Simulation and Analysis of Distributed Systems in Klaim

Francesco Calzolai, Michele Loreti

▶ To cite this version:

Francesco Calzolai, Michele Loreti. Simulation and Analysis of Distributed Systems in Klaim. 12th International Conference on Coordination Models and Languages (COORDINATION) Held as part of International Federated Conference on Distributed Computing Techniques (DisCoTec), Jun 2010, Amsterdam, Netherlands. pp.122-136, 10.1007/978-3-642-13414-2_9 . hal-01054619

HAL Id: hal-01054619

https://inria.hal.science/hal-01054619

Submitted on 7 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Simulation and analysis of distributed systems in KLAIM

Francesco Calzolai and Michele Loreti

Dipartimento di Sistemi e Informatica
Università di Firenze

Abstract. Network and distributed systems typically consists of a large number of actors that act and interact with each other in a highly dynamic environment. Due to the number of involved actors and their strong dependence on mobility and interaction, performance and dependability issues are of utmost importance for this class of systems. STOKLAIM is a stochastic extension of KLAIM specifically thought to facilitate the incorporation of random phenomena in models for network-aware computing. In this paper we show how STOKLAIM can be used to specify and verify quantitative properties of distributed systems. To support the analysis an automatic tool is introduced and used.

1 Introduction

Network and distributed systems typically consist of a large number of actors that act and interact with each other in a highly dynamic environment. Many programming and specification formalisms have been developed that can deal with issues such as (code and agent) mobility, remote execution, security, privacy and integrity. Important examples of such languages and frameworks are, among others, Obliq [6], Seal [7], ULM [5] and KLAIM (*Kernel Language for Agents Interaction and Mobility*) [8,4].

Performance and dependability issues are of utmost importance for "network-aware" computing, due to the number of involved actors and their strong dependence on mobility and interaction. Spontaneous computer crashes may easily lead to failure of remote execution or process movement, while spurious network failures may cause loss of code fragments or unpredictable delays.

Correctness in network and distributed systems, as well as their safety guarantees, is not a rigid notion "*either it is correct or not*" but has a less absolute nature: "in 99.7% of the cases, safety can be ensured"'.

To facilitate the incorporation of random phenomena in models for network-aware computing a stochastic extension of KLAIM [8,4], named STOKLAIM, has been proposed in [9]. KLAIM is an experimental language for distributed systems that is aimed at modelling and programming mobile code applications, i.e., applications for which exploiting code mobility is the prime distinctive feature. In STOKLAIM, every action has a random duration governed by a negative exponential distribution.

In [10], MoSL (*Mobile Stochastic Logic*), a logic that allows one to refer to the spatial structure of the network for the specification of properties for StoKlaim models as been proposed. MoSL is a stochastic logic (inspired by CSL [2,3]) that, together with qualitative properties, permits specifying time-bounded probabilistic reachability properties, such as "the likelihood to reach a goal state within t time units while visiting only legal states is at least 0.92". MoSL is also equipped with operators that permit describing properties resulting from resource production and consumption. In particular, state properties incorporate features for resource management and context verification. Context verification allows the verification of assumptions on resources and processes in a system at the logical level, i.e. without having to change the model to investigate the effect of each assumption on the system behaviour.

In this paper we show how StoKlaim can be used for specifying and verifying quantitative properties of distributed systems. In the paper we use StoKlaim to model three classical leader election algorithm. To support the analysis of StoKlaim systems we use SAM (Stochastic Analyser for Mobility). This is a *command-line* tool that provide functionalities for executing, simulating and model-checking specifications.

Structure of the paper In Section 2 we recall the modelling language StoKlaim, while in Section 3 we briefly describe $MoSL$ and its intuitive semantics. In Section 4 we describe SAM and its main functionalities. This tool is used for analysing three leader election protocols. This analyses are described in Section 5. Section 6 concludes the paper.

2 Klaim

Klaim [4] is a formalism introduced for specifying concurrent and distributed systems. It has been designed to provide programmers with primitives for handling physical distribution, scoping and mobility of processes. Klaim is based on process algebras but makes use of Linda-like asynchronous communication and models distribution via multiple shared tuple spaces [13]. Tuple spaces and processes are distributed over different localities and the classical Linda operations are indexed with the location of the tuple space they operate on.

The Linda communication model was originally proposed for parallel programming on isolated machines. The model permits *time uncoupling* (data life time is independent of the life time of the producer process), *destination uncoupling* (the producer of a datum does not need to know the future use or the final destination of that datum) and *space uncoupling* (communicating processes need to know a single interface, i.e. the operations over the tuple space).

A Klaim system, called a *net*, is a set of *nodes*, each of which is identified by a *physical locality. Physical Localities* can be seen as the addresses of network nodes. Every node has a computational component (a set of processes running in parallel), an *allocation environment* and a data component (a tuple space). Processes interact with each other either locally or remotely by posting and

retrieving tuples to and from a tuple space. Processes can refer nodes by using either physical localities or *logical localities*. While physical localities have a global meaning, logical localities have a local meaning and their interpretation depends on the node where processes run. Indeed, each node is equipped with an allocation environment mapping logical localities to physical localities. When a process uses a logical locality, the allocation environment is used to resolve this name into a physical address.

Processes interact with each other by means of messages, named *tuples*, inserted into located tuple spaces. Tuples are retrieved from tuple spaces via *pattern matching* using *templates*. Processes can also be spawned to be evaluated remotely.

2.1 StoKlaim: Stochastic Klaim

In order to deal with performance and dependability issues Klaim has been extended by adding distribution rates to its actions [9]. In the proposed extension, actions are assumed to have a random duration governed by a negative exponential distribution. System specifying by means of the stochastic extension of Klaim can be formally analysed by using the logic and the model checking technique presented in [10].

Syntactic categories We distinguish the following basic syntactic categories.

- Val, ranged over by $v, v', v_1, \ldots$, is a set of (basic data) *values*;
- LLoc, ranged over by $l, l', l_1, \ldots$, is a set of *logical localities*, also called *localities*; we assume the locality $\mathsf{self} \in \mathsf{LLoc}$;
- PLoc, ranged over by $s, s', s_1, \ldots$, is a set of *physical localities*, also called *sites*;
- $\mathsf{Val\text{-}var}$, ranged over by $x, x', x_1, \ldots$, is a a set of *value variables*;
- $\mathsf{Loc\text{-}var}$, ranged over by $u, u', u_1, \ldots$, is set of *locality variables*;
- $\mathsf{Proc\text{-}var}$, ranged over by $X, X', X_1, \ldots$, be a set of *process variables*.

All the above sets are countable and are mutually disjoint. Let ℓ, ℓ', ℓ_1 range over $\mathsf{Loc} = \mathsf{LLoc} \cup \mathsf{PLoc} \cup \mathsf{Loc\text{-}var}$. We will also use $e, e', e_1, \ldots$ to denote value expressions. The precise syntax of expressions e is not specified since it is irrelevant for the purpose of the present paper. We assume that expressions contain, at least, basic values Val and variables $\mathsf{Val\text{-}var}$.

We adopt the $\tilde{(\cdot)}$-notation for sequences; e.g., $\tilde{l} = l_1, l_2, \ldots, l_n$ denotes a sequence over Loc and $\tilde{x} = x_1, x_2, \ldots, x_m$ is a sequence over $\mathsf{Val\text{-}var}$. For sequence $\tilde{s} = s_1, \ldots, s_n$, let $\{\tilde{s}\}$ denote the set of elements in $\tilde{s}$, i.e., $\{\tilde{s}\} = \{s_1, \ldots, s_n\}$. One-element sequences and singleton sets are denoted as the element they contain, i.e., $\{s\}$ is denoted as s and $\tilde{s} = s'$ as s'. The empty sequence is denoted by ϵ.

$$
\begin{aligned}
N &::= \mathbf{0} \mid l :: E \mid N \parallel N \\
E &::= P \mid \langle \tilde{d} \rangle \\
d &::= P \mid l \mid e \\
P &::= \mathbf{nil} \mid (A, \lambda).P \mid P + P \mid P \mid P \mid X \\
A &::= \mathbf{out}(\tilde{d})@\ell \mid \mathbf{in}(\tilde{t})@\ell \mid \mathbf{read}(\tilde{t})@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(!x) \\
t &::= d \mid ?X \mid ?x \mid ?u
\end{aligned}
$$

Table 1. StoKlaim syntax

Syntax Syntax of StoKlaim nets is reported in Table 1. Specifications in StoKlaim consist of nets and processes. The most elementary net is the null net, denoted $\mathbf{0}$. A net consisting of a single node with *locality* l is denoted $l :: E$ where E is a *node element*. In general, nets consist of several nodes composed in parallel.

Node elements are either processes executing at a node ($l :: P$) or data represented as a *tuple* $\langle \tilde{d} \rangle$ that is stored at a node ($(l :: \langle \tilde{d} \rangle)$),

Processes are built up from the terminated process **nil**, a set of randomly delayed actions, and standard process algebraic constructors such as prefix, choice, parallel composition and process instantiation with optional parameters.

The process $(A, \lambda).P$ executes action A with a random duration that is distributed exponentially with rate $\lambda \in \mathbb{R}^+$.

A process can write tuple $\tilde{d}$ in repository ℓ by the action $\mathbf{out}(\tilde{d})@\ell$. With an input action $\mathbf{in}(t)@\ell$ a process can withdraw a tuple matching pattern $\tilde{t}$ from ℓ.

A pattern is a sequence of *template fields*. These can be either actual fields or a *formal fields*, i.e. a variables prefixed with an question mark to indicate binding of such a variable. Matching predicate *match* is formally defined in Table 2. This leads to a substitution Θ associating values to the corresponding variables, where $\Theta_1 \triangleleft \Theta_2$ denotes the usual composition of substitutions Θ_1 and Θ_2.

$$
match(l, l) \ =_{\text{def}} \ [] \qquad\qquad match(v, v) \ =_{\text{def}} \ []
$$

$$
match(?u, l) \ =_{\text{def}} \ [l/u] \qquad match(?x, v) \ =_{\text{def}} \ [v/x] \qquad match(?X, P) \ =_{\text{def}} \ [P/X]
$$

$$
\frac{match(t_1, d_1) = \Theta_1 \quad \ldots \quad match(t_n, d_n) = \Theta_n}{match((t_1, \ldots, t_n), (d_1, \ldots, d_n)) \ =_{\text{def}} \ \Theta_1 \triangleleft \ldots \triangleleft \Theta_n}
$$

Table 2. Pattern-matching of tuples against templates

Action $\mathbf{read}(\tilde{t})@\ell$ is similar to $\mathbf{in}(\tilde{t})@\ell$ except that retrieved tuple is not deleted from the tuple spaces at ℓ. The action $\mathbf{eval}(P)@\ell$ spawns process P at site ℓ.

Action **newloc**$(?u)$ creates a new node. This action will have also the effect of creating a fresh new address, say l, that identifies the new created nodes. The allocation environment associated to l is obtained by extending the allocation environment ρ of the node where the action is executed, so to bind locality self to l.

Specifications A StoKlaim specification $\mathcal{S}$ is a triple $\mathcal{E}, \Delta \vdash N$ where:

- $\mathcal{E}$ is a function associating to each site in the net N an *allocation environment*;
- Δ is the set of process definitions ($X \stackrel{\Delta}{=} P$);
- N is a net describing both the structure and the behaviour of a system.

Allocation environments are used to associate *logical localities* to *physical localities*. Formally, an allocation environment ρ is a (total) function from Loc to PLoc.

We say that a StoKlaim *specification* $\mathcal{E}, \Delta \vdash N$ is *well-formed* if and only if it is type-correct and:

- for each $s \in \mathsf{PLoc}$, if $\mathcal{E}(s) = \rho$ then:
  - $\forall s' \in \mathsf{PLoc}$: $\rho(s') = s'$;
  - $\rho(\mathsf{self}) = s$
- for each $X \stackrel{\Delta}{=} P \in \Delta$, process variable X occurs *guarded*, i.e. prefixed by an action, in P or as the argument of an **eval** action;
- processes use only localities that really exist.

In the remainder of this paper we assume specifications to be *well-formed*. Moreover, we will also omit $\mathcal{E}$ and Δ from the specification when their definition is clear from the context. Finally we let Spec denotes the set of StoKlaim specifications.

Operational semantics Stochastic behaviour of StoKlaim specifications is defined by means of an *action-labelled* CTMCs (AMCs). These are Continuous Time Markov Chains where transitions are equipped with a label:

Definition 1. *An* action-labelled *CTMC (AMC)* $\mathbb{A}$ *is a triple* $(S, ACT, \mapsto)$ *where* S *is a set of states,* ACT *is a set of actions, and* $\mapsto$ *is the transition function, which is a total function from* $S \times ACT \times S$ *to the set of non-negative real numbers* $\mathbb{R}_{\geq 0}$.

Semantics of StoKlaim specifications is described as an AMC $\mathcal{R}_{SK} = (\mathsf{Spec}, \Lambda, \mapsto)$. Where Spec is the set of StoKlaim specifications, while Λ is the set of transition labels $\mathcal{A}$ defined according to the grammar below:

$$\mathcal{A} ::= s_1 : \mathbf{n}(s) \mid s_1 : \mathbf{o}(\tilde{d}, s_2) \mid s_1 : \mathbf{i}(\tilde{d}) s_2 \mid s_1 : \mathbf{r}(\tilde{d}, s_2) \mid s_1 : \mathbf{e}(P, s_2)$$

Each label identifies the site where the action is performed, the argument of the action and the site where the action takes effect.For instance, $i_1 : \mathbf{o}(v, i_2)$ represents the uploading of value v from site i_1 to site i_2. Interested reader can refer to [10] for a complete description of StoKlaim semantics.

3 MoSL: Mobile Stochastic Logic

Performance and dependability properties of StoKlaim systems can be specified by means MoSL [10] (Mobile Stochastic Logic). Key features of this logic are:

- it is a *temporal logic* that permits describing the dynamic evolution of the system;
- it is both *action*- and *state*-based;
- it is a *real-time* logic that permits the use of real-time bounds in the logical characterisation of the behaviours of interest;
- it is a *probabilistic logic* that permits expressing not only functional properties, but also properties related to performance and dependability aspects; and, finally
- it is a *spatial logic* that references the spatial structure of the network for the specification.

In this section we briefly recall syntax (Table 3) and the intuitive semantics of $MoSL$.

$$
\begin{array}{ll}
\Phi & ::= \mathtt{tt} \quad | \quad \aleph \quad | \quad \neg\Phi \quad | \quad \Phi \vee \Phi \quad | \quad \mathcal{P}_{\bowtie p}(\varphi) \quad | \quad \mathcal{S}_{\bowtie p}(\Phi) \\
\varphi & ::= \Phi \ {}_{\Delta}\mathcal{U}^{<t}_{\Omega}\ \Psi \mid \Phi \ {}_{\Delta}\mathcal{U}^{<t}\ \Psi \\
\aleph & ::= PTF@\imath \to \Phi \mid \langle \tilde{F} \rangle @\imath \to \Phi \mid Q(\tilde{Q}', \tilde{\ell}, \tilde{e})@\imath \leftarrow \Phi \mid \langle \tilde{f} \rangle @\imath \leftarrow \Phi \\
PTF & ::= Q(\tilde{Q}', \tilde{\ell}, \tilde{e}) \mid !X \\
\Delta & ::= \top \mid \{\} \mid \{\xi_1, \ldots, \xi_n\} \\
\xi & ::= g : \mathbf{N}(g) \mid g : \mathbf{O}(\tilde{F}, g) \mid g : \mathbf{I}(\tilde{F}, g) \mid g : \mathbf{R}(\tilde{F}, g) \mid g : \mathbf{E}(PTF, g)
\end{array}
$$

Table 3. Syntax of MoSL formulae

As in the branching-time temporal logic CTL, also in MoSL two classes of formulae are considered: *state* formulae $\Phi, \Phi', \Phi_1, \ldots$ and *path* formulae $\varphi, \varphi', \varphi_1, \ldots$.

There are three categories of state formulae. The first category includes formulae in propositional logic, where the atomic propositions are $\mathtt{tt}$ and the basic state formulae. The second category includes statements about the likelihood of paths satisfying a property. Finally there are the so-called long-run properties.

Basic state formulae are built using a variant of $MoMo$ [11] *consumption* ($\to$) and *production* ($\leftarrow$) operators. Production and consumption operators permit formalising properties concerning the availability of resources (i.e. located tuples and processes) and system's reactions to placement of new resources in a state. For instance, a consumption formula $Q@\imath \to \Phi$ holds for a network whenever in the network there exists a process Q running at a node, of site $\imath$, and the "remaining" network, namely Q's context, satisfies Φ.

Path formulae basic rely on the CTL *until* operator $\Phi \ \ \mathcal{U}\ \Psi$. In order to be able to refer also to actions executed along a path, a variant of the *until*

operator as originally proposed in action-based CTL [12] is used. To that end, the until-operator is parameterised with two *action specifiers.*

Here, $\top$ stands for "any set" and can be used when no requirement on actions is imposed. A set of action specifiers is satisfied by an action if the latter satisfies at least one of the elements of the set. Action specifiers (ξ) are a kind of templates for actions. The action specifier $s_1 : \mathbf{o}(\tilde{d}, s_2)$, is satisfied only by action $(s_1 : \mathbf{o}(\tilde{d}, s_2))$. Action specifiers may contain binders that bind their variables to corresponding values in actions in the path; e.g., the action specifier $?u_1 : \mathbf{o}(\tilde{t}, !u_2)$ is satisfied by any action, executed at some site, which uploads a tuple matching $\tilde{t}$ to some site. Action specifiers and their matching to actions generate substitutions that bind variables in subformulae. The meaning of the other action specifiers is now self-explanatory.

A path satisfies $\Phi \ {}_{\Delta}\mathcal{U}_{\Omega} \ \Psi$ whenever eventually a state satisfying Ψ—in the sequel, a Ψ-state—is reached via a Φ-path—i.e. a path composed only of Φ-states—*and*, in addition, while evolving between Φ states, actions are performed satisfying Δ and the Ψ-state is entered via an action satisfying Ω. Finally, we add a time constraint on path formulae. This is done by adding time parameter t—in much the same way as in timed CTL [1]—which is either a real number or may be infinite. In addition to the requirements described just above, it is now imposed that a Ψ-state should be reached within t time units. If $t = \infty$, this time constraint is vacuously true, and the until from action-based CTL is obtained. Similarly, a path satisfies $\Phi \ {}_{\Delta}\mathcal{U}^{<t} \ \Psi$ if the initial state satisfies Ψ (at time 0) or eventually a Ψ state will be reached in the path, by time t via a Φ-path, and, in addition, while evolving between Φ-states, actions are performed satisfying Δ.

4 SAM

In this section we present SAM (Stochastic Analyser for Mobility) [22]. This is a *command-line* tool, developed in OCaML, that supports the stochastic analysis of StoKlaim specifications. SAM can be used for:

- executing interactively specifications;
- simulating stochastic behaviours;
- model checking MoSL formulae.

Running a specification SAM provides an environment for interactive execution of StoKlaim specification. When a specification is executed, a user can select interactively possible computations.

Simulating a specification To analyse behaviour of distributed systems specified in StoKlaim, SAM provides a simulator. This module randomly generates possible computations. At each step of the simulation the next state is determined by using the Gillespie Algorithm [14]. A simulation continues until in the considered computation either a *time limit* or a deadlock configuration is reached.

Fixed a *sampling time*, each computation is described in term of the number of resources (located tuple) available in the system during the computation. At the end of a simulation, the average amount of resources available in the system at specified time intervals is provided.

To identify the values to collect in a simulation a sequence of elements (named *experiments*) of the form:

$\mathsf{label} : \langle \tilde{d} \rangle @ lp$

is provided. An experiments associate a label (label in the example above) to a locate tuple ($\langle \tilde{d} \rangle @ lp$) where lp can be either a site or a wildcard (*). In the former case, the number of tuples in the considered localities are counted. In the latter, the tuples in all the localities are summed.

For instance what follows are two experiments that can be used for computing the number of tuple ⟨"FOLLOWER"⟩ and ⟨"LEADER"⟩ available in a net:

follower: ⟨"FOLLOWER"⟩@$*$
leader: ⟨"LEADER"⟩@$*$

Model checking SAM permits verifying whether a given StoKlaim specification satisfies or not a MoSL formula. This module, which implements the model checking algorithm proposed in [10], use an existing state-based stochastic model-checker, the Markov Reward Model Checker (MRMC) [19], and wrapping it in the MoSL model-checking algorithm. After loading a StoKlaim specification and a MoSL formula, it verifies, by means of one or more calls to MRMC, the satisfaction of the formula by the specification.

Unfortunately, even simple StoKlaim specification can generate a very large number of states. For this reason, the *numerical* model checking cannot always be applied. To overcome the state explosion problem, a *statistical model-checker* has been also implemented in SAM. The statistical approach has been successfully used in existing model checkers [15,21,24,25].

While in a numerical model checker the exact probability to satisfy a path-formula is computed up to a precision ϵ, in a *statistical model-checker* the probability associated to a path-formula is determined after a set of independent observations. This algorithm is parametrised with respect to a given *tolerance* ε and *error probability* p. The algorithm guarantees that the difference between the computed values and the exact ones are greater than ε with a probability that is less than p.

5 Leader Election in StoKlaim

In this Section we use StoKlaim for specifying a system where n distributed sites (nodes) have to elect a leader (a uniquely designed process): we will consider three well known leader election protocols [23,17]: *All The Way*, *As far as it can* and *Asynchronous leader election.* The first two protocols are modelled by relying on agents and code mobility, while the third is modelled by relying on message passing.

In all the considered algorithms, it is assumed that the nodes are always arranged in a ring: in this particular network topology every node is connected to two other nodes (called *prev* and *next*). This is a common assumption for leader election algorithms [23].

In StoKlaim the system consists of n nodes each of which hosts the execution of an agent or a process. We assume these nodes are identified by sites $(s_0,\ldots, s_{n-1})$. The index associated to a site identifies the position of the node in the ring: the process located at s_i precedes (follows) the one located at $s_{i+1 \mod n}$ $(s_{i-1 \mod n})$ in the ring. We also assume that the allocation environment ρ_i of sites s_i, beside the standard mapping $\mathsf{self} \mapsto s_i$, maps logical locality next to $s_{i+1 \mod n}$ and logical locality pred to $s_{i-1 \mod n}$. In the performance analysis we will consider four ring sizes: 25, 50, 75 and 100. Notice that, for these configurations, standard model checking techniques are note easily applicable. Indeed, if n is the number of nodes in the ring, the first two algorithms generate models with more than $n!$ states while the models generated from the third algorithm are composed of more than 2^n states.

In the considered models we assume that local communications are performed with rate 15.0 while remote communications have rate 1.0.

In the analysis we will use simulations and (statistical) model checking. Simulations will be performed by considering 300 iterations and will be used to determine how, in the average, process change their state. Statistical model checking will be used to compute the probability, when t varies from 0 to 600, of:

$$\mathsf{tt}\ {}_{\top}\mathcal{U}^{<t} \bigvee_i \langle\text{“}\mathsf{LEADER}\text{”}\rangle @ s_i \rightarrow \mathsf{tt}$$

This formula identifies the paths that leads one of the site to become a leader within time t.

5.1 All the way

In this algorithm every participants is univocally identified by an *id* selected randomly. For this reason, at the beginning each process retrieve a value (the *localID*) from a specific site (rg) that acts as random generator. The leader will be the node with the minimum *id*.

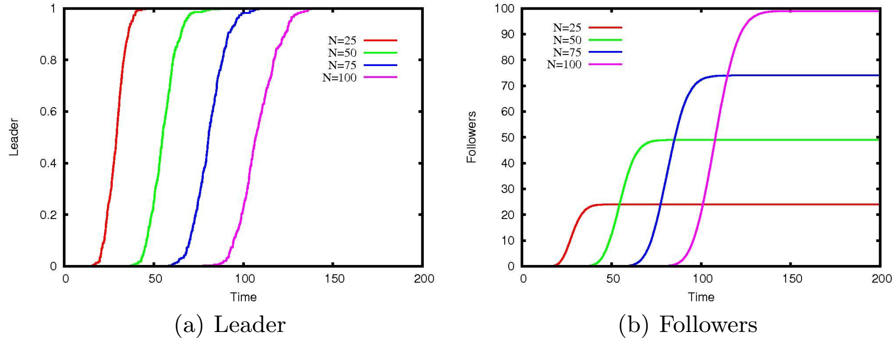
When a process has determined its *id*, an agent is sent to the next node in the ring. This agent carries a node *id* and the minimum seen so far (*min*). Agent travels *all the way* along the ring in order to find the minimum value in the net. When an agent arrives in a node, it reads the site's *id* and updates the minimum accordingly. As soon as an agent returns to the originating node, it is able to determine the node's roles: *leader* if its *id* is the smallest in the ring, *follower* otherwise.

What follows is the agent devoted to find the minimum *id* on the ring and that decides if its starting node can be the leader[1]:

[1] SAM uses a richer syntax for specifying processes. This contains standard command like selection and iterations and typed variables.

$agent[int\ id,\ int\ min] \triangleq$
  (**read**("ID", $?lId$)@self, $local$).
  if $(id = lId)$ **then**
    if $(id = min)$ **then**
      (**out**("LEADER")@self, $local$)
    else
      (**out**("FOLLOWER")@self, $local$)
  else
    if $(min < lId)$ **then**
      (**eval**($agent(id,\ min)$)@next, $remote$)
    else
      (**eval**($agent(id,\ lId)$)@next, $remote$)

The result of the simulation of the system is reported in Figures 1(a) and 1(b): on the left-hand side it is shown the average number of leaders, while on the right-hand side it is shown the average number of followers. From these pictures it is clear that, in the average, all the nodes (both *leader* and *followers*) terminate the algorithm approximately at the same time. This means that a *follower* continues to play an active role in the system till the end of the computation.



(a) Leader (b) Followers

Fig. 1. *AllTheWay* simulation

5.2 As far as it can

The algorithm considered in the previous section is not very efficient. Indeed, every agent keeps travelling even if its *id* is not the smaller. An improvement of the *All the way* algorithm is the *As far as it can*. In this algorithm an agent moves to the next node if and only if its *id* is smaller than the local identifier. An agent travels along the ring "as far as it can", until it stops in a site with a smaller (or equals) *id*. Only the agent with the smaller *id* is able to return to its starting site, all the others will be eventually stopped. After the survivor agent

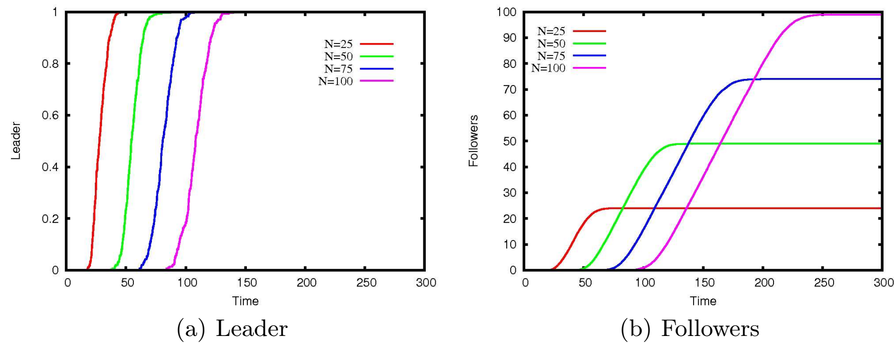
travelling all the ring, it sets the starting node as *leader* and then creates the agent notify that revisits all the other nodes setting them as *follower*.

The following is the agent used in the StoKlaim specification to visit the ring:

$agent[int\ id,\ int\ min] \triangleq$
  $(\mathbf{read}(\text{“ID”},\ ?lId)@\mathsf{self}, local).$
  $\mathbf{if}\ (id = lId)\ \mathbf{then}$
    $(\mathbf{out}(\text{“LEADER”})@\mathsf{self}, local).$
    $(\mathbf{eval}(notify(id))@\mathsf{next}, remote)$
  $\mathbf{else}$
    $\mathbf{if}\ (min < lId)\ \mathbf{then}$
      $(\mathbf{eval}(agent(id,\ min))@\mathsf{next}, remote)$

$notify[int\ id] \triangleq$
  $(\mathbf{read}(\text{“ID”},\ ?lId)@\mathsf{self}, local).$
  $\mathbf{if}\ (id! = lId)\ \mathbf{then}$
    $(\mathbf{out}(\text{“FOLLOWER”})@\mathsf{self}, local).$
    $(\mathbf{eval}(notify(id))@\mathsf{next}, remote)$

Simulation results, reported in Figure 2, shows that, differently from *All the way*, the leader is selected rapidly.



(a) Leader (b) Followers

Fig. 2. *AsFarAsItCan* simulation

5.3 Asynchronous leader election

The algorithm considered in the previous sections identify the leader as the node with the minimum *id*. In this section we consider a *randomised* protocol that is an adaptation of the *asynchronous leader election protocol* proposed in [17].

Processes in the system can be either *active* or *inactive*. Until a process becomes inactive, it performs the following steps:

1. Chooses 0 or 1 each with a given probability, and sends the choice to the next process.

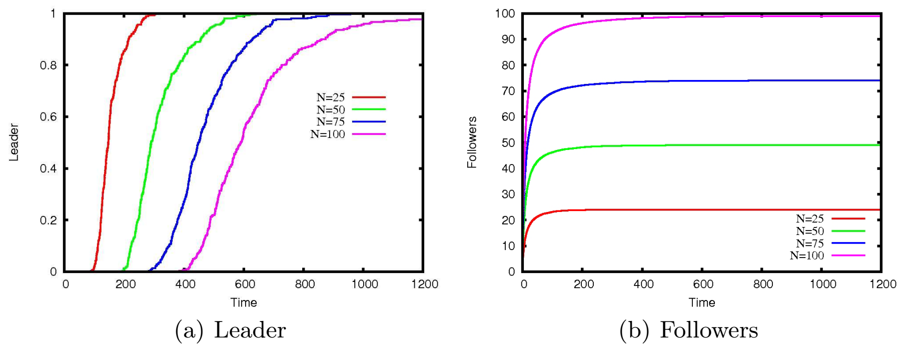
2. If the process chose 0 and the active process preceding it in the ring chose 1 it becomes inactive and only continues to relay received messages.
3. If it is still active, it sends a counter around the ring to check whether it is the only active process. In that case it becomes the leader, otherwise another round of the algorithm is repeated.

The model for an *active* process is shown below. Notice that process *testLeader* is the process used for verifying if the current node is a leader. An *active* process is modelled in StoKlaim as follows:

```
active[int id] ≜
  (out(0)@next, remote).
    (in(?val)@self, local).
    if (val = 1) then
      (out("FOLLOWER")@self, local).
      inactive()
    else
      testLeader(id)
  +(out(1)@next, remote).
    (in(?val)@self, local).
    testLeader(id)
```

```
testLeader[int id] ≜
  (out("CHECK", id)@next, remote).
  (in("CHECK", ?c :)@self, local).
  if (c = id) then
    (out("LEADER")@self, local)
  else
    active(id)
```

The simulation of considered system (Figure 3) shows that, differently from the specifications considered in previous sections, a large number of nodes become *inactive* before the leader is selected. Unfortunately, the time needed to determine the leader increases significantly.



(a) Leader (b) Followers

Fig. 3. *Asynchronous* simulation

5.4 Final considerations

We now use statistical model checking to contrast the three considered algorithms. In particular we study how the probability to select a leader within t time units varies in the three algorithms. In Figure 4 are reported the model checking results for a configuration with $n = 75$ nodes in the ring. The results respect the one obtained with the simulation. *All the way* algorithm and *As far as it can*, in the average, are faster then the *Asynchronous leader election*.

Notice that to compute the probability of the considered formulae, due to the size of considered system, the use of numerical model checking is not practicable. On the contrary, statistical model checker allow us to compute an approximation of the requested probability.

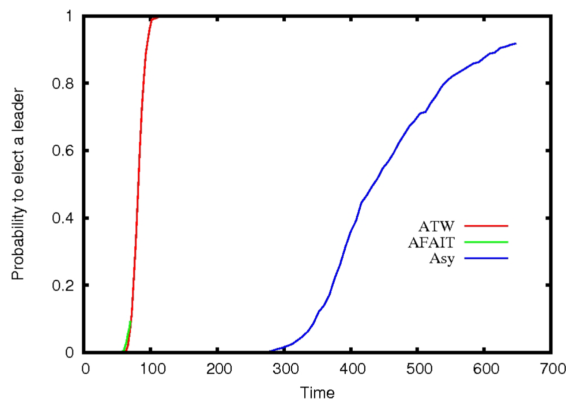


Fig. 4. Comparison between the three algorithm

6 Conclusions and Future works

In this paper we have shown how StoKlaim can be used for specifying and verifying quantitative aspects of distributed systems. In the paper the analysis of three well known leader election algorithms are considered. The proposed analysis have been performed by relying on SAM: Stochastic Analyser for Mobility. This is a *command-line* tool that provide functionalities for executing, simulating and model-checking specifications.

Several software tools have been developed for supporting verification (via model checking) of quantitative aspects of concurrent systems. We just mention PRISM [20], ETMCC [16] and *MRMC* [18]. However, many of these frameworks either do not use a process algebra to support system specification (like, for instance, MRMC and ETMCC), or the considered process algebra does not provide linguistic primitives for describing distribution and mobility. On the contrary, the use of StoKlaim, where *localities* and *process mobility* are first class citizens, simplifies the specification and the analysis of distributed systems.

As a future work we plan to study the possibility to extend the model checker in order to take into account rewards. Notice that the algorithms considered in Section 5 are analysed by considering the time needed to select a leader. By introducing rewards in the logic, other interesting aspects, like for instance the average number of messages exchanged by the different algorithms, can be taken into account.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoret. Comput. Sci.*, 126:183–235, 1994.
2. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking Continuous Time Markov Chains. *Transations on Computational Logic*, 1(1):162–170, 2000.
3. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate Symbolic Model Checking of Continuous-Time Markov Chains. In J. Baeten and S. Mauw, editors, *Concur '99*, volume 1664 of *Lect. Notes in Comput. Sci.*, pages 146–162. Springer, 1999.
4. L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim Project: Theory and Practice. In C. Priami, editor, *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, volume 2874 of *Lect. Notes in Comput. Sci.*, pages 88–150. Springer, 2003.
5. G. Boudol. ULM: a core programming model for global computing: (extended abstract). In D. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming (ESOP)*, volume 2986 of *Lect. Notes in Comput. Sci.*, pages 234–248. Springer, 2004.
6. L. Cardelli. A Language with Distributed Scope. In *22nd Annual ACM Symposium on Principles of Programming Languages*, pages 286–297. ACM, 1995.
7. G. Castagna and J. Vitek. Seal: A framework for Secure Mobile Computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, volume 1686 of *Lect. Notes in Comput. Sci.*, pages 47–77. Springer, 1999.
8. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–329, 1998.
9. R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. KLAIM and its stochastic semantics. Technical report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2006. Available at `http://rap.dsi.unifi.it/~loreti/papers/TR062006.pdf`.
10. R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. Model checking mobile stochastic logic. *Theoretical Computer Science*, 382(1):42–70, 2007.
11. R. De Nicola and M. Loreti. Multiple-Labelled Transition Systems for nominal calculi and their logics. *Mathematical Structures in Computer Science*, 18(1):107–143, 2008.
12. R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Proceedings of LITP Spring School on Theoretical Computer Science*, volume 469 of *Lect. Notes in Comput. Sci.*, pages 407–419. Springer, 1990.
13. D. Gelernter. Generative Communication in Linda. 7(1):80–112, 1985.
14. D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

15. G. N. H. Younes, M. Kwiatkowska and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, June 2006.
16. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Tool for Model-Checking Markov Chains. *International Journal on Software Tools for Technology Transfer*, 4(2):153–172, 2003.
17. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.
18. J.-P. Katoen, M. Khattri, and I. Zapreev. A Markov reward model checker. In *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 243–244. IEEE Computer Society, 2005. ISBN 0-7695-0418-3.
19. J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE CS Press, 2005.
20. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking using PRISM: A Hybrid Approach. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.
21. P. Quaglia and S. Schivo. Approximate model checking of stochastic cows. In *Proc. of TGC 2010*. To appear., 2010.
22. Sam: Stochastic analyser for mobility. http://rap.dsi.unifi.it/SAM/.
23. N. Santoro. *Design and Analysis of Distributed Algorithms*. Wiley, 2006.
24. K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 202–215. Springer, 2004.
25. K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2005.