A Flexible Framework for Modeling and Simulation of Multipurpose Wireless Networks

Vinicius Bohrer, Ramon Fernandes, César Marcon, Thais Webber,
Ricardo M. Czekster, Letícia B. Poehls, Fabiano Hessel

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga 6681, Porto Alegre, Brazil
cesar.marcon@pucrs.br

Abstract— **The emergence of wireless networks has contributed to a growing number of studies and protocols regarding its performance and reliability requirements, among others. Several issues have to be considered when deploying such devices under harsh environmental conditions. These issues often force the designer to adopt decisions that are usually difficult to verify in real world settings. In order to mitigate such problems, an alternative resides in the use of simulation models for both homogeneous and heterogeneous devices. This paper describes an event-based Wireless Network Simulator (WiNeS) for devices operating in several topologies and configurations of networks. WiNeS is a Java-based framework specially built to support customized network options that offers hybrid simulation for virtual and physical nodes in the same environment. Some of WiNeS' features include the computation of maximum communication distances among devices in 2D and 3D spatial node distributions as well as pairing rules to evaluate the nodes connectivity.**

Keywords - Hybrid simulation, Event-based simulation, Network topologies, Wireless communication.

I. INTRODUCTION

Nowadays, many proposals of wireless networks for joining some technologies such as Bluetooth [1], Wi-Fi [2], and Zigbee [3] on the same network have been presented. These networks use different frequencies and communication protocols, forming multiprotocol systems for a range of applications. However, several problems arise: (i) how to choose the best node disposition in the environment in order to ensure the correct pairing between devices, and (ii) how to identify the most efficient device type to perform a given communication. Such issues demand alternatives to evaluate test scenarios before the systems are deployed.

Several non-commercial open source network simulators of general purpose, like NS-2 [4], NS-3 [5], OMNet++ [6], GloMoSim [7], JiST/SWANS [8], are currently available to perform modeling and simulation analysis. The simulators present differences in terms of architectures, features and applicability. Many of them are platform dependent and have issues related to scalability and end user facilities [9], [10], [11]. The main concern addressed in the present work is the need of simulation environments able to bear heterogeneous devices and several inner and external simulation events (originated from virtual and physical nodes respectively, in a controllable environment).

The Wireless Network Simulator (WiNeS) presented in this paper is a platform independent framework written in Java. It supports the following Models of Computation (MoCs) [12]: Discrete Events (DE), Synchronous Reactive Events (SRE) and Asynchronous Reactive Events (ARE). DE enables the simulation of a series of timed events, whereas SRE are capable of describing synchronous-reactive network elements. In fact, the combination of both simulation models can cope with a wide variety of situations. However, these models are inadequate when dealing with unpredictable behaviors and therefore WiNeS incorporates ARE as an alternative simulation model, allowing the simulation of non-deterministic events. Such model is advantageous for the integration of virtual nodes with physical nodes that are very susceptible to unpredictable behavior, from a simulation point of view.

WiNeS' goal is to provide a flexible simulation environment to accommodate any wireless network topology, communication protocols and device types, while remaining simple enough for the end user to implement such features, without a deep understanding of the underlying simulator architecture and its operations. Additionally, the simulation engine is constructed to be as lean as possible, allowing nodes scalability as well as providing facilities to heterogeneous devices modeling with multiple communication capabilities.

The paper is organized as follows. Section II presents the WiNeS framework and Section III describes the framework capabilities and features in detail, instantiating simulation scenarios. Section IV presents related simulators characteristics in comparison with WiNeS framework. Finally, Section V concludes our contribution with discussions about WiNeS future improvements.

II. WINES FRAMEWORK

WiNeS framework presents the following characteristics: (i) verification of nodes connectivity; (ii) simulation of different network topologies; (iii) simulation of homogeneous and heterogeneous architectures; and (iv) simulation of user-implemented protocols in a hybrid environment with virtual and physical nodes. The framework intends to be generic to support a wide range of topologies, architectures, and communication types, depending exclusively on the designer's implementations, while guaranteeing the enforcement of node communication rules, i.e. based on protocols, frequencies and geographic locations. Note that WiNeS is not readily aware of details pertaining protocols or device types. Thus, any additional information required for network operations should be developed and informed by the designer. Figure 1 depicts that WiNeS consists of a flexible simulation environment for wireless networks. The **Communication infrastructure** implements TCP/IP sockets as the message exchanging mechanism, which allows the transmission of any type of data between simulated devices. Each data sent from one node to another passes through the **Environment emulation**, which checks whether both devices are communicating properly.

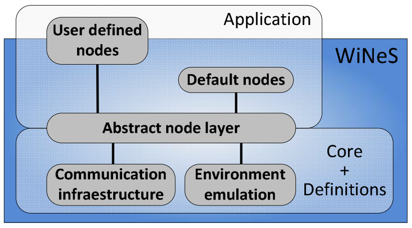

Figure 1. Software layers of WiNeS framework.

Although sockets lead to a larger overhead during the simulation, especially when compared to straightforward and less resource demanding models (i.e. shared memory), their adoption as a message exchanging mechanism between nodes allows the use of hybrid simulation scenarios. Unlike shared memory models, the sockets can be seamlessly used to connect the simulation core to external nodes, such as other simulation instances running distributed on offsite locations, or the direct interaction between virtual and real world nodes in a flexible simulation environment. Indeed, WiNeS presents an **Abstract node layer** for node's behavior description, which enables the addition of several network functionalities and protocols. Thus, to define the application, the framework provides: (i) a standard node behavior (**Default nodes**) based on predefined parameters, defining a discrete event model used for traffic generation and (ii) a customizable node layer (**User defined nodes**) that also can be used with default nodes as basis. The next Section describes the framework's architecture.

A. WiNeS' Architecture

Figure 2 shows WiNeS' architecture composed of 3 parts: (i) Definitions Block; (ii) Application Layer; and (iii) Core.

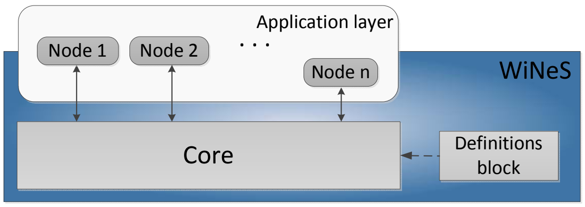

Figure 2. Main components of the WiNeS framework.

(i) Definitions Block

The Definitions Block is composed of a series of simulation rules containing characteristics of wireless technologies, such as total simulation time and parity rules for defining nodes communication and time/distance scales.

Definitions can be provided to WiNeS either by an XML or by a Java description file using the simulator's Java API. Observing Figure 3, it is possible to see that the XML file contains the following tags: (i) CONNECTIVITY_RULES: represent the set of rules specified by the designer for each pairing between two device types. Each rule describes the maximum distance between nodes to communicate on their frequencies. It is worth mentioning that these rules are optional in the simulation process, being dependent on the designer's choices to consider these values as useful information to take actions, perform numerical evaluations, or just use them as reference label. (ii) TIME_SCALE: indicates how often the simulator should increase a unit of simulation time, e.g. every 30 seconds the total simulation time should be incremented by one time unit. (iii) SIMULATION_TIME: indicates the time unit for the simulation run, e.g. the simulation time is set to 20x30=600 seconds. (iv) DISTANCE_SCALE: indicates the actual distance that each simulation unit represents.

```
<CONNECTIVITY_RULES>
    <RULE freq="2.4GHz" Dev1="antenna" Dev2="antenna" dist="100"/>
    <RULE freq="2.4GHz" Dev1="antenna" Dev2="WSN" dist="20"/>
</CONNECTIVITY_RULES>
<TIME_SCALE>
        <UNITY unity="seconds"/>
        <VALUE value="30"/>
</TIME_SCALE>
<SIMULATION_TIME>20</SIMULATION_TIME>
<DISTANCE_SCALE>
        <UNITY unity="km"/>
        <VALUE value="3"/>
</DISTANCE_SCALE>
```
Figure 3. Example of a XML input file for the Definitions Block.

(ii) Application Layer

The **Application Layer** is composed of nodes communicating with the Core via sockets. Nodes can be created from information provided in an XML file or in API methods. Figure 4 illustrates an XML file for mapping a heterogeneous node, containing temperature and humidity sensors integrated with an active RFID tag.

```
<NODE>
        <NODE_INFO>
                <POSITION X="3.0" Y="5.5" Z="8.0"/>
                <TYPE nodeType="0"/>
        </NODE_INFO>
        <COMPONENT_SPECIFICATION>
                <TYPE typeDevice="sensor"/>
                <OPERATION protocol="ZigBee" freq="2.4GHz" power="1.3"/>
        </COMPONENT_SPECIFICATION>
        <COMPONENT_SPECIFICATION>
                <TYPE typeDevice="activeTag"/>
                <OPERATION protocol="EPC Gen2" freq="5GHz" power="1"/>
        </COMPONENT_SPECIFICATION>
        <BEHAVIOR_SPECIFICATION>
                <TEMPERATURE time="0" value="12.8"/>
                <TEMPERATURE time="125" value="10.92"/>
                <HUMIDITY time="0" value="23.8"/>
                <HUMIDITY time="25" value="24.9"/>
        </BEHAVIOR_SPECIFICATION>
</NODE>
```
Figure 4. Example of an active RFID tag description.

The example depicts a non-passive node located in a 3D position (X=3,Y=5,Z=8). It has two antennas for transmission/reception. Its sensor part uses Zigbee protocol at a frequency of 2.4GHz for communication, whereas its RFID part uses EPC Gen2 protocol at 5GHz.

The node's information is mapped within two tables: (i) **Specification Table**: recognized by the tags <NODE_INFO> and <COMPONENT_SPECIFICATION>. This table contains the data referring to the node's general characteristics (e.g. protocol, frequency and position), representing its communication capabilities. Remark that the node specification, thus describing a heterogeneous device, may have more than one specification tag. (ii) **Behavior Table**: recognized by the tag <BEHAVIOR_SPECIFICATION>. This table contains data related to the node behavior considering the simulation time units, i.e. the node may express all instants in which a given parameter changes, with a discrete event list. The *nodeType* information obtained from the tag <NODE_INFO> indicates if the node is passive ('1') or non-passive ('0'). If a node is passive, it has no autonomy to initiate communication, and responds to stimuli originated from other nodes only. When creating a node, the designer can specify a customized **Application Layer**, thus eliminating the need of the predefined **Behavior Table**. However, it is still possible to rely on the events defined in this table if needed.

Figure 5 illustrates the node's architecture, where dashed arrows indicate a read access, dotted arrows illustrate messages exchange between node and Core, and continuous lines with arrows indicate the simulation flow.

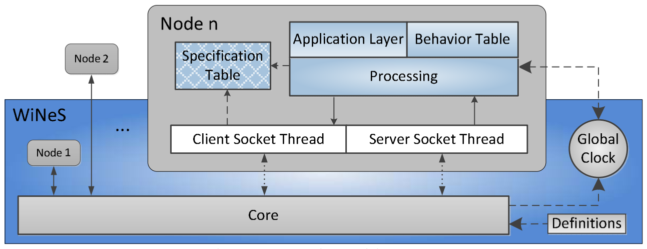

Figure 5. Node architecture.

The node's operation depends on the designer's implementation. If a customized **Application Layer** is defined for a node, it behaves accordingly; otherwise the node relies on the following default behavioral patterns: (i) current-time behavior, i.e. the node checks its **Behavior Table** to verify if it has some data to be sent to the simulation environment, based on the current simulation time; (ii) when the simulation Core forwards data from a node to other node, i.e. each time a node may communicate with others by sending requests to a specific IP address and port number.

The **Application Layer** implemented by the designer may be coupled to the default processing system within the node. The framework provides an API with node functions enabling this layer to receive and to send packets to other nodes. Therefore, the node may assume the processing implemented by the designer, i.e. simulating protocols, network topologies, packet loss, or any other desired functionality, and/or rely on predefined behaviors described on the node's **Behavior Table**. The Node-Core communication is performed by two threads: (i) **Client Socket Thread** and (ii) **Server Socket Thread**; connecting the elements through a socket.

The **Global Clock** maintains temporal coherence in communicating events. This clock starts at zero and is sequentially incremented by a time unit, defined on **Definitions Block**. The clock can be observed by all elements included in the framework during the simulations. A node without a customized Application Layer uses only the Behavior Table and consequently, employs the Global Clock to determine the set of events that should occur during the simulation's execution.

Furthermore, there are two types of simulation: (i) virtual, composed of virtual nodes only and (ii) hybrid, composed of virtual and physical nodes. For virtual simulation, the Global Clock is internally defined while hybrid simulation is performed based on the physical node's clock. The WiNeS API contains functions for defining the internal clock based on the external one. Internally, the simulation clock is updated asynchronously and periodically to avoid clock skewing.

(iii) Core

The **Core** performs all computational logic necessary for the communications between nodes. Figure 6 depicts the **Core** architecture composed of (i) a thread for receiving data from the nodes (**Server Socket Thread**); (ii) a set of threads (**Processing Threads**) that process all received packets and determine to which nodes the data should be forwarded when generating event logs; (iii) a thread for sending data to the nodes (**Client Socket Thread**); and (iv) a **Connections Table**, composed of simulated node information. Additionally, the **Core**, if not previously specified, automatically assigns a port number to the nodes.

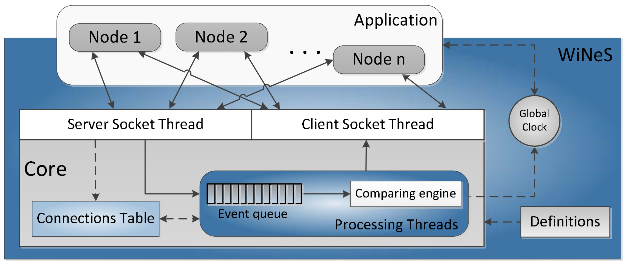

Figure 6. Core architecture.

In more detail, the **Server Socket Thread** component waits for messages sent from the application nodes. If the **Core** receives a request message from a new node in the network, it should: (i) verify if that information is valid and interpretable; (ii) add the node in the **Connections Table** and (iii) reply with a message of acknowledgment. However, if the information is not valid, the **Core** replies with an error message. In the case, the **Core** receives any other type, the message will be inserted into an **Event queue** to be treated by the **Processing Threads** component. **Processing Threads** get the first element of **Event queue** and send it to the **Comparing engine**, where the forwarding decision is made.

The **Connections Table** comprises a list of nodes containing: (i) NodeID: unique identifier generated by the Core; (ii) IpAddress: TCP/IP network address; (iii) PortNum: TCP port on the network; (iv) Parameters: list of node's characteristics according to <COMPONENT_SPECIFICATION>; (v) Timeslot: variable storing the clock time received from the last packet of the corresponding node, which is used to control the time difference between consecutive packets; and (vi) Coordinates (X, Y, Z) with the node's geographic positions.

When a node sends a packet to the **Core**, it automatically requests its inclusion. The **Core** then creates a new entry to the **Connections Table** and stores the new node's IP address and port number, along with other relevant data provided by the node. After being registered in the **Connection Table**, the node becomes visible to other nodes, and capable of exchanging data on the simulation environment.

B. Message Protocol

The WiNeS package format consists of five fields. The **packet type** field specifies the message's nature: (i) ENTRY_REQUEST is used when a node requires entering in the simulation; (ii) ENTRY_ANSWER is used in response to an ENTRY_REQUEST message; (iii) EXIT_REQUEST is used when a node wants to leave the simulation; (iv) SEND_DATA is used for all elements of WiNeS to send generic data; and (v) END_SIMULATION is used by the **Core** to inform the nodes that the simulation ended, releasing all of them. ENTRY_REQUEST and EXIT_REQUEST allow dynamically insertion and removal of nodes on/from the simulation environment.

The **device type** field indicates the type of node that originates the packet, i.e. the Core or a node identification.

The **nodeId** field is the element identifier, marking the node that is sending the packet. If the Core is sending the packet, this field is set to '0'; otherwise, it is set to NodeID, which is the identification number of the node that originated the packet. The **timeslot** field contains the packet generation time, obtained from the Global Clock. The **payload** is a generic field used to store any kind of data originated from the device type related to the packet type.

The **payload** format is categorized according to the following operations: (i) ENTRY_REQUEST: the node must send a TCP port ('0' if none), its **Specification table**, and its coordinates; (ii) ENTRY_ANSWER: the **Core** must confirm the requested port number from the node, or send the automatically generated port number and NodeID; (iii) EXIT_REQUEST: the node must send its NodeID to the Core to allow its exclusion from the Connections Table; and (iv) SEND_DATA: indicates a generic payload that depends on the device type field.

C. *Simulation Operation*

When the simulation starts, the mapping of all devices' specifications occurs either from the supplied XML definitions' file or using the parsed information originated from the API. Each device is then instantiated as a node, based on these definitions, and registered on the **Connections Table** in the **Core**. In this table, a specific port number is assigned for each registered node for later communications; note that a heterogeneous node is recognized as a single node by the simulation engine. These activities, along with other events that occur during simulation execution, are logged into text files for later analysis. The **Connections Table** also stores additional information about each connected node such as operating frequencies, protocols and geographical coordinates. The information is used by the **Core** to determine possible communications between simulated nodes.

Figure 7 depicts the basic simulation flow. When a device intends to join the network, it should send an ENTRY_REQUEST message with information for its Specifications Table to the Core. Afterwards the node waits for an ENTRY_ANSWER response from the Core. Meanwhile, the Core allocates resources for node communication, generates a unique NodeID, and updates the Connections Table. Then, the Core sends the response to the node, with an ENTRY_ANSWER message, informing the attributed NodeID and TCP port number, and the node is able to send and receive messages. At any time, the Core is awaiting data. When any packet, with the exception of ENTRY_REQUEST, is received by the Core the packet is placed in the Event queue to be processed by the Processing Threads (Figure 6).

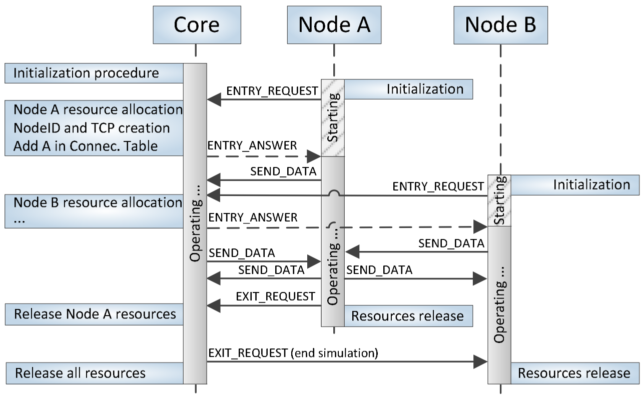

Figure 7. Message chart diagram for the Core operation.

Parallel to any procedure, the Core always checks if the Event queue still contains events. If there is anything to be processed, the **Core** removes the event from the queue, and processes it in the Comparing engine. During simulation, a log of every communication is generated and when the simulation time ran out, the **Core** generates a final log entry with general statistics, e.g. the data exchanged in the simulation.

D. *Comparing engine*

The Event queue contains all SEND_DATA messages originated from the nodes. The **Comparing engine** processes and analyses each message at a time, defining which node has the technological capacity (i.e. devices with compatible frequencies and protocols) and geographic location to receive each message. The Comparing engine, aware of this message, processes the following steps: (i) unpack the SEND_DATA message to recognize the NodeID, the global clock, and payload; (ii) find the corresponding node (e.g. N1) in the Connections Table and get its Specification Table data using its NodeID, (iii) look at all the nodes in the Connections Table. For each different node in the table (e.g. N2), the engine compares the specifications with N1 and evaluates if both nodes are within communication range. For each successful comparison, the **Core** generates a packet with the same payload received from N1, forwarding the message to N2. For a comparison to succeed, the nodes must: (i) use the same communication protocols; (ii) operate at the same frequency; (iii) have a distance between them smaller than or equal to the maximum distance informed in the Definitions.

Finally, when the simulation ends, the Core sends an EXIT_REQUEST for all nodes, and WiNeS finalizes all open sockets and threads, exiting the simulation.

III. MODELING EXAMPLES

Some tests were performed to demonstrate WiNeS functionalities for virtual and hybrid networks simulation.

A. *Heterogeneous nodes*

Figure 8 presents a star topology with: seven passive RFID tags, a wireless sensor node, and a central node. The central node represents a heterogeneous sensor/RFID reader with the ability to communicate with RFID tags and sensors. This example demonstrates that WiNeS describes and simulates message exchanges between heterogeneous nodes. For this application, a wireless sensor node (NodeID=100 and 3D coordinates (0,1,0) requires the values contained in the passive tags. A heterogeneous node located at the central position, which is able to communicate with all nodes, accomplishes the data acquisition. It is important to note that even if all nodes are within communication range, the simulation engine ensures that the wireless sensor is unable to directly read passive tags, because (i) the sensor node is not intended to be a reader, and (ii) the node respects the hierarchy defined by star topology.

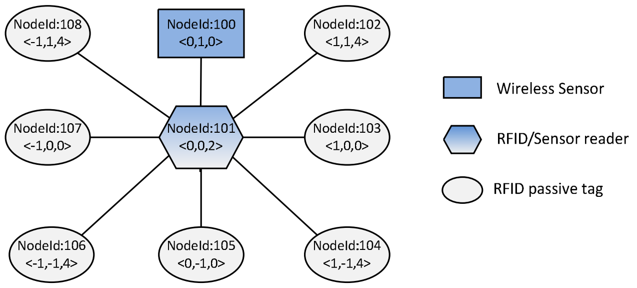

Figure 8. Heterogeneous nodes in a star topology

Each node has two types of address: (i) physical address, which is an XYZ Cartesian coordinate to place the node on the environment; and (ii) logical address, which is the ID used by WiNeS to communicate all nodes within the system. Thus, the simulation scenario instantiation is guided by the following specifications: (i) creation of three types of devices (wireless sensor, RFID/sensor reader and RFID tag) on the Application Layer; (ii) the wireless sensor node (NodeID=100) requires the reading of all values contained in the RFID tags; (iii) the RFID/sensor reader node operates on two different frequencies (heterogeneous component specification); and (iv) the seven passive RFID tags have different high level identifications (ID) and they cannot start any communication alone; all of them wait for reader's requests. Figure 9 shows the Java specification.

```
public class Star {
    public static void main(String[] args) {
  // Simulation parameters
        SimulatorSpecification specs = new SimulatorSpecification(15,
                TIMESCALE.SECONDS, DISTANCE.KILOMETER, "");
        specs.addRule("2.4GHz", "sensor", "sensor", 2);
        specs.addRule("433KHz", "reader", "passiveTag", 2);
        WiNeS sim = new WiNeS(6000, specs, LOG);
  // Sensor node
        NodeSpecification sensorSpecification = new NodeSpecification();
        sensorSpecification.setCoordinates(0.0, 1.0, 0.0);
        sensorSpecification.setNodeType(0);
        sensorSpecification.addComponent("sensor", "ZigBee", "2.4GHz", "1");
        sensorSpecification.addBehavior("EVENT", "3", "DATA");
        sensorSpecification.addBehavior("EVENT", "11", "DATA");
  // Sensor node / reader
        NodeSpecification sensorReaderSpecification = new NodeSpecification()
        sensorReaderSpecification.setCoordinates(0.0, 0.0, 0.0);
        sensorReaderSpecification.setNodeType(0);
        sensorReaderSpecification.addComponent("sensor","ZigBee","2.4GHz","1"
        sensorReaderSpecification.addComponent("reader","ZigBee","433Khz","1"
  // Nodes insertion
        sim.addNode(sensorSpecification, new Sensor());
        sim.addNode(sensorReaderSpecification, new SensorReader());
          // Passive nodes
        for(int y = -1; y < 2; y++) {
            for(int x = -1; x < 2; x++) {
                if((y == 0 && x == 0) || (y == 1 && x == 0))
                    continue;
                NodeSpecification tag = new NodeSpecification();
                tag.setCoordinates(x, y, 0.0);
                tag.setNodeType(1);
                tag.addComponent("passiveTag", "ZigBee", "433KHz", "1");
                tag.addBehavior("DATA", "0", "5");
                tag.addBehavior("DATA", "5", "8");
                tag.addBehavior("DATA", "10", "9.3");
                sim.addNode(tag, new PassiveTag());
            }
        }
        sim.start();
    }
}
```

Figure 9. Example of *heterogeneous system* specification on WiNeS.

Figure 10 illustrates the network traffic related to RFID tags reading data in a star topology. When an information request packet is received from the wireless sensor (REQUEST_DATA message), the central node (NodeID=101) forwards the message as a new type of package, which is understandable only for the RFID tags. The reverse process also occurs; the packets originated from RFID tags are transformed into messages recognizable by the wireless sensor. The RFID tags reply to the central node's message REQUEST_DATA with a DATA message (which contains their ID), while the central node forwards DATA to the wireless sensor node. Remark that the nodes are also definable as either static or mobile. In both cases the initial coordinates must be informed in the Application Layer. During the simulation mobile node coordinates are updated by the Core according to the predefined mobility function.

```
Source:100 Target:101 Packet: [REQUEST_DATA 102]
Source:101 Target:102 Packet: [Node:100 REQUEST_DATA]
Source:102 Target:101 Packet: [DATA 100 ID=334.332.789]
Source:101 Target:100 Packet: [DATA ID=334.332.789 Node:102]
Source:100 Target:101 Packet: [REQUEST_DATA 103]
Source:101 Target:103 Packet: [Node:100 REQUEST_DATA]
Source:103 Target:101 Packet: [DATA 100 ID=334.332.790]
Source:101 Target:100 Packet: [DATA ID=334.332.790 Node:103]
…
Source:100 Target:101 Packet: [REQUEST_DATA 108]
Source:101 Target:108 Packet: [Node:100 REQUEST_DATA]
Source:108 Target:101 Packet: [DATA 100 ID=334.332.795]
Source:101 Target:100 Packet: [DATA ID=334.332.795 Node:108]
```

Figure 10. Example of event log for testing star topology.

B. *Heterogeneous topology*

Figure 11 illustrates an example of multiple mesh networks linked by a ring network, i.e. an example of heterogeneous WSN (Wireless Sensor Network). The ellipses are the RFD nodes (Reduced Function Device of IEEE.802.15.4 protocol) composing the lowest network level. The rectangular nodes represent the mesh nodes coordinators called FFD (Full Function Device) that link all WSNs, establishing a highest level in a ring topology.

This example shows the simulator's capability to enable the node definition with functionalities in hybrid topologies. On top of the IEEE.802.15.4 protocol there is, for example, a hierarchical protocol enabling node communication. Each network level has a node coordinator that receives packets on the mesh, and forwards them to the next ring coordinator. The mesh routing algorithm is XY; thus, the message goes primarily moving horizontally left and then vertically upward to reach the coordinator. Each node has a hierarchical addressing with XY coordinates for internal mesh routing, and an ID for inter mesh routing throughout the ring topology.

Since all nodes have similar information, it is sufficient to create user defined nodes classes, such as GridNode, that extends the Abstract node layer of WiNeS, and thereby becoming an Application Layer. Above the GridNode class, the RFD and FFD class are added. Therefore, the classes RFD and FFD are aware of their hierarchical addresses.

The Default node layer, together with the GridNode, was used for traffic generation. The classes RFD and FFD implement routing, while the Default node layer initializes the packets generation. After the simulation's execution, the node communications can be analyzed in a log file. In this method, the date and local time of the machine, the simulation timeslot, and the message itself are stored.

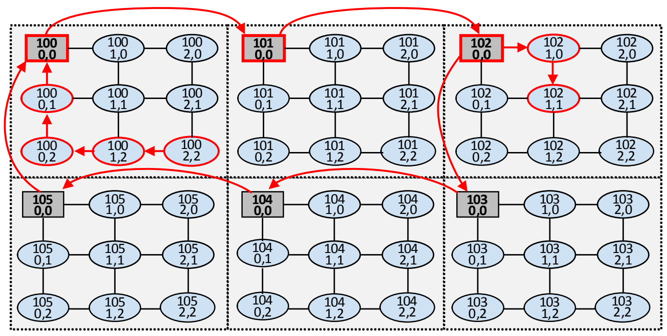

Figure 11. WSN with heterogeneous topology.

As an example, Figure 12 presents a fragment of a simulation log, which could be used to analyze the nodes communication with the exchanged messages path. Note that the message: (<102, 1, 1> 5ºC <100, 2, 2> 10:42) means that at 10:42 the node <100, 2, 2> (i.e. ID=100, X=2,Y=2) detected 5ºC of temperature and tries to send this information to the node <102, 1, 1> (i.e. ID=102, X=1,Y=1). The message originated at node <100, 2, 2> is transmitted through the nodes following XY routing, aiming to reach the coordinator node (FFD <100, 0, 0>). Then, this message arrives in the ring topology and passes by FFD nodes 100, 101 and 102. When the message reaches FFD <102, 0, 0> it starts an XY routing protocol on the target mesh until it reaches the target node.

```
<100, 2, 2> sent <100, 1, 2> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
<100, 1, 2> sent <100, 0, 2> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
<100, 0, 2> sent <100, 0, 1> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
<100, 0, 1> sent <100, 0, 0> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
<100, 0, 0> sent <101, 0, 0> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
<101, 0, 0> sent <102, 0, 0> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
<102, 0, 0> sent <102, 1, 0> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
<102, 1, 0> sent <102, 1, 1> (<102, 1, 1> 5ºC <100, 2, 2> 10:42)
```
Figure 12. Example of event log for heterogeneous topology.

C. *Hybrid simulation scenario*

This section explains the capability of WiNeS to perform simulations integrating physical nodes (NodeP) with virtual nodes (NodeV). Specific APIs for network communication and node definition are able to create the hybrid environment. Figure 13 demonstrates the additional software layer (Physical) to provide this hybrid vision. The Virtual layer corresponds to the Application layer from Figure 2.

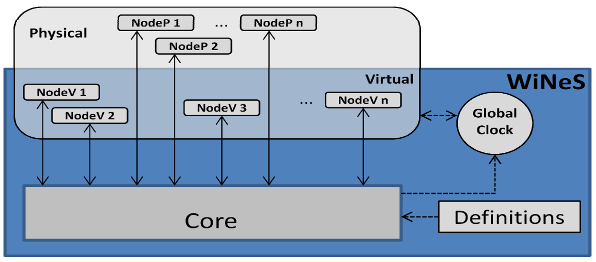

Figure 13. Hybrid simulation with virtual and physical nodes.

In this hybrid model one can include the modeling of ARE to stimulate the virtual or physical network with unforeseen events. The WiNeS framework provides an API to represent ARE behaviors. Suppose a wired network, for example in an industrial environment, where a wireless gateway can add more coverage to the environment. WiNeS allows the emulation of these gateways functioning without necessarily having all physical nodes in the physical environment. This means that one is capable of creating a simulation environment able to connect physical nodes and synchronize them within simulation environment. The most important component to read is the Global Clock followed by the implementation of the remaining specifications and customized APIs.

Designers interested in pursuing this direction are given a flexible framework to introduce their own Java code able to communicate physical and virtual nodes, providing node clock synchronization based on Global Clock and ways to interpret messages from both types of nodes.

IV. Related Works

Simulators can be classified by a variety of aspects, such as their support related to network topologies and node types, scalability, development platform, model of computation, programming language, exploration of current trends in performance optimization [5], [10], among others.

Simple simulators usually involve the simulation of different network topologies, deployment scenarios and data traffic between the simulated nodes only, while complex simulators are capable of simulating network protocols and thereby provide the development framework for customized simulation environments. WiNeS framework aims to be a simple network simulator with a few complex characteristics, for example, being a development framework.

Currently, there are open source simulators such as [4], [5], [6], [7] and [8]. Moreover, there are many other simulators, emulators and test beds available in the literature covering a multitude of purposes [11]. However, in the following only a selection of the well-known network simulators is described, highlighting their purposes and available features.

Table 1 displays an overview of the key features of five network simulators, along with WiNeS itself. Remark that scalability is not shown comparatively, but we acknowledge that it is an important issue to analyze in further experiments.

V. Conclusion

The technological trend of forming heterogeneous wireless networks creates a need to design flexible simulators. The main goal is saving time for these networks' deployment, while allowing the evaluation of performance and costs during the design phase. Nevertheless, WiNeS framework was developed to be a simulation system for wireless networks enabling flexible scenarios design within a simple API. The modeling examples showed possibilities and features provided by WiNeS, highlighting that the framework contains mechanisms to verify node communication rules during simulation. Moreover, the heterogeneous elements simulation example shows that WiNeS is able to simulate new emerging trends such as the combined use of wireless technologies.

However, WiNeS is susceptible to some disadvantages due to its flexibility in modeling wireless networks and some design approaches. The following downsides were identified: (i) The operating system is able to accommodate only a limited number of open sockets; (ii) the number of efficiently handled threads is limited; It is important to remember that in a network scenario with a thousand nodes, the framework faces

3,000 threads to simulate these nodes, 1,000 client sockets, and 1,000 server sockets. (iii) The overall simulation performance dependents on the nodes' operation complexity.

To counterweight the indicated problems, WiNeS presents the following advantages: (i) The project was developed in Java, which is multiplatform. (ii) It allows defining simulation scenarios with a high degree of flexibility, including predefined templates for nodes description, with heterogeneous characteristics. (iii) It allows the inclusion of new modules in Java, and (iv) the simulation of wired networks, if necessary, only ignoring environmental settings such as the distance verification between communicating nodes, since connection is already established through wire. Comparing WiNeS with major simulators, we conclude that all tools intend to facilitate the creation of communication network topologies, exploiting different levels in protocols modeling, allowing a flexible scalability. The designer should rely on the most appropriate simulator for their specific applications and evaluations, taking into consideration relevant aspects such as the simulator's scalability and its provided libraries. A robust simulator with a wide range of devices readily available could shorten deployment times; however, if scalability is lacking, this could become a key determinant in choosing the most appropriate simulator.

Acknowledgment

This work is partially funded by FAPERGS Docfix (SPI n.2843-25.51/12-3) and PqG 12/1777-4. Financial support also granted by CAPES AEX 5967-13/9, CNPQ and FAPESP to the INCT-SEC (National Institute of Science and Technology Embedded Critical Systems Brazil), processes 573963/2008-8 and 08/57870-9.

References

[1] Y.-S. Chen; Y.-W. Lin; C.-Y. Chang. An overlapping communication protocol using improved time-slot leasing for Bluetooth WPANs. Journal of Network and Computer Applications, v. 32, n. 1, pp. 273-292, Jan. 2009.

[2] Institute of Electrical and Electronics Engineers, Inc. Std. 802.11 IEEE - 1999 ed.: Wireless LAN Medium Access Control (MAC) and Physical Layer (LHY) Specification, IEEE Press, 90p, 1999.

[3] S. Farahani. ZigBee and IEEE 802.15.4 Protocol Layers. ZigBee Wireless Networks and Transceivers, Chapter 3, pp. 33-135, 2008.

[4] NS-2. The Network Simulator ns-2: Documentation. Captured on www.isi.edu/nsnam/ns/ns-documentation, Sep. 2012.

[5] NS-3. ns-3 Network Simulator. Captured on www.nsnam.org/ns-3-15/documentation, Sep. 2012.

[6] J. Chen et al. The Development of a Realistic Simulation Framework with OMNeT++. Conference on Future Generation Communication and Networking, pp. 497-500, 2008.

[7] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, M. Gerla. GloMoSim: A Scalable Network Simulation Environment. UCLA Computer Science Department Technical Report, Vol. 990027, 1999.

[8] R. Barr, Z. J. Haas, R. van Renesse. JiST: Embedding Simulation Time into a Virtual Machine. 5th EUROSIM Congress on Modeling and Simulation, pp.1-16, 2004.

[9] E. Weingartner, H. vom Lehn; K. Wehrle. A Performance Comparison of Recent Network Simulators. IEEE International Conference on Communications, ICC '09, pp.1-5, 2009.

[10] J. Font et al. Analysis of source code metrics from ns-2 and ns-3 network simulators. Simulation Modelling Practice and Theory, v. 19, n. 5, pp. 1330-1346, May 2011.

[11] M. Imran, A. Md Said, H. Hasbullah. A Survey of Simulators, Emulators and Testbeds for Wireless Sensor Networks. International Symposium in Information Technology, pp. 897-902, 2010.

[12] Edwards, S. ; Lavagno, L. ; Lee, E.A. ; Sangiovanni-Vincentelli, A. Design of embedded systems: formal models, validation, and synthesis. Proceedings of the IEEE, v. 85, n. 3, pp. 366-390, Mar. 1997.

Table 1. Some networks simulators characteristics and comparisons.

Simulator	MoC	NC; SC; PT; PR	General Characteristics
ns-2	*DE*	NC: Wired/Wireless; SC: Limited (nodes are objects, thus when instantiating it creates a large number of dependencies to be checked – memory usage and simulation runtime); PT: Linux, FreeBSD, Mac OS; PR: C++, OTcl.	Event scheduler runs independently from the simulation control system, facilitating the customization of different events; predefined simulation components; Scripts in OTcl configure simulation environment, initialize event scheduler and trigger data traffic sources; numerous protocols models and traffic generators available; lacks of available customized wireless network models. The experiments show that ns-2 scale up to 3 thousand nodes.
ns-3	*DE*	NC: Wired/Wireless; SC: Limited (Higher if using virtualization); PT: Linux, FreeBSD, Mac OS; PR: C++, Python.	Models with no compatibility with ns-2; pre-implemented devices; customized wireless networks; support the integration of actual implementation codes providing standard APIs.
OMNet++	*DE*	NC: Wired/Wireless; SC: Limited; PT: Windows, Mac OS, Linux; PR: C++.	General purpose DES framework not specific for network simulation; kernel library for creating new modules; provides Internet protocol models; facilitates the simulation of ad hoc networks and wireless sensor networks; supports the specification of variable parameters in the network description, such as number of nodes can be dynamic.
GloMoSim	*Parallel DE*	NC: Wireless; SC: High (parallel simulation, node and layer aggregations); PT: Windows, Linux; PR: PARSEC (extension of C).	Focuses on mobile wireless devices, providing two mobility models; 2D plane; aims at very large network models; layered approach as OSI model; communication based on nodes distance; simulates only IEEE 802.11 protocol; packet collision analysis; simple APIs.
JiST/ SWANS	*DE*	NC: Wireless; SC: High (claims to scale networks of wireless networks with better performance than ns-2 and GloMoSim); PT: Any platform; PR: Java.	Embeds simulation engine on the bytecodes; lacks of enough protocol models; provides ad hoc network simulator; allows writing simulation on a known programming language; objects communicate by passing messages (represented by object method invocation); simulation events are method invocations; - eliminates the need of explicit event simulation event queue.
WiNeS	*DE SRE ARE*	NC: Wired/Wireless; SC: High; PT: Any platform; PR: Java.	Uses XML to describe simulation environment (e.g. nodes, parameters), allows writing simulation on a known programming language; shares a few similarities with ns-2 such as the event scheduler that maintains synchronized the simulated devices with the simulation clock, and each device has access to the event scheduler; is very similar to the internal architecture of GloMoSim, using the same concept of a network with many nodes running in parallel; enables physical nodes within simulation core control, thus allowing hybrid simulation environment. Experimental results show that WiNeS supports up to 10 thousand nodes simulation, each node transmitting around of 100 packets simultaneously.

Legend: MoC – Model of Computation; DE – Discrete Event; SRE – Synchronous Reactive; ARE - asynchronous Reactive; Network connections type (NC); Scalability (SC); Platform (PT) and Programming language (PR).