SICS Technical Report
T2006:05

ISSN 1100-3154
ISRN:SICS-T–2006/05-SE

A Sensor Network Simulator for the Contiki OS

Fredrik Österlind
fros@sics.se

February 2006

Abstract

This report introduces a new sensor network simulator for the Contiki OS - the COOJA Simulator.

The Contiki OS is a portable operating system designed specifically for resource limited devices such as sensor nodes. It is built around an event-driven kernel but supports pre-emptive multithreading at a per-process basis. It also supports a full TCP/IP stack via uIP and the programming abstraction Protothreads.

The main design goal of the COOJA Simulator is extendibility, for which Interfaces and Plugins are used. An Interface represents a sensor node property or device, such as a position, a button or a radio transmitter. A Plugin is used to interact with a simulation, for example to control the simulation speed or to watch all network traffic between the simulated nodes. Both new Plugins and Interfaces can easily be created and added to the simulation environment. A number of other parts of the simulator, for example a radio medium responsible for forwarding radio network data, can also be implemented and added to the simulator. And by supporting several different simulation environments at the same time in one simulation, different underlying hardware platforms can be simulated in heterogeneous networks.

Java Native Interface is used to connect the new simulator with Contiki, allowing simulated applications to run in a real Contiki system. By using this approach, any simulated application can then be run on a real sensor node unaltered.

Keywords: Sensor Networks, Contiki, simulator.

CONTENTS

1 Introduction **1**
- 1.1 Problem formulation . . . 1
- 1.2 Approach, Work phases and Results . . . 2
- 1.3 Report structure . . . 2

2 Background **4**
- 2.1 About Wireless Sensor Networks . . . 4
- 2.2 The Contiki Operating System . . . 5
  - 2.2.1 Event-based kernel . . . 5
  - 2.2.2 Threads and Protothreads . . . 5
  - 2.2.3 The system loop . . . 5
  - 2.2.4 uIP . . . 6
  - 2.2.5 Previous Contiki Simulator . . . 7
- 2.3 Process memory structures . . . 7
- 2.4 Java Native Interface . . . 8

3 Related work **10**
- 3.1 TOSSIM and TinyViz . . . 10
- 3.2 PowerTOSSIM . . . 10
- 3.3 ATEMU . . . 11

4 Design choices **12**
- 4.1 Extendibility . . . 12
- 4.2 Java and JNI . . . 12
- 4.3 Application simulator . . . 13
- 4.4 Automated Contiki environment setup . . . 13

5 Implementation **15**
- 5.1 Design overview . . . 15
- 5.2 Core communications . . . 16
- 5.3 COOJA interfaces . . . 17
- 5.4 Ticking a node . . . 19
- 5.5 COOJA plugins . . . 19
- 5.6 Code generation and compilation . . . 21
- 5.7 Graphical User Interface . . . 21
- 5.8 Radio Mediums . . . 22
- 5.9 Positioners and IP distributors . . . 22
- 5.10 Configuration system . . . 23

5.11 Extended simulation environment 23
5.12 An extensive example 24
5.12.1 The Contiki application 24
5.12.2 Creating the node type 25
5.12.3 Adding the nodes 25
5.12.4 Starting the simulation 25
5.12.5 Clicking the button 26
5.12.6 Sending the radio packet 28
5.12.7 Forwarding the radio packet 28
5.12.8 Turning on the leds 28

6 Evaluation **30**

7 Results and Future work **32**
7.1 Results 32
7.2 Future work 32

BIBLIOGRAPHY **33**

APPENDIX **34**

LIST OF FIGURES

2.1 uIP's global buffer usage 6
2.2 Previous Contiki Simulator screenshot 7
2.3 The typical memory structure of a process 8

4.1 Device drivers on a hardware vs. simulation platform 13

5.1 The simulation loop 15
5.2 A simulated sensor node 16
5.3 Java Native Interface usage 17
5.4 The connection between simulation and core interfaces 19
5.5 Pseudo-code of a node tick from the simulation loop to the core 19
5.6 COOJA screenshot with started plugins 20
5.7 Compilation of a new node type 22
5.8 Compilation with or without a user platform 23
5.9 Example - creating the node type 25
5.10 Example - adding 10 nodes 26
5.11 Example - starting the simulation 26
5.12 Example - clicking a button 27
5.13 Example - radio medium forwarding packet 28
5.14 Example - the resulting leds 29
5.15 Example - summary of events 29

CHAPTER
ONE

Introduction

An interesting research area which has gained a lot of attention lately is that of **Wireless Sensor Networks** (WSN). Communication and electronics advances have opened up the possibility to use these sensor networks in real-life situations, and there are numerous possible application areas; for example military, medicine, disaster areas or smart homes.

A wireless sensor network consists of small devices with sensing abilities, connected over the air via radio. The devices, called sensor nodes, often have limited resources in terms of energy, memory and processing power. Given these circumstances and the problems that arise, a lot of research is being performed in this area, and interesting questions concern both hardware and software. For example, which communication protocols are most energy-efficient and enable the longest network lifetime, or how should a sensor node operating system be designed?

When working with sensor networks, simulators are very useful. Developing, testing and debugging code against an intended hardware is hard and time-consuming work. Deployment of code and setup of networks are expensive, both in terms of time and money. By using simulators code development time can be shortened - the code is uploaded only once finished. And the resulting programs can be evaluated in highly customized simulated surroundings not feasible for real-life tests.

The **Swedish Institute of Computer Science** (SICS) conducts research in the area of WSN, ranging from theoretical research such as sensor network lifetime modelling to real applications developed together with industry. This requires a good development and deployment environment.

One of the contributions from SICS in this area is the **Contiki Operating System**, a lightweight and portable operating system designed for constrained environments such as typical sensor nodes. Some interesting features of Contiki are the support for TCP/IP and pre-emptive multithreading on a per-process basis.

1.1 Problem formulation

The purpose of this master thesis is to design and implement a simulator for Contiki OS - the **COOJA Simulator** (**Co**ntiki **OS** **Ja**va). Since the purpose of using a simulator varies greatly between users, development phases and what kind of application is simulated, the main design goal is extendibility.

A basic version must support the typical sensor node platforms, but more important, new devices must be easily added by users. Interactions with simulations must be customizable, from setup of networks to advanced analysis of certain properties of a communication protocol. In fact, implementing new functionality should be encouraged by

the design of the simulator. The main benefit of using the simulator should be during code development and testing phases, as well as larger-scale tests such as communication protocol behaviour. Other design goals are a usable, flexible and scalable simulator that fits into the surrounding development and deployment environment.

1.2 Approach, Work phases and Results

This master thesis can be divided into the following overlapping work phases:

- study
- design
- implementation
- test
- release
- report

The thesis began with gaining knowledge about Contiki as well as existing sensor network simulators. The existing simulators have solved similar problems in many different ways depending on what they aim to accomplish, may it be efficiency, scalability etc., and there is much to be gained from studying their solutions. Early on, the main focus for COOJA was set on extendibility, and eventually a simulator design was formed. During the implementation phase, the system has been redesigned several times and additional studying has been needed.

Both Windows and Unix-based systems have been used the entire implementation phase. This is mainly because of the many external tools that are used from inside the program, something that often causes problems when switching operating systems at a late stage of development. By developing on both systems simultaneously, compability is ensured from the start.

During the test and release phase, the COOJA Simulator has been demonstrated at several occasions. Persons from several companies, such as Ericsson, SICS, KTH, EmwiTech and ABB, have seen and tested the simulator. The response has been very positive and the discussions have led to valuable feedback and suggestions, which in many cases have been integrated into the resulting product. The COOJA Simulator is currently being used by SICS in ongoing research, where both the basic functionality as well as the extendible design are exploited and appreciated.

1.3 Report structure

This report is structured into background, related work, design choices, implementation and evaluation. Please note that it is not intended as a user manual. For detailed information and usage guides suitable for both users and developers, a user manual is available at `http://www.sics.se/~fros/cooja`.

The *background* chapter will give the reader an understanding of techniques used during development, relevant to the implementation chapter. It is followed by *related work*, which compares features and techniques of the COOJA Simulator with a few other common sensor network simulators.

In the *design choices* chapter, overall design choices such as programming language and simulator features are explained and motivated, to later be evaluated and discussed in the *evaluation* chapter.

The *implementation* chapter's purpose is to explain how the simulator actually works. The report ends with results and future work.

CHAPTER
TWO

Background

This chapter will introduce various techniques and systems used in and during development of the COOJA Simulator, directly relevant for understanding the implementation chapter.

2.1 About Wireless Sensor Networks

There is a wide range of application areas in which sensor networks can be used. Military applications include surveillance and reconnaissance, and in the health area sensor nodes could help monitor and aid patients. As deployment time can be very short, sensor networks can be used for monitoring disaster areas. Another example is to use sensor nodes in smart homes, as alarms or controlling devices [1].

Advances in wireless communication and electronics enable development of cheaper and smaller sensor nodes. Basically, a sensor node has processing power, wireless communication abilities and sensing devices. But to minimize costs they are often very limited, a typical sensor node has a short communication range, low initial amount of energy and low processing power. Of these, the especially limiting factor is the energy source, as that often is the main factor corresponding to the length of the sensor node life [1].

A sensor network may consist of up to thousands or even millions of sensor nodes, densely deployed and without pre-determined positions. The network has to be fault tolerant, scalable and of low production costs. These circumstances demand a new set of ad-hoc protocols and algorithms to be developed. Those used in traditional networks are often not well suited for sensor networks, for reasons such as the larger amount of nodes in communication range, high failure rates and limited resources of individual nodes [1].

Sensor networks are very application-specific, different networks' ideal nodes may differ in both hardware and algorithms. And of course, sensor node hardware can be constructed in several different ways. A simulator should therefore be adjustable to easily simulate different software as well as different underlying hardware platforms.

Also, similar wireless sensor networks may include different kinds of sensor nodes, with different purposes. As an example consider a network with two different types of nodes, a *cheap* type and an *expensive* type. The cheap nodes are simple, they only gather temperature data and forwards it to the nearest expensive node. The expensive nodes analyse all data from their cheap node neighbours, and present the average temperature on the Internet. Note that these sensor node types differ not only in their running software, but also in their hardware platforms. To simulate the above example, a simulator must support such heterogeneous networks, with several node types differing in both hardware and software.

2.2 The Contiki Operating System

Contiki is an operating system designed for memory constrained environments, such as the nodes used in WSN. It is built around an event-driven kernel, and features include dynamic loading and unloading of individual programs and services, and optional per-process pre-emptive multi-threading. It also supports a full TCP/IP stack via the uIP library, as well as the programming abstraction Protothreads. Contiki is implemented in the C language and has been designed to be easily portable to new platforms. It has been ported to more then 20 different platforms since its release 2003 [3].

2.2.1 Event-based kernel

In a purely event-based system, a process is implemented as an event handler, letting different blocks of code execute depending on which event is given. These blocks are always allowed to run to completion once called. Since a single code block will never be interrupted, these blocks can be designed so that they may all share the same stack. Compared to a multi-threaded model this requires less memory and computation overhead when having several concurrent processes. In Contiki, a process consists of an event handler and an optional poll handler function. The Contiki kernel holds the event scheduler that dispatches events to processes and periodically polls processes that registered a poll handler function. It uses a single stack for all processes, which is rewound between each invocation of an event handler [6].

2.2.2 Threads and Protothreads

Unlike most other event-based systems, Contiki supports pre-emptive multithreading [6]. This is performed via a library which can optionally be linked with programs that require it. This allows nodes to run applications that normally do not fit well in purely event-based systems, such as cryptographic computations. Such a computation would otherwise occupy the entire systems for a long time [6].

Apart from pre-emptive threads, Contiki also supports *Protothreads* [7]. Normally when writing programs for event-driven systems, these have to be written as explicit state machines with a resulting code that can be hard to understand and maintain. *Protothreads* is a programming abstraction used on top of these systems, with the purpose to simplify implementations of high-level functionality. By using Protothreads, programs can perform conditional blocking without the overhead of regular threads; Protothreads are stackless and require only 2 bytes of RAM each [7]. A regular Contiki process consists of one single Protothread [2].

2.2.3 The system loop

At a regular start-up of Contiki the system, among other things, initializes a few processes. It then repeatedly calls the system function `process_run()` [2]. This function calls all registered poll handlers, and then processes one event from the current system event queue. After the single event is processed, the function returns the number of unhandled events still in the queue. If the event queue is emptied, a system may choose to go to sleep in order to save energy. If an external interrupt awakes the system later, it loops around the same function `process_run()` again and until all new events are handled.

```
int
main(void)
{
  ...
```

```
  beep();

  while(1) {
    /*    watchdog_restart();*/
    while(process_run() > 0);
    LPM_SLEEP();
}

return 0;
}
```

The above code snippet is from the Contiki main source file of a specific hardware platform, and demonstrates a common usage of `process_run()`. The call to `LPM_SLEEP()` returns when the node wakes up again, often due to an external interrupt triggers calling `LPM_AWAKE()`.

2.2.4 uIP

`uIP` (micro IP) is a small TCP/IP implementation suitable for sensor nodes and other resource limited devices. It is designed to have only the absolute minimum of required features for a full TCP/IP stack, and focuses on the TCP, ICMP and IP protocols [4]. uIP uses a *single global buffer* for holding packets, large enough to contain only one maximum sized packet. When a new data packet arrives from the network, the network device driver puts it in the global buffer and calls upon uIP to handle the new data. After analysing the incoming packet data, uIP notifies the intended application. Because of the single buffer, this application must act on it right away to avoid the data being overwritten by another incoming packet. At this point the application may also choose to immediately send a response using the same global buffer. Similarly, when an application wants to send data, it passes a pointer to the data as well as the length to uIP, which writes the headers and finally calls the network device driver to send the packet out on the network. Figure 2.1 shows an overview of the connections between the device driver, uIP and the application, and how they use the global buffer.

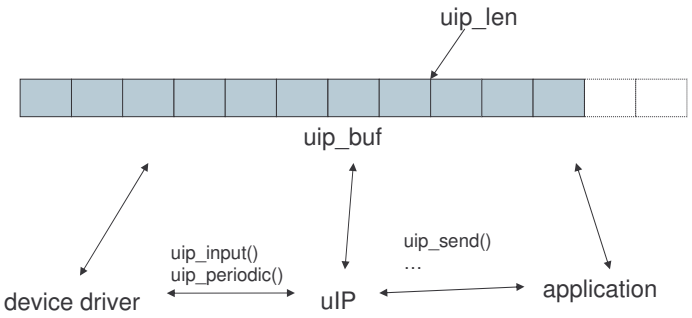

Figure 2.1: uIP's global buffer usage

The global buffer variable is named `uip_buf`, and its current data content size is stored in the integer variable `uip_len` [2]. The device driver operates against uIP by two functions. The first function, `uip_input()`, should be called when the device driver has received a packet and stored it in the global buffer. When it returns, the device

driver must check if there now is any outbound packet in the buffer. The second function, `uip_periodic()`, should be called periodically by the device driver to discover if any retransmissions are needed. The application may use several different functions to operate against uIP, among them the simple send data function named `uip_send` [2].

2.2.5 Previous Contiki Simulator

Previous to this master thesis, a basic Contiki simulation environment already existed. Contiki was ported to run as a user-level process in FreeBSD, each simulated node was represented by its own started process, and they were all connected by a simulated network layer [5]. However, since all nodes were represented by their own running process without any synchronization, the resulting simulations were not deterministic. In order to evaluate and compare different protocols simulations must be reproducible. Also, there was no support for heterogeneous networks and adding new functionality required a lot of work. Figure 2.2 shows a screenshot of the simulator.

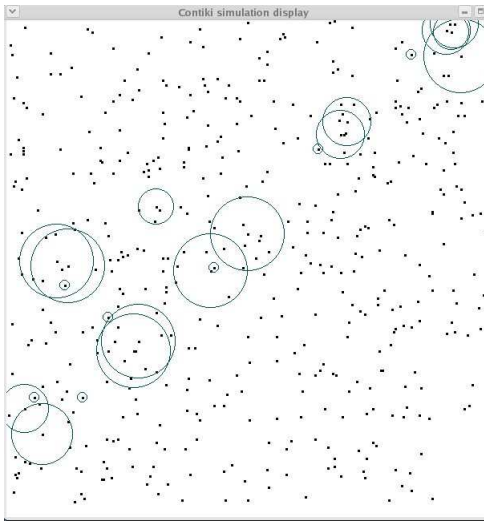

Figure 2.2: Previous Contiki Simulator screenshot

2.3 Process memory structures

A regular process memory consists of several different memory areas, or memory sections. Each section is a range of addresses without gaps and all data in a section is treated the same. Which sections exist varies between platforms but simplified there are at least three sections, these are the "‘text"’, "‘data"’ and "‘bss"’ section. The text section holds the program code and constants, and is usually unalterable when the program is executing. The data section holds initialized variables and is alterable. The bss section is similar to the data section but holds uninitialized variables. The reason for having a bss section is to save space in compiled binaries. Since all of the data in the section is zeroed when the program is started, only the length of the section has to be saved in the binary, not all the zeroes.

Additional memory areas are the heap and the stack. The heap enables dynamic memory. During an execution a program may want to allocate new memory, and this memory is placed in the heap. The stack is used for storing local non-static variables and function call parameters. See Figure 2.3 for an overview of the typical memory structure of a process.

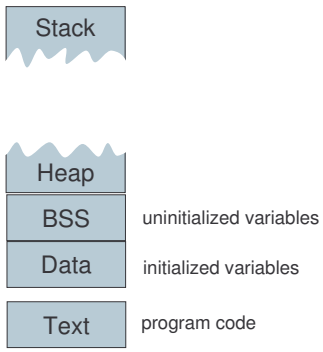

Figure 2.3: The typical memory structure of a process

2.4 Java Native Interface

Java Native Interface (JNI) is built into the Java virtual machine (VM) and provides a way to locate and invoke native methods on a platform. This way code running inside the JVM can interoperate with applications written in other programming languages such as C or Assembly. Reasons for using JNI may be to reuse libraries and APIs not implemented in Java, or to speed up calculations by using Assembly code.

A library is loaded in a Java class using the `System.loadLibrary` method, and certain Java methods, *native*, are then mapped to functions in the library. As a simple example we may have a native method "‘tick"’ in the class "‘Lib1"’ residing in package "‘se.sics.cooja.corecomm"’. The Java side code would look something like this:

```
package se.sics.cooja.corecomm;
...
public class Lib1 ... {
static {
    System.loadLibrary("mySharedLibrary");
  }
  native void tick();
}
```

Then the corresponding C function in the library would look like the following:

```
JNIEXPORT void JNICALL
Java_se_sics_cooja_corecomm_Lib1_tick(JNIEnv *env, jobject obj)
{
 ...
 ...
}
```

As can be seen in the Java code the library identified by "'mySharedLibrary"' is loaded. Which actual file this corresponds to is platform specific; usually on Unix/Linux it is mapped to "'libmySharedLibrary.so"' and on Windows "'mySharedLibrary.dll"'. The C function name is constructed by the string Java, the packages, the class name and finally the method name. In this example, every call to the Java method `tick` is forwarded to the corresponding C function `Java_se_sics_cooja_corecomm_Lib1_tick`.

In the Java 2 SDK, the VM maintains a list of every loaded native library for each class loader. Once loaded, a library will not be unloaded until the class loader is garbage collected.

Several libraries can be loaded by the same class loader if they are named differently. However, once loaded the function names will be "'occupied"' and any later loaded library cannot implement functions of the same signature. Looking at the example above, assume we also want to load another library "'mySharedLibrary2"', which implements another tick function with the same signature. If both of these libraries were loaded, only the first library's native function would be executed by calling the native Java tick method. This implies that if several libraries implementing different functions must be loaded, either the function names must be different ("'tick1"', "'tick2"' etc) or they must be loaded from different classes (which also generates a different signature).

CHAPTER

THREE

Related work

Several simulators exist that are either adjusted or developed specifically for wireless sensor networks. The COOJA Simulator's main contribution is that of an application simulator for *the Contiki OS*, with the purpose to ease and speed-up development and testing phases. It is designed to be extendible and usable, with less focus on efficiency. A few commonly used simulators are discussed in this chapter, to point out similar and distinct features.

3.1 TOSSIM and TinyViz

The simulator TOSSIM [9] is in many ways similar to COOJA. It simulates the event-driven operating system TinyOS. The visualization tool TinyViz provides a graphical user interface, and is connected to TOSSIM over a TCP socket. In COOJA, the simulator and the user interface are not separated, and the corresponding communication between the executed Contiki code and the surroundings is via JNI. (This is explained in greater detail in 5.2.) Both TOSSIM and COOJA support adding plugins for customized simulation control as well as selecting different or creating new radio mediums.

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 [9].

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. Although the latter approach requires more overhead during node-switches, it gives a less complex structure where the code executed during a simulation is equal to the code executed on a real node. This makes for example debugging simulated code very intuitive. And since the entire memory of each node is available in the simulator at all times, advanced memory interactions are made possible such as saving a node memory and restoring it at a later time.

3.2 PowerTOSSIM

PowerTOSSIM [10] is an extension to TOSSIM for estimating per-node power consumption. Each TinyOS component corresponding to a hardware peripheral (such as a LED),

reports its actions during the simulation. This is later translated to how much energy the device required. For simulating the CPU power consumption, PowerTOSSIM uses a code-transformation technique to estimate the number of CPU cycles of executed code blocks [10]. The generation and processing of the power data is decoupled due to efficiency and flexibility; the gathered data is analysed offline to calculate for example how much energy the node used during the simulation.

In COOJA, *interfaces* (explained in 5.3) represent the simulated hardware peripherals and report how much energy they use during simulations. But opposite to PowerTOSSIM there is no analysis of the code executing on the simulated CPU, the estimated power consumption of the CPU is instead a value only depending on what current energy state the node is in (dead, low power mode or active). Also, the energy calculations are performed directly in COOJA since the focus on efficiency is not as central. An advantage of performing calculations immediately is to test how protocols perform when central nodes run out of energy and are therefore shut down. The network then has to find new routes.

3.3 ATEMU

The simulator/emulator ATEMU [8] (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 [8]. COOJA does not emulate nodes; a better description is that COOJA simulates Contiki applications, compiled for a port of Contiki. The port for which Contiki applications are compiled can instead easily be adjusted to represent a target hardware platform, having different hardware devices and power consumption settings.

CHAPTER
FOUR

Design choices

In this chapter overall design goals and choices will be discussed. The different parts of the simulator that are mentioned here will be explained in more detail in the next chapter.

4.1 Extendibility

The purpose of simulating a sensor network differs greatly between different simulations. During development and implementation of a routing protocol, a user wants to confirm that the code is executing properly. When the protocol is implemented, factors such as network lifetime or total number of packets sent become more important.

But the same user may also want to investigate some protocol specifics or properties. For example, consider a routing protocol that needs an algorithm which automatically figures out the relative positions of all nodes in a network. A user may then want to graphically show these positions in comparison with their "'real"' simulated positions. Therefore, while already having support for base-functionality in the simulator, it is important that a user can easily extend that functionality, concerning both hardware and software.

To achieve such extendibility the simulator mainly uses two different parts - **plugins** and **interfaces**, both of which can easily be created and added to the simulator. The plugins interact with a simulation, or parts of a simulation. An interface interacts with a sensor node - it simulates hardware peripherals or monitors interesting values. Both of these will be explained in Chapter 5.

4.2 Java and JNI

Common programming languages for similar simulators such as COOJA include C and Java. Java offers a quick and easy way both to develop and to later extend applications, hence a good choice for the COOJA Simulator.

The ability to compile and execute real Contiki OS code in the simulator minimizes the step between simulating and actually running the code on real platforms. Java Native Interface (JNI) offers a connection between the C language of Contiki code and the pure Java part of the simulator. But by using JNI, the otherwise platform independent Java part will now need a platform that can compile and execute Contiki code. An alternative could be using a TCP connection between the Java part and a proxy calling the compiled Contiki OS code. This way the Java part would still be platform independent and simulations could be run remotely on any platform able to compile Contiki and the proxy.

After comparing the different alternatives, JNI was chosen mainly due to extendibility reasons. For example, when using JNI the entire simulation is run in the same process. A COOJA plugin (explained in 5.5) is able to, without adding any new intelligence to the JNI connection or any other part of the simulator, start up a common debugger in a new process, set relevant breakpoints and then start debugging Contiki code running in the simulator process. The same thing would be impossible with a TCP connection without adding debugging support to the interface between the Java part and the proxy. Hence, in this case a regular user can easier add new functionality when JNI is used. Also, to run a simulation on two different hosts over a TCP socket would introduce a major performance bottleneck, which means that in most cases the Java part would still need to run on the same platform as the native code.

4.3 Application simulator

The COOJA Simulator is a Contiki OS application simulator as opposed to a sensor node emulator. When a user creates a new node type, the resulting Contiki system is compiled for the simulation platform, in the regular Contiki environment. While the drivers of a hardware platform operate on hardware, the drivers of the simulation platform work against the Java part of the simulator. But since the simulation platform supports the same devices as any hardware platform, there is no difference between them from an application viewpoint. The same application code compiles and executes on both platforms. So the hardware peripherals of a platform are not emulated, they are replaced by other simulated devices. An alternative would be to emulate all hardware of a sensor node, which could give more precise results but also would limit the simulator to one or a couple of supported hardware platforms. By using the above method, different hardware platforms can be supported simply by puzzling together all wanted drivers. Figure 4.1 shows a comparison of device drivers on a hardware platform and in the simulator. The left part illustrates how an application uses a driver to interact with hardware, whereas in the right part the driver pretends to be hardware but instead communicates with the simulator.

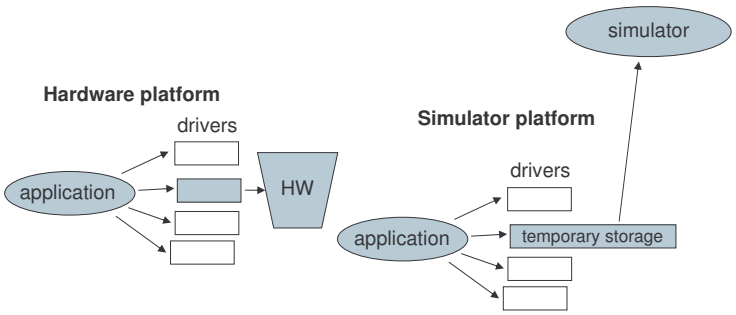

Figure 4.1: Device drivers on a hardware vs. simulation platform

4.4 Automated Contiki environment setup

In order to simplify usage, as much as possible should be controlled from inside the simulator.

This includes setting up and loading the Contiki system. For example, since the Contiki applications are central to the node types, and they are started from the Contiki main file at start-up, the main file should automatically be generated and compiled from the simulator. This way a user chooses which processes the node should run and then the simulator generates the needed source files, compiles them and finally loads the created library. Apart from the Contiki applications, the generated main file also specifies which interfaces and sensors each node has, as well as contains the entire JNI interface towards the Java part. All of this should be easily controlled and altered by the user.

CHAPTER

FIVE

Implementation

This chapter explains the main parts of the COOJA Simulator and how they are connected. Earlier in this report, Interfaces and Plugins have been mentioned. An Interface represents a sensor node property or hardware peripheral, whereas a Plugins is used to interact with a simulation. Both of these will be explained in this chapter, as well as how the simulator uses the process memory structure (2.3) to simulate several nodes of the same type.

An extended version of this chapter intended for developers is available in the user manual, available from `http://www.sics.se/~fros/cooja`. The chapter ends with an extensive example of what happens when the user interacts with a simulation. The example gives the reader an easier explanation of how the earlier discussed parts are connected.

5.1 Design overview

Simplified, a COOJA simulation consists of a number of nodes being simulated. Each node is connected to a node type, or "'of a node type"'. When the simulation is running, all of the nodes get to act in turn. And when all nodes have acted once the simulation time is updated and then the process is repeated (the simulation loop). See Figure 5.1.

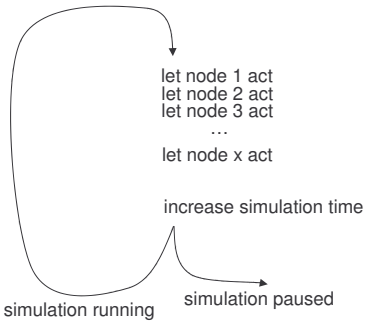

Figure 5.1: The simulation loop

More specifically, each node also has its own node memory and a number of node interfaces. The memory consists of one or several memory segments, each with a start address and data. Together the memory segments must define all interesting and needed parts of an entire simulated Contiki OS (explained in 5.2).

The interfaces act on the memory and simulate node devices such as a clock or a radio transmitter. For example, when the time changes a clock interface should update some specific time variable. And that variable resides in the node memory of that node.

The node type is the bridge between the node explained above, and a loaded Contiki OS executing node specific code. This is from where the simulated Contiki OS ("‘the core"’) is initialized, and the initial memory is created. And all nodes of the same type are linked to the same loaded Contiki OS. The node type also performs variable name to address mapping. This implies that if the above clock interface wants to change the core time variable "‘timevar"’, the node type is asked what address that variable is at. When a node gets to act, the node type is responsible for linking the node to its corresponding Contiki OS. See Figure 5.2 (note that the node type can also be connected to other nodes).

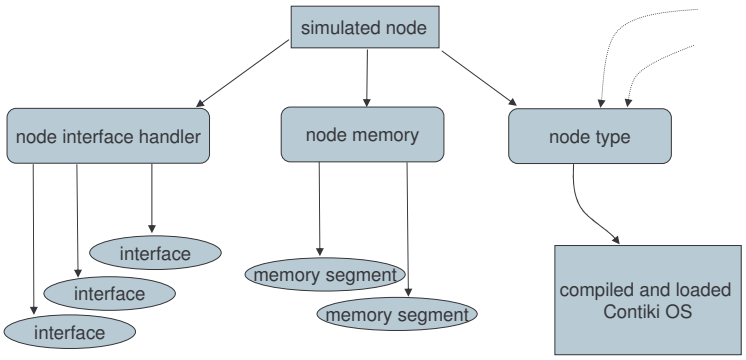

Figure 5.2: A simulated sensor node

There is one running Contiki OS for each existing node type. This means that all nodes of the same type share the same Contiki OS. It consists of Contiki code compiled as a shared library towards a simulation platform, and the communication between the Java part and the core is via Java Native Interface. Observe the difference between a Contiki OS process and a Contiki application process; the first one is the entire Contiki operating system compiled and executed, while the latter is a process existing only inside the Contiki environment. When referring to the Contiki OS process the term "‘core"’ will be used, and when referring to the rest of the simulator the term "‘simulator"’ will be used. Obviously the core is also a part of the simulator but to simplify the text this distinction will be made. There are only a few native functions connecting the core to the simulator, of which the most important three are a "‘set memory"’ function, a "‘get memory"’ function and a function that is called when a node gets to act (the "‘tick"’ function - see below).

The simulator is pure Java based while the core is implemented in the C language (as Contiki OS itself). As a rule, the simulator is responsible for everything external to a specific node such as how the surrounding radio medium works or the current simulation time. The core is responsible for the inner workings of the node; it executes real Contiki code, allowing the same simulated application code to be used unaltered on real physical nodes. The process of letting a node act will be called letting a node "‘tick"’, or "‘ticking"’ a node, and is explained in 5.4.

5.2 Core communications

A Contiki application is always compiled for a specific platform, such as the ESB platform. The platform defines, among other things, which devices (leds, buttons etc.) are available.

When Contiki is compiled to be used in the COOJA Simulator (the platform "'cooja"'), the main source file defines a number of JNI functions. On a "'regular"' Contiki platform, this main part processes events from the event queue until empty. In COOJA, a node is initialized at startup, and then all the processing of events is performed remotely via the JNI functions. There are only five JNI functions, these are an **initialization** function, a **get memory** function, a **set memory** function, a function that returns the **absolute memory address** of a special reference variable and finally a **tick** function. The initialization function starts up the process handler, networking and the pre-specified application processes, much like on any other platform. Get and set memory functions are very simple, without any error checking they just set or return a specified byte array from the current process memory.

The function which returns an absolute address of a reference variable is used for mapping between relative and absolute memory addresses. Each node type knows the relative addresses of all memory sections and variables in its core. When the simulator switches between different nodes, all of the node relevant process memory has to be replaced. COOJA does this by copying the entire BSS and DATA memory sections back and forth between the simulator and the core (the stack and heap sections are not needed, as explained in 2.2). By comparing the reference variable absolute address with the known relative address (via node type), the absolute addresses of the memory sections (as well as all variables) can be calculated. See Figure 5.3 for an illustration of all the JNI functions and their purposes.

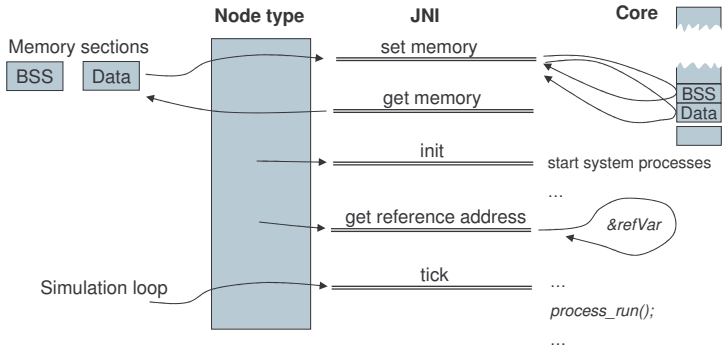

Figure 5.3: Java Native Interface usage

The last JNI function is the tick function. This is as the name suggests called during a node tick, and this is where the Contiki system function `process_run()` (that processes an event from the event queue) is called. For more information about the JNI tick function see Section 5.4.

5.3 COOJA interfaces

The COOJA interfaces are the main and preferred way to analyse and interact with simulated nodes. Not only do they simulate all the hardware devices, but by exploiting the way the interfaces are treated, highly customized simulation behaviour can be achieved. Instead of strictly being interfaces to node devices, a better explanation may be that they are interfaces to node properties. For example, one of those properties is the node position. Often a node does not even know its actual position - these kinds of interfaces are called virtual. By customizing the position interface, a simulation with moving nodes can be

created. This could be useful if one were to simulate nodes rolling down a hill or on moving robot arms.

COOJA interfaces exist both in the core and in the simulator. Interfaces implemented in the simulator ("'simulation interfaces"') have full access to the node memory, and interfaces implemented in the core ("'core interfaces"') can access Contiki system functions. Often relationships and dependencies exist between core and simulator interfaces. One example is the radio transmitter, a radio interface must exist both in the simulator and in the core. When radio data is transmitted or received the interfaces communicate with *each other* rather than with the Contiki OS system directly. When the node is ticked, the core interface can then deliver incoming radio data to the Contiki system the same way as a regular hardware device driver would, by storing it in the global buffer and calling `uip_input()` (see 2.2.4).

A distinction of simulator interfaces can also be made; active and passive, and has to do with the *node state*. The only node property not handled via interfaces is the node state, which is either active, sleeping (to save energy) or dead. A dead node still exists, but will never be ticked and can never leave that state. When a sleeping node is ticked, only passive interfaces are allowed to act, and the tick is not delivered to the core. An example of an interface that should be passive is the battery interface. That interface must act even though the node is sleeping (the node still requires battery energy). On the other hand, the active interface representing a PIR sensor may not discover light changes if the node is sleeping. Active interfaces may still wake up a sleeping node by triggering an external interrupt. A button interface is such an example - it is active but wakes up a sleeping node whenever the button is pressed.

When a node interface wants access to a variable, it is the node type that actually performs the mapping between the variable name and the memory address. To get an extendable design, each interface is responsible for its own dependencies. Thus, when an interface is constructed, it should check that all needed variables and interfaces are available.

All simulation interfaces can also be *observed*. Any entity of the simulator can register as an observer, and is notified whenever the interface decides so. For example, a radio interface may choose to notify its observers when it is about to send data. A standard observer of radios is the radio medium, it can then fetch the new data and decide which of the other radios should receive it.

The observer-observable approach enables very dynamic interactions between different parts of the simulator, and not only simulation interfaces use this approach. For example a simulation can be observed, it notifies all observers when the simulation is started or paused, when new node types have been created or when nodes have been added or removed. The simulation loop can also be observed, it notifies its observers whenever it has completed one loop.

A simulation interface optionally also has a graphical representation, where it offers information and interaction with a user. These graphical representations are implemented as regular Java panels and can be displayed in a COOJA plugin (see Section 5.5). Figure 5.4 shows how simulation and core interfaces are connected. As can be seen in the figure, interfaces only interact with the node memory. When that node memory later is copied to the core, the core interfaces can discover any changes made.

Which interfaces a certain node type has is chosen at compilation time, see Section 5.6 for more information. The easiest way is to select which simulation interfaces should be supported, all dependency core interface will automatically be added (although these highly recommended interfaces also can be manually configured).

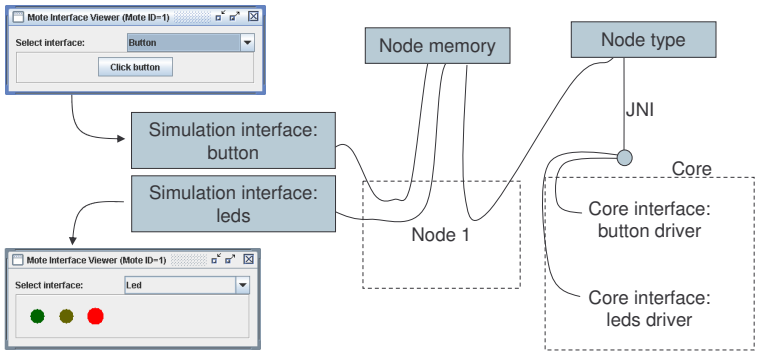

Figure 5.4: The connection between simulation and core interfaces

5.4 Ticking a node

During a node tick the Contiki system function `process_run()` is called once, which handles one event and polls a process once. In order to tick the "'right node"', the memory of that node has to be set before going down to the core. Also both before and after this function is run node interfaces are allowed to act, which ones depend on the node state as discussed earlier.

Information about pending events and timers are returned from the core back to the simulator. This is used for deciding if a node should go to sleep. Figure 5.5 contains pseudo-code following a tick from the simulation loop down to the core.

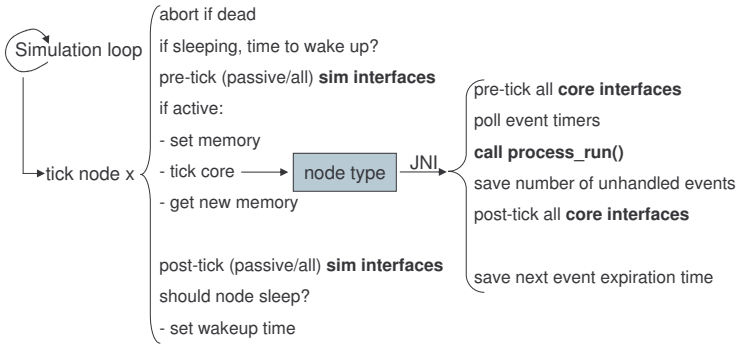

Figure 5.5: Pseudo-code of a node tick from the simulation loop to the core

5.5 COOJA plugins

Whereas the COOJA interfaces are the best way to interact with simulated nodes, plugins are the best way for a user to interact with a simulation. The plugins are registered at runtime before they can be used, often at startup of the simulator. The user then creates instances of the available registered plugins during simulations. The plugins are implemented like a regular Java panel, and hence a user can create new advanced graphical interfaces in a straight-forward way. Plugins can be of four different types, and they are treated slightly different.

The first and simplest type is the **GUI plugin type**. The GUI plugin type only needs a running GUI to be constructed, and this is passed as an argument when a user initializes the plugin. Via the GUI, relevant information such as the current simulation (if any) as well as all simulated nodes can be accessed. Since this plugin only depends on the GUI, it is not removed when the current simulation is removed. Plugins of this type are the ones that can outlive a simulation. And since only one simulation can be active at the same time in COOJA, this plugin type may be used to transfer information between different simulations. An example of a GUI plugin may be a testrun controller which loads up a simulation, performs some tests, saves the data and loads another simulation.

The second and third types both depend on a simulation, the **Simulation plugin types**. When such a plugin is created, the current simulation is passed as an argument, and if that simulation is removed, so will the plugin be. An example of a simulation plugin may be displaying information about the current active simulation, such as the number of simulated nodes and types or the current simulation status. Another very useful example is a graphical representation of the positions of all simulated nodes. The difference between the two simulation plugin types is that one of them (called the simulation standard plugin) will automatically be created when a new simulation is created, the other is optionally started by a user.

The last plugin type depends on a simulated node, and if that node is removed, so will the plugin be. It is called the **Mote plugin type**. An example of this plugin may be to watch a certain node variable, and stop the simulation whenever that variable changes. Another example is an energy usage history plugin, which monitors the amount of energy left in the battery of a node and presents a graph over time using this data. See Figure 5.6 for a screenshot with some different types of plugins started.

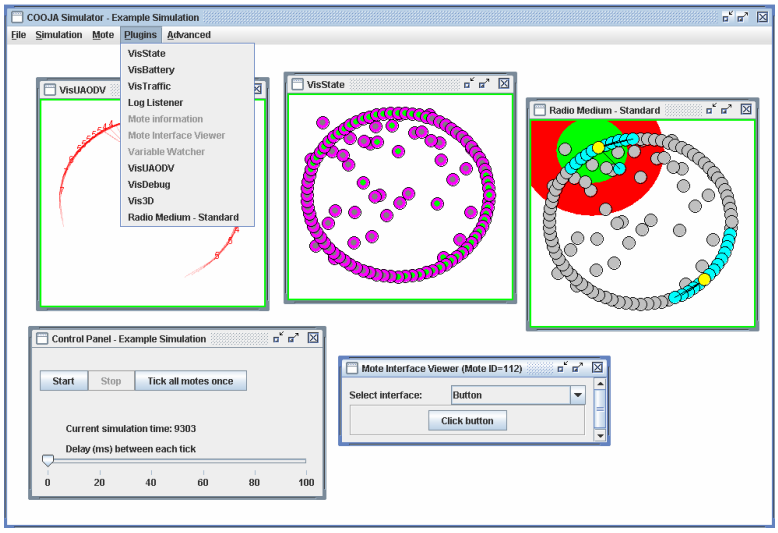

Figure 5.6: COOJA screenshot with started plugins

Most plugin types should be loaded at startup and be available until COOJA is terminated. However, COOJA also supports registering and un-registering plugins - depending on the current simulation. These plugins are called **dynamic plugins** and the only difference is the way they are registered, i.e. any plugin type can be registered as a dynamic plugin. The reason for supporting dynamic plugins is that some parts of a simulation may want to register a plugin of their own. An example is radio mediums (see Section 5.8).

When a radio medium is used, that is the only part in COOJA responsible for and with information about how radio data should be handled - which nodes can reach and interfere with other nodes, transmission bit errors etc. And since radio mediums can be very advanced, simply implementing a generic visual interface to all of them is not enough. By using dynamic plugins, a radio medium can register one or several plugins of its own and let a user see or alter radio medium specific parameters using them. In Figure 5.6 the radio medium used is named "'Radio Medium - Standard"', and it registered a dynamic plugin of the same name. This plugin simply allows a user to view and change transmission and interference ranges.

5.6 Code generation and compilation

A node type is defined by a number of parameters such as which Contiki processes are loaded at start-up, which standard sensors the Contiki environment knows about and which core interfaces have been registered. Another important factor is which Contiki environment the library is compiled against, different node types may use different versions of Contiki. When a user creates a new node type he must enter the Contiki OS path, against which the shared library will be compiled. Available processes, sensors and interfaces are automatically detected by scanning the simulation platform and can be chosen from. These settings define how the resulting Contiki main source file will look.

When the configuring is done, COOJA generates the source file and compiles it. The output from this compilation is a map file and the shared library with the JNI functions (see Section 5.2). The map file is outputted from the linker and is a text file that contains names and addresses of all memory sections and variables of the compiled library. This file is parsed when the node type is created, and enables variable address lookups.

The compiled library corresponds to the newly created node type. But due to JNI (see Section 2.4), each new library must be loaded from a unique class, they can not all be loaded from a single node type class. COOJA supports a limited number of simultaneously loaded libraries each via its own class, called library classes. Since the contribution of a library class only is its unique name, it is extremely simple. It has the native functions and when constructed it just loads a given library file. During node type creation, before the library is compiled, a free library class is allocated and the JNI functions in the source file are named with regard to that class. See Figure 5.7.

5.7 Graphical User Interface

The core Graphical User Interface (GUI) is kept as simple as possible. It is based on a desktop pane and by using menus and dialogs a user may create new, load or save simulations. The user can also create new node types, add nodes, change simulator settings and create instances of registered plugins.

When loading a simulation, stored values are recommended to the user, but can be altered. For example, if a user loads a simulation with two different node types and 100 nodes, he must recompile both node types before the nodes are added. At this point the user may for example choose to change which Contiki processes are run on each node. This can be useful if he wants to compare different algorithms while using the same network setup, or how routing protocols behave when different radio mediums are used.

All windows in the desktop pane are started plugins, so without plugins there is no way to interact with a simulation, except for creating and adding nodes. A number of plugins that offer basic functionality is implemented in the basic version of COOJA. This

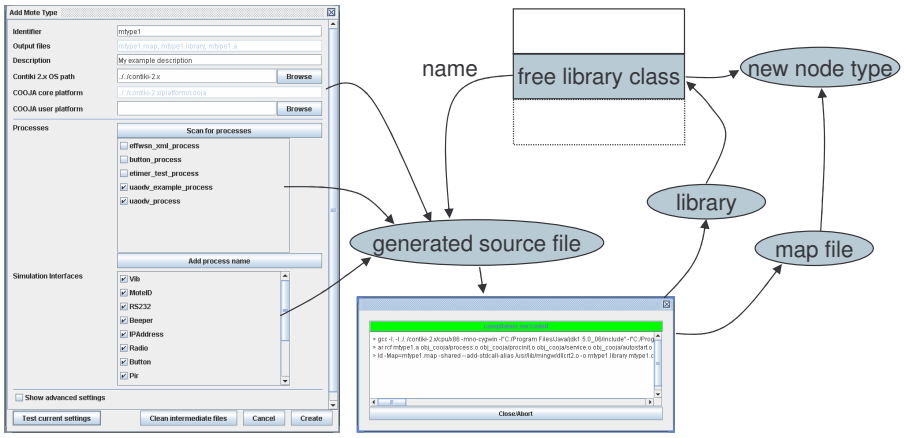

Figure 5.7: Compilation of a new node type

includes viewing and moving nodes, observing radio traffic, watching node interfaces and node log outputs as well as controlling simulations.

5.8 Radio Mediums

As network communication is central to a WSN, the COOJA Simulator supports adding and using different radio mediums.

Each radio medium is listening to a number of registered radio transmitters. Every radio transmitter has its own position, and the radio medium discovers whenever new data arrives at any of the radios. It is then responsible for calculating which of the other radios should hear the data. And the current radio medium used during a simulation is the only part forwarding radio data in the network. By using this approach a radio medium can be made arbitrarily advanced. For example bit errors can be introduced depending on distance, or obstacles can be simulated in order to block or corrupt data between different radios. As mentioned above, to increase user interaction each radio medium may register dynamic COOJA plugins.

In the basic version of COOJA there are only two simple radio mediums implemented and registered at the simulator start-up. The first one is completely silent, no data is transferred. The second uses two distance parameters, one for transmission range and the other for interference range. Data packets actually delivered to a receiving radio device are never corrupted in any way.

5.9 Positioners and IP distributors

Positioners and IP distributors are both registered the same way as radio mediums - this means that it is easy to create and add new ones. They are used when new nodes are added to a simulation, and enable a user to distribute positions and IP addresses using customized schemes.

Positioners determine node positions; included in the basic COOJA is random, linear and positioning in an ellipse shaped form. Examples of positioners may be to read positions from an external file, or to distribute them according to some algorithm.

IP distributors determine the node IP addresses, those already included are random, defined from a unique node ID and defined from the node position (spatial IP addressing).

5.10 Configuration system

The configuration system is used by the Java part of the simulator, including both the base simulator system and added implemented plugins. Configuration files are read at startup or during a node type creation, and by creating new a user may add extra or override the default configuration. The base simulator system reads for example which positioners, IP distributors, interfaces, plugins and radio mediums should be registered at start-up. The user may choose only between these when the simulator is started, but by loading another configuration file new ones can be made available.

An example of what a battery interface may use the configuration system for is to read its initial energy. In the same way a led interface may read how much energy it requires during its different states. This way, by only changing configuration files, the simulated hardware can differ significantly depending on which configuration files are loaded. A typical usage is to create different configuration files depending on which hardware platform should be simulated, either loading these files at simulation startup or when the node type is created.

5.11 Extended simulation environment

The preferred way to use COOJA is to create a customized working environment. The working environment, called user platform, is a directory where users can extend COOJA functionality without changing the base system. A working environment can add both Contiki implemented devices (in C language), and Java classes. This means that the simulated hardware platform can be extended to have new core interfaces (peripherals) depending only on which directory is chosen as current user platform.

At the Java side, plugins and simulation interfaces should extend certain abstract classes in the base system. When compiled against COOJA they can be loaded and used by the simulator, without the need to recompile the base system.

Which of the new interfaces and plugins are available up is specified the usual way - by the earlier explained configuration system. See Figure 5.8 for a comparison when creating node types with or without a user platform.

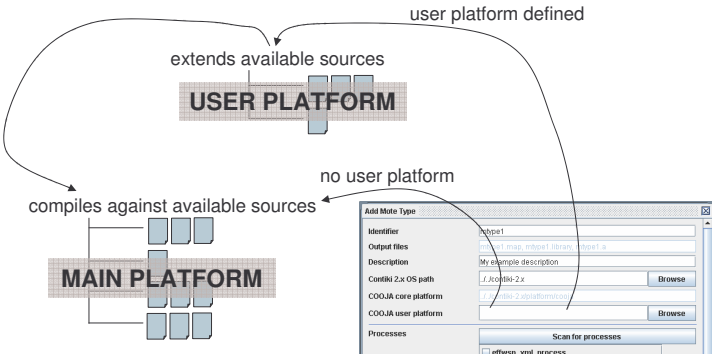

Figure 5.8: Compilation with or without a user platform

5.12 An extensive example

In this section a simple example will be run and explained. The simulation will contain several simulated nodes, all running the same Contiki application on the same simulated hardware. Note that this example is run using a specific configuration and the results would be different if another was used, such as other interfaces, plugins, radio mediums, uIP versions, MAC protocols etc.

The Contiki application is very simple - if the onboard button is pressed the node will send a network broadcast message to all other nodes. And when such a packet is received all three leds will be turned on.

5.12.1 The Contiki application

The Contiki process is implemented in `cooyah.c` using Protothreads, and is named `cooyah_example_process`. Pseudo code of the process body follows (for the real code see Appendix):

```
PROCESS BODY OF cooyah_example_process

LOG "Example process started"

ACTIVATE NETWORK
ACTIVATE BUTTON SENSOR

LOOP FOREVER {
- wait for incoming event

* if shut down event
  LOG "An event occurred: shutting down"
  shut down

* if button was pressed
  LOG "An event occurred: the button is pressed, sending packet"
  send network packet containing "cooyah COOJA"

* if button was released
  LOG "An event occurred: the button was released again, doing nothing"

* if network packet was received
  LOG "An event occurred: a packet was received, turning on leds"
  LOG "PACKET DATA:" + packet_data
  turn on all three leds
}

END OF PROCESS BODY
```

As can be seen in the code, four different events are handled; an exit event, a button down event, a button up event and a received network data event. The source file was placed in the main simulation platform in order for the simulator to find it.

5.12.2 Creating the node type

A simulation was created using the standard radio medium, and a node type was created with the above process selected (see Figure 5.9). All simulation interfaces were used, although only a few are really needed for the above application.

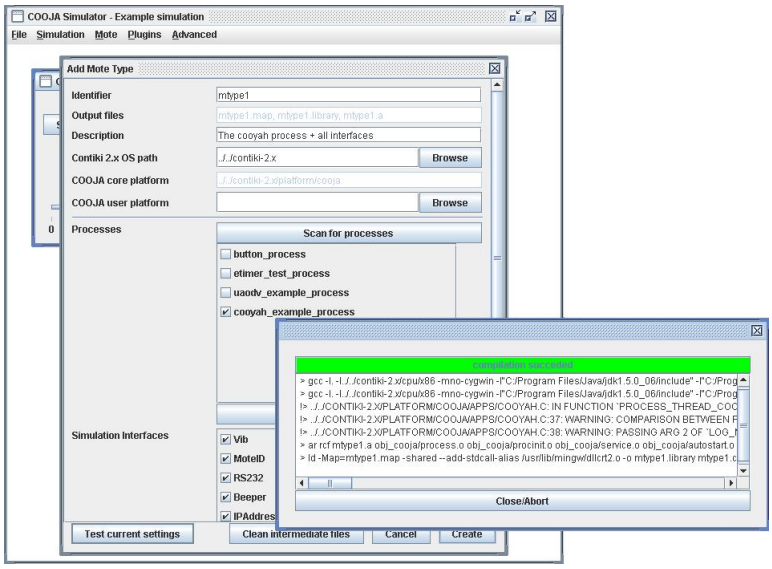

Figure 5.9: Example - creating the node type

When the "'Test current settings"'-button was pressed, the free library class "'Lib1"' was selected, and a Contiki main source code file was generated; mtype1.c. This file contains the JNI functions tuned to be called from the "'Lib1"'-class, all start-up processes including the above, and finally all core interfaces and sensors. It was compiled against the main simulation platform which resulted in both the library file and the map file. Later when the "'Create"'-button was pushed, the library class loaded the library "'mtype1.library"' and the node type was created. Also, the map file was loaded and the variable and memory section addresses parsed. Finally the JNI function "'init"' was called upon the newly created library and the current memory was fetched. This memory represents a newly started node, and is stored in the node type to be used when adding new nodes.

5.12.3 Adding the nodes

10 nodes of the above node type were randomly added. All of these nodes received the basic startup memory from the node type, and all interfaces specified in the node type were created for each of the new nodes. See Figure 5.10.

5.12.4 Starting the simulation

When the simulation is started by using the plugin "'Control Panel"', the simulation loop begins. After just a few ticks, the initial log messages (specified in the Contiki code above) can be seen in the plugin "'Log Listener"'. Also, all of the node states are set to be sleeping, because there are no unhandled events or pending event timers. Some of the simulation interfaces (the passive) are still allowed to act; one example is the battery interface. A sleeping node can now only be awoken by an external interrupt, such as a clicked button. In this simulation, incoming radio traffic also triggers external interrupts.

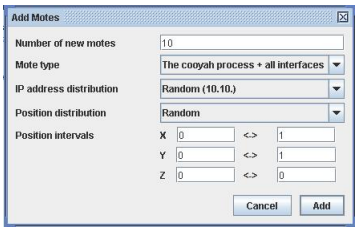

Figure 5.10: Example - adding 10 nodes

By using the plugin "'Mote Interface Viewer"' we can view and interact with simulation interfaces. In Figure 5.11 we can see that node 2 and node 8 both have all their leds turned off, and we also have the option to click the button of node 8.

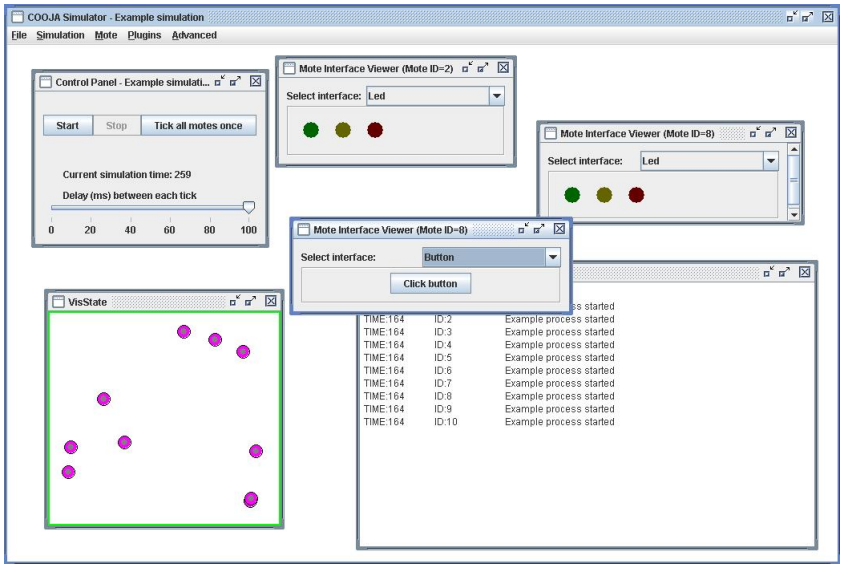

Figure 5.11: Example - starting the simulation

5.12.5 Clicking the button

Now we, by using the graphical representation of the button simulation interface, click the button on node 8. The interface operates on the node memory and changes the character variables "'simButtonIsDown"' to 1 and "'simButtonHasChanged"' to 1. It also changes the node state to active, imitating an external interrupt. The following Java code is from the button simulation interface, and is run when the button is pressed. (Note that this code also checks if the button sensor is activated at all.)

```
/**
 * Presses the button and flags a change.
 */
public void pressButton() {
  moteMem.setByteValueOf("simButtonIsDown", (byte) 1);

  if (moteMem.getByteValueOf("simButtonIsActive") == 1) {
```

```
      moteMem.setByteValueOf("simButtonChanged", (byte) 1);

      // If mote is inactive, wake it up
      if (RAISES_EXTERNAL_INTERRUPT)
        mote.setState(Mote.STATE_ACTIVE);
    }
  }
```

The simulation loop continues as before but now node 8 is active. This causes the tick to go down to the core, and there the corresponding button core interface is allowed to act. The following code is from the button core interface - the driver of the simulated button. Note that both the simulation interface and the core interface operate on the same variables.

```
doInterfaceActionsBeforeTick(void)
{
  // Check if button value has changed
  if (simButtonChanged && simButtonIsActive) {
    sensors_changed(&button_sensor);
    simButtonChanged = 0;
  }
}
```

The core interface now discovers that the simulated button has been pressed, and signals a change to the Contiki system - the exact same way a hardware driver would; by calling `sensors_changed(&button_sensor)`. The Contiki system adds an unhandled event to its list, and deals with it during the next `process_run()` function call, which is when all core interfaces have acted.

After a few ticks, that event has been passed on to the cooyah application which now sends a packet onto the network. The button simulation interface automatically releases the simulated button one tick after it has been pressed. That causes another event to be passed to the application, which can be seen in the "'Log Listener"' plugin, Figure 5.12.

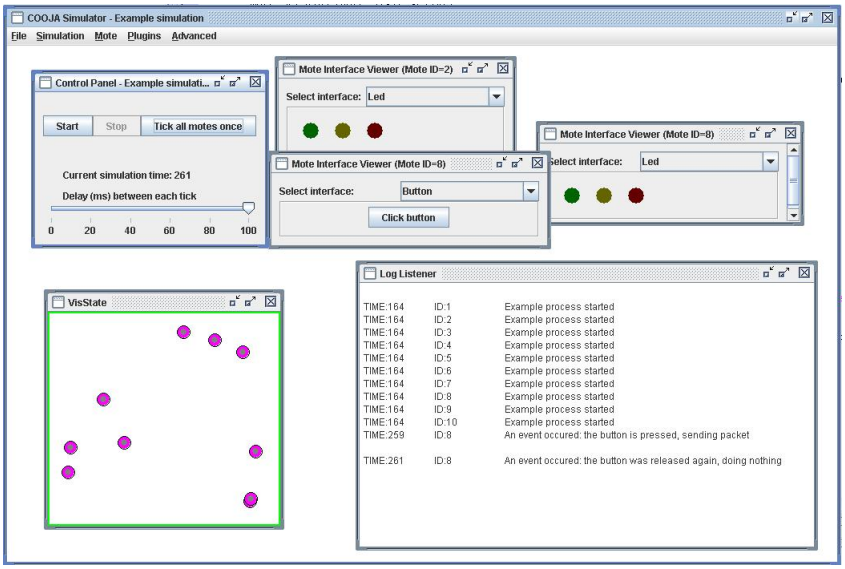

Figure 5.12: Example - clicking a button

5.12.6 Sending the radio packet

The cooyah application sends the packet by calling a uIP function:
`uip_send("cooyah COOJA", 12)`, which only copies the data into the global buffer. A few ticks later, the running Contiki default uIP process discovers that it has unhandled data in the global buffer. Eventually that data is passed down to the device driver; implemented in the core radio interface.

The radio core interface copies data from the global buffer to a buffer of its own. The same way the core and simulation button interfaces communicate with each other, this packet is later transferred to the radio simulation interface. And the simulation radio interface then notifies all its observers that new data is available.

5.12.7 Forwarding the radio packet

The radio medium used in this simulation has registered as an observer on all radio simulation interfaces. When the above interface received a packet, the radio medium was informed of this and fetched the new data. It then decides on which other radio interfaces should receive this data - in this example the two nearest neighbours; node 2 and node 5. In the plugin registered by the radio medium, "'Radio Medium - Standard"', the current transmission and interference ranges are visualized as well as the current connections. See the lower-left corner of Figure 5.13 for a screenshot of when the radio medium forwards the network packet.

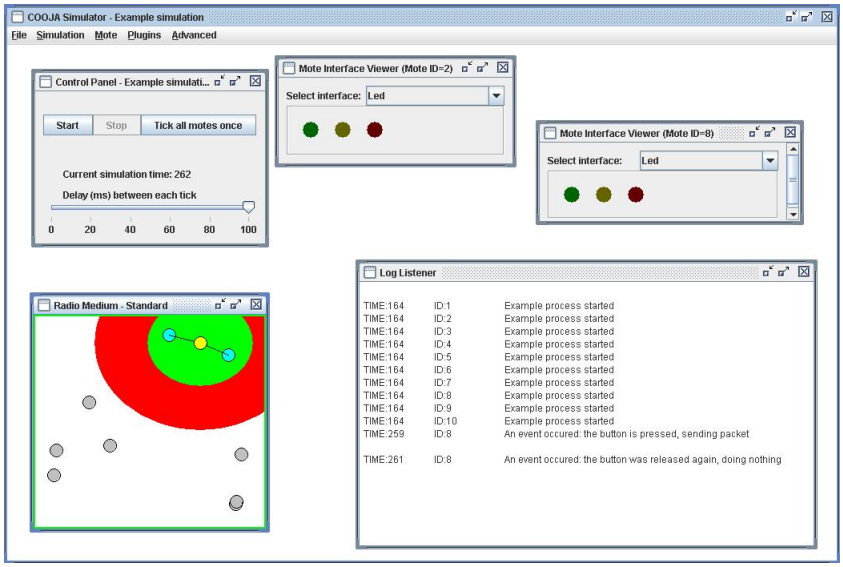

Figure 5.13: Example - radio medium forwarding packet

Since the radio interface was allowed to trigger external interrupts in this simulation, it will wake up both nodes 2 and 5. The received data is passed down to the radio core interfaces in the same fashion as before, and after being handled by the uIP process, the cooyah process is eventually given an "'incoming network data"' event.

5.12.8 Turning on the leds

The simulation and core interfaces representing leds are connected in a similar way as buttons, radios etc. In Figure 5.14 the resulting leds and logs are displayed. A summary of the events during this example can be seen in Figure 5.15.

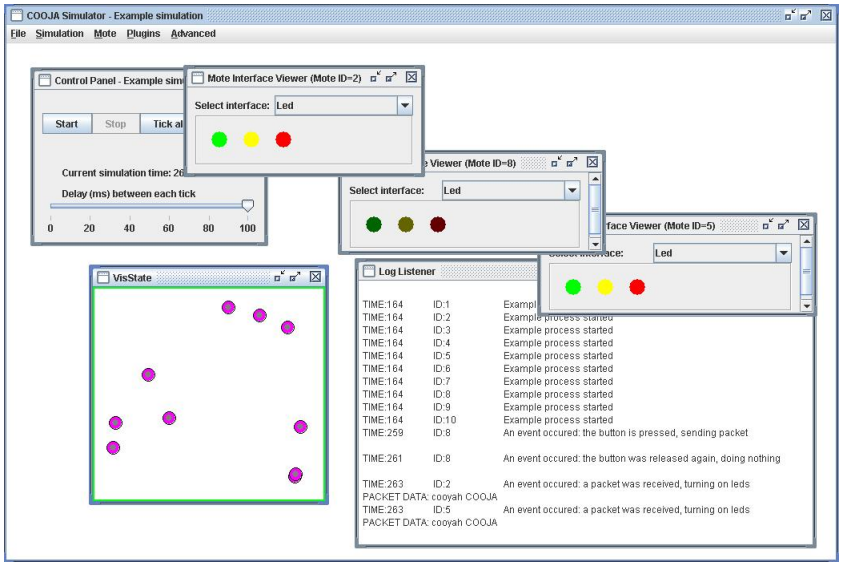

Figure 5.14: Example - the resulting leds

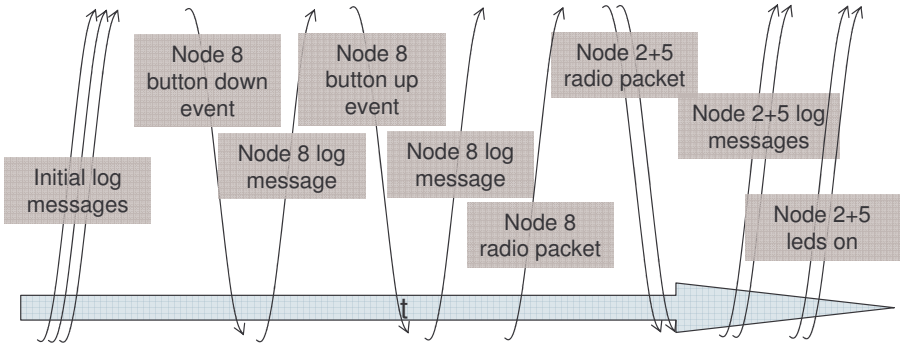

Figure 5.15: Example - summary of events

CHAPTER
SIX

Evaluation

In this chapter an overall evaluation of the simulator will be discussed, considering different potential users and the design choices in Chapter 4. As argued earlier, the simulator may be used for different purposes; a user may just want to test and debug some application code, develop entirely new applications or evaluate algorithms and routing protocols in highly customized simulated surroundings. There may also be a need for adding new functionality, such as new radio mediums or advanced interactions and analyses of simulated nodes and networks.

COOJA's main design goal is to be extendable, and at the same time usable. For a first-time user the windowing system should be easy and intuitive to understand and navigate in. When he creates a new node type, the simulator automatically finds his applications that he may select from. Since the basic simulator supports the most common hardware peripherals, chances are very good that his applications can be simulated right away. The simulator then generates sources, compiles and loads the generated library. When adding nodes he is presented with alternatives of how to position them, and how the IP addresses of the new nodes should be distributed. And plugins included in the basic simulator allows the user to interact with and view information about the simulation.

For the user that wants to create new applications, the create node type dialog may also be used to test-compile the code. However, since created node types never can be removed, and COOJA only supports a limited number of simultaneous node types, the simulator has to be restarted once and a while. A faster way to test code, and to avoid too many node types, is to after creating a node type and adding one or several nodes, save the current simulation. When the simulator is restarted and the previous simulation is loaded, all settings will already be preset which gives a short time-overhead between code changes.

The advanced user may add new customized interfaces, extending both hardware peripherals (core and simulation interfaces) and property interactions (virtual simulation interfaces only), without the need to change anything in the base simulator. He can create different configuration files, or even different user platforms, for each hardware node that is being simulated, and then simulate all of these different node types in the same simulation. By extending existing plugins, new functionality can be added quickly, or he may create entirely new plugins working with the usual Java tools for creating graphical user interfaces - a simulation interface is implemented the same way as a JPanel. And because of the observer-observable-approach, all parts of a simulation can be controlled. Finally, since simulated code is regular Contiki application code, the step between simulating and uploading applications to real nodes is small.

However, a drawback of the simulator is its relatively low efficiency. Simulating many

nodes with several interfaces each requires a lot of calculations, especially when plugins are started and registered as observers to those interfaces.

CHAPTER
SEVEN

Results and Future work

7.1 Results

The resulting simulator works well for the intended usage. It helps new users to quickly and easily start up a simulation, and is very useful during development and test phases. It supports heterogeneous networks, concerning both simulated hardware and software. Larger-scale behaviour protocols and algorithms can be observed by using the basic set of plugins or by easily extending them. However, due to its extendibility it is not extremely efficient.

7.2 Future work

An interesting future add-on to the simulator is a more advanced radio medium. The radio medium implemented in the basic simulator is very simple - it only depends on the distance between the radio devices. An interesting and more realistic radio medium could introduce radio absorbing obstacles such as walls and transmission error probabilities.

A feature that would improve the base system is the ability to "'skip to the next event"'. By temporarily disabling a user from triggering simulated external interrupts, the simulation would be a lot faster. Instead of ticking a node a thousand times, waiting for something to happen, the simulation would be sure that nothing could happen and immediately calculate how much energy is needed during those thousand ticks etc. This feature can also support only watching subsets of interfaces and nodes, as an example the simulation can run until the first node battery is dead.

Another useful add-on would be, implemented as a GUI plugin, a test case controller. By reading a pre-defined specification it loads and starts simulations, logs interesting data, loads another simulation and repeats. By using this plugin, extensive and time-demanding tests could be performed during nights and holidays.

To fully minimize the step between simulating and deploying code on real nodes, and since the current way of compiling and uploading code to different sensor node platforms is standardized in Contiki, this functionality can be added to the simulator, either as a base functionality or as a plugin. This way a user may, inside the simulator, choose to compile and upload the applications of a node type to a currently connected real node.

BIBLIOGRAPHY

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks, 2002. IEEE Commun. Mag. 40.

[2] A. Dunkels. The contiki operating system 2.x online documentation (webpage). `http://contiki.sourceforge.net/html/`; accessed January 22, 2006.

[3] A. Dunkels. The contiki operating system (webpage). `http://www.sics.se/~adam/contiki/`; accessed January 22, 2006.

[4] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys)*, San Francisco, May 2003.

[5] A. Dunkels, L. M. Feeney, B. Grönvall, and T. Voigt. An integrated approach to developing sensor network solutions. In *Proceedings of the Second International Workshop on Sensor and Actor Network Protocols and Applications*, Boston, Massachusetts, USA, Aug. 2004. Invited paper.

[6] 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.

[7] A. Dunkels, O. Schmidt, and T. Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[8] M. Karir. Atemu - sensor network emulator / simulator / debugger. Center for Satellite and Communication Networks, Univ. of Maryland, 2003. `http://www.isr.umd.edu/CSHCN/research/atemu/`.

[9] P. Levis. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[10] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In Sensys, 2004.

APPENDIX

Basic support of the COOJA Simulator

Positioners

- EllipsePositioner
- LinearPositioner
- RandomPositioner

IP Distributors

- IdIPDistributor
- RandomIPDistributor
- SpatialIPDistributor

Plugins

- LogListener
- MoteInformation
- MoteInterfaceViewer
- SimControl
- SimInformation
- VariableWatcher
- VisBattery
- VisState
- VisTraffic

Radio Mediums

- SilentRadioMedium
- StandardRadioMedium

Interfaces

- Battery (virtual)
- Beeper
- Button
- IPAddress
- Led
- Log
- MoteID
- Pir
- Position (virtual)
- RS232
- Radio
- Time
- Vib

Contiki application code - cooyah.c

```
#include <stdio.h>
#include <stdlib.h>
#include "net/uip.h"
#include "lib/sensors.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include "sys/log.h"
#include "node-id.h"

#define COOYAH_PORT 1234

PROCESS(cooyah_example_process, "Example process for report");
AUTOSTART_PROCESSES(&cooyah_example_process);

static struct uip_udp_conn *broadcast_conn;
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(cooyah_example_process, ev, data)
{
  PROCESS_BEGIN();

  log_message("Example process started", "");

  broadcast_conn = udp_broadcast_new(COOYAH_PORT , NULL);

  button_sensor.activate();
```

```
  while(1) {
    PROCESS_WAIT_EVENT();
    log_message("An event occurred: ", "");

    if(ev == PROCESS_EVENT_EXIT) {
      log_message("shutting down\n", "");
      break;
    }

    if(ev == sensors_event && data == &button_sensor && button_sensor.value()) {
      log_message("the button is pressed, sending packet\n", "");

      tcpip_poll_udp(broadcast_conn);
      PROCESS_WAIT_UNTIL(ev == tcpip_event && uip_poll());
      uip_send("cooyah COOJA", 12);
    }

    if(ev == sensors_event && data == &button_sensor && !button_sensor.value()) {
      log_message("the button was released again, doing nothing\n", "");
    }

    if(ev == tcpip_event && uip_newdata()) {
      log_message("a packet was received, turning on leds\n", "");
      log_message("PACKET DATA: ", uip_appdata);
      leds_on(LEDS_ALL);
    }

  }

  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```