The *Madhoc* Metropolitan Adhoc Network Simulator

Luc Hogie[1], Frédéric Guinand[2] and Pascal Bouvry[1]

[1]Université du Luxembourg, Campus Kirchberg
6, rue R. Coudenhove-Kalergi. L-1359 Luxembourg
{luc.hogie, pascal.bouvry}@uni.lu

[2]Université du Havre, Laboratoire d'Informatique
25, rue Philippe Lebon. 76600 Le Havre. France
frederic.guinand@univ-lehavre.fr

March 22, 2006

Contents

1 Introduction **5**

2 Models **7**
2.1 Network model 7
2.1.1 Hardware 7
2.1.2 Software 10
2.2 Ad hoc application model 10
2.2.1 Application 10
2.2.2 Monitor 12
2.3 Simulation model 12
2.3.1 Languages 13
2.3.2 Discrete-event, discrete time, resolution... 13
2.3.3 Simulation area and projections 14
2.3.4 Random number generation 15
2.3.5 Scheduling 15
2.3.6 Termination condition 16
2.3.7 Application deployment 16

3 Scenarios **19**
3.1 Random waypoint 19
3.1.1 City place 20
3.1.2 Market place 20
3.1.3 Concert place 21
3.2 Human mobility 21
3.2.1 On metropolitan mobility 22
3.2.2 Human mobility, a mobility model driven by the *intentions* of humans 22
3.3 Real-world mobility based on human mobility 24
3.3.1 Highway 25
3.3.2 Mall 25
3.3.3 Streets 26
3.3.4 Supermarket 27
3.3.5 Train station 28

4 Using *madhoc* **31**
4.1 Requirements . . . 31
4.2 Packaging . . . 31
4.3 Installation . . . 32
4.3.1 Installation on UNIX operating systems . . . 32
4.3.2 Installation on Windows operating systems . . . 33
4.4 Using *madhoc* . . . 33
4.4.1 Configuring *madhoc* as a standalone application . . . 34
4.4.2 Configuring *madhoc* as a framework . . . 35
4.5 Using *madhoc* as a standalone application . . . 35

5 The graphical user interface **37**
5.1 Monitor views . . . 38
5.1.1 Standard views . . . 38

6 Advanced usage **43**
6.1 Creating a new module . . . 43
6.1.1 Example . . . 43
6.2 Graphical and command-line user interfaces . . . 46
6.2.1 Using the GUI . . . 46
6.2.2 Graphical 2D plotter . . . 46
6.3 Using *madhoc* as a framework . . . 46
6.4 Accelerating the simulation process . . . 48
6.4.1 Acceleration tricks specific to broadcasting simulation . . . 49

7 Open issues **51**
7.0.2 Initialization of the mobility . . . 51
7.0.3 Altering the resolution of the simulation . . . 51

8 Targetted applications **53**
8.0.4 Broadcasting . . . 53
8.0.5 Mobility models . . . 53
8.0.6 Ad hoc computing: towards the ad hoc grid . . . 54

9 OOP implementation in java **55**

10 Grid **57**

Chapter 1

Introduction

Note:
This document stands for a user manual and a technical reference for the madhoc *simulator. Writing such a document is difficult because several approaches are possible in defining its structure and there do not exist any precise guidelines of how to proceed. In this document, we will try to keep things as simple and logical as possible. Obviously, we cannot give all details of* madhoc *in one single document. Hence, in addition to this pages, the reader should browse the source code, which is publicly available. Note that if the source code is not completely documented, its structure should be easily understandable to people used to the OO programming paradigm and to the Java language.*

Mobile ad hoc networks (MANETs) are networks composed of a set of communicating devices (called *node*s all along this article) able to spontaneously interconnect without any pre-existing infrastructure. Stations in range to one another communicate in a point-to-point fashion. In addition to that, these nodes are generally mobile. The main interest in MANETs is growing. Not only their im- importance in military applications is tightening, but also their impact on business is becoming clear. The wide spread of mobile phones, PDAs, Pocket PCs—which now embed Bluetooth and WiFi (IEEE 802.11) network adapters—enables the spontaneous creation of *metropolitan ad hoc networks* [10]. These networks could then constitute the infrastructure of numerous applications such as emergency and health-care systems [19], groupware [8], gaming [25][13][24], advertisements, customer-to-customer applications (like the UbiBay project [12]), etc.

Up to now, MANETs have been mostly investigated at the lowest layers of the network stack. Indeed, many studies tackle issues like routing [26], broadcasting [30], security [**?**], etc. The vast majority of them resort to MANETs simulation. Unfortunately, the simulation platforms available do not model the networks in a realistic manner and are limited. For example, by simulations, broadcasting [30] has shown to operate well on non-partitioned networks composed of few identical stations moving according the random waypoint mo-

bility model. But how would broadcasting protocols operate on large ($> 10{,}000$ stations) heterogeneous networks? More generally, is broadcasting merely practicable over this structure? Such questions are still unanswered because no tool actually enable researchers to solve them. Unfortunately, the investigation of metropolitan ad hoc networks has an even greater need of simulating

Indeed, investigating ad hoc networks poses a fundamental problem: due to the lack of existing networks, carrying out experiments on real devices is not possible. At best this could be done on a small set of devices, but doing this would not help much in producing relevant results. The only workable alternative turns out to be software-based MANETs simulation. Unfortunately simulation does not really solve the issue: most simulators do not feature realistic network models (in terms of station mobility and propagation of the radio waves) and do not permit the simulation of sensibly large networks. For example, the most popular mobility model (random waypoint) has been found by Yoon and al. [31] to have harmful side-effects on the simulation. Efforts towards realistic models have been observed only recently through projects like the NAB [15] and Jane [11, 20] simulators, the UDEL [7] models, etc. These initiatives all attempt to allow the simulation of metropolitan ad hoc networks. Unfortunately these are on-going research projects, they are hence hardly re-usable as is. Therefore, the studies of metropolitan MANETs may require the development of custom simulators.

Our initial purpose is the investigation of the ad hoc grid issue. By looking for an experimentation platform, we realized that developing a custom solution was the most sensible choice, hence this article.

This article presents the *madhoc* simulator. *madhoc* is a metropolitan ad hoc network simulator targeting the investigation of ad hoc grid-computing [3]. It features the components required for both realistic and large-scale simulations, as well as the tools essential to an effective monitoring of the simulated applications.

Chapter 2

Models

In this section, we will present, sequentially, the models for the network, for the ad hoc applications and for the simulation engine.

2.1 Network model

A good place to start with *madhoc* is probably to understand what does it simulate. In this section we distinguish the hardware and software parts of the network model.

2.1.1 Hardware

madhoc models an heterogeneous mobile ad hoc network. Mobility does not mean that all nodes are in a continuous move. It simply defines that nodes have the ability to move.

Different devices

On today's electronics market, mobile communicating devices can be found embedded on laptops, PDAs and mobile phones. These machines are very different in their technical aspects and in the way they are used. More precisely, Laptops are much more capable (in terms of computations) than a PDA, and the latter is in turn much more capable than a mobile phone. A mobile phone is mobile, as well as PDA and laptops. However, when switched on, PDAs and laptops usually do not move (except in the case of people working in a vehicle). This constitutes the major difference between these devices.

Different networking technologies

Mobile phones, PDAs and laptops do not feature (for the moment) the same communicating technologies. More precisely, the very limited energy storage of a mobile phone refrains it from using powerful radio signals. Hence mobile phones

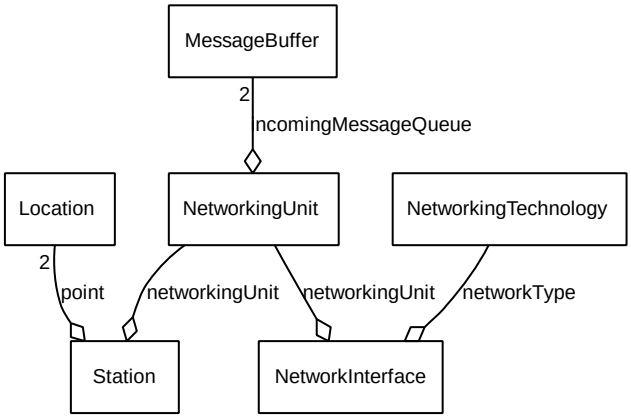

Figure 2.1: *The station model*

(as well as PDAs) are generally operating in a 8-12 meters range, thanks to their built-in Bluetooth adapter (although some mobile phone/PDAs equipped with WiFi interface are being appearing for a short while). On the other hand, all today's laptops feature WiFi network interfaces, all new devices come with a high-speed WiFi interface which provide a bandwidth up to 54Mbps. Some of them also integrate a Bluetooth adapter. These divergences in the way devices are equipped have a serious impact on the topology of the network. Imagine a Bluetooth mobile phone moving in an area populated with dozens of WiFi laptops. Since WiFi and Bluetooth are incompliant, the mobile phone won't be able to establish any connection with its geographical neighbors.

madhoc currently supports WiFi (IEEE802.11b), Bluetooth and Wireless USB. These protocols, which mostly stand at the PHY and MAC layers, are not modeled with detail. Protocols are represented in terms of:

- bandwidth: the bandwidth is shared by all communicating devices operating on a common media. All devices have the same chance to send/receive data.
- range of coverage: define the maximum distance to/from which the devices can receive data;
- packet size: emitted data is organized into packets. A packet is the smallest chunk of data that can be emitted over the network, for a given protocol.
- data transfer cost (see figure2.2): define the price for emitting one byte over the network. *madhoc* defines several basic cost models.

This model permits to consider the impact on communication technologies on network connectivity and bring heterogeneity.

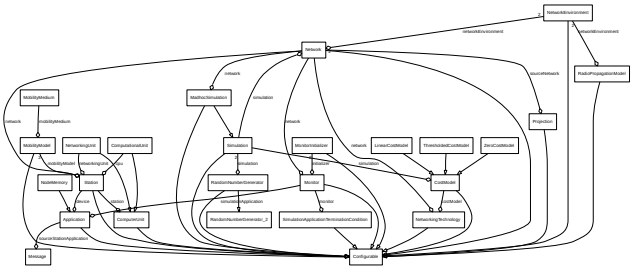

Figure 2.2: *madhoc features a cost model.*

Different computational/storage capacities

Initially targeting the analysis of the future ad hoc grids, *madhoc* integrate a basic model for the ad hoc grid node. More precisely, it endows the nodes with computational power and storage ability. *madhoc* assumes that those abilities depends on the type of device considered. A laptop will generally embed a fast processor coupled with a quite large central memory and a several gigabytes hard-disk. PDAs and mobile phone embeds energy-efficient components, hence their computation capacities is severely limited.

Different mobility models

madhoc defines a set of mobility models. A mobility model define how nodes move. Currently *madhoc* implements the random mobility model, the random waypoint mobility model and the human mobility model. The latter is *madhoc* specific an is highlighted in section 3.2.

madhoc's architecture permits node to have distinct mobility models. This is achievable by using the simulation in framework mode (see section 4.4.2).

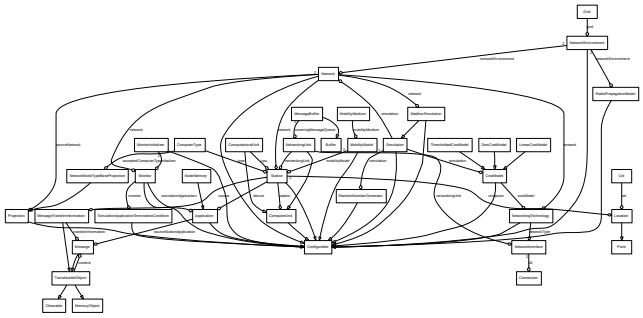

Figure 2.3: *The general mobility framework on top of which all mobility models are implemented.*

2.1.2 Software

In order to enable node-2-node communication, *madhoc* implements some basic mechanisms which allow the nodes to asynchronously send messages to each others.

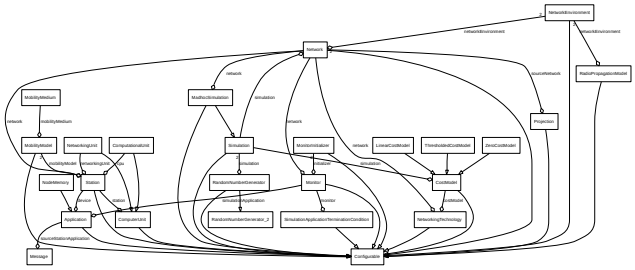

Figure 2.4: *The model for the network.*

PHY and MAC layers

As explained in section 2.1.1, *madhoc* models the PHY and MAC layers only in terms of available bandwidth, signal power and packet size. Collisions, interferences are not modeled by statistical models. This makes the simulator lightweight and consequently enables the simulation of large networks. In the code, *madhoc* does not make any clear difference between PHY and MAC layers. Their properties are both expressed within the same algorithms.

Network and transport layers

madhoc consider that in the context of large heterogeneous MANETs, multi-hop networking is unachievable. Then it does not feature any internal mechanism for multi-hop applications like routing. This limitation is not a problem as *madhoc* aims at simulating localized applications.

2.2 Ad hoc application model

2.2.1 Application

Definition

There is not yet any universally accepted definition of what is an ad hoc application. In our context, an ad hoc application is a piece of software that is executed in the mobile devices and is able to communicate in its steadily changing environment. The application is not the global thing that run on all nodes

in a network, it is the piece of code that run on one particular node. The application can be executed on only a subset of the nodes in the network. Moreover, several applications can be executed on one single node.

Because the environment is changing, the application must then be robust to connections/disconnections of peers. As this robustness depends on the application, it cannot be defined at a middleware-level. The middleware cannot do more than providing convenient facilities for the applications to obtain information about their changing environment.

Connection/disconnections

In order to enable the nodes to be aware of new neighbors, simulators usually use event-driven notification. More precisely, when the simulator finds a new link, it dynamically invokes a method on the two nodes involved. This method, initially abstract, needs to be implemented by the application that defines what to do in case of a connection/disconnection event. This application of the *observer* design pattern, based on polymorphism, has proved elegant and flexible. However, *madhoc* does not uses this because, based on our experience of ad hoc programming, we assumed that event-based strategies bring more complexity than flexibility. *madhoc* then does not dynamically invoke anything on nodes newly in range. Instead it provides all nodes with an utility method that returns a set of the nodes that were not yet in range last time it was invoked. Applications are then free to invoke this method or not.

Note: Another drawback of the event-based notification is that the code of handler methods is executed by the simulator in a routine dedicated to the construction of the interconnection network: a routine that should not execute any application-level code.

Application identification

As nodes can execute several applications, there is the need to have a way of identifying the different applications running on one given node. *madhoc* defines than applications running on a given node are identified by their class (similarly, IP identifies processes by their port numbers). Consequently, the situation in which two applications of the same class would run on the same node (conflicting services) cannot happen.

Node-2-node communication

Application communicate exclusively in asynchronous and unconnected mode by using the message-passing paradigm. *madhoc* does not implement any "connected-mode" connection patterns such as sockets. A message can be sent in unicast (one recipient), multicast (multiple recipients) and broadcast (all nodes in the neighborhood are implicitely recipients) modes. *madhoc* does not strongly define what a message is. It only impose that a message should be transferable across the network links. Then by defining a new kind of message, the user should implement some specific method that will allow *madhoc*

to know the number of bytes used to code the message and to make a copy of the message.

2.2.2 Monitor

Ad hoc applications are inherently distributed over a highly dynamic network infrastructure. Consequently, deployment and monitoring are challenging issues. Simulation allows to put in place things that would not be possible in the real world. Then *madhoc* defines the self-explanatory concept of *monitor*.

Definition

A monitor is an abstraction defined at the level of the simulation. It does not have any instance in the real world. A monitor aims at maintaining a global view on all applications of a given class and at carrying out operations applicable to a set of applications, such as their deployment, their initialization and the observation of the decentralized process.

Measurement

The greatest benefit of monitors is that they provide the user with virtual sensors which allow the user to observe what is going on in the simulation process. Just like a physical sensor (thermometer, barometer, etc), a *madhoc* sensor will take measures on a given system. Measures are taken at each iteration. Each new measure is appended to an history.

There exist several sorts of measures as the phenomenon that are sensed are of various nature. If the phenomenon can be sensed as a numerical value (like the network throughput and the simulation time that can be expressed as numbers) the measure is said to be *numerical*. It is important to distinguish numerical measure from the other as their nature allow specific things to be done with them (e.g. mathematical operations, graphical rendering, etc).

Unfortunately, sensing takes time. The more sensors are defined, the longer will take the simulation. Then, for the development process, it is advisable to use as many sensors are available in order to detect the slightest flaw of the code, but for the research experimentations, the user should enable only the metrics he is interested in, by discarding the others.

2.3 Simulation model

This section explains how *madhoc* models the concept of simulation and details some of its important aspects, for our point of view, in the case of the simulation of MANETs.

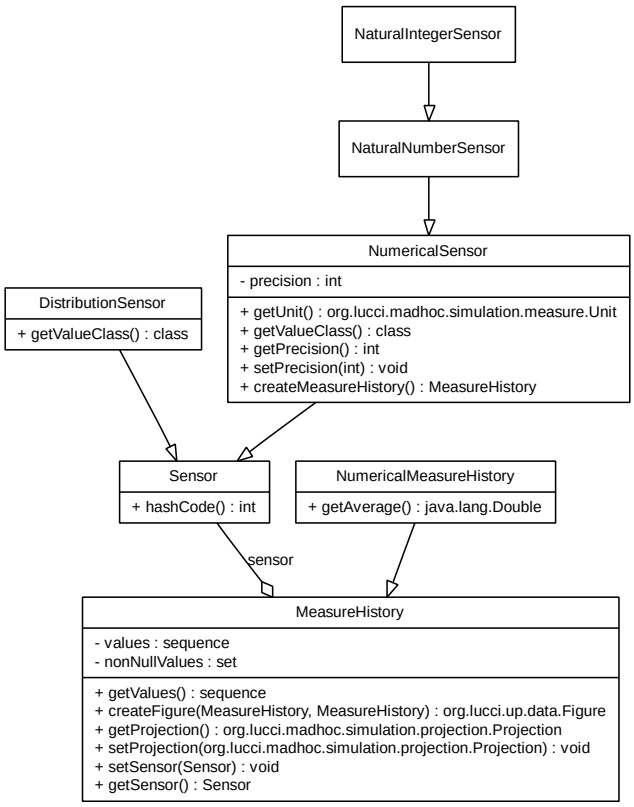

Figure 2.5: *The measurement model.*

2.3.1 Languages

All the executable code within *madhoc* is written in Java. Dissimilarly, *ns-2* and many other simulators use a natively compiled language such as C or C++ in their core and require the user to write extension in some interpreted language like TCL. Nothing like this exists in *madhoc*. Here the only things that are not written in Java are the configuration files, if any. Then, a new extension consists of some `.class` files that contain the executable code and optional configuration file that enables the user to interact with the extension.

2.3.2 Discrete-event, discrete time, resolution...

Unlike many other simulators, *madhoc* does not rely on the discrete-event simulation paradigm. Instead of jumping from event to event, *madhoc*'s kernel iterates upon time. Time is discrete. The time-interval between two iterations is called the *resolution*.

The resolution of a simulation process is defined by the user. The configuration key *simulation_resolution* allows the user to modify it. You should of course use the same resolution for all the experiments otherwise your result cannot be assumed to be consistent. The greatest is the resolution, the faster—and the less accurate—is the simulation process. The resolution of the simulation ideally depends on the application you want to simulate. In the specific case of RAD-based broadcasting, the resolution should be at least twice lower than the maximum RAD otherwise the benefit of using a RAD is simply lost. In the specific case of the simulation of mobility, the resolution should ensure that mobile node move with reasonably small steps, otherwise some connections that would have occurred in the real world would not be simulated. In our case—RAD-based broadcasting in mobile network, the definition of the resolution should fit the constraints brought by both broadcasting and mobility.

2.3.3 Simulation area and projections

Simulation area

For the sake of simplicity, *madhoc* defines that nodes evolve in a square-shape surface. The surface of the simulation area can be defined by using the $simulation\_area\_surface$. It is expressed in square meters. Because *madhoc* relies on a specific technique called *bining* (bining is a general modeling technique which consists in dividing a considered zone in a grid—multi-dimensional griding is possible—so as operations on the zone are executed with a lower complexity), the surface exhibits the following constraint: it must be multiple of 10,000. The number of nodes in the simulation area is constant: nodes cannot step out of the simulation area as well as no new node can step in.

Projections

In many cases the user will want to consider only a sub-network. For example, the user may want to observe what is going on only on a specific zone of the simulation area, he may want to observe what happens only on Bluetooth nodes, etc. In order to make this possible, *madhoc* defines the concept of *network projection*. Just like the mathematical definition of projection, a projection of the simulated network will result in a set of the nodes that match the criteria defined by the projection itself. The communication links retained in the projection are the links whose two participants are part of the projection. The sensors available in the simulation will be used on each projection, generating several measure repository. Several projections has been predefined.

As convenient as projections are, processing the projections is a cumbersome task for the simulator: at each iteration it needs to find out which nodes belong to a projection, and then apply the measurement available on this newly created set of nodes. It is then important that the user define only the projections that are relevant to him.

madhoc has a set of pre-defined projections. Some of them are described in the following:

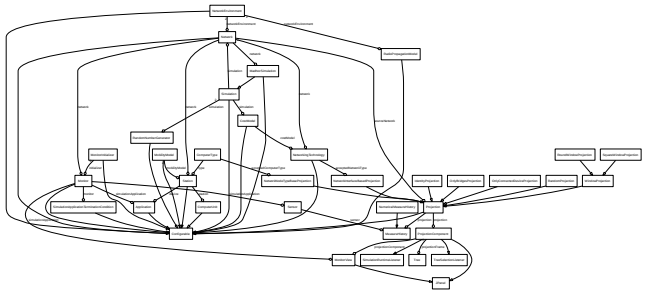

Figure 2.6: *The model of network projections.*

identity projection gathers all the stations in the initial network. This is the most basic projection as it projects everything.

window projection considers a sub-area of the simulation area whose center is the center of the latter. The window can be either a square or a circle. The size of the window is parametrizable.

network technology-based projection considers the subset of the nodes in the network that embeds a network adapter of a given type.

only-connected-nodes projection gathers all the node that have at least one neighbor.

random projection is a projection made of a set of nodes that are randomly selected. This projection is useful if the user wants to operate on only a subset of the simulated nodes—for performance reasons, for example—while keeping a certain degree of statistical confidence.

2.3.4 Random number generation

madhoc provides a build-in customizable random number generator.

The configuration key *random_number_generator_seed* allows the user to specify the seed for the random number generator. Any integer number is allowed. The value *ip* uses the IP address of the local computer as the seed. The value *time* uses the current time. The value $time + ip$ uses a mix of both. This allows the experiments to be potentially reproducible.

2.3.5 Scheduling

The application scheduler implemented in the core of the simulator works in a sequential manner. This had be chosen for performance issue: dedicating one thread by node would have been way too cumbersome and the number of simulated nodes would have been a lot lower.

Practically, the scheduler manages a list of the simulated nodes. It iterates on this list by invoking the `run()` method on every nodes.

On the use of stochasticity for scheduling

Initially the list of nodes on which the scheduler iterated always featured the same order. On runtime, we could notice that some nodes processed faster that others, that some nodes transmitted over the network without any problems while other nodes constantly could not. We then noticed that on the low-bandwidth communication links, only a few nodes were able to transmit before the link is saturated, the others—if any— were blocked. Moreover these privileged few nodes were always the same, the scheduling was hence unfair.

In order to give all application the same chance to execute and to emit data on the network during an iteration of the simulator, the application scheduler executes in a stochastic fashion. More precisely, along one single iteration, all applications are asked to execute multiples times, each time for a very short period. The order the applications is randomized. At each turn all applications are invoked, but in an non-predefined order.

2.3.6 Termination condition

madhoc is a general simulator. As such, it provides a general model for the termination condition of the simulations, as explained in this section. A simulation is said to be terminated if all the applications it executes are terminated, if the concept of "termination" makes sense. More precisely, a broadcasting application can be said to be terminated when all the nodes are reached, or when the broadcast message has expired. Whatever it is, the termination condition is easy to define. In the case of beaconing applications (the application that steadily sends "hello" messages in the neighborhood), the concept of "termination" does not make sense. A beaconing application should never terminate, unless the node may want to become invisible, which is out of scope here.

Then, if all the applications that define a termination condition are terminated (the termination condition is true), the simulation itself can be said to be terminated.

As the termination condition of an application depends to itself, it must be defined by the programmer.

2.3.7 Application deployment

In this section we will explain how an application gets deployed within *madhoc*.

Deploying the application

A monitor is not only in charge of taking measures on the applications its represents. It is also in charge of deploying the application accross the network. On startup, *madhoc* checks the configuration to get the list of monitors that

needs to be created. From each monitor class found, *madhoc* instantiates a monitor object and delegate to it the deployment of the application. For every node in the network, the monitor will instantiate an application object. When this is done, the application is ready to be executed.

Taking measures

However, if nothing more is done, the application does not give any feedback to the user of how it works. Then the monitor instantiates the sensors that will, at each iteration, take measures on the application. In the case of the graphical model, these measures can be displayed at once on the interface. In the case of the console mode, they will be printed on stdout when the simulation will have finished.

Keeping an eye

In the specific case of the graphical interface, *madhoc* allows the user to create graphical views on the applications it develops. This views, which are standard Swing components, will be dynamically integrated into the standard interface. Moreover, their content will be updated at each iteration of the simulation process. The user can then have a precise view of the simulated application.

Chapter 3

Scenarios

In [6], Bohacek states that a simulator does not always need to be realistic but only must stress the protocol so as we can be sure that it will operate in the real world. Up to now, most studies have relied on randomized mobility models [4][30][18][22], especially on the Random Waypoint Mobility Model [31]. Thanks to several studies focusing on the wallop of random waypoint mobility model [31][5][27], researchers are now aware of the harmful impact of stochastic mobility patterns. One the other hand, an effort towards more realistic mobility models can be observed through papers and projects like the Group Mobility Model [14], the Graph-based Mobility Model [28], the Obstacle Mobility Model [17][16], the UDEL model [6][7], and the GEMM project [23]. Surveys of mobility models can be found in [9][23]. Studying the impact of realistic mobility model on distributed applications, Tugcu and Ersoy [29] have shown that the choice of the mobility model has a significant impact on the performance of the mobile systems. The effect on the relative performance becomes more important especially when the algorithms try to predict the mobility of the nodes.

madhoc currently defines two mobility models: random waypoint and human mobility.

The following table gives the name of the configuration keys that need to be set in order to define a given value for the simulation.

Parameter	*Configuration key*
Simulation area surface	`simulation_area_surface`
Mobile phone density	`network_phone_density`
PDA density	`network_pager_density`
Laptop density	`network_laptop_density`
Hotspot density	`network_hotspot_density`

3.1 Random waypoint

Random waypoint is a commonly used mobility model. In the random waypoint mobility model, as described in [**?**], each station chooses randomly a destination

in the simulation area and moves towards this destination with a randomly chosen velocity. When the destination is reached, the station remains at the same place for a while. This process is repeated by each station until the end of the simulation. In addition to that, within *madhoc*, in order to reduce the "harmful" behavior inherent to the random waypoint mobility model [31], each simulation process is initialized using the stationary distribution pattern [21].

Random waypoint is supported in *madhoc* for the sake of compatibility and respect of the previous works in the domain.

Parameter name	*Configuration key*
Node velocity interval	`random_waypoint_mobility_velocity_interval`
Pause time interval	`random_waypoint_mobility_pause_interval`

3.1.1 City place

Parameter name	*Value*	*Unit*
Simulation area surface	200×200	m^2
Mobile phone density	3.000	$node/km^2$
PDA density	500	$node/km^2$
Laptop density	100	$node/km^2$
Velocity interval	[0.3 1]	m/s^{-1}
Pause duration interval	[0 10]	s

As illustrated on figure 3.1, the network which is formed by the presence of people walking accross a city place. Because a city place is an open area, no obstacle prevent the propagation of radio waves. Consequently the network is well connected. The devices not equipped with a WiFi adapter tend to be isolated due to their short coverage radius.

3.1.2 Market place

Parameter name	*Value*	*Unit*
Simulation area surface	200×200	m^2
Mobile phone density	5.000	$node/km^2$
PDA density	1000	$node/km^2$
Laptop density	10	$node/km^2$
Velocity interval	[0.3 1]	m/s^{-1}
Pause duration interval	[0.01 60]	s

Figure 3.2 shows the result in terms of node location of the market place environment.

As illustrated in figure 3.3, the resulting network is heavily connected. This comes because of the high density of devices and the lack of obstacles.

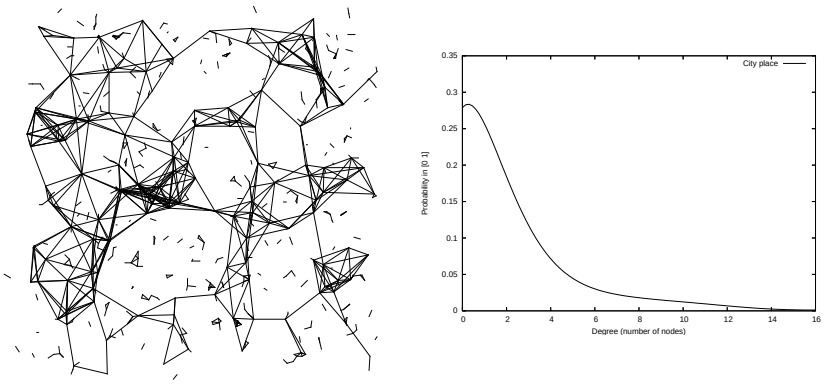

Figure 3.1: *On the left, an example of a network formed by people wandering accross a place in a city. On the right, the average degree distribution for such networks.*

3.1.3 Concert place

Parameter name	*Value*	*Unit*
Simulation area surface	50×50	m^2
Mobile phone density	5000	$node/km^2$
PDA density	1000	$node/km^2$
Laptop density	0	$node/km^2$
Velocity interval	[0 0.2]	m/s^{-1}
Pause duration interval	[5 600]	s

The concert place and the market place environment lead to a similar connectivity. This is because of similar densities. The main difference between these two environment resides in the velocity. More precisely, while nodes in a market place have the speed of a pedestrian, those in a concert place barely move. This difference of velocity of the nodes is not visible on static views as these presented here but has a great impact on how protocol process. In the specific case of a dynamic broadcasting protocol, the movements in the market place environment, which result to many connection/disconnections between nodes, may entail lots of re-broadcasting of a packet.

3.2 Human mobility

Human mobility is the default mobility model in *madhoc*. It is a generic mobility model that roughly represent the "clever" mobility of people in metropolitan areas.

Figure 3.2: *4,000 nodes moving according to the market place environment.*

3.2.1 On metropolitan mobility

Simulating metropolitan mobility is difficult because urban zones feature an extensive list of dissimilar configurations (avenues, pedestrian areas, places, shopping malls, etc). Building a generic model that take into consideration all the components of a city is a daunting task. So far, attempts at modeling metropolitan mobility focus on specific configurations. The Manhattan mobility model is the most relevant initiative. The city-chapter and the graph-based mobility models [9] are generalizations of it. In the Manhattan model, the station is allowed to move along the horizontal or vertical streets on the urban map. Unfortunately the model fails at representing important characteristics of metropolitan environment, such as the existence of shopping zones, the width of the streets, etc. Moreover, nothing is said about the propagation of radio waves.

3.2.2 Human mobility, a mobility model driven by the *intentions* of humans

Humans do not move randomly. When moving, they have a determined target spot and move towards it. The target may be a few meters away (next shop, next crossroads, other sidewalk, etc) as well as far away (next district, next city, etc).

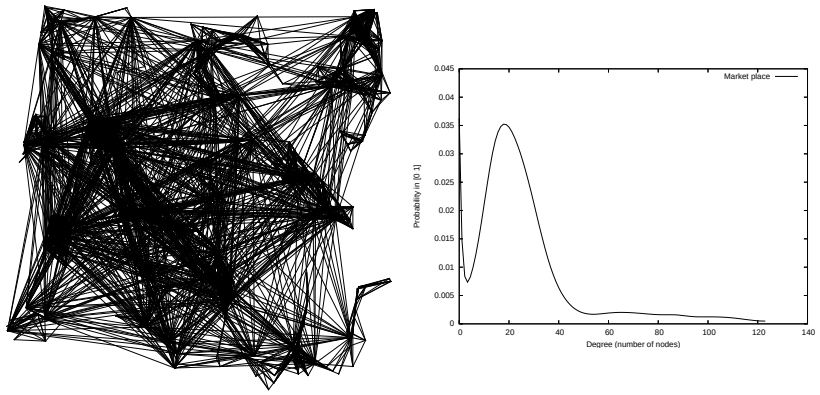

Figure 3.3: *On the left, an example of a network formed by people wandering accross in a market. On the right, the average degree distribution for such networks.*

Upon time, their target change. Most of the time, people have a dynamically changing list of targets. This list can be composed of two spots (like a secretary regularly moving from her office to her boss' office and vice versa) or more (people shopping). This list of target spots is dynamically changing because of various parameters that will appear upon time (locations of the target places, closure times, high frequentation times, etc). The behavior described in this paragraph is different from what is proposed by random waypoint. Our scenario can be said to be some constrained waypoint (the constrained waypoint mobility model does not rely on randomness, then the destinations and pause times are given by a scenario) enriched with advanced rules based on human's decision. The human mobility model proposed by *madhoc* models this.

The general principles of human mobility

The human mobility defines that the simulation area is populated with fixed places that constitutes targets for the nodes in move. Places are randomly located within the mall so as the distance between two given places must not be lower than 10m. A place has a round shape (whose radius is randomly chosen in a given interval) surrounded with walls. Walls are important because they obstruct mobility and attenuate the radio signal. People within a given place move according to the random mobility model. The speed of nodes moving within a spot is not defined by the human mobility model as it depends on the kind of environment that is modeled.

By randomly moving within a given place, nodes soon or later reach the edge of the place, then they are considered to be gone out of the place. At this moment the next target place it chosen. The decision of the next target place is based on the list of places that still have not yet visited and on the distance

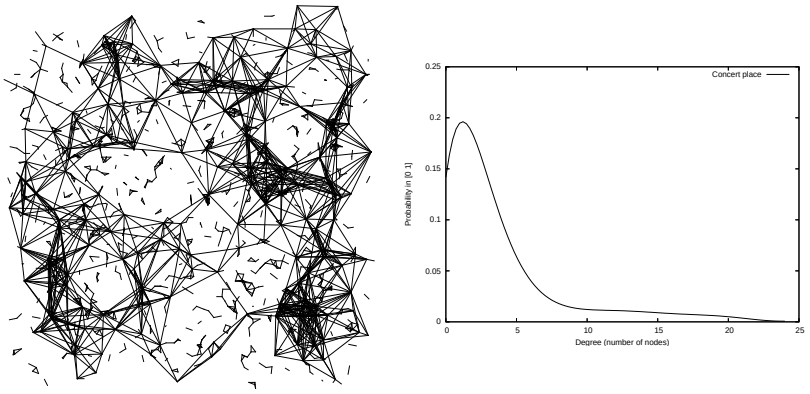

Figure 3.4: *On the left, an example of a network formed by people attending a concert. On the right, the average degree distribution for such networks.*

between the current location of the node and the targets so as the closest places are preferred. Once the list of places that have not yet visited is empty, it is refilled with all the places in the simulation.

Emerging topology of human mobility

As shown on figure 3.6, the human mobility model can generates the emergence of dense and highly connected regions. These regions are sometimes isolated, meaning that they are not connected to the rest of the network. At runtime, one can observe that they sporadically get connected to other regions thin paths (like chains). This kind of topology illustrates that applications—as well as protocols—must deal with high variations of the density.

By changing the parameters of the human mobility, we can achieve the simulation of various types of mobility: mobility within mall centers, in the city, in a train station and so forth. In the next section we propose some mobility models built on top of the human mobility model.

3.3 Real-world mobility based on human mobility

Mobility impacts the topology on the network, but not only. The speed of the nodes also have a very important impact on the dynamicity. More precisely, a network in which the nodes move quickly could perfectly have the same topology than one in which the nodes move slowly—meaning that the two networks exhibit the same degree distribution, the same number of partitions, etc. But if the nodes move faster, the network is very likely to be extremely dynamic.

This section provides the parameters used for modeling various metropolitan networks with *madhoc*.

Parameter name	*Value*
Spot density	`human_environment_spot_density`
Spot radius	`human_environment_spot_radius`
Wall obstruction ratio	`human_environment_wall_obstruction`
Out-spot velocity	`human_mobility_out_spot_speed`
In-spot velocity	`human_mobility_in_spot_speed`

3.3.1 Highway

Parameter name	*Value*	*Unit*
Simulation area surface	1000×1000	m^2
Mobile phone density	50	$node/km^2$
PDA density	20	$node/km^2$
Laptop density	10	$node/km^2$
Spot density	1	$spot/km^2$
Spot radius	[10 10]	m
Wall obstruction ratio	0.1	
Out-spot velocity	[20 40]	m/s^{-1}
In-spot velocity	[20 40]	m/s^{-1}

The highway mobility environment is characterized by nodes moving at a very high speed. The density of spot is defined so as there are preferably only 3 spots in the simulation area. Numerous spots would result in too many intersections of nodes which would lead to unrealistic behavior. The highway environment is very different from the other ones in the sense that the connectivity it produce it specific: most links are organized into chains of nodes moving in opposite senses.

3.3.2 Mall

The mall mobility model represents the mobility of nodes with an area made of shops connected by corridors. Spot model shops.

Parameter name	*Value*	*Unit*
Simulation area surface	300×300	m^2
Spot density	1000	$place/km^2$
Spot radius	[1 10]	m
Wall obstruction ratio	0.7	
Mobile phone density	5000	$node/km^2$
PDA density	1000	$node/km^2$
Laptop density	500	$node/km^2$
Hotspot density	0	$node/km^2$
Out-spot velocity	[0.3 1]	m/s^{-1}
In-spot velocity	[0.3 0.8]	m/s^{-1}

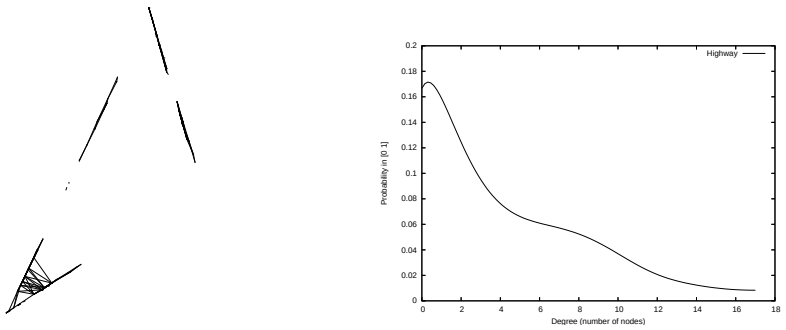

Figure 3.5: *On the left, an example of a network formed by cars driving on a highway section. On the right, the average degree distribution for such networks.*

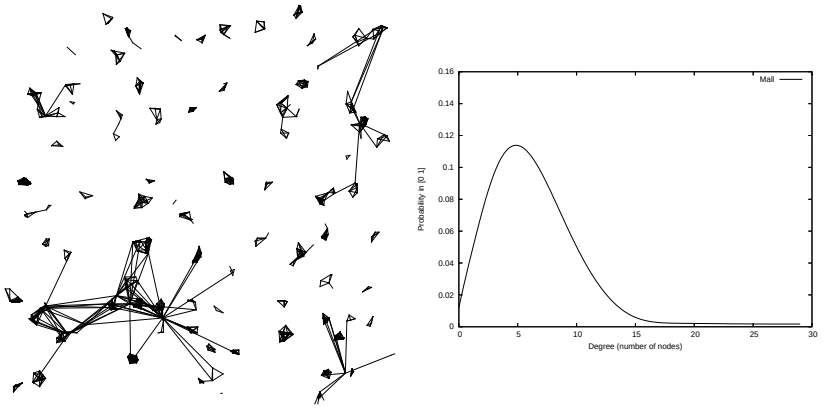

Figure 3.6: *On the left, an example of a network formed by people shopping in a mall. On the right, the average degree distribution for such networks.*

3.3.3 Streets

The street environment model the network formed by vehicle driving in a city section. Here spots represent the crossroads. The resulting connectivity exhibit some very high dense regions (in terms of communication links), which are the crossroads, connected by thin paths formed by nodes moving along streets.

Parameter name	*Value*	*Unit*
Simulation area surface	1000×1000	m^2
Spot density	50	$place/km^2$
Spot radius	[3 15]	m
Wall obstruction ratio	0.9	
Mobile phone density	5000	$node/km^2$
PDA density	1000	$node/km^2$
Laptop density	500	$node/km^2$
Hotspot density	0	$node/km^2$
Out-spot velocity	[1 20]	m/s^{-1}
In-spot velocity	[0.3 20]	m/s^{-1}

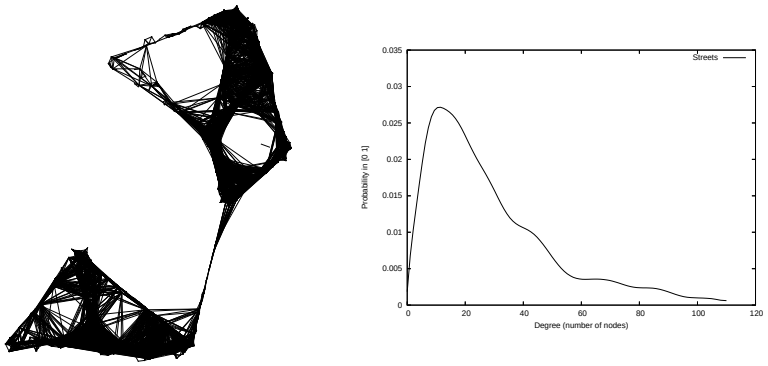

Figure 3.7: *On the left, an example of a network formed by cars driving in a city section. On the right, the average degree distribution for such networks.*

3.3.4 Supermarket

The spot represent the edges of the

Parameter name	*Value*	*Unit*
Simulation area surface	50×50	m^2
Spot density	$100,000$	$place/km^2$
Spot radius	[3 15]	m
Wall obstruction ratio	0.2	
Mobile phone density	$50,000$	$node/km^2$
PDA density	$30,000$	$node/km^2$
Laptop density	500	$node/km^2$
Out-spot velocity	[0.3 1.5]	m/s^{-1}
In-spot velocity	[0.1 0.5]	m/s^{-1}

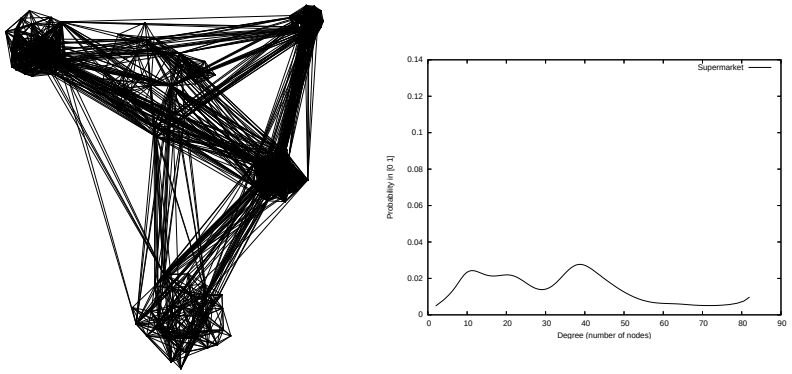

Figure 3.8: *On the left, an example of a network formed by people shopping in a supermarket. On the right, the average degree distribution for such networks.*

3.3.5 Train station

The train station environment model pedestrian walking in the hall of a train station, going to shops, buying tickets... The resulting movement for the node is slow, with lots of pauses.

Parameter name	*Value*	*Unit*
Simulation area surface	100×100	m^2
Spot density	$10,000$	$place/km^2$
Spot radius	[1 2]	m
Wall obstruction ratio	0.1	
Mobile phone density	$8,000$	$node/km^2$
PDA density	$2,000$	$node/km^2$
Laptop density	500	$node/km^2$
Out-spot velocity	[0.3 1]	m/s^{-1}
In-spot velocity	[0.3 0.3]	m/s^{-1}

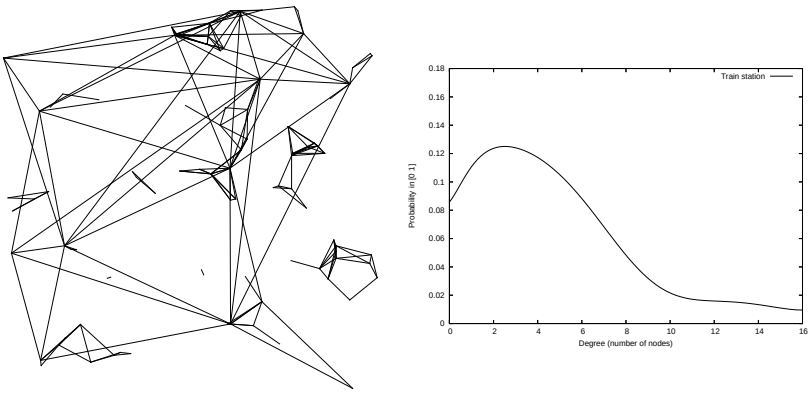

Figure 3.9: *On the left, an example of a network formed by people roaming in a train station. On the right, the average degree distribution for such networks.*

Chapter 4

Using *madhoc*

4.1 Requirements

madhoc requires the version 1.5 of the java runtime environment. This software is freely available on Sun's website. It also makes use of the following projects:

Up is a Swing component dedicated to the rendering of numeric data. It may be used for scientific, statistic, geometric data rendering. More generally, it is able to render any object that can be said to be composed of points. Up keeps things easy using a very common object-oriented model: the composite design pattern. Practically, Up merely renders a figure, assuming that a figure is composed of points and child figures. Up also automatically adapts the axis system according to the graphical environment.

EPSGraphics is a package that allow Java application (*Up* in our case) to produce PostScript files. It is required only if you want to generate PostScript files.

iText is a package that allow Java application (*Up* in our case) to produce PDF files. It is required only if you want to generate PDF files.

Javamail is not mandatory but it allows *madhoc* to send me a e-mail message in case it encounters an problem. This is totally transparent to the user. Javamail is Sun product and is freely downloadable from Sun's website. Note that Javamail requires the Sun's activaction toolkit which is also free.

4.2 Packaging

madhoc comes as a set of jar files. The jar files which come with the *madhoc* distribution contain both the source code and the compiled bytecode of *madhoc*. In the distribution you have, the source code might be obsfucated for

performance reasons. If it is not the case, feel free to unjar the source and look at it. The following list give some explanations on the content and role of each jar file.

Madhoc.jar is the core of the *madhoc* simulator, which is absolutely necessary for running it ;

Madhoc-Broadcasting.jar contains the implementation of standard broadcasting protocols ;

Madhoc-HumanMobility.jar contains the code for the mall mobility model ;

Madhoc-NetworkMonitor.jar contains the code (sensors, graphical views, etc) for monitoring the general behavior of the network ;

Madhoc-AdHocComputing.jar contains the code for ad hoc scattering which is the reverse operation to broadcasting ;

Madhoc-BroadcastingMalaga.jar features some specific code developed for the need of the researchers at Màlaga university ;

Toools.jar is a set of utility classes used by *madhoc* ;

Up.jar permits the 2D graphical representation of the data collected by the sensor plus various data structures ; Up is a standalone project hosted at `http://amy.sunsite.dk/up` ;

skinlf.jar permits the use of the Skin Look&Feel, which is required by some version of *madhoc*. This jar file is needed only if you use the graphical interface and want to have the use this look&feel.

4.3 Installation

As *madhoc* is a pure Java application, installing it on any computer is trivial. *madhoc* comes with a set of `.jar` files. These files can be freely downloaded at `http://www-lih.univ-lehavre.fr/~hogie/madhoc`. Please take care at downloading the most recent files, which can be identifiable by looking at its date, which is part of its filename.

Copy these files anywhere in your file system and include them in the classpath. At this step, *madhoc* is completely installed on your computer.

4.3.1 Installation on UNIX operating systems

On UNIX-type operating systems (including Linux), the easiest way of installing *madhoc* (as well of any pure-java application) is to copy the `.jar` files in your `$HOME/lib` directory. Then the `$HOME/.profile` file (or another file, depending on your shell) should have the following command, which automatically include

into the `CLASSPATH` environment variable all the jar files found in the `$HOME/lib` directory.

```
for j in $HOME/lib/*.jar
do
    CLASSPATH=$CLASSPATH:"$f"
done

export CLASSPATH
```

4.3.2 Installation on Windows operating systems

There is no de-facto standard for installing Java programs on the Windows operating system. The installation of *madhoc* will then depend on your java runtime environment. Note that on Windows, the PATH separator character which seperate the jar file paths within the CLASSPATH variable is the ";" character (on UNIX, the path separator is ":").

Installation on Cygwin environments

Cygwin is an UNIX environment built on top of the Windows platform. It is a very good option to those who do not wish to set up a dual-boot system, and turns out to me more flexible solution than the latter. Installing *madhoc* on Cygwin is no problem. The user simply got to pay attention to the fact that Cygwin does not runs a UNIX JDK but a Windows one. Consequently, the PATH separator the user need to specify on in Cygwin configuration is ";". The way Cygwin manages the file system poses also some troubles. For example, it turns out to be better to write `c:` then `/cygdrive/c`, which theoretically both refer to the C logical disk drive.

4.4 Using *madhoc*

madhoc can be used in two different ways: as a framework or as a standalone application. On the one hand, using it as a framework consits of manipulating its API (application programming interface) from a user program. Doing this requires a advanced knowledge of the Java programming language and a good understanding of *madhoc* object-oriented model. On the other hand, using *madhoc* as a standalone application consists of executing the *madhoc* application just like any other application: within a process of the operating system. In this case *madhoc* works pretty much like a Unix filter: it accepts its input data from the command line or as a command-line argument and generate its output on the standard output or to a file.

Whether *madhoc* is used as a framework or as a standalone application, it needs to be fed with some configuration information. This information defines what to do and how to do it.

Formally speaking, the configuration is a set of key/value entries in which each key is a unique string and the value is a set of strings. Pragmatically speaking, the way the configuration information is passed to *madhoc* depends on the way the simulator is used.

4.4.1 Configuring *madhoc* as a standalone application

In the case of running *madhoc* as a standalone application, the configuration information must be passed either by the standard input of the process or by providing it as an ASCII text file whose name is specified on *madhoc*'s command line. Except from the lines beginning with the "#" character which are considered as comment lines and blank lines which all are skipped, all lines constituting a configuration text must strictly respect the following BNF (Bacus Normal Form) definition:

$$line := key = \{value(, value)*\} key := [^{=}] + value := [^{,}\}]*$$

Here follows as example of correct configuration entries:

```
# some comment
simulation_name = {my simulation}
simulation_resolution = {10}    # some other comment
```

Each line is called a configuration entry. Entries are mandatory as soon as *madhoc* ask for them, since *madhoc* does not make any assumption of default values. If a configuration entry cannot be found in the configuration provided by the user, *madhoc* immediately interrupts and prints an explicit error message on the standard error.

The configuration can be wholly contained in a single file as well as it can be split in several files. Indeed, when scanning its command-line parameters, *madhoc* check if a given parameter is a directory or not. If the parameter is a normal file (not a directory) whose the name ends with `.madhoc`, it is considered to be a configuration file and is used as such. If the given command-line parameter specifies a directory, the latter is scanned to configuration files. A full example of a configuration (split in several files for the sake of clarity) can be downloaded at:

`http://www-lih.univ-lehavre.fr/~hogie/madhoc/code/config.zip`

There exists a long list of configuration keys available in the default *madhoc*. The example configuration file whose the url was given in the previous paragraph contains all the configuration keys allowed in *madhoc* as it comes by default. We did not wish to give long explanation on al configuration keys herein. The example configuration file is wealthly documented. Please refer to it for further explanations.

4.4.2 Configuring *madhoc* as a framework

When using *madhoc* as a framework, there is no need to handle text files. Here you need to instantiate a `org.lucci.madhoc.config.ConfigurationKeys` object and set its fields to the appropriate values. The name of the fields of the configuration object The following example is the same of the one in the previous section, adapted to a framework-mode.

```
ConfigurationKeys config = new ConfigurationKeys();
config.simulation\_name = "my simulation";
config.simulation\_resolution = "10";
```

The `ConfigurationKeys` object features all the configuration keys available as public static fields. For each of them, it also define some default values. Please pay attention at setting all the configuration keys you need at an appropriate value. Leaving configuration keys to unappropriate default value is indeed a fairly frequent mistake.

Configuring a new module

If you have developed a new module for *madhoc* (a new protocol, a new monitor or whatsoever), you might want to give the user the ability to configure it.

In the case of *madhoc* as a standalone application, the user just had to add new items in the configuration by editing the configuration files. In the case of using *madhoc* as a framework, there is no possibility to directly modify the configuration object nor the its class as the latter is already compiled and contained in a jar file. Then the user need to subclass the class of the configuration object and define the new configuration items as fields of the newly created subclass.

Why using a configuration object?

In the early versions of *madhoc*, no configuration object existed. The user was then required to directly manipulate the classes of the simulator, just like developers usually use frameworks (like Swing, Collections, etc). But as *madhoc* evolved, it became more and more complex. Manipulating the classes of the recent versions of *madhoc* turns out to be a cumbersome task.

Introducing the configuration object was then a good way of breaking the complexity of the system by turning the *madhoc* user's view to a planar one.

4.5 Using *madhoc* as a standalone application

The executable class that must be invoked for starting *madhoc* is:

```
org.lucci.madhoc.Madhoc
```

There are two ways of invoking it.

You can specifiy a directory and/or configuration files in the command line. Configuration files will be loaded directly while directories will be scanned for configuration files to be loaded.

```
java org.lucci.madhoc.Madhoc config/ a.madhoc b.madhoc
```

Then, the following command loads all the configuration files in the current directory:

```
java org.lucci.madhoc.Madhoc .
```

If you do not specify anything on the command line, *madhoc* reads its configuration for its standard input. You can then invoke *madhoc* this way:

```
cat config.madhoc | java org.lucci.madhoc.Madhoc
```

Warning! In this case, you should however make sure that all the configuration files start or end with a newline character. If not, by concatening two files whose the first does not end with a newline character, the pipe will append the first line of the second file to the last line of the first file, and will hence generate an invalid syntax.

Chapter 5

The graphical user interface

If you wish to launch the graphical user interface, you need to set the `simulation_interaction_mode` to `graphical`. This will completely initialize the simulation and then instantiate a graphical application that will give the user the ability to observe in many ways the simulation process.

When the GUI starts, it opens a desktop environment in which every windows correspond to a distinct projection 2.3.3 of the simulated network, as illustrated on image 5.1 Projections are easily identifiable since their name entitles the corresponding window.

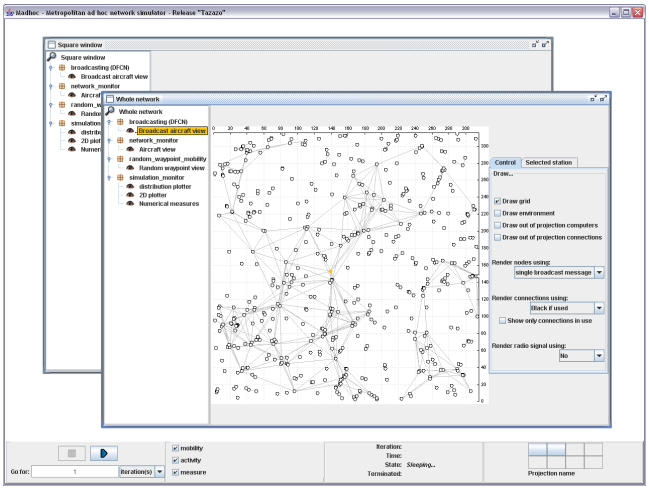

Figure 5.1: On start-up, *madhoc* open one window for each projection available

All windows have the same features. A window is split vertically in two zones. As shown on image 5.2 the left side of the window shows a tree widget (graphical component). This tree is three-levels-deep and is organized as follows:

root is the name of the simulation

first level shows the monitors 2.2.2 available

second level (the deepest one) shows the *views* made available by each monitor (see section 5.1).

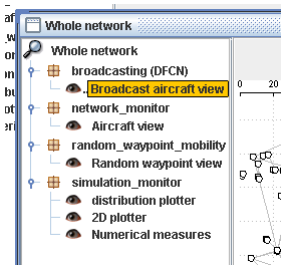

Figure 5.2: A tree for selecting the views.

5.1 Monitor views

A view is a widget that permit the graphical rendering of some information exposed by its monitor. Views are shown on the right side of the projection window. More than one view is showable simultaneously. Showing one single views is achievable by simply selecting the requested one on the tree widget (note that selecting a tree node at zero and first level does not do anything). You can select multiple views by keeping the `Ctrl` key pressed while clicking on the corresponding leafs (this way of doing is very common on graphical user interfaces).

5.1.1 Standard views

madhoc comes with a set of standard monitors that features views which are directly usable. Before starting *madhoc*, ensure that the configuration key `monitors_class` include the `org.lucci.madhoc.broadcast.NetworkMonitor` and `org.lucci.madhoc.simulation.monitor.SimulationMonitor` classes

Aircraft views

The aircraft views comes with the network monitor. It provides an 2D representation of the simulated network, as if it is was observed from an aircraft or a satellite. Image 5.3 shows and example of an aircraft view.

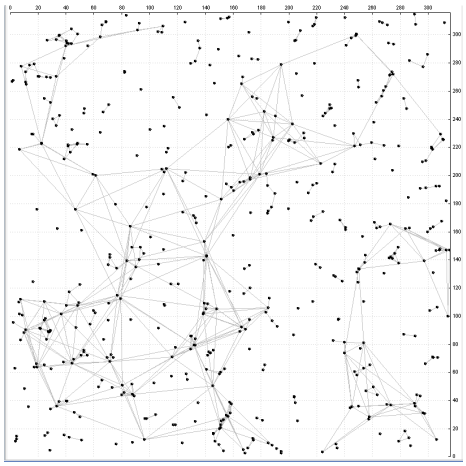

Figure 5.3: An aircraft view of a network.

Numerical measures list

The numerical measures list view comes with the simulation monitor. It provides a table whose rows represent a numerical measure (see section measures). All numerical measures in the simulator are represented here. As shown on image 5.4, different colors permit to recognize which monitor the measure come from (the colors used here are the same as the colors used on the tree widget, on the left side of the projection window).

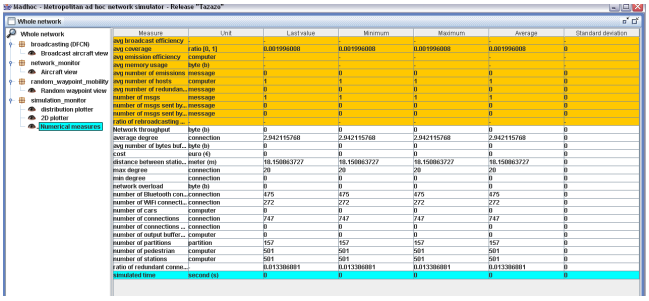

Figure 5.4: A view that give an overview of all the numerical measures available in the simulation process.

2D representation view

The 2D representation view aim to improve the functionality of the numerical measure list view. It permits the user to select two measures on the latter view

and graphical expose the relations between them, on a 2D axis system. The way the measure have to be selected is intuitie. The control panel on the right side of the 2D representation view shows two checkbox that indicate which dimension is currently being configured. By default, the X dimension is selected, as shown on image 5.5. Then click on whatever measure within the numerical measure list view. It can observed that the X axis of the 2D representation is immediately updated with the name and color of the selected measure. Now change the selected dimension using the appropriate checkbox on the 2D representation view's control pane. Then click on some other measure in the numerical measure list view. Just like what happened for the X axis, the Y one is updated. *madhoc* now shows a color 2D mathematic function which highlight the relation betweent the two selected measures, as shown in the example image 5.6. Most of the time, the user will want to use the `simulated time` measure on the X axis (image 5.6).

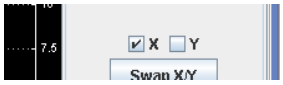

Figure 5.5: The widget that select the dimension to be set.

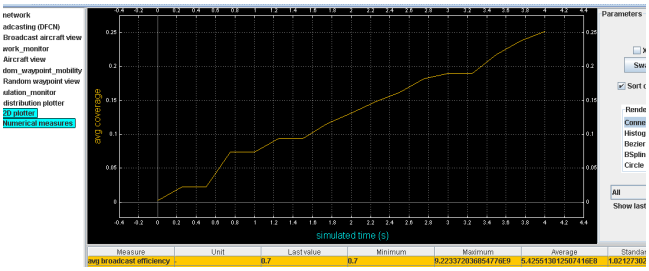

Figure 5.6: The evolution of network coverage (in the case of a broadcasting application) upon time.

Distribution view

The distribution view (an example is given on image **??**) permits the 2D reprentation of every distribution measures 2.2.2 available in the current simulation. The export function described in section 5.1.1 is also available here.

Image exportation

Often, it turns out to be very useful to export the displayed image to a disk file. For doing this, right-click on the rendered image. A popup menu will

then propose a `Save as...` menu item that will give you the ability to save the image to various common image formats. If the image is targeted to an scientific paper, vector format (PostScript, PDF or SVG) is obviously the way to go as vector-based image representation ensure perfect scaling or the produced image. People willing to re-work the rendering with a tool like GNUPlot [1] have the possibility to export the rendered data as a tab-separated file that will be easily red by GNUPlot. If the image is targeted to a web-site, bitmap format will be preferred. Note that JPG is a lossy compression technique that is better forget about in our concern, since it produce dirty graphics.

This functionality, as well as the rendering capability, is provided by the `UP` project [2].

Chapter 6

Advanced usage

6.1 Creating a new module

Section gives the initialization process of a simulated application. This helps at understanding what needs to be written by the programmer at the development stage.

What need to be hardcoded in Java are the monitor and the code of the application. These two components are enough for deploying and executing the application. It is very likely that the user will want to have some feedback on the application he is developing. Then he needs to write one or more sensors that will periodically take measures on the application and report them to the user interface, be it graphical or console-based.

Optionally, if the user wants to use the advanced features of the graphical user interface, he has the possibility to define new views for his applications. These views, which are Swing components, will be dynamically integrated in the *madhoc*'s GUI.

6.1.1 Example

The best way to know how to create a new application for *madhoc*is to look at the following example. In the following we will define a simple application that merely periodically sends "hello´´ messages to its neighbors.

The code of the application

First you need to define what your application does. This is done by deriving the class:

```
org.lucci.madhoc.network.Application
```

To do this, you need to implements the following methods:

```
String getName()
```

```
    void configure()
    void doIt(double time)
```

`String getName()` requires that you simply define and return a name for this application. The name identifies the application, so take care at not using a name that is already in use.

```
public String getName()
    {
        return "hello_message_application";
    }
```

The `void configure()` method codes how the application will handle its configuration parameters. In our present case, the configuration simply indicates to which frequency the application will emit hello messages.

```
public void configure() throws Throwable
    {
        this.helloFrequency = 1000 *
getMonitor().getNetwork().getSimulation().getConfiguration()
                .getInteger("hello_application_frequency");
    }
```

Obviously, the `helloFrequency` field must have been defined. In order this to work, the *hello_application_frequency* configuration key must be defined in *any* configuration file. Here it is advisable to create a new configuration file that will be dedicated to our new application.

The `void doIt(double time)` method is invoked at each iteration of the simulator. This is where what the application do is actually coded. An application that periodically sends its identity to its neighbors could be implemented in the following way:

```
public void doIt(double time)
    {
        if (getSimulatedTime() % this.helloFrequency == 0)
        {
            Message message = new Message();
            TransferableDouble id = new TransferableDouble();
            id.setValue(getComputer().getIdentifier());
            message.setContent(id);
            message.setSourceStationApplication(this);

getComputer().getNetworkingUnit().getOutgoingMessageQueue().add(message);
        }

        for (Message msg : getIncomingMessages())
        {
            ++incomingMessageCount;
        }
```

```
}
```

Obviously, the `incomingMessageCount` field must have been defined.

Deploying the application

The code of the application must now be attached by some means to the simulator itself. To do this, a *monitor* for the application must be defined. A monitor is a subclass of:

```
org.lucci.madhoc.simulation.Monitor
```

By deriving this class, you just need to give a name to the monitor. A name that looks like *my_application_monitor* is not a bad idea. The last thing you need to do is to provide the minimum configuration keys for the newly created application.

If you want the new application to be the only application to be executed, then simply use:

```
monitors_class={my_package.MyMonitor}
```

Otherwise, add the class name of the monitor to the list of monitor classes. This configuration entry makes the simulator aware of the new application.

The application itself requires a few configuration entries to start. Indicate the simulator which application the monitor will deploy:

```
my_application_monitor_protocol_class={my_package.MyApplication}
```

Indicate the termination condition of the application. In our case, the application never terminates.

```
my_application_monitor_termination_conditions={none}
```

Furthermore, the application does not need specific code to be initialized:

```
my_application_monitor_initializer={none}
```

We did not define any measure for the application, so filtering measure makes no sense. Just allow everything or nothing, it does not matter.

```
my_application_monitor_measures_regex={.*}
```

The application wants to know how often it must send hello message. This is given by the following configuration entry. In this case, a message is sent every 2 seconds.

```
hello_application_frequency={2}
```

6.2 Graphical and command-line user interfaces

madhoc can be used in 3 differents ways: as a console program, as a graphical application as well as a framework.

In either case, the user needs to define the configuration for the simulation.

Whether you want to use the graphical user interface or the command-line tool, you need to provide the configuration for the network you want to simulate.

Their entry point is the same, that is the `org.lucci.madhoc.Madhoc` class. Whether *madhoc* route the execution to the GUI or the console mode depends on the `simulation_interaction_mode` configuration key. You can set it either to `graphical` or `console`.

6.2.1 Using the GUI

The graphical user interface aims at providing the user with excellent visualization capabilities on the application he is developing. Organized as a desktop, each internal window represent a projection of the network. All projections feature the same tools, called views. These views are organized are classified (using the tree on the left side of the window) with regards to the monitor which made them available. One or several views can be activate at the same time by clicking on them (for activating multiple views, click on them by pressing the `Ctrl` key).

Numerical measure list

This view lists all the sensors available in the simulator. The values presented are the ones extracted by considering the given projection only. So the values for the same sensors on different projections may be different. The measures are classified according to the monitor that made them available.

6.2.2 Graphical 2D plotter

This view allows the graphical representation of any measure in relation with any other. In order to define which measure to represent as a 2D plot, select them using the numerical measure list view. Both views must hence be active in order to do that.

6.3 Using *madhoc* as a framework

By using the framework, the user (actually the programmer) needs to define the configuration by setting the values of the public fields of a `ConfigurationKeys` object.

```
ConfigurationKeys keys = ConfigurationKeys();

# show how to set a parameters
```

```
# there are plently of such parameters available to the user
keys.simulation_name = "my simulation";
```

As explained in section **??**, the configuration is a set of key/values pair. At the level of the code, it is a data structure that has nothing to do with the configuration keys available. However in the framework mode, the configuration object must be initialized with the set of keys provided by the user.

Decoupling the keys from the data structure that is actually used by the simulation may sound strange. Indeed it is strange but it has the great advantage that the keys available (actually the keys allowed) are hardly coded in Java. This way the user cannot use non-existing keys. More over, programmers who use a Java editor which provide code completion can use the latter for completing the name of the keys. The configuration data structure must then be initialized in the following way:

```
Configuration config = new Configuration();
config.load(keys);
```

Then a simulation object must be created and initialized with the configuration. The configuration contains enough data to allow a complete initialization of the simulator. In case some configuration keys are missing or have been fed with unsuitable values, explicit errors will be shows as java exceptions.

```
Simulation simulation = new Simulation();
simulation.configure(config);
```

At this step, the simulation is completely initialized and is hence ready to use.

```
# execute one single iteration of the simulator
simulation.iterate();
```

madhoc does not run until the simulation has completed. The simulation must be progressively advanced by the user. A typical way of executing a simulation is a invoke the `iterate()` method in the body of a `while` loop whose the termination condition (consequently the termination condition of the simulation) depends on the simulated application. The following code simulates one second of network activity.

```
# iterate until the simulated time is lower than 1 second
while (simulation.getSimulatedTime() < 1)
{
    simulation.iterate();
}
```

When the simulation is being running (in the body of the `while` loop) or when it has completed, after the block) it is possible to get some information on the simulation process. Indeed, the framework features an extensive list of utility methods (like `getSimulatedTime()` that can be used for retrieving informations on the simulated network and applications. In order to obtain some information that is specific to a given application, it is advisable to get the instance of the latter application, which is exposed within the simulation as a `Monitor` object.

```
# gets the monitor that is in charge of observing what's
# going on in the network
Monitor networkMonitor = simulation.getNetwork()
    .getMonitorMap(NetworkMonitor.class);
```

Sensors are designed for application-specific purposes. They are hence exposed by monitors. Obtaining a sensor can be done using the following code.

```
# retrives the sensor that is in charge of measuring the network throughput
Sensor throughputSensor = networkMonitor.getSensorMap().get(ThroughputSens
```

As explained in section 2.2.2, sensors are asked to take measure on every projections at the end of each iteration. Measures are then stored into an history that is taken care of by the projection object. The latter history can be retrieve in the following way.

```
# first gets the instance of the projection we are
# interested in (all the network)
Projection projection = simulation.getNetwork()
    .getProjectionMap(IdentityProjection.class);

# and then get the history for the measure we are interested in (the throug
MeasureHistory history = projection.getHistory(throughputSensor);
```

The set of values that have be archived in the history can be obtained by invoking the `getValues()` method on the `MeasureHistory` object. If the measure is a numerical measure (see section 2.2.2), metrics like the average or the standard deviation can then easily be obtained out of this list.

6.4 Accelerating the simulation process

If you need to increase the surface of the area you want to simulate, or increase the number of nodes in the networks, you may want the simulator the operate more efficiently. There are several ways of doing this, as described hereafter.

- Increasing the resolution will fasten up the simulator in a linear way. This is the first thing you should look at. But increasing the resolution will make the simulation less accurate, depending on what you want to simulate. Then you should be able to define which is the maximum resolution under which the behavior of the simulation will change insignificantly. However you should be extremely cautious by altering the resolution of the simulation because it may lead to unpredictable results.

- Reducing the number of projections also improve the performance of the simulation. It is important that you define only the projections you need for your simulations. Adding/discarding projections has no effect on the simulation engine.

- reducing the sensors by selecting those only which provide the metrics you are interested in. Several sensors (like the ones which counts the number of partition, which measure the degree distribution in the network, etc) require a considerable amount of time to proceed. If you are not interested in their measures, just discard them.

6.4.1 Acceleration tricks specific to broadcasting simulation

- Discarding the RAD is a trick specific to broadcasting. If the resolution of the simulation is high, the RAD becomes useless because it is a low level mechanism. In some broadcasting protocols (such as DFCN, AHBP, etc), the RAD is important because it has a serious impact on the order protocol-level information is delivered to the node. Hence discarding the RAD may change the behavior of the broadcasting protocol. Discarding the RAD in DFCN makes it loose some benefit of its cumulative neighborhood mechanism, which head to more message emissions. Depending on what you want to simulate, this harm can be simply ignored.

- Making the nodes moving faster is also a broadcasting-specific trick. It increases the number of connection/disconnection which permits some broadcasting protocols (like DFCN and AHBP_EX which use the "new connection event" as a trigger for considering a re-emission) to process significantly faster.

Chapter 7

Open issues

madhoc targets the simulation of metropolitan networks. So far, only few research projects have focused on this. Along the conception/development of *madhoc*, we have been facing several complex issues.

7.0.2 Initialization of the mobility

The initialization of mobility is difficult. How can we make sure that the initial network is valid? The structure of the network is a consequence of the mobility. But because the mobility rules are complex, the topology it will generate cannot be predicted. A solution is to run the simulator by discarding the first measures. How long this initialization process should last?

This issue is a real problem because it might prevent researchers from building relevant models for certain mobility patterns.

7.0.3 Altering the resolution of the simulation

The concept of resolution is described in section 2.3.2. *madhoc* has initially been developed considering a fixed resolution of 4 iterations per second. Lately we had to change this by supporting a user-defined resolution. But simulating a network with a resolution that is not static is extremely difficult. The most difficult point is probably the simulation of the data transfer which is tightly linked to time.

Chapter 8

Targetted applications

8.0.4 Broadcasting

madhoc were primarily designed to make easy the development and experimentation of broadcasting. protocols. Two efficient protocols were defined using *madhoc*: DFCN and CABP. DFCN is a bandwidth efficient protocols for mobile networks. CAPB is an extension of the DFCN protocol. CAPB adapts its strategy to the degree of urgency of the message that is being broadcasted. The more urgent is a message, the less greedy will CABP react.

8.0.5 Mobility models

Mobility models

A second application of the *madhoc* simulator is the development of novel mobility and radio propagation models for the simulation of metropolitan networks. Then we have developped the mall mobility model. The mall mobility model simulates the mobility of people walking in a mall center. It defines a mall as a wide area made of shops and corridors which connect them. People in the mall then move from shop to shop using the corridors. The mall mobility model uses the constraint waypoint mobility model at two different scales (at the levels of the shops and, at the bigger scale, the level of the mall).

First, a shop is a round place (whose radius is randomly chosen in a given interval) surrounded with walls, which obstruct human mobility and attenuate the radio signal. Shops are randomly located within the mall so as the distance between two given shops must not be lower than 10m. People within a given shop move according to the random waypoint mobility model, at a speed chosen in between 1 and 4 km/h. Pauses last between 1 and 20 seconds. At most 20 pauses are allowed. When a station goes out of a shop, it randomly choses its next destination shop.

Second, a corridor is a bi-directional path connecting two shops. There is always one corridor connecting two given shops. People walk in corridors at a speed randomly chosen between 2 and 6 km/h. In order to model the broadness

of the corridors, a variation of the direction of people walking into them is tolerated. The broader is the corridor, the bigger variation is tolerated.

The mall mobility model generates the emergence of dense and highly connected regions. These regions are sometimes isolated. At runtime, one can observe that they get sporadically get connected to other regions thin paths (like chains). This kind of topology illustrates that applications—as well as protocols—must deal with high variations of the density.

Pedestrian area mobility

The pedestrian area mobility model simulates the mobility of people walking in a pedestrian-only city area. This mobility model is an extension of the graph-based mobility model [28]. More precisely the street network is represented as a grid, each edge being a street section and each node being a crossroads. Similarly to the mall mobility model, in order to model the broadness of the streets, a variation of the direction of people walking into them is tolerated. People walk at a speed randomly chosen between 2 and 6 km/h. Just like in the random waypoint and mall mobility models, people make short pause whose duration is randomly chosen between 1 and 20s.

The radio propagation model is constrained by the street graph. More precisely, in order to model the signal attenuation of the walls, a given node can communicate with other nodes located on the same edge and on aligned edges, using the path loss model. The broadness of crossroads is represented by allowing all nodes close to a given crossroads to communicate with one another.

The pedestrian area mobility model is still under development.

8.0.6 Ad hoc computing: towards the ad hoc grid

Our main target is the investigation of computations distributed over mobile ad hoc networks. *madhoc* were developed with this idea in mind.

We are currently working at realistically modeling the properties of the future ad hoc grid and at predicting its behavior.

We are now focusing on building a basic grid application which consist of the distribution of on single task. It may sound simple, but actually many questions arise: According to what policy the distribution should occur? How to ensure that the result of a task will come back the client node? And so on. This research is on progress.

Chapter 9

OOP implementation in java

The *madhoc* project has begun in January 2004. Since then, 20,000 lines of code have been written. The code organized in 205 classes, classified in 42 packages. All along the development process, a great attention has been paid to the elegancy of the design and to the quality of the algorithms. The resulting code is clean and stable.

madhoc simulations deal with numerous entities of the same class: the stations. Such architecture is used many other software, like in multi-agent frameworks. Experience has shown that modeling the entities as columns in a huge array of primitive objects could improve the simulations efficiency by factor 10. However doing this merely discards the advantages of OO programming (modularity, reusability...). But we initially considered that using such a technique would have prevented us from developing *madhoc* as it is now, and would greatly compromise its "open" model.

Chapter 10

Grid

madhoc

What constitutes the main obstacle to the constitution of large grid is general network security.

Generally, the client/server defines that only one daemon (server software) of a given type runs on a given computer. Things do not work like this in *madhoc*'s grid. It defines that exactly one server (called *simulation server*) runs on every processor. Then a workstation equiped with n processors will run n simulation servers. There are no relations between simulation servers running on a same given computer: they operate in an autonomous and independant manner.

On every computer participating to *madhoc*'s grid, there is a server running. This server periodically looks for tasks to execute. The number of task it can execute simulatenously depends on the number of processor of its hosting computer. As a *madhoc* simulation is a monothread processus, a single computer can simulatenously run as many simulation as it embeds processors.

Bibliography

[1] *GNUPlot. http://www.gnuplot.info/.*

[2] *Ultimate Plotter. Up. http://amy.sunsite.dk/up/.*

[3] Kaizar Amin, Gregor von Laszewski, and Armin R. Mikler. Toward an Architecture for Ad Hoc Grids. In *12th International Conference on Advanced Computing and Communications (ADCOM 2004)*, Ahmedabad Gujarat, India, 15-18 December 2004.

[4] Christian Bettstetter. Smooth is better than sharp: a random mobility model for simulation of wireless networks. In *MSWIM '01: Proceedings of the 4th ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 19–27. ACM Press, 2001.

[5] Christian Bettstetter, Giovanni Resta, and Paolo Santi. The node distribution of the random waypoint mobility model for wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 2(3):257–269, 2003.

[6] Stephan Bohacek and Vinay Sridhara. The graph properties of manets in urban environments. In *(In Submission)*, 2004.

[7] Stephan Bohacek and Vinay Sridhara. The udel models - manet mobility and path loss in an urban. In *(In Submission)*, 2004.

[8] Dominik Buszko, Wei-Hsing (Dan) Lee, and Abdelsalam (Sumi) Helal. Decentralized ad-hoc groupware api and framework for mobile collaboration. In *GROUP'01: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 5–14. ACM Press, 2001.

[9] T. Camp, J. Boleng, and V. Davies. A Survey of Mobility Models for Ad Hoc Network Research. *Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2(5):483–502, 2002.

[10] Marco Conti, Silvia Giordano, Gaia Maselli, and Giovanni Turi. Mobileman: Mobile metropolitan ad hoc networks. In *Proceedings of the 8th International IFIP-TC6 Conference, Lecture Notes in Computer Science LNCS 2775*, pages 194–205, September 2003.

[11] Hannes Frey, Daniel Görgen, Johannes K. Lehnert, and Peter Sturm. A java-based uniform workbench for simulating and executing distributed mobile applications. *Scientific Engineering of Distributed Java Applications*, november 2003.

[12] Hannes Frey, Johannes K. Lehnert, and Peter Sturm. Ubibay: An auction system for mobile multihop ad-hoc networks. In *Workshop on Ad hoc Communications and Collaboration in Ubiquitous Computing Environments*, 2002.

[13] D. Görgen, H. Frey, and C. Hutter. Information dissemination based on the en-passant communication pattern. *KiVS: Fachtagung Kommunikation in Verteilten Systemen*, 2005.

[14] Xiaoyan Hong, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. A group mobility model for ad hoc wireless networks. In *MSWiM '99: Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 53–60. ACM Press, 1999.

[15] EPFL Information Sciences Institute. The nab (network in a box) wireless network simulator. In *http://nab.epfl.ch*, 2004.

[16] Amit Jardosh, Elizabeth M. Belding-Royer, Kevin C. Almeroth, , and Subhash Suri. Real world environment models for mobile ad hoc networks. *Journal on Special Areas in Communications - Special Issue on Wireless Ad hoc Networks*, 14(2), January 2005.

[17] Amit Jardosh, Elizabeth M. Belding-Royer, Kevin C. Almeroth, and Subhash Suri. Towards realistic mobility models for mobile ad hoc networks. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 217–229. ACM Press, 2003.

[18] Per Johansson, Tony Larsson, Nicklas Hedman, Bartosz Mielczarek, and Mikael Degermark. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 195–206. ACM Press, 1999.

[19] C. Kunze, U. Grossmann, W. Storka, and KD. Muller-Glaser. Application of ubiquitous computing in personal health monitoring systems. In *DGBMT: Jahrestagung der Deutschen Gesellschaft Für Biomedizinische Technik*, pages 360–362, 2002.

[20] Johannes K. Lehnert, Daniel Görgen, Hannes Frey, and Peter Sturm. A scalable workbench for implementing and evaluating distributed applications in mobile ad hoc networks. In *WMC'04: Western Simulation Multi-Conference*, pages 154–161, 2004.

[21] William Navidi and Tracy Camp. Stationary distributions for the random waypoint mobility model. *IEEE Transactions on Mobile Computing*, 3(1):99–108, 2004.

[22] Guangyu Pei, Mario Gerla, Xiaoyan Hong, and Ching-Chuan Chiang. A wireless hierarchical routing protocol with group mobility. In *WCNC1999; IEEE Wireless Communications and Networking Conference*, number 1, pages 1538–1542. IEEE, IEEE, September 1999.

[23] Ray and Suprio. Realistic mobility for manet simulation, December 2004.

[24] Sebastien Matas Riera, Oliver Wellnitz, and Lars Wolf. A zone-based gaming architecture for ad-hoc networks. In *NETGAMES '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 72–76. ACM Press, 2003.

[25] Hartmut Ritter, Min Tian, Thiemo Voigt, and Jochen H. Schiller. A highly flexible testbed for studies of ad-hoc network behaviour. In *LCN*, pages 746–752, 2003.

[26] E. Royer and C. Toh. A review of current routing protocols for ad-hoc mobile wireless networks. In *IEEE Personal Communications*, 1999.

[27] Christian Schindelhauer, Tamas Lukovszki, Stefan Ruhrup, and Klaus Volbert. Worst case mobility in ad hoc networks. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 230–239. ACM Press, 2003.

[28] Jing Tian, Jörg Hähner, Christian Becker, Illya Stepanov, and Kurt Rothermel. Graph-based mobility model for mobile ad hoc network simulation. In *Annual Simulation Symposium*, pages 337–344, 2002.

[29] Tuna Tugcu and Cem Ersoy. How a new realistic mobility model can affect the relative performance of a mobile networking scheme. *Wireless Communications and Mobile Computing*, 4(4):383–394, 2004.

[30] B. Williams and T. Camp. Comparison of broadcasting techniques for mobile ad hoc networks. In *MOBIHOC: Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 194–205, 2002.

[31] Jungkeun Yoon, Mingyan Liu, and B. Noble. Random waypoint considered harmful. In *INFOCOM: Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1312–1321, March 2003.