Regression Testing Framework for WSNs
(Demo Abstract)

Wolf-Bastian Pöttner, Daniel Willmann, Felix Büsching, and Lars Wolf
Technische Universität Braunschweig,
Institute of Operating Systems and Computer Networks,
Mühlenpfordtstraße 23, 38106 Braunschweig, Germany
[poettner|dwill|buesch|wolf]@ibr.cs.tu-bs.de

***Abstract*—Quality assurance of software for WSN nodes is both: important and cumbersome. Testing software in simulators is possible, but the reality is only reproduced up to a certain extent. Therefore, tests on real nodes cannot be replaced by simulators. Especially when multiple nodes are involved, running tests with reproducible results on real nodes becomes a challenge. Software has to be compiled for each node, all nodes have to be flashed and reset to establish the same initial state for all tests. In this demo we show a system that continuously monitors source code management systems (such as GIT or SVN) to trigger tests for new revisions. Multiple predefined tests are run with a defined initial state and the system collects results in terms of performance numbers and success rates. Furthermore, call graphs are automatically generated to allow for locating performance bottlenecks in the software.**

I. Introduction

Program code for Wireless Sensor Network (WSN) nodes is becoming more and more complex. Since debugging complex code on embedded platforms is cumbersome, proper quality assurance mechanisms have to be installed to ensure a high code quality. To make matters worse, many WSN deployments do not have means to update code in the field, essentially making them a fire and forget solution. If the software fails in the fields, the nodes have to reprogrammed which is often connected to significant effort and potentially long timespans of unavailability. We argue, that continuous testing of WSN software during development is indispensable to ensure a high code quality.

Contiki OS [1] uses a simulator-based regression testing approach [2]. Using simulated nodes has the advantage that tests can be easily run in parallel to speed up time for testing and that networks with many nodes can be simulated. However, simulators reproduce the real world with limited accuracy and therefore are only one tool for testing, debugging and performance evaluation. Especially generating accurate information about execution time of programs is problematic on simulated nodes. In [3] we have shown, that call graphs for code running on actual nodes can be generated to allow in-depth understanding of where a program spends most of its time.

In this paper we present our approach to regression testing of code for WSN nodes that does the following:

- Automatically trigger tests based on Source Control Management (SCM) changes
- Perform tests with defined initial state for reproducible results
- Arbitrary number of tests per change
- Keep track of results and performance figures
- Generate call graphs for test runs

II. Concept

To fulfil the goal stated in the introduction, we make use of the open source Continuous Integration (CI) server Jenkins[1]. In Jenkins we define "jobs" that are executed when a change is committed to the SCM. Jenkins allows fine grained control to make sure that only tests that are relevant for a specific SCM change are triggered. Jenkins allows to run multiple tests in parallel if enough "slaves" (PCs with nodes attached) are available. Finally, Jenkins can run certain longer-running tests with a greater interval, e.g. once a week if a change in the SCM occurred. In our system, Jenkins jobs trigger a python script that conducts a certain test as described in the next section.

A. PC-based Test Framework

Our test framework running on the PC is written in python and conducts a single test. The tool ensures defined initial states for all nodes by making sure that all nodes have the latest software or a dummy image. While the test tool can be started by Jenkins, it can also be used stand-alone for repeated tests while writing software. The outcome of a test consists of logfiles as well as performance numbers and a binary flag indicating if the test was successful or not. Tests are specified in the form of code to conduct the actual test as well as two configuration files. One file is test-specific and defines which code is flashed onto which node. This configuration further allows to parameterize the tested code via compiler flags and can enable a timeout after which the test is aborted if no result has been returned. Finally, the test-specific configuration determines which source code files are compiled with instrumentation enabled. In addition, a computer-specific configuration file specifies which nodes (type and role) are connected to which ports of the computer, where the source code resides in the file system and where the logs shall be stored.

The python script (illustrated in Figure 1) then compiles the source code for each involved node, whereas first of all the program is compiled without instrumentation. If files are selected for instrumentation, the modification time of those files is changed and the program is recompiled with

[1] http://jenkins-ci.org/

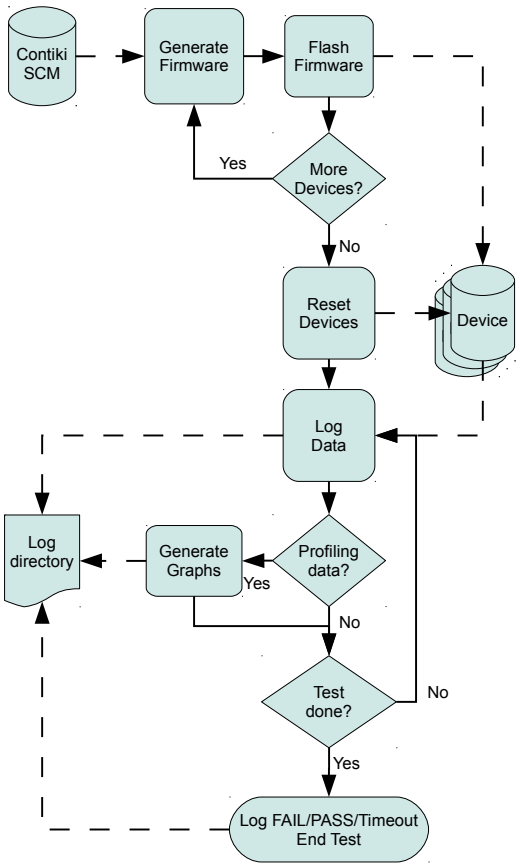

Fig. 1: Flowchart of a single test (simplified)

instrumentation enabled. The GNU *make* utility will then automatically recompile the files with newer modification time and link those together with the files that have been previously compiled. Afterwards, a simple dummy-program is compiled for all nodes that are specified in the configuration but are not used in the current test. This program ensures, that no old code fragments on unused nodes disturb the radio communication of the nodes that are involved in the test.

After compiling the code, the binary images are flashed onto all nodes, one after the other. After flashing all nodes, the nodes are reset and logs of the nodes (transported via serial port) are stored. The python tool skims the stream of outgoing data for certain markers that are used for signaling. Those markers include:

- *Reboot*: Node has rebooted, test is automatically aborted and counts as FAIL
- *Call Graph Information*: Information is collected and passed on to a script that generates a call graph in pdf format
- *Performance Numbers*: Information is collected and passed on to Jenkins for trend graph generation
- *Pass/Fail*: Reports if a node judges the test to be successful or unsuccessful

The test is completed successful if all involved nodes report a "pass" within the time limit. If at least one node either reports a fail, reboots or reports nothing within the time limit, the test is failed. After the test is completed, the python tool collects log files of all the nodes, binary images flashed onto the nodes as well as the call graph files and stores them in the log file directory. Optionally, the tool generates an XML file containing the performance number for Jenkins.

In [3] we have shown that instrumenting code has a variable impact on performance. Furthermore, we have seen that certain tests would only complete when they are either instrumented or not instrumented because instrumentation changes the timing behavior. We therefore run each test with and without instrumentation to measure the raw performance and to be able to find certain timing-related bugs and misbehavior.

B. Node-based Test Software

On the nodes we use Contiki OS; however, the solution presented here is not limited to Contiki. Nodes print out a distinct string during the boot process to allow detecting reboots. Furthermore, nodes can print out regular logs over the serial port while the test is running. If none of the markers is used, those information is ignored by the python script and simply written to a logfile. Our profiling infrastructure [3] requires that the programmer explicitly triggers the output of the collected profiling information over the serial port. Furthermore, the programmer of the test case has to explicitly decide whether a test is a success or not and call the respective functions. The programming interface is illustrated in Figure 2.

```
void TEST_FAIL(char * reason);
void TEST_PASS();
void TEST_REPORT(char * description,
    uint32_t value, uint32_t scale, char *
     unit);
```

Fig. 2: Interface of the test framework on the nodes

III. Experiences

We use the regression testing framework presented in this paper to ensure the correct function of μDTN [4]. μDTN is a bundle protocol [5] implementation for Contiki OS (essentially a network stack) consisting of more than 10.000 lines of code. To test μDTN we currently use 15 regression tests that are automatically executed on every SCM change. We use the test framework on the one hand to make sure all tests complete successful as expected and on the other hand to continuously monitor the performance impact of changes. To analyze the change of performance over time, all performance figures reported by the tests are automatically plotted by Jenkins as shown in Figure 4. If a drop in performance becomes apparent, we look at the log files of the test as well as the call graph to determine the source of this drop. We furthermore use the python framework for repeated tests while developing and optimizing software. Since the tool will flash all involved nodes, mistakes such as old software version on certain nodes or glitches of the build system when changing the compiler flags are prevented.

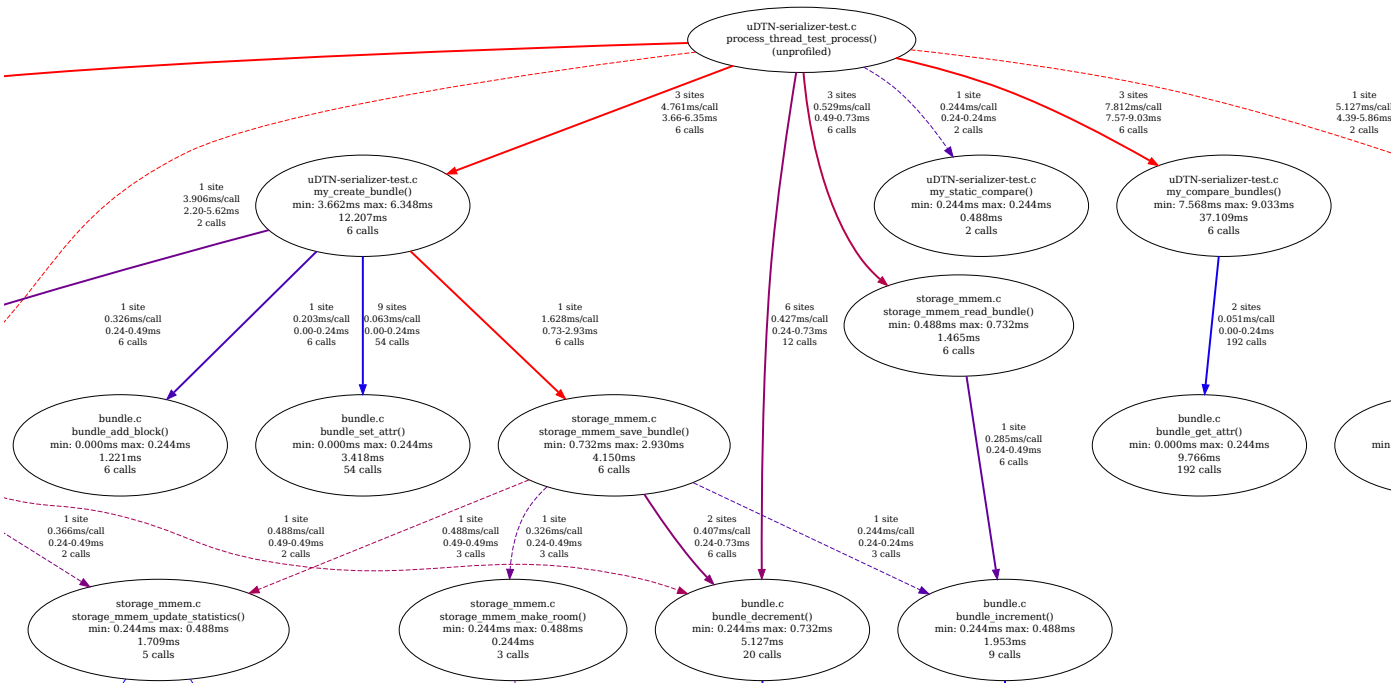

Fig. 3: Excerpt from the call graph of a μDTN test for the bundle serializer

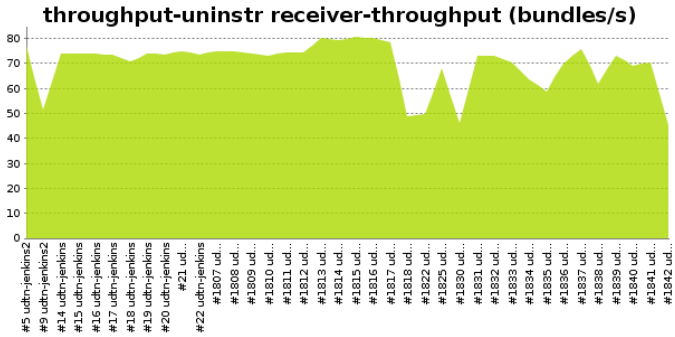

Fig. 4: Throughput in bundles per second over multiple builds in Jenkins

μDTN is based on Contiki and should therefore be able to run on (almost) any platform that is supported by Contiki. However, regularly interoperability problems occur due to byte ordering or memory alignment issues. In our test environment we use TMote Sky [6] as well as INGA [7] nodes to allow automated interoperability testing of μDTN running on both nodes. In Figure 4 we exemplarily show the throughput in bundles per second over multiple builds (and associated revisions) in Jenkins.

IV. In this Demo

In this demo we show a live sensor node running instrumented and non-instrumented code using the test framework discussed in this paper. We show how a call graph is generated from code running on the node on the example of a μDTN test case and how the call graph can be used to optimize the code.

In the test case, the bundle serializer of μDTN is tested locally (i.e. without interaction with other nodes) and an excerpt of the call graph is shown in Figure 3.

For the demo we will need:

- Table (for laptop)
- Wall outlet (for laptop)
- Space for a poster

References

[1] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[2] A. Dunkels, "Contiki Regression Tests: 9 Hardware Platforms, 4 Processor Achitectures, 1021 Network Nodes," http://contiki-os.blogspot.de/2012/12/contiki-regression-tests-9-hardware.html, Dec 2012.

[3] W.-B. Pöttner, D. Willmann, F. Büsching, and L. C. Wolf, "All eyes on code: Using call graphs for WSN software optimization," in *Eight IEEE Workshop on Practical Issues in Building Sensor Network Applications 2013 (IEEE SenseApp 2013)*, Sydney, Australia, Oct. 2013.

[4] G. von Zengen, F. Büsching, W.-B. Pöttner, and L. Wolf, "An Overview of μDTN: Unifying DTNs and WSNs," in *Proceedings of the 11th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze" (FGSN)*, Darmstadt, Germany, 9 2012.

[5] K. Scott and S. Burleigh, "Bundle Protocol Specification," RFC 5050 (Experimental), Internet Engineering Task Force, Nov. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5050.txt

[6] Moteiv Corporation, "Tmote Sky Datasheet," http://www.snm.ethz.ch/pub/uploads/Projects/tmote_sky_datasheet.pdf, 2006.

[7] F. Büsching, U. Kulau, and L. Wolf, "Architecture and evaluation of inga - an inexpensive node for general applications," in *Sensors, 2012 IEEE*. Taipei, Taiwan: IEEE, oct. 2012, pp. 842–845.