Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 369178, 16 pages
doi:10.1155/2008/369178

Research Article

Power Aware Simulation Framework for Wireless Sensor Networks and Nodes

Johann Glaser, Daniel Weber, Sajjad A. Madani, and Stefan Mahlknecht

Institute of Computer Technology, Technical University of Vienna, 1040 Wien, Austria

Correspondence should be addressed to Johann Glaser, glaser@ict.tuwien.ac.at

Received 1 October 2007; Revised 21 February 2008; Accepted 16 May 2008

Recommended by Sandeep Shukla

The constrained resources of sensor nodes limit analytical techniques and cost-time factors limit test beds to study wireless sensor networks (WSNs). Consequently, simulation becomes an essential tool to evaluate such systems.We present the power aware wireless sensors (PAWiS) simulation framework that supports design and simulation of wireless sensor networks and nodes. The framework emphasizes power consumption capturing and hence the identification of inefficiencies in various hardware and software modules of the systems. These modules include all layers of the communication system, the targeted class of application itself, the power supply and energy management, the central processing unit (CPU), and the sensor-actuator interface. The modular design makes it possible to simulate heterogeneous systems. PAWiS is an OMNeT++ based discrete event simulator written in C++. It captures the node internals (modules) as well as the node surroundings (network, environment) and provides specific features critical to WSNs like capturing power consumption at various levels of granularity, support for mobility, and environmental dynamics as well as the simulation of timing effects. A module library with standardized interfaces and a power analysis tool have been developed to support the design and analysis of simulation models. The performance of the PAWiS simulator is comparable with other simulation environments.

1. INTRODUCTION

The advances in distributed computing and micro-electro-mechanical systems (MEMSs) have fueled the development of smart environments powered by wireless sensor networks (WSNs). WSNs face challenges like limited energy, memory, and processing power and require detailed study before deploying them in the real world. Analytical techniques, simulations, and test beds can be used to study WSNs. Though analytical modeling provides a quick insight to study WSNs, it fails to give realistic results because of WSN-specific constraints like limited energy and the sheer number of sensor nodes. Real world implementations and test beds are the most accurate method to verify the concepts but are restricted by costs, effort, and time factors. Simulations provide a good approximation to verify different schemes and applications developed for WSNs at low cost and in less time. The available simulation frameworks are either general purpose or WSN specific. The general purpose network simulators do not address WSN specific unique characteristics while WSN specific simulators mostly lack the capability of capturing and analyzing the power consumption and timing issues at the desired level of granularity.

The proposed PAWiS simulation framework [1, 2] assists in developing, modeling, simulating, and optimizing WSN nodes and networking protocols. It particularly supports detailed power reporting and modeling of wireless environments. A typical WSN node may comprise various types of sensors (e.g., temperature, humidity, strain gage, pressure), a central processing unit (CPU) with peripherals, and a radio transceiver. The simulation covers the internal structure of these nodes as well as communication among them. Sensor nodes forming a network communicate with each other via an ad hoc multihop network. The range of applications that can be simulated covers many domains such as building automation, car-interior devices, car-to-car communication systems, container monitoring and tracking, and environmental surveillance.

The PAWiS simulation framework provides a way to reduce the overall power consumption by carefully optimizing various design aspects within the context of the application. Enhancing energy performance could propel

many new applications since the lack of sufficient battery lifetime or limited energy scavenging systems is still the main cause for the slowly spreading number of WSN applications.

In previous research performed at the Vienna University of Technology [3], several weaknesses of current WSN nodes were identified. These include the wakeup problem (i.e., how to wakeup a sleeping node), the voltage matching and power supply problem, the fairly long oscillator start-up time, and other hardware related problems. However, overall efficiency also strongly depends upon the application and its interaction with other nodes and the environment. Here, communication protocols play an important role, but considering the different layers of protocols independently and not taking into account adjacent layers as well as the hardware and environment, improvements can only be suboptimal [4]. PAWiS explicitly supports cross-layer design to exploit the synergy between layers. Several aspects regarding power aware wireless sensors are emphasized and directly supported by the PAWiS framework. The PAWiS framework hence helps to capture the whole system in one simulation and extracts power consumption figures from software and hardware modules uncovering leakages in early design stages. The main contributions of this work are as follows.

(i) One of the main contributions is to equip the user to program models of a wide variety of abstraction,

(ii) Another contribution is to model the internals of WSN nodes as well as the communication between them. The framework distinguishes between software and hardware tasks, yet it is easy to change the hardware/software partitioning,

(iii) one of the main contributions also is an elaborate power simulation with any level of accuracy which can still be balanced with complexity. The simulated power consumption can depend on the supply voltage, for example, for a nearly empty battery when supplying a microcontroller that operates at very low voltages. In WSN nodes, components with different supply voltages are combined resulting in the need for low dropout regulators (LDOs) and DC/DC converters. The PAWiS framework allows to model this hierarchical supply structure as well as the efficiency factor of the converters.

(iv) Powerful analysis and visualization techniques are provided to evaluate the simulation results and derive a path to optimization.

(v) The RF communication is modeled according to real-world wave propagation phenomena while still maintaining an efficient simulation. It includes interferers, noise, and attenuation due to distance to influence the bit error ratio of communication links. No preset topology is required because the packets are transmitted to all nodes within reach. The topology of network communication itself originates from the link quality and the routing algorithm. With this approach any routing protocol, especially ad hoc protocols, can be implemented. The transmission model implementation is entirely independent of the underlying modulation format enabling the simulation of any type of modulation. Multiple participants can utilize the RF channel by multiple access schemes and are separated by space, time, frequency, and code.

2. RELATED WORK

To have credible results through simulation, the choice of models and the simulation environment is very important. Key properties for WSN simulators must include a way to capture energy consumption at any level of abstraction, powerful scripting language, graphical user interface (GUI) support to animate, trace, and debug, and the ease to integrate new modules. Some of these key properties are discussed in [5].

NS-2 is a discrete event, object-oriented, general purpose network simulator written in C++ (http://www.isi.edu/nsnam/ns/). According to [6], it is the most widely used simulator and has a rich library of protocols but focuses mainly on IP networks. OTcl [7] is used as scripting language to control and configure simulations. It provides a GUI support with the network animator (Nam) which is not so good and only reproduces NS-2 trace [5]. For WSN, NS-2 does not scale well, and it is difficult to simulate 100+ nodes [8]. NS-2 lacks detailed support to measure the energy utilization of different hardware, software, and firmware components of a WSN node. "*One of the problems of ns2 is its object-oriented design that introduces much unnecessary interdependence between modules. Such interdependence sometimes makes the addition of new protocol models extremely difficult, which can only be mastered by those who have intimate familiarity with the simulator*" [9].

SensorSim [10] is an NS-2-based simulator for modeling sensor networks. The authors have provided a power model, a battery model, and a CPU model to address sensor network specific constraints but because of the "*unfinished nature of the software,*" the simulator is no longer available (http://www.nesl.ee.ucla.edu/projects/sensorsim/).

OMNeT++ [11] is a discrete event, component based, general purpose, public source, modular simulation framework written in C++ (http://www.omnetpp.org/). It provides a strong GUI support for animation and debugging. The mobility framework (MF) [12] for OMNeT++ is a specific purpose addon to simulate ad-hoc networks. In the MF, the links between pairs of neighbor nodes are specified with OMNeT++ gates with the help of an additional global module called channel control. Unlike "*visible communication paths*" which are created and freed in bulk, the air module of the PAWiS framework (see Section 3.3.7) decides about connection between pairs of nodes dynamically based solely on their position information and other radio transmission effects. Another difference is the capturing of power consumption with the power meter (see Section 3.3.8) that helps to identify major energy consuming modules. OMNeT++/MF does not provide a WSN specific module library [13] which can help expedite the design process.

SenSim [14] is an OMNeT++ based simulation framework for WSN. Some protocol layers and hardware units are implemented as simple OMNeT++ modules. They have provided implementations for different WSN specific protocols, battery, a simple CPU implementation, a simple radio, and a wireless channel. A coordinator module is implemented to assist in inter-communication between hardware and software modules. It does not add additional functionality to OMNeT++ other than a simulation template with a small number implemented modules.

NesCT (http://www.nesct.sourceforge.net/) is an add-on for OMNeT++ which allows the simulation of TinyOS-based sensor networks in OMNeT++ (http://www.tinyos.net/). It does not come with any additional functionality but acts as a language translator between two different environments (TinyOS and OMNeT++).

Global mobile information system simulator (GlomoSim) [15] is a library-based general purpose, parallel simulator for wired and wireless networks written in Parsec (http://www.pcl.cs.ucla.edu/projects/parsec/) (C-based discrete event simulation language for parallel programming). Being parallel, it is highly scalable and can simulate up to 10 000 nodes [5]. Using GlomoSim requires learning the new language Parsec. GlomoSim is superseded by QualNet (http://www.scalable-networks.com), a commercial network simulator and is not released with updated versions since 2000. However, sQualNet [16], an evaluation framework for sensor networks, built on the top of QualNet has been released recently.

OPNet Modeler is a commercial, well-established (1986), general purpose, object oriented simulation environment written in C++ (http://www.opnet.com). It supports discrete event, hybrid, and analytical simulation. It provides a very rich set of modules for all layers of protocol stacks including the IEEE 802.11 family, IEEE 802.15.4, and routing protocols like AODV [17], and DSR [18]. Each level of the protocol stack has to be implemented as a state machine but it is *"difficult to abstract such a state machine from a pseudo-coded algorithm"* [19]. The authors in [19] compared OPNet Modeler, GlomoSim, and NS-2 with a broadcast protocol. The results show that the performance of NS-2 and GlomoSim, and OPNet is barely comparable.

SENSE [9] is a sensor network specific, component-based simulator written in C++ built on the top of COST [20]. To address the issue of scalability, SENSE provides an optional way for parallel simulation as is done in GlomoSim. It provides a small library of module implementations like AODV and DSR, with simplistic battery and power models; but unlike PAWiS, it does not provide a detailed structure to capture energy consumption of different hardware and software components. It also lacks a visualization tool which is helpful in debugging and visual inspection.

Ptolemy II [21] is a component assembly-based software package to study concurrent, real time, and heterogeneous embedded systems and is written in Java. Ptolemy II provides a rich support to model, simulate, and design components in different domains (e.g., discrete time or component interaction), but according to recent Ptolemy II 7.0.beta release notes, its wireless domain is still in experimental phase (http://www.ptolemy.berkeley.edu/ptolemyII/). VisualSense [22] is an open source WSN visual simulation framework built on Ptolemy II.

J-Sim [23] is a general purpose, component-based, open-source simulation framework written in Java. It is glued to different scripting languages with TCL/Java and hence is a dual language framework like NS-2. Initially, designed for wired networks, its WSN specific package provision supports only 802.11 MAC scheme and some high-level models, for example, battery, CPU, wireless channel, and sensor channel.

Various WSN specific simulation and emulation tools have been released in the previous years. These tools include TOSSIM [24], EmStar [25], and ATEMU [26]. The advantage of using such tools is that the code that is used for simulation/emulation also runs on the real node (with minor modifications) reducing the effort to rewrite the code for the sensor node and giving detailed information about resource utilization. The main problem with such frameworks is that *"the user is tied to a single platform either software or hardware (typically MICA motes), and to a single programming language (typically TinyOS/NesC)"* [5]. Tython [27] and PowerTossim [28] are extensions of Tossim to capture the dynamic behavior of environment and power consumption, respectively.

In contrast to many of the above frameworks, the PAWiS simulation framework meets all the key properties outlined in [5]. It also utilizes the powerful GUI support of OMNeT++, it utilizes the widely used scripting language Lua to include environmental dynamics and mobility, and to reduce the test-debug cycle (http://www.lua.org/). It focuses on capturing energy consumption at hardware and software levels, provides a visualization tool to analyze energy consumption, a rich library of modules to get an optimized protocol stack, standardized interfaces to improve reusability, and provides a simulation template for the user to jump start any simulation study.

3. SIMULATION FRAMEWORK

A wireless sensor network is built of independent nodes which communicate via an ad hoc wireless network. The PAWiS simulation framework is designed to model, simulate, and optimize both the wireless communication protocols as well as the interior of the nodes. Each node is built as a virtual prototype in a way that its function, timing, and power consumption as well as system failures are simulated at any level of detailedness.

3.1. Methodology

The design of a WSN and its nodes follows a top-down approach embedded in a cyclic process. The functional specification defines requirements of the WSN which apply to the architecture as well as to the implementation. The implementation on the other hand imposes constraints on the architecture and the functional specifications in a bottom-up manner. For example, the functional specification of a tire pressure monitoring system (TPMS, see also Section 6.2) may require the sensors to measure the current pressure

every 20 seconds. Due to the power consumption of the current sensor technology and the available battery capacity, the implementation of a TPMS sensor node imposes the constraint that this function can only be maintained for 2 months, which is rather short compared to the lifetime of a tire.

3.1.1. Work flow

To design a WSN and its nodes, the functional specification is defined. For a full optimization, the work flow is a cyclic application of the following steps.

(a) Every node typically consists of multiple submodules. In this step, the node structure is defined and for every module type a certain implementation is chosen (*composition*). Initially, the modules only need to meet minimum functional requirements. For instance, for the aforementioned TPMS node, the user chooses a CPU, a pressure sensor, an AD converter, a timer, an RF transmitter, and memory. Additionally, software modules like the network stack and sensor handling are required.

(b) The modules chosen in the previous step are *integrated*. Their interfaces have to be adopted and their functions must be coordinated. For example, the chosen pressure sensor might require special treatment for its power-on sequence, which must be implemented accordingly in the module which operates it.

(c) The modules are *configured*, that include setting values for the clock frequency of the CPU, the resolution of analog-to-digital converters (ADCs), and so forth.

(d) In the previous steps, a fully functional model of a node and of the network was set up and is *simulated* in the current step.

(e) The simulation results are *evaluated*. This includes the verification of the function, analysis of the power consumption and timing, and detection of potential for further optimization (see Section 5).

(f) The issues identified in the previous step are considered for a *refinement* of the models as well as the design. Examples are increasing the detailedness and accuracy of power consumption and timing, dividing the functionality into more elaborate modules, configuration changes, and even a modification of the node composition by exchanging module implementations. The chosen pressure sensor might consume too much energy in every measure cycle due to its long-lasting power-up sequence, and hence should be exchanged by a sensor with faster startup. Another example is the physical layer model (including the RF transceiver) which might need a refinement of the power consumption reporting during intermediate states (e.g., when switching from transmit to receive mode) (see Section 3.3.8). With the refined node implementation, the procedure is started over again, until the optimization goal is achieved.

These *refinement cycles* are the main track to enhance the development and design [3]. After completing the optimization process, the final outcome comprises the verified function, the architecture of the node, the implementation details, and the power specification of every module.

The module library (see Section 4) is particularly intended for the composition and integration of modules to a node. It provides a collection of multiple module implementations for every module type which can be combined in numerous ways. The integration effort is minimized because these modules conform to the interface specification (Section 4.2).

3.1.2. Optimization

Several strategies for the optimization of WSN nodes are proposed in [3]. The PAWiS simulation framework is especially constructed to assist the designer in applying these strategies.

(i) *System-level optimization* involves a modification of the whole system behavior like choosing a different network layout or application patterns.

(ii) Exchanging the *module implementation is done* by selecting a different module from the library. For example, choose a dual-slope, a $\Sigma\Delta$ or an successive approximation ADC. Another example is to change a communication layer implementation (e.g., use another medium access (MAC) protocol).

(iii) Another strategy is exchanging *multiple module implementations* for adjacent modules that tightly work together. While modifying a single module might degrade the node performance, the interaction of the changes of multiple modules potentially leads to an overall improvement.

(iv) *Cross-layer optimization* works on more than one network layer where, for example, modifying the routing protocol benefits from a different physical layer.

(v) *Partitioning* of modules and/or functions is done by dividing the task between hardware and software, digital and analog, or RF and baseband. For example, a specific MAC protocol could be implemented in software, as dedicated hardware acceleration unit, or a combination of both.

(vi) *Another strategy is scaling* a module, for example, the resolution of an ADC or the register count of a CPU.

(vii) *Parameterization* of modules, for example, the timing, transmission power, and bit rate of a radio transceiver.

3.2. Structure

The PAWiS simulation framework is based on the OMNeT++ discrete event simulator [11] which is written in the C++ programming language (Figure 1). A discrete event simulation system operates on the basis of chronological

consecutive events to change a system's state. These events are processed by the simulation kernel. The simulation time itself does not progress continuously but is advanced with each occurring event (hence it is not possible to issue events that are scheduled before the current simulation time). The proposed framework handles timing-related issues according to this discrete event mechanism.

User-defined models are implemented as C++ classes and mostly utilize framework concepts. The user of the framework is only confronted with OMNeT++ to comprehend the simulation process. Node composition and network layout along with environmental and setup parameters are specified in configuration files as well as script files (see Section 3.4). The modules are compiled and linked with the simulation kernel, and result in the simulation application. This offers a GUI-based frontend which enables visual debugging of the communication processes of the model on a per-event basis at simulation runtime. An optional command line-based frontend can be utilized for increased simulation performance.

The framework is primarily focused on simulating inter- and intranode communication. Additionally, fine-grained aspects (e.g., CPU instruction set emulation as used by [26]) can be easily modeled with user extensions. However, a tradeoff between simulation details and execution performance (as discussed in [29]) has to be considered with increasing quantity of network nodes.

A promising feature is the possibility to use SystemC in combination with OMNeT++ (http://www.systemc.org/). This is achieved by combining the OMNeT++ simulation kernel and the SystemC simulation kernel in a way that events from OMNeT++ and SystemC are being processed together. This also allows the communication between both domains. OMNeT++ allows the use of a custom scheduler which is the central point to merge the messages and events of OMNeT++ and SystemC, respectively. Unfortunately, the OSCI SystemC kernel does not offer such an interface, so slight modifications in the source code were necessary.

Simulation results comprise timing and power consumption profiles as well as event records. The completed model itself contains information regarding the functional description and architecture specifications along with low-level implementation details.

3.3. Basic concepts

3.3.1. Modularization

A wireless sensor node is typically composed of multiple *modules* (e.g., CPU, timer, radio, network layers). Internally, every module is based on one or more *tasks*. The framework defines two types of tasks. One type models a *hardware component* (e.g., a timer, an ADC) whereas the second type is a *software task*, for example, application, routing, MAC, and physical layer. Every module is implemented as a C++ class derived from a framework base class. Tasks are implemented as methods within a module class. The execution of a single task is sequential but all active tasks are running in parallel. This form of concurrency is implemented as cooperative multithreading, where the program flow is suspended within a method when certain framework calls are made (e.g., to wait for some condition to be satisfied) and will continue execution after being dispatched again. This process is transparent for the user and is entirely handled by the framework.

Basically, the detailedness and granularity of the sensor node model strongly depend on the design and simulation requirements for hardware and software modules and are not restricted by the PAWiS framework.

3.3.2. Functional interfaces

Control flow transitions between two modules are specified by the so-called *functional interfaces* (FI). They can be thought of as subroutines with well known names and parameter specifications. An invocation of an FI is similar to a blocking subroutine call but may exceed the module boundary. The framework allows the passing of arguments to and from FIs. In the model, FIs are implemented as class methods (similar to tasks).

A collection of FIs grouped together under a well-known name can be thought of as a functional module-type description. This introduces a level of abstraction in the functional design process and hence enables reusability of functional design. Two modules that are completely satisfying the specification regarding their FIs can be said to be functionally equivalent (although they might have entirely different power consumption and timing profiles). This approach is utilized in the module library (see Section 4).

3.3.3. CPU

As already mentioned in Section 3.3.1 tasks can either model hardware or software. Software tasks of sensor nodes are executed by a CPU. It is important to note that multiple software tasks cannot run in parallel, since typically only one CPU is available and supported by the framework. The CPU module of the framework ensures that only one task's code simulation is executed at a time.

To model the power consumption and timing behavior of software tasks, the PAWiS simulation framework splits the simulation into two parts. The functional part is implemented in the C++ method of the task. The timing and power consumption part, on the other hand, is delegated to the CPU module which maintains its power consumption and delays execution of the software task for the calculated processing time (the time that the code execution on the CPU would take). This means that the whole functional part is executed at the very same simulation time instant. The model programmer has to insert special framework requests to the CPU module to simulate the execution time and power consumption.

These requests include the estimated execution time of the firmware code on the CPU. Now think that the CPU of a given node should be replaced during the optimization process. This would also require to modify all execution time estimates in all modules of the node. To allow for a CPU exchange without the need to adapt other modules the

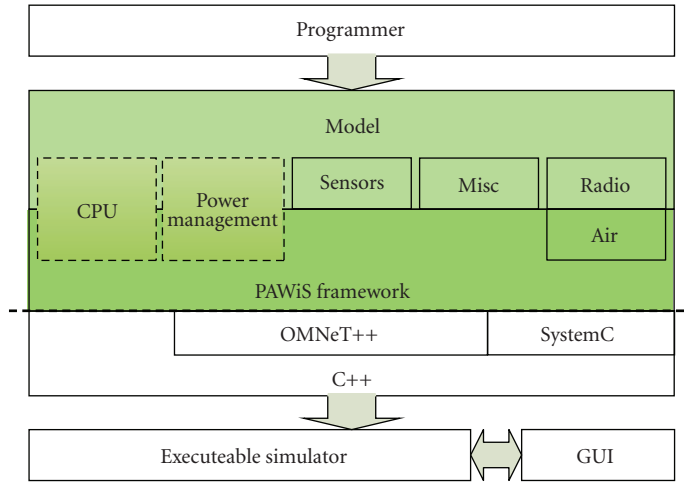

Figure 1: Structure of the PAWiS simulation framework.

execution time estimates are referred to the so-called *norm CPU*. This is an imaginary but well-defined CPU implementation (regarding its performance). The actual CPU model scales its processing time and power consumption according to its individual properties. For higher accuracy, the CPU request also supplies the percentage of integer, floating point, memory access, and flow control operations.

Many microcontrollers used for WSN nodes have CPUs which offer special low-power modes. The PAWiS simulation framework also supports modeling of these states. This is done by pausing code execution and setting the power consumption of the CPU module to a lower value. To exit the low-power mode an interrupt can be issued.

3.3.4. Timing

Modeling time delays differ whether they occur in firmware or hardware modules. For hardware modules the framework provides a simple wait method to suspend execution for a certain amount of time. Several distinct implementations of the wait method are available with support for fixed and conditional timeouts.

Using wait is not valid for software tasks because it is not possible to wait and do nothing in software (even for an infinite loop without body, the CPU does something). In fact, if delays are needed in software, the corresponding module has to use a loop (or a similar construct) to wait for a certain time and therefore utilize the CPU to achieve the delay. The framework offers a variety of methods to utilize the CPU for timing and flow control purposes. Alternatively the CPU can be put to a low-power mode which stops execution too and therefore delays until an external or timer event occurs.

An important consequence of this timing model is that consecutive user code lines without a wait call or a CPU utilization request take place in the same simulation time instant (i.e., no simulation time elapses during that code execution). Simulation time only advances when these special methods are invoked.

3.3.5. Interrupts

The framework provides a basic mechanism to model interrupt handling in a two-step process that maps

(i) interrupt sources to interrupt vectors; and

(ii) interrupt vectors to interrupt service routines.

Whenever an interrupt request is issued, the framework handles the necessary task scheduling according to the interrupt priorities and the currently running task.

The implemented interrupt model supports several user configurable interrupt sources (potentially coming from different modules, e.g., a timer, an analog-digital-converter, etc.). Each of these sources is mapped to an interrupt vector. Additionally, it is possible to map multiple sources to one specific interrupt vector. Furthermore, every vector maintains a priority and an interrupt service routine (ISR). As the model allows multiple vectors to share one ISR, the framework provides means to identify the triggering vector from the ISR. The framework's CPU module entirely handles the interrupt processing except for prioritizing of interrupt vectors that needs to be provided in the user model by overriding the CPU base class.

The user can register ISRs for interrupt vectors within software modules. When interrupt sources trigger interrupt requests, the CPU module determines the appropriate interrupt vector, checks its priority, and if appropriate transfers control to the ISR (which is always a software task). In case of a control transfer, the currently executed CPU task is preempted and continues execution after the ISR has finished.

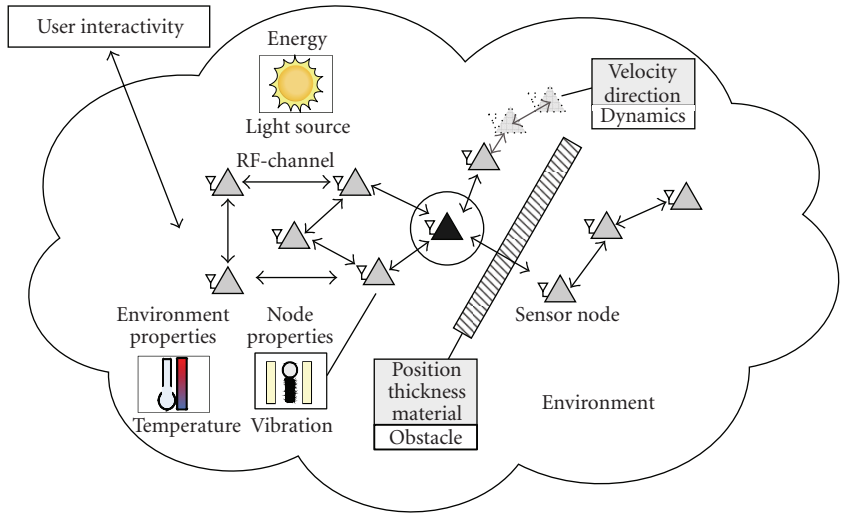

Figure 2: The Environment with properties, objects, and sensor nodes.

3.3.6. Environment

All sensor nodes are placed at 3D positions within the *environment*. This is a representation of the outer world and surroundings of all nodes including the RF channel.

Besides the nodes themselves, additional objects like walls, floors, trees, interferers, heaters, light sources, global properties (e.g., the attenuation exponent b (see Section 3.3.7)), and more are defined within the environment. The entire *environment* can be configured with configuration and scripting files.

3.3.7. Air

The *air* is an essential part of the *environment* to handle the RF channels, which are defined by 3D node placement in space and obstacles between the nodes. A real RF signal is subject to wave propagation phenomenons like attenuation, reflection, refraction, and fading (multipath propagation) from the transmitter to the receiver. In the PAWiS simulation framework, these effects can be modeled but currently we use a distance-based path loss radio model, which only considers the distance between transmitter and receiver.

The packet transmission is modeled without the definition of a predefined topology (similar to a wired network). Instead of that, every RF message is transmitted to all other nodes and the received RF power is calculated from the transmitter power, antenna properties and especially the distance and obstacles between the transmitter and the receiver. The topology of the network results from the reachability between nodes which is limited by the minimum received signal quality.

Signal power

The received signal power of a node is proportional to the transmitter power, only scaled by wave propagation effects and node properties. These constant attenuation factors between all nodes can be conflated to a matrix which is referred to as *adjacency matrix* within the framework. A simple row multiplication is used to calculate the received signal power for all nodes. The matrix is a precisely defined interface from the *Environment* setup (i.e., node positions, obstacles) to the data communication. Therefore, it can also be calculated by an external RF channel simulation tool.

The current implementation of the *Air* supports only isotropic antennas with uniform antenna gain. Obstacles are considered for the adjacency matrix by explicitly given additional attenuation factors between pairs of nodes in the *Environment* configuration.

Packet transmission

Whenever a data packet is transmitted by a node, the *Air* calculates the received signal power for all other nodes and notifies every node (above a certain threshold) about the start of the transmission. The nodes confirm the acceptance of the data packet, if their receiver is currently in listen mode and if the signal power is above the sensitivity threshold. During the transmission, the receiving nodes calculate the signal-to-noise ratio (SNR) between the received signal power P_{signal} and the received and internal noise power P_{noise} : $\text{SNR} = P_{\text{signal}}/P_{\text{noise}}$.

From this SNR, the bit error ratio (BER) is calculated, which is a function of the SNR depending on the (fixed) modulation format (the formula can be provided by the user). From the BER and the bit length of the transmission, the bit error count is calculated. Consequently, the user's module decides whether the received packet is valid or treats it as corrupted.

From the size of the transmitted data packet and the bit rate, the Air calculates the duration for the transmission. At the time when the transmission is finished, the Air notifies all receiving nodes again. This notification contains the user data, its length, and the number of bit errors. Additionally, as some protocols decide whether to process an incoming

packet after the arrival of some header fields, the framework provides a mechanism to stop listening to a transmission after a specified number of bits have arrived.

Collisions

If a node starts to send a packet while another packet is already being transmitted, this second signal is uncorrelated to the first sender. The framework models this signal as noise and therefore decreases the SNR at the receiver of the first packet. Such events can happen several times during the reception of a data packet, therefore the receiver has to deal with changing SNR throughout the packet receiving process. The final count of bit errors thus results from this sequence of different SNR values and is assembled from the portions of constant SNR. So the bit errors are accumulated for all portions of the packet with constant SNR. The user has to provide the method to calculate the bit error count for a constant portion of SNR, everything else is handled by the framework.

Multiple access

To utilize the RF communication by multiple participants, three multiple access methods are supported. For separate services, it is likely to utilize different frequency bands (e.g., the 2.4 GHz ISM band and the 868 MHz ISM band). Within these bands, a separation using dedicated frequency channels (frequency division multiple access, FDMA) is typically implemented to increase the number of node sending in parallel. The same purpose is served by overlaying the RF signals in time and frequency domain but coding these with different keys (code division multiple access, CDMA). These three multiple access schemes are generalized by the framework. The user provides a function to implement the adjacent channel interference which handles the filter suppression ratio or the coding gain. This is incorporated by the Air to calculate signal and noise power and consequently the SNR. The other two common multiple access formats, space and time division multiple access (SDMA, TDMA) are supported trivially due to the principle of the Air.

3.3.8. Power simulation

A key feature of the framework (regarding PAWiS requirements) is given by the power consumption simulation of tasks. Therefore, a central power meter object logs the power consumption values that are reported by all modules of all nodes. Only tasks that simulate dedicated hardware directly report power consumption. Software tasks report their CPU utilization, and the CPU module calculates its power consumption and reports it on behalf of the requesting task.

Every hardware task that consumes power reports this to the central reporting facility. It can have different electrical behavior, that is, the current I depends in different ways on the supply voltage U. The current can be constant and therefore independent of the supply voltage. A resistive behavior ($I = U/R$) and a combination ($I = I_{\text{const}} + U/R$) can be modeled. Additionally, a user defined, for example, nonlinear characteristic can be implemented and is supported by the framework. The power consumption as well as the electrical behavior can be updated by a task at any point in time.

The reporting of power consumption is accomplished by calls to special methods offered by PAWiS framework classes. The model programmer has to provide the appropriate figures (current, equivalent resistance). These numbers can be determined in several ways. The most accurate numbers result from measurements of real-world devices (e.g., a test chip or prototype PCB) which should be modeled with the PAWiS framework (though this requires the device to be already available). Alternatively the user can obtain the consumption parameters by electrical simulations of the circuitry using, for example, Spice. This is particularly interesting if the model is programmed in parallel to designing the chip of the planned module. For commercially available components (e.g., a microcontroller or RF transceiver) the data sheet provides the appropriate power consumption values.

It is important to mention that modules of a sensor node do not consume constant power throughout their lifetime. On the contrary, the power consumption varies with the operating state. For example, the CPU consumes less power when being in sleep mode (and hence does not execute instructions), the power consumption of the radio differs whether in transmit, in receive, in PLL-locked or in idle mode. The model programmer has to report new power consumption figures every time the state of the module changes.

The framework supports the modeling of a supply hierarchy where the power input of a module (e.g., an ADC), is supplied by the power output of another module (e.g., an LDO). Since a power supply has varying efficiency according to load and input voltage as well as nonzero output resistance, its output voltage and internal consumption are calculated from its output current. This output current is the sum of all supply currents of the supplied modules. Additionally, the framework provides a mechanism to specify different power supply behaviors (particularly the output resistance). This power consumption model results in a simple electrical network.

During the simulation, a task calculates its current, reports this to its power supply and is notified about the actual input voltage by the power supply in return. Most of this is handled automatically by the framework. This mechanism recursively propagates up the supply tree (breadth-first) and is finally reported to the central *power meter* which stores this values to an external data file. In this way for every module and task of every node in the simulated WSN, the power consumption is calculated and logged. With this approach, the power consumption of the whole node is covered and the simulation results plausibly reflect the reality. These results are analyzed and visualized by the data postprocessing tool (see Section 5). So the power simulation values are not actively evaluated during the simulation run but analyzed after the simulation is finished.

3.4. Dynamic behavior

The PAWiS framework supports dynamic behavior (e.g., mobility, environmental dynamics) via an embedded scripting language. Generally, scripting languages are platform independent and need a virtual machine for execution. Although, it degrades execution time compared to compiled languages, it enables code adjustments and algorithm tweaks even at runtime. Scripting languages can be considered as high-level languages as they usually feature dynamic typing, implicit memory management (garbage collection), and often multithreading or support for coroutines intrinsically. An application utilizing an embedded scripting engine needs to provide glue code in order to make internals accessible via scripts. (Glue code is code that does not provide additional functionality but "glues" application specific objects or functions to the scripting engine.) For the PAWiS framework, the scripting language LUA [30] has been chosen due to its simplicity, extensibility, widespread usage, large community, fast execution, and maturity (http://www.lua.org/).

A large portion of the PAWiS framework's basic functionality is glued to the scripting engine. A main part comprises the module's flow control and power consumption functionality which enables the user to provide *functional interfaces* entirely in the scripting language. This is intended to serve as a rapid prototype development scheme without the need to re-compile the entire simulation. Scripts can be hooked to various events fired by the framework, for example, for set up purposes scripts can be hooked to node or network creation events. Usually simple topologies can be setup with the OMNeT++ intrinsic Ned language but more complex topologies can be created utilizing these initialization scripts with less effort. PAWiS scripts can be used to interface with the framework on network-wide and intranode levels during runtime or in the network setup phase.

A real-world scenario for the necessity of introducing dynamic behavior of sensor nodes is shown in Figure 3 where two cars are passing each other. Each car is equipped with a WSN and both networks affect each other when they come close. In order to simulate and observe this simple yet realistic scenario, it is necessary that the two networks can be moved at a certain velocity and in some direction during runtime. This is achieved by grouping the nodes in two distinct networks and then moving these groups via a script in opposite directions. When sensor nodes change their position in space, the PAWiS framework automatically handles the respective change in the network connectivity and the signal strengths without the need of user intervention (i.e., no method needs to be called to update the adjacency matrix). Accordingly, the radio model is rendered dirty and recalculated when the next activity on the air occurs.

Besides mobility, scripting is useful to handle dynamic and reproducible effects that occur within the sensor network's environment [27]. Generally, sensors (even simulated ones) need to monitor a certain phenomenon. Although the PAWiS simulation framework does not provide a detailed model to capture phenomena (e.g., humidity, and luminance), it defines an abstract interface for sensors and the environment. This sensor/phenomenon pair is handled like a typical subscriber/producer pattern. Sensors subscribe to a phenomenon to get it's current value and will be notified of changes instead of frequently polling the phenomenon thus reducing the computational load. The timing behavior of the notification can be set up (e.g., an update frequency or whenever a significant change of a phenomenon's value occurs) for each sensor individually. Sensor properties maintained by the framework are the position of the sensor and its orientation in 3D space. Whenever a sensor reads its associated phenomenon's value, these properties affect the reading as they define the distance and whether the sensor is facing the value's source.

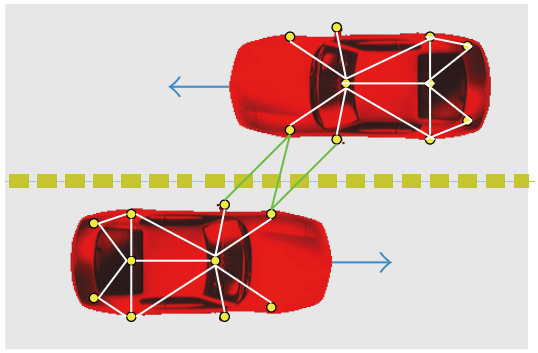

Figure 3: Two cars equipped with wireless sensor networks passing each other.

The semantic of the environmental values has to be introduced by the user of the framework. Though this can be done with C++, it is recommended to use scripts for the environment model as it is more portable and can be exchanged easily for a simulation run. With the help of scripts, even complex environmental scenarios can be easily modeled and simulated. Additionally, the framework supports the usage of prelogged data that can be used as values for phenomena. Currently, effects of interest within the intended field of application for the framework comprise day and night cycles, season changes, weather conditions, and phenomena that support energy scavenging mechanisms.

Utilizing scripting for environment and phenomena as mentioned above introduces reproducible pseudo realistic (opposed to pseudorandom) values to the simulation. While further methods to support more accurate and realistic values exist some of them come up with inherent drawbacks (at least for the targeted field of application). The EmStar [25] framework has the ability to use a hybrid simulation scheme to provide realistic sensor readings. This means that it uses real-sensor readings as input for the simulation. While this brings real-world values into the simulation, it cannot be reproduced over multiple simulation runs. Furthermore, it is not possible to simulate large-time spans, for example, when season-dependent effects need to be considered or the intended network lifetime is up to a year or more.

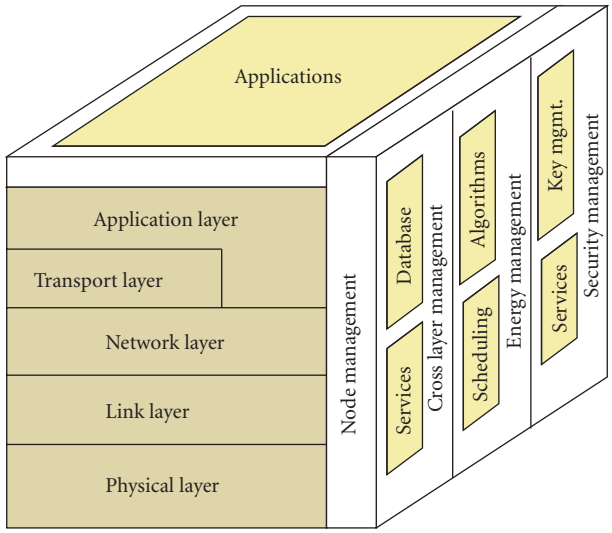

Figure 4: WSN protocol architecture.

4. MODULE LIBRARY

The implementation within a module has to meet the specification of the *functional interfaces* according to the type of the module. This forms another key idea of the modeling process, that is, to provide a library with various distinct implementations for a specific module type. The resulting library can be used to evaluate, refine, and test architectural issues of the user's model. We provide a module library for the proposed protocol architecture (see Section 4.1). These modules can be used in any combination to get an optimized protocol stack for a particular application. Protocol architecture and standardized interfaces (discussed below) make it easy to integrate new modules or interchange different implementations to get optimized results.

4.1. Protocol architecture

The wireless communication characteristics (mobility, rapidly changing link quality, limited resources, and environmental obstructions) and new design paradigms (e.g., wake-up radio) motivate to divert from traditional layered architectures. At the same time "*plug and play*" like features of the layered architecture are important for extensibility and interchangeability.

We define a protocol architecture [31] which provides the benefits of traditional layered architectures [32] and focuses on exploiting synergy across layers (e.g, to extend network life time) [33]. The proposed architecture (see Figure 4) comprises traditional layers (application, transport, network, link, and physical layer) with management planes (cross layer, energy, security, and node). All layers and planes are connected through well-defined interfaces.

The cross-layer management plane (CLAMP) [4] provides a mechanism to exchange cross-layer information but in an optional way that the concept of modularity of layered architectures is still maintained. The CLAMP provides a rich set of parameters available to all the modules of the protocol stack by a publish-notify-update-query mechanism. A discussion of benefits when using cross layer information can be found in [4, 34].

Limited energy sources (e.g., AA battery), insufficient energy from scavenging techniques [35], and the difficulty to replace batteries (cost and geographic reasons) motivated the introduction of an energy management plane (EMP). The EMP can be used to implement algorithms (e.g., [36]) to compute remaining battery capacity or to schedule different events (e.g., updating timers, periodic listening) in order to save energy.

Generally, security is not considered as integrated component of the system architecture (at the start) which affects network security adversely [37]. A security management plane (SMP) is included so that security-related issues (key management algorithms, light-weight encryption and decryption, etc.) can easily be integrated into every component as discussed in [37]. Additionally, network diagnosis and management (NM) used for resetting nodes, remote firmware deployment, address assignment, querying the availability of the node, and so forth, are introduced.

4.2. Interfaces

Standardized interfaces (see Figure 5) between different protocol layers and management planes expedite the process of new module integration and module interchangeability and hence reduces the overall design time. We kept the interfaces small in number and simple to reduce the communication overhead between layers and avoid complexity. Each interface has its own syntax and semantics. The user-defined interfaces can also be used if required. Figure 5 shows the unified view of the protocol layers, management planes, and interfaces among them. The advantage of such interfaces is that it makes the development of protocol layers independent from one another and at the same time provides the functionality to exploit synergy between different layers for cross-layer optimization benefits. Detailed information about the interfaces can be found in [31].

5. DATA POSTPROCESSING

A data postprocessing tool (DPPT) was developed to analyze and visualize the information (power consumption, timing, events, and module interaction) logged during the simulation run. The DPPT (see Figure 6 for a screenshot) along with the PAWiS framework runs on Linux as well as Windows platforms. The DPPT visualizes the simulation results in a hierarchical way by network nodes, modules, submodules, and user-defined categories (e.g., the user can visualize energy consumed by the whole node or by one of its modules like routing or MAC). Each of these elements can be shown or hidden. Elements can be presented in different display colors to be distinguished in the graph. Several navigational helpers (e.g., panning, zooming, scrolling, and snapping) are provided to navigate the simulation data. The DTTP also provides operations on data rows like integration of power consumption, and so forth.

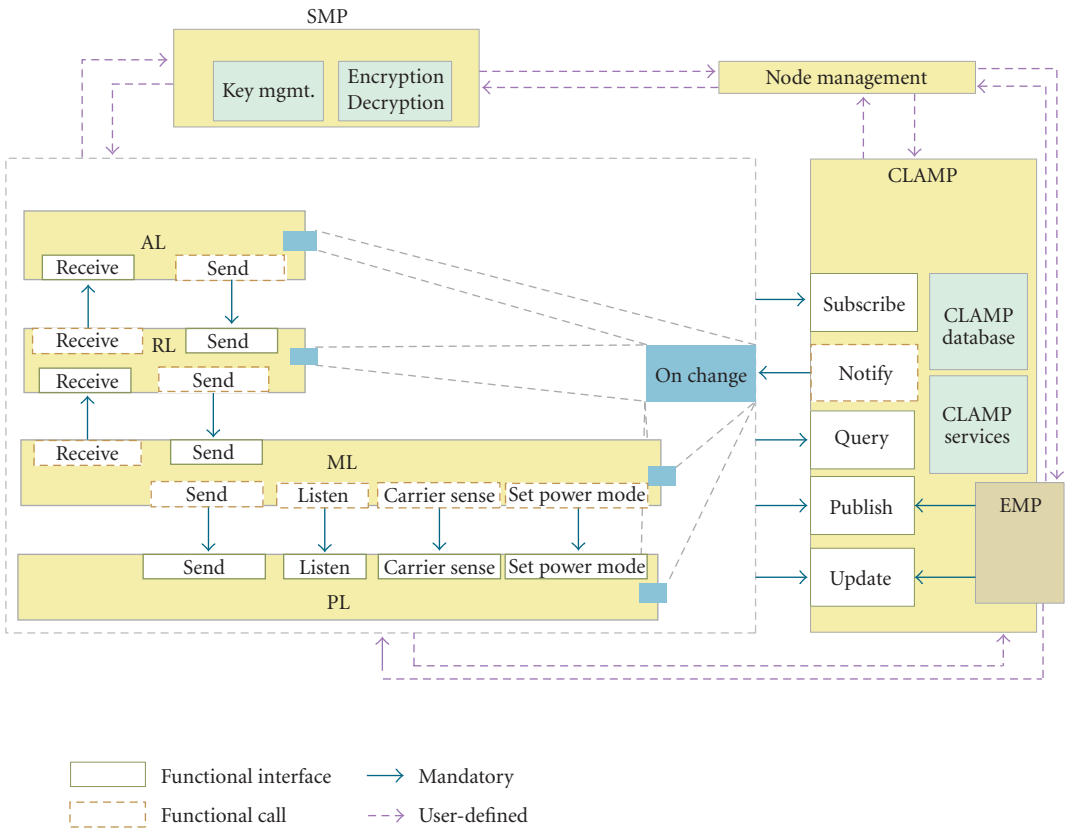

Figure 5: Interfaces between different layers and management planes.

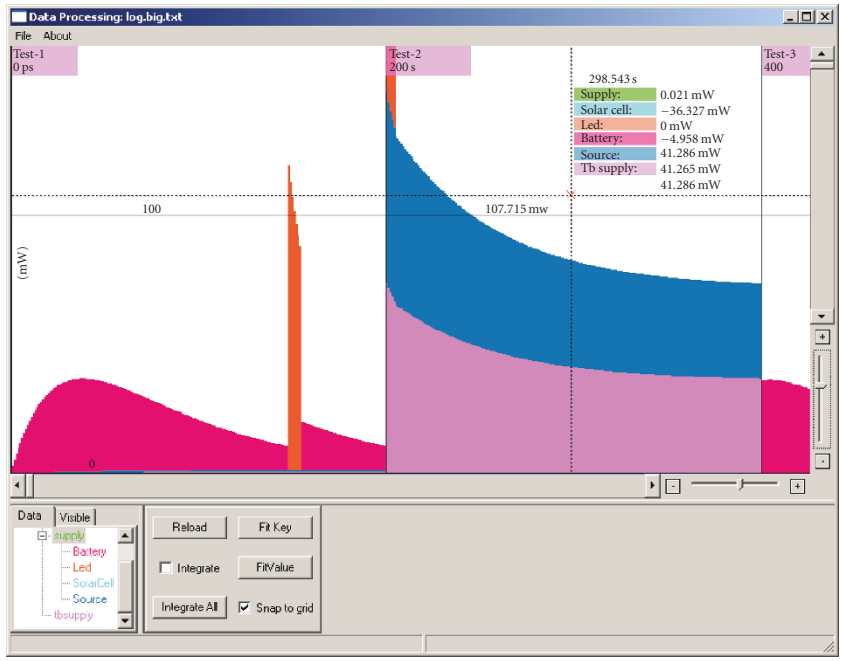

Figure 6: Data postprocessing tool.

The functions of the DPPT are divided into two categories, analysis and visualization of the data. The analysis techniques include the following:

(i) power consumption;

(ii) energy consumption (integral of power consumption over time);

(iii) reveal network topology from routing tables;

(iv) calculate remaining battery capacity from power consumption;

(v) sum of power consumption of a certain module type of all nodes;

(vi) statistical analysis of end-to-end packet delay, number of retransmissions, unnecessary CPU wakeups, and more;

(vii) power and/or energy consumption distribution across several nodes.

The analysis results are visualized by the following:

(i) chart (pie, bar, stack, etc.);

(ii) tables (showing numbers);

(iii) 3D maps (power and/or energy consumption distribution);

(iv) vectors (network topology, routing decisions).

The main display as shown in Figure 6 is the power consumption of the modules of a node plotted over time. The individual power values are stacked and colored according to the module. As the user moves the mouse, an overlay shows the power values at the cursor.

One major difficulty for displaying the data is the high dynamic range of values and times. For WSN nodes, it is typical that within a very short time interval many events (and therefore changes in power consumption) occur followed by a long period of no actions. The user would have to constantly zoom in and out to inspect the points of interest. The same problem applies to power consumption values, for example, the power consumption of the RF transceiver is higher than the consumption of the AD converter by a factor of several hundreds. We tackle this visualization problem by applying a non-linear monotonic concave function (e.g., the logarithm) to the time and/or value *differences*. (We do not transform the absolute time and sum of power values (as for logarithmic plots) but the differences from one discrete event to the next.) This reduces the dynamic range of the differences while equal time intervals and power values are displayed equally at every absolute point.

6. RESULTS

In this section, we present a performance evaluation of the PAWiS framework as well as a case study to show its benefit in designing, simulating, and optimizing real-world scenarios.

Table 1: Performance results for different network sizes.

Nodes	Sim s/s	Event/s
20	207.6790	32,808
50	52.8075	23,975
100	7.9105	16,666
200	2.6811	9,756
500	0.2249	4,025
1,000	0.1278	2,298

6.1. Performance evaluation

To get performance figures, we abstracted an application layer by probabilistic sampling based on uniformly distributed intervals (a node waits between 20 to 70 seconds and then sends data with a probability of 20%). A position-based routing scheme (a scheme which forwards data based on location of the immediate source, the node itself, and that of the destination node) which maximizes progress (minimizes distance) towards the destination node was implemented on the routing layer. At the MAC layer, a simple CSMA/CD (carrier sense multiple access with collision detection) scheme was implemented. A linear battery discharge model and a first-order radio model was used. Many to one communication was considered where all the nodes send data to the sink node located at the center of the network area. Multiple simulation runs with various network sizes and approximately equal node density were made for the performance evaluation. The network nodes were distributed on a grid with a jitter around the grid positions. Scripting was only used to set up the network and simulation environment, but no dynamic effects during simulation runtime have been implemented. The used simulation model put more emphasis on higher-level effects (on protocol level) than on intranode communication or instruction set simulation. For the execution of each simulation run, a common desktop PC equipped with an AMD Athlon 64 3000 CPU (2.0 GHz) and 1.5 GB of RAM has been used.

The simulation framework generates messages for a lot of events that occur during simulation. These messages are dispatched by the OMNeT++ simulation kernel. Therefore, the number of messages (events) generated during a simulation run is a key figure of the performance evaluation. An additional important figure constitutes the ratio between simulated seconds and wall-clock seconds for each run. These figures were acquired by simulating one million events for different network sizes.

Table 1 shows the simulation results for 20, 50, 100, 200, 500, and 1000 nodes. It should be noted that for this particular simulation configuration, the ratio between simulation time and wall-clock time is just an example and should not be seen as a general figure for any simulation. More important are the figures regarding the processed events per second. As the framework handles many aspects of wireless communication, the duration for processing events with lots of nodes increases significantly. This is

related to the fact that the more nodes are simulated, the more signal-dependent calculations have to be computed at each sending operation thus resulting in less processed events per second. Evidently, the node's granularity (in terms of hardware and software) additionally affects the overall simulation performance. The finer grained a node is, the more events are needed to simulate one node which further drops the ratio between simulation time and wall-clock time.

In principle, a finer model granularity results in an increased number of OMNeT++ modules and submodules. OMNeT++ supports parallel discrete event simulation (PDES) strategies to enhance overall simulation performance but special measures need to be taken to ensure the synchronization of modules on distinct processors thus reducing the local simulation throughput. However to benefit from PDES, the model needs to consider some message exchange constraints to ensure the causality of events on distinct processors. Currently, the PAWiS framework does not comply with all of these constraints. Therefore, it is not yet possible to utilize PDES in order to speed up the simulation (though it is planned to make future versions of the framework PDES compliant).

The performance figures for this particular simulation show that with a network size of 500 nodes the simulation time progresses slower than wall clock time which makes it rather impractical to simulate larger networks over long-time periods (e.g., simulating 30 days for 1000 nodes would take 235 days!). As expected, these figures suggest that the framework needs to process more events per second and node with increasing network sizes. Though these figures seem to be not very promising, it should be noted that simulating this large networks at fine granularity is of little practical use. If one wants to simulate at fine granularity within one node, smaller network sizes also help identifying functionality issues and inefficiencies. In a second step larger networks may be simulated by omitting implementation details which helps to reduce the number of events. Moreover, simulating networks with sizes up to 30 nodes does better reflect currently commercially developed sensor network applications.

A scalability evaluation in [13] targeting various aspects of WSN simulation (some of them comparable to the PAWiS framework) with OMNeT++ confirms the above-mentioned figures and conclusions. The authors showed performance figures for WSN models with different complexities and sizes (a full model contains radio, environment, and battery) and finally conclude that simulating larger networks with current approaches is not practical.

6.2. Case study

A case study of a WSN for a tire pressure monitoring system (TPMS) in automobiles (as depicted in Section 3.4 and Figure 3) has been modeled and simulated with the PAWiS framework. The TPMS model consists of four wireless sensor nodes and a sink node. Each sensor node is attached to a different wheel while the sink node is placed in the central console. All sensor nodes are battery powered whereas the sink is connected to the car-power supply. Each sensor periodically (every 20 seconds) reads the sensor values, and in case of a significant change from previous readings (this is modeled with a sending probability of 10%) it sends a packet to the sink node. As the focus of this example is not power awareness, the CPU never enters sleep mode but permanently executes code from the application. If the sink node receives the packet without bit errors, it sends an acknowledgment back to the originator node. At most, three retransmissions are attempted in case of acknowledgment timeouts by the sender. A sensor node in the model consists of modules for application, MAC and routing, timer, and radio. The application module is further divided into a sensor and an ADC for acquiring the tire pressure. In this model, MAC (utilizing CSMA strategy) and routing are combined as there is no explicit routing needed. The simulation of this TPMS model shows the power consumption of each of these modules. Resulting data can be further analyzed to figure out major power consuming modules and optimize them in order to increase the network life time. Two application scenarios have been simulated: the first scenario with a single car (consisting of the four sensor nodes and one sink node) and a second scenario including two cars within each other's proximity to observe the differences in power profiles of different modules. The two-car scenario should show the impact regarding power consumption and packet delivery ratio of potential interferers (packets from the opposing car are treated as noise) on the local sensor network in each car. After simulating both scenarios, the aforementioned DPPT (see Section 5) was used to integrate the power consumption figures of all the modules. The results are shown in Figure 7 where the power consumption of a specific sensor node from both scenarios is presented. As expected, the major part of power is consumed by the application module (indirectly as the CPU is actually consuming the power for software modules) with almost the same value for both scenarios. The sensor/ADC and timer modules contribute only with very small amounts to the overall power consumption since they are inactive for a significant portion of time. In the single-car scenario, the radio and MAC/routing consumed significantly less power than in the two-car scenario. This happens due to the fact that an increased number of sensor nodes within the vicinity of each other results in more network collisions which raise the bit error probability. Furthermore, bit errors result in packet retransmissions which additionally keep the radio in listening state for a longer time to wait for acknowledgments. The same applies for the MAC as more occurrences of timeouts and retransmits have to be processed.

We simulated this simple and straightforward case study of a TPMS on purpose. It took only about 20 man-hours to model and simulate the real world scenario from scratch. Different scenarios were simulated (e.g., without acknowledgments, moving cars, static single car, etc.) with minor adjustments in the configuration file in less time. With the framework, module inefficiencies can easily be identified (the absolute figures may not be entirely accurate, although relative information may provide a deep insight) and can also be corrected in very short development cycles.

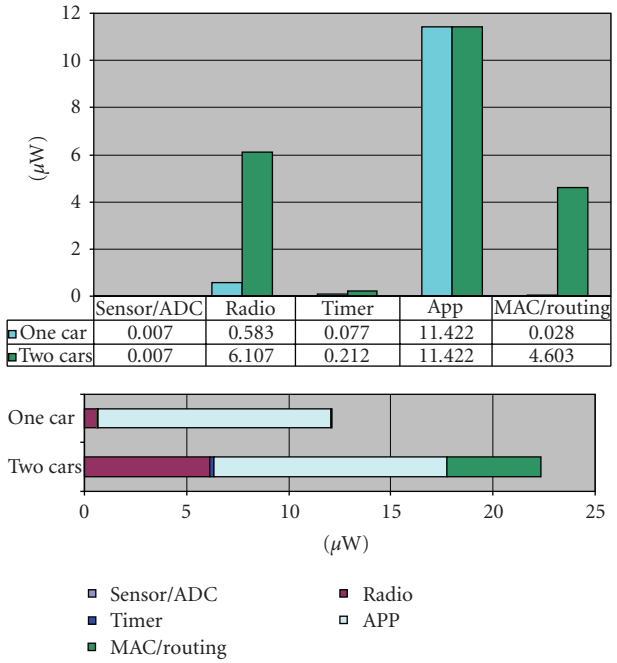

Figure 7: Power consumption results of the TPMS simulation for one node.

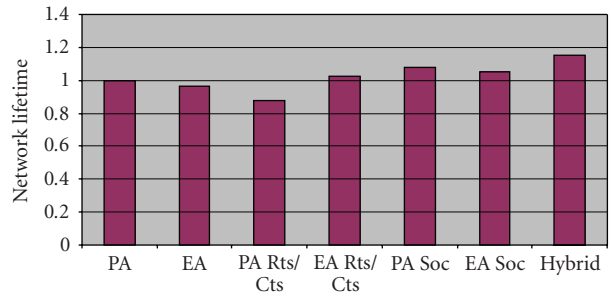

Figure 8: Network life time for various position-based routing implementations.

6.3. Optimization study

In this case study, we discuss the optimization methodology for the development of a routing protocol with the PAWiS framework. This study was aimed to develop and optimize a table-less position-based routing protocol. Initially, we created a skeleton model (the same as used for the study in Section 6.1) for a simple node consisting of application layer (probabilistic sampling), MAC layer (CSMA), and physical layer (simple radio) as well as an implementation of the initial version of a table-less routing scheme which we call progress aware (PA). As the composition of the protocol stack is easily managed with the configuration file without the need to recompile the particular simulation, we executed many trials with numerous node compositions. After analyzing the simulation results, we identified several enhancements and alternatives for various aspects of the routing protocol. Consequently, multiple refinement cycles with distinct implementations of the position based routing strategy were applied. These strategies include PA, energy aware (EA), PA and EA with Rts/Cts packet exchanges, PA and EA with state of charge (SOC) packet exchanges, and even hybrid approaches (to dynamically change between progress aware and energy aware routing) to route packets. First, the performance figures acquired in each refinement iteration were analyzed to identify the main contributors to power consumption in the routing layers. In the next step, the gained insights were reapplied to the routing implementations resulting in additional refinement/analysis cycles until the results met the targeted power consumption constraints. The chart in Figure 8 depicts the network lifetime for various implementations of position-based routing compared to our initial version of a protocol that utilized a PA scheme (with network lifetime scaled to 1 in the chart). The chart shows that the EA scheme exhibits lower lifetime compared to the PA scheme which can be perceived as a contradiction. The reason for this is that in this scheme packet forwarding is based on timers. For example, assume a four-node network comprising a source node S, a destination node D and two potential forwarders A and B. If S has some data to transmit, it sends them blindly. A and B being within transmission range of S receive them and start their respective timers. The timer of the node which provides maximum progress expires first (i.e., the timer of the node with more remaining energy), and further forwards the data while the other node upon listening to that data kills its timer and drops that data. EA scheme always try to balance energy consumption across nodes by routing through nodes with more remaining energy, therefore, once energy balance (all nodes having almost equal remaining energy) across the network is achieved, duplicate packets occur as timers of potential forwarders expire at the same time. These duplicate packets cause the routing layer to utilize the radio more often and hence drain the battery quicker which results in reduced network lifetimes.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a power aware discrete event simulation framework for wireless sensor networks and nodes. Based on OMNeT++, it provides additional features to capture energy consumption (at any desired level), introducing a model for the RF communication (enabling complex topologies) and environmental effects with scripting capabilities. It provides a visualization tool to analyze and visualize energy usage. The framework focuses on extensibility and reusability by outlining a protocol architecture and provides module library. The results show that the performance (execution time) of the PAWiS simulation framework is comparable with other frameworks and appropriate for the targeted field of application. The case studies showed that the modularization of OMNeT++ models combined with the abstract component concept of the PAWiS framework generally results in a reduced design-debug cycle.

The framework in its current version handles already much of the functionality and effects that are important for

simulating wireless sensor networks. However, there are still some extensions and features that need to be enhanced or included. Performance of the simulation framework needs to be further enhanced, as the current results show scalability issues for larger networks (see Section 6.1).

Furthermore, an important aspect of the simulation framework is the postprocessing tool chain. The visualization tool will be extended by additional functions for analyzing the output of the simulation application. Currently, the output comprises power consumption, fired events, and module occupation. Additionally, analysis tools will be included to allow the comparison of nodes regarding various properties gained from the simulation and features like significant power peak detection, min-max over sliding window, integration features.

Additional and updated information regarding the PAWiS project and the simulation framework is available at http://www.ict.tuwien.ac.at/pawis/.

REFERENCES

[1] D. Weber, J. Glaser, and S. Mahlknecht, "Discrete event simulation framework for power aware wireless sensor networks," in *Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN '07)*, vol. 1, pp. 335–340, Vienna, Austria, June 2007.

[2] J. Glaser and D. Weber, "Simulation framework for power aware wireless sensors," in *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN '07)*, K. Langendoen and T. Voigt, Eds., pp. 19–20, Delft, The Netherlands, January 2007.

[3] S. Mahlknecht, J. Glaser, and T. Herndl, "PAWiS: towards a power aware system architecture for a SoC/SiP wireless sensor and actor node implementation," in *Proceedings of 6th IFAC International Conference on Fieldbus Systems and Their Applications*, pp. 129–134, Puebla, Mexico, November 2005.

[4] S. A. Madani, S. Mahlknecht, and J. Glaser, "CLAMP: cross layer management plane for low power wireless sensor networks," in *Proceedings of the 5th International Workshop on Frontiers of Information Technology*, Islamabad, Pakistan, December 2007.

[5] E. Egea-López, J. Vales-Alonso, A. Martínez-Sala, P. Pavón-Marñio, and J. García-Haro, "Simulation tools for wireless sensor networks," in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '05)*, Philadelphia, Pa, USA, July 2005.

[6] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET simulation studies: the incredibles," *Mobile Computing and Communications Review*, vol. 9, no. 4, pp. 50–61, 2005.

[7] D. Wetherall and C. J. Lindblad, "Extending Tcl for dynamic object-oriented programming," in *Proceedings of the 3rd USENIX Annual Tcl/Tk Workshop*, p. 19, Toronto, Canada, July 1995.

[8] V. Naoumov and T. Gross, "Simulation of large ad hoc networks," in *Proceedings of the 6th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM '03)*, pp. 50–57, San Diego, Calif, USA, September 2003.

[9] G. Chen, J. Branch, M. Pflug, L. Zhu, and B. K. Szymanski, "Sense: a wireless sensor network simulator," in *Advances in Pervasive Computing and Networking*, chapter 1, pp. 249–267, Springer, New York, NY, USA, 2006.

[10] S. Park, A. Savvides, and M. B. Srivastava, "SensorSim: a simulation framework for sensor networks," in *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '00)*, pp. 104–111, Boston, Mass, USA, August 2000.

[11] A. Varga, "The OMNeT++ discrete event simulation system," in *Proceedings of the 15th European Simulation Multiconference (ESM '01)*, pp. 319–324, Prague, Czech Republic, June 2001.

[12] W. Drytkiewicz, S. Sroka, V. Handziski, A. Koepke, and H. Karl, "A mobility framework for OMNeT++," in *Proceedings of the 3rd International OMNeT++ Workshop*, Budapest, Hungary, January 2003.

[13] E. Egea-López, J. Vales-Alonso, A. Martínez-Sala, P. Pavón-Marñio, and J. García-Haro, "Simulation scalability issues in wireless sensor networks," *IEEE Communications Magazine*, vol. 44, no. 7, pp. 64–73, 2006.

[14] C. Mallanda, A. Suri, V. Kunchakarra, S. S. Iyengar, R. Kannan, and A. Durresi, "Simulating wireless sensor networks with OMNeT++," submitted to *IEEE Computers*.

[15] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: a library for parallel simulation of large-scale wireless networks," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS '98)*, pp. 154–161, Banff, Canada, May 1998.

[16] M. Varshney, D. Xu, M. B. Srivastava, and R. L. Bagrodia, "SenQ: a scalable simulation and emulation environment for sensor networks," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*, pp. 196–205, Cambridge, Mass, USA, April 2007.

[17] C. Perkins and E. Royer, "Ad hoc on-demand distance vector routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pp. 90–100, New Orleans, La, USA, February 1999.

[18] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing*, Kluwer Academic Publishers, Norwood, Mass, USA, 1996.

[19] D. Cavin, Y. Sasson, and A. Schiper, "On the accuracy of MANET simulators," in *Proceedings of the 2nd ACM International Workshop on Principles of Mobile Computing (POMC '02)*, pp. 38–43, Toulouse, France, October 2002.

[20] G. Chen and B. K. Szymanski, "COST: a component-oriented discrete event simulator," in *Proceedings of the 34th Winter Simulation Conference (WSC '02)*, vol. 1, pp. 776–782, San Diego, Calif, USA, December 2002.

[21] J. Liu, X. Liu, and E. A. Lee, "Modeling distributed hybrid systems in Ptolemy II," in *Proceedings of the American Control Conference*, vol. 6, pp. 4984–4985, Arlington, Va, USA, June 2001.

[22] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao, "Modeling of sensor nets in Ptolemy II," in *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks (IPSN '04)*, pp. 359–368, Berkeley, Calif, USA, April 2004.

[23] J. A. Miller, R. S. Nair, Z. Zhang, and H. Zhao, "JSIM: a Java-based simulation and animation environment," in *Proceedings of the 30th Annual Simulation Symposium*, pp. 31–42, Atlanta, Ga, USA, April 1997.

[24] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pp. 126–137, Los Angeles, Calif, USA, November 2003.

[25] L. Girod, T. Stathopoulos, N. Ramanathan, et al., "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 201–213, Baltimore, Md, USA, November 2004.

[26] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, and M. Karir, "ATEMU: a fine-grained sensor network simulator," in *Proceedings of the 1st Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON '04)*, pp. 145–152, Santa Clara, Calif, USA, October 2004.

[27] M. Demmer, P. Levi, A. Joki, E. Brewer, and D. Culler, "Tython: a dynamic simulation environment for sensor networks," Tech. Rep. UCB/CSD-05-1372, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Calif, USA, 2005.

[28] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pp. 188–200, Baltimore, Md, USA, November 2004.

[29] J. Heidemann, N. Bulusu, J. Elson, et al., "Effects of detail in wireless network simulation," in *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 3–11, USC/Information Sciences Institute, Society for Computer Simulation, Phoenix, Ariz, USA, January 2001.

[30] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua—an extensible extension language," *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[31] S. A. Madani, S. Mahlknecht, and J. Glaser, "A step towards standardization of wireless sensor networks: a layered protocol architecture perspective," in *Proceedings of the International Conference on Sensor Technologies and Applications*, pp. 82–87, Valencia, Spain, October 2007.

[32] V. Kawadia and P. R. Kumar, "A cautionary perspective on cross-layer design," *IEEE Wireless Communications*, vol. 12, no. 1, pp. 3–11, 2005.

[33] W. Su and T. L. Lim, "Cross-layer design and optimization for wireless sensor networks," in *Proceedings of the 7th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD '06)*, pp. 278–284, Las Vegas, Nev, USA, June 2006.

[34] V. T. Raisinghani and S. Iyer, "Cross-layer design optimizations in wireless protocol stacks," *Computer Communications*, vol. 27, no. 8, pp. 720–724, 2004.

[35] S. Roundy, D. Steingart, L. Frechette, P. Wright, and J. Rabaey, "Power sources for wireless sensor networks," in *Proceedings of the 1st European Workshop on Wireless Sensor Networks (EWSN '04)*, vol. 2920, pp. 1–17, Springer, Berlin, Germany, January 2004.

[36] D. Rakhmatov, S. Vrudhula, and D. A. Wallach, "A model for battery lifetime analysis for organizing applications on a pocket computer," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 1019–1030, 2003.

[37] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, pp. 53–57, 2004.