UDC 004.78

Wireless Sensor Network Application Programming and Simulation System

Žarko Živanov[1], Predrag Rakić[1] and Miroslav Hajduković[1]

[1] Faculty of Technical Sciences, Trg D. Obradovića 6,
21000 Novi Sad, Serbia
{zzarko,pec,hajduk}@uns.ns.ac.yu

Abstract. We present, a wireless sensor network application programming and simulation system, suitable for wireless sensor network application development for both resource constrained and unconstrained hardware. Developed programs can be tested inside simulator, or (with source unchanged) executed directly on hardware. Main contribution of our project is uniform object oriented programming model with predefined basic concurrency abstractions.

Keywords: wireless sensor networks; operating systems; simulation; code generation.

1. Introduction

Wireless sensor networks (WSNs for short) are networks consisting of large number of battery powered sensor devices (sensor nodes, or just nodes for short), interconnected by radio waves. The main task of such network is to collect physical data in the given environment and to send it to one or more collector (sink) nodes. Since it is often expensive or impossible to charge or replace node's battery, prolonging the node's lifetime is essential.

Topology of the network is usually dynamic. Although in most cases nodes are not movable, lifetime of each node is different. And because of energy saving, RF range of each node is usually limited to nearest neighbors. This implies that communication with sink must be done by using point to point protocols. When a neighbor dies, node usually must find another route to send its data. There are many protocols addressing this issue [1], [2].

Each node consists of several parts:

- microcontroller (CPU) with RAM and some kind of ROM memory (usually flash or EEPROM);
- one or more sensors (accessed through analog-to-digital converters - ADC);
- battery;
- RF module;
- optional actuators;

– optional additional flash memory.

CPU can be 8, 16 or 32-bit, while available memory is usually a couple of tens kilobytes or hundreds of kilobytes. This imposes constrains on node's program size and complexity. Today's nodes vary from nodes with 8-bit micro controllers like MICA and MICA2 [3], to nodes with 32-bit 400 MHz micro controllers and megabytes of memory, like Intel Mote 2 [4].

Since hardware characteristics vary in wide range, characteristics of their operating systems vary, too. Operating systems of heavily constrained nodes are event-based or cooperative. More powerful nodes (usually with permanent power source) are preemptive.

In this paper we present WAPAS (Wireless sensor network Application Programming And Simulation system) project suitable for wireless sensor network application development for both (power, memory, processor, etc.) constrained and unconstrained hardware. Depending on target hardware characteristics, appropriate concurrency model can be selected (Fig 1).

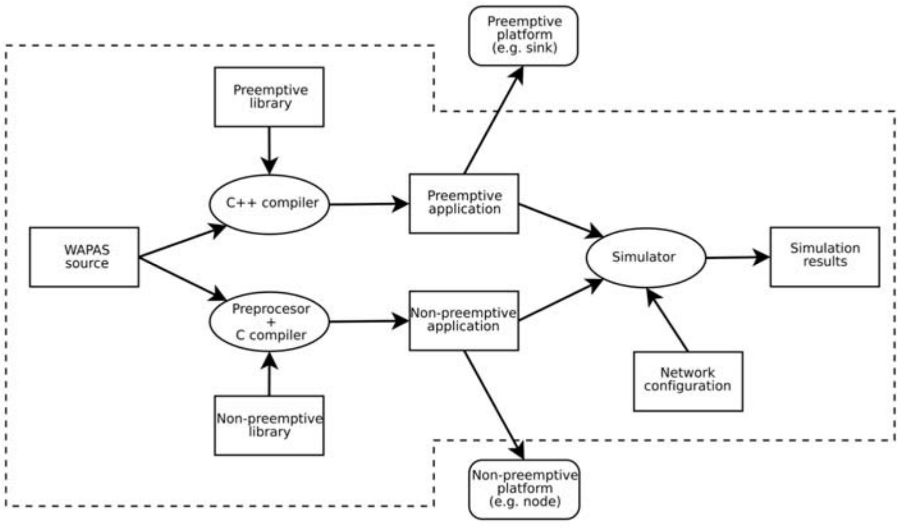

Fig. 1. WAPAS project structure overview

Main contribution, in this paper, is uniform object oriented programming model with predefined basic concurrency abstractions. This programming model can be used with both preemptive and cooperative concurrency models, i.e. it is suitable for both hardware constrained and unconstrained platforms.

Our programming model offers opportunity to application programmers to develop code for both nodes and sinks using almost identical programing model and to test it together, in the same simulation environment.

WAPAS simulator offers complete virtual environment for simulating WSN behavior. Node's hardware is simulated by simulation libraries (sensors, RF module, energy consumption). All functions are accessible from graphical user interface, allowing simulation starting, stopping, modifying and monitoring.

Chapter 2 of the paper presents related work. Chapter 3 introduces proposed programming model. Chapter 4 describes details of the programming model implementation. Chapter 5 contains simulator description. Conclusions are given in Chapter 6.

2. Related Work

2.1. Operating Systems

In literature we recognized two distinctive categories of wireless sensor network operating systems. One category is intended for cheap sensor nodes [5] with finite state machine programing models and event based concurrency, like TinyOS [6] and Contiki [7]. The other is designed for high-end nodes (e.g. sinks), with preemptive operating systems and multi-threaded programming models, like Mantis [8].

Preemptive multitasking with blocking procedure calls is, what today's programmers consider, comfortable environment. Unfortunately, it can not be efficiently implemented on energy constrained hardware platforms, at least not natively.

To fill in the gap, number of projects offer improved programing and concurrency models for low-end nodes operating systems as library extensions or virtual machines:

- TinyThreads [9] is implemented as library on top of TinyOS, allowing intermixing event-driven and threaded programming. It contains library functions that provide blocking I/O operations. Every thread has its own stack.
- Coroutines for TinyOS [10] offer a programming model which combines event-driven system with cooperative multitasking, enabling the programmer to structure his application as sequential code instead of as a state machine.
- Protothreads [11] supports an extremely lightweight stackless type of thread, and provides conditional blocking on top of event-driven system without the overhead of per thread stacks.
- Fibers for TinyOS [12] allows user to use blocking I/O calls using just one (system) stack with limitation that there can be only one user fiber. Implementation is similar to protothreads but instead of jumping back to main loop, a blocking call actually calls scheduler. This allows user to use local variables and block inside subroutines.
- Mate [13] is a virtual machine environment for TinyOS. Among other things, Mate provides a way for threating split-phase operations as though they were strait-line pseudo-blocking operations. However, since

Mate is virtual machine these operations introduce cost of byte-code interpretation.

These projects provide improved models to underline operating systems, but their concurrency model improvements are available only together with certain overhead and programming interface change.

2.2. Simulators

First sensor network simulators were made as an extension of already existing network simulators. These simulators mainly focus on protocols used in WSNs. Recently, several simulators are made specifically for WSNs. Some of them are general simulators, while others are simulators for a specific hardware platform or operating system.

SensorSim [14] is an extension to ns-2 network simulator. This simulator provides models needed for WSN modeling: battery, CPU, sensors, RF module, etc. Power consumption of all components is also modeled. It also supports hybrid simulation: ability to interconnect real and simulated nodes.

GTSNetS [15] is a simulator for large-scale sensor networks, capable of simulating hundreds of thousand nodes. This simulator is an extension of GTNetS network simulator. It provides several models for the different functional units composing a sensor node. It is mainly a communication protocol simulator.

SENS [16] is a customizable WSN simulator, consisting of components for every aspect of the simulation. It is a platform independent simulator with a thin compatibility layer which allows portability between simulator and real sensor nodes.

J-Sim [17] is a simulation environment for WSN developed in Java. It is mainly oriented towards communication protocol simulation. WSN modeling is based on inheritance of classes in the simulation framework.

TOSSIM [18] is a simulator for TinyOS applications. The main goal of TOSSIM is to provide a simulator that bridges the gap between algorhytms and implementation. Most of the code written for TOSSIM can be directly compiled for TinyOS.

TOSSF [19] is a scalable simulator for TinyOS applications. It allows heterogeneous collection of sensor nodes and allows dynamic network topology. TOSSF simulates execution of TinyOS applications at source level. This simulator is an adaptation of SWAN, a simulator for wireless ad-hoc networks.

3. Programming model

The traditional approach to deal with concurrency is to use preemptive multithreading. In preemptive systems, programmer has no control over the moment at which control switch occur. This imposes context switching and shared variables protection overhead. This overhead can be significant in hardware constrained systems. Since preemptive concurrency model is convenient and comfortable for hardware unconstrained platforms and at the same time inappropriate for hardware constrained platforms both preemptive and non-preemptive concurrency models are used in WSN operating systems. We are developing uniform programming model suitable for both types of platforms: preemptive and non-preemptive.

Our programming model is inspired by COLIBROS operating system [20], [21]. It is based on C++ language because object oriented view is much more natural and comfortable than procedural or state machine views are [22]. Concurrency is described (and implemented in preemptive systems) by inheritance and polymorphisms.

We identified basic concurrency abstractions suitable for wireless sensor programming and integrated them in our programming model. These abstractions are:

– threads;
– shared variables.

These concurrency abstractions, threads and shared variables, relate like speaking language subjects and objects. Thread, like subject, represents somebody conducting actions. Shared variable, like object, represents something actions are conducted on. Together they are used to model wireless sensor network application.

3.1. Threads

Thread represents context of execution. Separate tasks in application are implemented as threads.

We concluded that any activity in WSN application that is needed once will be probably needed again. Activities that are really needed only once are some kind of initialization and their place is in thread object constructors. So, we offer no dynamic thread instantiation. Thread objects are statically instantiated. Thread can be started only once and when it is finished it's memory is not freed. For that reason, threads should never end. Instead, they should contain endless loop. In the loop, thread activity can be blocked (postponed):

– waiting for some time to expire;
– waiting for some condition to be fulfilled;
– waiting for some event to happen.

WAPAS application entry point is not `main()` function but instead system declared and user defined `Initial::run()` member function.

Example of thread definition, instantiation and start is shown in Program 1. In this example one user thread class called `Simple` is defined. Member function `run()` contains thread instructions: endless loop the body of which is executed every 500ms. First, current time is saved in local variable `start_time`. After that analog-to-digital (A/D) conversion is started (adc.getData()) and when conversion result is returned in local variable data, it is sent through RF module (rf.send()). When cycle (A/D conversion and sending) is finished thread activity is stopped. Thread activity is resumed (cycle repeated) 500ms after beginning of previous cycle (delay_till()). Function delay_till() provides mechanism for scheduling periodic activities independent of activity duration. Objects adc and rf represent ADC and RF module devices, respectively. Operation adc.getData() acquires data from ADC. It is presented in Program 3. Operation rf.send() transmits data. It is not presented in this article.

Thread object simple is statically instantiated. In initial thread, node's duty cycle is defined through working (working_period()) and sleeping (sleeping_period()) period. Duty cycle is set to 2% (10ms working period and 490ms sleeping period). After that, thread simple is scheduled (start()).

Program 1: Simple Thread Example

```
class Simple : public Thread {
  public:
    void run(void) {
        unsigned data;
        unsigned start_time;
        int result;
        for (;;) {
            start_time = time_get();
            data = adc.getData();
            result = rf.send(data, ...);
            ...
            delay_till(start_time+500);
        }
    }
};

Simple<DEFAULT_STACK_SIZE> simple;

void Initial::run() {
    working_period(10);
    sleeping_period(490);
    simple.start();
}
```

3.2. Shared variables

During application execution threads need to communicate (exchange data) with other threads as well as with environment (timers, sensors, communication devices, etc.). Depending on concurrency model used, communication (accessing shared memory locations) can lead to race conditions. Shared variables are developed with appropriate synchronization protocols which provide safe communication, independent of concurrency model used.

We recognize two types of shared variables:

– exclusive;
– atomic.

Exclusive Variables

Exclusive variables are used for communication between threads. They provide mechanisms for thread synchronization – mutual exclusion and conditional synchronization.

Mutual exclusion is achieved through instancing `Exclusive_block` class objects and conditional synchronization is achieved using member attributes of type `Condition`.

Exclusive variable definition example is shown in Program 2. In this example exclusive class `Communicate` is defined. This class guarantees safe communication between two threads, i.e. communication is atomic. One message can not be read more then once and new message can not overwrite old, unread one. Objects of this type are used for communication in producer-consumer situations. The `Communicate` class has two member functions `send()` and `receive()`, bodies of which are executed in exclusive blocks. Constructor and destructor of object of `Exclusive_block` type borders exclusive block.

In member function `send()` first it is checked if object is `EMPTY`. If it's not, calling thread execution is suspended (`empty.await()`) until object is emptied. When object becomes empty, thread execution is resumed, new message is placed in object, object state is changed to `FULL` and thread (if any) waiting for state `FULL` is resumed (`full.signal()`).

Member function `receive()` is symmetrical. So, first it is checked if object is `FULL`. If it's not, calling thread execution is suspended (`full.await()`) until message is placed in object. After that, object state is changed to `EMPTY`, thread (if any) waiting for state `EMPTY` is resumed (`empty.signal()`) and message is returned to calling thread. Message consistency is preserved because message (`data`) is returned before destruction of local variable `set_up`.

Program 2: Exclusive variable Example

```
class Communicate : public Exclusive {
  enum States { FULL, EMPTY };
  unsigned data;
  Condition full, empty;
  States state;
 public:
  Comunicate() : state(EMPTY) {}
  void send(unsigned d) {
    Exclusive_block set_up(this);
```

```
      if(state != EMPTY)
        empty.await();
      data = d;
      state = FULL;
      full.signal();
    }
    unsigned receive() {
      Exclusive_block set_up(this);
      if(state != FULL)
        full.await();
      state = EMPTY;
      empty.signal();
      return data;
    }
  };
```

Atomic Variables

Atomic variables usually represent hardware components. They are used for communication between threads and devices to ensure communication atomicity.

Atomic class definition example is shown in Program 3. In this example, `Adc` class represents ADC assigned `ADC_INT_NUMBER` interrupt vector number. This class has two member functions. One, `getData()` is blocking and designed for thread to ask ADC for new sample and expect for conversion completion. The other is interrupt handler invoked by ADC to notify system that conversion is completed.

Bodies of these two member functions are atomic. Atomicity of `getData()` is protected by variable `set_up` of `Atomic_block` type. `Atomic_block` constructor disables interrupt handling and destructor returns it in previous state, thus protecting atomicity of region in which variable of this type lives. Atomicity of interrupt handler is protected by hardware mechanisms.

In function `getData()` first it is checked if A/D conversion is already started by another thread (`busy`). If it is, calling thread execution is suspended (`ready.expect()`) until ADC is ready for new cycle. When ADC is ready, it is declared busy and is instructed to start conversion. After that, calling thread is unconditionally suspended (`dataReady.expect()`) to wait for conversion completion indicated by interrupt handler. After resuming, ADC is declared ready for new cycle, thread waiting for ADC (if any) is resumed (`ready.notify()`) and acquired sample is returned to calling function.

Interrupt handler is called when sampling cycle is completed. Interrupt handler places the sample in attribute `data` and resumes activity of thread that initiated sampling cycle, i.e. thread that called `getData()`.

Program 3: Atomic variable example

```
  class Adc : public Atomic <ADC_INT_NUMBER> {
    Event ready, dataReady;
    bool busy;
    unsigned data;
   public:
    Adc() : busy(false) {start_interrupt_handling();}
    unsigned getData() {
      Atomic_block set_up;
      if(busy) ready.expect();
      busy = true;
      ...; // instruct hardware to start sampling
           // and place result in attribute "data"
      dataReady.expect();
      busy = false;
      ready.notify();
      return data;
    }
    void interrupt_handler(void) {
      data = ...; // io access
      dataReady.notify();
    }
  }
```

4. Programming Model Implementation

Programs written using previously described programming model can be efficiently compiled for both preemptive and non-preemptive system. For preemptive system we are developing system library. For event-based systems we are developing preprocessor. This preprocessor translates object oriented C++ source code to plain and efficient C code.

On both platforms application code is compiled together with system libraries similar to exokernel application [23].

4.1. Preemptive System

In preemptive system which is generally not resource constrained, whole program is compiled with C++ compiler and linked with appropriate system libraries (target hardware platform or simulation system).

Threads and shared variables (exclusive and atomic) are statically defined global objects. Order of thread execution is determined by preemptive priority scheduler.

Every thread has its own memory region containing descriptor followed by stack. Stack size is defined with `Thread` class template parameter. In simulation, block of memory with no access rights is placed between descriptor and stack. Access to this block can be used to detect stack overflow.

Exclusive variables are protected against race conditions. Exclusive regions are used for mutual exclusion. Part of code in which threads access exclusive variables (actually their attributes) is called critical section. Execution of these parts of code should be serialized (mutually exclusive). Mutual exclusion is achieved by enclosing critical sections in exclusive region blocks.

Every exclusive variable contains ticket, that thread should obtain before accessing it. Thread that obtains ticket enters exclusive region. Other threads suspend their execution until the ticket is granted to them.

Ticket is obtained in constructor of `Exclusive_block` type and released in its destructor. Thus, exclusive regions are represented by blocks of code in which local variable of `Exclusive_block` type exist.

Conditional synchronization is achieved through member objects of `Condition` type. Object of this type contains list in which threads that wait fulfillment of that condition are linked. Thread can choose place for itself in list using operations: `first()`, `next()` and `last()` and link itself using `await()` operation. Condition fulfillment is indicated with `signal()` operation. This operation transfers first thread out of the list.

Race condition protection in access to atomic variables is achieved through atomic regions. Atomic regions are implemented in constructor and destructor of `Atomic_block` class. In constructor interrupt handling is disabled and in destructor it is enabled.

Thread can suspend its activity until arrival of event. The events are represented with objects of `Event` class. Object of this class contains list in which threads, expecting the event, are linked. Thread is always linked at the last position in the list (`expect()`). Event arrival is usually indicated in interrupt handler (`notify()`).

4.2. Non-preemptive System

WAPAS offers natural programing model for preemptive systems. In non preemptive system whole WAPAS program is transformed to fit in cooperative multitasking paradigm and hardware limitations.

Thread bodies (body of member function `run()`) of all thread classes are combined in one `body()` function with certain changes (described later). The `body()` function is executed in infinite loop. Interrupt handlers can preempt `body()` function.

In WAPAS application time is divided in working and sleeping periods (Fig 2). One working period is called epoch. Epoch consists of many cycles. One cycle is one execution of `body()` function.

Most of the time, node is in power-saving (sleeping) mode. When sleeping time expires, system becomes on-line (active) and executes `body()` function i.e. first cycle begins. After cycle is finished, system checks if active time expired, and if it isn't, next cycle is started. If active time expired, system goes to sleep mode unless some thread called `operation_in_progress()` system call. In that case, cycles are executed until `operation_completed()` system call. After completion of the cycle in which system call `operation_completed()` is executed, system goes to sleep mode.

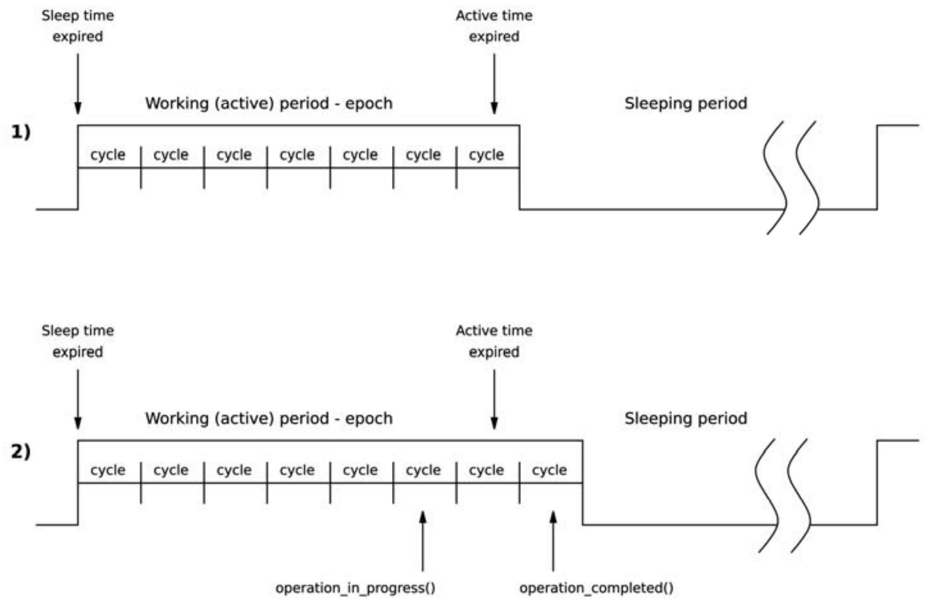

Fig. 2. Application time line in non-preemptive system: showing one working and one sleeping period. Working period begins when sleep time expires. It consists of cycles. Working period ends after (1) completion of cycle in which active time expired, if `operation_in_progress()` has not been called, or either (2) after completion of cycle in which `operation_completed()` system call is executed after active time expired.
Note: The sleeping period is usually much longer than active one [24].

`operation_in_progress()` and `operation_completed()` system calls are designed for operations (e.g. communication) that span over multiple cycles but should not be suspended in one and resumed in next epoch. It is up to the programmer to use this mechanism carefully, taking care not to disturb node's duty cycle significantly. These system calls are ignored in preemptive system, since duty cycle of preemptive system is 100% (it is never in sleep mode).

The first phase during code generation is preprocessing. Preprocessing is largely based on code earlier developed for kiosk application generator [25]. Preprocessor first translates application's source code into C code, with inherently less overhead (binary code compiled directly from object oriented source code introduces certain overhead). In order to optimize programs for such systems, some restrictions and semantic changes on program model are present if the target is non-preemptive:

– Only one thread can wait on a single condition;

- Calls to `Condition::await()` and `Event::notify()` are implemented as conditions. If condition is not fulfilled, thread's code is not executed in the given cycle;
- `Condition::await()` and `Condition::signal()` can only be used inside member functions of exclusive class;
- `Event::expect()` and `Event::notify()` can only be used inside member functions of atomic class;
- `first()`, `next()` and `last()` methods of `Exclusive` class are not allowed;
- Only periodic threads are allowed (that is, only threads that run from beginning to end in each epoch). Thread that have long execution can monopolize CPU time;
- Declaration of local variables is not allowed inside blocks contained in any of the functions (it is allowed only at the function body level).

All thread classes, exclusive variable classes and atomic variable classes are registered. Also, all objects representing recognized classes are registered. Conditional jumps are inserted to enable continuation of execution of each thread's code in the cycle (if, for example, thread waits for `signal()` or `notify()`, it will not start from the beginning, but from inserted conditional jump that decides if thread's code should continue its execution). Modifications of the source code are:

1. For each registered object (threads, shared variables), appropriate structure is created, holding all needed data, including local variables (if present);
2. Accessing local variables is converted into accessing member of appropriate structure;
3. For each atomic variable, its interrupt handler's code is extracted and placed in appropriate interrupt handler function;
4. `run()` method of all threads are combined into `body()` function, in descending priority order. In front of each thread's code a conditional jump is inserted, that decides if thread's code should be executed in the given cycle. Thread's `run()` code is sliced into code segments divided by calls to `await()`, `expect()` and `yield()`(willingly give-up the CPU) methods. A switch statement is introduced for jumping into appropriate place in code;
5. Calls to shared variable methods are replaced by complete code from that method;
6. Calls to `await()` or `expect()` methods are replaced by appropriate conditional jump;
7. Calls to `signal()` and `notify()` methods are replaced with code that sets attribute of appropriate structure. These attributes are checked inside inserted conditional jumps.

After execution of last cycle in the current epoch, hardware is adjusted for next epoch. After that, node goes to sleep mode, until awakening.

5. Simulation Environment

In order to simulate wireless sensor network, one must provide complete virtual environment to code executing in each node. This includes:

– simulating sensor readings;
– simulating actuator actions;
– simulating message sending and receiving;
– simulation-data collecting, presenting and analysis.

For sensor readings simulation we need to model sensor behavior and to supply data values for each sensor. Most of the sensors are, from the node's code point of view, AD converters and are relatively easy to simulate. AD converter can be described with its data size, conversion time, and level of oscillations in accuracy (needed for modeling malfunctions). Depending on physical signal and required level of realism, physical signal propagation can be simulated (e.g. sound waves).

Actuator actions can be modeled according to their influence on the environment. Each action must be separately programmed, depending on its nature.

Message sending and receiving simulation includes RF module simulation and wave propagation simulation. Again, depending on required level of realism, wave propagation can be simply modeled as RF module sending range/strength, or it can include influence of obstacles.

The sole purpose of WSN simulation is to provide insight information on WSN functionality. Thus, simulation environment must be able to collect relevant data and to display it on demand.

Our simulation environment consists of two main parts:

– simulation libraries;
– simulation server.

Simulation libraries are used in node's program code, replacing code used for hardware handling (sensors, actuators, RF module, battery, ...). Simulation server coordinates all actions needed for simulation, supplies physical data to virtual sensors, handles RF communication and collects simulation data.

5.1. Simulation Libraries

Simulation libraries are linked with node's program code in order to simulate node's hardware. Simulation includes:

– RF module simulation;
– AD converters simulation;
– actuator simulation;
– hardware timers simulation;
– battery consumption simulation.

RF module simulation is done via TCP/IP communication with simulation server.

AD converters and actuator simulation is also done via communication with simulation server, which provides all physical data. Internal timings of the AD converters are realized inside library.

Hardware timers simulation provides all the functionality of node's internal timers (including watchdog timer, external timers, etc.).

Battery consumption simulation is based on energy consumption of node's parts (CPU - for all power saving modes, RF module - for sending and receiving, sensors, actuators, flash memory).

5.2. WSN Simulation Server

WSN Simulation server's (server, for short) main role is to simulate node's environment. That includes:

- supplying data for sensor readings and reacting to actuator commands;
- simulating message sending and receiving.

In order to achieve these goals, server needs additional functionality:

- reading simulation configuration files and creating simulation environment;
- receiving messages from all nodes and determining which message can go to which node;
- calculating every node's visibility to all other nodes;
- calculating current physical data for all sensors;
- starting, stopping or pausing simulation;
- displaying all relevant data;
- user interaction with simulation;
- monitoring of the whole simulation process.

At the start of the simulation, server first reads configuration files. Every node is described with its ID, coordinates, battery state, node's software (name of the file with software image), start and stop time(s). Terrain is described with simplified obstacles. Each obstacle has its type (square/circle), coordinates, RF attenuation factor and, eventually, description of its movement during the simulation. Since the simulation can be done on a computer network, server also needs a list of all computers available for simulation (their IP addresses). For each simulated node, a new process is created on one of available computers.

Calculating RF visibility between nodes is done by using each node's range/strength and by including influence of obstacles. For each node it is determined which nodes can receive its messages, and from which nodes it can receive messages. If there are moving obstacles, this calculation must be done after each obstacle configuration change.

RF interference is simulated using RF interference sources (each described with its coordinates and strength, similar to RF module strength) and by calculating packet collision (based on sending time of each packet). Random RF interference is also included via probability factor.

Sensor data is described as a two-column table with time and value in each row. Server calculates actual value for the current time using linear interpolation.

Separate configuration file describes conditions for automatic simulation pausing. Simulation can be paused when certain amount of time passes, when (average) value of some variable or parameter (e.g. software version, battery capacity, working status, internal variables, etc.) for a node or group of nodes reaches certain value, etc.

5.3. Graphical Interface

Important part of the server's functionality is a graphical representation of simulated WSN. Each node is represented with a square and some basic text information. Square's color roughly describes node's status. Detailed information about node can be obtained by clicking on it (node's range is displayed as a transparent circle and more text data is shown). Graphical interface allows many operations on simulated WSN:

- loading and saving current WSN configuration;
- starting, pausing and stopping of the whole simulation;
- changing of parameters for one or more nodes (working state, variables, coordinates, etc.);
- changing of parameters for one or more obstacles (coordinates, size, etc.);
- viewing graph representation of any parameter;
- launching text editor to manually edit configuration files or to edit node's source code;
- compiling node's source code.

5.4. Additional programs

Additional programs are used to generate some of data needed for simulation. Their functions are:

- generating regular/random node placement;
- generating sensor data using mathematical functions;
- generating data for obstacle movement during time.

Additional programs can be executed from GUI, or independently from the command line.

6. Conclusion and Further Work

Main contribution of WAPAS is uniform object oriented programming model, suitable for both hardware constrained and unconstrained platforms (with some limitations for the hardware constrained platforms).

Preemptive part of WAPAS is based on COLIBROS operating system. For non-preemptive system, the whole application is transformed to fit in cooperative multitasking paradigm and hardware limitations. Application programmer may or may not understand all details of this transformation, but (s)he must be aware of certain limitations. Level of application programmer's expertise necessary for efficient usage of this programming model (especially in non-preemptive environments) remains to be determined.

Though WAPAS is still in development phase, we are simultaneously developing simulation environment with graphical user interface, to ease application development.

7. References

1. Uludag, S. and Lui, K. and Nahrstedt, K. and Brewster, G.: Analysis of Topology Aggregation techniques for QoS routing. ACM Comput. Surv., ACM, 39, 7. (2007)
2. Nakamura, E. F. and Loureiro, A. A. F. and Frery, A. C.: Information fusion for wireless sensor networks: Methods, models, and classifications. ACM Comput. Surv., ACM, 39, 9. (2007)
3. http://www.xbow.com
4. http://www.intel.com/research/exploratory/motes.htm
5. Akyildiz, I. F. and Su, W. and Sankarasubramaniam, Y. and Cayirci, E. Wireless sensor networks: a survey. Comput. Networks, Elsevier North-Holland, Inc., 38, 393-422. (2002)
6. Hill, J. and Szewczyk, R. and Woo, A. and Hollar, S. and Culler, D. and Pister, K.: System architecture directions for networked sensors. SIGPLAN Not., ACM Press, 35, 93-104. (2000)
7. Dunkels, A. and Gronvall, B. and Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. lcn, IEEE Computer Society, 00, 455-462. (2004)
8. Bhatti, S. and Carlson, J. and Dai, H. and Deng, J. and Rose, J. and Sheth, A. and Shucker, B. and Gruenwald, C. and Torgerson, A. and Han, R.: MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. Mob. Netw. Appl., Kluwer Academic Publishers, 10, 563-579. (2005)
9. McCartney, W. P. and Sridhar, N.: Abstractions for safe concurrent programming in networked embedded systems. SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems, ACM Press, 167-180. (2006)
10. Rossetto, S. and Rodriguez, N.: A cooperative multitasking model for networked sensors. ICDCSW '06: Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems, IEEE Computer Society, 91. (2006)

11. Dunkels, A. and Schmidt, O. and Voigt, T.: Using Protothreads for Sensor Node Programming. Proceedings of Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks. (2005)
12. Welsh, M. and Mainland, G.: Programming sensor networks using abstract regions. NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation, USENIX Association, 3-3. (2004)
13. Levis, P. and Culler, D.: Maté: a tiny virtual machine for sensor networks. SIGOPS Oper. Syst. Rev., ACM Press, 36, 85-95. (2002)
14. Park, S. and Savvides, A. and Srivastava, M. B.: SensorSim: a simulation framework for sensor networks. MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems, ACM Press, 104-111. (2000)
15. Ould-Ahmed-Vall, G. H. B. R. D.: Simulation of large-scale sensor networks using GTSNetS. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on, IEEE Computer Society, 211-218. (2005)
16. Sundresh, S. and Kim, W. and Agha, G.: SENS: A Sensor, Environment and Network Simulator. anss, IEEE Computer Society, 00, 221. (2004)
17 Sobeih, A. and Chen, W. and Hou, J. C. and Kung, L. and Li, N. and Lim, H. and Tyan, H. and Zhang, H.: J-Sim: A Simulation Environment for Wireless Sensor Networks. anss, IEEE Computer Society, 00, 175-187. (2005)
18. Levis, P. and Lee, N. and Welsh, M. and Culler, D.: TOSSIM: accurate and scalable simulation of entire tinyOS applications. SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems, ACM Press, 126-137. (2003)
19. Perrone, L. F. and Nicol, D. M.: Network modeling and simulation: a scalable simulator for TinyOS applications. WSC '02: Proceedings of the 34th conference on Winter simulation, Winter Simulation Conference, 679-687. (2002)
20. Hajdukovic, M.: Operativni sistemi (problemi i struktura). Fakultet tehnickih nauka, 257, (2004)
21. Rakic, P.: Migracija konkurentne biblioteke COLIBRY sa MS/DOS na GNU/Linux platformu. Fakultet tehnickih nauka. (2006)
22. Cao, Q. and Abdelzaher, T.: liteOS: a lightweight operating system for c++ software development in sensor networks. SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems, ACM Press, 361-362. (2006)
23. Engler, D. R. and Kaashoek, M. F. and J. O'Toole, J.: Exokernel: an operating system architecture for application-level resource management. SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles, ACM Press, 251-266. (1995)
24. Hill, J. and Horton, M. and Kling, R. and Krishnamurthy, L.: The platforms enabling wireless sensor networks. Commun. ACM, ACM Press, 47, 41-46. (2004)
25. Živanov, Ž.: Generator for kiosk applications (Masters thesis). Author's reprint, Library of Faculty of Technical Sciences, Novi Sad, Serbia. (2006)

Žarko Živanov has graduated at the Faculty of Technical Sciences, University of Novi Sad, in 2000. He got Master Degree in 2006, at the Faculty of Technical Sciences, University of Novi Sad. He is currently assistant at Chair for Applied Computer Science at the Faculty of Technical Sciences, Computing and Control Department, University of Novi Sad.

Predrag Rakić has graduated at the Faculty of Technical Sciences, University of Novi Sad, in 2001. He got Master Degree in 2006, at the Faculty of Technical Sciences, University of Novi Sad. He is currently assistant at Chair for Applied Computer Science at the Faculty of Technical Sciences, Computing and Control Department, University of Novi Sad.

Miroslav Hajduković has graduated at the Faculty of Electrical Engineering, University of Sarajevo, in 1977. He got Master Degree in 1980 and his Ph.D. in 1984 from Faculty of Electrical Engineering, University of Sarajevo. In 1985 he was on post-doctoral studies in Computer Laboratory at the Cambridge University in Great Britain. He is currently a Professor of Computer Science at the Faculty of Technical Sciences, Computing and Control Department, University of Novi Sad.

Received: November 8, 2007; Accepted: November 22, 2007