RAPPORT DE RECHERCHE

L R I

PRACTICALLY SELF-STABILIZING PAXOS REPLICATED STATE-MACHINE

BLANCHARD P / DOLEV S / BEAUQUIER J / DELAET S

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

08/2014

Rapport de Recherche N° 1573

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Practically Self-Stabilizing Paxos Replicated State-Machine

Peva Blanchard[1]*, Shlomi Dolev[2]**, Joffroy Beauquier[1]***, and Sylvie Delaët[1]†

[1] LRI, Paris-Sud XI Univ., Orsay, France
[2] Dept. of Computer Science, Ben-Gurion Univ. of the Negev, Beer-Sheva, 84105, Israel

Abstract. We present the first (practically) self-stabilizing replicated state machine for asynchronous message passing systems. The scheme is based on a variant of the Paxos algorithm and ensures that starting from an arbitrary configuration, the replicated state-machine eventually exhibits the desired behaviour for a long enough execution regarding all practical considerations.

1 Introduction

To provide a highly reliable system, a common approach is to replicate a state-machine over many servers (replicas). From the system's client point of view, the replicas implement a unique state-machine which acts in a sequential manner. This problem is related to the Consensus problem. Indeed, if all the replicas initially share the same state and if they execute the same requests in the same order, then the system is coherent from the client's point of view. In other words, we can picture the system as a sequence of Consensus instances that decide on the request to execute at each step. In an asynchronous message-passing system prone to crash failures, solving a single consensus instance has been proven impossible [10]. This hinders the possibility of a state-machine replication protocol.

Lamport, however, has provided an algorithmic scheme, namely Paxos [14,15], that partially satisfies the requirements of state-machine replication in the following sense. The safety property (two processes cannot decide to execute different requests for the same step) is always guaranteed. On the other hand, the liveness property (every non-crashed process eventually decides) requires additional assumptions, usually any means to elect a unique leader for a long enough period of time. Note that the original formulation [15] presented Paxos as a (partial) solution to the Consensus problem, but its actual purpose is to implement a replicated state-machine. Since then, many improvements have been proposed, e.g., Fast Paxos [17], Generalized Paxos [16], Byzantine Paxos [18], and the study of Paxos has become a subject of research on its own. The extreme usefulness of such an approach is proven daily by the usage of this technique by the very leading companies [5].

Unfortunately, none of these approaches deal with the issue of transient faults. A transient fault may put the system in a completely arbitrary configuration. In the context of replicated state-machine, the consequences (among many other unanticipated scenarios) may be the following: (a) the states of the replica are incoherent, (b) the replicas never execute the same requests in the same order, (c) the replicas are blocked even if the usual liveness conditions (e.g., unique leader) are satisfied. The issues (a) and (b) hinder the linearizability of the state-machine, whereas the issue (c) hinders the liveness of the state-machine.

A self-stabilizing system is able to recover from any transient fault after a finite period of time. In other words, after any transient fault, a self-stabilizing system ensures that eventually the replicas have coherent states, execute the same requests in the same order and progress is achieved when the liveness conditions are satisfied.

Nevertheless, completing this goal is rather difficult. One of the main ingredients of any Paxos-based replicated state-machine algorithm is its ability to distinguish old and new messages. At a very abstract level, one uses natural numbers to timestamp data, i.e., each processor is assumed to have an infinite memory. At a more concrete level, the processes have a finite memory, and the simplest timestamp structure is given by a natural number bounded by some constant $2^{\mathfrak{b}}$ ($\mathfrak{b}$-bits counter). Roughly speaking, this implies that the classic Paxos-based replicated state-machine approach is able to distinguish messages in a window of size $2^{\mathfrak{b}}$.

* blanchard@lri.fr

** Partially supported by Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences, Israeli Internet Association, Israeli Ministry of Science, Lynne and William Frankel Center for Computer Sciences, and Israel Science Foundation (grant number 428/11). dolev@cs.bgu.ac.il

*** jb@lri.fr

† delaet@lri.fr

This constant is so large that it is sufficient for any practical purposes, as long as transient faults are not considered. For example, if a 64-bits counter is initialized to 0, incrementing the counter every nanosecond will last about 500 years before the maximum value is reached; this is far greater than any concrete system's timescale. But, a transient fault may corrupt the timestamps (e.g., counters set to the maximum value) and, thus, lead to replicas executing requests in different order or being permanently blocked although the usual liveness related conditions (e.g., unique leader) are satisfied.

This remark leads to a realistic form of self-stabilizing systems, namely *practically self-stabilizing systems*. Roughly speaking, after any transient fault, a practically self-stabilizing system is ensured to reach a finite segment of execution during which its behavior is correct, this segment being "long enough" relatively to some predefined timescale. We give details in Sec. 2.

In this paper, we provide a new bounded timestamp architecture and describe the core of a practically self-stabilizing replicated state-machine, in an asynchronous message passing communication environment prone to crash failures.

Related work. If a process undergoes a transient fault, then one can model the process behaviour as a Byzantine behaviour. In [4], Castro and Liskov present a concrete[1] replicated state-machine algorithm that copes with Byzantine failures. Lamport presents in [18] a Byzantine tolerant variant of Paxos which has some connections with Castro and Liskov's solution. Note, however, that in both cases, the number of Byzantine processes must be less than one third of the total number of processes. This is related to the impossibility of a Byzantine tolerant solution to Consensus where more than one third of the system are Byzantine. The approach of self-stabilization is comprehensive, rather than addressing specific fault scenarios (risking to miss a scenario), and thus is somehow orthogonal to Byzantine fault tolerance. The issue of bounded timestamp system has been studied in [6] and [12], but these works do not deal with self-stabilization. The first work, as far as we know, on a self-stabilizing timestamp system is presented in [1], but it assumes communications based on a shared memory. In [2] and [9], the authors present the notion of practically stabilizing algorithm and provide an implementation of practically self-stabilizing replicated state machine using shared memory, and the message passing implementation of a practically self-stabilizing single-write multi-reader atomic register. Doing so, they introduce a self-stabilizing timestamp system. However, their approach assumes that a single processor (the writer) is responsible for incrementing timestamps and stabilization is conditional on such write executions. Our timestamp system is a generalization which allows many processors to increment timestamps. A first formulation of the present work has been given in [3].

The paper starts with a background and description of techniques and correctness in a nutshell. Then we turn to a more formal and detailed description.

2 Overview

In this section, we define the Replicated State-Machine (RSM) problem and give an overview of the Paxos algorithm. In addition, we give arguments for the need of a self-stabilizing algorithm that would solve the Replicated State-Machine Problem. Doing so, we investigate the recently defined kind of self-stabilizing behaviour, namely the *practically self-stabilizing* behaviour, and also briefly present the core idea of our algorithm.

Replicated State-Machine. Replicated State-Machine (RSM) aims at providing a reliable service to clients. From the client point of view, it is only required that the RSM acts as a correct sequential machine, and that every client request eventually gets a response. Formally, the problem is defined by the two following properties: *(Safety)* every execution yields a history of client requests and responses that is linearizable [11], *(Liveness)* in this history, every request has a corresponding response.

Original Paxos. Although the original Paxos algorithm [15] has been formulated as a (partial) solution to the Consensus problem, its actual purpose is to implement a RSM. Hence, in the following, our presentation of Paxos will include aspects related to the RSM problem.

The original Paxos algorithm allows to implement a RSM property in an asynchronous complete network of processors communicating by message-passing such that less than half of the processors are prone to crash failures.

[1] In their paper, "practical" is not related to our notion of practical self-stabilization.

Precisely, the safety of the RSM is always guaranteed, whereas the liveness is guaranteed if some conditions (e.g., unique leader) are satisfied. We refer to these conditions as the *liveness conditions*. The algorithm uses unbounded integers and also assumes that the system starts in a consistent initial configuration.

If it were possible to elect a unique leader in the system, then implementing a replicated state-machine would be easy: this leader receives the client requests, chooses an order, and tells the other processors. But, since the leader may crash (no more leader), and since it is impossible to reliably detect the crashes (many leaders at the same time), a take-over mechanism is required. To do so, the Paxos algorithm defines three roles: *proposer* (or *leader*), *acceptor* and *learner*.

Basically, a proposer is a willing-to-be leader. It receives requests from clients, orders them (using a step number s, natural number) and proposes them to the acceptors. The acceptor accepts a request for a step s according to some specific rules discussed below. A request can be decided on for step s when a majority of acceptors have accepted it in step s. Finally, the learner learns when some request has been accepted by a majority of acceptors for some step and decides accordingly. The learner has a local copy of the state-machine, and it applies the decided requests in an increasing step order.

There are many possible mappings of these roles to the processors of a concrete system. In our case, we assume that every processor is both an acceptor and a learner. We also assume that some unreliable failure detector elects some processors; the elected processors, in addition to their other roles, become proposers.

To deal with the presence of many proposers, the Paxos algorithm uses ballot numbers [1] (unbounded natural numbers). Every proposer can create new ballot numbers (two proposers include their identifiers to produce mutually distinct ballot numbers). Every acceptor records a ballot number which roughly represents the proposer it is attached to. When a processor becomes a proposer, it executes the following *prepare phase* or *phase 1*. It creates a new ballot number t, and tries to recruit a majority of acceptors by broadcasting its ballot number ($p1a$ message) and waiting for replies ($p1b$) from a majority. An acceptor adopts the ballot number t (i.e., is recruited by the proposer) only if its previously adopted ballot number is strictly smaller. In any case, it replies to the proposer. If the proposer does not manage to recruit a majority of acceptors, it increments its ballot number and tries again.

An acceptor α adds to its $p1b$ reply, the lastly accepted request $accepted_\alpha[s]$ for each step s (if any), along with the corresponding ballot number at the time of acceptance. Thanks to this data, at the end of the prepare phase, the proposer knows the advancement of a majority of acceptors, and can compute requests to propose which do not interfere with possibly previous proposals. It selects, for each step s, the most recent (by refering to the ballot numbers of the accepted requests) accepted request, and if there are no such requests, it can pick any requests it has personally received from clients.

Then for each step s for which the proposer has a request to propose, the proposer executes the following *accept phase* or *phase 2*. The proposer broadcasts to the acceptors a $p2a$ message containing its ballot number t, the step s, and the proposed request p. An acceptor accepts this request for step s if the ballot number t is greater than or equal to its previously adopted ballot number, and acknowledges the proposer. If the proposer sees an acceptor with a greater ballot number, it reexecutes phase 1. Otherwise, it receives positive answers from a majority of acceptors, and it tells the learners to decide on the request for the corresponding step.

Phase 2 can be thought as the "normal case" operation. When a proposer is unique, each time it receives a request from a client, it assigns to it a step number and tell the acceptors. Phase 1 is executed when a processor becomes a proposer. Usually, a processor becomes a proposer when it detects the crash of the previous proposer, e.g., according to some unreliable failure detector. Phase 1 serves as a "take-over" mechanism: the new proposer recruits a majority of acceptors and records, for each of them, their lastly accepted requests. In order for the proposer to make sure that these lastly accepted requests are accepted by a majority of acceptors, it executes phase 2 for each corresponding step.

The difficulty lies in proving that the safety property holds. Indeed, since the failure detection is unreliable, many proposers may be active simultaneously. Roughly speaking, the safety correctness is given by the claim that once a proposer has succeeded to complete phase 2 for a given step s, the chosen request is not changed afterwards for step s. Ordering of events in a common processor that answers two proposers yields the detailed argument, and the existence of such a common processor stems from the fact that any two majorities of acceptors always have non-empty intersection. The liveness property, however, is not guaranteed. A close look at the behaviour of Paxos shows why it is so. Indeed, since every proposer tries to produce a ballot number that is greater than the ballot numbers of a majority of acceptor, two such proposers may execute many unsuccessful phases 1. Intuitively though, if there is

[1] These ballot numbers are not used to indexed the requests like the step numbers above

a single proposer in the system during a long enough period of time, then requests are eventually decided on, and progress of the state-machine is ensured.

Practically Self-Stabilizing Replicated State-Machine. As we pointed out in the previous section, the Paxos algorithm uses unbounded integers to timestamp data (ballot and step numbers). In practice, however, every integer handled by the processors is bounded by some constant $2^{\mathfrak{b}}$ where $\mathfrak{b}$ is the integer memory size. Yet, if every integer variable is initialized to a very low value, the time needed for any such variable to reach the maximum value $2^{\mathfrak{b}}$ is actually way larger than any reasonable system's timescale. For instance, counting from 0 to 2^{64} by incrementing every nanosecond takes roughly 500 years to complete. Such a long sequence is said to be practically infinite. This leads to the following important remark from which the current work stems. *Assuming that the integers are theoretically unbounded is reasonable only when it is ensured, in practice, that every counter is initially set to low values, compared to the maximum value. In particular, any initialized execution of the original Paxos algorithm with bounded integers is valid as long as the counters are not exhausted.*

In the context of self-stabilization, a transient fault may hinder the system in several ways as explained in the introduction. First, it can corrupt the states of the replicas or alter messages leading to incoherent replicas states. Second, and most importantly, a transient fault may also corrupt the variables used to timestamp data (e.g., ballot or step number) in the processors memory or in the communication channels, and set them to a value close to the maximum value $2^{\mathfrak{b}}$. This leads to an infinite suffix of execution in which the State-Machine Replication conditions are never jointly satisfied. This issue is much more worrying than punctual breakings of the State-Machine Replication specifications.

Intuitively though, if one can manage to get every integer variable to be reset to low values at some point in time, then there is consequently a finite execution (ending with ballot or step number reaching the maximum value $2^{\mathfrak{b}}$) during which the system behaves like an initialized original Paxos-based State-Machine Replication execution that satisfies the specifications. Since we use bounded integers, we cannot prove the safe execution to be infinite (just like the original Paxos cannot), but we can prove that this safe execution is as long as counting from 0 to $2^{\mathfrak{b}}$, which is as long as the length of an initialized and safe execution assumed in the original Paxos prior to exhausting the counters. This is what we call a *practically self-stabilizing* behaviour.

More formally, a finite execution is said to be practically infinite when it contains a causally ordered (Lamport's happen-before relation [13]) chain of events of length greater than $2^{\mathfrak{b}}$. We then formulate the *Practically Self-Stabilizing Replicated State-Machine* (PSS-RSM) specification as follows: *(Safety)* Every infinite execution contains a practically infinite segment that yields a linearizable history of client requests and responses, *(Liveness)* In this history, every request has a corresponding response.

Tag System. Our algorithm uses a new kind of timestamping architecture, namely a tag system, to deal with the overflow of integer variables. We first describe a simpler tag system that works when there is a single proposer, before adapting it to the case of multiple proposers.

One of the key ingredient of Paxos is the possibilty for a proposer to increment its ballot number t. We start with t being a natural number between 0 and a large constant $2^{\mathfrak{b}}$, namely a bounded integer. Assume, for now, that there is a single proposer in the system. With an arbitrary initial configuration, some processors may have ballot numbers set to the maximum $2^{\mathfrak{b}}$, thus the proposer will not be able to produce a greater ballot number. To cope with this problem, we redefine the ballot number to be a couple $(l\ t)$ where t is a bounded integer (the integer ballot number), and l a label, which is not an integer but whose type is explicited below. We simply assume that it is possible to increment a label, and that two labels are comparable. The proposer can increment the integer variable t, or increment the label l and reset the integer variable t to zero. Now, if the proposer manages to produce a label that is greater than every label of the acceptors, then right after everything is as if the (integer part of the) ballot numbers of the processors have all started from zero, and, intuitively, we get a practically infinite execution that looks like an initialized one. To do so, whenever the proposer notices an acceptor label which is not less than or equal to the proposer current label (such an acceptor label is said to cancel the proposer label), it records it in a history of canceling labels and produces a label greater than every label in its history.

Obviously, the label type cannot be an integer. Actually, it is sufficient to have some finite set of labels along with a comparison operator and a function that takes any finite (bounded by some constant) subset of labels and produces a label that is greater than every label in this subset. Such a device is called a finite labeling scheme (see Sec. 3).

In the case of multiple proposers, the situation is a bit more complicated. Indeed, in the previous case, the single proposer is the only processor to produce labels, and thus it manages to produce a label greater than every acceptor label once it has collected enough information in its canceling label history. If multiple proposers were also producing labels, none of them would be ensured to produce a label that every other proposer will use. Indeed, the first proposer can produce a label l_1, and then a second proposer produces a label l_2 such that $l_1 \prec l_2$. The first proposer then sees that the label l_2 cancels its label and it produces a label l_3 such that $l_2 \prec l_3$, and so on.

To avoid such interferences between the proposers, we elaborate on the previous scheme as follows. Instead of being a couple (l,t) as above, a ballot number will be a couple (v,t) where t is the integer ballot number, and v is a tag, i.e., a vector of labels indexed by the identifiers of the processors. We assume that the set of identifiers is totally ordered. A proposer μ can only create new labels in the entry μ of its tag. By recording enough of the labels that cancel the label in the entry μ, μ is able to produce a greatest label in the entry μ; therefore the entry μ becomes a valid entry (it has a greatest label) that can be used by other proposers. In order for the different processors to agree on which valid entry to use, we simply impose that each of them uses the valid entry with the smallest identifier.

Finally, in the informal presentation above, we presented the tag system as a means to deal with overflows of ballot numbers, but the same goes for overflows of any other kind of ever increasing (but bounded) sort of variables. In particular, in any implementation of Paxos, the processors record the sequence of executed requests (which is related to the step number); our tag system also copes with overflows of this kind of data.

3 System Settings

Model. All the basic notions we use (state, configuration, execution, asynchrony, ...) can be found in, e.g., [7,19]. Here, the model we work with is given by a system of **n** *asynchronous processors* in a *complete communication network*. Each communication channel between two processors is a *bidirectional asynchronous communication channel of finite capacity* **C** [8]. Every processor has a unique identifier and the set Π of identifiers is totally ordered. If α and β are two processor identifiers, the couple (α,β) denotes the communication channel between α and β. A *configuration* is the vector of states of every processor and communication channel. If γ is a configuration of the system, we note $\gamma(\alpha)$ (resp. $\gamma(\alpha,\beta)$) for the state of the processor α (resp. the communication channel (α,β)) in the configuration γ. We informally[1] define an *event* as the sending or reception of a message at a processor or as a local state transition at a processor. Given a configuration, an event induces a transition to a new configuration. An *execution* is denoted by a sequence of configurations $(\gamma_k)_{0 \leq k < T}$, $T \in \mathbb{N} \cup \{+\infty\}$ related by such transitions[2]. A *local execution* at processor λ is the sequence of states obtained as the projection of an execution on λ.

We consider transient and crash faults only. The effect of a transient fault is to corrupt the state of some processors and/or communication channels; but it does not corrupt the memory where the program is located[3]. As usual in self-stabilization, it is assumed that all the basic services related to message transmission (in particular identifiers) are reliable. Also, we only consider the suffix of execution after the last transient fault; though crash faults may occur in this suffix. This amounts to assume that the initial configuration of every execution is arbitrary and at most **f** processors are prone to crash failures.

A *quorum* is any set of at least $\mathbf{n} - \mathbf{f}$ processors. The maximum number of crash failures **f** satisfies $\mathbf{n} \geq 2 \cdot \mathbf{f} + 1$. Thus, there always exists a responding majority quorum and any two quorums have a non-empty intersection. We also use the "happened-before" strict partial order introduced by Lamport [13]. In our case, we note $e \rightsquigarrow f$ and we say that e happens before f, or f happens after[4] e. Each processor plays the role of a proposer, acceptor and learner. A proposer can be active or inactive[5]. We simply assume that at least one processor acts as a proposer infinitely often. This proposer is not required to be unique in order for our algorithm to stabilize. A unique proposer is required only for the liveness of the state-machine (Sec. 6). Finally, we fix a state-machine $\mathscr{M}$, and each processor has a local copy of $\mathscr{M}$. A *request* corresponds to a transition of the state-machine. We assume that the machine $\mathscr{M}$ has a predefined initial state.

Data Structures. Given a positive integer $\mathfrak{b}$, a $\mathfrak{b}$-bounded integer, or simply a bounded integer, is any non-negative integer less than or equal to $2^{\mathfrak{b}}$. A *finite labeling scheme* is a 4-tuple $\overline{\mathscr{L}} = (\mathscr{L}, \prec, d, \nu)$ where $\mathscr{L}$ is a finite set whose elements are called *labels*, $\prec$ is a partial relation on $\mathscr{L}$ that is irreflexive ($l \not\prec l$) and antisymmetric ($\nexists (l,l')\ l \prec l' \wedge l' \prec l$), d is an integer, namely the *dimension* of the labeling scheme, and ν is the *label increment function*, i.e.,

[1] For a formal definition, refer to, e.g., [7,19].

[2] For sake of simplicity, the events and the transitions are omitted.

[3] This would create Byzantine processes, and is outside of our scope.

[4] Note that the sentences "f happens after e" and "e does not happen before f" are not equivalent.

[5] How a proposer becomes active can be modeled by a the output of a failure detector.

a function that maps any finite set A of at most d labels to a label $\nu(A)$ such that for every label l in A, we have $l \prec \nu(A)$. We denote the reflexive closure of $\prec$ by $\preccurlyeq$. The definition of a finite labeling scheme imposes that the relation $\prec$ is not transitive. Hence, it is not a preorder relation. Given a label l, a *canceling label* for l is a label cl such that $cl \not\preccurlyeq l$. See [2] for a concrete construction of finite labeling scheme of any dimension.

A *tag* is a vector $v[\mu] = (l\ cl)$ where $\mu \in \Pi$ is a processor identifier, l is a label, cl is either the null symbol $\perp$, the overflow symbol ∞ or a canceling label for l. The entry μ in v is said to be *valid* when the corresponding canceling field is null, $v[\mu].cl = \perp$. If v has at least one valid entry, we denote by $\chi(v)$ the *first valid entry of* v, i.e., the smallest identifier μ such that $v[\mu]$ is valid. If v has no valid entry, we set $\chi(v) = \omega$ where ω is a special symbol (not in Π). Given two tags v and v', we note $v \prec v'$ when either $\chi(v) > \chi(v')$ or $\chi(v) = \chi(v') = \mu \neq \omega$ and $v[\mu].l < v'[\mu].l$. We note $v \simeq v'$ when $\chi(v) = \chi(v') = \mu$ and $v[\mu] = v'[\mu]$. We note $v \preccurlyeq v'$ when either $v \prec v'$ or $v \simeq v'$.

A *fifo label history* H of size d, is a vector of size d of labels along with an operator $+$ defined as follows. Let $H = (l_1, \ldots, l_d)$ and l be a label. If l does not appear in H, then $H + l = (l, l_1, \ldots, l_{d-1})$, otherwise $H + l = H$. We define the *tag storage limit* $\mathbf{K}$ and the *canceling label storage limit* $\mathbf{K}^{cl}$ by $\mathbf{K} = \mathbf{n} + \mathbf{C}\tfrac{\mathbf{n}(\mathbf{n}-1)}{2}$ and $\mathbf{K}^{cl} = (\mathbf{n}+1)\mathbf{K}$.

4 The Algorithm

In this section, we describe the Practically Self-Stabilizing Paxos algorithm. In its essence, our algorithm is close to the Paxos scheme except for some details. First, in the original Paxos, the processors decide on a unique request for each step s. In our case, there is no actual step number, but the processors agree on a growing sequence of requests of size at most $2^{\mathfrak{b}}$ as in [16] (see Remark 1 below). Second, our algorithm includes tag related data to cope with overflows.

The variables are presented in Alg. 1. The clients are not modeled here; we simply assume that each active proposer α can query a stream $queue_\alpha$ to get a client request to propose. The variables are divided in three sections corresponding to the different Paxos roles: proposer, acceptor, learner. In each section, some variables are marked as Paxos variables[1] while the others are related to the tag system.

The message flow is similar to Paxos. When a proposer λ becomes active, it executes a prepare phase (phase 1), trying to recruit a majority of acceptors. An acceptor α is recruited if the proposer ballot number is (strictly) greater than its own ballot number. In this case, it adopts the ballot number. It also replies (positively or negatively) to the leader with its latest accepted sequence of requests $accepted_\alpha$ along with the corresponding (integer) ballot number. After recruiting a quorum of acceptors, the proposer λ records the latest sequence (w.r.t. the associated integer ballot numbers) of requests accepted by them in its variable proposed $proposed_\lambda$. If this phase 1 is successful, the proposer λ can execute accept phases (phase 2) for each request received in $queue_\lambda$. For each such request r, the proposer λ appends r to its variable $proposed_\lambda$, and tell the acceptors to accept $proposed_\lambda$. An acceptor accepts the proposal $proposed_\lambda$ when the two following conditions are satisfied: (1) the proposer's ballot number is greater than or equal to its own ballot number, and (2) if the ballot integer associated with the lastly accepted proposal is equal to the proposer's ballot integer, then $proposed_\lambda$ is an extension of the lastly accepted proposal. Roughly speaking, this last condition avoids the acceptor to accept an older (hence shorter) sequence of request. In any case, the acceptor replies (positively or negatively) to the proposer. The proposer λ plays the role of a special learner in the sense that it waits for positive replies from a quorum of acceptors, and, sends the corresponding decision message. The decision procedure when receiving a decision message is similar to the acceptation procedure (reception of a $p2a$ message), except that if the acceptor accepts the proposal, then it also learns (decides on) this proposal and execute the corresponding new requests.

We now describe the treatment of the variables related to the tag system. Anytime a processor α (as an acceptor, learner or proposer) with tag v_α receives a message with a tag v', it updates the canceling label fields before comparing them, i.e., for any μ, if $v_\alpha[\mu].l$ (or $v_\alpha[\mu].cl$) is a label that cancels $v'[\mu].l$, or $v_\alpha[\mu].cl = \infty$ is the overflow symbol, then the field $v'[\mu].cl$ is updated accordingly[2], and vice versa. Also, if the processor α notices an overflow in its own variables (e.g. its ballot integer, or one of the request sequence variables, has reached the upper bound), it sets the overflow symbol ∞ in the canceling field of the first valid entry of the tag. If after such an update, the label $v_\alpha[\alpha].l$ is canceled, then the corresponding canceling label is added to H_α^{cl} as well as the label $v_\alpha[\alpha].l$, and $v_\alpha[\alpha].l$ is set to the

[1] They come from the original formulation of Paxos.

[2] i.e., the field $v'[\mu].cl$ is set to $v_\alpha[\mu].(l\ or\ cl)$. In case, there is a canceling label and the overflow symbol, the canceling label is prefered.

new label $\nu(H^{cl}_\alpha)$ created from the labels in H^{cl}_α with the label increment function. The purpose of H^{cl}_α is to record enough canceling labels for the proposer to produce a greatest label. In addition, if, after the update, it appears that $v_\alpha \preccurlyeq v'$, then α adopts the tag v', i.e., it copies the content of the first valid entry $\mu = \chi(v')$ of v' to the same entry in v_α (assuming $\mu < \alpha$). Doing so, it also records the previous label in v_α in the label history $H_\alpha[\mu]$. If there is a label in $H_\alpha[\mu]$ that cancels $v_\alpha[\mu].l$, then the corresponding field is updated accordingly. The purpose of $H_\alpha[\mu]$ is to avoid cycle of labels in the entry μ of the tag. Recall that the comparison between labels is not a preorder. In case $\mu = \alpha$, then α uses the label increment function on H^{cl}_α to produce a greater label as above.

We say that there is an *epoch change* in the tag v_λ if either the first valid entry $\chi(v_\lambda)$ has changed, or the first valid entry has not changed but the corresponding label has changed. Whenever there is an epoch change in the tag v_λ the processor cleans the Paxos related variables. For a proposer λ, this means that the proposer ballot integer t^p_λ is reset to zero, the proposed requests $proposed_\lambda$ to the empy sequence; in addition, the proposer proceeds to a new prepare phase. For an acceptor (and learner) α, this means that the acceptor ballot integer is reset to zero, the sequences $accepted_\alpha$ and $learned_\alpha$ are reset to the empty sequence, and the local state q^*_α is reset to the predefined initial state of the state-machine.

The pseudo-code in Algorithms 2 and 3 sums up the previous description. Note that, the predicate $(v_\alpha, t_\alpha) < (v_\lambda, t_\lambda)$ (resp. $(v_\alpha, t_\alpha) \leq (v_\lambda, t_\lambda)$) means that either $v_\alpha \prec v_\lambda$, or $v_\alpha \simeq v_\lambda$ and $t_\alpha < t_\lambda$ (resp. $t_\alpha \leq t_\lambda$).

Remark 1. Note that, in our algorithm, the replicas agree on growing sequences of requests, of length at most $2^{\mathfrak{b}}$. Our goal in this paper is not to provide an optimized solution, but to pave the way to it. Yet, a means to control the length of the sequences would be to replace a prefix of request sequence by the state reached from the initial state when applying the prefix. Then the replicas can agree on (possibly conflicting) states by the latest found in a quorum.

Algorithm 1: Variables at processor α

```
(tag system)
    $v_\alpha$ : tag
    canceling label history, $H^{cl}_\alpha$ : fifo history of size $(\mathbf{K}+1)\mathbf{K}^{cl}$
    for each $\mu \in \Pi$, label history, $H[\mu]$ : fifo history of size $\mathbf{K}$
(proposer)
    client requests, $queue_\alpha$ : queue (read-only)
    [Paxos] proposer ballot integer, $t^p_\alpha$ : bounded integer
    [Paxos] proposed requests, $proposed_\alpha$ : requests sequence of size $\leq 2^{\mathfrak{b}}$
(acceptor)
    [Paxos] acceptor ballot integer, $t^a_\alpha$ : bounded integer
    [Paxos] accepted requests, $accepted_\alpha = (t, seq)$ : $t$ bounded integer, $seq$ requests sequence of size $\leq 2^{\mathfrak{b}}$
(learner)
    [Paxos] learned requests, $learned_\alpha$ : requests sequence of size $\leq 2^{\mathfrak{b}}$
    [Paxos] local state, $q^*_\alpha$ : state of the state-machine
```

5 Proofs

Due to lack of space, proofs are only sketched. More details can be found in [3].

5.1 Tag Stabilization

Definition 1 (Interrupt). *Let λ be any processor (as a proposer, or an acceptor) and consider a local subexecution $\sigma = (\gamma_k(\lambda))_{k_0 \leq k \leq k_1}$ at λ. We denote by v^k_λ the λ's tag in $\gamma_k(\lambda)$. We say that an interrupt has occurred at position k in the local subexecution σ when one of the following happens*

- $\mu < \lambda$, *type* $[\mu, \leftarrow]$ *: the first valid entry moves to μ such that $\mu = \chi(v^{k+1}_\lambda) < \chi(v^k_\lambda)$, or the first valid entry does not change but the label does, i.e., $\mu = \chi(v^{k+1}_\lambda) = \chi(v^k_\lambda)$ and $v^k_\lambda[\mu].l \neq v^{k+1}_\lambda[\mu].l$.*

Algorithm 2: Prepare phase (Phase 1)

```
Processor $\lambda$ becomes a proposer:
    increment $t_\lambda$
    if $t_\lambda$ reaches $2^b$ then
        set $v_\lambda[\chi(v_\lambda)].cl$ to $\infty$
        update the entry $v_\lambda[\lambda]$ with $H^{cl}$ if it is invalid
        clean the proposer Paxos variables
    broadcast $\langle p1a, v_\lambda, t_\lambda, \lambda \rangle$
    collect replies $R$ from some quorum $Q$
    update (if necessary) the tag $v_\lambda$ and the label histories
    if no epoch change in $v_\lambda$ and all replies are positive then
        order $R$ with lexicographical order $(accepted_\alpha.t, |accepted_\alpha.seq|)$
        $proposed_\lambda \leftarrow accepted_\alpha.seq$ the maximum in $R$ (break ties if necessary)
        if $proposed_\lambda$ has reached max length then
            set $v_\lambda[\chi(v_\lambda)].cl$ to $\infty$
            update the entry $v_\lambda[\lambda]$ with $H^{cl}$ if it is invalid
            clean the Paxos variables
            repeat phase 1
    else
        if epoch change in $v_\lambda$ then
            clean the Paxos variables
        repeat phase 1

Processor $\alpha$ receives $p1a$ message from $\lambda$:
    update canceling fields in $(v_\alpha, v_\lambda)$
    if $(v_\alpha, t_\alpha) < (v_\lambda, t_\lambda)$ then
        adopt $v_\lambda, t_\lambda$
    if epoch change in $v_\alpha$ then
        clean Paxos variables
    reply to $\lambda$, $\langle p1b, v_\alpha, t_\alpha, accepted_\alpha, \alpha \rangle$
```

Algorithm 3: Accept phase (Phase 2) and Decision

```
Once $\lambda$ gets requests in $queue_\lambda$:
    append requests to $proposed_\lambda$
    broadcast $\langle p2a, v_\lambda, t_\lambda, proposed_\lambda \rangle$
    collect replies $R$ from some quorum $Q$
    update (if necessary) the tag $v_\lambda$ and the label histories
    if no epoch change in $v_\lambda$ and all replies are positive then
        broadcast $\langle dec, v_\lambda, t_\lambda, proposed_\lambda \rangle$
    else
        if epoch change in $v_\lambda$ then clean the Paxos variables
        proceed to phase 1

Processor $\alpha$ receives $p2a$ or $dec$ message from $\lambda$:
    update canceling fields in $(v_\alpha, v_\lambda)$
    if $(v_\alpha, t_\alpha) \leq (v_\lambda, t_\lambda)$ then
        adopt $v_\lambda, t_\lambda$
        if epoch change in $v_\alpha$ then clean the Paxos variables
        if $accepted_\alpha.t < t_\lambda$ or $accepted_\alpha.seq$ is a prefix of $proposed_\lambda$ then
            accept $(t_\lambda, proposed_\lambda)$
            if it is a dec message then
                learn $proposed_\lambda$
                update $q^*_\alpha$ by executing the new requests
    if it is a $p2a$ message then
        reply to $\lambda$, $\langle p2b, v_\alpha, t_\alpha, accepted_\alpha, \alpha \rangle$
```

- *$\mu < \lambda$, type $[\mu, \rightarrow]$: the first valid entry moves to μ such that $\mu = \chi(v_\lambda^{k+1}) > \chi(v_\lambda^k)$.*
- *type $[\lambda, \infty]$: the first valid entry is the same but there is a change of label in the entry λ due to an overflow of one of the Paxos variables; we then have $\chi(v_\lambda^{k+1}) = \chi(v_\lambda^k) = \lambda$ and $v_\lambda^k[\lambda].l \neq v_\lambda^{k+1}[\lambda].l$.*
- *$[\lambda, cl]$: the first valid entry is the same but there is a change of label in the entry λ due to the canceling of the corresponding label; we then have $\chi(v_\lambda^{k+1}) = \chi(v_\lambda^k) = \lambda$ and $v_\lambda^k[\lambda].l \neq v_\lambda^{k+1}[\lambda].l$.*

For each type $[\mu, *]$ ($\mu \leq \lambda$) of interrupt, we denote by $|[\mu, *]|$ the total number (possibly infinite) of interrupts of type $[\mu, *]$ that occur during the local subexecution σ.

If there is an interrupt like $[\mu, \leftarrow]$, $\mu < \lambda$, occurs at position k, then necessarily there is a change of label in the field $v_\lambda[\mu].l$ (due to the adoption of received tag). In addition, the new label l' is greater than the previous label l, i.e., $l \prec l'$. Also note that, if $\chi(v_\lambda^k) = \lambda$, the proposer λ never copies the content of the entry λ of a received tag, say v', to the entry λ of its tag, even if $v_\lambda^k[\lambda].l \prec v'[\lambda].l$. New labels in the entry λ are only produced with the label increment function applied to the union of the current label and the canceling label history H_λ^{cl}.

Definition 2 (Epoch). *Let λ be a processor. An epoch σ at λ is a maximal (for the inclusion of local subexecutions) local subexecution at λ such that no interrupts occur at any position in σ except for the last position. By the definition of an interrupt, all the tag's values within a given epoch σ at λ have the same first valid entry, say μ, and the same corresponding label, i.e., for any two processor states that appear in σ, the corresponding tag values v and v' satisfies $\chi(v) = \chi(v') = \mu$ and $v[\mu].l = v'[\mu].l$. We denote by μ_σ and l_σ the first valid entry and the corresponding label common to all the tag values in σ.*

Definition 3 (h-Safe Epoch). *Consider an execution E and a processor λ. Let Σ be a subexecution in E such that the local subexecution $\sigma = \Sigma(\lambda)$ is an epoch at λ. Let γ^* be the configuration of the system right before the subexecution Σ, and h be a bounded integer. The epoch σ is said to be* h-safe *when the interrupt at the end of σ is due to an overflow of one of the Paxos variables. In addition, for every processor α (resp. communication channel (α, β)), for every tag x in $\gamma^*(\alpha)$ (resp. $\gamma^*(\alpha, \beta)$), if $x[\mu_\sigma].l = l_\sigma$ then any corresponding integer variables (ballot integers, or lengths of request sequences) have values less than or equal to h.*

If there is an epoch σ at processor λ such that $\mu_\sigma = \lambda$ and λ has produced the label l_σ, then necessarily, at the beginning of σ, the Paxos variables have been reset. However, other processors may already be using the label l_σ with, for example, arbitrary ballot integer value. Such an arbitrary value may be the cause of the overflow interrupt at the end of σ. The definition of a h-safe epoch ensures that the epoch is truly as long as counting from h to $2^{\mathfrak{b}}$.

Since a processor λ always checks that the entry $v_\lambda[\lambda]$ is valid (updating with $\nu(H_\lambda^{cl})$ if necessary), it is now assumed, unless stated explicitly, that we always have $\chi(v_\lambda) \leq \lambda$.

Consider a configuration γ and a processor identifier μ. Let $S(\gamma)$ be the set of every tag present either in a processor memory or in some message in a communication channel, in the configuration γ. Let $S^{cl}(\mu, \gamma)$ be the set of labels l such that either l is the value of the label field $x[\mu].l$ for some tag x in $S(\gamma)$, or l appears in the label history $H_\alpha[\mu]$ of some processor α, in the configuration γ. Then, we have $|S(\gamma)| \leq \mathbf{K}$ and $|S^{cl}(\mu, \gamma)| \leq \mathbf{K}^{cl}$. In particular, the number of label values $x[\mu].l$ with x in $S(\gamma)$ is less than or equal to $\mathbf{K}$.

Lemma 1 (Cycle of Labels). *Consider a subexecution E, a processor λ and an entry $\mu < \lambda$ in the tag variable v_λ. The label value in $v_\lambda[\mu].l$ can change during the subexecution E and we denote by $(l^i)_{1 \leq i \leq T+1}$ for the sequence of successive distinct label values that are taken by the label $v_\lambda[\mu].l$ in the entry μ during the subexecution E. We assume that the first T labels $l^1, \ldots, l^T$ are different from each other, i.e., for every $1 \leq i < j \leq T$, $l^i \neq l^j$. If $T > \mathbf{K}$, then at least one of the label l^i has been produced*[1] *by the processor μ during E. If $T \leq \mathbf{K}$ and $l^{T+1} = l^1$, then when the processor λ adopts the label l^{T+1} in the entry μ of its tag v_λ, the entry μ becomes invalid.*

Proof (Sketch). This stems from the fact that in any configuration there are at most $\mathbf{K}$ different tags in the system, and that λ records the last $\mathbf{K}$ label values of the entry μ of its tag. □

Lemma 2 (Counting the Interrupts). *Consider an infinite execution E_∞ and let λ be a processor identifier such that every processor $\mu < \lambda$ produces labels finitely many times. Consider an identifier $\mu < \lambda$ and any processor $\rho \geq \lambda$. Then, the local execution $E_\infty(\rho)$ at ρ induces a sequence of interrupts such that $|[\mu, \leftarrow]| \leq R_\mu = (J_\mu + 1) \cdot (\mathbf{K} + 1) - 1$ where J_μ is the number of times the processor μ has produced a label since the beginning of the execution.*

[1] Precisely, it has invoked the label increment function to update the entry μ of its tag v_μ.

Proof (Sketch). Assume the contrary. Then there are $R_\mu + 1$ successive distinct label values in the field $v_\rho[\mu].l$, $l^1 \prec \cdots \prec l^{R_\mu+1}$. We can divide this sequence in $J_\mu + 1$ segments of length $\mathbf{K}+1$. Due to the previous lemma, there is one segment containing a cycle of labels of length $\leq \mathbf{K}$; this is a contradiction since ρ records the last $\mathbf{K}$ labels in $H_\rho[\mu]$. □

Theorem 1 (Existence of a 0**-Safe Epoch).** *Consider an infinite execution E_∞ and let λ be a processor such that every processor $\mu < \lambda$ produces labels finitely many times. We denote by $|\lambda|$ the number of identifiers $\mu \leq \lambda$, J_μ for the number of times a proposer $\mu < \lambda$ produces a label and we define*

$$T_\lambda = (\sum_{\mu<\lambda} R_\mu + 1) \cdot (|\lambda|+1) \cdot (\mathbf{K^{cl}}+1) \cdot (\mathbf{K}+1) \tag{1}$$

where $R_\mu = (J_\mu + 1) \cdot (\mathbf{K}+1) - 1$. Assume that there are more than T_λ interrupts at processor λ during E_∞ and consider the concatenation $E_c(\lambda)$ of the first T_λ epochs, $E_c(\lambda) = \sigma^1 \ldots \sigma^{T_\lambda}$. Then $E_c(\lambda)$ contains a 0*-safe epoch.*

Proof (Sketch). The bound given by the previous lemma and successive applications of the pigeonhole principle yield a segment $E_2(\lambda)$ of $(\mathbf{K}^{cl}+1)(\mathbf{K}+1)$ successive epochs with interrupts like $[\lambda,\infty]$ and $[\lambda,cl]$ only. If there is in E_2 a segment E_3 of $\mathbf{K}+1$ successive epochs with interrupts like $[\lambda,\infty]$ only, λ must have a created a label that was not present in the system; and the corresponding epoch is 0-safe. Otherwise, there is at least $\mathbf{K}^{cl}+1$ interrupts like $[\lambda,cl]$. This implies that λ has collected all the possible canceling labels. At the end, it produces a greatest label, and the corresponding epoch is necessarily 0-safe. □

Note that the epoch found in the proof is not necessarily the unique 0-safe epoch in $E_c(\lambda)$. The idea is only to prove that there exists a practically infinite epoch. If the first epoch σ at λ ends because the corresponding label l_σ in the entry μ_σ gets canceled, but lasts a practically infinite long time, then this epoch can be considered, from an informal point of view, safe. One could worry about having only very "short" epochs at λ due to some inconsistencies (canceling labels or overflows) in the system. Theorem 1 shows that every time a "short" epoch ends, the system somehow loses one of its inconsistencies, and, eventually, the proposer λ reaches a practically infinite epoch. Note also that a 0-safe epoch and a 1-safe or a 2-safe epoch are, in practice, as long as each other. Indeed, any h-safe epoch with h very small compared to $2^{\mathfrak{b}}$ can be considered practically infinite. Whether h can be considered very small depends on the concrete timescale of the system. Besides, every processor α always checks that the entry α is valid. Doing so the processor α still works to find a "winning" label for its entry α. In that case, if the entry μ becomes invalid, then the entry α is ready to be used, and a safe epoch can start without waiting any longer.

5.2 Safety

To prove the safety property within a subexecution, we have to focus on the events that correspond to deciding a proposal, e.g., (v,t,p) at processor α (v being a tag, t a ballot integer, p a sequence of requests). Such an event may be due to corrupted messages in the communication channels an any stage of the Paxos algorithm. Indeed, a proposer computes the proposal it will send in its phase 2 thanks to the replies it has received at the end of its phase 1. Hence, if one of these messages is corrupted, then the safety might be violated. However, there is a finite number of corrupted messages since the capacity of the communication channels is finite. To formally deal with these issues, we define the notion of scenario that corresponds to specific chain of events involved in the Paxos algorithm. Consider a subexecution $E = (\gamma_k)_{k_0 \leq k \leq k_1}$. A *scenario in E* is a sequence $U = (U_i)_{0 \leq i < I}$ where each U_i is a collection of events in E. In addition, every event in U_i happens before every event in U_{i+1}.

Definition 4 (Phase Scenario). *Consider a proposer ρ, an acceptor α, quorums S and Q of acceptors, a tag v, a ballot integer t, and a sequence of requests p.*

A phase 1 scenario *is defined as follows. The proposer ρ broadcasts a message $p1a$ containing the tag v, and ballot integer t. Every acceptor in the quorum S receives this message and adopts*[1] *the tag v. Every processor α in the quorum S replies to the proposer ρ a $p1b$ message telling they adopted the couple (v,t), and containing the last proposal they accepted. These messages are received by ρ. We denote this scenario by $\rho \xrightarrow{p1a} (S,v,t) \xrightarrow{p1b} \rho$.*

[1] Recall that this means the acceptor, say α, copies the entry $v[\chi(v)]$ in the entry $v_\alpha[\chi(v)]$.

A phase 2 scenario with acceptation *is defined as follows. The proposer ρ broadcasts a p2a message containing the tag v, the ballot integer t, and the proposed sequence of requests p. The acceptor α accepts the proposal (v,t,p). We denote this scenario by $\rho \xrightarrow{p2a} (\alpha,v,t,p)$.*

A phase 2 scenario with quorum acceptation *is defined as follows. The proposer ρ broadcasts a p2a message containing the tag v, the ballot integer t, and the proposed sequence of requests p. Every acceptor in the quorum Q accepts the proposal (v,t,p). Every acceptor α in the quorum Q sends to the proposer ρ a p2b message telling that it has accepted the proposal (v,t,p). The proposer ρ receives these messages. We denote this scenario by $\rho \xrightarrow{p2a} (Q,v,t,p) \xrightarrow{p2b} \rho$.*

A phase 2 scenario with decision *is defined as follows. The proposer ρ broadcasts a p2a message containing the tag v, the ballot integer t, and the proposed sequence of requests p. Every acceptor in the quorum Q accepts the proposal (v,t,p). Every acceptor α in the quorum Q sends to the proposer ρ a p2b message telling that it has accepted the proposal (v,t,p). The proposer ρ receives these messages. The proposer ρ sends a decision message containing the proposal (v,t,p). The processor α receives this message, accepts and decides on the proposal (v,t,p). We denote this scenario by $\rho \xrightarrow{p2a} (Q,v,t,p) \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha,v,t,p)$.*

In all the previous cases, we say that the phase scenarios are conducted by the proposer ρ *and* use the ballot (v,t).

Definition 5 (Simple Acceptation Scenario). *A* simple acceptation scenario *is the concatenation of a phase 1 scenario, followed by a finite number of phase 2 scenarios with quorum acceptation, and ending with a phase 2 scenario with either acceptation, or decision; all the phase scenarios being conducted by the same proposer ρ, and using the same ballot (v,t). Let S be the quorum of acceptors in the phase 1 scenario, p be the sequence of requests accepted (or decided on) in the last event of the scenario, and α be the corresponding acceptor. If the last phase scenario is a phase scenario with acceptation, then we denote the simple acceptation scenario by $\rho \xrightarrow{p1a} (S,v,t) \rightsquigarrow \rho \xrightarrow{p2a} (\alpha,v,t,p)$. If the last phase scenario is a phase scenario with decision, then we denote the simple acceptation scenario by $\rho \xrightarrow{p1a} (S,v,t) \rightsquigarrow \rho \xrightarrow{p2a} (Q,v,t,p) \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha,v,t,p)$. When we want to indicate that both cases are possible, we simply denote the simple acceptation scenario by $(\rho,S,v,t) \rightsquigarrow (\alpha,v,t,p)$.*

A simple acceptation scenario is simply a basic execution of the Paxos algorithm that leads a processor to either accept a proposal, or decide on a proposal (accepting it by the way).

Definition 6 (Fake Message). *Given a subexecution $E = (\gamma_k)_{k_0 \leq k \leq k_1}$, a* fake message relative to the subexecution *E, or simply a fake message, is a message that is in the communication channels in the first configuration γ_{k_0} of the subexecution E.*

This definition of fake messages comprises the messages at the beginning of E that were not sent by any processor, but also messages produced in the prefix of execution that precedes E.

Definition 7 (Fake Phase Scenario). *Consider a proposer ρ, an acceptor α, quorums S and Q of acceptors, a tag v, a ballot integer t, and a sequence of requests p. Fix a subexecution E. A* fake phase scenario relative to *E is one of the following scenario.*

(Fake phase 1 scenario) *The proposer ρ sends a p1a message with ballot (v,t). It receives positive replies from a quorum S, one of these replies at least being fake (i.e. it was not actually sent by an acceptor). We denote this fake phase scenario by $\rho \xrightarrow{p1a} (S,v,t) \xrightarrow{fake\ p1b} \rho$.*

(Fake phase 2 scenario with acceptation) *The acceptor α receives a fake p2a with proposal (v,t,p) that seems to come from the processor ρ. The acceptor α accepts the proposal. We denote this scenario by $\rho \xrightarrow{fake\ p2a} (\alpha,v,t,p)$.*

(Fake phase 2 scenario with quorum acceptation) *The proposer ρ sends a p2a message with proposal (v,t,p). The proposer ρ receives positive replies from a quorum Q, one of these replies, at least, being fake. Then ρ sends a decision message with proposal (v,t,p) to the acceptor α, and α decides accordingly. We denote this scenario by $\rho \xrightarrow{p2a} (Q,v,t,p) \xrightarrow{fake\ p2b} \rho \xrightarrow{dec} (\alpha,v,t,p)$.*

(Fake phase 2 scenario with decision) *The acceptor α receives a fake decision message with proposal (v,t,p) which seems to come from the proposer ρ. The acceptor α decides accordingly. We denote this scenario by $\rho \xrightarrow{fake\ dec} (\alpha,v,t,p)$.*

Definition 8 (Simple Fake Acceptation Scenario). *A* simple fake acceptation scenario *is either a fake phase 2 scenario with acceptation, a fake phase 2 scenario with quorum acceptation, a fake phase 2 scenario with decision, or the concatenation of a fake phase 1 scenario, followed by a finite number of (non-fake) phase 2 scenarios with quorum acceptation, and ending with a (non-fake) phase 2 scenario with either an acceptation, or a decision; all the scenarios being conducted by the same proposer ρ, and using the same ballot (v,t). We often denote this kind of scenarios by $fake \rightsquigarrow (\alpha,v,t,p)$ where (α,v,t,p) refers to the last acceptation (or decision) event.*

A simple fake acceptation scenario is somehow similar to a simple acceptation scenario except for the fact that at least one fake message (relative to the given subexecution) is involved during the scenario.

Definition 9 (Composition). *Consider two simple scenarios*

$$U = X \rightsquigarrow (\alpha_1,v_1,t_1,p_1)$$
$$V = (\rho_2,S_2,v_2,t_2) \rightsquigarrow (\alpha_2,v_2,t_2,p_2)$$

where $X = fake$ or $X = (\rho_1,S_1,v_1,t_1)$ such that the following three conditions are satisfied. (1) The processor α_1 belongs to S_2 (2) Let e_2 be the event that corresponds to α_1 sending a $p1b$ message in scenario V. Then the event "α_1 accepts the proposal (v_1,t_1,p_1)" from U is the last acceptation event before e_2 occurring at α_1. In addition, the proposer ρ_2 selects the proposal (t_1,p_1) as the highest-numbered proposal at the end of the Paxos phase 1*. In particular, p_1 is a prefix of p_2, i.e., $p_1 \sqsubset p_2$. (3) All the tags involved share the same first valid entry, the same corresponding label.*

Then the composition *of the two simple scenarios is the concatenation the scenarios U and V. This scenario is denoted by $X \rightsquigarrow (\alpha_1,v_1,t_1,p_1) \rightarrow (\rho_2,S_2,v_2,t_2) \rightsquigarrow (\alpha_2,v_2,t_2,p_2)$. Note also that the ballot integer is strictly increasing along the simple scenarios.*

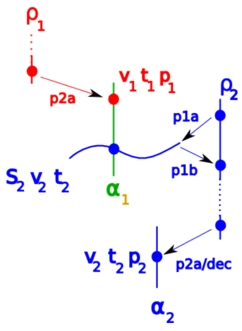

Fig. 1. Composition of scenarios - Time flows downward, straight lines are local executions, arrows represent messages.

Definition 10 (Acceptation Scenario). *Given a subexecution E, an* acceptation scenario *is the composition U of simple acceptation scenarios $U_1,\ldots,U_r$ where U_1 is either a simple acceptation scenario or a simple fake acceptation scenario relative to E, whereas the other are real (i.e. non-fake) simple acceptation scenarios. We denote it by $X \rightsquigarrow (\alpha_1,v_1,t_1,p_1) \rightarrow (\rho_2,S_2,v_2,t_2) \rightsquigarrow (\alpha_2,v_2,t_2,p_1)\ldots(\rho_r,S_r,v_r,t_r) \rightsquigarrow (\alpha_r,v_r,t_r,p_r)$ where X is either $fake$ or some (ρ_1,S_1,v_1,t_1).*

An acceptation scenario whose first simple scenario is not fake relative to E is called real acceptation scenario relative to E. *An acceptation scenario whose first simple scenario is fake relative to E is called* fake acceptation scenario relative to E.

Given an acceptation event or a decision event, there is always at least one way to trace back the scenario that has lead to this event. If one of these scenarios involve a fake message, then we cannot control the safety property. Besides, all the tags involved share the same first valid entry μ and the same corresponding label l. Also, the ballot integer value, as well as the sequence of requests, is increasing along the acceptation scenario; i.e., if $i < j$, then $t_i < t_j$ and $p_i \sqsubset p_j$.

Definition 11 (Fake event). *Consider an event e that corresponds to some processor accepting a proposal, let U be the simple acceptation scenarios that ends with the event e. The event e is said to be* fake relative to a subexecution *E if U is a fake simple acceptation scenario relative to E. The event e is sait to be* real relative to *E otherwise.*

Definition 12 (Simple Scenario Characteristic). *The* characteristic *of a simple acceptation scenario U with tag v, ballot integer t, is the tuple $char(U) = (\chi(v), v[\chi(v)].l, t)$.*

Definition 13 (Observed Zone). *Consider an execution E. Let λ be a proposer and let Σ be a subexecution such that the local execution $\sigma = \Sigma(\lambda)$ at λ is a h-safe epoch. We denote by F the suffix of the execution that starts with Σ. Assume that λ hears from at least two quorums during its epoch σ. Let Q^0, Q^f be the first and last quorums respectively whose messages are processed by the proposer λ during σ. For each processor α in Q^0 (resp. Q^f), we denote by $e^0(\alpha)$ (resp. $e^f(\alpha)$) the event that corresponds to α sending to λ a message received in the phase that corresponds to Q^0 (resp. Q^f).*

The zone observed by λ during the epoch σ, *namely $Z(F,\lambda,\sigma)$, is the set of acceptation scenarios relative to F described as follows. An acceptation scenario relative to F belongs to $Z(F,\lambda,\sigma)$ if and only if it ends with a real acceptation (or decision) event (relative to F) that does not happen after the end of σ and it contains a real simple acceptation scenario $U = (\rho,S,v,t) \rightsquigarrow (\beta,v,t,p)$ such that there exists an acceptor α in $S \cap Q^0 \cap Q^f$ at which the event $e^0(\alpha)$ happens before the event e that corresponds to sending a $p1b$ message in U, and the event e happens before the event $e^f(\alpha)$ (cf. Figure 2).*

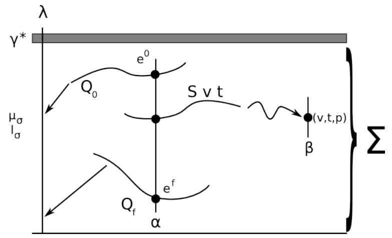

Fig. 2. Scenario $(\rho,S,v,t) \rightsquigarrow (\beta,v,t,p)$ in $Z(F,\lambda,\sigma)$ - Time flows downward, straight lines are local executions, curves are send/receive events, arrows represent messages.

The observed zone models a globally defined time period during which we will prove, under specific assumptions, the safety property (cf. Theorem 3).

Lemma 3 (Epoch and Cycle of Labels). *Consider an execution E. Let λ be a processor and consider a subexecution Σ such that the local execution $\sigma = \Sigma(\lambda)$ is an epoch at λ. We denote by F the suffix of the execution E that starts with Σ. Consider a processor ρ and a finite subexecution G in F as follows: G starts in Σ and induces a local execution $G(\rho)$ at ρ such that it starts and ends with the first valid entry of the tag v_ρ being equal to μ_σ and containing the label l_σ, and the label field in the entry $v_\rho[\mu_\sigma]$ undergoes a cycle of labels during $G(\rho)$. Assume that, if $\mu_\sigma < \lambda$, the processor μ_σ does not produce any label during G. Then $\mu_\sigma = \lambda$ and the last event of σ happens before the last event of $G(\rho)$.*

Proof. By Lemma 1, since the entry $v_\rho[\lambda]$ remains valid after the readoption of the label l at the end of $G(\rho)$, the proposer μ_σ must have produced some label l' during G (hence $\mu_\sigma = \lambda$) that was received by ρ during G. Necessarily, the production of l' happens after the last event of σ at λ, thus the last event of $G(\rho)$ at ρ also happens after the last event of σ at λ. □

Theorem 2 (Safety - Weak Version). *Consider an execution E. Let λ be a processor and let Σ be a subexecution such that the local execution $\sigma = \Sigma(\lambda)$ at λ is an h-safe epoch. We denote by F the suffix of the execution that starts with Σ. Consider the two simple scenarios $U_1 = \rho_1 \xrightarrow{p1a} (S_1,v_1,t_1) \rightsquigarrow \rho_1 \xrightarrow{p2a} (Q_1,v_1,t_1,p_1) \xrightarrow{p2b} \rho_1 \xrightarrow{dec} (\alpha_1,v_1,t_1,p_1)$ and $U_2 = (\rho_2,S_2,v_2,t_2) \rightsquigarrow (\alpha_2,v_2,t_2,p_2)$ with characteristics $(\mu_\sigma,l_\sigma,t_1)$ and $(\mu_\sigma,l_\sigma,t_2)$ respectively.*

We denote by e_i the acceptation event (α_i,v_i,t_i,p_i). Assume that the events e_1 and e_2 occur in F and that $h \leq t_1 \leq t_2$. In addition, assume that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. We then have two *cases: (a) If $t_1 = t_2$, then either $p_1 \sqsubset p_2$, or $p_2 \sqsubset p_1$, or the last event of σ happens before one of the event e_1 or e_2. (b) If $t_1 < t_2$, then $p_1 \sqsubset p_2$ or the last event of σ happens before one of the event e_1 or e_2.*

Proof (Sketch). We assume that both events e_1 and e_2 do not happen after the last event of σ and we prove the result. We denote by γ^* the configuration right before the subexecution Σ. We prove the result by induction on the value of t_2.

(Bootstrapping). We first assume that $t_2 = t_1$. Recall the ballot integers include the identifiers of the proposer, hence $\rho_1 = \rho_2$. If $p_1 \not\sqsubseteq p_2$ and $p_2 \not\sqsubseteq p_1$, then ρ_1 has sent two $p2a$ messages with different proposals and the same ballot. Let e and f be the events corresponding to these two sendings. None of the events e and f occurs in the execution prefix A, otherwise, since e_1 and e_2 occur in F, the configuration γ^* would contain a ballot (x,t) with $x[\mu_\sigma].l = l_\sigma$ and $t \geq h$; this is a contradiction since σ is h-safe. We will refer to this argument as the *safe epoch* argument. Hence, e and f occur in F. The fact that $p_1 \not\sqsubseteq p_2$ and $p_2 \not\sqsubseteq p_1$ implies that there must be a cycle of labels in the entry $v_{\rho_1}[\mu_\sigma]$ between the e and f. By Lemma 3, this implies that the last event of σ happens before the event e_1 or e_2; this is a contradiction. We will refer to this argument as the *cycle of label* argument. Hence, $p_1 \sqsubset p_2$ or $p_2 \sqsubset p_1$.

(Induction). Now, $t_1 < t_2$ and we assume the result holds for every value t such that $t_1 \leq t < t_2$. Pick some acceptor β in $Q_1 \cap S_2$. From its point of view, there are two events f_1 and f_2 at β that respectively correspond to the acceptation of the proposal (v_1, t_1, p_1) in the scenario U_1 (reception of a $p2a$ message), and the adoption of the ballot (v_2, t_2) in the scenario U_2 (reception of a $p1a$ message).

First, the events f_1 and f_2 occur in the suffix F (same argument as in bootstrapping). Since $t_1 < t_2$, by the *cycle of labels* argument, f_1 happens before f_2. The $p1b$ message the acceptor β has sent contains a non-null lastly accepted proposal (t,p) such that $t_1 \leq t < t_2$ and $p_1 \sqsubset p$. Otherwise, the *cycle of labels* argument would show (again) a contradiction.

Now, the proposer ρ_2 receives a set of proposals from the acceptors of the quorum S_2, including at least one non-null proposal from β. Then, it selects among the replies, the accepted proposal (t_c, p_c) with the highest ballot integer, and highest request sequence length (lexicographical order). Since ρ_2 has received the proposal (t,p) from β, we then have $h \leq t_1 \leq t \leq t_c < t_2$ and $(t, |p|) \leq (t_c, |p_c|)$ (lexicographically).

Let β_c be the proposer in S_2 which has sent to ρ_2 the proposal (t_c, p_c) in the $p1b$ message. By the *safe epoch* argument, there is an event f_c in F that corresponds to β_c accepting the proposal (t_c, p_c). Consider the simple acceptation scenario V_c that ends with f_c, and let $char(V_c) = (\mu_c, l_c, t_c)$ be its characteristic. Since f_c is the last acceptation event before β_c replies to ρ_2 (with a $p1a$ message), we must have $(\mu_c, l_c) = (\mu_\sigma, l_\sigma)$; otherwise, the accepted variable $accepted_{\beta_c}$ would have been cleared (epoch change at β_c), and β_c would have not sent the non-null proposal (t_c, p_c) to ρ_2. Because of the *safe epoch* argument, V_c cannot be a fake simple acceptation scenario; thus V_c is a real simple acceptation scenario.

By applying the induction hypothesis to V_c, and since f_c cannot happen after the last event of σ (otherwise e_2 would also happen after it), we have two cases. The case (A) $t_1 = t_c$. Then $p_1 \sqsubset p_c$ or $p_c \sqsubset p_1$. But, the fact that $(t, |p|) \leq (t_c, |p_c|)$ (lexicographically) and $p_1 \sqsubset p$ implies that $|p_c| \geq |p| \geq |p_1|$, and thus $p_1 \sqsubset p_c$. The case (B) $t_1 < t_c$. But then $p_1 \sqsubset p_c$.

In all cases, we have $p_1 \sqsubset p_c$. But, we also have $p_c \sqsubset p_2$ (scenario U_2), hence $p_1 \sqsubset p_2$. □

Corollary 1. *Consider an execution E. Let λ be a processor and let Σ be a subexecution such that the local execution $\sigma = \Sigma(\lambda)$ at λ is an h-safe epoch. We denote by F the suffix of the execution that starts with Σ.*

Consider two decision events $e_i = (\alpha_i, v_i, t_i, p_i)$, $i = 1,2$, such that $\chi(v_i) = \mu_\sigma$, $v_i[\mu_\sigma].l = l_\sigma$ and $t_i \geq h$. Assume that both events e_1 and e_2 are real decision events relative to F. In addition, assume that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. Then either $p_1 \sqsubset p_2$, $p_2 \sqsubset p_1$ or the last event of σ happens before one of the event e_1 or e_2.

Theorem 3 (Safety). *Consider an execution E, a proposer λ proposer and a subexecution Σ such that the local execution $\sigma = \Sigma(\lambda)$ at λ is a h-safe epoch for some bounded integer h. We denote by F the suffix of execution that starts with Σ. Assume that the observed zone $Z(F, \lambda, \sigma)$ is defined and that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. Consider two scenarios U_1 and U_2 in $Z(F, \lambda, \sigma)$ ending with acceptation events $e_1 = (\alpha_1, v_1, t_1, p_1)$ and $e_2 = (\alpha_2, v_2, t_2, p_2)$. Let $\mu_i = \chi(v_i)$ and $l_i = v_i[\mu_i]$, $i = 1,2$, and assume that $\mu_\sigma \leq \min(\mu_1, \mu_2)$ and $t_1, t_2 \geq h$. Then $(\mu_1, l_1) = (\mu_2, l_2) = (\mu_\sigma, l_\sigma)$, and $p_1 \sqsubset p_2$ or $p_2 \sqsubset p_1$.*

Proof (Sketch). The definition of the observed zone imply that $(\mu_1, l_1) = (\mu_2, l_2) = (\mu_\sigma, l_\sigma)$ because the corresponding scenarios has been "seen" by λ during its epoch. Then the previous corollary applies. □

In the case $\mu_\sigma < \lambda$, assuming that μ_σ does not produce any label during F means that the proposer λ should be the live processor with the lowest identifier. To deal with this issue, one can use a failure detector.

6 Self-Stabilizing Failure Detector

Liveness in Paxos is not guaranteed unless there is a unique proposer. The original Paxos algorithm assumes that the choice of a distinguished proposer is done through an external module. In the sequel, we present an implementation of a self-stabilizing failure detector that works under a partial synchronism assumption. Note that this assumption is strong enough to implement an eventual perfect failure detector, but such a failure detector is not mandatory for our tag system to stabilize. This brief section simply explains how a *self-stabilizing* implementation can be done; which is, although not difficult, not obvious either. Each processor α has a vector L_α indexed by the processor identifiers; each entry $L_\alpha[\mu]$ is an integer whose value is comprised between 0 and some predefined maximum constant W. Every processor α keeps broadcasting a hearbeat message $\langle hb, \alpha \rangle$ containing its identifier (e.g., by using [7,8]). When the processor α receives a heartbeat from processor β, it sets the entry $L_\alpha[\beta]$ to zero, and increments the value of every entry $L_\alpha[\rho]$, $\rho \neq \beta$ that has value less than W. The detector output at processor α is the list F_α of every identifier μ such that $L_\alpha[\mu] = W$. In other words, the processor α assesses that the processor β has crashed if and only if $L_\alpha[\beta] = W$.

(Interleaving of Heartbeats). For any two live processors α and β, between two receptions of heartbeat $\langle hb, \beta \rangle$ at processor α, there are strictly less than W receptions of heartbeats from other processors. Under this condition, for every processor α, if the processor β is alive, then eventually the identifier β does not belong to the list F_α. A distinguished proposer ρ can be defined as follows: $\rho = \min(\mu;\ L_\rho[\mu] < W)$.

References

1. U. Abraham. Self-stabilizing timestamps. *Theor. Comput. Sci.*, 308(1-3):449–515, Nov. 2003.
2. N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. Pragmatic self-stabilization of atomic memory in message-passing systems. In *SSS*, pages 19–31, 2011.
3. P. Blanchard, S. Dolev, J. Beauquier, and S. Delaët. Self-stabilizing paxos. *CoRR*, abs/1305.4263, 2013.
4. M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.
5. 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, pages 398–407, New York, NY, USA, 2007. ACM.
6. D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2):418–455, Apr. 1997.
7. S. Dolev. *Self-stabilization*. MIT Press, 2000.
8. S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract). In *SSS*, pages 133–147, 2012.
9. S. Dolev, R. I. Kat, and E. M. Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76:884–900, December 2010.
10. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.
11. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
12. A. Israeli and M. Li. Bounded time-stamps. *Distrib. Comput.*, 6(4):205–209, July 1993.
13. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
14. L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
15. L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.
16. L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.
17. L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.
18. L. Lamport. Byzantizing paxos by refinement. In *DISC*, pages 211–224, 2011.
19. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.