CHAPTER

Simulating wireless and mobile systems 16

The Integration of DEUS and Ns-3

M. Amoretti, M. Picone, F. Zanichelli, and G. Ferrari
Università degli Studi di Parma, Parma, Italy

1 INTRODUCTION

Mobile and distributed systems are the result of the interconnection of several nodes, characterized by decentralized goals and control, that as a whole exhibit one or more properties (i.e., behavior) which are not easily inferred from the properties of the individual parts. Such systems are complex, because the interactions of the nodes determine their future individual states and that of the system [1]. Moreover, they usually exhibit high levels of concurrency and asynchrony and their performance may be highly influenced by the changing environmental conditions of the environment (e.g., if they move).

For the qualitative and quantitative analysis of such systems, discrete event modeling and simulation (in which time jumps from event to event) are usually adopted [2]. In order to choose the proper simulation environment, the following criteria are taken into account: simulation architecture (the operation and the design of the simulator), usability (how easy the simulator is to learn and use), extensibility (the possibility to modify the standard behavior of the simulator in order to support specific protocols), configurability (how easily the simulator can be configured and with what level of detail), scalability (the ability to simulate how a decentralized protocol scales with thousands, or more, nodes), statistics (how meaningful and easy to manipulate the results are), reusability (the possibility to use the simulation code to write the real application). Moreover, the design of mobile ubiquitous applications can be achieved efficiently only by taking into account multiple aspects: networking, user behavior, environment dynamics. Depending on the problem to be studied, omitting some of these points of view may lead to less-than-useful simulation results.

By looking at the state of the art, it is evident that almost every simulation tool targets a specific problem class. Only a few of them are truly general-purpose. Among these, in our opinion, the most advanced is CD++ [3], a modeling environment that enables the definition and execution of Discrete

Event System Specification (DEVS) models [2]. OMNeT++ is another well-known general-purpose discrete event simulation tool, which has been publicly available since 1997 [4]. Like CD++, OMNeT++ is based on the concept of simple and compound modules. The user defines the structure of the model (the modules and their interconnection) using a topology description language called NED. OMNeT++ has been used in numerous domains from queueing network simulations to wireless and ad-hoc network simulations, from business process simulation to peer-to-peer network, optical switch and storage area network simulations.

Unfortunately, all of these simulation tools are not particularly suitable for the analysis of distributed systems with thousands of nodes, characterized by a high level of churn (node joins and departures) and reconfiguration of connections among nodes. Trying to fill this gap, in 2009 we started a project for the development of an open source, Java-based, general-purpose discrete event simulation tool, called DEUS [5]. To simulate a distributed system at the application level, DEUS is particularly convenient, because of its extreme ease of use and flexibility. However, it does not provide packages for simulating networking layers, and we do not foresee implementing them. For this reason, until this point the scheduling of application-level events to simulate the exchange of messages among nodes has been necessarily configured by the user, using reasonable values—which could be considered a naive approach.

In this chapter, we present a general co-simulation methodology to obtain realistic DEUS-based simulations of mobile and distributed systems, leveraging on a highly reliable and complete open source tool for the discrete event simulation of Internet systems, namely ns-3 [6]. Such a tool relies on high-quality contributions of the community to develop new models, debug or maintain existing ones, and share results. As a proof of concept, we describe our positive experience in integrating ns-3's LENA LTE-EPC package (see Section 4) to support the network-aware simulation of a peer-to-peer overlay scheme called Distributed Geographic Table (DGT), which allows mobile nodes to efficiently share geo-referenced information without centralized control. To the best of our knowledge, OVNIS [7] is the only other tool which integrates ns-3 with a higher level discrete event platform, namely the SUMO road traffic simulator [8]. However, the only available release of OVNIS is the initial one, which includes an outdated version of ns-3.

The chapter is organized as follows. Section 2 analyzes related work on wireless and mobile systems co-simulation. Section 3 recalls the main features of DEUS. Section 4 is devoted to ns-3. Section 5 illustrates the methodology we propose to use ns-3 to improve the realism of DEUS-based simulations. Section 6 describes a challenging case study (regarding mobile nodes that form a peer-to-peer overlay network operating on top of LTE), and compares the results obtained with the proposed methodology with those obtained with a naive approach that models only the application layer. Finally, Section 7 concludes the chapter with a discussion of open problems and future work.

2 CO-SIMULATION OF WIRELESS AND MOBILE SYSTEMS

Although an intuitive way to simulate complex systems is to engineer a new tool from scratch that would contain building modules for communication components and others for physical components (such as mobile devices and vehicles), a good practice and a basic principle in engineering is to avoid reinventing the wheel and to rely on well-developed ideas as much as possible. Thus, adapting and integrating existing simulation tools provides a practical and convenient approach. In particular, co-simulation (cooperative simulation) is a methodology that allows individual components to be simulated by different simulation tools running sequentially or simultaneously, and exchanging information in a collaborative manner. In general, the type of information exchanged during co-simulation may be boundary conditions such as pressure, flow rate and temperature, or simulation parameters such as time steps or control signals. In the context of wireless and mobile systems, co-simulation is implemented by integrating a network simulator, producing information like accurate packet delays and transmission ranges, with other simulation tools, either specialized or general-purpose.

A brief description of co-simulation tools for network control systems (NCSs)—e.g., MATLAB®, Jitterbug, TrueTime—has been proposed by Årzén and Cervin [9]. For wireless network control systems (WNCS) simulations, a network simulator has been implemented as C MEX S-functions, to execute simultaneously with the SIMULINK control system [10]. Co-simulation of control and network based on MATLAB/SIMULINK has been proposed in several research works [11–14] that investigated NCS performance for various data rates, traffic, loads, network delays, networked predictive control, and compensation of transmission delay. However, the MATLAB/SIMULINK environment does not provide sufficient support for simulation of real-time implementation issues. MATLAB is also limited in simulating important aspects of wireless networks, such as node movement models and wireless signal propagation models. Jitterbug and TrueTime have been used to investigate the effects on system performance of the sampling period, communication delay, jitter, control-task scheduling, and blocking of real time tasks [15,16]. However, TrueTime does not support wireless networks and uses simplified network models. Moreover, it is not possible to use Jitterbug to evaluate the performance of a feedback scheduling system where the CPU loads change, and where the sampling periods of the controllers are changing over time. Another limitation of Jitterbug is that only linear systems can be analyzed [15].

Other research works [17–20] combined two simulation packages to achieve a more efficient co-simulation approach. A co-simulation platform that combines the ns-2 network simulator with the Modelica framework has been presented by Al-Hammouri et al. [17], where ns-2 models the communication network and Modelica simulates sensors and actuators. SIMULINK and OPNET co-simulation for WNCS over MANET has been considered by Hasan et al. [18], to investigate the situation where the controller communicates with the simulated stationary

MANET and plant nodes, over a real wireless link. In a more recent work [19], Hasan et al. have presented a SIMULINK-OPNET co-simulation methodology, with comprehensive simulation results, also considering the impact of different network sizes with stationary and mobile nodes. Leclerc et al. have developed a multi-modeling platform called AA4MM (Agent and Artefact for Multiple Models) [20]. Its main goal is reusability and interoperability of different simulators with a software architecture that is completely decentralized and based upon the multi-agent paradigm. Each simulator is controlled by a simulator manager (formally an agent) which is an autonomous entity. All of these manager/agents cooperate in order to run the whole simulation and to take care of the interaction problematics. Such an approach has been validated by coupling a user behavior simulator (MASDYNE) with a MANET simulator (JANE [21]). The most interesting result of such an approach is the ability to take mutual influences of user behaviors and network performances into account. However, this is not always possible. For example, ns-3 is not designed for being used as an on-demand provider of data items (e.g., the delay of a specific packet transmitted wirelessly in a complex environment); instead, it is particularly suitable to collect general network statistics. Fortunately, co-simulation can be also implemented by means of sequential integration of powerful, independent tools. In Section 5 we present our integration of DEUS and ns-3.

3 DEUS

DEUS is a multi-platform tool, developed in Java language (the code can be downloaded from the official site [22]). Its API, by subclassing, enables the implementation of (i) *nodes*, i.e. the entities which interact in a complex system, leading to emergent behaviors such as humans, pets, cells, robots or intelligent agents; (ii) *events*, e.g., node births and deaths, interactions among nodes, interactions with the environment, logs and so on; and (iii) *processes*, either stochastic or deterministic ones, constraining the timeliness of events.

Once specific Java classes have been implemented, it is possible to configure a simulation by means of the DEUS graphical user interface, which includes:

- the Visual Editor, for the generation of XML documents describing specific simulations;
- the Automator, for the execution of parametric simulations and the automatic generation of statistics in a Gnuplot-compliant format.

Figure 16.1 illustrates how DEUS simulation models are created (using also a Visual Editor), and then executed by the Engine, which is the core of DEUS, managing the event queue and the simulation loop. The Automator allows sensitivity analysis to be performed, by setting ranges for node and process parameters.

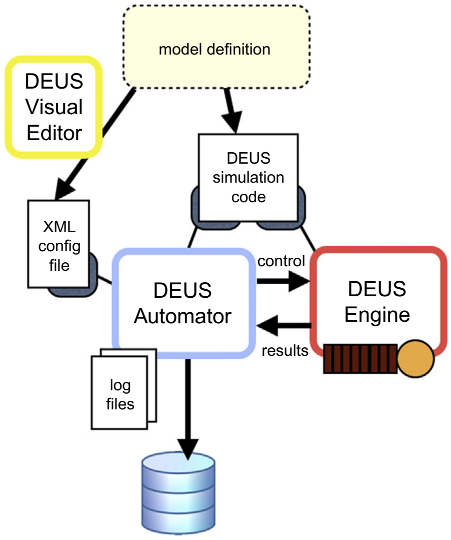

FIGURE 16.1

Discrete event simulation with DEUS.

A node may represent a dynamic system characterized by a set of possible states, whose transition functions may be implemented either in the source code of the events associated to the node, or in the source code of the node itself. Multi-scale modeling of complex systems can be achieved by defining nodes of different complexity, and connecting them. DEUS comes with a library of predefined, common processes, and many others can be implemented by the user.

3.1 DEUS API STRUCTURE AND FEATURES

Since DEUS is a general-purpose simulator, basic interfaces and classes are kept separated from more specific ones. By means of subclassing, it is possible to create specific modules for the simulation of any type of complex system. An extension package related to peer-to-peer resource sharing networks is provided by default.

The experience we acquired during the development of other simulation code (mainly using ns-2 and PeerSim [23]) showed us how difficult it is to manage memory when it comes to the simulation of systems with a large number of interacting parts (nodes, if systems are described as graphs). Java is an extremely powerful language and the flexibility of its object orientation, plus the reflection mechanism, make it highly suitable to build such a type of project. However, the difficulties in managing the garbage collection mechanism require a good design of the memory management. For these reasons, as we describe in more detail later on, DEUS relies on an efficient cloning mechanism: the initial process loads configuration objects into memories and new instances of those objects are obtained through deep cloning.

3.2 SIMULATION OBJECTS AND BEHAVIOR

The development of DEUS started from the definition of the basic simulation objects and the design of the configuration procedure, having in mind all the dynamics of complex systems that one may need to simulate. The goal was to achieve high flexibility and usability, allowing developers to specify a section with simulation objects and another one with simulation behavior, maximizing the possibility to reuse components and providing self-validation constraints so that the engine could process the configuration file through reflection and without any further validation. In particular, we have recently demonstrated that DEUS allows testing of deployment software on simulated devices and environments [24]. Simulation objects are events, nodes, and resources, while simulation behavior is managed through processes and engine objects.

An *event* represents the base simulation unit: i.e., the piece of code that is going to be scheduled by the system. Moreover, as complex systems are made by interacting components, we introduced the concept of *node*, which also corresponds to a data structure collector the event could rely on. Each node can have a set of *resources*, a structured way to represent objects the node can share or use through the event code. The association between events and nodes is given by *process* objects, which are responsible for event schedule timing calculation. The *engine* object puts everything together by linking events that are scheduled at the beginning of the simulation.

The simulation behavior follows the standard model of discrete event simulations: initialization of system state variables and clock, scheduling of initial events and, until the ending condition is true, calculation of next clock time and processing of the next event in the scheduling queue. However, a few additions have been made to make the model more flexible. For each event, it is possible to specify whether its execution is *one-shot*, so that the event will be removed from the schedule after its completion, or not, so that the event will be rescheduled according to the timing given by its associated process. Moreover, each event is provided with a listening mechanism over the scheduling process so that the latter will be able to schedule other events, namely *referenced events*, right after the event's execution. The ending condition of the simulator happens once the maximum simulation time has been reached or the scheduling queue is empty.

3.3 DEUS CORE

DEUS has been divided into packages, each one addressing a specific aspect of the simulation. The root package is `it.unipr.ce.dsg.deus`, containing the following subpackages:

- `core` – base system components including simulation object interfaces, configuration parser and engine;
- `schema` – object model representing the configuration file;

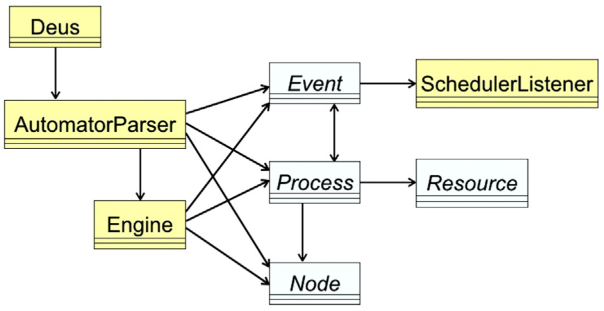

FIGURE 16.2

Class diagram of DEUS `core` package.

- `util` — support classes for the simulation engine;
- `impl.event` — reference implementations of the event object;
- `impl.node` — reference implementations of the node object;
- `impl.resource` — reference implementations of the resource object;
- `impl.process` — reference implementations of the process object.

In the following we provide a detailed description of the main classes contained in each package. A class diagram of the core package is shown in Figure 16.2.

3.3.1 `core` *package*

The `Event` class represents the simulation object being scheduled by the Engine. Each event is identified by a configuration id, a set of properties, a flag indicating if the event should be executed only once, a set of referenced events, a parent process, the triggering time and a listener to handle the execution of referenced events. In order to keep the simulation memory area as small as possible, each event is created by cloning the original event obtained from the simulation configuration parser; therefore, each implementing class should provide the code for cloning the event ensuring that its internal state is consistent, by reinitializing the event members that do not have to be cloned.

The `Node` class represents a generic data structure collector inside the simulation, so the main use is to store, read and delete information useful to characterize the simulation state. Each node is identified by a configuration id, a set of properties and a set of resources. Similarly to the `Event` class, there is the same cloning mechanism to keep the memory requirements small for the simulation execution.

The `Resource` class represents a generic resource associated to a node, with getter and setter methods.

The `Process` class represents the simulation object responsible to determine the timestamps of the events to be scheduled. Each process is identified by a configuration id, a set of properties, a set of referenced nodes and a set of referenced events.

The `Engine` class represents the simulation engine of DEUS. After the configuration file has been parsed, the obtained configured simulation objects (nodes, events and processes) are passed to the Engine, to let it properly initialize the queue of events to be run. The simulation is a standard discrete event simulation where each event has an associated triggering time, used as a sorting criteria. The events inserted into the simulation queue are processed individually one after the other, each time updating the current simulation virtual time. The run method of the engine will process each event in the event queue until a maximum virtual time is reached or the queue is empty. In each cycle the first event of the queue is removed (the one with the lowest triggering time), the virtual time of the simulation is updated and the event is executed. If the event has some referenced events, those will be scheduled right after the event execution. If the event is not one-shot and has a parent process, then it will be scheduled for execution with a triggering time calculated according to the parent process strategy.

The `AutomatorParser` class is responsible for handling the simulation configuration file, according to the *DEUS XML schema*. The configuration can be seen as a set of nodes, resources, events, processes and engine parameters. The `AutomatorParser` class handles the configuration of each simulation object and stores them in a set of array data structures. Each simulation object has a set of base features, plus references to other simulation objects: nodes can have a set of resources, events can have a set of referenced events, and processes can have references to both nodes and events. At the end of the configuration file parsing process, the `AutomatorParser` initializes the Engine object enabling the simulation execution.

3.3.2 `impl.event` package

The `BirthEvent` class represents the birth of a simulated node. During its execution, an instance of the node associated to the event is created.

The `DeathEvent` class represents the death of a simulated node. During the execution of the event the associated node is killed or, if nothing is specified, a random node is chosen instead.

The `LogPopulationSizeEvent` class is used to simulate a logging event that stores the number of nodes in the simulation, each time it is scheduled. It demonstrates that an event can really be anything, in the context of the complex system to be simulated.

3.3.3 `impl.node` package

The `BasicNode` class is the default implementation of the node abstract class, without any specific properties. A specific implementation is provided in the p2p package, which is described later in the chapter.

3.3.4 `impl.resource` package

The `AllocableResource` class represents a generic allocable resource, having a type/amount pair parameter which must be specified through the configuration file.

The `ResourceAdv` class represents a resource advertisement, i.e., a document that describes a `ConsumableResource` (with a name and an amount), and the interested node. Once the resource described by a `ResourceAdv` has been discovered, the owner of the resource should be registered into the `ResourceAdv`, and the found flag set to true.

3.3.5 `impl.process` *package*

The `PeriodicProcess` represents a generic periodic process. It has a parameter called *period*, which is used to generate the triggering time. Each time the process receives a request to generate a new triggering time, it computes it by adding the period value to the current simulation virtual time. An extension of this class is provided through the `TwoSpeedsPeriodicProcess` class that allows the specification of two different periods; the switch between first period and second period is made using a virtual time threshold.

The `PoissonProcess` represents a generic Poisson process. It has one parameter called *meanArrival*, which is used to generate the triggering time. Each time the process receives a request to generate a new triggering time, it computes it by adding the current simulation virtual time to the value of a homogeneous Poisson process with the rate parameter calculated as 1/*meanArrival* time.

Similarly to the `TwoSpeedPeriodicProcess`, there is the `TwoSpeedPoissonProcess` class to provide a Poisson Process that changes its speed after a virtual time threshold has been reached. Other classes allow for limiting the event scheduling to a time period, by specifying a starting time and an ending time.

3.4 EXTENSION PACKAGE FOR THE SIMULATION OF PEER-TO-PEER SYSTEMS

To simulate a particular class of complex systems, namely peer-to-peer resource sharing networks, we implemented the `it.unipr.ce.dsg.deus.p2p` package, which contains the following subpackages:

- `node` – the model of peer;
- `event` – the events characterizing a P2P network.

In the following we provide a detailed description of the main classes contained in each package. The related class diagram is illustrated in Figure 16.3.

3.4.1 `node` *package*

The `Peer` class is an extension of the `Node` class that represents the concept of peer in a network. Each peer is identified by a unique key generated by the engine (in the given key space) and is characterized by a list of neighbors, i.e., peers with whom it has an active link connection, and a status regarding peer connection to the network (whether is connected or not). Some methods have been implemented to manage neighborhood and notification messages.

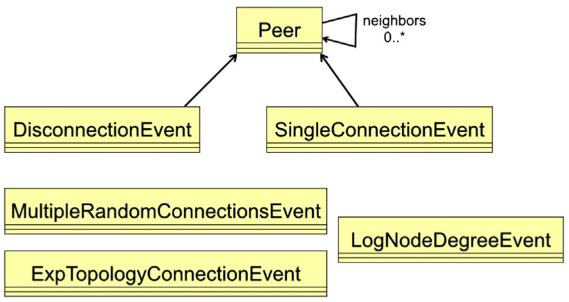

FIGURE 16.3

Class diagram of the `p2p` package.

3.4.2 `event` *package*

The `SingleConnectionEvent` class simulates the connection event of a peer in the network. The peer can connect to a randomly chosen node, or to a specific one.

An extension of this class is provided through the class called `MultipleRandomConnectionsEvent`, which enables a connection to more than one node, randomly chosen in the network.

The `DisconnectionEvent` class is used to disconnect a specific node from the network. Alternatively, it can be used to disconnect a random node from the network.

The `LogNodeDegreeEvent` class provides a logger that computes the node degree distribution for each peer of the network. The result is an array, whose index represents the node degree, and each value is the number of nodes that have the node degree corresponding to the considered index.

4 NS-3

Ns-3 is a discrete-event network simulator for Internet systems. It is a free, open source software project (with GPLv2 licensing) organized around research community development and maintenance. Like its predecessor ns-2, ns-3 relies on C++ for the implementation of the simulation models. However, ns-3 no longer uses oTcl scripts to control the simulation, thus overcoming the problems which were introduced by the combination of C++ and oTcl in ns-2. Instead, network simulations in ns-3 can be implemented in pure C++, while parts of the simulation optionally can be realized using Python as well.

Moreover, ns-3 integrates architectural concepts and code from GTNetS [25], a simulator with good scalability characteristics. Such design decisions were made at the expense of compatibility—porting ns-2 models to ns-3 must be done in a manual way. Besides performance improvements, the simulator has an extended feature set. For example, ns-3 supports the integration of real implementations code by providing standard APIs, such as Berkeley sockets or POSIX threads, which are transparently mapped to the simulation.

Among the packages being developed for ns-3, the LENA LTE-EPC is particularly rich and efficient [26]. In the LTE-EPC simulation model, there are two main components. First, the LTE Model, which includes the LTE Radio Protocol stack (RRC, PDCP, RLC, MAC, PHY). Such entities reside entirely within the User Equipment (UE) and the E-UTRAN Node B (eNB) nodes. Second, the EPC Model, including core network interfaces, protocols and entities, which reside within the SGW, PGW and MME nodes, and partially within the eNB nodes.

5 INTEGRATION OF DEUS AND NS-3

As illustrated in Section 3, to simulate a distributed system with DEUS, it is necessary to write the classes that represent nodes, events and processes. Node may represent devices, servers, virtual machines, applications, etc. Events may be associated to specific nodes (e.g., start, connection, disconnection, internally/externally triggered state change, stop, etc.), or involving several nodes (it is the case of logging events). To simulate a message delivery from one node to another, it is necessary to define the sender, the destination and to schedule a "delivered message" event in the future (in terms of virtual time of the simulation). The scheduling time of such an event must be set using a suitable process, selected among those that are provided by the DEUS API, or defined by the user, possibly.

For example, if the purpose of the simulation is to measure the average delay of propagating multi-hop messages within a network of nodes (e.g., a peer-to-peer network), the value of each link's delay must be realistic, taking into account the underlying networking infrastructure. In particular, if the communication is wireless, estimating the delay of point-to-point communication is a challenging task.

The direct integration of DEUS with ns-3, with the former that "calls" the latter to compute a delay value every time a node must send a message to another node, taking into account current surrounding conditions, is unpractical and would highly increase the duration of the simulation. Instead, a more effective and efficient solution (illustrated in Figure 16.4) includes the following steps:

1. given a complex system to be simulated, identify the main subsystem types, each one being characterized by specific networking parameters;
2. with ns-3: create detailed simulation models of the subsystems (i.e., submodels) and measure their characteristic transmission delays, taking into account both message payloads and proper headers;
3. with DEUS: simulate the whole distributed system, with refined scheduling of communication events, taking into account the transmission delays computed at step 2.

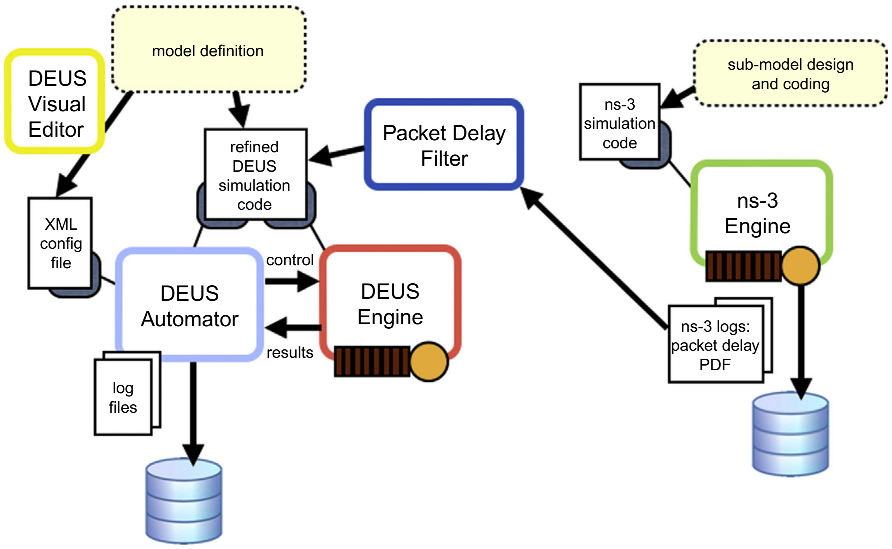

FIGURE 16.4

Discrete event simulation with DEUS and ns-3.

For example, if the overlay network relies on a cellular network, the submodel to be characterized with ns-3 could be a set of cells. Multicell communication may be very fast, if base stations are connected by optical fibers [27]. However, intercell interference and horizontal handover could be taken into account, when simulating mobile nodes. Moreover, the simulation of each cell should take into account the presence of other mobile nodes, which are not directly involved in the distributed application of interest, but consume significant resources. Finally, the same subsystem could be simulated with different geographic conditions, e.g., in a city (with small cells, buildings, and noisy channel), or in a rural area (with larger cells and a less disturbed channel).

Regarding step 2, with reference to the LTE package, it is necessary to modify the C++ class that logs the uplink and downlink delays. The modified class must log a discretized probability density function (PDF) of the RLC packet delay. Such a discretized PDF is then used to generate realistic packet delays in the DEUS-based simulations, using the well-known *inversion method* [28], which is based on the inverse probability theorem:

- choose the cumulative distribution function $F(x)$ of the random variable to be sampled;
- generate a set of uniform random numbers such that $R \sim U(0,1)$;
- compute the random variate $X_i = F^{-1}(R_i)$.

The Packet Delay Filter, illustrated in Figure 16.4, is a Java module that approximates the discretized PDF by a piecewise constant function, whose numerical inversion is straightforward and computationally inexpensive. The Packet Delay Filter implements the following algorithm:

1. put the points of the discretized PDF in a list L
2. divide L into n sub-lists
3. for each sub-list, compute the mean value of the points
4. merge two neighbor sub-lists, if the difference of their mean values is below t
5. repeat from step 2 until the set of sub-lists converges

The algorithm is repeated several times, for different values of t, in order to find the best set of sub-lists—i.e., the one that corresponds to a piecewise constant function whose integral is closer to 1. An example PDF approximated with the aforementioned algorithm is shown in Figure 16.7(b).

6 EVALUATION

We have applied the proposed methodology to model and simulate the Distributed Geographic Table (DGT), which is a peer-to-peer overlay scheme with the main objective to provide support for mobile node localization. Compared to centralized localization approaches, the DGT is more scalable, since its performance (in terms of responsiveness, completeness and robustness) remains valuable also for a large number of nodes, and when the nodes' dynamics are very high [29]. In a DGT-based system, the responsibility for maintaining information about the position of active peers is distributed among nodes, for which a change in the set of participants causes a minimal amount of disruption.

Every peer maintains a set of geo-buckets (GB), each one being a regularly updated list of known peers, sorted by their distance from the Global Position of the peer itself. GBs can be represented as concentric circles, each one having a different (application-specific) radius and thickness. The distance between two DGT peers is defined as the actual geographic distance between their locations in the world. The neighborhood of a geographic location is the group of nodes located inside a given region surrounding that location.

The main service provided by the DGT overlay is to route requests to find available peers in a specific area, i.e., to determine the neighborhood of a generic global position (Figure 16.5). The routing process is based on the evaluation of the region of interest centered in the target position. The idea is that each peer involved in the routing process selects, among its known neighbors, those that presumably know a large number of peers located inside or close to the chosen area centered in the target point. If a contacted node cannot find a match for the request, it does return a list of closest nodes, taken from its routing table. This procedure can be used both to maintain the peer's local neighborhood and to find available nodes close to a generic target.

Further details about the DGT can be found, for example, in recent articles by Picone et al. [29,30]. Simulation results presented there were obtained by means

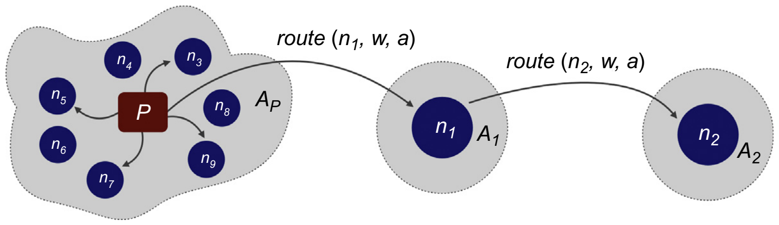

FIGURE 16.5

Propagation of a DGT query between nodes to retrieve the neighborhood of a local or remote region of interest.

of a DEUS simulation model, integrated with Google Maps to have a realistic characterization of the urban environment (the city of Parma). However, simplistic assumptions on the packet transmission delay were made. In the following we illustrate how the methodology illustrated in this chapter has been used to simulate the DGT with more realistic packet transmission delays.

The simulation considers a number of vehicles that move over 100 km of realistic paths, generated using the Google Maps API. Each simulated vehicle selects a different path and starts moving over it. Using the features provided by the Google Maps API, we created a simple HTML and Javascript control page, which allows the monitoring of the time progression of the simulated system, where any node can be selected to view its neighborhood (demo videos are available online [31]).

The simulation covers 10 hours of DGT system life, with 500 to 2000 mobile nodes, 5 virtual paths with bad road surface (due to either ice, water, snow, or pothole), accident events scheduled during the simulation according to a Poisson stochastic process and with different message types to disseminate information about sensed data and traffic situation. Simulations with DEUS have been repeated with 20 different seeds for the random number generator, which are sufficient to obtain a narrow I95 confidence interval (5% of the steady state value, in the worst case). Obtained graphs consider means and standard deviations obtained by averaging over the whole set of simulated nodes, and over the 20 different simulation runs.

The considered DGT configuration is the one that gives the best performance in urban scenarios [29]. Each node has 4 GBs with a 0.5 km thickness and a peer discovery limiting number equal to 10 nodes, covering a region of interest of 12.5 km^2 and an adaptive discovery period ranging from 1.5 min to 6 min, depending on the number of new discovered nodes during each lookup process. The period increases with the knowledge degree of the node neighborhood, corresponding to the decrement of the number of new discovered peers in the same area of interest.

The transmission delay of a DGT packet has been computed by simulating with ns-3 the subsystem illustrated in Figure 16.6 (by averaging over 20 simulation runs), using the LTE package illustrated in Section 4 [32]. To match the previously described DGT configuration (i.e., DGT peers having GBs with radius of 2 km), we defined a square area having side length $l = 2$ km, with a grid of $r = 10$

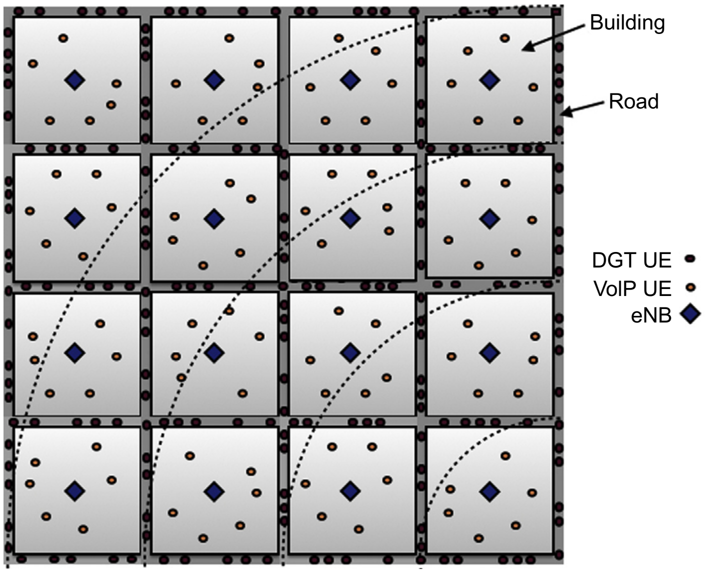

FIGURE 16.6

Bird's-eye view of the simulated scenario, with $n = 200$ DGT nodes and $n_v = 96$ other UEs randomly placed within the buildings. The geo-buckets of the DGT node in the bottom right corner of the map are also drawn, to show that the side length of the considered area equals the GB radius.

roads (5 in the N-S direction, and 5 in the W-E direction) and vehicles running over them (with linear density δ). The total amount of DGT User Equipments (UEs) is $n = r\ \delta\ l$. Parallel roads are spaced by $l/4 = 0.5$ km. In the map, there are 16 large buildings with square footprint, each one having seven floors. Randomly located within each building, there are $n_v/16$ other UEs, where n_v is their total amount. The path loss model is `ns3::BuildingsPropagationLossModel`. On top of each building, exactly in the middle, there is an eNB, i.e., a base station that serves a subset of the $n + n_v$ UEs. Such a dense deployment of eNBs may appear to be quite optimistic. We plan to test other models with $500-1000$ m radius cells, and 200 active users each, which should be the best estimates for near-term LTE deployment.

The configuration of the eNBs includes FDD paired spectrum, with 50 Resource Blocks (RBs) for the uplink (which corresponds to a nominal transmission rate of 50 Mbps) and the same for the downlink—this is coherent with currently deployed LTE systems. DGT UEs use UDP to send four types of DGT packets to each other. The first type, called `Descriptor` (24 bytes), is for neighborhood consistency maintenance purposes. The second type of packet is the `Lookup Request` (20 bytes), which is used to search for remote nodes placed around a specified location. The third packet type is the `Lookup Response`

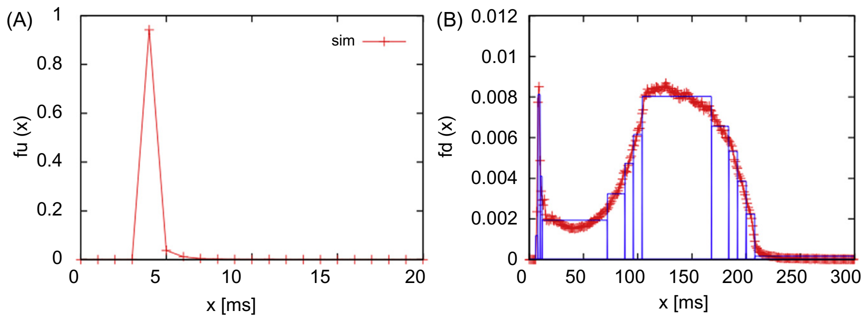

FIGURE 16.7

PDFs of the uplink (left) and downlink (right) delays for DGT packets (for the case with $\delta = 10$ vehicles/km), obtained with ns-3. The downlink PDF produced by means of ns-3 has 300 points. Its approximation obtained from the Packet Delay Filter has 13 levels > 0.

(500 bytes), which is sent by a DGT node as a reply to a lookup request, if the node owns the searched resource or information. Finally, there are traffic information packets (66 bytes). All packet types have also a 12 byte header. We set an interpacket interval of 50 ms for all types of DGT messages. Thus, the maximum rate is about 10 kB/s, while the minimum is $32 \times 20 = 0.64$ kB/s.

In a dynamic DGT scenario (the one simulated with DEUS), packets are not sent periodically—descriptors are sent only every ε meters; lookup requests are sent only when necessary, as well as lookup responses; traffic information messages are sent only when something interesting can be communicated to the other nodes (for example, a traffic jam or an incident). To simulate the presence of non-DGT traffic over LTE networks, we also included $n_v = 96$ other UEs, transmitting and receiving VoIP packets (using UDP) with a remote host located in the Internet. Such packets have a 12-byte header and a 13-byte payload, with interpacket interval set to 20 ms (we considered the AMR 4.75 kbps codec). The PDF of the resulting uplink delay is basically a Dirac delta function, shown in Figure 16.7a. Instead, the PDF of the downlink delay can be approximated with a corresponding piecewise constant function, with 13 levels, shown in Figure 16.7b.

Such delay profiles scale from small scenarios to larger ones, as they refer to intra-GB communications only. A DGT message could be propagated across the whole city, from one peer to another, relayed by intermediate peers. Each message propagation would be affected only by the data traffic within the GB of the forwarding peer, where the obtained delay profiles apply.

Such a packet delay model is a considerable improvement with respect to the one we used in our previous DEUS-based DGT simulations, which used, for every transmission, an exponential delay with mean value obtained by considering the nominal uplink and downlink.

Then, while simulating the whole overlay network with DEUS, we logged the average packet delay and amount of sent data per node. Figures 16.8 to 16.10

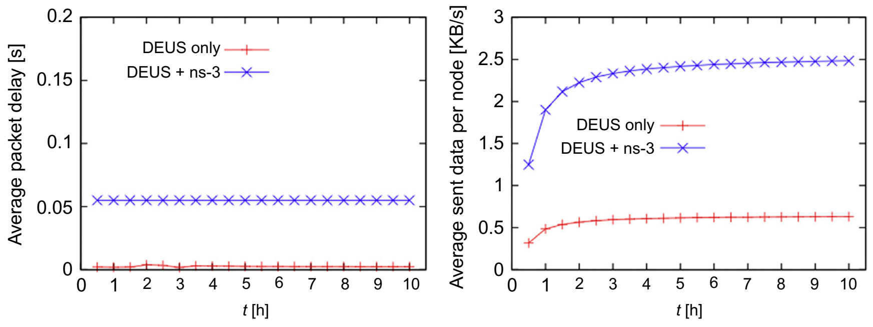

FIGURE 16.8

Average packet delay (left) and amount of sent data per node (right), measured with DEUS, for the simulated DGT overlay network with 500 vehicles.

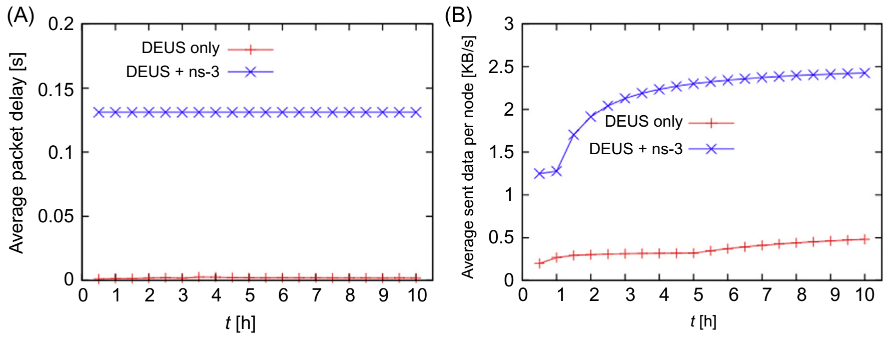

FIGURE 16.9

Average packet delay (left) and amount of sent data per node (right), measured with DEUS, for the simulated DGT overlay network with 1000 vehicles.

compare the results obtained with the old simulation model, and those obtained with the refined one, for different network sizes.

As we expected, in the refined model the average delay is higher than the one obtained with the naive model, which is based on nominal uplink and downlink values. Also the average amount of sent data is higher, as the refined model takes into account also packet headers.

7 CONCLUSION

In this chapter we have presented an effective and efficient co-simulation solution for wireless and mobile systems, based on the general-purpose simulation tool called DEUS, and the network-specific simulation tool called ns-3. With respect

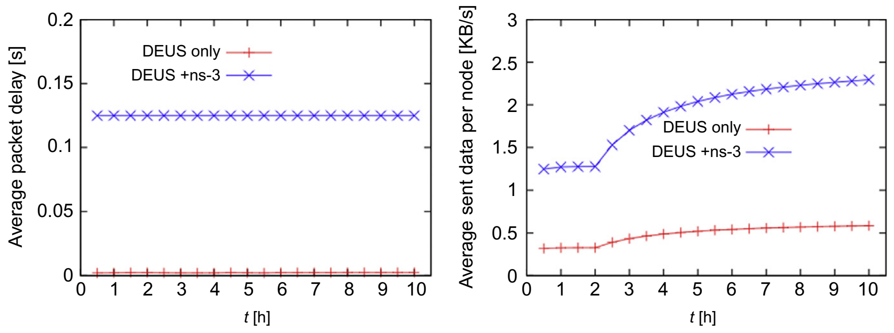

FIGURE 16.10

Average packet delay (left) and amount of sent data per node (right), for the simulated DGT overlay network with 2000 vehicles.

to the state of the art, such a solution has two main advantages. First, ns-3 allows us to obtain highly detailed statistics, encompassing all network layers. Second, DEUS is highly flexible, allowing us to simulate any type of mobility model, and to use deployment software on simulated devices and environments. The proposed approach has been successfully applied to the simulation of a peer-to-peer overlay with mobile nodes, associated to vehicles in a urban scenario.

REFERENCES

[1] Gershenson C, Heylighen F. How can we think the complex? In: Richardson K, editor. Managing organizational complexity: philosophy, theory and application. Information Age Publishing; 2005, Chapter 3.

[2] Zeigler BP, Praehofer H, Kim TG. Theory of modeling and simulation. 2nd ed. Academic Press; 2000.

[3] Wainer G. CD++: a toolkit to develop devs models. Software – Pract Exp 2002;32 (13):1–46.

[4] Varga A, Hornig R. An overview of the OMNeT++ simulation environment. In: First international conference on simulation tools and techniques for communications, networks and systems (SIMUTools 2008), Marseille, France, Mar. 2008.

[5] Amoretti M, Agosti M, Zanichelli F. DEUS: a Discrete Event Universal Simulator, 2nd ICST/ACM International conference on simulation tools and techniques (SIMUTools 2009), Roma, Italy; March 2009. ISBN 978-963-9799-45-5.

[6] NS-3 Consortium. ns-3. Official website. Available from: <http://www.nsnam.org>; 2014.

[7] University of Luxembourg. The OVNIS platform. Website. Available from: <http://ovnis.gforge.uni.lu>; 2014.

[8] Behrisch M, Bieker L, Erdmann J, Krajzewicz D. SUMO – simulation of urban mobility: an overview, in SIMUL 2011. In: Third international conference on advances in system simulation, Barcelona, Spain; October 2011. p. 63–8.

[9] Årzén KE, Cervin A. Control and embedded computing: survey of research directions, presented at the 16th IFAC World Congress, Prague, Czech Republic; 2005.

[10] Colandairaj J, Irwin GW, Scanlon WG. An integrated approach to wireless feedback control, presented at the UKACC international control conference, Glasgow, UK; 2006.

[11] Colandairaj J, Irwin GW, Scanlon WG. Analysis and co-simulation of an IEEE 802.11B wireless networked control system, in 16th IFAC world congress, Prague, Czech Republic; 2005.

[12] Chen Z, Liu L, Zhang J. Observer based networked control systems with network-induced time delay. In: IEEE International conference on systems, man and cybernetics, Hague, The Netherlands; 2004. p. 3333−7.

[13] Liu GP, Rees D, Chai SC. Design and practical implementation of networked predictive control systems. In: International conference on networking, sensing and control, Arizona, USA; 2005. p. 336−41.

[14] Yang Y, Wang Y, Yang SH. A networked control system with stochastically varying transmission delay and uncertain process parameters. In: 16th IFAC world congress, Prague, Czech Republic; 2005.

[15] Cervin A, Hanriksson D, Lincoln B, Eker J, Arzen KE. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. IEEE Control Syst Mag 2003;23(3):16−30.

[16] Andersson M, Henriksson D, Cervin A, Årzén KE. Simulation of wireless networked control systems, presented at the 44th IEEE conference on decision and control and European control conference (ECC); 2005. p. 476−81.

[17] Al-Hammouri A, Liberatore V, Al-Omari H, Al-Qudah Z, Branicky MS, Agrawal D. A co-simulation platform for actuator networks. In: Fifth international conference on embedded networked sensor systems, Sydney, Australia; 2007. p. 383−4.

[18] Hasan MS, Yu H, Griffiths A, Yang TC, Interactive co-simulation of MATLAB and OPNET for networked control systems. In: 13th international conference on automation and computing, Stafford, UK; 2007. p. 237−42.

[19] Hasan MS, Yu H, Carrington A, Yang TC. Co-simulation of wireless networked control systems over mobile ad hoc network using SIMULINK and OPNET. IET Commun 2009;3(8):1297−310.

[20] Leclerc T, Siebert J, Chevrier V, Ciarletta L, Festor O. Multi-modeling and co-simulation-based mobile ubiquitous protocols and services development and assessment. In: Seventh international ICST conference on mobile and ubiquitous systems − mobiquitous 2010. Sydney, Australia; 2010.

[21] Gorgen D, Frey H, Hiedels C. Jane - the Java Ad hoc network development environment, In: ANSS '07; 2007. p. 163−76, USA.

[22] Deus: a simple tool for complex simulations. Official website. Accessed at: <https://code.google.com/p/deus/>; 2015.

[23] Montresor A, Jelasity M. PeerSim: a scalable P2P simulator. In: Ninth IEEE international conference on Peer-to-Peer (P2P'09), Seattle, WA, USA; September 2009.

[24] Brambilla G, Grazioli A, Picone M, Zanichelli F, Amoretti M. A cost-effective approach to software-in-the-loop simulation of pervasive systems and applications, in PerCom 2014, WiP Session, Budapest, Hungary, March 2014.

[25] Riley G. Large scale network simulations with GTNetS. In: Winter simulation conference, New Orleans, Louisiana, USA; Dec. 2003.

[26] Baldo N, Requena-Esteso M, Nin-Guerreo J, Miozzo M. A new model for the simulation of the LTE-EPC data plane, in fifth ICST/ACM international conference on simulation tools and techniques (SIMUTools 2009), Sirmione, Italy; Mar. 2012.

[27] Nagate A, Hoshino K, Mikami M, Fujii T. A field trial of multicell cooperative transmission over LTE system. In: IEEE international conference on communications (ICC 2011), Kyoto, Japan; Mar. 2011.

[28] Papoulis A. Probability, random variables, and stochastic processes. 3rd ed. McGraw Hill; 1991.

[29] Picone M, Amoretti M, Zanichelli F. Proactive neighbor localization based on distributed geographic table. Int J Pervasive Comput Commun 2011;7(3):240−63.

[30] Picone M, Amoretti M, Zanichelli F. Evaluating the robustness of the DGT approach for smartphone-based vehicular networks. In: Fifth IEEE workshop on user mobility and vehicular networks, Bonn, Germany; Oct. 2011.

[31] Distributed Systems Group (DSG). DGT − Distributed Geographic Table. Demo videos. Available from: <http://dsg.ce.unipr.it/?q=node/38#media>; 2015.

[32] Amoretti M, Picone M, Zanichelli F, Ferrari G. Simulating mobile and distributed systems with DEUS and Ns-3, international conference on high performance computing and simulation 2013, Helsinki, Finland; July 2013. Proceedings published by IEEE.