DrySim: Simulation-Aided Deployment-Specific Tailoring of Mote-Class WSN Software

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

Bijun Li, Rüdiger Kapitza
IBR, TU Braunschweig, Germany
{bli, rrkapitz}@ibr.cs.tu-bs.de

ABSTRACT

Despite intensive research in the field of mote-class Wireless Sensor Networks in recent years, real-life deployments are still challenging and systems are prone to failures. This can typically be attributed to fragile hardware or misbehaving software. Issues caused by software, often induced by the inherent constraints of resources, can be countered using simulations. However the simulation results often do not reflect those of the specific deployment.

We suggest analyzing the actual environment conditions of a deployed network and map them to a simulator. Then, based on simulations, software and parameters can be tailored to the specific deployment.

We developed two tool chains, RealSim and DryRun, and compared results from simulation runs to those acquired from two different testbeds using Tmote Sky nodes. This was done in two campaigns, each altering 2 configuration parameters from the hardware to the application layer. The presented data is based on over 1100 experiments, respectively over 270 h, on real hardware and almost 7000 simulations. The close relation of simulation and real measurements shows that our DrySim approach is feasible.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless Communication*; I.6.0 [**Simulation and Modeling**]: General

Keywords

Wireless Sensor Networks; Simulation; Testbed; Deployment

1. INTRODUCTION

Despite more than one decade of intensive research in the field of mote-class Wireless Sensor Networks (WSNs), real-life deployments are still considered difficult and systems are fragile in many ways. In fact there is a substantial record of failed experiments [2, 4]. The reasons are multifold, ranging from fragile hardware to faulty and misconfigured software. These issues are aggravated, and to a certain extent caused, by the serious resource constraints of the nodes.

Although the typical mote evolved only slightly in terms of memory and computing power during the last decade, the software running on these nodes had a great leap of its own technology. For example, providing an IPv6-based web-server is not out of the ordinary. WSN Operating Systems (OS) like Contiki [10] or TinyOS [20] provide a huge set of libraries, protocols and services. These OS target highly specialized deployments and therefore support many configuration options to tune the system to the specific needs.

Estimating the exact impact of changing a certain configuration parameter or choosing a different network protocol is difficult, even for a domain expert. Often the impact can only be determined by testing the different versions in the target environment. This is especially the case as the results are influenced by the network topology, the quality of the connections and the interactions between different software modules. If a specific configuration performs well in a certain environment (testbed), there is no guarantee that it will also perform well in another environment (real deployment).

A way of getting sound results is to test different configurations in the final environment. Alternatively one can choose a configuration that performs well in similar environments (testbed) and hope that it performs as expected in the final environment, too. While the first solution is suitable for a specific deployment, the second will work only if a generic solution is required (e.g. house automation). Both have in common that running the required amount of experiments is very laborious. The possible configurations quickly multiply up to huge numbers, and experiments must be repeated multiple times to get sound results. Not only changing environmental conditions, but also subtle effects like boot-up order, random offsets, packet loss and clock skew may have a significant impact on the outcome. For a real *wireless* deployment running a sufficient number of experiments is often not feasible.

Unlike the typical WSN mote, the average computer has developed tremendously. This provides us with the ability to simulate resource-constrained sensor nodes faster and more accurately than ever. If the simulator resembles a concrete network, it enables us to run a huge number of experiments and find a suitable setup for that specific deployment. To support this, we developed two tool chains, RealSim to map a deployment to the simulator and DryRun to support setting up experiments to test combinations of multiple different configuration options. Both are publicly available [8].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSWiM'14, September 21–26, 2014, Montreal, QC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3030-5/14/09 ...$15.00.
http://dx.doi.org/10.1145/2641798.2641838.

Protocol	Boolean	Numeric
ConitikiMAC	9	19
IP	8	8
UDP	1	1
TCP	2	8
ARP	0	2
6LoWPAN	2	2
RPL	2	11
	24	51

Table 1: Configuration options per network protocol

The paper is structured as follows: In the *problem statement* we go into detail why it is necessary to run many experiments in the target environment to get a good setup and why this is not feasible (Section 2). An *overview* of our idea of simulating the target environment to gain a suitable setup is given in Section 3. This is followed by the *implementation details* (Section 4). We then discuss *related work* (Section 5). Our approach is *evaluated* by comparing results acquired from the simulation to those from the testbed (Section 6). In *discussion* we look at the current limits and opportunities of our approach (Section 7), before we end with a *conclusion* (Section 8).

2. PROBLEM STATEMENT

Modern WSN operating systems like Contiki and TinyOS usually provide a huge amount of possible configuration options. They are supposed to provide maximum flexibility and to allow tailoring the system to the requirements (e.g., bandwidth, power, memory, network size, network topology, network protocol, timeliness, and many others). In this context configuration does not only include adjustments of parameters but also selecting alternative code paths or modules (e.g. MAC-Protocol). Although most configuration options are within the network layer, they can be found in all layers of the system. As a result, one has to be an expert on all layers of a system to get optimal results. Based on the failed deployments in the past [4] this is a challenge even for an experienced WSN-system developer, let alone an application developer.

For example Contiki's IPv6 over Low Power Wireless Personal Area Network (6LoWPAN) stack consists of 7 protocols, providing a total of 75 configuration options (Table 1). This does not include flags that adjust the code due to other features like debugging, tracing and encryption, but only the protocol specific options.

Interactions.

The situation of having many configuration options is aggravated by the fact that different options closely interact with each other. These interactions are not only within a certain layer but often also cross-layer: Increasing the sampling frequency of a sensing application is likely to also require adjustments at the network layer. This can include having to change to a network protocol that is better suitable for more traffic. Not always obvious to see, these dependencies between the configuration options make it inherently complicated to adjust them.

Network Portability.

It is not only necessary to adjust the network stack to the application but also to the underlying network. A big network usually needs a different configuration (e.g. bandwidth, size of the routing table) than a small one. In addition to the size of the network, the topology and link attributes have a big impact on the performance of a network. Consequently, experiences gained in one environment can only be transferred to another environment with great care.

Running Trials.

The most reliable way to find a good configuration is running trials. In a testbed, flashing different firmware versions and collecting data can normally be done using a reliable, wired connection. It is still time-consuming, as testing 25 different configurations for 20 min each will take over 8 h. If repeating them four times to get more robust results, this already takes more than a day. As we show in our evaluation, a detailed analysis that is supposed to uncover interactions requires testing a lot more configurations and they must be repeated more often.

When testing in a real, *wireless* deployment, Over The Air (OTA) programming is required. To the additional overhead of reliably distributing different firmware to all nodes, the risk of bringing the network into an indeterminate state or even bricking a node is added. The additional overhead and possible manual interactions make this approach unfeasible.

3. OVERVIEW

We aim at a generic approach that allows tailoring the software system to a specific deployment. As discussed before, achieving this for complex software requires a rather large testing campaign that cannot be executed on the target deployment. We therefore propose *trace-based* simulation for *deployment-specific* testing of WSN software. It can be structured in five consecutive steps:

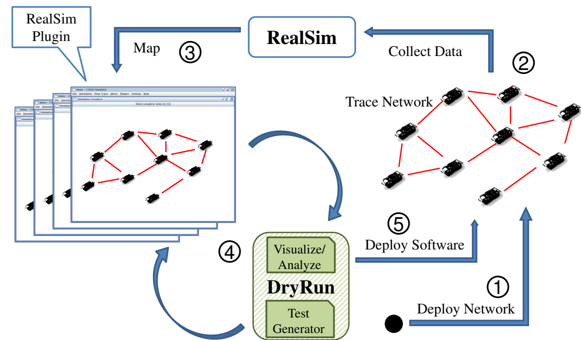


Figure 1: Simulation-aided deployment-specific tailoring of WSN software at a glance

1. ① Of course our approach does not eliminate thorough testing before deploying a WSN, but it does target the phase during or right after a deployment; therefore deploying the WSN is the first step.
2. ② After that the connectivity between the deployed nodes is profiled to obtain the network topology.
3. ③ The acquired data are used to configure a simulator in a way that it resembles the tested network.
4. ④ Running multiple Monte Carlo simulations in parallel allows testing a huge configuration space within a short time.
5. ⑤ After evaluating the simulation, a suitable configuration can be chosen and programmed to the nodes.

Simulation ④.

The selection of the simulator is crucial for our approach, as its accuracy has a direct impact on the results. One of our goals is to show the effects of tuning certain parameters while taking the whole software system into account. Using a WSN simulator that is able to emulate the mote's hardware, it is possible to run the same binary code as on the real nodes. It is therefore ignorant of the implementation and is not dependent on a certain OS or library. Further on it allows uncovering hardware specific issues like unaligned reads or problems caused by the tool chain. These are not necessarily triggered when compiling for a different target.

Due to the simplicity of the RISC-based micro controllers used in mote-class WSNs, these simulators are also very timing accurate. Therefore the side effects caused by concurrency or by using algorithms that perform badly on a specific platform and then miss timing constraints, can be observed as well.

Based on these considerations we chose the Cooja Simulator [24]. While the network models are rather simple compared to some sophisticated network simulators like NS2 or OMNeT++, it integrates MSPSim [12], which supports emulating the Sky mote we used in our testbed, including its CC2420 radio hardware. By setting the seed of Java's Pseudo Random Number Generator (PRNG), Cooja supports reproducible Monte Carlo Simulations. In our setup the PRNG is used to determine start-up offsets and whether a packet is received based on the Packet Reception Ratio (PRR).

Data Acquisition ②.

There are multiple ways of acquiring connectivity data [6, 18, 22, 28]. Most of these have a rather sophisticated underlying protocol, often triggering measurements. For RealSim, of which we already presented an early version [26], we decided to choose a simple approach that does not rely on any synchronization between nodes.

Each node sends beacons with increasing IDs. When a node receives such a beacon it obtains the Received Signal Strength Indication (RSSI) and Link Quality Index (LQI) from the radio chip and derives the PRR from the ascending order of the ID packed into the beacon. While a node is not receiving data, it regularly samples the RSSI of the background noise.

Mapping data ③.

As Cooja provides a Direct Graph Radio Medium (DGRM), it is the perfect target for our data. It allows us to set PRR, RSSI and LQI for each directed connection. Additionally we extended Cooja to support setting the background noise for each node.

Using the RealSim tool chain it is possible to extract a certain time frame and load it into the simulation. The RealSim plugin then replays the sample by adjusting the DGRM.

Testing Configurations ④.

While RealSim is responsible for making the simulation more realistic, DryRun supports running large campaigns, testing many configurations. To create a campaign DryRun provides a test generator. As input, the test generator takes a script that describes how to set up the test environment and which configuration options to test. For each experiment a shell-script is generated, which initiates an isolated test environment (e.g. copy files), executes the experiment and collects the results. Setting up separate environments is necessary to be able to run the experiments in parallel or even distribute them to multiple machines without side effects.

The experiment scripts collect only the raw logs. The *dataextractor* extracts the relevant information, brings it in correlation with the input parameters of each experiment and saves them in a format that can be processed using tools like R. As the area of interest depends on the specific research question and due to the volume of the data there are currently no tools to further automatically analyze the data.

4. IMPLEMENTATION DETAILS

Challenges.

Developing a tool like RealSim is not as trivial as it seems, even if major parts of the infrastructure are already provided by Cooja, MSPSim and Contiki. For RealSim to work, all components of a huge software stack have to work together seamlessly. If the results are not as expected, this might be because the model is not accurate enough, or there is a bug somewhere in the tool surveying the network, the OS it is built upon, the processing tools, MSPSim, Cooja, the RealSim plugin, the experiment setup tools or the scripts evaluating the results; maybe our hypotheses that it is possible to simulate real networks was wrong.

To demonstrate this we will discuss the way of a packet's RSSI-value from acquisition to simulation. The radio hardware averages the signal strength over the first 8 symbol periods. It then calculates RSSI value that corresponds to the signal strength in dBm with an offset as *signed byte*. In Contiki it is handled as *unsigned word*, which must be casted back, before pre-processing the data in the node. After passing through the serial and being processed, the RealSim plugin sets the DGRM configuration in dBm. The averaging over the 8 symbols is done by Contiki's MSPSim adaption layer, before it is passed to MSPSim and written to the virtual register of the CC2420 emulator.

Data Acquisition ②.

To allow a random distribution, as well as a constant beacon rate, we randomly distribute the points of time to send a beacon over a certain time frame/episode (e.g. 6 beacons within 80 s). For each received packet we extract the RSSI and LQI provided by the radio chip. Based on the ID we detect whether a new episode started and calculate the PRR and average RSSI and LQI for that neighbor node and episode. It is possible to either send the aggregated data to a sink using the network, or print it directly to the serial.

We sample the background noise at a rate of 300 ± 50 ms, unless the radio chip indicates that it is currently sending or receiving. At the end of the local episode, the average RSSI value is calculated.

For some nodes in our testbed we were able to observe a background noise ranging from a RSSI value of -52 to -15. This roughly maps to -97 to -60 dBm. Considering that a delta of 37 dBm represents a factor of over 5000, averaging the RSSI value instead of the corresponding energy levels gave quite accurate simulation results. It is also much simpler and requires fewer resources.

```
implicit val exp = new Experiment

val files = new GetFile("src/*")
exp.addstep(files)

val mk = new Make
mk.addConf("CCA_THRESH",-54, -20, 2)
mk.addConf("TEST_RATE", 5, 30, 5)
exp.addstep(mk)

val cooja = new Cooja("test1.csc")
cooja.addRandRange(0, 9)
exp.addstep(cooja)
```

Listing 1: Example experiment setup

Mapping Data ③.

The data acquired by the nodes is printed to the serial and then saved together with a time stamp in a log file. From the log a certain time span can be selected, which is then converted to simple format supporting commands (e.g. `setedge`, `rmedge`) and a time when they are to be executed. At the beginning of a simulation RealSim loads this data and executes the commands at the given simulation-time.

Experiment Generator ④.

The main part of the DryRun toolkit is the experiment generator. The generator uses the Scala runtime compiler to script the setup. Listing 1 shows a simplified setup. First a new Experiment object is created (l. 1). Then all the files from the `src` directory are copied to the build environment (l. 3). Explicitly adding it to the experiment ensures the correct order (l. 4). Lines 6 to 9 configure the experiment to be built using `Make` with `CCA_THRESH` ranging from −54 to −20 in steps of 2, and `TEST_RATE` from 5 to 30 in steps of 5. Finally Cooja is run with `test1.csc` as configuration file using 10 different seeds for Cooja's PRNG.

The listing will create a campaign of 850 experiments. Each experiment consists of a folder containing the configuration of the experiment and a shell-script. The shell-script sets up the environment in the temporary directory, runs the experiment and copies the resulting logs back to the experiment-folder. As the scripts are self-sufficient they can be executed in parallel or distributed over multiple machines. Additional functions include creating symbolic links, checking out files from git and retrieving additional information from the experiment environment. It is also possible to select a certain length of a network trace using different start offsets.

For our evaluation we used the same infrastructure to generate the experiments run on the testbed. Instead of running the simulation we flashed the firmware and collected the serial output.

5. RELATED WORK

Similar approaches have been published before. For example Marchiori et al. traced their testbed and developed their own simulator called WsnSimPy to replay these traces [22]. There are also quite some other works that tried to reproduce testbed results based on generic network simulators like NS2 or OMNeT++ [14, 17, 23] or the WSN Simulator Castalia [3, 19, 25]. As opposed to our approach, all of them only focus on simulating the network layer. It is therefore possible to quickly make a conceptual evaluation of a network protocol, but effects caused by the concrete implementation, OS, libraries used, timing and hardware are neglected.

In their position paper Greg et al. give an overview of different approaches for realistic simulations [13]. As opposed to us they suggest improving trace-based simulation in OMNeT++/MiXiM and TOSSIM. Both simulators do not emulate the target nodes but are directly interfaced. As TOSSIM is implemented as target platform for TinyOS, it is at least possible to see effects caused by the actual implementation and OS, but not the platform itself.

Using a very simple setup, Gama et al. show that the code running on the node does influence the results [1]. Instead of emulating the node, they add delays to the different processing layers (Hardware, Media Access Control (MAC), Application, etc.) to improve their simulation results. Though this approach provides a better performance than emulating each instruction, it has the drawback of inaccuracy and side-effects caused by concurrency, which cannot be detected.

Besides our generic approach of testing the whole system, there are also approaches that target certain layers. pTunes [28], for example, continuously optimizes the MAC layer. While these approaches probably yield better results for supported use-cases, they must be specifically adjusted to the code in use. Our solution, in contrast, is able to test any compile time parameter, independent of the code it addresses.

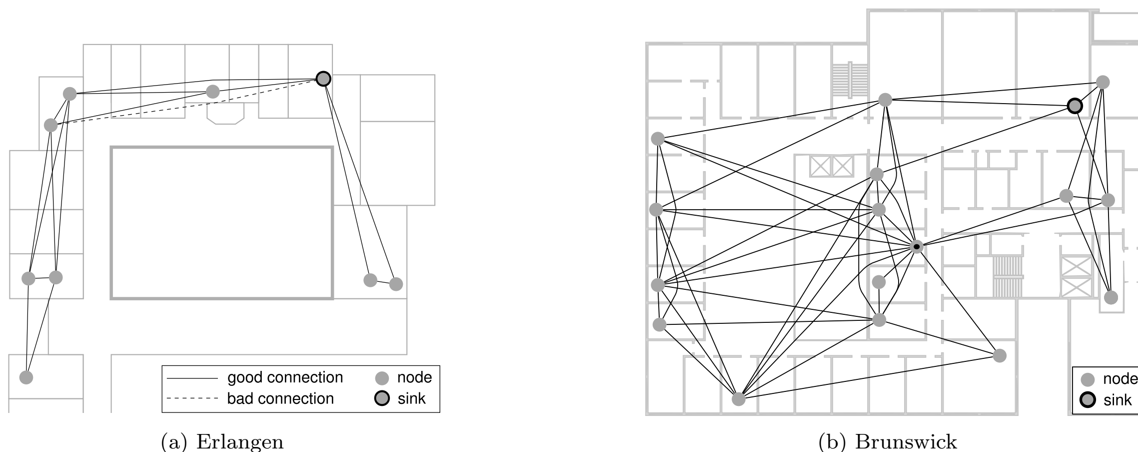
In [16] He et al. use simulation to predict the PRR for the connections in an office environment before the deployment. Considering the strong fluctuations and the effects we monitored when changing the position of a node, we do not think it possible to get reliable results without actual measurements. Their approach seems promising to gain a good initial setup, though.

The WiseML-plugin [21] for Cooja also supports adjusting the DGRM and is very similar to our RealSim plugin. The WiseML format itself is more generic and supports detailed descriptions of the environment and nodes. This includes, for example, positions and sensor data. Although the plugin does support setting the temperature read by a node, it can only set the PRR, but not LQI and RSSI. As we show in our evaluation the RSSI has a significant impact on sophisticated network protocols like ContikiMAC.

6. EVALUATION

To verify our approach, we did not evaluate the approach itself, but evaluated the crucial point: Does the network that is mapped to the simulator resemble the traced, real network? For this we chose four different configuration parameters and compare the simulation results to two testbeds. The parameters were not selected by the expected novelty of the results, but because they are often used for system tuning. With our experiments we want to show that the simulation is able to yield results that are similar to those of the testbed, and that these results differ between testbeds. Therefore the results in general are probably not surprising for a domain expert.

As code base we used Contiki's UDP server/client example, where each client regularly sends data to the sink. We extended the code to allow querying the Energest [11] statistics at the end of an experiment. Energest, is part of Contiki and accounts the time of the system being in a certain state. For example, the time of the CPU being in low power mode or the radio being turned on. Multiplied with the energy input of the system being in that specific state, the energy consumption can be estimated.



(a) Erlangen (b) Brunswick

Figure 2: The testbed in Brunswick is larger and closer meshed. The solid lines represent good, the dashed line a flaky connection. The sink is marked with a black border.

In our evaluation we tested the following parameters:

CCA Threshold The Clear Channel Assessment (CCA) threshold is a hardware configuration parameter of the `CC2420` radio chip. Based on its value the hardware decides whether the channel is clear to send data. The current CCA status can also be queried via a hardware pin; a feature used by the ContikiMAC protocol.

RDC/MAC Contiki distinguishes between the MAC and Radio Duty Cycling (RDC) layer. Currently Contiki supports four different RDC options: ContikiMAC [9], CX-MAC, an adjusted version of the X-MAC protocol [5], SICSLoWMAC that puts packets into 802.15.4 frames and nullrdc which passes the data on to the MAC layer. Choosing an RDC does not only change a simple parameter, but also includes different code into the binary.

CCR The Channel Check Rate (CCR) is the rate at which the RDC layer wakes up and checks for other nodes to send data. A low CCR lets the receiver wake up less frequently while the sender must potentially send more packets until the receiver wakes up to receive the packet.

Packet Rate In our test application we altered the packet rate, at which the "user"-program sends data to the sink.

At the end of each experiment we collected energy and network statistics for each node – over 50 parameters in total. When investigating why the specific configuration behaves the way it does, these can be very helpful. As we want to compare simulation and testbed, rather than understand the behavior of a specific protocol, we chose two metrics that are important for WSN deployments: The time spent with the radio turned on (rx and tx) and the number of packets that arrive at the sink. The CC2420 radio chip is one of the biggest energy consumers and uses about the same amount of energy for sending and receiving when in the default settings.

To aggregate the data, we took the mean of all nodes except the sink. We excluded the sink because it distorts the two metrics we chose: It does not send any packets and, in the example we derived our experiments from, the radio is always on.

As discussed we try to circumvent the limitations of running multiple experiments on a real deployment. Yet, to get a sound ground truth for our evaluation we had to do exactly that. Consequently all our experiments were run in a testbed and we used the serial as convenient method of gathering data.

When running experiments on real hardware, it is not unlikely that the results show artifacts caused by random effects like packet loss or changes of the environment. We tried to mitigate this by running each experiment several times and executing them in a random order. In this way each configuration had the same chance of being executed at daytime or at night, when conditions were typically more stable. We tried to reconstruct this behavior for our simulations by not only using a different PRNG-seed for each repeated simulation, but also a different snippet from our trace. For each configuration we used the same seeds and snippets. Thus the simulation yielded the same results for different configurations, if the change had no impact on the behavior of the node. This effect can be seen if Figure 3a where the best result for SICSLowMAC at the Brunswick-testbed does not change for a CCA threshold above -39.

6.1 Testbeds

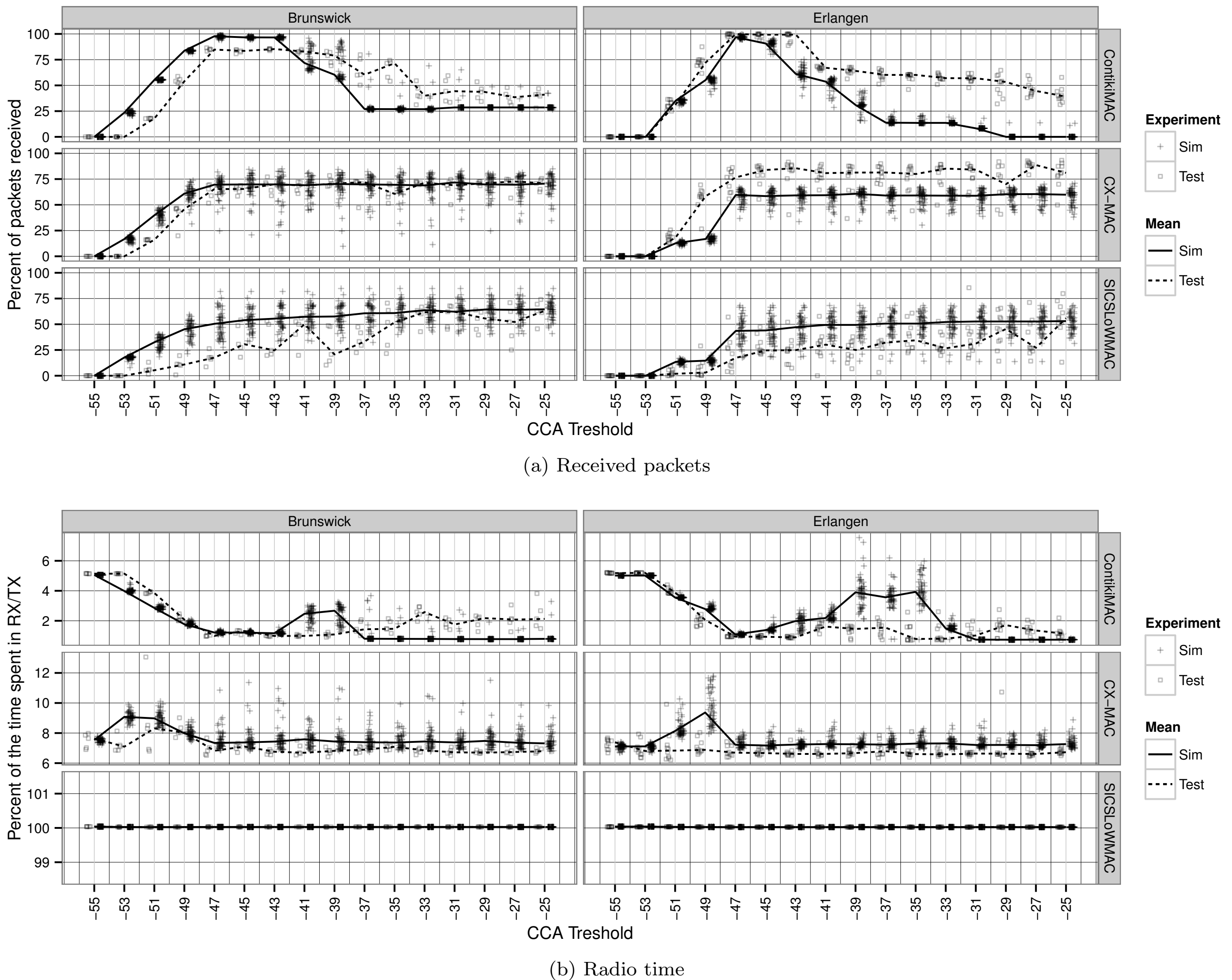
Our two testbeds are located in an office environment at the universities in Erlangen and Brunswick, Germany. Both testbeds are managed using Wisebed [7]. Due to network delays and the Wisebed infrastructure, it is not possible to accurately control the node bootup order. To minimize the effects of nodes booting in a system-inherent order, we randomly delayed the reset command of each node at the beginning of an experiment.

Erlangen.

The network in Erlangen (Figure 2a) consists of 9 Sky nodes that were placed as far apart as possible while still providing a stable connection. The room in the middle is a lecture hall with stronger walls, blocking the connectivity. Solid lines typically have a PRR of 100 % while the dashed line has around 10 %. The short ranged connections have a RSSI value of about -40 dB while the long range connections are at around (-85 ± 5) dB. This sink is marked with a black border.

Brunswick.

The testbed in Brunswick is also located in an office environment and consists of 17 Sky nodes. Figure 2b shows



(a) Received packets

(b) Radio time

Figure 3: Impact of the CCA threshold on different RDC layers in simulation and test bed. Each dot represents the mean of the nodes of one run. The line connects the median of all runs with the same configuration.

the good connections between the nodes. In comparison to Erlangen the network is meshed much closer. We chose a sink (dark border) outside the central mesh to increase the number of packets that must be routed. The figure also shows that, especially in an office environment, the distance is not necessarily related to the connectivity. In the center there is a node marked with black dot, which has extraordinary connectivity in all directions, while the node right next to it, to the lower left, is only connected to two nodes. Similarly the node to the top right of the sink has better connectivity than the sink itself.

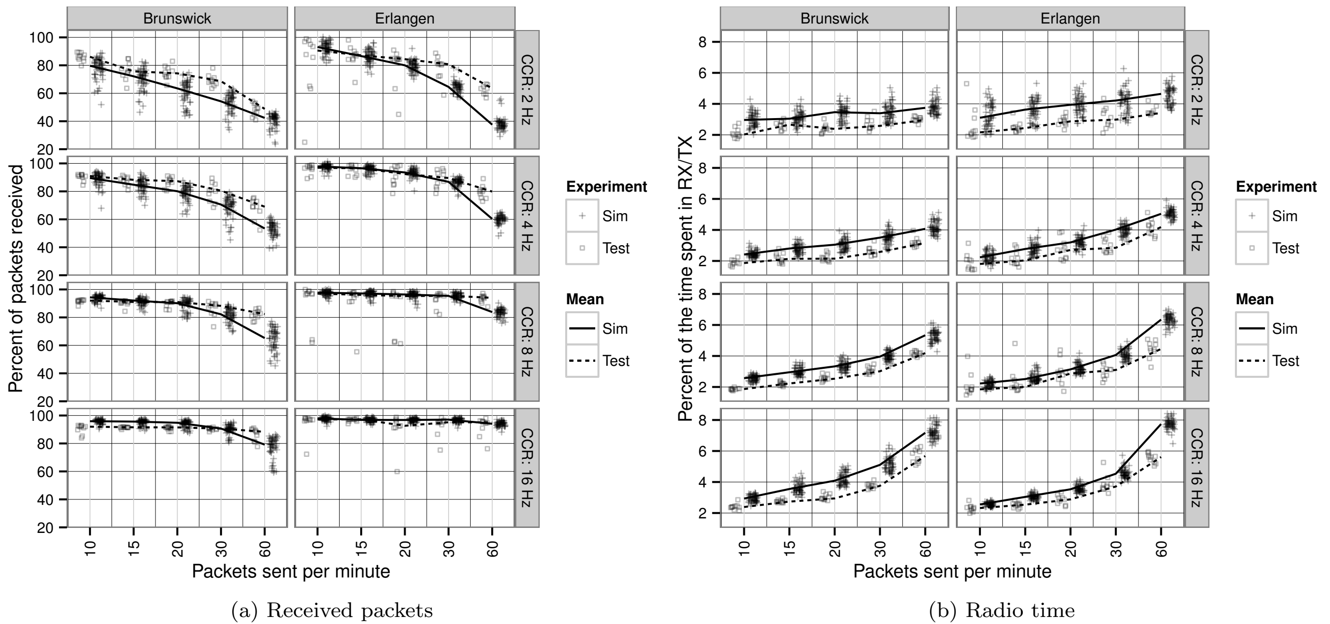
6.2 CCA vs RDC

In this campaign we investigate the impact of changing the hardware-configurable CCA as well as the RDC network layer. We tested the three available RDC layers (ContikiMAC, CX-MAC and SICSLoWMAC) and configured the CCA threshold from -55 to 25 in steps of 2. The runtime of an experiment was 20 min. We simulated each configuration 50 times, which resulted in a total of 2400 simulation runs for each location. In Erlangen each configuration ran 10 and in Brunswick 5 times. Unfortunately this also means that the jitter had a stronger impact. In Erlangen it took over a week to run the campaign and in Brunswick almost 4 days, flashing each node 480 and 240 times, respectively .

Figure 3 shows the results of the experiment. Every point shows the mean of all nodes except the sink. The results from the simulation are shown as cross and are placed to the right, while the results from the testbed are represented as square and placed to the left of the corresponding discrete value. The median of the experiments are connected with a line that is solid for the simulation and dashed for the testbeds.

The results show that although there are deviations, there is a clear similarity between simulation and testbed. Part of deviation can be accounted to the insufficient noise model.

The impact of this flaw in the radio model can be seen in the average number of packets received from each node (Figure 3a). If the measured noise is higher than the CCA threshold, the hardware does not send any data. Therefore no packets are sent if the CCA threshold is too low. This can be seen for all three protocols. While this matches quite well in



(a) Received packets

(b) Radio time

Figure 4: If the number of packets being sent is too high or the CCR too low, the networks starts losing packets. Each dot represents the mean of the nodes of one run. The line connects the median of all runs with the same configuration.

Erlangen, there is an offset of about two between simulation and testbed in Brunswick. In Brunswick, measured peaks of the background noise were not as high as in Erlangen and therefore the average value was lower. This allowed the simulated hardware to send packets at a lower CCA threshold.

The strong variation of the background noise in Erlangen is also accountable for the strong drop of the simulated ContikiMAC for higher CCA thresholds in Erlangen. ContikiMAC also uses the CCA threshold to test whether any other node is sending data. If the measured noise level is below the threshold, the radio is turned off right away, not being turned on long enough to receive a packet. In the noisy environment in Erlangen, the radio was kept on due to the noise and then received a packet by chance. Half of the nodes had a direct connection to the sink, which did not do any radio duty cycling. It was therefore sufficient to receive a single packet from the sink, to get the routing information and send data to the sink for the rest of the experiment. In Brunswick there was no such drop in the simulation because the nodes were placed closer together and therefore their signal was strong enough to keep the receiving radio on. If we extend our campaign to higher CCA thresholds we can probably see this effect in Brunswick as well. This cannot be seen for CX-MAC and SICSLowMAC, which only evaluate the CCA threshold when sending packets.

We did not investigate why CX-MAC outperforms the simulation in Erlangen and SICSLowMAC performs so much worse in the testbed. Nonetheless there are some effects of SICSLowMAC that can be seen in both the simulation and the testbed. For example, in Brunswick the results are better than in Erlangen. For CX-MAC we do see a slight positive trend towards higher CCA thresholds. A similar trend was also observed on the testbed.

As the SICSLowMAC has no duty cycling the radio is turned on all the time (Figure 3b). Besides the offset of the CCA threshold by two in Brunswick, which we already discussed, one can see that especially in Erlangen the simulated radio on time is significantly higher around a CCA threshold of about 37. This again is caused by the non-existing fluctuation of the radio signals in the simulation. In the simulation the signal strength is only adjusted every episode ($\approx 80\,\mathrm{s}$ in our setup). Thus, if the signal strength oscillates around the CCA threshold, the sending node must wait at least until the next episode to be able to successfully send its packets. In the testbed the next change, and therefore the chance to receive the packet, typically is in the next RDC and thus within less than a second.

6.3 Application vs. MAC

To show the interactions of application, MAC and testbed we varied the rate at which the packets are sent to the sink. To avoid the effects already discussed, we chose to use ContikiMAC with a CCA threshold of -45, which performed well in both testbeds and the simulation. As second parameter we chose the CCR value. It tells ContikiMAC how often per second to check the channel for other nodes sending data. If it is too high, a lot of energy is spent checking for radio traffic, while if it is too low, the bandwidth is reduced and packets might be dropped.

For the CCR the lowest value supported is 2 Hz and the default is 8 Hz. We therefore chose 2, 4, 8 and 16 Hz. For the packets we chose a rate of 10, 15, 20, 30 and $60\,\mathrm{min}^{-1}$. Based on the high number of packets we reduced the time of the experiment to 5 min. Each configuration was simulated 50 and tested 10 times on both testbeds.

The results are presented in Figure 4. We prepared the data the same way as in the previous trial: We calculated the average number of packets received and the average radio time for each experiment and plotted a cross for the simulation and a square for the testbed results. We then connected the median of the experiments with a line – solid for the simulation and dashed for the testbed.

Figure 4a shows the percentage of sent packets that were received. Even in a perfect environment, chances are high that not all packets arrive at the sink, as some are likely to be on their way when the experiment is ended. As expected, the much larger testbed in Brunswick goes into an overload situation much earlier than Erlangen. As the radio model drops all packets in case of a collision, the testbed outperforms the simulation in the overload situation. Considering the jitter of the measurements the results are quite close, though and the point clouds almost always overlap.

In terms of radio on time (Figure 4b), the testbed always outperforms the simulation, not only in an overload situation. We also account this to the radio model, as packets need to be re-transmitted more often. Even though, there is a clear correlation between simulation and testbed, and the dot clouds typically overlap.

7. DISCUSSION

Our evaluation shows that it is possible to get quite realistic results, even with a very simple radio model and simplified assumptions when tracing the network. To the best of our knowledge it was also the most comprehensive comparison between testbed and simulation using such a holistic approach. Actually, due to the many trials preceding the presented numbers, we found multiple bugs in all layers of the system. One of them, located in the implementation of the radio hardware, ignored the configured CCA threshold. Before we started varying the threshold in our experiments, it caused the simulations to yield unexplainable discrepancies to our real world experiments and was quite hard to find. Although we are certainly not the first to suggest verifying simulators using testbeds, with our tools this can now be accomplished by running a sufficient number of experiments with reasonable efforts.

We see multiple leverage points to improve our results. One is a more dynamic model for the background noise and the signal strength. This will likely have a significant influence on protocols like ContikiMAC, which heavily rely on the RSSI. To allow tracing networks outside a testbed, such a model must also be suitable to aggregate the data on the nodes themselves, though.

Further on, the signal strength is currently only evaluated by the radio hardware, but not the radio model. Therefore weak signals can interfere with strong ones. In this context Halkes et al. had very promising results with a Signal-to-Noise Ratio (SNR)-based model [15].

Yet another problem we faced while comparing the simulations with the testbed was the amount of data we collected. The numbers presented show only 2 of the over 50 different attributes we collected. With simulators the amount of data that can be collected suddenly becomes unlimited. They allow monitoring state that is too complex or changes too often to be printed to the serial [27]. While this provides great opportunities, it also requires the support of specialized tools to handle the data.

The amount of data that can be collected, as well as the huge number of different configurations when combining multiple parameters requires further tooling support. Once our simulation results are accurate enough, it is possible to use machine learning and evolutionary algorithms to find better configurations.

As soon as the number of experiments is only limited by the available processing power, many other things to investigate come to mind. For example, it is possible to test each node with an individual configuration. This might allow leaf-nodes to save more energy. It is also possible to derive scenarios from the traces, for example to test whether failing nodes can be tolerated. Yet another possibility is to enrich the simulation with an energy model and try to increase the network lifetime opposed to node lifetime.

Currently we are only targeting the pre-deployment stage. Extracting the required information from the packets sent over the network anyway would make this approach even more powerful. It would allow running simulations on an updated model of the network, without having to flash a special firmware. It is then possible to detect potential issues caused by the changed environment, or just test the next firmware version to be deployed.

8. CONCLUSION

To gain a reliable, robust and long-living mote-class WSN network, it is often not sufficient to use the default configuration, but the system must be tailored to the specific used case and deployment. This is likely to become a tedious task, even for a domain expert. We therefore presented our DrySim-approach of first mapping a deployment to the simulator and then tailoring the system based on simulations. By parallelizing the simulations it is possible to systematically test a huge number of different configurations.

To verify our approach we created a set of tools called RealSim that we used to trace testbeds and map them into the Cooja WSN simulator. We chose the Cooja simulator because it allows us to emulate binary code and to execute it in simulation time. To allow testing different parameters by instrumenting Cooja and RealSim we developed a second set of tools, DryRun, which simplify setting up campaigns to test different configurations.

In the evaluation we made a comprehensive comparison between the testbed and the simulation. Although there is plenty of room for improvements, it shows that our DrySim-approach is feasible and can support finding a suitable configuration for a specific deployment using simulation.

9. ACKNOWLEDGMENTS

This work was partly supported by the Bavarian Ministry of State for Economics, Traffic and Technology under the (EU EFRE funds) grant no. 0704/883 25 and the German Research Foundation (DFG) under grants no. FOR 1508.

10. REFERENCES

[1] Modelling the impact of software components on wireless sensor network performance. pages 1 – 6.

[2] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments. In Proc. of the 6th ACM conf. on Embedded network sensor systems (SenSys 2008), pages 43–56, 2008.

[3] L. Bergamini, C. Crociani, A. Vitaletti, and M. Nati. Validation of WSN simulators through a comparison with a real testbed. In Proc. of the 7th ACM workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks (PE-WASUN 2010), pages 103–104, 2010.

[4] J. Beutel, K. Römer, M. Ringwald, and M. Woehrle. Deployment Techniques for Sensor Networks. In Sensor

Networks, Signals and Communication Technology, chapter Deployment, pages 219–248. Springer, 2009.

[5] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In Proc. of the 4th int. conf. on Embedded networked sensor systems (SenSys 2006), page 307, 2006.

[6] M. Chini, M. Ceriotti, R. Marfievici, A. L. Murphy, and G. P. Picco. Demo: TRIDENT, untethered observation of physical communication made to share. In Proc. of the 9th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2011), SenSys '11, pages 409–410, 2011.

[7] G. Coulson, B. Porter, I. Chatzigiannakis, C. Koninis, S. Fischer, D. Pfisterer, D. Bimschas, T. Braun, P. Hurni, M. Anwander, G. Wagenknecht, S. P. Fekete, A. Kröller, and T. Baumgartner. Flexible experimentation in wireless sensor networks. Communications of the ACM, (1):82–90, 2012.

[8] DryRun and RealSim authors. Dryrun and realsim git repositories. `https://github.com/cmorty/{dryrun|realsim}`.

[9] A. Dunkels. The contikimac radio duty cycling protocol. Technical report, Swedish Institute of Computer Science, 2011.

[10] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proc. of the 1st IEEE Workshop on Embedded Networked Sensors (Emnets-I 2004), 2004.

[11] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In Proc. of the 4th workshop on Embedded networked sensors (EmNets 2007), pages 28–32, 2007.

[12] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. Poster Abstract: MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards. In European Conf. on Wireless Sensor Networks (EWSN 2007), Poster/Demo session), pages 1–2, 2007.

[13] K. Garg, A. Förster, D. Puccinelli, and S. Giordano. Towards Realistic and Credible Wireless Sensor Network Evaluation. In Proc. of the 3rd Int. ICST Ad Hoc Networks (ADHOCNETS 2011), pages 49–64, 2011.

[14] C. Guo, M. Jacobsson, and R. V. Prasad. A Case Study of Networked Sensors by Simulations and Experiments. In Proc. of the 11th Int. Conf. on Thermal, Mechanical Multi-Physics Simulation, and Experiments in Microelectronics and Microsystems (EuroSimE 2010), pages 1–5, 2010.

[15] G. P. Halkes and K. G. Langendoen. Experimental evaluation of simulation abstractions for wireless sensor network MAC protocols. EURASIP Journal on Wireless Communications and Networking, pages 24:1—-24:2, 2010.

[16] D. He, G. Mujica, J. Portilla, and T. Riesgo. Simulation tool and case study for planning wireless sensor network. In Proc. of the 38th Annual Conf. on IEEE Industrial Electronics Society (IECON 2012), pages 6024 –6028, 2012.

[17] P. Hurni and T. Braun. Calibrating Wireless Sensor Network Simulation Models with Real-World Experiments. In Proc. of the 8th Int. IFIP-TC 6 Networking Conf., LNCS, pages 1–13, 2009.

[18] A. Kamthe, M. A. Carreira-Perpiñán, and A. E. Cerpa. M&M: Multi-level Markov Model for Wireless Link Simulations. In Proc. of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2009), SenSys '09, pages 57–70, 2009.

[19] E. Kolega, V. Vescoukis, and D. Voutos. Assessment of network simulators for real world WSNs in forest environments. In Proc. of the 2011 IEEE Int. Conf. on Networking, Sensing and Control (ICNSC 2011), pages 427 –432, 2011.

[20] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. In Ambient Intelligence, pages 115–148. Springer, 2005.

[21] Q. Li, F. Österlind, T. Voigt, S. Fischer, and D. Pfisterer. Making wireless sensor network simulators cooperate. In Proc. of the 7th ACM workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks (PE-WASUN 2010), pages 95–98, 2010.

[22] A. Marchiori, L. Guo, J. Thomas, and Q. Han. Realistic performance analysis of WSN protocols through trace based simulation. In Proc. of the 7th ACM workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks (PE-WASUN 2010), pages 87–94, 2010.

[23] G. Möstl, R. Hagelauer, G. Müller, and A. Springer. A Network and System Level Approach towards an Accurate Simulation of WSNs. In Computer Aided Systems Theory (EUROCAST 2011), LNCS, pages 17–24. Springer, 2012.

[24] 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, 2007.

[25] H. N. Pham, D. Pediaditakis, and A. Boulis. From Simulation to Real Deployments in WSN and Back. In IEEE Int. Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2007), pages 1–6, 2007.

[26] 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 2011), pages 95–96, 2011.

[27] M. Strübe, F. Lukas, and R. Kapitza. Demo Abstract: CoojaTrace, Extensive Profiling for WSNs. In Poster and Demo Proc. of the 9th European Conf. on Wireless Sensor Networks (EWSN 2012), pages 64–65, 2012.

[28] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele. pTunes: runtime parameter adaptation for low-power MAC protocols. In Proc. of the 11th int. conf. on Information Processing in Sensor Networks (IPSN 2012), pages 173–184, 2012.