Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Study thesis Nr. 2412

Design and Implementation of a Coordination Service for Distributed Applications (In-memory Paxos)

Christian Mayer

Course of Study:	Computer Science
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	Dr. rer. nat. Frank Dürr
Commenced:	2012-11-12
Completed:	2013-05-14
CR-Classification:	C.2.4, C.4, H.3.4

Abstract

Coordination of different, independent processes is a very important aspect in the area of distributed systems. In order to coordinate each other, participants of a distributed system often have to agree on some common knowledge such as locking of a shared resource. The general problem how to reach agreement on some value is also known as **consensus problem**. In many practical systems, the consensus problem is outsourced to a distributed system consisting of multiple servers to increase its availability. Each server can be contacted by clients that intend to reach consensus about a specific value. Examples are Google's locking service Chubby or Yahoo's distributed file system Zookeeper. The standard Paxos algorithm solves this problem in an environment where nodes may recover after a crash and messages can have infinite delay. However, a system based on the classical Paxos algorithm makes use of expensive stable storage operations to guarantee that a crashed and recovered Paxos server is still able to participate in the protocol. Studies have shown that these disk costs are the bottleneck of the whole system. In this work a performance-oriented version of Paxos will be investigated that still solves the consensus problem, but trades availability of the consensus system against performance by not using stable storage operations. Without careful design this can be problematic, because a Paxos server that has lost its memory can be dangerous for the success of the protocol. This can be solved by not allowing a crashed and recovered Paxos server to participate in the protocol anymore. Instead, a recovered server rejoins the protocol with a new id, so that no active processes assume anything about the recovered process. In order to join the group of active processes, a majority of servers has to be active and agree on this. Our evaluations show, that a coordination system based on the in-memory Paxos approach has a very short response time of only one millisecond and high throughput up to 18000 write requests per second.

Contents

List of Figures

1 Introduction

The consensus problem is a basic problem in the area of distributed systems. Assume a set of processes P building the distributed system and a set V containing possible consensus values. Some processes $p \in P$ may propose values $v \in V$ to other processes in P. A protocol solving the consensus problem will reach an agreement about exactly one of the proposed values, even if there are multiple values proposed. Agreement means, that two processes p, q can never assume different values $v_p, v_q \in V$ and $v_p \neq v_q$ to be consensus values.

Many common problems in distributed computing can be solved on top of the consensus problem for instance leader election, locking or synchronization. Consider the leader election example. If we could not use the consensus function, we would have to implement a whole new protocol to solve this. But if we already are able to reach consensus, we will just propose some process as leader and wait until the consensus algorithm acknowledges agreement about that.

A consensus algorithm can be seen as a very important tool for building and running distributed systems. Common knowledge across the whole distributed system enables processes to work together by coordinating and synchronizing each other. Practical examples for coordination services are:

Locking services manage access to shared resources and can be implemented easily and fault-tolerant using a coordination service. Entities that need access to the resource have to get a lock from the distributed coordination service. Google's Chubby Locking Service [1] and Yahoo's Zookeeper [2] are two famous examples of such systems.

Cloud storage provider usually replicate their data to guarantee higher availability, reliability and performance. The replicas have to run a synchronization algorithm to maintain consistency of data. This can be done via the state machine approach [3]. Each replica stores a list of $(slot, command)$-pairs (e.g. $(0, write), (1, write), (2, read), ...$) that were applied one after another to a local database. A consensus algorithm can be used to reach consensus about each command c, issued in slot (=position) i. A replica will be only allowed to apply a command c in slot i, if it is sure about consensus and hence about the fact, that each other replica either also applies c in slot i or no command at all. If all commands are deterministic, each replica will have the same database state after a certain number of commands, because each replica will issue the same list of commands to its local database.

Software defined networking is an emerging networking paradigm that separates network switches from routing tables (see for example [5]). The data plane (packet forwarding) is handled by switches that are implemented in very fast hardware. The control plane determines the routing tables of the switches. An administrator or a complex, perhaps distributed

application build the controller that computes dynamically optimal routing tables for each switch. Switches communicate with the controller by any network protocol. Such a controller has to be very fast and failure-tolerant, hence it could be implemented as a distributed system. A number of servers handle the computation of optimal routing tables and issue those to the switches. The servers have to coordinate each other, so that no two servers overwrite each others routing table updates. This coordination can be done by locking the resources (switches) so that servers can not overwrite other servers entries.

In big systems or companies (for example Google and Yahoo) the consensus problem is often encapsulated by offering *consensus as a service*. As a consensus service is often crucial for client applications, it is not sufficient to only use one consensus server. In order to reach higher availability, multiple servers (often five) have to be responsible for the agreement about a value. Clients, that can be huge applications on their own (e.g. Google BigTable is a client of Google Chubby), may send at least two kinds of requests to the consensus service: read and write requests.

In this work, we will concentrate on a specific consensus protocol, the Paxos protocol, and develop a variant of the Paxos protocol, called in-memory Paxos. After that there will be some performance evaluation and correctness investigation of the in-memory Paxos protocol.

The Paxos protocol solves consensus in an environment where processes may fail and recover from crashes and where messages can get lost. It was first presented by Leslie Lamport [3], at least in the terminology that will be introduced next. We already mentioned that some processes may propose values, they intend to reach consensus about. These processes are called **proposers**. Processes that receive proposals and decide whether a proposal should be accepted or rejected are called **acceptors**. If a majority of acceptors accepts a value $v \in V$, this value will be called the *chosen value*. Some processes, called **learners**, are interested in the chosen value, otherwise the consensus protocol would be pointless. They could get notified by a proposer about the chosen value v. Together, the three sets build the distributed system for consensus, hence $P = proposers \cup acceptors \cup learners$.

The terms can be illustrated by the following example: A few politicians (proposers) want to become president. Every politician proposes itself as president to a set of voters (acceptors). If a politician falls out and there are still politicians trying to become president, everything will be still alright. If a majority of voters agrees on one president, there will be network wide consensus about who is the new president. This fact will be only important, if anybody reads newspapers and learns about this (learners). Assume newspapers only print the truth (no byzantine failures), each learner will either learn, that the one elected politician is president or nothing about the new president at all. It is not possible that two learners learn about different presidents, because no two presidents can have collected a majority set of votes.

If a client wants to reach consensus via the Paxos based consensus service, it will submit a write request. The consensus servers internally try to reach consensus about the client's value in a specific slot. If there is consensus in slot i, the client will be notified about success. In this situation, there may never be consensus about another value in slot i.

It is also possible, that clients send read requests to the consensus service to determine which consensus values were reached in past slots. The consensus service responses to these read

requests with the corresponding consensus values. If a consensus server gets a read request (some client wants to know the consensus value in slot i), it will often be able to answer it locally, because if it knows a consensus value of slot i, it will be sure, that this was actually the consensus value in slot i. Therefore, read requests can be answered very fast in general.

Less performant are write requests. A client want to reach consensus about a value v (e.g. to get a lock for a certain resource). As a consensus server need to perform the Paxos protocol before it responds to the client about the success status of the request, all consensus server are included to determine the client answer.

During the classical Paxos protocol, it is often necessary for participants to save protocol specific information on stable storage so that they can still participate in the protocol after they have crashed and recovered. Practical experiences [4] have shown, that the time needed to write this information to disk, can dominate the whole execution time, when the consensus servers are in close network proximity.

If requests to the consensus service are mainly read requests, these costs will be somewhat compensated by the fact, that they do not have to be paid very frequently. This assumption is not valid for each system. We have already mentioned the need of a consensus service in software defined networking environments. Servers of the distributed controller have to coordinate their access to the routing tables of switches. Most of the requests to the consensus service will be write requests with the purpose to lock resources. It is not that interesting for the servers of the distributed controller to read who exactly possessed a lock in the past. This concrete scenario actually motivated this work.

If there is an environment present where writes dominate the load for the consensus service and performance is very important, how can the performance be improved by guaranteeing that the protocol is still correct? In this work, we will try to eliminate the bottleneck of synchronous writes to stable storage. After an investigation of the original Paxos protocol it will become clear that writing to stable storage guarantees, that the protocol will still work right, if processes crash and recover.

This is not the case for an in-memory server. After it recovers, all state information before the crash is lost. Therefore it cannot safely participate in the Paxos algorithm any more, because it doesn't remember promises he has given to other servers. But if processes do not recover at all, writes to stable storage will not be necessary. However, it should be allowed that processes recover to ensure high availability and fault-tolerance of the service. A few options to make this work will be named in chapter 5. One solution is that a crashed and recovered server rejoins the protocol with a new id, so that no old promise can be violated. The recovered process will only be allowed to participate if it learns that there was consensus about its rejoin request.

The rest of this work is structured as follows:

Chapter 2 introduces the **system model** that is assumed and the **problem specification** of the consensus problem. A generic **interface** to the coordination service will be given to complete our assumptions about the system.

Chapter 3 presents the **original Paxos** algorithm that solves the consensus problem and the in-memory version based upon. The theoretical algorithm to solve consensus for one value is the **single-decree Paxos algorithm**, but in practice consensus has to be solved in consecutive slots. This is addressed by the **multi-decree Paxos algorithm** that is more efficient than just using multiple instances of the single-decree Paxos. As we are interested in better performance, we will investigate the concrete **stable storage** operations used in Paxos.

Chapter 4 is about **possible alternatives** that do not make use of stable storage operations, but still guarantee consistency of the servers local databases. An approach based on **group management**, the ability to change the group of active participants, will be chosen. The necessary changes to the multi-decree Paxos protocol will be given in the **protocol description**.

Chapter 5 shows that the **safety** properties from the original Paxos protocol are still valid and consistency is guaranteed. The **liveness** property has to be changed to be still valid, which will be shown at the end of this chapter.

Chapter 6 gives practical **performance evaluations** of the in-memory consensus system and **chapter 7** represents a summary of this work.

2 Background and Related Work

The Paxos algorithm was proposed by Leslie Lamport in 1998 to solve consensus in an asynchronous environment. Many terms used in this thesis like *proposer*, *acceptor* and *learner* are introduced by Lamport. A reader, who is familiar with Lamport's *The Part-Time Parliament* [3] or *Paxos Made Simple* [6] will be also familiar with the first part of this thesis, but it is not necessary to be acquainted with his work as we will introduce the Paxos algorithm in detail.

In the Paxos algorithm a number of rounds were iterated, where each round consists of two phases: the preparing and the proposing phase. The proposer starts the preparing phase by choosing a proposer number n and sending prepare messages with its proposer number to a majority of acceptors. If an acceptor never received messages from any higher numbered proposer, it would acknowledge the prepare request with a promise not to accept messages from lower numbered proposers. If a proposer received promises from a majority of acceptors, it would see itself as coordinator, who may propose values. Now, the coordinator is allowed to move on to the proposing phase by proposing a value to a majority. As it is possible that another lower numbered proposer has already proposed some value that was accepted by a majority of acceptors, the coordinator proposes either the value that was proposed by the highest numbered coordinator or an arbitrary value, if none was proposed before. An acceptor will accept a proposal, if this doesn't violate any promise it has given in the meantime. If a majority of acceptors accepts a proposal, the value will be chosen and there will never be any other value chosen.

Figure 2.1 shows the protocol graphically. An acceptor has to store its answer on stable storage before the message is actually sent. There were different ideas to optimize the protocol. Some approaches like the multi-decree Paxos or Fast Paxos [7] reduce the number of message delays that is three in the classical Paxos (one extra round to inform the learners that there was consensus). Another approach is shown with Ring Paxos [8], where the number of concrete messages, that has to be sent, is reduced by using ip-multicast. The approach to be presented in this thesis is shortly proposed in *Overcoming CAP with Consistent Soft-State Replication* from Birman, Freedman, Qi Huang and Dowell [9] but was not investigated closer there. The idea is to minimize the latency of one instance of a Paxos algorithm by not using stable storage operations and to solve problems like inconsistencies that would arise, if proposers simply participated after a crash. As stable storage operations are very expensive and the network delay is usually very short, these costs often dominate the latency as described in *Paxos made live: an engineering perspective* [4].

As the goal of this work is to develop and implement a fast coordination service, there are other more practical oriented studies that influenced the design as well as the implementation of the

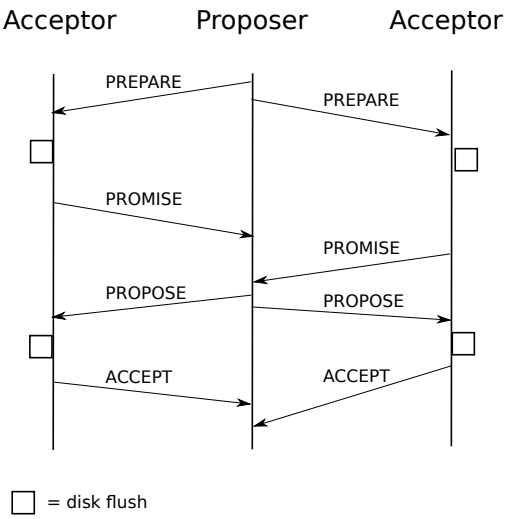

Figure 2.1: Illustration of the original Paxos protocol with access to stable storage

in-memory approach. So it has become clear that group management functionality is needed while reading *The Chubby lock service for loosely-coupled distributed systems* from M.Burrow [1] and *Paxos made live: an engineering perspective* [4]. To compare our system with the capabilities of other systems, we used the widely-known Zookeeper coordination service from Yahoo's researchers [2] and the Paxos based consensus service OpenReplica [10].

This thesis was first motivated by the need for a fast locking service for the distributed controller of a software-defined network (see [5] for more information). Many of the concepts and ideas however are not constrained to this area and can be adapt everywhere, where a fast coordination system is needed and one is interested in trading availability against performance.

3 System Model and Problem Description

In this chapter, system components and assumptions about the distributed system will be named, followed by a formal specification of the problem to be solved.

3.1 System Model

We will use Tanenbaums [11] definition of a distributed system:

> "[In] a distributed system, a collection of independent computers appears to its users as a single coherent system."

In this sense, the distributed consensus system is implemented by a fixed set of servers that communicate with each other to achieve consensus. To ensure higher availability and prevent a single point of failure, multiple servers should run the system. Clients can use the distributed system through a predefined interface to hide the fact that it is distributed(see Figure 3.1). Two important operations must be offered by the interface: read a consensus value and write a consensus value. If a client wants to reach consensus about a value, it will contact a server of the system and will hope that the system reaches consensus about this value. If there is consensus about a specific value, clients will be able to read this value, respectively.

As we have already mentioned, the system is presented to the clients as one single service. Internally, each server process implements at least one of the following three roles, but it is also possible that a process implements all of them:

- **Proposer:** may propose values to acceptors that were issued by clients.
- **Acceptor:** decides, whether to accept or reject a value, communicates with proposers
- **Learner:** if there is consensus about a value, the learners will be informed.

The servers internally run the consensus protocol to iteratively reach consensus about values that were issued by clients.

The system is assumed to be asynchronous, there are no bounds for process execution, message delivery and clock drift. Processes may crash and restart, channels can omit and duplicate messages. There are no byzantine failures.

In order to keep it simple, we assume that the set of servers is fixed. In practice, the system should be able to add and remove servers to be more flexible. However, as we will see, we can easily implement group management (a mechanism, so that all servers agree on the group

building the consensus service and so that servers can join and leave this group) on top of the paxos protocol.

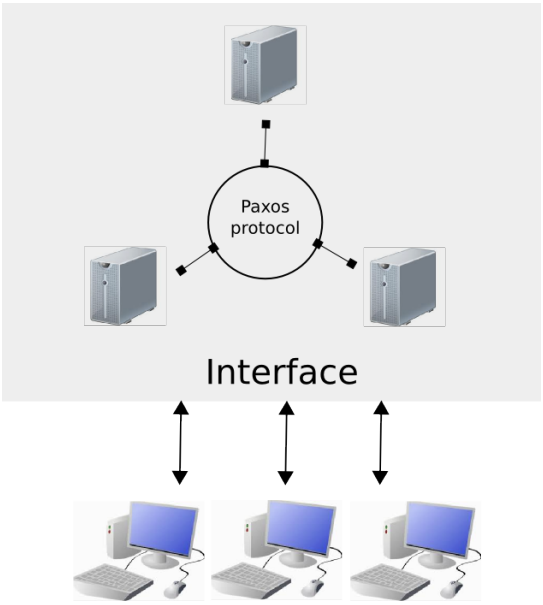

Figure 3.1: Consensus servers offer "consensus as a service" to clients

Each process implements the following interface (see also fig.3.2), so that clients can use the consensus service in an easy manner. A client intending to communicate with the abstract "consensus service" contacts one arbitrary server that is part of the consensus service. Therefore, a list with a subset of the consensus servers must be published or well known. The contacted server offers the concrete interface to the client and sees itself as broker between the clients intentions and the consensus service. This is a minimal interface so that a consensus service makes sense at all. It can be extended (and will be) by other services like group management.

void $write(i, v)$ If a client wants consensus about value v in slot i, it will call this method. If the call is successful it won't be sure yet that there actually will have been consensus. It just means that the consensus service is notified about the clients intention to reach consensus in slot i. There is no return value, see below for an explanation.

value $read(i)$ Returns the consensus value in slot i or a default value if no consensus value is known.

The client could be notified about success or failure of his request to get consensus about a value, but as the interface should be minimal, this is omitted here. To get the same behaviour, the client could call $write(i, v)$ followed by $read(i)$.

An example, how the consensus system and its interface can be used, would be the following. Consider the simple scenario of two clients, which intend to share their work (to update and read resources) and therefore have to agree on a single task list $T = t_1, t_2, t_3, ..., t_n$, so that they can split the tasks. A task $t_i, i \in \{1, ..., n\}$ could be an update of a resource or a status request (read). Each client can publish new tasks on the consensus service. The order, in

which the tasks were executed, matters, hence it is important that both clients agree on the same sequence of tasks. Otherwise it would be possible that a client assumes, a resource is already updated, while it is actually not, because the other client intends to perform the update on a later point in time. If a client wants to issue a new task “update A” in slot 5, it will call *write*(5,“update A”) to one specific server out of the set of servers implementing the Paxos protocol. If the other client wants to handle the task t_5 in slot 5, he will ask the system about the consensus value in slot 5 by issuing *read*(5). The system returns “update A” and consensus via the consensus system is reached.

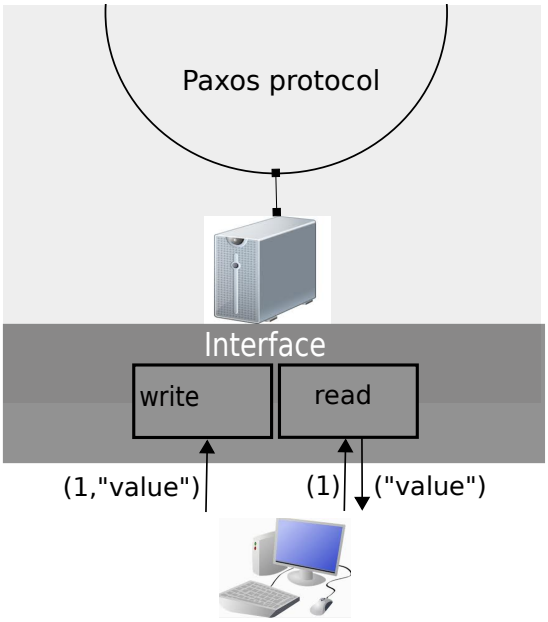

Figure 3.2: Client communication with interface of consensus service

3.2 Problem Statement

There are n proposers $p_1, ..., p_n$. A few proposers actually propose values to a set of m acceptors $a_1, ..., a_m$. Let the proposed values build the set V. The goal of a consensus protocol is, that enough acceptors accept one value $v \in V$ and will always accept v. In more detail, a consensus algorithm ensures the safety and liveness properties[6].

Safety1. Only a value that has been proposed may be chosen (hence it is impossible that all acceptors simply choose "1")

Safety2. Only a single value is chosen, so that different learners cannot learn different values.

Safety3. A learner never learns that a value has been chosen unless it actually has been, hence the learner can be sure about consensus.

Liveness1. Some of the proposed values is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

A simple solution to this problem would be to maintain a fixed coordinator, which chooses one of the proposals and is responsible to get the value accepted by all acceptors. This approach is used in the two phase commit protocol. The problem is, that the coordinator may fail, which, on the one hand, would cause the protocol to block. On the other hand, such a single point of failure is a severe security problem and decreases the availability of the system. Thus, multiple servers were used to implement a distributed system that solves consensus. This leads to new challenges: the servers implementing the system have to synchronize and communicate in the presence of message losses in an asynchronous system. A protocol solving this will be introduced in the next chapter: the Paxos protocol, that guarantees correct behaviour of the system (safety properties) in an asynchronous environment.

The Paxos protocol uses stable storage to handle process failures: if a process fails, it will be still able to remember essential assurances it has given before it failed. In contrast to that, a server using only volatile storage, won't be able to recover promptly after a crash, because it also has forgotten all important assurances.

But using stable storage is slow and therefore the throughput of a consensus system decreases. The goal of this work is to increase performance by weakening the availability of the system. It is often sufficient to guarantee, that the system survives the simultaneous crash of a minority of servers. This is assumed to happen very rarely, if enough servers are distributed over the whole network or even more efficient the whole datacenter to machines with different power supplies. If the whole datacenter failed, the system would have to be rebooted anyway, because many client applications have crashed, too.

A possible scenario in the original Paxos protocol would be the simultaneous crash of all acceptors exactly after they have accepted a certain value v. If the acceptors rejoin the protocol, they will need to know that they have already accepted v. Otherwise a new coordinator may learn, that no value has been accepted yet and propose a new value v' with $v' \neq v$, which violates the safety property.

The original Paxos protocol solves this by make use of stable storage. So how can safety and liveness be guaranteed without using stable storage?

Assume that no acceptor make use of stable storage operations. If an acceptor can not remember its last accepted value (for example after a crash), but still participates in the protocol, safety won't be ensured any more as different values will be allowed to be chosen. But if it simply does not participate in the protocol after a crash, liveness can not be ensured as a value could have been chosen, but learners might not be able to learn it.

To handle this, a new liveness property **Liveness2** is introduced, which is not as strong as the classical one, but sufficient for many applications. This is a typical trade-off between performance and reliability.

> **Liveness2.** As long as not more than $\frac{n}{2}$ out of n servers fail simultaneously, some of the proposed values is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

In the traditional protocol, it will be no problem, if all processes crash simultaneously. Eventually, each process will recover from the crash and a majority of processes will be up for a sufficient long time, so that consensus can be reached.

We give up this nice property to be able not to use stable storage operations. A crash of a majority of processes will be serious in our approach as a process cannot rely on its knowledge on stable storage: it has forgotten the promises it has given and may not participate in the protocol. If all processes crash, there will be no knowledge in the network and no recovery from this dead-lock at all.

So, can we live with this new property? This can not be answered in general, but in many scenarios it is sufficient. The probability of a majority crash can be minimized by using enough servers and spreading these over different data centers. By the way, safety will always be guaranteed even if a majority of processes crashes.

We will introduce now the original Paxos protocol in its two main variants and investigate the expensive stable storage operations in more detail.

4 Original Paxos

The Paxos protocol can be used to reach consensus in a distributed system in the presence of node and link failures and unbounded message delay. There are many variants of the Paxos protocol (see chapter 2). The protocol family can be divided into **single-decree** and **multi-decree** Paxos protocols, that solve the consensus problem for a single value or a sequence of values. In this chapter, we will cover the most basic variants, introduced by Lamport [3]. After proposing the single-decree Paxos protocol, Lamport has constructed the multi-decree protocol as an efficient variant of multiple instances of the single-decree protocol. If there has to be consensus in slots $1, 2, ..., n$ (multi-decree), one can simply perform the single-decree Paxos protocol, one protocol for each slot, one slot after another. The slot number $i \in 1, ..., n$ (also called instance number in this work) can be sent with each protocol message so that the participants know the slot, where the received message belongs to. This procedure can be optimized, but is basically the solution of the more complex consensus problem. After introducing both variants, the chapter ends with an investigation of the expensive stable storage operations, that were needed in the original Paxos implementation.

4.1 Single-Decree Paxos

To reach consensus about a single value v, the single-decree Paxos algorithm iterate a number of rounds, each round can be divided into three phases [4]

1. Elect a proposer p to be coordinator, so that the protocol won't be forced to block, if a coordinator fails in a round.
2. The coordinator p decides on a value v, it intends to reach consensus about, and proposes it to a majority of acceptors. Acceptors may acknowledge or reject the value.
3. When a majority of acceptors have acknowledged the proposed value, the coordinator broadcasts commit messages about consensus.

We will start with an informal explanation of the protocol and end with a more formal description.

4.1.1 Informal Protocol Description

Assume some processes $p_1, p_2, ..., p_n$ participate on the Paxos protocol. Each process p_i plays at least one role in the protocol: It may propose values to acceptors as *proposer*, receive proposals as *acceptor* or learn about chosen values as *learner*.

There can be multiple proposers, but ideally only one proposer p sees itself as coordinator. To become coordinator, a proposer has to maintain a unique coordinator number. Only the proposer with the highest coordinator number can get coordinator. To ensure this, the coordinator candidate p first collects promises from a majority of acceptors not to accept proposals from lower-numbered proposers, so that it is ensured, that a majority of acceptors see also p as the current coordinator. Acceptors may accept multiple proposers, so that the system is failure tolerant, although processes may crash. It is possible, that p sees itself as coordinator (because it has collected a majority of promises), but in the meantime another process q with higher coordinator number has collected promises from a majority of processes. In this scenario, p would propose values to the acceptors, but could never collect a majority of accepts because a majority of acceptors has promised coordinator q not to accept any lower-numbered proposals. So it is sure, that only one proposer can get a majority of accepts for its proposal.

There is a problem with this approach: Consider the following scenario: A proposer q has collected a majority of promises, so that it sees itself as coordinator, that may propose values. Hence, the coordinator q proposes a value v to a majority of acceptors. The majority of acceptors accepts the value v. This value is chosen (=majority accepted) now and because of the safety criteria, that only a single value is chosen (**Safety2**), every later chosen value v' must already be equal to v. Because of the possibility that messages could have infinitely delay, the accepts messages could be in transit for a very long time. In the meantime another proposer p with a higher coordinator number as q tries to become new coordinator. It collects a majority of promises and propose a different value $v' \neq v$. Acceptors must be able to accept different values (see below why). Coordinator p collects a majority of accepts for value v' and a learner l learns about the chosen value v'. Now, the forgotten accept messages for value v appear at the scene and l learns also about v as chosen value. This violates **Safety2**.

Why must acceptors be able to accept different values? If an acceptor could only accept one value, the above problem wouldn't appear. If a value v was chosen, no other coordinator could make $v' \neq v$ chosen, because no majority of acceptors would accept v'. Anyway, this approach is not practicable, as some of the proposed values must be eventually chosen (**Liveness1**). So that this criteria is not violated, an acceptor could never accept a value unless it is sure, that this value is also accepted by a majority of acceptors, otherwise it would be possible that each acceptor accepts another value and no majority could ever be reached, although some values were proposed. But if no acceptor could accept a value without global knowledge, how could a value actually have been accepted? It is better to allow acceptors to accept different values and make the coordinator responsible only to propose values that were already accepted by a majority of acceptors. If no such values exist, the coordinator can propose any value.

To learn about already accepted values, the coordinator also collects knowledge about all previously accepted values. The acceptors simply append this information to their promises. With the collected global knowledge, the coordinator knows about already accepted values, proposed by the previous coordinator (the one with the highest coordinator number smaller than its own coordinator number). If the last coordinator got a value v through, a majority of acceptors would have accepted this value. The current coordinator has the information about

previously accepted proposals from a majority of acceptors and therefore it knows also about v (each two majorities overlap).

The coordinator may crash at arbitrary time and so the acceptors must be able to select a new leader. An acceptor will be allowed to choose another coordinator by sending a promise, if it has a higher coordinator number and ignore messages from past coordinators.

If an acceptor receives a proposed value from his current coordinator, it will accept, otherwise it will reject the value. This ensures, that if a coordinator receives a majority of accepts, every later (higher-numbered) coordinator will learn about this majority accepted value. Because of the promises of the acceptors not to accept lower-numbered coordinators, there can be no lower-numbered coordinators getting any value through. This means all coordinators push the same or no value through the distributed system. Next, the algorithm will be presented in more detail.

4.1.2 Formal Protocol Description

We will define the variables each participant has to maintain, the actions each participant may perform and when it is allowed to perform these actions.

Each participant needs to store some values persistently, i.e. on stable storage and some values non-persistent, i.e. in-memory. The memory type of the variable is indicated after its declaration. Although we mention the memory type now, we will examine the reason for this later. It should be kept in mind, that stable storage operations are expensive, while non-persistent storage operations are much cheaper (but of course also non-persistent).

- Proposer p:

  $coordNumber$ ***(stable)*** The coordinator number, which has to be globally unique (so that $owner(coordNumber) = p$ can be determined by any process) and which has to grow monotonically.

  $currVal$ ***(volatile)*** The current value to propose, when $status = proposing$.

  $currPromises$ ***(volatile)*** A set consisting of all promises, he already got for the current proposal with coordinator number $coordNumber$. If $p.status \neq preparing$, $currPromises$ has no meaning and should be the empty set.

  $status$ ***(volatile)*** Current phase in the protocol, $status \in \{idle, preparing, proposing\}$. If the proposer doesn't know in which phase he is (e.g. after a crash), he will initialize $status$ with $idle$.

  $currAccepts$ ***(volatile)*** A set consisting of all accepts, he already got for the current proposal $coordNumber$. If $p.status \neq proposing$, $currAccepts$ has no meaning and should be the empty set.

- Acceptor a:

  id ***(stable)*** Constant id of acceptor.

promiseNumber ***(stable)*** The highest promise number, it has given, initially $-\infty$.

lastAccepted ***(stable)*** The highest numbered accepted proposal (n, v), initially the coordinator number n is $-\infty$ and the proposal value v is *null*.

- Learner l:

  consensusValue ***(stable)*** Value there was consensus about, initially *null*.

Now we will introduce condition action pairs, so that each action will be allowed to be performed by a participant, if the condition is fulfilled. The conditions are above the actions. Each action is assumed to be atomic, which means, it is performed either completely or not at all. (e.g. if the action is interrupted by a crash).

Initiate new round:
Proposer p is free to initiate a new round n by proposing itself as coordinator.

Update $p.status = preparing$.

Update $p.coordNumber = next(p.coordNumber)$, where $x < next(x) \forall x$.

Update $p.currPromises = \emptyset$.

Send $PREPARE(n)$ request to a set of acceptors, where $n = p.coordNumber$. This actually encodes two requests to an acceptor:

Give promise never to accept a lower numbered proposal $PROPOSE(n', v')$, where $n' < n$ and a value v'.

Give highest-numbered accepted proposal smaller n.

Send promise:
Acceptor a receives $PREPARE(n)$ request, where $n > a.promiseNumber$.

Update $a.promiseNumber = n$.

Send $PROMISE(n, a.id, a.lastAccepted)$ to $owner(n)$.

Collect promises:
Proposer p receives $PROMISE(n, a, v')$, where $n = coordNumber$ and $p.status = preparing$.

$currPromises = currPromises \cup \{(n, a, v')\}$.

Propose value:
Proposer p, with $p.status = preparing$ got promises from a majority of acceptors.

(sees itself as current coordinator, that may propose values, because it knows the value v with $(n, a, v) \in currPromises$ and $n = max\{n' | (n', a', v') \in currPromises\}$ of the highest numbered accepted proposal in $currPromises$ and therefore also the last chosen value. If there was no value accepted, v can be chosen arbitrary by p. Without the promises, p could never be sure, that an acceptor will not accept a lower numbered proposal sometimes.)

Update $p.status = proposing$.

Update $currAccepts = \emptyset$.

Update $currVal = v$ (see above for a definition of v).

Coordinator p sends $PROPOSE(p.coordNumber, v)$ to some majority set of acceptors.

Accept proposal:
Acceptor a receives $PROPOSE(n, v)$, where $n = a.promiseNumber$.

Update $a.lastAccepted = (n, v)$.

Send $ACCEPT(n, a.id)$ to $owner(n)$.

Collect accepts:
Proposer p receives $ACCEPT(n, a)$, if $n = coordNumber$ and $p.status = proposing$.

Update $currAccepts = currAccepts \cup \{(n, a)\}$.

Send consensus:
Proposer p has got $ACCEPT$ messages from a majority of the acceptors and $p.status = proposing$.

Send $CONSENSUS(currVal)$ to the learners (that learn about the value).

4.2 Multi-Decree Paxos

In practice, there is often the need to reach consensus not only about one single value, but about a sequence of values. Imaging following scenario: There are many clients, that want to use a distributed database [1, 2, 4] by issuing commands to it (e.g. read, write, delete). Initially, each process of the distributed database has the same state (e.g. an empty local database). If each process maintains a log of a sequence of commands, that are issued to the distributed database, and there is consensus about the actual command sequence, each process can execute the commands locally and in sequence. After the execution of the sequence, all processes will have the same internal state, if the commands are deterministic. The distributed log of commands can be reached via Multi Paxos.

4.2.1 Informal Protocol Description

A simple solution to this problem is using multiple instances of the single-decree Paxos protocol [3, 6]: one instance for each consensus value. If there was consensus about a value in the i-th instance, this value would be the consensus value in the i-th position of the sequence.

There is one efficient optimization: The $PREPARE(n)$ and $PROMISE(n, v)$ messages, exchanged in the first phase of the protocol are not needed for each instance of the Paxos protocol. A proposer can be coordinator for many instances. If proposer p with coordinator number n wants to become coordinator, it will send a $PREPARE(n)$ message to a majority of acceptors. For each instance of the algorithm, it has to be sure about the accepted values, that were lower numbered than n. Because of that, each acceptor answers the $PREPARE(n)$ request with a promise, which is valid for each Paxos instance. If a proposer sees itself as coordinator for all instances, it can propose values for concrete instances (by sending $PROPOSE(i, n, v)$, i is the instance number, n is the coordinator/proposal number and v is the value of the highest numbered accepted proposal in instance i). If it learns, that some value has been already accepted in this instance, this value will be proposed, otherwise it will be free to propose own values (for example values, which clients have sent to it). Obviously, the first steps are independent of a concrete instance and can be performed just once, if a new coordinator-candidate intends to take over leadership.

Another easy and effective optimization is to prevent the execution of the protocol for those instances, where the results are already known by the coordinator. Instead it sends a $PREPARE(i, n)$, where the protocol is performed only for each instance greater than i.

4.2.2 Formal Protocol Description

We will give now a formalization of the multi-decree Paxos protocol.

- Proposer p:

  $coordNumber$ ***(stable)*** The coordinator number, which has to be globally unique (so that $owner(coordNumber) = p$ can be determined by any process) and which has to grow monotonically.

  $instance$ ***(volatile)*** It is often sufficient not to perform the algorithm for all instances, but for all instances, that have a greater or equal instance number than this variable.

  $currVal_i$ ***(volatile)*** The current value to propose in instance i, when $status = proposing$.

  $currPromises$ ***(volatile)*** A set consisting of all promises, it already got for the current proposal $coordNumber$. If $p.status \neq preparing$, $currPromises$ has no meaning and should be the empty set.

  $status$ ***(volatile)*** Current phase in the protocol $status \in \{idle, preparing, proposing\}$ If the proposer doesn't know in which phase it is (e.g. after a crash), it will initialize $status$ with $idle$.

$currAccepts_i$ ***(volatile)*** A set consisting of all accepts, the proposer p already got for the current proposal $PROPOSE(i, p.coordNumber, p.currVal_i)$ in an instance i. If $p.status \neq proposing$, $currAccepts$ has no meaning and should be the empty set.

- Acceptor a:

  id ***(stable)*** Constant id of acceptor.

  $promiseNumber$ ***(stable)*** The highest promise number, it has given, initially $-\infty$.

  $lastAccepted_i$ ***(stable)*** Highest numbered accepted proposal in instance i.

- Learner l:

  $consensusValue$ ***(stable)*** Set of $(instanceNumber, Value)$ pairs there was consensus about, initially $\emptyset$.

As in the formal description for the single-decree Paxos protocol, we will give now condition action pairs.

Initiate new round:
Proposer p is free to initiate a new round by proposing itself as coordinator for all instances greater than $p.instance$.

Update $p.status = preparing$.

Update $p.coordNumber = next(p.coordNumber)$, where $x < next(x) \forall x$.

Update $p.currPromises = \emptyset$.

Send prepare:
Proposer p may send a PREPARE-message to an acceptor, if $p.status = preparing$.

Send $PREPARE(p.instance, p.coordNumber)$ request to a set of acceptors, where $p.instance$ is the lowest instance, for which p wants to reach consensus and $p.coordNumber$ is the coordinator number. This actually encodes two requests to an acceptor:

Give promise never to accept a lower numbered proposal $PROPOSE(i, n', v)$, where the coordinator number n' is lower than n, v is any value and i is any instance number.

Give highest-numbered accepted proposal smaller n for each instance greater or equal than $p.instance$.

Receive prepare:
Acceptor a receives $PREPARE(i, n)$ request, where $n > a.promiseNumber$.

Update $a.promiseNumber = n$.

Send promise:
Acceptor a sends a promise to the current coordinator.

Send $PROMISE(a.promiseNumber, a.id, lastAccepted_{\geq i})$ to $owner(a.promiseNumber)$, where $lastAccepted_{\geq i}$ is the set $\{a.lastAccepted_{i'} | i' \geq i\}$.

Collect promises:
Proposer p with $p.status = preparing$ or $p.status = proposing$ receives $PROMISE(n, a, V)$, where $n = coordNumber$.

$currPromises = currPromises \cup \{(n, a, V)\}$.

Start proposing phase:
Proposer p with $p.status = preparing$ got promises from a majority of acceptors in instance i.

Update $p.status = proposing$.

Update $p.currAccepts_i = \emptyset \ \forall i \geq p.instance$.

Update $p.currVal_i$ as the value of the highest numbered accepted proposal in $p.currPromises$ for instance i.

Propose value:
Proposer p with $p.status = proposing$ wants to propose a value for instance $i \geq p.instance$.

Update $p.currVal_i$ as the value of the highest numbered accepted proposal in $p.currPromises$ for instance i.

Send $PROPOSE(i, p.coordNumber, p.currVal_i)$ to a majority of acceptors.

Accept proposal:
Acceptor a receives $PROPOSE(i, n, v)$, where $n \geq a.promiseNumber$.

Update $a.lastAccepted_i = (n, v)$.

Send $ACCEPT(i, n, a.id)$ to $owner(n)$.

Collect accepts:
Proposer p receives $ACCEPT(i, n, a)$, $n = p.coordNumber$ and $p.status = proposing$.

Update $currAccepts_i = currAccepts_i \cup \{(n, a)\}$.

Send consensus:
Proposer p has got $ACCEPT$ messages from a majority of the acceptors in any instance $i \geq instance$ and $p.status = proposing$.

Send $CONSENSUS(i, currVal_i)$ to the learners (that learn about the value).

Receive consensus:
Learner receiving $CONSENSUS(i, v)$ from proposer p.

Update $consensusValue = consensusValue \cup \{(i, v)\}$.

If each process implements all three roles (proposer, acceptor and learner), an acceptor receiving $PREPARE(i, n)$ can ask the proposer $owner(n)$ about all consensus values for instances $\leq i$, it doesn't know already. Furthermore the acceptor can inform the proposer about all consensus values for instances $\geq i$, it has already learned about.

The protocol can be extended by retransmitting messages after time-outs, so that the loss of a few messages cannot cause the stopping of the protocol.

We have introduced now the classical Paxos protocol. This protocol works very well and many practical systems are based on Paxos. However, Paxos make use of stable storage to allow crashed and recovered processes joining the protocol again without any complex bootstrapping and initialization. This is a reasonable decision, but there are environments, where performance is more important than this kind of failure tolerance. To understand the importance of this costs, we will end this chapter with a short examination of the stable storage operations.

4.3 Stable Storage In Paxos

The above protocol for multi-Paxos does many of the operations on stable storage. Practical implementations of the protocol have shown, that these costs can be the bottleneck of the whole protocol:

> "In a system where replicas are in close network proximity, disk flush time can dominate the overall latency of the implementation." [4]

Therefore, it may be a good idea to minimize these costs by using mainly volatile storage. However, this change affects the safety properties, because acceptors would lose the information about the promises they have given after a crash. Recovering from a crash would be impossible.

How problematic are these costs for the concrete multi-Paxos protocol? We will investigate the stable variables one after another.

1. Proposer p:

   coordNumber ***(stable)*** The coordinator number, which has to be globally unique and which has to grow monotonically.

   The variable changes in each round in phase 1, but as we have seen, phase 1 must not be executed in each instance. So if phase 1 is performed only from time to time, these costs will not be critical.

2. Acceptor a:

   id ***(stable)*** Constant id of acceptor.

   The variable does not change in the multi-decree protocol and can be loaded into memory once.

promiseNumber ***(stable)*** The highest promise number, acceptor a has given, initially $-\infty$.

The variable changes in each round in phase 1 and hence does not change frequently in the ideal case.

$lastAccepted_i$ ***(stable)*** Highest numbered accepted proposal in instance i.

This is the most problematic variable, because it changes in each instance of the multi-decree protocol. The coordinator even has to wait until the last acceptor of a majority has flushed the value to disk before it actually accepts it. Hence the lowest-bounded latency of each Paxos instance is the slowest disk flush time of an acceptor from a majority of acceptors.

3. Learner l:

consensusValue ***(stable)*** Set of $(instanceNumber, value)$ pairs there was consensus about, initially $\emptyset$.

The variable can be changed asynchronously and in parallel, because a majority-accepted value can be learnt for sure: each acceptor stores the last accepted value on its disk. The changing of this variable does not affect the latency of a Paxos instance.

In summary, an acceptor stores its promise on stable storage before it is sent to a proposer. As proposers can be coordinators for many instances, these costs have not to be paid very frequently. More problematic is the issue, that an acceptor has to store a value on disk, it intends to accept. This latency will occur in each instance. If an acceptor crashes and recovers, it will still know the highest promise and the last accepted value, it has sent. Because of that, it is able to answer prepare and propose requests and, what is also important, it can inform the learners about possible consensus values in each instance. Therefore the learners are able to determine eventually the chosen values, which satisfies **Liveness1**.

As we can see, the fastest possible instance still needs one disk-flush time. As the networks get faster and faster, this can be the bottleneck of the multi-Paxos protocol in an environment with a handful consensus servers communicating via a fast network (for a similar configuration see [1]). The purpose of this work is to reach a better performance by not using the critical stable storage operations and still guaranteeing the safety criteria.

5 In-Memory Paxos

We will introduce next *in-memory Paxos*, a new, performance-oriented Paxos variant. First of all, there will be a quick investigation of possible alternatives solving the same problem. The selected approach will be explained in more detail later in this chapter.

5.1 Overview of possible alternatives

We have already formulated a different liveness property (**Liveness2**) that requires a majority of processes to be active. This assumption enables new possibilities to solve consensus. We will give a few of them:

- A simple technique to ensure safety without writes on stable storage is to not allow processes to join the protocol after they have crashed. This approach ensures safety and liveness as long, as a majority of processes have never crashed. Anyway, it is by no means fault-tolerant to simply not allow joining the protocol again. A mechanism should be introduced that enables a crashed and recovered process to rejoin the protocol.
- A more fault-tolerant solution would be to replicate promises and last accepted values across the network into volatile memory other processes to ensure, that a crashed and recovered acceptor is able to get the necessary information. A problem hereby is, that an acceptor has to wait for acknowledgements instead of waiting for disk writes before it actually accepts a proposal. Also the number of messages will increase as n^2, if n is the number of processes, as each acceptor has to replicate state information on a majority of processes (if a majority of processes has to be active, at least one process will still possess state information that are up to date).
- Another quite logical solution that reduces the number of disk writes is to log the highest seen instance number i. If an acceptor sees an instance number $i' \geq i + m$ for a certain window size $m > 1$, it will update $i := i'$. If the acceptor crashes, it will participate only in instances $i' > i + m$ and can be sure, that it has not accepted any value for these instances. If m is chosen big enough, stable storage operations will be very rare. On the other hand, a very high m will slow down the recovery from a crash, because a recovered acceptor will have to wait for a long time until it will be allowed to participate in the protocol again.
- In practical systems it is often necessary to implement group management, so that processes can join and leave the protocol[1, 4, 3]. If group management has to be implemented anyway, another solution will be, that processes rejoin the protocol with a

new id just after they recovered from a crash. Rejoining can be seen as a leave, followed by a join request. Group management is often designed so that the group is fixed for each concrete instance. Changes are only possible between instances so that there is consensus about the group in each instance. Messages that are sent to former processes on the same physical address must be filtered out by comparing the specified receiver id with the own id.

The first solution does not allow group changes, the second introduces additional message delay as well as a big number additional messages. It is hard to believe that this results in any performance improvements. The third solution reduces the number of stable writes, while no extra messages are needed. An advantage over the last alternative is that recovering from a crash is cheap. On the contrary, there are actually some stable writes that are expensive and to choose the right m during design time will be hard. The last option has another advantage: it is the most simple solution, because functionality is used that often has to be present anyway. Therefore, we have chosen the last possibility that will be examined in the following.

5.2 Informal Protocol Description

As the set of processes participating in the protocol is often not fixed, there must be some management of protocol membership, so that there is consensus about the participants of the protocol in each instance. Lamport proposed in [3], that consensus about the group membership can be reached through Paxos itself. If a process p intends to leave the protocol, it will contact the current coordinator, which tries to push the leave-request of p through. The membership in instance i can be defined by the consensus values in all instances up to $i - k$ for a certain window size k. Joining the protocol works just as leaving.

Imaging three processes A,B and C, initially forming the group $\{A, B, C\}$. Each process plays all three roles: proposer, acceptor, learner. The consensus values in the first three instances of the multi-Paxos protocol were the following: (1,“some random value”), (2,“42”), (3,“B leaves the group”). Say process C wants to propose a value in instance 4. It has to determine the current group of acceptors, so that it can decide to whom to send the prepare and propose messages and whether it has received promises and accepts from a majority of processes. If the window size is $k = 1$, process C will have to know all consensus values up to instance $4 - k = 3$. It is now clear that the group that is responsible for instance 4 has changed and is now $\{A, C\}$.

The protocol is very similar to the multi-Paxos protocol with the following changes: There is no logging of accept and promise messages. If an acceptor a crashes, all these information will be lost. Because of that, the recovered acceptor does nothing, except to ask the coordinator to leave the protocol. If the coordinator is sure about that (by reaching consensus via one instance of Paxos), it will acknowledge the leave-request. After that the acceptor may join the protocol by contacting again the coordinator (as optimization, the two steps can also be merged). To preserve confusion, the acceptor changes its id, so that old a-messages in transit can be filtered out by the new acceptor. If this consensus is reached in instance i, the new

participant a' will be allowed to participate in all instances greater than $i + k$, because the group in instance $i + k$ is defined by the instances $1, 2, ..., i$.

A crashed and recovered proposer has lost its state and if it intends to initiate new rounds, it will have to start the protocol again. The issue, that the proposal number has to grow monotonically can easily be ensured by increasing an epoch number on stable storage after each crash.

A crashed and recovered learner has lost all consensus values. But the fact, that a crashed and recovered process does start as completely new process allows the old process to forget the consensus values without affecting any safety properties. A majority of processes has to be up and so the new process can learn the consensus values anyway (if it is interested in the consensus value in instance i, it will simply start a new round for instance i to learn about all previously accepted values). It is worth to mention here that an optimization with respect to reliability would be to log consensus values asynchronously on stable storage, so that recovery is more efficient.

5.3 Formal Protocol Description

The protocol is similar to the original multi-Paxos protocol. If something has changed with respect to the original multi-Paxos protocol, it will be marked with "new". A process plays all three roles. After a crash, there is no state from the old process, except id information.

- Proposer p:

  Old $coordNumber$ ***(stable)*** The coordinator number, which has to be globally unique (so that $owner(coordNumber) = p$ can be determined by any process) and which has to grow monotonically.

  New $coordNumber$ ***(volatile)*** The coordinator number, which has to be globally unique and which has to grow monotonically. The coordinator number can be a composition of the acceptors id (proposer, acceptor and learner build one Paxos process, so the proposer has access to the acceptors id) and the actual proposal number, so that it is ensured, that every process has infinitely many, unique and fresh coordinator numbers.

  $instance$ ***(volatile)*** It is often sufficient to perform the algorithm for all instances, that have a greater or equal instance number than this variable.

  $currVal_i$ ***(volatile)*** The current value to propose in instance i, when $status = proposing$.

  $currPromises$ ***(volatile)*** A set consisting of all promises, it already got for the current proposal $coordNumber$. If $p.status \neq preparing$, $currPromises$ has no meaning and should be the empty set.

status ***(volatile)*** Current phase in the protocol $status \in \{idle, preparing, proposing\}$ If the proposer doesn't know in which phase it is (e.g. after a crash), he will initialize *status* with *idle*.

$currAccepts_i$ ***(volatile)*** A set consisting of all accepts, the proposer p already got for the current proposal $PROPOSE(i, p.coordNumber, p.currVal_i, a)$ in an instance i, where a is the id of an acceptor. If $p.status \neq proposing$, *currAccepts* has no meaning and should be the empty set.

New *windowSize* ***(volatile)*** The set of active participants in instance i (and therefore the size of a majority of acceptors) is defined by the chosen values up to instance $i - windowSize$. This variable is introduced due to the group management requirement and can be initialized, when the proposer starts up the first time.

New *clientValues* ***(volatile)*** This new variable stores values, that can be actually proposed. Clients, that want to initiate consensus about a value, contact a proposer and wait for acknowledgement. The proposer stores the values temporarily in this variable.

- Acceptor a:

  Old: *id* ***(stable)*** Constant id of acceptor.

  New: *id* ***(stable)*** Composition of the actual id of the process and an epoch number, which increases each time, a process recovers and is 0 at the beginning. Therefore an acceptor possesses an infinite set of unique ids. After a crash and recovery and a successful rejoining the protocol, the process sees itself as new process without any knowledge about the old process on the same physical address. It simply ignores messages, that are addressed to some old process.

  Old: *promiseNumber* ***(stable)*** The highest promise number, it has given, initially $-\infty$.

  New: *promiseNumber* ***(volatile)*** The highest promise number, it has given, initially $-\infty$. Because an acceptor with a certain id does only participate as long as it has never crashed, there is no need to remember promises over crash boundaries.

  Old: $lastAccepted_i$ ***(stable)*** Highest numbered accepted proposal in instance i.

  New: $lastAccepted_i$ ***(volatile)*** Highest numbered accepted proposal in instance i. A crashed and recovered acceptor has forgotten all values it has ever accepted, but as it stops and starts up as new process, there can be no violated promises.

- Learner l:

  Old: *consensusValue* ***(stable)*** Set of $(instanceNumber, Value)$ pairs there was consensus about, initially $\emptyset$.

New: $consensusValue$ ***(volatile)*** Set of $(instanceNumber, Value)$ pairs there was consensus about, initially $\emptyset$. If a process (and therefore also a learner) crashes, it will have forgotten previously learned values. This will be no problem as it does not violate the safety properties. The values can be learnt anyway, if a majority of processes is running.

The following condition action pairs have changed with respect to multi-paxos:

Receive client request:
Proposer p receives $write(v)$ from a client of the consensus system.

Update $clientValues = clientValues \cup \{v\}$

Group management joining:
Proposer receiving $join(ip, port, recovered)$ from a process that want to participate as new acceptor.

Update $clientValues = clientValues \cup \{"JOIN : (ip, port, recovered)"\}$

Group management leaving:
Proposer receiving $leave(ip, port, recovered)$ from a process that want to leave the protocol.

Update $clientValues = clientValues \cup \{"LEAVE : (ip, port, recovered)"\}$

Send prepare:
Proposer p may send a PREPARE-message to an acceptor with id a, if $p.status = preparing$.

Send $PREPARE(p.instance, p.coordNumber, a.id)$ request to acceptor a, where $p.instance$ is the lowest instance, for which p wants to reach consensus and $p.coordNumber$ is the coordinator number. Because the acceptor has to know, if the message is destined to itself or to another, older acceptor at the same physical address, the id of the receiver acceptor has to be appended. This actually encodes two requests to an acceptor a:

Give promise never to accept a lower numbered proposal $PROPOSE(i, n', v, a.id)$, where the coordinator number n' is lower than n, v is any value and i is any instance number.

Give highest-numbered accepted proposal smaller n for each instance greater or equal than $p.instance$.

Receive prepare:
Acceptor a receives $PREPARE(i, n, id)$ request, where $n > a.promiseNumber$ and $id = a.id$. The acceptor ignores messages, that are not destined to itself, so that no old message disturbs its protocol.

Update $a.promiseNumber = n$.

Propose value:
Proposer p with $p.status = proposing$ wants to propose a value for instance $i \geq p.instance$.

> Update $p.currVal_i$ as the value of the highest numbered accepted proposal in $p.currPromises$ for instance i.
>
> Send $PROPOSE(i, p.coordNumber, p.currVal_i, a.id)$ to a majority of acceptors, if $a.id$ is the id of each individual acceptor to which the message was sent.

Accept value:
Acceptor a receives $PROPOSE(i, n, v, id)$, where $n \geq a.promiseNumber$ and $id = a.id$.

> Update $a.lastAccepted_i = (n, v)$.
>
> Send $ACCEPT(i, n, a.id)$ to $owner(n)$.

Handle get request:
Learner receiving $read(i)$.

> Send v back, if $(i, v) \in consensusValue$. This can be seen as fast read. Another slower variant will be to ask enough other processes, if they know about a consensus value in instance i, so that the client can be sure about the fact whether there was or was not consensus.

To put it altogether, we will give the condition action pairs that haven't changed. The rest of this chapter can also be skipped by readers that want to avoid redundancy.

Initiate new round:
Proposer p is free to initiate a new round by proposing itself as coordinator for all instances greater than $p.instance$.

> Update $p.status = preparing$.
>
> Update $p.coordNumber = next(p.coordNumber)$, where $x < next(x) \forall x$.
>
> Update $p.currPromises = \emptyset$.

Send promise:
Acceptor a sends a promise.

> Send $PROMISE(a.promiseNumber, a.id, lastAccepted_{\geq i})$ to $owner(a.promiseNumber)$, where $lastAccepted_{\geq i}$ is the set $\{a.lastAccepted_{i'} | i' \geq i\}$.

Collect promises:
Proposer p with $p.status = preparing$ or $p.status = proposing$ receives $PROMISE(n, a, V)$, where $n = coordNumber$.

> $currPromises = currPromises \cup \{(n, a, V)\}$.

Start proposing phase:
Proposer p with $p.status = preparing$ got promises from a majority of acceptors in instance i.

Update $p.status = proposing$.

Update $p.currAccepts_i = \emptyset \ \forall i \geq p.instance$.

Update $p.currVal_i$ as the value of the highest numbered accepted proposal in $p.currPromises$ for instance i.

Collect accepts:
Proposer p receives $ACCEPT(i, n, a)$, $n = p.coordNumber$ and $p.status = proposing$.

Update $currAccepts_i = currAccepts_i \cup \{(n, a)\}$.

Send consensus:
Proposer p has got $ACCEPT$ messages from a majority of the acceptors in any instance $i \geq instance$ and $p.status = proposing$.

Send $CONSENSUS(i, currVal_i)$ to the learners.

Receive consensus:
Learner receiving $CONSENSUS(i, v)$ from proposer p.

Update $consensusValue = consensusValue \cup \{(i, v)\}$.

In summary, there were only a few changes due to the necessity of group management (handling join and leave requests), another few changes, so that an interface to clients can be offered (issue values and get consensus values), and the critical changes of the kind of storage on some variables.

6 Formal Proof

We already know, that the original Paxos protocol satisfies **Safety1**,**Safety2** and **Safety3**. For the proof, we assume, that the multi-decree protocol is already correct and show, that the introduced changes do not affect the safety properties. The new liveness criteria is shown afterwards.

6.1 Safety

We first show an important observation, that eases the actual proof, because it makes clear, that it is not needed for a Paxos process to recover eventually:

Theorem 1. *If a normal Paxos process (playing all three roles: acceptor, proposer and learner) does not participate in the protocol, safety will be still guaranteed.*

Proof. Assume safety won't be guaranteed, if a normal Paxos process does not participate in the protocol for a certain timespan t

> A possible scenario in the original Paxos protocol is, that all messages send to and from a concrete Paxos process p get lost for a timespan $t' > t$.
>
> In this scenario, safety would not be guaranteed in the original Paxos protocol, because of the assumption.
>
> But safety is guaranteed by the original Paxos protocol

□

Now, we show for each change of the original Paxos protocol, that the safety property is not affected.

- The variables $coordNumber$, $promiseNumber$, $lastAccepted_i$ and $consensusValue$ become volatile. If the processes participated in the protocol after they crashed and recovered, this would be problematic, because the state before the crash would be lost. However, crashed processes do never participate in the protocol. They join in with a new id as new process. So with theorem 1, it is clear that only making stable variables volatile and not participating in the protocol after a crash does not affect the safety property. Processes with volatile instead of stable variables only participate in the protocol while the actual values of the variables are the same as the values of the stable variables. At

the moment, where the variables could differ, the in-memory processes do not participate anymore.

- The variable *id* of the acceptor stays constant as long as a process participate. Hence there is actually no change to the original protocol.
- There were three new condition action pairs: receiving values from clients, join requests and leave requests. As we can see easily, the actions only change the new variable $clientValues$. This can't make the safety properties false.
- Each message to an acceptor is associated with the intended receiver of the message. This is done, so that an acceptor can prove, that a prepare or proposed message it receives is destined for itself and not for an acceptor, that had been listened on the same port in the past. So we have a bit more traffic in the network, but the same situation like before, each acceptor receives only messages, that are destined to itself. If the protocol satisfies the safety properties already, after this change safety is still valid.
- The learner reacts on a new condition: it sends consensus values to clients, if they ask for them. As this has nothing to do with the protocol, it does not change safety.

6.2 Liveness2

Liveness2. If a majority of processes is up at any point in time, some of the proposed values is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

This are actually two criteria that will be shown one after another (Theorem 2 and Theorem 3).

Theorem 2. *If a majority of processes is up at any point in time, some of the proposed values is eventually chosen.*

Proof.
1. A value v is proposed in instance i from the coordinator with coordinator number n, iff at least $\lceil \frac{m+1}{2} \rceil$ acceptors out of m acceptors have received $PROPOSE(i, n, v, a)$, where a is the unique id of each acceptor, so that it is able to identify messages, that were sent to itself.
2. Assume there were k proposer $p_1, p_2, ..., p_k$, seeing themself as coordinators that may propose values. Proposer p_1 sends $PROPOSE(i, n_1, v_1, a)$ and hence proposes value v_1 in instance i with coordinator number n_1 to acceptor a for enough different acceptors. Proposer $p_j, j \in \{1, 2, ..., k\}$ sends $PROPOSE(i, n_j, v_j, a)$ to enough different acceptors.
3. Without loss of generality we say, that $n_1 > n_j$, $\forall j \in \{1, 2, ..., k\}$.
4. Because of 1. and 2. we know that a majority of acceptors received $PROPOSE(i, n_1, v_1, a)$, because of 3. we know that $n_1 > n_j$, $\forall j \in \{1, 2, ..., k\}$.

5. An acceptor a accepts $PROPOSE(i, n, v, a)$ if $n \geq a.promiseNumber$. The coordinator with number n_1 has the highest coordinator number, hence for each Acceptor a it is true that $a.promiseNumber \leq n_1$.
6. Out of 4. and 5. follows, that the proposal with coordinator number n_1 is majority accepted (chosen).
7. Some of the proposed values is chosen.

□

Theorem 3. *If a majority of processes is up at any point in time and a value v was chosen, then a process can eventually learn the value.*

Proof. Assume the negation: a process can not learn the chosen value v, when a majority is up and v was chosen in instance i. The value v was proposed by a proposer with coordinator number n.

- If a new proposer sends $PREPARE(i, n', a)$ with the coordinator number $n' > n$, acceptor a will answer with $PROMISE(n', a, a.lastAccepted_{\geq i})$. As a reminder: the set $a.lastAccepted_{\geq i}$ contains the accepted values (by acceptor a), that were proposed by the highest numbered coordinator in each instance $\geq i$.
- Each two majorities overlap, that means a proposer receiving a majority of promises knows at least one pair (n, v) (a majority of acceptors has saved (n, v) in $lastAccepted_{\geq i}$.
- The new proposer p from above learns $p.currVal_i = v$, because n is the highest numbered accepted proposal in instance i.
- Some process can actually learn the value, which is a contradiction

□

7 Performance Evaluation

In this chapter, our practical implementation of the in-memory Paxos protocol will be evaluated with respect to throughput and latency of the in-memory approach and compared to other systems using persistent state like Zookeeper and Open Replica.

7.1 Evaluation Setup

The consensus system as well as the clients of our system were implemented in Java. We have evaluated the system on 9 servers, each possesses two processors (Intel(R) Core(TM)2 Duo CPU with 2.40GHz) and 2 GB RAM. The servers were connected via Ethernet with 0.518 ms average roundtrip time for 64 Bytes (ping).

In each scenario, five servers were used to run the consensus system (in-memory Paxos and Zookeeper) and the others to run clients of the system. Although Zookeeper does not use the Paxos protocol (it uses a special protocol that reminds on Paxos: Zookeeper Atomic Broadcast(ZAB), see [12]), it is a very fast consensus service which makes use of stable storage operations. As we focus on write-intensive workloads, the clients only issue write requests. Each client iteratively asks the system to write an integer, waits for acknowledgement and counts positive replies. These indicate that the system has processed the write request successfully. We used the Zookeeper client library to write Java clients, so that the type of clients is the same in both systems: a client issues one integer value after another, as fast as possible.

In order to collect measurements for various different number of clients, the number of clients increases linearly during execution time.

7.2 Results

We have measured throughput and latency of both systems. Latency means the time needed for one request. The number of successful requests per second summed over all clients is taken as the system's throughput.

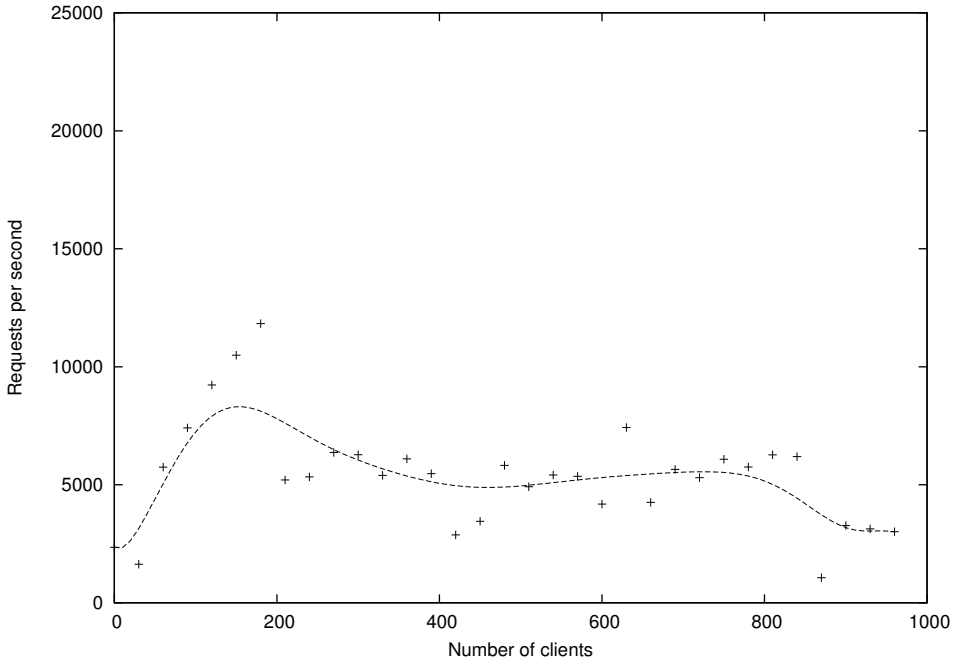

Figure 7.1: Throughput Zookeeper for 100 percent writes

7.2.1 Throughput

Initially, we compare the throughput of Zookeeper and the in-memory system. We give the client distribution window for which both systems scale optimal.

Zookeeper reaches an optimal throughput of 12000 requests per second, if 200 Zookeeper clients issue requests to the system. When the number of clients increase further, a negative effect on the throughput of the system is visible. However, for most configurations the throughput rate was between 3000 and 7000 requests per second (Figure 7.1). For a higher number of clients (=1000) the overall throughput performance decreases slowly.

Figure 7.2 shows the throughput of our in-memory Paxos system. The system reaches much higher throughput rates between 10000 and 18000 requests per second. In most configurations the throughput is two or three times higher than Zookeeper's. The highest throughput rate that was measured for in-memory Paxos is 18000 requests per second for 1500 clients. This seems to be also the limit of scalability, for more clients the throughput rate decreases.

The throughput evaluation shows that the in-memory approach outnumbers the highly-optimized Zookeeper system for write intensive workloads.

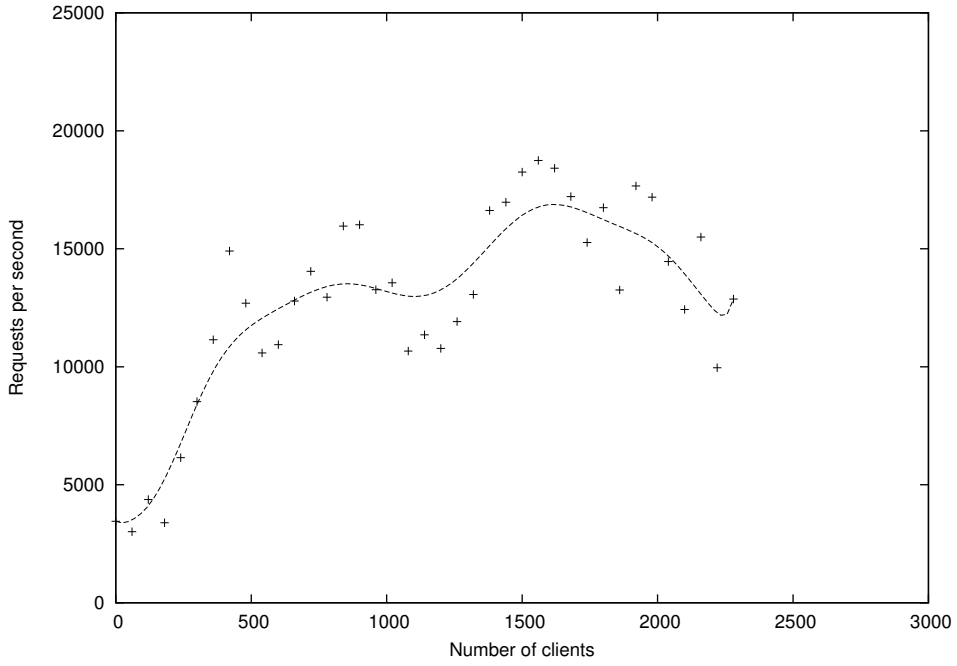

Figure 7.2: Throughput in-memory Paxos for 100 percent writes

There are of course other consensus systems that are based on the classical Paxos protocol, for example OpenReplica [10], which is also a performant consensus system, but that makes use of stable storage to provide higher availability and reliability. We have not evaluated OpenReplica on our servers, but the paper gives a throughput rate of 327 requests/second, even though faster servers with higher RAM were used to evaluate OpenReplica.

Google's Chubby, another famous system that is based on Paxos, has throughput rates up to 640 requests per second for 5 bytes requests (what is near to our 4 bytes integer requests). This rate was given in [4].

7.2.2 Latency

Another important performance metric is the latency of client requests. As already mentioned, the number of clients is increased linearly during the evaluation. In both systems, the number of clients does not have great influence on the latency for successful requests which means that we did not reached the scalability limit of both systems with respect to latency.

Zookeeper achieves latencies between 10-20 ms for more than 65% of the requests (see Figures 7.4 and 7.3). Very few client requests were answered in less than 5 ms (<1%).

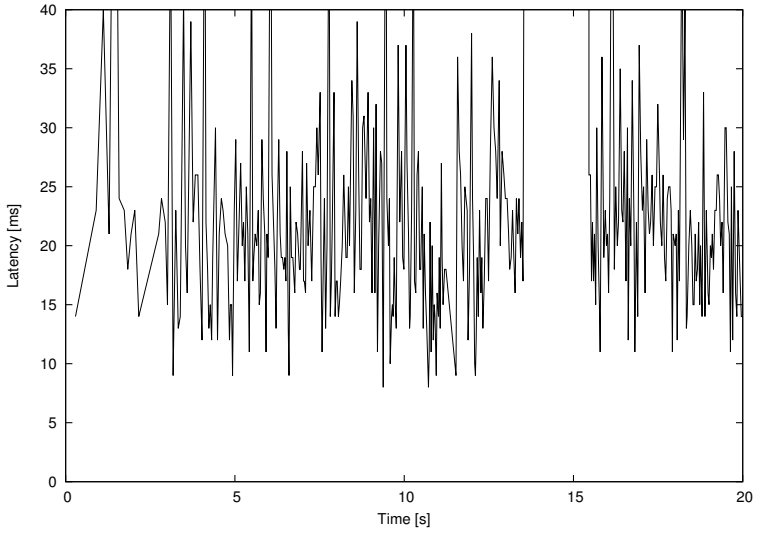

Figure 7.3: Latency Zookeeper during runtime

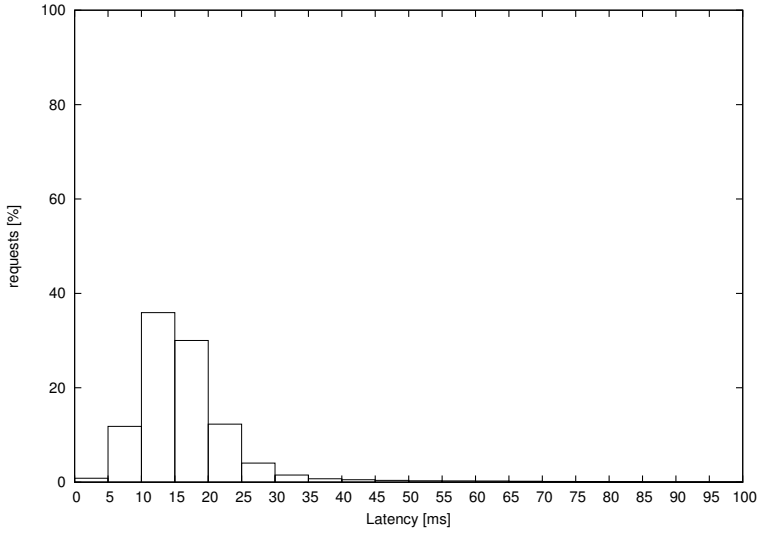

Figure 7.4: Latency distribution Zookeeper

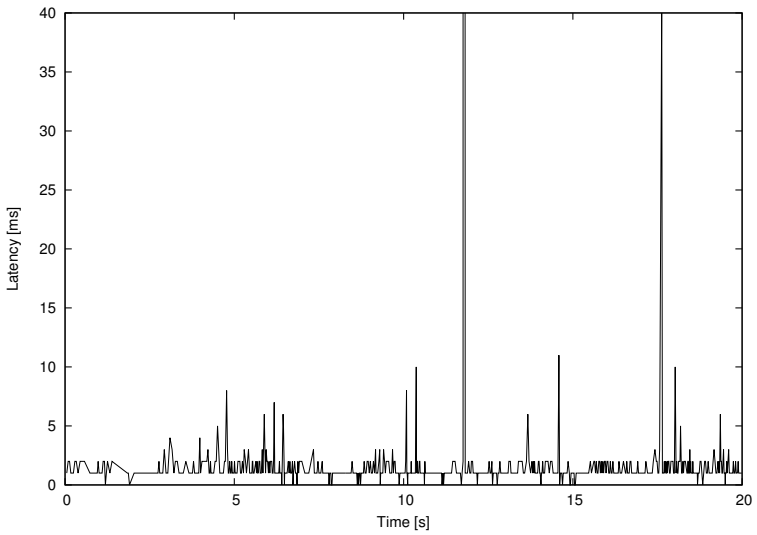

Figure 7.5: Latency in-memory Paxos during the system runs

The in-memory approach handles more than 97% of the requests in less than five milliseconds, most requests can be answered in one or two milliseconds (see Figures 7.6 and 7.5).

Summarily, the in-memory approach has very fast response times and achieves high throughput rates. As in many distributed systems, our system is very sensitive to some parameters for example how many requests will be handled at once in a Paxos instance, how many clients may issue values and how often a proposer will try to take over leadership via prepare messages. We have reached more than 15000 instead of 1000 requests per second only by varying some system parameters and allowing the system to handle an variable instead of a fixed number of client requests per Paxos instance.

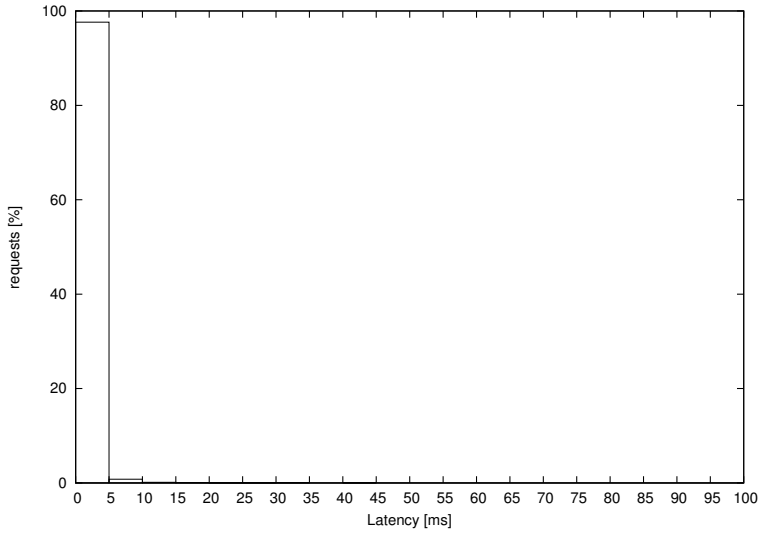

Figure 7.6: Latency distribution in-memory Paxos

8 Summary

In practice there is often a big dependency between a system that offers "consensus as a service" and the numerous client applications, that make use of the service (e.g. Google's Chubby is used by Google BigTable and Google File System). As these client programs become faster, the consensus service has to have low latency and high throughput. The performance bottleneck of the consensus service itself is often the disk flush time, which is hundreds of thousand times slower than access to volatile memory.

In order to improve performance, we have investigated, implemented and evaluated an in-memory version of the Paxos protocol, that still guarantees the desired safety properties so that there are no inconsistencies.

As we have seen in the last chapter, the in-memory Paxos protocol has beneficial practical implications in terms of higher performance. But it is also clear, that the probability of data loss will be higher, if each server holds its database in memory only. This will be the case, if more than f out of $2f + 1$ processes fail.

An optimization of the in-memory Paxos approach with respect to reliability could be to additionally store learned values asynchronously on disk. If a majority of processes fails, the protocol will stop, but it will be possible to do some recovery after the system has rebooted. Consider an in-memory system that has reached consensus in slots $0, 1, ..., i$ and processes have stored asynchronously consensus values for slots $0, 1, ..., i'$ with $i' < i$. After a disastrous crash, where all servers fail completely, the system has to be rebooted, but can be initialized with consensus values for the first i' slots.

If more availability and reliability is needed, the number of servers can be increased accordingly. In this case, we have to give back some of the high performance. This kind of trade-off is very typical. It may be interesting to examine in more detail how to keep the system still high performant by guaranteeing enough availability and reliability.

We think that the in-memory Paxos approach is very promising and flexible and can be used on its own or even extended and combined with other approaches to build a highly performant system that still guarantees consistency.

Bibliography

[1] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, (Berkeley, CA, USA), pp. 335–350, USENIX Association, 2006. (Cited on pages 5, 10, 21, 26 and 27)

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010. (Cited on pages 5, 10 and 21)

[3] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998. (Cited on pages 5, 6, 9, 17, 22, 27 and 28)

[4] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, (New York, NY, USA), pp. 398–407, ACM, 2007. (Cited on pages 7, 9, 10, 17, 21, 25, 27 and 41)

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008. (Cited on pages 5 and 10)

[6] L. Lamport, "Paxos Made Simple," *SIGACT News*, vol. 32, pp. 51–58, Dec. 2001. (Cited on pages 9, 13 and 22)

[7] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, pp. 79–103, Oct. 2006. (Cited on page 9)

[8] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012. (Cited on page 9)

[9] K. Birman, D. Freedman, Q. Huang, and P. Dowell, "Overcoming cap with consistent soft-state replication," *Computer*, vol. 45, no. 2, pp. 50–58, 2012. (Cited on page 9)

[10] D. Altınbüken and E. G. Sirer, "Commodifying replicated state machines with openreplica," tech. rep., Computer Science Department, Cornell University, 2012. (Cited on pages 10 and 41)

[11] A. Tanenbaum, *Computer Networks*. Prentice Hall Professional Technical Reference, 4th ed., 2002. (Cited on page 11)

[12] F. P. Junqueira, B. C. Reed, and M. 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, (Washington, DC, USA), pp. 245–256, IEEE Computer Society, 2011. (Cited on page 39)

[13] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," *Distrib. Comput.*, vol. 13, pp. 99–125, Apr. 2000.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[15] B. W. Lampson, "How to build a highly available system using consensus," in *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, (London, UK, UK), pp. 1–17, Springer-Verlag, 1996.

All links were last followed on May 11, 2013.

Decleration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature