Fully-Distributed Debugging and Visualization of Distributed Systems in Anonymous Networks

Cédric Aguerre, Thomas Morsellino, Mohamed Mosbah

FULLY-DISTRIBUTED DEBUGGING AND VISUALIZATION OF DISTRIBUTED SYSTEMS IN ANONYMOUS NETWORKS

Keywords: Distributed algorithm, Visualization, Debugging, Anonymous network, Snapshot, Global predicate evaluation.

Abstract: The debugging of distributed algorithms is a major challenge which greatly benefits from the help of an interactive and informative human-computer interface. In this paper we present ViSiDiA, a platform for the visualization, simulation and debugging of distributed algorithms. Our approach respects real-life constraints such as process anonymity and privacy, network synchronicity. We propose a new fully-distributed method for the debugging and monitoring of distributed systems, based on the computation of global states and global predicates from local information in anonymous and asynchronous networks. We show how the debug information can be visualized concurrently with the algorithm execution.

1 INTRODUCTION

Many complex information systems require a large number of servers and machines, interconnected on either local or remote networks. Such distributed applications aim at making several processes collaborate to the execution of a same task. Problems are raised concerning concurrent access to resources, critical failure detection, or even process communication strategy. The generalization of heterogeneous and large networks thus involves substantial studies in the distributed algorithm field.

The design and validation of distributed applications depend on the analysis and understanding of underlying algorithms that must be proved, implemented, debugged and tested. The emergence of simulation softwares is an important step towards this objective. A visualization of running algorithms, with the ordered observation of data transfers and property changes over the network, is of great help. A quantification of all events occurring over the network leads to the evaluation of distributed algorithms in terms of complexity or global performance.

Among services required in a distributed system, monitoring the execution of every process in a distributed way can be a critical issue but is useful to evaluate network properties or to debug the entire execution. By definition, in a fully distributed asynchronous system there is no global clock and thus it is not possible to take an instant picture of the whole system in a distributed manner. Every process knows neither the states of other processes nor the state of any communication channel. This impossibility is reinforced by the analysis proposed by Guerraoui and Ruppert (Guerraoui and Ruppert, 2005) in which they consider that a vast majority of papers on distributed computing assume that processes have unique identifiers or do not want to share their private data.

Hence, there exist three important challenges in developing a visual interface for distributed algorithm debugging. First a debugger needs snapshots of the system, which are composed of the state of each process and the state of each communication channel. How can we compute such a global network snapshot using only local information processes exchange? Second, what if processes do not have unique identifiers or do not wish to divulge them for privacy reasons? Finally, these debug information must be available along with the algorithm execution. How can we obtain an efficient computation and visualization of debug information along with simulation events display?

To address the aforementioned challenges, this paper presents a complete, stable framework named ViSiDiA for the simulation, visualization and debugging of distributed algorithms in anonymous networks. Our approach relies on both the development of a new fully-distributed debugging technique with solid theoretical fundaments, and the design of an interactive simulation viewer offering a high level, simple interface to implement and test new algorithms. This paper has the following contributions.

- We propose an original method to debug the execution of distributed algorithms with which global information can be computed from local data collection in the context of anonymous networks.
- Debug information is visualized along with the execution of asynchronous distributed algorithm. The user can observe and interact with the network.

- We present a platform for creating, visualizing and simulating distributed networks. This is of particular interest when testing and thus debugging new algorithms.
- We incorporate a visual debugger to this platform, and extend its Application Programming Interface (API, for short) to easily design new distributed algorithms.

The rest of this paper is organized as follows. First, we present our model and recall in Section 2 notions we use in our approach. Section 3 describes and assesses our prototype used to monitor the behavior of a distributed system. The design of debugging algorithms is detailed and explained in Section 4. Finally, we discuss related work in Section 5 before drawing conclusions and sketching current and future work in Section 6.

2 PRELIMINARIES

Model. Our model is the usual asynchronous message passing model (Tel, 2000; Yamashita and Kameda, 1996). A network is represented by a simple connected graph $G = (V(G), E(G)) = (V, E)$ where vertices correspond to processes and edges to direct communication links. The state of each process is represented by a label $\lambda(v)$ associated to the corresponding vertex $v \in V(G)$; we denote by $\mathbf{G} = (G, \lambda)$ such a labelled graph. We assume the network to be anonymous: the identities of processors are not necessarily unique or for privacy reasons, processes do not share their identities during computation steps.

We assume that each process can distinguish the different edges that are incident to it, i.e., for each $u \in V(G)$ there exists a bijection δ_u between the neighbors of u in G and $[1, \deg_G(u)]$. We will denote by δ the set of functions $\{\delta_u \mid u \in V(G)\}$. The numbers associated by each vertex to its neighbors are called *door-numbers* (also called *port-numbers*) and δ is called a *door-numbering* of G. We will denote by $(\mathbf{G}, \delta)$ the labelled graph $\mathbf{G}$ with the door-numbering δ.

Each process in the network represents an entity that is capable of performing computation steps, exchanging (sending/receiving) messages with its neighbors via the corresponding doors. We consider asynchronous systems, i.e., no global time is available and each computation may take an unpredictable (but finite) amount of time. Note that we consider only reliable systems: there are no message loss or duplication. We also assume that the channels are FIFO (First In First Out), i.e. for each channel, the messages are delivered in the order they have been sent. In this model, a distributed algorithm is given by a local algorithm that all processes should execute. A local algorithm consists of a sequence of computation steps interspersed with instructions to send and to receive messages.

Snapshots and global predicates. As explained by Tel (Tel, 2000) (p. 335-336), the construction of snapshots is motivated by:

- it can be involved in debugging distributed algorithms,
- the evaluation of global predicates (properties which remain true as soon as they are verified) of the distributed system,
- if the system crashes (due to a failure of a component), it may be restarted from the last known snapshot (and not from the initial configuration).

A *consistent snapshot* of a distributed system is a global state of the distributed system or a global state that the system could have reached. Since the seminal paper of Chandy and Lamport (Chandy and Lamport, 1985) which presents an algorithm to compute a consistent snapshot, many papers give such algorithms according to the model of the distributed system. They assume that processes have unique identifiers and/or there is exactly one initiator. Many papers give also specific algorithms to detect some specific predicates that hold in a system such as termination or deadlock (Mattern, 1987; Bracha and Toueg, 1987; Marzullo and Sabel, 1994; Kshemkalyani and Wu, 2007; Kshemkalyani, 2010). Among well-known global predicates of distributed systems detected with snapshots, one can also consider loss of tokens and garbage collection (see (Tel, 2000; Santoro, 2007; Kshemkalyani and Singhal, 2008)).

The main motivation of the global predicate evaluation (GPE, for short) problem is induced by the need to react against particular situations which can occur in distributed systems. As an example, consider situations of natural disasters, such as earthquakes or floods. Spatial areas must be covered with a large number of sensors to assist rescue teams. The spontaneous and distributed features of such a network make the monitoring of what is currently happening an issue. For instance, one would like to check whether the water level is under a specified threshold for each sensor. When the water is lower than the threshold at one sensor, it verifies if the condition also holds at its neighbors and the neighbors of its neighbors and so on. If a sensor have detected a problem, a domino-effect can occur, i.e., sensors with no problem become wrong (flood propagation) and the monitoring has to be done again from the beginning.

As far as we know, no solutions have been proposed for visualization and debug in anonymous networks without any knowledge on the underlying network such as processes identifiers or spanning trees. A distributed approach to the GPE problem must be considered and addressed for which the only possible assumption is the number of processes (sensors) that have been deployed. Many notions and algorithms concerning snapshots and global predicates evaluation can be found in (Kshemkalyani and Singhal, 2008).

3 THE VISIDIA PLAFTORM

ViSiDiA* (Visualization and Simulation of Distributed Algorithms) is a tool aiming at simulating and visualizing the execution of distributed algorithms (Bauderon et al., 2001; Derbel and Mosbah, 2003; Bauderon and Mosbah, 2003). ViSiDiA is used as an educational and research utility, and has been developped and validated by various students and researchers. We have recently added value to ViSiDiA by incorporating it into a web interface, by making it consider both local and remote client/server systems, and by making it platform independent. For these developments, we have entirely redesigned and reimplemented ViSiDiA, although preserving its original concepts.

We here present this new design, with which our fully-distributed debugger is developed. First, a new object-oriented architecture has been defined to make our platform robust and extensible. Second, the Graphical User Interface (GUI) has been relooked to gain clarity, to be compatible with a client-server system and used into most of web browsers. Third, we have incorporated various standard distributed models such as message passing, mobile agents, graph relabeling rules, sensors, considering both synchronous and asynchronous systems. Finally, all these concepts have been made accessible in an API allowing users to implement new distributed algorithms and test them on ViSiDiA. We here focus on the visual components for executing and debugging algorithms on ViSiDiA.

3.1 System architecture

We adopt the classic Model-View-Controller pattern as a logical part of ViSiDiA implementation. The model is the distributed network, the view is a GUI, and the controller is a simulator of distributed algorithm execution. This simulation console receives

*http://visidia.labri.fr

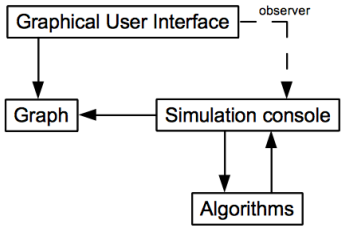

Figure 1: Structural and logical organization of ViSiDiA. Plain arrows represent direct associations (both GUI and console have access to the graph; console and algorithms can communicate and call each other). The dashed arrow is an indirect association; the GUI observes the console (e.g. to update the display) but the console does not directly transmit information to the GUI.

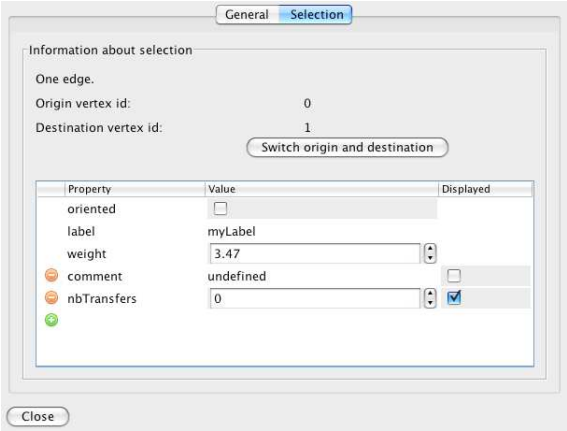

Figure 2: Example of setting an edge properties. Three properties are specific to edges and cannot be removed (oriented, label, weight). The user has added two other properties (comment, nbTransfers) which are both displayable, but only nbTransfers is displayed on the graph.

events relative to user actions from the GUI, and manages update and synchronization events with both model and view. This pattern is illustrated by Figure 1.

ViSiDiA contains an API to implement distributed algorithms thanks to a set a simple primitives, such as node/edge states and communication strategies between processes. This API is linked to the simulation core (including the network graph and a simulation console), on top of which the GUI is built. This forms a 3-layers system which defines the structural part of the application.

3.2 Network graph and processes

ViSiDiA contains a graph editor used to define distributed networks. By the means of mouse clicks, the user can easily perform standard operations on nodes and edges: add, delete, duplicate, move, connect.

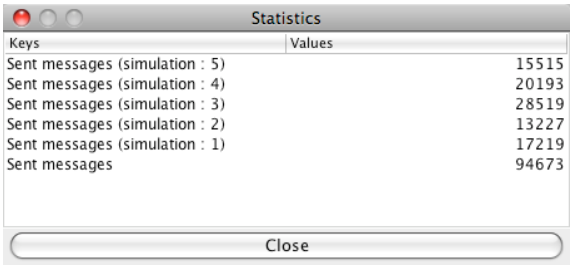

Figure 4: Some statistics on number of sent messages over 5 cloned simulations executed in a raw.

Undo/redo operations, as well as graph input/output, are also available. Nodes and edges have specific properties, such as labels or weights. The user can adjust these predefined properties and add new ones, deciding if they are displayed or not, through a simple dialog window (Figure 2).

From a given graph, it is then possible to run one or more simulations, for example changing initial settings and comparing the final results obtained on the same graph. The user can interact with the network (e.g. changing some properties), whilst the simulation executes.

Each graph node corresponds to a process, and graph edges represent communication channels between processes on which messages transit. A process is implemented as an autonomous thread associated to a local copy of the simulated algorithm. The reason for this copy is that as the network is anonymous, in our approach we must run the same algorithm on every node. As processes have a very restricted knowledge of their neighborhood, they mainly operate on their state and send messages on communication channels (through doors). These operations are visualized using colors, line styles and text animations (Figure 3).

The user can select displayable messages according to their types. Since many information may be displayed at the same time, the simulation can also be paused or recorded, and its speed adjusted. Real-time statistics about number of delivered messages or number of property changes are computed and can be displayed during the simulation. A simulation can be automatically run several time, and the various statistics can be compared over simulations (Figure 4).

3.3 Sending and receiving messages

A process can send (resp. receive) messages through any door but no information is available on the receiver (resp. sender) identity.
As a consequence, processes define messages to send, then ask the simulation console to deliver them to the appropriate recipient. A message is delivered when

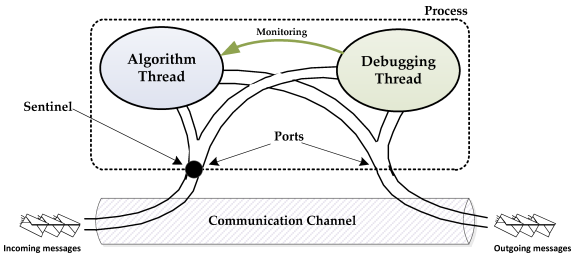

Figure 5: Implemented architecture within ViSiDiA. Each process contains an algorithm thread and a debugging thread. Once messages are received, the sentinel dispatches them to the threads according their types. Note that each message is always considered by the debugging thread to compute the snapshot and the GPE algorithms.

the console has pushed it on the receiver message queue. According to our model, reception is FIFO.

3.4 Simulation console and events scheduling

The console if responsible for scheduling all operations on the network, including message delivery. These operations are self-executable commands managed by the console using a event/acknowledgement system. When a process requires for a new command to be treated, the console first generates an event and locks it until the command is completely processed. The console then asks for the command to execute and waits for an acknowledgment which indicates that the command has terminated. The lock is finally released and a new command can be treated. This system is used to synchronize simulation events and display on the GUI, and guaranty events priority and scheduling agreements.

3.5 Monitoring and debugging feature

Any distributed system may suffer from any local failures, then it might crash. It can be the case if an algorithm suffers from a bug or if a process is disconnected from the network. Our distributed system must be robust enough to react to any local failures, blocking failure propagation to the entire system. For instance, we want our debugger to detect if a process is *deadlocked* and to inform other processes which are still waiting for its termination. As a consequence we cannot use the mono-thread per process solution as a support for debugging algorithm execution.

From our assumptions, we adopt a solution which is not coupled with the thread that executes the underlying algorithm. For a sake of simplicity, we add another thread to each process (see Figure 5). This thread deals with the debugging algorithm. A process

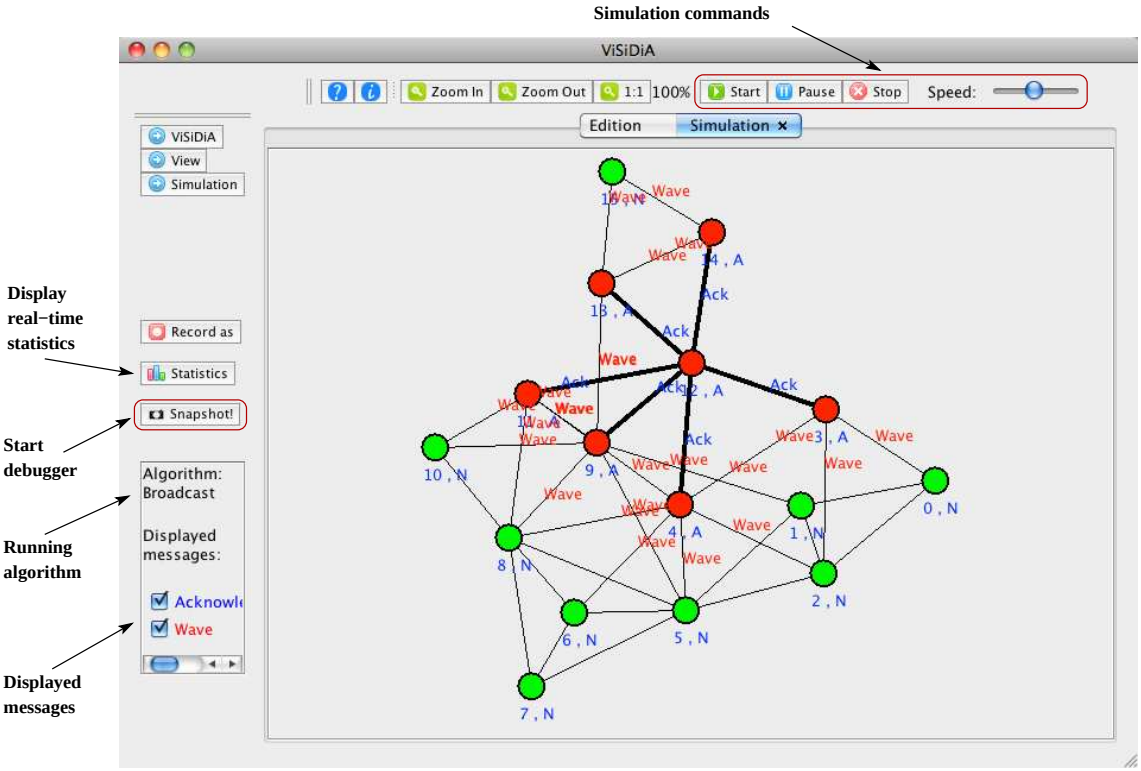

Figure 3: Overview of the visual interface. Settings are accessible on the top and left panels. The distributed network is displayed while an algorithm executes. The user can observe changes on node and edge states, as well as transiting messages. A button to launch the debugger is available on the left panel.

is endowed with a sentinel that checks every incoming message. Each received message that belongs to the underlying algorithm is sent to both the debug and the algorithm threads. Debugging messages are only sent to the debugging thread. Thus, we ensure that if the execution of the underlying algorithm crashes, the debug thread remains alive and still monitor the execution.

The GUI of ViSiDiA is upgraded with two new functionalities. We add a button in the simulation panel that launches a snapshot computation during the execution of the underlying algorithm and we develop a new panel that appears once the snapshot computation is over. This panel sums up the output of snapshot computation in one tab and presents the results of the global predicates evaluation in another tab. The first tab consists of a tree of the local snapshot of each process in which incoming channels states and specified variables values are listed (Figure 6).

3.6 Extension of the API and case study

In order to monitor the value of specified variables and test if a global predicate occurs in the system we also extend the existing API. We present how to use the debugging feature with our extension of the ViSiDiA API. Our case study is based on a simple distributed broadcast algorithm: each process broadcasts each incoming message to every neighbors except the message sender (Algorithm 1). At the end of an execution of this algorithm, a spanning tree is computed. A frequent designing error comes from the fact that acknowledgment messages are not sent back to sender or are sent to all neighbors. Thus, it results in the end of the broadcast and therefore the end of the spanning tree computation. Hence, in this example, a developer would like to know the evolution of the process state (e.g. the label value) and the exchanged messages (e.g. the content of each channels). A naive approach could be to write in the standard output (or in a file per process) the local snapshot of each process, or each process could send its local snapshot to the simulator to be displayed within the visualization software. But as explained throughout this paper, since the broadcast algorithm is asynchronous one cannot rely on this debugging method. Our distributed debugging mechanism reliably addresses these problems. It is enabled by using the *registerVariable()* method. This method is sufficient to detect above typical errors. Once the snapshot computation is terminated, even if the broad-

Figure 6: Screenshots from ViSiDiA that show the result of a snapshot taken during an execution of the broadcast algorithm. (a) and (b): monitoring result of the label of the process 2 with the value of its label and the state of its incoming channels. (c): list of GPE such as termination and token loss.

cast failed to reach all processes, the ViSiDiA panel (see Figure 6(b) and (c)) shows the state of every process and its corresponding incoming channels. Thus, designing errors could be detected.

We also provide a set of methods to apply a GPE. One of these methods allows to specify that the underlying algorithm is terminated (*setTerminated()* method). At the end of the snapshot computation if the underlying algorithm is terminated then it will be notified in the ViSiDiA panel (Figure 6(c)). Furthermore, we allow developers to create their own global predicate (e.g. flood propagation detection). They only have to create this new global predicate and use the *addGlobalPredicate()* method, to inform the GPE algorithm that it has to check this predicate for every process. Note that more than one predicate can be evaluated at the same time during algorithm execution.

4 DEBUGGING DISTRIBUTED ALGORITHMS

This section presents the corresponding algorithms we design (Chalopin et al.,) and add to ViSiDiA to address the global snapshot and the global predicate evaluation problems.

We give algorithms based on the composition of an algorithm by Szymanski, Shy, and Prywes (Szymanski et al., 1985) with the Chandy-Lamport (Chandy and Lamport, 1985) algorithm which enable each process to detect an instant where all processes have obtained their local snapshot and to evaluate global predicates anonymously.

```
begin
    static Message wave = new Message("Wave", true);
    static Message ack = new Message("Acknowledgment", true,
    java.awt.Color.blue);
    int arity = getArity();
    String label = getProperty("label");
    registerVariable("label start", label);
    addGlobalPredicate(sp);
    if label.compareTo("A") == 0 then
        for neighbor = 0 to arity-1 do
            sendTo(neighbor, Broadcast.wave);
    else
        Door door = receiveMessage();
        int doorNum = door.getNum();
        sendTo(doorNum, Broadcast.ack);
        putProperty("label", new String("A"));
        registerVariable("label in progress", label);
        setDoorState(new MarkedState(true), doorNum);
        for neighbor = 0 to arity-1 do
            if neighbor != doorNum then
                sendTo(neighbor, Broadcast.wave);
    registerVariable("label end", label);
    setTerminated(true);
end
```

Algorithm 1: Example of a broadcast algorithm written in Java using the API of ViSiDiA. In black appears the algorithm as written without any debug procedure. In blue, the only 5 lines of code needed to debug this algorithm on the visual interface.

4.1 The Chandy-Lamport snapshot algorithm

The aim of a snapshot algorithm is to construct a system configuration defined by the state of each process and the state of each channel.

This section presents the Chandy-Lamport snapshot algorithm (Chandy and Lamport, 1985); it is pre-

sented as Algorithm 2. Each process p is equipped with:

- a boolean variable $taken_p$ which is initialized at $false$, it indicates if the process p has already recorded its state;
- a boolean variable $local\text{-}snapshot_p$ initialized at $false$, it indicates if the process p has recorded its state and the state of incoming channels;
- a multiset of messages $M_{p,i}$, initially $M_{p,i} = \emptyset$, for each incoming channel i of p.

We assume that Algorithm 2 is initiated by at least one process which: saves its state, sends a marker on each outcoming door and for each incoming door memorizes messages which arrive until it receives a marker through this door. When a process receives for the first time a marker, it does the same thing that an initiator; the incoming channel by which it receives for the first time a marker is set as empty.

I*nit-CL*$_p$: {To initiate the algorithm by at least one process p such that $taken_p = false$}
begin
  record(state(p)) ;
  $taken_p := true$;
  send$< mkr >$ to each neighbor of p;
  For each door i the process p records messages which arrive via i
end

R-*CL*$_p$: {A marker has arrived at p via door j}
begin
  receive$< mkr >$;
  mark door j;
  if *not* $taken_p$ **then**
    $taken_p := true$;
    record(state(p)) ;
    send$< mkr >$ via each door;
    For each door $i \neq j$ the process p records messages which arrive via i in $M_{p,i}$
  else
    The process p stops to record messages from the channel j of p;
    record($M_{p,j}$)
  if *p has received a marker via all incoming channels* **then**
    $local\text{-}snapshot_p := true$
end

Algorithm 2: The Chandy-Lamport snapshot algorithm.

If we consider an execution of the Chandy-Lamport algorithm we obtain a consistent snapshot within finite time after its initialization by at least one process (see (Tel, 2000) Theorem 10.7). In particular:

Fact 1. *Within finite time after the initialization of the Chandy-Lamport algorithm, each process p has computed its local snapshot (*$local\text{-}snapshot_p = true$*).*

Once the computation of local snapshots is completed (for each process p the boolean $local\text{-}snapshot_p$ becomes true), the knowledge of the snapshot is fully distributed over the system. The next question is "how to exploit this distributed knowledge?"

A first answer is obtained by the construction of the global state of the system centralized on a process. As is explained by Raynal (Raynal, 1988): *Providing an algorithm for the calculation of a global state is a basic problem in distributed systems.* Several assumptions can be done to obtain a global state: exactly one initiator for the Chandy-Lamport algorithm, there exists a global clock, processes have unique identifiers, global colors associated to each computation of a global state.

A global clock can be simulated by local logical clocks (Raynal, 1988), nevertheless it does not enable iterated computations of snapshots.

Another way to exploit local knowledge is based on wave algorithms: a message is passed to each process by a single initiator following the topology of the network or a virtual topology (ring, tree, complete graph, ...), see (Matocha and Camp, 1998).

These solutions are not available in the context of anonymous networks with no distinguished processes and no particular knowledge on the topology.

4.2 Termination detection of the Chandy-Lamport snapshot algorithm

A first step to answer the above requirement is given by the termination detection of the computation of all local snapshots. More precisely, it requires that all processes certify, in a finite computation, that they have completed the computation of the local snapshot. The algorithm by Szymanski, Shy, and Prywes (the SSP algorithm for short) (Szymanski et al., 1985) does this for a region of pre-specified diameter; the assumption is necessary that an upper bound of the diameter of the entire network is known by each process (note that the network size also suffices). In the sequel this upper bound is denoted by β and we assume that each process knows it. This knowledge is available at any process in the ViSiDiA API.

We consider a distributed algorithm which terminates when all processes reach their local termination

conditions. In particular in Algorithm 2, each process is able to determine only its own termination condition. SSP's algorithm detects an instant in which the entire computation is achieved.

Let G be a graph, to each process p is associated a predicate $P(p)$ and an integer $a(p)$. Initially $P(p)$ is false and $a(p)$ is equal to -1. Transformations of the value of $a(p)$ are defined by the following rules.

Each local computation acts on the integer $a(p_0)$ associated to the process p_0; the new value of $a(p_0)$ depends on values associated to neighbors of p_0. More precisely, let p_0 be a process and let $\{p_1, ..., p_d\}$ the set of processes adjacent to p_0.

- If $P(p_0) = false$ then $a(p_0) = -1$;
- if $P(p_0) = true$ then $a(p_0) = 1 + Min\{a(p_k) \mid 1 \leq k \leq d\}$.

We assume that for each process p the value of $P(p)$ eventually becomes true and remains true thereafter.

To apply the SSP algorithm, each process is endowed with three variables:

- $a(p) \in \mathbb{Z}$ is a counter and initially $a(p) = -1$, $a(p)$ represents the distance up to which all processes have the predicate true;
- $A(p) \in \mathcal{P}_{\text{fin}}(\mathbb{N} \times \mathbb{Z})^{\dagger}$ encodes the information p has about the values of $a(q)$ for each neighbor q. Initially, $A(p) = \{(i,-1) \mid i \in [1, \deg_G(p)]\}$.

A precise description is given in Algorithm 3. From this algorithm, we can compose the application of the Chandy-Lamport algorithm and the SSP algorithm to enable each process to detect an instant where all processes have completed the computation of their local snapshot. In some sense, $a(p)$ represents the distance up to which all processes have completed the computation of the local snapshot.

If a process has completed the computation of its local snapshot then it changes the value of $a(p)$ to 0 and it informs its neighbors. When a process p receives a value $a(q)$ for some neighbor q via the door i then it substitutes the new value $(i, a(q))$ to the old value (i,x) in $A(p)$. Finally, p computes the new value $a(p) = 1 + Min\{x \mid (i,x) \in A(p)\}$.

A process p knows that each process has completed the computation of its local snapshot as soon as $a(p) \geq \beta$ (we recall that β is an upper bound on the number of processes of the network or its diameter). Thus we add a boolean variable $snapshot_p$ initialized at false; it indicates if the process knows whether all processes have completed the computation of the local snapshots.

[†]For any set S, $\mathcal{P}_{\text{fin}}(S)$ denotes the set of finite subsets of S.

I*nit-SSP*$_p$: {To initiate termination detection on the process p such that $P(p) = true$ }
begin
  $a(p) := 0$;
  $m := Min\{x \mid (i,x) \in A(p)\}$;
  if $m \geq a(p)$ **then**
    $a(p) := m + 1$;
  send$< a(p) >$ to each neighbor of p
end

R-*SSP*$_p$: {An integer $< \alpha >$ has arrived at p via door j}
begin
  receive$< \alpha >$;
  $A(p) := (A(p) \setminus \{(j,x)\}) \cup \{(j,\alpha)\}$;
  $m := Min\{x \mid (i,x) \in A(p)\}$;
  if $(m \geq a(p)$ *and* $P(p) = true)$ **then**
    $a(p) := m + 1$;
  if $a(p) \geq \beta$ **then**
    p detects the entire termination: the predicate P is true for each process
  else
    send$< a(p) >$ via each door
end

Algorithm 3: The SSP algorithm.

4.3 GPE algorithm

Let $\mathcal{A}$ be a distributed algorithm. Let $\mathcal{E}$ be an execution of $\mathcal{A}$. Our aim is to detect the termination of $\mathcal{E}$.

An execution $\mathcal{E}$ has terminated if and only if all the processes are passive and all the channels are empty. Thus to detect the termination of the execution $\mathcal{E}$, it suffices that from time to time (to be defined) at least one process initializes the computation of a snapshot and if its state is passive and its incoming channels are empty it must detect if the same property holds for all the processes. This is done by using an occurrence of the SSP algorithm. If variables of a process p indicate that the execution is not completed then p emits a signal through the network to inform each process.

In this way, we obtain an algorithm to detect global termination of the execution of a distributed algorithm. These repeated termination queries are similar to the solution described by Santoro in Section 8.3 of (Santoro, 2007).

As a corollary, this procedure can be extended to evaluate other global predicates. It requires SSP to be done over some local predicates. Thus, from local snapshots, one can detect an instant of the execution in which a global predicate holds. An example of such

a local global predicate is depicted in Section 2. The main ideas are:

1. at least one process initiates the computation of a snapshot (Algorithm 2),
2. each process p detects an instant where the computation of its local snapshot is completed,
3. each process p detects an instant where the computation of all local snapshots is completed: $snapshot_p = true$,
4. if the local snapshot of the process p is such that the local predicate holds then p initiates an occurrence of the SSP algorithm over the local predicate to check whether the predicate holds in the whole network,
5. if the local snapshot of the process p is such that the predicate is not satisfied then it sends a signal to inform every process that the execution of $\mathcal{A}$ is not completed and at least another snapshot must be computed. Variables of Algorithm 2 and Algorithm 3 are reseted and $snapshot_p = false$.

4.4 Theoretical evaluation

We are interested in characterizing the theoretical complexity of these three algorithms. As Tel(Tel, 2000), we define the time complexity by supposing that internal events need zero time units and that the transmission time (i.e. the time between sending and receiving a message) is at most one time unit. This corresponds to the number of rounds needed by a synchronous execution of the algorithm. Let A be an algorithm we want to debug and let G be a graph of size n with m edges and of diameter D. We denote $time(A)$ its time complexity, $mess(A)$ its message complexity and $size(A)$ the size required by each exchanged message.

From the complexity viewpoint, Algorithm 2 yields $O(2m)$ messages of size one bit. Moreover, it requires the size of a local snapshot of memory at any process p. More precisely, it requires $O(\Delta time(A) mess(A) size(A))$ bits of memory. We store each incoming channel state in a list. Hence, the memory depends on the number of incoming channels and the size induced by each message. Its time complexity is $O(D)$.

We now study the complexity of the SSP algorithm which is used by Algorithm 3 and by the global predicate evaluation. One know that once the confidence level $a(p)$ of every process is greater than the size of the network, the underlying algorithm is terminated (Algorithm 3) and more generally the global predicate is satisfied on the network (Section 4.3). Thus, one deduces that to reach a step in which a global predicate is checked, it requires $O(n)$ time units. The message complexity yielded by the SSP layer is equal to $O(mD)$ while the size of each message is $O(log(D))$ bits.

5 RELATED WORK

In recent years, several tools have assessed the question of simulation and visualization of distributed algorithms (Moses et al., 1998; Stasko and Kraemer, 1993; Koldehofe et al., 2003; Ben-Ari, 2001; Carr et al., 2003; Pongor, 1993; Chang, 1999).

As opposed to ViSiDiA, none fully satisfies to our needs: a distributed algorithm simulation based on a strong theoretical basis, ensuring event scheduling and using a message passing model available for asynchronous systems and anonymous networks; a visual interface for network creation and a visualization of both local and global properties and statistics; a programming interface to allow designing new algorithms; a visual debugger running concurrently to the distributed algorithm. In regard to debugging algorithms, there exist different approaches to add debugging and GPE features to a simulation software. Most of existing softwares allow the simulation of distributed systems but require a centralized entity to get the state of each process and thus to compute a snapshot of the whole network. Our approach overcomes this problem and makes the ViSiDiA debugger to be a fully distributed component.

Many notions and algorithms concerning snapshots and global predicates evaluation can be found in (Kshemkalyani and Singhal, 2008). A dedicated process is usually in charge of determining if the global state of a distributed system satisfies a global predicate. Such a process starts the Chandy-Lamport algorithm, collects processes and channels states, computes a network map, and finally tests if the labelled network satisfies the given property. To collect or to analyze local snapshots, different assumptions may be done (see (Kshemkalyani et al., 1995)): processes have unique identifiers, there is exactly one initiator or one collector process. The analysis may be done thanks to a wave. As is explained in (Kshemkalyani et al., 1995): *A wave is a flow of control messages such that every process in the system is visited exactly once by a wave control message, and at least one process in the system can determine when this flow of control messages terminates.* Furthermore waves sequence may be implemented through a traversal structure such as a tree or a ring. Some papers present specialized algorithms to obtain efficient algorithms to evaluate particular properties of networks (Mattern,

1987; Bracha and Toueg, 1987; Marzullo and Sabel, 1994; Kshemkalyani and Wu, 2007; Kshemkalyani, 2010). In any case, it is assumed that processes have identifiers and/or that there is exactly one initiator.

No such possibilities exist under our assumptions. As for as we know, our method is a new emerging solution for snapshots and GPE over anonymous networks without any knowledge about the underlying network.

6 CONCLUSION

In this paper, we have proposed a new method for debugging distributed algorithms. Our solution respects a fully-distributed scheme, releasing strong constraints usually imposed (such as process identification and network synchronicity). We assessed the question of visualizing debug information along with algorithm execution in the context of anonymous and asynchronous networks. We have proposed a new design of ViSiDiA, a simulation and visualization platform which incorporates our fully-distributed debugger. It allows several processes to initiate the debugger at any time; processes only have to initially know the network size. Hence, our tool helps in monitoring what happens locally on the network and in finding the stage when something goes wrong. It also enables to determine in a distributed manner the state of the whole system during the execution of an algorithm.

In terms of future work, several points will be dealt with. First, a checkpoint and rollback recovery system is currently being designed and implemented for ViSiDiA. It will be an extension of the existing *play*, *pause* and *stop* buttons by adding a *previous* and a *next* buttons. With these buttons, users will be able to define the number of checkpoints to compute and to browse the algorithm execution history. Besides, we are developing another layer to our debugging algorithm which permits each process to compute a *weak snapshot* of the network i.e., the maximal information it can get anonymously from this network by only knowing its size. Finally, we are also planning to extend our theoretical and practical results on debugging to local computations, mobile agents and broadcast multi-hop models (e.g. radio networks).

REFERENCES

Bauderon, M., Gruner, S., Métivier, Y., Mosbah, M., and Sellami, A. (2001). Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, volume 50 of *ENTCS*, pages 229–239.

Bauderon, M. and Mosbah, M. (2003). A unified framework for designing, implementing and visualizing distributed algorithms. *Electronic Notes in Theoretical Computer Science*, 72(3):13 – 24. GT-VMT'2002, Graph Transformation and Visual Modeling Techniques (First International Conference on Graph Transformation).

Ben-Ari, M. (2001). Interactive execution of distributed algorithms. *J. Educ. Resour. Comput.*, 1.

Bracha, G. and Toueg, S. (1987). Distributed deadlock detection. *Distributed Computing*, 2(3):127–138.

Carr, S., Fang, C., Jozwowski, T., Mayo, J., and Shene, C.-K. (2003). Concurrent mentor: A visualization system for distributed programming education. In *PDPTA'03*, pages 1676–1682.

Chalopin, J., Métivier, Y., and Morsellino, T. On snapshots and stable properties detection in anonymous fully distributed systems. *submitted, 2011*.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75.

Chang, X. (1999). *Network simulations with OPNET*, pages 307–314. ACM.

Derbel, B. and Mosbah, M. (2003). Distributing the execution of a distributed algorithm over a network. In *Information Visualization, 2003. IV 2003. Proceedings. Seventh International Conference on*, pages 485 – 490.

Guerraoui, R. and Ruppert, E. (2005). What can be implemented anonymously? In *DISC*, pages 244–259.

Koldehofe, B., Papatriantafilou, M., and Tsigas, P. (2003). Integrating a simulation-visualisation environment in a basic distributed systems course: a case study using lydian. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, ITiCSE '03, pages 35–39, New York, NY, USA. ACM.

Kshemkalyani, A. D. (2010). Fast and message-efficient global snapshot algorithms for large-scale distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 21(9):1281–1289.

Kshemkalyani, A. D., Raynal, M., and Singhal, M. (1995). An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4):224–233.

Kshemkalyani, A. D. and Singhal, M. (2008). *Distributed computing*. Cambridge.

Kshemkalyani, A. D. and Wu, B. (2007). Detecting arbitrary stable properties using efficient snapshots. *IEEE Trans. Software Eng.*, 33(5):330–346.

Marzullo, K. and Sabel, L. S. (1994). Efficient detection of a class of stable properties. *Distributed Computing*, 8(2):81–91.

Matocha, J. and Camp, T. (1998). A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221.

Mattern, F. (1987). Algorithms for distributed termination detection. *Distributed computing*, 2:161–175.

Moses, Y., Polunsky, Z., Tal, A., and Ulitsky, L. (1998). Algorithm visualization for distributed environments. In *Proceedings of the 1998 IEEE Symposium on Information Visualization*, pages 71–78, Washington, DC, USA. IEEE Computer Society.

Pongor, G. (1993). Omnet: Objective modular network testbed. In *MASCOTS '93: Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 323–326, San Diego, CA, USA. The Society for Computer Simulation, International.

Raynal, M. (1988). *Networks and distributed computation*. MIT Press.

Santoro, N. (2007). *Design and analysis of distributed algorithm*. Wiley.

Stasko, J. T. and Kraemer, E. (1993). A methodology for building application-specific visualizations of parallel programs. *J. Parallel Distrib. Comput.*, 18:258–264.

Szymanski, B., Shy, Y., and Prywes, N. (1985). Synchronized distributed termination. *IEEE Transactions on software engineering*, SE-11(10):1136–1140.

Tel, G. (2000). *Introduction to distributed algorithms*. Cambridge University Press.

Yamashita, M. and Kameda, T. (1996). Computing on anonymous networks: Part i - characterizing the solvable cases. *IEEE Transactions on parallel and distributed systems*, 7(1):69–89.