Wireless Netw (2010) 16:1525–1539
DOI 10.1007/s11276-008-0134-1

Hybrid simulation of Sensor and Actor Networks with BARAKA

Thomas Halva Labella · Isabel Dietrich · Falko Dressler

Published online: 11 September 2008
© Springer Science+Business Media, LLC 2008

Abstract We present BARAKA, a new hybrid simulator for Sensor and Actor Networks (SANETs). This tool provides integrated simulation of communication networks and robotic aspects. It allows the complete modelling of co-operation issues in SANETs including the performance evaluation of either robot actions or networking aspects while considering mutual impact. This hybrid simulation enables new potentials in the evaluation of algorithms developed for communication and co-operation in SANETs. Previously, evaluations in this context were accomplished separately. On the one hand, network simulation helps to measure the efficiency of routing or medium access. On the other hand, robot simulators are used to evaluate the physical movements. Using two different simulators might introduce inconsistent results, and might make the transfer on real hardware harder. With the development of methods and techniques for co-operation in SANETs, the need for integrated evaluation environment increased. To compensate this demand, we developed BARAKA.

Keywords Sensor and Actor Networks · Simulation · Rigid-body simulation · OMNeT++

T. H. Labella (✉) · I. Dietrich · F. Dressler
Autonomic Networking Group, Department of Computer Science 7, University of Erlangen-Nuremberg, Martensstr. 3, 91058 Erlangen, Germany
e-mail: hlabella@ulb.ac.be

I. Dietrich
e-mail: isabel.dietrich@informatik.uni-erlangen.de

F. Dressler
e-mail: dressler@informatik.uni-erlangen.de

1 Introduction

Sensor and Actor Networks (SANETs) are challenging research objects. The field was born from the intersection of research on Wireless Sensor Networks (WSNs) and mobile robotics. The result is an heterogeneous system, made of fixed nodes capable only of sensing the environment (the *sensor nodes*, sometimes called also *motes*), and *mobile nodes* that are also able to change it (the robots).[1] Akyildiz et al. [1] cite as unique features of a SANET: node heterogeneity, real-time requirements, different deployment strategies for motes and robots, mobility and co-ordination paradigm—mote/robots and not only mote/sink as typically in Wireless Sensor Network. Research issues include power management, routing, co-ordination algorithms, design, and many other topics. Many algorithms and methods have been proposed to optimise the efficiency of the networking part as well as of the co-ordination between single nodes [2]. Examples are the optimised navigation of robot systems using a WSN [3], communication architectures for mobile SANETs [4], and efficient actuation control in SANETs [5].

All these approaches must be carefully evaluated in order to show the feasibility of the promised capabilities. Usually, a simulation environment is preferred to a lab setup. Even if advances in electronics allow us to experiment with compact hardware sensors and with robots in real environments, simulation is still useful if we want to test larger networks, or if preliminary experiments with real objects might risk to break them.

When we started our research in this area (we present our work in [6]), we discovered a major problem. There is no comprehensive tool for integrated SANET simulation. In

[1] We use the word *agent* in the following when we refer to either the entities of a SANET.

other words, there is no simulator that takes equally care of both the networking of the nodes and the realistic movements of the robots (see our overview of up-to-date simulation tools in the next section). We think that an integrated simulation is necessary for the following reasons. First, it allows to study deeper form of interactions between robots and motes. Second and most importantly, it reduces the gap between simulation and reality. An integrated detailed simulator allows the experimenter to develop algorithms in simulation and to immediately use them also on real hardware.

We would like to stress this point a bit more. A SANET developer usually has to focus both on the communication protocols between agents the movements of the robots. The developer could use two different simulation tools if this two aspects were independent, but this is not the case in SANETs. When developing protocols, one tends to simplify and underestimate the effects of robots' movements. The networking community often uses real-world traces to tune the protocols they are studying. This approach makes no sense in SANETs, since the developer has control on the movements of the robots, which can be foreseen since their behaviours are known. Roboticians tend also to simplify and underestimate the role of the network condition on the outcome of robots' behaviours. For instance, it is often assumed that messages can reliably sent in nearly no time. The effects of errors or delays on the resulting behaviours are seldom taken into account. A further argument against the use of two different simulators is that it might introduce inconsistent results between the two simulators. This is because one simulator takes particular care only of some aspects and approximates some others, that might be fundamental for the other simulator.

Let us take an often-cited example in the SANET literature: a robot moves to a part of the network which is congested or where a node has just failed, in order to work as extra gateway. However, the decision if and where to go depends, from the robot's point of view, on several factors: how far is the place, if the robot has enough energy to reach the place, if there is an obstacle on the way, if it is possible to bypass the obstacle, if the failing part of the network is important for the current work of the other robots, if there are other tasks that are more important, and so on. Just to mention one, the problem of recognising an obstacle is a hard problem in itself for autonomous robots.[2]

[2] Robots can sense obstacles using several devices: infra-red sensors, lasers, radars, cameras, bumpers, etc. Each of them has some advantages and disadvantages. Infrared-sensors, for instance, are cheap but work at short range and cannot return the detailed shape of an object. It might difficult to tell an obstacle, which has to be avoided, from a target, which has to be reached. Cameras can give more detailed information, but image processing requires a lot of computational power. It is possible to combine data coming from different sensors to overcome such problems, but this increases the complexity of sensor data elaboration. The reader can refer to [7] and [8] for some examples.

We think this is a strong argument for the use of comprehensive tools for SANET development. Accordingly, we decided to create BARAKA, the simulator we are going to present in this paper.

BARAKA now allows us to create realistic physical environments in which we can model the robot systems and their behaviour as well as the communication protocols and corresponding properties in a single simulation setup. Basically, BARAKA is built upon OMNeT++, a well known network simulation tool, and Open Dynamics Engine (ODE), a library used to simulate rigid-body physics.

The main contribution of this paper is to present a new integrated simulation tool for SANETs. BARAKA features the following characteristics:

- integrated simulation of communication networks and robotics
- complete modelling of co-operation issues in SANETs
- performance evaluation of either robot actions or networking aspects considering mutual impact.

The rest of the paper is organised as follows. Section 2 describes network simulators and robot simulators. We describe OMNeT++ and ODE, which we used to build BARAKA, in Sects. 3 and 4, respectively. In Sect. 5, we present our new SANET simulator in more detail. Section 6 shows a case study to demonstrate the capabilities of BARAKA using a comprehensive set-up. Section 7 concludes the paper.

2 Simulators

A look at the current status of network and robot simulators helps to understand the need of a simulator like BARAKA. The following sections describe how network and robot simulations usually take place, and why they are not enough for SANETs.

2.1 Network simulators

Network simulators are typically used to study the interactions between entities (such as routers, links or packets) in communication networks. Because of the discrete nature of the simulated entities, network simulations are most efficiently carried out as *discrete event simulations* [9]. Discrete event simulators assume that a system can be represented by a set of state variables. The variables change values only at a countable number of points in time. The simulator maintains a set of future events (such as message arrivals or timer firings) and processes this set one event at a time. It starts with the earliest event in the list and continues with the following ones. The simulation

times flows with the time associated to the events. It might not advance if two events are concurrent, or advance with big steps if two events are separated in time.

A large number of network simulation tools are available. Among the more well-known and popular tools, there are the commercial simulators OPNET[3] and Qualnet,[4] and the free open source simulators ns-2[5] and OMNeT++.[6]

Many simulators come with a number of ready-to-use protocols, mostly including the common Internet protocols, and often also a selection of protocols for ad hoc networks (such as DSR [10] or AODV [11]). The simulators can model wired as well as wireless connections. Mobile nodes are simulated specifying the parameters of the physical transmission devices (e.g., the transmitting power) and a mobility model of the nodes.

The performance of simulation programs is the topic of a lot of ongoing research. Fjord et al. discussed the scalability of simulations and investigated parallel simulations as a measure to counteract the performance problems of many simulation programs [12]. Another approach is discussed by Breslau et al. [13]. They use several levels of abstraction to adjust the trade-off between detail and performance. Heidemann et al. analyse the effect of varying levels of detail in network simulations [14]. They emphasise that an inappropriate level of detail can lead to very slow simulation execution, but also to misleading or incorrect results.

Pawlikowski et al. describe several methods to increase the credibility of network simulations [15]. They point out that the selection of an appropriate random number generator and appropriate output data analysis are very important not only for the credibility, but also for the outcome of a simulation.

With the rise of sensor networks, many simulators have focus more and more on topics such as energy consumption or interactions of the sensors with the environment. A brief overview of some new simulators and the problems encountered in sensor network scenarios is provided by Sundresh et al. [16].

Simulation results depend also heavily on the mobility model used. There are two types of mobility models: traces and synthetic models [17]. Traces are real data collected in real networks. Synthetic models attempt to represent the real behaviour of a network without using traces. Examples of synthetic models are the Random Waypoint model, the Gauss–Markov model, and the Nomadic Community model. These and many more were well reviewed by Camp et al. [17] and by Bai and Helmy [18].

[3] http://www.opnet.com/

[4] http://www.scalable-networks.com/

[5] http://www.isi.edu/nsnam/ns/

[6] http://www.omnetpp.org/

The use of models is sound when dealing with mobile networks whose nodes follow unknown behaviours. In case of SANETs however, the designer of the system does know the behaviours of the nodes. It is more reasonable then to use directly these behaviours instead of trying to approximate them with synthetic model or with traces. The latter might not be yet available at development time and it will always tend to produce unrealistic patterns as direct interactions between the communication and the co-ordination parts are not possible. Using directly the nodes' programmed behaviours has also the advantage of reducing the gap between simulation and real world. Additionally, one could program the network to exploit any behavioural pattern that might be typical for the nodes.

Programming robots' behaviours is however not straightforward. The programmer usually needs a good knowledge of the environment and its dynamics. Robot simulators take care of this by simulating realistic environments. None of the network simulators that we know supports the simulation of realistic environments, which include nodes mobility based on their physical construction and terrain characteristics.

2.2 Robot simulators

Robot simulators are used to implement control algorithms for existent pieces of hardware. It is usually preferred to start the implementation of any algorithm in simulation because it is safer, and there is no risk to break a robot or to lose its control. Because of this, the target of the simulator is to ease the transfer of any program from simulation to real robots.

A robot is nowadays a complex object which can be simulated at several level of details. One could be interested for instance in simulating the working of the engines connected to the wheel, or the analysis the pictures taken from the camera, or the mechanical stress of a arm-like actuator. We suppose in this paper that the designer of the SANET is mostly interested in the development of the control algorithm of the robots. The control algorithm is the part of the robot's software that receives as input the sensors' data (raw or elaborated) and sends as output commands to the actuators.

A simulator used for the development of a control algorithm needs obviously to be able to simulate the sensor readings and the actuator effects on the robots and in the environment. A good simulation of sensor readings can take place if the environment is well simulated in details.

Nowadays, every desktop computer is powerful enough to accurately simulate the robot and its environment. A number of simulators have been developed that can simulate the world physics. One of the most used is the commercial simulator Webots.[7] The world physics

[7] http://www.cyberbotics.com/

simulation is based on the ODE library (Sect. 4). ODE is not the only solution for rigid-body simulation. There are other commercial libraries available, such as Vortex[8] and Havok.[9]

The accurate environment simulation is a recent trend in robot simulators. The RoboCup Simulator,[10] used for simulated football matches during the RoboCup [19] competitions, began with a two dimensional environment and has recently added the third dimension and the simulation of players' movements.

The focus of robot simulators is an accurate simulation of a robot's behaviour. They usually use the API that is going to be used on the real hardware (e.g., to get the value of the infrared sensors or to set the speed). They do not consider the problem of modelling the networking of the robots. Although there are simulators that run *over* a network, like Player/Stage [20], they do not *simulate* the network.

Most of the control algorithms developed for robotics use quite simple communication schemata for co-ordination: blackboard, or point to point communication. When using a blackboard, which is the common solution in robotics, each robot is aware of every message that other robots have sent. This kind of communication is important for well known algorithms, like ALLIANCE [21]. While blackboard communication is quite easy to implement in a simulator, researchers in robotics usually do not realise that it requires to broadcast, possibly by flooding, all the messages in the network. Additionally, effects like delays and congestion can hardly be simulated with a robotics simulator and their effect on the robots' performance might be underestimated. The outcome is that the control algorithms might work well in simulation, but fail when used in a real network.

3 OMNeT++

OMNeT++ [22] is a discrete event simulator. It simulates modules, which can send messages to each other through communication channels. Modules connected together form a network. Modules can be compound, made of several sub-modules which form a sub-network between them.

Modules, channels and messages are implemented as C++ objects. Each message represents an event and is stored in the scheduler of OMNeT++ (also called the future event list). The simulator, after having initialised the modules, takes the first event in the list and delivers it to its destination. The delivery occurs by calling a method of the module and giving the message as parameter. Modules can send messages to others or to themselves. In this case, they mostly simulate internal timers. A delivery time is associated to each message and determines its position in the scheduler list. The simulation continues till there are no more messages to be delivered or until a specified time limit has been reached.

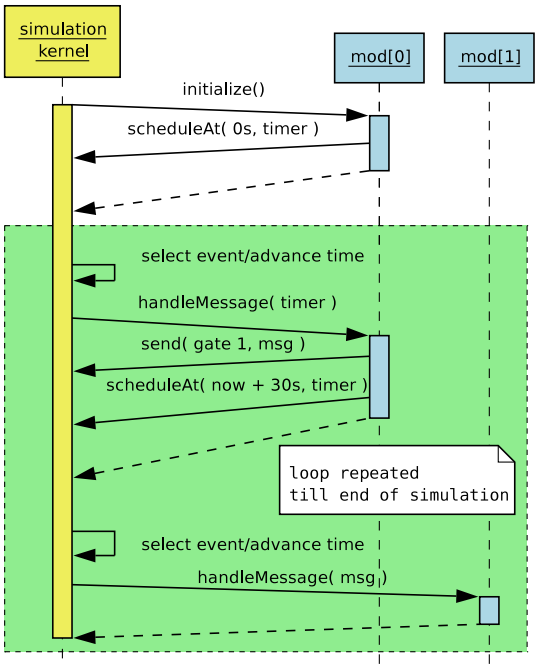

Fig. 1 UML sequence diagram of the OMNeT++ simulation kernel. The left bar represents the simulation kernel, the others two modules. The continuous-line arrows from one bar to another represent normal C++ methods call. The parameters of the calls are those between brackets

Let us see, for example, how it is possible to simulate a module, called mod[0], that sends a message every 30 s to another, mod[1]. The situation is depicted in Fig. 1. During the initialisation phase, mod[0] schedules a message for itself at time 0 s, that is, at the beginning of the simulation. This is how module's internal timers can be simulated. When the simulation starts, the scheduler of OMNeT++ takes this message out of the list, and delivers it to mod[0]. The module is waken up by this message and prepares the message to send to mod[1]. mod[0] sends the message, that is, calls a function of OMNeT++ to store the message in the scheduler. Given the length of the message, the speed of the connection between the modules and the current status of the channel (busy or free), OMNeT++ calculates the delivery time of the message to mod[1]. After having sent the message, mod[0] sets another timer for 30 s later. The control returns to the simulation kernel. It takes the following event in the list, the message to mod[1], and

[8] http://www.cm-labs.com/products/vortex/

[9] http://www.havok.com/

[10] http://sserver.sourceforge.net/

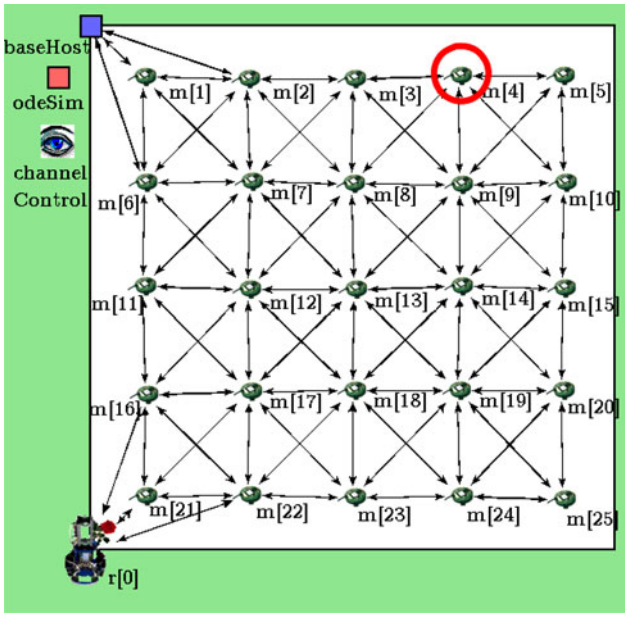

Fig. 2 Snapshot of BARAKA: network view. The icons on a grid represent the position of motes. They are named m[X], where X is just a progressive number. One robot (r[0]) is at the bottom left corner. The arrows shows the connections, i.e., which are the nodes that can be reached by each agents. Connections are handled by the channelcontrol module of the MF. The circled mote is the target that the robot has to reach in the experiment of Sect. 6

delivers it to its destination. The loop continues till the end of the simulation.

OMNeT++ is a very general simulation tool. There are plenty of extensions that have been created to simulate, among other things, communication networks. We used the "Mobility Framework (MF)"[11] for our simulation of SANETs. MF provides a structure to simulate communicating mobile nodes and additional modules that ease the implementation of a reliable simulation.

A special module, the channelcontrol module, connects hosts that could theoretically communicate with each other, as shown in Fig. 2. After every movement of any host, the channelcontrol updates its connections according to the new positions.

Each host is simulated as a compound module. It contains five other modules (Fig. 3). Three of them simulate the standard networking layers: application layer, networking layer and physical transmission device (the NIC). Any message received by the host goes upward from the NIC, through the network layer to the application layer. The inverse path is followed by a message generated by the application layer and directed to other hosts.

There is an additional module (mobility) to communicate the new position of the host to channelcontrol, and a blackboard for cross-layer communication.

[11] http://mobility-fw.sourceforge.net/

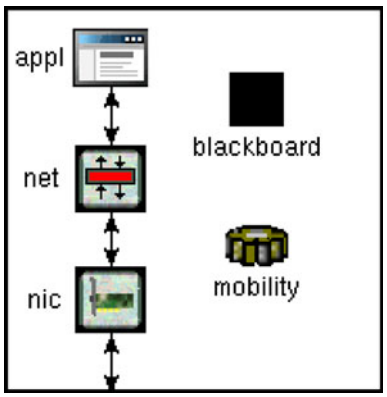

Fig. 3 Implementation of a mobile host in the Mobility Framework. Each host is made of five modules: application layer, network layer, NIC, blackboard and mobility

A number of reason drove our choice to OMNeT++. First of all, it is open source: we could analyse its architecture and implementation in order to improve our hybrid simulator. The highly modular code of OMNeT++ eased the integration process. Additionally, OMNeT++ comes with a highly functional Graphic User Interface (GUI), which allows to inspect and control nearly all the details of the simulation. The GUI is however an optional module. It was very useful during development, but it slowed terribly down the performances when we needed to perform our experiments. For such cases, OMNeT++ can also use a command line interface. The output goes to the console or to a file. Last but not least, OMNeT++ has a scripting language to describe the network, and a very detailed and helpful user manual.

4 Open Dynamics Engine

The Open Dynamics Engine[12] is a library used to simulate rigid bodies. It provides primitives to define a *body* by its mass, momentum of inertia, initial position and velocity. A body corresponds to the dimensionless mass point used in elementary physics. Different bodies can be attached to each other through a number of *joints*: free, extensible, hinges, ball&sockets, and so on. Each body can also have more *geometries* attached to it, which are used to give a shape to the body. The library also offers primitives to apply forces and torques to the body (as a motor does on the wheels of a car).

The library offers two very important functions. The first one takes the state of the environment at time t and computes the new state at time $t + \Delta$, where Δ is a user defined parameter. This function integrates the equation of motion and returns the solution at time $t + \Delta$, thus it is called

[12] http://ode.org

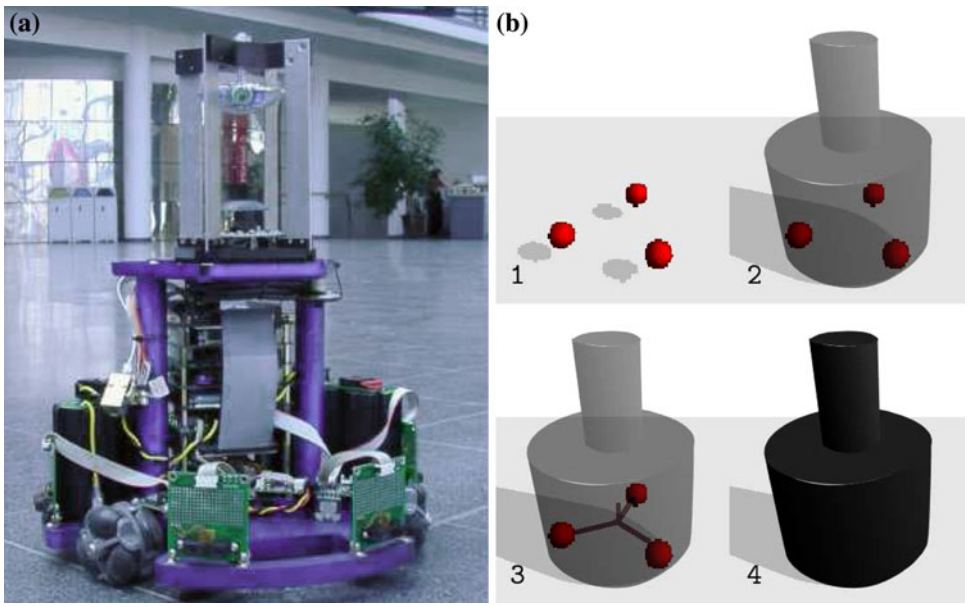

Fig. 4 Left: picture of a Robertino robot. Right: the sequence shows how the Robertino robots can be built and simulated in Open Dynamics Engine: (1) three spheres simulate the omnidirectional wheels; (2) the main body of the robot is approximated by two cylinders; (3) three hinges connect the wheels to the robot's main body along radial axes; (4) the simulated robot is ready to move

integration step. The second function checks whether any two objects are colliding. If it is the case, the libraries takes some measures in order to avoid the penetration of the bodies at the next integration step. Collision detection is also used to compute friction at the contact points. In this way, if we apply a torque to the hinges that connect four spheres to a parallelepiped, the spheres touch the ground, and we set some friction between spheres and ground, we can simulate the wheels of a car on a road.

It is up to the program using ODE to set-up the objects correctly and to iteratively call the two functions to advance the simulation. Between two calls to the integration step, the program can perform whatever task it needs to do. It can change, for instance, the torque applied to some robots' wheel in order to avoid an obstacle.

An example can help to understand better how ODE works. Our laboratory has a number of Robertino robots[13] (Fig. 4(a)), that we want to simulate later together with a WSN. Robertino is a three-wheeled omnidirectional robot, with six infrared sensors around the body and an omnidirectional camera. The robot uses Swedish wheels, which have a strong grip in the direction of the rotation of the motor, but a very low friction along the axis of the motor. Three such wheels grant the robot the capability to reach every configuration (position and rotation) on a plane.

Figure 4(b) illustrates how we simulate Robertino with ODE. The shape of the robot is approximated by two cylinders (two geometries). The cylinders are connected to the body placed in the centre of mass of the robot. Weight of the body and dimensions of the geometries respect those of the real Robertino. Wheels are simulated with three spheres. The spheres are connected to the main body through three hinge joints. The hinges are free to turn around the radial axes. The motors of the robot are simulated by applying torques to the spheres. The rotational axis of each sphere corresponds to the motor axis of the real robot.

ODE allows to specify two friction coefficients for each geometry. We set very low friction between wheels and ground in the direction of the radial axes and high friction in the perpendicular direction. This is the direction of rotation of the turns. This can effectively simulate a Swedish wheel.

The main reason for our choice of ODE is that it is open source. We did not need to change the original code much, except for a small patch to simulate cylinders. Having access to the code however allowed us to tune better our simulator. We had past experience with commercial tools [23], and the fact of being closed source gave us some problems. The worst one was probably the dependency to the license policy of the companies. If, for instance, the licenses became prohibitively expensive, we could not use our tools any more. ODE is in this regard highly safe. Additionally, ODE uses algorithms that were optimised for speed. Although we did not perform rigorous measurements, we had the feeling that it run faster than the tools we previously used.

5 BARAKA

BARAKA is the name that we gave to our simulator. It is the result of the integration of ODE into OMNeT++.

The advantage of BARAKA w.r.t. OMNeT++ and ODE taken individually is that BARAKA is better when

[13] http://www.openrobertino.org/

one has to focus on both the networking and the physics of the system at the same time. In [6], for instance, we study a system where the robots' controllers (implemented in the application layers) heavily interact with the network layers of the nodes. If we had run two different sets of experiments, one with a network simulator and one with a robot simulator, we would not have been able to understand and exploit the effects of physics on communication and the other way round.

We first provide some details about BARAKA's architecture. The description goes into some technicalities of the implementation, but it helps to explain how we could merge discrete event simulation with the simulation of rigid bodies. To better understand the interplay between OMNeT++ and ODE, we also explain how we simulate the infrared sensors of our robots. Finally, we conclude the section with some considerations about the simulator's performance.

5.1 Architecture

BARAKA's architecture was designed to tackle the problem of integrating two different type of simulators: a discrete event network simulator and a rigid-body simulator. The points to solve are two: first, to merge the collision detection/integration step loop in the OMNeT++ flow; second, to create modules that simulate the robots and the motes both in their physical and networking aspects. These modules are used by the agents' programs to control the behaviours of the agents in the simulated world.

The ODE loop takes place in an OMNeT++ module called odesim (it is represented as a square in the top left corner of Fig. 2, above channelcontrol). It has no connection to any other module in the simulation. odesim neither receives messages from nor sends messages to the others. During its initialisation, it sets up a timer in the OMNeT++ scheduler. When waken up, odesim performs the collision detection and the integration step of ODE. It then sets up the same timer for Δ seconds later. odesim behaves like mod[0] in Fig. 1, only without the sending of a message (Fig. 5).

We defined a set of interface classes. They allow to modularise the whole simulation and to separate the objects in charge of the simulation from the objects in charge of the agent control. `RealWorldObject` (Fig. 6) formalises the API common to each simulated agents. It includes, for instance, the methods to send messages. `Robot` adds the methods typical for robots (setting the speed of the wheels, getting the sensor readings, etc.). The interface `Controller` specifies the methods that the agents' controllers have to implement. They include most notably a method that is called at each control cycle and a method to handle incoming messages. These interfaces allows a more

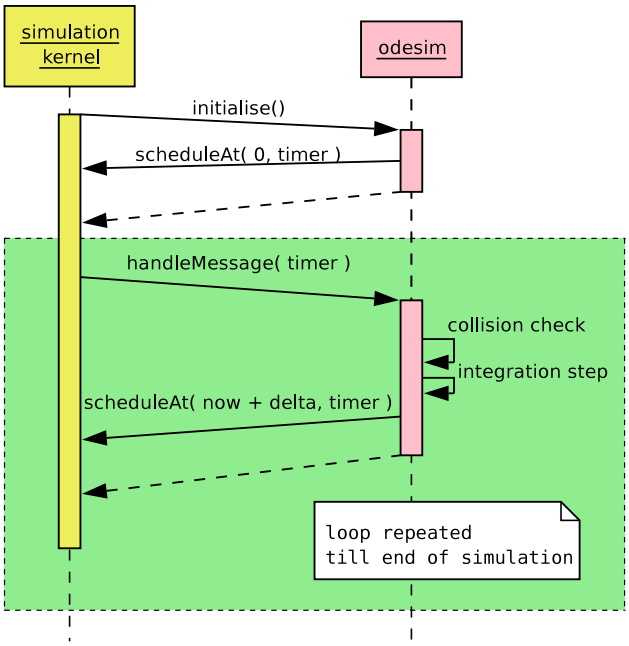

Fig. 5 UML sequence diagram for the integration between OMNeT++ and ODE. The left bar refers to the simulation kernel of OMNeT++. The right bar refers to the OMNeT++ module which implements the real world simulation with ODE

painless switch from simulation to real hardware. It will be not necessary to rewrite the controllers of the agents, but only the classes that implement the interfaces.

Two classes take care of simulating the agents. They are the ones handled by OMNeT++ simulation kernel: `SimulatedMote` and `SimulatedRobot`. They implement respectively the interfaces described by `RealWorldObject` and `Robot`. We used the multiple inheritance mechanism to allow these classes to simulate both the networking behaviour and the physical-world behaviour.

On the one side, they inherit from `ODEObject`. This class is used as gateway to ODE library and odesim. It allows to create the bodies, geometries and joints related to one object, to get and set the speeds, and so on. When `SimulatedMote` and `SimulatedRobot` are initialised at the beginning of the simulation, they create the objects in the physical world. Motes are simply simulated as light cubes (5 cm side, 50 g mass). Robots are created as shown in Fig. 4(b), respecting the real masses and sizes.[14] Figure 7 shows the resulting simulated world.

During the simulation, the `SimulatedRobot` applies the torques to the object's wheel according to the commands given by the controller. If, for instance, the controller decides that the robot has to move forward, the controller calls the method in `SimulatedRobot` to set the speed of the wheels. The implementation in `SimulatedRobot`

[14] http://www.openrobertino.org/hw/dimensions/overview.html

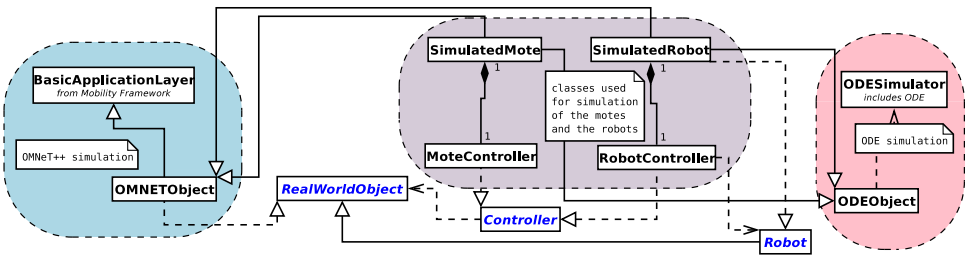

Fig. 6 UML class diagram of the most relevant classes of BARAKA. See the text for the description

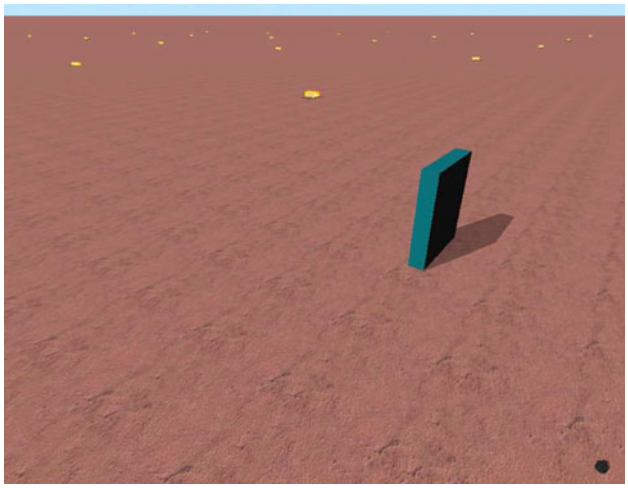

Fig. 7 Snapshot of BARAKA: three-dimensional world view. This picture shows a view of the three dimensional world associated with the network view of Fig. 2. The real dimension of the motes, usually few centimetres, were increased to make them visible. For comparison, the robot in the bottom right corner in the simulated world view is 42 cm tall. The picture shows also one obstacle, a wall, placed between the robot and the first mote it has to reach

calculates which is the required rotational speeds of the wheels. It then calls via `ODEObject` functions of the ODE library to set the desired rotational speed by applying a torque on the hinges. `SimulatedRobot`'s work ends here. During the next integration step, odesim will let the wheels rotate. Given the friction with the ground, the rotation of the wheels will result also in a forward translation of the wheels and their connected bodies, i.e., the robot.

On request from the controller, `SimulatedRobot` uses information obtained by odesim in order to simulate the robot's sensors. `SimulatedRobot` can get, for instance, a list of nearby objects and use it to calculate the value of the infrared sensors to return to the controller (Sect. 5.2).

The other inheritance branch comes from `OMNETObject`. This class deals with everything that has to do with the networking of the node. It is derived from the application layer class specified by the MF. If the controller wants to send a message, `SimulatedMote` and `SimulatedRobot` take the message from the controller, wrap it into a OMNeT++ message and put it in the scheduler of the simulation kernel. When an agent receives a message from others, the message passes through the NIC, the network layer till the application layer, that is, an instance of either `SimulatedMote` or `SimulatedRobot`. The message is then forwarded to the controller to be processed.

The controllers of the motes and the robots are implemented respectively by the classes `MoteController` and `RobotController`. They both implement the interface `Controller`. Each instance of `SimulatedMote`(`Simulated` *Robot*) contains one instance of `MoteController`(`Robot` *Controller*) and simulates one mote (robot).

5.2 OMNeT++ /ODE interplay

Figure 8 summarises how the classes described above work together. The figure depicts the typical working flow of the simulation of a robot. During initialisation, `SimulatedRobot` creates all the elements for the simulation of the robot, and then registers the robots into odesim. During simulation, `SimulatedRobot` can receive two type of events from the OMNeT++ simulation kernel: new incoming messages or the beginning of a new control cycle.

Let us give a look to what happens during the control cycle, since it is the most complex and richest part. Objects of the class `SimulatedRobots` set a timer event for every control cycle. When they are waken up by this event, they call the robot's control program, implemented in an object of type `Controller`. A robot's control program usually starts by querying the sensors to know the current state of the environment. In our case they are the infrared sensors. The method to obtain the infrared readings is implemented in `SimulatedRobot`. `SimulatedRobot` asks odesim for a list of nearby objects. This list, together with a model of the infrared sensors, is used by `SimulatedRobot` to calculate the values to return to the control algorithm. The latter then evaluates the speed and direction of the robot, which are set through a method of `SimulatedRobot`. As we already explained before, `SimulatedRobot` asks then the ODE library to set the right torques to the robot's wheels.

It should be noted that it is quite expensive to calculate the list of nearby objects every time the controller wants to read the infrared sensors. If every controller did it, it would imply to check at every control time step the distances between nearly N^2 pairs of objects, where N is the number of objects in the environment. This is obviously a likely

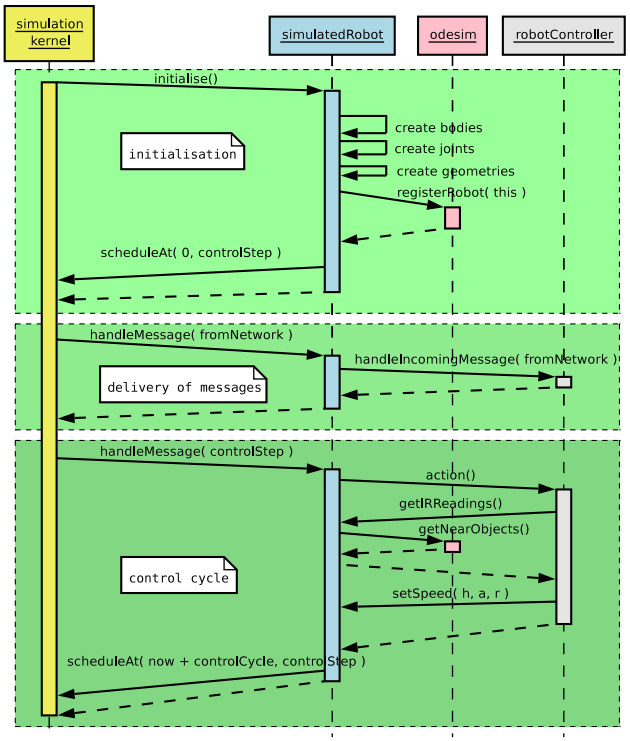

Fig. 8 UML sequence diagram that summarises how a robot is simulated in BARAKA. During the initialisation, simulatedRobot (an instance of `SimulatedRobot`), creates the body in the world handled by odesim. The simulation kernel delivers messages to simulatedRobot, which forwards it to robotController, the controller of the robot. simulatedRobot sets a periodic timer to simulate the robot's control cycle. During the control cycle, the controller might call methods of `SimulatedRobot` to obtain, e.g, the value of the IR sensors. `SimulatedRobot` calculates the value using information coming from odesim

bottleneck of the simulation. However, we can skip this step: we can exploit the collision detection step of Open Dynamics Engine, which basically does the same, but in an optimised way.

ODE first controls if the bounding boxes of any two objects intersect. The bounding box is the box with minimum volume that contains the object and whose faces are parallel to the XY, XZ and YZ planes. If two bounding boxes intersect, ODE will call another function provided by BARAKA. This function has to calculate more accurately the contact points of the shapes of the two objects. We exploited this mechanism by artificially increasing the robots' bounding boxes in order to be as big as the area covered by infrared sensors. When ODE calls BARAKA's functions to find the contact points, and if one of the two objects is a robot, we add the object under examination to the a list of near objects. Each `SimulatedRobot` instance has its own list. The list is updated at every collision detection step. Afterwards, the normal bounding-box check takes place and in case BARAKA calculates the real contact points.

The network connections to other agents are kept up to date by the mobility module related to each simulated agent. The mobility node regularly queries odesim for the position of the agents, that is, of the objects created by `SimulatedMote` and `SimulatedRobot`.

5.3 Performances

It is hard to give some sound measure of 'performances', mostly because we miss similar tools for comparison. We think however that we might give the reader a sort of a 'feeling': in [6] we performed 240 runs, each run one simulated hour long. The whole experiment took less than 1 day on a 20-CPU cluster. Even with our most complex configuration (25 motes plus 12 robots) the simulation was faster than real time. All experiments were done obviously using the command line interface to OMNeT++.

A serious bottleneck is in the way network connections are handled. At every movement of a robot, the MF checks the distance of the robot with *all* the other objects in order to find out new connections and delete the old ones. The developer of MF are aware of this problem and are already considering faster solutions for future versions.

6 Case study

We now describe an experiment we ran in order to test our simulator. The following set-up might seem too complex and some design decision that we took might seem unmotivated. This is due to the fact that what we discuss now is in fact only a part of a bigger scenario that we described and analysed in [6]. There is unfortunately not enough space to justify our choices here. We think however that it is not so important now to give a sound justification of our set-up. Our purpose is to test BARAKA and to give some examples of what we can do with it. The following set-up, albeit complex, can effectively test both the networking and the physical behaviour of a SANET at the same time.

6.1 Experiment description

In this experiment, both networking and physical simulation of the environment are important. We simulate a SANET with 25 motes placed on a grid in a square environment of side 500 m, as shown in Figs. 2 and 7. One mote (highlighted by a circle in Fig. 2) broadcasts a message requesting for a robot's intervention. There is one robot in the bottom left corner that listens for incoming requests. When it receives one, it answers and drive to the requesting mote, avoiding to collide against other objects.

The environment size is smaller than the one commonly used for SANETs. On the one hand, we need a large area to have a realistic simulation of the system; on the other hand, larger area increases the simulation time, since the speed of the robot is fixed. The size of our environment is a trade-off between different between the two criteria and is enough to test BARAKA's features.

The robot is too far away to directly communicate with the mote, thus the network requires a routing mechanism. Our routing algorithm, derived from *AntHocNet* [24] is thoroughly described in [6] and in [25]. We need however to highlight some characteristics that we require later. Each node i keeps routing information in a number of tables ${}_c\mathbf{R}^i$ for each class of messages c. Classes are be used to identify messages belonging to different tasks of the SANET. Each entry ${}_c\mathbf{R}^i_{nd}$ is a tuple that contains some statistics about the path from node i to node d using node n as next hop for a message belonging to class c. A single tuple can contain for instance the energy required for transmission, the minimal signal-to-noise ratio or the end-to-end delay. The routing discovery mechanisms allows to find multiple routes to destination. Each packet is randomly forwarded to one of the node i's neighbour n for destination d with probability

$$\mathcal{P}^i_{nd} = \frac{r({}_c\mathbf{R}^i_{nd})^\beta}{\sum_{j\in N^i_d} r({}_c\mathbf{R}^i_{jd})^\beta}, \tag{1}$$

where N^i_d is the set of neighbours for which a path to d is known, $r(\cdot)$ is a function that takes a tuple in ${}_c\mathbf{R}^i$ and returns are real positive value, β is a constant bigger than or equal to 1. For medium access protocol, we use the IEEE 802.11 as implemented by the modules provided by MF.

The robot needs to know the path (in the environment) to its destination. The robot does not require a map of the environment since it can obtain enough information from the routing table of its network layer. The topology of the WSN can be seen as an approximation of the topology of the environment. The robot can exploit this feature instead of trying to build a map of the environment. Assuming that the robot can know the direction of a nearby mote,[15] it can travel in the SANET in the same way in which network packets do.

6.2 Agents' controllers

After the mote has broadcast its help request, its controller works as depicted in Fig. 9:

request

The mote waits for any robots to reply. In our case, only one robot can reply. If the mote does not receive a positive answer within 30 s, it broadcasts the request again. It repeats this for a maximum of 3 times and then gives up.

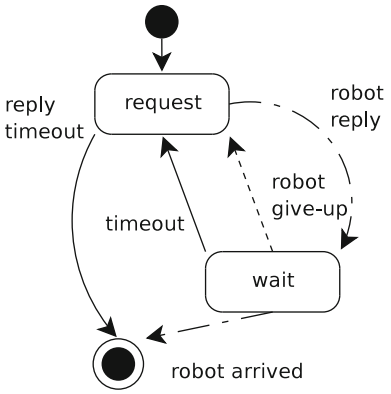

Fig. 9 Motes' behaviour for help-request task. The dash-dotted arrow represents a transition that occurs thanks to an incoming packet, in this case a robot that answers the mote's request or that signals its arrival. The dotted arrow stands for an incoming packet from the robot which gave up the task. Continuous-line arrows are internal events which the mote evaluates at each control step. See the text for the description of the states

wait

The mote waits for the robot that was assigned the task. During this period, the robot is travelling through the network to reach its destination. It regularly sends messages to the mote. Messages are both used as 'keep alive' and to update the robot's route. They contain also the expected maximum time the robot requires to travel one hop. If the mote does not receive a message from the robot again within this time, the mote 'drops' the robot, broadcasts the request again and returns to **request**. Upon arrival, the robot sends a message to signal it is on the place, and the mote considers the request fulfilled.

The robots starts acting after the arrival of the mote's request. The controller of the robot works as shown in Fig. 10:

select destination

In [6], the robot probabilistically chooses one host among the set *HD* of motes that are waiting for help. In our case there is however only one mote requesting, and thus it is chosen with probability 1.

request assignment

The robot informs the destination that it is willing to take on the request. The mote decides whether the robot can continue or not. The mote may reject because the request was already fulfilled, or because another robot is working on it. If the robot does not receive an answer after 30 s, it sends the request again for a maximum of 3 times, then it gives up.

[15] It is not the purpose of our work to address this problems, but it might be done, e.g., by triangulating the signal emitted by a node, by using directional antennas, or by means of a vision system.

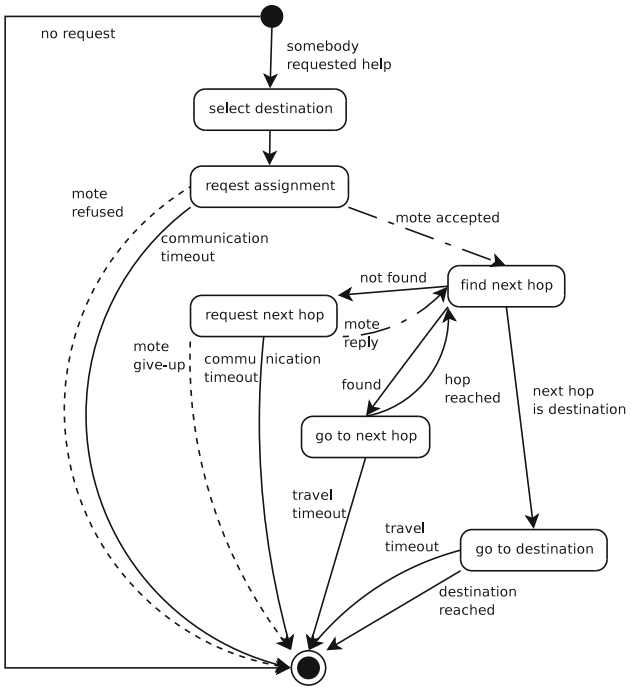

Fig. 10 Robots' behaviour for help-request task. The meaning of dash-dotted, dotted and continuous-line arrows is as in Fig. 9. See the text for the description of the states

find next hop

The controller looks into the routing table of the network layer to select the next hop of the route to its destination. The next hop is chosen as in (1). We refer only to the information in the routing table regarding this task only (request for help), therefore we drop the class specification from the notation of (1). The probability $\mathcal{P}_{nd}$ that robot r selects neighbour n as next hop to go to d is given by:

$$\mathcal{P}_{nd} = \frac{r(\mathbf{R}^r_{nd})^2}{\sum_{i \in N^r_d} r(\mathbf{R}^r_{id})^2},$$

$$r(\mathbf{R}^r_{id}) = \begin{cases} H & \text{if } i = d, \\ \frac{1}{h} & \text{otherwise.} \end{cases}$$

where N^r_d is the set of the robot's neighbours that know a way to d, h is the distance measured in number of hops, and H is a high value constant. This functions selects the next hop in the same way as the network layer does to route data packets. A robot going to its destination can be seen as a special kind of packet travelling the network.

request next hop

If there is no entry in the routing table about destination d, the robot sends a message to the destination and waits for a reply. This message is used to start the route discovery process at the network layer. As in **request assignment**, the robot waits 30 s before sending another request, for a maximum of 3 times, then it gives up.

go to next hop

The robot proceeds towards the next hop. It periodically checks the IR sensors to see whether it is going to collide with an obstacle, and avoids it if necessary. When the robot is at 10 m from the mote, it invalidates the hop's entry in the routing table, and sends a message to the destination, in order to start a new route discovery process. When it reaches the distance 4 m, it considers the hop reached and searches for a new one. If the previous message did not get lost, the routing table should already contain the information to find the new hop immediately. If the robot has been trying to reach the hop for more than 300 s, it gives up.

go to destination

In this state, the robot behaves mostly as in **go to next hop**, only the robot does not need to send a message to the destination when it is at 10 m from it. When the robot is at less that 10 m from destination, it signals the mote that it has arrived.

It should be clear now to the reader that to simulate this set-up we need a good simulation of both the network and the physical world. The former is required to accurately simulate the communication between robots and motes, the latter to allow the robot to move. Although this set-up might seem to the reader somehow crafty (we recall that this is due to the fact that is only a part of our work in [6]), it is a good test bed for our simulator.

The messages exchanged between the robot and the mote consist of three Boolean fields: `ra`, `ma` and `a`. The robot sets `ra` (it stands for "`robot acknowledgement`") to `true` when it asks to be assigned the help request, or when it sends messages to find the next hops in the route. The field is set to `false` when the robot gives up. The mote sets `ma` ("`mote acknowledgement`") to `true` to inform the robot that it is in charge of the request. The mote sets it to `false` if the mote gives up on the task. Finally, the robot sets `a` ("`arrived`") to true to inform the mote that it arrived at the destination.

6.3 Measurements

We run 10 experiments with the setup explained above. In this section, we show selected examples of these experiments in order to demonstrate the possibilities of what can be measured with BARAKA.

One might be interested in the trajectory of the mobile nodes. It is easy to modify the mobility modules of each node to log the position. We prepared such a modification in order to appropriately evaluate the mobility of the robot

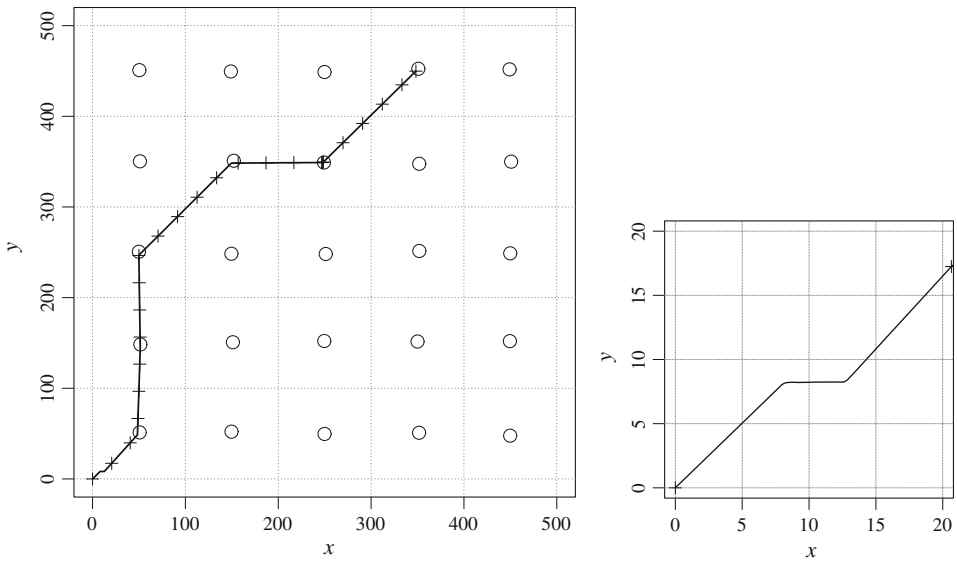

Fig. 11 Left: Trajectory that the robot followed to reach its destination. Ticks mark the position every 30 s. Note the obstacle avoiding manoeuvre at the bottom left corner. Right: close up on the avoiding manoeuvre. The values on the x and y-axes are in meters

systems. The result of one example run is shown in Fig. 11.[16]

While analyzing the mobility models of the robot systems according to the ODE based physical simulation, it is at the same time possible to perform measurements in the network using the features provided by OMNeT++. At this place, all typical measures can be taken as requested by standard network simulators.

For example, if one wanted to see which parts of the network are under load, one could plot the number of packets received by each node, as we do in Fig. 12. This measure provides a rough idea of the overall resource utilization in the entire network. As can be seen, the nodes in the core of the network have to perform more operations compared to border nodes. We can derive a number of questions regarding the quality of the algorithm under test from these numbers, e.g., to analyze the global fairness or the average energy consumption.

Additionally, it is also easy to measure the time it takes to discover a particular route. This measure allows to study the real-time behavior of applications using the analyzed routing scheme. If the route setup is either unpredictable or just too high, real-time applications cannot be used in combination with this routing scheme. For the given example, the distribution of the time to discover a route is shown in Fig. 13, and it is summarised by the following numbers: minimum 0.96 ms, first quartile 2.11 ms, median 10.27 ms, third quartile 60.72 ms, maximum 10.02 s.

Finally, the end-to-end delay is a typical measure used to characterise the behaviour of routing protocols. Again, some kinds of applications essentially rely on a low variation of this measure (similarly to the route setup delay). In

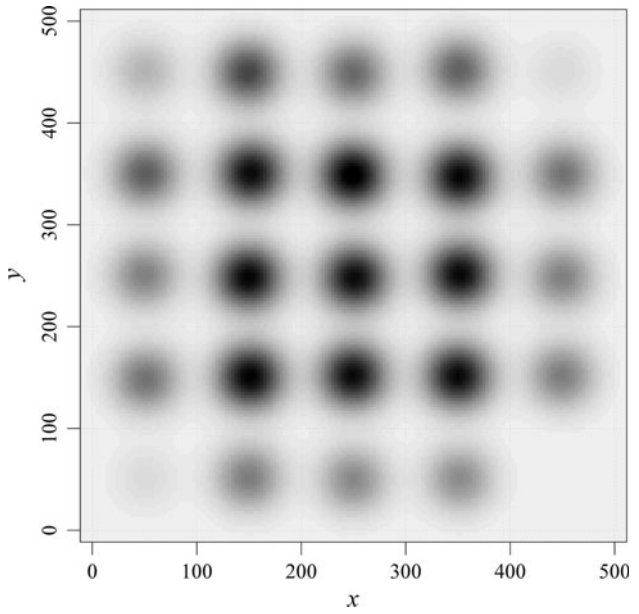

Fig. 12 Number of packets received by each node during the experiment. Each node is represented by a fuzzy circle. The darker the circle, the more packets the node received. Data refers to 10 replications. The node with the highest number of received packets is the one in the second row from the top and the third column from the left (4,272 packets). The one with the lowest number is the lowest right node (1,848 packets). The values on the x and y-axes are in meters

measurement results for our scenario are shown in Fig. 14 and summarised by: minimum 0.7 ms, first quartile 0.7 ms, median 2.52 ms, third quartile 8.7 ms, maximum 5.66 s.

There are several more things that can be measured with BARAKA. In fact, BARAKA can be used to estimate or measure whatever observable might interest the researcher, as long as the measurements can be expressed in the form of C/C++ code to insert in the source. For instance in [6],

[16] Some movies from this experiment, can be seen at http://www7.informatik.uni-erlangen.de/~labella/comsware07.html.

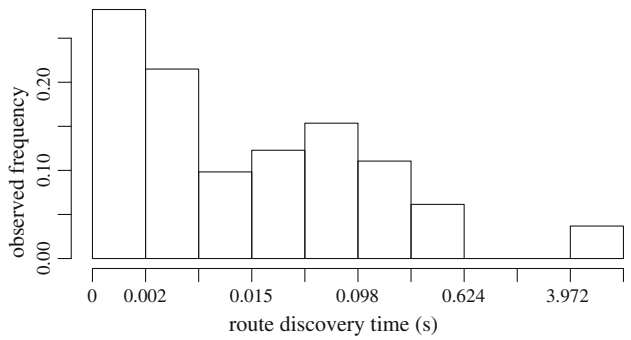

Fig. 13 Distribution of the time to discover a route in the simulated SANET. The *x*-axis uses a log scale. Data refers to 10 replications

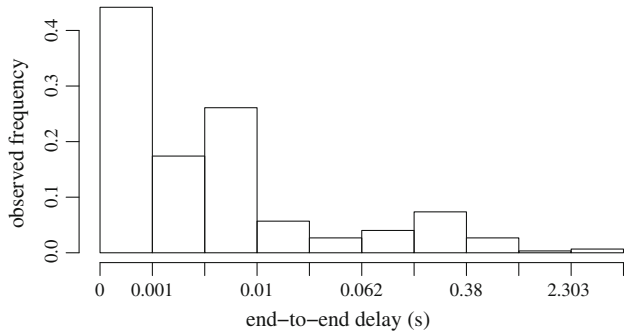

Fig. 14 Distribution of the end-to-end delay in the simulated SANET. The *x*-axis uses a log scale. Data refers to 10 replications

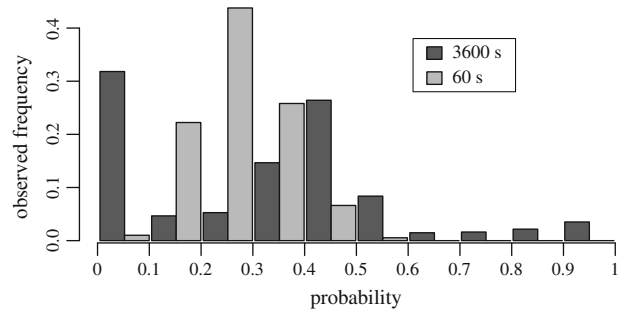

Fig. 15 Analysis of agent behaviours with BARAKA. This plot, described more properly in [6], shows the development in time of agents' probabilities of choosing a task. It used here to show the possibility of performing complex analyses with BARAKA

we studied an architecture for division of labour. The robots and the motes choose randomly one task to perform and adapt the task during time. Thanks to BARAKA, we were able to study the development of the robots' and motes' probability, depicted in Fig. 15.

7 Conclusions

We described BARAKA, the simulation tool that we developed for a comprehensive analysis of SANETs. This simulator is more advanced than the ones currently available in the sense that it can accurately simulate the networking and the physical world of the system. This goes at the cost of some more computation time. The overhead is however negligible: in our experiments the speedup with reality was more than satisfying. The advantage of BARAKA is that it allows a researcher to do experiments on new forms of robot/mote and application/network layer interactions. It also reduces the work necessary to port the algorithms on the real hardware.

However, the simulator still lacks a validation in more complex scenarios, and our future work will for sure go in this direction. Our preceding experience with simulation both of networks and of robots make us confident in a successful validation: when designing BARAKA, we used our experience to improve its reliability.

The current development status of BARAKA is more similar to a prototype than to a proper simulator. Robots and motes are well simulated, and the simulation can be easily controlled through the user interface of OMNeT++. If the user wanted, for instance, to change the shape of the robots, this would be possible only by reimplementing the `SimulatedRobot` class. For the future, we plan to expand BARAKA by means of a script file that can describe the robots, and possibly choose between different control algorithms. This would help BARAKA to become even more flexible.

Acknowledgements Thomas Halva Labella thanks the DAAD (Deutscher Akademischer Austausch Dienst), grant number 331 4 03 003, for the fellowship that funded this work.

References

1. Akyildiz, I., & Kasimoglu, I. (2004). Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks, 2*, 351–367.
2. Melodia, T., Pompili, D., Gungor, V., & Akyildiz, I. (2005). A distributed coordination framework for wireless sensor and actor networks. In: *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing (ACM Mobihoc 2005)* (pp. 99–110). New York, NY: ACM Press.
3. Batalin, M., & Sukhatme, G. (2004). Using a sensor network for distributed multi-robot task allocation. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA2004)* (Vol. 1, pp.158–164). New York, NY: IEEE Press.
4. Melodia, T., Pompili, D., & Akyildiz, I. (2006). A communication architecture for mobile wireless sensor and actor networks. In: *Proceedings of IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON 2006)*. New York, NY: IEEE Press.
5. Dressler, F. (2006). Network-centric actuation control in sensor/actuator networks based on bio-inspired technologies. In: *3rd IEEE International Conference on Mobile Ad Hoc and Sensor Systems (IEEE MASS 2006): 2nd International Workshop on Localized Communication and Topology Protocols for Ad hoc Networks (LOCAN 2006)*, Vancouver, Canada.
6. Labella, T., & Dressler, F. (2006). A bio-inspired architecture for division of labour in SANETs. In: *Proceedings of the First IEEE/*

ACM International Conference on Bio Inspired Models of Network, Information and Computing Systems (BIONETICS 2006). Italy: Cavalese.
7. Manduchi, R., Castano, A., Talukder, A., & Matthies, L. (2005). Obstacle detection and terrain classification for autonomous off-road navigation. *Autonomous Robots, 18*, 81–102.
8. Jia, S., Sheng, J., Chugo, D., & Takase, K. (2007). Obstacle recognition for a mobile robot in indoor environments using RFID and stereo vision. In: *Proceedings of International Conference on the Mechatronics and Automation, (ICMA 2007)* (pp. 2789–2794). New York, NY: IEEE Press.
9. Law, A., & David Kelton, W. (2000). *Simulation modeling and analysis* (3rd ed.). Boston: McGraw-Hill.
10. Johnson, D., Hu, Y. C., & Maltz, D. (2007). *The dynamic source routing protocol (DSR) for mobile ad hoc networks for IPv4*. IETF RFC 4728.
11. Perkins, C., Belding-Royer, E., & Das, S. (2003). *Ad hoc on demand distance vector (AODV) routing*. IETF RFC 3561.
12. Fujimoto, R., Perumalla, K., Park, A., Wu, H., Ammar, M., & Riley, G. (2003). Large-scale network simulation: How big? how fast? In: *Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS 2003)* (pp. 116–123). New York, NY: IEEE Press.
13. Breslau, L., Estrin, D., Fall, K., Floyd, S., Heidemann, J., Helmy, A., et al. (2000). Advances in network simulation. *IEEE Computer, 33*, 59–67.
14. Heidemann, J., Bulusu, N., Elson, J., Intanagonwiwat, C., Lan, K. C., Xu, Y., et al. (2001). Effects of detail in wireless network simulation. In: *SCS Multiconference on Distributed Simulation*, pp. 3–11
15. Pawlikowski, K., Jeong, H. D., & Lee, J. S. (2002). On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine, 40*(1), 132–139.
16. Sundresh, S., Kim, W., & Agha, G. (2004). SENS: A sensor, environment and network simulator. In: *Proceedings of the 37th Annual Simulation Symposium* (pp. 221–228). New York, NY: IEEE Press.
17. Camp, T., Boleng, J., & Davies, V. (2002). A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing: Special Issue on Mobile Ad Hoc Networking: Research, Trends and Applications, 2*, 483–502.
18. Bai, F., & Helmy, A. (2004). A survey of mobility modeling and analysis in wireless ad hoc networks. In: *Wireless ad hoc and sensor networks*. Dordrecht, The Netherlands: Kluwer Academic Publishers.
19. Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., & Matsubara, H. (1997). Robocup: A challenge AI problem. *AI Magazine, 18*, 73–85.
20. Gerkey, B., Vaughan, R., & Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In: *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)* (pp. 317–323). Portugal: Coimbra.
21. Parker, L. (1997). L-ALLIANCE: Task-oriented multi-robot learning in behavior-based systems. *Journal of Advanced Robotics, 11*(4), 305–322.
22. Varga, A. (2001). The OMNeT++ discrete event simulation system. In: *Proceedings of the 15th European Simulation Multiconference (ESM'2001)*. Nottingham, UK: European Council for Modelling and Simulation.
23. Dorigo, M., Trianni, V., Şahin, E., Groß, R., Labella, T., Baldassarre, G., et al. (2004). Evolving self-organizing behaviors for a *Swarm-Bot*. *Autonomous Robots, 17*, 223–245.
24. Di Caro, G., Ducatelle, F., & Gambardella, L. (2005). AntHocNet: An adaptive nature-inspired algorithm for routing in mobile ad hoc networks. *European Transactions on Telecommunications, Special Issue on Self-organization in Mobile Networking, 16*, 443–455.
25. Labella, T. (2007). Division of Labour in Groups of Robots. PhD thesis, Université Libre de Bruxells.

Author Biographies

Thomas Halva Labella graduated in Computer Science Engineering at Politecnico di Milano (Italy) in 2001. He received his Diplôme d'Études Approfondies (DEA) degree in 2003 and his Ph.D. in 2007 from the Université Libre de Bruxelles (Belgium). In 2005 and 2006 he was a visiting student at the Computer Networks and Communication Systems group at the Department of Computer Science, University of Erlangen. His main research topic is about self-organised division of labour in autonomous robots by means of bio-inspired algorithms.

Isabel Dietrich received her M.Sc. in Computer Science from the University of Erlangen, Germany in 2005. She is currently a Ph.D. student in the Computer Networks and Communication Systems group at the Department of Computer Science, University of Erlangen. Her research interests include UML-based discrete-event simulation, and performance analysis of communication networks and wireless sensor networks.

Falko Dressler received his M.Sc. and Ph.D. from the University of Erlangen in 1998 and 2003, respectively. In 2003, he joined the Computer Networks and Internet group at the Wilhelm-Schickard-Institute for Computer Science, University of Tuebingen. Since 2004, he is an assistant professor in the Computer Networks and Communication Systems group at the Department of Computer Science, University of Erlangen, where he coordinates the Autonomic Networking group. Dr. Dressler is an Editor for the Elsevier Ad Hoc Networks journal, the ACM/Springer Wireless Networks (WINET) journal, and the Journal of Autonomic and Trusted Computing (JoATC). He was guest editor of special issues on self-organization, autonomic networking, and bio-inspired computing and communication for IEEE Journal on Selected Areas in Communications (JSAC), Elsevier Ad Hoc Networks, and Springer Transactions on Computational Systems Biology (TCSB). Dr. Dressler published two books including Self-Organization in Sensor and Actor Networks,

published by Wiley in 2007. He co-authored more than 100 reviewed research papers. Dr. Dressler is Senior Member of the IEEE (IEEE Communications Society, IEEE Computer Society), member of ACM and GI (Gesellschaft fnr Informatik). He is actively participating in several working groups of the IETF. His research activities are focused on (but not limited to) Autonomic Networking addressing issues in Wireless Ad Hoc and Sensor Networks, Self-Organization, Bio-inspired Mechanisms, Network Security, Network Monitoring and Measurements, and Robotics.