RAPPORT DE RECHERCHE

L R I

SELF-STABILIZING PAXOS

BLANCHARD P / DOLEV S / BEAUQUIER J / DELAET S

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

02/2013

Rapport de Recherche N° 1558

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

Self-Stabilizing Paxos

[Full Paper]

Peva Blanchard
LRI, Université Paris-Sud XI
Orsay, France
blanchard@lri.fr

Shlomi Dolev*
Dept. of Computer Science
Ben-Gurion Univ. of the Negev
Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il

Joffroy Beauquier
LRI, Université Paris-Sud XI
Orsay, France
jb@lri.fr

Sylvie Delaët
LRI, Université Paris-Sud XI
Orsay, France
delaet@lri.fr

ABSTRACT

We present the first self-stabilizing consensus and replicated state machine for asynchronous message passing systems. The scheme does not require that all participants make a certain number of steps prior to reaching a practically infinite execution where the replicated state machine exhibits the desired behavior. In other words, the system reaches a configuration from which it operates according to the specified requirements of the replicated state-machine, for a long enough execution regarding all practical considerations.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault tolerance*

General Terms

Algorithms, Theory, Reliability

Keywords

Distributed Algorithms, Consensus, Replicated State-Machine, Self-stabilization, Fault Tolerance, Paxos

1. INTRODUCTION

One of the most influential results in distributed computing is Paxos, where repeated asynchronous consensus is used to replicate a state machine using several physical machines. The task is to use asynchronous consensus, where safety is guaranteed, and liveness is almost always achieved by using an unreliable failure detector, to implement an abstraction of very reliable single state machine on top of physical machines that can crash, though usually at least several machines stay alive at any particular moment. The extreme usefulness of such an approach is proven daily by the usage of this technique, by the very leading companies, to ensure their availability and functionality.

Unfortunately Paxos is not self-stabilizing and therefore a single transient fault may lead the system to stop functioning even when all the cluster machines operate. One example is the corruption of the time-stamp used to order operations in Paxos, where a single corruption of the value of this counter to the maximal value will cause the system to be blocked. In another scope, the occurrence of transient fault with the same nature caused the Internet to be blocked for a while [9]. Self-stabilization is a property that every on-going system should have, as self-stabilizing systems automatically recover from unanticipated states, i.e., states that have been reached due to insufficient error detection in messages, changes of bit values in memory [5], and in fact any temporary violation in the assumptions made for the system to operate correctly. The approach is comprehensive, rather than addressing specific fault scenarios (risking to miss a scenario that will later appear), the designer considers every possible configuration of the system, where configuration is a cartesian product of the possible values of the variables. Then the designer has to prove that from every such a configuration, the system converges to exhibit the desired behavior.

Self-stabilizing systems do not rely on the consistency of a predefined initial configuration and the application of correct steps thereafter. In contrast, self-stabilizing systems assume that the consistency can be broken along the execution and need to automatically recover thereafter. The designers assume an arbitrary configuration and prove convergence, not because they would like the system to be started in an arbitrary configuration, but because they are aware that the specified initial configuration and the defined steps consistency maybe temporarily broken, and would like the system to regain consistency. Therefore, although the system may

*Partially supported by Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences, Intel, MFAT, MAGNET and Lynne and William Frankel Center for Computer Sciences.

lose safety properties, the safety is automatically regained, leading to a safer behavior than of non-stabilizing systems, namely, initially safe and eventually safe [2].

Self-stabilizing consensus and replicated state machine for shared memory system appeared in [6], the case of message passing being left to future investigation. One approach to gain a self-stabilizing consensus and replicated state machine in message passing is to implement the read-write registers used in [6], using message passing[1]. A self-stabilizing implementation of such a single-writer multiple-reader register appeared in [1]. Unfortunately, the implantation had to assume that the writer is active forever. Thus, the implementation of self-stabilizing Paxos under the original assumptions was left open. In this paper we present the first self-stabilizing Paxos in message passing systems.

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. SELF-STABILIZING PAXOS OVERVIEW

In this section, we define the Repeated Consensus 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 Repeated Consensus Problem. Doing so, we investigate a new kind of self-stabilization, namely the *practical self-stabilization*. Then, we informally describe the main theorems of this paper.

2.1 Repeated Consensus Problem and Paxos Algorithm

2.1.1 Repeated Consensus

The processors have to perform successive instances of consensus on values proposed by some of them. Every processor is assumed to have an integer variable s, namely the step variable, that denotes the current consensus instance it is involved in. In each consensus instance, processors decide on a value. For example, in the context of replicated state machines, the step variable denotes the current step of the state machine, and at each step, a processor may decide to apply a command to its copy of the state machine. Processors may have different views on what is the current step since some of them may have progressed faster than others. The Repeated Consensus Problem is defined by the following conditions :

- *(Safety)* For every step s, if two processors decide on values in step s, then the two decided values must be equal.
- *(Integrity)* For every s, if a processor decides on some value, then this must have been proposed in step s.
- *(Liveness)* Every non-crashed processor decides in every step.

2.1.2 How Paxos Works ?

The original Paxos algorithm guarantees the safety and the integrity 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. The algorithm uses unbounded integers and also assumes that the system starts in a consistent initial configuration. To guarantee the liveness property, additional assumptions must be made and are discussed below. The Paxos algorithm defines three roles as follows :

- Proposer : it basically tries to impose a consensus value for its current step. Note that there might be more than one proposer in each step.
- Acceptor : it accepts consensus values according to some specific rules. A value can be decided on for step s when a majority of acceptors have accepted it in step s.
- Learner : it learns when some value has been accepted by a majority of acceptors for some step and decides accordingly. In some implementations, the learner notifies other learners for them to decide on the same value for this step.

In practice, the actual mapping of the roles in the system depends on the implementation. Here, we assume that every processor is an acceptor while some of them can also be proposers. In addition, every proposer is a learner that notifies every other processor about decisions it takes. Every proposer has its own idea of what should the value be for step s. In each step s, a proposer tries to impose some consensus value. However, its trial may succeed or fail according to the activity of other proposers. When a trial fails, the proposer may move to the next step because, for instance, the current step has already been completed by another proposer, or it may try again. Hence, every processor has an integer variable t, namely the trial variable, that denotes the current trial it is involved in. A Paxos tag is defined to be a couple $(s\ t)$ where s denotes a step, i.e., a consensus instance, and t a trial within this step. The Paxos algorithm assumes that all the step variables and the trial variables are natural integers, hence unbounded, and initially set to zero.

In Paxos, for every proposer, every trial is composed of two phases. In phase one, the proposer first tries to recruit a majority of acceptors by sending to all the acceptors its tag $(s\ t)$ (phase 1, message $p1a$) containing its current step and its current trial, and waits for replies from a majority of acceptors. An acceptor always tries to participate in the last step (consensus instance) and trial. Hence, upon receiving the proposer tag $(s\ t)$, the acceptor checks whether the tag $(s\ t)$ is lexicographically greater than the tag containing its current step and current trial, (in which case the acceptor agrees to be recruited by the proposer, i.e., to adopt the step s and the trial t as its current step and current trial) and replies (phase 1, message $p1b$) to the proposer with its current tag (updated or not) and the last consensus value it has accepted along with the tag at the time of its acceptation, if any, or a null value otherwise. By adopting the tag $(s\ t)$, the acceptor also promises not to accept any value coming with a tag less than $(s\ t)$.

Upon receiving the replies from a majority of acceptors, the proposer knows if it has managed to recruit all of them by

[1]A suggestion made by Eli Gafni.

checking if the acceptors have adopted its tag. If it is not the case, it is because some acceptor tag is not less than or equal to the current tag of the proposer. Then the proposer increments (lexicographically) its tag until it is greater than every tag the proposer has received, and repeats phase one. In case phase one is successful, the proposer also knows the values that were lastly accepted by some majority of acceptors for the proposer's current step along with the corresponding tags. The proposer then adopts the value which has the maximal tag, or keeps its own value if no values were reported by the acceptors, before starting phase two.

The proposer sends (phase 2, message $p2a$) to all the acceptors, the successful tag with the consensus value obtained as above and waits for replies from a majority of acceptors. An acceptor with a tag smaller than or equal to the tag in the message, receiving the proposed couple tag and consensus value from the proposer, updates its variables (adopting the proposer's tag, accepting the consensus value and recording the corresponding tag), and acknowledges the proposer (phase 2, message $p2b$). Once the proposer gets replies from a majority of the processors, the proposer examines the replies. In case the replies are all accepting the proposed pair of tag and consensus value, the proposer sends a decision message with the tag and the consensus value to all the processors, otherwise, when at least one processor replies with a bigger tag, the leader increments its tag and restarts from phase one again.

2.1.3 Why Paxos Works ?

The integrity property is guaranteed by the fact that a decided value always comes from a proposer in the system. The difficulty lies in proving that the safety property is ensured. Roughly speaking, the safety correctness is yielded by the claim that once a proposer has succeeded to complete the second phase, the consensus value is not changed afterwards for the corresponding step. Ordering of events in a common processor that answers two proposers yields the detailed argument. Consider λ_1 being the first proposer to finish phase two for step s, and consider the very next proposer[1] λ_2 that also completes phase two for the same step s. Note that the tag used by λ_2, $(s\ t_2)$, must be greater than the tag $(s\ t_1)$ used by λ_1. There must be a common acceptor α that answered λ_2 in phase one of λ_2 and answered λ_1 in phase two of λ_1. Since $t_1 < t_2$, α has accepted the consensus value of λ_1 before replying to λ_2 and in turn λ_2 has received the consensus value of λ_1 along with the tag $(s\ t_1)$, which leads to λ_2 using the consensus value of λ_1 in phase two of λ_2.

The liveness property, however, is not guaranteed. This is obvious since the authors in [7] have proven that the Consensus Problem is impossible in asynchronous systems prone to at least one crash failure. However, a close look at the behaviour of Paxos shows that only the liveness property cannot be guaranteed and why it is so. Indeed, since every proposer tries to produce a tag that is lexicographically greater than a majority of the acceptor tags, two such proposers may execute many trials for the same step without ever succeeding to complete a phase two. For example, the first proposer succeeds in phase one within step 0 with tag (0 0), and then the second proposer succeeds in phase one with tag (0 1). When the first proposer executes its phase two with tag (0 0), it cannot recruit a majority of acceptors, and hence its phase two fails. The first proposer then increments its tag to (0 2), executes a phase one and succeeds. Now the second proposer cannot succeed in its phase two with the tag (0 1); it has to increment its tag to (0 3) and executes another phase one. This execution can be iterated, and it shows that none of the two proposers is able to complete a phase two, i.e., no decision is ever taken for step 0. Intuitively though, it is clear that if there is a single proposer in the system during a long enough period of time, then the processors decide in every step, up to the last step the proposer is involved in.

[1] In case where only one proposer has executed a successful phase two, then the safety property for this step is trivial.

2.2 Practical Self-Stabilization

2.2.1 Bounded Integers and Self-Stabilization

As we pointed out in the previous section, the Paxos algorithm uses unbounded integers to tag data. 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. Hence, assuming that the integers are theoretically unbounded is reasonable only when it is ensured, in practice, that every step and trial variables are initially set to low values.

In the context of self-stabilization, however, a transient fault may produce fake decision messages in the communication channels, or make an acceptor accepting a consensus value that was not proposed. Such transient faults only break the Repeated Consensus conditions punctually and nothing can be done except waiting. However, a transient fault may also corrupt the Paxos step and trial variables 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 Repeated Consensus conditions are never jointly satisfied. This issue is much worrying than punctual breakings of the Repeated Consensus specifications.

Intuitively though, if one can manage to get every integer variable (step and trials) to be reset to low values at some point in time, then there is consequently a finite execution (ending with step or trial variables reaching the maximum value $2^{\mathfrak{b}}$) during which the system behaves like an initialized original Paxos execution that satisfies the Repeated Consensus Problem conditions[2]. Since we use bounded integers, we cannot prove the execution to be infinite, but we can prove that this execution is as long as counting from 0 to $2^{\mathfrak{b}}$, which is, in practice, as long as the length of execution assumed in the original Paxos prior to exhausting the counters. This is what we call *practical self-stabilization*.

[2] Modulo the unavoidable punctual breakings due to, e.g., fake decision messages.

2.2.2 What our Algorithm does

Clearly, most of the Paxos properties relies on a clever management of the tags $(s\ t)$ at each processor. Since there might be tags with step and trial values near the maximum $2^{\mathfrak{b}}$, there must be a mechanism to reset these fields to zero. In the sequel (Section 3), we define a new kind of tag that encapsulates the Paxos tag. We can associate such a tag with a specific step and trial value, and two tags can be compared. Our algorithm is an adaptation of the Paxos algorithm that uses this new tag structure.

As in the original Paxos algorithm, there are two functions, ν^s and ν^t, to increment a tag. First, we can increment the step, $b \leftarrow \nu^s(b)$, which can be translated in the original Paxos as the operation $(s\ t) \leftarrow (s+1\ 0)$ of moving to the next consensus instance. Or we can increment the trial, $b \leftarrow \nu^t(b)$, which can be translated as the operation $(s\ t) \leftarrow (s\ t+1)$ of moving to the next trial within the same consensus instance. These increment functions[1] implement a kind of reset mechanism that is triggered only when necessary. At each processor, the current tag undergoes a sequence of modifications (increments, or adoption of a tag from another processor, ...). If one associates with this sequence of tags $(b^i)_{i\in\mathbb{N}}$, the corresponding sequence of original Paxos tags $(s^i\ t^i)_{i\in\mathbb{N}}$, then there will be, more or less occasionally, several time instants at which the Paxos tags $(s^i\ t^i)$ do not behave as in Paxos, e.g., a reset of the variable, jumps, and alike. Between two such interrupts, the projected sequence of Paxos tags $(s^i\ t^i)$ behaves exactly as in Paxos.

The first main theorem (Section 6, Theorem 1) states that there exists a processor λ that eventually performs a finite execution σ within which the sequence of λ's tags corresponds to a sequence of Paxos tags that begins with low step and trial values, and ends with the step or the trial reaching the maximum value $2^{\mathfrak{b}}$. In other words, σ is a practically infinite execution within which everything seems like in the original Paxos, at least from the point of view of λ. The second main theorem (Section 6, Theorem 3) makes the link between the local execution σ occuring at the processor λ and the safety property on the global system. Roughly saying, it states that, within a practically infinite period of time, if two processors decide on values p_1 and p_2, with tags b_1 and b_2 respectively, such that b_1 and b_2 points to the same step s, then either $p_1 = p_2$ or one of the decision event happens after a practically infinite period of time.

3. TAG SYSTEM

This section is divided into two parts. In the first part, we informally describe the actual tag system we will be using in the algorithm. However, for didactic reasons, we first describe a simpler tag system that works when there is a single proposer, before adapting it to the case of multiple proposers. The second part formally defines the notions of bounded integer, label and tag.

3.1 Motivation

3.1.1 Single Proposer

We start by looking at Paxos tags $(s\ t)$ where the step s and trial t variables are integers bounded by a large constant $2^{\mathfrak{b}}$. Assume, for now, that there is a single proposer in the system, and let's focus on its tag. The goal of this proposer is to succeed in imposing a consensus value for every step ranging from 0 to $2^{\mathfrak{b}}$, or at least from a low step value to a very high step value. The proposer can do a step increment, $(s\ t) \leftarrow (s+1\ 0)$, or a trial increment within the same step, $(s\ t) \leftarrow (s\ t+1)$. To impose some value in step s, it must reach a trial t such that the tag $(s\ t)$ is lexicographically greater than every other processor tags in a majority of acceptors. Note that if the proposer reaches a step s such that every other processor are in a step lower than s, then the proposer will succeed in the trials $(s\ 0), (s+1\ 0), \ldots, (2^{\mathfrak{b}}\ 0)$. If every processor have tags with low step and trial values, then the proposer soon manages to produce a sequence of tags like $\sigma = (s\ 0), (s+1\ 0), \ldots, (2^{\mathfrak{b}}\ 0)$ starting with a low step value, i.e., a practically infinite sequence of steps.

With an arbitrary initial configuration, some processors may have tags with step or trial value set to the maximum $2^{\mathfrak{b}}$, thus the proposer will not be able to produce a greater tag, thus it cannot reach a sequence like σ. To deal with this issue, we introduce a third field, namely the label field, such that a tag is now defined by a triple $(l\ s\ t)$ where s and t are the step and trial fields, and l a label, which is not an integer but whose type is voluntarily not explicited here. We simply assume that it is possible to increment a label, and that two labels are comparable. The proposer has now three possibilities to increment its tag. It can first do a step increment, $(l\ s\ t) \leftarrow (l\ s+1\ 0)$ or a trial increment, $(l\ s\ t) \leftarrow (l\ s\ t+1)$; in both cases the label is constant. The proposer can also increment its label and resets its step and trial fields to zero, $(l\ s\ t) \leftarrow (l'\ 0\ 0)$ with l' greater than l. Incrementing the label can be thought as resetting the step and trial variable to zero. Now imagine that initially the label that the proposer uses is greater than every other label in the acceptor tags, and that the proposer step and trial values are equal to zero. Then, the proposer manages to succeed in a practically infinite number of steps that mimicks the behaviour of the original Paxos tags.

Therefore, for the proposer to reach a long sequence that mimicks the original Paxos, it is only needed to ensure that the proposer is always able to produce a label that is greater than any label in some finite set of labels, namely the labels of the acceptors. 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 labels. The history is only required to be large enough to contain every label in the system. When the proposer increments its label, it produces a label that is greater than every label in its history. Doing so, the proposer eventually manages to produce a label greater than every other label in the system, and starts an execution with initially low step and trial values that mimicks the behaviour of Paxos tag.

The label type cannot be an integer, otherwise we would postpone the problem of handling two bounded integers to the problem of handling three bounded integers. Actually, according to the previous remarks, 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

[1] Along with other mechanisms (see Section 3).

than every label in this subset. Such a device is called a finite labeling scheme. An implementation of such a finite labeling scheme was suggested in [1], and is formally presented in Appendix A.1. Roughly saying, a label is a fixed length vector of integers from a bounded domain in which the first integer is called sting and the others are called antistings. A label l_1 is greater than a label l_2, noted $l_1 \prec l_2$, if the sting of l_1 does not appear in the antistings of l_2 but not vice versa. Given a finite set of labels $l_1,\ldots,l_r$, we can build a greater label l by choosing a sting not present in the antistings of the l_i, and choosing the stings of the l_i as antistings in l.

3.1.2 Multiple Proposers

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 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 assume that the set of proposer identifiers is totally ordered and we define a tag to be a vector, say a, whose entries are indexed by the proposer identifiers. Each entry $a[\mu]$ of the tag a contains a tuple $(l\ s\ t\ id\ cl)$ where l is a label, s and t are step and trial bounded integers, id is the identifier of the proposer that owns the tag, and cl is either a label that cancels l or the null value[1] denoted by $\perp$. The identifier of the proposer that owns the tag is included, so that two proposers never share the same content in any entry of their respective tags. The canceling field tells the proposer whether the corresponding label has been canceled by some label. The role of cancellation is explained below.

Therefore, a proposer, say λ, has the possibility to use one of the entries of its tag, say a, to specify the step and trial it is involved in. However, the entry used must be valid, i.e., the entry must contain a null canceling field value along with step and trial values strictly less than the maximum value $2^{\mathfrak{b}}$. The entry actually used by the proposer is determined by the lowest proposer identifier μ such that the entry corresponding to μ is valid. The entry $a[\mu]$ is referred to as the first valid entry in the tag. If the first valid entry is located at the left of the entry indexed by the proposer identifier, i.e., the identifier μ is less than the proposer identifier λ, then the proposer can increment the step and trial values stored in the entry $a[\mu]$, but it cannot increment the label in the entry $a[\mu]$. The proposer can only increment the label, and thus reset the corresponding step and trial variables, stored in the entry indexed by its own identifier. In addition, whenever the entry indexed by the proposer identifier λ becomes invalid, the proposer λ produces a new label in the entry $a[\lambda]$ and resets the integer variables to zero and the canceling field to the null value $\perp$; this makes $a[\lambda]$ a valid entry in the proposer tag. The important point is that, from a global point of view, the proposer identified by λ is the only proposer to introduce new labels in the entries indexed by λ in tags of the system. Besides, this also shows that any proposer λ has to record in its label history only the canceling labels that are stored in the entry λ of tags.

A comparison relation is defined on tags so that every processor (proposer or acceptor) always try to use the valid entry with the lowest identifier. A tag b_1 is less than b_2 when either the first valid entry of b_1 is located at the right of the first valid entry of b_2, or both first valid entries are indexed by the same identifier μ and the tuple $b_1[\mu].(l\ s\ t\ id)$ is lexicographically less than the tuple $b_2[\mu].(l\ s\ t\ id)$. If there is no valid entry in both tags, or if the labels are not comparable, then the tags are not comparable.

3.2 Formal Definitions

Definition 1 (Bounded Integer) *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 $2^{\mathfrak{b}}$.*

Definition 2 (Finite Labeling Scheme) *A* finite labeling scheme *is a 4-tuple $\overline{\mathcal{L}} = (\mathcal{L}, \prec, d, \nu)$ where $\mathcal{L}$ is a finite set whose elements are called* labels, *$\prec$ is a partial relation on $\mathcal{L}$ 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., a function that maps any finite set of at most d labels to a label such that for every subset A in $\mathcal{L}$ of at most d labels, for every label l in A, we have $l \prec \nu(A)$. We denote the reflexive closure of $\prec$ by $\preccurlyeq$.*

Remark 1. The definition of a finite labeling scheme imposes that the relation $\prec$ is not transitive. Hence, it is not an order relation.

Definition 3 (Canceling Label) *Given a label l, a* canceling label *for l is a label cl such that $cl \not\preccurlyeq l$.*

Definition 4 (Tag System) *A tag system is given by a 4-tuple $(\mathfrak{b}, \Pi, \omega, \overline{\mathcal{L}})$ where $\mathfrak{b}$ is positive integer, Π is the totally ordered finite set of processor identifiers, ω is a special symbol such that $\omega \notin \Pi$ and $\overline{\mathcal{L}}$ is a finite labeling scheme. In addition the order on Π is extended as follows : for every $\mu \in \Pi$, $\mu < \omega$.*

Definition 5 (Tag) *Given a tag system $(\mathfrak{b}, \Pi, \omega, \overline{\mathcal{L}})$, a* tag *is a vector $a[\mu] = (l\ s\ t\ id\ cl)$ where μ and id are identifiers, l is a label, cl is either the null value noted $\perp$ or a canceling label for l, and s and t are $\mathfrak{b}$-bounded integers respectively called the* step *and* trial *fields. The* entry indexed by μ *in the tag a, or simply the entry μ in a, refers to the entry $a[\mu]$. The entry μ is said to be* valid *when the corresponding canceling field is null, $a[\mu].cl = \perp$, and both the corresponding step and trial values are strictly less than the maximum value, i.e., $a[\mu].s < 2^{\mathfrak{b}}$ and $a[\mu].t < 2^{\mathfrak{b}}$.*

Definition 6 (First Valid Entry) *Given a tag a, the first valid entry in the tag is defined by*

$$\chi(a) = \min\left(\{\mu \in \Pi \mid a[\mu] \text{ is valid}\} \cup \{\omega\}\right)$$

[1] Which means that the label l is not canceled.

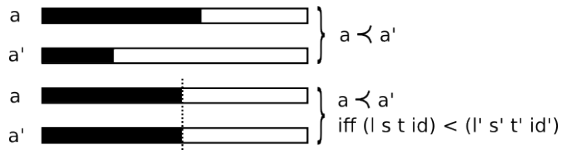

Figure 1: Comparison of tags

Definition 7 (Comparison of Tags) *Given two tags a and a', we note $a \prec a'$ when either $\chi(a) > \chi(a')$ or $\chi(a) = \chi(a') = \mu < \omega$ and[1] $a[\mu].(l\ s\ t\ id) < a'[\mu].(l\ s\ t\ id)$. We note $a \simeq a'$ when $\chi(a) = \chi(a')$ and $a[\chi(a)] = a'[\chi(a)]$. We note $a \preccurlyeq a'$ when either $a \prec a'$ or $a \simeq a'$.*

4. SYSTEM SETTINGS

All the basic notions we use (state, configuration, execution, asynchrony, ...) can be found in, e.g., [3, 10]. Here, the model we work with is given by a system of $\mathbf{n}$ asynchronous processors in a complete communication network. Each communication channel between two processors is a bidirectional asynchronous communication channel of finite capacity $\mathbf{C}$ [4].

Every processor has an identifier. The set 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[2] 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[3].

Consider an execution $E = (\gamma_k)_{0 \leq k < T}$. A *subexecution* F in E is the empty sequence or a segment $F = (\gamma_k)_{k_0 \leq k < k_1}$ $(0 \leq k_0 \leq k_1 \leq T)$ of consecutive configurations in E. The *local execution* at processor λ is the projected sequence $E(\lambda)$ of states of λ, i.e. $E(\lambda) = (\gamma_k(\lambda))_{0 \leq k < T}$. A *local subexecution* at λ in E is the empty sequence or the projected sequence of states $F(\lambda) = (\gamma_k(\lambda))_{k_0 \leq k < k_1}$ $(0 \leq k_0 \leq k_1 \leq T)$ from a subexecution F of E. A subexecution (resp. local subexecution) σ^1 is *included* in a subexecution (resp. local subexecution) σ^2 if it is a segment of consecutive configurations (resp. states) in σ^2.

We use the "happened-before" strict partial order introduced by Lamport [8]. Informally, if e and f are two events that occur on the same processor, then $e \rightsquigarrow f$ if and only if e occurs before f. If e is the sending of a message and f the reception of that message (not necessarily occuring on the same processor), then $e \rightsquigarrow f$. The full relation $\rightsquigarrow$ is the transitive closure of these basic cases. A formal presentation is presented in [8]. In our case, when $e \rightsquigarrow f$, we say that e happens before f, or f happens after[4] e.

The initial configuration of every execution is arbitrary and at most $\mathbf{f}$ processors are prone to crash failures. A quorum is any set of at least $\mathbf{n} - \mathbf{f}$ processors. For any execution E, we note $Live(E)$ the set of processors that do not crash during E, and we note $Crashed(E)$ the complement of $Live(E)$. We make the following resilience assumption.

Assumption 1 (Resilience) *The maximum number of crash failures $\mathbf{f}$ 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.*

In addition, every processor has access to a read-only boolean variable Θ_α, e.g., from an unreliable failure-detector (Section 7) that satisfies the following condition.

Assumption 2 (Module $\boldsymbol{\Theta}$) *For every infinite execution $E_\infty = (\gamma_k)_{k \in \mathbb{N}}$, there is a non-empty set $\mathcal{P}(E_\infty)$, namely the* proposers *in E_∞, of processors in $Live(E_\infty)$, such that*

$$\forall \lambda \in \mathcal{P}(E_\infty),\ \forall k \in \mathbb{N},\ \Theta_\lambda^k = \mathbf{true} \tag{1}$$

$$\begin{aligned} \forall \lambda \in\ & Live(E_\infty)\ s.t.\ \lambda \notin \mathcal{P}(E_\infty) \\ & \exists k_0,\ \forall k \geq k_0,\ \Theta_\lambda^k = \mathbf{false} \end{aligned} \tag{2}$$

where Θ_α^k denotes the value of the read-only variable Θ_α in the configuration γ_k.

5. THE ALGORITHM

In this section, we describe the self-stabilizing Paxos algorithm. We first present the variables before giving an overview of the algorithm. The last section describes the details of the algorithm. The pseudo-code of the algorithms is given in Appendix A.3. In the sequel, we refer to the following datastructure.

Definition 8 (Fifo History) *A fifo history H of size d on a set V, is a vector of size d of elements of V along with an operator $+$ defined as follows. Let $H = (v_1, \ldots, v_d)$ and v an element in V. If v does not appear in H, then $H + v = (v, v_1, \ldots, v_{d-1})$, otherwise $H + v = H$.*

We define the *tag storage limit* $\mathbf{K}$ and the *canceling label storage limit* $\mathbf{K}^{cl}$ by $\mathbf{K} = \mathbf{n} + \mathbf{C}\frac{\mathbf{n}(\mathbf{n}-1)}{2}$ and $\mathbf{K}^{cl} = (\mathbf{n}+1)\mathbf{K}$. We consider a tag system $(\mathfrak{b}, \Pi, \omega, \mathcal{L}, \prec, d, \nu)$ such that Π is the set of processor identifiers and the labeling scheme dimension is equal to $(\mathbf{K}+1)\mathbf{K}^{cl}$.

5.1 Variables

The state of a processor α is defined by the following variables : the *processor tag* a_α, the *processor proposal* p_α (a consensus value), the *canceling label history* H_α^{cl} (fifo label history of size $\mathbf{M} = (\mathbf{K}+1)\mathbf{K}^{cl}$), the *accepted proposal record* r_α and the *label history* H_α described as follows. The accepted proposal record r_α is a vector indexed by the processor identifiers. For each identifier μ, the field $r_\alpha[\mu]$ contains either the null value $\perp$ or a couple composed of a tag and

[1] Lexicographical comparison using the corresponding relation on labels, integers and processor identifiers.
[2] For a formal definition, refer to, e.g., [3, 10].
[3] For sake of simplicity, the events and the transitions are omitted.
[4] Note that the sentences "f happens after e" and "e does not happen before f" are not equivalent.

consensus value. The variable H_α is a vector indexed by the processor identifiers. For each identifier μ, the field $H_\alpha[\mu]$ is a fifo label history of size $\mathbf{K}$.

5.2 Tag Procedures

Algorithm 1 defines a procedure `clean` that cleans the canceling fields of a given tag as follows. The procedure takes as input a processor identifier λ and a tag a. After the completion of the procedure, for every entry μ in the tag a, if the canceling field $a[\mu].cl$ is not null, then its value is a canceling label for the label in $a[\mu].l$. In addition, every identifier value in $a[\mu].id$ is equal to λ. The second procedure `fill_cl` updates the canceling fields of two given tags x and y as follows. After the completion of the procedure, for any $\mu \in \Pi$, if the label $x[\mu].l$ or $x[\mu].cl$ (not equal to $\perp$) cancels $y[\mu].l$, then $y[\mu].cl$ is not null. And if $x[\mu].l = y[\mu].l$ with one of the integer fields in $x[\mu]$ being equal to the maximum value $2^{\mathfrak{b}}$, then both step and trial fields $y[\mu].cl$ are equal to $2^{\mathfrak{b}}$. The previous remarks also hold when exchanging x and y.

Algorithm 2 defines the function `check_entry` whose arguments are a processor identifier λ, a tag x, and an history of labels L. This function checks whether the entry $x[\lambda]$ is valid or not. If this entry is invalid, it stores the label value $x[\lambda].l$ in the history L, produces[1] a new label in $x[\lambda]$ with the labels in the history L and resets the step and trial fields to zero. Algorithm 2 also defines the step increment function, ν^s, and the trial increment function, ν^t. Both functions arguments are a processor identifier λ, a tag x, and a fifo history of labels L, and they both return a tag (the incremented tag). First, a copy y of the tag x is created. Then the tag y is cleaned with the procedure `clean`. The step increment function then increments the step in the first valid entry of y and resets the corresponding trial field to zero. The trial increment function only increments the trial field in the first valid entry of y. Then, in both functions, it is checked whether the entry $y[\lambda]$ is valid or not, and updated accordingly thanks to the function `check_entry`. Both functions return the tag y.

5.3 Algorithm Overview

Each processor can play two roles, namely, the acceptor role and the proposer role. A processor α plays both[2] the acceptor role and the proposer role as long as Θ_α is equal to **true**. When Θ_α is equal to **false**, the processor α only plays the acceptor role.

The current step and trial of a processor are determined by the step and trial values in the first valid entry of its tag. A proposer tries to impose some proposal for its current step. To do so, it executes the following two phases (cf. Algorithm 4).

(Phase 1). The proposer, say λ, reads a new proposal and tries to recruit a quorum of acceptors by broadcasting a message (phase 1, message $p1a$) with its tag a_λ (Algorithm 4, line 7). It waits for the replies from a majority of acceptors. When an acceptor α receives this $p1a$ message, it either adopts the proposer tag if the proposer tag is greater than its own tag a_α, or leaves its tag unchanged otherwise. The acceptor replies (phase 1, message $p1b$) to the proposer with its tag (updated or not) and the proposal, either null or a couple (tag, consensus value), stored in its accepted proposal variable $r_\alpha[\chi(a_\alpha)]$.

Upon receiving the acceptor replies, the proposer λ knows if it has managed to recruit a majority of acceptors. In that case, the proposer λ can move to the second phase. Otherwise, λ has received at least one acceptor reply whose tag is not less than or equal to the proposer tag of λ. At each reception of such an acceptor tag, the proposer λ modifies its tag in order for the proposer tag to be greater than the acceptor tag received. When messages are received from at least half of the processors, the proposer begins another phase one with its updated tag.

(Phase 2). When the proposer λ reaches this point, it has managed to recruit a quorum of acceptors and it knows all the latest proposals that they accepted for the entry $\chi(a_\lambda)$. Assume for instance that the proposer tag points to step s, i.e., the step value in the first valid entry μ of the proposer tag is equal to s. Then (Algorithm 4, line 11 to line 23) the proposer λ first checks that the tags associated with the received proposals all share the same first valid entry and the same corresponding label as the tag of λ. If it is not the case, then λ moves to the next step, i.e., it uses the step increment function to update its tag a_λ and goes to phase one. Otherwise, it looks for non-null proposals for step s and if there are some, it copies the proposal with the maximum tag (among those that point to step s) in its proposal variable. If there are more than two different proposals associated with this maximum tag, then λ increments its proposer tag with the step increment function and starts a new phase one. In any other case, it keeps its original proposal.

Next, the proposer λ sends to all the acceptors a message (phase 2, message $p2a$) containing its tag along with the proposal it has computed (Algorithm 4, line 26) and waits for the replies of a majority of acceptors. When an acceptor α receives this $p2a$ message, if the proposer tag is greater than or equal to its own tag, then the acceptor adopts the proposer tag and stores the proposal in the variable r_α. Otherwise, the acceptor leaves its tag and the accepted proposals record unchanged. Next, it replies (phase two, message $p2b$) to the proposer with its tag (updated or not).

After having received the replies from a majority of acceptors, the proposer λ knows if a majority have accepted its proposal. In that case, it broadcasts a decision message containing its proposer tag and the successful consensus value (Algorithm 4, line 28). At the reception of this message, any acceptor with a tag less than or equal to the proposer tag decides on the given proposal. The proposer λ can then move to the next step. Otherwise, the proposer λ has received tags that are not less than or equal to the proposer tag, and thus λ updates its proposer tag accordingly, and starts another phase one.

Remark 2. By "α adopts the tag b", we mean that α copies the content of the first valid entry in b to the same entry in

[1] With the label increment function from the finite labeling scheme (cf. Definition 2).

[2] One can think of having two threads on the same processor.

α's acceptor tag[1], i.e., $a_\alpha[\chi(b)] \leftarrow b[\chi(b)]$. Furthermore, every time a processor α modifies its tag, it also does the following. If the label l, in some entry μ of the tag, is replaced by a new label, then the label l is stored in the label history $H_\alpha[\mu]$ that corresponds to the identifier μ and a label that cancels the new label is looked for in the label history $H_\alpha[\mu]$, updating the corresponding canceling field accordingly. If the label in the entry α, i.e., the only entry in which the proposer α can create a label, gets canceled, then the associated canceling label is stored in the canceling label history H_α^{cl}. Any new label produced in the entry α of the tag at processor α is also stored in H_α^{cl}. In addition, for every μ, the accepted proposal $r_\alpha[\mu]$ is cleared, i.e., $r_\alpha[\mu] \leftarrow \perp$, whenever there is a label change in the entry $a_\alpha[\mu]$. A non-null field $r_\alpha[\mu] = (b, p)$ is also cleared whenever the label in the entry $b[\mu]$ is different than the label in the entry $a_\alpha[\mu]$, or the labels are equal but the entry $b[\mu]$ is lexicographically greater than the entry $a_\alpha[\mu]$. Finally, any processor α always checks that the entry α of its tag is valid, and updates it accordingly with the function `check_entry`.

Remark 3. Note that the relation $\prec$ on labels is not an order (neither a partial order), since it is not transitive. This implies the existence of cycle of labels like, e.g., $l_1 \prec l_2 \prec l_3 \prec l_1$. Indeed, the purpose of the label history $H_\alpha[\mu]$ is to avoid such cycling on labels in the entry μ. The mechanism ensures that, in any entry μ of the tag, the label field cannot undergo a cycle of labels whose length is less than the tag storage limit of the system. However, longer cycles are possible but this implies that the processor μ has produced new labels. This is due to the fact that no processor α ever adopts in any entry μ of one of its tag, a label that comes from the label history $H_\alpha[\mu]$. The label history $H_\alpha[\mu]$ only serves as a source of canceling labels for the entry μ.

5.4 Algorithm Details.

We focus on the reception of a proposer message by an acceptor (Algorithm 3). Say an acceptor α receives a message $\langle p1a, \lambda, b\rangle$ from proposer λ. The acceptor α first records in the canceling label history H_α^{cl} any label in the entry $b[\alpha]$ that cancels the label $a_\alpha[\alpha].l$ in the acceptor tag (line 4). Using the procedure `fill_cl` presented in Algorithm 1, the acceptor α updates the canceling fields of both tags a_α and b. Then, it checks the validity of the entry $a_\alpha[\alpha]$ with the procedure `check_entry` and updates it accordingly (line 5). If the updated tags satisfy $a_\alpha \prec b$, then α adopts the tag b, i.e., it copies the content of the first valid entry $b[\chi(b)]$ to the entry $a_\alpha[\chi(b)]$ in a (line 7). If there has been a change of label in the entry $a_\alpha[\chi(b)]$, then the accepted proposal variable $r_\alpha[\chi(b)]$ is cleared, the old label is stored in the history $H_\alpha[\chi(b)]$, and α looks in this history for labels that cancel the new label $a_\alpha[\chi(b)].l$, updating the corresponding canceling field accordingly (lines 9 to 11). Next, the acceptor checks for every identifier μ if either the tag b in the accepted proposal $r_\alpha[\mu]$ uses a label different than the label in the entry $a_\alpha[\mu]$, or if the tuple $a_\alpha[\mu].(l\ s\ t\ id)$ is less than the tuple $b[\mu].(l\ s\ t\ id)$; in such a case, the entry $r_\alpha[\mu]$ is cleared. In any case, whether it adopts the tag b or not, the acceptor α replies to the proposer λ with a message $\langle p1b, \alpha, a_\alpha, r_\alpha[\chi(a_\alpha)]\rangle$ where a_α is its updated (or not) acceptor tag and $r_\alpha[\chi(a_\alpha)]$ is the lastly accepted proposal for the entry $\chi(a_\alpha)$ (line 18).

When an acceptor α receives a $p2a$ message or a decision message containing a proposal (b, p), the procedure is similar. It first updates the canceling label history H_α^{cl}, the canceling fields of a_α and b, and checks the validity of the entry $a_\alpha[\alpha]$ (lines 21 and 22). The difference with the previous case is that the condition to accept the proposal (b, p) is $a_\alpha \preccurlyeq b$. In this case, the acceptor α adopts the tag b, updating the canceling field and the label history as in the case of a $p1a$ message, stores the couple (b, p) in its accepted proposal variable $r_\alpha[\chi(b)]$ and, in case of a decision message, decides on the couple (b, p) (lines 24 and 25). In addition, if there has been a change of label in the entry $a_\alpha[\chi(b)]$, then[2] the old label is stored in the history $H_\alpha[\chi(b)]$, and α looks in this history for labels that cancel the new label $a_\alpha[\chi(b)].l$, updating the corresponding canceling field accordingly (lines 27 and 28). We say that the acceptor α has accepted the proposal (b, p). Next, the acceptor checks for every identifier μ if either the tag b in the accepted proposal $r_\alpha[\mu]$ uses a label different than the label in the entry $a_\alpha[\mu]$, or if the tuple $a_\alpha[\mu].(l\ s\ t\ id)$ is less than the tuple $b[\mu].(l\ s\ t\ id)$; in such a case, the entry $r_\alpha[\mu]$ is cleared. In case of a $p2a$ message, whether it accepts the proposal or not, the acceptor α replies to the proposer λ with a message $\langle p2b, \alpha, a_\alpha\rangle$ containing its updated (or not) acceptor tag (line 35). In case of a decision message, the acceptor does not reply.

At the end of any phase, a proposer executes a procedure named the preempting routine (Algorithm 5) that mainly consists in waiting for the replies from a majority of acceptors and suitably incrementing the proposer tag. The phase is considered successful if the routine returns *ok* and failed otherwise. In this routine, the processor λ waits for $\mathbf{n} - \mathbf{f}$ replies from the acceptors. Note that, although the pseudo-code suggests λ receives only acceptor replies (Algorithm 5, line 5), the processor λ, as an acceptor, also processes messages ($p1a$ or $p2a$) from other proposers. The variable a_{sent} stores the value of a_λ that λ has sent at the beginning of the phase. The variable b is an auxiliary variable that helps filtering messages and is reset to a_{sent} at the beginning of each new loop (line 4). For each message with tag a_α and proposal r_α received from a processor α, the procedure updates the canceling fields of both b and a_α (line 6).

If the current phase is a phase one, then a reply is considered positive when the acceptor α has adopted the tag λ sent, i.e., when[3] $a_\alpha \simeq b$. If the current phase is a phase two, the reply is considered positive when the acceptor α has adopted the tag λ has sent and has accepted the corresponding proposal, i.e, $a_\alpha \simeq b$ and $p_\lambda = r_\alpha[\chi(b)].p$. The condition C^+ (line 7) summarizes these two cases. A reply is considered negative when the received acceptor tag is not less than or equal to the tag the proposer λ has sent, i.e., a an acceptor tag a_α such that $a_\alpha \not\preccurlyeq b$ (condition C^-, line 8). The procedure discards any acceptor reply that does not satisfy the conditions C^+ nor C^-. The variable M counts the number of positive replies. The routine returns *ok* if all the replies are positive,

[1]Note that only the entry $a_\alpha[\chi(b)]$ is modified. In fact, we have $a_\alpha \simeq b$ and not $a_\alpha = b$.

[2]In this case, the variable $r_\alpha[\chi(b)]$ is not cleared.

[3]Recall that $a_\alpha \simeq b$ means $\chi(a_\alpha) = \chi(b)$ and $a_\alpha[\chi(b)] = b[\chi(b)]$

i.e., $M = \mathbf{n} - \mathbf{f}$, and *nok* otherwise (lines 36 and 37).

At each negative reply received, the routine updates the variable a_λ so that it is always greater than the tag received. Precisely, it updates the canceling label history H^{cl}_λ (line 13), the canceling fields of a_α and a_λ (line 14) and checks the validity of the entry $a_\lambda[\lambda]$ (line 15). Recall that this implies $\chi(a_\lambda) \leq \lambda$. Then, the routine checks if a_α is less than or equal to a_λ. If it is so, then the routine does not modify a_λ. Otherwise (lines 16 to 30), it checks if a_α has its first valid entry located at the left of a_λ's first valid entry, i.e., $\chi(a_\alpha) < \chi(a_\lambda)$. In that case, the content of the entry $a_\alpha[\chi(a_\alpha)]$ is copied[1] to the entry $a_\lambda[\chi(a_\alpha)]$ and the trial value is incremented. In addition, the previous label in $a_\lambda[\chi(\alpha)].l$ is stored in the label history $H_\lambda[\chi(\alpha)]$ and possible canceling labels for the new label in $a_\lambda[\chi(\alpha)].l$ are searched for in $H_\lambda[\chi(\alpha)]$ (lines 19 to 22). If the first valid entry $\chi(a_\alpha)$ in a_α is not located at the left of a_λ's first valid entry, then necessarily $\chi(a_\alpha) = \chi(a_\lambda) = \mu$, since $a_\alpha \not\preccurlyeq a_\lambda$. In that case, the routine compares the content of the entries indexed by μ in a_α and a_λ (lines 24 to 30). Note that, since the routine has updated the canceling fields, the corresponding labels are equal[2]. If both entries $a_\alpha[\mu]$ and $a_\lambda[\mu]$ share the same step value, then a_λ is updated with the time increment function ν^t (line 27). Otherwise, the step increment function is used (line 30).

6. PROOFS

6.1 Basics

Lemma 1 *Any phase of the proposer algorithm eventually ends.*

PROOF. Let ϕ be a phase executed by some proposer λ. At the beginning of ϕ, the proposer λ has broadcast a message with its proposer tag a_λ, along with a consensus value p in case of a $p2a$ message. Assumption 1 (Section 4) ensures that at least $\mathbf{n} - \mathbf{f}$ acceptors eventually reply. The only reason why ϕ would be endless is λ discarding real replies from these acceptors in the preempting routine. For each such acceptor α, when it receives the message sent by λ, it first updates the canceling fields in a_α and a_λ. Let a, b respectively be the updated versions of a_α, a_λ, and $\mu = \chi(a)$. According to the acceptor Algorithm 3, if the acceptor α adopts the tag b then we have $a \simeq b$, and in case of a $p2a$ message, it also accepts the consensus value, i.e., $r_\alpha[\chi(a)] = (b, p)$; otherwise, we must have $a \not\preccurlyeq b$. These two cases correspond exactly to the conditions C^+ and C^- in the Algorithm 5. In other words, real replies are not discarded by λ, and since there are at least $\mathbf{n} - \mathbf{f}$ such replies, phase ϕ eventually ends. □

Given any configuration γ of the system and any processor idenditifer μ, let $S(\gamma)$ and $S^{cl}(\mu, \gamma)$ be two sets as follows. The set $S(\gamma)$ is the set of every tag present either in a processor memory or in some message in a communication channel, in the configuration γ. The set $S^{cl}(\mu, \gamma)$ denotes the collection 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.

Lemma 2 (Storage Limits) *For every configuration γ and every identifier μ, 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 $\mathbf{K}$.*

PROOF. Consider a configuration γ. For each processor α, there is one tag value (tag a_α) in the processor state $\gamma(\alpha)$ of α. For each communication channel (α, β), there are at most $\mathbf{C}$ different messages in the channel state $\gamma(\alpha, \beta)$; all these messages have one tag each. Hence, the maximum number of tags present in the configuration γ is $\mathbf{n}$ plus $\mathbf{C}$ times the number of communication channels. The network being complete, the number of communication channels is $\mathbf{C}\frac{\mathbf{n}(\mathbf{n}-1)}{2}$, thus we have $\mathbf{K} \geq |S(\gamma)|$. For every α, the maximum size of the history $H_\alpha[\mu]$ is $\mathbf{K}$. Hence, the size of $S^{cl}(\mu, \gamma)$ is bounded above by $\mathbf{K}$ (labels $x[\mu].l$ for x in $S(\gamma)$) plus $\mathbf{K}$ times the number of processors (labels from $H_\alpha[\mu]$ for every processor α), i.e., $(\mathbf{n}+1) \cdot \mathbf{K} = \mathbf{K}^{cl}$. □

6.2 Tag Stabilization - Definitions

Definition 9 (Interrupt) *Let λ be any processor and consider a local subexecution $\sigma = (\gamma_k(\lambda))_{k_0 \leq k \leq k_1}$ at λ. We note a^k_λ for the value of λ's tag in $\gamma_k(\lambda)$. We say that an interrupt has occurred at position k in the local subsexecution σ when one of the following happens*

- *$\mu < \lambda$, type $[\mu, \leftarrow]$: the first valid entry moves to μ such that $\mu = \chi(a^{k+1}_\lambda) < \chi(a^k_\lambda)$, or the first valid entry does not change but the label does, i.e., $\mu = \chi(a^{k+1}_\lambda) = \chi(a^k_\lambda)$ and $a^k_\lambda[\mu].l \neq a^{k+1}_\lambda[\mu].l$.*
- *$\mu < \lambda$, type $[\mu, \rightarrow]$: the first valid entry moves to μ such that $\mu = \chi(a^{k+1}_\lambda) > \chi(a^k_\lambda)$.*
- *type $[\lambda, \max]$: the first valid entry is the same but there is a change of label in the entry λ due to the step or trial value having reached the maximum value $2^{\mathfrak{b}}$; we then have $\chi(a^{k+1}_\lambda) = \chi(a^k_\lambda) = \lambda$ and $a^k_\lambda[\lambda].l \neq a^{k+1}_\lambda[\lambda].l$.*
- *$[\lambda, cl]$: 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 $\chi(a^{k+1}_\lambda) = \chi(a^k_\lambda) = \lambda$ and $a^k_\lambda[\lambda].l \neq a^{k+1}_\lambda[\lambda].l$.*

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

Remark 4. 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 $a_\lambda[\mu].l$. In addition, the new label l' is greater than the previous label l, i.e., $l \prec l'$. Also note that, if $\chi(a^k_\lambda) = \lambda$, the proposer λ never copies the content of the entry λ of a received tag, say a, to the entry λ of its proposer tag, even if $a^k_\lambda[\lambda].l \prec a[\lambda].l$. 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^{cl}_λ.

[1]Note that since the canceling fields have been updated with the procedure `fill_cl`, necessarily the labels $a_\alpha[\chi(\alpha)].l$ and $a_\lambda[\chi(\alpha)].l$ are different.

[2]Otherwise, one would cancel the other and contradict the definition of the first valid counter.

Definition 10 (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, every tag 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 a and a' satisfies $\chi(a) = \chi(a') = \mu$ and $b[\mu].l = b'[\mu].l$. We note μ_σ and l_σ for the first valid entry and associated label common to all the tag values in σ.*

Definition 11 (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 one of the integer fields in $a_\lambda[\mu_\sigma]$ having reached the maximum value $2^\mathfrak{b}$. 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 the step and trial values in $x[\mu_\sigma].l$ have values less than or equal to h.*

Remark 5. 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 step and trial value in $b_\lambda[\lambda]$ are equal to zero. However, other processors may already be using the label l_σ with arbitrary corresponding step and trial values. The definition of a h-safe epoch ensures that the epoch is truly as long as counting from h to $2^\mathfrak{b}$.

6.3 Tag Stabilization - Results

Lemma 3 *Let λ be any processor. Then the first valid entry of its proposer tag is eventually always located at the left of the entry indexed by λ, i.e., $\chi(a_\lambda) \leq \lambda$.*

Proof. This comes from the fact that whenever the entry $a_\lambda[\lambda]$ is invalid, the processor λ produces a new label in $a_\lambda[\lambda]$ and resets the step, trial and canceling field (cf. procedure `check_entry`, Algorithm 2). Once $\chi(a_\lambda) \leq \lambda$, every consequent tag values is obtained either with the step or trial increment functions (ν^s or ν^t), or by copying the content of a valid entry $\mu < \lambda$ of some tag to the entry $a_\alpha[\mu]$. Hence the first valid entry remains located before the entry λ. □

Remark 6. Thanks to this lemma, for every processor λ, it is now assumed, unless stated explicitly, that the entry $\chi(a_\lambda)$ is always located before the entry λ, i.e., $\chi(a_\lambda) \leq \lambda$.

Lemma 4 (Cycle of Labels) *Consider a subexecution E, a processor λ and an entry $\mu < \lambda$ in the tag variable a_λ. The label value in $a_\lambda[\mu].l$ can change during the subexecution E and we note $(l^i)_{1 \leq i \leq T+1}$ for the sequence of successive distinct label values that are taken by the label $a_\lambda[\mu].l$ 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 a_λ, the entry μ becomes invalid.*

Proof. First note that a processor adopts a new label in the entry μ of one of its tag, only when the old label is less than the new label. Hence, we have for every $1 \leq i \leq T$, $l^i \prec l^{i+1}$ and, in particular, if $l^1 = l^{T+1}$, $l^2 \not\prec l^{T+1}$. Assume $T > \mathbf{K}$. Since in every configuration there is at most $\mathbf{K}$ tags in the system, and μ is the only source of labels in the entry μ, the fact that λ has seen more than $\mathbf{K}$ different label values in the entry μ is possible only if μ has produced at least one label during E. If $T \leq \mathbf{K}$ and $l^1 = l^{T+1}$, i.e., there is a cycle of length T, then when λ adopts the label $l^{T+1} = l^1$, the label history $H_\lambda[\mu]$ contains the whole sequence $l^1, \ldots, l^T$ since its size is $\mathbf{K}$. Hence, λ sees the label l^2 that cancels the label l^{T+1}, and the entry μ becomes invalid. □

Lemma 5 (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*

$$|[\mu, \leftarrow]| \leq R_\mu = (J_\mu + 1) \cdot (\mathbf{K} + 1) - 1 \qquad (3)$$

where J_μ is the number of times the processor μ has produced a label since the beginning of the execution.

Proof. We note $(a_\rho^k)_{k \in \mathbb{N}}$ the sequence of ρ's tag values appearing in the local execution $E_\infty(\rho)$. Assume on the contrary that $|[\mu, \leftarrow]|$ is greater than R_μ. Note that after an interrupt like $[\mu, \leftarrow]$, the first valid entry $\chi(a_\rho)$ is equal to μ. In particular, the entry μ is valid after such interrupts. Also, the label value in the entry $a_\lambda[\mu].l$ does not change after an interrupt like $[\mu, \rightarrow]$. We define an increasing sequence of integers $(f(i))_{1 \leq i \leq R_\mu + 1}$ such that the i-th interrupt like $[\mu, \leftarrow]$ occurs at $f(i)$ in the sequence $(a_\rho^k)_{k \in \mathbb{N}}$. The sequence $l^i = a_\rho^{f(i)+1}[\mu].l$ is the sequence of distinct labels successively taken by $a_\rho[\mu].l$. We have $l^i \prec l^{i+1}$ for every $1 \leq i \leq R_\mu$.

Divide the sequence $(l^i)_{1 \leq i \leq R_\mu + 1}$ in successive segments u^j, $1 \leq j \leq J_\mu + 1$, of size $\mathbf{K}+1$ each. For any j, if all the $\mathbf{K}+1$ labels in u^j are different, then, by Lemma 4, the processor μ has produced at least one label. Since the processor μ produces labels at most J_μ many times, there is some sequence u^{j_0} within which some label appears twice. In other words, in u^{j_0} there is a cycle of length less than or equal to $\mathbf{K}$. By Lemma 4, this implies that the entry μ becomes invalid after an interrupt like $[\mu, \leftarrow]$; this is a contradiction. □

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 note $|\lambda|$ for the number of identifiers $\mu \leq \lambda$, J_μ for the number of times a proposer $\mu < \lambda$ produces a label and we define*

$$T_\lambda = (\sum_{\mu < \lambda} R_\mu + 1) \cdot (|\lambda| + 1) \cdot (\mathbf{K^{cl}} + 1) \cdot (\mathbf{K} + 1) \qquad (4)$$

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

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,* $E_c(\lambda) = \sigma^1 \dots \sigma^{T_\lambda}$. *Then* $E_c(\lambda)$ *contains a* 0*-safe epoch.*

PROOF. By Lemma 5, we have $\sum_{\mu<\lambda} |[\mu, \leftarrow]| \leq \sum_{\mu<\lambda} R_\mu$ in the local execution $E_\infty(\lambda)$, a fortiori in the execution $E_c(\lambda)$. By the pigeon-hole principle, there must be a local subexecution $E_1(\lambda) = \sigma^i \dots \sigma^{i+X-1}$ in $E_c(\lambda)$, where $X = (|\lambda| + 1) \cdot (\mathbf{K}^{cl} + 1) \cdot (\mathbf{K} + 1)$, that contains only interrupts like $[\mu, \rightarrow]$, $[\lambda, \max]$ or $[\lambda, cl]$. Naturally, the number of interrupts like $[\mu, \rightarrow]$ in $E_1(\lambda)$ is less than or equal to $|\lambda|$. Hence, another application of the pigeon-hole principle gives a local subexecution $E_2(\lambda) = \sigma^j \dots \sigma^{j+Y-1}$ in $E_1(\lambda)$ where $Y = (\mathbf{K}^{cl} + 1) \cdot (\mathbf{K} + 1)$ that contains only interrupts like $[\lambda, \max]$ or $[\lambda, cl]$.

Assume first that within $E_2(\lambda)$, there is a subexecution $E_3(\lambda) = \sigma^k \dots \sigma^{k+Z-1}$ where $Z = \mathbf{K} + 1$ in which there are only interrupts like $[\lambda, \max]$. Since $\mathbf{K} + 1 \leq \mathbf{M}$ the size of the canceling label history[1], we have $l_{\sigma^k}, \dots, l_{\sigma^{h-1}} \prec l_{\sigma^h}$, for every $k < h < k + Z$. In particular, all the labels $l_{\sigma^k}, \dots, l_{\sigma^{k+Z-1}}$ are different. Since $Z = \mathbf{K} + 1$ and since there is at most $\mathbf{K}$ tags in a given configuration, there is necessarily some $k \leq h < k+Z$ such that the label l_{σ^h} does not appear[2] in the configuration γ^* that corresponds to the last position in σ^{h-1}. Also, by construction, we have $\mu_{\sigma^h} = \lambda$ and σ^h ends with an interrupt like $[\lambda, \max]$. Hence, σ^h is 0-safe.

Now, assume that there is no subexecution E_3 in E_2 as in the previous paragraph. This means that if we look at the successive interrupts that occur during $E_2(\lambda)$, between any two successive interrupts like $[\lambda, cl]$, there is at most $\mathbf{K}$ interrupts like $[\lambda, \max]$. Since the length of $E_2(\lambda)$ is $(\mathbf{K^{cl}}+1) \cdot (\mathbf{K}+1)$, there must be at least $\mathbf{K^{cl}} + 1$ interrupts like $[\lambda, cl]$. Let $E_4(\lambda)$ be the local subexecution that starts with the epoch associated with the first interrupt like $[\lambda, cl]$ and ends with the epoch associated with the interrupt $[\lambda, cl]$ numbered $\mathbf{K}^{cl}$. Let σ in $E_2(\lambda)$ be the epoch right after $E_4(\lambda)$. By construction, there is at most $\mathbf{K}^{cl} \cdot (\mathbf{K} + 1)$ epochs in $E_4(\lambda)$ which is the size $\mathbf{M}$ of the history H_λ^{cl}. Hence, at the beginning of σ, the history H_λ^{cl} contains all the labels the processor λ has produced during E_4 as well as all the $\mathbf{K}^{cl}$ (exactly) labels it has received during E_4. Since there is at most $\mathbf{K^{cl}}$ candidates label for canceling in the system, necessarily, in the first configuration of σ, the history H_λ^{cl} contains every candidates label for canceling present in the whole system. And since l_σ is greater, by construction, than every label in the history H_λ^{cl}, l_σ was not present in the entry λ of some tag in the configuration that precedes σ and it cannot be canceled by any other label present in the the system. In addition, by construction, E_2 only contains interrupts like $[\lambda, \max]$ or $[\lambda, cl]$. From what we said about l_σ, the interrupt at the end of σ is necessarily $[\lambda, \max]$. In other words, the epoch σ is a 0-safe epoch. □

[1]Recall that the canceling label history also records the label produced in the entry λ.

[2]Note that λ is the only processor to produce labels in entry λ, so during the subexecution that correspond to an epoch σ^h at λ, the set of labels in the entry λ of every tag in the system is non-increasing.

Remark 7. 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 entries with high values in the step and trial fields) 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.

Remark 8. Besides, every processor α always checks that the entry α is valid, and, if not, it produces a new label in the entry $a_\alpha[\alpha]$ and resets the step, trial and canceling label field. Doing so, even if α's first valid entry μ is located before the entry α, 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.

6.4 Practical Safety - Definitions

To prove the safety property within a subexecution, we have to focus on the events that correspond to deciding a proposal, e.g., (b, p) at processor α. Such an event may be due to corrupted messages in the communication channels an any stage of the Paxos algorithm. Indeed, a proposer selects the proposal it will send in its phase two thanks to the replies it has received at the end of its phase one. 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. Hence, violations of the safety do not happen very often. To formally deal with these issues, we define the notion of scenario that corresponds to specific chain of events involved in the Paxos algorithm.

Definition 12 (Scenario) *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}. *We use the following notations*

- $\rho \xrightarrow{p1a} (S, b)$ *: The proposer* ρ *broadcasts a message* p1a *containing the tag* b. *Every acceptor in the quorum* S *receives this message and adopts*[3] *the tag* b.
- $(S, b) \xrightarrow{p1b} \rho$ *: Every processor* α *in the quorum* S *sends to the proposer* ρ *a* p1b *message telling they adopted the tag* b, *and containing the last proposal* $r_\alpha[\chi(a_\alpha)]$ *they accepted. These messages are received by* ρ.

[3]Recall that this means it copies the entry $b[\chi(b)]$ in the entry $a_\beta[\chi(b)]$.

- $\rho \xrightarrow{p2a} (Q,b,p)$ *: The proposer ρ broadcasts a p2a message containing a proposal (b,p). Every acceptor in the quorum Q accepts the proposal (b,p).*
- $(Q,b,p) \xrightarrow{p2b} \rho$ *: Every acceptor α in the quorum Q sends to the proposer ρ a p2b message telling that it has accepted the proposal (b,p). The proposer ρ receives these messages.*
- $\rho \xrightarrow{dec} (\alpha,b,p)$ *: the proposer ρ sends a decision message containing the proposal (b,p). The processor α receives this message, accepts and decides on the proposal (b,p).*

Definition 13 (Simple Acceptation Scenario) *Given S a quorum of acceptors, b a tag, p a consensus value, ρ a proposer and α an acceptor, a* simple acceptation scenario U of the first kind *is defined as follows.*

(U_0) *A proposer ρ broadcasts a p1a message with tag b.*

(U_1) *Every processor β from a quorum S receives this p1a message, adopts the tag b and replies to ρ a p1b message containing its tag $a_\beta \simeq b$ and the lastly accepted proposal $r_\beta[\chi(a_\beta)]$.*

(U_2) *The proposer ρ receives these messages at the end of its Paxos phase one, moves to the second phase of Paxos, and sends a p2a message to a processor α telling it to accept the proposal (b,p).*

(U_3) *The processor α receives the p2a message and accepts the proposal (b,p).*

Given quorums S and Q, b a tag, p a consensus value, ρ a proposer and α an acceptor, a simple acceptation scenario V of the second kind *is defined as follows.*

(V_0) *A proposer ρ broadcasts a p1a message with tag b.*

(V_1) *Every processor β from a quorum S receives this p1a message, adopts the tag b and replies to ρ a p1b message containing its tag $a_\beta \simeq b$ and the lastly accepted proposal $r_\beta[\chi(a_\beta)]$.*

(V_2) *The proposer ρ receives these messages at the end of its Paxos phase one, moves to the second phase of Paxos, and sends a p2a message to every processor in Q telling it to accept the proposal (b,p).*

(V_3) *Every processor in Q receives the p2a message, accepts the proposal and replies to the proposer ρ with a p2b message.*

(V_4) *The proposer ρ receives the replies from the acceptors in Q, and sends to the acceptor α a decision message containing a proposal (b,p).*

(V_5) *The acceptor α receives the decision message, accepts and decides on the proposal (b,p).*

With the notations introduced, we have

$$\textit{(1-st kind)}\ \rho \xrightarrow{p1a} (S,b) \xrightarrow{p1b} \rho \xrightarrow{p2a} (\alpha,b,p) \tag{5}$$

$$\textit{(2-nd kind)}\ \rho \xrightarrow{p1a} (S,b) \xrightarrow{p1b} \rho \xrightarrow{p2a} (Q,b,p) \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha,b,p) \tag{6}$$

If the kind of scenario is not relevant, we note

$$S \rightsquigarrow (\alpha,b,p) \tag{7}$$

Remark 9. 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 14 (Fake Message) *Given a subexecution $E = (\gamma_k)_{k_0 \leq k \leq k_1}$, a* fake message relatively 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.*

Remark 10. 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 15 (Simple Fake Acceptation Scenario) *Given a subexecution E, we note $\bigcirc \to X$ if there exists an event e in X that corresponds to the reception of a fake message relatively to E. With the previous notation, a* simple fake acceptation scenario relatively to *E is one of the following scenario.*

$$\bigcirc \xrightarrow{p2a} (\alpha,b,p) \tag{8}$$

$$\bigcirc \xrightarrow{p1b} \rho \xrightarrow{p2a} (\alpha,b,p) \tag{9}$$

$$\bigcirc \xrightarrow{dec} (\alpha,b,p) \tag{10}$$

$$\bigcirc \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha,b,p) \tag{11}$$

$$\bigcirc \xrightarrow{p2a} (Q,b,p) \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha,b,p) \tag{12}$$

$$\bigcirc \xrightarrow{p1b} \rho \xrightarrow{p2a} (Q,b,p) \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha,b,p) \tag{13}$$

If the exact type is not relevant, we note $\bigcirc \rightsquigarrow (\alpha,b,p)$.

Remark 11. A simple fake acceptation scenario is somehow similar to a simple acceptation scenario except the fact that at least one fake message (relatively to the given subexecution) is involved during the scenario.

Definition 16 (Composition) *Consider two simple scenarios $U = X \rightsquigarrow (\alpha_1,b_1,p_1)$, where $X = \bigcirc$ or $X = (S_1,b_1)$, and $V = S_2 \rightsquigarrow (\alpha_2,b_2,p_2)$ such that the following conditions are satisfied.*

- *The processor α_1 belongs to S_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 (b_1,p_1)" is the last acceptation event before e_2. In addition, the proposer involved in the scenario V selects the proposal (b_1,p_1) as the highest-numbered proposal at the beginning of the Paxos phase two. In particular, $p_1 = p2$.*

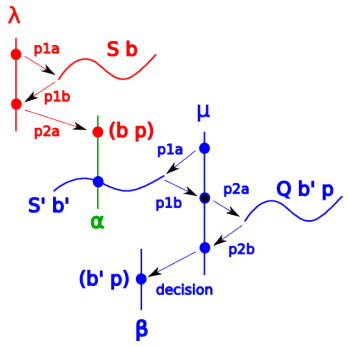

Figure 2: Composition of scenarios of the 1-st kind (red) and the 2-nd kind (blue) - Time flows downward, straight lines are local executions, arrows represent messages.

- *All the tags involved share the same first valid entry, the same corresponding label and step value.*

Then the composition *of the two simple scenarios is the concatenation the scenarios U and V. This scenario is noted*

$$X \rightsquigarrow (\alpha_1, b_1, p_1) \rightarrow S_2 \rightsquigarrow (\alpha_2, b_2, p_2) \quad (14)$$

Note that the trial value is strictly increasing along the simple scenarios.

Definition 17 (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 relatively to E. We note*

$$X \rightsquigarrow (\alpha_1, b_1, p) \rightarrow S_2 \rightsquigarrow (\alpha_2, b_2, p) \ldots S_r \rightsquigarrow (\alpha_r, b_r, p) \quad (15)$$

An acceptation scenario whose first simple scenario is not fake relatively to E is called real acceptation scenario relatively to E. *An acceptation scenario whose first simple scenario is fake relatively to E is called* fake acceptation scenario relatively to E. *Given an event e that corresponds to some processor accepting a proposal, we note $Sc(e)$ the set of acceptation scenarios that ends with the event e.*

Remark 12. 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 for the corresponding step. Besides, note that all the tags involved share the same first valid entry μ, the same corresponding label l, step value s and consensus value p. Also, the trial value is increasing along the acceptation scenario.

Definition 18 (Scenario Characteristic) *The* characteristic *of an acceptation scenario U in which all tags have first valid entry μ, corresponding label l, step value s and consensus value p, is the tuple $char(U) = (\mu, l, s, p)$.*

Definition 19 (Fake Characteristics) *Consider a subexecution $E = (\gamma_k)_{k_0 \leq k \leq k_1}$. Given a scenario characteristic (μ, l, s, p), we note $\mathcal{E}(E, \mu, l, s, p)$ the set of events in E that correspond to accepting a proposal (b, p) with $\chi(b) = \mu$ and $b[\mu].(l\ s) = (l\ s)$. A characteristic (μ, l, s, p) is said to be* fake relatively to E *if there exists an event e in $\mathcal{E}(E, \mu, l, s, p)$ such that the set $Sc(e)$ contains a fake acceptation scenario relatively to E. We note $\mathcal{FC}(E)$ the set of fake characteristics relatively to E.*

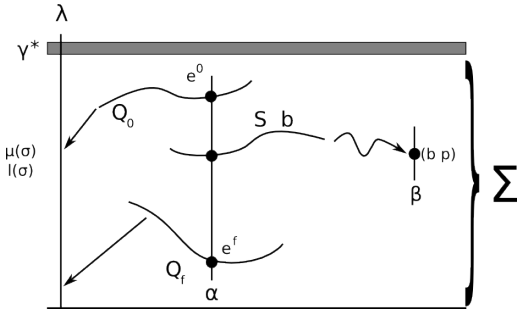

Figure 3: Scenario $S \rightsquigarrow (\beta, b, p)$ in $Z(F, \lambda, \sigma)$ - Time flows downward, straight lines are local executions, curves are send/receive events, arrows represent messages.

Definition 20 (Unsafe Steps) *If we fix the identifier μ and the label l, we define the set of* unsafe step *values $\mathcal{US}(E, \mu, l)$ as the set of values s such that there exists a consensus value p with $(\mu, l, s, p) \in \mathcal{FC}(E)$.*

Remark 13. Given an identifier μ and the label l, an unsafe step s is a step in which an accepted proposal might be induced by fake messages, and thus, we cannot control the safety for this step.

Definition 21 (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 note F the suffix of the execution that starts with Σ. Assume that λ executes at least two trials 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 note $e^0(\alpha)$ (resp. $e^f(\alpha)$) the event that corresponds to α sending to λ a message received in the trial that corresponds to Q^0 (resp. Q^f).*

The zone observed by λ during the epoch σ, *noted $Z(F, \lambda, \sigma)$, is the set of real acceptation scenarios relatively to F described as follows. A real acceptation scenario relatively to F belongs to $Z(F, \lambda, \sigma)$ if and only if it ends with an acceptation event that does not happen after the end of σ and its first simple acceptation scenario $U = (S, b) \rightsquigarrow (\beta, b, p)$ is 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 3).*

Remark 14. The observed zone models a globally defined time period during which we will prove, under specific assumptions, the safety property (cf. Theorem 4).

6.5 Practical Safety - Results

Lemma 6 (Fake Acceptation Scenarios) *Consider a fake message m, and two acceptation scenarios of charactestics (μ, l, s, p) and (μ', l', s', p') that begins with the reception of m. Then both scenarios share the same characteristics, i.e., $(\mu, l, s, p) = (\mu', l', s', p')$.*

PROOF. We have two scenarios that begins with the reception of m. Focus on the first simple scenario of each acceptation scenario. Assume, for instance, that the message m is a $p1b$ message and both simple fake acceptation scenarios are as follows

$$\bigcirc \xrightarrow{p1b} \rho \xrightarrow{p2a} (\alpha, b, p) \tag{16}$$

$$\bigcirc \xrightarrow{p1b} \rho' \xrightarrow{p2a} (\alpha', b', p') \tag{17}$$

Since once a message is received, it is not in the communication channels anymore, the event "reception of m at ρ" and "reception of m at ρ'" must be the same. In particular $\rho = \rho'$. Thanks to the messages it has received, the processor ρ computes a proposal (b, p) and broadcasts it. Hence, the processors α and α' receives (and accepts) the same proposal (b, p). Hence, $(b, p) = (b', p')$. By definition, $\chi(b) = \mu$, $b[\mu].(l\ s) = (l\ s)$ and $\chi(b') = \mu'$, $b[\mu'].(l\ s) = (l'\ s')$. Therefore, $(\mu, l, s, p) = (\mu', l', s', p')$. The other cases are analogous. □

Theorem 2 (Fake Characteristics) *Given a subexecution F, we have*

$$|\mathcal{FC}(F)| \leq \mathbf{C}^{\frac{\mathbf{n}(\mathbf{n}-1)}{2}} \tag{18}$$

$$\forall \mu, l,\ |\mathcal{US}(F, \mu, l)| \leq \mathbf{C}^{\frac{\mathbf{n}(\mathbf{n}-1)}{2}} \tag{19}$$

PROOF. On the contrary, note $r = |\mathcal{FC}(F)|$ and assume that $r > \mathbf{C}^{\frac{\mathbf{n}(\mathbf{n}-1)}{2}}$. We note $c_1, \ldots, c_r$ the distinct values in $\mathcal{FC}(F)$, and $c_i = (\mu_i, l_i, s_i, p_i)$. By definition, for each c_i, there is an event $e_i \in \mathcal{FC}(F, \mu_i, l_i, s_i)$ and a scenario U_i in $Sc(e_i)$ that begins with a simple fake acceptation scenario. Let m_i be the fake message consumed in this simple fake acceptation scenario. Since there are at most $\mathbf{C}^{\frac{\mathbf{n}(\mathbf{n}-1)}{2}}$ fake messages in the starting configuration of F, and since the set of fake messages is non-increasing (channels do not produce messages), there are two indices $1 \leq i < j \leq r$ such that $m_i = m_j$. By Lemma 6, the scenarios U_i and U_j have the same characteristic, i.e., $c_i = c_j$; contradiction. The second inequation is straightforward since $|\mathcal{US}(F, \mu, l)| \leq |\mathcal{FC}(F)|$ □

Lemma 7 (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 note 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 a_ρ being equal to μ_σ and containing the label l_σ, and the label field in the entry $a_\rho[\mu_\sigma]$ undergoes a cycle of labels during $G(\rho)$. Assume that, if $\mu_\sigma < \lambda$, then 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 4, since the entry $a_\rho[\lambda]$ 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 3 (Weak Practical Safety) *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. We note F the suffix of the execution that starts with Σ. Consider a step value s and the two following simple scenarios*

$$U_1 = \rho_1 \xrightarrow{p1a} (S_1, b_1) \xrightarrow{p1b} \rho_1 \xrightarrow{p2a} (Q_1, b_1, p_1) \xrightarrow{p2b} \rho_1 \xrightarrow{dec} (\alpha_1, b_1, p_1) \tag{20}$$

$$U_2 = (S_2, b_2) \rightsquigarrow (\alpha_2, b_2, p_2) \tag{21}$$

with characteristics $(\mu_\sigma, l_\sigma, s, p_1)$ and $(\mu_\sigma, l_\sigma, s, p_2)$ respectively. In addition, we assume that $b_i[\mu_\sigma].t > h$ and $\tau_1 \leq \tau_2$ where $\tau_i = b_i[\lambda].(t\ id)$. We note e_i for the acceptation event (α_i, b_i, p_i). Assume that both events e_1 and e_2 occur in F and $s \notin \mathcal{US}(F, \mu_\sigma, l_\sigma)$. In addition, assume that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. Then either $p_1 = p_2$ or the last event of σ happens before one of the event e_1 or e_2.

PROOF. We assume that both events e_1 and e_2 do not happen after the last event of σ and we prove that $p_1 = p_2$. Since s is not in $\mathcal{US}(F, \mu_\sigma, l_\sigma)$, every scenario in $Sc(e_1)$ or $Sc(e_2)$ are real acceptation scenarios relatively to F. We note γ^* the configuration right before the subexecution Σ. We prove the result by induction on the value of τ_2.

(Bootstrapping). We first assume that $\tau_2 = \tau_1$. In particular, $\rho_1 = \tau_1.id = \tau_2.id = \rho_2$. If $p_1 \neq p_2$, this means that ρ_1 has sent two $p2a$ messages with different proposals and the same tag[1]. Note e and f the events that correspond 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 tag x with $x[\mu_\sigma].l = l_\sigma$ and $x[\mu_\sigma].t > h$; this is a contradiction since σ is h-safe. Hence, e and f occur in F. Then, there must be a cycle of labels in the entry $a_{\rho_1}[\mu_\sigma]$ between the e and f. By Lemma 7, this implies that the last event of σ happens before the event e_1 or e_2; this is a contradiction. Hence, $p_1 = p_2$.

(Induction). Now, τ_2 is any value such that $\tau_1 < \tau_2$ and we assume the result holds for every value τ such that $\tau_1 \leq \tau < \tau_2$. 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 (b_1, p_1) in the scenario U_1 (reception of a $p2a$ message), and the adoption of the tag b_2 in the scenario U_2 (reception of a $p1a$ message). First, the events f_1 and f_2 do not occur in the execution prefix A. Otherwise there would exist a tag value x in γ^* such that $x[\mu_\sigma].l = l_\sigma$ and $x[\mu_\sigma].t > h$; this is a contradiction, since σ is h-safe. Hence, f_1 and f_2 occur in the suffix F.

We claim that f_1 happens before f_2. Otherwise, since $\tau_2 > \tau_1$, there must be a cycle of labels in the field $a_\beta[\mu_\sigma].l$. By

[1]Modulo $\simeq$.

Lemma 7, this implies that the last event of σ happens before the event f_1, and thus before the event e_1; contradiction. Hence, f_1 happens before f_2. We claim that the $p1b$ message the acceptor β has sent contains a non-null lastly accepted proposal $r_\beta[\mu_\sigma] = (b, p)$ such that $\chi(b) = \mu_\sigma$, $b[\mu_\sigma].(l\ s) = (l_\sigma\ s)$ and $\tau_1 \leq b[\mu_\sigma].(t\ id) < \tau_2$. Otherwise, there must be a cycle of labels between f_1 and f_2, which implies that f_2, and thus e_2, happens after the end of σ.

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. It first checks that every tag received uses the entry μ_σ and the label l_σ and that there is no two different proposals with two tags that share the same content in entry μ_σ before continuing to the second phase of Paxos, and if it is not the case, it updates its proposer tag and executes another phase one of Paxos. Hence, since ρ_2 has moved to the second phase of Paxos, it means that no such issue has happened. Then, it selects among the proposals whose tags point to the step s the proposal (b_c, p_c) with the highest tag. In particular, $\chi(b_c) = \mu_\sigma$, $b_c[\mu_\sigma].(l\ s) = (l_\sigma\ s)$. Since ρ_2 has received the proposal (b, p) from β, we have $\tau_1 \leq \tau_c < \tau_2$, where $\tau_c = \beta_c[\mu_\sigma].(t\ id)$. Let β_c be the proposer in S_2 which has sent to ρ_2 the proposal (b_c, p_c) in the $p1b$ message. There is an event f_c in F that corresponds to β_c accepting the proposal (b_c, p_c). Otherwise there would exist a tag value x in γ^* such that $x[\mu_\sigma].l = l_\sigma$ and $x[\mu_\sigma].t > h$; this is a contradiction, since σ is h-safe. Next, since $s \notin \mathcal{US}(F, \mu_\sigma, l_\sigma)$, $\chi(b_c) = \mu_\sigma$, and $b_c[\mu_\sigma].(l\ s) = (l_\sigma\ s)$, the set $Sc(e_2)$ does not contain any fake acceptation scenario relatively to F, thus neither the set $Sc(f_c)$. We can pick a real scenario in $Sc(f_c)$ and apply the induction hypothesis, which shows that $p_c = p_1$. Hence, $p_1 = p_2$, since p_c is the consensus value the proposer ρ_2 sends during the corresponding Paxos phase two. □

Corollary 1 (Weak Practical Safety) *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 note F the suffix of the execution that starts with Σ. Consider a step value s and two decision events $e_i = (\alpha_i, b_i, p_i)$, $i = 1, 2$, such that $\chi(b_i) = \mu_\sigma$, $b_i[\mu_\sigma].(l\ s) = (l_\sigma\ s)$ and $b_i[\mu_\sigma].t > h$. Assume that both events e_1 and e_2 occur in F and $s \notin \mathcal{US}(F, \mu_\sigma, l_\sigma)$. In addition, assume that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. Then either $p_1 = p_2$ or the last event of σ happens before one of the event e_1 or e_2.*

Proof. Since e_1 and e_2 are decision events, and since s is not in $\mathcal{US}(F, \mu_\sigma, l_\sigma)$, there are two real acceptation scenarios in $Sc(e_1)$ and $Sc(e_2)$ relatively to F respectively that contains simple acceptation scenarios of the second kind as follows :

$$U_1 = \rho_1 \xrightarrow{p1a} (S_1, c_1) \xrightarrow{p1b} \rho_1 \xrightarrow{p2a} (Q_1, c_1, p_1) \xrightarrow{p2b} \rho_1 \xrightarrow{dec} (\beta_1, c_1, p_1) \quad (22)$$

$$U_2 = \rho_2 \xrightarrow{p1a} (S_2, c_2) \xrightarrow{p1b} \rho_2 \xrightarrow{p2a} (Q_2, c_2, p_2) \xrightarrow{p2b} \rho_2 \xrightarrow{dec} (\beta_2, c_2, p_2) \quad (23)$$

with characteristics $(\mu_\sigma, l_\sigma, s, p_1)$ and $(\mu_\sigma, l_\sigma, s, p_2)$ respectively and trial values $c_i[\mu_\sigma].t$ greater than h. We note $\tau_i = c_i[\mu_\sigma].(t\ id)$. Whether $\tau_1 \leq \tau_2$ or $\tau_2 \leq \tau_1$, Theorem 3 gives the result. □

Theorem 4 (Practical 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 note 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, U_2 in $Z(F, \lambda, \sigma)$ with characteristics (μ_1, l_1, s_1, p_1) and (μ_2, l_2, s_2, p_2) such that $\mu_\sigma \leq \min(\mu_1, \mu_2)$ and both scenarios contain simple acceptation scenarios with tags whose associated trial values are greater than h. Then $(\mu_1, l_1) = (\mu_2, l_2) = (\mu_\sigma, l_\sigma)$, and if $s_1 = s_2$ then $p_1 = p_2$.*

Proof. Assume that the scenario U_1 is such that $\mu_1 > \mu_\sigma$. Let $V = (S, b) \rightsquigarrow (\beta, b, p)$ be its first simple acceptation scenario. By definition of the observed zone $Z(F, \lambda, \sigma)$, there exists an acceptor α in $S \cap Q^0 \cap Q^f$ such that we have the happen-before relations $e^0(\alpha) \rightsquigarrow e \rightsquigarrow e^f(\alpha)$, where e is the event that corresponds to α sending a $p1b$ message in the scenario V. At $e^0(\alpha)$ and $e^f(\alpha)$, messages are sent to λ and are processed during σ. Hence, the corresponding tag values of the variable a_α must use the entry μ_σ and the label l_σ. Otherwise, the message either is not processed or causes an interrupt at processor λ. Now, at event e, the first valid entry of the variable a_α is $\mu_1 > \mu_\sigma$ which implies that the entry μ_σ is invalid. Hence, between $e^0(\alpha)$ and $e^f(\alpha)$, the entry $a_\alpha[\mu_\sigma]$ becomes invalid and valid again. There must be a cycle of labels in the label field $a_\alpha[\lambda].l$. Lemma 7 implies that the last event of σ happens before $e^f(\alpha)$; by the definition of $e^f(\alpha)$, this is a contradiction. Therefore $\mu_1 = \mu_\sigma$. If $l_1 \neq l_\sigma$, then there must also be a cycle of labels in the entry $a_\alpha[\mu_\sigma]$ between $e^0(\alpha)$ and $e^f(\alpha)$, which leads to a contradiction again, thanks to the same argument. Therefore, $l_1 = l_\sigma$. Of course, the previous demonstration also shows that $(\mu_2, l_2) = (\mu_\sigma, l_\sigma)$. If $s_1 = s_2$, then Corollary 1, the fact that the trial values associated to the scenarios U_1 and U_2 are greater than h and the fact that the two acceptation events in scenarios U_1 and U_2 do not happen after the end of σ imply that $p_1 = p_2$. □

Remark 15. 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.

7. FAILURE DETECTOR

7.1 Overview

Liveness for some step s in Paxos is not guaranteed unless there is a unique proposer for this step s. The original Paxos algorithm assumes that the choice of a distinguished proposer for a given step is done through an external module. It is not necessary for this external module to be a proper leader election module. It is only needed that some processor be the unique proposer for a "long enough" period of time. Of course, in a purely asynchronous system, there is no canonical definition of what is a "long enough" period of time.

Usually, failure detectors are used to deal with this issue. The exact implementation of a failure detector is hidden

from the user point of view, and usually assumes some conditions such as synchrony or partial synchrony, that depend mostly on the concrete representation of the system. Hence, the solution of the initial problem is solved without refering to a more concrete representation of the system. For the system builders, the real question then concerns the possibility to implement a failure detector that satisfies the initial problem requirements. In the context of self-stabilization, it is not obvious that a failure detector is implementable in any settings. Hence, in the sequel, we present an implementation of a self-stabilizing failure detector that works under a partial synchronism assumption.

7.2 Self-Stabilizing Failure Detector

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 [4]). 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 $L_\alpha[\beta] = W$.

Assumption 3 (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. The connection with the external module Θ in Section 4 can be defined as follows

$$\Theta_\alpha = \textbf{true} \Leftrightarrow \alpha = \min(\mu;\ L_\alpha[\mu] < W) \qquad (24)$$

Under Assumption 3, we see that the module Θ eventually satisfies the conditions in Assumption 2, Section 4.

8. CONCLUSION

The original Paxos algorithm provides a solution to the problem, for a distributed system, to reach successively several consensus on different options, e.g., the different requests to apply in the case of a distributed database. A proper tagging system using natural integers is defined so that, although the liveness property, i.e., the fact that, in every consensus instance, every processor eventually decides, is not guaranteed, the safety property is ensured : no two processors decide on different values in the same consensus instance. The original formulation, however, does assume a consistent initial state and assumes that consistency is preserved forever by applying step transitions from a restricted predefined set of step transitions. This line of consistency preserving argument is fragile and error prone in any practical system that should exhibit availability and functionality during very long executions. Hence, there is an urgent need for self-stabilizing on-going systems, and in particular for the very heart of asynchronous replicated state machine systems used by the leading companies to ensure robust services. One particular aspect of self-stabilizing systems is the need to re-examine the assumption concerning the use of (practically) unbounded time-stamps. While in practice it is reasonable for Paxos to assume that a bounded value, represented by 64 bits, is a natural (unbounded) number, for all practical considerations, in the scope of self-stabilization the 64 bits value may be corrupted by a transient fault to its maximal value at once, and still recovery following such a transient fault must be guaranteed. More generally, the designer of self-stabilizing systems, does not try to protect its system against specific "bad" scenarios. She assumes that some transient faults, whatever their origin is, corrupt (a part of) the system and ensures that the system recovers automatically after such fault occurrences.

Using the finite labeling scheme presented, we have defined a new kind of tag system that copes with such transient faults. The tag is defined as a vector indexed by the processor identifiers and such that each entry contains a label, a step and a trial value. Incrementing the label becomes a way to properly reset the step and trial values in a given entry of a tag. Each processor is responsible for producing labels only in the entry that corresponds to its identifier. Therefore, once it collects enough information about the labels present in its attributed entry, a processor is able to produce a label that no other processor can cancel. Hence, in a tag, there might be several entries with "winning" labels, and the owner of the tag uses the entry with the lowest identifier. The first part of the proof (Sections 6.2 and 6.3) aims at proving that if a processor is active for a long enough period of time, then this processor reaches a practically infinite local execution, namely an epoch, during which its tag behaves exactly as in the original Paxos algorithm (Theorem 1). This result mainly holds thanks to the fact that each processor maintains histories of labels that allow the detection of cycle of labels.

The second part of the proof (Sections 6.4 and 6.5) focuses on the global point of view. We show that, given a proposer λ which has reached a practically infinite local execution, a specific kind of safety is ensured. Indeed, if there are two decisions on proposals with tags that have the same first valid entry and the same label as the proposer λ, and if the two tags point to the same step then the two consensus values are equal, or one of the decision happens after a practically infinite period of time (Theorem 3). However, due to either fake messages (messages that were not sent by any processor) or messages produced within the stabilization period, the safety and integrity property might be violated during the practically safe period. Theorem 2 shows that the number of decisions due to fake messages (or messages produced within the stabilization period) is bounded. Finally, Theorem 4 shows that the safety property is satisfied within a globally defined period, namely the zone observed by λ during its epoch, given that λ is the live processor with the lowest identifier (for the sake of simplifying the proof). We presented (Section 7) an implementation of a tunable self-stabilizing failure detector that works under a partial synchrony assumption. Such a device is mandatory to ensure the liveness property of the consensus problem. The implementation we presented also highlights the fact that the distinguished proposer must have the lowest possible identifier, in order for our self-stabilizing Paxos algorithm to behave efficiently. Note, however, that once every live

processors use the same first valid entry and the same corresponding label, then the proposer can be any live processor, as this execution is exactly analogous to an initialized Paxos execution.

9. REFERENCES

[1] 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.

[2] S. Delaët, S. Dolev, and O. Peres. Safe and eventually safe : Comparing self-stabilizing and non-stabilizing algorithms on a common ground. In T. Abdelzaher, M. Raynal, and N. Santoro, editors, *Principles of Distributed Systems*, volume 5923 of *Lecture Notes in Computer Science*, pages 315–329. Springer Berlin / Heidelberg, 2009.

[3] S. Dolev. *Self-stabilization.* MIT Press, 2000.

[4] 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.

[5] S. Dolev and Y. A. Haviv. Self-stabilizing microprocessor: Analyzing and overcoming soft errors. *IEEE Transactions on Computers*, 55:385–399, 2006.

[6] S. Dolev, R. I. Kat, and E. M. Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76:884–900, December 2010.

[7] 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.

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

[9] E. C. Rosen. Vulnerabilities of network control protocols: an example. *SIGCOMM Comput. Commun. Rev.*, 11(3):10–16, July 1981.

[10] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2001.

APPENDIX

A. APPENDIX

A.1 Construction of a Finite Labeling Scheme

We show how to construct a finite labeling scheme $(\mathcal{L}, \prec, d, \nu)$. First, consider the set of integers $X = \{1, 2, ..., K\}$ with $K = d^2 + 1$. We define the set $\mathcal{L}$ to be the set of every tuple (z, A) where $z \in X$ is the *sting*, and $A \subset X$ with $|A| \leq d$ is called the *antistings*. The relation $\prec$ is defined as follows

$$l = (z, A) \prec l' = (z', A') \Leftrightarrow (z \in A') \wedge (z' \notin A) \qquad (25)$$

The function ν is defined as follows. Given r labels (s_1, A_1), ..., (s_r, A_r) with $r \leq d$, the label $\nu(l_1, \ldots, l_r) = (s, A)$ is given by

$$s = \min \{X - (A_1 \cup \cdots \cup A_r)\} \qquad (26)$$

$$A = \{s_1, \ldots, s_r\} \qquad (27)$$

The function is well-defined since $r \leq d$ and $|A_1 \cup \cdots \cup A_r| \leq d^2 < |X|$. In addition, for every i, we have $s \notin A_i$ and $s_i \in A$, thus $(s_i, A_i) \prec (s, A)$.

A.2 Pigeonhole Principle

Lemma 8 *Consider a sequence $u = (u^i)_{1 \leq i \leq N}$ such that $\forall 1 \leq i \leq N, u^i \in \{0, 1\}$, and $N = (n+1)m$ for some $n, m \in \mathbb{N} - \{0\}$. Assume that the cardinal of $\{i \mid u^i = 1\}$ is less than or equal to n. Then there exists $1 \leq i_0 \leq N$ such that for every $i_0 \leq i \leq i_0 + m - 1$, $u^i = 0$.*

PROOF. Divide the sequence u in successive subsequences σ^j, $1 \leq j \leq n+1$ such that each σ^j length is m. If for every $1 \leq j \leq n+1$, the sequence σ^j contains at least one 1, then the number of 1 appearing in u is at least $n+1$, which leads to a contradiction. Hence, there is some j_0 such that the sequence σ^j only contains 0. □

A.3 Algorithms

Algorithm 1: Tags - Procedures

```
1  function clean(λ : processor identifier, a : tag)
2      foreach μ ∈ Π do
3          if a[μ].cl ≼ a[μ].l then a[μ] ← ⊥
4          a[μ].id ← λ
5      end
6  end
7  function fill_cl(x, y : tags)
8      x_c ← x, y_c ← y
9      foreach μ ∈ Π do
10         if y_c[μ].(l or cl) ⋠ x[μ].l then
           x[μ].cl ← y_c[μ].(l or cl)
11         if y_c[μ].l = x[μ].l ∧ y_c[μ].(s or t) = 2^b then
12             x[μ].(s t) ← (2^b 2^b)
13         idem by exchanging (x,x_c) and (y, y_c)
14     end
15 end
```

Algorithm 2: Tags - Increment functions

```
1  function check_entry(λ : identifier, x : tag, L : history
   of labels)
2      if x[λ] is invalid then
3          L ← L + x[λ].l
4          x[λ].(l s t id) ← (ν(L) 0 0 λ)
5          x[λ].cl ← ⊥
6  end
7  function ν*(λ : identifier, x : tag, L : label history)
8      y ← x
9      clean(λ, y)
10     if χ(y) ≤ λ then
11         (case ν^s) y[χ(y)].(s t) ← (1 + y[χ(y)].s 0)
12         (case ν^t) y[χ(y)].t ← 1 + y[χ(y)].t
13     check_entry(λ, y, L)
14     return y
15 end
```

Algorithm 3: Acceptor α

switch `receive()` **do**
  case $\langle p1a, \lambda, b\rangle$
    $a_{old} \leftarrow a_\alpha$
    if $b[\alpha].(l \text{ or } cl) \not\preccurlyeq a_\alpha[\alpha].l$ **then** $H_\alpha^{cl} \leftarrow H_\alpha^{cl} + b[\alpha].(l \text{ or } cl)$
    `fill_cl`(a_α, b), `check_entry`$(\alpha, a_\alpha, H_\alpha^{cl})$
    if $a_\alpha \prec b$ **then**
      $a_\alpha[\chi(b)] \leftarrow b[\chi(b)]$
      if $a_{old}[\chi(b)].l \neq a_\alpha[\chi(b)].l$ **then**
        $r_\alpha[\chi(b)] \leftarrow \bot$
        $H_\alpha[\chi(b)] \leftarrow H_\alpha[\chi(b)] + a_{old}[\chi(b)].l$
        if $\exists l \in H_\alpha[\chi(b)],\ l \not\preccurlyeq a_\alpha[\chi(b)].l$ **then** $a_\alpha[\chi(b)].cl \leftarrow l$
    end
    foreach $\mu \in \Pi$ **do**
      $c \leftarrow r_\alpha[\mu].b$
      if $c[\mu].l \neq a_\alpha[\mu].l \vee a_\alpha[\mu].(l\ s\ t\ id) \prec c[\mu].(l\ s\ t\ id)$ **then**
        $r_\alpha[\mu] \leftarrow \bot$
    end
    `send`$(\lambda, \langle p1b, \alpha, a_\alpha, r_\alpha[\chi(a_\alpha)]\rangle)$
  case $\langle p2a, \lambda, b, p\rangle$ *or* $\langle decision, \lambda, b, p\rangle$
    $a_{old} \leftarrow a_\alpha$
    if $b[\alpha].(l \text{ or } cl) \not\preccurlyeq a_\alpha[\alpha].l$ **then** $H_\alpha^{cl} \leftarrow H_\alpha^{cl} + b[\alpha].(l \text{ or } cl)$
    `fill_cl`(a_α, b), `check_entry`$(\alpha, a_\alpha, H_\alpha^{cl})$
    if $a_\alpha \preccurlyeq b$ **then**
      $a_\alpha[\chi(b)] \leftarrow b[\chi(b)]$, $r_\alpha[\chi(b)] \leftarrow [b, p]$
      if *it is a decision message* **then** `decide`(b, p)
      if $a_{old}[\chi(b)].l \neq a_\alpha[\chi(b)].l$ **then**
        $H_\alpha[\chi(b)] \leftarrow H_\alpha[\chi(b)] + a_{old}[\chi(b)].l$
        if $\exists l \in H_\alpha[\chi(b)],\ l \not\preccurlyeq a_\alpha[\chi(b)].l$ **then** $a_\alpha[\chi(b)].cl \leftarrow l$
    end
    foreach $\mu \in \Pi$ **do**
      $c \leftarrow r_\alpha[\mu].b$
      if $c[\mu].l \neq a_\alpha[\mu].l \vee a_\alpha[\mu].(l\ s\ t\ id) \prec c[\mu].(l\ s\ t\ id)$ **then**
        $r_\alpha[\mu] \leftarrow \bot$
    end
    if *it is a p2a message* **then** `send`$(\lambda, \langle p2b, \alpha, a_\alpha, r_\alpha\rangle)$
endsw

Algorithm 4: Proposer λ - Main loop

loop As long as Θ_λ = **true**
  $p \leftarrow$ `input`()
  $a_\lambda \leftarrow \nu^s(\lambda, a_\lambda, H_\lambda^{cl})$

  [Ph. 1]
  $p_\lambda \leftarrow p$
  $\forall \alpha \in \Pi$, `send`$(\alpha, \langle p1a, \lambda, a_\lambda\rangle)$
  if `PR(1)` *returns nok* **then** go to [Ph. 1]

  [Ph. 2]
  let $\mu = \chi(a_\lambda)$, and Γ be the set of non-null proposals $r_\alpha[\mu]$ received at the end of [Ph. 1] in
  if $\Gamma \neq \emptyset$ **then**
    if $\forall x, y \in \Gamma, \chi(x.a) = \chi(y.a) = \mu \wedge x.a[\mu].l = y.a[\mu].l = a_\lambda[\mu].l$ **then**
      $\Gamma_0 \leftarrow \{(a, p) \in \Gamma \mid a = \max(b | \exists q, (b, q) \in \Gamma, b[\mu].s = a_\lambda[\mu].s)\}$
      if Γ_0 *contains more than one element* **then**
        $a_\lambda \leftarrow \nu^s(\lambda, a_\lambda, H_\lambda^{cl})$
        go to [Ph. 1]
      else
        we note $\Gamma_0 = \{(a, p)\}$
        $p_\lambda \leftarrow p$
    else
      $a_\lambda \leftarrow \nu^s(\lambda, a_\lambda, H_\lambda^{cl})$
      go to [Ph. 1]
    end
  end
  $\forall \alpha \in \Pi$, `send`$(\alpha, \langle p2a, \lambda, a_\lambda, p_\lambda\rangle)$
  if `PR(2)` *returns nok* **then** go to [Ph. 1]
  $\forall \alpha \in \Pi$, `send`$(\alpha, \langle decision, \lambda, a_\lambda, p_\lambda\rangle)$
end loop

Algorithm 5: Proposer λ - Preempting Routine

```
1  function PR(φ : phase 1 or phase 2)
2      N ← ∅, M ← 0, a_sent ← a_λ
3      while |N| < n − f do
4          b ← a_sent
5          ⟨pφb, α, a_α, q_α⟩ ← receive(⟨pφb, ∗, ∗, ∗⟩)
6          fill_cl (a_α, b)
7          C+ = (a_α ≃ b) ∧ (φ = 2 ⇒ p_λ = q_α.p))
8          C− = (a_α ⋠ b)
9          if α ∉ N then
10             if C+ ∨ C− then N ← N ∪ {α}
11             if C+ then M ← M + 1
12             else
13                 if a_α[λ].(l or cl) ⋠ a_λ[λ].l then
                   H_λ^cl ← H_λ^cl + a_α[λ].(l or cl)
14                 fill_cl(a_α, a_λ)
15                 check_entry(λ, a_λ, H_λ^cl)
16                 let μ = χ(a_α) in
17                 if a_α ⋠ a_λ then
18                     if μ < χ(a_λ) then
19                         H_λ[μ] ← H_λ[μ] + a_λ[μ].l
20                         a_λ[μ] ← a_α[μ]
21                         if ∃l ∈ H_λ[μ], l ⋠_l a_λ[μ].l then
                           a_λ[μ].cl ← l
22                         a_λ ← ν^t(λ, a_λ, H_λ^cl)
23                     else
24                         (we have χ(a_λ) = μ and
                           a_λ[μ].l = a_α[μ].l)
25                         if a_α[μ].s = a_λ[μ].s then
26                             a_λ[μ].t ← a_α[μ].t
27                             a_λ ← ν^t(λ, a_λ, H_λ^cl)
28                         else
29                             a_λ[μ].s ← a_α[μ].s
30                             a_λ ← ν^s(λ, a_λ, H_λ^cl)
31                         end
32                     end
33                 end
34         end
35     end
36     if M = n − f then return ok
37     else return nok
38 end
```