Revisiting the Paxos algorithm

Roberto De Prisco*, Butler Lampson, Nancy Lynch

MIT Laboratory for Computer Science
545 Technology Square NE43, Cambridge, MA 02139, USA.

Abstract. This paper develops a new I/O automaton model called the Clock General Timed Automaton (Clock GTA) model. The Clock GTA is based on the General Timed Automaton (GTA) of Lynch and Vaandrager. 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" behavior. It can be used for practical time performance analysis based on the stabilization of the physical system.
We use the Clock GTA automaton to model, verify and analyze the PAXOS algorithm. 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, it is 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.

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

1 Introduction

I/O automata are simple state machines with transitions labelled with named actions. They are suitable for describing asynchronous and partially synchronous distributed systems. The general timed automaton (GTA) model, introduced by Lynch and Vaandrager [12, 13, 14], has formal mechanisms to represent the passage of time and is suitable for modelling partially synchronous distributed systems. 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. Real distributed systems are subject to failures that may cause a temporary abnormal timing behavior. Hence the above mentioned bounds of ℓ and d can be occasionally violated (timing failures). In this paper we develop an I/O automaton model, called the *Clock GTA*, which provides a systematic way of describing both the normal and the abnormal timing behaviors of a distributed system. The model is intended to be used for performance and fault-tolerance

* Contact author. E-mail: robdep@theory.lcs.mit.edu

analysis of practical distributed systems based upon the stabilization of the system. We use the Clock GTA to formally describe and analyze the PAXOS algorithm, devised by Lamport [8] to solve the consensus problem.

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*, 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 [7]. We refer the reader to [11] 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. 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.

The PAXOS algorithm 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 considered; 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. In [8] 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[2]. 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 [8], 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.

---

[2] PAXOS is the name of the ancient civilization studied in [8]. 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.

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 GTA.

The correctness proof uses automaton composition and invariant assertion methods. 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. The correctness of each piece is proved by means of invariants, i.e., properties of system states that are always true in an 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 $24\ell + 10n\ell + 13d$ time and uses $10n$ messages, where n is the number of processes. This performance is for a worst-case scenario. We also discuss the MULTIPAXOS protocol and provide a data replication algorithm using MULTIPAXOS. With MULTIPAXOS the high availability of the replicated data is combined with high fault tolerance.

Related work. The consensus algorithms of Dwork et al. [5] and of Chandra and Toueg [2] bear some similarities with PAXOS. The algorithm of [5] also uses rounds conducted by a leader, but the rounds are conducted sequentially, whereas in PAXOS a leader can start a round at anytime and multiple leaders are allowed. The strategy used in each round by the algorithm of [5] is different from the one used by PAXOS. The time analysis provided in [5] 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 [6].

The algorithm of Chandra and Toueg is based on the idea of an abstract failure detector. It turns out that failure detectors provide an abstract and modular way of incorporating partial synchrony assumptions in the model of computation. One of the algorithms in [2] uses the failure detector $\diamond\mathcal{S}$ which incorporates the partial synchrony considered in this paper. That algorithm is based on the rotating coordinator paradigm and as PAXOS uses majorities to achieve consistency. The performances of the Toueg and Chandra algorithm and of the PAXOS algorithm seem to be comparable.

Both the Chandra and Toueg algorithm and the Dwork et al. algorithm consider a distributed setting that does not allow process restarts and channel failures (however the Chandra and Toueg algorithm can be modified to work with loss of messages). The PAXOS algorithm tolerates process restarts and channel failures; this makes PAXOS more suitable in practice.

MULTIPAXOS can be easily used to implement a data replication algorithm.

In [10, 15] Liskov and Oki provide a data replication algorithm. It incorporates ideas similar to the ones used in PAXOS.

PAXOS bears some similarities with the standard three-phase commit protocol [17]. However the standard commit protocol requires a fixed leader while PAXOS does not.

In [9] Lampson provides a brief overview of the PAXOS algorithm together with key ideas for proving the correctness of the algorithm.

Cristian's *timed asynchronous model* [3] is very similar to the distributed setting considered in this paper. Our Clock GTA provides a formal way of modelling the stability property of the timed asynchronous model.

In [16] Patt-Shamir introduces a special type of GTA used for the clock synchronization problem. Our Clock GTA automaton considers only the local time; our goal is to model good timing behavior starting from some point on and thus we do not require synchronization of the local clocks.

2 Models

Our formal framework is provided by I/O automata models. I/O automata models are simple type of state machines suitable for describing asynchronous and partially synchronous distributed systems. We use the general timed automaton (GTA), model (see [11], Section 23.2). We introduce a new type of GTA, called *Clock GTA*. We assume that the reader is familiar with the GTA model; briefly, it is a labelled transition system model that includes a time-passage action $\nu(t)$ that represents the passage of (real) time t.

2.1 The Clock GTA model

The Clock GTA model provides a systematic way of describing systems that may exhibit timing failures for portions of their executions, but may behave nicely for other portions. The ability to talk about such changing is crucial for realistic performance fault-tolerance analysis of practical algorithms.

A Clock GTA is a GTA with a special component included in the state; this special variable is called *Clock* and it assumes values in the set of real numbers. 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, 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. 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 any amount t' we are able to model the passage of (local) time by the amount t'. As a special case, we have that $t' = t$; in this case 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 1. A time-passage 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.

Definition 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 steps is irregular.

2.2 The distributed setting

We consider a complete network of n processes communicating by exchange of messages in a partially synchronous setting. Each process of the system is uniquely identified by its identifier $i \in \mathcal{I}$, where $\mathcal{I}$ is a totally ordered finite set of n identifiers, known by all processes. Each process of the system has a local clock. Local clocks may 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 GT automata to model both processes and channels. We assume that processes take actions within ℓ time and that messages are delivered within d time, for given constants ℓ and d. A *timing failure* is a violation of these time bounds. A timing failure can be modelled with an irregular time-passage step.

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 that 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 and can be either `stopped` or `alive`. It is updated by actions Stop_i and Recover_i. A process i is *alive* (resp. stopped) in a given state if in that state we have $Status_i =$ `alive` (resp. $Status_i =$ `stopped`). 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.

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; however, for simplicity, we assume that the entire state of a process is in a stable storage.

Channels. We consider unreliable channels that can lose and duplicate messages. Reordering of messages is allowed and it is not considered a failure. Timing failures are possible. Figure 1 shows the signature[3] of a Clock GT automaton $\text{CHANNEL}_{i,j}$ which models the channel from process i to process j. Channel failures are formally modelled as input actions $\text{Lose}_{i,j}$ (which deletes one of the message currently in the channel), and $\text{Duplicate}_{i,j}$ (which duplicates one of the message currently in the channel).

System stabilization. In the introduction we have pointed out that PAXOS satisfies termination when the system stabilizes. The definition of "nice" execution fragment given below captures the requirements needed to guarantee termination.

[3] The code of this automaton, as well as the code of the other automata we will see later, are omitted from this extended abstract and are deferred to the full paper. The full code can be found in [4].

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

Fig. 1. Automaton CHANNEL$_{i,j}$. The code is deferred to the full paper.

Definition 3. Given a distributed system, we say that an execution fragment α is *stable* if every process is either alive or stopped in α, no Lose$_{i,j}$ and Duplicate$_{i,j}$ actions occur in α and α is regular.

Definition 4. Given a distributed system, we say that an execution fragment α is *nice* if α is a stable execution fragment and a majority of the processes are alive in α.

The next lemma provides a basic property of CHANNEL$_{i,j}$.

Lemma 5. *In a stable execution fragment α of* CHANNEL$_{i,j}$ *beginning in a reachable state s and lasting for more than d time, we have that (i) all messages that in state s are in the channel 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 The consensus problem

In this section we formally define the consensus problem (we remark that several variations of the definition of the consensus problem have been considered in the literature). Each process i in the network receives as input an initial value v, provided by an external agent by means of an action 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 Init$(v)_i$ actions.

To solve the consensus problem means to give an algorithm that, for any execution α, satisfies

– **Agreement:** No two processes output different values in α.
– **Validity:** Any output value in α belongs to V_α.

and, for any admissible infinite 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 occurred in α, then any process alive in γ eventually outputs a value.

The PAXOS algorithm solves the consensus problem defined above.

4 A failure detector and a leader elector

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

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

Fig. 2. Automaton DETECTOR for process i. The code is deferred to the full paper.

A failure detector. Figure 2 shows the signature of Clock GTA DETECTOR$_i$, which detects failures. Automaton DETECTOR$_i$ works by having each process constantly sending "Alive" messages to each other process and checking that such messages are received from other processes. The strategy used by DETECTOR$_i$ is a straightforward one. For this reason it is very easy to implement. The failure detector so obtained is not reliable in the presence of failures (Stop$_i$, Lose$_{i,j}$, irregular executions). However, in a stable execution fragment, automaton DETECTOR$_i$ is guaranteed to provide reliable information on stopped and alive processes.

A leader elector. It is easy to use a failure detector to elect a leader: actions InformStopped$(j)_i$ and InformAlive$(j)_i$ are used to update the current set of alive processes and a common rule to elect the leader is used (the alive process with the biggest identifier is elected leader). Figure 3 shows the signature of automaton LEADERELECTOR$_i$. We denote with S_{LEA} the system consisting of DETECTOR$_i$ and LEADERELECTOR$_i$ automata for each process $i \in \mathcal{I}$ and CHANNEL$_{i,j}$ for each $i, j \in \mathcal{I}$. Processes have a state variable *Leader* that contains the identifier of the current leader. Formally we consider a process i to be *leader* if $Leader_i = i$. This definition allows multiple or no leaders. In a state s, there is a *unique leader* if and only if there exist an alive process i such that $s.Leader_i = i$ and for all other alive processes $j \neq i$ it holds that $s.Leader_j = i$. The following lemma holds.

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

Fig. 3. Automaton LEADERELECTOR for process i. The code is deferred to the full paper.

The following lemma holds.

Lemma 6. *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.*

5 The PAXOS algorithm

PAXOS was devised a very long time ago (the most accurate information dates it back to the beginning of this millennium) but its discovery, due to Lamport, dates back only to 1989 [8]. In this section we provide a new and detailed description of PAXOS.

The core part of the algorithm is BASICPAXOS$_i$. In BASICPAXOS$_i$ processes try to reach a decision by leading what we call a round. A process leading a round is the leader of that round. DETECTOR$_i$ and LEADERELECTOR$_i$ are used to elect leaders. STARTERALG$_i$ makes the current leader start new rounds if necessary. The description of BASICPAXOS$_i$ is further subdivided into three components, namely BPLEADER$_i$, BPAGENT$_i$ and BPSUCCESS$_i$. We will prove (Theorem 13) that the system S_{PAX} ensures agreement and validity, and (Theorem 18) that S_{PAX} guarantees also termination within $24\ell + 10n\ell + 13d$, when the system executes a nice execution fragment. It is worth to remark that some automata need to be able to measure the passage of time, while others do not. For the latter, time bounds are used only for the analysis.

5.1 Automaton BASICPAXOS

We begin with an overview, then provide the code and the analysis.

Overview. The basic idea, which is the heart of the algorithm, is to 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. Since different rounds may be carried out concurrently (several leaders may concurrently initiate rounds), we need to distinguish them. Every round has a unique identifier. 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 $i < j$. If $r < r'$ we say that round r *precedes* round r'. We remark that the ordering on the round numbers is not related to the actual time when rounds are started, i.e., a round with a bigger round number can be conducted before a round with a smaller round number.

Informally, the steps for a round are the following.

1. To initiate a round, the leader sends a "Collect" message to all agents[4].
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. It also commits to not accept any previous round. If the agents is already committed for a round with a bigger round number then it just sends an "OldRound" message.

[4] 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.

3. Once the leader has gathered more than $n/2$ "Last" messages, it decides, according to some rules, the value to propose for its round and sends to all agents a "Begin" message. The set of processes from which the leader gathers information is called the *info-quorum* of the round. 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.
4. An agent that receives a message sent in step 3 from the leader of the round, responds with an "Accept" message by accepting the value proposed in the current round. If the agent is committed for a round with a bigger number then it just sends an "OldRound" message.
5. If the leader gets "Accept" messages from a majority of agents, then the round is successful and the leader sets its own output value to the value proposed in the round. The set of agents that accepts the value proposed by the leader is called the *accepting-quorum*.

Since a successful round implies that the leader of the round reaches a decision, after a successful round the leader needs to broadcast the reached decision.

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. Agreement is guaranteed by choosing the values of new rounds exploiting the information about previous rounds from at least a majority of the processes 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 every agent responds with the number of the latest round in which it accepted the value and the accepted value (or with `nil` if the agent has not yet accepted a value). Once the leader gets such information from a majority of the processes, 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 with its initial value if all processes 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 accepted round is round r', $r' < r$, then implicitly the agent commits itself to not accept any other round r'', $r' < r'' < r$.

To end up with a decision value, rounds must be started until at least one is successful. BASICPAXOS$_i$ guarantees agreement and validity, however, it is necessary to make BASICPAXOS$_i$ start rounds to get termination. We deal with this problem in section 5.3.

The code. In order to describe BASICPAXOS$_i$ 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 and the third one is called BPSUCCESS$_i$ and it simply broadcasts a reached decision (this can be thought of as part of the leader behavior, though we have separated it since it is not part of a round). Automaton BASICPAXOS$_i$ is simply the composition of

Signature of BPLEADER$_i$	
Input:	Receive$(m)_{j,i}$, $m \in \{$"Last", "Accept", "Success", "OldRound"$\}$
	Init$(v)_i$, NewRound$_i$, Stop$_i$, Recover$_i$, Leader$_i$, NotLeader$_i$
Internal:	Collect$_i$, GatherLast$_i$, Continue$_i$ GatherAccept$_i$, GatherOldRound$_i$
Output:	Send$(m)_{i,j}$, $m \in \{$"Collect", "Begin"$\}$
	BeginCast$_i$, RndSuccess$(v)_i$

Fig. 4. Automaton BASICPAXOS for process i. The code is deferred to the full paper.

BPLEADER$_i$, BPAGENT$_i$ and BPSUCCESS$_i$. Our code is "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 loss of messages and responses are expected to be received within given time bounds (we actual deal with this in Section 5.3). 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 starting a new round and abandoning the current one (starting rounds is dealt with in Section 5.3). We remark that in practice it is efficient to cope with some failures by, for example, re-sending messages.

Signature of BPAGENT$_i$	
Input:	Receive$(m)_{j,i}$, $m \in \{$"Collect", "Begin"$\}$
	Init$(v)_i$, Stop$_i$, Recover$_i$
Internal:	LastAccept$_i$, Accept$_i$
Output:	Send$(m)_{i,j}$, $m \in \{$"Last", "Accept", "OldRound"$\}$

Fig. 5. Automaton BPAGENT for process i. The code is deferred to the full paper.

Figures 4 and 5 show the signature of, respectively, BPLEADER$_i$ and BPAGENT$_i$. We remark that BPSUCCESS$_i$ simply takes care of broadcasting a reached decision.

Messages. In this paragraph we describe the messages used for communication between the leader and the agents. The description assumes that i is the leader.

1. "Collect" messages, $m = (r,$"Collect"$)_{i,j}$. Starts round r.
2. "Last" messages, $m = (r,$"Last"$,r', v)_{j,i}$. Provides the last round r' accepted by the agent, and its value v. If the agent did not accept any previous round, then v is either `nil` or the initial value of the agent and r' is $(0, j)$.
3. "Begin" messages, $m = (r,$"Begin"$,v)_{i,j}$. Announces the value v of round r.
4. "Accept" messages, $m = (r,$"Accept"$)_{j,i}$. The agent accepts the value and commits for round r.
5. "Success" messages, $m = ($"Success"$,v)_{i,j}$. Announces the decision v.
6. "Ack" messages, $m =($"Ack"$)_{j,i}$. The agent received the decision.

7. "OldRound" messages, $m = (r, \text{"OldRound"}, r')$. The agent is committed for round $r' > r$.

Partial Correctness. Let us define the system S_{BPX} to be the composition of system S_{LEA} and an automaton BASICPAXOS_i for each process $i \in \mathcal{I}$. In this section we prove the partial correctness of S_{BPX}: 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).

Next we give some definitions that we use in the proofs.

Definition 7. 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. This implies that if a round r is dead, there cannot be a majority of the processes accepting it, thus round r cannot be successful. We denote by $\mathcal{R}_V$ the set of rounds for which the value has been chosen. Next we formally define the concept of *anchored* round which is crucial to the proofs.

Definition 8. 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 $\texttt{Hvalue}(r') = \texttt{Hvalue}(r)$.

Next we prove that S_{BPX} guarantees agreement. The key invariant used in the proof is the following.

Invariant 9. *In any state of an execution of S_{BPX}, any non-dead round $r \in \mathcal{R}_V$ is anchored.*

To prove it we use a sequence of auxiliary invariants. In the following we provide the crucial ones.

Invariant 10. *In any state s of an execution of S_{BPX}, if message $(r, \text{"Last"}, r'', v)_{j,i}$ is in $\text{CHANNEL}_{j,i}$, then $j \in \texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$.*

Invariant 11. *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 $j \in \texttt{Hreject}(r')$.*

Validity is easier to prove since values for new rounds come from either initial values or values of previous rounds.

Invariant 12. *In any state of an execution α of S_{BPX}, for any $r \in \mathcal{R}_V$ we have that $\texttt{Hvalue}(r) \in V_\alpha$.*

The next theorem follows from Invariants 9 and 12.

Theorem 13. *In any execution of the system S_{BPX}, agreement and validity are satisfied.*

5.2 Analysis

In this section we analyze the performance of S_{BPX}. Before turning our attention to the time analysis, let us give the following lemma which provides a bound on the number of messages sent in any round.

Lemma 14. *If an execution fragment of the system S_{BPX}, starting in a reachable state, is stable then at most $4n$ messages are sent in a round.*

Next we consider the time analysis. We remark that in order for the leader to execute step 3, i.e., action BeginCast$_i$, it is necessary that an initial value be provided. If the leader does not have an initial value and no agent sends a value in a "Last" message, the leader needs to wait for the execution of the Init$(v)_i$ to set a value to propose in the round. Clearly the time analysis depends on the time of occurrence of the Init$(v)_i$. For simplicity we assume that an initial value is provided to every process at the beginning of the computation.

We remark that a leader reaches a decision when it conducts a successful round. Formally, a round is successful when action RndSuccess$_i$ is executed.

Lemma 15. *Suppose that for an execution fragment α of the system S_{BPX}, starting in a reachable state s in which no decision has been reached yet, it holds that: (i) α is stable; (ii) in α there exists a unique leader, say process i; (iii) α lasts for more than $7\ell + 4n\ell + 4d$ time; (iv) process i is conducting round r, for some round number r; (v) round r is successful. Then we have that action RndSuccess$_i$ is performed by time $7\ell + 4n\ell + 4d$ from the beginning of α.*

Lemma 16. *If an execution fragment α of the system S_{BPX}, starting in a reachable state and lasting for more than $3\ell + 2n\ell + 2d$ time, is stable and there is a unique leader which has decided before the beginning of α, then by time $3\ell + 2n\ell + 2d$, every alive process has decided, the leader knows that every alive process has decided and at most $2n$ messages are sent.*

Lemmas 14,15 and 16, state that if in a stable execution a successful round is conducted, then it takes a linear amount of time and a linear number of messages to reach consensus. However it is possible that, due to committed agents, 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.

5.3 Starting rounds

Figure 6 shows the signature of Clock GT automaton STARTERALG$_i$. This automaton checks if an ongoing round has been successful within the expected time bound. By Lemma 15, if action RndSuccess$_i$ does not happen within time $7\ell + 4n\ell + 4d$ from the start of the round, then the round may not achieve success and a new round has to be started. This is done by action CheckRndSuccess$_i$. When, in a nice execution fragment, a second round has been started, there is nothing that can prevent the success of the new round. Indeed in the newly started round 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.

Signature of STARTERALG$_i$	
Input:	Leader$_i$, NotLeader$_i$, BeginCast$_i$, RndSuccess$_i$, Stop$_i$, Recover$_i$
Internal:	CheckRndSuccess$_i$
Output:	NewRound$_i$
Time-passage:	$\nu(t)$

Fig. 6. Automaton STARTERALG for process i. The code is deferred to the full paper.

Correctness and analysis. Let S_{PAX} be the system obtained by composing system S_{BPX} with one automaton STARTERALG$_i$ for each process $i \in \mathcal{I}$. Since this system contains as a subsystem the system S_{BPX} then it guarantees agreement and correctness. However, in a long enough nice execution of S_{PAX} termination is achieved, too.

Lemma 17. *Suppose that for an execution fragment α of S_{PAX}, starting in a reachable state s, it holds that (i) α is nice; (ii) there is a unique leader, say process i; (iii) α lasts for more than $16\ell + 8n\ell + 9d$ time. Then by time $16\ell + 8n\ell + 9d$ the leader i has reached a decision.*

Notice that if the execution is stable for enough time, then the leader election will eventually come up with only one leader (see Lemma 6). Thus we have the following theorem.

Theorem 18. *Let α be a nice execution fragment of S_{PAX} starting in a reachable state and lasting for more than $24\ell + 10n\ell + 13d$. Then the leader i executes Decide$(v')_i$ by time $21\ell + 8n\ell + 11d$ from the beginning of α and at most $8n$ messages are sent. Moreover by time $24\ell + 10n\ell + 13d$ from the beginning of α any alive process j executes Decide$(v')_j$ and at most $2n$ additional messages are sent.*

A recover may cause a delay. Indeed if the recovered process becomes leader, it will start new rounds, possibly preventing the old round from success.

6 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 (as long as there are initial values for the k-th consensus problem).

Clearly 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. Few modifications to the code provided in the previous section are needed. 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$_i$ and STARTERALG$_i$ automata. To distinguish instances of BASICPAXOS$_i$ we use an additional parameter that specifies the ordinal number of the instance. So, we have BASICPAXOS$(1)_i$, BASICPAXOS$(2)_i$, BASICPAXOS$(3)_i$, etc., where BASICPAXOS$(k)_i$ is used to agree on the k-th value. This additional parameter will be present in each action. For instance, the Init$(v)_i$ action becomes Init$(k, v)_i$ in BASICPAXOS$(k)_i$. Similar modifications are needed for all the other actions. The STARTERALG$_i$ automaton has to be modified in a similar way. Theorem 18 can be restated for each instance of PAXOS.

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.

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 replication 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. However liveness is not guaranteed: it is possible that a requested operation is never installed.

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.

7 Concluding remarks

This paper introduces a special type of general timed automaton, called the *Clock GTA*, suitable for describing partially synchronous systems subject to timing failures. It can be used for practical time performance analysis based on the stabilization of the physical system. Using the Clock GTA, Lamport's PAXOS algorithm is modelled, verified and analyzed. Future work may encompass on one hand the use of the Clock GTA for modelling other algorithms that work in partially synchronous systems subject to timing failures, and on the other hand improvements and implementation of PAXOS.

References

1. T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, in *Proceedings of the* 11^{th} *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), pp. 225–267. A preliminary version appeared in the *Proceedings of the* 10^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, August 1991.
3. F. Cristian and C. Fetzer, The timed asynchronous system model, Dept. of Computer Science, UCSD, La Jolla, CA. Technical Report CSE97-519.
4. R. De Prisco, Revisiting the Paxos algorithm, M.S. Thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, June 1997. Technical Report MIT-LCS-TR-717, Lab. for Computer Science, MIT, Cambridge, MA, USA, June 1997.
5. C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, *J. of the ACM*, vol. 35 (2), pp. 288–323, April 1988.
6. 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.
7. 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.
8. L. Lamport, The part-time parliament, Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.
9. B. Lampson, How to build a highly available system using consensus, in Proceedings of the 10^{th} *International Workshop on Distributed Algorithms* WDAG 96, Bologna, Italy, pages 1–15, 1996.
10. 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.
11. N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, San Francisco, 1996.
12. 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.
13. 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.
14. 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.
15. B. Oki, Viewstamped replication for highly-available distributed systems, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1988.
16. B. Patt-Shamir, A theory of clock synchronization, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, October 1994.
17. D. Skeen, Nonblocking Commit Protocols, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 133–142, May 1981.

This article was processed using the LaTeX macro package with LLNCS style