# Cross-Level Sensor Network Simulation with COOJA

Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, Thiemo Voigt
Swedish Institute of Computer Science
{fros,adam,joakime,nfi,thiemo}@sics.se

## Abstract

*Simulators for wireless sensor networks are a valuable tool for system development. However, current simulators can only simulate a single level of a system at once. This makes system development and evolution difficult since developers cannot use the same simulator for both high-level algorithm development and low-level development such as device-driver implementations.*

*We propose cross-level simulation, a novel type of wireless sensor network simulation that enables holistic simultaneous simulation at different levels. We present an implementation of such a simulator, COOJA, a simulator for the Contiki sensor node operating system. COOJA allows for simultaneous simulation at the network level, the operating system level, and the machine code instruction set level. With COOJA, we show the feasibility of the cross-level simulation approach.*

## 1. Introduction

Code development for wireless sensor networks is difficult and tedious [6, 11]. Reasons include the distributed nature of sensor networks, as well as the longer compile-run-debug cycle caused by the need to transfer the compiled program onto the set of sensor nodes used for development and testing.

Software development for sensor networks can be simplified by using a system simulator which allows to develop algorithms, study system behaviour and observe interactions in a controlled environment [4]. Current WSN simulators perform simulation at a specific, fixed, level such as the application, operating system or hardware level. The level at which the simulation is performed affects both the level at which software development can occur and the execution efficiency of the simulator. A simulator that simulates a particular sensor node platform at the hardware level enables the development of low-level software such as device drivers but at the price of longer simulation times and higher code complexity since low-level programming languages must be used. Conversely, a high-level simulator that does not model node hardware may provide short simulation times but enables the development of high-level algorithms only.

The main contribution of this paper is COOJA, a novel simulator for the Contiki operating system [1] that enables *cross-level simulation*: simultaneous simulation at many levels of the system. COOJA combines low-level simulation of sensor node hardware and simulation of high-level behavior in a single simulation. COOJA is flexible and extensible in that all levels of the system can be changed or replaced: sensor node platforms, operating system software, radio transceivers, and radio transmission models.

The simulator is implemented in Java, making the simulator easy to extend for users, but allows sensor node software to be written in C by using the Java Native Interface. Furthermore, the sensor node software can be run either as compiled native code for the platform on which the simulator is run, or in a sensor node emulator that emulates an actual sensor node at the hardware level.

We show that the ability to perform cross-level simulations has several advantages over traditional simulations restricted to one level. We also show that COOJA is a suitable simulator for such cross-level simulations.

The remainder of this paper is structured as follows. We present the COOJA simulator in Section 2 and its implementation in Section 3. In Section 4 we evaluate the system. Related work is reviewed in Section 5. Finally, Section 6 summarizes our main conclusions.

## 2. COOJA

In this section we first present a high-level overview of COOJA before describing how COOJA achieves cross-level simulation. We demonstrate the flexibility of COOJA by describing COOJA's radio models and their usage during simulation.

COOJA is a flexible Java-based simulator designed for simulating networks of sensors running the Contiki operating system [1]. COOJA simulates networks of sensor nodes where each node can be of a different type; differing not

only in on-board software, but also in the simulated hardware. COOJA is flexible in that many parts of the simulator can be easily replaced or extended with additional functionality. Example parts that can be extended include the simulated radio medium, simulated node hardware, and plug-ins for simulated input/output.

A simulated node in COOJA has three basic properties: its data memory, the node type, and its hardware peripherals. The node type may be shared between several nodes and determines properties common to all these nodes. For example, nodes of the same type run the same program code on the same simulated hardware peripherals. And nodes of the same type are initialized with the same data memory. During execution, however, nodes' data memories will eventually differ due to e.g. different external inputs.

COOJA currently is able to execute Contiki programs in two different ways. Either by running the program code as compiled native code directly on the host CPU, or by running compiled program code in an instruction-level TI MSP430 emulator. COOJA is also able to simulate non-Contiki nodes, such as nodes implemented in Java or even nodes running another operating system. All different approaches have advantages as well as disadvantages. Java-based nodes enable much faster simulations but do not run deployable code. Hence, they are useful for the development of e.g. distributed algorithms. Emulating nodes provides more fine-grained execution details compared to Java-based nodes or nodes running native code. Finally, native code simulations are more efficient than node emulations and still simulate deployable code. Since the need of abstraction in a heterogeneous simulated network may differ between the different simulated nodes, there are advantages in combining several different abstraction level in one simulation. For example, in a large simulated network a few nodes may be simulated at the hardware level while the rest are implemented at the pure Java level. Using this approach combines the advantages of the different levels. The simulation is faster than when emulating all nodes, but at the same time enables a user to receive fine-grained execution details from the few emulated nodes.

COOJA executes native code by making Java Native Interface (JNI) calls from the Java environment to a compiled Contiki system. The Contiki system consists of the entire Contiki core, pre-selected user processes, and a set of special simulation glue drivers. This makes it possible to deploy and simulate the same code without any modifications, minimizing the delay between simulation and deployment.

The Java simulator has full control over the memory of simulated nodes. Hence the simulator may at all times view or change Contiki process variables, enabling very dynamic interaction possibilities from the simulator. Another interesting consequence of using JNI is the ability to debug Contiki code using any regular debugger, such as gdb, by attaching it to the entire Java simulator and breaking when the JNI call is performed. Also entire simulation states may be saved and later restored, skipping back simulations over time.

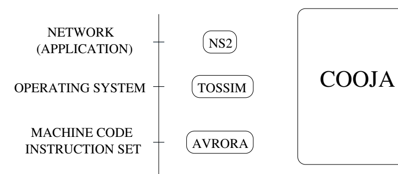



**Figure 1. COOJA can simultaneously simulate at several levels.**

The hardware peripherals of simulated nodes are called *interfaces*, and enable the Java simulator to detect and trigger events such as incoming radio traffic or a LED being lit. Interfaces also represent properties of simulated nodes such as positions that the actual node is not aware of.

All interactions with simulations and simulated nodes are performed via *plugins*. An example of a plugin is a simulation control that enables a user to start or pause a simulation. Both interfaces and plugins can easily be added to the simulator, enabling users to quickly add custom functionality for specific simulations.

## 2.1. Cross-Level Simulation

As depicted in Figure 1, COOJA allows for simultaneous simulations at three different levels, namely the networking (or application) level, the operating system level and the machine code instruction level.

Throughout this paper nodes at all different levels are said to be simulated, although nodes at the instruction level are hardware emulated since the executable is compiled for another architecture than the simulation platform. Nodes at the operating system level could similarly be called high-level emulated.

**Networking Level** During design and implementation of for example routing protocols, the specific hardware is often not as important as the networking itself. The most important factors may instead concern the radio medium, radio devices and perhaps sleep duty cycles of the sensor nodes. When performing such a design and implementation task it may be possible, but not necessary, to use a fine-grained simulation environment such as an instruction-level simulator.

COOJA supports code development by enabling the user to easily exchange certain simulator modules such as device drivers or radio medium modules. A simulation can be

saved and reloaded using other more or less detailed modules, still with the other simulation parameters unaltered. Furthermore, new radio mediums and interfaces such as radio devices can easily be developed in Java and be added to the COOJA simulation environment.

To further simplify and speed up development in such scenarios, COOJA also supports adding pure Java code nodes. Without any connection to Contiki, these can be useful when developing high-level algorithms which when tested and evaluated will be ported to deployable sensor node code. Pure Java nodes can also be used in heterogeneous networks where the user only needs to focus on a subset of all the simulated nodes. Since such Java nodes require less memory and processing power, larger heterogeneous network can be simulated more efficiently. For example, using Java nodes, users may rapidly implement the functionality of several different nodes together forming a network. And then later users can, node by node, port the Java code to deployable Contiki node code, while still maintaining full functionality of the network.

**Operating System Level** COOJA simulates the operating system by executing native operating system code as described in the previous section. As the entire Contiki OS, including any user processes, is executed it is also possible to alter Contiki core functionality. This is useful for example to test and evaluate changes in included Contiki libraries.

**Machine Code Instruction Set Level** Using COOJA, it is possible to create new nodes with a very different underlying structure than the typical nodes. We have evaluated this statement by adding nodes connected to a Java-based microcontroller emulator instead of a compiled Contiki system. The emulator represents an ESB node [7], emulating at the bit level.

The emulated nodes are controlled in a similar way as the native code nodes. Each simulated node is allowed to run for maximum a fixed period of time or long enough to handle one event. Events are then, by using the current node memory, transferred via the hardware interfaces to and from the simulator.

**COOJA's support for Cross-level Simulation** As explained earlier COOJA supports simulations at these three different abstraction levels. Note that the individual node is always simulated at *one* of these levels. The main advantage of COOJA's cross-level simulations is that nodes from each of the levels can co-exist and interact *in the same simulation*. Thus, for example, an emulated node can send a radio packet to a Java based node.

### 2.2. Radio Models

Each simulation in COOJA uses a radio model that characterizes radio wave propagation. New radio models may be added to the simulation environment. The radio model is chosen when a simulation is created. This enables a user to, for example, develop a network protocol using a simple radio model, and then testing it using a more realistic model, or even a custom made model to test the protocol in very specific network conditions. Often a radio model provides one or several plugins in order to configure and view the current simulated network conditions.

COOJA supports, except from a completely silent model, a simple model that uses an interference and a transmission range parameter that can be changed during a simulation run. Ongoing work on better radio models will provide COOJA with a general ray-tracing based model supporting radio absorbing material.

## 3. Implementation

This section discusses in more detail the implementation of the COOJA Simulator. The focus will be on how actual Contiki code is executed and controlled from the Java simulator as well as on how different important modules of COOJA communicate with each other.

### 3.1. Compiling Contiki for COOJA

The Contiki operating system is event-driven. Each handled event in the system is allowed to run to completion. This is a common approach well-suited for memory scarce devices such as sensor nodes and is also used in e.g. TinyOS [2]. COOJA exploits this property by calling the loaded Contiki system so that each simulated node only handles one event in turn. These calls all start in the simulation loop, where the simulator "ticks" each available node. During a node tick, both before and after the Contiki code actually executes, each node interface is allowed to check for any new events. For example a radio interface may have received new incoming data from the radio medium, and will make sure that the new data is discovered by the node software. Beside deciding which nodes should be allowed to act, the simulation loop also increases the overall simulation time and notifies the simulator that another loop has been performed.

Each Contiki system is always compiled for a specific hardware platform. The platform holds the available drivers and thus defines how to communicate with the actual hardware. When a node is emulated, for example in the MSP430 emulator, the Contiki system is still compiled for the MSP430 processor architecture. But when a node is to be simulated using the native code approach,

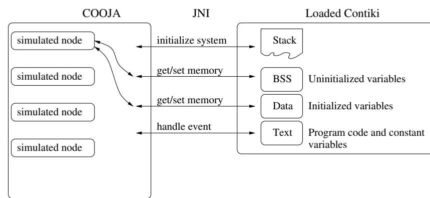



**Figure 2. Controlling Contiki by manipulating memory segments.**

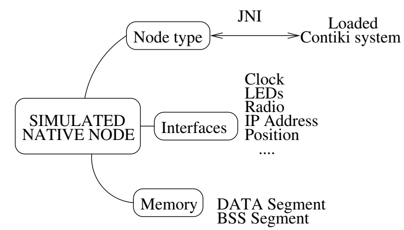



**Figure 3. The node type acts as a link to the loaded Contiki library.**

the Contiki system is compiled for a special glue simulation platform. This simulation platform offers a variety of drivers, enabling most Contiki processes to be compiled right away. The entire compilation and loading of the library is controlled from within the simulator, and a user may choose which processes and hardware peripherals (interfaces) should be available. The Contiki main source code file is then generated, compiled and loaded into the simulator.

### 3.2. Operating System Simulation

When simulating native code nodes, the node type acts as a link between a node and the compiled Contiki system. The node type loads the compiled shared library and all interaction between the simulator and the library is through this type. The node type only has a few functions by which it interacts with a loaded Contiki system. These include functions for copying and replacing memory segments, initializing the Contiki system and finally a function that tells the system to handle an event - to tick the node at the Contiki system level.

All nodes of the same type share the same program code memory - the Contiki operating system and a set of Contiki programs - whereas the data memory (program state) is different for each simulated sensor node. For every node, COOJA stores a copy of the memory of the C environment.

When a sensor node is scheduled to run, its data memory is copied into the C environment, and a JNI call is made to the event-driven Contiki kernel. The Contiki kernel dispatches a single event from its event queue to one of the processes running in the system, after which is returns control back to COOJA. At this point the potentially updated data memory is fetched back up to the Java environment.

For each loaded Contiki system, COOJA must be able to find the addresses of functions and variables. This is done by parsing the map-file generated at link-time. The map-file contains, among other, information about symbols' addresses in the library. By comparing the absolute memory address of a variable at runtime, to its relative address specified in the map-file, COOJA is able to calculate the addresses of all the library variables.

The rest of the parsed map-file allows the simulator to both fetch and alter variable values during simulations. This is the main way an interface or a plugin communicates with the Contiki system - they watch and alter values of certain variables. For example, a plugin could be written that watches a specific variable throughout a simulation and triggers some action depending on it.

Note that the only memory segments COOJA needs to copy from and to the loaded library, are the data and the BSS segments. Due to the memory constrained platforms the Contiki OS is designed for an event driven approach is used, where every event is allowed to run to completion. This, among other advantages, enables processes to share a single stack. Correctly written Contiki applications should thus never use the stack as a storage point between events. Hence there is no need for COOJA to save and restore the stack memory when simulating different nodes using the same loaded Contiki system. And since the text memory segment is identical for every simulated node of the same type, the only memory segments COOJA must copy are the data and the BSS segments (see Figure 2).

For an overview of the different parts of a simulated native node in COOJA see Figure 3.

### 3.3. COOJA's Configuration System

COOJA uses a configuration system that enables a user to alter parts of the simulation environment without changing any COOJA main code. The system can both be used for adding new parts such as interfaces, plugins and radio mediums, or to reconfigure existing parts. The new files are placed in project directories and can be activated and used from within the simulator.

### 3.4. Plugins and Interfaces

The typical interaction with a simulated node is via an available node interface. Each interface represents some property of the node, for example a radio or a position.

Since the node itself does not know its simulated position this interface has no need for any communication with the node memory and the underlying Contiki system. When the position of a node is altered, the position interface signals a change to its observers. Examples of such observers may be the current radio medium and an active node visualizer plugin.

The node interfaces that need to communicate with the running Contiki code typically have a corresponding part in the Contiki system, not only in the Java environment. The communication between these parts is always performed by manipulating the node memory. For example a button interface being clicked signals a flag in the node memory. When the memory has been copied to the Contiki system a corresponding button interface there discovers the click and informs Contiki about it the same way a regular hardware interrupt would. Similar to the position interface, the button interface also signals a change when the button is being pressed or released.

A plugin is used to interact with a simulation. Often it provides a user with some graphical interface and observes something interesting in the simulation. A plugin can be of a few different types, either it regards a simulation, or a node, or none of them (called a GUI plugin). A node plugin may for example watch some counter variable of a node and pause the simulation when it reaches 100. A simulation plugin may display the positions of all nodes in a simulation. And a GUI plugin may start new simulations, perform some tests, log the results and repeat.

Throughout COOJA an observer-observable approach is used. Several different parts of COOJA may have observers, for example the simulation state, the simulation loop, the radio medium and all node interfaces. This enables very dynamic interactions; for example the radio medium simply observes all position and radio interfaces.

# 4. Evaluation

We evaluate COOJA using four metrics: efficiency, scalability, flexibility and extensibility. We define the efficiency of the simulation as the running time for simulating a number of nodes running the same software. A number of factors influence the efficiency of a simulation in COOJA, of which two significant are the number of active plugins and the number of interfaces of each node. Interfaces are polled each time a node runs. Thus an increase in the number of interfaces should linearly increase the running time. Plugins themselves do not require processing power, but often register as listeners to simulations, nodes, node interfaces or even other plugins, thus increasing the running time. The actual increase depends on the activity of the observed object.

## 4.1. Simulation Performance

The efficiency of a simulation mainly depends on the number of nodes, the number of interfaces of each node and the number of simulation observers. As several plugins are called each time the simulation time increases, they may require relatively much processing power. One of those plugins is the standard simulation control, which allows a user to start and stop the simulation as well as see the current simulation time.

We simulated a network consisting of 100 native code nodes, all of the same type. The application executed on the nodes is very simple - it toggles all LEDs periodically, 10 times per second. Each simulated node has four interfaces; a position interface, an ID interface, a clock interface and a LED interface. One observer plugin is active throughout the simulation: the simulation control plugin. The simulation plugin is called once every simulation loop, when it updates a user interface widget showing the current simulation time. The simulated time is increased with 1 ms every completed simulation loop.

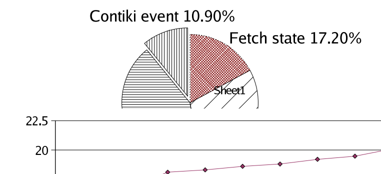
With the above settings we simulated the network for 180 simulated seconds (3 minutes). The running time is 5.87 seconds on a 2.4 MHz Pentium with 512 Mb RAM running Fedora Core 5.

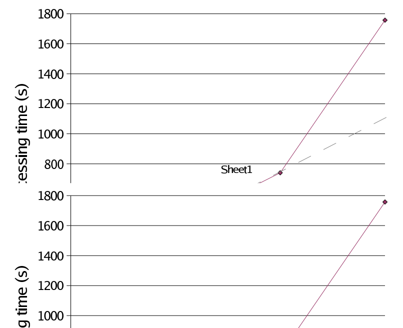
During the simulation the nodes sleep most of the time and only wake up when the LEDs are toggled. In order to analyze where the running time is spent during regular node ticks we disable the ability for the nodes to sleep. When a simulation is run with zero delay, approximately 98% of the time is spent ticking the nodes, and the rest on the single active plugin. The total simulation time increases from 5.87 seconds to 275.0 seconds when nodes are unable to sleep.

Figure 4 shows the relative running times for each node tick. The major parts of each node tick are polling the interfaces (30.1%), copying the memory back and forth (41.8%), letting Contiki handle one event (10.9%) and fetching the new state from the Contiki system (17.2%). We see that the Contiki event handling part is very small compared to the other parts. This is partly due to the few events that are actually handled during these calls.

The running time also increase as the number of plugins observing any part of the simulation is increased. We add more copies of the same simulation control already active in the above simulation. Figure 5 shows the running time relative to the total number of active simulation controls. We can see that the running time increases linearly with the number of plugins, since they all perform the same tasks when notified by the simulator.

Apart from plugins and interfaces the major factor affecting the running time of a simulation is the number of simulated nodes. Figure 6 shows the running time with an increasing number of nodes.All nodes are of the same type.




Figu ng a reg- ular

Figu odes in- creas

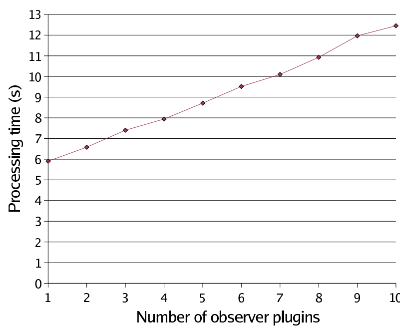



**Figure 5. The number of active plugins affects the running time.**

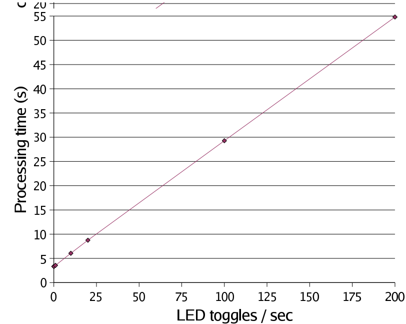



**Figure 7. The activity of each simulated node affect running times.**

The increase in running time is linear until 10000 nodes. At 10000 nodes the running time increase sharply which likely is due to the memory resources of the machine running the simulation being exhausted. With 20000 nodes the simulator crashed because of a shortage of Java heap memory.

The last performance measurement concerns the node activity. The simulation is faster if the nodes sleep. In Figure 7 the LED toggle rate is changed in order to show how different node activity affects the running time. Each node sleeps between the LED toggles, only waking up during a short period when it toggles the LEDs.

In summary, our results show that the running time of the simulation increases linearly with the number of nodes, the number of plugins, and when increasing the activity of the simulated program.

## 4.2. Cross-Level Simulation

To demonstrate the advantages of cross-level simulations we simulate a sensor network consisting of nodes simulated at all three different levels: the network level, the operating system level, and the machine code level. Each simulated node ran a simple process that toggle the LEDs once per second.

We compile the blinker application for the ESB platform and load it into the TI MSP430 emulator. We also implement a very simple Java-based node. The Java-based node imitates the Contiki blinker application by manually toggling its LEDs once per second. Finally, we load the blinker application into a native code node in COOJA. This setup enables us to compare the performance between the three different abstraction levels by measuring the running time of a fixed set of tasks.

As expected, the running times for simulations differ depending on which abstraction level is used. Also, the memory requirements differ between the three levels. Around 60 emulated nodes require the same amount of memory as 12000 native code nodes. The Java-based nodes require less memory than the native nodes.

In Figure 8 the different running times depending on the number of nodes are shown. On average a simulation with

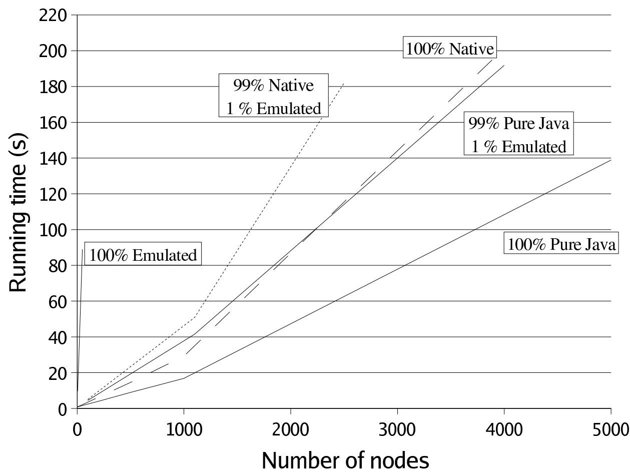



**Figure 8. Mixing nodes from different abstraction levels enable more effective simulations.**

only pure Java-based nodes is more than 1.7 times faster than the corresponding simulation with native code nodes, and more than 50 times faster than with emulated nodes.

When we combine nodes from several levels in the same simulation we keep some of the performance advantages of the higher levels but have the possibility to receive more fine-grained execution details from the rest. In Figure 8 running times of simulations with 1% emulated nodes are shown. For example, when using 1% emulated nodes and 99% pure Java-based nodes, we can simulate more than 2000 nodes (20 emulated, 1980 Java-based) in the same time as a simulation with only 50 emulated nodes.

### 4.3. Flexibility and Extensibility

One of COOJA's main advantages is the flexibility it provides to the user. To illustrate COOJA's flexibility we simulate a simple heterogeneous network, where several nodes gather information using a standard vibration sensor, and report this information via radio to a sink node. The sink node simply counts the number of events. In order to simulate this network we need two different node types, one running the source software and the other running the sink software. We also create a custom-made plugin to easier watch and control the status of the simulation.

In order to simulate new Contiki applications we first create a simple project directory where we put the application sources and a project configuration file. The configuration file enables the simulator to recognize and use sources in the project directory. After creating the two node types in COOJA and setting up a network, we may easily change the running Contiki processes on each node type. This enables us to switch between different versions of a process and speeds up development and testing phases.

Throughout the development, in order to test the processes, we can interact with the vibration interfaces on any of the source nodes to trigger vibration events and then watch the packets transferred in the radio medium. We can also watch the current value of the counter variable of the sink node. To customize the functionality we created a new plugin to control and watch the simulation. The plugin watches the counter value of the sink node and randomly triggers vibration events on the other nodes. When the counter reaches a fixed number the plugin pauses the simulation.

To be able to use the new plugin, the project directory only needs minor changes; the configuration file needs to be updated with a pointer to the plugin Java classes. Using the same approach new radio mediums and node interfaces can be added to the simulation environment.

## 5. Related Work

In this section we review existing simulators for wireless sensor networks. Different simulators allow for simulation at different levels of the system; at the instruction set level, the operating system level, or at the network level. COOJA differs from current simulators in that COOJA is not restricted to a single level, but allows for simultaneous simulation of a holistic wireless sensor network at different levels at once.

### 5.1. Network Level Simulators

Wittenburg and Schiller [11] have ported the Scatterweb API to the ns-2 simulator. This way, applications can be run both in the simulated environment and on real hardware without modifying the source code. While using a popular simulator such as ns-2 is very useful for simulating at the network level, extending their approach to include e.g. an operating system level simulator is not possible since ns-2 is a pure network simulator.

Also, SensorSim uses ns-2 as its base but extends is in several ways by including for example battery models, sensing channels and sensor models as well as protocol stacks for sensors nodes [5].

Other network level simulators used for the simulation of wireless sensor networks include GloMoSim [12] and OMNet++ [10].

### 5.2. Operating System Level Simulators

TOSSIM [4] is a simulator for the event-driven TinyOS operating system [2]. The main difference between

TOSSIM and COOJA is how several nodes are represented in the different simulators. In TOSSIM, the problem of simulating several nodes is solved by changing the sensor node code. All variables are replaced with arrays, where each element in an array belongs to a corresponding node. This is done automatically when the code is compiled for the simulator environment with the result that all nodes are simulated in the same process. In COOJA, the executed code remains unchanged, and when simulating several nodes (of the same type) all of these are executed one by one in the same process. Which node is currently active is identified only by the current process memory; different sets are copied back and forth when switching between nodes.

PowerTOSSIM [8] is an extension to TOSSIM for estimating per-node power consumption. However, both TOSSIM or PowerTOSSIM only supports simulations of nodes at the operating system level.

### 5.3. Instruction Level Simulators

The simulator/emulator ATEMU [3] (ATmel EMUlator) uses a hybrid approach; the operations of individual nodes are emulated and the communication between them is simulated. The emulation supports the MICA2 platform but can be extended to support other platforms.

Avrora [9] is another sensor network simulator simulating nodes at the instruction-level. Avrora achieves better scalability than ATEMU while still maintaining accuracy. In contrast to these approaches, COOJA is also able to simulate nodes at several different levels such as the network level and the operating system level.

## 6. Conclusions

In this paper, we have presented COOJA, a cross-level simulator for the Contiki operating system. COOJA enables simultaneous simulations at the network, operating system and machine code instruction set level. We have shown that cross-level simulation has advantages in terms of effectiveness and memory usage. It allows a user to combine simulated nodes from several different abstraction levels. This is especially useful in heterogeneous networks where fine-grained execution details are only needed for a subset of the simulated nodes.

## Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems, and the European Commission under contract IST-004536-RUNES.

## References

[1] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, Nov. 2004.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[3] M. Karir. Atemu - sensor network emulator / simulator / debugger. Technical report, Center for Satellite and Communication Networks, Univ. of Maryland, 2003.

[4] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 126–137, 2003.

[5] S. Park, A. Savvides, and M. B. Srivastava. Sensorsim: A simulation framework for sensor networks. In *International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Boston, MA, USA, Aug. 2000.

[6] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *ACM SenSys*, pages 255–267, 2005.

[7] J. Schiller, H. Ritter, A. Liers, and T. Voigt. Scatterweb - Low Power Nodes and Energy Aware Routing. In *Hawaii International Conference on System Sciences*, Hawaii, USA, Jan. 2005.

[8] 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 ACM SenSys'04*, 2004.

[9] B. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *International Conference on Information Processing in Sensor Networks (IPSN)*, 2005.

[10] A. Varga. The omnet++ discrete event simulation system. In *European Simulation Multiconference*, Prague, Czech Republic, June 2001.

[11] G. Wittenburg and J. Schiller. Running realworld software on simulated wireless sensor nodes. In *Proc. of the ACM Workshop on Real-World Wireless Sensor Networks (ACM REALWSN'06)*, Uppsala, Sweden, June 2006.

[12] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.