Demo Abstract: CoojaTrace, Extensive Profiling for WSNs

Moritz Strübe, Florian Lukas
Friedrich-Alexander University Erlangen-Nuremberg
{struebe,lukas}@cs.fau.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

***Abstract*—CoojaTrace extends the Cooja WSN-Simulator by offering extensive logging capabilities for debugging and analyses of multi-node WSN deployments. This is implemented using Scala-based Functional Reactive Programming (FRP)-techniques, enabling flexible and easily programmable in-depth access to the internal node, as well as simulator execution state.**

CoojaTrace is part of the DryRun framework; a set of tools that instrument a WSN-Simulator for extensive analysis of WSN deployments. To achieve this, not only external accessible state like energy usage or serial output is needed, but also internal state like routing tables or operation state in general, which often requires the instrumentation of pointers, and can hardly be monitored by observing static memory addresses. Although some WSN-Simulators provide debug interfaces like a GDB-stub, an easy interface to log and analyze system state of a whole deployment is still lacking. This gap is filled by CoojaTrace by providing a simple and scriptable interface to access this data.

I. Introduction

CoojaTrace[1] is part of the RealSim/DryRun framework. These two frameworks work together to allow *Deployment-Targeted* development. RealSim automatically configures the simulator to match a real deployment as close as possible, by collecting information from a previously deployed network [4]. DryRun is a collection of tools that use the preconfigured simulator to improve the quality of the deployed network. Examples are finding optimal configuration settings or testing and comparing multiple software revisions.

While most WSN-simulators provide some kind of scripting interface that allows to interact with the serial interfaces of the motes, easy access to internal data and state is often neglected. For example program variables or the state of different hardware components can often only be accessed by directly extending the simulator. As a consequence these extensions are usually highly specialized and therefore not very flexible in terms of extensibility. With the threshold for extending the simulator being quite high, the required data is often exposed via the serial interface, which is not very satisfying solution, as it alters the behavior of the node.

We therefore present CoojaTrace, a plugin for Cooja [2], which allows to access an extensive amount of system state using a scripting language. It provides wrappers to most interfaces to enable monitoring them using Functional Reactive Programming (FRP). The results can be logged to different output formats, as well as displayed at runtime.

[1]http://rdsp.cs.fau.de

II. System Architecture

Most components within Cooja interact using the observer-pattern, which allows other objects to be notified upon changes to the component's state. This pattern can easily be mapped to the Functional Reactive Programming (FRP) programming paradigm, which introduces a type of variable that can propagate changes automatically. FRP variables can be derived from other FRP variables through arbitrary functional expressions. Once a variable changes, this change is automatically propagated to all derived variables, comparable to the way most spreadsheet programs update cells. Wrapping Cooja's observers into FRP variables allows further processing and logging steps to be described using a concise functional syntax, while the actual data propagation is handled by the FRP framework.

The implementation is based on Scala and the reactive-core framework[2]. As Scala is compiled into Java byte code and its objects are compatible with Java objects, the Java based Cooja simulator can not distinguish the Scala code from any Java code. This allows an easy integration into the simulator. Further on Scala provides a runtime compiler and thus allows to write code without the need to restart the simulator.

Part of CoojaTrace is a comprehensive library, which maps the observer-pattern to FRP, and in addition provides abstractions to simplify the access to Cooja's data. Most notably is the easy access to the memory of the application running on a mote. While Cooja already provides symbol resolution (i.e. mapping a variable name to an address), as well as viewing and changing memory, there currently is no easy way of logging a variable. In addition to logging, CoojaTrace allows to dereference pointers and do pointer arithmetics. This is a very powerful feature as operating systems like Contiki make extensive use of pointers and structs. Even if the target of a pointer is not changed at runtime, obtaining the address of a certain data word currently is manual work, and it is unlikely that the address is unchanged after the next translation, thus requiring manual interaction after each compilation.

Especially when simulating long running experiments, or monitoring values that change often, like the stack pointer, data aggregation can significantly improve performance, as less data must be saved, as well as analyzed. For this CoojaTrace provides different operators that can, amongst others, calculate

[2]http://www.reactive-web.co.cc

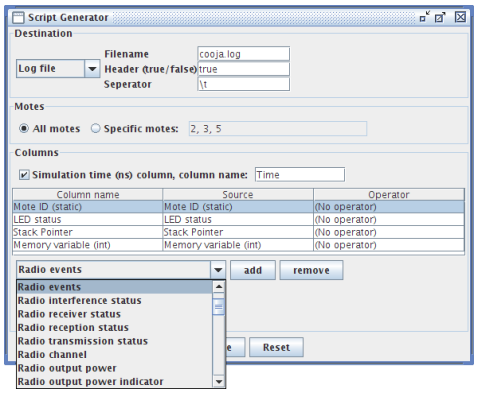

Fig. 1. CoojaTrace script generator

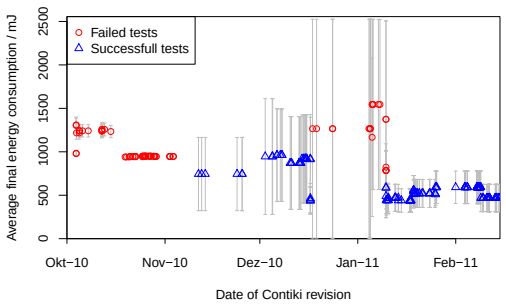

Fig. 2. Average energy usage and standard deviation of 20 nodes in an experiment instrumenting the collect protocol

the maximum, minimum, average or standard deviation based on a sliding window.

While Cooja's integrated script editor is probably better suited to write test cases, this is of course possible using CoojaTrace, too. For one, as already mentioned, CoojaTrace can access all of Cooja's objects, and can therefore interact with it. Further on CoojaTrace provides an `assert()` statement, which can be used to either stop and analyze the simulation (e.g. using GDB) or terminate the simulation when running unattended. The latter is especially interesting to improve runtime when running a batch of simulations (e.g. to find an optimal configuration).

For logging CoojaTrace currently supports a log window, a simple text file format, as well as a SQLite[3] database. Besides that, an output window, similar to the one provided by Timeline [3] is in development. In addition to the markers provided by Timeline, the visual log-target will also support plotting graphs, which will also allow to plot information like the stack pointer in the same window.

To lower the threshold of creating new queries CoojaTrace provides a wizard (Fig. 1), which supports the user in creating simple queries without the need to understand the syntax.

III. Examples

To show the power of CoojaTrace we present two examples. For both examples, which can be concatenated, we will use a very simple log format, using three columns `mote`, `what` and `val`, where the first two are used to distinguish what is logged. The time stamp is automatically added as an additional column.

```
1 val logt = LogFile("ct.log", List("mote", "what", "val"))
```

A very simple example is the logging of the currently running process of every node.

```
1 for (mote <- sim.allMotes) {
2     log(logt, mote, "process", mote.currentProcess.name)
3 }
```

The `for` statement loops through each object in the `sim.allMotes` collection. For each `mote` the mote's FRP-variable `currentProcess.name` is assigned to the log-target. Thus each time the running process changes, this is propagated to the log-destination, and a new row is added. `mote` and `"process"` are needed to distinguish the log-entries by mote or variable.

A more complex example is the logging of the energy usage of each node, based on the model of Energest [1].

```
1 for(mote <- sim.allMotes) {
2   val rx    = timeSum(mote.radio.receiverOn)
3   val tx    = timeSum(mote.radio.transmitting)
4   val act   = timeSum(mote.cpuMode === "active")
5   val idle  = timeSum(mote.cpuMode =!= "active")
6   val energy: Signal[Double] =
7       rx * 60 + tx * 53.1 + act * 5.4 + idle * 0.1635
8   log(logt, mote, "energy", energy)
9 }
```

The `timeSum`-Function sums up the time a certain condition is true. By multiplying this value with the energy required in this state the energy usage of the mote can be estimated. Due to the FRP based approach the energy is logged each time the state of the node or radio changes.

Using the listing above we measured the energy required to receive 10 messages from every node using the collect protocol. After about twice the expected time the experiment failed. The samples chosen in figure 2 show that there was a severe problem with the network stack in October 2010. While we did not investigate the cause, it also shows that the changes made around New Year improved the situation.

[3] http://www.sqlite.org/

Acknowledgements

This work was supported by the Bavarian Ministry of State for Economics, Traffic and Technology under the (EU EFRE funds) grant no. 0704/883 25.

References

[1] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based sensor node energy estimation. In *Proc. of the 5th Int. Conf. on Embedded Networked Sensor Systems (SenSys 2007)*, pages 409–410. ACM, 2007.

[2] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level simulation in cooja. In *European Conf. on Wireless Sensor Networks (EWSN 2007), Poster/Demo session*. IEEE, 2007.

[3] F. Österlind, J. Eriksson, and A. Dunkels. Cooja timeline: A power visualizer for sensor network simulation. In *Proc. of the 8th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2010)*, pages 385–386. ACM, 2010.

[4] M. Strübe, S. Böhm, R. Kapitza, and F. Dressler. RealSim: Real-time Mapping of Real World Sensor Deployments into Simulation Scenarios. In *Proc. of the 6th ACM Int. Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WiNTECH '11)*, pages 95–96. ACM, 2011.