Simulation of Mobile Ad hoc Network Protocols in ReactiveML

Louis Mandel [1] Farid Benbadis [2]

Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
Paris, France

Abstract

This paper presents a programming experiment of a complex network routing protocol for mobile ad hoc networks within the ReactiveML language.

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, 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.

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 feasible. Experimental results show that the ReactiveML implementation is two orders of magnitude faster than the original C version; it was able to simulate more than 1000 nodes where the original C version failed (after 200 nodes) and compares favorably with the version programmed in NAB.

Key words: Network Simulation, Reactive Programming

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Mobile ad hoc networks are highly dynamic networks characterized by the absence of any physical infrastructure. They are composed of nodes which evolve concurrently and have to synchronize continuously with other nodes in order to route packets and to update their knowledge of the network topology. 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 only on the coordinates of their direct neighbors and of the destination [7]. This avoids the need for topology knowledge beyond one-hop.

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. Being highly dynamic, they must clearly be simulated (that is, programmed!) before any implementation start. The simulation is useful as a simple graphical observation of the network and in order to measure the route lengths and network overhead of a particular protocol. Of course, this simulation is memory and time consuming: the intrinsic complexity of a simulation step of a network of n nodes is $O(n^2)$ in memory (every node builds a location table for every node of the network) and is $o(n \times m)$ in time where m is the number of neighbors of a node. This means that the simulation will call for very efficient data-structures in order to be able to consider real-size networks, 1000 nodes being already an interesting limit.

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 [12] or OPNET [13] inappropriate. Indeed, NS2 has been originally designed for conventional (wired) networks and does not treat well wireless networks. In particular, it is only able to simulate small networks (1000 nodes network seems to be barely conceivable) whereas we consider large scale networks. Moreover, these two simulators need that the layers 1 to 3 be described while we are only interested in layer 3 (the network layer). Finally, these two simulators appear hard to use. This is why we decided to program the protocol directly in a conventional programming language (here C). Nonetheless, getting an efficient programming of age and position based protocols routing was more than an issue in such a language.

In this paper, we show that the *synchronous reactive model* introduced by Boussinot [5] 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

[1] Email: `louis.mandel@lip6.fr`
[2] Email: `farid.benbadis@lip6.fr`

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 [1,4,6,14]. 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 implementation has been done in ReactiveML [3], an embedding of the reactive model inside a statically typed, strict functional language [10]. ReactiveML provides reactive programming constructs with most of the features of Ocaml. Reactive constructs give a powerful way to describe the dynamic part of the system whereas the host language Ocaml provides data-structures for programming the algorithmic (combinatorial) part. Moreover, it provides an efficient execution scheme for reactive constructs which made the simulation of real-size examples feasible.

Experimental results show that the ReactiveML implementation is two order of magnitude faster than the original C version which was made prior to the ReactiveML implementation; it was able to simulate more than 1000 nodes where the original C version failed (after 200 nodes); it appeared to be robust (we ran it for 20 days without any memory increase nor degradation); it is twice faster than the ad hoc version programmed in NAB [11].

The purpose of this paper is not to present a new protocol [4] but more 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 [1,6,14].

It would be difficult to implement the simulator in a synchronous language like Lustre [9], Esterel [3] or Signal [8] 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.

The paper is organized as follows. Section 2 presents briefly the principles to the routing protocol we have considered. Section 3 describes the ReactiveML implementation. The language is very young and the paper can thus be considered as a tutorial introduction of the language through a real example. 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 experimental results. Section 5 discusses several possible extensions of the implementation and we

[3] The distribution can be accessed as: `http://www-spi.lip6.fr/~mandel/rml`.

[4] The protocol is described in [2] where numbers have been obtained with the ReactiveML implementation.

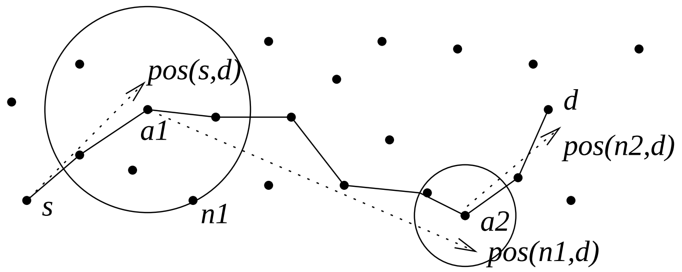


Fig. 1. Routing a packet from s to d: anchor nodes a_1 and a_2 refine estimation of d's position.

conclude in section 6.

2 Age and Position Based Routing

The main principle of Age and Position Based (APB) routing protocols is that each node may have an information about each other node 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) [7] routing method, where a source node s needs to communicate with a destination d, as follows: [5]

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 $age(n_i, d) \leq age/2$;
  $age := age(n_i, d)$;
  While not the closest node of $pos(n_i, d)$ do
    forward toward $pos(n_i, d)$
  done;
  $i := i + 1$;
  $a_i :=$ the closest node of $pos(n_i, d)$
done

where a_i are anchor nodes, [6] $pos(n_1, n_2)$ 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 [7], uses the

[5] For more details about EASE, see [7]

[6] Anchor nodes search for a better estimation of destination position than the one included in the packet.

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) [2], uses also the encounter between nodes, but disseminates nodes locations in data packets. 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 [7], 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 Implementation in ReactiveML

3.1 ReactiveML Survival Kit

REACTIVEML is built above OCAML such that every OCAML program (without objects, labels and functors) is a valid REACTIVEML program and REACTIVEML code can be linked to any OCAML library.

A program is a set of definitions. Definitions introduce, like in OCAML, types, values or functions. We illustrate the syntax with the example of positions. We define 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 pos42 = { x = 4; y = 2 }
let distance2 p1 p2 =
  (p2.x - p1.x) * (p2.x - p1.x) + (p2.y - p1.y) * (p2.y - p1.y)
```

This is regular OCAML code. 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 timeless.[7] Let us consider the process

[7] In circuit terminology, processes are *sequential* functions whereas OCAML functions are

`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"
```

Then to execute a process we use the `run` primitive: `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 `suspend_resume` that controls the instant where a process is executed.

We first define a process `sustain` parameterized by a signal `s`. `sustain` emits the signal `s` at every instant.

```
let process sustain s = loop emit s; pause end
```

The `loop/end` construct is the unbounded iteration.

`switch` is a process parameterized by two signals, `s_in` and `s_out`. Its behavior is to start the emission of `s_out` when `s_in` is emitted and to sustain this emission while `s_in` is absent. When `s_in` is emitted again, the emission of `s_out` is stopped and the process returns in its initial state.

```
let process switch s_in s_out =
  loop
    await immediate s_in;
    pause;
    do run (sustain s_out) until s_in done
  end
```

We can see in this example the `await` which stops the execution of the process until `s_in` is emitted, and the preemption construct `do/until` which kills, at the end of the instant, its body when the signal `s_in` is present.

We define now the process `suspend_resume` parameterized by a process `p` and a signal `s`. This process awaits the first emission of `s` to start the execution of `p`. Then, each emission of `s` alternatively suspends the execution of `p` and resumes it. We implement this process with the parallel composition of (1) a `do/when` construct that executes `p` only when the signal `active` is present and (2) the execution of a switch that controls the emission of `active` with the signal `s`.

```
let process suspend_resume p s =
  signal active in
  do run p when active
  ||
```

considered to be *combinatorial*.

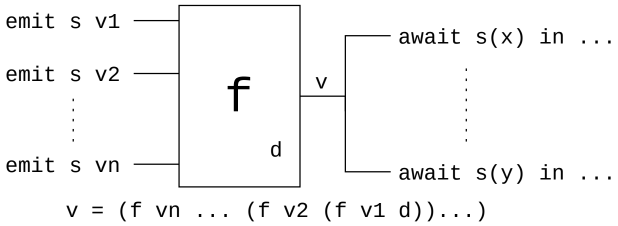


Fig. 2. Multi-emission on signal `s`, combined with function `f`, gives the value `v` at the next instant.

```
run (switch s active)
```

Notice that `suspend_resume` is an example of higher-order process since it takes a process `p` as a parameter.

ReactiveML also provides valued signals. They can be emitted (`emit` *signal value*) or awaited to get the associated value (`await` *signal* `(`*pattern*`)` `in` *expression*). Valued signals call for a particular treatment in case of multi-emission. 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` *expression*

The behavior of multi-emission is illustrated in Fig. 2. We assume signal `s` declared with the default value `d` and the gathering function `f`. If values `v1`, ..., `vn` are emitted during an instant, then all the `await` receive the value `v` at the next instant. [8]

For example, if we want to define a signal `sum` that computes the sum of the emitted values, then we can write:

```
signal sum default 0 gather (+) in ...
```

In this case, the program `await sum(x) in print_int x` awaits the first instant in which `sum` is emitted and then, at the next instant, prints the sum of the values emitted.

Another very useful signal declaration is the one that collects all the values emitted during the instant:

```
signal s default [] gather fun x y -> x :: y in ...
```

Here, the default value is the empty list and the gathering function builds the list of emitted values.

We stop this short introduction here. Various examples of ReactiveML programs can be accessed at www-spi.lip6.fr/~mandel/rml.

[8] `v = (f vn ... (f v2 (f v1 d))...)`

3.2 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. The a's entry in n's table, looks like this: $[ID_a, pos(n,a), date(n,a)]$ where $pos(n,a)$ is an estimation of a's position, and $date(n,a)$ indicates when n 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` its current position and, `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 LE 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.

3.3 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.

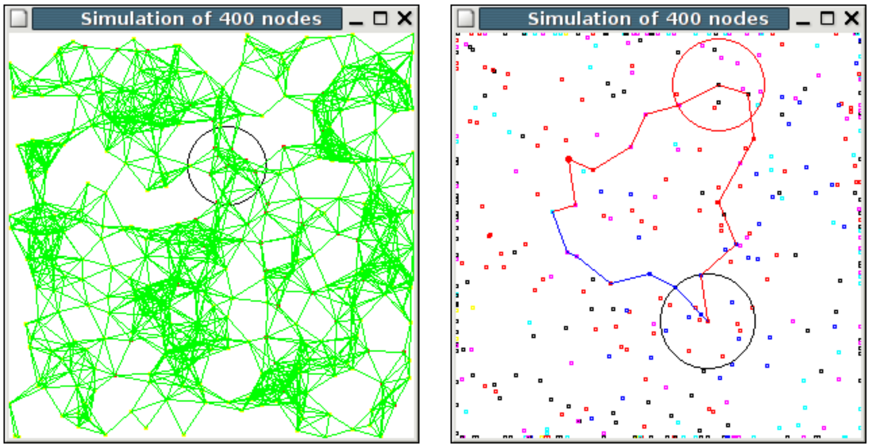
The behavior of each node is composed of three steps. A node (1) moves, (2) discovers its neighborhood, and (3) routes packets. These steps are combined in a process node [9] which is parameterized by the initial position of the node pos_init, a function move that computes its next position, and a function 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;

    (* 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. [10] At

[9] http://www-spi.lip6.fr/~mandel/rml/slap/simulator/node.rml.html

[10] A screenshot is given in Fig. 3



(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.

Fig. 3. Screenshots of the simulator graphical interface.

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.

3.3.1 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.

3.3.2 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. Moreover, 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.

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. 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. 4. In the one hand, n sends its position on the signals present in the 4 squares touched by its radio transmission (the 4 gray squares in this figure). In the other hand, nodes a and c transmit their position on the signal present in 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 present in the square where n is located.

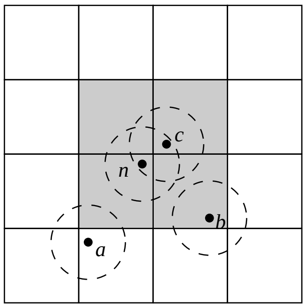
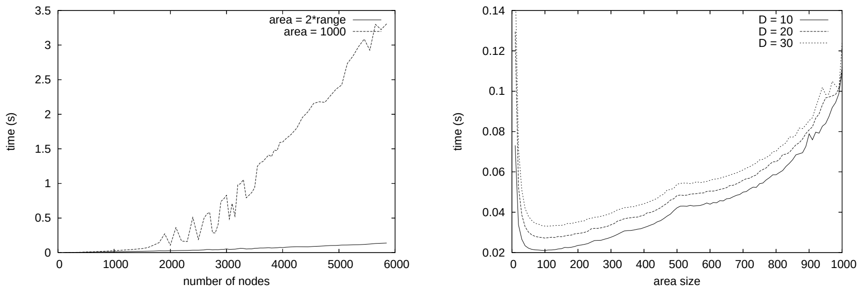


Fig. 4. Topology split into multiple squares. Node n emits its position on the gray squares, while it listens on the one it is located.



(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.

Fig. 5. Simulation times for neighborhood discovery.

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:

```
let (i,j) as local_area, neighbor_areas =
  get_areas self.pos.x self.pos.y
in
List.iter
  (fun (i,j) -> emit hello_array.(i).(j) self)
  (local_area::neighbor_areas);
await hello_array.(i).(j) (all) in
self.neighbors <- get_neighbors self all;
```

Fig. 5 shows the effect of the area split on execution time. In Fig. 5(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 1000 nodes the second method is 2 times faster than the first one. Then for 2000 nodes it is 4 times faster and for 5000 nodes it is more than 20 times faster.

We focus now on the second method, which is more appropriate. As we can see in Fig. 5(b), the execution time depends heavily on the area size. In this figure, which represents times required for simulating a 2000 nodes topology using three different densities,[11] we observe that dividing the topology in a big number of squares is not efficient. In this case, each node emits its position on a large number of signals, which requires resources. In 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 both densities simulated.

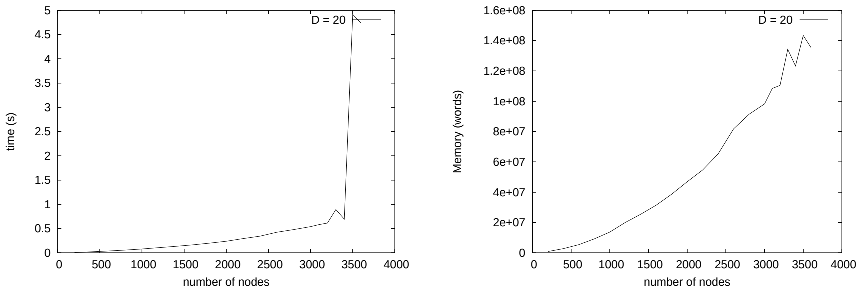
3.3.3 Routing

The last step in a node execution is the packets routing, which is described in section 2.[12] 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 at light 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 guaranties that this action is not interruptible such that there is no need to protect the access to share data like in the thread model.

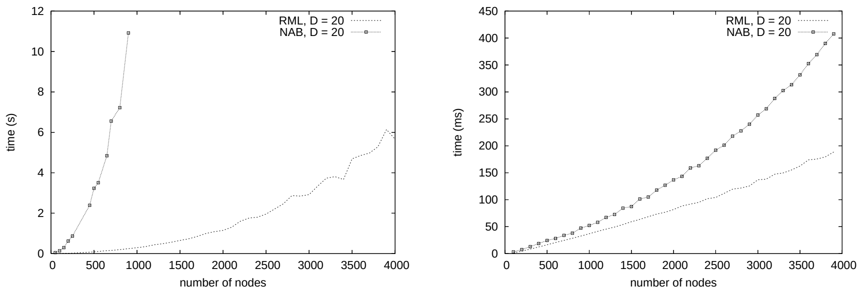
[11] We recall that the density represents the average number of nodes per coverage region, which depends on the nodes radio range.

[12] http://www-spi.lip6.fr/~mandel/rml/slap/simulator/routing.rml.html



(a) Simulation times. (b) Memory usage.

Fig. 6. Simulations depending on the number of nodes with a topology density D=20.



(a) With packets emission. (b) Without packet emission.

Fig. 7. Comparison of simulation times, between NAB and ReactiveML simulator, depending on the number of nodes with a topology density D=20.

4 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.

First, we analyze our program capability to simulate large networks. Fig. 6(a) represents simulation times depending on number of nodes. We observe that at about 3000 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. We can see in Fig. 6(b) that 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. 7(a) represents the execution time for a simulation

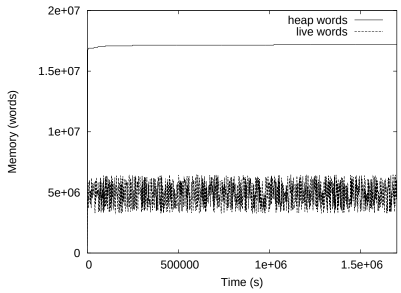


Fig. 8. Memory usage during 20 days.

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. 7(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.

The last point on which we want to put emphasis is the memory usage stability. Figure 8 shows the size of the heap and the number of live words in the heap during the execution of a simulation. We can notice that during 20 days of execution, the size of the heap is constant.

5 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`:

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

Figure 9, we present the memory usage of a simulation that removes a node at each instant. It shows that the garbage collector works on processes.

A more interesting point is the dynamic creation of processes. In ReactiveML, dynamic creation is made through recursion. We define the recursive

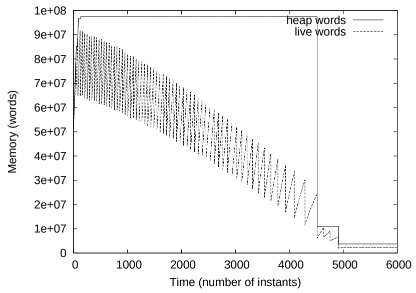


Fig. 9. Memory usage of a simulation that removes a node at each instant.

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.

6 Conclusion

In this paper, we addressed the problem of simulating mobile ad hoc networks. We believe that existing tools are not totally satisfactory because they are both difficult to use and not adapted to large scale and/or highly dynamic scenarios.

We decided to implement our simulator in ReactiveML because this language allows a large number of parallel processes to be executed. Moreover, the fact that ReactiveML is based on Ocaml allowed us to easily define complex functions like moving or routing. The expressive broadcast communication makes it simple to modify a naive method for neighborhood discovery into an efficient one. ReactiveML compilation techniques, associated to the Ocaml runtime, results in efficient simulations.

Efficiency is a central aspect in ReactiveML. The design and semantics have been tuned for that purpose. For example, the parallel construct is deterministic but the evaluation order is not specified, giving the opportunity to execute parallel branches in any appropriate order. Signals are efficiently represented (as Ocaml values are, and can thus be automatically garbage collected) and the access to a signal is done in constant time. There is no busy waiting during run-time: suspended processes (*e.g.*, awaiting for some condition or signal to be emitted) are only waked-up when necessary. Several experiments have shown that the execution scheme of REACTIVEML competes with the best existing execution schemes for reactive languages [1,6,10].

Because the protocol was described in a programming language (and not in a dedicated simulator), it is easy to extend and change some of the internal data-structures involved in the routing protocol. Moreover, the graphical observation itself could be programmed with reactive constructs (*e.g.*, suspending the simulation, drawing the topology). That is, observing a reactive program is also a reactive problem.

The perspective of this work is to simulate another kind of network (sensor network), and take into account other dynamic aspects like energy consumption. The goal of the simulator is to understand the interactions between the different protocol layers in order to increase the lifetime of the whole network. In this simulator we can find dynamic aspects, the need of scalability and also the simulation of the MAC layer.

References

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

[2] Farid Benbadis, Marcelo Dias de Amorim, and Serge Fdida. ELIP: Embedded location information protocol. In *IFIP Networking 2005 Conference*, 2005.

[3] 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.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. Technical report, Laboratoire d'Informatique de Paris 6, 2005. Available at http://www-spi.lip6.fr/~mandel/rml.

[11] Network in A Box. http://nab.epfl.ch/.

[12] The network simulator. http://www.isi.edu/nsnam/ns.

[13] OPNET Modeler. http://www.opnet.com.

[14] 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.