Increasing the Reliability of Wireless Sensor Networks with a Distributed Testing Framework

Matthias Woehrle
ETH Zurich
8092 Zurich, Switzerland
woehrle@tik.ee.ethz.ch

Christian Plessl
ETH Zurich
8092 Zurich, Switzerland
plessl@tik.ee.ethz.ch

Jan Beutel
ETH Zurich
8092 Zurich, Switzerland
beutel@tik.ee.ethz.ch

Lothar Thiele
ETH Zurich
8092 Zurich, Switzerland
thiele@tik.ee.ethz.ch

ABSTRACT

Designing Wireless Sensor Networks (WSNs) has proven to be a slow, tedious and error-prone process due to the inherent intricacies of designing a distributed, wireless, and embedded system. A systematic design approach accompanied by a test methodology supports the development of WSN software conforming to all design requirements including robustness and reliability. In this paper, we propose the fundamentals of such a test methodology. We present essential features of a framework for testing a broad range of WSN applications. We demonstrate with a case study that our test methodology is a feasible approach by integrating a number of existing design-tools for the TinyOS operating system. While we target TinyOS in the case study the proposed test methodology is general and not tailored to a specific WSN platform or operating system.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

Keywords

wireless sensor networks, testing, reliability

1. INTRODUCTION

Wireless sensor networks combine the challenges from different research domains, A WSN is a distributed system that shows a large degree of parallelism. Each WSN node is essentially an autonomous embedded system with stringent constraints on energy, storage and memory resources. The wireless nature introduces an unreliable communication channel requiring robust and resilient communication protocol design. When deployed in an unprotected environment, external influences such as noisy radio channels or wide temperature variations can considerably influence the application and require adequate countermeasures to be taken at design time. Many WSNs aim at continuous and autonomous operation with only limited or no access and supervision. This unique combination of system characteristics requires novel design methodologies, since deploying a correct WSN application is of utmost importance.

Motivated by many reported pitfalls [2], our own experiences and discussions with industrial partners, we argue that a systematic design approach accompanied by an end-to-end test methodology is key to overcome these intricacies and to build sustainable WSN systems. A test methodology shall enable a design team to continuously monitor the correctness of an implementation.

Figure 1 illustrates possible paths in the process of developing a typical, large scale WSN application. The top-right corner represents the goal of the development process, an application running in real-world on 1000 WSN nodes. Typically, such a large scale application is developed by simulating a small number of nodes first. After the simulation passes a formal or informal test procedure, the developer faces two choices: either the application is refined and im-

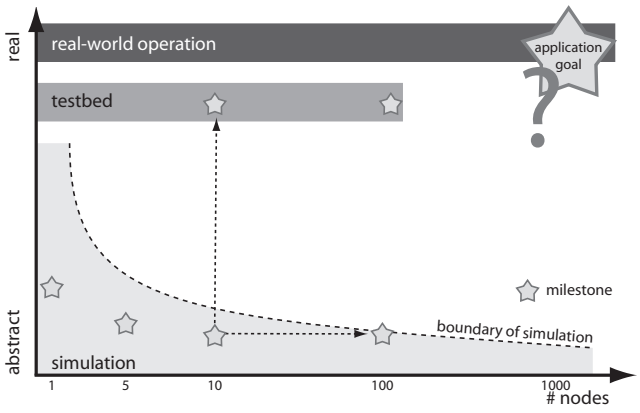

Figure 1: WSN development process: moving from abstract simulation of few nodes to real-world operation of many nodes

EmNets '07, June 25-26, 2007, Cork, Ireland

Published in: *Proc. Workshop on Embedded Networked Sensors (EmNets)*
© ACM 2007, DOI: 10.1145/1278972.1278996

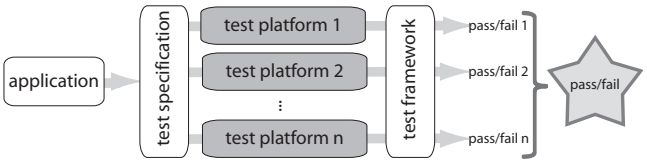

Figure 2: Our test framework allows for testing WSN software on different test platforms (e. g., in simulation, or distributed execution on a testbed) given a common test specification.

plemented on a testbed, or the simulation is extended to include more WSN nodes. Detailed simulation of WSNs requires significant computing resources. Hence, large-scale WSNs can be simulated only with a reduced accuracy and fidelity of the simulation results. In general, porting an application that has been validated in simulation to the testbed is not trivial, due to inaccurate simulation assumptions, limited debugging capabilities in the testbed, and possibly stricter resource constraints on the WSN target nodes.

In this paper, we propose a *new methodology* for testing WSNs and we present an implementation of the methodology in a *test framework*. To the best of our knowledge, this is the first work that applies self-checking tests to the domain of distributed WSNs. Based on a test specification, the methodology bundles the WSN software, the test inputs and the expected outputs into a test case, which allows for automated verification.

Figure 2 illustrates the capability of our test framework to seamlessly transfer test cases between different test platforms. This supports the development process by continuously verifying the correctness of the software under test on different levels of abstraction. The test framework automatically runs the tests and verifies that the test results conform to the specification.

This paper is organized as follows. We discuss related work in Section 2. Section 3 introduces our test methodology and defines the components of the test framework. Section 4 presents an implementation of our test methodology for a TinyOS 2 platform. Finally, we summarize our work and draw conclusions.

2. RELATED WORK

There have been numerous research efforts to ameliorate the WSN development process towards robust deployments. Design methodologies—such as WSN specific languages (Attributed State Machines [7], nesC [4]) or design and architectural principles (MNet [2])—are othogonal to software testing efforts. In-networking debugging techniques (Marionette [12], Sympathy [9], VigilNet [6]) or packet sniffing [10] provide passive monitoring to increase the visibility of the system. While these techniques help to analyze problems in a WSN deployment, they do not try to actively identify errors in the software by injecting appropriate or corner case scenarios.

The focus of the WSN community concerning software testing has been the implementation of individual test platforms such as simulators (TOSSIM [8], Viptos [1]) or testbeds (MoteLab [11], EmStar [5], DSN [3]). Our test methodology integrates test platforms into a comprehensive test framework (see Sec. 4) leveraging the individual benefits.

3. WSN TEST METHODOLOGY AND FRAMEWORK

In this section, we propose a WSN test methodology and an according framework and introduce the necessary terminology.

3.1 Testing Framework Architecture

We propose a test methodology that bases on the idea to integrate self-testing capabilities into the tests. Self-testing is achieved by explicitly specifying the inputs and the expected outputs of the software under test in an executable format. This allows for full automation of the test procedure. Automated testing promotes frequent execution of the tests and enables the designer to continuously compare the implementation's behavior with the specification.

Thorough testing is important for building robust applications, however it is no guarantee for a correct application. Testing is fundamentally limited to detecting the presence of errors in an implementation and is not capable of proving the absence of errors.

A key property of our proposed methodology is that it allows the designer to provide a unified specification of a test case. The same test case can be executed on different test platforms: real hardware (e. g., using a testbed) or a simulator. The simulation can target different levels of abstraction, e. g., pure functional simulation, functional simulation with a refined radio channel model, cycle-accurate execution on an instruction set simulator, etc. The framework does neither implement nor imply the capability to automatically translate a design across abstraction levels. Instead, the framework provides the mechanism for executing test cases that are valid on different abstraction levels on the respective platforms.

The framework provides the generation of test components tailored to a specific test platform relying on a unified specification. Hence, the resulting reusability of the application-specific test specification promotes multi-platform testing. This increases the ease of use of testing, allows for automatic verification of the software under test, and thus helps to detect possible regression errors.

3.2 Test Specification

By *testing a WSN* we understand the execution of a test suite that is a collection of *test cases*. The architecture of a test case is illustrated in Fig. 3 and is discussed in the following.

3.2.1 Software under test

We denote the application software that is subject to the test procedure as *software under test (SUT)*. A high-level model for simulating interaction between multiple nodes, or a model that focusses solely on a specific aspect of a node may be preferred at the beginning of the design process. Details are added over time until the SUT is equivalent to the software that is actually executed on the hardware mote.

3.2.2 Test drivers

A test driver implements the inputs, so called stimuli, that are applied to the SUT during the execution of the test. Each stimulus is characterized by value (or type) and order (e. g., timestamp) when it is applied to the SUT. The sequence of stimuli generated by the test driver can be arbitrarily complex. *Static drivers* apply a single event to

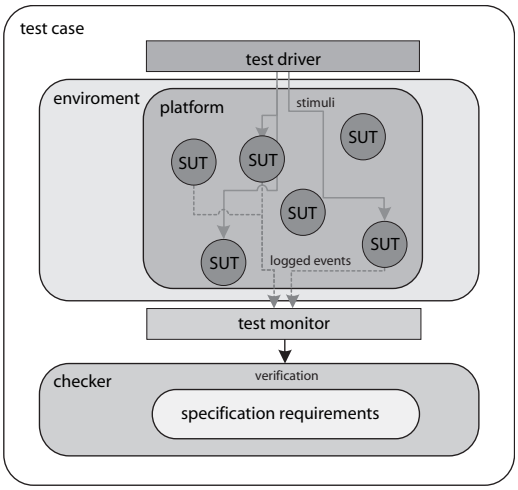

Figure 3: Exemplary test case specification comprising 6 SUT instances

the SUT (e. g., a start signal) or stimulate the application with a predefined sequence of events. More general *dynamic drivers* read the SUT's state or outputs and react to dynamic responses of the system. The framework provides the infrastructure for distributed test platforms to apply stimuli to remote SUT instances.

3.2.3 Test monitors

Test monitors observe functional and non-functional properties at runtime and provide these results to the test checkers. Some properties may only be determined after the test has been executed, e. g., the total packet loss rate. For these test cases the distributed monitors collect the intermediate data during the test, convert or group the data to the suitable representation for the checkers after the execution of the test.

A test monitor can also be implemented as a part of the application running on the WSN node to allow for fine grained observations on real system implementations. These monitors are denoted as *target monitors*. Using such "in-situ" monitoring however requires meticulous care in the implementation and retrieval of the results generated, since every cycle executed and every byte transfered on a WSN node ultimately alters the application's behavior.

3.2.4 Test environment

The test environment is a collection of functions, that incorporate and model all parameters of the environment that are relevant for the tests. The relevant parameters for a specific test are determined by the scenario defined in the test case. The test case defines the position and mobility of nodes and hence the network topology.

Depending on the abstraction level certain aspects of the environment can be included, simplified or excluded.

3.2.5 Test checkers

Test checkers are binary decision functions for validating the system's execution according to functional and non-functional requirements of the specification. The inputs to the checkers are provided by the test monitors. The output of a checker is either "test passed" or "test failed".

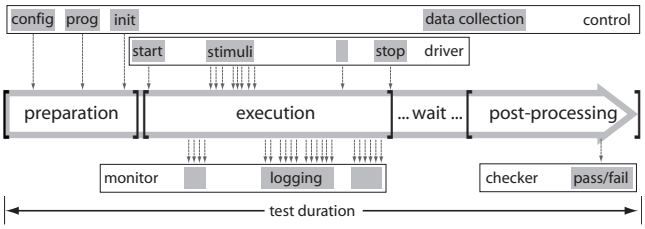

Figure 4: Timeline of the test procedure of an exemplary test case including interactions with the test framework components.

This concept is sufficiently flexible for many scenarios. A test case can verify a single property of the implementation, e. g., whether two WSN nodes can communicate or whether a certain communication pattern results in a deadlock. But test cases can also capture more complex behavior as long as the result of the test is binary, e. g., check whether the data-loss rate is below an acceptable threshold. Test cases can also check for non-functional properties, such as time and energy used, if these properties are accessible.

When implementing a test case, the designer must be aware that the checker is applied to different abstraction levels, e. g., to the simulation and to the testbed execution. Therefore, the designer should take care to implement the checker as independent as possible from specific test platform properties.

3.3 Test Procedure

The test procedure is divided in three phases: test preparation, test execution, and result verification. Figure 4 visualizes the distinct phases of the test execution.

3.3.1 Test preparation

In the test preparation phase, the application code is compiled for the execution platform. Depending on the test case, the appropriate test drivers and test monitors are loaded. On a simulation platform, the SUT program is loaded into the simulator and the environment is configured according to the specification in the test case. For executing the test on a testbed, the application code is distributed and programmed to all WSN nodes. Communication channels between the test framework host and the WSN nodes must be established for the drivers and monitors.

The WSN nodes are initialized to a predefined state that can correspond to a reset state or to a checkpoint representing the SUT at a particular point in operation by setting buffers or queues to a specific state derived by analysis or simulation.

3.3.2 Test execution

In the execution stage the SUT executes reacting to the test driver's stimuli and subsequent internal events in the selected environment. The test monitors observe and log all functional and non-functional data as defined for each instance of the SUT. The test executes for a fixed time or until a predefined state is reached.

3.3.3 Result verification

In the result verification phase, the data collected by the monitors is retrieved and compared to the expected results defined in the checker. On a simulation platform the data

is typically available in logfiles. On a distributed test platform,e. g., a testbed, it may be required to request and retrieve the data from a distributed logging facility.

4. CASE STUDY

In this section we present a case study that validates the feasibility of our approach for two selected test platforms. It demonstrates an example implementation of the methodology for the TinyOS 2 operating system and Moteiv Tmote Sky nodes and integrates existing tools into an automated test framework.

4.1 Software under test

The SUT is a sensor data collection application based on the MultihopOscilloscope application, which is part of the TinyOS 2 distribution. The application is a typical WSN application that sends locally collected sensor data to sink nodes. The data is routed over a multihop collection tree. We have extended the application such that oscilloscope packets are sent and forwarded by the nodes only when an enable flag is set. The TinyOS dissemination protocol is used to distribute changes to this flag at runtime to all nodes.

4.2 Test platforms

We selected two commonly used platforms for WSN testing, a standard WSN simulator and a testbed. For the simulation of our TinyOS application we selected the TOSSIM simulator [8]. TOSSIM is a prominent example of an event-driven simulator allowing to use the actual TinyOS application code. It can simulate multiple nodes and features simulation primitives for modelling the communication channels of the nodes.

For the testbed platform, we have selected the Deployment Support Network (DSN)[3]. The DSN is a distributed wireless testbed infrastructure, that allows for testing different types of WSN nodes. Each Tmote is attached to a DSN node providing local communication via the UART. The DSN nodes build a wireless network that provides communication between the Tmotes and the test host.

4.3 Test case

The case-study aims at testing the quality of service of a data collection application. The test case defines 10 Tmotes set up in a typical office environment. We fit the TOSSIM radio model parameters to reflect the topology and communication characteristics of this environment. The parameter fitting is based on data from previous link quality measurements for the Tmote targets.

At the beginning of the test the test driver powers on all target nodes. Initially, the flag that controls the data collection is disabled. After an idle interval, allowing the nodes to setup the routing tree, the test driver sets the enable flag on the sink node. Sensor data is collected for a predefined duration. Subsequently, the enable flag is reset and the test case ends after a phase-out interval.

For the TOSSIM simulation, the stimuli are directly applied using the built-in python interface. The testbed platform infrastructure services the test driver by communicating to the Tmotes. The DSN provides the capability to apply the stimuli to all or a subset of the target nodes in the distributed system.

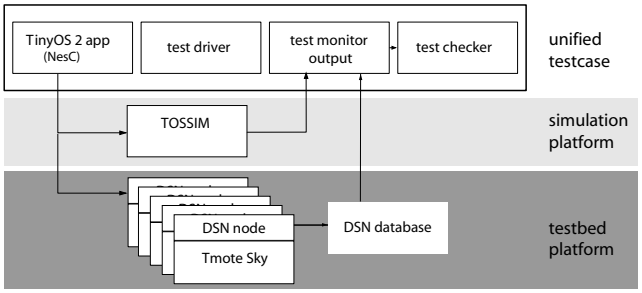

Figure 5: Case study test flow for both platforms

A TinyOS-based target monitor was developed to probe variables and state changes in an application. For TOSSIM, the test monitor writes its output to a logfile. The DSN implementation writes the local monitor data via UART to the DSN node. A database collects the distributed monitor information from the DSN nodes and thus provides a common data access point for post-processing.

We attach the target monitor to the following functions in our SUT: 1. sending local sensor data packets, 2. packet forwarding to host on sink node, 3. boot sequence including code version and node id, and 4. enable flag changes.

The checker uses the monitor information from item 1) and 2) above to derive the average packet yield, which is defined as the ratio of the number of packets the sink node forwards to the host and the sum of local sensor data packets sent by all nodes.

The pass or fail condition defined in this test case is an average packet yield of 90%.

4.4 Test flow

Figure 5 depicts the test flow for the example application with a unified test case, and the according tool chain for the two test platforms. The starting point of the test flow is an instrumented TinyOS 2 application. In a (so far manual) pre-processing step, the generic test primitives are replaced by platform-specific test framework interfaces.

The driver for the simulation communicates with TOSSIM via the built-in python interface generating monitor output in a common format. For the testbed, an according python script executes commands via an XML-RPC interface to the DSN nodes. Monitor data is retrieved from a database and subsequently formatted. The common format allows for using the same checker for both test platforms. The checker outputs a pass or fail notification based on the previously specified requirement.

4.5 Test results

We ran each test case 40 times on both test platform. Running a test case on the simulation platform takes about 0.5 min whereas a run on the testbed takes approximately 11 min. The test time differences are due to differences in the execution time on the platform and significant overheads in pre- and post-processing steps. These overheads motivate the aggregation of test cases to test suites to avoid extensive use of expensive operations like code distribution and target programming for the testbed platform.

Figure 6 presents the test results. Additional to the binary checker output, indicated by the horizontal line, the average packet yield derived by the checker is displayed. The data

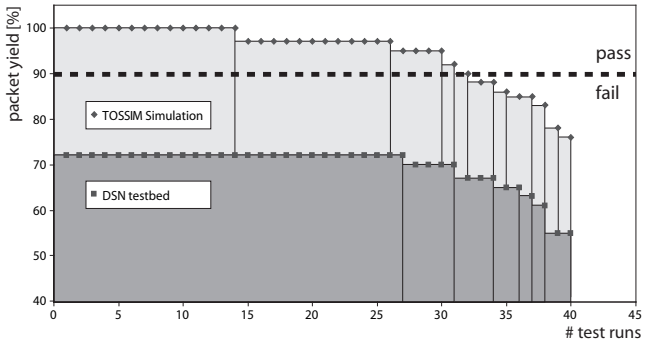

Figure 6: 40 independent test runs ordered according to the resulting packet yield. 32 out of 40 simulation results passed the test criteria of an average packet yield $\geq$ 90%.

in the graph is sorted with respect to the packet yield to point out its distribution.

While 20% of the TOSSIM test cases failed, no test result on the testbed platform passed the defined criteria. The simulation assesses the packet yield overly optimistic. However, the measured packet yield is consistent with literature (Great Duck Island yield 58%, California redwoods yield 40%, Ecuador volcano yield 68% [2]).

The distributions of yields for both platforms follow a similar trend but differ significantly in absolute values. The current coarse grained test case cannot provide a detailed explanation for this mismatch and additional fine grained test cases are needed to determine the cause.

5. CONCLUSIONS

The main contribution of this paper is the proposal of a test methodology and framework for WSNs. The methodology applies the idea of self-checking tests to WSNs, i. e., distributed embedded systems. Our test framework allows for executing the same test cases on different test platforms (e. g., simulation or testbed) and for automatically verifying the correctness of the application.

We have shown that an implementation of the test methodology is feasible by integrating existing tools into a test framework for TinyOS applications. We have successfully applied the test framework for verifying an example application on two abstraction levels: in simulation and running on a testbed. We argue that the proposed design methodology leads to an improved WSN design flow and are confident that it allows for building more robust WSN applications. Applying our framework to real-world applications will reveal if the pay-off in terms of software reliability and robustness justifies the additional effort of the proposed methodology. Thus, we are planning to integrate additional platforms and to make the framework available to other researchers.

6. ACKNOWLEDGEMENTS

The work presented here was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322. We kindly acknowledge the support by Mustafa Yuecel and Roman Lim.

7. REFERENCES

[1] E. Cheong, E. Lee, and Y. Zhao. Joint modeling and design of wireless networks and sensor node software. Technical Report UCB/EECS-2006-150, Nov. 2006.

[2] J. Choi, J. Lee, M. Wachs, and P. Levis. Opening the sensornet black box. Technical Report SING-06-03, Stanford Information Networks Group, Stanford University, CA, 2006.

[3] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum. Deployment support network - a toolkit for the development of WSNs. In *Proc. 4th European Workshop on Sensor Networks (EWSN 2007)*, volume 4373 of *Lecture Notes in Computer Science*, pages 195–211. Springer, Berlin, Jan. 2007.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation (PLDI 2003)*, pages 1–11. ACM Press, New York, June 2003.

[5] L. Girod, J. Elson, A. Cerpa, T. Stathapopoulos, N. Ramananthan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proc. USENIX 2004 Annual Tech. Conf.*, pages 283–296, June 2004.

[6] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. Stankovic, T. Abdelzaher, J. Hui, and B. Krogh. VigilNet: An integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks*, 2(1):1–38, 2006.

[7] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05)*, pages 45–52. IEEE, Piscataway, NJ, Apr. 2005.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137. ACM Press, New York, Nov. 2003.

[9] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. 3rd ACM Conf. Embedded Networked Sensor Systems (SenSys 2005)*, pages 255–267. ACM Press, New York, 2005.

[10] M. Ringwald and K. Römer. Passive inspection of sensor networks. In *Proc. 3rd IEEE Int'l Conf. on Distributed Computing in Sensor Systems (DCOSS 2007)*, Santa Fe, New Mexico, USA, June 2007.

[11] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05)*, pages 483–488. IEEE, Piscataway, NJ, Apr. 2005.

[12] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proc. 5th Int'l Conf. Information Processing Sensor Networks (IPSN '06)*, pages 416–423. ACM Press, New York, 2006.