Paxos made code

Implementing a high throughput Atomic Broadcast

Master's Thesis submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Dependable Distributed Systems

presented by
Marco Primi

under the supervision of
Prof. Fernando Pedone

May 2009

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Marco Primi
Lugano, 29 May 2009

Abstract

The PAXOS algorithm is used to implement Atomic Broadcast, an important communication primitive useful for building fault-tolerant distributed systems. Transforming a formal description into an efficient, scalable and reliable implementation is a difficult process that requires addressing a number of practical issues and making careful design choices. In this document we share our experience in building, verifying and benchmarking different Paxos-based implementations of Atomic Broadcast.

Acknowledgements

Daniele Sciascia is coauthor of *libpaxos-T* and *libfastpaxos*.
Thanks to Nicolas Schiper for some interesting insights and tests about the issue described in Section 4.6.1.

Contents

Contents **viii**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**
  1.1 Related work 3
  1.2 LibPaxos Motivation 4

2 The Paxos algorithm **5**
  2.1 Atomic Broadcast and Multipaxos 5
  2.2 Actors 5
    2.2.1 Learner 6
    2.2.2 Acceptor 6
    2.2.3 Proposer 7
    2.2.4 Leader Proposer 7
  2.3 Phases 7
    2.3.1 Phase 1 7
    2.3.2 Phase 2 9
  2.4 Example execution 9

3 LibPaxos: an open-source Paxos implementation **13**
  3.1 Implementing Paxos 13
  3.2 Design choices 14
    3.2.1 Network model 14
    3.2.2 Client model 15
  3.3 Implementing the learner 16
  3.4 Implementing the acceptor 17
    3.4.1 Snapshots 18
  3.5 Implementing the proposer 18

3.5.1 Pending list size 22
3.5.2 Ballots and rejects 22
3.5.3 Calibration and event counters 23
3.5.4 Memory allocation strategies 24
3.6 Handling Failures 25
3.6.1 Acceptor Failures 25
3.6.2 Learner Failures 25
3.6.3 Proposer Failures 25
3.7 Optimizations 26
3.7.1 Phase 1 pre-execution 26
3.7.2 Parallel Phase 2 27
3.7.3 Message batching and compression 27
3.7.4 Value batching 27
3.7.5 Learning acceptors 28
3.7.6 Relaxed stable storage 28
3.7.7 Selective quorums 28

4 Performance evaluation **31**
4.1 Experiments setup 31
4.1.1 Infrastructure 31
4.1.2 Benchmark client 31
4.2 An ideal Atomic Broadcast 32
4.3 Example experiment 32
4.4 Impact of acceptors durability 33
4.5 Impact of values size 34
4.6 Scalability 36
4.6.1 Multicast switching issues 37
4.6.2 Number of acceptors 38
4.6.3 Number of learners 39
4.7 Recovery times for proposers 40

5 Conclusions **43**
5.1 Discussion 43
5.2 Future work 43

Bibliography **45**

Figures

2.1 The phases of the Paxos algorithm from ABroadcast to ADeliver. This procedure is required to successfully bind a value to an instance. 8

3.1 The finite state machine depicting the states of any instance of the algorithm 19

4.1 Delivery rate for different client value sizes in values per second and kilobytes per second. Higher is better. 35
4.2 90% confidence intervals of delivery latency for different client value sizes. Smaller is better. 35
4.3 Delivery rate (left axis) and delivery latency (right axis) with different number of acceptors. 39
4.4 Delivery rate (left axis) and delivery latency (right axis) with different number of learners. 40
4.5 Delivery rate over time, around second 65, we force the leader election service to choose a new coordinator among the proposers. 41

Tables

4.1 Comparison of serializer against PAXOS. Best throughput obtained with a single client sending multiple values concurrently. 33
4.2 Comparison of durability methods for PAXOS and the serializer. Strict durability (*TR* and *BD* columns) has a significant impact on delivery rate. 34

Chapter 1

Introduction

Historical notes

The PAXOS algorithm was originally presented by Leslie Lamport in 1990. However, at that time, only few people realized its potential. In fact, very few people understood that the paper was about distributed systems at all, probably because it was written like a story about ancient greek legislators. It took almost ten years and a complete rewrite to get the paper published in 1998 [Lam98].

After that, PAXOS quickly gained a special spot in the field of distributed systems because of its simplicity and its weak, realistic assumptions. Numerous studies decomposed, improved and built upon PAXOS in the following years [Lam02; CSP07]. The algorithm is nowadays used as a fundamental building block of many fault-tolerant production systems. It is for example the base for Google's Chubby Distributed Lock Service [Bur06], a component used in commercial products like Analytics and Earth.

What is Paxos

PAXOS is an elegant solution to the *consensus* problem in distributed systems, informally defined as follows: a set of processes starts, each one with some initial value. Through some communication protocol, all of them should come to an agreement and unambiguously choose a single value among the initial ones.

PAXOS can solve this problem in an asynchronous model despite having very weak, realistic assumptions. Channels can, for example, lose, reorder or duplicate messages. Moreover only three message delays are required to make each participant aware of the final value. The protocol is fully decentralized, crashed processes can be replaced without having to stop the system, unless a critical number of them fail. Agreement on the chosen value is never violated, despite the number of failures.

Solving consensus in an efficient way allows us to build an *Atomic Broadcast* protocol (a.k.a. *total order broadcast*), which ensures that messages are received reliably and in the same order by all participants. This is accomplished by executing different consensus *instances*. The Atomic Broadcast usually consists of two primitives:

- $ABroadcast(v)$: An action invoked by clients when they want to submit the value v to the network.
- $v = ADeliver()$: An action invoked on each client listening to the broadcast whenever the next value v is decided.

The broadcast, and therefore PAXOS, must satisfy the following conditions [DSU04]:

- Validity: if a correct process ABroadcasts a value v, then it eventually ADeliver v.
- Uniform Agreement: if a process ADelivers v, then all correct processes eventually deliver v.
- Uniform Integrity: for any value v, every process delivers v at most once, and only if v previously submitted by some process.
- Uniform Total Order: if a process delivers value v before delivering value v', then all processes deliver v before v'.

What is Paxos used for?

In distributed systems, it is often the case that processes need to take coordinated actions in order to maintain consistency. An easy solution to this problem is to have a single process acting as a coordinator, serializing and distributing a stream of submitted commands that all participants will receive and execute in the same order. This technique is generally known as the Finite state machine approach for distributed systems [Lam96].

In such a system, however, the coordinator is a single point of failure and most likely a bottleneck. By using an Atomic Broadcast protocol instead, one can get rid of the coordinator, making the system truly decentralized and fault tolerant since it is equivalent to having a coordinator that never crashes.

Other than being provably correct for this task, PAXOS has also a lot of potential for having good throughput and low latency, if correctly implemented. For this reasons, it is currently used in numerous production systems, like Google's Chubby [Bur06].

As a simple example, observe the following scenario for a fully-replicated distributed database: clients perform concurrent operations through one of the servers, which is currently acting as master. Normally this requires some expensive communication protocol to ensure that all replicas update their local copy before answering a client (e.g., Two Phase Commit). Failures during those operations may lead to even more complicate recovery schemes.

If we instead can rely on a powerful primitive like Atomic Broadcast, building such a system becomes trivial: the master submits each client operation through the ABroadcast, when that command is ADelivered, the master applies the changes and answer the client. The replicas will eventually apply the same changes in the same order, therefore reaching the same state. The failure of the master does not involve complex recovery mechanisms, the clients just have to switch to another replica. In fact, in this situation, a master is not required anymore, clients submit commands directly through ABroadcast. Notice how the complexity of building the system reduces to the complexity of implementing a total order broadcast protocol. If we can build a scalable, reliable and fast Atomic Broadcast, we have an easy way of building large systems on top of it.

1.1 Related work

Although it was introduced 10 or so years earlier, the PAXOS algorithm was published in 1998 by Leslie Lamport in a paper titled "The Part-Time Parliament" [Lam98]. Initially, few people understood and worked on the protocol, mostly providing alternative decompositions with corresponding proofs of correctness [PLL97; RPSR03]. Later, Lamport published another paper containing a from-scratch, simpler description of the algorithm [Lam01a].

The protocol gained momentum as more people understood its usefulness for building fault-tolerant distributed systems. Alternative versions of the algorithm started to surface, providing agreement in different models, like in presence of byzantine failures or in disk-based scenarios [Lam01b; Lam02; GL03; Zie04]. Other revisions proposed faster and cheaper modifications [LM04; Lam06; CSP07].

Up to this point, the literature is still mostly theoretical. Only recently, PAXOS started to be used as a building block for real-world production systems like Chubby [Bur06], a fault-tolerant distributed lock service used in many Google systems like BigTable[CDG+06] and the Google File System[GGL03], which are in turn components used in commercial products like Google Earth, Google Analytics and the engine behind Google's web crawling engine.

A particularly relevant paper for us is [CGR07], which for the first time discusses the practical point of view of implementing PAXOS, providing interesting insights on how Google engineers proceeded in developing, evaluating and testing their system.

1.2 LibPaxos Motivation

As described in more detail in the next chapter, the PAXOS algorithm looks very simple. However we find, as well as others [CGR07], that translating the algorithm into well-performing code is non-trivial. Moreover it requires making a number of design choices and addressing different practical issues.

Since we spent much time building different PAXOS implementations, we share our experience and motivate our choices. We think this document can be a useful guide for anyone willing to implement a Paxos-based protocol from scratch.

The main contribution that we make is releasing *LibPaxos* as open-source software. The project consists of a collection of PAXOS implementations useful as reference and example of the ideas described in the rest of this document. We hope *LibPaxos* can one day be used as a brick for easily building fault-tolerant distributed systems.

Chapter 2

The Paxos algorithm

2.1 Atomic Broadcast and Multipaxos

Atomic Broadcast is a powerful communication primitive for distributed systems; it allows different processes to reliably receive an ordered sequence of values. Using this primitive leads to simpler application design, since the latter can be built on top of an algorithm proven to be correct.

The PAXOS algorithm for solving consensus is used to implement a fault-tolerant Atomic Broadcast. This is done by executing subsequent instances of consensus, each one uniquely identified by a monotonic increasing number (*instance identificator*, or *iid*). The resulting protocol (known also as MULTI-PAXOS), has been shown correct [Lam01a]: it satisfies the safety requirement of *agreement*, *validity* and *integrity* despite the number of failures. Progress is granted as long as a subset of processes is alive and communicating, meaning that adding more machines makes the system more resilient. PAXOS only tolerates crash-stop failures, although it can be modified to survive byzantine failures too [Lam01b; Lam02; Zie04].

2.2 Actors

A distributed application that uses PAXOS as a black-box providing Atomic Broadcast, has different processes that are interested in receiving values, submitting them, or both. We refer to this processes as *clients*.

Within the black box instead, there is another set of processes performing specific protocol tasks, we refer to those as *proposers*, *acceptors* and *learners*. Notice that this is a logical modularization for the sake of simplicity; in reality a client process could also be an acceptor, a learner, a proposer, a combination of them or neither of those.

2.2.1 Learner

The task of the learner consists of listening to acceptors decisions, in order to deliver the ordered sequence of values. A client interested in listening to the stream is either a learner or receives values from one.

Whenever the learner realizes that a *majority* of acceptors has been reached for an instance, it knows that the value for that instance is permanently and unambiguously chosen. Reaching a majority of acceptors means receiving a certain number of valid acknowledgements from them, this mechanism is covered in more details later.

Values are delivered in the same order by all learners, starting from instance number 1. Knowing the chosen value for an instance is therefore not sufficient for delivering it: all instances from 1 to the current iid must have been delivered already. For example, assume a learner delivered values for instances up to number 10. The value for instance 11 is not known yet but the values for instances 12 and 13 are; we call instance 11 a *hole*. In this situation, a learner should contact the acceptors and ask them to repeat their decision for the missing instance. Once iid:11 is closed, it can be delivered, followed by iid:12 and iid13.

2.2.2 Acceptor

The task of the acceptor is relatively simple: it sits waiting for messages from proposers or learners and answers to them. For each instance, the acceptor keeps a state record, consisting of $< iid, B, V, VB >$, where B is an integer, the highest *ballot* number that was accepted or promised for this instance, V is a client value and VB is the ballot number corresponding to the accepted value. The three fields are initially empty.

The proposer sends two kinds of requests: *prepare* and *accept*, the acceptors react to them in the following way.

A prepare is a tuple $< i, b >$, where i is the instance number and b a ballot. Unless the ballot B in the acceptor record for instance i contains a number higher than b, the acceptor grants the request. It sets $B = b$ and answers with a *promise* message, consisting of $< i, b, V, VB >$, where V and VB are null if no value was accepted yet.

An accept is a tuple $< i, b, v >$, where i is the instance number, b is a ballot and v is a value. Unless the ballot B in the acceptor record for instance i contains a number higher than b, the acceptor grants the request. It sets $B = b$, $V = v$, $VB = b$ and answers with a *learn* message, consisting of $< i, b, v >$. The acceptor should periodically repeat the state of the highest instance for which some value was accepted. By doing so, it helps learners to stay up-to date when message loss occurs, since they can realize if they have holes and act accordingly.

Notice that for the protocol to work correctly, acceptors are not allowed to lose any information in case of a crash. This means that before answering any request, the updated state must be saved into stable storage (i.e., synchronously written to disk).

The number of acceptors is pre-determined and fixed. Progress can be achieved only as long as a majority of them is alive.

2.2.3 Proposer

The proposer is responsible for pushing values submitted by the clients until those are delivered. Proposers relies on an external leader election service, which should nominate a *coordinator* (or *leader*) among them. Even a weak failure detector is sufficient for this task. Proposers that are not the current coordinator can be idle; their only task is to be ready to take over the leadership, if the leader election service says so. The Leader proposer instead is the "active" component in the system that tries to deliver values as fast as possible.

2.2.4 Leader Proposer

The leader proposer sends client values through the broadcast. For each client value submitted, it chooses the next unused instance and tries to bind the the value to it. This process is executed in two phases, the second phase can be started only on successful completion of the first one, as described in more details in the next section.

2.3 Phases

Figure 2.1 depicts the sequence of events required in each instance in order to deliver a value.

2.3.1 Phase 1

In the first phase, the proposer sends a *prepare* message to the acceptors consisting of $< i, b >$, where i is the instance number and b a ballot. It should also set a timeout for this instance at some point in the future. Acceptors that did not acknowledge a higher ballot, answer with a *promise*, consisting of $< i, b, V, VB >$, where V and VB are null if no value was accepted yet. The Leader has to wait until either *(i)* a majority (i.e., $\lfloor n/2 \rfloor + 1$ where n is the number of acceptors) of promises are received from distinct acceptors, or *(ii)* the timeout expires. In the

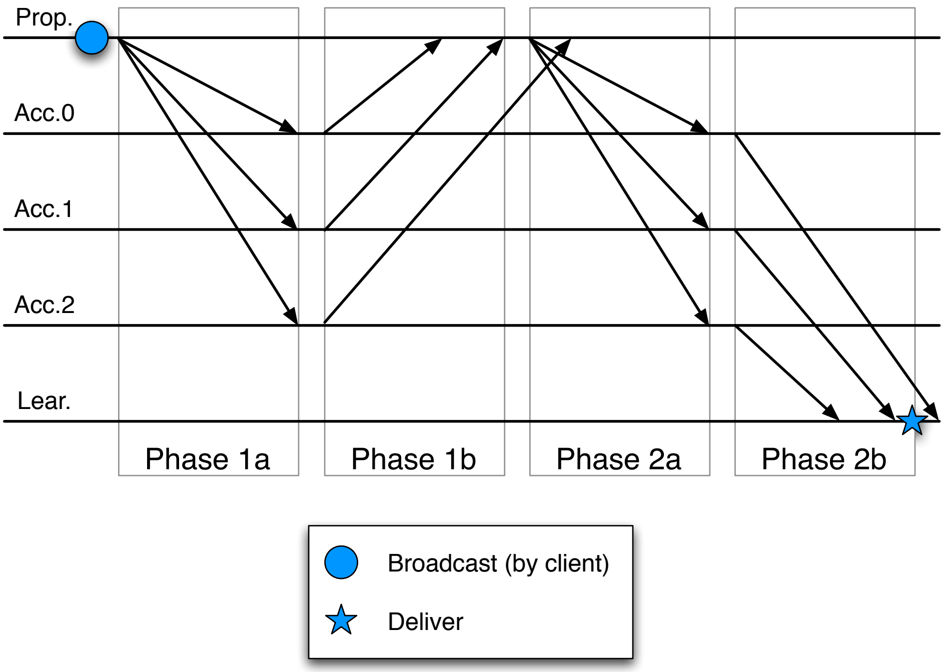
Prop.
Acc.0
Acc.1
Acc.2
Lear.
Phase 1a
Phase 1b
Phase 2a
Phase 2b
Broadcast (by client)
Deliver

Figure 2.1: The phases of the Paxos algorithm from ABroadcast to ADeliver. This procedure is required to successfully bind a value to an instance.

first case the instance is declared *ready*, and phase 2 can begin. In the second case the proposer will increment its ballot and retry executing phase 1; it cannot execute phase 2.

Notice that for this mechanism to work, ballot numbers produced by different proposers should be distinct, for example proposer number 1 uses ballot numbers 101, 201, 301, ..., while proposer 5 uses ballot numbers 105, 205, 305, ... (in this case at most 100 proposers are allowed).

2.3.2 Phase 2

Depending on the outcome of phase 1, the proposer may need to take different actions in phase 2. Specifically, if all the promises received contained a null value V, the instance is *empty*. In this case the leader can send a value submitted a client. If instead some value was found in the promises, the instance is *reserved*. The coordinator is forced to select the value V with the highest associated ballot VB and execute phase 2 with it.

Once a value is picked with the above rule, the proposer sends an *accept* and again sets a timeout for this request. The accept consists of $< i, b, v >$, where i is the instance number, b is the ballot used in phase 1 and v is a value. Acceptors that did not acknowledge a higher ballot will accept the request and answer with a *learn* message, consisting of $< i, b, v >$. If a majority of acceptors accepted the value, it is safe to assume that the instance is *closed* with v permanently associated to it. Nothing can happen in the system that will changes this fact, therefore learners can deliver the value to their clients. In case of request timeout, the Leader has to start over by incrementing the ballot and executing phase 1 again.

We previously said that learn messages are sent by acceptors to learners. The Leader should receive those messages too or it can internally use a learner and wait for it to deliver.

2.4 Example execution

Let us observe an example execution for one instance of the protocol. The network is composed of a single client C, two proposers $P1$ and $P2$, three acceptors $A1$,$A2$,$A3$. $L1, L2$ and $L3$, are three learners started by the client and by the proposers. $P1$ is initially the leader.
Initialization: C sends the value v to the current leader $P1$.
Phase 1a: $P1$ sends a prepare message consisting of an instance number (1 in this case) and a ballot number (i.e., 101).
$P1$ does not receive any promise from the acceptors because of message loss. It

increments the ballot (to 201) and retries.
Phase 1b: The three acceptors receive the prepare request, since none of them acknowledged any ballot higher than 201 for instance 1, they update their state in stable storage and send the corresponding promise message. $P1$ receives two promises from $A2$ and $A3$, it can declare this instance ready since the value in both promises is null.
Phase 2a: Instance 1 is ready with no value in it. The proposer sends an accept message containing the previously used ballot 201 and the value v received from the client.
Phase 2b: Again the acceptors did not acknowledge a ballot higher than 201, therefore they accept the request. After updating their state permanently, they send a learn message to all learners, announcing their decision to accept v for instance 1 with ballot 201. As soon as the learners realize that a majority of acceptors granted the same request (same ballot), they can deliver v. In this way, $P1, P2$ and C are notified that the value was accepted. In case of a timeout in phase 2, $P1$ must restart from phase 1, using ballot 301.

As another example, let us go through the worst possible scenario we can think of. The network is composed of two proposers $P1$ and $P2$, three acceptors $A1$,$A2$,$A3$ and a client C. $C, P1$ and $P2$ internally start three learners, respectively $L_c, L1$ and $L2$.

$P1$ is the current leader and manages to deliver instances up to $i - 1$ (included). $P1$ successfully completes phase 1 for instance i, it sends an accept containing the value v_i submitted by C. Acceptors $A1$,$A2$ receive the valid request, accept it, and update their state in stable storage. At this point the value v_i is unambiguously chosen for instance i, since a majority of acceptors accepted it. However, assume that both acceptors crash while trying to inform the learners about their decision. To make things worse, $P1$ crashes in the same moment. Due to a temporary network failure, $A3$ does not receive $P1$'s message, it stays alive but knows nothing about what happened.

$P2$ takes over the leadership: since it uses a learner, it knows that instance up to $i-1$ are already closed. It executes phase 1 for instance i. However, since not enough acceptors are responding to its requests, phase 1 keeps expiring; $P2$ can do nothing but increment the ballot and keep trying. After some time, acceptor $A1$ recovers. A majority of acceptors is now online, progress is (eventually) granted. The next phase 1 executed by $P2$ receives two acknowledgements and the one from $A1$ contains value v_i. $P2$ is forced to execute phase 2 using v_i rather than any other value. $A1$ and $A3$ grant the request, they accept, log to stable storage and inform the learners, which can deliver value v_i for instance i.

No matter how tragic the scenario is, the safety of the protocol reduces to a simple fact: if a majority of acceptors accepted the same request, an instance is closed. Any proposer trying to do something for that instance will realize during phase 1 that there is a value already. It is then forced by the protocol to *help* with the current situation rather than try to push a different client value.

Chapter 3

LibPaxos: an open-source Paxos implementation

3.1 Implementing PAXOS

Although the laws governing a PAXOS network are very simple, translating the algorithm into a high-performance, reliable implementation proves to be non-trivial, as observed by others before us [CGR07]. In the past two years we developed a number of PAXOS-based systems, exploring different design alternatives with the objective of creating the ultimate high-throughput PAXOS implementation. The following projects are currently published as open-source under the name of *LibPaxos*[1].

ErlangPaxos: We think the Erlang programming language can be a powerful tool for prototyping of distributed algorithms. Among other features, it provides network communications and failure detection directly embedded in the language. For this reason, our first, exploratory implementation was written in Erlang. ErlangPaxos is structured like a PAXOS "simulator" governed by a shell that allows to inject specific events (message loss, process crash, network partitioning, etc.) and track reactions of the participants to them. This proved to be extremely useful for understanding the dynamics of the protocol and debugging it.

libpaxos-T: Our first attempt to produce a high performance library to be used within prototypes of distributed databases for clustered environments. The system is written in C and uses the *pthread*[2] library for concurrency. This proved to be a major problem since the time cost of pthread calls varies greatly across different systems, making the library occasionally unpredictable. The same is true when using UDP multicast as the communication method: performances are sensibly influenced by the operating system network stack and by the capacity of the underlying network switch.

---

[1] `http://libpaxos.sourceforge.net/`

[2] `http://en.wikipedia.org/wiki/POSIX_Threads`

libpaxos-E: Motivated by the fact that PAXOS is clearly the bottleneck in our distributed database prototype, we decided to restart from scratch and design a better structured library. We dropped threads in favor of an event-based model, using *libevent*[3], a library that allows to handle heterogeneous events (network traffic, timeouts, periodic alarms) with ease. The resulting code is easier to extend and modify to benchmark advanced optimizations. The system is complete with a simple leader election service (which can be replaced). We hope it will become mature enough to be used out-of-the-box in production environments.

libfastpaxos: a UDP Multicast, libevent-based implementation of the FAST-PAXOS algorithm [Lam06]. Implements the basic protocol without sophisticated optimizations. We built this library to better understand the loss in performance expected with growing number of client-proposers.

3.2 Design choices

Implementing PAXOS requires addressing a number of practical issues that are abstracted in the algorithm, which has to stay simple to be provably correct. Many design choices do not become evident until one hits against them, requiring to modify part of the system. For this reason one should consider those issues in advance and account for them at design time.

3.2.1 Network model

The single most influential choice when creating a PAXOS system, is probably the type of communication primitive used. Our implementations are based on UDP multicast, except for ErlangPaxos which abstracts the network layer. The advantage of Multicast is simplicity. If acceptors subscribe to a given address/port pair, then to send a message *to all* of them it is sufficient to send a message to that address. In a connection-oriented network this is more complicated since processes may leave, and come back or move to a different hosts. Multicast also pushes the cost of sending multiple copies of a message down to the network switch, since the sender calls `send` only once.

There are two major drawbacks of using UDP: the message size is limited by the MTU of the host OS and network switch, therefore there is a bound on the size of values that clients can submit. The second constraint is the performance of the network equipment when delivering high throughput multicast traffic. In particular, we found the behavior of the switches used in our experiments to be very unpredictable when multiple senders produce high volumes of traffic.

---

[3] `http://monkey.org/~provos/libevent/`

Many of the design choices in the rest of this document are based on the use of multicast, they may not work well in connection-oriented scenarios.

3.2.2 Client model

Depending on how PAXOS is used by the application on top of it, one has to make important design choices regarding clients/proposers interactions.

Client-proposers vs. clients and proposers

Should each client that wants to broadcast *be* a proposer? While this may be the case for a FASTPAXOS implementation (thus allowing delivery in only two message delays), we do not think it is a good choice in general. Making proposers and clients two disjoint sets of processes allows simpler management, for example, concerning leader election. Clients are free to join, leave and crash without interfering with PAXOS itself. This however introduces the cost of one more message delay for each value.

To submit values, clients connect to the current leader. In case the coordinator crashes, they connect to the next elected leader. Alternatively, clients can send their values by multicasting them on the "leader network" (an address where the current leader is listening to). A coordinator crash is completely transparent to them in this case.

Semantic of ABroadcast

Another important question is about the guarantees provided by the ABroadcast call invoked by clients. For example, should the call return only once the value has been delivered? Or can it return immediately, without any guarantee, allowing to submit multiple values concurrently?

Creating an ABroadcast call that returns exactly the state of the operation, either success or failure, is very difficult or even impossible. Consider a client that connects (e.g., using TCP) to the coordinator to submit a value. The best that proposer can do is send to status updates as the value is used. It starts by saying "Value received", then "Value submitted in phase 2" and hopefully then "Value delivered". However, what happens if the leader crashes at some point during this procedure? The clients sees the connection falling, but has no way of knowing if the value reached the acceptors and will be delivered eventually.

We think is better for different reasons to build a submit function (ABroadcast) that has weaker guarantees. This approach is the most general. The clients can decide what to do, like keeping trying to submit a value until it is delivered, risking to deliver it twice, or simply forgetting about it if it did not go through the first time (the client may need to use a learner to know about delivered values).

In case the application using PAXOS has strict ordering guarantees, e.g., it requires FIFO order with respect to each proposer, it is quite easy to enforce those requirements on top of the the Atomic Broadcast layer, for example by embedding a vector clock[Lam78] or a ⟨client ID,sequence number⟩ pair directly into the value.

Allowing each client to submit multiple values is also a key ingredient for more efficient network usage, allowing many optimizations described in the rest of this document.

3.3 Implementing the learner

In libpaxos-E, the learner is used by clients, but it's also used as a building block to implement proposers and acceptors. Therefore it needs to be flexible enough. In our case, it is started by passing 2 arguments: a *custom initialization function* and a *deliver callback function*. The learner starts the main events loop. Passing a custom initialization allows to add events to it (i.e., create a periodic callback, set timeouts, listen to a network socket and react accordingly). The deliver callback instead, is invoked whenever the current instance is closed, as previously described, it has the final value chosen for that instance as argument.

The learner is organized roughly as follows. Initially it sets the current instance number (lower instance not yet delivered) to 1, then it starts listening on the "learners network", where learn messages from acceptors are received. Those messages are stored in a structure indexed by the instance ID.

When the current instance is n, the learner may already know the value for some future instance $(n+1, n+2, \dots)$, without having n closed yet. Such a case is called a *hole* and requires it to take a special action, since the future instances cannot be delivered before n. The learner periodically checks for holes and asks the acceptors to retransmit their decision for the corresponding instances. Instance n is eventually closed and can be delivered, followed by $n+1, n+2, \dots$.

Once an instance is delivered, the current instance number is incremented and the learner can completely forget about the associated value, avoiding the need of infinite space over time. Allowing the learner to deliver out-of-order is a straightforward modification and does not require changing the hole-checking mechanism.

In some situations, a client using a learner may join the network after a significant number of instances have been already delivered. The default behavior for a learner is to fetch and deliver all such values by sending a number of repeat requests. Depending on the application, this may be useful or not. A learner should have an alternative startup method, which starts delivering from the present instance rather than starting from number 1.

Another ability an application may need from the learner is to fetch the value for some specific instance closed in the past. This can be easily achieved using retransmission requests.

3.4 Implementing the acceptor

Despite being the simplest component in the network, an acceptor can be tricky to implement, specifically because stable storage must be provided to it. This follows from the fact that an acceptor crashing must be able to recover its state entirely. Without doubts, this stable storage requirement is the factor with the most impact on performance. However, as we shall see later, there are ways to get around this issue.

In our prototypes, we use Berkeley DB [OBS99] as a stable storage layer. The code is made so that we can easily change the "durability mode": from no durability at all, to strict log-based transactional storage with synchronous writes to disk, passing from other intermediate setting.

In libpaxos-E, the acceptor starts on top of a learner, however it normally shuts down the learner functionalities (learning/delivering values), it only uses its event loop (one of the optimization described later on requires the acceptor to be a learner too). To this event loop the acceptor adds a periodic event which reminds to repeat its most recent (highest instance number) accept, this is useful to keep learners up-to-date in low-traffic situations. Other than that, the acceptor simply waits for requests from proposers (prepare/accept messages) and answers to them if the case. Any change to the state must be made permanent before sending the corresponding acknowledgement.

For practical reasons, acceptors need send one further type of message, a so called *reject*. An acceptor receives a promise that has a ballot too small, rather than just dropping the request silently, it informs the sender about the currently accepted ballot with a reject message. This allows the proposer to skip a few ballots ahead when generating the next request.

3.4.1 Snapshots

Over time, the space required to store the acceptor state grows to infinity. A solution to this problem, originally proposed in [CGR07], is to allow the application running on top of PAXOS to invoke a special routine: $snapshot(i)$. When receiving this command, each acceptor knows that it can safely drop informations about instances up to i, truncating their database. The semantic of this operation and its recovery procedure, in case of failure, depends on the application built on top.

3.5 Implementing the proposer

The proposer is clearly the most complex role among the three. Not considering disk storage, it is likely to be the next bottleneck in a PAXOS system, since all traffic goes through it.

As discussed previously, we think it is a good idea to have clients that send values to the proposers rather than having clients that *are* proposers; this allows to treat Atomic Broadcast as a standalone component. A proposer that is not currently coordinator can sit idle. However, nothing prevents it from listening to messages addressed to the leader (with multicast, those messages are received anyway). This may be helpful in case the leader election service nominates this proposer as the next coordinator. For the same reason, non-coordinating proposers may also listen for values submitted by clients.

The leader instead is busy broadcasting client values as fast as possible. When received, those values are temporarily stored in a *pending list*. A leader that simply executes one instance at the time, can be implemented almost directly by following the algorithm. However, if high performance is required, it is probably the case that multiple instances should be executed concurrently, which complicates the design significantly.

In libpaxos-E, the proposer starts on top of a learner. Its state consists of a window of instances. The lower bound for this window is the lowest instance not yet closed; the higher bound must be chosen so that it fits in memory. This structure can be easily implemented over a properly sized array, using instance number modulo array size to directly access the relative instance informations. The window slides forward when the internal learner delivers the next value.

Periodically, the Leader L should go over the current window and, based on the state of each instance, take the appropriate action. For example if instance i successfully completed phase 1 with a null value, the leader can pop the next client value from the pending list, assign it to i and execute phase 2. When timeouts occur, L should take appropriate action, namely retrying with a higher ballot for phase 1 timeouts, and restarting from phase 1 for phase 2 timeouts.

Trying to keep the proposer code as simple and close as possible to the algorithm specification is quite difficult, especially when multiple parallel instances are executed. Consequently it is more difficult to catch protocol violations that may be present in the code. An approach that we found extremely useful in this task is to model the life of any instance as a finite-state machine. This map makes it easier to verify that the appropriate action is always taken, since those are likely to end up scattered around in the code.

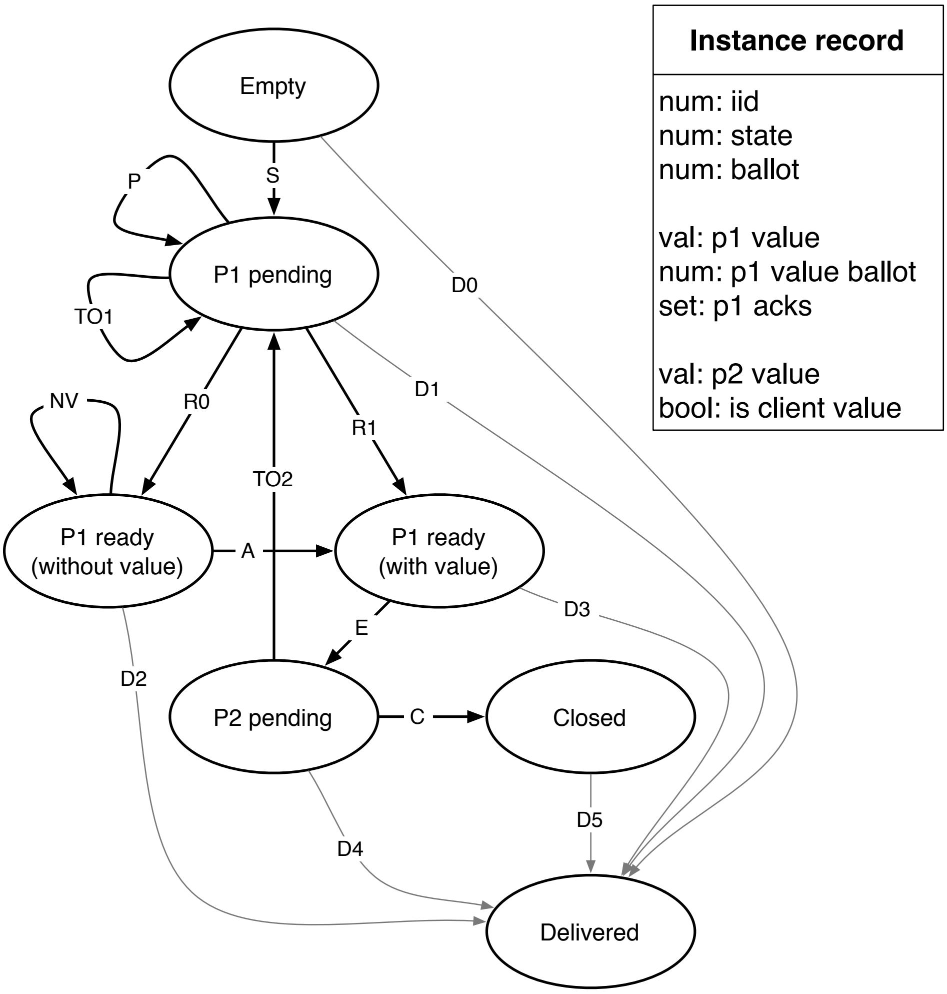
Instance record
num: iid
num: state
num: ballot
val: p1 value
num: p1 value ballot
set: p1 acks
val: p2 value
bool: is client value
Empty
S
P
P1 pending
TO1
D0
D1
NV
R0
R1
TO2
P1 ready (without value)
A
P1 ready (with value)
D3
E
D2
P2 pending
C
Closed
D5
D4
Delivered

Figure 3.1: The finite state machine depicting the states of any instance of the algorithm

The finite state machine in Figure 3.1 depicts the possible sequence of events and states that happen during the life of an instance. Each instance record is a tuple $\langle i, S, b, pset, v1, vb, pset, v2, cv \rangle$, where i is the instance number, S is a symbol representing a state (as they appear in the figure), b is a ballot, $pset$ is a set of promises received, $v1$ is the value found after a successful phase 1, vb is the corresponding ballot, $v2$ is the value to use for phase 2, cv is a flag indicating wether $v2$ is a value received from a client. Each instance is initialized as $\langle 0, empty, 0, \{\emptyset\}, null, 0, null, false \rangle$. We now go through the possible state transitions.

- **S**: the proposer executes phase 1 for the first time in this instance, numbered I. It generates its first ballot. For example if the proposer ID is 2, the first ballot b is 102. After sending the prepare request to the acceptors, the record is updated to $\langle I, \mathbf{p1\_pending}, \mathbf{b}, \emptyset, null, 0, null, false \rangle$. The proposer sets a timeout for this instance.

- **TO1**: the timeout for this instance is expired (a majority of promises were not received). The proposer increments the ballot, sends the prepare requests, clears the promises and sets the timeout. All other fields are not relevant. The updated record consists of $\langle I, p1\_pending, \mathbf{b{+}{+}}, \emptyset, \mathbf{null}, \mathbf{0}, v2, cv \rangle$

- **P**: the proposer received a promise p_a from some acceptor a. The ballot number must match the one stored, otherwise the message is dropped. The record is updated to $\langle I, p1\_pending, b, \{\mathbf{p_a} \cup \mathbf{pset}\}, \mathbf{v1}, \mathbf{vb}, v2, cv \rangle$. $v1$ and vb are updated with the following rule: if p_a contains a value, and the corresponding value ballot is higher than vb, then $v1$ is set to the value in the promise and vb to its value ballot. If the size of $pset$ is now equal to the majority of acceptors, one of **R0** or **R1** is triggered.

- **R0**: a majority of promises was received, none of which contained a value. The record is updated to $\langle I, \mathbf{p1\_ready}, b, \{\emptyset\}, \mathbf{null}, \mathbf{0}, v2, cv \rangle$. If $v2$ is non-null, this proposer previously assigned a client value to this instance, action **A** is triggered. If $v2$ is null, the proposer assigns to it the next element from the pending list of client values and sets cv to true. Then **A** is executed.

- **NV**: phase 1 was completed, $v2$ is null and the pending list happens to be empty. This instance is not used until a value is submitted by some client.

- **A**: some client value $v2$ was assigned to this instance, the record consists of $\langle I, p1\_ready, b, \{\emptyset\}, null, 0, v2, cv \rangle$. Phase 2 can start with action **E**.

- **R1**: a majority of promises was received, at least one of them contained a value, $v1$. Different cases are possible:

  - **v2 is null**: update the record to $\langle I, p1\_ready, b, \{\emptyset\}, null, 0, \mathbf{v1}, \mathbf{false}\rangle$. We assigned no client value to this instance. Phase 2 is executed with the value, among the promises, that has the highest value ballot.
  - $\mathbf{v1} = \mathbf{v2}$: update the record to $\langle I, p1\_ready, b, \{\emptyset\}, null, 0, v2, cv\rangle$. The value found in phase 1 is the client value that we assigned to this instance (this can happen after a phase 2 timeout).
  - $\mathbf{v1} \neq \mathbf{v2} \wedge \mathbf{cv} = \mathbf{true}$: push back $v2$ to the head of the pending list, then update the record to $\langle I, p1\_ready, b, \{\emptyset\}, null, 0, \mathbf{v1}, \mathbf{false}\rangle$. While trying to send a client value for this instance, we discovered another value and we must use it. The client value will be sent in another instance.
  - $\mathbf{v1} \neq \mathbf{v2} \wedge \mathbf{cv} = \mathbf{false}$: discard the current $v2$, then update the record to $\langle I, p1\_ready, b, \{\emptyset\}, null, 0, \mathbf{v1}, \mathbf{false}\rangle$. We had some non-client value for this instance, but another value came along and replaced it.

  In all above cases the result is the same: the value found in phase 1 is used to execute phase 2, by triggering action **E**.

- **E**: the proposer executes phase 2 with ballot b and value $v2$. The record is updated to $\langle I, \mathbf{p2\_pending}, b, \{\emptyset\}, null, 0, v2, cv\rangle$. A timeout is set.

- **C**: this action is triggered if the proposer realizes that instance i is closed (i.e., by querying its learner) with any value. The record is updated to $\langle I, \mathbf{closed}, b, \{\emptyset\}, null, 0, v2, cv\rangle$. This instance can be ignored until it is delivered (**D5**).

- **TO2**: the timer for phase 2 expired. It is necessary to restart from phase 1. The record is updated to $\langle I, \mathbf{p1\_pending}, \mathbf{b{+}{+}}, \{\emptyset\}, null, 0, v2, cv\rangle$.

- **D0, . . . , D5**: at any time during the life of the instance, something else may happen, the learner may kick-in and deliver some value v' for it. Maybe that value got accepted "slowly" making the proposer timeout, maybe another proposer is sending values. Independently of the current state, the event is handled with the following rule:

  - $\mathbf{v}' = \mathbf{v2} \wedge \mathbf{cv} = \mathbf{true}$: our client's value was delivered, inform it if the case.
  - $\mathbf{v}' \neq \mathbf{v2} \wedge \mathbf{cv} = \mathbf{true}$: push back $v2$ to the head of the pending list, since we could not deliver it in this instance.
  - Any other case requires no action.

This finite-state machine approach allows to create a test suite to verify that the proposer takes the appropriate action for every $\langle state, event\rangle$ that can happen. Given that the most modern techniques for formally verifying code cannot cope directly with thousands lines of C code, this is probably as close as one can get to showing the correctness of a particular implementation with respect to the algorithm description.

3.5.1 Pending list size

In case the submit function used by the clients is non-blocking, special care needs to be taken when handling the list of pending values of the leader proposer. For example, assume C is a client that acts in the following way: it submits ten values and sets a timer for each one of them. If a timer expires, C tries again to submit that value. The client is also a learner. When one of its values is delivered, C deletes it, generates a new value and submits it, setting the corresponding timeout. In other words, at any moment, C is trying to push ten distinct values. Assume the timeout for values used by C is 1 second, the average delivery rate of the underlying PAXOS network is 1 value per second. At time 0, C submits; the pending list size of the leader contains 10 values. After 1 second, 1 value was delivered, however 9 of C's values timed-out and were re-submitted. The pending list size is now 18. After another second, another value is delivered, while 9 others timed-out, and so on.

A simple way to solve this issue could be to have the leader checking that a value is not already in the pending list before adding it. This solves the issue in the case described above, but not in the general case. In fact if C generates a different value once an old one times-out, we have the same effect: the leader pending list grows too quickly for the capacity of the network.

A more general solution is to limit the maximum pending list size. Values that don't fit are simply ignored. The client will retry to submit them later on and eventually succeed.

3.5.2 Ballots and rejects

From the PAXOS algorithm description, ballot numbers generated by proposers must fulfill the following requirements:

- Two distinct proposers never generate the same ballot for the same instance.
- The operation *increment* must be defined for any ballot (we use b++ to denote the result of incrementing ballot b).

- The binary operations *greater than* and *equal to* must be defined for any two ballots.

In our implementation we use the following scheme: fix the maximum number of proposer n as a power of 10, i.e., 100. Each proposer is given a unique ID in the range $0 \ldots n-1$. Proposer p's first ballot number for each instance is created as an integer $n+p$ (i.e. 103 if p is 3). Incrementing ballot b is straightforward: $b{+}{+} \equiv \ (b+n)$ (i.e., $103 \rightarrow 203 \rightarrow 303 \rightarrow \ldots$).
When receiving a reject message containing the current ballot "to beat" b', the proposer can specifically generate the next one rather than just incrementing the current b:

Algorithm 1 IncrementSkip(b, b')

```
$my\_seq = b - p$
$b'\_owner = b'\ mod\ n$
$b'\_seq = b' - b'\_owner$
$use\_seq = max(my\_seq, b'\_seq)$
$use\_seq\ += 1$
**return** $(use\_seq + p)$
```

3.5.3 Calibration and event counters

Two important parameters for the proposer are the time intervals used for phase 1 and phase 2 timeouts. An interval too small may time-out before the acceptors have the chance to send the answer; an interval too large may make the proposer less reactive. There is no magic number to pick those values, since they depend almost entirely on *(i)* the disk latency of acceptors and *(ii)* the network latency.

For example, in some of the libpaxos-E experiments that we run, we set the phase 1 interval in the order of seconds when running using the acceptor in strict durability mode. If instead we use a non-durable setting, the interval is set to a few millisecond.

It is therefore essential to calibrate those parameters once the system is deployed. A simple and effective way to do this is to count the number of phase 1 and phase 2 timeouts.

We start by setting both intervals to a very large value, and putting the system under heavy load (with clients sending random values). Very few or no timeouts should happen. Then we keep decreasing the time interval until only a few timeouts occur. Although this is not the case in any of our implementations, one could use those counter to dynamically increase or reduce the timeout intervals at runtime.

In libpaxos-E, the proposer includes a simple framework for events counting, which is useful to calibrate those and other parameters described later on in this document. Count reports are periodically logged if the proposer is the current leader. Events that we find useful to monitor include:

- Values dropped from the pending list of the leader. An indicator that clients are submitting at a rate too high for the proposer to digest.

- Number of times the leader cannot execute phase 2 for a new instance because phase 1 is not completed yet. This value grows quickly if the phase 1 pre-execution window of the leader is too small. Pre-execution of phase 1 is an optimization described in section Section 3.7.1.

- Number of retransmissions requested or sent (in the learner and in the acceptor respectively). Useful to quantify the loss of messages in the network if a non-reliable protocol is used (e.g., UDP).

3.5.4 Memory allocation strategies

At any moment in time, a proposer is only interested in a subset of all instances. It does not care about instances with number lower than the highest one delivered, since those are closed and there is nothing to do about them. Neither the leader is interested in instances too far in the future, since before delivering those, the ones in between must be closed. There is therefore a "window" of currently relevant instance numbers that starts from the current iid (the lowest not closed/delivered). This windows slides up of one position whenever the next value is delivered.

Exploiting this fact allows to map the conceptually infinite array of instances over a fixed size array that can be pre-allocated. For example, if the array holds 100 instance slots, the maximum window size is 99. The leader must not try to access instances already delivered or higher than the lowest not delivered plus 99. Instances in the window are found with a direct memory access in the array at position $iid \pmod{100}$. In languages like C and C++, this simple trick can save a lot of memory allocations, which are costly and bug-prone. In languages like JAVA this mechanism helps the garbage collector by cutting all references from the root set.

A similar strategy is used to implement the learner.

3.6 Handling Failures

The PAXOS algorithm tolerates any number of process failures, provided that those are crash-stop (crash-resume for acceptors). In our implementation, we try to model this behavior by "mining" the code with runtime validity checks. If one of those assertions fails, the actor immediately shuts down. It is by far simpler to restart some process from a valid known state, even if it is the initial one, than modify the system to handle unexpected incorrect behavior.

3.6.1 Acceptor Failures

Particular care should be taken of the acceptors, since the system cannot progress unless a majority of them is alive. A practical way to do that is to wrap the process or monitor it. When a failure is detected, this monitor should restart the acceptor in recovery mode rather than from scratch. The time required by an acceptor to recover may depend on the size of data stored and therefore by the number of values delivered so far.

3.6.2 Learner Failures

When crashed, a learner can be restarted "clean", that is, with an empty state. After that, it eventually closes some instance numbered i, just by listening to the learn message that it receives. When this happens, the learner realizes that it has to fill a hole that goes from instance 1 to instance $i-1$. It does so by sending repeat requests. Later on we discuss some optimizations to make this procedure faster.

Depending on who is using this learner however, the above procedure could be totally irrelevant. We think the learner has to have a second initialization procedure that simply ignores instances up to $i - 1$. The instance number from which to start can be forced or chosen automatially, i.e., "start delivering from instance I" or "start delivering from the lowest you can close right now".

For example, in libpaxos-E, when a proposer crashes and restarts, it does not care about values delivered in the past. The only relevant information is the number of the highest instance already delivered, which is used when the proposer becomes a leader.

3.6.3 Proposer Failures

Handling proposer failures depends very much on the semantic of the submit function used by clients. If submit has no guarantee of success, then the failure of the leader is transparent to the clients (like in libpaxos-E), no special action needs to be taken.

The proposer can simply be restarted by its wrapper/monitor process. It instructs the internal learner not to go over the previously delivered values. This is because the only relevant information to a non-leader proposer is the number of the highest delivered instance. In fact when a proposer is promoted to leader, it starts phase 1 for instance (highest delivered + 1).

A leader crash must be detected promptly by the failure detector, which nominates the next coordinator. The leader election service may also decide to give the leadership to some process even if the current leader did not crash. This means that all proposers should be always ready to switch back and forth between "idle" and leader mode.

3.7 Optimizations

When implementing PAXOS, there are a number of practical tricks that are essential for improving performance. Some of those are well known in the literature, some others are specific to our implementations.

3.7.1 Phase 1 pre-execution

A proposer should only execute phase 2 for an instance after completing phase 1. However nothing prevents from executing phase 1 "in advance", that is, without waiting the next value from the client[Lam98]. An instance whose phase 1 is completed can be used later, provided that the current leader does not crash. Therefore the coordinator can pre-execute a large number of instances. When values are submitted, phase 2 starts immediately, saving the two message delays required by phase 1.

For maximum throughput, this pre-execution window should be large enough so that, when receiving a value to broadcast, the leader should never execute phase 1. For calibrating this value we find useful to count the number of times the proposer cannot directly jump to phase 2. If the pre-execution window is large enough, this counter stays 0 for the whole execution.

Notice that this shortcut can be used only the first time phase 2 is executed in an instance. In case of timeout, the proposer is forced by the protocol to go through phase 1 again.

3.7.2 Parallel Phase 2

Unless the broadcast requires some strict ordering guarantees (i.e., FIFO), two instances are completely independent. Therefore the leader can concurrently execute multiple phase 2 with different values.

This form of pipelining can increase throughput tenfold, but it requires some special attention. For example, the leader should not assign a value from the pending list to instance i unless all instances from the lowest not delivered up to $i-1$ already have a value assigned. Otherwise the network may get "stuck" since i can be delivered but $i-1$ is not closed simply because there is no value to send for it.

Notice that FIFO order cannot be granted even with respect to a single client, since the leader pops different items from the pending list but it may be forced to re-enqueue them in a different order.

In libpaxos-E, the maximum number of instances to execute in parallel can be set to 1: no concurrency. This ensures FIFO ordering with respect to each client.

3.7.3 Message batching and compression

Sending a single message for each request or acknowledgement is a waste of network resources, especially since messages not containing a value are small (i.e., a prepare request contains just 2 numbers). In each packet there is room for multiple PAXOS messages batched together.

Combined with the two previous optimizations, batching allows the leader to execute phase 1 for hundreds of instances with only a few messages. Not only this saves network bandwidth, it allows the acceptors to process different requests and update their state in stable storage with a single atomic operation, requiring a single (larger) disk write.

In our implementations, we batch only messages of the same type, however different kinds of messages can be batched too if they have the same recipients (i.e., prepare and accept from leader to acceptors). To further reduce traffic, messages can be compressed before being sent.

3.7.4 Value batching

To digest client values faster, the leader can use a single instance to broadcast different values. This is done by popping different values from the pending list and sending them as one composite value. At the other side of the system, the learners unpack the batch and deliver the originally submitted values.

3.7.5 Learning acceptors

If each acceptor is also a learner, it eventually knows the final value chosen for each instance.
Storing this final value and refusing to modify it later, does not violate the protocol. Furthermore it speeds up the recovery procedure for learners lagging behind. When answering to accept and repeat requests, the acceptor can set a flag implying that the instance is closed with that value. When receiving such an answer, a learner does not need to wait for a majority, it can deliver immediately that value.

This update to the state in the acceptor does not require an immediate commit to stable storage.

3.7.6 Relaxed stable storage

In a PAXOS implementation, acceptors must be able to recover their state entirely after a failure, therefore they synchronously write to disk before acknowledging any valid request. Even adopting a smarter policy for such writes (i.e., batching many of them into a single atomic operation), still involves I/O that requires tens of milliseconds to complete.

An acceptor that crashes either recovers its state entirely, or it is not allowed to re-enter the system. Therefore if we make their storage volatile, failed acceptors should not be restarted.

By making each acceptor a learner, and assuming that at least one up-to-date acceptor-learner is alive at any moment, we can relax the durability requirement. If it is feasible to assume that either *(i)* a majority of acceptors does not crash simultaneously, or *(ii)* at least one up-to-date learning acceptor does not crash, then it is possible to weaken the durability mode required. The storage can be made volatile and thus much faster.

Crashed acceptors are not allowed to re-enter the network, but rather than restarting them we take a different approach. When some number (strictly less than a majority) of failures is detected, the leader should stop opening new instances. A snapshot is taken to ensure all learners delivered all values decided so far. The system can be then entirely restarted, with the initial number of acceptors.

3.7.7 Selective quorums

If strict durability is required, we can use selective quorums to spread the cost of disk writes across different acceptors.
For example, a system with 6 acceptors is divided into two groups of 3 each, $G1$

and $G2$. Acceptors in $G1$ ignore all requests regarding odd-numbered instances, considering only even-numbered ones. Processes in $G2$ do the opposite. The majority for any instance consists of 2 acceptors.

This approach allows better parallelism and reduces the latency introduced by disk writes. In this case, for example, two instances i and $i+1$ can be closed with a single disk delay.

Chapter 4

Performance evaluation

In this section we perform different experiments aimed at better understanding the runtime behavior of libpaxos-E. Notice that libpaxos-E only recently completed development, therefore the objective here is not to simply measure its performances, rather it is to understand in order to improve. For example with libpaxos-T, the predecessor of libpaxos-E, it took us different months of targeted benchmarks and usage within other systems before reaching stable, reproducible results and understanding which were the true bottlenecks.

4.1 Experiments setup

4.1.1 Infrastructure

The experiments described in the rest of this section were performed within a cluster of 16 Apple Xserve G5 RackMac3,1 machines running Mac OS X Server 10.4.11. Each machine has two PowerPC G5 2.3GHz CPUs and 2GB of RAM. Nodes are connected through a GigaBit ethernet 1000baseT dedicated switch. The maximum MTU size is around 8 Kbytes; client values are therefore limited to less than 7 Kbytes. Round-trip times for ping among nodes are around 0.2 milliseconds. Unless otherwise specified, each process (including clients) runs on a different machine.

4.1.2 Benchmark client

Client processes used in our experiments behave in the following way. Values are randomly generated, the minimum and maximum allowed size is a parameter. A number n of values (30 by default) is submitted asynchronously. Each client starts a learner in order to be notified with broadcast values. When a delivery occurs with value v, the client checks to see if v belongs to the set of submitted values. If so, v is deleted from the list, a new value v' is created and sent to the leader. In other words, at each moment in time each client tries to broadcast n different values. If a value submitted is not delivered in a few seconds (3 by default), the client will re-send it. Each client stores the turnaround time for each value, which is the time from the first submission to the delivery of that value.

Another kind of client which does not submit values but only listens for them is used to measure throughput in values per second and bytes per second.

4.2 An ideal Atomic Broadcast

In order to better understand the performance of our implementation, we try to create an ideal Atomic Broadcast to compare against. For this purpose we use a *serializer* node (sometimes known as a *sequencer*). The task of this process is listening to client values, assigning each one of them to a different instance and broadcasting them to the learners (with a special flag specifying that this is the final value). This node can alone implement Atomic Broadcast in only two message delays, but has of course to make stronger assumptions than PAXOS. If the node stops, the system stops, and if the node loses data there is no way to recover it.

The serializer is compatible with clients and learners of libpaxos-E, therefore our benchmarks for the two are identical. Moreover it uses the same stable storage layer used by acceptors, allowing different kinds of durability.

4.3 Example experiment

To show the procedure behind each of our benchmarks, we start with a simple test. The objective is to maximize throughput in a network with a single client sending values.

Setup: A client submits values for 5 minutes, at most 30 are sent concurrently. The size of values is random between 20 and 2000 bytes. The stable storage is configured to be non-durable (we assume no failures). PAXOS runs with 3 acceptors.

It takes different runs to calibrate the library parameters for the best performance, specifically the timeout intervals need to be adjusted iteratively, observing the event counters. Before considering an experiment "valid" we execute it different times to ensure it is repeatable and not just a case.

	Paxos	Serializer	unit
Average delivery rate	5278	9614	val/s
Average delivery rate	5323	9523	kB/s
Minimum Latency	0.56	0.51	ms
Maximum Latency	6552.82	4795.14	ms
90% C.I. Latency	[1.72, 5.38]	[2.11, 2.76]	ms

Table 4.1: Comparison of serializer against PAXOS. Best throughput obtained with a single client sending multiple values concurrently.

Table 4.1 presents the best results obtained in terms of throughput for both PAXOS and the serializer. Although the serializer runs faster, this experiment is not a good indicator since a single client is unlikely to produce enough data to saturate the broadcast. In a high-traffic situation, we expect the serializer to beat PAXOS by some orders of magnitude.

More interesting in this case is the latency. For example, in the best case, (minimum latency) the difference between the two broadcasts is extremely small. As one may expect, the maximum latency is worse for PAXOS, which may have to re-execute an instance from phase 1 in case of timeout. This is also evident from the latency confidence interval (the last row in the table). This value is obtained by randomly sampling the latency of values in one of the clients. We think a confidence interval is a much more valid indicator than the mean latency, mainly because the geometric average is not robust and greatly influenced by outliers in the sample population.

4.4 Impact of acceptors durability

We want to quantify the performance penalty of enforcing strict durability in the acceptors' stable storage layer.
Setup: We use the same workload as in the previous experiment, consisting of a single client sending different small values concurrently. PAXOS runs with three acceptors. We try different durability settings for them and the serializer:

- *ND*: No durability, the acceptors write to disk only when the memory cache is full and some database page need to be swapped out.

- *TR*: Default transactional mode for Berkeley DB, it wraps state modifications into atomic operations logged to disk before proceeding. This setting ensures durability in case of process crash but not in case of an OS failure, since disk writes may be buffered by the disk driver.

Paxos				
	ND	TR	BD	unit
Avg. Delivery rate	5234	55	48	val/s
Avg. Delivery rate	5439	58	49	kB/s
90% C.I. Latency	[1.67, 5.02]	[211, 259]	[311, 350]	ms

Serializer				
	ND	TR	BD	unit
Avg. Delivery rate	9457	109	71	val/s
Avg. Delivery rate	9820	113	73	kB/s
90% C.I. Latency	[2.22, 2.91]	[119, 132]	[147, 169]	ms

Table 4.2: Comparison of durability methods for Paxos and the serializer. Strict durability (*TR* and *BD* columns) has a significant impact on delivery rate.

- *BD*: Durability is enforced by "manually" forcing a full database synchronization on disk before answering any request.

The results in Table 4.2 clearly show how stricter durability negatively affects performances for both Paxos and the serializer. In the best case we have a 100x slowdown in the delivery rate and a corresponding increase of latency.
A positive result is that the performance loss of the serializer seems similar to the one of Paxos, meaning that the latter is not "more affected" by the cost of disk storage.

4.5 Impact of values size

So far, we considered clients sending mixed-size values of around a kilobyte (on average). This size turns out to be a very relevant factor for both delivery rate and latency. For example, small values may be delivered quickly since they require less bandwidth and less time to be stored. Larger values however make more efficient use of each multicast message sent.
Setup: We use a single client sending multiple values concurrently, the size of those is fixed in each of the experiments. The three Paxos acceptors and the serializer are configured for using non-durable storage. We start with small values, 30 bytes, and increase up to 6000 bytes, which is the maximum the network allows. Especially for Paxos, the network and the clients must be carefully calibrated to get the best performance. For each different value size we try to obtain the best delivery rate, then we fix the parameters and run different executions with those; each one lasts at least few minutes.

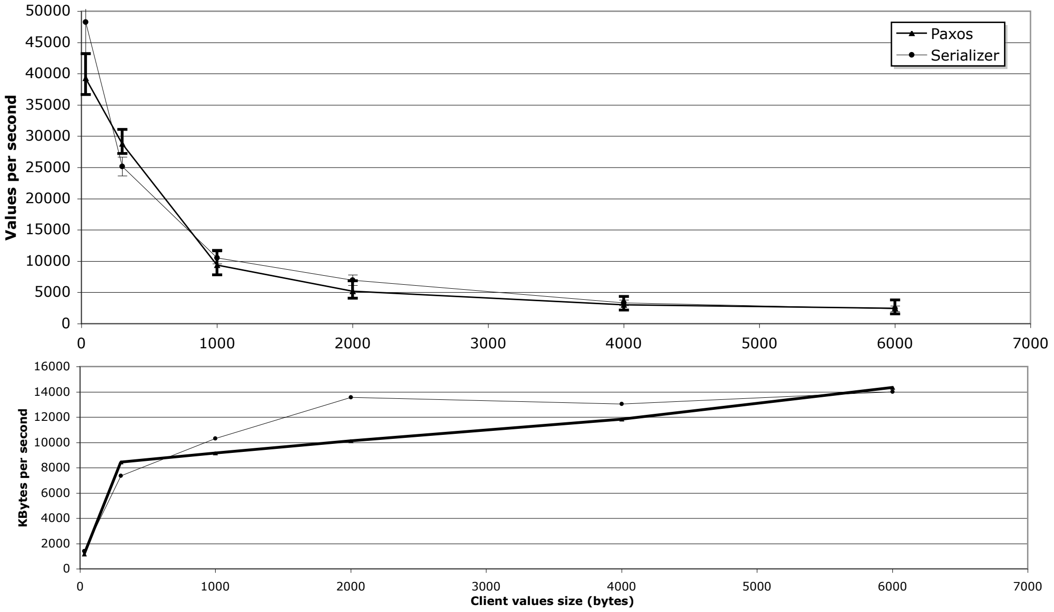

Figure 4.1: Delivery rate for different client value sizes in values per second and kilobytes per second. Higher is better.

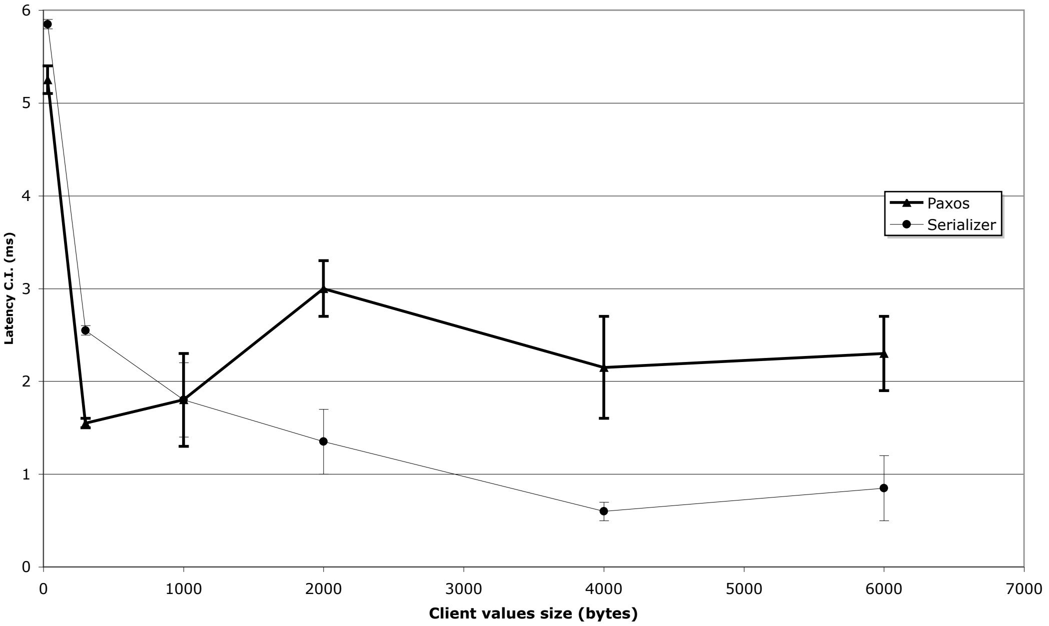

Figure 4.2: 90% confidence intervals of delivery latency for different client value sizes. Smaller is better.

In Figure 4.1 each point represents the average of the experiments with "good" parameters, the error bar display the best and the worst performance recorded with those settings. Figure 4.2 reports the corresponding latency, the error bar represent the bounds of the 90% confidence interval while the point is the middle value.

There are a few interesting remarks about this set of experiments. The first discovery is that, once calibrated, PAXOS performances are not that distant from that of the serializer, at least in terms of delivery rate. In fact for some sizes (300 and 6000 bytes), PAXOS is actually better than the serializer. Although we did not expect such a good result, it can be explained by the fact that the serializer acts very similarly to a single acceptor. Given that we are far from the throughput limit of the underlying switched network, learners can deliver more or less at the same frequency after a single message from the serializer or after 2 of the 3 messages from the acceptors.

Latency values are more than doubled for PAXOS, going through the proposer has clearly a cost. Very small values have in both cases high latency since many of them need to be sent by the client and batched by the proposer/serializer before filling a packet and flushing it.

The serializer performances are very stable, as visible from the (almost not visible) error bars. PAXOS results instead fluctuate up to 50% in some cases.

A relevant parameters for the performance of PAXOS is the number of instances that the proposer is allowed to execute concurrently. We notice that setting it to more than twice the number of values that should normally fit in a packet, dramatically reduces the throughput and increases the number of timeouts.

With both broadcast methods we notice that sometimes the experiments report very bad result (i.e., 10 or 100 times slower than other executions with the same parameters). Since this is unfrequent, we filter those executions and repeat the benchmark. This issue is further discussed in Section 4.6.1.

Despite the high delivery rate with small values, the quantity of data delivered is small, around 1 kilobyte per second, despite the high delivery rate. With larger values at a lower rate we instead reach 14 Mbytes per second. Batching values as described in Section 3.7.4 may therefore have a beneficial effect in networks where clients cannot aggregate small values themselves.

4.6 Scalability

Previous benchmarks were executed with the minimum possible number of processes. In this set of experiments we explore the cost of adding more, either to make the system more resilient or to allow more application processes to listen to the broadcast.

With this particular implementation, it is not worth to benchmark with different number of processes submitting values. Since ABroadcast just sends a multicast message on the "leader network", having a single client sending 10 values concurrently or 10 clients sending 1 value each is the same from the proposers point of view.

4.6.1 Multicast switching issues

We previously said that sometimes the network produces very bad results for configurations that, if just restarted, work properly. We suspected this was related to some learners lagging behind and asking for numerous retransmissions to the acceptors.

While running the set of experiments that we are about to present, this behavior becomes more and more frequent as the number of processes involved grows. After some other test aimed at observing the performance of multicast on this and other network switches, we have a more precise idea of what is causing the problem.

Multicast is built on top of UDP, which is a non-reliable transmission mechanism. When a process sends data, this is copied into an operating system buffer. From there it is copied to the network card and finally sent on the wire. At this point the operation is considered successful despite the fact the the datagram may never reach any of the receivers.

As long as a single process in the network is sending, the throughput reaches decent values. However adding more senders literally kills the performance, i.e., the decrease in bandwidth is not linear in the number of processes sending: if a single sender delivers regularly at 100 MBps, two senders running concurrently cannot reach 50 MBps each. This is the actual behavior on the switch used in our experiments. Messages loss rate rate is proportional to the loss in throughput, but is yet to be clarified if (for this particular network) messages are lost by all-or-none or different processes lose different messages.

As long as the network has only a few processes sending at a relatively low rate, the switch manages to push most of the packets through. When we run close to the capacity limit instead, Paxos is affected in a significant way. As soon as one of the learners loses some relevant message, the situation degenerates. The learner starts asking retransmissions, producing more traffic. The acceptors have to answer and produce even more traffic. The fact that they slow down may also result in timeouts in the leader which is forced to re-execute different instances. Of course this causes further message loss and yet more learners will need to ask retransmissions.

At the moment, we have no concrete solution for this issue. Since it depends on the capabilities of the underlying switch, it is difficult to solve in a general way. Maybe a benefit could be gained by introducing some network coding, such that processes can tolerate message loss since informations are repeated and spread across datagrams. Another approach may be to force the leader to slow down whenever some learner is lagging behind, however this is probably not good for scalability.

For the following experiments, we assume to have found a way to magically configure the switch for not losing messages. This means that we will discard executions in which the problems manifest itself. Unlike previously, this is much more frequent even when just adding two more acceptors. The values presented are therefore not very realistic since they are measured for extremely short executions (i.e., 60 seconds) and after different trials. Nevertheless they may provide an idea of how the system scales in a situation where the switch is well-behaving.

4.6.2 Number of acceptors

In Paxos, adding more acceptors makes the system more resilient. The number f of failures tolerated is the total number of acceptors minus a majority (i.e., $f = \lceil \frac{\#of\,acceptors}{2} \rceil - 1$). We would like to assess the cost of more resilience.

Setup: We use a single client proposing short values of 500 bytes each. We voluntarily keep the delivery rate well below the expected limit for this configuration, meaning that the client sends only a few values concurrently. We use the same parameters for executions with different number of acceptors.

The results in Figure 4.3 confirms what one may expect intuitively: the latency increases, as each learner needs to wait for more acknowledgements. Since those are sent concurrently from all acceptors however, the increase is only a fifth of a millisecond after tripling the number of acceptors and seems linear in the number of processes. The drop in throughput looks linear too but it is steeper. Moreover the network becomes less predictable. With 7 acceptors, for example, we record an execution that obtains more than 7000 values per second while most of the others are around half of that.

Unfortunately, we only have nine machines free in the cluster. Adding a tenth acceptor on a node that is also playing some other role (proposer, learner or client) would expose that machine to a traffic rate that is at least double to all other processes.

Acceptors are relatively lightweight on the CPU (i.e., during our benchmarks we observe they rarely use more than 5% of the CPU time). Nevertheless they do lots of I/O, a profiling of one of this processes in non-durable mode reveals that

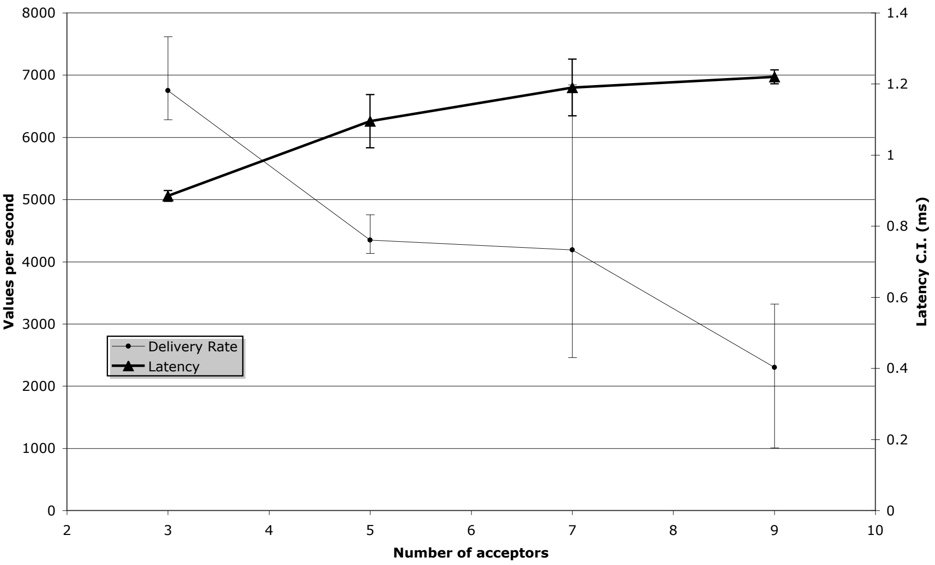

Figure 4.3: Delivery rate (left axis) and delivery latency (right axis) with different number of acceptors.

it is spending roughly 40% of the time in storage procedures and the remaining 60% almost exclusively in network related system calls. With stricter durability settings where synchronous disk I/O is required, the ratio becomes around 90% disk and 10% network.

We think the most relevant factor for adding more acceptors is the multicast capacity of the underlying switch. For leader and learners, having to wait for more messages is not a big penalty, however the network equipment must be able to digest lots of additional data without dropping too much packets, since also the recovery procedures are more costly.

4.6.3 Number of learners

The number of learners that a network can sustain is likely to be an important factor when running an application on top of PAXOS. Since in a switch packets are transmitted on all segments anyway, adding more listeners should come "for free" and the change is transparent to the rest of the network. As we already discussed in Section 4.6.1 this is not true in practice. At least in our case where more learners increase the probability of network congestions due to retransmit requests.

Setup: We use a single client proposing short values of 500 bytes each. The submit rate for this client is below the maximum expected capacity. We start with 3 learners (started respectively by the client, the leader proposer and the bandwidth monitor). Then we add other learner processes (with a delivery function that does nothing) on other cluster nodes. We only have 6 free machines so the test with 10, 15 and 24 processes are executed with more than a learner on the same node. Figure 4.4 confirms that, at least up to a certain point, adding more learners does not affect the network. The best single result for this set of benchmark is in fact recorded with 10 processes.

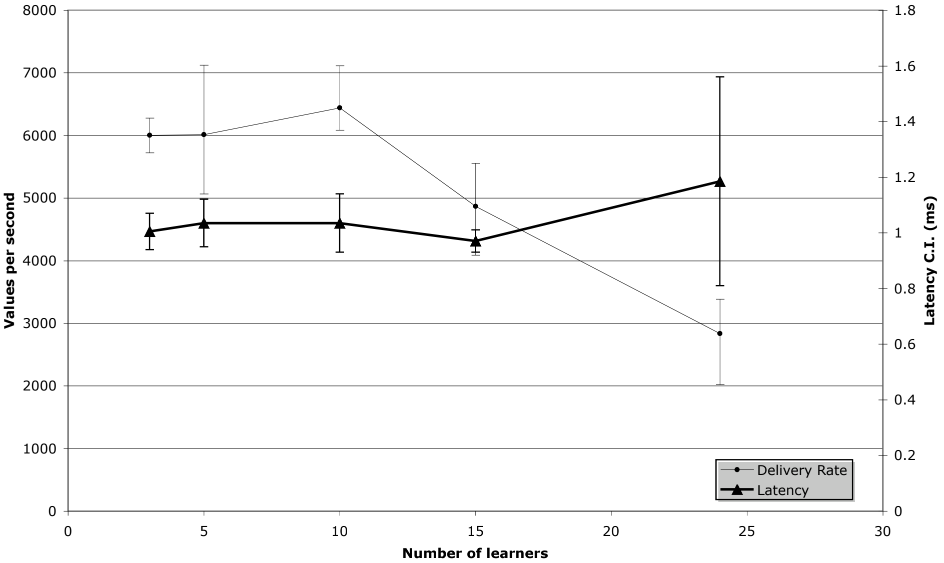

Figure 4.4: Delivery rate (left axis) and delivery latency (right axis) with different number of learners.

Learners are also quite lightweight on the CPU (3% to 5% usually), nevertheless we have the impression that the slowdown measured with more of them is related to the fact that they need to share a single machine. For example, the configuration with 24 learners has 4 processes on each one of the 6 spare machines. The number of retransmissions requested is higher in this case; adding yet another learner to one of those nodes jams the network after a few seconds.

4.7 Recovery times for proposers

In our implementation, proposers rely on an external failure detector to provide a simple leader election service. The latter is responsible of nominating the current coordinator. Strategies for electing the leader are out of the context of this document. What is relevant for us is that when a proposer is promoted, it has to

pick up where its predecessor left, as fast as possible.
Setup: We start a network composed of 3 acceptors, 2 proposers and a total of 4 learners. One of them logs delivery rate statistics every second. The client submits 10 values of 500 bytes each concurrently; if those values are not delivered in the next 3 seconds, it will try to resubmit them. We also run a leader election oracle in "manual" mode. This component normally decides to change the leader if a certain number of heartbeat messages are lost, but in this case it sticks to the first proposer until we command to switch to the other one.

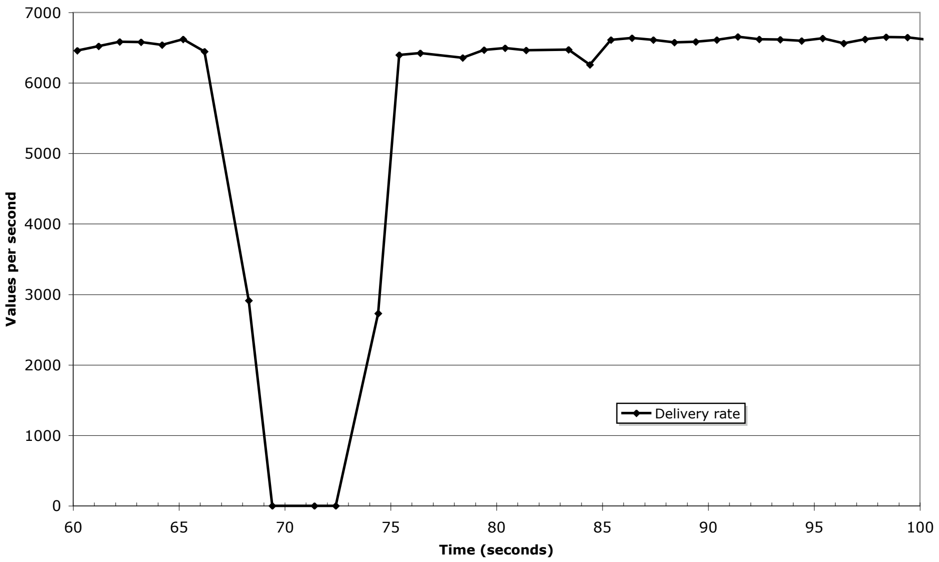

Figure 4.5: Delivery rate over time, around second 65, we force the leader election service to choose a new coordinator among the proposers.

Figure 4.5 depicts an example execution where we force the oracle to demote the initial leader. The other proposer is eventually notified of the change and begins pre-executing phase 1 for some instances, starting from the lowest one not yet delivered. By the time the client realizes the values timed-out and resubmits them, the new leader is ready and can immediately start executing phase 2 for different instances (the duration of the interval where the throughput is zero is exactly 3 seconds).

If the acceptors answer with *reject* messages to invalid proposals (phase 1b), the bootstrap time for the new leader depends solely on the number of instances to pre-execute. If rejects are not implemented, it may take variable time since the previous leader may have timed-out and incremented the ballot for those instances. The new leader has to increment and wait for the timeout at least the same number of times before finally producing a ballot that is acknowledged in phase 1.

Another technique for reducing this switch time is to keep track of submitted, not-yet-delivered values in proposers that are not the current leader. This way the broadcast can continue as soon as the new leader completes phase 1 for the next empty instance.

Chapter 5

Conclusions

5.1 Discussion

In this document we revisited the literature about PAXOS, collecting different efforts to make the algorithm more efficient. We then showed how to use those techniques in practice when creating an implementation.

Although we tried to be as general as possible when implementing libpaxos-E, it is evident for us that the application using PAXOS as an Atomic Broadcast primitive may require very specific features from it. It is therefore important to understand this requirements before selecting the algorithm (simple, fast, byzantine, etc.), the network layer, the semantic of the submit/deliver interface and finally the optimizations. The implementation should be the last step that almost directly follows from the above choices.

When creating the code, proper engineering methods should be used to ensure that the requirements are effectively granted at run-time. We did not stress this aspect which was already explored in [CGR07]. We also propose different benchmark cases that were useful for us to gain a better understanding of the system.

While using multicast has different advantages, like transparent reconfiguration and cheap delivery to multiple recipients, it also has a few problems and its behavior may depend on the network equipment.

5.2 Future work

There are two main areas which we think are worth further exploration. The first one is disk-based stable storage: there may be more efficient ways to ensure strict durability by better tailoring the persistence layer to the workload of the acceptors. The second one is performance of IP-Multicast, which proved to be very unpredictable in some situations. Both of those issues may significantly enhance the performance of a Paxos network. Most of the design issues addressed in this document are specific to LAN-based implementations using UDP Multicast. Using some other messaging scheme (i.e., TCP) may require a very different approach and yield to very different results.

Another issue we would like to deepen is using *LibPaxos* to provide Atomic Broadcast to a set of geographically distant processes over the internet.

Bibliography

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[CSP07] Lásaro J. Camargos, Rodrigo M. Schmidt, and Fernando Pedone. Multicoordinated paxos: Brief announcement. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 316–317, New York, NY, USA, 2007. ACM Press.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[GL03] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam96] B. W. Lampson. How to build a highly available system using consensus. In Babaoglu and Marzullo, editors, *10th International Workshop on Distributed Algorithms (WDAG 96)*, volume 1151, pages 1–17. Springer-Verlag, Berlin Germany, 1996.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

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

[Lam01b] Butler W. Lampson. The abcd's of paxos. In *PODC*, page 13, 2001.

[Lam02] Leslie Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[Lam06] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LM04] Leslie Lamport and Mike Massa. Cheap paxos. In *DSN*, pages 307–314, 2004.

[OBS99] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[PLL97] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the paxos algorithm. In *WDAG*, pages 111–125, 1997.

[RPSR03] Boichat Romain, Dutta Partha, Frolund Svend, and Guerraoui Rachid. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.

[Zie04] Piotr Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge, Computer Laboratory, 2004.