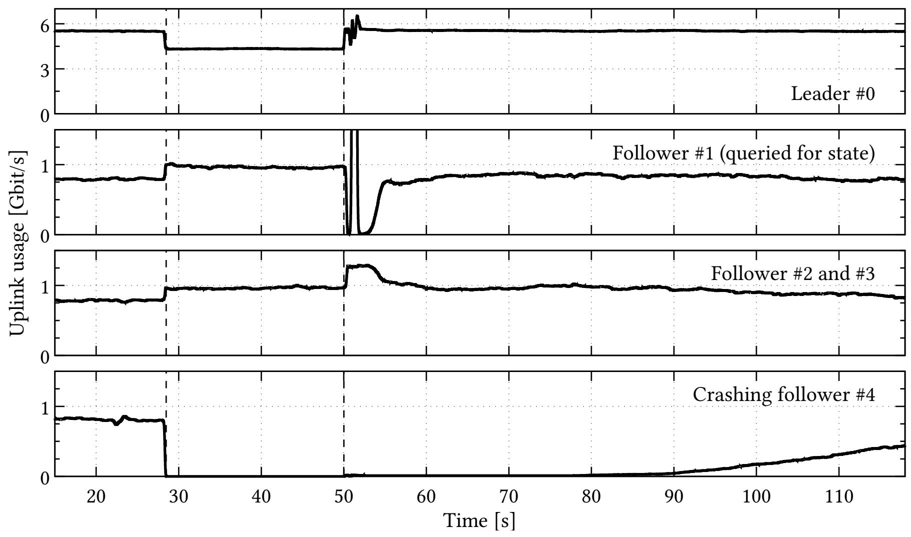


Figure 7.14: Outgoing link usage while a follower crashes and later restarts in mPaxosSM@pmem with five replicas.

throughput is not immediately restored, as while R_r knows all decided commands, it still needs to execute them.

The throughput of the system is fully restored as soon as R_r responds to clients in a timely manner. A good indication of this is the R_r's network outgoing link usage. As long as R_r is not responding in time, clients infrequently send new request to it, so it sends out mainly Accept messages and infrequently forwards batches of client request to the leader, so the uplink use is low. Once R_r is responding in time, then it frequently sends the responses to the clients and forwards batches with new requests to the leader, so the uplink use is moderate. As seen in Figure 7.12, at R_r's crash (28s) the clients it handled move to other replicas. R_f's uplink usage rises, and leader's uplink drops by the data that it no longer sends to R_r, but also rises by the data sent to extra clients. At recovery, when R_r has updated its state from R_f's snapshot, the updated state is 2s outdated (this is the time it takes to create, transfer and apply a snapshot). So, R_r must not only execute newly decided commands, but also commands decided for these 2 seconds. Once it completes catching up with execution, its outgoing link usage rises by the data sent back to the clients (70÷85s). So, it takes about 35 seconds from the start of R_r until the throughput from before the crash is restored in a system with three replicas. With five replicas (see Figure 7.14) restoring throughput takes 90 seconds. It takes longer because the system throughput drops less at crash in a system with five replicas, as handling the clients previously connected to R_r spreads over four rather than two replicas, so R_r has less CPU time left to catch up with execution, as the CPU is more occupied by the current requests.

Chapter 8

Conclusions

This thesis proposes methods for supporting crash recovery in Paxos-based State Machine Replication systems that improve over the well-known method which persists vital Paxos protocol data on a storage device which survives replica crashes (and is denoted for the purpose of the thesis as FullSS). The first two proposed approaches, ViewSS and EpochSS, require an assumption that always enough replicas are up to guarantee availability of the system (or else no replica will be able to recover anymore). Such assumption is not a whim. The designers of Paxos-based systems already put tremendous effort to keep all the time a majority of replicas up to provide uninterrupted operation of the systems. ViewSS and EpochSS minimize the use of stable storage during normal (non-faulty) system operation, and upon recovery fetch from other replicas the data that was the lost on crash. mPaxos and mPaxosSM, the two other approaches for supporting recovery, leverage persistent memory to reduce the cost of storing durably the vital Paxos data as well as to improve the recovery time. Unlike ViewSS and EpochSS, mPaxos and mPaxosSM do not impose any restriction on the number of simultaneous replica failures. ViewSS, EpochSS, mPaxos and mPaxosSM were discussed in detail and contrasted with FullSS. All the five were implemented in JPaxos for the purpose of throughout evaluation and experimental comparison.

Robustness of recovery Not surprisingly, restoring lost state is the main factor of the total time of recovery in FullSS, ViewSS and EpochSS, even in tests with an application as simple as an EchoService. mPaxos and mPaxosSM store their state in pmem, so it is ready to use right after starting a replica anew, yet mPaxos needs to restore the state of the state machine, which still takes less time than in FullSS, ViewSS or EpochSS. mPaxosSM is the fastest to restore the state of a replica to a state it had on crash, mPaxos second, then FullSS followed by, almost on par, ViewSS and EpochSS (the last two restore a more recent state). Once the state is restored, the recovering replica can order new commands. However, to execute the commands, it first needs to learn all that it missed during downtime, and catch up with executing the commands with the rest of the replicas. So, in all systems the recovering replica has to run a catch-up procedure to learn the commands passed while it was down. ViewSS and EpochSS use this procedure to recover data, and are first to complete it (in a comparable time duration). mPaxosSM is the third best, for it uses pmem of the state machine as an ever-fresh snapshot, so the recovering replica neither has to fetch many commands following the snapshot nor has to reconstruct state of the state machine from the snapshot. Worst in catch-up completion time are FullSS and mPaxos, that need also to store durably all the data fetched from other replicas. FullSS also reads the lost state from stable storage first and later recovers the state again using data received from peers, what takes extra effort. While mPaxosSM recovers and catches up all missing Paxos instances fast, it is the slowest to catch up executing the commands, because the state machine in mPaxosSM has to persist its state by maintaining it in pmem, and the current pmem hardware has limited performance.

Impact of recovery on the system When a replica recovers, it informs other replicas about it (in ViewSS and EpochSS) and fetches the current state from some replica (always in ViewSS/EpochSS, and in case of FullSS, mPaxos and mPaxosSM in the most common situation when the system advanced while the recovering replica was down). Thus, the recovery process affects, to a varying degree, the performance of the system. In case of ViewSS, it is likely that a new leader has to be elected amidst recovery procedure, which effectively prevents the system from deciding commands for a while. This had an observable negative impact on the throughput of whole system when ViewSS used an HDD as the stable storage, and negligible impact with stable storage that has better performance. In mPaxosSM a snapshot is crafted only on demand, and recovery of a process creates such demand (assuming that the system advanced since the crash). Hence, starting a replica anew in mPaxosSM usually halts executing commands on the replica that sends the state to the newly restarted one. In a system with three replicas, this results in a brief but noticeable throughput drop, for only one of the replicas can answer client requests (the second is out of date, the third is creating snapshot). With more replicas in the system the effects dwindles. Regardless of the recovery method, when one replica has to catch up, some other process is slowed down, as it has to transfer data to the recovering process. However, except the mPaxosSM case discussed above this only slightly slows down the whole system. In the tests, 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 catch up finished.

Cost of supporting recovery during fault-free operation Supporting crash recovery in Paxos-based systems has its cost. The thesis explained and evaluated the cost of each proposed system compared to a system that does not support recovering crashed replicas (denoted as crash-stop). FullSS is know to negatively impact latency and throughput, and while it is possible to marginalize this impact in some workloads, FullSS is still considered bad for its performance impact on fault-free operation. In the evaluation the throughput of whole system dropped due to the ability of recover from failures using FullSS from as much as 34% when the system was serving small requests (and using RAM disk as stable storage), to as little as 1% with suitably large requests. In contrast, EpochSS and ViewSS retain full performance of a crash-stop system, and can recover crashed replicas as long as a majority of the replicas is up. Noteworthy, no downsides of using EpochSS instead of a crash-stop system were identified. ViewSS only increases the latency of leader election, what can hardly be considered a performance problem. However, neither EpochSS nor ViewSS is suitable if one must support recovery in a case when a majority (or more) of replicas crash at the same time. The mPaxos system supports crash of any number of replicas at a time, and performs strictly better than FullSS using an SSD as the stable storage. When compared to FullSS that uses pmem as the stable storage, mPaxos has higher throughput for some workloads, and is on par for other workloads. Importantly, these results were obtained with the current pmem hardware which is still immature, so one shall expect an improvement of the performance as pmem matures. No-crash systems, FullSS, ViewSS, EpochSS and mPaxos assume circumstances in which the state machine creates periodically a snapshot of its state, or else they must not truncate the ever-growing Paxos log. The mPaxosSM system changes what is required from the state machine: now it has to persist its state after every single operation it executes. With pmem, unlike other non-volatile storage devices, this can be done with a reasonable latency and throughput. In the evaluation the cost of doing so was clearly visible – the thread executing commands was always busy, and this limited the throughput. However, this change has a deeper significance: mPaxosSM does not suffer from the cost of creating snapshots periodically. While the performance of other systems drops with growing cost of creating snapshots, mPaxosSM performance is indifferent to how often the snapshot is created and how much resources are consumed upon crafting it. In systems that need to create snapshots periodically the time period between each two snapshots must be short enough for the size of all commands issued since the last snapshot stays within sane bounds, and in the evaluation this size grows by 400MB/s when the network is saturated. Hence, regardless of the resources consumed by creating a snapshot, it must be created sufficiently frequently. In many workloads this degrades performance of FullSS, ViewSS, EpochSS and mPaxos to such extent that mPaxosSM outperforms all the competitors. For instance, in the evaluated KV-Map, with 120k unique keys with 6kB value each, mPaxosSM outperforms both FullSS and mPaxos assuming one wants to keep main memory usage below 3GB, similarly with 33k unique keys with 20kB value each, mPaxosSM outperforms EpochSS for the same main memory usage limit.

Persistent memory The Intel Optane DC Persistent Memory Modules (PMM) and Persistent Memory Development Kit (PMDK) libraries emerged quite recently. Their strengths and weaknesses as well as implications of main memory being persistent are not yet fully recognized. While traits such as latency, scalability, endurance and energy use were widely discussed with every new non-volatile memory technology, the systems used in this thesis stumbled upon the limited write bandwidth, especially if a single thread needs to write the data – in such case pmem is slower than modern mass storage devices. The software support for pmem is already quite powerful, but has its pitfalls. Some major programming languages still lack a good support for pmem. While to support pmem in an application it is sufficient to select which data has to be stored in pmem and to add adequate transactions, the efficiency of resulting code is suboptimal. Replacing readily available library solutions with custom routines in key points in the code improved performance of mPaxosSM by 20%, which is unexpectedly high gain for replacing dedicated libraries with hand written code.

Selecting the right recovery support method To choose the right approach for implementing recovery, one has first to consider the workload. That is, estimate the resources needed to create a snapshot. If creating the snapshot is expected to be particularly resource-consuming, then mPaxosSM is preferred. In case when creating snapshots periodically does not degrade performance too much, one must answer the question whether support for catastrophic failures is required. Without such need, EpochSS should be considered, as it provides the performance of a crash-stop system while offering a decent ability to recover from replica crashes. Otherwise, when one must lose no data upon crash of more than half of the replicas and be able to recover the system if such catastrophic crash occurs, mPaxos and FullSS remain for choice. While the former is the better choice, it requires pmem that will still take time to become a widespread hardware.

Acknowledgements

A part of the presented research is supported by the Foundation for Polish Science, within the TEAM programme co-financed by the European Union under the European Regional Development Fund, grant No. POIR.04.04.00-00-5C5B/17-00. Project web site: http://www.cs.put.poznan.pl/persistentdatastore

The experimental evaluation presented in Section 7.3 was conducted on hardware loaned by Intel Poland.

Bibliography

[ABD95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[ACT98] Marcos K. Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing, DISC '98*, September 1998.

[ADGFT03] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceedings of the 22th annual symposium on Principles of Distributed Computing, PODC '03*, July 2003.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[BAM19] Daniel Bittman, Peter Alvaro, and Ethan L. Miller. A persistent problem: Managing pointers in NVM. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems, SOSP '19*, 2019.

[BBCR22] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent memory: A survey of programming support and implementations. *ACM Computing Surveys*, 54(7):152:1–152:37, 2022.

[BBH$^+$11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI '11*, March-April 2011.

[BDFG03] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, March 2003.

[BFAP16] Peter Bailis, Camille Fournier, Joy Arulraj, and Andrew Pavlo. Research for practice: Distributed consensus and implications of NVM on database management systems. *Communications of the ACM*, 59(11):52–55, 2016.

[BHC$^+$13] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW '13*, November 2013.

[BKK⁺12] Bartosz Bosak, Jan Kończak, Krzysztof Kurowski, Mariusz Mamoński, and Tomasz Piontek. Highly integrated environment for parallel application development using QosCosGrid middleware. In *Building a national distributed e-Infrastructure - PL-Grid - scientific and technical achievements*, volume 7136 of *Lecture Notes in Computer Science*, pages 182–190. Springer, 2012.

[BPvR14] Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. Scalable State-Machine Replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '14*, June 2014.

[BSF⁺13] Alysson Neves Bessani, Marcel Santos, João Felix, Nuno Ferreira Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proceedings of USENIX Annual Technical Conference, USENIX ATC '13*, June 2013.

[Bur06] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, November 2006.

[CDE⁺12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI '12*, October 2012.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '07*, August 2007.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[CJ15] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. In *Proceedings the 41st International Conference on Very Large Data Bases, VLDB '15*, Aug-Sept 2015.

[CSP08] Lásaro J. Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated agreement protocols for higher availabilty. In *Proceedings of the 7th IEEE Internat. Symposium on Networking Computing and Applications, NCA '08*, July 2008.

[CWB⁺20] Wentao Cai, Haosen Wen, H. Alan Beadle, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '20*, February 2020.

[DHK+15] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review*, 49(2):18–26, 2015.

[DHL+18] Huynh Tu Dang, Jaco Hofmann, Yang Liu, Marjan Radi, Dejan Vucinic, Robert Soulé, and Fernando Pedone. Consensus for non-volatile main memory. In *2018 IEEE 26th International Conference on Network Protocols, ICNP '18*, September 2018.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[EBR+20] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Fifteenth EuroSys Conference 2020, EuroSys '20*, April 2020.

[Fed20] Sasha Fedorova. We Replaced an SSD with Storage Class Memory. Here is What We Learned. https://engineering.mongodb.com/post/we-replaced-an-ssd-with-storage-class-memory-here-is-what-we-learned, August 2020.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[GKL20] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the 47th International Conference on Very Large Data Bases, VLDB '21*, 14(4), August 2020.

[GVVS20] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference, USENIX ATC '20*, July 2020.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for Internet-scale systems. In *2010 USENIX Annual Technical Conference, USENIX ATC '10*, June 2010.

[HMS16] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: quorum intersection revisited. In *20th International Conference on Principles of Distributed Systems, OPODIS '16*, December 2016.

[HPM+18] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC '18*, July 2018.

[HS21] Morteza Hoseinzadeh and Steven Swanson. Corundum: statically-enforced persistent memory safety. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21*, April 2021.

[HT20] Takahiro Hirofuchi and Ryousei Takano. A prompt report on the performance of Intel Optane DC Persistent Memory Module. *IEICE Transactions on Information and Systems*, 103-D(5):1168–1172, 2020.

[Isa07] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[IYZ+19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[JLM15] Leander Jehl, Tormod Erevik Lea, and Hein Meling. Replacement: Decentralized failure handling for replicated state machines. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS '15*, September-October 2015.

[JM14] Leander Jehl and Hein Meling. Asynchronous Reconfiguration for Paxos State Machines. In *Distributed Computing and Networking - 15th International Conference, ICDCN '14*, January 2014.

[JPa22] JPaxos, mPaxos, mPaxosSM – state machine replication tools based on Paxos with support of crash failure recovery. https://github.com/JPaxos, 2014–2022.

[JRS11] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance Broadcast for Primary-backup Systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks, DSN '11*, June 2011.

[KA08] Jonathan Kirsch and Yair Amir. Paxos for System Builders: An Overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS '08*, September 2008.

[KSZ+11] Jan Z. Kończak, Nuno Santos, Tomasz Żurkowski, Paweł T. Wojciechowski, and André Schiper. JPaxos: State Machine Replication based on the Paxos protocol. Technical Report EPFL-IC-TR-167765, EPFL, July 2011.

[KW21] Jan Z. Kończak and Paweł T. Wojciechowski. Failure recovery from Persistent Memory in Paxos-based State Machine Replication. In *40th International Symposium on Reliable Distributed Systems, SRDS '21*, September 2021.

[KWG16] Jan Kończak, Paweł T. Wojciechowski, and Rachid Guerraoui. Ensuring irrevocability in wait-free Transactional Memory. In *11th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT '16*, March 2016.

[KWG17] Jan Z. Kończak, Pawel T. Wojciechowski, and Rachid Guerraoui. Operation-level wait-free Transactional Memory with support for irrevocable operations. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3570–3583, 2017.

[KWS+21] Jan Z. Kończak, Paweł T. Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. Recovery algorithms for Paxos-based State Machine Replication. *IEEE Transactions on Dependable and Secure Computing*, 18(2):623–640, March-April 2021.

[Lam89] Leslie Lamport. The part-time Parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp., Palo Alto, September 1989.

[Lam96] Butler W. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms, WDAG '96*, volume 1151 of *LNCS*, October 1996.

[Lam98] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam01a] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, 2001.

[Lam01b] Butler W. Lampson. The ABCD's of Paxos. In *Proceedings of the twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC '01*, August 2001.

[Lam05] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[Lam06] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2), 2006.

[LC12] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, Computer Science and Artificial Intelligence Lab, MIT, July 2012.

[LFE+19] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. DynaStar: optimized dynamic partitioning for Scalable State Machine Replication. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS '19*, 2019.

[LMZ09] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC '09*, August 2009.

[LMZ10] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in Egalitarian parliaments. In *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles, SOSP '13*, November 2013.

[MBP14] Parisa Jalili Marandi, Carlos Eduardo Benevides Bezerra, and Fernando Pedone. Rethinking State-Machine Replication for Parallelism. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS '14*, July 2014.

[MDP16] Odorico Machado Mendizabal, Fernando Luís Dotti, and Fernando Pedone. Analysis of checkpointing overhead in parallel state machine replication. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16*, April 2016.

[MDP17] Odorico Machado Mendizabal, Fernando Luís Dotti, and Fernando Pedone. High performance recovery for Parallel State Machine Replication. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS '17*, June 2017.

[MDSG20] Tony Mason, Thaleia Dimitra Doudali, Margo I. Seltzer, and Ada Gavrilovska. Unexpected performance of Intel® Optane™ DC Persistent Memory. *IEEE Computer Architecture Letters*, 19(1):55–58, 2020.

[MJM08] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machine for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI '08*, December 2008.

[MPP12] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-Ring Paxos. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '12*, June 2012.

[MPSP10] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *International Conference on Dependable Systems and Networks, ICDCN '10*, June–July 2010.

[MPSS17] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering shared objects without stable storage. In *31st International Symposium on Distributed Computing, DISC '17*, October 2017.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC '88*, August 1988.

[OO14] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14*, June 2014.

[PIL$^+$19] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using Intel Optane DC Persistent Memory Modules. In *Proceedings of the International Symposium on Memory Systems, MEMSYS '19*, October 2019.

[PLL00] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243(1-2):35–91, 2000.

[RD17] Amitabha Roy and Subramanya R. Dulloor. Cyclone: High availability for persistent key value stores. *CoRR*, abs/1711.06964, 2017.

[RR03] Luís Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1206–1217, 2003.

[RST11] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. In *Proceedings of the 37th International Conference on Very Large Data Bases, VLDB '11*, August-September 2011.

[Sca20] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020.

[SS12] Nuno Santos and André Schiper. Tuning Paxos for high-throughput with batching and pipelining. In *Proceedings of the 13th International Conference on Distributed Computing and Networking, ICDCN '12*, January 2012.

[vRA15] Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys*, 2015.

[vRSS15] Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. Vive la différence: Paxos vs. Viewstamped Replication vs. ZAB. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2015.

[WK12] Paweł T. Wojciechowski and Jan Kończak. A formal model of crash recovery in distributed Software Transactional Memory. In *Proceedings of the 4th Workshop on the Theory of Transactional Memory, WTTM '12*, July 2012.

[WZL$^+$18] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*, March 2018.

[WZM$^+$19] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. On the parallels between Paxos and Raft, and how to port optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC '19*, August 2019.

[XJXS17] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of 2017 USENIX Annual Technical Conference, USENIX ATC '17*, July 2017.

[YKH$^+$20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST '20*, February 2020.

Streszczenie w języku polskim (abstract in Polish)

Algorytmy odtwarzania stanu dla
zreplikowanej maszyny stanowej
z pamięcią ulotną i nieulotną

We wstępie rozprawa opisuje przyczyny powstania i obszary zastosowań systemów opartych o zreplikowaną maszynę stanową, oraz charakteryzuje problem odtwarzania replik po awarii i wskazuje na słabe strony jego istniejących rozwiązań.

Następnie podane są definicje podstawowych pojęć potrzebnych do precyzyjnego przedstawienia poruszanych w rozprawie zagadnień, między innymi system rozproszony, częściowo synchroniczny, detektory awarii, rodzaje awarii sieci i węzłów, kryteria poprawności i postępu. Rozprawa rozważa model awarii pozwalający na odtwarzanie maszyn (crash-recovery) bez awarii bizantyjskich. Dalej definiowany jest problem konsensusu, atomowe rozgłaszanie, maszyna stanowa i replikowana maszyna stanowa, kworum.

Kolejna część rozprawy pokrótce przedstawia algorytm Paxos. Paxos pozwala, w systemach częściowo synchronicznych lub asynchronicznych z detektorem awarii, rozwiązać problem konsensusu, a tym samym zrealizować równoważne mu atomowe rozgłaszanie. Podany jest pseudokod dla algorytmu Paxos w dwóch wariantach – jednym w pełni zgodnym z artykułem L. Lamporta opisującym Paxos, drugi z wersją przystosowaną do replikowania maszyny stanowej, będący również odwzorowaniem algorytmu zaimplementowanego w prototypie użytym w ocenie wydajności proponowanych w rozprawie rozwiązań. Działanie algorytmów jest wyjaśnione, a algorytmy uzupełnione o procedurę nadganiania i inne znane z literatury optymalizacje. Następnie opisany jest framework replikacji maszyny stanowej. Określono przechowywanie jakich danych jest kluczowe zarówno dla samego algorytmu Paxos, jak i dla części odpowiedzialnej za faktyczne zreplikowanie maszyny stanowej i obsługę zewnętrznych klientów. Opisano sposób działania frameworku analizując kolejne kroki przetwarzania żądania klienta.

Po wprowadzeniu algorytmu Paxos i podejścia zreplikowanej maszyny stanowej dokonano przeglądu literatury. Na wstępie przedstawiono zarys dotychczasowych badań nad algorytmem Paxos. Dalej skupiono się na publikacjach poruszających temat odtwarzania stanu po awarii. Wskazano dotychczas proponowane metody zarówno dla omawianego w rozprawie algorytmu Paxos, jak i dla innych, zbliżonych algorytmów konsensusu takich jak Viewstamped Replication, ZAB czy Raft. Następnie wskazano proponowane w literaturze sposoby zmniejszenia kosztu nadganiania. Ostatnią poruszaną kwestią jest stan badań nad pamięcią trwałą, ze szczególnym

uwzględnieniem prac łączących pamięć trwałą i algorytmy konsensusu.

Kolejny rozdział rozprawy poświęcono pamięci trwałej. Opisano i wyjaśniono podstawowe własności pamięci trwałej, porównując ją do pamięci DRAM i masowej dla lepszego zobrazowania poszczególnych cech. Omówiono od kiedy przetwarzane dane mają zagwarantowaną trwałość zapisu, określono co oznacza adresowanie bajtowe i bezpośredni dostęp oraz scharakteryzowano parametry dostępnych na rynku produktów, takie jak opóźnienie, przepustowość (ze szczególnym naciskiem na przepustowość zapisu z jednego wątku) i pojemność. Następnie wskazano sposoby konfiguracji i administracji pamięci trwałej. Omówiono sposób w jaki można z tej pamięci korzystać w programach uruchomionych z poziomu użytkownika i wskazano główne wyzwania w tworzeniu takich programów, podając popularne i wystarczająco wydajne ich rozwiązania. Temat pamięci trwałej zakończono przeglądem bibliotek programistycznych i narzędzi użytych do tworzenia i oceny prototypów.

Pozostała część rozprawy poświęcona jest własnym badaniom i podzielona na opis proponowanych metod wspierania odtwarzania stanu replik po awarii i ocenę eksperymentalną i porównanie działania systemów zreplikowanej maszyny stanowej korzystających z tych metod. Opisane metody zostały pogrupowane na takie które pozwalają odtworzyć replikę niezależnie od ilości równoczesnych awarii, oraz takie w których jednocześnie mniej niż połowa replik może być niesprawna.

Do pierwszej kategorii należą systemy mPaxos i mPaxosSM, oraz opisany dla porównania z nimi, a znany z literatury algorytm FullSS (nazwany tak w rozprawie dla odróżnienia od innych) który zapisuje w pamięci nietraconej przy awarii kluczowe zmiany stanu przetwarzania. Wadą algorytmu FullSS jest konieczność częstego synchronicznego zapisu do takiej pamięci. Pamięć nietracona przy awarii zwykle jest realizowana pomocy dysków HDD lub SSD, a ich opóźnienie i ograniczona ilość operacji na sekundę (IOPS) ograniczają przepustowość systemów korzystających z FullSS. Systemy mPaxos i mPaxosSM korzystają z pamięci trwałej i zamiast, jak FullSS, duplikować wybrane dane w pamięci i na dysku, trzymają wybrane dane w nieulotnej pamięci operacyjnej. W rozprawie uszczegółowiono jakie dane należy trzymać w pamięci trwałej dla zapewnienia poprawności i minimalizowania czasu odtwarzania. System mPaxos trzyma na bieżąco wszystkie kluczowe dane algorytmu Paxos w pamięci trwałej, natomiast dane części odpowiedzialnej za zreplikowanie maszyny stanowej i obsługę klientów są przechowywane zarówno w pamięci trwałej (w wersji spójnej z ostatnią migawką maszyny stanowej) i pamięci ulotnej (w wersji bieżącej). Dzięki takiemu podziałowi możliwe jest odtworzenie stanu na podstawie zawartości pamięci trwałej bez zmian w replikowanej maszynie stanowej. System mPaxosSM zakłada zmianę interfejsu między frameworkiem i maszyną stanową. W mPaxosSM maszyna stanowa musi utrwalać efekt wykonania każdej operacji w pamięci trwałej przed zwróceniem odpowiedzi do frameworku, natomiast z interfejsu usunięto całkowicie tworzenie migawek i od maszyny stanowej wymaga się tylko wsparcia chwilowego wstrzymania pracy na czas dostępu frameworku do obszaru pamięci trwałej maszyny stanowej. W wyniku takich zmian system mPaxosSM nie potrzebuje wykonywać i utrwalać regularnych migawek stanu, a w porównaniu do systemu mPaxos przechowuje również dane części odpowiedzialnej za replikację na bieżąco w pamięci trwałej. Pozwala to na dalsze skrócenie czasu odtwarzania, a brak konieczności regularnego wykonywania migawek pozwala poprawić sprawność maszyny stanowej, która cały czas może przetwarzać komendy bez poświęcania zasobów na wykonywanie migawek.

Druga grupa metod wspierania odtwarzania uległych awarii replik zakłada że w każdej chwili większość replik jest dostępna, a co za tym idzie dane które uległa awarii replika musi odtworzyć mogą być uzyskane przez odpytanie działających replik. Z jednej strony takie założenie ogranicza zastosowanie omawianych algorytmów. Choć w przypadku jego niespełnienia zachowane są warunki poprawności, dalsza praca systemu nie jest możliwa, a dane mogą zostać utracone. Z drugiej strony systemy które w taki sposób wspierają odtwarzanie są nie gorsze wydajnościowo niż systemy bez wsparcia dla odtwarzania, ponadto w praktycznych zastosowaniach systemów zreplikowanej maszyny stanowej dąży się do zagwarantowania takiego założenia jako koniecznego dla nieprzerwanej dostępności systemu. Rozprawa omawia dwa algorytmy odtwarzania stanu zakładające dostępność większości replik: ViewSS i EpochSS. Oba algorytmy w trakcie odtwarzania muszą dowiedzieć się od innych replik jaki stan potrzebują przywrócić oraz muszą zapewnić że wiadomości wysłane przed awarią przez odtwarzającą się replikę po otrzymaniu przez inne repliki po zakończeniu procedury odtwarzania nie naruszą poprawności działania systemu. W obu algorytmach odtwarzająca się replika R_r odczytuje numer utrzymywany w pamięci nietraconej w razie awarii (w ViewSS jest to najwyższy znany numer lidera, w EpochSS liczbę odtworzeń repliki) i wysyła go do pozostałych replik. Te odpowiadają najwyższym znanym numerem instancji i numerem lidera. Po zebraniu odpowiedzi od większości replik, R_r wykonuje procedurę nadganiania do osiągnięcia wskazanego przez inne repliki stanu, z której końcem kończy odtwarzanie. Dla radzenia sobie z wiadomościami wysłanymi przez R_r przed awarią we ViewSS pozostałe repliki muszą zmienić lidera przed wysłaniem odpowiedzi do odtwarzającego się R_r (o ile taka zmiana nie nastąpiła od awarii R_r). W tym samym celu w EpochSS pozostałe repliki zapamiętują liczbę odtworzeń R_r i w fazie wyboru lidera porównują tą liczbę z wartością zawartą w komunikatach tej fazy (które w EpochSS jako jedyne muszą być zabezpieczone przed wspomnianym problemem).

Dla każdego sposobu wspierania odtwarzania wyszczególnione są dodatkowe czynności które replika musi wykonywać w trakcie rutynowej pracy, wraz z omówieniem ich wpływu na wydajność. Wszystkie metody są porównane ze sobą przez podzielenie odtwarzania repliki na kolejne etapy i omówienie jakie działania która metoda w danym etapie wykonuje. Następnie przedstawione są brzegowe dla wydajności każdej metody odtwarzania sytuacje i wskazane są możliwe wąskie gardła ograniczające wydajność procesu odtwarzania.

Proponowane metody, uznane za referencyjne algorytm FullSS oraz podejście niewspierające odtwarzania zostały zaimplementowane w oprogramowaniu JPaxos. Rozprawa przedstawia zarys istotnych kwestii implementacyjnych tych systemów i przedstawia użyte do eksperymentów warianty tych systemów. W wyniku przebiegu badań nad tematem i dostępności sprzętu ocena eksperymentalna została podzielona na dwa etapy: w pierwszym porównano system niewspierający odtwarzania, FullSS, ViewSS i EpochSS, w drugim porównano FullSS, EpochSS, mPaxos i mPaxosSM. Każdy etap bada wpływ wybranej metody wsparcia odtwarzania stanu na przepustowość systemu w rutynowej pracy, wpływ awarii i odtwarzania na wydajność systemu, oraz czas trwania procesu odtwarzania repliki, nadganiania stanu repliki i wykonania nadgonionych komend na maszynie stanowej.

W pierwszym etapie porównanie wydajności systemu niewspierającego odtwarzania i opartych o ViewSS i EpochSS potwierdziło utrzymanie pełnej przepustowości tych systemów. Przepustowość FullSS spada w różnym stopniu w zależności od charakteru obciążenia, będąc niższa nawet o ⅓ od wydajności porównywanych systemów dla małych żądań. Czas odtwarzania systemu opartego o FullSS okazał się mniejszy niż pozostałych dwóch, co pozwala szybciej przywrócić odporność systemu na kolejne awarie replik. Jednak po uwzględnieniu konieczności nadgonienia stanu

EpochSS i ViewSS szybciej doprowadza replikę do pełnej sprawności. Sam proces odtwarzania stanu okazał się mieć zauważalny, choć nieznaczny wpływ na wydajność całego systemu, z pewnymi wyjątkami. W przypadku użycia magnetycznego dysku twardego (zamiast symulowanego wysokowydajnego dysku SSD) wymuszona przy odtworzeniu repliki przez algorytm ViewSS zmiana lidera doprowadziła do istotnego chwilowego spadku przepustowości.

Testy z pierwszego etapu używały maszyny stanowej mającej minimalny wpływ na wydajność systemu. Drugi etap testów został przeprowadzony dla replik powielających magazyn klucz-wartość i na nowszym sprzęcie, wyposażonym w pamięć trwałą Intel Optane DCPMM. Dla miarodajnego porównania algorytmy FullSS i EpochSS również używały pamięci trwałej jako pamięć nietraconą przy awarii. W rutynowej pracy, po uwzględnieniu odpowiednich poprawek, przepustowość systemów kształtuje się w kolejności od najszybszego: EpochSS, mPaxos, FullSS, mPaxosSM. Pierwsze trzy systemy są w stanie wykorzystać dostępną przepustowość najbardziej obciążonego łącza, a więc ich dalsze skalowanie ogranicza sieć. System mPaxosSM okazał się zauważalnie wolniejszy, z tego powodu został on poddany profilowaniu. W jego wyniku można określić że przepustowość mPaxosSM jest ograniczana przez przepustowość zapisu do pamięci trwałej, oraz że liczby ani rozmiaru zapisów nie można zredukować. Poza szczegółową analizą wydajności systemów dla wybranego rozmiaru żądań przeprowadzono też ocenę pracy dla szerokiego zakresu rozmiarów żądań, jak również różnych ustawień tworzenia migawek. To pozwoliło na określenie dla jakich zastosowań koszt tworzenia migawek przewyższa spadek wydajności cechujący mPaxosSM, a więc na określenie dla jakich zastosowań mPaxosSM osiąga większą przepustowość od pozostałych systemów.

Analiza zachowania systemu w trakcie awarii i odtwarzania repliki dla systemów FullSS, EpochSS i mPaxos potwierdziła wyniki podobnej analizy wykonanej w pierwszym etapie eksperymentów, ponadto pokazała że mPaxos odtwarza się w czasie znacznie krótszym niż pozostałe, jednak proces późniejszego nadganiania przebiega wolniej niż dla FullSS. Odtwarzanie repliki w mPaxosSM nie jest konieczne, bo wszystkie krytyczne dane przetwarzana znajdują się w trwałej pamięci operacyjnej. Konieczne jest, jak w każdym przypadku, nadgonienie stanu, które w mPaxosSM wymaga wykonania migawki na żądanie i w systemie z trzema replikami skutkuje istotnym spadkiem wydajności przez okres tworzenia takiej migawki. Dla pięciu replik spadek wydajności jest dużo niższy, bo poza repliką wykonującą migawkę i nadganiającą stan w systemie pracują trzy repliki (a nie jedna) na które rozkłada się obsługa klientów. Z racji wspominanych limitów wydajnościowych pamięci trwałej przepustowość systemu mPaxosSM jest uzależniona od przepustowości wykonywania komend, co w procesie odtwarzania przełożyło się na znacznie wydłużony czas wykonania nadgonionych komend.

Rozprawę zamykają wnioski z badań pozwalające wybrać właściwy dla danej aplikacji sposób wspierania odtwarzania replik uległych awarii, pokazujące ograniczenia każdego podejścia i streszczające istotne wyniki oceny eksperymentalnej systemów.