Best of Websim99

ANEJOS: a Java based simulator for ad hoc networks

Miguel Sánchez*, Pietro Manzoni

DISCA — Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Camino de Vera, 14, 46071 Valencia, Spain

Abstract

Ad hoc networks are multi-hop wireless networks where mobile devices communicate using a shared, low power, low bandwidth channel without any wired infrastructure: each node acts also as a router of its neighbors. Ad hoc networks protocol design is still a new and growing research area; there currently is an IETF Working Group working on the definition of a standard for such networks. Various proposals are being evaluated and the need for a common comparison framework arises. This paper presents a simulator called ANEJOS (ad hoc networks Java simulator). The simulator is written in Java and is based on the SimJava tool and allows to be adapted to various routing protocols. It also allows considering other relevant aspects to ad hoc networks, like mobility patterns and traffic generation patterns. ANEJOS shared our models with other researchers through the internet, we can use Java applets inside HTML pages containing the whole simulator. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Ad hoc networks; Mobility; Simulation environments; Java

1. Introduction

An ad hoc network is a multi-hop wireless network where mobile devices communicate using a shared, low power, low bandwidth channel. These mobile devices can establish and maintain network connections while moving inside the covered area. The network topology may vary rapidly, typically due to node migration and to signal interference and power outages. No wired backbone or any sort of centralized network management system is present. Control is distributed and every node is required to cooperate to support messages routing. Due to either the high rate of changes in the network topology and the characteristics of the communication channel, it is not possible to employ the same routing algorithms used in static networks.

Various dedicated algorithms have been proposed to solve routing issues for this environment (see [1,2]). In [3] we presented a dedicated routing protocol designed to provide communication support to a collection of mobile robots. Our project consisted in arranging the mobile robots over a lake to monitor water conditions. Each robot, powered by a solar panel, is provided with a micro-controller to execute all the processing. Inside the robot there are a few sensors for temperature, pH level, and oxygen concentration and an air-pumping device. Data transmission is carried out by a wireless modem operating at a common carrier frequency of 433 MHz and with a short reaching range (a few hundreds of meters due to their low power, only 1 mW). The task of the robots was to pump air into the water to rise the oxygen levels in required areas. Robots could change their position due to the water drift, the
wind or human intervention. We wanted the system to be able to react autonomously to the possible changes in the topology.

Ad hoc networks protocol design is still a new and growing research area; there currently is an IETF Working Group working on the definition of a standard for such networks. As said before, various proposals are issued and the need for a common comparison framework arises. With this in mind we designed a simulator called ANEJOS (ad hoc networks Java simulator). The simulator is written in Java and is structured to be extensible and adaptable to cope with the various routing protocols. As Section 3 illustrates, ANEJOS is based on a few basic classes which models the communication medium, the mobile devices and the interchanged messages. These classes can be implemented and extended depending on the characteristics of the protocol to be analyzed. Other classes, namely the mobility class and the traffic generation class, provide complementary extensibility aspects to the simulator. The mobility class allows experimenting with the behavior of the routing protocol with different mobility patterns; in this work we propose the details of three mobility patterns. The traffic generation class allows simulating the behavior of the routing protocols with different data generation patterns or even with transport or application layer protocols, providing extensive features to the simulator.

The paper is organized as follows. Section 2 shortly presents the WARP as an example of a routing protocol for an ad hoc network. Section 3 presents the simulator structure. Section 4 illustrates the details of the used mobility models. In Section 5, we demonstrate how the simulator can be extended to analyze other routing protocols, traffic generation patterns or data-link layer protocols. Finally, Section 6 concludes the paper resuming the results obtained and the on-going work.

2. The routing protocol WARP

We designed an ad hoc networks routing protocol called WARP (wireless “Albufera” routing protocol). WARP is a lightweight protocol based on the activity of a dedicated node called the coordinator. The coordinator is a fixed node that sends all the control information required from the mobile devices to operate and that receives all the information collected by the robots. The protocol constituted of two parts: the forward routing protocol and the backward routing protocol. The forward routing protocol is used to send data from the coordinator towards a destination node, while the backward routing protocol allows a robot to send data to the coordinator. All the nodes take part in the routing of packets toward a destination node or toward the coordinator. A route between node \( v_1 \) and node \( v_n \) is defined as a finite set of distinct nodes \( v \in V \) and a finite set of distinct links \( l \in E(t) \) for which \( v_1 l_1 v_2 l_2 \cdots l_{n-1} v_n \) is verified (see Fig. 1). WARP ensures that each node knows at each time the set of its neighbors while no node is required to know the complete lay-down of the network. A small set of status variables is maintained in each node to allow the protocol to work.

We defined a general structure for the packets used in WARP. From the basic structure three sub-types are derived: data for regular data, hello to propagate topology updates, and frreq, frmain and frsec for the routing protocols. We will not give the details of the forward and the backward routing. These two tasks can be defined as on-demand tasks since they are activated when the coordinator wants to send data towards a node or vice-versa. In WARP there are also periodic tasks that are to be completed. The most important of these is the generation of the hello packets. These messages are periodically sent and are used by the receiving nodes to update their neighbor’s tables keeping also track of the quality of these links. We define three quality levels for the links: strong, weak and loose; by default, all links are marked as strong. Before sending an hello packet, each node has to browse an internal status list. If the time elapsed from
the last hello packet received is greater than a prede-

on the protocol can be found in [3].

3. The simulator structure

The starting point on building the simulator in Java was the idea shown by the SimJava package authors [4]:

“You can build a simulator that lets other people view your results on their own workstation”.

SimJava is a process based discrete event simulation package for Java, similar to Jade’s Sim++, with animation facilities that allow the user to see the progress of the simulation [5]. This possibility proves to be very helpful when defining the details of a new piece of software like, in our case, a routing protocol. Also the Java language WORA (write once, run anywhere) feature seemed to be very interesting. It allows to eliminate the difficulties often found when using different simulation packages, which are too dependent on a particular compiler type or version or on a particular machine architecture. Java allows distributing the source code with the trust that potential users will encounter minimal difficulties on running and modifying it.

Our simulator allows modeling a graph-based ad hoc network, like the one used for WARP. The simulator allows spreading a collection of mobile nodes on an area. Each node executes all the tasks related to a certain routing protocol and all the data transmission operations required from applications. At the same time, each node is capable of moving on the test area following a specified mobility pattern (see Section 4). The data transmission system is built on wireless transmitters deploying radio waves (it could also be implemented by means of infrared beams). We are not trying to include in the simulator all the characteristics of the physical transmission, because of the increment on the complexity and because we are focussing on the routing and mobility aspects.

The basic components of the simulator are nodes, links and the transmission medium. These components are modeled with the following Java classes:

- **Node.** Each instance of this class represents a node (i.e., a mobile robot). A simulation starts launching any number of nodes you want. Every node has a spatial coordinate and sends to and receives packets from an Ether object. When a node receives a packet, it performs a call to the abstract method `process()` of class `Packet` which results in a call to the proper packet type treatment routine. Although every instance of `Node` does the same work, we use a `get_id()` function to differentiate between regular nodes and the coordinator (which plays a special role in our protocol while being another node).

- **Ether.** It is a static class representing the characteristics of the data transmission subsystem. It models a wireless transmission with limited range. Ether needs to know the current position of any network node. On receiving a packet from a node, Ether first checks which nodes are in the range of the source node. Then it sends a copy of the packet to these in-range nodes. This class receives messages from `Node` objects containing either the transmitted packets or control events to inform the Ether of a Node location change.

The relationship between these two classes is shown in Fig. 2. A Node instance requests packet transmission to the Ether class and the Ether
class handles the transmission to each of the close-by nodes.

Some other classes are included in the simulator to handle node location information, packet types (and processing) and mobility models:

- **Pos.** It is the class holding Cartesian coordinates and the methods to deal with them (e.g., distance measurements).

- **Packet.** This is an abstract class, which is the base class for different packet types. It implements the Cloneable interface and redeﬁnes the clone() method to let the Ether class retransmit copies of received packets to the other nodes. This class can be extended in order to include each packet type needed by the routing protocol. Simulating WARP we deﬁned the following three subclasses:
  - **DataPkt.** This is the data packet class, intended to send user data. It includes all the needed ﬁelds not only present in the Packet class but also the code that a node must run when it receives a data packet. This code includes all the routing functions to be performed by the node when the received packet is not destined to it.
  - **HelloPkt.** This class is related to a periodic packet used to maintain status information and intended to be sent only between neighbor nodes in order to advertise our presence to establish a route to the coordinator node. There is no forwarding code because these packets are not routed.
  - **RoutePkt.** The coordinator uses these packets node to discover a route to a certain node. There are different types of these packets labeled by the value of the type ﬁeld. RoutePkt packets are forwarded by nodes and can produce changes in the routing tables. To do this, node identity is passed as parameter in the process method calls (in fact this is done in the super class and therefore in the other two packet classes).

- **Mobility.** This abstract class holds the mobility model structure and the methods that calculate the motion of nodes. Each new motion model extends this class.\(^1\)
  - **Scan, Pursue and Nomadic.** These are descendents of mobility class, implementing the different motion models described in the next section. Start instantiating the desired mobility model, and then add the nodes using addNode() method. Then the current location of this node with the getLocation() function or the new one with getNewLocation() function is obtained. Method setTarget() and function getTarget() lets you change the target position of the model (where available).

- **Comm.** This is the main class, which basically starts all the objects of the simulator. It uses the Sim_system.initialise() method to start the simulation engine and then adds all the entities (Node and Ether objects) by calling Sim_system.add() method. Eventually, when all the entities have been added, it calls Sim_system.run() to actually run the simulation.

The whole simulator is built on top of the SimJava library, which provides the basic threading mechanism and the inter-thread messaging subsystem. Those classes performing concurrent work must extend (inherit) the Sim_entity class. The body() method is rewritten with the code you want this entity to perform. Each entity has a unique identiﬁer and the method sim_schedule() lets you send messages to any entity of the system (including yourself) with a programmed time delay. The sim_wait() method blocks the calling entity until there is a message (event) available. With all these pieces we built the data transmission subsystem and also some of the time-triggered procedures.

We would like to point out that SimJava entities can only process one event at a time. If several events occur at the same time, only one of them is returned by the sim_wait() function. The other events are queued for deferred treatment. To overcome this behavior, we use the sim_get_next() function that returns the next available event independent of whether it is a normal event or a deferred event. In this way we do not miss any event.

Another worthwhile feature to remark is that, although SimJava includes several communication resources, such as the port structures, we choose not to use them because they are conceived for point-to-point links. In our case, communication is based on broadcasts; therefore, an implementation based on point to point links we will end-up with a complex and uncomfortable structure.

\(^1\) The motion models presented in this paper can be found at: http://www.disca.upv.es/misan/mobmodel.htm.
4. Mobility models

Testing protocol behavior in different scenarios of mobility is what led us to include various mobility patterns in our model. In fact, we think that the most important aspect to perform a comparative evaluation of several protocols is to test them in similar mobility scenarios. Anyway, the lack of such kind of models also led us to design some of them, trying to model real world moving patterns associated to some common task performed either by humans or by robots. The idea was to put a mobility model in charge of telling us where every node on the network is moving to, at predefined or random time intervals.

We were only aware of two mobility models: the Brownian motion (described mathematically by Einstein in 1926), and the exponential correlated random model. The Brownian motion represents the motion of molecules inside a solution. At temperatures above absolute zero, the individual atoms that constitute any substance are in movement. Indeed, the amount of this movement is what constitutes temperature itself. The resultant of this motion is sometimes called a random walk and has a zero mean, meaning that there is not really a net movement. The average displacement in a given direction is a linear function of the square root of the time period (t) during which its movement is measured. The relationship is expressed as

$$\Delta Y = \sqrt{\frac{kT}{f}t},$$

where $k$ is the Boltzmann constant, $T$ the absolute temperature and $f$ the frictional coefficient of the particle. As can be seen, movement increases with temperature and also with the time duration of the observation period (sample time) meaning that a longer observation period gives a longer displacement.

In the sample window of Fig. 3, we show the path followed by only one node after a certain amount of time. Because of the distribution we use in the code (normal bell-shaped Gauss distribution), the movement is usually of very short length, producing this fuzzy result. Our sample code to implement such movement is as follows:

```java
// Pos class is dealing with location info
public static Pos move_random(Pos n) {
    double alpha=angle.sample();
    double r=radius.sample();
    // x axis offset
    int xi=(int)(Math.sin(alpha)*r);
    // y axis offset
    int yi=(int)(Math.cos(alpha)*r);
    return new Pos(xi + n.x, yi + n.y);
}
```

We choose a random distance to move ($r$ from a normal distribution with mean=0 and variance=10). We select a random angle ($alpha$, from a uniform distribution between 0 and $\pi$). The x-axis and y-axis increments are calculated by projecting the resulting vector in the two axes.

The second model was the exponential correlated random model developed at BBN Systems and Technologies in 1996. The model has groups of nodes. Each group is a circle containing nodes, which can be described as having a certain number of nodes orig-
inally centered at a specific coordinate with a group speed. The nodes can initially be spread out from the center. We describe each group by moving a determined distance with a certain angle. The movement of the nodes inside a group is like Brownian motion model, but the whole group moves somehow linearly in a certain direction. Group movement is performed according to the equation:

\[ \mathbf{p}_{t+1} = \mathbf{p}_t \cdot e^{-1/\tau} + \varphi \cdot X \cdot \sqrt{1 - (e^{-1/\tau})^2} \]

where \( X \) is the random Gauss variable and \( \varphi \) is the random angle (unit vector) of the displacement. These two parameters are similar to the previous \( \alpha \) and \( r \). The latter parameter, \( \tau \), is a “filtering” function factor which modulates the sharpness (or softness) of movement. The kind of tasks that can be modeled by this latter model could be troops movement in the battlefield, where each group represents a platoon.

### 4.1. Our proposed models

Our models are based on the center idea of the “random orbit”. Each moving node has a home location (or reference point) and moves around it. Each movement is calculated by determining a random \( x \)-axis and \( y \)-axis positive increments and then by comparing the actual location of the node with the home position. If the actual position is above the home position (\( y \)-axis) then the new node position will be formed by applying the \( y \)-axis increment to fall node position. The same is true in the \( x \)-axis, if the node actual position is on the left of the home position the node will suffer a right displacement.

Fig. 4, shows the random orbit model in action. You can see that each new node position produces a new restriction on the space to be used on the next motion. On the right of Fig. 4 you can see an example of a random orbit with 500 movements. (If you take a closer look you will probably see that not all of the available points have been visited. This is due to the random nature of the motion and the limited number of samples.)

The second idea in our models is the group motion. While each of the node moves in a random orbit around its home location, the whole reference system can move following some known or unknown pattern. This is shown in Fig. 5.

Next, we will describe several node motion models, which assume some form of target location. This target can also move, but this motion is not defined.

Fig. 5. The idea of group motion.
in the model. Therefore, we are only specifying the motion model at the node level and not at the target level.

We have developed three new models trying to replicate other real world activities that could be useful in our target system. All of our models are based on a key idea. These activities are:

1. **Scanning an area.** In this model we have tried to represent the moving pattern of a row of robots moving in a certain direction. This behavior can be found on a searching activity (e.g., anti-personal mines deactivation robots). But the model is not limited to a forward direction normal to the row axis, but any angle could be possible to, for example, form a “one behind the other” motion also present in some robotics activities (e.g., transportation convoy). We have called this the “column model”. This model is shown in Fig. 6.

   In the previous figure node zero is representing the target direction of the column of nodes and the small dots represent the reference positions of each node (the plus sign represents the center of gravity of the five points).

```java
// moves a node in column model
public static Pos move_column(int node) {
    // get node real position
    Pos p = Pos(loc.get(new Integer(node))); // get node ref. position
    Pos r = (Pos)ref.get(new Integer(node));
    double d = p.distance(r);
    int xi = (int)(Math.random() * (d + 5));
    int yi = (int)(Math.random() * (d + 5));
    if (p.isLeft()) xi = p.x;
    else xi = p.x + xi;
    if (p.isUp()) yi = p.y;
    else yi = p.y + yi;
    return new Pos(xi, yi);
}
```

To implement such behavior we have of line reference points (“ref” hash table) and each node current position is held in the “loc” hash table. Nodes move randomly around its reference point.

Here the range of movement is proportional to the distance to the reference point of the node. Further the node is from its reference, the more movement it can experience (greater range in the random number). However, if we consider a true random movement (as the Brownian motion) it is possible for a node to reach a great distance from the reference point and therefore, the column of nodes would disappear. We introduce a modification on how we apply the random (positive) increments calculated. If we are on the left of the reference position we add the increment to the x-axis. We do the same in the y-axis. Therefore, all the node movement is something like a “random orbit” around its reference point.

2. **Pursue a target.** Here we have a collection of robots (nodes) trying to catch a single target. This kind of behavior is found in multiple robotics activities (e.g., people or equipment tracking). Here we have taken into account the fact that physics does not let a pursuer robot to follow any position change of the target but it is speed limited and so, the tracking is usually done with some error that may also be due to other factors like the environment. In this model we suppose also certain randomness of movement when the target is stopped and tracked. (For example, you have an apple-pie in your hand and some flies are trying to land on it. You move the pie to avoid them to reach it. This image could describe better than any other model behavior. Flies are the nodes.)

   In Fig. 7, node zero is the target point and nodes number one to nine are trying to approach it. The dot locations are merely the starting point of nodes (with the exception of the target). Again the plus sign represents the center of gravity.

   Similar to the random movement of the column model, here the reference point is substituted by the target point. All the nodes try to reach this target doing random movements, but again, the randomness is constrained by the `isUp()` and `isLeft()` functions. The second difference is the limited values of the range multiplier (`d`) that limits the maximum and minimum range of movement. The upper limitation models the maximum speed limitation. While the lower bound limitation of “`d`” is to maintain a certain “random orbit” around the target point when it is reached.
Fig. 6. Column model (left: motion schema, right: sample window).

Fig. 7. The pursue model (left: motion schema, right: sample window).

Fig. 8. Nomadic community model (left: grid formation, right: sample window).
3. Nomadic community. In this model, all nodes move, from time to time, from one location to another. After the community is established in one target area, each node maintains its own private one, inside of which the node moves more or less randomly. This motion pattern may be useful in some military operations and also (with a slower movement) in agriculture robotics and is shown in Fig. 8.

Sometimes we need to do an operation to an area if each robot is capable of covering let us say one tenth, we can deploy 10 robots and devote each one to a different part of the referred area. After the operation completes, the whole system is able to move to another place to repeat its work. For the moment we are not presenting a community moving-pattern and we are experimenting with manual community moving. The calculations for this model are identical to the ones used in the column model, because again each node “orbits” around its reference position. The initial grid creation is done by means of a square spiral whose code is shown below.

The \texttt{turn\_right} vector holds the position increments to turn “d” pixels to the right in any of the four directions (north, west, south, east). The algorithm always tries to turn to the right as the next move, however, if it finds a busy place, then it continues in the same current forward direction. This builds the square spiral that serves us as reference points.

```
public static
Pos move\_pursue(int node) {
    Pos p=(Pos) loc\_get(new Integer(node));
    double d=p.distance(target);
    // min movement
    if(d<15.0)d=15.0;
    // max movement
    if(d>50.0)d=50.0;
    int xi=(int)(Math.random()\*d/3);
    int yi=(int)(Math.random()\*d/3);
    if (p.isLeft(target)) x1+=p.x;
    else x1+=p.y-x1;
    if (p.isUp(target)) y1+=p.y;
    else y1+=p.y-y1;
    return new Pos(x1,y1);
}
```

```
int d=50, dir=0;
Pos turn\_right[]=new Pos[4];
turn\_right[0]=new Pos(0,d);
turn\_right[1]=new Pos(-d,0);
turn\_right[2]=new Pos(0,-d);
turn\_right[3]=new Pos(d,0);

for(int i=1;i<10;i++)
    loc\_put(new Integer(i),
        new Pos(i*10+100,
            i*10+100));

//starting point
Pos p=new Pos(300,200);
ref\_put(new Integer(1),p);
for(int i=2;i<loc\_size()+1;i++) {
    Pos def=new Pos(p.add(
        turn\_right[dir]));
    // empty place
    if(!ref\_contains(def)) {
        // insert point in the grid
        ref\_put(new Integer(i),def);
        // rotate heading
        dir=(dir+1)%4;
        p=def;
    } else { // continue ahead
        p=p.add(
            turn\_right[((dir+3)%4)]);
        ref\_put(new Integer(i),
            new Pos(p));
    }
}
```

These three models enable us to simulate node communication in a known environment, where we can adjust all the parameters to evaluate its influences. We have also conceived a possible hierarchical extension. At the lower level you found nodes. Several nodes can be joined to build higher level entities which also move conforming some mobility model (e.g., if, at the node level, you have a Brownian motion and each group of nodes moves with the column model then you get something like the exponential correlated random mobility model).

5. Simulator extensions

The current model is only dealing with a certain routing algorithm and let us choose the mobility pattern you want. The current structure allows users to implement their own algorithms to measure their behavior. However, several improvements are possible in order to study other phenomena, like:
• **Media access issues.** In the current version, the simulator does not take into account what is the real media access protocol. This means that we are not covering the possible interactions between the routing algorithm and the access method. We are now considering fixed (or random) access time, which each packet suffers before it is successfully transmitted.

• **Physical level problems.** Wired data transmission is not always error free, sometimes the transmitted signal is corrupted by several physical effects and what really arrives at the destination is a garbled packet. Wireless links are usually worse than their wired counterparts, therefore a more accurate simulator should consider the different link impairments causes (and effects), which would lead to model the real signal propagation, attenuation and noise issues.

• **Realistic traffic patterns.** Simulator traffic is usually built by a simple random traffic generator function. This approach allows studying the overall behavior of a protocol, however, real traffic conditions are usually obtained from the execution of the real transport protocols (and services) over the network simulator. We are evaluating the implementation of the possibility to obtain real traffic patterns from traces or by running the applications and transport protocols over the simulation network level.

Using a layered approach can extend this simulator. The Ether class is the one that deals with packet transmission. Although currently, it is not using a media access protocol, the Ether class is the proper place to include things like MAC or physical transmission issues.

We are developing a Traffic class to be attached on top of the nodes to signal each node when it is time to send a packet. In this implementation, Traffic object will also be Sim_Entity object, running concurrently with other simulator’s threads.

We would like also to remark the graphic capability of the simulator. The Abstract Window Toolkit Java class library allows developing GUI applications that run smoothly in almost any platform. Up to now we have not yet developed a proper graphic support, but we are also working on those aspects. Our idea is to allow users to obtain an animated view of the simulation progress. This kind of observation can help users to quickly detect some kind of uncovered problems and also to interact with the model, for example changing one node’s position at any moment to verify that the routing protocol is correctly handling this situation.

6. Conclusions and future work

Ad hoc networks protocol design is still a new and growing research area; there currently is an IETF Working Group working on the definition of a standard for such networks. Various proposals are issued and the need for a common comparison framework arises. This paper presented a simulator called ANE-JOS (ad hoc networks Java simulator). The simulator is written in Java and is structured to be extensible and adaptable to cope with the various routing protocols. Some researchers of ad hoc networks are working with other simulators, mainly written in C or C++. Most of them require you to download a huge compressed file and build the whole code, which means “Makefiles” and recompilation, and sometimes strong computing power. On the other hand, existing simulators are not always dealing well with wireless links or with moving nodes. But what is the more compelling reason for us to switch to Java is the ability to include your entire simulator on a web page and running it in almost any browser.

Java run-time performance is a hot issue. Being an interpreted language you will found some performance penalty over C++ native code (ranging from two to ten times slower). Just-in-time compilers (JIT) try to rise Java language to a better performer level with varying results. But we have specially found the fact that it lets you program with less effort and errors due to some library constructs (like Vector or Hashtable classes) that eases some of the programming tasks.

It is difficult to estimate which is the benefit in developing time with respect to other languages. The SimJava class library also eases greatly programming tasks of a simulator (because it is a porting from other C++ library you can also work with if you are a C++ programmer). As we have said in the previous section, our future work is aimed at developing lower and higher levels support (MAC, a transport protocol) and also the graphical representation of the informa-
tion. For this task we probably use some of the useful features included in the SimJava package.

The other open path is user documentation and code documentation. Because of the rapid changes and the beginning stage of the project, we do not plan to document it up to simulator structure which will be more stable.

An open problem is to develop the mobility pattern of the several structures we use in the paper as reference point’s grid or target points, because we have shown only the evolution of the model and not the motivation and path of these references to follow. For example, what kind of movement is doing the target in the pursue model. We have left open this because we think it is not very important for us while the speed of grid motion is maintained low (in this case one node has, more or less, the same neighborhood).

References


Miguel Sánchez obtained his B.S. in Computer Science in 1988 and his M.S. in Computer Science in 1992, both from the Universidad Politecnica de Valencia, Spain, where he is now an Assistant Professor at the CS department. His current research is in routing and media access control in ad hoc networks on which he is pursuing his Ph.D.

Pietro Manzoni received his M.Sc. degree in Computer Science from the “Università degli Studi”, Milan, Italy, in 1989, and his Ph.D. degree in Computer Science from the “Politecnico di Milano”, Milan, Italy, in 1995. Since 1995, he is an Assistant Professor of Computer Networks at the “Universidad Politecnica de Valencia”, Valencia, Spain. His research interests are in the theory, design and implementation of communication protocols.