MOMOSE
A Mobility Model Simulation Environment for Mobile Wireless Ad-hoc Networks*

Stefano Boschi
Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Viale Morgagni 65
50134 Firenze, Italy
boschis@gmail.com

Pilu Crescenzi
Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Viale Morgagni 65
50134 Firenze, Italy
piluc@dsi.unifi.it

Miriam Di Ianni
Dipartimento di Matematica
Università degli Studi di Roma II
Viale della Ricerca Scientifica
00133 Roma, Italy
diianni@mat.uniroma2.it

Gianluca Rossi
Dipartimento di Matematica
Università degli Studi di Roma II
Viale della Ricerca Scientifica
00133 Roma, Italy
gianluca.rossi@uniroma2.it

Paola Vocca
Dipartimento di Matematica
Università degli Studi del Salento
Via per Arnesano
73100 Lecce, Italy
paola.vocca@unile.it

ABSTRACT

This paper describes `MOMOSE`, a highly flexible and easily extensible environment for the simulation of mobility models. `MOMOSE` not only allows a programmer to easily integrate a new mobility model into the set of models already included in its distribution, but it also allows the user to let the nodes of the MANET move in different ways by associating any mobility model to any subset of the nodes themselves. Moreover, `MOMOSE` can be easily adapted in order to record, during the simulation time, all the data necessary for the evaluation of the performance of any communication protocol or of any MANET-based application.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*environments*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*wireless communication*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol verification*

General Terms

Experimentation; Measurement; Performance; Verification

Keywords

MANET, Mobility model, NS-2, simulation environment

*This work was partially supported by the EU under the EU/IST Project 15964 *AEOLUS*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIMUTOOLS 2008, March 03-07, Marseille, France
Copyright © 2008 ICST 978-963-9799-20-2
DOI 10.4108/ICST.SIMUTOOLS2008.3036

1. INTRODUCTION

A *mobile wireless ad hoc network* (in short, *MANET*) is a computer network in which no pre-existing communication infrastructure exists, communication links are wireless, and nodes are free to move and organize themselves in an arbitrary fashion. These networks are expected to have several applications because of the minimal configuration and the quick deployment they require: natural or human-induced disasters, inter-vehicular communication, law enforcement, military conflicts, and emergency medical situations are just a few examples of application areas in which MANETs are expected to play an important role.

Since the nodes of a MANET are mobile, the network topology may change rapidly and unpredictably over time. It is then important, in order to evaluate the performance of any communication protocol or of any MANET-based application, to be able to accurately simulate the mobility traces of the nodes that will eventually utilize the protocol or the application. To this aim, we need a *mobility model* that describes the movement pattern of mobile users, and how their location, velocity and acceleration change over time.

We can distinguish two basic approaches in order to obtain a mobility model. The first approach consists of constructing the mobility model on the ground of accurate information about the mobility traces of users: however, obtaining real mobility traces is usually a great challenge. For this reason, various researchers proposed different kinds of mobility models that are not trace-driven and that are called *synthetic mobility models*. A great variety of such mobility models have been proposed in the literature, which differ according to at least one of the following criteria [11]: the *geographic constraints* that a mobile node has to deal with, the *scale* the model is designed to work for, and *the individuality* which is determined by the node aggregation level of the model. Some examples of mobility models that have been proposed in the past are[1] the random walk model [15, 18, 27, 28], the random waypoint model and its many variations [20], the random direction model and its many variations [6, 29], the boundless simulation area model [10], the Gauss-Markov model [10], the city section model [10], the exponential correlated random model [17], the column model [30], the nomadic community model [30], the pursue model [30], the reference point group model [17], and, more recently, the real-world environment model [19], the virtual track group model [36], the ripple model [12], the clustered model [22], the social model [24], and the TCP-based worm spread model [1].

In this paper we describe the `MOMOSE` tool, which is a highly flexible and easily extensible environment for the simulation of mobility models. Indeed, `MOMOSE` not only allows a programmer to easily integrate a new mobility model into the set of models already included in its distribution, but it also allows the user to let the nodes of the MANET move in different ways by associating any mobility model to any subset of the nodes themselves. Moreover, `MOMOSE` can be easily adapted in order to record, during the simulation time, all the data necessary for the evaluation of the performance of any communication protocol or of any MANET-based application.

The paper is structured as follows. In the rest of this section we briefly describe of our simulation tool. In Section 2, we introduce the software architecture of `MOMOSE`, while in Section 3 we describe the experiments that we have executed in order to evaluate the performance differences between the two simulation engines included in `MOMOSE`. In Section 4 we briefly explain how a programmer can use `MOMOSE` in order to develop a new mobility model and/or a new data recorder. Finally, in Section 5 we summarize the tools that, according to our opinion, are the most related to our application.

1.1 A brief description of MOMOSE

`MOMOSE` is a mobility model simulation tool for `MANET`s. `MOMOSE` main characteristics are its easy extendibility and its high adaptability to the information needed in order to evaluate a specific protocol. Within the `MOMOSE` framework, indeed, a programmer can easily implement new mobility models and successively simulate the movement of a set of nodes by using any combination of the mobility models included in the `MOMOSE` distribution and of the newly implemented mobility models. Moreover, the user can define appropriate *data recorders*, that is, sets of data structures and methods, in order to collect, during the simulation, the data necessary for the evaluation of a specific protocol.

During each simulation, the set of the nodes of the networks is partitioned into an arbitrary number of subsets, each one corresponding to the specific mobility model governing the movement of the nodes in the subset: however, since each node has an own logic unit which is independent from the other nodes, different nodes in the same subset are allowed to play different roles within the same mobility model (for example, in the `PursueModel` [17] it is necessary to allow one node to act as the *leader* of the subset).

`MOMOSE` also allows the user to simulate the movement of the nodes within a "realistic" environment, where obstacles (such as buildings and barriers) are present: these obstacles not only limit the movement of the nodes, but they also attenuate the transmission signals sent by the communication units. The definition of a scenario is flexible enough to allow the user to define significantly different situations, ranging from people moving within a building or a campus to robots moving within a disaster recovery environment or to vehicles moving within an urban environment (such as in a `VANET`).

During a simulation, one or more data recorders can be used, which allows the user to collect the interesting data: for example, a data recorder could compute the distribution of the nodes during the simulation time in a given area, while another data recorder could compute the degree of each node (that is, the number of active connections for each node): both data recorders could store these information into the same output file, which could be subsequently used in order to produce statistics or reports. The current distribution of `MOMOSE` includes a data recorder that produces trace files compatible with the `NS-2` network simulation environment [14], which is one of the most popular network simulator within the research community. As far as we know, the mobility modules produced for this simulator include only very simple mobility models, such as the random waypoint model [21] and the random walk model [26]: `MOMOSE` can hence be used in order to produce more realistic mobility patterns, which can be subsequently used by `NS-2` while simulating and evaluating any network protocol.

The behavior of a mobility model is typically determined by the value of some model-specific parameters: for example, in the case of the *Gauss-Markov mobility model* [31] the parameter α which determines the randomness degree of the model has to be specified, while in several other models typical parameters are the acceleration value or the angular velocity value [7]. In general, a unique set of parameters which can be used for any mobility model does not exist: for this reason, `MOMOSE` allows the user to define and to subsequently use *configuration windows* whose content depends on the mobility model. In this way, it is very easy to tune all the parameters of a specific model: moreover, this parameter tuning can be saved in appropriate files, which can be successively reloaded. A similar approach can be also followed in the case of data recorders whose behavior depends on the value of specific parameters: even in this case the user can define specific configuration windows.

The graphical user interface (in short, GUI) of `MOMOSE` is based on the `Java` *Swing* classes and on the *OpenGL* standard [33]. The simulation engine, instead, has been developed both in `Java` and in `C++`: the former one is deeply

[1]This list is certainly not exhaustive: our goal, however, is just to give a flavor of the huge quantity of mobility models that have been proposed in the literature and of how important the mobility simulation topic is within the MANET research area.

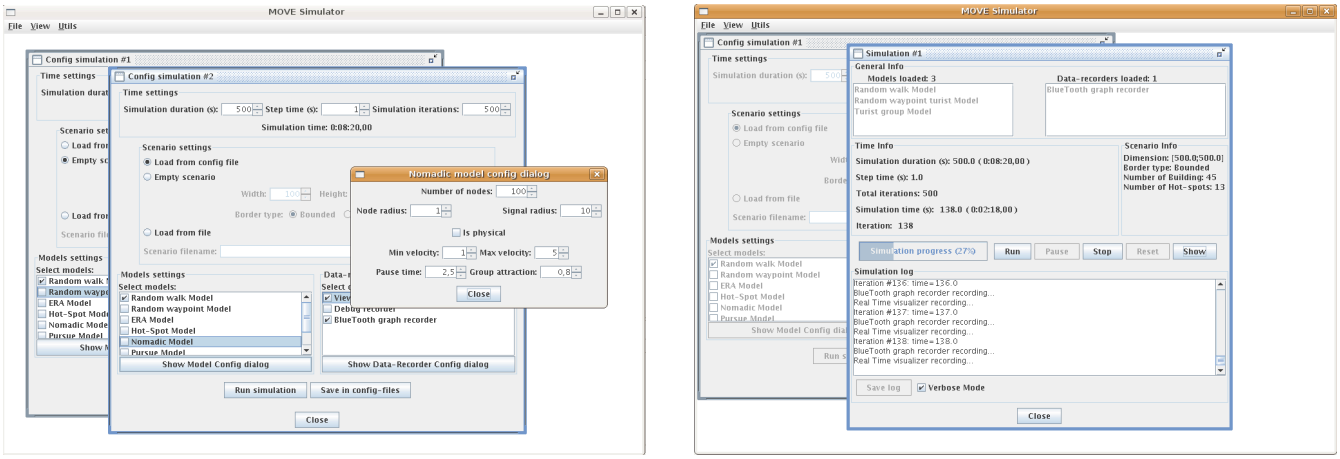

Figure 1: The configuration and the simulation windows

integrated with the GUI and allows the user to interactively control the simulation and its visualization, while the latter is optimized from a performance point of view and is accessible only by means of command lines. The reason why we decided to include two different simulation engines is strictly connected to the main characteristics of the two programming languages. `Java` is highly portable and the Swing classes behave essentially in the same way, independently from the used processor/operating system platform: on the contrary, several different graphical libraries have been developed in `C++`, and it does not seem that any of them can be considered as the standard one. Hence, the `Java` version of the simulation engine is particularly appropriate during the development phase of a mobility model and/or of a data recorder: in this case, indeed, the GUI allows the user to easily configure the simulation parameters and to observe in real-time the node movement and the evolution of the simulation itself. On the other hand, the `C++` engine, which has to be compiled for any possible platform/operating system platform, has the advantage of being significantly faster than the `Java` engine: hence, this version of the engine is more appropriate for the execution of simulations with a long simulation time and/or with a huge number of nodes.

By means of the `MOMOSE` GUI, the user can control any aspect of a simulation, both before its beginning by interacting with its configuration window and during its execution by interacting with the simulation window. The *configuration window* allows the user to set up the simulation time and the simulation scenario (which can range from an empty area to any environment specified in an appropriate file). It also allows the user to select and configure the mobility models and the data recorders to be used during the simulation: each mobility model and each data recorder can also be configured by interacting with its configuration window which allows the user to tune the mobility model parameters and to set up the data recorder parameters (see the left part of Figure 1). By means of the mobility model configuration window, it is possible to tune both parameters common to all mobility models (such as the number of nodes moving according to the model or the maximum node transmission range) and parameters which are meaningful for that specific model only (such as the attraction degree in the case of the *nomadic model* [31]). The data recorder configuration window allows the user to set up parameters such as the name of the file into which the collected data will be written. The user can define and implement new mobility model and data recorder configuration windows, which will be automatically integrated into the `MOMOSE` framework. Starting from the simulation configuration window, the user can directly start the simulation itself or can save the current configuration into an appropriate file, which can be subsequently used by one of the two engines previously described in order to execute the simulation and collect the required data (clearly, a configuration file can be reloaded and modified at any subsequent moment).

The *simulation window* (see the right part of Figure 1) allows the user to manage and control the execution of a simulation: in particular, at any moment the user can pause and restart the execution, can stop it, can see log messages produced by the simulation engine, and can activate a graphical window which shows, in real-time, the movement of all the nodes within the specified scenario (along with other informations related to the simulation). A *scenario* is defined by means of an XML file which contains the list of obstacles that are present in the simulation area. Each obstacle is formed by one or more polygons (in particular, squares, rectangles and/or circles): for each polygon, the XML code specifies its position, its rotation angle, its color, its name and its attenuation factor (which is a number between 0 and 1). A scenario can also contains a set of *hot-spots*, that is, specific points of the simulation area which are of particular interest for the nodes: by means of this feature, it is possible to define within the scenario a graph, which can be used by a mobility model while deciding the movement of a node (such as in the case of the *pathway model* [32]). The XML standard allows the user to easily define new scenarios: however, `MOMOSE` includes the possibility of generating a scenario starting from a *Scalable Vector Graphics* (in short, *SVG*) file. Hence, the user can create the scenario by using any drawing program, in order to subsequently export it in the SVG format and, hence, to translate it into the XML code required by the `MOMOSE` simulation engine.

`MOMOSE` includes an OpenGL *player* which allows the user to visualize and graphically analyze the evolution of a simu-

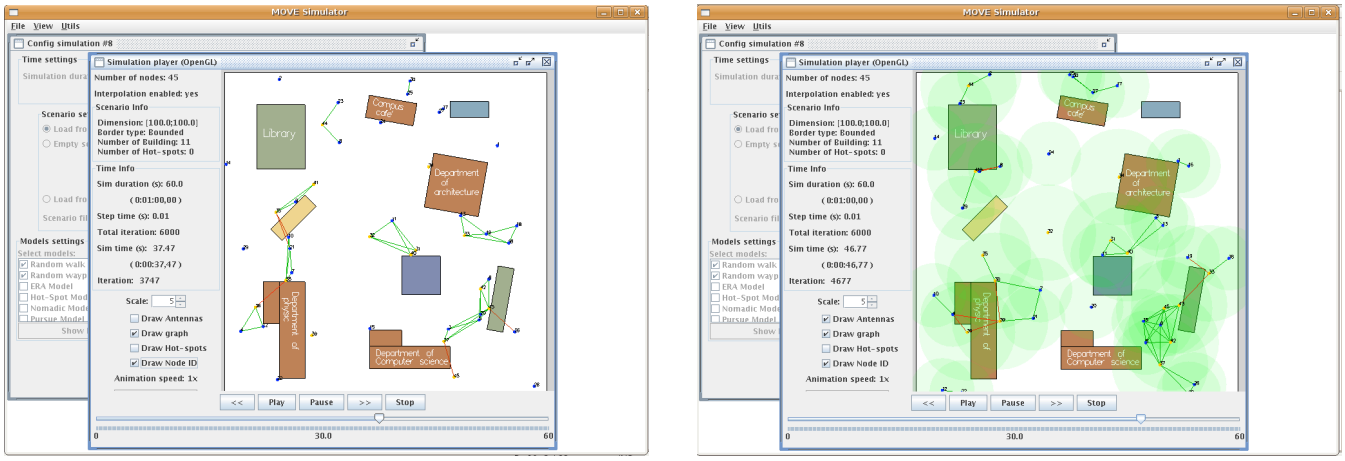

Figure 2: The OpenGL player without and with the drawing of the transmission ranges

lation, which was previously saved into appropriate files by a default data recorder included in the MOMOSE distribution. Within the main drawing area of the player, the simulation scenario and the movement of the nodes are shown (see the left part of Figure 2): on the left of this area, some basic information about the scenario and the simulation time is given and some tools for controlling the simulation itself are given (as with any other player, the user is also allowed to pause and restart the simulation and to fast advance it both forward and backward). During the simulation, some additional information can be visualized within the drawing area, such as the node IDs, the node transmission ranges, the hot-spot graph and the communication graph. In this latter graph, the color of the edges indicates the power of the transmission signal between any pair of nodes: in particular, a green color indicates that no attenuation is present, while different red gradations indicate different levels of transmission power. Since long simulations with a large number of nodes might produce huge trace files, MOMOSE allows the user to save these files in a compressed form (in particular, by using the `gzip` standard): these compressed files can be directly loaded and visualized by the player.

Finally, it is worth noting that, since the simulation configuration files, the scenario definition files and the player trace files are written by using the XML technology and they are all independent from each other, they can be immediately ported on different processor/operating system platforms, provided that an instance of MOMOSE has been installed.

2. SOFTWARE ARCHITECTURE

Apart from the GUI, the main software components of MOMOSE are the simulation engine,[2] the models, the nodes and the data recorders. Each of the latter three components is represented by means of an abstract `Java` class. The MOMOSE distribution includes several *template classes* that can be extended by the programmer in order to develop personalized mobility models and data recorders (see also the next section).

The simulation engine contains several components managing the following different aspects of a simulation.

- The *mobility model manager* is in charge of the nodes and of the mobility models during the simulation.
- The *physical engine manager* computes the collisions between the nodes and the obstacles which are present into the scenario and, hence, moves the nodes within the simulation area.
- The *scenario manager* is in charge of the logical representation of the simulated environment and of all the objects which are contained in the environment itself.
- The *data recorder manager* allows the data recorders to store the data collected during the simulation.
- The *time manager* is in charge of the advance of the simulation clock.

A simulation execution is divided into three phases: the *simulation setup* phase, the *simulation cycle* phase, and the *simulation end* phase (see the left part of Figure 3).

During the setup phase all the data structures necessary for the simulation execution are initiliazed: the behavior of this phase is determined by the information read from the simulation configuration file, produced by means of the GUI or directly written by the user. The different components of the simulation engine are initialized one after the other starting from the time manager data structures and, subsequently, the scenario and all the objects that are contained within it. Successively, the mobility models and the node generation are initialized: during this step, each model can setup its own data structures and create the nodes that will move into the simulation area. The nodes have some common properties (such as the ID, the transmission range, and the velocity vector); moreover, each node can be defined as a physical object, so that it can collide with other nodes. During their generation, the nodes are also assigned an initial position: to this aim, the mobility model can analyze the scenario, if necessary (for example, in order to

[2]From a functional point of view, the `Java` and the `C++` simulation engines are equivalent: all the classes that represent the different simulator components have the same interface and do the same task. In this way, the programmer can switch from one engine to the other without having to modify a single line of the code.

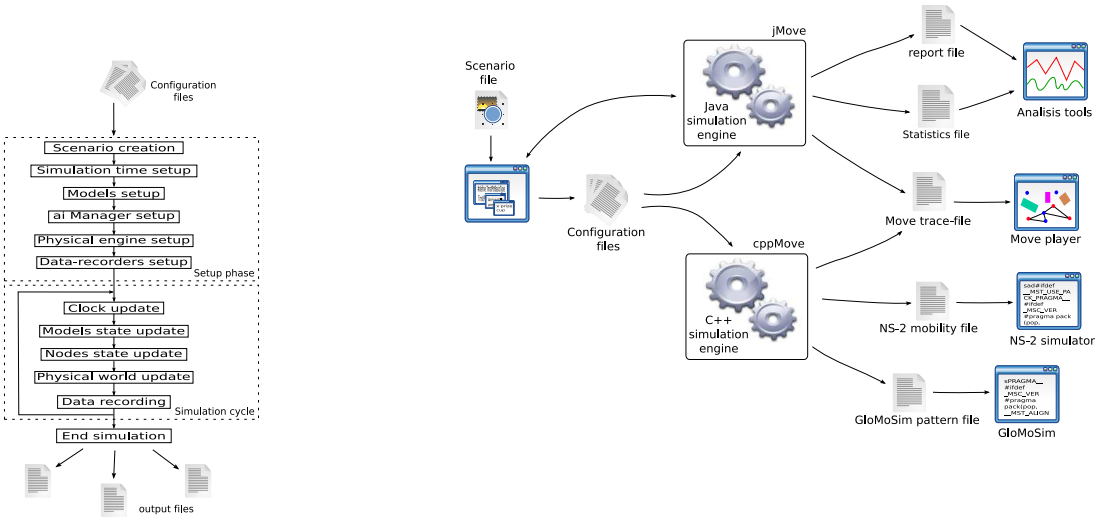

Figure 3: The simulation and MOMOSE flow diagrams

position a node onto an hot-spot or to avoid to position a node onto a wall). At the end of this step, all the information concerning the models and the nodes are passed to the initialization step of the mobility model manager and of the physical engine manager. The last step of the setup phase consists of the initialization of the data recorders and of their manager: similarly to the mobility model initialization, each data recorder setup its own data structures (such as the output file or counter variables).

Once the setup phase is done, the simulation cycle starts. At each cycle, the time manager updates the internal clock of the simulator and checks for the ending conditions. If the simulation is not ended up, the mobility model manager makes each node and model choose the next operation to be performed. Both models and nodes may perform any required operation: for instance, a model may generate a new target for its nodes, may change the role of some or of all its nodes and may interact with other models, while a node may switch on or off its own transmission device, may change its own role, may change its speed and direction, and may modify its transmission range (notice that, by modifying the transmission range, energy saving arguments can be also taken into account). While such operations are performed, the simulator keeps models and nodes informed about the simulation time, the scenario and the states of the other nodes in order to let them take the correct decisions. For instance, a node may need information about the scenario in order to determine whether collisions may occur while moving at a given speed and direction, or it may need information about the hot-spot list in order to choose its next target destination. After the mobility model manager step, the physical engine manager gets the simulation control and computes the new node positions, taking into consideration the collisions between nodes and scenario objects and among nodes. In order to speed up these operations, the physical engine manager represents the simulation area by a *BSP tree*:[3] such a tree is built during the setup phase and it allows the physical engine manager to save computational time, since only the collisions between a node and its surrounding physical objects are considered. At the end of any simulation cycle, each data recorder collects the required data and records system informations at current time. Data recorders may access all simulation data (such as time, scenario, and system state) and may access all the information about models and nodes involved in the simulation: for instance, one data recorder might compute the communication graph and draw its diameter, the node degrees, and the number of connected components, while another data recorder might write the logs of node positions and states at the current simulation cycle.

The last phase of a simulation is the simulation end, during which a procedure is invoked for any data recorder that allows it to execute some final tasks. For example it is possible to close the open files, to evaluate some performance values by using the collected data, or to create reports and the similar. During this phase, the simulation engine erases all temporary data structures. Finally, the simulation ends and the output files created by the data recorders can be used for the analysis or can be exploited by other tools (see the right part of Figure 3).

3. PERFORMANCE COMPARISON

In this section a performance comparison is presented between the `Java` and the `C++` simulation engines. The analysis is performed by comparing the running times of the two engines when executing on the same simulation framework (that is, the same simulation time and the same number of nodes). In order to evaluate the real performances, all additional I/O components have been removed. The simulations have been executed on a `Intel Pentium 4 2.4 GHz` processor, with `512 MB` of RAM running a `Linux` (kernel version `2.6.17-10`) operating system.

The first comparison focuses on the number n of nodes. Ten different values of n have been considered: for each of them, twenty simulations have been executed and the average execution time has been computed. In particular, for each execution the simulation time has been set equal

[3]The Binary Space Partitioning [16] technique is a recursive partitioning of the space into convex sub-spaces, which is described by means of a binary tree (called BSP tree).

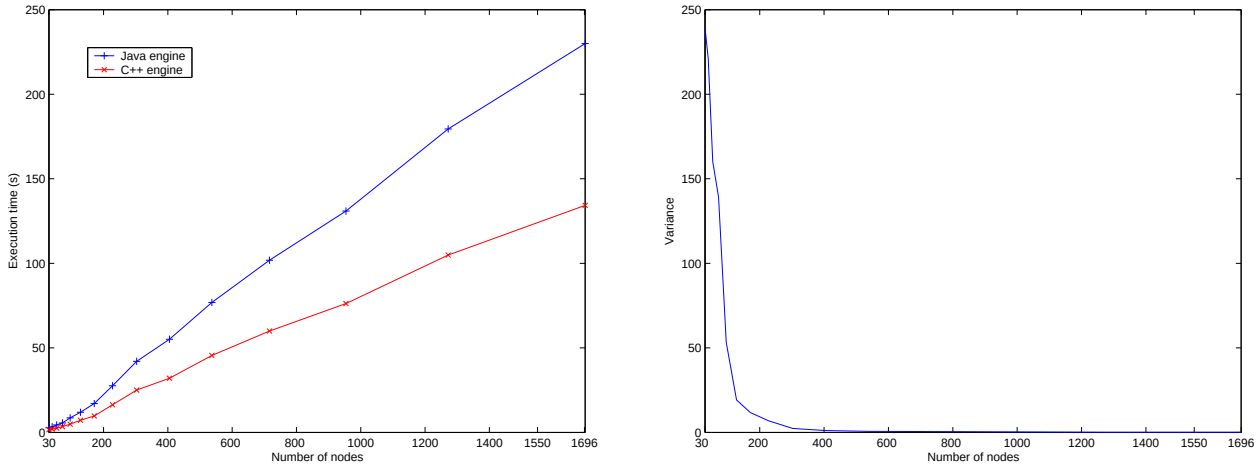

Figure 4: A comparison of the Java and the C++ engines with respect to the number of nodes: on the left the average execution time, on the right the variance value

to 10800 seconds, while the n nodes have been partitioned into three equally sized sets moving accordingly to the random waypoint, the random walk, and the nomadic mobility model, respectively. In the left part of Figure 4, the average execution time is shown, while in the right part of the figure the corresponding variance values are drawn. It is evident that the C++ engine is significantly more performing than the Java engine: it also seems that this better performance does not depend on the number of nodes.

However, one might think that the execution time is too short and that the initial overhead, that a Java program has to usually pay for,[4] cannot be compensated in such short execution times. For this reason, we have performed a second kind of comparison which focuses on the simulation time t. Ten different values of t have been considered: for each of them, twenty simulations have been executed and the average execution time has been computed. In particular, for each execution 900 nodes have been simulated, partitioned in a way similar to the previously described one. In the left part of Figure 5, the average execution time is shown (in a logarithmic scale), while in the right part of the figure the corresponding variance values are drawn.

Even in this case, the better performance of the C++ engine is quite evident. Even though the performance relative difference slightly decreases while the simulation time increases, it seems that asymptotically this difference tends to a value close to 40%.

4. EXTENDING MOMOSE

As already stated in Section 1, MOMOSE allows the programmer to create new mobility models and new data recorders starting from a set of template classes. In the next two sections, we show how this can be done by also describing an example of a new mobility model and of a new data recorder.

4.1 Creating a Mobility Model

In order to develop a new mobility model, the programmer has to implement three classes: the ModelBuilder class, which manages the creation of the model, the Model class, which represents the model itself, and the Node class, which represents a node moving according to the model. Optionally, the programmer can implement two additional classes, that is, the ModelConfigDlg class, which represents the model configuration window, and the ModelParser class, whose task is reading all the necessary information starting from a configuration file: these two classes should allow the user to easily manage the model parameters (MOMOSE furnishes, however, a default version of these two classes that allows the user to set up some parameters common to all mobility models).

The ModelBuilder class collects the model configuration data either from the simulation configuration window or from a configuration file (by making use of the parser): by using these data, the builder creates and set up the object that actually represents the model and that will be passed to the simulation engine. The programmer can personalize the ModelBuilder class by rewriting three methods: createFromDlg, which is needed for creating a model starting from the data taken from a configuration window, createFromFile, which is needed for creating a model starting from the data taken from a configuration file, and toConfigFile, which is used in order to save the configuration data onto a configuration file: The first two methods return an instance of the Model class, which is then passed to the simulation engine.

The Model class is characterized by two attributes: the node array, which contains the nodes moving according to the mobility model, and the Boolean flag isThinker, which specifies whether the model is a "thinking" entity or all the reasoning is directly performed by the nodes. The programmer has to implement two methods. The setup method is invoked by the simulation engine during the model setup step and receives as parameters an object representing the simulation clock and a scenario: in this way, the model can setup all the necessary data structures required by the simulation. Within the body of the setup method, the nodes have to be created and inserted into the node array. The think method, instead, receives as parameters only an object representing the simulation clock and is invoked by the

[4]For instance, the Java interpreter usually performs a code validity check, which is done only the first time a method is invoked.

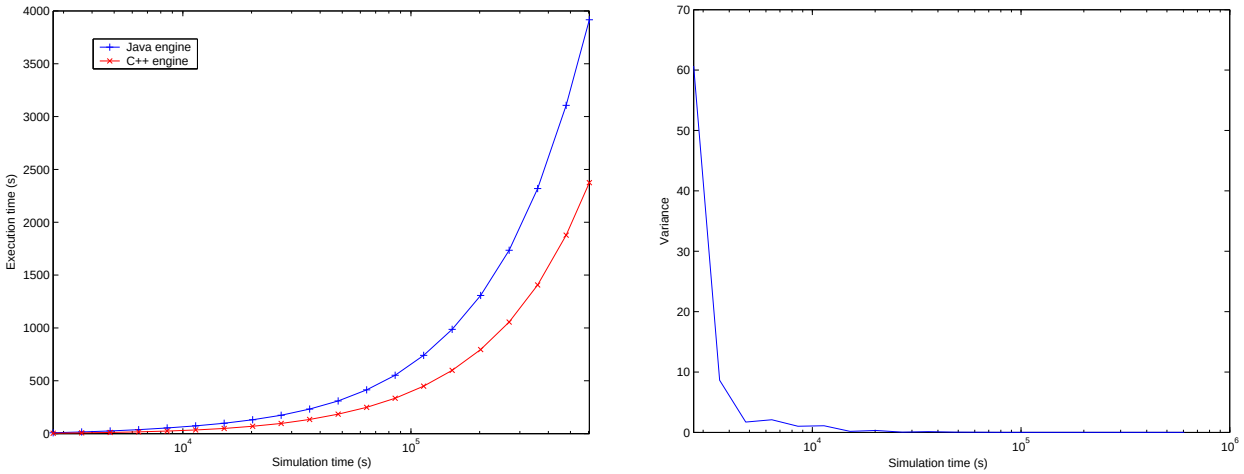

Figure 5: A comparison of the `Java` and the `C++` engines with respect to the simulation time: on the left the average execution time (logarithmic scale), on the right the variance value

mobility model manager if the `isThinker` flag is `true`: in this way the model can reason about the operations that have to be performed at the current simulation time (clearly, if no operation has to be done by the model, then the programmer does not need to rewrite this method).

The `Node` class has several attributes such as the velocity vector, the transmission range and the node ID: moreover, it has the two Boolean flags `isPhysical`, which specifies whether the node has to be considered a physical object, and `collided`, which specifies whether at the previous simulation cycle the node collided with a scenario object or with another node. The personalization of this class simply consists of rewriting the `think` method, which is very similar to the homonymous method of the `Model` class.

In order to integrate the new mobility model into the `MOMOSE` framework, it suffices to position the above described classes in a common package (in the case of the `Java` engine) or in a subdirectory of the main path (in the case of the `C++` engine), to add a reference to the new model builder class within the `SimulationManager` class and to recompile the source code.

4.2 Creating a Data Recorder

The structure of the classes that implement a data recorder is quite similar to the one of the classes that implement a mobility model. In particular, the programmer has to define two classes: the `DataRecorderBuilder` class, which manages the creation of the data recorder, and the `DataRecorder` class, which actually implements the data recorder itself. Optionally, the programmer can implement a configuration window class, extending the `DataRecorderConfigDlg` class, and a `DataRecorderParser` class, which collect the data recorder set up information starting from a configuration file (even in this case, `MOMOSE` furnishes a default implementation of these two optional classes).

The `DataRecorderBuilder` class is very similar to the `ModelBuilder` class: even in this case, the programmer has to rewrite the three methods `createFromDlg`, `createFromFile`, and `toConfigFile`.

The `DataRecorder` class is used to define the data recorders that are used by the simulation engine and that record all the necessary information concerning the evolution of the system. In particular, the programmer has to rewrite the three methods `setup`, `record`, and `endSimulation`. The first method is invoked during the simulation setup phase and initializes the recorder data structures (such as the output file): it receives as parameters an object representing the simulation clock, a scenario, and an array containing the list of models used by the simulation engine: by means of this array, the recorder can access both the models and the nodes. The `record` method is invoked by the data recorder manager during the simulation cycle, in order to record the information concerning the state of the system, the nodes and the models at the current simulation time: for instance, this method could append into the output file the position and the transmission range of every node at the current simulation time. Finally, the `endSimulation` method is invoked during the end simulation phase: as we have already said, by means of this method the recorder can close the output file, can eliminate all the data structures or can create statistical reports.

Similarly to the creation of a new mobility model, in order to integrate the new data recorder into the `MOMOSE` framework, it suffices to add a reference to the new data recorder builder class within the `SimulationManager` class and to recompile the source code.

5. RELATED WORK

The two most popular network simulators which are used within the wireless ad hoc network research community are the Network Simulator-2 (in short, NS-2) [40] and the GloMoSim environment [37].

NS-2 is a discrete event simulator that supports the simulation of TCP, routing, and multicast protocols over wired and wireless networks. Indeed, NS began as a simulator of wired networks, but due to the extension developed within the Rice university *Monarch project* [39], it now allows the user to simulate wireless networks (in particular, ad hoc wireless networks). Within NS-2, a simulation is defined by means of the `C++` and the `oTcl` languages. The `C++` classes are used in order to develop the protocols that have to be simulated, while `oTcl` is used as an interface between the

user and the C++ classes: in particular, the programmer creates a text file that describes the network architecture and lists the events that must occur during the simulation.

GloMoSim is a simulation library for wired and wireless networks, which has been developed at the *Parallel computing laboratory* of the University of California at Los Angeles. GloMoSim uses *Parsec* [2], which is a simulation language for sequential and parallel execution of discrete-event simulation models. Within GloMoSim a simulation is described by means of a text file which is passed to the simulator: by means of this file, it is possible to specify the network architecture, the total simulation time, the protocols that have to be executed by the nodes and the simulation events. The Glomosim distribution includes the implementation of several wireless protocols (such as *802.11*, *MACA*, and *CSMA*) and of some routing protocols for ad hoc networks (such as *DSR*, *Fisheye*, and *AODV*).

Both NS-2 and GloMoSim include tools for the generation of node mobility patterns. In particular, within NS-2 the information concerning the node movements are collected into a file, called *ns2-mobility-file*, which is then given as input to the simulator. These files are generated by a tool, called *IMPORTANT* [3], which currently supports the following mobility models: random waypoint, reference point group mobility, freeway mobility and Manhattan mobility. The tool for the generation of mobility pattern within GloMoSim is called *MobiGen* [38] and currently uses the random waypoint model only.

In our opinion, both generation tools were not designed to be extended, so that it does not seem to be very easy to add new mobility models to them. Moreover, only one of the available mobility models can be used during each simulation. Finally, the currently included models can simulate only the movement within an empty area, that is, without any obstacle. On the contrary, it is more and more important to simulate the behavior of protocol within a realistic scenario and to allow the nodes to move themselves according to different mobility patterns: it is indeed known that using simplistic environments and mobility patterns can lead the simulation to produce wrong performance results [35].

6. CONCLUSION

In this paper we have described MOMOSE, a new environment for the development and simulation of mobility models for mobile wireless ad-hoc networks, whose main characteristics are flexibility and extensibility. MOMOSE has already been applied in three interesting and non trivial case studies, thus showing how it can be used while analyzing different aspects of MANETs.

In the first case study we have replicated the experiments described in [5] concerning a localization algorithm based on the estimate of the received signal power: the localization problem is one of the most important research topic within the field of sensor networks, and it is strictly related to routing protocols and energy consumption. In order to replicate these experiments, we used the random waypoint model (which was already included in MOMOSE) and we designed a new parametric data recorder, which computes, during the simulation, the real position of a sensor, the position computed by the algorithm proposed in [5], and the error between these two values. Our experimental results strongly agree with the ones presented in [5], thus validating the correctness of our simulation tool and proving the easiness of using it in order to design and realize a new set of experiments.

The second case study concerned the development of a new mobility model. Considering that different nodes may move according to different mobility models and that the mobility behavior of a node may vary during time because of changes of its environment, we let the nodes of a network move according to mobility models that are determined by the roles played by the nodes themselves: these roles, in turn, can be determined by computing colorings of the graph induced by the communication network [9]. Observe that prior applications of social network analysis to the development of MANET mobility models assume that the structure of the social network is known *a priori* and that this structure does not change over time [24, 25]: in the role assignment based approach, instead, the social network structure is determined by the topology of the MANET, which in turn changes over time due to the movement of the nodes. Our experiments (that will be reported in a forthcoming paper) show that the combination of a role assignment algorithm (called ecological [8]) with a simple mobility model (that is, the random waypoint model) produce movement patterns with significantly higher mobility metrics [34] than the original mobility model itself. Moreover, these experiments allowed us to confirm the easiness of designing and developing new mobility models within the MOMOSE framework.

The goal of the third and last case study was to evaluate the connectivity performance of part of the *Blue pleiades* protocol proposed in [13] in order to more efficiently perform the device discovery phase of the Bluetooth standard [23]. In particular, during this phase each Bluetooth device attempts at discovering other devices contained within its visibility range and at establishing reliable communication channels with them: however, since requiring each device to discover *all* of its neighbors is too time consuming [4], the Blue pleiades protocol forces each device to stop the device discovery phase as soon as a constant number of neighbors has been detected (typically, 7). We have then implemented this protocol and we have evaluated its connectivity performance when used for a MANET formed by thousands of devices moving around the historic centre of Florence, Italy (see Figure 6). According to our experiments, the Blue pleiades protocol works quite well if the number of selected neighbors is at least 5: indeed, in this case the number of connected components do not increase too much with respect to the case in which all the neighbors are selected.

In conclusion, the above described experiments seem to confirm the flexibility and extensibility of MOMOSE: for this reason, we believe that our framework can become a very useful tool for evaluating the effects of mobility on the performance of a protocol for MANETs and we hope that a wide use of the tool itself will allow us to further improve it.

7. REFERENCES

[1] M. Abdelhafez, G. F. Riley, R. G. Cole, N. Phamdo. *Modeling and Simulations of TCP MANET Worms*, Proc. 21st International Workshop on Principles of Advanced and Distributed Simulation, 123–130, 2007.

[2] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song. *PARSEC: A Parallel Simulation Environment for Complex Systems*, IEEE Computer, 77–85, October 1998.

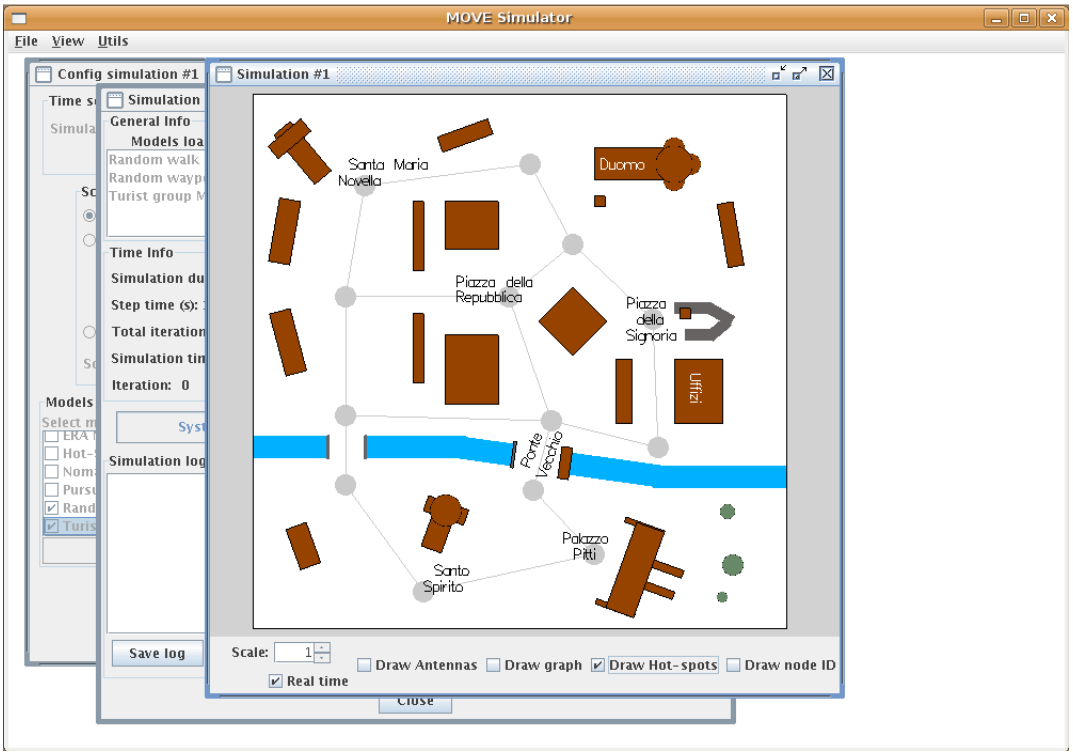

Figure 6: The Florence's historic centre scenario used while evaluating the Blue pleiades protocol

[3] F. Bai, N. Sadagopan, and A. Helmy. *User Manual for IMPORTANT Mobility Tool Generators in ns-2 Simulator*, University of Southern California, February 2004.

[4] S. Basagni, R. Bruno, G. Mambrini, and C. Petrioli. *Comparative performance evaluation of scatternet formation protocols for networks of Bluetooth devices*, Wireless Networks, 10, 197–213, 2004.

[5] P. Bergamo and G. Mazzini. *Localization in sensor networks with fading and mobility*, Proc. 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, 750–754, 2002.

[6] C. Bettstetter. *Mobility modeling in wireless networks: Categorization, smooth movement, border effects*, ACM Mobile Computing and Communication Review, 3, 55–67, 2001.

[7] C. Bettstetter. *Smooth is Better than Sharp: A Random Mobility Model for Simulation of Wireless Networks*, Proc. ACM Intern. Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems, 19–27, 2001.

[8] S. P. Borgatti and M. G. Everett. *Ecological and perfect colorings*, Social Networks, 16, 43–55, 1994.

[9] U. Brandes and T. Erlebach, Eds. *Network Analysis: Methodological Foundations*, Lecture Notes in Computer Science, Vol. 3418, 2005.

[10] T. Camp, J. Boleng, and V. Davies. *A Survey of Mobility Models for Ad Hoc Network Research*, Wireless Communication and Mobile Computing, 2, 483–502, 2002.

[11] J. Capka and R. R. Boutaba. *A mobility management tool-the realistic mobility model*, Proc. IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, 242–246, 2005.

[12] C.-H. Chen, H.-T. Wu, and K.-W. Ke. *Flexible mobility models towards uniform nodal spatial distribution and adjustable average speed*, Proc. IEEE Vehicular Technology Conference, 2292–2296, 2005.

[13] D. Dubhashi, O. Häggström, G. Mambrini, A. Panconesi, and C. Petrioli. *Blue pleiades, a new solution for device discovery and scatternet formation in multi-hop Bluetooth networks*, Wireless Networks, 13, 107–125, 2007.

[14] K. Fall and K. Varadhan. *The ns Manual*, University of California at Berkeley, 2007.

[15] W. Feller. *An Introduction to Probability Theory and its Applications, Volume 1*, Wiley, 1968.

[16] H. Fuchs, Z.M Kedem and B.F. Naylor. *On Visible Surface Generation by A Priori Tree Structures*, Proc. 7th annual conference on Computer graphics and interactive techniques, 124–133, 1980.

[17] X. Hong, M. Gerla, G. Pei, and C. Chiang. *A group mobility model for ad hoc wireless networks*, Proc. ACM Intern. Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems, 53–60, 1999.

[18] B. D. Hughes. *Random walks and random environments*, Oxford University Press, 1995.

[19] A. Jardosh, E. M. Belding-Royer, K. C. Almeroth, and S. Suri. *Real world environment models for mobile ad hoc networks*, IEEE Journal on Special Areas in Communications, 23, 2005.

[20] D. B. Johnson and D. A. Maltz. *Dynamic source*

routing in ad hoc wireless networks, in Mobile Computing, Kluwer Academic Publishers, 1996.

[21] E. Hyytiä, H. Koskinen, P. Lassila, A. Penttinen, J. Roszik, and J. Virtamo. *Random Waypoint Model in Wireless Networks*, Networks and Algorithms: complexity in Physics and Computer Science, Helsinki, June 2005.

[22] S. Lim, C. Yu, and C. R. Das. *Clustered mobility model for scale-free wireless networks*, Proc. IEEE Conference on Local Computer Networks, 231–238, 2006.

[23] G. Manthos and P. Johansson. *Bluetooth: an enabler of personal area networking*, IEEE Network, 15, 28–37, 2001.

[24] M. Musolesi, S. Hailes, and C. Mascolo. *An ad hoc mobility model founded on social network theory*, Proc. 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems, 20–24, 2004.

[25] M. Musolesi and C. Mascolo. *Designing Mobility Models based on Social Network Theory*, ACM SIGMOBILE Mobile Computing and Communications Review, to appear.

[26] G. Noubir and R. Rajaraman. *Mobility Models for Ad Hoc Network Simulation*, Proc. of IEEE Information Communications Conference, 454–463, 2004.

[27] S. Pólya. *Über eine Aufgabe der Wahrscheinlichkeitstheorie betreffend die Irrfahrt im Strassennetz*, Mathematische Annalen, 84, 149–160, 1921.

[28] P. Révész. *Random walk in random and non-random environments*, World Scientific, 1990.

[29] E. Royer, P. Melliar-Smith, and L. Moser. *An analysis of the optimum node density for ad hoc mobile networks*, Proc. IEEE International Conference on Communications, 857–861, 2001.

[30] M. Sanchez and P. Manzoni. *A java-based ad hoc networks simulator*, SCS Western Multiconference Web-based Simulation Track, San Francisco, January 1999.

[31] D. Shukla. *Mobility models in ad-hoc networks*, KReSIT-IIT Bombay, November 2001.

[32] J. Tian, J. Hahner, C. Becker, I. Stepanov, and K. Rothermel. *Graph-based Mobility Model for Mobile Ad Hoc Network Simulation*, Proc. of 35th Annual Simulation Symposium, 337, 2002.

[33] M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide: The official guide to learning OpenGL*, Addison Wesley, 2003.

[34] S. Xu, K. L. Blackmore, and H. M. Jones. *An analysis framework for mobility metrics in mobile ad hoc networks*, EURASIP Journal on Wireless Communications and Networking, 2007.

[35] J. Yoon, M. Liu, and B. Noble. *Random Waypoint Considered Harmful*, Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, 1312–1321, 2003.

[36] B. Zhou, K. Xu, and M. Gerla. *Group and swarm mobility models for ad hoc network scenarios using virtual tracks*, Proc. Military Communications Conference, 289–294, 2004.

[37] Global Mobile Information Systems Simulation Library, `http://pcl.cs.ucla.edu/projects/glomosim/`.

[38] MobiGen, `http://www.soe.ucsc.edu/~mmosko/mobigen/`.

[39] The Rice University Monarch Project, `http://www.monarch.cs.rice.edu/`.

[40] The network simulator NS-2, `http://www.isi.edu/nsnam/ns/`.