Simulation of Ad hoc Networks in ReactiveML [*]

Farid Benbadis
LIP6, Université Paris 6
Farid.Benbadis@lip6.fr

Louis Mandel
VERIMAG
Louis.Mandel@imag.fr

Marc Pouzet
LRI, Université Paris-Sud 11
Marc.Pouzet@lri.fr

Ludovic Samper
France Telecom R&D
Ludovic.Samper@imag.fr

Abstract

This paper presents a programming experiment of complex network routing protocols for mobile ad hoc networks within the reactive language REACTIVEML.

Mobile ad hoc networks are highly dynamic networks characterized by the absence of physical infrastructure. In such networks, nodes are able to move, evolve concurrently and synchronize continuously with their neighbors. Due to mobility, connections in the network can change dynamically and nodes can be added or removed at any time. All these characteristics — concurrency with many communications and the need of complex data-structure — combined to our routing protocol specifications make the use of standard simulation tools (*e.g.*, NS-2, OPNET) inadequate. Moreover network protocols appear to be very hard to program efficiently in conventional programming languages.

In this paper, we show that the *synchronous reactive model* as introduced in the pioneering work of Boussinot matters for programming such systems. This model provides adequate programming constructs — namely synchronous parallel composition, broadcast communication and dynamic creation — which allow a natural implementation of the hard part of the simulation. This thesis is supported by two concrete examples: the first example is a routing protocol in mobile ad hoc networks and the simulation focuses only on the network layer. The second one is a routing protocol for sensors networks and the wholes layers are faithfully simulated (hardware, MAC and network layers). More importantly, the physical environment (e.g., clouds) has also been integrated into the simulation using the tool LUCKY.

The implementation has been done in REACTIVEML, an embedding of the reactive model inside a statically typed, strict functional language. REACTIVEML provides reactive programming constructs together with most of the features of OCAML. Moreover, it provides an efficient execution scheme for reactive constructs which made the simulation of real-size examples (with several thousand of nodes) feasible.

1. Introduction

Ad hoc networks are highly dynamic networks characterized by the absence of any physical infrastructure. In this paper, we study two kinds of ad hoc networks : mobile and sensor networks.

Mobile ad hoc networks are composed of nodes which evolve concurrently and have to synchronize continuously with other nodes. Among existing routing protocols, age and position based protocols have recently emerged because of their relatively simple and efficient policies: no location service is required, the destination position discovery is achieved during the packets forwarding step where nodes make elementary forwarding decisions based solely on the coordinates of their direct neighbors and of the destination [18]. This avoids the need for topology knowledge beyond one-hop.

Sensor networks consist in ad hoc networks but with specific constraints. A sensor network is composed by a large number of sensors (several thousands). Those nodes are designed to be as small and cheap as possible. Sensor networks can be deployed in situation with difficult access and/or no available energy. Thus, the nodes are power-constrained. Indeed, the network has to achieve a certain service as long as possible, and because there is no or very few infrastructure, and because of the size of the network, nodes that ran out of energy are not replaced.

These networks are typical examples of *complex dynamic systems*, that is, dynamic systems where not only the state of system evolves during the execution but also its internal structure. Ensuring a correct behavior of such a network is challenging, and the better way to tackle this problem is to build *models* that can be simulated.

For example, power consumption is crucial in sensor networks. All the elements of a network have some influence on power consumption: the nodes architecture, the radio access functionalities, the communication protocols, the application, and even network environment which stimulates the sensors. Thus, power consumption has to be estimated in advance. This can be achieved through simulation.

The characteristics of these networks — concurrency with many synchronizations and the need of complex data-structures — make the use of standard simulation tools like NS-2 [1] or OPNET [27] inappropriate. Indeed, NS-2 has been originally designed for wired networks and does not treat well wireless networks. In particular, it is only able to simulate small networks (1000 nodes networks seems to be barely conceivable) whereas we consider large scale networks.

In this paper, we show that the *synchronous reactive model* introduced by Boussinot [10, 11, 34] strongly matters for programming those systems. We argue that this model provides the good programming constructs — synchronous parallel composition with a common global time scale, broadcast communication and dynamic creation — making the implementation of the hard part of the network surprisingly simple and efficient. We can remark that the reactive synchronous model is not contradictory with the asynchronous aspect of these networks. Synchrony only gives the ability to all nodes to react in a fair way as it could be done in an imperative implementation. The model provides *language concurrency* as opposed to *run-time concurrency*: reactive parallel programs are translated into conventional single-thread, yet efficient programs [2, 9, 13, 36]. Whereas a similar formulation is possible in any conventional programming language using one run-time thread per node, it would not allow to simulate large networks for clear efficiency reasons.

The programs have been written in REACTIVEML (RML for short) [1] an embedding of the reactive model inside a statically

[*] Submitted to publication. June 2006.

[1] The distribution can be accessed as: `http://ReactiveML.org`.

typed, strict functional language [25, 24]. REACTIVEML provides reactive programming constructs with most of the features of OCAML [23]. Reactive constructs give a powerful way to describe the dynamical part of the system whereas the host language OCAML provides data-structures for programming the algorithmic (combinatorial) part. Moreover, REACTIVEML provides an efficient execution scheme for reactive constructs which made the simulation of real-size examples feasible.

The purpose of this paper is to convince of the adequacy of the reactive model for real-size simulation problems like network protocols. As a side-effect, this protocol can also serve as an interesting benchmark for validating and comparing the various implementations of the reactive model [2, 13, 36].

The paper is organized as follows. Section 2 discuss about the adequacy of the programming model on which REACTIVEML is based for programming network simulators. The section 3 presents briefly a routing protocol we have considered and its REACTIVEML implementation. The language is very young and the paper can thus be considered as a tutorial introduction of the language through two real examples. In order to ease the presentation, we start with a survival kit which can easily be skipped. We only give the hard part of the code and give hyperlinks to the complete distribution. Section 4 presents the simulation of a second protocol used in sensor networks, taking the physical environment into account. Finally, section 5 discuss related works and we conclude in section 6.

2. Why RML Matters to Program Simulators

One first observation is that even if there exists many different network simulators, people continueto develop their own simulator. Why? Creating your own simulator guarantees that this simulator will perfectly fit your needs. Indeed, even if some simulators provide several levels of detail, a custom simulator can exactly address the faced problem. For example, NS-2 or OPNET need that the layers 1 to 3 be described even if the designer is interested only in layer 3 (the network layer).

Of course, writing a simulator from scratch is time consuming but it avoids the cost of learning an other simulator. In order to reduce the time and effort needed for to write its own simulator, a high level language is required.

For this purpose, we claim that REACTIVEML which combines reactive constructs for describing the dynamics of the system with the expressiveness of OCAML reduces this effort. Indeed, because REACTIVEML is built above OCAML, it keeps its main properties [2]: a powerful type system, user-definable algebraic data types, definitions through pattern matching and automatic memory management. These features are important in order to manage complex data structures such as nodes or packets. The interest of OCAML for programming the simulation of ad hoc networks has been already identified by the authors of NAB [15]. By adding reactive constructs to OCAML, REACTIVEML provides a means to describe each node and their parallel composition in a more natural way. This is a extra but essential advantage of the REACTIVEML simulator with respect to the NAB version developed in OCAML.

Authors of [12] insist on the importance of a visualization tool that helps users to understand the complex behavior in the simulation. Nam (Network Animator) [16], the visualization tool of NS-2 was proposed during the VINT project. A visualization tool for a network simulator is not only useful in order to give intuition about protocols to develop but also to aid in debugging both the simulator and the protocol stack. Since REACTIVEML is being built above OCAML, it can use any OCAML program. It is for example possible to generate a trace file that is compatible with Nam or to write its own visualization tool.

More interesting is the ability to dynamically add or remove observers (and printers) during the simulation. Such observers run in parallel with the simulation without modifying it. The reactive features of REACTIVEML give also means to act dynamically on the behavior of the network. For example, a function that creates a new node with a mouse-click is about ten lines long (see section 3.4). This feature is interesting to provoke a certain behavior of the simulated network. An other feature could be the dynamic creation of observers. Indeed, the graphic window for instance can be removed during execution (to speed up the simulation) and then displayed again to monitor the simulation. Moreover, the visualization tool is also an REACTIVEML program, letting the user display just is needed on the graphic output (e.g., collision, emission of packets).

In wireless networks, data transmission impacts the behavior of a network. To generate messages, simulators can use statistic laws on the nodes. Poisson processes for instance are often used. This environment modeling shows its limit, especially in the case of sensor networks where a more accurate modeling of the environment is sometimes needed. Sridharan et al. [35] linked the sensor network simulator TOSSIM with MATLAB. It is possible to describe an environment in REACTIVEML. An environment process will then run in parallel with the rest. A better way is to rely on a dedicated language to model the environment. This is why we have interfaced REACTIVEML with LUCKY, a language to describe stochastic reactive programs [22].

Network simulators are computer intensive applications and efficiency of the programming language is thus a key point. Of course, a dedicated simulator will certainly be more efficient than the one obtained by using a general purpose one. The programming language must itself be efficiently compiled. Because REACTIVEML programs are compiled into OCAML code without any use of run-time concurrency (e.g., *threads* or unix processes) and which are in turn compiled into native code, we were able to simulate large scale networks with several thousand of nodes. More information about the implementation of REACTIVEML can be found in [24].

We illustrate these points on two examples: a mobile ad hoc network and a sensor network.

3. Simulation of Mobile Ad hoc Networks

Our first example is a simulator that evaluates dissemination methods for Age and Position Based (APB) routing protocols in mobile ad hoc networks.

3.1 Age and Position Based Routing

The main principle of APB routing protocols is that each node may have an information about each other node's location. This information is stored in a position table and associated to an *age* that represents the time elapsed since the last time the information has been updated. The position table is queried by a packet to estimate destination position.

In this routing methods, destination location discovery is performed during packet transfer: a source node does not know destination location when it sends the packet, it only has an estimation about it. We describe the EASE (Exponential Age SEarch) routing method, where a source node s needs to communicate with a destination d, as follows:[3]

```
Set $i := 0$, $age := \infty$, $a_0 := s$ in
While $a_i \neq d$ do
  search around $a_i$ a node $n_i$ such that $age(n_i, d) \leq age/2$;
```

[2] http://caml.inria.fr/about/index.en.html

[3] For more details about EASE, see [18]

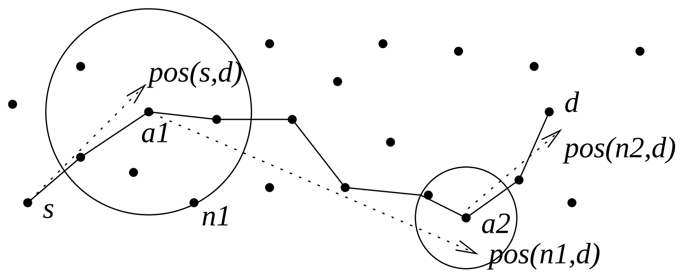


Figure 1. Routing a packet from s to d: anchor nodes a_1 and a_2 refine estimation of d's position.

```
  $age := age(n_i, d)$;
  Set $m := a_i$ in
  While $m$ is not the closest node of $pos(n_i, d)$ do
    $m :=$ next neighbor toward $pos(n_i, d)$
  done;
  $i := i + 1$;
  $a_i := m$ (* the closest node of $pos(n_i, d)$ *)
done
```

where a_i are anchor nodes that search for a better estimation of destination position than the one included in the packet. $pos(n_1, n_2)$ is n_2's position as known by n_1, and $age(n_1, n_2)$ is the age of this information. An illustration of this algorithm is represented in Fig. 1.

Two different methods are used to update position tables in APB routing protocols. The first one, LE (for Last Encounter), introduced in [18], uses the encounter between nodes. Each node remembers the location and time of its last encounter with every other node. The second method, ELIP (Embedded Location Information Protocol), uses also the encounter between nodes, but disseminates nodes locations in data packets [6]. In this method, a source node can include its current coordinates in every message it sends in such a way that all the nodes that participate to the forwarding procedure update their knowledge about the source.

To simulate these two protocols, we have to represent a set of nodes that evolve in parallel. All of them move, communicate and update their local position tables, which contains an estimation of the position of all other nodes, at every simulation instant.

The goal of our simulator is to compare two dissemination methods to be used in an APB ad hoc routing algorithm. We did not conceive a generic simulator which can be used for any routing protocol. Moreover, we do not focus on the efficiency of the routing protocol EASE, which has been proven in [18], but on the performance of ELIP and LE, two dissemination algorithms. The important point is that the two dissemination algorithms are evaluated in the same conditions. For this reason, we do not have to consider the physical and link layers and do not take into account the interferences and packets loss. We only focus on the network layer, and consider that when a node broadcasts a packet, all its direct neighbors receive it.

3.2 Implementation in ReactiveML

We present here the structure of the simulator and detail some key points. The full implementation is available at
`http://ReactiveML.org/eurasip`.

3.2.1 ReactiveML Survival Kit

REACTIVEML is built above OCAML. Every OCAML program (without objects, labels and functors) is a valid REACTIVEML program and REACTIVEML code can be linked to any OCAML library. In the following, we assume that the reader is rather familiar with OCAML.

A program is a set of definitions. Definitions introduce, like in OCAML, types, values or functions. We illustrate the syntax with the the following example. It defines the type of positions as a record and an example of a position $(4, 2)$. Then, we define the function `distance2` that computes the square of the Euclidean distance between two positions.

```
type position = { x: int; y: int }
let pos = { x = 4; y = 2 }

val pos : position

let distance2 p1 p2 =
  (p2.x - p1.x) * (p2.x - p1.x)
  + (p2.y - p1.y) * (p2.y - p1.y)

val distance2 : position -> position -> int
```

This is regular OCAML code and the REACTIVEML compiler automatically computes the type signatures (printed in *italic* font).

REACTIVEML adds to this functional language, the *process* definition. Processes are state machines whose behavior can be executed through several instants. They are opposed to regular OCAML functions which are considered to be instantaneous [4]. Consider the process `hello_world` that prints "hello" at the first instant and "world" at the second one (the `pause` statement suspends the execution until the next instant):

```
let process hello_world =
  print_string "hello␣";
  pause;
  print_string "world"

val hello_world : unit process
```

This process can be instantiated using the `run` primitive and typing: `run hello_world`.

Communication between parallel processes is made by broadcasting signals. A signal can be emitted (`emit`) and awaited (`await`). There is also suspension (`do/when`) and preemption (`do/until`) constructs that use signals. We illustrate these constructs with a process `ping_pong` that prints alternatively `ping` and `pong`.

```
let process ping_pong =
  signal s1, s2 in
  loop
    await s1;
    print_string "ping";
    emit s2
  end
  ||
  emit s1;
  loop
    await s2;
    print_string "pong";
    emit s1
  end
val ping_pong : unit process
```

The construct `signal/in` declare the two signals `s1` and `s2` then two expressions are executed in parallel. The first one prints `ping`

[4] In circuit terminology, processes are *sequential* functions whereas OCAML functions are considered to be *combinatorial*.

and the other one prints pong. Synchronizations are made through the signals s1 and s2.

Valued signals call for a particular treatment in case of multi-emission. For example, what is the value of x in the following example where the values 1 and 2 are emitted during the same instant?

```
emit s 1 || emit s 2 || await s(x) in ...
```

There are several solutions. So, when a valued signal is declared, we have to define how to combine values in the case of multi-emission on a signal during the same instant. This is achieved with the construct:

signal *name* default *value* gather *function* in *expr*

Thus, if we want to define the signal s such that it computes the sum of the emitted values, we can write:

```
signal s default 0 gather (+) in
emit s 1 || emit s 2 || await s(x) in print_int x
(* s : (int, int) event *)
```

The expression `await s(x) in print_int x` awaits the first instant in which s is emitted and then, at the next instant, prints 3 which is the sum of the emitted values. The type `(int, int) event` of the signal s states that the emitted values and the combined values are integers.

The type of emitted values on a signal and the type of the combined value are not necessarily the same. If τ_1 is the type of the values emitted on s and τ_2 is the type of the combined value then s type is of type (τ_1,τ_2) event. In this case, the default value must have type τ_2 and the gathering function must have type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_2$.

In the following example, the signal s collects all the values emitted during the instant:

```
signal s default [] gather fun x y -> x :: y in
emit s 1 || emit s 2 || await s(x) in ...
(* s : (int, int list) event *)
```

Here, the default value is the empty list and the gathering function builds the list of emitted values. So the value of x is the list [2; 1] [5] The notation `signal s in ...` is a shortcut for this gathering function.

We stop this short introduction to ReactiveML here. Various examples of programs can be found at http://ReactiveML.org.

3.2.2 Data structures

We consider a node n. To use an age and position based routing protocol, n must be aware about its position. n stores the information it has about other nodes positions in a local position table. Each entry in this position table looks like this:

$$[ID_a, pos(n,a), date(n,a)]$$

Here this entry is about a node a. $pos(n,a)$ is an estimation of a's position, and $date(n,a)$ indicates when n has got this information. n knows its immediate neighborhood represented by the set of all the nodes under its radio range.

We then define the type of a node as a record:

```
type node =
  { id: int;
    mutable pos: position;
    mutable date: int;
    pos_tbl_le: Pos_tbl.t;
    pos_tbl_elip: Pos_tbl.t;
    mutable neighbors: node list; }
```

where id is the unique identifier of the node. pos is its current position which is its coordinates on a grid with squares of one meter square. neighbors the list of nodes that are under its coverage range. date is the current local date of the node, essentially used to compute the age of other nodes position information. pos_tbl_le and pos_tbl_elip are the position tables used to simulate the LE and ELIP dissemination protocols.

The record contains mutable fields which can be modified, and non-mutable fields which are fixed at the creation of the concerned record. pos_tbl_le and pos_tbl_elip are not mutable because we implement them as imperative structures in the module Pos_tbl. The position tables associate a position and a date to each node.

Packets for age and position based routing protocols contain the following fields: the source and destination identifiers, an estimation of destination position, the age of this information, and data to be transmitted. When using ELIP, the packets can contain also source node location.

In the simulator, packets do not contain data but contain other information used for statistics computation. This information is also useful for the graphical interface.

```
type packet =
  { header: packet_header;
    src_id: int;
    dest_id: int;
    mutable dest_pos: position;
    mutable dest_pos_age: int;
    (* to compute statistics *)
    mutable route: node list;
    mutable anchors: node list; }
```

src_id, dest_id, dest_pos and dest_pos_age are used for routing. route is the list of nodes that the packet traveled through, and anchors is the list of anchor nodes. header indicates if the packet is a LER or an ELIP packet.

```
type packet_header =
  | H_LE
  | H_ELIP of position option
```

The type `position option` indicates that ELIP packets can contain the position of the source node or not.

3.2.3 Behavior of a node

The heart of the simulator is the description of a node's behavior. Indeed, the simulator execution is the parallel composition of all the nodes execution.

The behavior of each node is composed of three steps. A node (1) moves, (2) discovers its neighborhood, (3) routes packets. These steps are combined in a process node [6] which is parameterized by the initial position of the node pos_init, a function move that computes its next position, and a function make_msg that creates a list of destinations to reach.

```
let process node pos_init move make_msg =
  let self = make_node pos_init in
  loop
    self.date <- self.date + 1;

    (* Moving *)
    self.pos <- move self.pos;
    emit draw self;
```

[5] The parallel composition is associative and commutative, so the order of the elements of the list associated to s is not specified.

[6] http://ReactiveML.org/eurasip/elip/node.rml.html

```
    (* Neighborhood discovering *)
    ...
    update_pos_tbl self self.neighbors;

    (* Routing *)
    pause;
    let msg = make_msg self in
    ...
    pause;
  end
```

This process creates a record of type `node` that represents the internal state of the node. Then it enters in the permanent behavior which is executed through three instants. In the first one, a node updates the local date, moves and emits its new position on the global signal `draw` for the graphical interface (a screen-shot is given in Fig. 2). At the end of the first and during the second instant, the new neighborhood is computed and the position tables are updated using encounters between nodes. The third and last instant is the routing. By enclosing this part between two `pause` statements, we have the guaranty that the topology can not change. We detail now the main steps of the process.

Mobility Nodes movements are parameterized by a mobility function `move`. This function computes the new position of a node according to the current position. The `move` function must have the following signature:

```
val move : position -> position
```

We can implement very simple mobility functions like random moves where a node can move to one of its eight adjacent positions.

```
let random pos = translate pos (Random.int 8)

val random : position -> position
```

(`Random.int 8`) is the call of the function `Random.int` of the OCAML standard library and `translate` which is a function that returns a new position.

We can also implement more realistic mobility models like the random way-point one. With this mobility model, a point is chosen randomly in the simulation area and the node moves up to this point. When it reaches this point, a new one is chosen. This function is interesting because it must keep an internal state.

```
let random_waypoint pos_init =
  let waypoint = ref pos_init in
  fun pos ->
    if pos = !waypoint
    then waypoint := random_pos();
    (* move in the direction of !waypoint *)
    ...

val random_waypoint :
  position -> position -> position
```

The partial application of this function with only one parameter:

```
  random_waypoint (random_pos())
```

returns a mobility function that can be given as an argument to a node.

Neighborhood In real networks, the neighborhood of a node is obtained thanks to the physical layer. By contrast, in the simulator it has to be computed. Neighborhood discovery is the key point of the efficiency of the simulator. We first give a simple method to compute the neighbors of a node, then we explain how it can be improved.

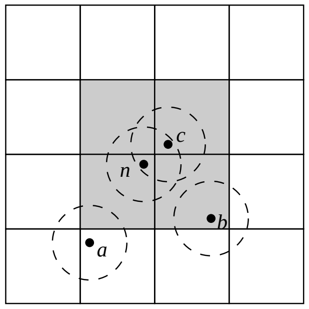


Figure 3. Topology split into multiple squares. Node n emits its position on the gray squares, while it listens on the one it is located.

To compute its neighborhood, a node needs to know the position of other nodes. In this first method, we use a signal `hello` to gather all nodes coordinates. Each node emits its position on `hello` such that the value associated to the signal is the list of all nodes. Thus the code of a node looks like the following (`self` is the internal state of the node):

```
  emit hello self;
  await hello(all) in
  self.node_neighbors <- get_neighbors self all;
```

The function `get_neighbors` returns the `all`'s sublist that contains the nodes under the coverage range of `self`.

This neighborhood discovery method is very simple but its drawback is that each node has to compute its distance with all other nodes leading to a quadratic complexity in the number of nodes. To improve this method, we split the simulation area in small areas and associate a `hello` signal to each area. That way, a node has only to compute its distance with the nodes in the areas under its range.

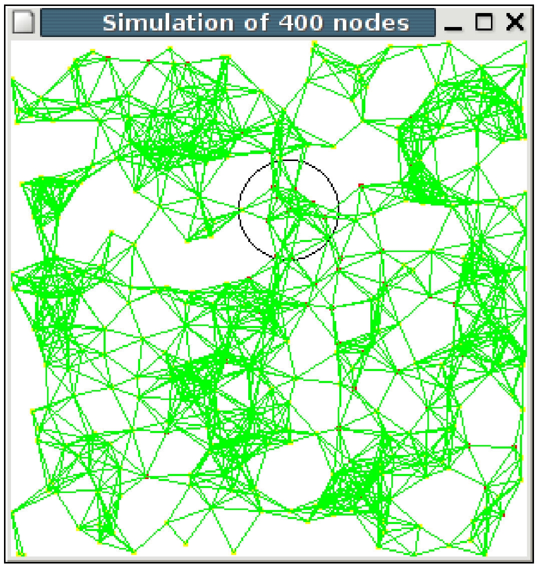
We consider node n in Fig. 3. A `hello` signal is associated to each square. Node n sends its position on the signals associated to the 4 squares touched by its radio transmission (the 4 gray squares in this figure). In the same way, nodes a, b, and c emit their position on the signals associated to the squares that intersect their coverage range. So, nodes a and c transmit their position on the signal associated to the square where n is. n receives then positions of a and c. Using this information, n computes its distance from a and c and concludes that c is a neighbor while a is not. n does not consider node b because this node does not emit its position on the signal associated to the square where n is located.

All the `hello` signals are stored in a two dimensional array `hello_array`. We define a function `get_areas` that returns the area of a node and the list of neighbor areas that are under its range.

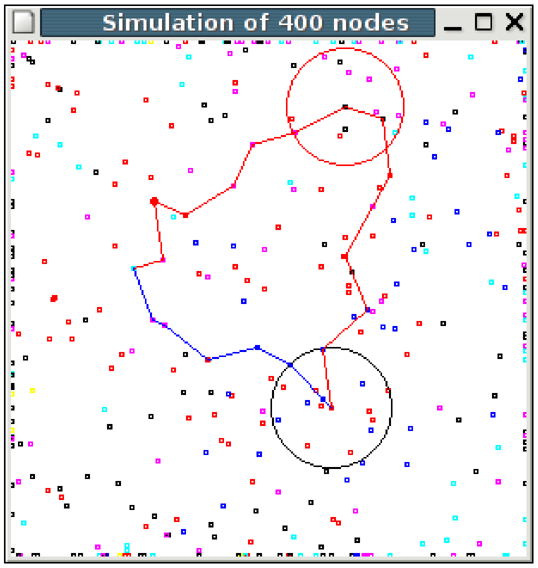
```
val get_areas :
  position -> (int * int) * (int * int) list
```

Now the behavior of a node is to emit its position in all the areas under its range and to compute its distance with all the nodes which have emitted their position in its area. So the code of the neighborhood discovery becomes:

```
  (* Compute areas under the coverage range *)
  let (i,j) as local_area, neighbor_areas =
    get_areas self.pos.x self.pos.y
  in
  (* Emit the position on each of these areas *)
  List.iter
    (fun (i,j) -> emit hello_array.(i).(j) self)
    (local_area::neighbor_areas);
```



(a) Topology connectivity. Each green line represents two neighbor nodes, while the black circle represents one node coverage region.



(b) An example of routing paths using ELIP (blue) and LE (red) dissemination methods. The red circle represents the search performed by the anchor node when using LE.

Figure 2. Screen-shots of the simulator graphical interface.

```
(* Get the nodes that emits their position *)
await hello_array.(i).(j) (all) in
self.neighbors <- get_neighbors self all;
```

Fig. 4 shows the effect of the area split on execution time. In Fig. 4(a), we compare the first method, where all the nodes emit and listen on the same signal, to the second one, where each nodes emits only on the areas under its radio range. Because, in the first method, each node computes its distance to every other node, the neighborhood discovery procedure spends much more time than in the second method, where each node computes its distance to the nodes that emit on its adjacent areas only. We observe that for the simulation of 1500 nodes the second method is 2 times faster than the first one. Then for 2500 nodes it is 5 times faster and for 5000 nodes it is more than 10 times faster.

We focus now on the second method, which is more appropriate. As we can see in Fig. 4(b), the execution time depends heavily on the area size. This figure represents the time required for the simulation of a 3000 nodes topology using three different densities (average number of neighbors a node have). We observe that dividing the topology in too many squares is not efficient. In this case, each node emits its position on a large number of signals, which requires resources. On the other hand, dividing the topology in large squares makes that a node receives large number of nodes positions on its signal. It spends then long time to compute distances with nodes placed far from it. Simulation results show that 2-ranges-sided squares seems to be a good compromise for the three densities simulated.

Routing The last step in a node execution is the packets routing, which is described in section 3.1[7] The important point is that we assume that routing is instantaneous, which means that the topology is fixed during routing. This scenario is realistic because we assume that nodes move at human speed, while packets travel as radio waves speed. Topology is then supposed to change at time scale of seconds or longer, while packets spend at most tens of milliseconds from source to destination. We can then use OCAML functions. which are supposed instantaneous, to implement the routing protocols.

In the simulator, we compare two location dissemination methods, both of them combined with the same forwarding algorithm. This algorithm computes the next node which will receive the packet. We use a classical geographical method. The packet is forwarded to the neighbor that is the nearest (for the Euclidean distance) to the destination. The interesting point in the function `forward` is that a node can access to the internal state of other nodes executed in parallel. REACTIVEML guarantees that this action is not interruptible such that there is no need to protect the access to share data like in the thread model.

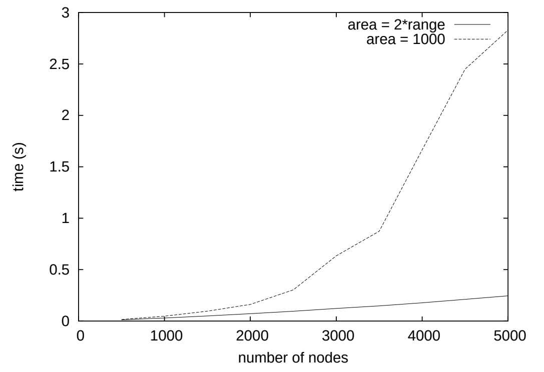
3.2.4 The main process

The main process that executes the simulation starts with an initialization part to define simulation parameters. Then it executes `n` nodes in parallel (`for/dopar` is a parallel iterator), the graphical interface and others synchronous observers.

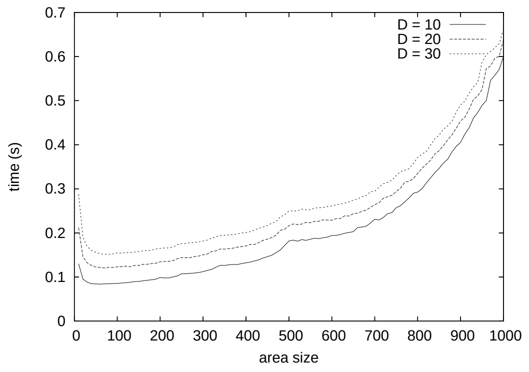
```
let process main =
  (* Initialization part *)
  ...

  (* Main part *)
  begin
    for i = 1 to n dopar
      let pos = random_pos() in
      run (node pos (Move.random_waypoint pos)
                Msg.make)
    done
    ||
    run (draw_simul draw)
    ||
    ...
```

[7] `http://ReactiveML.org/eurasip/elip/routing.rml.html`



(a) Comparison of simulation times depending on the number of nodes and the neighborhood discovery method.



(b) Comparison of simulation times depending on the areas size for the improved method.

Figure 4. Simulation times for neighborhood discovery.

```
  end
val main : unit process
```

The structure of this process is the classical structure of the main process of a simulator.

3.3 Analysis

The simulation speed depends on the parameters: number of nodes, coverage range, number of emitted packets, simulation area size, etc. These parameters are linked through the relative density, given by the number of nodes per coverage zone, in order to get a realistic simulation environment.

The simulations have been done on the following computer:

PC Dual-PIV 3.2Ghz, RAM 2GB
running Debian Linux 3.1

First, we analyze our program capability to simulate large networks. Fig. 5(a) represents simulation times depending on number of nodes. We observe that at about 8000 nodes the execution time becomes suddenly more important. This is due to memory usage, when there is enough nodes so that the process has to swap. In Fig. 5(b), the memory usage looks like being quadratic in the number of nodes. This result is natural because each node has a position table that contains positions of all other nodes. To overcome this limitation, we can limit the number of destination nodes such that only a subset of nodes have to be in the position tables.

Now, we compare our simulator with NAB, a simulator developed by the authors of EASE. The Fig. 6(a) represents the execution time for a simulation where each node emits a packet at each instant. This type of simulation with a lot of mobility and communications is interesting to evaluate the dissemination algorithms. The numbers shows that NAB is less efficient than the REACTIVEML implementation but this comparison is unfair. Indeed NAB simulates the MAC layer such that routing a packet is much more time consuming than in our simulator. Because neighborhood discovery is time consuming (about 25% of the simulation time with the optimized version), an interesting comparison with NAB is thus the packets-free simulations. In this case, we compare only the neighborhood discovery. The MAC layer does not affect the simulation such that, the two simulators have to do exactly the same thing. The execution time is given in Fig. 6(b). We can observe that the expressiveness of the signal communication gives us a very simple way to define an efficient algorithm. Moreover, our simulator use less memory than NAB.

3.4 Dynamic Extension

In ad hoc networks, protocols must be robust to topology changes, which includes nodes join and leave. Thus, nodes can be added or removed dynamically.

Preemptible nodes are defined using the construct `do/until` that executes its body until a signal is emitted:

```
let process
    preemptible_node pos_init move make_msg kill =
  do
    run (node pos_init move make_msg)
  until kill done
```

Here, when the signal `kill` is emitted, the node is removed from the simulation.

Figure 7 gives the memory usage of a simulation that removes a node at each instant. It shows that the garbage collector works well and deallocate processes which are removed.

A more interesting point is the dynamic creation of processes. In REACTIVEML, dynamic creation is made through recursion. We define the recursive process `add` that creates new nodes as follow:

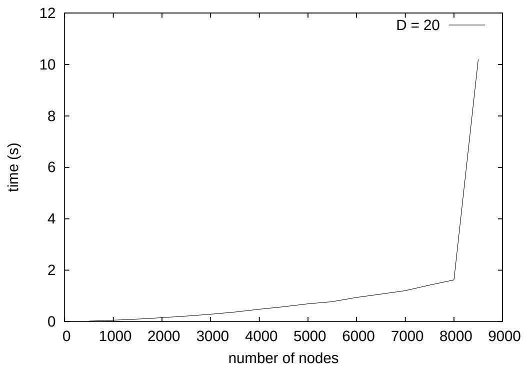
```
let rec process add new_node start =
  await new_node (pos) in
  run (add new_node start)
  ||
  await immediate start;
  run (node pos
          (random_waypoint (random_pos()))
          make_msg)
```

This process is parameterized by two signals: `new_node` and `start`. `new_node` is emitted (with an initial position) when a new node is created. The signal `start` is emitted at each new moving step, it is used to synchronize the new node with the other ones. Indeed, the new node must start with its moving step when all nodes move.

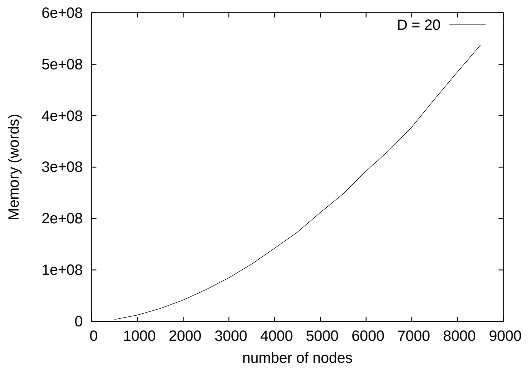
4. Simulation of Sensor Networks

In this part, we will detail the programming of an other simulator in REACTIVEML. This is simulator for sensor networks and, this time, we do a finer simulation with all the layers of the network. We call it GLONEMO(for **glo**bal **ne**twork **mo**del [8].

[8] Screen-shot Fig. 8

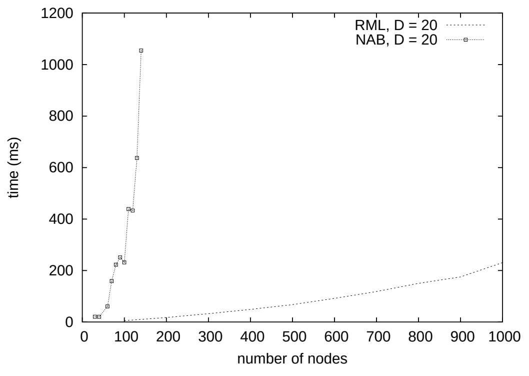


(a) Simulation times.

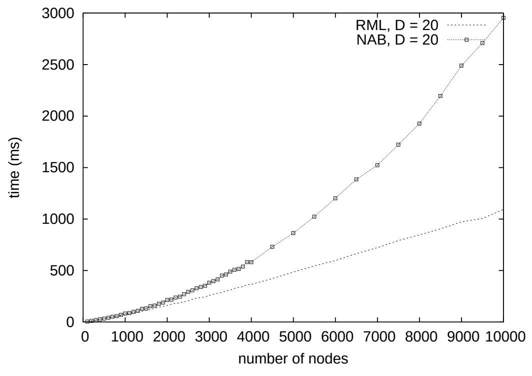


(b) Memory usage.

Figure 5. Simulations depending on the number of nodes with a topology density D=20.



(a) With packets emission.



(b) Without packet emission.

Figure 6. Comparison of simulation times, between NAB and REACTIVEML simulator, depending on the number of nodes with a topology density D=20.

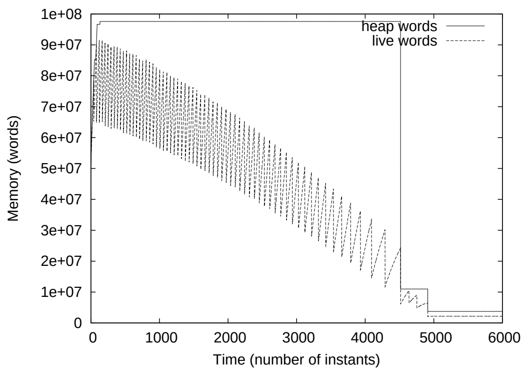


Figure 7. Memory usage of a simulation that removes a node at each instant.

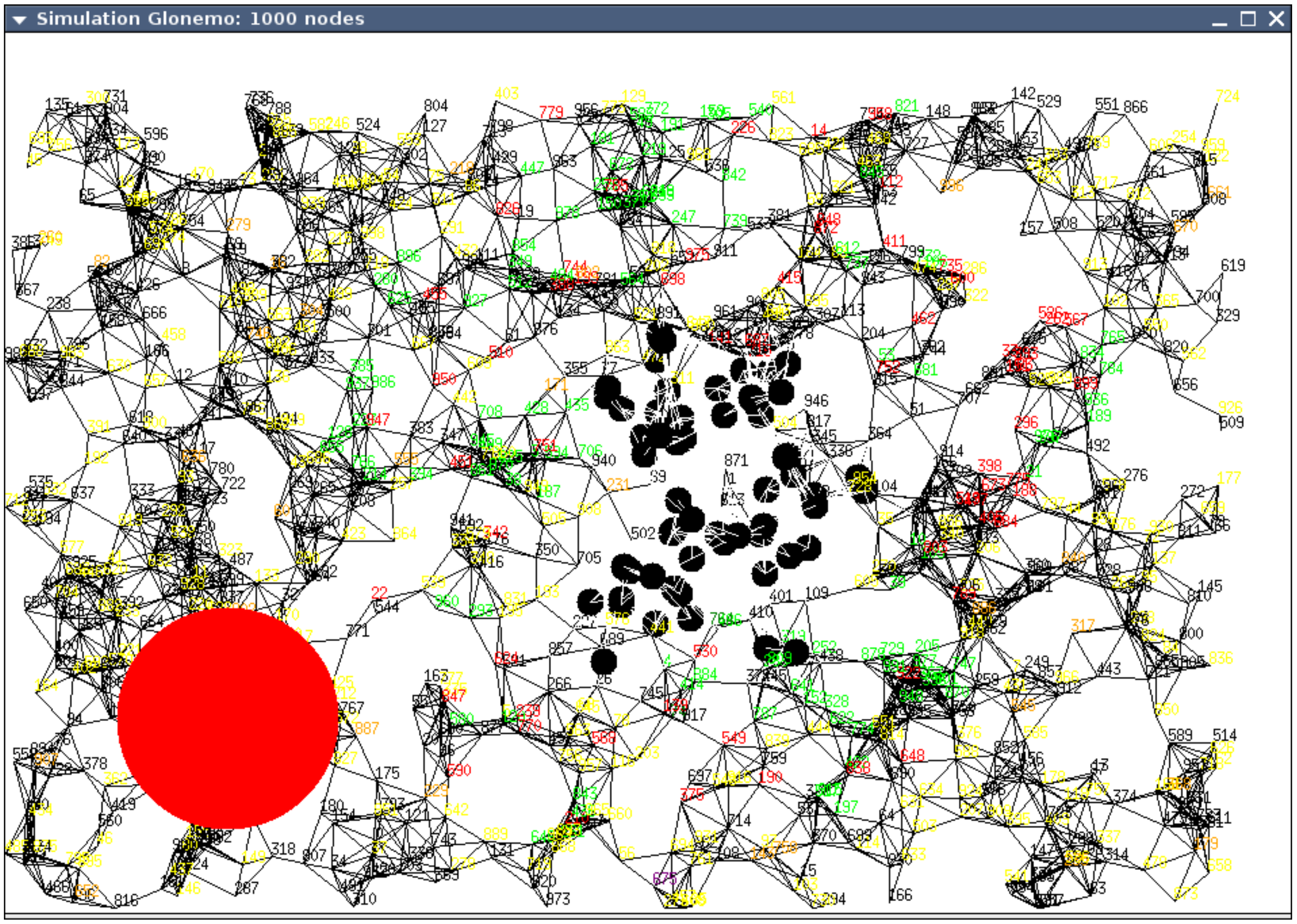


Figure 8. Screen-shots of the simulator graphical interface. The red disk represents a toxic cloud. Black disks are nodes without energy.

In our example, the network has to warm when a toxic cloud is detected and the goal is to design low energy consumption routing protocols.

4.1 Structure of the Simulator

4.1.1 Hardware Model

In order to have an accurate model of the energy consumption, a model of the hardware is needed. Indeed, without this modeling, the energy would have to be evaluated using other observations (like the number of packet sent) and abstractions. The accurate model of the hardware was easily described in REACTIVEML, it contains several automata one for each consuming part, the radio, the CPU and the memory.

4.1.2 Medium Access Control

The radio module is an important source of consumption for the sensor nodes. To reduce that consumption, there exists specific Medium Access Control(MAC) protocols for sensor networks. Those protocols minimize the time the radio is alight [28, 14]. Thus, to analyze the energy consumption of a sensor network, the MAC layer cannot be omitted. Furthermore, this simulator could be used to evaluate different MAC protocols.

The sensor networks MAC protocol that has been implemented here is a Preamble Sampling MAC protocol (see fig 9) like WiseMAC [14] and BMAC [28]. In the preamble sampling

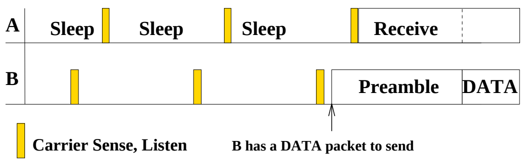


Figure 9. Medium Access Control: the preamble sampling technique.

technique, a preamble precedes each data packet for alerting the receiving node. All nodes in the network sample the medium with a common period, but their relative schedule offsets are independent. If a node finds the medium busy after it wakes up and samples the medium, it continues to listen until it receives a data packet or the medium becomes idle again. The size of the preamble is set to be equal to the preamble sampling period.

It was rather easy (about 150 lines) to implement this protocol with REACTIVEML. Moreover, even if REACTIVEML is a synchronous language, we simulate the clock drift of the nodes.

4.1.3 Routing

As for the MAC layer, the routing protocols are also specific in sensor networks. Two of these are flooding and Directed Diffusion [21]. We implemented both. In flooding, each node receiving

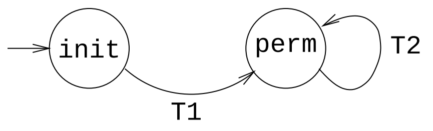


Figure 11. The automaton described by Lucky

a packet repeats it by broadcasting unless it had previously sent this packet. This mechanism is useful for the management of the network indeed some messages have to reach the whole network.

Directed diffusion is a data-centric routing that is used to collect data in sensor networks. In the network there is one (possibly several) node called sink that collects the data of the network. This routing protocol has three steps, see figure 10. (a) The sink first floods an interest message, which is a task description to the whole network, (b) the sensors set up gradients and (c) when a source has data for the interest, it sends the packet to the sink along the interest's gradient path.

4.1.4 Application

A sensor network is dedicated to a particular application. The whole protocol stacks depends on the application. In our example, the network that we simulate has to send an alarm to a specific node called the sink in case of danger. The danger is here related to the environment, in fact the role of the network is to avert when a toxic cloud is detected.

4.1.5 Environment

The environment is the source of (almost) all activity that occurs in the network. It is not realistic to have independent stimuli that activate the sensors [33]. A sensor network simulator has to include a model of the environment. Here, the model of the environment has been implemented using LUCKY.

LUCKY [22] is a programming language for the description of non deterministic reactive systems. It is a part of the LURETTE [31] tool box, an automatic testing tool for reactive programs.

A LUCKY program defines a set of input variables, a set of output variables and an automaton with constraints on transitions. The outputs generated respect the constraints that may involve the inputs and the previous values of the outputs. The execution of a LUCKY program is a synchronous system. At each step, the LUCKY process reads the inputs takes a transition where constraints can be satisfied and generates random outputs that satisfy the constraints.

In GLONEMO, the environment is a (toxic) cloud moving according to the direction and speed of the wind. The model consists in two processes, one for a two-dimensional wind, which does not vary a lot and an other for a cloud. The LUCKY code for the `wind` process is the following:

```
inputs { }
outputs {
    wind_x : float;
    wind_y : float;
}
start_node { init }
transitions {
 init -> perm // transition T0
   ~cond
     wind_x = 0.0 and wind_y = 0.0;

 perm -> perm // transition T1
   ~cond
     abs (wind_x - pre wind_x) < 1.0 and
     abs (wind_y - pre wind_y) < 1.0 and
     abs wind_x < 5.0 and abs wind_y < 5.0
}
```

This LUCKY program defines a two states automaton (see fig 11) with two output variables `wind_x` and `wind_y`. The constraints on the outputs are defined at the transitions. For those conditions, the keyword of the language in LUCKY is `~cond`. Here, transition `T0` sets the initial values of `wind_x` and `wind_y` to `0.0` and the transition `T1` guaranty that at each activation the values of the output variables are closed to their previous values.

The cloud is a disk whose center has the coordinates `cloud_x` and `cloud_y`. Similarly, it is defined by an automaton where `wind_x` and `wind_y` are the inputs and `cloud_x` and `cloud_y` the outputs.

```
inputs {
  wind_x : float;
  wind_y : float;
}
outputs {
  cloud_x: float;
  cloud_y: float;
}
start_node { init }
transitions {
 init -> perm // transition T0
   ~cond
     cloud_x = 0.0 and cloud_y = 0.0;

 perm -> perm // transition T1
   ~cond
     (if wind_x >= 0.0
      then ((cloud_x - pre cloud_x) >= 0.0
        and (cloud_x - pre cloud_x) <= wind_x)
      else ((cloud_x - pre cloud_x) <= 0.0
        and (cloud_x - pre cloud_x) >= wind_x))
    and
     (if wind_y >= 0.0
      then ((cloud_y - pre cloud_y) >= 0.0
        and (cloud_y - pre cloud_y) <= wind_y)
      else ((cloud_y - pre cloud_y) <= 0.0
        and (cloud_y - pre cloud_y) >= wind_y))
}
```

These LUCKY programs can be imported into REACTIVEML and turned into processes parameterized by their input and output variables. Parameters become REACTIVEML signals. The behavior of the process is to read the value associated to the input signals and to emit the value computed by LUCKY on the output signal at each step.

Let us illustrate it with the example of the cloud.

```
external.luc cloud_lucky
  {wind_x : float; wind_y : float;}
  {cloud_x: float; cloud_y: float;} = ["cloud.luc"]
val cloud_lucky :
  ('a, float) event * ('b, float) event ->
  (float, 'c) event * (float, 'd) event ->
  unit process
```

Here, we create a process named `cloud_lucky`. The inputs `wind_x` and `wind_y` must be signals of type *('a, float) event* such that the value associated to the signals are floats. The outputs `cloud_x` and `cloud_y` have type *(float, 'a) event* since the process emits values of type *float*. In the same way, we can create the process `wind_lucky`.

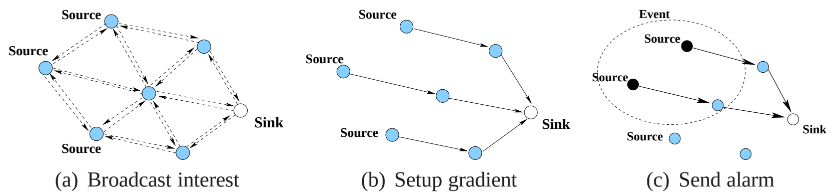


Figure 10. Routing: An example of directed diffusion.

To observe particular behaviors for the cloud without having to program them, it is useful for the user to be able to modify the could position during the simulation.

When the simulator is executed with a graphical interface, the `fan` process reads keyboard inputs and generates a particular wind. This process has the following interface:

```
val fan :
  (float, 'a) event * (float, 'b) event ->
  unit process
```

So, interactive simulations can simply be done by the parallel composition of the processes `wind_lucky` and `fan`. Winds produced by the LUCKY process and the `fan` are combined through the signals `wind_x` and `wind_y`:

```
  signal wind_x default 0.0 gather (+.) in
  signal wind_y default 0.0 gather (+.) in
  run (wind_lucky () (wind_x,wind_y))
  ||
  run (fan (wind_x,wind_y))
```

4.2 Benchmarks and Scalability

Sensor networks are huge systems composed by thousands or even millions of nodes. Thus, a simulator dedicated to sensor networks must be able to simulate such a high number of elements. In this section, we discuss about the capacity of GLONEMO to execute such networks. We measure both the time of the simulation and the memory usage.

GLONEMO focuses on the energy consumption. That is why a fine grain simulation is needed. Indeed, to model the energy consumption in an accurate way, a model of the hardware is introduced in the execution of the simulator. The time scale for that is really small comparing to the times involved at the network layers. In GLONEMO, the execution of one logical instant represents 10^{-2} seconds. Thus to simulate the behavior of a network during one hour, we need 360000 instants. For a 10000 nodes network, such a simulation takes about 11 hours. On figure 12(a), we printed the execution time of one single instant in terms of the number of nodes. This time appears to be linear with the number of nodes. For a 140000 nodes network, the execution time of one instant takes about 1.4 second. This is a long simulation but regarding the memory (fig 12(b)), this simulation takes only 700 Megabytes and can thus be done on everyday computers.

On figure 13, we plot the speed and memory of a given simulation with a fix number of nodes. The memory needed to run the simulation is constant (see fig 13(b)). This ensures that a simulation will not begin to swap in the middle of the execution. Moreover, the time taken to execute one instant is constant during the whole simulation (see fig 13(a)). This is important to run long simulations.

GLONEMO, written in REACTIVEML, is able to simulate in an accurate way more than 100000 nodes.

5. Related Works

Because network simulators are extensively used in the network community research, many relevant simulators have been developed. Let us describe the distinguishing features of some of them.

In 2000, Breslau et al. [12] defend the need of a single simulator for the research community. This was the VINT project leading to the NS-2 simulator [1]. Indeed, NS-2 is one of the most popular simulator in the research community. It is a packet-level simulator that was first designed for wired networks. NS-2 is a discrete event simulator. The interest of having one single simulator is to enable comparison between different protocols without the need to implement the protocol we want to compare with. Indeed, NS-2 offers a large protocol library. However, even if NS-2 provides several levels of abstraction (four according to [12]), it is more effective to implement the exact level of abstraction needed. This is why some people still write stand-alone simulators. Moreover, NS-2 is not really scalable and is convenient for simulating a few hundred nodes only.

To overcome the limitation in scalability of NS-2, people propose *Parallel Discrete Event Simulation* [17] where the simulator is distributed among several machines. GTNetS [32] is developed with this paradigm. This is a complementary approaches to a centralised implementation as provided in REACTIVEML. The two dedicated simulators implemented in REACTIVEML appear to be scalable enough to run this way.

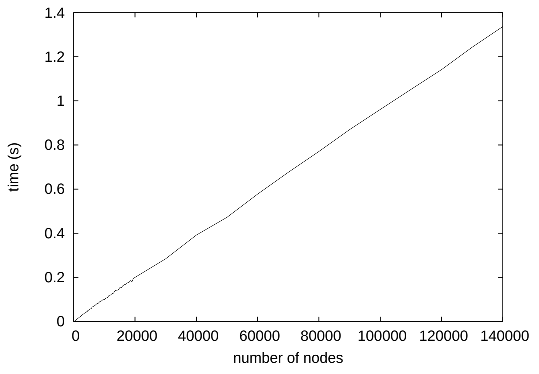
Sensor networks are new kind of ad hoc networks that interest the research community. Those networks have different characteristics and new constraints, thus new simulators are needed. Because one of the key issue in sensor networks is the power consumption, people began to develop simulators that take into account the energy consumption. Avrora [37] and Atemu [29] are cycle-accurate simulators (RTL level). With that level of detail scalability is probably hopeless.

Finally, it would have been difficult to implement the two simulators in a synchronous language like LUSTRE [20], ESTEREL [7] or SIGNAL [19] for at least two reasons: the use of complex data structures that are shared between the reactive part and the computational one, and the dynamic creation that is not allowed in these languages.

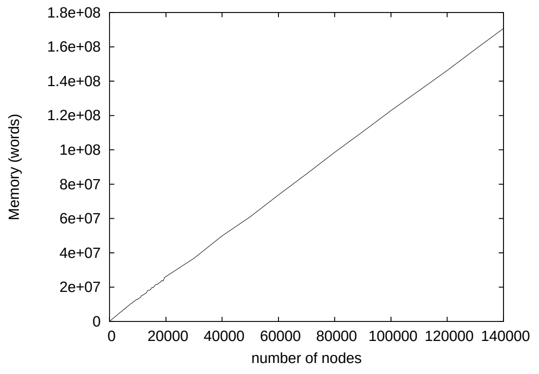
6. Conclusion and Perspectives

From the observation that generic network simulators are not always satisfactory and that users still develop their own simulators from scratch, we propose the use of the reactive model to program them. This model is dedicated to the programming of systems with a lot of parallel processes and communications and this is typically the case of network simulators.

Two different simulators have been considered: a coarse-grained one (ELIP) and a fine-grained one (GLONEMO). Both simulators, with the graphical interface, were defined in less than 2000 lines of REACTIVEML. It is easy to define the data structures describing nodes and packets. Moreover, the reactive model appeared to be

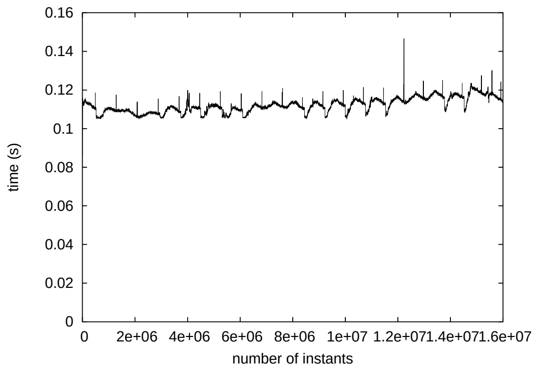


(a) Simulation times of one logical instant.

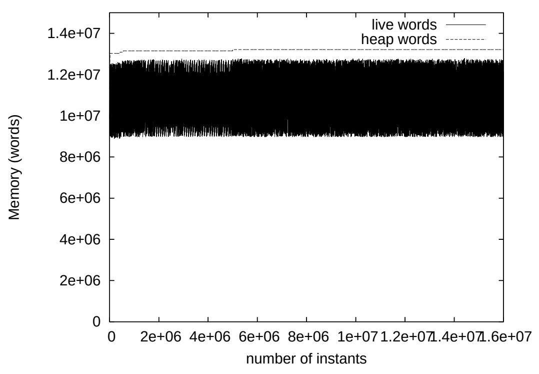


(b) Memory usage.

Figure 12. Simulation times and memory as a function of number of nodes.



(a) Simulation times of one logical instant.



(b) Memory usage.

Figure 13. Simulation of 10000 nodes during 20 days.

well adapted for both the description of mobility in ELIP and to the modular description of different the protocol layers in GLONEMO. Finally, the underlined model of concurrency of reactive programs states that all the node of the network react synchronously during a reaction. This makes the correspondence between the logical time and the simulation time.

The link between REACTIVEML and LUCKY allowed to simulate the physical external environment in GLONEMO. This point is particularly important for sensor networks since a naive model of the environment does not give relevant simulation results.

The simulators were efficient enough and robust to obtain the useful simulation metrics [4, 5, 6]. It is clearly possible to develop more efficient simulators than ELIP and GLONEMO but it appears that there were a good compromise between the development time and the simulators efficiency.

This works offers several perspectives, some concerning the simulators by themselves and some concerning REACTIVEML. For the GLONEMO simulator, it would be interesting to have several levels of simulation: a fine-grained simulator in order to have an accurate estimation of the energy consumption and then a faster simulator that gives information about the higher layers. Understanding how to write such a multi-level simulator is a challenging direction.

A natural extension of the language is to equip it with a debugger. A top-level *à la* OCAML and inspired by the REACTIVE SCRIPTS [8] has been implemented. It allows to define interactively REACTIVEML programs and to control the execution. Defining a more conventional debugger for a reactive language is largely an open problem.

Another direction is the use of formal validation techniques and tools for reactive programs. Technically, this means extracting models in a form usable by the validation tools. For GLONEMO, for example, we would like to prove two kinds of properties. The first is the validation of the *abstractions* that are needed for the model to be of a reasonable complexity. For instance, we think that we should never include in the model a full description of the hardware, at the abstraction level that is needed for precise energy evaluations, i.e., the RTL level. But if we include an abstraction of it, we should *prove* that: 1) it is indeed an abstraction of the real hardware, and 2) the composition with the rest of the model preserves this abstraction. The second kind is the verification of *global* properties such as: *after time T, the system still has more than x % of the nodes alive*.

Verifying reactive program with dynamic creation of processes is still largely an open problem. Establishing close relations between the reactive model and process algebra could give some useful insight [3].

The key perspective is to use REACTIVEML not only to simulate ad hoc networks but also others embedded systems. There is a first experiment with the simulation of a gyroscopic system. This example is taken from the avionic industry, it deals with the treatment of position variations of an airplane [26]. An interesting point here is how to extract the embedded (real-time) software from the REACTIVEML program.

References

[1] The Network Simulator - ns-2.

[2] Raúl Acosta-Bermejo. *Rejo - Langage d'Objets Réactifs et d'Agents*. PhD thesis, Ecole des Mines de Paris, 2003.

[3] R.M. Amadio and F. Dabrowski. Feasible reactivity for synchronous cooperative threads. In *Extended abstract presented at the workshop Expressiveness in Concurrency*, San Francisco, September 2005.

[4] F. Benbadis, M. Dias de Amorim, and S. Fdida. 3P: Packets for positions prediction. In *Proceedings of IEEE INFOCOM students workshop'05*, Miama, FL, USA, 2005.

[5] F. Benbadis, M. Dias de Amorim, and S. Fdida. Dissémination prédictive des coordonnées pour le routage géographique basé sur l'âge. In *Proceedings of CFIP 2005 Conference*, Bordeaux, France, 2005.

[6] F. Benbadis, M. Dias de Amorim, and S. Fdida. ELIP: Embedded location information protocol. In *Proceedings of IFIP Networking 2005 Conference*, Waterloo, Canada, 2005.

[7] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[8] F. Boussinot and L. Hazard. Reactive scripts. In *RTCSA '96: Proceedings of the Third International Workshop on Real-Time Computing Systems Application (RTCSA '96)*, page 270, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Frédéric Boussinot. Concurrent programming with Fair Threads: The LOFT language, 2003.

[10] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *Software Engineering*, 22(4):256–266, 1996.

[11] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes tool box : A reactive java framework. *Software Practice and Experience*, 28(4):1531–1550, 1998.

[12] Lee Breslau, Deborah Estrin, Kevin R. Fall, Sally Floyd, John S. Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, 2000.

[13] Christian Brunette. *Construction et simulation graphiques de comportements: le modèle des Icobjs*. PhD thesis, Université de Nice-Sophia Antipolis, 2004.

[14] Christian C. Enz, Amre El-Hoiydi, Jean-Dominique Decotignie, and Vincent Peiris. Wisenet: An ultralow-power wireless sensor network solution. *IEEE Computer*, 37(8):62–70, 2004.

[15] EPFL. Network in A Box.

[16] Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. Network visualization with nam, the vint network animator. *Computer*, 33(11):63–68, 2000.

[17] Richard M. Fujimoto, Kalyan S. Perumalla, Alfred Park, Hao Wu, Mostafa H. Ammar, and George F. Riley. Large-scale network simulation: How big? how fast? In *MASCOTS*, page 116. IEEE Computer Society, 2003.

[18] Matthias Grossglauser and Martin Vetterli. Locating nodes with EASE: Last encounter routing in ad hoc networks through mobility diffusion. In *Proceedings of IEEE Infocom*, March 2003.

[19] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Lemaire. Programming real-time applications with signal. *Proc. of the IEEE*, 79(9):1321–1336, September 1991.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[21] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67, 2000.

[22] Erwan Jahier and Pascal Raymond. The lucky language reference manual. Technical report, Unité Mixte de Recherche 5104 CNRS - INPG - UJF, 2004.

[23] Xavier Leroy. The Objective Caml system release 3.09. Documentation and user's manual. Technical report, INRIA, 2006.

[24] Louis Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[25] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[26] Lionel Morel and Louis Mandel. Executable contracts for incremental prototypes of embedded systems. Submitted to publication, 2006.

[27] OPNET Modeler. `http://www.opnet.com`.

[28] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[29] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, and John S. Baras. ATEMU: A Fine-grained Sensor Network Simulator. *Secon*, 2004.

[30] P. Raymond and Y. Roux. Describing non-deterministic reactive systems by means of regular expressions. In *Synchronous Languages Applications and Programming - SLap02*, volume 65.5. Electronic Notes in Theoretical Computer Science, 2002.

[31] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[32] George F. Riley. The georgia tech network simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12, New York, NY, USA, 2003. ACM Press.

[33] Ludovic Samper, Florence Maraninchi, Laurent Mounier, Erwan Jahier, and Pascal Raymond. On the importance of modeling the environment when analyzing sensor networks. In *Proceedings of International Workshop on Wireless Ad-Hoc Networks 2006 (IWWAN 2006)*, page 7, New York, United States, June 2006.

[34] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214. ACM Press, 2004.

[35] Avinash Sridharan, Marco Zuniga, and Bhaskar Krishnamachari. Integrating environment simulators with network simulators. Technical report, University of Southern California, 2004.

[36] Jean-Ferdinand Susini. *L'approche réactive au dessus de Java : sémantique et implémentation des SugarCubes et de Junior*. PhD thesis, Ecole des Mines de Paris, 2001.

[37] Ben L Titzer, Daniel K Lee, and Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. *Proceedings of IPSN*, 2005.