Mobile Agents For Implementing Local Computations in Graphs

Bilel Derbel, Stefan Gruner, Mohamed Mosbah

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Mobile Agents For Implementing Local Computations in Graphs

Bilel Derbel — Stefan Gruner — Mohamed Mosbah

N° 6506

April

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6506--FR+ENG

Mobile Agents For Implementing Local Computations in Graphs

Bilel Derbel* , Stefan Gruner† , Mohamed Mosbah‡

Thème COM — Systèmes communicants
Équipes-Projets DOLPHIN

Rapport de recherche n° 6506 — April — 17 pages

Abstract: Mobile agents are a recent paradigm to facilitate the design and programming of distributed applications. However, whilst their popularity continues to grow, a uniform theory of mobile agent systems is not yet sufficiently elaborated, in comparison with classical models of distributed computation. In this paper we show how to use mobile agents as an alternative model for implementing distributed local computation rules. In doing so, we approach a general and unified framework for local computations which is consistent with the classical theory of distributed computations based on graph relabeling systems.

Key-words: Distributed algorithms, Mobile agents, Graph relabeling systems.

* Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université de Lille 1 (USTL). Supported by the DOLPHIN inria project team.
† Laboratoire Bordelais de Recherche en Informatique (LaBRI), Université Bordeaux 1 - ENSEIRB, France.
‡ Department of Computer Science, University of Pretoria, South Africa

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley 59650 Villeneuve d'Ascq (France) Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Mobile Agents For Implementing Local Computations in Graphs

Résumé : Mobile agents are a recent paradigm to facilitate the design and programming of distributed applications. However, whilst their popularity continues to grow, a uniform theory of mobile agent systems is not yet sufficiently elaborated, in comparison with classical models of distributed computation. In this paper we show how to use mobile agents as an alternative model for implementing distributed local computation rules. In doing so, we approach a general and unified framework for local computations which is consistent with the classical theory of distributed computations based on graph relabeling systems.

Mots-clés : Distributed algorithms, Mobile agents, Graph relabeling systems.

1 Introduction

Models of *local computations*, described by *graph relabeling systems* provide a powerful theoretical framework to specify and reason about various aspects of distributed computation with distributed algorithms [11, 12, 2]. Assuming that the reader is already familiar with this theoretical background, we will only briefly recapitulate the basic characteristics and features of modeling distributed systems by local computations and graph relabeling systems. This well-known paradigm will be our starting point from where we shall proceed towards a more recent paradigm of distributed computation by *mobile agents*.

Our aim is to demonstrate that all "basic building blocks" of the graph relabeling paradigm can be implemented by the activities of *mobile agents*, leading to the conjecture that mobile agents are as powerful as classical distributed systems, i.e., message passing systems [4]. In *practice* the use of mobile agents for the implementation of distributed algorithms can have *advantages* over classical implementations, because roaming agents can better cope with temporary network failures and also consume less computational resources, in comparison with the global network activities induced by classical implementations of distributed algorithms. In addition, mobile agents allow to bring a new level of abstraction in distributed computing. For instance, in the message passing model, the nodes represent both the topology of the network and the autonomous computation entities. In opposite, in the mobile agent model, the nodes define only the topology of the network, while the agents define the computation entities of the network.

The consideration (description, reconstruction) of agent systems in terms of graph transformation systems is not a new idea; take for example [10] as an early contribution to this field of study. In [10], however, graph transformation techniques are used to model *internal* properties and/or actions of agents, whereas the focus of our paper is on their *external* properties, mainly *motion* between network places, motivated by our intention to demonstrate the possibility of expressing (respectively implementing) classical distributed algorithms in terms of mobile agent systems. To this end, graph transformation systems can be regarded as the "bridge" formalism between the domain of classical distributed algorithms and the domain of mobile agent systems.

Graph Relabeling Systems: Processor networks, which are the substrate of distributed computation, are represented by labeled graphs $G = (V, E, L, \lambda)$ with a set of labels L and a (possibly partial) labeling function $\lambda : (V \uplus E) \longrightarrow L$ that attaches labels to vertices (nodes) and/or edges (arcs) of the network graph. The labels, which may lexically appear arbitrarily complex, are used to model the internal states of the network components during the run of a distributed algorithm on the network. A final label configuration represents the result of a terminated algorithm. Thereby, the models must be designed in such a way that three *locality conditions* are always fulfilled:

c1: Relabeling does not modify the underlying graph structure (from a topological point of view);

c2: Each step can only relabel a limited, connected sub graph (fixed in size);

c3: The applicability of a relabeling step in a "neighborhood" is constrained only by the local conditions within such a neighborhood, not by the global state of the entire network.

Distributed algorithms described in such a framework are usually composed of *basic units* which correspond to certain types of relabeling rules. These various rule types, which are classified and explained in [5], comprise constructs such as:

- single node relabeling depending on only one neighbor, i.e., relabeling of half an edge,
- two neighbor relabeling, i.e., relabeling of an entire edge,
- single node relabeling depending on labels of all neighbors, i.e., only the label of the center of a *star* is relabeled according to the labels attached to the star (radius 1),
- relabeling of an entire star depending on the labels attached to the star,
- single node relabeling in the center of a *ball* of radius 2,

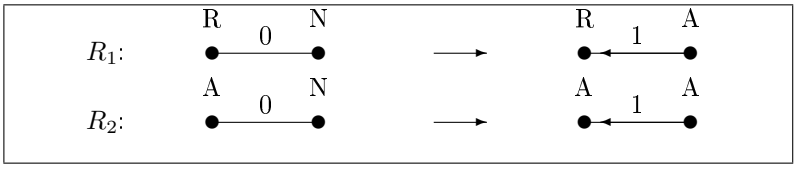


Figure 1: Rules for a distributed construction of a rooted spanning tree

- single node relabeling in the center of larger balls or stars,
- relabeling of an entire ball of radius k with k an integer parameter,

Note that not every rule type is suitable for composing (describing) a particular distributed algorithm. For example, whereas the distributed computation of a spanning tree across the underlying network can well be described in terms of the most simple rule type (relabeling one node depending on only one neighbor), others, more complicated distributed algorithms can only be described in terms of more complicated types of relabeling rules [13]. Anyway, a relabeling system gives us a uniform and unified methodology of thinking and proving distributed algorithms. For example, Figure 1 shows a simple relabeling rule system, consisting of two rules only, by means of which a spanning tree of a graph can be computed in a distributed and self-organizing fashion[1].

Mobile Agent Systems: One can ask how to turn out a set of relabeling rules into an executable distributed algorithm. In other words how to implement a distributed algorithm described with a relabeling system into a practical distributed setting. Because we can find many types of distributed systems relying on the type of communication (e.g., messages, shared memory), the type of synchrony (e.g., synchronous, asynchronous), and the type of computation entities (e.g., processors, mobile agents), many solutions are possible. For instance, some algorithms are known for the case of classical message passing systems [15, 16, 14, 9]. In this paper we are interested in a uniform and practical mobile agent solution.

When a distributed algorithm is to be implemented by means of mobile agents, a variety of issues must be considered. Amongst those, there are some especially important considerations concerning the nature of *synchronization*, the notion of *agent*, as well as the organization of *agent processing* whenever an agent arrived at a particular node in the network.

In order to perform local relabeling 'classically', some type of synchronization is needed for a short period of time between the involved nodes. In the usual implementation of a local relabeling step [3], *messages* are sent between the involved nodes such that, depending on the information contents of those messages, synchronization can be achieved. In a pure mobile agent system, however, there are no messages; there are only agents moving from node to node. Consequently, the notion of "synchronization" looses its traditional meaning: In a classical distributed system, all nodes are active during the same time. They might not have a common clock and might follow their own local speed of pace, but no node is supposed to fall asleep until the termination of the algorithm. In a mobile agent system, on the contrary, a network node is asleep as long as no agent is locally present: consequently, the notion of "synchrony" in a mobile agent system can only be circumscribed in terms of particular *patterns of agent moves* between two quasi-synchronized neighbor nodes.

Remainder of the Paper: In the remainder of the paper we present some mobile agent implementations of relabeling systems.

- In the next section, we broadly classify various possible ways of agent system specification, including the "inner workings" of the underlying network and its nodes. This shall help the reader to comprehend the particular contributions of our paper in a wider context.

[1] In the example of Fig. 1, it is implicitly assumed that at the beginning there exists a unique node with label R (root) in the network graph, and all the other nodes have label N

- In Section 3, we describe two implementations of two basic classes of local relabeling rules called type LC0 and LC1, and we classify these approaches in terms of the previously outlined classification.
- In Section 4 we describe a *general* methodology (or framework) for implementing *any* class of relabeling rules, thereby following an interleaving semantics, stateful and Concurrent approach (in terms of our classification).
- Summary and acknowledgments conclude the paper and hint at possible future work.

2 Classification of Agent Systems and Network Nodes

Mobile agent systems, as well as the kinds of networks in which they operate, are manyfold. To better understand a particular piece of work (including our contribution) in this field of study, it is useful to have some kind of classification at hand, according to which various solutions or approaches can be compared and appreciated. This section sketches such a classification — not in great detail (which is not the central purpose of our paper), yet sufficient for placing our contribution into a wider context for the sake of better understanding.

Agent Population: As far as the agent population of a network is concerned, we can distinguish *homogeneous* populations, in which all agents are of the *same type*, from *heterogeneous* populations, the agents of which can be of various *different types*.

Agent Description: As far as the *description* of an *agent algorithm* is concerned, we can distinguish *regional* from *local* descriptions. A local description describes the activities of an agent with respect to *one node* only, on a conceptually low level, in a "terminate-and-restart" mode, in which the agent shows the same behaviour on every network node. A regional algorithm, on the contrary, describes the activities of an agent on a *set* of nodes, in a "break-and-continue" mode, including the migrations from node to node: according to this high-level behaviour description, an agent can show different types of behaviour at different places.

Agent States: Closely related to the form of agent behavior description (regional or local — see above) is the question of how an agent must be "equipped" in order to be able to perform regional or local algorithms. In this context we can distinguish *state-rich* from *state-poor* agents. State-poor agents are rather minimalist creatures which do not need to carry their own program-pointer from node to node, because they simply reboot and restart themselves whenever they arrive at a new node that can execute their code. State-rich agents, on the contrary, who can interrupt their algorithm and continue it at another node, need to be able to carry their own program pointer with them from node to node for this purpose. Every agent, however, needs to be endowed with a minimum of memory, such that he can at least remember on his journey where he came from. If an agent is nothing else but a piece of software code, then "memory modification" to this agent cannot mean anything but some form of self-modification of the agent's own software code. Our algorithms presented in this paper, however, "abstract away" from this technical detail.

Agent Cooperation: Related to the issue of agent population (see above) is the question how an agent system behaves which has more than one *instance of* agents coexisting in the network at the same time. Here we can distinguish *independent* (or *competitive*) from *interdependent* (or *collaborative*) agent systems. In independent agent systems, the notion of "task" is defined in terms of one agent, who does his job as if he were completely alone in his world. In collaborative agent systems, on the contrary, the notion of "task" is defined on a higher level and needs the activities of more than one work-sharing agents for its completion. Whether or not an agent system is classified as "collaborative" depends thus on the level of abstraction (high or low) at which the notion of "task" is defined.

Network Nodes: We regard our underlying network (in which our agents operate) to be a network of *mono processors*. All these mono processors together achieve parallel or concurrent

computation. Thus, an individual node in the network is by itself *simple*, and not a transputer or other form of parallel or concurrent system by itself. Under these conditions, mobile agents in such a network are *programs* which are executed whenever they arrive at a processor place in the network. This leads to the question of *how shall a mono node behave* in the case that *more than one* agent (program) arrive and are present at such a node during an overlapping period of time. This problem must be solved carefully, otherwise the arrival of several agents at the same node could lead to "chaos" or, more precisely speaking, a not well defined *operational semantics*.

Pseudo-Concurrency: To clarify these operational issues (arising from the presence of more than one agent at the same network node) we could choose from a variety of conventions, for example: FIFO nodes. This would be the simplest *technical* solution (from a hardware point of view), as it is described in the standard literature on *operating systems* [6]. With this technique, every network node (processor) is equipped with a *waiting queue* for incoming mobile agents (processes) that can either be empty ([]) or non-empty ($[a_1|a_2|a_3|\ldots]$). If the waiting queue is non-empty, the waiting agents will control the node in a FIFO sequence and leave the node as soon as their job is done. (In case the queue is empty, the corresponding node is passive, or "asleep".) On a higher level of abstraction, however, one could ignore those technical details and treat the presence of two agents a_1, a_2 at the same node p by random sequentialization, denoted in terms of "interleaving" as: $(a_1|a_2)_p$.

In the following sections of this paper we will present several interesting agent specifications (implementing different types of graph relabeling) which belong to different branches of the classification scheme that we have outlined in this section.

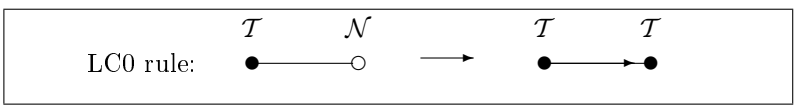
3 Basic Agent Operations

As mentioned in the Introduction, there are several classes of local relabeling rules, according to the various types of neighborhoods in which those rules can be 'anchored' (e.g. edge-shaped neighborhood, star-shaped neighborhood, ball-shaped neighborhood, and the like). In order to implement a distributed local graph relabeling system by means of mobile agents it is crucial that to every type of relabeling rule we can specify at least one corresponding agent type which is able to perform the application of such kind of relabeling rule on an underlying host graph (network). In the following sub-sections we present novel solutions for rules of type LC0 and LC1.

Performing the application of such rules, agents must also simulate the according *synchronization* mechanisms [15, 16, 14, 9] which would have to be applied in a classical agent-free implementation of a distributed graph relabeling system. Such synchronization mechanisms for relabeling an entire edge (see LC0) can be obtained by adapting the *'handshake'* algorithm of [7] in the case of a *synchronous* mobile agent system, i.e., in the case where there exists a global clock shared by all mobile agents. The algorithms presented in next sections are different from [7]'s algorithm in many points. Although they could be less efficient, they are simpler and more general since they apply for asynchronous networks, use less powerful assumptions and hold for LC1 type rules.

3.1 Blocking-Free LC0:

The LC0 rule (which is well-known to be suitable for the distributed computation of *spanning trees*) looks like this:



It replaces Nonterminal nodes by Terminal nodes, and increases thus its own applicability by every actual application. One should note that an initial labeling of the graph that will allow the application of such a rule must have at least one node with a type $\mathcal{T}$ label.

In the following we present a simple implementation which does not use any blocking in the agent code at all. Instead, mutual exclusion will be provided by the 'operating' systems of the nodes themselves which provides internal *waiting queues* to cope with the arrival of more than one agent at the same node at the same time. Thus, the here presented approach makes only minimalist requirements as far as the internal structure (code) of agents are concerned.

Our solution is 'competitive' in the terminology of above (see Introduction), because an agent A_0 is trying to apply an LC0 rule against the efforts of the other agents A_i who are trying to do the same.

3.1.1 Preliminaries, Part A: Agents

- All LC0 agents are assumed to be *identical* which means that (*a*) they carry the *same program code* and (*b*) they do *not* carry any static unique ID that could distinguish them from each other once and for all.
- In this model an agent can *not* 'continue' the execution of the program code after migrating from one node n to another node n'. Thus they must always start with their very first line of program code again, whenever they arrive at another processing place; in other words: they do not carry a persistent program pointer which would point to the current line of execution of their own program code.
- However, an agent needs *one bit* of *persistent memory* C in order to remember the context from where he came. Because of the specification of the LC0 Rule, namely $\{\mathcal{T}—\mathcal{N}\} \Longrightarrow \{\mathcal{T} \rightarrow \mathcal{T}\}$, an agent must have visited a $\mathcal{T}$ node before he may change an $\mathcal{N}$ node to $\mathcal{T}$. Coming from another $\mathcal{N}$ node the agent may *not* update an $\mathcal{N}$ node, because otherwise the agent would implement a *wrong* rule, namely $\{\mathcal{N}—\mathcal{N}\} \Longrightarrow \{\mathcal{N} \rightarrow \mathcal{T}\}$, for which there is no specification. Without any kind of persistent memory, which the agent can carry along in his 'rucksack' while traveling from node to node, the agent could not remember the type of node from where he came and could thus not correctly implement LC0[2]. At agent creation time we set $C :=$ false and update the value to 'true' as soon as the agent has found its very first LC0 context node of type $\mathcal{T}$.

These types of agents are deliberately specified minimalistically to be the most primitive and 'non-intelligent' agents we can think of; yet these primitive entities will be sufficient to implement the above-mentioned LC0 rule, if only the underlying network environment provides the following features:

3.1.2 Preliminaries, Part B: Environment

- According to the LC0 rule of above, a node possesses one out of two distinguishable types T: These are $\mathcal{T}$, respectively $\mathcal{N}$.
- A node shall also be equipped with a waiting queue for incoming agents. Because of the system being fully asynchronous, the *waiting time* of an agent in a node's waiting queue is completely arbitrary; an agent could 'vanish' in a queue for several hours as well as for just a few micro seconds.
- We assume that an agent gets *exclusive access* to the processor of a node from the beginning to the end of its agent code, while other agents are waiting in the queue until that agent has finished its task. There is *no* 'round robin' (or any other pseudo-simultaneous) processor sharing amongst a multitude of agents sitting in the same node at the same time, which means that two agents can never disturb each other while sitting in the same network node at the same time.

[2]Technically speaking, the agent must modify its own program code —like in the 'core war' game— when modifying its own persistent memory, for the agent consists of nothing else but program code.

- We assume that a node maintains *locally unique* channel names (port names) to each of its adjacent edges e_i.
- We assume that a node will be able to inform an incoming agent a_0 about local identity of the channel e_i through which that agent entered that node. This information will be stored in the 'operating system' of the node even while the according agent is waiting in the node's internal queue.
- After an LCO rule $\{\mathcal{T}—\mathcal{N}\} \Longrightarrow \{\mathcal{T} \rightarrow \mathcal{T}\}$ has been successfully applied to a node v, i.e. another branch ($\rightarrow$) of a spanning tree has been constructed and v has become part of it by changing its type from $\mathcal{N}$ to $\mathcal{T}$, v will internally mark the according channel port. A link memory L shall store this information. Note that a node can have at most *one incoming* edge in a spanning tree, such that L is either empty or it carries the name of one of the ports of its node.

3.1.3 Algorithm (Pseudo-Code)

```
PROCEDURE AGENT[C] ..... //C is persistent during migration!
BEGIN
ARRIVE @ node;
BEGIN ATOMIC SECTION

if( T(node) == 'T' ) ..... // found potential LC0 rule context
C := true;
p := getAnyPort(node); ..... // try to find node type 'N'
LEAVE(p);

if( T(node) == 'N' AND C == true ) ..... // application possible
i := getMyIncomingPort(node);
L(node) := i;
T(node) := ' T'; ..... // update accomplished
p := getAnyPort(node); ..... // try to find further work
LEAVE(p);

if( T(node) == 'N' AND C == false ) ..... // not seen T-context
p := getAnyPort(node); ..... // try to find node type ' T'
LEAVE(p);

END ATOMIC SECTION
END
```

Figure 2: LC0 implementation: code for an agent.

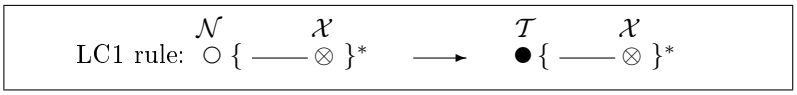
Based on the assumptions of above, the algorithm of Fig. 2 (the code of which is carried by the mobile agents), implements relabeling rules of type LC0. *Deadlock-Freeness* of the procedure is guaranteed because no kinds of blocking techniques (semaphores, etc.) are used at all. *Mutual Exclusion* of agents in one node is guaranteed by the underlying 'operating system' of that node which is assumed to provide a FIFO queue for incoming agents. *Correctness* of the spanning tree construction is guaranteed by the fact that any node can have at most one incoming link, and any agent can create at most one such links at the same time, and at most one agent can be active in the same node at the same time.

Also note that the agent's memory C is actually a *monotonous* function: As soon as the agent has found his first context node of type $\mathcal{T}$, C will switch to 'true' and will never be switched back

to 'false' again, for any node visited immediately afterward is either a $\mathcal{T}$ too, or will switch from $\mathcal{N}$ to $\mathcal{T}$ in the course of the operation. This means that our agent implementation of above is *almost state-less*, and can indeed be made completely state-less if it can be externally guaranteed that the starting place of an agent (at creation time) is a node which has type $\mathcal{T}$.

3.2 LC1 with two different Types of Agents

LC1 is the 'star' rule type that updates a single node in relation to all its neighbors. In other words, LC1 works like a *generalized* cellular automaton rule in the sense of [8]. We can thus sketch the update type of LC1 as:



whereby $\mathcal{X}$ stands for any node label in the neighborhood which will remain the same; only the center of the star is updated when the neighborhood condition is fulfilled.

In the following implementation of this rule we will use a *blocking* technique, such that two agents who wish to update neighbor nodes cannot interfere with each other. The according agents of type 'Star' will be used to implement the core of LC1.

However, whenever blocking is allowed, the resulting system is deadlock-prone. To break the symmetry of a mutual-block situation, an agent of type 'Lamport' will crawl through the web and assign priority labels wherever a mutual-block situation is detected. Consequently an area with a higher priority can be served first by the agents of type 'Star'. In the following we first present the code of the 'Star' agents, thereafter the code of 'Lamport'.

3.2.1 Preliminaries

Without loss of generality (only for the sake of intuitive description) we assume that a star center is connected by with its neighbor nodes by means of one *hyper edge*. Given a node set $\mathcal{V}$, a hyper edge is a structure $h = (v, V)$, whereby $v \in \mathcal{V}$ and $V \subseteq \mathcal{V}$. Thus an agent shall be able to use the information $h(v) \in V$ (and for any $v' \in V$, $h^-(v') = v$) for the purpose of traveling between a center of a star and its fringes (orientation). A node shall be endowed with a rich internal state, made up of the following components:

- $p \in \mathbb{N}_0 \uplus \{-1\}$ is a priority flag which will be used to solve conflicts between competing neighbor activities. (The value -1 means that this node has not yet been ranked in any priority order.)
- m is the node's main label, which can be updated as a result of any LC1 rule application.
- h is the node's hyper edge information which is used by a 'Star' agent to navigate within a star shaped neighborhood.
- $M = [m_1, \ldots, m_n]$ is a field with a buffer m_i for every neighbor node $v_i = h(v) \in V$. According to rule LC1 the center node can be updated as soon as information from all its neighbors are collected, and M will be used exactly for this purpose.

Similar to the previous example (LC0), a 'Star' agent shall possess a small, persistent runtime environment which the agent can carry from node to node during migration. The main components of this runtime environment are

- a number memory ('my-prio', init.:*nil*), such that priority considerations can be made;
- a hyper edge memory, ('my-label', init.:*nil*) such that the agent has orientation within a star-shaped neighborhood of nodes;
- a work-mode flag ('my-counter', init.:0) such that the agent can determine whether he is in the center of a star, or at the fringe of a star, or in search for another job;

- a memory ('my-memo', init.:*nil*) for reading a node's label and transporting this information back into the center of a star.

3.2.2 Agent 'Star' (Pseudo-Code)

Based on the preliminaries of above, the algorithm of Fig. 3 for the 'Star' agent should be more or less self-explanatory — note, however, that the agent code is started from the very first line of the program whenever the agent arrives at a new node, (thus: no persistent program pointer and consequently no code-continuation in the process of migration):

Basically the algorithm says: When you have collected information from all the neighbors then you must apply the LC1 rule. However, if a neighbor is prior, then you cannot collect its information and you must return to the center undone; and try again later. The priority labels are allocated by the supportive 'Lamport', agent which is described in the following.

3.2.3 Agent 'Lamport' (Pseudo-Code)

This agent is very simple, see Fig. 4 for the detailed description. However, to ensure *uniqueness* of the priority numbers, we stipulate that there be only *one* instance of 'Lamport' in the network. Because the code of 'Lamport' is only short, we can assume that 'Lamport' will work sufficiently fast to do his job across the network. Whenever 'Lamport' finds a critical node (with number 0), 'Lamport' will allocate a unique number $n > 0$ to it. This is also the reason why the 'Star' agent has to update his own priority memory whenever he comes back into the center — because 'Lamport' could have visited the center in the meantime while the 'Star' agent was in the fringe.

Because of the uniqueness of the priority numbers allocated by the 'Lamport' agent, the 'Star' agents can never deadlock, though they can temporarily protect their current neighborhoods against other 'Star' agents roaming in that area of the network.

4 A General Mobile Agent Framework for Relabeling Systems

After having presented two particular examples (LC0, LC1) in the previous section, we are now aiming for a constructive and general method of implementing *any* local graph relabeling system by means of mobile agents. In doing so, we approach a general and unified framework for local computations which is consistent with the classical theory of local computation based on graph relabeling systems.

In the rest of the paper, we consider a *k-locally generated* relabeling system R. We recall that R is called *k-locally generated* if any relabeling rule of R is entirely defined by the precondition and the relabeling of a generic ball of radius at most k. Intuitively speaking, only the labels of nodes and edges in a ball of radius k are changed. One application of this type of relabeling systems is for studying graph reduction rules and graph recognizers in a distributed and static environment. For instance, in [17] it is shows how to encode *handy reduction rules* envolving vertex (edge) deletion (addition) in a distributed environement by mean of k-locally generated relabeling systems.

Assume that we have n agents which have been scattered over the entire network. Our goal is to make the agents apply the relabeling rules given by R in a distributed way. The examples of the previous section have shown that the major challenge consists in making the agents execute the rules in an independent and concurrent way, that is, if an agent is being executing some rule in some region, then no other agent should execute a rule simultaneously on the same region — otherwise the relabeling may be wrong or ill-defined. We first present an algorithm for the case there is exactly one agent in the network ($n = 1$), thereafter we extend the solution for the more general case of many agents ($n > 1$).

```
AG DEF PERSISTENT VAR:
   my-label(init:nil), my-prio(init:nil),
   my-memo(init:nil), my-counter(init:0) ;
BEGIN
if IF (my-count = 0) AND (host-prio = −1) then
   host-prio := 0 ; ..... // mark center active
   my-prio := host-prio ;
   my-count := 1 ; ..... // prepare for work
if (my-count = 1) then
   my-label := host-hyp ; ..... // remember hyper edge
   my-prio := host-prio ;
   my-count := 2 ; ..... // prepare for fringe
   DO select neighbor N with host-M[N] = nil ;
   DO move and enqueue into [my-label → N] ;
if (my-count = 2) AND ((host-prio = −1) OR (my-prio < host-prio)) then
   my-memo := host-m ; ..... // collect info
   my-count := 3 ; ..... // prepare for center
   DO move and enqueue into [my-label⁻] ; ..... // go back
if (my-count = 2) AND (my-prio ⩾ host-prio) then
   my-memo := nil ; ..... // fringe is blocked
   my-count := 3 ; ..... // prepare for center
   DO move and enqueue into [my-label⁻] ; ..... // go back
if (my-count = 3) then
   DO update host-M ← my-memo ; ..... // bring info
   if IF (host-M contains nil) then my-count := 1 else my-count := 4
if (my-count = 4) then
   // all neighbors checked
   DO update host-m = • ; ..... // rule application in center
   host-prio := −1 ; ..... // job done
   my-count := 0 ;
   DO move away to another job ;
if (otherwise) then
   // nothing to do here my-count := 0 ;
   DO move away to another job ;
```

Figure 3: LC1 implementation: code for a 'Star' agent

```
if (host-prio = 0) then
   // found node in critical section
   host-prio := my-number ; ..... // allocate priority
   my-number := my-number +1 ;
   DO move away to another job ;
```

Figure 4: LC1 implementation: code for 'Lamport' agent

4.1 Model

In the following, we assume that each node is equipped with a 'whiteboard' where agents can read and write information under mutual exclusion. The label of a node is stored in its whiteboard. The whiteboard $WB(v)$ of a node v contains also other variables allowing agents to exchange information and to communicate together (e.g., to decide whether a node may be relabeled). More precisely, for every node v we will denote by $WB(v).c$ the couple (X, i) with X a label from set $\{\mathbf{M}, \mathbf{Locked}\}$ and i an integer value. In our general approach, we will assume that every agent has a unique identifier. In fact, if the agents (and the network) are anonymous and if $k > 2$, there exists *no deterministic* distributed algorithm in the asynchronous mobile agent model allowing to execute a k-locally generated relabeling system for *any* graph. This claim can be proved using the equivalence result of [4]. Roughly speaking, the equivalence result there says that mobile agents and message passing systems have the same power from a computability point of view[3]. Since it is well known that it is impossible to implement a k-locally generated relabeling system for any graph using messages (see e.g., [1, 15]), our claim is straightforward. For simplicity and clarity, we assume that the identifier of agent A_i (with $i \in \{1, \cdots, n\}$) is i.

4.2 Single Agent Implementation

For now we assume that we have only one agent in the network to *implement* a distributed algorithm *specified* by a local graph relabeling system. Two problems must be solved in this scenario:

- How shall the agent traverse the entire network without omitting any node?
- How does the agent recognize the neighborhood of a node in order to apply a relabeling rule on this node in that neighborhood?

The traveling problem can be solved by means of a *spanning tree*. Thus, first we make the agent construct a rooted spanning tree T of the entire network. Many spanning tree algorithms are described in the literature, and any kind of spanning tree will do (see also the next section for a Depth First Search (DFS) tree algorithm). Now, the agent can use T as a *map* for traveling across the network. For instance, from the root of T, the agent could perform a DFS-traversal of T. Whenever the agents visit a new node, he temporarily interrupts his DFS-traversal in order to apply a local relabeling rule. Thereafter the agent continues the DFS-traversal to visit another node in T. Once the entire network is traversed the agent will start a new DFS-traversal, and so on, until no further relabeling rules are applicable. This method ensures that all the nodes of the graph will be visited at some time by the agent, such that node starvation is impossible.

Now we need to describe how the agent can execute a graph relabeling rule after arrival at some node v. The idea is to make the agent "learn" the k-neighborhood of v in order to be able to check if a relabeling rule can be applied. In order to learn the node's k-neighborhood, the agent first constructs a Breadth First Spanning (BFS) tree $T_{B(v,k)}$ of the ball $B(v, k)$ rooted at v (for instance this can be done in a layered fashion). Then the agent 'collects' the entire topology of $B(v, k)$ by traversing the neighborhood tree $T_{B(v,k)}$. In case the network nodes have *unique identifiers* the learning of a node's k-neighborhood is quite straightforward. In case that no such unique node identifiers are available it is also not too difficult to let the agent himself create such identifiers for the visited nodes (e.g., when constructing the initial spanning tree T). Having "learned" the topology of $B(v, k)$, and having noticed that some relabeling rule r is applicable in the context of $B(v, k)$, the agent visits $B(v, k)$ again (using the neighborhood tree $T_{B(v,k)}$) and attaches new labels according to rule r.

In this context it is important to note that we clearly distinguish between the *application of one local rule* and the *execution of a rule-based algorithm* across the entire host-graph. For the latter purpose a spanning tree might be useful, but it might also be done as a random-walk without

[3] In other words, what can be computed by message passing can also be computed by mobile agents and vice versa.

any global spanning-tree being constructed. Within our general framework we choose to adopt a deterministic approach which seems to be easier to understand.

4.3 Multiple Agent Implementation

In the remainder of this section, we extend the previous single agent approach and describe our generic framework for implementing a k-locally generated relabeling system for any integers $k, n \geqslant 1$.

4.3.1 Initializing and Traveling the Network

The key idea of our approach is to partition the graph G into a set of n regions $(G_i)_{i\in\{1,\ldots,n\}}$ and to assign a region G_i to every agent A_i. Each agent then applies the applicable relabeling rules in its own region, independent of other agents. Thereby we have to consider how the regions are assigned to the agents, and how the application of rules is managed at the borderline between two regions.

Without loss of generality, we can assume that a node contains no more than one agent at the beginning. In fact, if this assumption is not satisfied then the agents with the lowest identifiers travel the network searching for a new departure node. If no free node is found (which can be detected by performing a DFS-traversal of the network), the agent searching for a departure node vanishes (it dies).

At the beginning, each agent executes algorithm of Fig. 5. This algorithm is an adaptation of the classical DFS-tree algorithm for a mobile agent system. For simplicity, we have omitted the details showing how an agent marks a node or an edge (which is straightforward using the above-mentioned 'whiteboards' of the nodes). After termination, every agent has computed a spanning tree denoted by T_{G_i}. In other words, the region G_i is defined to be the subgraph of G induced by the tree constructed by agent A_i. Note that it might possibly happen that an agent

```
mark the initial departure node as root of T_{G_i};
find a new un-explored node neighboring the current node; .... // search for a non-marked
neighbor
if a new un-explored node v is found then
    mark the new explored node v as part of sub-graph G_i;
    update the rooted tree T_{G_i} ;
    continue the exploration (DFS-traversal) from node v (go to line 2);
else
    move back to the previous parent node u using the rooted tree T_{G_i} ;
    if node u is the root and all outgoing edges of u were explored then
        stop the exploration;
        T_{G_i} is ready;
    else
        continue the exploration from node u (go to line 2);
```

Figure 5: Algorithm InitNetwork for constructing a region G_i: high level code for agent number i

fails to compute a tree. In this case, the agent should vanish and the actual number of agents in the network is decreased. Moreover, the case of a unique agent corresponds to the case where there is only one region (the whole graph). However, the algorithm InitNetwork of Figure 5 allows to construct a spanning forest of G even when the agents do not have unique identifiers which could be of independent interest. We also remark that algorithm InitAgent can be easily encoded in a high level way using rules type LC0 or LC1.

4.3.2 Executing the local relabeling

Now that the regions $(G_i)_{i\in\{1,\ldots,n\}}$ are constructed, every agent is responsible for executing relabeling rules in its own region. In the interior of a region, the rules could be executed like specified

by our single agent implementation. However, some conflicts may occur at the borderline between two adjacent regions. The main purpose of the following paragraphs is to show how to deal with these conflicts. First, each agent A_i constructs a BFS-spanning tree $T_{B_i(v,k)}$ of $B(v,k)$ for each node $v \in G_i$ (note that $T_{B_i(v,k)}$ may contain nodes in another region $G_j \neq G_i$). Then, each agent A_i traverses G_i in a DFS fashion using T_{G_i}. When agent A_i is at a node $v \in G_i$, it tries to apply a rule using the following four phase strategy:

1. In the first phase, agent A_i traverses $T_{B_i(v,k)}$ and collects the labels of $B(v,k)$ in order to check if a rule can be applied. If no rule can be applied, then A_i continues the traversal of T_{G_i}. Otherwise, A_i goes to the second step.

2. In the second phase, agent A_i traverses $T_{B_i(v,k)}$ and tries to mark the $WB(w).c$ field of all nodes $w \in B(v,k)$ using an extra label **(M,i)** as following:

   - If a node $w \in B(v,k)$ is marked with label **(locked,j)** for any $j \neq i$, then agent A_i waits until node w is unlocked by agent A_j (see next phase).
   - If a node $w \in B(v,k)$ is already marked **(M,j)** by another agent $A_j \neq A_i$, then there are two cases:
     - If $i < j$ then A_i unmarks all the nodes he has already marked and continues the traversal of T_{G_i} (go to step 1).
     - Otherwise, A_i marks w with label **(M,i)** and continues the traversal of $T_{B_i(v,k)}$ (exploration of $B(v,k)$).

3. In the third phase, if A_i succeeds in marking all the nodes of $B(v,k)$ with **(M,i)**, then it traverses $T_{B_i(v,k)}$ once again in order to lock all the nodes in $B(v,k)$ by marking them with the extra label **(locked,i)**, i.e., the neighborhood ball is ready to be relabelled according to a rule. If the label of at least one node $w \in B(v,k)$ is not **(M,i)** then A_i unmarks all nodes marked with label **(M,i)** or those locked with label **(locked,i)** and continues the DFS-traversal of T_{G_i} (in other words, it reinitializes the $WB(w).c$ field of nodes $w \in B(v,k)$ marked by himself and goes to phase 1). When an agent A_i traverses $T_{B_i(v,k)}$ in order to lock the nodes, it also collects the topology of $B(v,k)$ at the same time in order to prepare executing a rule which avoids to make another traversal.

4. The fourth phase is executed if and only if the agent A_i has succeeded locking all nodes in $B(v,k)$. Hence, the agent traverses $B(v,k)$ for the fourth time in order to apply a rule. At the same time, it unlocks the nodes in $B(v,k)$. Finally, the agent continues the DFS-traversal of T_{G_i} and starts another cycle in the first phase again.

Note that an agent executes the second phase if and only if it finds a rule to execute after the first traversal in the first phase. Nevertheless, it may happen that in the fourth phase, no rule can be applied since the label of some nodes in $B(v,k)$ may change. In addition, a node w marked **(locked,i)** by an agent A_i can be updated *only* by agent A_i himself. In other words, if an agent j wants to mark node w, then he must wait until agent A_i unmarks it. Moreover: *several* traversals of $B(v,k)$ are needed only in the case that v belongs to the frontier of some other regions, i.e., there exists some $j \neq i$ such that $B(v,k) \cap G_j \neq \varnothing$. Based on this observation the agents can make further pre-computations in order to mark the nodes at their frontier and thus they can avoid traversing the ball $B(v,k)$ several times if node v does not belong to the frontier.

4.3.3 Correctness analysis

First, we remark that the relabeling done by an agent A_i locally on a ball $B(v,k)$ is correct. In fact, the relabeling of a ball is always done according to a valid relabeling rule described by the relabeling system given in input. Furthermore, whenever an agent is being relabeling a node w (or an edge) of a ball $B(v,k)$ in phase 4, *no* other agent could be relabeling a node w' in $B(v,k)$ at the same time. The latter property is easily proved by remarking that: an agent A_i begins

relabeling a ball $B(v,k)$ in phase 4 if and only if the entire ball $B(v,k)$ has been marked with label **(locked,i)**, and nodes marked with label **(locked,i)** cannot be unlocked by other agents.

Now, it remains the prove that the relabeling is globally correct.

Lemma 1 *Our framework is deadlock free, i.e., an agent cannot be blocked infinitely often in any node.*

Proof. The only case where an agent A_i may wait at a node w is when w is marked **(locked,j)** with $i \neq j$ (phase 2). In other words, the agent A_i may wait if the node w was locked by another agent A_j. From the description of phase 3 and 4, we are sure that node w will be unlocked by agent A_j, thus avoiding deadlocks. In fact, since node w was locked by agent A_j, then this means that agent A_j has succeeded into applying phase 2, i.e., it has marked all nodes in the corresponding $B(v,k)$ ball with label **(M,j)**. Thus, agent A_j is applying either phase 3 or phase 4, while agent A_i is waiting in node w. From the description of phase 3 and 4, agent A_j is never blocked and it always unlocks the nodes in $B(v,k)$. □

The deadlock free property stated in the previous lemma is not sufficient to prove the correctness of our framework. In fact, it only ensures that the agents will not be blocked waiting for each others, but it does not ensure that the relabeling rules will be effectively applied. In the following, we argue that if a rule r has to be executed in any node v in order to continue the relabeling of the graph, then there exists an agent A_i that succeeds in relabeling $B(v,k)$ within a finite time according to r.

Observe that in the first stage of our framework, an agent at node v always verifies whether a rule can be applied. Thus if an agent starts marking the nodes of some ball $B(v,k)$, then this means that some rule can be applied in $B(v,k)$. Now, observe that if an agent A_i fails preparing a ball $B(v,k)$ in phase 3 i.e., it fails locking the nodes of $B(v,k)$, then there must exist another agent A_j applying a rule in a ball $B(w,k)$ such that $j > i$ and $B(v,k) \cup B(w,k) \neq \varnothing$. The agent A_j may also fail preparing ball $B(w,k)$ because of a neighboring agent A_ℓ with a higher identifier. Using the 'deadlock free' property we are sure that among all agents who passe the first phase, at least the agent having the highest identifier will succeed applying a rule.

Now, suppose that some rule r has to be executed in some ball $B(v,k)$ in order to continue the relabeling of the graph, that is no other rule can be applied in any other node before rule r is applied in $B(v,k)$. Then, the agent A_i in the region G_i containing v will be the only agent who passes the first stage of our framework and will not be disturbed by other neighboring agents when preparing the ball $B(v,k)$ in phase 3.

Thus, from the discussion above, the following is straightforward:

Theorem 1 *Our generic framework is correct, that is:*

- *there is no deadlock,*
- *there is no rule starvation,*
- *the relabeling performed by agents is correct.*

5 Conclusion

In this paper, we have argued that mobile agent paradigm is well suitable for implementing distributed algorithms based on relabeling systems. By doing so, we are approaching a more comprehensive theory of distributed algorithms in which *(i)* relabeling systems are considered as a formal tool-box for designing algorithms and *(ii)* our mobile agent algorithms are considered as a practical tool-box for implementing them. Consequently, the mobile agent algorithms given in this paper can be considered as the key to a complete solution for designing, proving and implementing distributed algorithms using relabeling systems.

It is easy to see that our algorithms could also be adapted to message passing systems by using *tokens* to simulate agents. Nevertheless, we think that a practical implementation or just a detailed description would be rather complicated compared with our algorithms since nodes will have to manage complicated data structures and message types. Therefore we believe that mobile agents will play an important role into bringing a new theoretical and a practical approach to some classical distributed problems. Indeed, the abstraction provided by mobile agents allows both an encapsulation and a modularization of distributed computations over a network, which should lead to feasible solutions.

References

[1] D. Angluin. Local and global properties in networks of processors. In 12^{th} *Symposium on theory of computing*, pages 82–93, 1980.

[2] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. *Technical Report, LaBRI*, RR-1271-02, 2002.

[3] M. Bauderon and M. Mosbah. A unified framework for designing, implementing and visualizing distributed algorithms. In P. Bottoni and M. Minas, editors, *Graph Transformation and Visual Modeling Techniques (GT-VMT'02)*, volume 72 of *Electronic Notes in Theoretical Computer Science*, Barcelona, Spain, 2003.

[4] J. Chalopin, E. Godard, Y. Métivier, and R. Ossamy. Mobile agent algorithms versus message passing algorithms. In *10th International Conference On Principles Of Distributed Systems (OPODIS'06)*, volume 4305 of *Lecture notes in computer science*, pages 187–201. Springer, 2006.

[5] J. Chalopin and D. Paulusma. Graph labelings derived from models in distributed computing. In 32^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science (WG'06)*, volume 4271 of *Lecture Notes in Computer Science*, pages 301–312. Springer, 2006.

[6] H. M. Deitel, P. J. Deitel, and D. R. Choffnes. *Operating Systems (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[7] B. Derbel. Efficient distributed handshake using mobile agents. In 8^{th} *International Conference on Distributed Computing and Networking (ICDCN'06)*, pages 294–305. LNCS 4308, 2006.

[8] S. Gruner. Mobile agent systems and cellular automata. Technical Report 1400-06, LaBRI - University of Bordeaux 1, 2006.

[9] A. E. Hibaoui, Y. Métivier, J. Robson, N. Saheb-Djahromi, and A. Zemmari. Analysis of a randomized dynamic timetable handshake algorithm. Technical Report 1402-06, LaBRI, 2006.

[10] P. Knirsch and H.-J. Kreowski. A note on modeling agent systems by graph transformation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, volume 1779 of *Lecture notes in computer science*, pages 79–86. Springer, 2000.

[11] I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.

[12] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[13] I. Litovsky, Y. Métivier, and W. Zielonka. On the Recognition of Families of Graphs with Local Computations. *Information and Computation*, 118(1):110–119, 1995.

[14] Y. Métivier, N. Saheb, and A. Zemmari. Randomized rendez vous. In *Mathematics and computer science: Algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194. Birkhäuser, 2000.

[15] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Information Processing Letters*, 82:313–120, 2002.

[16] Y. Métivier, N. Saheb, and A. Zemmari. Analysis of a randomized rendez vous algorithm. *Information and Computation*, 184:109–128, 2003.

[17] R. Ossamy. An algorithmic and computational approach to local computations. In *Ph. D Thesis.* LaBRI, University of Boredaux 1, France, Dec. 2005.

Centre de recherche INRIA Futurs : Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399