Revisiting the Paxos algorithm [⋆]

Roberto De Prisco [a] Butler Lampson [a] Nancy Lynch [a]

[a] *MIT Laboratory for Computer Science, 545 Technology Square NE43, Cambridge, MA 02139, USA. Email:* {robdep,lampson,lynch}@theory.lcs.mit.edu.

Abstract

The PAXOS algorithm is an efficient and highly fault-tolerant algorithm, devised by Lamport, for reaching consensus in a distributed system. Although it appears to be practical, seems to be not widely known or understood. This paper contains a new presentation of the PAXOS algorithm, based on a formal decomposition into several interacting components. It also contains a correctness proof and a time performance and fault-tolerance analysis.

The formal framework used for the presentation of the algorithm is provided by the Clock General Timed Automaton (Clock GTA) model. The Clock GTA provides a systematic way of describing timing-based systems in which there is a notion of "normal" timing behavior, but that do not necessarily always exhibit this "normal" timing behavior.

Key words: I/O automata models, formal verification, distributed consensus, partially synchronous systems, fault-tolerance

1 Introduction

Reaching consensus is a fundamental problem in distributed systems. Given a distributed system in which each process starts with an initial value, to solve a consensus problem means to give a distributed algorithm that enables each process to eventually output a value of the same type as the input values, in such a way that three conditions, called *agreement*, *validity* and *termination*,

[⋆] A preliminary version of this paper appeared in proceedings of the 11^{th} International Workshop on Distributed Algorithms, Saarbrücken, Germany, September 1997, Lecture Notes in Computer Science, vol. 1320 (1997), pp. 111–125. The first author is on leave from the Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (SA), Italy.

hold. There are different definitions of the problem depending on what these conditions require. Distributed consensus has been extensively studied. A good survey of early results is provided in [13]. We refer the reader to [24] for a more recent treatment of consensus problems.

Real distributed systems are often partially synchronous systems subject to process, channel and timing failures and process recoveries. In a partially synchronous distributed system, processes take actions within ℓ time and messages are delivered within d time, for given constants ℓ and d. However these time bounds hold when the system exhibits a "normal" timing behavior. Hence the above mentioned bounds of ℓ and d can be occasionally violated (timing failures). Processes may stop and recover; it is possible to keep the state of a process, or part of it, in a stable storage so that the state, or part of it, survives the failure. Messages can be lost, duplicated or reordered. Any practical consensus algorithm needs to consider the above practical setting. Moreover the basic safety properties must not be affected by the occurrence of failures. Also, the performance of the algorithm must be good when there are no failures, while when failures occur, it is reasonable to not expect efficiency.

Lamport's PAXOS algorithm [19] meets these requirements. The model considered is a partially synchronous distributed system where each process has a direct communication channel with each other process. The failures allowed are timing failures, loss, duplication and reordering of messages, and process stopping failures. Process recoveries are allowed; some stable storage is needed. PAXOS is guaranteed to work safely, that is, to satisfy agreement and validity, regardless of process, channel and timing failures and process recoveries. When the distributed system stabilizes, meaning that there are no failures, nor process recoveries, and a majority of the processes are not stopped, for a sufficiently long time, termination is also achieved and the performance of the algorithm is good. Hence PAXOS has good fault-tolerance properties and when the system is stable it combines those fault-tolerance properties with the performance of an efficient algorithm, so that it can be useful in practice. In the original paper [19], the PAXOS algorithm is described as the result of discoveries of archaeological studies of an ancient Greek civilization. That paper contains also a proof of correctness and a discussion of the performance analysis. The style used for the description of the algorithm often diverts the reader's attention. Because of this, we found the paper hard to understand and we suspect that others did as well. Indeed the PAXOS algorithm, even though it appears to be a practical and elegant algorithm, seems not widely known or understood.

In [19] a variation of PAXOS that considers multiple concurrent runs of PAXOS for reaching consensus on a sequence of values is also presented. We call this variation the MULTIPAXOS algorithm[1].

This paper contains a new, detailed presentation of the PAXOS algorithm, based on a formal decomposition into several interacting components. It also contains a correctness proof and a time performance and fault-tolerance analysis. The MULTIPAXOS algorithm is also described, together with an application to data replication. The formal framework used for the presentation is provided by the Clock General Timed Automaton (Clock GTA), which has been developed in [5]. The Clock GTA is a special type of Lynch and Vaandrager's General Timed Automaton (GTA) model [26–28]. The Clock GTA uses the timing mechanisms of the GTA to provide a systematic way of describing both the normal and the abnormal timing behaviors of a partially synchronous distributed system subject to timing failures. The model is intended to be used for performance and fault-tolerance analysis of practical distributed systems based upon the stabilization of the system.

The correctness proof uses automata composition and invariant assertion methods. Automata composition is useful for representing a system using separate components. We provide a modular presentation of the PAXOS algorithm, obtained by decomposing it into several components. Each one of these components copes with a specific aspect of the problem. In particular there is a "failure detector" module that detects process failures and recoveries. There is a "leader elector" module that copes with the problem of electing a leader; processes elected leaders by this module, are used as leaders in PAXOS. The PAXOS algorithm is then split into a basic part that ensures agreement and validity and into an additional part that ensures termination when the system stabilizes; the basic part of the algorithm, for the sake of clarity of presentation, is further subdivided into three components. The correctness of each piece is proved by means of invariants, i.e., properties of system states which are always true in any execution.

The time performance and fault-tolerance analysis is conditional on the stabilization of the system behavior starting from some point in an execution. Using the Clock GTA we prove that when the system stabilizes PAXOS reaches consensus in $O(1)$ time and uses $O(n)$ messages, where n is the number of processes. We also briefly discuss the MULTIPAXOS protocol and a data replication algorithm which uses MULTIPAXOS. With MULTIPAXOS the high availability of the replicated data is combined with high fault tolerance.

[1] PAXOS is the name of the ancient civilization studied in [19]. The actual algorithm is called the "single-decree synod" protocol and its variation for multiple consensus is called the "multi-decree parliament" protocol. We use the name PAXOS for the single-decree protocol and the name MULTIPAXOS for the multi-decree parliament protocol.

Related work. The consensus algorithms of Dwork et al. [9] and of Chandra and Toueg [2] bear some similarities with PAXOS. The algorithm of [9] also uses "rounds" conducted by a leader, but the strategy used in each round is different from the one used by PAXOS. Also [9] does not consider process restarts. The time analysis provided in [9] is conditional on a "global stabilization time" after which process response times and message delivery times satisfy the time assumptions. This is similar to our stabilized analysis. A similar time analysis, applied to the problem of reliable group communication, can be found in [12].

The algorithm of Chandra and Toueg is based on the idea of an abstract failure detector [2]. It turns out that failure detectors provide an abstract and modular way of incorporating partial synchrony assumptions in the model of computation. A $\diamond\mathcal{P}$ failure detector incorporates the partial synchrony considered in this paper. One of the algorithms in [2] uses a $\diamond\mathcal{S}$ failure detector, which is weaker than a $\diamond\mathcal{P}$ failure detector. This algorithm is based on the rotating coordinator paradigm and as PAXOS uses majorities to achieve consistency. However it takes, in the worst case, longer time than PAXOS to achieve termination. Chandra, Hadzilacos and Toueg [1] identified the "weakest" failure detector that can be used to solve the consensus problem. This weakest failure detector is $\diamond\mathcal{W}$ and it is equivalent to $\diamond\mathcal{S}$. The Chandra and Toueg algorithm does not consider channel failures (however it can be modified to work with loss of messages but the resulting algorithm is less efficient than PAXOS with respect to the number of messages sent).

The failure detector provided in this paper differs from those classified by Chandra and Toueg in that it provides reliability conditional on the system stabilization. If the system eventually stabilizes then our failure detector can be classified in the class of the eventually perfect failure detectors. However it should be noted that in order for PAXOS to achieve termination it is not needed that the system become stable forever but only for a sufficiently long time.

Dolev et al. [8] have adapted the Chandra and Toueg's definition of failure detector to consider also omission failures and have given a distributed consensus protocol that allows majorities to achieve consensus.

MULTIPAXOS can be easily used to implement a data replication algorithm. The data replication algorithms in [23,30,18,22] are based on ideas similar to the ones used in PAXOS.

PAXOS bears some similarities with the three-phase commit protocol [35]. However the three-phase commit protocol does not always guarantee majorities to progress. The commit algorithm of Keidar and Dolev [17] is similar to PAXOS in that it always guarantees majorities to progress. Also, PAXOS is more efficient than the three-phase commit protocol when the system is stable and consensus has to be reached on a sequence of values (a three-phase protocol is needed only for the first consensus problem, while all the subsequent ones can be solved with a two-phase exchange of messages).

Cristian's timed asynchronous model [4] is similar to the distributed setting considered in this paper. It assumes, however, a bounded clock drift even when the system is unstable. Our model is weaker in the sense that makes no assumption on clock drift when the system is unstable. The Clock GTA provides a formal way of modelling the stability property of the timed asynchronous model. In [31] Patt-Shamir introduces a special type of GTA used for the clock synchronization problem. The Clock GTA considers only the local time; our goal is to model good timing behavior starting from some point on and thus we are not concerned with synchronization of the local clocks.

In [21] Lampson provides a brief overview of the PAXOS algorithm together with key ideas for proving the correctness of the algorithm. We used these ideas in the correctness proof provided in this paper.

Road map. Section 2 describes the I/O automaton models used and Section 3 describes the distributed system considered. Section 4 gives a formal definition of the consensus problem. In Section 5 a failure detector and a leader elector are presented; they are used by the PAXOS algorithm. The PAXOS algorithm itself is described and analyzed in Section 6. Section 7 describes MULTIPAXOS and Section 8 discusses how to use MULTIPAXOS to implement a data replication algorithm.

2 Models

Our formal framework is provided by I/O automaton models, specifically by the Clock GTA model developed in [5]. In this section we briefly describe essential notions about I/O automata needed to read the rest of the paper. We refer the interested reader to [24], Chapters 8 and 23, for more information and references about I/O automaton models, and to [5] for a more detailed presentation of the Clock GTA model.

2.1 I/O automata and the GTA

The I/O automata models are formal models suitable for describing asynchronous and partially synchronous distributed systems. An I/O automaton is a simple type of state machine in which transitions are associated with named *actions.* These actions are classified into categories, namely *input, output, internal* and, for the timed models, *time-passage.* Input and output actions are used for communication with the external environment, while internal actions are local to the automaton. The time-passage actions are intended to model the passage of time. The input actions are assumed not to be under the control of the automaton, that is, they are controlled by the external environment, which can force the automaton to execute the input actions. Internal and output actions are controlled by the automaton. The time-passage actions are also controlled by the automaton (though this may at first seem somewhat strange, it is just a formal way of modeling the fact that the automaton must perform some action before some amount of time elapses).

The General Timed Automaton (GTA) uses *time-passage* actions called $\nu(t)$, $t \in \mathbb{R}^+$ to model the passage of time. The time-passage action $\nu(t)$ represents the passage of time by the amount t.

A GTA consists of four components: (i) the *signature*, consisting of four disjoint sets of actions, namely, the input, output, internal and time-passage actions; (ii) the set of *states*; (iii) the set of *initial states*, which is a nonempty subset of the set of states; (iv) the *state-transition relation*, which specifies all the possible state to state transitions.

A state to state transition, usually called a *step*, is a triple (s, π, s') where s and s' are states of the automaton and π is an action that takes the automaton from s to s'. If for a particular state s and action π, there is some transition of the form (s, π, s'), then we say that π is *enabled* in s. Input actions are enabled in every state.

A *timed execution fragment* of a GTA is defined to be either a finite sequence $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ or an infinite sequence $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots,$ where the s's are states, the π's are actions (either input, output, internal, or time-passage), and $(s_k, \pi_{k+1}, s_{k+1})$ is a step for every k. Note that if the sequence is finite, it must end with a state. The *length* of a finite execution fragment $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ is r. A timed execution fragment beginning with a start state is called a *timed execution.* If α is any timed execution fragment and π_r is any action in α, then we say that the *time of occurrence* of π_r is the sum of all the reals in the time-passage actions preceding π_r in α. A timed execution fragment α is said to be *admissible* if the sum of all the reals in the time-passage actions in α is ∞. A state is said to be *reachable* if it is the final state of a finite timed execution of the GTA.

In the rest of the paper we will often refer to timed executions (resp. timed execution fragments) simply as executions (resp. execution fragments).

2.2 The Clock GTA

A Clock GTA is a GTA with a special component included in the state; this special variable is called *Clock* and it can assume values in $\mathbb{R}$. The purpose of *Clock* is to model the local clock of the process. The only actions that are allowed to modify *Clock* are the time-passage actions $\nu(t)$. When a time-passage action $\nu(t)$ is executed by the automaton, the *Clock* is incremented by an amount of time $t' \geq 0$ independent of the amount t of time specified by the time-passage action[2]. Since the occurrence of the time-passage action $\nu(t)$ represents the passage of (real) time by the amount t, by incrementing the local variable *Clock* by an amount t' different from t we are able to model the passage of (local) time by the amount t'. As a special case, we have some time-passage actions in which $t' = t$; in these cases the local clock of the process is running at the speed of real time.

In the following and in the rest of the paper, we use the notation $s.x$ to denote the value of state component x in state s.

Definition 2.1 *A step* $(s_{k-1}, \nu(t), s_k)$ *of a Clock GTA is called* regular *if* $s_k.Clock - s_{k-1}.Clock = t$*; it is called* irregular *if it is not regular.*

That is, a time-passage step executing action $\nu(t)$ is regular if it increases *Clock* by $t' = t$. In a regular time-passage step, the local clock is increased by the same amount as the real time, whereas in an irregular time-passage step $\nu(t)$ that represents the passage of real time by the amount t, the local clock is increased either by $t' < t$ (the local clock is slower than the real time) or by $t' > t$ (the local clock is faster than the real time).

Definition 2.2 *A timed execution fragment* α *of a Clock GTA is called* regular *if all the time-passage steps of* α *are regular. It is called* irregular *if it is not regular, i.e., if at least one of its time-passage step is irregular.*

In a partially synchronous distributed system processes are expected to respond and messages are expected to be delivered within given time bounds. A timing failure is a violation of these time bounds. An irregular time-passage step can model the occurrence of a timing failure. We remark that a timing failure can actually be either an upper bound violation (a process or a channel is slower than expected) or a lower bound violation (a process or a channel is faster than expected). Obviously, in a regular execution fragment there are no timing failures.

Though we have defined a regular execution fragment so that it does not

[2] Formally, we have that if $(s, \nu(t), s')$ is a step then also $(s, \nu(\tilde{t}), s')$, for any $\tilde{t} > 0$, is a step. Hence a Clock GTA cannot keep track of the real time.

contain any of the timing failures, we remark that for the the scope of this paper we actually need only that the former type of timing failures (upper bound) does not happen. That is, for the scope of this paper, we could have defined a regular step $\nu(t)$ as one that increases the clock time by an amount t', $t' \geq t$.

Using MMTAs to describe Clock GTAs. GTAs encode timing restrictions explicitly into the code of the automata. This provides a lot of flexibility but requires more complicated code to explicitly handle the time and the time bounds. In many situations however we do not need such a flexibility and we only need to specify simple time bounds (e.g., an enabled action is executed within ℓ time). The MMTA[3] model is a subclass of the GTA model suitable for describing such simple time bounds. The MMTA does not have time-passage actions but each action is coupled with its time of execution so that the execution of an MMTA is a (not necessarily finite) sequence $\alpha = s_0$, (π_1, t_1), s_1, $(\pi_2, t_2), \ldots, (\pi_r, t_r)$, $s_r, \ldots$, where the s's are states, the π's are actions, and the t's are times in $\mathbb{R}^{\geq 0}$. To specify the time bounds an MMTA has a fifth component (with respect to the four components of a GTA) called *task partition*, which is an equivalence relation on the locally controlled actions (i.e., internal and output action). Each equivalence class is called a *task* of the automaton. A task C having at least one enabled action is said *enabled.* Each task C has a lower bound, *lower*(C), and an upper bound *upper*(C), on the time that can elapse before an enabled action belonging to the task C is executed. If the task is not enabled then there is no restriction.

There is a standard technique that transforms any MMTA into a GTA (see [24], Section 23.1). This technique can be extended to transform any MMTA into a Clock GTA (see [5]). In the rest of the paper we will some times use MMTAs to describe Clock GTAs and when using MMTAs we will always use *lower*$(C) = 0$ and *upper*$(C) = \ell$. The following lemma [5] holds.

Lemma 1 *Consider a regular execution fragment α of a Clock GTA described with the MMTA model, starting from a reachable state s_0 and lasting for more than ℓ time. Then (i) any task C enabled in s_0 either has a step or is disabled within ℓ time, and (ii) any new enabling of C has a subsequent step or disabling within ℓ time, provided that α lasts for more than ℓ time from the enabling of C.*

[3] The name MMT derives from Merritt, Modugno, and Tuttle who introduced this automaton[29].

2.3 Composition of automata

The composition operation allows an automaton representing a complex system to be constructed by composing automata representing simpler system components. The most important characteristic of the composition of automata is that properties of isolated system components still hold when those isolated components are composed with other components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving action π, so do all component automata that have π in their signatures. Since internal actions of an automaton A are intended to be unobservable by any other automaton B, automaton A cannot be composed with automaton B unless the internal actions of A are disjoint from the actions of B. (Otherwise, A's performance of an internal action could force B to take a step.) Moreover, A and B cannot be composed unless the sets of output actions of A and B are disjoint. (Otherwise two automata would have the control of an output action.) When A and B can be composed we say that they are *compatible*. The transitions of the composition are obtained by allowing all the components that have a particular action π in their signature to participate, simultaneously, in steps involving π, while all the other components do nothing. Note that this implies that all the components participate in time-passage steps, with the same amount of time passing for all of them.

For a formal definition of the composition operation we refer the reader to [24], Section 23.2.3. Here we recall the following theorems.

Theorem 2 *The composition of a compatible collection of GTAs is a GTA.*

Given the execution $\alpha = s_0, \pi_1, s_1, ...$, of a GTA A obtained by composing a compatible collection $\{A_i\}_{i \in I}$ of GTAs, the notation $\alpha | A_i$ denotes the sequence obtained from α by deleting each pair π_r, s_r for which π_r is not action of A_i and by replacing each remaining s_r by $(s_r)_i$, that is, automaton A_i's piece of s_r.

Theorem 3 *Let $\{A_i\}_{i \in I}$ be a compatible collection of GTAs and let A be the composition of A_i, for all $i \in I$. If α is an execution of A, then $\alpha | A_i$ is an execution of A_i, for every $i \in I$.*

The above theorem is important because it enables us to claim that properties proven to be true for a particular automaton A are still true for a bigger automaton obtained by composing automaton A with other automata. We will make extensive use of this theorem in the rest of the paper.

Clock GTAs are GTAs; hence, they can be composed as GTAs are composed. However we point out that the composition of Clock GTAs does not yield a

Clock GTA but a GTA.

3 The distributed setting

In this section we discuss the distributed setting. We consider a partially synchronous distributed system consisting of n processes. The distributed system provides a bidirectional channel for every two processes. Each process is uniquely identified by its identifier $i \in \mathcal{I}$, where $\mathcal{I}$ is a totally ordered finite set of n identifiers. The set $\mathcal{I}$ is known by all the processes. Moreover each process of the system has a local clock. Local clocks can run at different speeds, though in general we expect them to run at the same speed as real time. We assume that a local clock is available also for channels; though this may seem somewhat strange, it is just a formal way to express the fact that a channel is able to deliver a given message within a fixed amount of time, by relying on some timing mechanism (which we model with the local clock). We use Clock GTAs to model both processes and channels.

Throughout the rest of the paper we use two constants, ℓ and d, to represent upper bounds on the time needed to execute an enabled action and to deliver a message, respectively. These time bounds do not necessarily hold for every action and message in every execution; a violation of these bounds is a timing failure.

3.1 Processes

We allow process stopping failures and recoveries, and timing failures. To formally model process stops and recoveries we model process i with a Clock GTA which has a special state component called $Status_i$ and two input actions Stop$_i$ and Recover$_i$. The state variable $Status_i$ reflects the current condition of process i. The effect of action Stop$_i$ is to set $Status_i$ to `stopped`, while the effect of Recover$_i$ is to set $Status_i$ to `alive`. Moreover when $Status_i = \texttt{stopped}$, all the locally controlled actions are not enabled and the input actions have no effect, except for action Recover$_i$. We say that a process i is *alive* (resp. *stopped*) in a given state s if we have $s.Status_i = \texttt{alive}$ (resp. $s.Status_i = \texttt{stopped}$). We say that a process i is alive (resp. stopped) in a given execution fragment, if it is alive (resp. stopped) in all the states of the execution fragment. An automaton modelling a process is called a *process automaton*.

Between a failure and a recovery a process does not lose its state. We remark that PAXOS needs only a small amount of stable storage (see Section 6.6); however, for simplicity, we assume that the entire state of a process is stable.

We also assume that there is an upper bound of ℓ on the elapsed clock time if some locally controlled action is enabled. This time bound can be violated if timing failures happen.

Finally, we provide the following definition of "stable" execution fragment of a given process automaton. This definition is used later to define a stable execution of a distributed system.

Definition 3.1 *Given a process automaton* PROCESS_i*, we say that an execution fragment* α *of* PROCESS_i *is stable if process* i *is either stopped or alive in* α *and* α *is regular.*

3.2 Channels

We consider unreliable channels that can lose and duplicate messages. Reordering of messages is allowed, i.e., is not considered a failure. Timing failures are also possible. Figure 1 shows the code of a Clock GTA $\text{CHANNEL}_{i,j}$, which models the communication channel from process i to process j; there is one automaton for each possible choice of i and j. Notice that we allow the possibility that the sender and the receiver are the same process. We denote by $\mathcal{M}$ the set of messages that can be sent over the channels.

```
CHANNEL$_{i,j}$

Signature:
  Input:          Send$(m)_{i,j}$, Lose$_{i,j}$, Duplicate$_{i,j}$
  Output:         Receive$(m)_{i,j}$
  Time-passage:   $\nu(t)$

State:
  $Clock \in \mathbb{R}$, initially arbitrary
  $Msgs$, a set of elements of $\mathcal{M} \times \mathbb{R}$, initially empty

Actions:

input Send$(m)_{i,j}$
  Eff:  add $(m, Clock)$ to $Msgs$

output Receive$(m)_{i,j}$
  Pre:  $(m,t)$ is in $Msgs$, for some $t$
  Eff:  remove $(m,t)$ from $Msgs$

input Lose$_{i,j}$
  Eff:  remove one element of $Msgs$

input Duplicate$_{i,j}$
  Eff:  let $(m,t)$ be an element of $Msgs$
        let $t'$ such that $t \leq t' \leq Clock$
        place $(m,t')$ into $Msgs$

time-passage $\nu(t)$
  Pre:  Let $t' \geq 0$ be such that
          for all $(m,t'') \in Msgs$
            $Clock + t' \leq t'' + d$
  Eff:  $Clock := Clock + t'$
```

Fig. 1. Automaton $\text{CHANNEL}_{i,j}$

The time-passage actions of $\text{CHANNEL}_{i,j}$ do not let pass the time beyond $t'' + d$ if a message (m, t''), that is, a message m sent at time t'', is in the channel. Clearly this restriction is on the local time and messages can also be lost. However if the execution is regular and no messages are lost then a particular message is delivered in a timely manner. The following definition of "stable" execution fragment for a channel captures the condition under which messages are delivered on time.

Definition 3.2 *Given a channel* $\text{CHANNEL}_{i,j}$*, we say that an execution fragment* α *of* $\text{CHANNEL}_{i,j}$ *is stable if no* $\mathit{Lose}_{i,j}$ *and* $\mathit{Duplicate}_{i,j}$ *actions occur in* α *and* α *is regular.*

We remark that the above definition requires also that no $\text{Duplicate}_{i,j}$ actions happen. This is needed for the performance analysis (duplicated messages may introduce delays in the PAXOS algorithm).

The next lemma follows from the above discussion.

Lemma 4 *In a stable execution fragment* α *of* $\text{CHANNEL}_{i,j}$ *beginning in a reachable state* s *and lasting for more than* d *time, we have that* (i) *all messages* (m, t) *that in state* s *are in* $\mathit{Msgs}_{i,j}$ *are delivered by time* d*, and* (ii) *any message sent in* α *is delivered within time* d *of the sending, provided that* α *lasts for more than* d *time from the sending of the message.*

3.3 Distributed systems

A distributed system is the composition of automata modelling channels and processes. We are interested in modelling bad and good behaviors of a distributed system; in order to do so we provide some definitions that characterize the behavior of a distributed system. The definition of "nice" execution fragment given later in this section, captures the good behavior of a distributed system. Informally, a distributed system behaves nicely if there are no process failures and recoveries, no channel failures and no irregular steps—remember that an irregular step models a timing failure—and a majority of the processes are alive.

Definition 3.3 *A communication system for the set* $\mathcal{I}$ *of processes, is the composition of channel automata* $\text{CHANNEL}_{i,j}$ *for all possible choices of* $i, j \in \mathcal{I}$*.*

Definition 3.4 *A distributed system is the composition of process automata modeling the set* $\mathcal{I}$ *of processes and a communication system for* $\mathcal{I}$*.*

We define the communication system S_{CHA} to be the communication system for the set $\mathcal{I}$ of all processes.

Next we provide the definition of "stable" execution fragment for a distributed system exploiting the definition of stable execution fragment given previously for channels and process automata.

Definition 3.5 *Given a distributed system S, we say that an execution fragment α of S is stable if: (i) for all automata* PROCESS_i *modelling process i, $i \in S$ it holds that $\alpha|\text{PROCESS}_i$ is a stable execution fragment for process i; (ii) for all channels $\text{CHANNEL}_{i,j}$ with $i, j \in S$ it holds that $\alpha|\text{CHANNEL}_{i,j}$ is a stable execution fragment for $\text{CHANNEL}_{i,j}$.*

Finally we provide the definition of "nice" execution fragment that captures the conditions under which PAXOS satisfies termination.

Definition 3.6 *Given a distributed system S, we say that an execution fragment α of S is nice if α is a stable execution fragment and a majority of the processes are alive in α.*

The above definition requires a majority of processes to be alive. As is explained in Section 6.6, any quorum scheme could be used instead of majorities.

In the rest of the paper, we will often use the word "system" to mean "distributed system".

4 The consensus problem

Several different but related agreement problems have been considered in the literature. All have in common that processes start the computation with initial values and at the end of the computation each process must reach a decision. The variations mostly concern stronger or weaker requirements that the solution to the problem has to satisfy. The requirement that a solution to the problem has to satisfy are captured by three properties, usually called *agreement, validity* and *termination.* It is clear that the definition of the consensus problem must take into account the distributed setting in which the problem is considered.

We assume that for each process i there is an external agent that provides an initial value v by means of an action $\text{Init}(v)_i$. We denote by V the set of possible initial values and, given a particular execution α, we denote by V_α the subset of V consisting of those values actually used as initial values in α, that is, those values provided by $\text{Init}(v)_i$ actions executed in α. A process outputs a decision v by executing an action Decide$(v)_i$. If a process i executes action Decide$(v)_i$ more than once then the output value v must be the same.

To solve the consensus problem means to give a distributed algorithm that, for any execution α of the system, satisfies

- **Agreement:** All the Decide(v) actions in α have the same v.
- **Validity:** For any Decide(v) action in α, v belongs to V_α.

and, for any admissible execution α, satisfies

- **Termination:** If $\alpha = \beta\gamma$ and γ is a nice execution fragment and for each process i alive in γ an Init$(v)_i$ action occurs in α while process i is alive, then any process i alive in γ, executes a Decide$(v)_i$ action in α.

The agreement and termination conditions require, as one can expect, that processes "agree" on a particular value. The validity condition is needed to relate the output value to the input values (otherwise a trivial solution, i.e., always output a default value, exists).

5 Failure detector and leader elector

In this section we provide a failure detector algorithm and then we use it to implement a leader election algorithm, which, in turn, is used in Section 6 to implement PAXOS. The failure detector and the leader elector we implement here are both sloppy, meaning that they are guaranteed to give accurate information on the system only in a stable execution. However, this is enough for implementing PAXOS.

5.1 A failure detector

In this section we provide an automaton that detects process failures and recoveries. This automaton satisfies certain properties that we will need in the rest of the paper. We do not provide a formal definition of the failure detection problem, however, roughly speaking, the failure detection problem is the problem of checking which processes are alive and which ones are stopped.

Figure 2 shows a Clock GTA, called DETECTOR$(z,c)_i$, which detects failures. In our setting failures and recoveries are modeled by means of actions Stop$_i$ and Recover$_i$. These two actions are input actions of DETECTOR$(z,c)_i$. Moreover DETECTOR$(z,c)_i$ has InformStopped$(j)_i$ and InformAlive$(j)_i$ as output actions which are executed when, respectively, the stopping and the recovering of process j are detected.

```
DETECTOR$(z, c)_i$

Signature:
  Input:          Receive$(m)_{j,i}$, Stop$_i$,Recover$_i$
  Output:         InformStopped$(j)_i$, InformAlive$(j)_i$, Send$(m)_{i,j}$
  Internal:       Check$(j)_i$
  Time-passage:   $\nu(t)$

State:
                                                for all $j \in \mathcal{I}$:
$Clock \in \mathbb{R}$                   init. arbitrary     $Prevrec(j) \in \mathbb{R}^{\geq 0}$     init. arbitrary
$Status \in \{alive, stopped\}$          init. alive         $Lastinform(j) \in \mathbb{R}^{\geq 0}$  init. $Clock$
$Alive \in 2^{\mathcal{I}}$              init. $\mathcal{I}$           $Lastsend(j) \in \mathbb{R}^{\geq 0}$    init. $Clock$
                                                    $Lastcheck(j) \in \mathbb{R}^{\geq 0}$   init. $Clock$

Actions:

input Stop$_i$
  Eff: $Status$ := stopped

output Send(“Alive”)$_{i,j}$
  Pre: $Status$ = alive
  Eff: $Lastsend(j) := Clock + z$

input Receive(“Alive”)$_{j,i}$
  Eff: if $Status$ = alive then
         $Prevrec(j) := Clock$
         if $j \notin Alive$ then
           $Alive := Alive \cup \{j\}$
           $Lastcheck(j) := Clock + c$

internal Check$(j)_i$
  Pre: $Status$ = alive
       $j \in Alive$
  Eff: $Lastcheck(j) := Clock + c$
       if $Clock > Prevrec(j) + z + d$
         then $Alive := Alive \setminus \{j\}$

input Recover$_i$
  Eff: $Status$ := alive

output InformStopped$(j)_i$
  Pre: $Status$ = alive
       $j \notin Alive$
  Eff: $Lastinform(j) := Clock + \ell$

output InformAlive$(j)_i$
  Pre: $Status$ = alive
       $j \in Alive$
  Eff: $Lastinform(j) := Clock + \ell$

time-passage $\nu(t)$
  Pre: none
  Eff: if $Status$ = alive then
         Let $t'$ be such that
           $\forall j, Clock + t' \leq Lastinform(j)$
           $\forall j, Clock + t' \leq Lastsend(j)$
           $\forall j, Clock + t' \leq Lastcheck(j)$
       $Clock := Clock + t'$
```

Fig. 2. Automaton DETECTOR for process i

Automaton DETECTOR$(z, c)_i$ works by having each process constantly sending “Alive” messages to each other process and checking that such messages are received from other processes. It sends at least one “Alive” message in an interval of time of a fixed length z (i.e., if an “Alive” message is sent at time t then the next one is sent before time $t + z$) and checks for incoming messages at least once in an interval of time of a fixed length c. Let us denote by S_{DET} the system consisting of system S_{CHA} and an automaton DETECTOR$(z, c)_i$ for each process $i \in \mathcal{I}$.

For simplicity of notation, henceforth we assume that $z = \ell$ and $c = \ell$, that is, we use DETECTOR$(\ell, \ell)_i$. In practice the choice of z and c may be different.

Using the strategy used by DETECTOR$(\ell, \ell)_i$ it is not hard to prove the following lemmas (for a detailed formal proof we refer the interested reader to [5]).

Lemma 5 *If an execution fragment α of S_{DET}, starting in a reachable state and lasting for more than $3\ell + 2d$ time, is stable and process i is stopped in α, then by time $3\ell + 2d$ from the beginning of α, for each process j alive in α, an action InformStopped$(i)_j$ is executed and no subsequent InformAlive$(i)_j$ action is executed in α.*

Lemma 6 *If an execution fragment α of S_{DET}, starting in a reachable state and lasting for more than $d + 2\ell$ time, is stable and process i is alive in α, then by time $d + 2\ell$ from the beginning of α, for each process j alive in α, an action InformAlive$(i)_j$ is executed and no subsequent InformStopped$(i)_j$ action is executed in α.*

The strategy used by DETECTOR$_i$ is a straightforward one. For this reason it is very easy to implement. However the failure detector so obtained is not reliable, i.e., it does not give accurate information, in the presence of failures (Stop$_i$, Lose$_{i,j}$, irregular executions). For example, it may consider a process stopped just because the "Alive" message of that process was lost in the channel. Automaton DETECTOR$_i$ is guaranteed to provide accurate information on faulty and alive processes only when the system is stable.

5.2 A leader elector

Electing a leader in an asynchronous distributed system is a difficult task. An informal argument that explains this difficulty is that the leader election problem is somewhat similar to the consensus problem (which, in an asynchronous system subject to failures is unsolvable [14]) in the sense that to elect a leader all processes must reach consensus on which one is the leader. It is fairly clear how a failure detector can be used to elect a leader. Indeed the failure detector gives information on which processes are alive and which ones are not alive. This information can be used to elect the current leader. We use the DETECTOR$(\ell, \ell)_i$ automaton to check for the set of alive processes. Figure 3 shows automaton LEADERELECTOR$_i$ which is an MMTA. Remember that we use MMTAs to describe in a simpler way Clock GTAs. Automaton LEADERELECTOR$_i$ interacts with DETECTOR$(\ell, \ell)_i$ by means of actions InformStopped$(j)_i$, which inform process i that process j has stopped, and

InformAlive$(j)_i$, which inform process i that process j has recovered. Each process updates its view of the set of alive processes when these two actions are executed. The process with the biggest identifier in the set of alive processes is declared leader. We denote with S_{LEA} the system consisting of S_{DET} composed with a LEADERELECTOR$_i$ automaton for each process $i \in \mathcal{I}$. Figure 4 shows S_{LEA}; it also shows S_{DET}, which is a subsystem of S_{LEA}.

```
LEADERELECTOR$_i$

Signature:
  Input:    InformStopped$(j)_i$, InformAlive$(j)_i$, Stop$_i$, Recover$_i$
  Output:   Leader$_i$, NotLeader$_i$

State:
  $Status \in \{\texttt{alive}, \texttt{stopped}\}$      initially alive
  $Pool \in 2^{\mathcal{I}}$                          initially $\{i\}$

Derived variable:
  Leader, defined as max of Pool

Actions:

  input Stop$_i$
    Eff: $Status :=$ stopped

  input Recover$_i$
    Eff: $Status :=$ alive

  output Leader$_i$
    Pre: $Status =$ alive
         $i = Leader$
    Eff: none

  output NotLeader$_i$
    Pre: $Status =$ alive
         $i \neq Leader$
    Eff: none

  input InformStopped$(j)_i$
    Eff: if $Status =$ alive then
           $Pool := Pool \setminus \{j\}$

  input InformAlive$(j)_i$
    Eff: if $Status =$ alive
           $Pool := Pool \cup \{j\}$

Tasks and bounds:
{Leader$_i$, NotLeader$_i$}, bounds $[0, \ell]$
```

Fig. 3. Automaton LEADERELECTOR for process i

Since DETECTOR$(\ell, \ell)_i$ is not a reliable failure detector, also LEADERELECTOR$_i$ is not reliable. Thus, it is possible that processes have different views of the system so that more than one process considers itself leader, or the process supposed to be the leader is actually stopped. However as the failure detector becomes reliable when the system S_{DET} executes a stable execution fragment (see Lemmas 5 and 6), also the leader elector becomes reliable when system S_{LEA} is stable. Notice that when S_{LEA} executes a stable execution fragment, so does S_{DET}.

Formally, we say that a state s of system S_{LEA}, is a *unique-leader state* if there exists an alive process i such that for all alive processes j it holds that $s.Leader_j = i$. In such a case, process i is the *leader* of state s. Moreover we

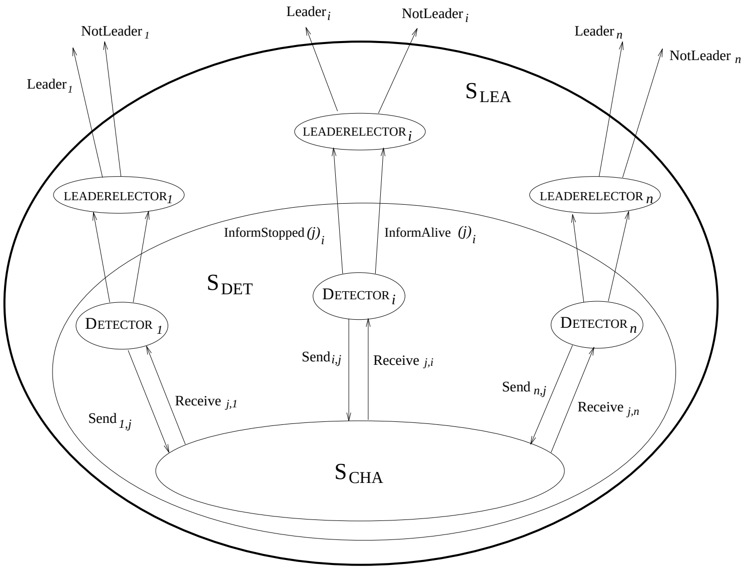


Fig. 4. The system S_{LEA}

say that an execution α of system S_{LEA}, is a *unique-leader execution* if all the states of α are unique-leader states with the same leader in all the states.

Next lemma states that in a stable execution fragment, eventually there is unique-leader state.

Lemma 7 *If an execution fragment α of S_{LEA}, starting in a reachable state and lasting for more than $4\ell + 2d$, is stable, then by time $4\ell + 2d$, there is a state occurrence s such that in state s and in all the states after s there is a unique leader. Moreover this unique leader is always the process with the biggest identifier among the processes alive in α.*

Proof: First notice that the system S_{LEA} consists of system S_{DET} composed with other automata. Hence by Theorem 3 we can use any property of S_{DET}. In particular we can use Lemmas 5 and 6 and thus we have that by time $3\ell+2d$ each process has a consistent view of the set of alive and stopped processes. Let i be the leader. Since α is stable and thus also regular, by Lemma 1, within additional ℓ time, actions Leader$_j$ and NotLeader$_j$ are consistently executed for each process j, including process $j = i$. The fact that i is the the process with the biggest identifier among the processes alive in α follows directly from the code of LEADERELECTOR$_i$. □

We remark that, for many algorithms that rely on the concept of leader, it is important to provide exactly one leader. For example when the leader election is used to generate a new token in a token ring network, it is important that there is exactly one process (the leader) that generates the new token, because the network gives the right to send messages to the owner of the token and two tokens may result in an interference between two communications. For these algorithms, having two or more leaders jeopardizes the correctness. Hence the sloppy leader elector provided before is not suitable. However for the purpose of this paper, LEADERELECTOR$_i$ is all we need.

6 The PAXOS algorithm

PAXOS was devised a very long time ago[4] but its discovery, due to Lamport, is very recent [19].

In this section we describe the PAXOS algorithm, provide an implementation using Clock GT automata, prove its correctness and analyze its performance. The performance analysis is given assuming that there are no failures nor recoveries, and a majority of the processes are alive for a sufficiently long time. We remark that when no restrictions are imposed on the possible failures, the algorithm might not terminate.

6.1 Overview

Our description of PAXOS is modular: we have separated various parts of the overall algorithm; each piece copes with a particular aspect of the problem. This approach should make the understanding of the algorithm much easier. The core part of the algorithm is a module that we call BASICPAXOS; this piece incorporates the basic ideas on which the algorithm itself is built. The description of this piece is further subdivided into three components, namely BPLEADER, BPAGENT and BPSUCCESS.

In BASICPAXOS processes try to reach a decision by running what we call a "round". A process starting a round is the leader of that round. BASICPAXOS guarantees that, no matter how many leaders start rounds, agreement and validity are not violated. This means that in any run of the algorithm no two different decisions are ever made and any decision is equal to some input value. However to have a complete algorithm that satisfies termination when there are no failures for a sufficiently long time, we need to augment BASICPAXOS with another module; we call this module STARTER. The functionality of STARTER is to make the current leader start a new round if the previous one is not completed within some time bound.

[4] The most accurate information dates it back to the beginning of this millennium [19].

Leaders are elected by using the LEADERELECTOR algorithm provided in Chapter 5. We remark that this is possible because the presence of two or more leaders does not jeopardize agreement or validity; however, to get termination there must be a unique leader.

Thus, our implementation of PAXOS is obtained by composing the following automata: $\text{CHANNEL}_{i,j}$ for the communication between processes, DETECTOR_i and LEADERELECTOR_i for the leader election, BASICPAXOS_i and STARTER_i, for every process $i, j \in \mathcal{I}$. The resulting system is called S_{PAX}.

Figure 5 shows the automaton at process i. Notice that not all of the actions are drawn in the picture: we have drawn only some of them and we refer to the formal code for all of the actions. Actions Stop_i and Recover_i are input actions of all the automata. The S_{PAX} automaton at process i interacts with automata at other processes by sending messages over the channels. Channels are not drawn in the picture.

Figure 6 shows the messages exchanged by processes i and j. The automata that send and receive these messages are shown in the picture. We remark that channels and actions interacting with channels are not drawn, as well as other actions for the interaction with other automata.

It is worth to remark that some pieces of the algorithm do need to be able to measure the passage of the time (DETECTOR_i, STARTER_i and BPSUCCESS_i) while others do not.

We will prove (Theorems 9 and 10) that the system S_{PAX} solves the consensus problem ensuring partial correctness—any output is guaranteed to be correct, that is, agreement and validity are satisfied—and (Theorem 17) that S_{PAX} guarantees also termination when the system executes a nice execution fragment, that is, without failures and recoveries and with at least a majority of the processes remaining alive.

Roadmap for the rest of the section. In Section 6.2 we provide automaton BASICPAXOS. This automaton is responsible for carrying out a round in response to an external request. We prove that any round satisfies agreement and validity and we provide a performance analysis for a successful round. Then in Section 6.3 we provide automaton STARTER which takes care of the problem of starting new rounds. In Section 6.4 we prove that the entire system S_{PAX} is correct and provide a performance analysis. In Section 6.5 we provide some comments about the number of messages used by the algorithm. Finally Section 6.6 contains some concluding remarks.

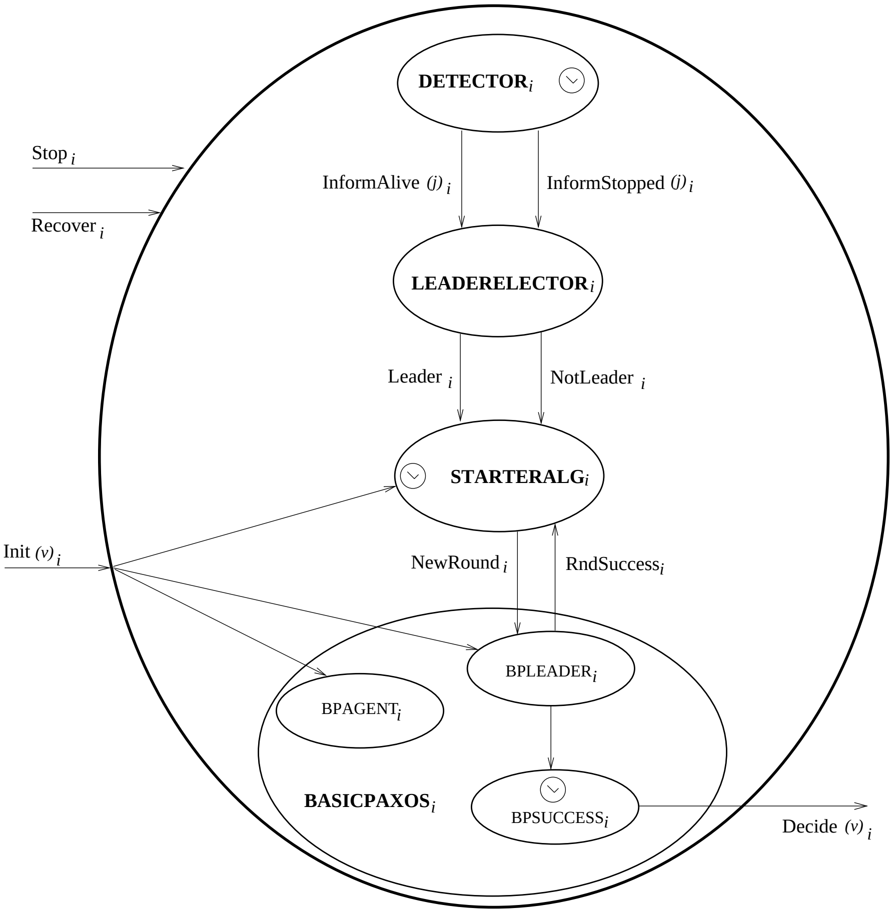


Fig. 5. PAXOS: process i. Some of the actions shown in the figure will be defined later in this section.

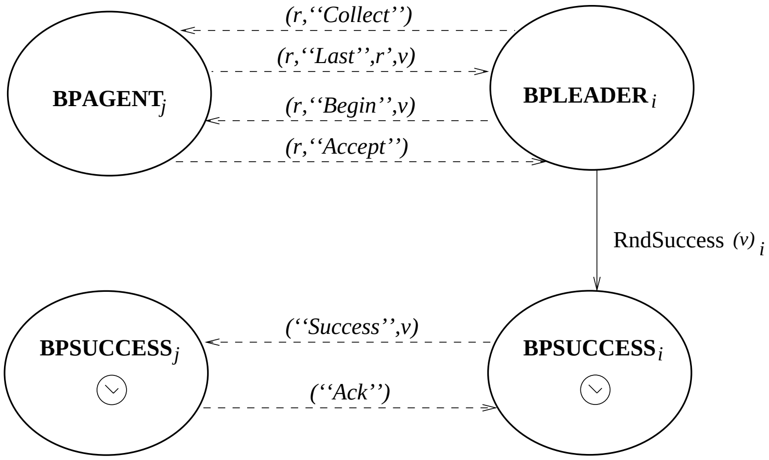


Fig. 6. BASICPAXOS: Messages.

6.2 Automaton BASICPAXOS

In this section we present the automaton BASICPAXOS which is the core part of the PAXOS algorithm. We begin by providing an overview of how automaton BASICPAXOS works, then we provide the automaton code along with a detailed description and finally we prove that it satisfies agreement and validity.

6.2.1 Overview

The basic idea is to have processes propose values until one of them is accepted by a majority of the processes; that value is the final output value. Any process may propose a value by initiating a *round* for that value. The process initiating a round is said to be the *leader* of that round while all processes, including the leader itself, are said to be *agents* for that round. Informally, the steps for a round are the following.

(1) To initiate a round, the leader sends a "Collect" message to all agents[5] announcing that it wants to start a new round and at the same time asking for information about previous rounds in which agents may have been involved.

(2) An agent that receives a message sent in step 1 from the leader of the round, responds with a "Last" message giving its own information about rounds previously conducted. With this, the agent makes a kind of commitment for this particular round that may prevent it from accepting (in step 4) the value proposed in some other round. If the agent is already committed for a round with a bigger round number then it informs the leader of its commitment with an "OldRound" message.

(3) Once the leader has gathered information about previous rounds from a majority of agents, it decides, according to some rules, the value to propose for its round and sends to all agents a "Begin" message announcing the value and asking them to accept it. In order for the leader to be able to choose a value for the round it is necessary that initial values be provided. If no initial value is provided, the leader must wait for an initial value before proceeding with step 3. The set of processes from which the leader gathers information is called the *info-quorum* of the round.

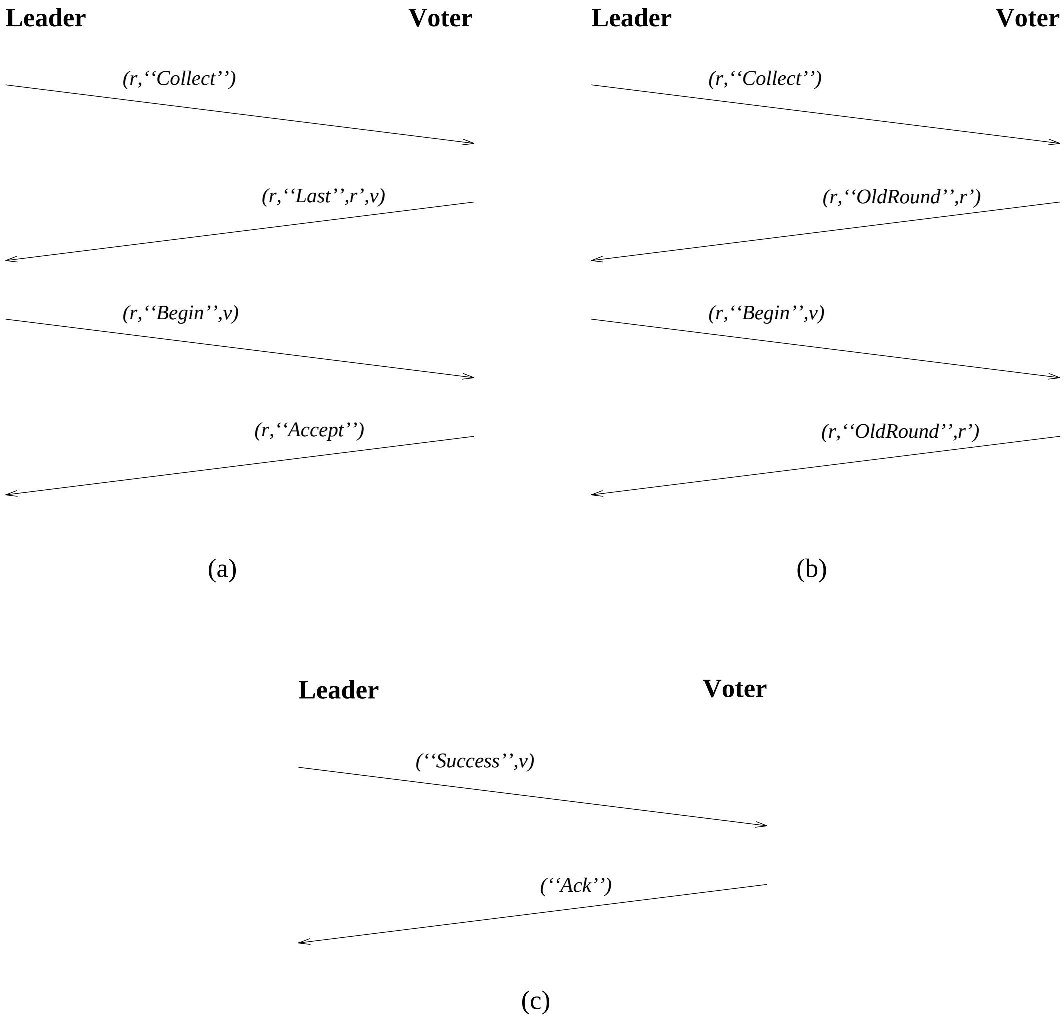
(4) An agent that receives a message from the leader of the round sent in step 3, responds with an "Accept" message by accepting the value proposed in the current round, unless it is committed for a later round and thus must reject the value proposed in the current round. In the latter

[5] Thus it sends a message also to itself. This helps in that we do not have to specify different behaviors for a process according to the fact that it is both leader and agent or just an agent. We just need to specify the leader behavior and the agent behavior.

case the agent sends an “OldRound” message to the leader indicating the round for which it is committed.

(5) If the leader gets “Accept” messages from a majority of agents, then the leader sets its own output value to the value proposed in the round. At this point the round is successful. The set of agents that accept the value proposed by the leader is called the *accepting-quorum*.

Since a successful round implies that the leader of the round reached a decision, after a successful round the leader still needs to do something, namely to broadcast the reached decision. Thus, once the leader has made a decision it broadcasts a “Success” message announcing the value for which it has decided. An agent that receives a “Success” message from the leader makes its decision choosing the value of the successful round. We use also an “Ack” message sent from the agent to the leader, so that the leader can make sure that everyone knows the outcome.



(a)

(b)

(c)

Fig. 7. Exchange of messages

Figure 7 shows: (a) the steps of a successful round r; (b) the responses from an agent that informs the leader that an higher numbered round r' has been already initiated; (c) the broadcast of a decision. The parameters used in the messages will be explained later. Section 6.2.2 contains a description of the messages.

Since different rounds may be carried out concurrently (several processes may concurrently initiate rounds), we need to distinguish them. Every round has a unique identifier. Next we formally define these round identifiers. A *round number* is a pair (x, i) where x is a nonnegative integer and i is a process identifier. The set of round numbers is denoted by $\mathcal{R}$. A total order on elements of $\mathcal{R}$ is defined by $(x, i) < (y, j)$ iff $x < y$ or, $x = y$ and $i < j$.

We say that round r *precedes* round r' if $r < r'$.

If round r precedes round r' then we also say that r is a *previous* round, with respect to round r'. We remark that the ordering of rounds is not related to the actual time the rounds are conducted. It is possible that a round r' is started at some point in time and a previous round r, that is, one with $r < r'$, is started later on.

For each process i, we define a "$+_i$" operation that given a round number (x, j) and an integer y, returns the round number $(x, j) +_i y = (x + y, i)$.

Every round in the algorithm is tagged with a unique round number. Every message sent by the leader or by an agent for a round (with round number) $r \in \mathcal{R}$, carries the round number r so that no confusion among messages belonging to different rounds is possible.

However the most important issue is about the values that leaders propose for their rounds. Indeed, since the value of a successful round is the output value of some processes, we must guarantee that the values of successful rounds are all equal in order to satisfy the agreement condition of the consensus problem. This is the tricky part of the algorithm and basically all the difficulties derive from solving this problem. Consistency is guaranteed by choosing the values of new rounds exploiting the information about previous rounds from at least a majority of the agents so that, for any two rounds, there is at least one process that participated in both rounds.

In more detail, the leader of a round chooses the value for the round in the following way. In step 1, the leader asks for information and in step 2 an agent responds with the number of the latest round in which it accepted the value and with the accepted value or with round number $(0, j)$ and `nil` if the agent has not yet accepted a value. Once the leader gets such information from a majority of the agents (which is the info-quorum of the round), it chooses the value for its round to be equal to the value of the latest round among all those it has heard from the agents in the info-quorum or equal to its initial value if all agents in the info-quorum were not involved in any previous round. Moreover, in order to keep consistency, if an agent tells the leader of a round r that the last round in which it accepted a value is round r', $r' < r$, then implicitly the agent commits itself not to accept any value proposed in any other round r'', $r' < r'' < r$.

Given the above setting, if r' is the round from which the leader of round r gets the value for its round, then, when a value for round r has been chosen, any round r'', $r' < r'' < r$, cannot be successful; indeed at least a majority of the processes are committed for round r, which implies that at least a majority of the processes are rejecting round r''. This, along with the fact that info-quorums and accepting-quorums are majorities, implies that if a round r is successful, then any round with a bigger round number $\tilde{r} > r$ is for the same value. Indeed the information sent by processes in the info-quorum of round $\tilde{r}$ is used to choose the value for the round, but since info-quorums and accepting-quorums share at least one process, at least one of the processes in the info-quorum of round r' is also in the accepting-quorum of round r. Indeed, since the round is successful, the accepting-quorum is a majority. This implies that the value of any round $\tilde{r} > r$ must be equal to the value of round r, which, in turn, implies agreement.

We remark that instead of majorities for info-quorums and accepting-quorums, any quorum system can be used. Indeed the only property that is required is that there be a process in the intersection of any info-quorum with any accepting-quorum.

Example. Figure 8 shows how the value of a round is chosen. In this example we have a network of 5 processes, A, B, C, D, E (where the ordering is the alphabetical one) and v_A, v_B denote the initial values of A and B. At some point process B is the leader and starts round $(1, B)$. It receives information from A, B, E (the set $\{A, B, E\}$ is the info-quorum of this round). Since none of them has been involved in a previous round, process B is free to choose its initial value v_B as the value of the round. However it receives acceptance only from B, C (the set $\{B, C\}$ is the accepting-quorum for this round). Later, process A becomes the leader and starts round $(2, A)$. The info-quorum for this round is $\{A, D, E\}$. Since none of this processes has accepted a value in a previous round, A is free to choose its initial value for its round. For round $(2, D)$ the info-quorum is $\{C, D, E\}$. This time in the quorum there is process C that has accepted a value in round $(1, B)$ so the value of this round must be the same of that of round $(1, B)$. For round $(3, A)$ the info-quorum is $\{A, B, E\}$ and since A has accepted the value of round $(2, A)$ then the value of round $(2, A)$ is chosen for round $(3, A)$. For round $(3, B)$ the info-quorum is $\{A, C, D\}$. In this case there are three processes that accepted values in previous rounds: process A that has accepted the value of round $(2, A)$ and processes C, D, that have accepted the value of round $(2, D)$. Since round $(2, D)$ is the higher round

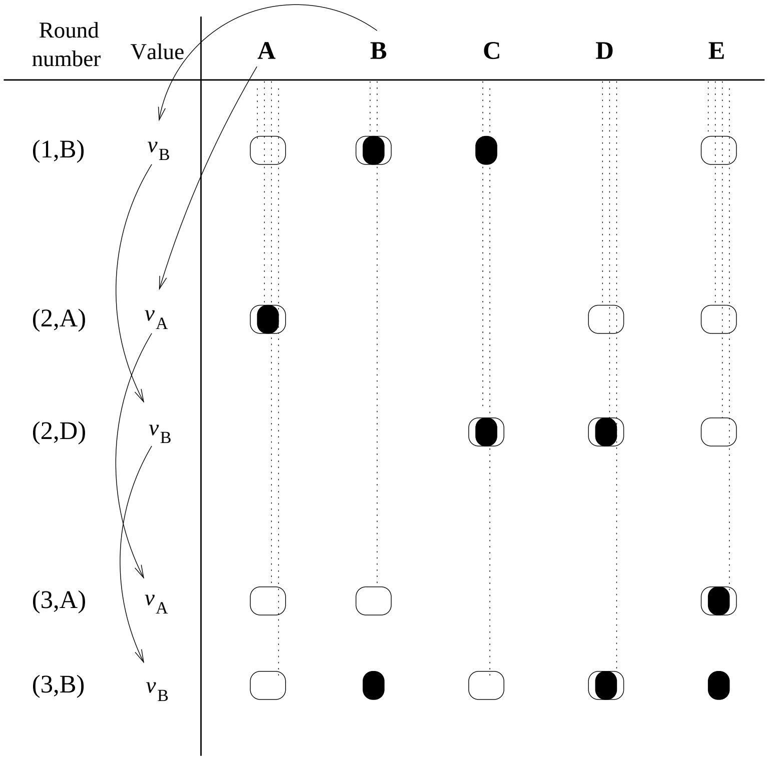


Fig. 8. Choosing the values of rounds. Empty boxes denote that the process is in the info-quorum, and black boxes denote acceptance. Dotted lines indicate commitments.

number, the value for round $(3, B)$ is taken from round $(2, D)$. Round $(3, B)$ is successful.

To end up with a decision value, rounds must be started until at least one is successful. The basic consensus module BASICPAXOS guarantees that a new round does not violate agreement or validity, that is, the value of a new round is chosen in such a way that if the round is successful, it does not violate agreement and validity. However, it is necessary to make BASICPAXOS start rounds until one is successful. We deal with this problem in Section 6.3.

6.2.2 The code

In order to describe automaton BASICPAXOS$_i$ for process i we provide three automata. One is called BPLEADER$_i$ and models the "leader" behavior of the process; another one is called BPAGENT$_i$ and models the "agent" behavior of the process; the third one is called BPSUCCESS$_i$ and it simply takes care of broadcasting a reached decision. Automaton BASICPAXOS$_i$ is the composition of BPLEADER$_i$, BPAGENT$_i$ and BPSUCCESS$_i$.

Figures 9 and 10 show the code for BPLEADER$_i$, while Figure 11 shows the code for BPAGENT$_i$. We remark that these code fragments are written using the MMTA model. Remember that we use MMTA to describe in a simpler way Clock GT automata. Figures 12 and 13 show automaton BPSUCCESS$_i$. The purpose of this automaton is simply to broadcast the decision once it has been reached by the leader of a round. Figures 6 and 7 describe the exchange of messages used in a round.

It is worth noticing that the code fragments are "tuned" to work efficiently when there are no failures. Indeed messages for a given round are sent only once, that is, no attempt is made to try to cope with losses of messages and responses are expected to be received within given time bounds. Other strategies to try to conduct a successful round even in the presence of some failures could be used. For example, messages could be sent more than once to cope with the loss of some messages or a leader could wait more than the minimum required time before abandoning the current round and starting a new one—this is actually dealt with in Section 6.3. We have chosen to send only one message for each step of the round: if the execution is nice, one message is enough to conduct a successful round. Once a decision has been made, there is nothing to do but try to send it to others. Thus once the decision has been made by the leader, the leader repeatedly sends the decision to the agents until it gets an acknowledgment. We remark that also in this case, in practice, it is important to choose appropriate time-outs for the re-sending of a message; in our implementation we have chosen to wait the minimum amount of time required by an agent to respond to a message from the leader; if the execution is stable this is enough to ensure that only one message announcing the decision is sent to each agent.

We remark that there is some redundancy that derives from having separate automata for the leader behavior and for the broadcasting of the decision. For example, both automata BPLEADER$_i$ and BPSUCCESS$_i$ need to be aware of the decision, thus both have a *Decision* variable (the *Decision* variable of BPSUCCESS$_i$ is updated when action RndSuccess$_i$ is executed by BPLEADER$_i$ after the *Decision* variable of BPLEADER$_i$ is set). Having only one automaton would have eliminated the need of such a duplication. However we preferred to separate BPLEADER$_i$ and BPSUCCESS$_i$ because they accomplish different tasks.

In addition to the code fragments of BPLEADER$_i$, BPAGENT$_i$ and BPSUCCESS$_i$, we provide here some comments about the messages, the state variables and the actions.

Messages. In this paragraph we describe the messages used for communication between the leader i and the agents of a round. Every message m is a

BPLEADER$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$“Last”, “Accept”,“OldRound”$\}$
Init$(v)_i$, NewRound$_i$, Stop$_i$, Recover$_i$, Leader$_i$, NotLeader$_i$
Internal: Collect$_i$, BeginCast$_i$,
GatherLast$(m)_i$, m is a “Last” message
GatherAccept$(m)_i$, m is a “Accept” message
GatherOldRound$(m)_i$ m is a “OldRound” message
Output: Send$(m)_{i,j}$, $m \in \{$“Collect”, “Begin”$\}$
Gathered$(v)_i$, Continue$_i$, RndSuccess$(v)_i$

State:

$\mathit{Status} \in \{$`alive,stopped`$\}$	init. `alive`	$\mathit{CurRnd} \in \mathcal{R}$	init. $(0, i)$
$\mathit{IamLeader}$, a boolean	init. `false`	$\mathit{HighestRnd} \in \mathcal{R}$	init. $(0, i)$
$\mathit{Mode} \in \{$`collect,gatherlast,`		$\mathit{Value} \in V \cup \{$`nil`$\}$	init. `nil`
`wait, begincast,`		$\mathit{ValFrom} \in \mathcal{R}$	init. $(0, i)$
`gatheraccept,`		$\mathit{InfoQuo} \in 2^{\mathcal{I}}$	init. $\{\}$
`decided,done`$\}$	init. `done`	$\mathit{AcceptQuo} \in 2^{\mathcal{I}}$	init. $\{\}$
$\mathit{InitValue} \in V \cup$ `nil`	init. `nil`	InMsgs, multiset of msgs	init. $\{\}$
$\mathit{Decision} \in V \cup \{$`nil`$\}$	init. `nil`	$\mathit{OutMsgs}$, multiset of msgs	init. $\{\}$

Derived Variable:

$\mathit{LeaderAlive}$, a boolean, `true` iff $\mathit{Status} =$ `alive` and $\mathit{IamLeader} =$ `true`

Actions:

```
input Stop_i
  Eff: $Status :=$ stopped

input Recover_i
  Eff: $Status :=$ alive

input Leader_i
  Eff: if $Status =$ alive then
         $IamLeader :=$ true

input NotLeader_i
  Eff: if $Status =$ alive then
         $IamLeader :=$ false

output Send$(m)_{i,j}$
  Pre: $Status =$ alive
       $m_{i,j} \in OutMsgs$
  Eff: remove $m_{i,j}$ from $OutMsgs$

input Receive$(m)_{j,i}$
  Eff: if $Status =$ alive then
         add $m_{j,i}$ to $InMsgs$

input Init$(v)_i$
  Eff: if $Status =$ alive then
         $InitValue := v$

input NewRound_i
  Eff: if $LeaderAlive =$ true then
         $CurRnd := HighestRnd +_i 1$
         $HighestRnd := CurRnd$
         $Mode :=$ collect
```

Fig. 9. Automaton BPLEADER for process i (part 1)

BPLEADER$_i$ (cont'd)

```
output Collect$_i$
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{collect}$
  Eff: $ValFrom := (0, i)$
       $InfoQuo := \{\}$
       $AcceptQuo := \{\}$
       $\forall j$ put $\langle CurRnd, \text{“Collect”}\rangle_{i,j}$
         in $OutMsgs$
       $Mode := \texttt{gatherlast}$

internal GatherLast$(m)_i$
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{gatherlast}$
       $m = \langle r, \text{“Last”}, r', v\rangle_{j,i}$
       $m \in InMsgs$
       $CurRnd = r$
  Eff: remove $m$ from $InMsgs$
       $InfoQuo := InfoQuo \cup \{j\}$
       if $ValFrom < r'$ and $v \neq \texttt{nil}$
       then
         $Value := v$
         $ValFrom := r'$
       if $|InfoQuo| > n/2$ then
         $Mode := \texttt{gathered}$

output Gathered$(Value)$
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{gathered}$
  Eff: if $Value = \texttt{nil}$ and
          $InitValue \neq \texttt{nil}$ then
            $Value := InitValue$
       if $Value \neq \texttt{nil}$ then
         $Mode := \texttt{begincast}$
       else
         $Mode := \texttt{wait}$
```

```
internal Continue$_i$
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{wait}$
       $Value = \texttt{nil}$
       $InitValue \neq \texttt{nil}$
  Eff: $Value := InitValue$
       $Mode := \texttt{begincast}$

internal BeginCast$_i$
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{begincast}$
  Eff: $\forall j$, let $m$ be
         $\langle CurRnd, \text{“Begin”}, Value\rangle_{i,j}$
         put $m$ in $OutMsgs$
       $Mode := \texttt{gatheraccept}$

internal GatherAccept$(m)_i$
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{gatheraccept}$
       $m = \langle r, \text{“Accept”}\rangle_{j,i}$
       $m \in InMsgs$
       $CurRnd = r$
  Eff: remove $m$ from $InMsgs$
       $AcceptQuo := AcceptQuo \cup \{j\}$
       if $|AcceptQuo| > n/2$ then
         $Decision := Value$
         $Mode := \texttt{decided}$

output RndSuccess$(Decision)_i$
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{decided}$
  Eff: $Mode := \texttt{done}$

internal GatherOldRound$(m)_i$
  Pre: $Status = \texttt{alive}$
       $m = \langle r, \text{“OldRound”}, r'\rangle_{j,i}$
       $m \in InMsgs$
       $HighestRnd < r'$
  Eff: remove $m$ from $InMsgs$
       $HighestRnd := r'$
```

Tasks and bounds:

$\{$Collect$_i$, Gathered$(v)_i$, Continue$_i$, BeginCast$_i$, RndSuccess$(v)_i\}$, bounds $[0, \ell]$
$\{$GatherLast$(m)_i$, $m \in InMsgs$, m is a “Last” message$\}$, bounds $[0, \ell]$
$\{$GatherAccept$(m)_i$, $m \in InMsgs$, m is a “Accept” message$\}$, bounds $[0, \ell]$
$\{$GatherOldRound$(m)_i$, $m \in InMsgs$, m is a “OldRound” message$\}$, bounds $[0, \ell]$
$\{$Send$(m)_{i,j}$, $m_{i,j} \in OutMsgs\}$, bounds $[0, \ell]$

Fig. 10. Automaton BPLEADER for process i (part 2)

BPAGENT$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$“Collect”, “Begin”$\}$
Init$(v)_i$, Stop$_i$, Recover$_i$
Internal: LastAccept$(m)_i$, m is a “Collect” message
Accept$(m)_i$, m is a “Begin” message
Output: Send$(m)_{i,j}$, $m \in \{$”Last”, “Accept”, “OldRound”$\}$

State:

$Status \in \{\texttt{alive,stopped}\}$	init. `alive`	$Commit \in \mathcal{R}$	init. $(0, i)$
$LastR \in \mathcal{R}$	init. $(0, i)$	$InMsgs$, multiset of msgs	init. $\{\}$
$LastV \in V \cup \{\texttt{nil}\}$	init. `nil`	$OutMsgs$, multiset of msgs	init. $\{\}$

Actions:

```
input Stop_i
  Eff: $Status := \texttt{stopped}$

input Recover_i
  Eff: $Status := \texttt{alive}$

output Send$(m)_{i,j}$
  Pre: $Status = \texttt{alive}$
       $m \in OutMsgs$
  Eff: remove $m_{i,j}$ from $OutMsgs$

input Receive$(m)_{j,i}$
  Eff: if $Status = \texttt{alive}$ then
         add $m_{j,i}$ to $InMsgs$

input Init$(v)_i$
  Eff: if $Status = \texttt{alive}$ then
         if $LastV = \texttt{nil}$ then
           $LastV := v$

internal LastAccept$(m)_i$
  Pre: $Status = \texttt{alive}$
       $m = \langle r, \text{“Collect”}\rangle_{j,i} \in InMsgs$
  Eff: remove $m$ from $InMsgs$
       if $r \geq Commit$ then
         $Commit := r$
         put $\langle r, \text{“Last”}, LastR, LastV\rangle_{i,j}$
           in $OutMsgs$
       else
         put $\langle r, \text{“OldRound”}, Commit\rangle_{i,j}$
           in $OutMsgs$

internal Accept$(m)_i$
  Pre: $Status = \texttt{alive}$
       $m = \langle r, \text{“Begin”}, v\rangle_{j,i} \in InMsgs$
  Eff: remove $m$ from $InMsgs$
       if $r \geq Commit$ then
         put $\langle r, \text{“Accept”}\rangle_{i,j}$ in $InMsgs$
         $LastR := r$, $LastV := v$
       else
         put $\langle r, \text{“OldRound”}, Commit\rangle_{i,j}$
           in $OutMsgs$
```

Tasks and bounds:

$\{$LastAccept$(m)_i$, $m \in InMsgs$, m is a “Collect” message$\}$, bounds $[0, \ell]$
$\{$Accept$(m)_i$, $m \in InMsgs$, m is a “Begin” message$\}$, bounds $[0, \ell]$
$\{$Send$(m)_{i,j}$, $m_{i,j} \in OutMsgs$ $\}$, bounds $[0, \ell]$

Fig. 11. Automaton BPAGENT for process i

```
BPSUCCESS$_i$

Signature:
  Input:          Receive$(m)_{j,i}$, $m \in \{$"Ack", "Success"$\}$
                  Stop$_i$, Recover$_i$, Leader$_i$, NotLeader$_i$, RndSuccess$(v)_i$
  Internal:       SendSuccess$_i$, Check$_i$
  Output:         Send$(m)_{i,j}$, $m \in \{$"Ack", "Success"$\}$
                  Decide$(v)_i$
  Time-passage:   $\nu(t)$

State:
$Clock \in \mathbb{R}$                          init. arbitrary    For each $j \in \mathcal{I}$
$Status \in \{$alive, stopped$\}$               init. alive          $Acked(j)$, a boolean   init. false
$IamLeader$, a boolean                          init. false          $LastSendAck(j) \in \mathbb{R} \cup \{\infty\}$ init. $\infty$
$Decision \in V \cup \{$nil$\}$                 init. nil            $LastSendSuc(j) \in \mathbb{R} \cup \{\infty\}$ init. $\infty$
$Prevsend \in \mathbb{R} \cup \{$nil$\}$        init. nil            $OutAckMsgs(j)$, set of msgs init. $\{\}$
$LastCheck \in \mathbb{R} \cup \{\infty\}$      init. $\infty$       $OutSucMsgs(j)$, set of msgs init. $\{\}$
$LastSS \in \mathbb{R} \cup \{\infty\}$         init. $\infty$

Actions:

input Stop$_i$
  Eff: $Status :=$ stopped

input Leader$_i$
  Eff: if $Status =$ alive and
          $IamLeader =$ false then
         $IamLeader :=$ true
         if $Decision \neq$ nil then
           $LastSS := clock + \ell$
           $PrevSend :=$ nil

output Send$(m)_{i,j}$
  Pre: $Status =$ alive
       $m_{i,j} \in OutAckMsgs(j)$
  Eff: $OutAckMsgs(j) := \{\}$
       $LastSendAck(j) := \infty$
output Send$(m)_{i,j}$
  Pre: $Status =$ alive
       $m_{i,j} \in OutSucMsgs(j)$
  Eff: $OutSucMsgs(j) := \{\}$
       $LastSendSuc(j) := \infty$

input Recover$_i$
  Eff: $Status :=$ alive

input NotLeader$_i$
  Eff: if $Status =$ alive then
         $IamLeader :=$ false
         $LastSS := \infty$
         $LastCheck := \infty$
         For each $j \in \mathcal{I}$
           $LastSendSuc(j) := \infty$

input Receive$(\langle$"Ack"$\rangle)_{j,i}$
  Eff: if $Status =$ alive then
         $Acked(j) :=$ true

input Receive$(\langle$"Success",$v\rangle)_{j,i}$
  Eff: if $Status =$ alive then
         $Decision := v$
         put $\langle$"Ack"$\rangle_{i,j}$ into $OutAckMsgs(j)$
         $LastSendAck(j) := Clock + \ell$
```

Fig. 12. Automaton BPSUCCESS for process i (part 1)

tuple of elements. The messages are:

(1) "Collect" messages, $m = \langle r,$"Collect"$\rangle_{i,j}$. This message is sent by the leader of a round to announce that a new round, with number r, has been started and at the same time to ask for information about previous

```
BPSUCCESS$_i$ (cont'd)

input RndSuccess$(v)_i$
  Eff: if $Status = \mathtt{alive}$ then
         $Decision := v$
         if $IamLeader = \mathtt{true}$ then
           $LastSS := Clock + \ell$
           $PrevSend := \mathtt{nil}$

internal Check$_i$
  Pre: $Status = \mathtt{alive}$
       $PrevSend \neq \mathtt{nil}$
       $t = PrevSend + (2\ell + 2d)$
       $Clock > t$
  Eff: $PrevSend := \mathtt{nil}$
       $LastSS := Clock + \ell$
       $LastCheck := \infty$

output Decide$(v)_i$
  Pre: $Status = \mathtt{alive}$
       $Decision \neq \mathtt{nil}$
       $Decision = v$
  Eff: none

internal SendSuccess$_i$
  Pre: $Status = \mathtt{alive}$
       $IamLeader = \mathtt{true}$
       $Decision \neq \mathtt{nil}$
       $PrevSend = \mathtt{nil}$
       $\exists j \neq i$, $Acked(j) = \mathtt{false}$
  Eff: $\forall j \neq i$ such that $Acked(j) = \mathtt{false}$
         put $\langle$"Success",$Decision\rangle_{i,j}$
           in $OutSucMsgs(j)$
         $LastSendSuc(j) := Clock + \ell$
       $PrevSend := Clock$
       $LastCheck := Clock + (2\ell + 2d) + \ell$
       $LastSS := \infty$

time-passage $\nu(t)$
  Pre: none
  Eff: if $Status = \mathtt{alive}$ then
         Let $t'$ be such that
           $Clock+t' \leq LastCheck$
           $Clock+t' \leq LastSS$
           and for each $j \in \mathcal{I}$
             $Clock+t' \leq LastSendAck(j)$
             $Clock+t' \leq LastSendSuc(j)$
       $Clock := Clock+t'$
```

Fig. 13. Automaton BPSUCCESS for process i (part 2)

rounds.

(2) "Last" messages, $m = \langle r$,"Last",r',$v\rangle_{j,i}$. This message is sent by an agent to respond to a "Collect" message from the leader. It provides the last round r' in which the agent has accepted a value, and the value v proposed in that round. If the agent did not accept any value in previous rounds, then v is either `nil` or the initial value of the agent and r' is $(0, j)$.

(3) "Begin" messages, $m = \langle r$,"Begin",$v\rangle_{i,j}$. This message is sent by the leader of round r to announce the value v of the round and at the same time to ask to accept it.

(4) "Accept" messages, $m = \langle r$, "Accept"$\rangle_{j,i}$. This message is sent by an agent to respond to a "Begin" message from the leader. With this message an agent accepts the value proposed in the current round.

(5) "OldRound" messages, $m = \langle r$, "OldRound", $r'\rangle_{j,i}$. This message is sent by an agent to respond either to a "Collect" or a "Begin" message. It is sent when the agent is committed to reject round r and it informs the leader about round r', which is the higher numbered round for which the agent is committed to reject round r.

(6) "Success" messages, $m = \langle$"Success",$v\rangle_{i,j}$. This message is sent by the leader to broadcast the decision.

(7) "Ack" messages, $m = \langle$"Ack"$\rangle_{j,i}$. This message is an acknowledgment, so that the leader can be sure that an agent has received the "Success" message.

We use the kind of a message to indicate any message of that kind. For example the notation $m \in \{$"Collect", "Begin"$\}$ means that m is either a "Collect" message, that is $m = \langle r,$"Collect"$\rangle$ for some r, or a "Begin" message, that is $m = \langle r,$"Begin"$,v\rangle$ for some r and v.

Automaton BPLEADER$_i$**.** Variable $Status_i$ is used to model process failures and recoveries. Variable $IamLeader_i$ keeps track of whether the process is leader. Variable $Mode_i$ is used like a program counter, to go through the steps of a round. Variable $InitValue_i$ contains the initial value of the process. Variable $Decision_i$ contains the value, if any, decided by process i. Variable $CurRnd_i$ contains the number of the round for which process i is currently the leader. Variable $HighestRnd_i$ stores the highest round number seen by process i. Variable $Value_i$ contains the value being proposed in the current round. Variable $ValFrom_i$ is the round number of the round from which $Value_i$ has been chosen (recall that a leader sets the value for its round to be equal to the value of a particular previous round, which is round $ValFrom_i$). Variable $InfoQuo_i$ contains the set of processes for which a "Last" message has been received by process i (that is, the info-quorum). Variable $AcceptQuo$ contains the set of processes for which an "Accept" message has been received by process i (that is, the accepting-quorum). We remark that in the original paper by Lamport, there is only one quorum which is fixed in the first exchange of messages between the leader and the agents, so that only processes in that quorum can accept the value being proposed. However, there is no need to restrict the set of processes that can accept the proposed value to the info-quorum of the round. Messages from processes in the info-quorum are used only to choose a consistent value for the round, and once this has been done anyone can accept that value. This improvement is also suggested in Lamport's paper [19]. Finally, variables $InMsgs_i$ and $OutMsgs_i$ are buffers used for incoming and outcoming messages.

Actions Stop$_i$ and Recover$_i$ model process failures and recoveries. Actions Leader$_i$ and NotLeader$_i$ are used to update $IamLeader_i$. Actions Send$(m)_{i,j}$ and Receive$(m)_{i,j}$ send messages to the channels and receive messages from the channels. Action Init$(v)_i$ is used by an external agent to set the initial value of process i. Action NewRound$_i$ starts a new round. It sets the new round number by increasing the highest round number ever seen. Action Collect$_i$ resets to the initial values all the variables that describe the status of the round being conducted and broadcasts a "Collect" message. Action GatherLast$(m)_i$ collects the information sent by agents in response to the leader's "Collect" message.

This information is the number of the last round accepted by the agent and the value of that round. Upon receiving these messages, GatherLast$(m)_i$ updates, if necessary, variables *Value*$_i$ and *ValFrom*$_i$. Also it updates the set of processes which eventually will be the info-quorum of the current round. Action GatherLast$(m)_i$ is executed until information is received from a majority of the processes. When "Last" messages have been collected from a majority of the processes, the info-quorum is fixed and GatherLast$(m)_i$ is no longer enabled. At this point action Gathered$(v)_i$ is enabled. If *Value*$_i$ is defined then the value for the round is set, and action BeginCast$_i$ is enabled. If *Value*$_i$ is not defined (and this is possible if the leader does not have an initial value and does not receive any value in "Last" messages) the leader waits for an initial value before enabling action BeginCast$_i$. When an initial value is provided, action Continue$_i$ can be executed and it sets *Value*$_i$ and enables action BeginCast$_i$. Action BeginCast$_i$ broadcasts a "Begin" message including the value chosen for the round. Action GatherAccept$(m)_i$ gathers the "Accept" messages. If a majority of the processes accept the value of the current round then the round is successful and GatherAccept$_i$ sets the *Decision*$_i$ variable to the value of the current round. When variable *Decision*$_i$ has been set, action RndSuccess$(v)_i$ is enabled. Action RndSuccess$_i$ is used to pass the decision to BPSUCCESS$_i$. Action GatherOldRound$(m)_i$ collects messages that inform process i that the round previously started by i is "old", in the sense that a round with a higher number has been started. Process i can update, if necessary, variable *HighestRnd*$_i$.

Automaton BPAGENT$_i$**.** Variable *Status*$_i$ is used to model process failures and recoveries. Variable *LastR*$_i$ is the round number of the latest round for which process i has sent an "Accept" message. Variable *LastV*$_i$ is the value for round *LastR*$_i$. Variable *Commit*$_i$ specifies the round for which process i is committed and thus specifies the set of rounds that process i must reject, which are all the rounds with round number less than *Commit*$_i$. We remark that when an agent commits for a round r and sends to the leader of round r a "Last" message specifying the latest round $r' < r$ in which it has accepted the proposed value, it is enough that the agent commits to not accept the value of any round r'' in between r' and r. To make the code simpler, when an agent commits for a round r, it commits to reject any round $r'' < r$. Finally, variables *InMsgs*$_i$ and *OutMsgs*$_i$ are buffers used for incoming and outcoming messages.

Actions Stop$_i$ and Recover$_i$ model process failures and recoveries. Actions Send$(m)_{i,j}$ and Receive$(m)_{i,j}$ send messages to the channels and receive messages from the channels. Action LastAccept$_i$ responds to the "Collect" message sent by the leader by sending a "Last" message that gives information about the last round in which the agent has been involved. Action Accept$_i$ responds to the "Begin" message sent by the leader. The agent accepts the value of the current round if it is not rejecting the round. In both LastAccept_i and Accept_i actions, if the agent is committed to reject the current round because of a higher numbered round, then an “OldRound” message is sent to the leader so that the leader can update the highest round number ever seen. Action $\text{Init}(v)_i$ sets to v the value of LastV_i only if this variable is undefined. With this, the agent sends its initial value in a “Last” message whenever the agent has not yet accepted the value of any round.

Automaton $\textsc{bpsuccess}_i$**.** Variable Status_i is used to model process failures and recoveries. Variable $\mathit{IamLeader}_i$ keeps track of whether the process is leader. Variable $\mathit{Decision}_i$ stores the decision. Variable $\mathit{Acked}(j)_i$ contains a boolean that specifies whether or not process j has sent an acknowledgment for a “Success” message. Variable $\mathit{Prevsend}_i$ records the time of the previous broadcast of the decision. Variables $\mathit{LastCheck}_i$, LastSS_i, and variables $\mathit{LastSendAck}(j)_i$, $\mathit{LastSendSuc}(j)_i$, for $j \neq i$, are used to impose the time bounds on enabled actions. Their use should be clear from the code. Variables $\mathit{OutAckMsgs}(j)_i$ and $\mathit{OutSucMsgs}(j)_i$, for $j \neq i$, are buffers for outcoming “Ack” and “Success” messages, respectively. There are no buffers for incoming messages because incoming messages are processed immediately, that is, by action $\text{Receive}(m)_{i,j}$.

Actions Stop_i and Recover_i model process failures and recoveries. Actions Leader_i and NotLeader_i are used to update $\mathit{IamLeader}_i$. Actions $\text{Send}(m)_{i,j}$ and $\text{Receive}(m)_{i,j}$ send messages to the channels and receive messages from the channels. Action $\text{Receive}(m)_i$ handles the receipt of “Ack” and “Success” messages. Action RndSuccess_i simply takes care of updating the $\mathit{Decision}_i$ variable and sets a time bound for the execution of action SendSuccess_i. Action SendSuccess_i sends the “Success” message, along with the value of $\mathit{Decision}_i$ to all processes for which there is no acknowledgment. It sets the time bounds for the re-sending of the “Success” message and also the time bounds $\mathit{LastSendSuc}(j)_i$ for the actual sending of the messages. Action Check_i re-enable action SendSuccess_i after an appropriate time bound. We remark that $2\ell + 2d$ is the time needed to send the “Success” message and get back an “Ack” message (see the analysis in the proof of Lemma 11).

We remark that automaton $\textsc{bpsuccess}_i$ needs to be able to measure the passage of time.

6.2.3 Partial Correctness

Let us define the system S_{BPX} to be the composition of system S_{CHA} and automaton $\textsc{basicpaxos}_i$ for each process $i \in \mathcal{I}$ (remember that $\textsc{basicpaxos}_i$ is the composition of automata $\textsc{bpleader}_i$, $\textsc{bpagent}_i$ and $\textsc{bpsuccess}_i$). In this section we prove the partial correctness of S_{BPX}: we show that in any execution of the system S_{BPX}, agreement and validity are guaranteed.

For these proofs, we augment the algorithm with a collection $\mathcal{H}$ of history variables. Each variable in $\mathcal{H}$ is an array indexed by the round number. For every round number r a history variable contains some information about round r. In particular the set $\mathcal{H}$ consists of:

$\texttt{Hleader}(r) \in \mathcal{I} \cup \texttt{nil}$, initially $\texttt{nil}$ (the leader of round r).
$\texttt{Hvalue}(r) \in V \cup \texttt{nil}$, initially $\texttt{nil}$ (the value for round r).
$\texttt{Hfrom}(r) \in \mathcal{R} \cup \texttt{nil}$, initially $\texttt{nil}$ (the round from which $\texttt{Hvalue}(r)$ is taken).
$\texttt{Hinfquo}(r)$, subset of $\mathcal{I}$, initially $\{\}$ (the info-quorum of round r).
$\texttt{Haccquo}(r)$, subset of $\mathcal{I}$, initially $\{\}$ (the accepting-quorum of round r).
$\texttt{Hreject}(r)$, subset of $\mathcal{I}$, initially $\{\}$ (processes committed to reject round r).

The code fragments of automata BPLEADER$_i$ and BPAGENT$_i$ augmented with the history variables are shown in Figures 14 and 15 . The figures show only the actions that change history variables. Actions of BPSUCCESS$_i$ do not change history variables.

Initially, when no round has been started yet, all the information contained in the history variables is set to the initial values. All but $\texttt{Hreject}(r)$ history variables of round r are set by the leader of round r, thus if the round has not been started these variables remain at their initial values. More formally we have the following lemma.

Lemma 8 *In any state of an execution of* S_{BPX}*, if* $\texttt{Hleader}(r) = \texttt{nil}$ *then* $\texttt{Hvalue}(r) = \texttt{nil}$*,* $\texttt{Hfrom}(r) = \texttt{nil}$*,* $\texttt{Hinfquo}(r) = \{\}$*,* $\texttt{Haccquo}(r) = \{\}$*.*

Proof: By an easy induction. □

Given a round r, $\texttt{Hreject}(r)$, is modified by all the processes that commit themselves to reject round r, and we know nothing about its value at the time round r is started.

Next we define some key concepts that will be instrumental in the proofs.

Definition 6.1 *In any state of the system* S_{BPX}*, a round* r *is said to be dead if* $|\texttt{Hreject}(r)| \geq n/2$*.*

That is, a round r is dead if at least $n/2$ of the processes are rejecting it. Hence, if a round r is dead, there cannot be a majority of the processes accepting its value, i.e., round r cannot be successful.

We denote by $\mathcal{R}_S$ the set $\{r \in \mathcal{R} | \texttt{Hleader}(r) \neq \texttt{nil}\}$ of *started* rounds and by $\mathcal{R}_V$ the set $\{r \in \mathcal{R} | \texttt{Hvalue}(r) \neq \texttt{nil}\}$ of rounds for which the value has

BPLEADER$_i$ (history variables)

```
input NewRound_i
  Eff: if $LeaderAlive = \texttt{true}$ then
         $CurRnd := HighestRnd + 1$
         • $\texttt{Hleader}(CurRnd) := i$
         $HighestRnd := CurRnd$
         $Mode := \texttt{collect}$

output BeginCast_i
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{begincast}$
  Eff: $\forall j$ put $\langle CurRnd,$"Begin"$, Value\rangle_{i,j}$
         in $OutMsgs$
       • $\texttt{Hinfquo}(CurRnd) := InfoQuo$
       • $\texttt{Hfrom}(CurRnd) := ValFrom$
       • $\texttt{Hvalue}(CurRnd) := Value$
       $Mode := \texttt{gatheraccept}$

internal GatherAccept(m)_i
  Pre: $LeaderAlive = \texttt{true}$
       $Mode = \texttt{gatheraccept}$
       $m = \langle r,$"Accept"$\rangle_{j,i} \in InMsgs$
       $CurRnd = r$
  Eff: remove $m$ from $InMsgs$
       $AcceptQuo := AcceptQuo \cup \{j\}$
       if $|AcceptQuo| > n/2$ then
         $Decision := Value$
         • $\texttt{Haccquo}(CurRnd) := AcceptQuo$
         $Mode := \texttt{decide}$
```

Fig. 14. Actions of BPLEADER$_i$ for process i augmented with history variables. Only the actions that do change history variables are shown. Other actions are the same as in BPLEADER$_i$, i.e. they do not change history variables. Actions of BPSUCCESS$_i$ do not change history variables.

been chosen. Clearly In any state s of an execution of S_{BPX}, we have that $\mathcal{R}_V \subseteq \mathcal{R}_S$.

Next we formally define the concept of *anchored* round which is crucial to the proofs. The idea of anchored round is borrowed from [20]. Informally a round r is anchored if its value is consistent with the value chosen in any previous round r'. Consistent means that either the value of round r is equal to the value of round r' or round r' is dead. Intuitively, it is clear that if all the rounds are either anchored or dead, then agreement is satisfied.

Definition 6.2 *A round $r \in \mathcal{R}_V$ is said to be anchored if for every round $r' \in \mathcal{R}_V$ such that $r' < r$, either round r' is dead or* $\texttt{Hvalue}(r') = \texttt{Hvalue}(r)$.

BPAGENT$_i$ (history variables)

```
internal LastAccept$(m)_i$
  Pre: $Status = \mathtt{alive}$
       $m = \langle r, \text{“Collect”}\rangle_{j,i} \in InMsgs$
  Eff: remove $m$ from $InMsgs$
       if $r \geq Commit$ then
         $Commit := r$
         • For all $r'$, $LastR < r' < r$
         •   $\mathtt{Hreject}(r') := \mathtt{Hreject}(r') \cup \{i\}$
         put $\langle r, \text{“Last”}, LastR, LastV\rangle_{i,j}$
           in $OutMsgs$
       else
         put $\langle r, \text{“OldRound”}, Commit\rangle_{i,j}$
           in $OutMsgs$
```

Fig. 15. Actions of BPAGENT$_i$ for process i augmented with history variables. Only the actions that do change history variables are shown. Other actions are the same as in BPAGENT$_i$, i.e. they do not change history variables. Actions of BPSUCCESS$_i$ do not change history variables.

Next we prove that S_{BPX} guarantees agreement, by using a sequence of invariants. The key invariant is Invariant 6.8 which states that all rounds are either dead or anchored. The first invariant, Invariant 6.3, captures the fact that when a process sends a “Last” message in response to a “Collect” message for a round r, then it commits to not vote for rounds previous to round r.

Invariant 6.3 *In any state s of an execution of S_{BPX}, if message $\langle r, \text{“Last”}, r'', v\rangle_{j,i}$ is in $OutMsgs_j$, then $j \in \mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no message is in $OutMsgs_j$. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

CASE 1. $\langle r, \text{“Last”}, r'', v\rangle_{j,i} \in s_k.OutMsgs_j$. By the inductive hypothesis we have $j \in s_k.\mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$. Since no process is ever removed from any $\mathtt{Hreject}$ set, we have $j \in s.\mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$.

CASE 2. $\langle r, \text{“Last”}, r'', v\rangle_{j,i} \notin s_k.OutMsgs_j$. Since by hypothesis we have $\langle r, \text{“Last”}, r'', v\rangle_{j,i} \in s.OutMsgs_j$, it must be that $\pi = \text{LastAccept}(m)_j$, with $m = \langle r, \text{“Collect”}\rangle$ and it must be $s_k.LastR_j = r''$. Then the invariant follows by the code of LastAccept$(m)_j$ which puts process j into $\mathtt{Hreject}(r')$ for all r' such that $r'' < r' < r$. □

The next invariant states that the commitment made by an agent when sending a "Last" message is still in effect when the message is in the communication channel. This should be obvious, but to be precise in the rest of the proof we prove it formally.

Invariant 6.4 *In any state s of an execution of S_{BPX}, if message $\langle r$, "Last",r'',$v\rangle_{j,i}$ is in* $\textsc{channel}_{j,i}$*, then $j \in \mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no messages are in $\textsc{channel}_{j,i}$. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1s_1...\pi_ks_k$ and consider the execution $s_0\pi_1s_1...\pi_ks_k\pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

Case 1. $\langle r$,"Last",r'',$v\rangle_{j,i} \in s_k.\textsc{channel}_{j,i}$. By the inductive hypothesis we have $j \in s_k.\mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$. Since no process is ever removed from any Hreject set, we have $j \in s.\mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$.

Case 2. $\langle r$,"Last",r'', $v\rangle_{j,i} \notin s_k.\textsc{channel}_{j,i}$. Since by hypothesis $\langle r$,"Last",r'', $v\rangle_{j,i}$ $\in s.\mathit{OutMsgs}_j$, it must be that $\pi = \mathrm{Send}(m)_{j,i}$ with $m =\langle r$,"Last",r'', $v\rangle_{j,i}$. By the precondition of action $\mathrm{Send}(m)_{j,i}$ we have that message $\langle r$,"Last",r'', $v\rangle_{j,i} \in$ $s_k.\mathit{OutMsgs}_j$. By Invariant 6.3 we have that process $j \in s_k.\mathtt{Hreject}(r')$ for all r' such that $r'' < r' < r$. Since no process is ever removed from any Hreject set, we have $j \in s.\mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$. □

The next invariant states that the commitment made by an agent when sending a "Last" message is still in effect when the message is received by the leader. Again, this should be obvious.

Invariant 6.5 *In any state s of an execution of S_{BPX}, if message $\langle r$, "Last",r'',$v\rangle_{j,i}$ is in InMsgs_i, then $j \in \mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no messages are in InMsgs_i. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1s_1...\pi_ks_k$ and consider the execution $s_0\pi_1s_1...\pi_ks_k\pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

Case 1. $\langle r$,"Last",r'',$v\rangle_{j,i} \in s_k.\mathit{InMsgs}_i$. By the inductive hypothesis we have $j \in s_k.\mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$. Since no process is ever removed from any Hreject set, we have $j \in s.\mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$.

CASE 2. $\langle r,\text{"Last"},r'',v\rangle_{j,i} \notin s_k.\mathit{InMsgs}_i$. Since by hypothesis $\langle r,\text{"Last"},r'',v\rangle_{j,i} \in s.\mathit{InMsgs}_i$, it must be that $\pi = \text{Receive}(m)_{i,j}$ with $m = \langle r,\text{"Last"},r'',v\rangle_{j,i}$. In order to execute action $\text{Receive}(m)_{i,j}$ we must have $\langle r,\text{"Last"},r'',v\rangle_{j,i} \in s_k.\text{CHANNEL}_{j,i}$. By Invariant 6.4 we have $j \in s_k.\texttt{Hreject}(r')$ for all r' such that $r'' < r' < r$. Since no process is ever removed from any `Hreject` set, we have $j \in s.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$. □

The following invariant states that the commitment to reject smaller rounds, made by the agent is still in effect when the leader updates its information about previous rounds using the agents' "Last" messages.

Invariant 6.6 *In any state s of an execution S_{BPX}, if process $j \in \mathit{InfoQuo}_i$, for some process i, and $\mathit{CurRnd}_i = r$, then $\forall r'$ such that $s.\mathit{ValFrom}_i < r' < r$, we have that $j \in \texttt{Hreject}(r')$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no process j is in $\mathit{InfoQuo}_i$ for any i. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

CASE 1. In state s_k, $j \in \mathit{InfoQuo}_i$, for some process i, and $\mathit{CurRnd}_i = r$. Then by the inductive hypothesis, in state s_k we have that $j \in \texttt{Hreject}(r')$, for all r' such that $s_k.\mathit{ValFrom}_i < r' < r$. Since no process is ever removed from any `Hreject` set and, as long as CurRnd_i is not changed, variable $\mathit{ValFrom}_i$ is never decreased, then also in state s we have that $j \in \texttt{Hreject}(r')$, for all r' such that $s.\mathit{ValFrom}_i < r' < r$.

CASE 2. In state s_k, it is not true that $j \in \mathit{InfoQuo}_i$, for some process i, and $\mathit{CurRnd}_i = r$. Since in state s it holds that $j \in \mathit{InfoQuo}_i$, for some process i, and $\mathit{CurRnd}_i = r$, it must be the case that $\pi = \text{GatherLast}(m)_i$ with $m = \langle r,\text{"Last"},r'',v\rangle_{j,i}$. Notice that, by the precondition of $\text{GatherLast}(m)_i$, $m \in \mathit{InMsgs}_i$. Hence, by Invariant 6.5 we have that $j \in \texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$. By the code of the $\text{GatherLast}(m)_i$ action we have that $\mathit{ValFrom}_i \geq r''$. Whence the invariant is proved. □

The following invariant is basically the previous one stated when the leader has fixed the info-quorum.

Invariant 6.7 *In any state of an execution of S_{BPX}, if $j \in \texttt{Hinfquo}(r)$ then $\forall r'$ such that $\texttt{Hfrom}(r) < r' < r$, we have that $j \in \texttt{Hreject}(r')$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state we have that for every round r, $\texttt{Hleader}(r) = \texttt{nil}$ and thus by Lemma 8 there is no process j in $\texttt{Hinfquo}(r)$. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

Case 1. In state s_k, $j \in \texttt{Hinfquo}(r)$. By the inductive hypothesis, in state s_k we have that $j \in \texttt{Hreject}(r')$, for all r' such that $\texttt{Hfrom}(r) < r' < r$. Since no process is ever removed from any $\texttt{Hreject}$ set, then also in state s we have that $j \in \texttt{Hreject}(r')$, for all r' such that $\texttt{Hfrom}(r) < r' < r$.

Case 2. In state s_k, $j \notin \texttt{Hinfquo}(r)$. Since in state s, $j \in \texttt{Hinfquo}(r)$, it must be the case that action π puts j in $\texttt{Hinfquo}(r)$. Thus it must be $\pi =$ BeginCast$_i$ for some process i, and it must be $s_k.\mathit{CurRnd}_i = r$ and $j \in s_k.\mathit{InfoQuo}_i$. Since action BeginCast$_i$ does not change CurRnd_i and $\mathit{InfoQuo}_i$ we have that $s.\mathit{CurRnd}_i = r$ and $j \in s.\mathit{InfoQuo}_i$. By Invariant 6.6 we have that $j \in \texttt{Hreject}(r')$ for all r' such that $s.\mathit{ValFrom}_i < r' < r$. By the code of BeginCast$_i$ we have that $\texttt{Hfrom}(r) = s.\mathit{ValFrom}_i$. ∎

We are now ready to prove the main invariant.

Invariant 6.8 *In any state of an execution of $S_{\rm BPX}$, any non-dead round $r \in \mathcal{R}_V$ is anchored.*

Proof: We proceed by induction on the length k of the execution α. The base is trivial. When $k = 0$ we have that $\alpha = s_0$ and in the initial state no round has been started yet. Thus $\texttt{Hleader}(r) = \texttt{nil}$ and by Lemma 8 we have that $\mathcal{R}_V = \{\}$ and thus the assertion is vacuously true. For the inductive step assume that the assertion is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that, for every possible action π the assertion is still true in state s. First we observe that the definition of “dead” round depends only upon the history variables and that the definition of “anchored” round depends upon the history variables and the definition of “dead” round. Thus the definition of “anchored” depends only on the history variables. Hence actions that do not modify the history variables cannot affect the truth of the assertion. The actions that change history variables are:

(1) $\pi =$ NewRound$_i$
(2) $\pi =$ BeginCast$_i$
(3) $\pi =$ GatherAccept$(m)_i$
(4) $\pi =$ LastAccept$(m)_i$

Case 1. Assume $\pi =$ NewRound$_i$. This action sets the history variable $\texttt{Hleader}(r)$, where r is the round number of the round being started by process i. The new round r does not belong to $\mathcal{R}_V$ since $\texttt{Hvalue}(r)$ is still undefined.

Thus the assertion of the lemma cannot be contradicted by this action.

CASE 2. Assume π = BeginCast$_i$. Action π sets $\texttt{Hvalue}(r)$, $\texttt{Hfrom}(r)$ and $\texttt{Hinfquo}(r)$ where $r = s_k.CurRnd_i$. Round r belongs to $\mathcal{R}_V$ in the new state s. In order to prove that the assertion is still true it suffices to prove that round r is anchored in state s and any round r', $r' > r$ is still anchored in state s. Indeed rounds with round number less than r are still anchored in state s, since the definition of anchored for a given round involves only rounds with smaller round numbers.
First we prove that round r is anchored. From the precondition of BeginCast$_i$ we have that $\texttt{Hinfquo}(r)$ contains more than $n/2$ processes; indeed variable $Mode_i$ is equal to $\texttt{begincast}$ only if the cardinality of $InfoQuo_i$ is greater than $n/2$. Using Invariant 6.7 for each process j in $s.\texttt{Hinfquo}(r)$, we have that for every round r', such that $s.\texttt{Hfrom}(r) < r' < r$, there are more than $n/2$ processes in the set $\texttt{Hreject}(r')$, which means that every round r', $s.\texttt{Hfrom}(r) < r' < r$, is dead. Moreover, by the code of π we have that $s.\texttt{Hfrom}(r) = s_k.ValFrom_i$ and $s.\texttt{Hvalue}(r) = s_k.Value_i$. From the code (see action GatherLast$_i$) it is immediate that in any state $Value_i$ is the value of round $ValFrom_i$. In particular we have that $s_k.Value_i = s_k.\texttt{Hvalue}(s_k.ValFrom_i)$. Hence we have $s.\texttt{Hvalue}(r) = s.\texttt{Hvalue}(s.\texttt{Hfrom}(r))$. Finally we notice that round $\texttt{Hfrom}(r)$ is anchored (any round previous to r is still anchored in state s) and thus we have that any round $r' < r$ is either dead or such that $s.\texttt{Hvalue}(s.\texttt{Hfrom}(r)) = s.\texttt{Hvalue}(r')$. Hence for any round $r' < r$ we have that either round r' is dead or that $s.\texttt{Hvalue}(r) = s.\texttt{Hvalue}(r')$. Thus round r is anchored in state s.

Finally, we need to prove that any non-dead round r', $r' > r$ that was anchored in s_k is still anchored in s. Since action BeginCast$_i$ modifies only history variables for round r, we only need to prove that in state s, $\texttt{Hvalue}(r') = \texttt{Hvalue}(r)$. Let r'' be equal to $\texttt{Hfrom}(r)$. Since r' is anchored in state s_k we have that $s_k.\texttt{Hvalue}(r') = s_k.\texttt{Hvalue}(r'')$. Again because BeginCast$_i$ modifies only history variables for round r, we have that $s.\texttt{Hvalue}(r') = s.\texttt{Hvalue}(r'')$. But we have proved that round r is anchored in state s and thus $s.\texttt{Hvalue}(r) = s.\texttt{Hvalue}(r'')$. Hence $s.\texttt{Hvalue}(r') = s.\texttt{Hvalue}(r)$.

CASE 3. Assume π = GatherAccept$(m)_i$. This action modifies only variable $\texttt{Haccquo}$, which is not involved in the definition of anchored. Thus this action cannot make the assertion false.

CASE 4. Assume π = LastAccept$(m)_i$. This action modifies $\texttt{Hinfquo}$ and $\texttt{Hreject}$. Variable $\texttt{Hinfquo}$ is not involved in the definition of anchored. Action LastAccept$(m)_i$ may put process i in $\texttt{Hreject}$ of some rounds and this, in turn, may make those rounds dead. However this cannot make false the assertion; indeed if a round r was anchored in s_k it is still anchored when another round becomes dead. $\square$

The next invariant follows from the previous one and gives a more direct statement about the agreement property.

Invariant 6.9 *In any state of an execution of S_{BPX}, all the Decision variables that are not* `nil`*, are set to the same value.*

Proof: We prove the invariant by induction on the length k of the execution α. The base of the induction is trivially true: for $k = 0$ we have that $\alpha = s_0$ and in the initial state all the $\mathit{Decision}_i$ variables are undefined.
Assume that the assertion is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that, for every possible action π the assertion is still true in state s. Clearly the only actions which can make the assertion false are those that set $\mathit{Decision}_i$, for some process i. Thus we only need to consider actions GatherAccept($\langle r$, "Accept"$\rangle)_i$ and actions RndSuccess$(v)_i$ and Receive($\langle$"Success"$, v\rangle)_{i,j}$ of automaton BPSUCCESS$_i$.

Case 1. Assume $\pi =$ GatherAccept($\langle r$, "Accept"$\rangle)_i$. This action sets $\mathit{Decision}_i$ to $\texttt{Hvalue}(r)$. If all $\mathit{Decision}_j$, $j \neq i$, are undefined then $\mathit{Decision}_i$ is the first decision and the assertion is still true. Assume there is only one $\mathit{Decision}_j$ already defined. Let $\mathit{Decision}_j = \texttt{Hvalue}(r')$ for some round r'. By Invariant 6.8, rounds r and r' are anchored and thus we have that $\texttt{Hvalue}(r') = \texttt{Hvalue}(r)$. Whence $\mathit{Decision}_i = \mathit{Decision}_j$. If there are some $\mathit{Decision}_j$, $j \neq i$, which are already defined, then by the inductive hypothesis they are all equal. Thus, the lemma follows.

Case 2. Assume $\pi =$ RndSuccess$(v)_i$. This action sets $\mathit{Decision}_i$ to v. By the code, value v is equal to the $\mathit{Decision}_j$ of some other process. The lemma follows by the inductive hypothesis.

Case 3. Assume $\pi =$ Receive($\langle$"Success",$v\rangle)_i$. This action sets $\mathit{Decision}_i$ to v. It is easy to see (by the code) that the value sent in a "Success" message is always the *Decision* of some process. Thus we have that $\mathit{Decision}_i$ is equal to $\mathit{Decision}_j$ for some other process j and the lemma follows by the inductive hypothesis. ▯

Finally we can prove that agreement is satisfied.

Theorem 9 *In any execution of the system S_{BPX}, agreement is satisfied.*

Proof: Immediate from Invariant 6.9. ▯

Validity is easier to prove since the value proposed in any round comes either from a value supplied by an Init$(v)_i$ action or from a previous round.

Invariant 6.10 *In any state of an execution α of S_{BPX}, for any $r \in \mathcal{R}_V$ we* *have that* $\texttt{Hvalue}(r) \in V_\alpha$.

Proof: We proceed by induction on the length k of the execution α. The base of the induction is trivially true: for $k = 0$ we have that $\alpha = s_0$ and in the initial state all the `Hvalue` variables are undefined.
Assume that the assertion is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that, for every possible action π the assertion is still true in state s. Clearly the only actions that can make the assertion false are those that modify `Hvalue`. The only action that modifies `Hvalue` is BeginCast. Thus, assume $\pi = \text{BeginCast}_i$. This action sets $\texttt{Hvalue}(r)$ to Value_i. We need to prove that all the values assigned to Value_i are in the set V_α. Variable Value_i is modified by actions NewRound_i and $\text{GatherLast}(m)_i$. We can easily take care of action NewRound_i because it simply sets Value_i to be $\mathit{InitValue}_i$ which is obviously in V_α. Thus we only need to worry about $\text{GatherLast}(m)_i$ actions. A $\text{GatherLast}(m)_i$ action sets variable Value_i to the value specified into the "Last" message if that value is not `nil`. The value specified into any "Last" message is either `nil` or the value $\texttt{Hvalue}(r')$ of a previous round r'; by the inductive hypothesis we have that $\texttt{Hvalue}(r')$ belongs to V_α. □

Invariant 6.11 *In any state of an execution of* S_{BPX}*, all the Decision variables that are not undefined are set to some value in* V_α.

Proof: A variable *Decision* is always set to be equal to $\texttt{Hvalue}(r)$ for some r. Thus the invariant follows from Invariant 6.10. □

Theorem 10 *In any execution of the system* S_{BPX}*, validity is satisfied.*

Proof: Immediate from Invariant 6.11. □

6.2.4 Analysis of S_{BPX}

In this section we analyze the performance of S_{BPX}. Since termination is not guaranteed by S_{BPX} in this section we provide a performance analysis (Lemma 14) assuming that a successful round is conducted. Then in Section 6.4, Theorem 17 provides the performance analysis of S_{PAX}, which, in a nice execution fragment, guarantees termination.

Let us begin by making precise the meaning of the expressions "the start (end) of a round".

Definition 6.12 *In an execution fragment whose states are all unique-leader states with process i being the unique leader, the start of a round is the execution of action* $\mathit{NewRound}_i$ *and the end of a round is the execution of action*

RndSuccess$_i$.

A round is *successful* if it ends, that is, if the RndSuccess$_i$ action is executed by the leader i. Moreover we say that a process i *reaches* its decision when automaton BPSUCCESS$_i$ sets its *Decision*$_i$ variable. We remark that, in the case of a leader, the decision is actually reached when the leader knows that a majority of the processes have accepted the value being proposed. This happens in action GatherAccept$(m)_i$ of BPLEADER$_i$. However, to be precise in our proofs, we consider the decision reached when the variable *Decision*$_i$ of BPSUCCESS$_i$ is set; for the leader this happens exactly at the end of a successful round. Notice that the Decide$(v)_i$ action, which communicates the decision v of process i to the external environment, is executed within ℓ time from the point in time when process i reaches the decision, provided that the execution is regular (in a regular execution actions are executed within the expected time bounds).

The following lemma states that once a round has ended, if the execution is stable, the decision is reached by all the alive processes within linear (in the number of processes) time.

Lemma 11 *If an execution fragment α of the system S_{BPX}, starting in a reachable state s and lasting for more than $3\ell + 2d$ time, is stable and unique-leader, with process i leader, and process i reaches a decision in state s, then by time $3\ell + 2d$, every alive process $j \neq i$ has reached a decision, and the leader i has Acked$(j)_i =$* `true` *for every alive process $j \neq i$.*

Proof: First notice that S_{BPX} is the composition of CHANNEL$_{i,j}$ and other automata. Hence, by Theorem 3 we can apply Lemma 4. Let $\mathcal{J}$ be the alive processes $j \neq i$ such that *Acked*$(j)_i =$ `false`. If $\mathcal{J}$ is empty then the lemma is trivially true. Hence assume $\mathcal{J} \neq \{\}$.

By assumption, the action that brings the system into state s is action RndSuccess$_i$ (the leader reaches a decision in state s). Hence action SendSuccess$_i$ is enabled. By the code of BPSUCCESS$_i$, action SendSuccess$_i$ is executed within ℓ time. This action puts a "Success" message for each process $j \in \mathcal{J}$ into *OutSucMsgs*$(j)_i$. By the code of BPSUCCESS$_i$, each of these messages is put on CHANNEL$_{i,j}$, i.e., action Send$(\langle$"Success",$v\rangle)_{i,j}$ is executed, within ℓ time. By Lemma 4 each alive process $j \in \mathcal{J}$ receives the "Success" message, i.e., executes a Receive$(\langle$"Success",$v\rangle)_{i,j}$ action, within d time. This action sets *Decision*$_j$ to v and puts an "Ack" message into *OutAckMsgs*$(i)_j$. By the code of BPSUCCESS$_j$, this "Ack" message is put on CHANNEL$_{j,i}$, i.e., action Send("Ack")$_{j,i}$ is executed, within ℓ time, for every process j. By Lemma 4 the leader i receives the "Ack" message, i.e., executes a Receive$(\langle$"Ack"$\rangle)_{j,i}$ action, within d time, for each process j. This action sets *Acked*$(j)_i =$ `true`.

Summing up the time bounds we get the lemma. □

In the following we are interested in the time analysis from the start to the end of a successful round. We consider a unique-leader execution fragment α, with process i leader, and such that the leader i has started a round by the first state of α (that is, in the first state of α, $\mathit{CurRnd}_i = r$ for some round number r).

We remark that in order for the leader to execute step 3 of a round, i.e., action $\mathrm{BeginCast}_i$, it is necessary that Value_i be defined. If the leader does not have an initial value and no agent sends a value in a "Last" message, variable Value_i is not defined. In this case the leader needs to wait for the execution of the $\mathrm{Init}(v)_i$ to set a value to propose in the round (see action $\mathrm{Continue}_i$). Clearly the time analysis depends on the time of occurrence of the $\mathrm{Init}(v)_i$. To deal with this we use the following definition.

Definition 6.13 *Given an execution fragment α, we define t^i_α to be 0, if variable $\mathit{InitValue}_i$ is defined in the first state of α; the time of occurrence of action $\mathit{Init}(v)_i$, if variable $\mathit{InitValue}_i$ is undefined in the first state of α and action $\mathit{Init}(v)_i$ is executed in α; ∞, if variable $\mathit{InitValue}_i$ is undefined in the first state of α and no $\mathit{Init}(v)_i$ action is executed in α. Moreover, we define T^i_α to be* $\max\{7\ell + 2d, t^i_\alpha + 2\ell\}$.

Informally, the above definition of T^i_α gives the time, counted from the beginning of a round, by which a $\mathrm{BeginCast}_i$ action is expected to be executed, assuming that the execution α is stable and the round being conducted is successful. More formally we have the following lemma.

Lemma 12 *Suppose that for an execution fragment α of the system S_{BPX}, starting in a reachable state s in which $s.\mathit{Decision} = \texttt{nil}$, it holds that:*

(i) *α is stable;*
(ii) *α is a unique-leader execution, with process i leader;*
(iii) *α lasts for more than T^i_α;*
(iv) *the action that brings the system into state s is action $\mathit{NewRound}_i$ for some round r;*
(v) *round r is successful.*

Then we have that action $\mathit{BeginCast}_i$ for round r is executed within time T^i_α of the beginning of α.

Proof: First notice that S_{BPX} is the composition of $\mathrm{CHANNEL}_{i,j}$ and other automata. Hence, by Theorem 3 we can apply Lemmas 1 and 4. Since the execution is stable, it is also regular, and thus by Lemma 1 actions of $\mathrm{BPLEADER}_i$ and $\mathrm{BPAGENT}_i$ are executed within ℓ time and by Lemma 4 messages are delivered within d time.

Action $\mathrm{NewRound}_i$ enables action $\mathrm{Collect}_i$ which is executed in at most ℓ time.

This action puts "Collect" messages, one for each agent j, into $OutMsgs_i$. By the code of BPLEADER$_i$ (see tasks and bounds) each one of these messages is sent on CHANNEL$_{i,j}$ i.e., action Send$_{i,j}$ is executed for each of these messages, within ℓ time. By Lemma 4 a "Collect" message is delivered to each agent j, i.e., action Receive$_{i,j}$ is executed, within d time. Then it takes ℓ time for an agent to execute action LastAccept$_j$ which puts a "Last" message in $OutMsgs_j$. By the code of BPAGENT$_i$ (see tasks and bounds) it takes additional ℓ time to execute action Send$_{j,i}$ to send the "Last" message on CHANNEL$_{j,i}$. By Lemma 4, this "Last" message is delivered to the leader, i.e., action Receive$_{j,i}$ is executed, within additional d time. By the code of BPLEADER$_i$ (see tasks and bounds) each one of these messages is processed by GatherLast$_i$ within ℓ time. Action Gathered$_i$ is executed within additional ℓ time.

At this point there are two possible cases: (i) $Value_i$ is defined and (ii) $Value_i$ is not defined. In case (i), action BeginCast$_i$ is enabled and is executed within ℓ time. Summing up the times considered so far we have that action BeginCast$_i$ is executed within $7\ell + 2d$ time from the start of the round. In case (ii), action Continue$_i$ is executed within ℓ time of the execution of action Continue$_i$, and thus by time $t^i_\alpha + \ell$. This action enables action BeginCast$_i$ which is executed within additional ℓ time. Hence action BeginCast$_i$ is executed by time $t^i_\alpha + 2\ell$. Putting together the two cases we have that action BeginCast$_i$ is executed by time $\max\{7\ell + 2d, t^i_\alpha + 2\ell\}$.

Hence we have proved that action BeginCast$_i$ is executed in α by time T^i_α. $\square$

Next lemma gives a bound for the time that elapses between the execution of the BeginCast$_i$ action and the RndSuccess$_i$ action for a successful round in a stable execution fragment.

Lemma 13 *Suppose that for an execution fragment α of the system S_{BPX}, starting in a reachable state s in which $s.Decision =$ `nil`, it holds that:*

(i) *α is stable;*
(ii) *α is a unique-leader execution, with process i leader;*
(iii) *α lasts for more than $5\ell + 2d$ time;*
(iv) *the action that brings the system into state s is action $BeginCast_i$ for some round r;*
(v) *round r is successful.*

Then we have that action $RndSuccess_i$ is performed by time $5\ell + 2d$ from the beginning of α.

Proof: First notice that S_{BPX} is the composition of CHANNEL$_{i,j}$ and other automata. Hence, by Theorem 3 we can apply Lemmas 1 and 4. Since the execution is stable, it is also regular, and thus by Lemma 1 actions of BPLEADER$_i$ and BPAGENT$_i$ are executed within ℓ time and by Lemma 4 messages are delivered within d time.

Action BeginCast$_i$ puts "Begin" messages for round r in $OutMsgs_i$. By the code of BPLEADER$_i$ (see tasks and bounds) each one of these messages is put on CHANNEL$_{i,j}$ by means of action Send$_{i,j}$ in at most ℓ time. By Lemma 4 a "Begin" message is delivered to each agent j, i.e., action Receive$_{i,j}$ is executed, within d time. By the code of BPAGENT$_j$ (see tasks and bounds) action Accept$_j$ is executed within ℓ time. This action puts an "Accept" message in $OutMsgs_j$. By the code of BPAGENT$_j$ the "Accept" message is put on CHANNEL$_{j,i}$, i.e., action Send$_{j,i}$ for this message is executed, within ℓ time. By Lemma 4 the message is delivered, i.e., action Receive$_{j,i}$ for that message is executed, within d time. By the code of BPLEADER$_i$ action GatherAccept$_i$ is executed for a majority of the "Accept" messages within additional ℓ time. At this point variable $Decision_i$ is defined and action RndSuccess$_i$ is executed within ℓ time. Summing up all the times we have that the round ends within $5\ell + 2d$. □

We can now easily prove a time bound on the time needed to complete a round.

Lemma 14 *Suppose that for an execution fragment α of the system S_{BPX}, starting in a reachable state s in which $s.Decision =$ `nil`, it holds that:*

- *(i) α is stable;*
- *(ii) α is a unique-leader execution, with process i leader;*
- *(iii) α lasts for more than $T^i_\alpha + 5\ell + 2d$;*
- *(iv) the action that brings the system into state s is action NewRound$_i$ for some round r;*
- *(v) round r is successful.*

Then we have that action BeginCast$_i$ for round r is executed within time T^i_α of the beginning of α and action RndSuccess$_i$ is executed by time $T^i_\alpha + 5\ell + 2d$ of the beginning of α.

Proof: Follows from Lemmas 12 and 13 □

The previous lemma states that in a stable execution a successful round is conducted within some time bound. However it is possible that even if the system executes nicely from some point in time on, no successful round is conducted and to have a successful round a new round must be started. We take care of this problem in the next section. We will use a more refined version of Lemma 14; this refined version replaces condition (v) with a weaker requirement. This weaker requirement is enough to prove that the round is successful.

Lemma 15 *Suppose that for an execution fragment α of $S_{\rm BPX}$, starting in a reachable state s in which $s.Decision = \texttt{nil}$, it holds that:*

(i) *α is nice;*
(ii) *α is a unique-leader execution, with process i leader;*
(iii) *α lasts for more than $T^i_\alpha + 5\ell + 2d$ time;*
(iv) *the action that brings the system into state s is action $NewRound_i$ for some round r;*
(v) *there exists a set $\mathcal{J} \subseteq \mathcal{I}$ of processes such that every process in $\mathcal{J}$ is alive and $\mathcal{J}$ is a majority, for every $j \in \mathcal{J}$, $s.Commit_j \leq r$ and in state s for every $j \in \mathcal{J}$ and $k \in \mathcal{I}$, $\text{CHANNEL}_{k,j}$ and $InMsgs_j$ do not contain any "Collect" message belonging to any round $r' > r$.*

Then we have that action $BeginCast_i$ is performed by time T^i_α and action $RndSuccess_i$ is performed by time $T^i_\alpha + 5\ell + 2d$ from the beginning of α.

Proof: Process i sends a "Collect" message which is delivered to all the alive voters. All the alive voters, and thus all the processes in $\mathcal{J}$, respond with "Last" messages which are delivered to the leader. No process $j \in \mathcal{J}$ can be committed to reject round r. Indeed, by assumption, process j is not committed to reject round r in state s and process j cannot commit to reject round r. The latter is due to the fact that in state s no message that can cause process j to commit to reject round r is either in $InMsgs_j$ nor in any channel to process j, and in α the only leader is i, which only sends messages belonging to round r. Since $\mathcal{J}$ is a majority, the leader receives at least a majority of "Last" messages and thus it is able to proceed with the next step of the round. The leader sends a "Begin" message which is delivered to all the alive voters. All the alive voters, and thus all the processes in $\mathcal{J}$, respond with "Accept" messages since they are not committed to reject round r. Since $\mathcal{J}$ is a majority, the leader receives at least a majority of "Accept" messages. Therefore given that α lasts for enough time round r is successful.

Since round r is successful, the lemma follows easily from Lemma 14. ∎

6.3 Automaton $S_{\rm PAX}$

To reach consensus using $S_{\rm BPX}$, rounds must be started by an external agent by means of the NewRound$_i$ action that makes process i start a new round. In this section we provide automata STARTER_i that start new round. Composing STARTER_i with $S_{\rm BPX}$ we obtain $S_{\rm PAX}$.

The system $S_{\rm BPX}$ guarantees that running rounds does not violate agreement and validity, even if rounds are started by many processes. However since running a new round may prevent a previous one from succeeding, initiating too many rounds is not a good idea. The strategy used to initiate rounds is to have a leader election algorithm and let the leader initiate new rounds until a round is successful. We exploit the robustness of BASICPAXOS in order to use the sloppy leader elector provided in Section 5. As long as the leader elector does not provide exactly one leader, it is possible that no round is successful, however agreement and validity are always guaranteed. This means that regardless of termination, in any run of the algorithm no two different decisions are ever made and any decision is equal to some input value. Moreover, when the leader elector provides exactly one leader, if the system S_{BPX} is executing a nice execution fragment then a round is successful.

Automaton $\textsc{starter}_i$ takes care of the problem of starting new rounds. This automaton interacts with $\textsc{leaderelector}_i$ by means of the Leader_i and $\mathrm{NotLeader}_i$ actions and with $\textsc{basicpaxos}_i$ by means of the $\mathrm{NewRound}_i$, $\mathrm{Gathered}(v)_i$, $\mathrm{Continue}_i$ and $\mathrm{RndSuccess}(v)_i$ actions. Figure 5, given at the beginning of the section, shows the interaction of the $\textsc{starter}_i$ automaton with the other automata.

The code of automaton $\textsc{starter}_i$ is shown in Figures 16 and 17. Automaton $\textsc{starter}_i$ does the following. Whenever process i becomes leader, the $\textsc{starter}_i$ automaton starts a new round by means of action $\mathrm{NewRound}_i$. Moreover the automaton checks that action $\mathrm{BeginCast}_i$ is executed within the expected time bound (given by Lemma 14). If $\mathrm{BeginCast}_i$ is not executed within the expected time bound, then $\textsc{starter}_i$ starts a new round. Similarly once $\mathrm{BeginCast}_i$ has been executed, the automaton checks that action $\mathrm{RndSuccess}(v)_i$ is executed within the expected time bound (given by Lemma 14). Again, if such an action is not executed within the expected time bound, $\textsc{starter}_i$ starts a new round. We remark that to check for the execution of $\mathrm{BeginCast}_i$, the automaton actually checks for the execution of action $\mathrm{Gathered}(v)_i$. This is because the expected time of execution of $\mathrm{BeginCast}_i$ depends on whether an initial value is already available when action $\mathrm{Gathered}(v)_i$ is executed. If such a value is available when $\mathrm{Gathered}(v)_i$ is executed then $\mathrm{BeginCast}_i$ is enabled and is expected to be executed within ℓ time of the execution of $\mathrm{Gathered}(v)_i$. Otherwise the leader has to wait for the execution of action $\mathrm{Init}(v)_i$ which enables action $\mathrm{Continue}_i$ and action $\mathrm{BeginCast}_i$ is expected to be executed within ℓ time of the execution of $\mathrm{Continue}_i$.

In addition to the code we provide some comments about the state variables and the actions. Variables $\mathit{IamLeader}_i$ and Status_i are self-explanatory. Variable Start_i is true when a new round needs to be started. Variable $\mathit{RndSuccess}_i$ is true when a decision has been reached. Variables $\mathit{DlineGat}_i$ and $\mathit{DlineSuc}_i$ are used to check for the execution of actions $\mathrm{Gathered}(v)_i$ and $\mathrm{RndSuccess}(v)_i$. They are also used, together with variable LastNR_i, to impose time bounds on enabled actions.

```
STARTER$_i$
Signature:
  Input:    Leader$_i$, NotLeader$_i$, Stop$_i$, Recover$_i$,
            Gathered$(v)_i$, Continue$_i$, RndSuccess$(v)_i$
  Internal: CheckGathered$_i$, CheckRndSuccess$_i$
  Output:   NewRound$_i$
  Time-passage: $\nu(t)$

State:
$Clock \in \mathbb{R}$                        init. arbitrary
$Status \in \{$alive,stopped$\}$             init. alive
$IamLeader$, a boolean                    init. false
$Start$, a boolean                        init. false
$DlineSuc \in \mathbb{R} \cup \{\infty\}$              init. nil
$DlineGat \in \mathbb{R} \cup \{\infty\}$              init. nil
$LastNR \in \mathbb{R} \cup \{\infty\}$                init. $\infty$
$RndSuccess$, a boolean                   init. false

Actions:
input Stop$_i$
  Eff: $Status :=$ stopped

input Recover$_i$
  Eff: $Status :=$ alive

input Leader$_i$
  Eff: if $Status =$ alive then
         if $IamLeader =$ false then
           $IamLeader :=$ true
           if $RndSuccess =$ false then
             $Start :=$ true
             $DlineGat := \infty$
             $DlineSuc := \infty$
             $LastNR := Clock + \ell$

input NotLeader$_i$
  Eff: if $Status =$ alive then
         $LastNR := \infty$
         $DlineSus := \infty$
         $DlineGat := \infty$
         $IamLeader :=$ false

output NewRound$_i$
  Pre: $Status =$ alive
       $IamLeader =$ true
       $Start =$ true
  Eff: $Start :=$ false
       $DlineGat := Clock + 6\ell + 2d$
       $LastNR := \infty$

input Gathered$(v)_i$
  Eff: if $Status =$ alive then
         $DlineGat := \infty$
         if $v \neq$ nil then
           $DlineSuc := Clock + 6\ell + 2d$

input Continue$_i$
  Eff: if $Status =$ alive then
         $DlineSuc := Clock + 6\ell + 2d$
```

Fig. 16. Automaton STARTER for process i (part 1)

Automaton STARTER$_i$ updates variable $IamLeader_i$ according to the input actions Leader$_i$ and NotLeader$_i$ and executes internal and output actions whenever it is the leader. Variable *Start* is used to start a new round and it is set either when a Leader$_i$ action changes the leader status *IamLeader* from false to true, that is, when the process becomes leader or when the expected time bounds for the execution of actions Gathered$(v)_i$ and RndSuccess$(v)_i$ elapse without the execution of these actions. Variable $RndSuccess_i$ is updated by the input action RndSuccess$(v)_i$. Action NewRound$_i$ starts a new round. Ac-

```
STARTER_i (cont'd)

input RndSuccess(v)_i
  Eff: if $Status = \texttt{alive}$ then
         $RndSuccess := \texttt{true}$
         $DlineGat := \infty$
         $DlineSuc := \infty$
         $LastNR := \infty$

internal CheckGathered_i
  Pre: $Status = \texttt{alive}$
       $IamLeader = \texttt{true}$
       $DlineGat \neq \texttt{nil}$
       $Clock > DlineGat$
  Eff: $DlineGat := \infty$
       $Start := \texttt{true}$
       $LastNR := Clock + \ell$

internal CheckRndSuccess_i
  Pre: $Status = \texttt{alive}$
       $IamLeader = \texttt{true}$
       $DlineSuc \neq \texttt{nil}$
       $Clock > DlineSuc$
  Eff: $DlineSuc := \infty$
       $Start := \texttt{true}$
       $LastNR := Clock + \ell$

time-passage $\nu(t)$
  Pre: none
  Eff: if $Status = \texttt{alive}$ then
         Let $t'$ be such that
           $Clock + t' \leq LastNR$
           and $Clock + t' \leq DlineGat + \ell$
           and $Clock + t' \leq DlineSuc + \ell$
       $Clock := Clock + t'$
```

Fig. 17. Automaton STARTER for process i (part 2)

tions CheckGathered$_i$ and CheckRndSuccess$_i$ check, respectively, whether actions Gathered$(v)_i$ and RndSuccess$(v)_i$ are executed within the expected time bounds. Using an analysis similar to the one done in the proof of Lemma 12 we have that action Gathered$(v)_i$ is supposed to be executed within $6\ell + 2d$ time of the start of the round. The time bound for the execution of action RndSuccess$(v)_i$ depends on whether the leader has to wait for an Init$(v)_i$ event. However by Lemma 13 action RndSuccess$(v)_i$ is expected to be executed within $5\ell + 2d$ time from the time of occurrence of action BeginCast$_i$ and action BeginCast$_i$ is executed either within ℓ time of the execution of action Gathered$(v)_i$, if an initial value is available when this action is executed, or else within ℓ time of the execution of action Continue$_i$. Hence actions Gathered$(v)_i$ and Continue$_i$ both set a deadline of $6\ell + 2d$ for the execution of action RndSuccess$(v)_i$. Actions CheckGathered$_i$ and CheckRndSuccess$_i$ start a new round if the above deadlines expire.

6.4 Correctness and analysis of S_{PAX}

Even in a nice execution fragment a round may not reach success. This is possible when agents are committed to reject the first round started in the nice execution fragment because they are committed for higher numbered rounds started before the beginning of the nice execution fragment. However in such a case a new round is started and there is nothing that can prevent the success of the new round. Indeed in the newly started round, alive processes are not committed for higher numbered rounds since during the first round they inform the leader of the round number for which they are committed and the leader, when starting a new round, always uses a round number greater than any round number ever seen. In this section we will prove that in a long enough nice execution fragment termination is guaranteed.

Remember that S_{PAX} is the system obtained by composing system S_{LEA} with one automaton BASICPAXOS$_i$ and one automaton STARTER$_i$ for each process $i \in \mathcal{I}$. Since this system contains as a subsystem the system S_{BPX}, it guarantees agreement and validity. However, in a long enough nice execution fragment of S_{PAX} termination is achieved, too.

The following lemma states that in a long enough nice, unique-leader execution, the leader reaches a decision. We recall that $T^i_\alpha = \max\{7\ell + 2d, t^i_\alpha + 2\ell\}$ and that t^i_α is the time of occurrence of action Init$(v)_i$ in α (see Definition 6.13).

Lemma 16 *Suppose that for an execution fragment α of S_{PAX}, starting in a reachable state s in which $s.Decision =$ `nil`, it holds that*

(i) α is nice;
(ii) α is a unique-leader execution, with process i leader;
(iii) α lasts for more than $T^i_\alpha + 20\ell + 7d$ time.

Then by time $T^i_\alpha + 20\ell + 7d$ the leader i has reached a decision.

Proof: First we notice that system S_{PAX} contains as subsystem S_{BPX}; hence by using Theorem 3, the projection of α on the subsystem S_{BPX} is actually an execution of S_{BPX} and thus Lemmas 14 and 15 are still true in α.

For simplicity, in the following we assume that $T^i_\alpha = 0$, i.e., that process i has executed an Init$(v)_i$ action before α. At the end of each case we consider, we will add T^i_α to the time bound to take into account the possibility that process i has to wait for an Init$(v)_i$ action. Notice that $T^i_\alpha = 0$ implies that $T^i_\beta = 0$ for any fragment β of α starting at some state of α and ending in the last state of α.

Let s' be the first state of α such that no "Collect" message sent by a process $k \neq i$ is present in CHANNEL$_{k,j}$ nor in *InMsgs*$_j$ for any j. State s' exists in α and its time of occurrence is less or equal to $d + \ell$. Indeed, since the execution is nice, all the messages that are in the channels in state s are delivered by time d and messages present in any *InMsgs* set are processed within ℓ time. Since i is the unique leader, in state s' no messages sent by a process $k \neq i$ is present in any channel nor in any *InMsgs* set. Let α' be the fragment of α beginning at s'. Since α' is a fragment of α, we have that α' is nice, process i is the unique leader in α' and $T^i_{\alpha'} = 0$.

If process i has started a round r' by state s' and round r' is successful, then round r' ends by time $T^i_{\alpha'} + 5\ell + 2d = 5\ell + 2d$ in α'. Indeed if the action that brings that system in state s' is a NewRound$_i$ action for round r' then by Lemma 14 we have that the round ends by time $T^i_{\alpha'} + 5\ell + 2d = 5\ell + 2d$. If action NewRound$_i$ for round r' has been executed before, round r' ends even more quickly and the time bound holds anyway. Since the time of occurrence of s' is less or equal to $\ell + d$ we have that round r' ends by time $6\ell + 3d$. Considering the possibility that process i has to wait for an Init$(v)_i$ action we have that round r' ends by time $T^i_\alpha + 6\ell + 3d$ in α. Hence the lemma is true in this case.

Assume that either (a) process i has started a round r' by state s' but round r is not successful or (b) that process i has not started any round by state s'. In both cases process i executes a NewRound$_i$ action by time $T^i_{\alpha'} + 7\ell + 2d = 7\ell + 2d$ in α'. Indeed in case (a), by the code of STARTER$_i$, action CheckRndSuccess$_i$ is executed within $T^i_{\alpha'} + 6\ell + 2d = 6\ell + 2d$ time and it takes additional ℓ time to execute action NewRound$_i$. In case (b), by the code of BPLEADER$_i$, action NewRound$_i$ is executed within ℓ time. Let r'' be the round started by such an action.

Let s'' be the state after the execution of the NewRound$_i$ action and let α'' be the fragment of α starting in s''. Since α'' is a fragment of α, we have that α'' is nice, process i is the unique leader in α'' and $T^i_{\alpha''} = 0$. We notice that since the time of occurrence of state s' is less or equal to $\ell + d$ the time of occurrence of s'' is less or equal to $8\ell + 3d$ in α.

We now distinguish two possible cases.

CASE 1. Round r'' is successful. In this case, by Lemma 14 we have that round r'' is successful within $T^i_{\alpha''} + 5\ell + 2d = 5\ell + 2d$ time in α''. Since the time of occurrence of s'' is less or equal to $8\ell + 3d$, we have that round r'' ends by time $13\ell + 5d$ in α. Considering the possibility that process i has to wait for an Init$(v)_i$ action we have that round r'' ends by time $T^i_\alpha + 13\ell + 5d$ in α. Hence the lemma is true in this case.

CASE 2. Round r'' is not successful.

By the code of STARTER$_i$, action NewRound$_i$ is executed within $T^i_{\alpha''} + 7\ell + 2d = 6\ell + 2d$ time in α''. Indeed, it takes $T^i_{\alpha''} + 5\ell + 2d$ to execute action CheckRndSuccess$_i$ and additional ℓ time to execute action NewRound$_i$. Let r''' be the new round started by i with such an action, let s''' be the state of the system after the execution of action NewRound$_i$ and let α''' be the fragment of α'' beginning at s'''. The time of occurrence of s''' is less or equal than $15\ell + 5d$ in α.

Clearly α''' is nice, process i is the unique leader in α'''. Any alive process j that rejected round r'' because of a round $\tilde{r}$, $\tilde{r} > r''$, has responded to the

"Collect" message of round r'', with a message $\langle r''$,"OldRound",$\tilde{r}\rangle_{j,i}$ informing the leader i about round $\tilde{r}$. Since α'' is nice all the "OldRound" messages are received before state s'''. Since action NewRound$_i$ uses a round number greater than all the ones received in "OldRound" messages, we have that for any alive process j, $s'''.Commit_j < r'''$. Let $\mathcal{J}$ be the set of alive processes. In state s''', for every $j \in \mathcal{J}$ and any $k \in \mathcal{I}$, $\text{CHANNEL}_{k,j}$ does not contain any "Collect" message belonging to any round $\tilde{r} > r'''$ nor such a message is present in any $InMsgs_j$ set (indeed this is true in state s'). Finally since α'' is nice, by definition of nice execution fragment, we have that $\mathcal{J}$ contains a majority of the processes.

Hence we can apply Lemma 15 to the execution fragment α'''. By Lemma 15, round r''' is successful within $T^i_{\alpha'''} + 5\ell + 2d = 5\ell + 2d$ time from the beginning of α'''. Since the time of occurrence of s''' is less or equal to $15\ell + 5d$ in α, we have that round r''' ends by time $20\ell + 7d$ in α. Considering the possibility that process i has to wait for an Init$(v)_i$ action we have that round r''' ends by time $T^i_\alpha + 20\ell + 7d$ in α. Hence the lemma is true also in this case. ▯

If the execution is stable for enough time, then the leader election eventually elects a unique leader (Lemma 7). In the following theorem we consider a nice execution fragment α and we let i be the process eventually elected unique leader. We recall that t^i_α is the time of occurrence of action Init$(v)_i$ in α and that ℓ and d are constants.

Theorem 17 *Let α be a nice execution fragment of S_{PAX} starting in a reachable state and lasting for more than $t^i_\alpha + 35\ell + 13d$. Then the leader i executes Decide$(v')_i$ by time $t^i_\alpha + 32\ell + 11d$ from the beginning of α. Moreover by time $t^i_\alpha + 35\ell + 13d$ from the beginning of α any alive process j executes Decide$(v')_j$.*

Proof: Since S_{PAX} contains S_{LEA} and S_{BPX} as subsystems, by Theorem 3 we can use any property of S_{LEA} and S_{BPX}. Since the execution fragment is nice (and thus stable), by Lemma 7 there is a unique leader by time $4\ell + 2d$. Let s' be the first unique-leader state of α and let i be the leader. By Lemma 7 the time of occurrence of s' before or at time $4\ell + 2d$. Let α' be the fragment of α starting in state s'. Since α is nice, α' is nice.

By Lemma 16 we have that the leader reaches a decision by time $T^i_{\alpha'} + 20\ell + 7d$ from the beginning of α'. Summing up the times and noticing that $T^i_{\alpha'} \leq t^i_{\alpha'} + 7\ell + 2d$ and that $t^i_{\alpha'} \leq t^i_\alpha$ we have that the leader reaches a decision by time $t^i_\alpha + 31\ell + 11d$. Within additional ℓ time action Decide$(v')_i$ is executed.

The leader reaches a decision by time $t^i_\alpha + 31\ell + 11d$. By Lemma 11 we have that a decision is reached by every alive process j within additional $3\ell + 2d$ time, that is by time $t^i_\alpha + 34\ell + 13d$. Within additional ℓ time action Decide$(v')_j$ is executed. ▯

6.5 Messages

It is not difficult to see that in a nice execution, which is an execution with no failures, the number of messages spent in a round is linear in the number of processes. Indeed in a successful round the leader broadcasts two messages and the agents respond to the leader's messages. Once the leader reached a decision another broadcast is enough to spread this decision to the agents. It is easy to see that, if everything goes well, at most $6n$ messages are sent to have all the alive processes reach the decision.

However failures may cause the sending of extra messages. It is not difficult to construct situations where the number of messages sent is quadratic in the number of processes. For example if we have that before i becomes the unique leader, all the processes act as leaders and send messages, even if i becomes the unique leader and conducts a successful round, there are $\Theta(n^2)$ messages in the channels which are delivered to the agents which respond to these messages.

Automaton BPSUCCESS keeps sending messages to processes that do not acknowledge the "Success" messages. If a process is dead and never recovers, an infinite number of messages is sent. In a real implementation, clearly the leader should not send messages to dead processes.

Finally the automaton DETECTOR sends an infinite number of messages. However the information provided by this automaton can be used also by other applications.

6.6 Concluding remarks

The PAXOS algorithm was devised in [19]. In this section we have provided a new presentation of the PAXOS algorithm. We conclude this section with a few remarks.

The first remark concerns the use of majorities for info-quorums and accepting-quorums. The only property that is used is that there exists at least one process common to any info-quorum and any accepting-quorum. Thus any quorum scheme for info-quorums and accepting-quorums that guarantees the above property can be used.

As pointed out in also [20], the amount of stable storage needed can be reduced to a very few state variables. These are the last round started by a leader (which is stored in the *CurRnd* variable), the last round in which an agent accepted the value and the value of that round (variables *LastR*, *LastV*), and the round for which an agent is committed (variable *Commit*). These variables are used to keep consistency, that is, to always propose values that are consistent with previously proposed values, so if they are lost then consistency might not be preserved. In our setting we assumed that the entire state of the processes is in stable storage, but in a practical implementation only the variables described above need to be stable.

A practical implementation of PAXOS should cope with some failures before abandoning a round. For example a message could be sent twice, since duplication is not a problem for the algorithm (it may only affect the message analysis), or the time bound checking may be done later than the earliest possible time to allow some delay in the delivery of messages.

A recover may cause a delay. Indeed if the recovered process has a bigger identifier than the one of the leader then it will become the leader and will start new rounds, possibly preventing the old round from succeeding. As suggested in Lamport's original paper, one could use a different leader election strategy which keeps a leader as long as it does not fail. However it is not clear to us how to design such a strategy.

7 The MULTIPAXOS algorithm

The PAXOS algorithm allows processes to reach consensus on one value. We consider now the situation in which consensus has to be reached on a sequence of values; more precisely, for each integer k, processes need to reach consensus on the k-th value. The MULTIPAXOS algorithm reaches consensus on a sequence of values; it was discovered by Lamport at the same time as PAXOS [19].

Informally, to achieve consensus on a sequence of values we can use an instance of PAXOS for each integer k, so that the k-th instance is used to agree on the k-th value. Since we need an instance of PAXOS to agree on the k-th value, we need for each integer k an instance of the BASICPAXOS and STARTER automata. To distinguish instances we use an additional parameter that specifies the ordinal number of the instance. So, we have BASICPAXOS(1), BASICPAXOS(2), BASICPAXOS(3), etc., where BASICPAXOS(k) is used to agree on the k-th value. This additional parameter will be present in each action. For instance, the $\text{Init}(v)_i$ and $\text{Decide}(v')_i$ actions of process i become $\text{Init}(k, v)_i$ and $\text{Decide}(k, v')_i$ in BASICPAXOS$(k)_i$. Similar modifications are needed for all other actions. The STARTER$_i$ automaton for process i has to be modified in a similar way. Also, messages belonging to the k-th instance need to be tagged with k.

This simple approach has the problem that an infinite number of instances must be started unless we know in advance how many instances of PAXOS are needed. Hence it is not practical. Furthermore, we have not defined the composition of Clock GTAs for an infinite number of automata (see Section 2).

We can follow a different approach consisting in modifying the BASICPAXOS and STARTER automata of PAXOS to obtain the MULTIPAXOS algorithm. This differs from the approach described above because we do not have separate automata for each single instance. The MULTIPAXOS algorithms takes advantage of the fact that, in a normal situation, there is a unique leader that runs all the instances of PAXOS. The leader can use a single message for step 1 of all the instances. Similarly step 2 can also be handled grouping all the instances together. As a consequence, less messages are used. Then, from step 3 on each instance must proceed separately; however step 3 is performed only when an initial value is provided.

Though the approach described above is conceptually simple, it requires some change to the code of the automata we developed in Section 6. To implement MULTIPAXOS we need to modify BASICPAXOS and STARTER. Indeed BASICPAXOS and STARTER are designed to handle a single instance of PAXOS, while now we need to handle many instances all together for the first two steps of a round. As the changes to the automata code are only technical, we do not provide the modified code; however we refer the interested reader to [5].

The correctness follows from the correctness of PAXOS. Indeed for every instance of PAXOS, the code of MULTIPAXOS provided in this section does exactly the same thing that PAXOS does; the only difference is that step 1 (as well as step 2) is handled in a single shot for all the instances. It follows that Theorem 17 can be restated for each instance k of PAXOS.

8 Application to data replication

Providing distributed and concurrent access to data objects is an important issue in distributed computing. The simplest implementation maintains the object at a single process which is accessed by multiple clients. However this approach does not scale well as the number of clients increases and it is not fault-tolerant. Data replication allows faster access and provides fault tolerance by replicating the data object at several processes.

One of the best known replication techniques is *majority voting* (e.g., [15,16]). With this technique both update (write) and non-update (read) operations are performed at a majority of the processes of the distributed system. This scheme can be extended to consider any “write quorum” for an update operation and any “read quorum” for a non-update operation. Write quorums and read quorums are just sets of processes satisfying the property that any two quorums, one of which is a write quorum and the other one is a read quorum, intersect (e.g., [12]). A simple quorum scheme is the write-all/read-one scheme (e.g., [3]) which gives fast access for non-update operations.

Another well-known replication technique relies on a *primary* copy. A distinguished process is considered the primary copy and it coordinates the computation: the clients request operations to the primary copy and the primary copy decides which other copies must be involved in performing the operation. The primary copy technique works better in practice if the primary copy does not fail. Complex recovery mechanisms are needed when the primary copy crashes. Various data replication algorithms based on the primary copy technique have been devised (e.g., [10,11,23]).

It is possible to use MULTIPAXOS to design a data replication algorithm that guarantees sequential consistency and provides the same fault tolerance properties of MULTIPAXOS. The resulting algorithm lies between the majority voting and the primary copy techniques. It is similar to voting schemes since it uses majorities to achieve consistency and it is similar to primary copy techniques since a unique leader is required to achieve termination. Using MULTIPAXOS gives much flexibility. For instance, it is not a disaster when there are two or more "primary" copies. This can only slow down the computation, but never results in inconsistencies. The high fault tolerance of MULTIPAXOS results in a highly fault tolerant data replication algorithm, i.e., process stop and recovery, loss, duplication and reordering of messages, timing failures are tolerated.

We can use MULTIPAXOS in the following way. Each process in the system maintains a copy of the data object. When client i requests an update operation, process i proposes that operation in an instance of MULTIPAXOS. When an update operation is the output value of an instance of MULTIPAXOS and the previous update has been applied, a process updates its local copy and the process that received the request for the update gives back a report to its client. A read request can be immediately satisfied returning the current state of the local copy. We refer the reader to [5] for automaton code implementing the above algorithm.

9 Conclusion

This paper revisits Lamport's PAXOS algorithm which is a practical and elegant algorithm for solving distributed consensus. Nevertheless, it seems to be not widely known or understood. A modular and detailed description of the algorithm is provided along with a formal proof of correctness and a performance analysis. The formal framework used is provided by the Clock GTA model which is a special I/O automaton model suitable for practical time performance analysis based on the stabilization of the physical system.

Possible future work encompasses an implementation of PAXOS and of data replication algorithms based on PAXOS. We recently learned that Lee and Thekkath [22] used an algorithm based on PAXOS to replicate state information within their Petal system which implements a distributed file server.

Acknowledgments. The first author would like to thank Idit Keidar and Sam Toueg for useful discussions on related work.

References

[1] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, *Journal of the ACM*, Vol. 43 (2), July 1996, pp. 685–722. A preliminary version appeared in the *Proceedings of the* 11th *Annual ACM Symposium on Principles of Distributed Computing*, pages 147–158, Vancouver, British Columbia, Canada, August 1992.

[2] T.D. Chandra, S. Toueg, Unreliable failure detector for asynchronous distributed systems, *Journal of the ACM*, Vol. 43 (2), March 1996, pp. 225–267. A preliminary version appeared in the *Proceedings of the* 10th *Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, August 1991.

[3] E.C. Cooper, Replicated distributed programs. UCB/CSD 85/231, University of California, Berkeley, CA, May 1985.

[4] F. Cristian and C. Fetzer, The timed asynchronous system model, Dept. of Computer Science, UCSD, La Jolla, CA. Technical Report CSE97-519.

[5] R. De Prisco, Revisiting the Paxos algorithm, M.S. Thesis, Massachusetts Institute of Technology, Lab. for Computer Science, Cambridge, MA, June 1997. Technical Report MIT-LCS-TR-717, Lab. for Computer Science, MIT.

[6] R. De Prisco, B. Lampson, N. Lynch, Revisiting the Paxos algorithm, in *Proceedings of the* 11th *International Workshop on Distributed Algorithms*, Saarbrücken, Germany, September 1997.

[7] D. Dolev, C. Dwork, L. Stockmeyer, On the minimal synchrony needed for distributed consensus, *Journal of the ACM*, vol. 34 (1), pp. 77-97, Jan 1987.

[8] Danny Dolev, Roy Friedman, Idit Keidar and Dahlia Malkhi, Failure Detectors in Omission Failure Environments, in *Proceedings of the* 16th *Annual ACM Symposium on Principles of Distributed Systems*, Santa Barbara, CA, August 1997, p. 286. Also TR 96-1608, Department of Computer Science, Cornell

University, September, 1996 and TR CS96-13, Institute of Computer Science, The Hebrew University of Jerusalem.

[9] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, *Journal of the ACM*, vol. 35 (2), pp. 288–323, April 1988.

[10] A. El Abbadi, D. Skeen, F. Cristian, An efficient fault-tolerant protocol for replicated data management, Proc. of the 4^{th} ACM SIGACT/SIGMOD Conference on Principles of Database Systems, 1985.

[11] A. El Abbadi, S. Toueg, Maintaining availability in partitioned replicated databases, Proc. of the 5^{th} ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems, 1986

[12] A. Fekete, N. Lynch, A. Shvartsman, Specifying and using a partitionable group communication service, to appear in *Proceedings of the* 16^{th} *Annual ACM Symposium on Principles of Distributed Computing*, August 1997.

[13] M.J. Fischer, The consensus problem in unreliable distributed systems (a brief survey). Rep. YALEU/DSC/RR-273. Dept. of Computer Science, Yale Univ., New Have, Conn., June 1983.

[14] M.J. Fischer, N. Lynch and M. Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM*, Vol. 32 (2), pp. 374–382, Apr 1985.

[15] D.K. Gifford, Weighted voting for replicated data. Proc. of the $7th$ ACM Symposium on Operating Systems Principles, *SIGOPS Operating Systems Review*, vol. 13 (5), pp. 150–162. Dec 1979.

[16] M.P. Herlihy, A quorum-consensus replication method for abstract data types. *ACM Trans. on Computer Systems* vol. 4 (1), 32–53, Feb 1986.

[17] I. Keidar and D. Dolev, Increasing the resilience of distributed and replicated database systems. *The Journal of Computer and System Sciences (JCSS)*, special issue with selected papers from PODS 1995. To appear.

[18] I. Keidar and D. Dolev, Efficient message ordering in dynamic networks, in *Proceedings of the* 15^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 68–76, May 1996.

[19] L. Lamport, The part-time parliament. *ACM Transactions on Computer Systems*, 16 (2), pp. 133–169, May 1998. Also Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[20] B. Lampson, W. Weihl, U. Maheshwari, Principle of Computer Systems: Lecture Notes for 6.826, Fall 1992, Research Seminar Series MIT-LCS-RSS 22, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, July 1993.

[21] B. Lampson, How to build a highly available system using consensus, in *Proceedings of the* 10^{th} *International Workshop on Distributed Algorithms*, Bologna, Italy, pages 1–15, 1996.

[22] E.K. Lee, C.A. Thekkath, Petal: Distributed virtual disks, In *Proceedings of the* 7^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 84-92, Cambridge, MA, October 1996.

[23] B. Liskov, B. Oki, Viewstamped replication: A new primary copy method to support highly-available distributed systems, in *Proceedings of the* 7^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, August 1988.

[24] N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, San Francisco, 1996.

[25] N. Lynch, M.R. Tuttle, An introduction to I/O automata, *CWI-Quarterly*, 2 (3), 219–246, CWI, Amsterdam, The Netherlands, Sep 89. Technical Memo MIT-LCS-TM-373, Lab. for Computer Science, MIT, Cambridge, MA, USA, Nov 88.

[26] N. Lynch, F. Vaandrager, Forward and backward simulations for timing-based systems. in *Real-Time: Theory in Practice*, Vol. 600 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 397–446, 1992.

[27] N. Lynch, F. Vaandrager, Forward and backward simulations—Part II: Timing-based systems. Technical Memo MIT-LCS-TM-487.b, Lab. for Computer Science, MIT, Cambridge, MA, USA, April 1993.

[28] N. Lynch, F. Vaandrager. Actions transducers and timed automata. Technical Memo MIT-LCS-TM-480.b, Lab. for Computer Science, MIT, Cambridge, MA, USA, October 1994.

[29] M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. *CONCUR 91: 2nd International Conference on Concurrency Theory*, Vol. 527 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 408–423, 1991.

[30] B. Oki, Viewstamped replication for highly-available distributed systems, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1988.

[31] B. Patt-Shamir, A theory of clock synchronization, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, October 1994.

[32] B. Patt-Shamir, S. Rajsbaum, A theory of clock synchronization, in *Proceedings of the* 26^{th} *Symposium on Theory of Computing*, May 1994.

[33] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, *Journal of the ACM* 27 (2), 228–234, Apr 1980.

[34] D. Skeen, D.D. Wright, Increasing availability in partitioned database systems, TR 83-581, Dept. of Computer Science, Cornell University, Mar 1984.

[35] D. Skeen, Nonblocking Commit Protocols, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 133–142, May 1981.