Approaching Wireless Sensor Networks Using Systematic Testing Strategies

TIK Report 284
Matthias Woehrle, Jan Beutel, Mustafa Yücel and Lothar Thiele
Computer Engineering and Networks Lab, ETH Zurich
8092 Zurich, Switzerland
matthias.woehrle@tik.ee.ethz.ch

Abstract—Wireless Sensor Network (WSN) applications today are hard to develop, often resulting in fragile systems with meager performance. We contribute to the coordinated development of WSN applications through the presentation of test and validation methods that allow for gradual refinement throughout the whole development flow. The methods described allow to systematically develop an application or part thereof while continuously and systematically integrating and testing the system under development. This approach is based on continuous integration with a special focus on networked embedded systems and their specific requirements concerning the testing architecture. Additionally a set of advanced metrics for the assessment of power traces are presented. The applicability of the tools and strategy presented is demonstrated using the showcase of a challenging environmental monitoring application.

I. INTRODUCTION

Wireless Sensor Networks have been anticipated to become a pervasive tool enabling detailed and unobtrusive observation of real-world phenomena, thus bringing substantial benefits to a variety of application areas. A number of advanced proof-of-concept systems have been successfully deployed, suggesting that the remaining gap between these prototypes and real-world applications of sensor networks has been significantly narrowed. However the reports of difficulties, failures and in general unacceptable performance both in the development phase as well as in operation are abundant. Even worse, in a number of popular cases analyzed, the errors are poorly understood and their causes remain largely unknown [1]. We argue, that there is still significant research necessary in order to close the gap between development and deployment using appropriate tools and methodologies. In particular, we believe that in the future the total cost of applying a sensor network will be dominated by the development and deployment process as per-device-cost are expected to drop to a few dollars. Hence, significant efforts are needed to approach the goal of consistent and coordinated design, development, validation, and deployment of applications for WSN, which are still in their infancy today.

The approach presented in this paper is based on methodologies derived from enterprise scale software engineering that is adapted and extended to the peculiarities of wireless networked embedded systems. Special attention has been given to the fact that the tools and methodologies developed are suitable in the domain of WSN where low-power operation, distribution and resource limitations prohibit for example the complete logging of system behavior. But exactly the details of the system behavior and its reaction to changes in the environment are paramount for an understanding here and as a result for successful development of WSN applications. The observation of the live system, under real-life operating conditions embedded in an environment as close as possible to the designated operating environment and under conditions ranging from the average to the worst case is the goal of our testing approach.

Our novel testing approach comprises a mixture of experimentation and analysis that allows for:

- Full automation of the build process
- Repeatability facilitating regression testing
- Minimal invasive observation of the system behavior using a multi-target testbed infrastructure
- Control of the environment, such as temperature, humidity or the attenuation of the RF channel
- Detailed control and analysis of the resources used, such as memory and power
- Definition and application of detailed metrics assessing the performance

The proposed testing strategy verifies functionally correct and satisfiable operation of the system in different resource and environmental conditions, increasing the test coverage for typically complex deployment scenarios. While analytical methods such as memory footprint analysis assure that the
system under test (SUT) adheres to stringent bounds such as a maximum memory footprint which are basic requirements that must be obeyed.

Test logs and measurements derived from a testbed setup are indicators of the average performance under realistic operating conditions. Current testbeds [2], [3] suffer from the fact that they are typically deployed in a specific location, e.g., an office building and do not encompass means to control the environment. However, this is critical, as direct control of the environment, in which the SUT is situated, can emulate realistic conditions or worst case scenarios that a system is being designed for. The addition of the capability to exert stimuli on the SUT, to record detailed feedback of both the functioning of the software and underlying algorithms and of physical parameters, the detailed reaction of a WSN to environmental changes and the resources available for operation can be investigated. By integrating a testbed with tools from software engineering that automate the build process [4], control of the environment and precise resource monitoring instrumentation, the methodology presented spans a multidimensional testing space (see Figure 1). Tests are performed continually based on the integration of each build. The testing architecture allows for a large number of different operating conditions that can be adapted according to the development progress. Thus, our testing strategy is not limited to a single or a few distinct operating configurations like a standard testbed [5]. This allows for evolution of the testing process, as testcases and test infrastructure grow with the evolving specification. In the pursuit of a concise testing and validation strategy for WSN the tools and strategy presented in this paper enables to investigate situations that are known to occur in a deployment but are ignored in a typical testbed in a coordinated and systematic way.

II. RELATED WORK

Agile methods is an umbrella term for different techniques in software engineering for developing software in iterations. Some of the most prominent examples of agile methods are extreme programming and test-driven development. The iterative approach is well suited to be combined with a continuous integration approach of continuous change commits and integration builds. Cohen [6] presents the history of agile methods, describing their essence and presenting empirical results and case studies. Continuous Integration is an established methodology in software engineering. Duvall [4] provides an introduction into CI, outlines its benefits and presents implementation advice for software projects. Junit [4] provides an introduction into CI, outlines its benefits and presents implementation advice for software projects. Junit is a unit testing framework for software development in Java [7] and provides testing strategies employed such as stubs and mock objects. TUnit [8] is a unit testing framework for TinyOS applications. It differs from standard unit testing in that tests are run distributedly on the target devices. While currently limited in platforms and scale, it is a novel and valuable approach for WSN. It focusses on some of the same software engineering processes, which are presented in this work, such as automated regression testing.

Haeberlen [9] discusses the problems of standardized, synthetic test platforms and traces for distributed systems pointing out five different possible pitfalls. Emstar [10] is a software environment for development and deployment of WSN applications. It supports heterogeneous networks by providing libraries, services and tools. EmTos allows the emulation of TinyOS applications. EmSim allows for simulating the heterogeneous network featuring an emulation mode, where execution information from real nodes are fed into a simulation server allowing for better accuracy in a mixed test environment.

Liu et al. [11] present a tool for increasing visibility at the deployment site. This valuable approach requires instrumentation code in deployed sensor nodes. It is orthogonal to our testing, since our testing strategies are concerned with the development process prior to deployment. The Sensor Network Inspection Framework [12] monitors the network traffic by the overhearing of traffic using a packet sniffer. This allows for non-intrusive, but unreliable monitoring of a sensor network. An analysis framework on a centralized host determines possible causes of problems such as routing loops.

Memento [13] an Sympathy [14] are both tools, which increase visibility by providing additional information on each sensor node, to collaboratively determine the network health and help detecting and debugging failures. Working distributively and inferring a low overhead in terms of protocol requirements and energy, these tools are typically used during and after deployment. Similarly, Visibility [15] is a new sensor network protocol, which strives for optimizing their visibility metric, which means reducing the energy cost of debugging a failure or certain behavior.

III. WSN DESIGN CHALLENGES

The key challenges behind wireless networked embedded systems are their distributed nature, the strict energy budget requiring efficient and low-power operation and the stringent resource limitations found on today’s devices. These characteristics imply that there is a chasm between the reactivity and observability of a system and its efficiency in the use of resources. A system spending the majority of the time in sleep state or accessible only by a highly bandwidth limited and unreliable wireless link is simply not easy to observe, which is a requirement for the successful analysis and identification of defects. It is known that the really hard to debug parts in system design are not in early stages, but to get the last mile right [16]. Reports about WSN failures and generally poor performance are abundant [1] and equally available from the early days [17] and from recent deployments [18]. Although still under investigation, this has lead to a recent push away from a focus on simulation [19] and design to tools and methodology to be used specifically in the deployment [3], inspection [12], [11] and testing [2], [5]. Turau [20] even suggested to precede deployments with an actual trial deployment. Although successful in his case in finding a number of errors in the system under development, the approach is prohibitively expensive and cannot give guarantees as certain details most
likely are different on the trial and real deployment. However from a methodological standpoint, we believe it is a good idea to consider a stepwise deployment strategy which is actually a similar approach.

A. Influence of the Embedded System

While optimization is typically performed on higher abstraction levels, the tight resource constraints of microprocessors including RAM and program memory and ultra-low power radios require programming close to the hardware in order to enable functionally correct operation and often tweak the hardware for its most efficient usage. In some cases this engineering at the max or even “over-optimization” can lead to the very unexpected. A driver that is developed and tested on one device might not perform in a same way on another device or in slightly differing operating conditions. A frequent complaint on the TinyOS mailing list has to do with the power consumption observed for certain power modes or applications. For example the Null application is used to check the lowest power mode on a TinyOS platform architecture. It can happen of course that the code is broken or the toolchain is differing slightly from what has been used by the original developer and thus the individually compiled binary differs from what was originally used to report a sleep value, e. g. for a Tmote platform. In a recent discussion on the TinyOS mailing list, it was reported that even using the older codebase certain nodes were yielding a value greatly differing from the target power consumption. The sleep current was magnitudes off the typical value. Since these were specific nodes only a hardware change was blamed and the manufacturer was contacted. A detailed analysis revealed that in fact no hardware change (schematic change) had occurred but a minor device variation was simply “expecting too much” from the hardware it was running on. This is not the only example of device variations impacting system performance as can be seen in an excerpt below from the TinyOS-Devel mailing list:

> I just tested some almost unused MicaZ motes and they show the
> expected low power consumption ( 2uA measured with the Extech).
> The one that shows high consumption was used in a real deployment
> for several months. I also tested 6 other motes from the same
> deployment and they also show a low power consumption.

Here are the tests using the McuPowerOverride. I used 3 good MicaZ (G1-G3) and the bad one (B). The results are:

<table>
<thead>
<tr>
<th>Mode</th>
<th>G1</th>
<th>G2</th>
<th>G3</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>3.53</td>
<td>3.46</td>
<td>3.63</td>
<td>3.88</td>
</tr>
<tr>
<td>ADC_NR</td>
<td>1.15</td>
<td>1.13</td>
<td>1.18</td>
<td>1.36</td>
</tr>
<tr>
<td>EXT_STANDBY</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.29</td>
</tr>
<tr>
<td>POWER_SAVE</td>
<td>2.6u</td>
<td>1.9u</td>
<td>2.2u</td>
<td>0.16</td>
</tr>
<tr>
<td>STANDBY</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.29</td>
</tr>
</tbody>
</table>

All the values are in mA except for POWER_SAVE for G1-G3.

B. Influence of the Environment

The environment is a commonly known cause of error. Han-eyveld [18] reports unanticipated obstructions: The leaves of the plants present on the deployment site changed transmission characteristics when there was rain. He observed a correlation between packet reception ratio, temperature and humidity. In the initial Great Duck Island deployment Szewczyc [17] discusses node failures and attributes a number of them to humidity and broken seals on the packaging. Apart from these relatively straightforward causes there are also influences of the environment that have deeper roots. Consider for example the heavily used MSP430. It is a micro-controller directly designed for ultra-low power system design. Power mode changes take effect immediately. Nevertheless, even such a targeted device needs to be handled with care for long-term outdoor deployments: One example is the negative temperature coefficient of the Digitally Controlled Oscillator (DCO) when the DCO is disabled for long low-power sleep states. In outdoor deployments, e. g. relying on scavenging energy from the sun, a system may be put to sleep for prolonged periods of time during the night. Such a long-term sleep period can result in a delayed wakeup of the DCO as the device is impacted by a variation in temperature. The effect starts to severely show with sleep periods of tens of minutes and a temperature variation of tens of degrees. And those are common parameters for WSN applications! This and similar effects are documented in the fineprint of the manufacturer documentation but often overlooked in practice.

The goals of designing and developing for a correct function and in the case of WSN power optimization are key to longevity and sustainability in sensor network applications. These examples show that there is indeed a need for comprehensive and widely spread tools and methodology for system testing. It is important to not just simply create one new tool here but an integrated methodology taking into account analytical and empirical methods on different levels of abstraction [19]. Verification of the system with systematic testing strategies is an indispensable part to improve the development process in order to arrive at a functioning and performing system implementation.

Testing of WSN system should cover the complete system state space concerning the hardware resources and environment. Current testbeds are fixed at a single point as depicted in Figure 1. However, it is vital to determine effects of a drained battery, cold temperature or unprecedented obstructions to determine a functionally correct system under such operating conditions in the actual deployment. In the following we present a test and instrumentation architecture that takes WSN testbeds to the next level, incorporating the environment and giving exact and detailed insight into the reaction to changing parameters and resource usage.
IV. THE TEST AND INSTRUMENTATION ARCHITECTURE

As depicted in Figure 2, the test and instrumentation architecture is composed of three basic components: a software repository, an infrastructure for continuous integration and a testing infrastructure. The software repository provides a central location for all software under development and associated tests. It allows to keep track of the development progress and allows multiple user interaction.

The core of the testing architecture is the infrastructure for continuous integration. While continuous integration is an established method and tool commonly used in enterprise software engineering [4], we propose to integrate it with a specialized testing infrastructure to account for the peculiarities of WSN development - execution on a real embedded target and in a distributed testbed environment [2], [3]. The continuous integration server regularly checks for changes in the project software repository and upon detection of a change initiates a clean checkout and then builds the project. The extension with a testing infrastructure allows to deploy and execute the software under test directly on the specific hardware platform it is being designed for. While this is actually a straightforward process often performed manually by developers, this is the first time the automation into a comprehensive integration framework is proposed. The tight integration with the test infrastructure allows to monitor the test and provides detailed information about functional and physical details of the execution on real target nodes. All data monitored during the test is collected and logged into a central database referencing the exact software version in the repository that was used to generate the respective instance.

In the following we will provide the details on the building blocks used in our testing architecture with a special focus on the integration of the physical test, i.e. the test on real motes in a real environment. The mechanics and equipment used in our prototypical setup is briefly explained in the following. A case study is given in section V.

A. Continuous Integration

Continuous integration (CI) [4] is a methodology, which promotes frequent integration, i.e. after each code check-in, in order to provide rapid feedback to developers. This facilitates identifying software defects, since differences are miniscule and error sources are easier to pinpoint, typically subject to a recent change. Up to this point, CI focusses on the integration of enterprise scale software projects designed by large teams and is a common and well known methodology, e.g. in agile development [6].

For integration of software in a team, CI allows for communication with the code repository, a build tool-chain to compile the software, software analysis tools and a test platform running unit tests such as JUnit for Java projects. The overall status of the project as well as the details of all associated builds are presented on a webpage, often also using a central indicator such as a lava lamp. Each build shows the software status providing visibility. A prominent example of a CI server is CruiseControl (CC).

1) Regular Builds with Statistics: Regular builds are either triggered upon code changes introduced to the repository, by user requests using a web interface or using periodic timers. All these builds can be configured separately allowing for usage of individual parameters and tools to be used. Builds are referenced to a specific version of the codebase in the repository using a unique build id. All data generated, logs, build artifacts and test results are stored in a central data structure available through a web based reporting interface.

2) Source Code Analysis and Code Checkers: Different tools for checking, e.g. coding (style) violations such as Findbugs, PMB, CheckStyle or the correctness of javadoc documentation, are readily available for integration into common CI frameworks such as CC. Recently, first specialized tools to check WSN specific software appear such as a checker for
TinyOS [21]. All such tools run during or after the build process and generate reports that can be visualized using the CC reporting front end. However they are typically tools allowing static code analysis only, without actually executing the binary generated.

3) Unit Test Integration: Unit testing is focussed on testing small components, in particular the smallest components available such as individual functions or methods. A piece of code, the test harness wraps around the component under test and allows to execute it isolated inside the test structure defined. This allows to check for the correctness of function of code written and persistence over a longer period of development with often complicated integration. The method typically brakes down the problem of checking for correct function into a number of smaller pieces. It is extremely popular in Java programming (JUnit [7]) and recently also available for TinyOS (TUunit by Rincon Corp. [8]). It is widely applicable especially for agile methods such as test-driven development. Unit testing is however not sufficient to test the comprehensive system, as the system is more than just the union of its components. Complex interactions induced by the inherent parallelism and non-determinism in distributed systems require additional testing on the complete system functionality. Unit testing is a powerful tool. It is complementary to our work, since unit tests can easily be integrated into the test architecture proposed here.

4) Memory Profiling: Embedded system design revolves around the efficient usage of resources and the correct execution of functions. Memory usage analysis is therefore a very important indicator of the status of a project. Simple but quite powerful test extension is a histogram of the memory usage (see Figure 11) and to test on a known upper bound of a specific platform, e.g. 48 kbyte for the case of an MSP430F1611 microcontroller.

5) Notification on Build Failures: CI typically contains a notification mechanism. In the case of CC, these can be configured, e.g. to send email notification to all developers involved upon a failed build. These email notifications contain complete coverage of build status and the error logs in question. Repetitive notifications can be used to emphasize urgency and as a result are a simple yet powerful mechanism to successful progress in teams. The results are anonymous and offer an objective way of telling everyone what went wrong based on a reference installation of both the toolchain and codebase. This eliminates defects stemming from individual developers computer setups, e.g. developers forgetting to include libraries that they use locally. Apart from the technical perspective, the social implication of this mechanism are obvious: No more personal confrontations of fellow developers neither across institutional borders nor in close vicinity of a shared office that are always detrimental to productivity. Anonymity in this case is very healthy since it provides objective, qualitative and quantitative measures.

6) Graphical Reporting: The features presented in the previous sections all rely on a tool running on some input and output data, generating reports as they are executed. While the original logs and text files generated are important when digging deep into details, a powerful graphical reporting interface helps to present a overview of the most critical aspects from the wealth of information and contexts. This provides the benefit to allow for depicting long periods of project development including dependencies in a concise representation.

B. Continuous Integration in the Sensor Network Context

The concept of regression builds, unit testing and test automation are not new and already very successful in the enterprise software world. In this context, Java or similar programming languages prevail and software is typically built for and run on large servers. Since this is a software-focussed approach that does not take into account the hardware intricacies in deeply embedded systems, such as the details of the sensor node hardware platform or the influence of the environment. This is a completely different scenario compared to the Wireless Sensor Network case. The distributed and embedded nature of today’s WSN systems does not allow for mapping the aforementioned procedures and tools directly from enterprise scale CI. Missing are the possibility to execute build results (executables) on a real WSN target device, e.g. a mote, to run distributed tests using a testbed such as MoteLab.

To this end, current CI practices are extended with the goal to provide a comprehensive development support especially targeted for wireless networked embedded systems. Thus, software builds are deployed on a real WSN testbed. Moreover, WSN testbeds extensions should be added to establish a testing infrastructure, which provides capabilities to capture physical parameters as well as instrumentation for stimulation and control of the environment. As depicted in Figure 1, testing for WSNs needs to provide control of environment and resource configuration. In order to guarantee test coverage and to be able to analyze test executions, the infrastructure has to allow for monitoring and controlling the resources available to the sensor nodes.
Testbeds allow for testing applications distributedly on a considerable scale. Further extensions to control the environment allow for a test setup, which closely matches the actual deployment conditions targeted. Combined with a laboratory style instrumentation to monitor and control physical parameters and resources available on the devices under test in the testbed, this is a powerful combination allowing to analyze the data extracted and compare it in real-time against expected results.

C. Testbed Integration

Our approach for testing WSN software combines established methods such as regression testing and unit testing with execution in a real environment. By using a testbed and realistic deployments scenarios, many factors that cannot be captured in simulation and analysis methods, can be used to narrow in on a realistic sensor network deployment and simultaneously extracting behavioral trace data from the system under test. Such a testbed can be located locally or even be integrated remotely. Even federated testbeds, each with specific operating environment characteristics or dedicated to a certain project or design team are feasible. Test jobs consist of a compilation process that is handled by the CI framework with a little additional instrumentation: (i) job formation with subsequent submission to a testbed, (ii) distribution of code to target devices, (iii) synchronous start of all target devices and (iv) log file collection. Depending on the context of the test, analysis can take place online, e.g. for the monitoring of operation or in more detail offline after completion of a test job.

1) Data Collection - Basic Logging: As already discussed in section IV-A, repetitive integration and tests based on changes of the software codebase allows for gradual refinement. This allows to continuously check the status and progress of the integrated base of a project. Using continuous testing strategies such as unit testing, a design point that has been reached once and tested to satisfaction will not be lost again. In combination with the execution on a testbed additional tasks can be performed such as the variation and tuning of parameters (sensitivity analysis) or the variation of the network topology by using different subsets of nodes or even different platforms in the testbed.

Without very elaborate customization or the incorporation of complicated tool setups, the integration of CI and a testbed allows to test repetitively using a number of different analysis methods. Through the integrated approach however, execution is greatly simplified and data from all test jobs is logged in a repository that references the actual software code version under test. This assures a maximum of transparency and the ability to post-facto analysis without using detail or context. A fact that is often overlooked when testing is carried out manually, notes are taken in a lab notebook and log files are organized in individual, non-persistent structures making it almost impossible to compare results from subsequent test runs.

2) Emulating the Environment: An apparent deficiency of current WSN testbeds is that they are typically set up in office environments and thus can offer only some realism with respect to the devices and an actual radio channel used. But they fail to capture the exact effects of a target deployment region, e.g. temperature variation, or changing operating conditions, e.g. depleting power resources. However these effects are important factors that have lead to a number of reported failures for real WSN deployments [18], [1]. The effect of temperature on electronic devices is well known, often basically considered at design time but not elaborated in detail in a system context towards the end of a development phase. Since software running on WSN devices is often engineered to the max (see example in section III), a simple change in the operating environment can lead to a malfunction of the system. In a similar way the capacity of any battery is influenced both by the discharge behavior and the environmental conditions (temperature). Today, this can neither be emulated or estimated and requires testing on a live object.

We therefore suggest to further augment the CI/testbed with means to emulate and control the environment in which nodes are deployed in a testbed. In practice this means that nodes under test are to be placed in a temperature and humidity cycling (TCT) chamber and powered from different power sources, e.g. real batteries or preferably a programmable power source (battery or solar cell emulator). This allows to test response of a system under test in both to cyclic (temperature cycles) and boundary (depletion of the battery) conditions. In combination with the ability of certain testbeds to insert asynchronous events [3], e.g. errors like a node failure or a delayed start of a subset of nodes, worst case scenarios can be emulated on the system under test. Of course mixed scenarios are also easily feasible, with a portion of nodes in a TCT chamber and a portion located outside subject only to the regular ambient conditions. Since in a mixed scenario the TCT chamber affect the propagation of the radio signals we propose to situate all antennas of the devices under test outside of the TCT chamber using extension cabling. In cases where this is not feasible and yet weak signal conditions need to be emulated, (programmable) attenuators could be incorporated in the infrastructure. We have so far refrained from the increase in complexity and simply performed test runs alternating antennas both inside the TCT chamber and outside.

D. Physical Parameter Extraction

For the physical characterization of motes, we employ two different approaches: On the one hand we observe long-term trends to determine the development process effects on the characteristics. With detailed snapshots of individual software builds, we perform an in-depth analysis allowing for regression testing of physical parameters.

1) Long-Term Trending: Current testbeds only has means of profiling power consumption on select nodes [2]. A supervision of all nodes under test is mandatory for assuring reliable operation and sufficient test coverage. We are using
a combination of a DSN [3] node pair with a custom power monitoring board (see Figure 5) that uses the internal ADC on the ATmega128l and a current sense amplifier in combination with the network logging tool Cacti. The device under test (DUT) can be powered from different sources (battery, line power) and the power status is sampled on request once every 5 minutes. Although limited in precision and not yielding very accurate details, basic long term trending with coarse granularity is very helpful for testbed supervision on long test sequences, giving an initial impression of what is going on (see Figure 10). Sampling more frequently and integrating the values could be an alternative but since it is infeasible to derive high quality power trace data from all nodes the benefit of the integrating solution is questionable.

2) Detailed Physical Parameter Characterization: For detailed characterization of the system performance and especially to pinpoint specific behavior detailed power traces incorporating the variation of the input are of utmost importance. In order to characterize device variations, but also to understand the interplay of the power supply (battery, regulated power, solar) and the system under varying load conditions instrumentation with a resolution >1000 samples/sec is required. Since such equipment is both costly and bulky it is not feasible to instrument every node in every testbed. Therefore we propose to instrument only a number of nodes for detailed analysis and automate the process using an interface to CI.

3) System Unit Tests for Power Traces: When trying to verify a comprehensive system, a test can nevertheless focus on an individual aspect of the execution. Focussing on the power consumption of the system or a part thereof such as a single node, power unit tests can be devised to determine if the power consumption, which is monitored during execution, is acceptable for the given test. While in standard unit testing, the test is performed on individual functional units, the power unit test is performed focussing on an individual physical parameter.

Power consumption is a crucial metric for sensor nodes, since energy-efficiency is of utmost importance for battery-operated motes. However it is not sufficient to merely look at average values. Sleep states and duty cycle patterns require intricate analysis of detailed power consumption traces. Employing our testing infrastructure, we can formulate power unit tests on the sensor node current traces.

E. Power Unit Tests

The formulation of a power unit test is based on a testcase execution for a given application. The testcase is deterministic and allows for formulating a representative reference function. Accounting for noise on measurements and variations in hardware or environmental conditions, we determine a set of bounds for a given reference function. These bounds allows for formulating a boolean valued checker that can be processed in our CI framework or even in the context of a larger testing framework by providing a single pass or fail result [5].

A reference may be determined in different ways: using a model, e.g. derived from a specification, and simulating the test case on the model or by using a golden measurement of the testcase on the target device. Reference functions differ in level of detail and accuracy, e.g. physical characterization of the sensor nodes may be incomplete.

1) Bounds for Power Unit Tests: The checking of a test measurement against a reference must allow for some variation in measurements. For hardware differences due to manufacturing variations or for differing temperature levels, we allow an offset of the range values in a specified interval. In order to compare a reference to an observation, we need to allow for a temporal shift of the reference function in time due to test run start and stop time variances introduced by the test architecture. To this end, we perform a least square analysis of the difference between the current observation and the reference trace. We compute a best fitting reference to determine shift and offset to the test specific reference in the specified intervals.

Starting point for our power unit test of the power trace is the shifted reference start: The checking of the power consumption occurs thereafter to be able to include a test initialization phase, not considering transient effects due to power-on or booting.

In order to check the power trace, the determined reference allows for determining an upper and a lower bound, which are the boundaries of an acceptance region. An error condition is asserted, if a window of consecutive readings lies outside this region. A small number of readings outside the acceptance region may be a spurious measurement device artifact. In the following we give a formal definition of the power unit test, i.e. the reference function and the upper and lower bound of the acceptance region and how they are derived for a given test scenario.

A reference function is a function $f : \mathbb{R} \rightarrow \mathbb{R}$ of time $t$ of a measured physical quantity, e.g. current dissipation. $f(t)$ is a piecewise, typically discontinuous function, which is composed of sub-functions $f_i : \mathbb{R} \rightarrow \mathbb{R}$ with discontinuities $t_i \in \mathbb{R}, i \in \mathbb{N}$ at the sub-interval boundaries of $f$.

This reference function is hulled by two bounding functions $(f^+, f^-)$, which are the boundaries of the acceptance region. In order to compute these bounding functions, we first determine intermediate upper and lower bound functions $f_{u/l}^{(i)}$, which are derived from the reference function by setting the discontinuities $t_i$ to be aligned with the test scenario.

$$f_{u/l}^{(i)}(t) = \begin{cases} f(t) & \text{if } t \leq t_i \\ f(t_{i+1}) & \text{if } t_i < t < t_{i+1} \\ f_{u/l}^{(i+1)}(t) & \text{if } t \geq t_{i+1} \end{cases}$$
which only account for the variance in value in each interval, but do not consider the sub-interval boundaries \( t_i \).

\[
f_{y,i}^-(t) = \begin{cases} 
    f_i(t) - \Delta y_i^- & \text{if } t \in [t_{i-1}, t_i) \\
    0 & \text{if } t \not\in [t_{i-1}, t_i)
\end{cases}
\]

The upper bound \( f_{y,i}^+ \) follows accordingly with adding bound values \( \Delta y_i^+ \). Variable bounds per interval allow for different granular checking. A transmitting node may have larger variation in its power than a node which is sleeping.

Additionally, we account for uncertainties in time with a symmetric variability in time \( \Delta t \) around the discontinuities \( t_i \). Thus, the lower bound of a reference function is defined as:

\[
\forall \tilde{t} \in [-\Delta t, \Delta t], i \in \mathbb{N} : 
\]

\[
f^-(t + \tilde{t}) = \begin{cases} 
    f_i^-(t_i^-) & \text{if } -f_i^-(t_i^-) + f_i^-(t_i^+) \geq 0 \\
    f_i^+(t_i^+) & \text{if } -f_i^-(t_i^-) + f_i^-(t_i^+) < 0
\end{cases}
\]

The upper bound follows accordingly:

\[
\forall \tilde{t} \in [-\Delta t, \Delta t], i \in \mathbb{N} : 
\]

\[
f^+(t + \tilde{t}) = \begin{cases} 
    f_i^+(t_i^+) & \text{if } -f_i^+(t_i^-) + f_i^+(t_i^+) \leq 0 \\
    f_i^-(t_i^-) & \text{if } -f_i^+(t_i^-) + f_i^+(t_i^+) > 0
\end{cases}
\]

Figure 6 illustrates the process of generating the final bounds used for the power unit test given the intermediate reference and the uncertainty \( \Delta t \).

Depending on the function value in the neighboring sub-interval in \([-\Delta t, +\Delta t]\) around each discontinuity \( t_i \), either the function value approaching from the left \( (t_i^-) \) or the right \( (t_i^+) \) is chosen.

Note that the definition of the reference trace is general in order to allow for granular modeling of power consumption. Although we reduce our power consumption models to linear models in the following, power unit test can be defined for models focussing on charge and discharge patterns, which are not considered in the linear models presented.

\[\text{F. Physical Stimulation}\]

Stimuli of physical parameters on the nodes under test can be used to trigger events but also to emulate sensor inputs or to exert specific, controlled changes on the nodes. As already discussed in section IV-C.2 a controlled RF environment using programmable splitters, multiplexers and attenuators would also be possible. This is a common technique in the test of RF systems requiring much care to the detail and high investments as the proper shielding of all signals is not trivial. So far the complexity of the implementation has only allowed us to use simple triggers based on voltage levels, slopes and external clock inputs.

\[\text{G. A Test and Instrumentation Architecture for Physical Characterization}\]

The following section describes our current implementation of the test and instrumentation architecture. The main component is the CC server, running a number of software projects. TinyOS applications or the PermaSense example discussed in section V. Our testing architecture is composed of a DSN testbed using a number of different target devices [3] and laboratory instrumentation using an Agilent Multifunction Switch Unit allowing to address multiple sensor node targets for hardware instrumentation, multimeters, precision counter and a programmable power source/power analyzer (see Figure 7). All instruments are accessible over TCP/IP and are controlled from the CI using a custom tool. For automated control of the environment, we have a prototypical setup of a TCT, built from a refrigerator, an integrated heater and fans for the airflow, which can be remotely controlled (see Figure 8).

As an example for an implementation of a power unit test, we look at the TinyOS Application MultihopOscilloscope. It is a multi-hop data collection application available in the TinyOS 2 distribution, which is regularly built on our test infrastructure.

The reference is described using an XML specification as depicted in Listing 1. It specifies the start and stop time of the reference checking. The reference function is specified using individual data points to define linear segments in

\[\text{Fig. 7. Logical and physical domain of the test architecture control apparatus used.}\]
the intervals. For MultihopOscilloscope, the period of the reference function is $1s$, divided into 2 distinct segments: (1) for radio idle and (2) for transmitting. We define a global time variance for a reference function of $\Delta t = 0.05s$. We use a variable bound for the reference function of $\Delta y_1 = 1.2mA$ and $\Delta y_2 = 0.5mA$. We use a variable bound for the reference function of $\Delta y_1 = 1.2mA$ and $\Delta y_2 = 0.5mA$.

<referenceTrace name='MultihopOscilloscope'>
  <start>0.4</start>
  <stop>9.5</stop>
  <xVariance>0.05</xVariance>
  <period>1.0</period>
  <points>
    <point>
      <time>0.0</time>
      <value>15.5</value>
      <yVarianceMinus>0.5</yVarianceMinus>
      <yVariancePlus>1.2</yVariancePlus>
    </point>
    ...
  </points>
</referenceTrace>

Listing 1. XML specification for MultihopOscilloscope

The resulting bound for the reference curve is displayed in Fig. 9. It also displays the start and the stop time at 0.4s and 9.5s respectively. Although a single reading is outside the reference bound, no error is asserted, as a single outlier may also be attributed to a measurement device artifact.

V. Test and Validation for Life on the Glacier - The PermaSense Case

The PermaSense project [22] aims at deploying a wireless sensor network to investigate the processes driving permafrost in steep alpine terrain and as a result of the retreat of the glaciers due to global warming, also rockfall. The targeted deployment regions in the Swiss alps are above 3500 m a.s.l. and are extremely hostile to any kind of technology, least of all tiny high-tech sensor nodes over a prolonged period of time. Extreme chilling and fast temperature variations, snow and ice, wind, avalanches and rockfall are properties that result as direct consequences of the area under investigation. Since the processes investigated are only happening very slowly, in some cases on a yearly scale, possibly even multi-year, the systems deployed for measurements must operate over very long periods of time as to minimize the servicing intervals. In a first analysis, logging only-solutions might provide a solution, alleviating the power hungry wireless link. Careful reconsideration of this approach reveals that in a significant amount of locations actually interesting to geoscientists, the avalanches and rockfall prevail and sensor stations might be swept away before a data retrieval by removal of the station could take place. Thus the reliability and power efficiency of a complete WSN-based solution is of primary importance.
In excess of a detailed specification and careful design phase, for the success of the PermaSense application, rigorous test procedures including temperature cycle testing emulating the condition on the mountain are of primary importance. Due to the fact that traveling to and from the deployment site takes days and can only be performed during the summer a failure of building blocks of the technology would result in a complete failure of the project. In the following we discuss the application of the test and validation methodology developed in the previous sections of this paper to this application. Apart from describing the steps taken and results obtained we critically discuss our experience gained during this work.

A. PermaSense System Architecture

The PermaSense system architecture under tests is based on the Shockfish TinyNode platform [23], a custom data acquisition and power board, a base station containing a satellite modem, specialized Li-SOCl$_2$ batteries, packaged in an IP68 rated protective steel enclosure. The software is based on an application that samples sensors and transmits the data over multihop to the base station using a synchronized protocol with acknowledgments. This protocol uses an interleaved scheme of 30 and 5 min for data transmission and synchronization messages respectively and 5 min synchronization intervals. It has been designed to offer 3 years lifetime based on the batteries used and depending on the influence of the environment and the amount of power used in the dominant communication. Overall the system targets the ambitious goal of a duty cycle of 1.35% and a mean current consumption per node of less than 300 uA.

B. PermaSense - Testing and Evaluation

After an inaugural development phase and a series of integration builds only using CI (see Figure 3) a first number of test jobs for the PermaSense networking and communication component were run on the testbed. This involved basic long-term power trending (see Figure 10) and log file analysis. During this early phase of the development this analysis was insightful and in cooperation with a number of detailed project meetings and code reviews lead to an overall improvement of the development process.

Memory profiling on every integration build was integrated into the toolflow as the next step (see Figure 11). Subsequently testing became more fine-grained with very specific questions on the roadmap, e.g. the influence of topology variations or excessive clustering of nodes on the networking performance. Sensitivity analysis was performed using specific parametrized versions of the software that was tested in individual test jobs with similar characteristics, e.g. 1 day, fixed number of nodes. For a detailed power/performance analysis, a combined setup of four nodes attached to the instrumentation setup described in section IV-G with two nodes inside the TCT chamber and two nodes outside as well as one node additionally attached to a precision counter for monitoring the stability of the system clock over longer periods of time was used. A basic current and voltage profile with the characteristics of the TDMA communication scheme of the PermaSense protocol can be seen in Figure 12. It can be seen that basically the protocol performs as expected with the four nodes in sync. Only occasionally a node in the TCT chamber needs to re-sync (longer active phase seen in period 3, 9 and 13) due to the influence of the cooling. A detail view on one of the communication rounds is shown in Figure 13. Here the effect of high power load on an already drained battery can be seen when load increases in the active phase of the protocol. With a lower operating limit of 2.4 V as given for the TinyNode, it is clear that we are already reaching a case of possible malfunction of the system here although the battery seems to be still healthy when the node is sleeping. Taking a closer look at the current consumption in sleep mode by zooming in on...
the current traces taken (see Figure 14) reveals considerable differences on the four nodes under test. Clearly there are device variations and spurious errors visible, but it seems as if there are also systematic errors (the lowest curve shows repetitive, unstable behavior). Even worse, this (mis-) behavior has been observed on different nodes, but it could not be observed on all nodes! To date we have not been able to find out all details of what is happening here, but the analysis has sharpened our mind and clearly requires attention to the effects visible in greater detail. However in the context of this paper we refrain from going into the full (numerical) details of the analysis of the PermaSense application as the objective here is to highlight the methodology, procedures and tool integration developed for a systematic analysis of WSN applications.

VI. DISCUSSION AND OUTLOOK

Based on our experience through collaboration in the PermaSense project, the devised testing strategy seems very promising, but it also reveals the need for multi-dimensional testing and integrated analysis based on a number of data sources. A comprehensive testing infrastructure allows for changing the configuration of the environment and the resources of the system. Additionally, detailed monitoring data collected by software and hardware instrumentation during test execution allows for an in-depth analysis of system characteristics. This helps to increase the understanding of system functionality and robustness under varying operating conditions and helps the debugging process in case of software defects.

Through the use of CruiseControl as the automation centerpiece, we have been able to develop accountable and reproducible methods for testing. The acceptance in the team has been extremely high and we have created a considerable amount of test data. However, this large amount of data, albeit being organized and structured, is exactly posing some immediate questions for the future: How to make sense out of all this data for better analysis? When is it justified to use the methodology presented and for which applications is it too much overhead? The infrastructure at this time has a high cost, both to set up but also in terms of equipment, a fact that should not be neglected. From a technical perspective, multi-user arbitration and the lack of a global time reference across all instrumentation components and an integration of more unit testing are the predominant issues.

The most significant experience gained was the integrating effect on the team. Using this methodology, it was easier to discuss issues and exchange knowledge, as there was a clear reference available to refer to the unique build and test id’s. A collaborative approach with a common reference drastically improves the development process, allowing for designing a robust and predictable system.

The test infrastructure described is not solely used for project specific work. The CruiseControl server at ETH Zurich is currently building, notifying developers and testing the TinyOS-2.x core components and applications (see Figure 9) and we are gradually merging the technology developed into a permanent setup using a testbed with a number of different target nodes (Mica, Tmote, TinyNode, EyesFXv2) as a service to the TinyOS community.

REFERENCES


