Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Debugging Wireless Sensor Network Simulations

Semester Thesis

Richard Huber
rihuber@ee.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Advisor:
Philipp Sommer

Supervisor:
Prof. Dr. Roger Wattenhofer

May 18, 2011

Abstract

Applications for wireless sensor networks are often developed in integrated development environments and tested and debugged in simulators. Since development environments and simulators normally use their own user interface, it is often hard for developers to link a bug found in the simulation to the correct position in the source code.
This thesis comes up with a way to debug TinyOS applications that run as a simulation in the COOJA simulator directly in the Yeti development environment. With the possibility to set breakpoints in the nesC source code and to inspect the memory of each simulated mote, we expect that the presented tool increases the productivity of TinyOS application developers.

Keywords: wireless sensor network, debug, simulation, Yeti, COOJA

Contents

Chapter 1

Introduction

Developing applications for wireless sensor networks is a challenging task. The limited memory, bandwidth and energy available on the embedded devices force developers to come up with small and efficient software. Frameworks and operating systems, such as TinyOS[3] or Contiki[1], help application developers to meet these requirements. But the limited hardware on the target devices not only affects the needs of the software itself, it has also a deep impact on the development process.

One of the most limiting facts is that, apart from some buttons or LEDs, there is usually no rich user interface on the mote platforms. This makes it difficult to output detailed information about the ongoing processes on the target device during the test and debug phase. One possible way to solve this problem is to connect the device to a JTAG adapter. Using this JTAG adapter, a host computer may perform typical debugging tasks on the device, such as reading and manipulating register and memory entries, setting and removing breakpoints or single stepping through the program. The drawback of this method is, that it delivers only information about one specific device. The JTAG adapter can neither provide any information about other devices, nor about the traffic not seen by this device.

Another problem in the test and debug phase is the deployment of new software versions. Collecting all motes of the network, program the new firmware to each device and redistribute them again for every little bug-fix is just not feasible. There exist smarter solutions for this problem (see [12] for example) where firmware updates are deployed using the wireless network. However, all these solutions require a running network, so if a bug causes a mote to loose its connection to the network, this mote has to be reprogrammed by hand anyway.

Simulators provide a much more comfortable environment for debugging wireless sensor networks. In the case of a failure in a particular mote, the whole simulation may be stopped immediately and a detailed investigation may be applied to all motes participating in the simulation. Additionally, the deployment of a new software version in the simulation comes with very little effort compared to a real network and the complete radio traffic may be recorded, regardless whether

the individual packets were disturbed or not. Furthermore, the network may be stressed by the environment with well defined conditions, which may be difficult to set up in a real world experiment.
The main contribution of this thesis is to provide a mechanism for controlling and debugging a simulated network directly from within the development environment by extending the debugging support of the Yeti [10] Eclipse plugin. As a typical example for a simulator we used the COOJA simulator [6]. The benefit of this mechanism is that developers of TinyOS applications are now able to code and simulate their programs within the same environment. We expect, that this leads to more comfort for the developers and to a shorter development time.

1.1 Related Work

TOSSIM [8] is a simulator for TinyOS applications. It uses the event-driven nature of TinyOS to map the execution of the application to a discrete event simulation. The TinyOS application is compiled directly into the TOSSIM framework. In this compilation procedure some low-level TinyOS systems that interact with the hardware are replaced.
This approach increases the scalability of the simulation but has the drawback that the simulated application differs in some points from that one which is executed on the physical node. Especially flaws in the hardware access may stay unexposed due to this difference.

Marionette [12] is a tool suite for remote procedure calls that may be automatically added to a nesC application at compile time. It is used to include a PC in the wireless sensor network in order to observe and control the network via a rich user interface. The PC acts as a client in the RPC architecture and may call any function on a specific node and read or write any variable on the node's heap. This insight in the network comes at the price of a larger program and an increased network traffic.
Marionette partly solves the problem of the time consuming deployment of new software versions by applying the following approach: in a first phase, the core functionality of each node is executed on the PC and the result is then transmitted back to the corresponding node. Once the algorithm on the PC is validated, it is shifted to the embedded device. This reduces the number of software updates that have to be deployed in the whole network.
The major disadvantage of Marionette is, that it requires an intact network communication. Since crashed nodes, or nodes with a flaw in the communication software, are no longer able to participate in the network communication, it is not possible anymore to inspect their internal state and deduce the point of failure.

Chapter 2

Background

2.1 TinyOS and the nesC programming language

TinyOS [3] is an operating system for embedded wireless low-power devices. Its event-driven nature maps well to the needs of wireless sensor network nodes. A lot of hardware abstraction modules as well as wireless communication stacks have been implemented for TinyOS. This dramatically reduces the programming effort a developer has to take for building an application for wireless sensor networks. TinyOS uses the nesC programming language, an adapted version of the C programming language [7].

2.1.1 Compiling TinyOS applications

In the first step of the compilation procedure of a nesC application, the various modules, interfaces and configurations are fetched by a preprocessor in order to translate them to a single generic C source file. Among other optimization steps, the preprocessor resolves every memory request by a static allocation and all symbols are mapped to a single global namespace. The component to which a symbol belongs, is encoded in the name of the function or variable. This dynamically produced C file is now fed to a C compiler, which builds the firmware for the specific microcontroller.

2.2 The GNU Compiler Collection (GCC)

The GCC compiler is one of the most frequently used tools for compiling C programs in the academic world. With the MSPGCC and the AVR-GCC projects, there are adaptions of the original C compiler, which produce firmware for microcontrollers of Texas Instrument's MSP series or Atmel's AVR series, respectively.

2.2.1 Debugging TinyOS applications

The fact, that the whole TinyOS application is first translated into an ordinary C file has a major impact in the way it might be debugged. The GNU Compiler Collection (GCC) includes a potent and handy debugger for C programs, the GNU Debugger (GDB). This debugger can now be used to debug the generic C file. Line directives, inserted in the generic C file by the nesC preprocessor, map the C code to the corresponding nesC source file and line number. GDB is able to resolve this mapping transparent to the user. This enables single stepping through the nesC code as well as setting and removing breakpoints that belong to a specific nesC source code line number.

2.2.2 Debugging remote targets

There are situations, where the debugger process and the observed application do not run on the same system. This applies also to wireless sensor networks, where the software to be debugged runs on embedded devices. For these scenarios, GDB offers the possibility to connect to a remote target. A GDB-proxy is required on the target system, which receives commands from the main GDB process, investigates the debugged program and sends a corresponding answer back to GDB. The communication between GDB and the GDB-proxy is based on the Remote Serial Protocol (RSP) [2]. This protocol adheres rigidly to the master-slave concept, where the GDB acts as a master and the GDB-proxy is only allowed to create exactly one answer to every request of the GDB.

2.2.3 GDB Machine Interface (GDB/MI)

GDB was initially designed as a command line tool. The GDB/MI is a machine oriented text interface to GDB. It is intended to be used in systems, where GDB is not directly manipulated by humans, but by another system. As an example, GDB/MI allows it to easily build a custom graphical user interface (GUI) which uses GDB as its debugging engine.

2.3 The Yeti TinyOS Plugin for Eclipse

Yeti is an Eclipse-based integrated development environment for TinyOS applications. Among other features, it provides a nesC editor with syntax highlighting and realtime code validation, a launch configuration wizard for the comfortable editing of make targets as well as a TinyOS Graph view that shows the wiring of an application's components.

2.3.1 Yeti Debug Plugin

With its debug plugin, YETI is also able to access a JTAG adapter in order to debug TinyOS applications running on a physical platform. Most of the work is thereby delegated to the C Development Tooling (CDT) Eclipse plugin. Further informations about the debug plugin is given in [9].

2.4 The COOJA Simulator

In the design of a simulator there is always a trade-off between the fidelity of the simulation and its resource consumption. For example, a model for the radio traffic in a wireless sensor network may use a simple probabilistic approach, leading to a fast but inadequate solution. On the other hand, the model may calculate the actual distribution of the electromagnetic waves in the environment as well as the noise pattern. The sophisticated result of this calculation comes at the price of a massive load for the host processor, causing the simulation to slow down. In a similar way, the platform models may be designed on different levels of abstraction.
COOJA is a Java-based simulator for wireless sensor networks, originally designed for simulating applications for the Contiki operating system. All its key functionality is encapsulated in plugins, including radio communication models, mote platform models and control panels. For every task, the user can decide which plugin should be used for it and can therefore assign a level of detail for this task.
COOJA even allows to use different models for the mote platforms in the same simulation, each of them operating on a different level of abstraction. This cross-level approach provides a developer with the possibility to run a few nodes on the instruction level, providing a detailed insight in the program execution. Meanwhile the large number of remaining nodes use a simple Java-based model on the application level which therefore hardly compromise the simulation performance.

2.4.1 MSPsim

MSPsim is an instruction-level emulator for the MSP430 microcontroller [6]. It is specially designed for using it as a COOJA plugin. The input it takes are unmodified firmware files compiled for the MSP430 microcontroller. MSPsim also supports the simulation of hardware peripherals such as sensors, radio modules or LEDs. Its architecture allows to easily adapt the simulator to new sensor boards and the built in support for breakpoints and watchpoints as well as for single stepping simplify the debugging procedure.
With the MSPsim at hand, TinyOS applications can easily be emulated in the COOJA simulator.

2.4.2 Avrora

The Avrora instruction-level simulator is the equivalent to MSPsim for Atmel's AVR microcontroller family [11]. Like MSPsim it may be used to emulate a specific hardware platform in a COOJA simulation.

Chapter 3

Design

The idea of this project is to establish a connection between the Yeti development environment and the Cooja simulator for the sake of debugging simulated TinyOS applications directly in the development environment.
To achieve this connection, a plugin is added to each of the two platforms.

3.1 Communication Levels

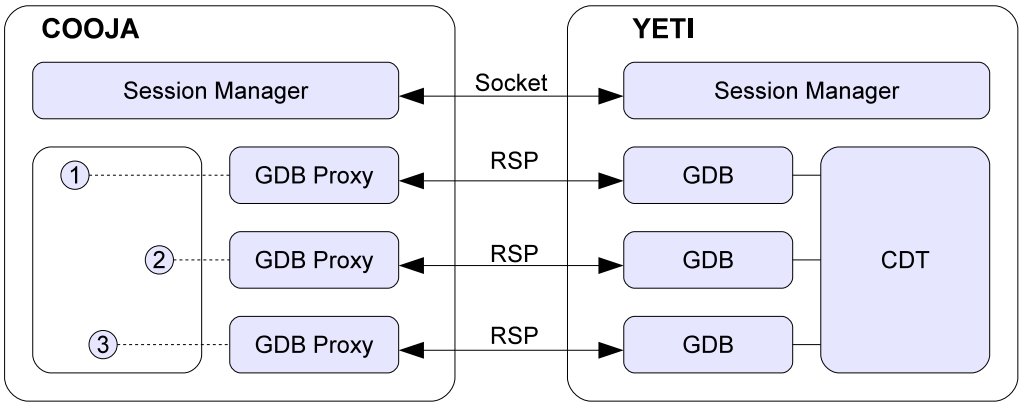

Figure 3.1: COOJA and Yeti communicate on two levels: a high-level control link between the session managers and a low-level GDB RSP connection between the GDB instances and the corresponding GDB proxies.

Figure 3.1 shows the connections that are established between Yeti and Cooja during a debug session.
The two session managers communicate over a TCP socket, where the session manager on the COOJA's side is acting as a server and the session manager on the Yeti's side is acting as a client. This high-level control link is mainly used for two tasks: exchanging metadata and commands. Metadata is for example the number of motes in the simulation and their corresponding ID. A command

might for example be the request to resume the simulation or the notification that a mote has received a breakpoint.
For every mote in the simulation, a GDB instance is created in Yeti and a GDB proxy instance is created in COOJA. The corresponding GDBs and GDB proxies communicate over TCP, using GDBs remote serial protocol (RSP). On these low level links, the actual debugging work is done. This includes setting and removing breakpoints or inspecting the memory and registers for a specific mote.
One might ask why this rather complex communication model is necessary. The model would be much simpler if the multi-process and multi-executable support of GDB was used to handle all motes with a single GDB and a single GDB proxy instance. The problem of this approach is that the full support of these features was added to GDB with version 7.2 [5]. Unfortunately the MSPGCC suite still uses the GDB 7.0.1 [4] and therefore the multi-process and multi-executable support is not available for this project.

3.2 Connection Setup

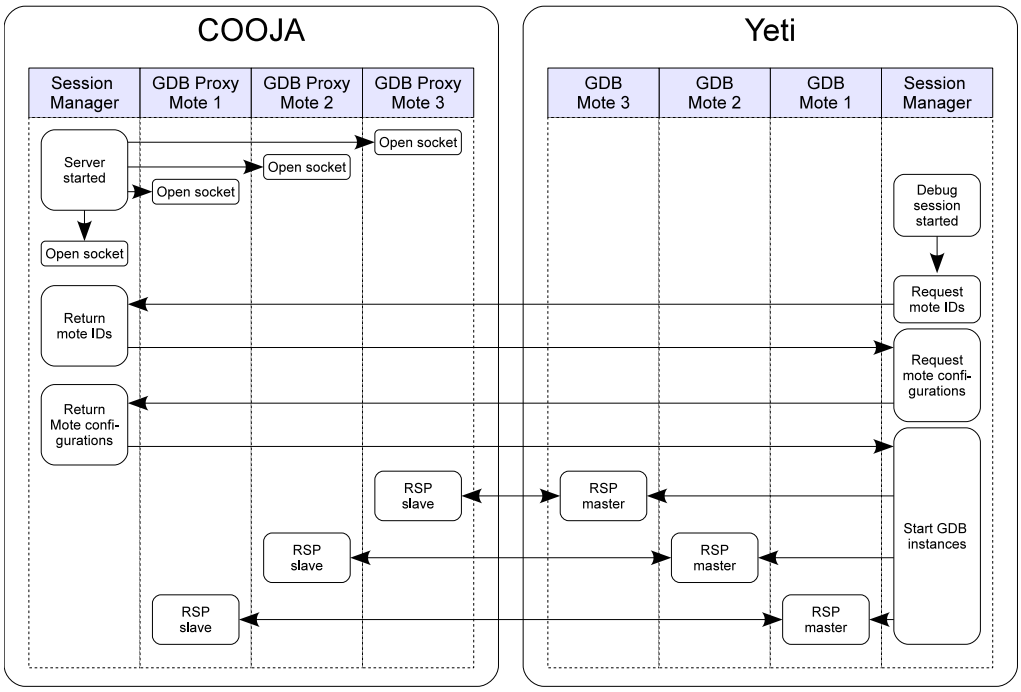

Figure 3.2: Activity diagram for the process of connecting COOJA and Yeti: session managers exchange metadata used to connect the GDBs and GDB proxies with each other.

In fig. 3.2, the activity diagram for the process of establishing the connections between COOJA and Yeti is shown. During the initialization of the simulation,

a GDB proxy process is started for each mote in the simulation. All these proxies open an individual socket and wait for an incoming RSP request on this socket. The session manager itself also opens a socket and waits for the Yeti's session manager to set up a connection on this socket.
Initialized by a user request, the Yeti session manager now starts a debug session. After a short handshake phase (not shown in the picture) the Yeti session manager requests the IDs of all debuggable motes in the simulation. Upon receiving this list of mote IDs, the Yeti session manager knows the number of debuggable motes in the simulation and their corresponding ID.
In order to get more information about the motes in the simulation, the Yeti session manager now sends a mote configuration request to the COOJA session manager for every mote. With the answer to this request, the Yeti session manager learns all the details about this specific mote, including the port on which its GDB proxy listens. With these information at hand, the Yeti session manager is now able to start a GDB process for every mote and let this process establish a connection to the corresponding GDB proxy.

3.3 Integration into Eclipse

In order to add the desired functionality to Yeti, an Eclipse plugin, called Yeti debug simulation plugin, was added to the project. The integration of this plugin into Eclipse is shown in fig. 3.3. The plugin has three important parts:

- **Simulation manager:** controls the whole debug session and maintains the high-level link to COOJA
- **View:** user interface that feeds the user with informations about the simulation and accepts commands from the user
- **Mote:** representation of a simulated mote

Whenever a mote should be connected, the simulation plugin delegates a launch to the CDT. The CDT then starts a GDB instance and creates a new debug target in the Eclipse debug framework. The simulation plugin registers itself as an event listener at the debug framework and therefore gets informed whenever a debug event happens. Additionally, the simulation plugin produces debug commands and delegates them to the debug framework.

3.4 Integration into COOJA

A plugin was also added to the COOJA platform. Figure 3.4 shows the integration of this plugin into COOJA. The plugin mainly consists of a simulation

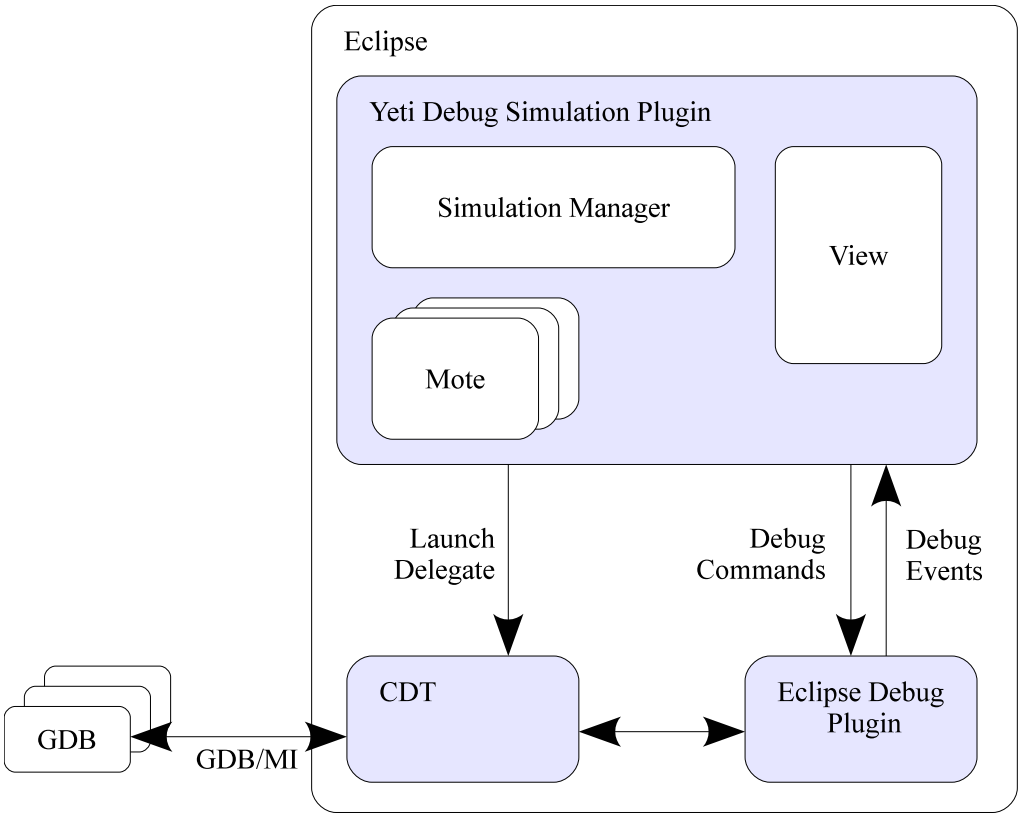

Figure 3.3: Integration of the Yeti simulation debug plugin into Eclipse: The actual debugging work is delegated to the CDT and the Eclipse debug framework.

manager and one GDB proxy for every simulated mote.
The simulation manager is connected to the simulation in order to receive events, obtain data and send commands. It also generates and controls the GDB proxy instances and receives events from them. The GDB proxies are directly connected to the MspMote objects, the simulation's internal representation of the simulated motes. Additionally, the GDB proxies are able to send commands to the simulation.

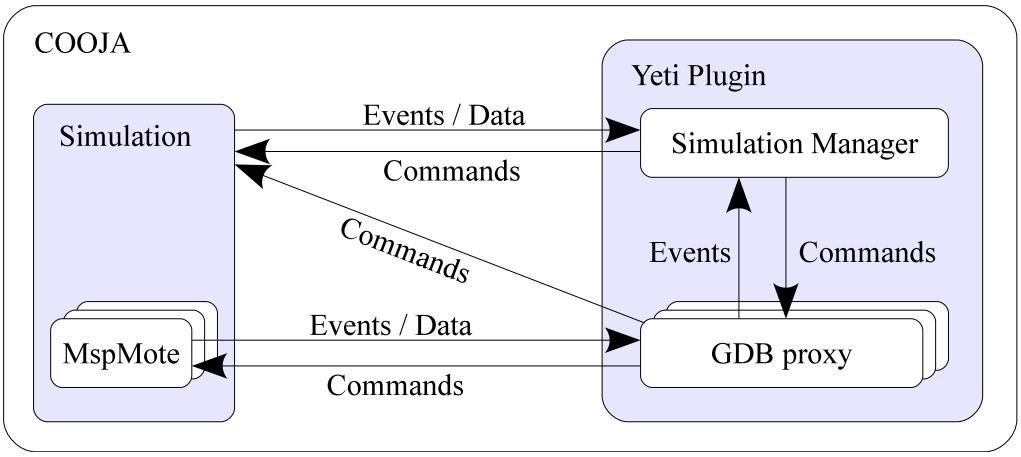

Figure 3.4: Integration of the Yeti plugin into COOJA: the simulation manager is connected to the simulation, the GDB proxies directly communicate with the MspMote objects.

Chapter 4

Implementation

4.1 The Yeti Debug Simulation Plugin for Eclipse

As mentioned in Chapter 3, an Eclipse plugin was created to add the debug facilities for simulated TinyOS applications to Yeti. This section describes the implementation of this plugin.

4.1.1 Plugin Organisation

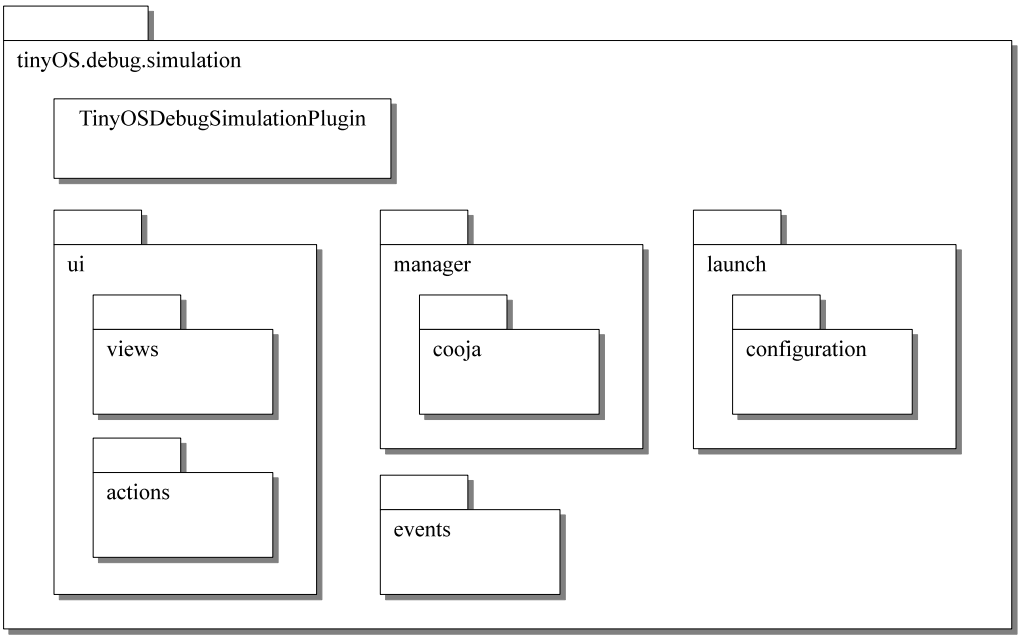

Figure 4.1: Package diagram of the Yeti debug simulation plugin.

Figure 4.1 shows the package diagram of the plugin. The most important package is the `tinyOS.debug.simulation.manager` package. In this pack-

age, the session managers and the corresponding mote representations are implemented. The `tinyOS.debug.simulation.launch` package provides the facilities for configuring and launching a new debug session. In the package `tinyOS.debug.simulation.ui`, a view is implemented to show the status of the simulation to the user and a set of user actions is defined. The package `tinyOS.debug.simulation.events` contains objects and interfaces used by the element of the plugin to notifying each other about the occurrence of events.

4.1.2 Session Manager

In order to avoid a restriction of the Yeti plugin to the COOJA simulator, the `tinyos.yeti.debugger.simulation.simulatorTab` extension point was defined. All plugins, that want to extend the simulation support to a new simulator, have to extend this extension point. With the extension, a configuration tab must be provided that that is able to configure the launch of a simulation in this new simulator and a session manager has to be implemented. The current version of the Yeti simulation debug plugin implements the extension for the COOJA simulator.

4.1.3 Starting the Debug Session

On the Eclipse platform, starting an application is done by executing a so called launch. Before this can be done, a user has to generate a launch configuration of the correct type.
The type of the launch configuration specifies the principal nature of the application and therefore the way, the launch has to be performed. For example a Java application is started in a totally different way than a PHP script. Therefore Java applications and PHP scripts each have their own launch configuration type.
The launch configuration itself is a collection of parameters needed to execute the launch. Such parameters may for example be paths to resources or the version of the Java virtual machine that should be used to start the application.
Every launch configuration type specifies a delegate class, which is responsible to execute launches of this type. Whenever a user performs a launch, the delegate's `launch()` method is called with a copy of the launch configuration as one of the arguments. Then it is the task of the delegate to start up the desired application based on the parameters extracted form the launch configuration.

The Yeti debug simulation plugin extends the `launchConfigurationTypes` extension point in order to provide a new launch configuration type for the debugging of TinyOS simulations. Figure 4.2 shows how a user can edit the parameters of the launch configuration. In the 'Simulator' tab, the user is asked which simulator should be used. In the 'Simulator Configuration' group, the user has to specify the attributes specific to the chosen simulator. For the COOJA

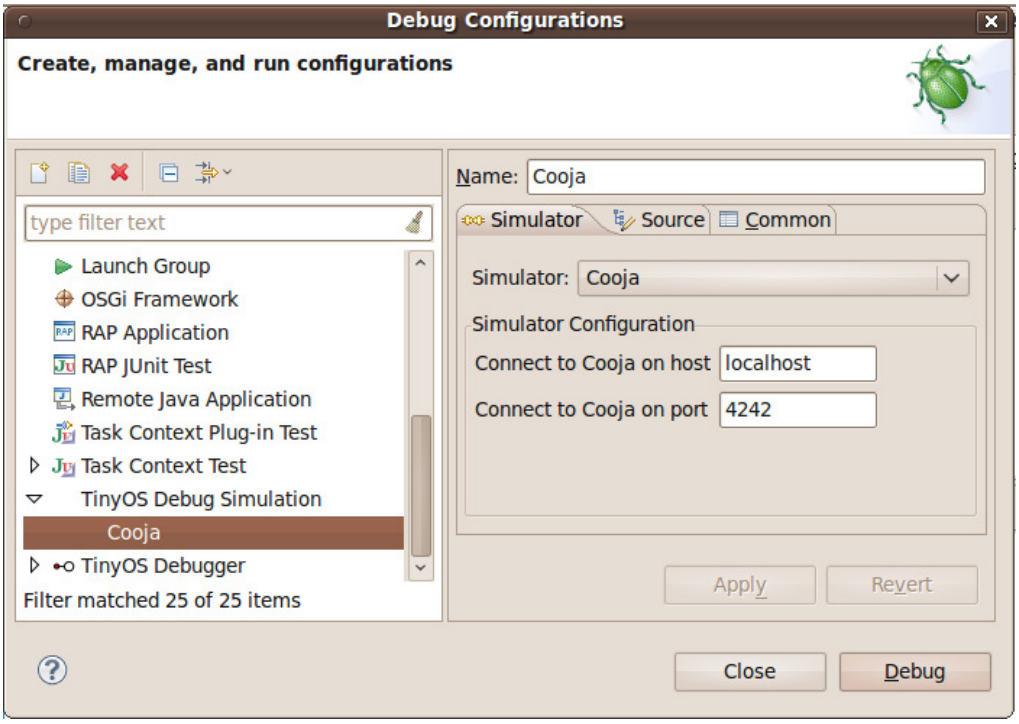

Figure 4.2: User interface for editing a launch configuration of the type 'TinyOS Debug Simulation'.

simulator, its the host and port pair, on which the COOJA session manager waits for Yeti.
When the user runs a launch of this type, the delegate instantiates a session manager and passes the launch configuration to it. The session manager then connects to COOJA using the informations extracted from the launch configuration. For every mote in the simulation, the session manager now creates an internal representation of the mote.
When the user wants to connect a specific mote, this mote takes the original launch configuration as a starting point and adapts the launch configuration's fields in a way that they now match to the configuration of the mote. This includes on one hand changing the port attribute to the port on which the corresponding GDB proxy waits for a connection. On the other hand, a parameter is added for the path to the firmware file running on the mote and the path to the GDB binary and the GDB init file that should be used to debug this specific mote. The Yeti session manager gets all these informations from the mote configuration sent by the COOJA session manager.
Finally the mote delegates a launch with the manipulated launch configuration to the CDT using the CDT abstraction layer created in [9]. The CDT is now responsible for the whole work of creating the GDB instances and registering the

debug target in the Eclipse debug framework.

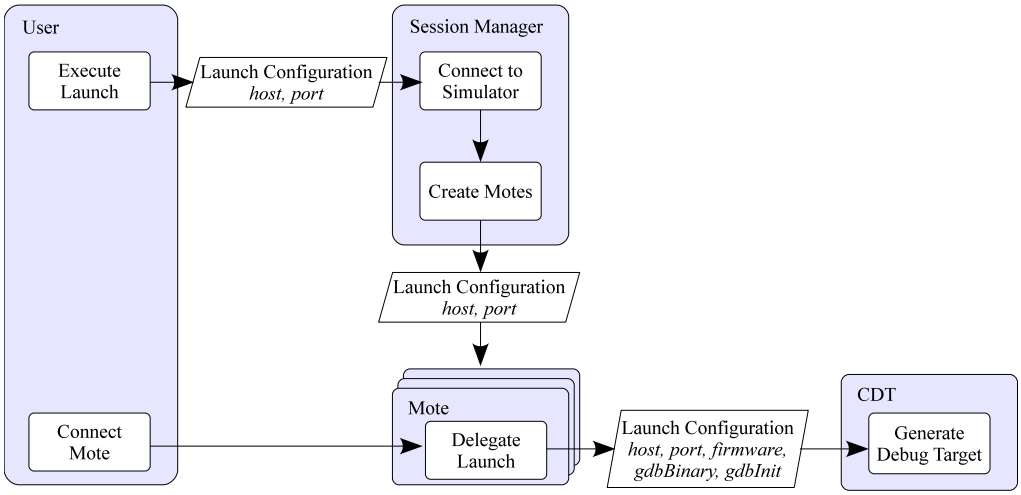

Figure 4.3: The motes adapt the original launch configuration and delegate them to CDT.

4.1.4 The Simulation Debug View

Since the CDT launch delegate registers a debug target for every debugged mote in the Eclipse debug framework, these targets appear in the Eclipse standard debug view. Although it is possible to control the debug session with this view, we do not recommend it. Instead, the Yeti debug simulation plugin provides its own debug view. This view shows all motes running in the simulation no matter whether they are connected or not. In the context menu of every mote, the user can activate commands for connecting and disconnecting motes.
In the Eclipse debug framework, every debug target has a set of threads. Since TinyOS applications are always single-threaded, all the motes will have exactly one thread in the debug framework. Since showing this thread to the user might be confusing, the plugins view omits this information and directly shows the stack trace of each mote.

4.2 The Yeti Session Plugin for COOJA

For the COOJA platform, a plugin of the type `SIM_PLUGIN` was created called `YetiSession`. In order to include it in the build, the configuration file

`CONTIKI_ROOT/cooja/apps/mspsim/cooja.config`

has to be adapted: in the line `sics.cooja.GUI.PLUGINS` the class name

`ch.ethz.dcg.cooja.mspsim.plugins.YetiSession`

has to be appended to the end of the line.

4.2.1 Plugin Organisation

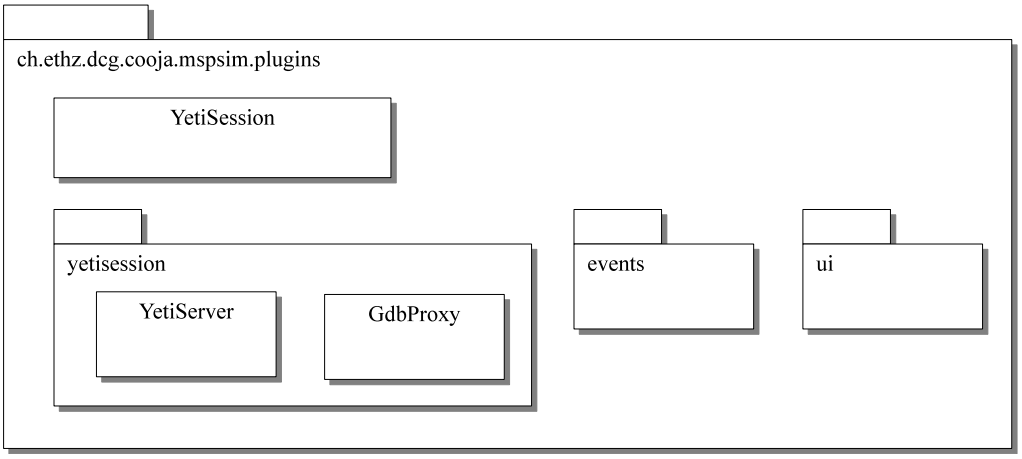

Figure 4.4: Package diagram of the YetiSession plugin for COOJA.

Figure 4.4 shows the package diagram for the `ch.ethz.dcg.cooja.mspsim.plugins` package, which contains the YetiSession plugin. The key functionality is collected in the `yetisession` package, namely the `YetiServer` class which implements the session manager and the `GdbProxy` class which implements the GDB proxy for the simulated motes. The `ui` package contains a dialog for configuring the plugin and the `events` package contains facilities used by the plugin's components to notify each other about events.

4.2.2 GDB Proxy

The class `GdbProxy` implements the proxy side of the GDB remote serial protocol (RSP). It is instantiated by the class `YetiServer` for every simulated mote and runs in its own thread. When the thread is started, it opens a socket and waits for RSP commands on this socket. A good introduction in the RSP is given in [2].

Chapter 5

Conclusion and Future Work

With this project, we were able to establish a connection between the Yeti development environment and the COOJA simulator by including a plugin in each of the two platforms. With these plugins it is possible to launch a debug session managed by Yeti with the targets running in a COOJA simulation. This leads to a simpler handling of the simulation and therefore a shorter development time. The software built for this project should be viewed as a proof of concept rather than a ready-to-use development tool. Further work should be invested to increase the number of the supported debugging functions. This may include finding a way to include some kind of assertions in the TinyOS code, or to build an automated testing procedure similar to JUnit.

Bibliography

[1] The contiki operating system. Website. Available online at `http://www.sics.se/contiki/`; visited on April 18th 2011.

[2] Mac os x developer library: Gdb remote serial protocol. Website. Available online at `http://developer.apple.com/library/mac/#documentation/DeveloperTools/gdb/gdb/gdb_33.html`; visited on April 18th 2011.

[3] The tinyos project. Website. Available online at `http://www.tinyos.net/`; visited on April 18th 2011.

[4] Gcc 4.x toolchain for texas instruments msp430 mcus. Website, 2010. Available online at `http://mspgcc4.sourceforge.net/`; visited on April 18th 2011.

[5] The gdb wiki. Website, 2010. Available online at `http://www.sourceware.org/gdb/wiki/MultiProcess`; visited on April 18th 2011.

[6] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. Cooja/mspsim: interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 27:1–27:7, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[7] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[8] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[9] Silvan Nellen. Eclipse plugin for tinyos debugging. Semester Thesis, 2009. ETH Zürich.

[10] Benjamin Sigg. Yeti 2 - tinyos 2.x eclipse plugin. Master's thesis, ETH, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Distribution Computing Group, 2008.

[11] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[12] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th international conference on Information processing in sensor networks*, IPSN '06, pages 416–423, New York, NY, USA, 2006. ACM.