

Diss. ETH No. X

Enabling Inspection of Wireless Embedded Systems

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by

ROMAN LIM

MSc ETH in Electrical Engineering and Information Technology
born September 21, 1981
citizen of Zug, Switzerland

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Kay Römer, co-examiner
Dr. Jan Beutel, co-examiner

2017



Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. y

Roman Lim

Enabling Inspection of Wireless Embedded Systems



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

Diss. ETH No. X

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Kay Römer, co-examiner
Dr. Jan Beutel, co-examiner

Examination date: January 10, 2017

Abstract

Wireless embedded systems are networks of wireless nodes that perform tasks that involve communication, sensing, and actuation, embedded in the environment. Since these networks can be large and must run unattended and safely for a long period of time, costs and energy are important design factors. As a consequence, system designs are commonly characterized by very little resources in terms of available energy, memory, communication bandwidth or processing power. Due to these resource constraints, inspecting and building such systems is a challenging task.

In this thesis, we enable increased observability and controllability when testing wireless embedded systems in a pre-deployment testbed setting. By measuring several functional and non-functional properties, tightly time synchronized across the entire network, studying these systems becomes possible in an unprecedented level of detail. For this purpose, we propose a new testbed architecture, present new time synchronization protocols and develop algorithms to trace the program execution.

The main contributions of this thesis are:

- We design and build a new testbed architecture that enables multi-modal inspection and control of devices under test. The combination of the testbed's services provide a previously unattained level of visibility into wireless embedded systems.
- We study the effect of time-of-flight in multi-hop time synchronization protocols, and we propose a new protocol that can effectively compensate propagation delays that stem from dissimilar wave propagation times between nodes. Compared to previous approaches, our protocol achieves up to $6.9 \times$ better synchronization accuracy.
- We design and implement a new distributed data acquisition system that combines fast data acquisition with accurate time synchronization to increase the possible level of detail of observations in a testbed.
- We describe a new algorithm that can be used to automatically place instrumentation code into existing programs for control flow

tracing. The novelty stems from using time information of the program to keep the induced overhead of instrumentation code low.

- We showcase the usability of the testbed by employing it in a project to count devices based on radio interference.

Contents

Abstract	i
List of Figures	v
List of Tables	ix
1 Introduction	1
1.1 Challenges of Inspecting Wireless Embedded Systems	2
1.2 State of the Art	4
1.3 Thesis Contributions and Road Map	5
2 FLOCKLAB: A Testbed for Tracing and Profiling of Wireless Embedded Systems	11
2.1 FlockLab Services	14
2.2 FlockLab Architecture	15
2.3 Benchmarking FlockLab	23
2.4 FlockLab in Action	30
2.5 Related Work	40
2.6 Summary	42
3 Time-of-Flight Aware Time Synchronization for Wireless Embedded Systems	45
3.1 Related Work	47
3.2 Impact of Propagation Delay	49
3.3 Time-of-Flight Aware Time Synchronization	53
3.4 Implementation	58
3.5 Evaluation	60
3.6 Summary	69
4 Fine-Grained Tracing of Time Sensitive Behavior in Wireless Sensor Networks	71
4.1 Related Work	73
4.2 Enabling Fine-Grained Tracing	75
4.3 Architecture	77
4.4 Implementation	83
4.5 Evaluation	84
4.6 Summary	89
5 Testbed Assisted Control Flow Tracing for Wireless Embedded Systems	91

5.1	Related Work	93
5.2	Control Flow Tracing	95
5.3	Implementation	106
5.4	Evaluation	109
5.5	Case Study	114
5.6	Summary	118
6	Passive, Privacy-preserving Counting of Smartphones via ZigBee Interference	119
6.1	Background and Terminology	123
6.2	DEV _{CNT} Overview	125
6.3	Estimating Smartphone Counts	127
6.4	Detecting and Counting Active Wi-Fi Scans	130
6.5	Implementation	140
6.6	Evaluation	142
6.7	Discussion	150
6.8	Related work	151
6.9	Summary	153
7	Conclusions and Outlook	155
7.1	Contributions	155
7.2	Possible Future Directions	157
	Bibliography	159
	List of Publications	173

List of Figures

2.1	The FLOCKLAB observer board with target nodes attached	12
2.2	High-level schematic of the FLOCKLAB observer hardware.	16
2.3	Processing of GPIO events, power samples, and serial data on an observer	18
2.4	Layout of the FLOCKLAB deployment	22
2.5	Distribution of the error on time intervals between GPIO events	24
2.6	Timing errors of power profiling	26
2.7	Power supply stability and power measurement accuracy	27
2.8	Comparative performance analysis of CTP/LPL on multiple platforms	32
2.9	GPIO trace showing a misconfiguration of CTP and LPL on TinyNodes	33
2.10	Energy measurements using FLOCKLAB	35
2.11	Clock drift measurements using FLOCKLAB	37
2.12	Simultaneous power profiling and GPIO tracing on FLOCKLAB	38
3.1	Experimental setup to show the influence of the capture effect on slot length measurements	53
3.2	Two-way round-trip measurement.	54
3.3	Round-trip measurements based on time information embedded into time synchronization packets	55
3.4	Unidirectional links prevent round-trip measurements	57
3.5	Time diagram of message transmission	59
3.6	Message delay distribution of the CC430 radio on a single link	60
3.7	Testbed layout and software enforced path for the <i>short</i> and <i>long</i> line topology	62
3.8	Comparison of available link delay estimates for immediate and delayed forwarding	63
3.9	Percentage of compensated links during floods	63
3.10	Cumulative distribution of synchronization errors for TATS, Glossy and PulseSync	66
3.11	Maximal synchronization error over time	67
3.12	Average synchronization errors over time per node	68
4.1	Overview of a single FLOCKDAQ observer.	77

4.2	Overview of clock control algorithm	79
4.3	Data format of GPIO tracing and power profiling packets	80
4.4	Data flow and memory structure of the data acquisition system	80
4.5	Generation of a synchronized PPS signal	82
4.6	Physical arrangement of the data acquisition board on FLOCKLAB	83
4.7	Cumulative distribution function of PPS signal error	86
4.8	Layout of the network during evaluation of time synchronization.	87
4.9	Distribution of synchronization error on node 4	88
4.10	Absolute time synchronization error of FLOCKDAQ	89
5.1	Overview of the tracing process.	95
5.2	Examples of leveraging time information to infer the exact execution path	97
5.3	Excerpt of a CPU clock speed measurement on an MSP430	97
5.4	Example program with a while loop and resulting annotated graph	99
5.5	Example of an initial witness set	101
5.6	Subgraph of the control flow graph that is examined by the admission test	101
5.7	Example graph with one strongly connected component	103
5.8	Number of non-blocking witnesses in each instrumented binary	112
5.9	Program memory utilization for the original program and instrumented variants	112
5.10	Runtime overhead caused by added instructions	114
5.11	Average pairwise distance of code coverage between nodes in the experiment	116
5.12	CPU speed profile at boot time for two different clock calibration algorithms.	117
6.1	IEEE 802.11b/g (Wi-Fi) and IEEE 802.15.4 (ZigBee) channels	123
6.2	High-level view of DEV _{CNT}	123
6.3	RSSI sampling, processing, and communication over time in DEV _{CNT}	125
6.4	Distribution of scan counts in real-world datasets	128
6.5	Fraction of devices that can be seen within a certain interval in the University Wi-Fi dataset	129
6.6	Example of three original signals and the corresponding trace of RSSI samples	131
6.7	State machine to determine begin and end of a signal	132
6.8	CDF of probe lengths in the University trace	133
6.9	RSSI trace of an active Wi-Fi scan	134
6.10	Classification of signal clusters	136
6.11	Sum of autocorrelations for an example binary vector	137
6.12	Scan detection rate across six smartphones	145

6.13	Average scan detection rate against received signal strength . . .	146
6.14	Estimated and real number of Wi-Fi enabled devices	147
6.15	Estimated number of smartphones in a real world trial	149

List of Tables

2.1	Clock cycles and time needed to set a GPIO pin on FLOCKLAB targets	15
2.2	Pairwise timing error of GPIO services	24
2.3	Minimum interval between consecutive GPIO events for 99% and 100% capture rate	29
2.4	Comparison of FLOCKLAB to other existing testbeds supporting distributed power measurements	40
3.1	Overview of reported synchronization errors	48
3.2	Slot times estimated during Glossy floods.	53
3.3	Structure of synchronization packets.	57
3.4	Standard deviations for message delays on different platforms. .	60
3.5	Accuracies measured for different protocols.	65
4.1	Maximal event rates of different target platforms.	75
4.2	Measured throughput burst sizes and maximum continuous rate.	85
4.3	Standard deviation and range of the error across the network . .	86
5.1	Recent node platforms	92
5.2	Binary size and number of program elements for applications used in the evaluation.	110
5.3	Number of witnesses used in instrumentation	111
5.4	Reliability and radio duty cycle of Glossy.	113
5.5	Code coverage of example applications.	115
6.1	Explored signal features	134
6.2	List of smartphones and operating systems	143
6.3	Scan detection performance with and without interference . . .	144

1

Introduction

Networked wireless embedded systems—known as sensor networks, cyber-physical systems or the Internet of Things—have been applied in various fields such as home automation, personal health and medicine, and surveillance, see e.g. [Sta14]. These networks consist of wireless sensor nodes that perform tasks that involve communication, sensing, and actuation. To enable such applications, unattended and safe operation for a long period of time is a prerequisite. Consequently, driving factors like energy sources, cost or size lead to system designs that are characterized by very little resources in terms of memory, energy, communication bandwidth and processing power. Working at resource limits increases the possibility and probability of faults. In addition to these resource constraints, unreliable wireless communication channels and the complexity inherent in distributed systems renders the task of building and deploying such networks challenging.

In this thesis, we focus on improving observability and controllability of such systems in a pre-deployment testbed setting. Better observability leads to better understandings of the system, thus helping to find errors and performance issues early on in the design process. System observability is the basis of methods for automated testing, verification and optimization [Woe10].

Testbeds, i.e., installations that facilitate experiments on real hardware, have become a fundamental part in the development cycle of wireless embedded systems. As opposed to simulation, testing a system in a real distributed environment exposes it to external factors like multi-path signal propagation, signal attenuation, temperature changes, or hardware variations. As it is difficult to accurately model and simulate

these influencing factors, testing on real hardware is needed for proper evaluation and validation.

Testbeds provide engineers and researchers with the services to facilitate testing of wireless embedded systems. A testbed not only has to implement means to measure and acquire information about functional properties (output values, program states), but also non-functional properties like *timing* or *power consumption* play an important role in wireless embedded systems:

- Low-power MAC protocols try to save energy by turning on the radio transceiver only when needed. For successful communication, sender and receiver nodes need to have their radio transceivers turned on at the same time. Coordination of such efforts are sensitive to timing errors—incorrect timing can severely hamper the system performance.
- The scarcity of energy resources in wireless embedded systems emphasizes the need to spend the available energy carefully. For example, in a multi-hop network, where information has to be relayed over several hops, running out of energy on relatively few nodes can lead to an unconnected network, and therefore a degraded system performance.

We extend the current state of art in testbed infrastructure by contributing a testbed architecture that is capable of measuring key properties of program execution and system state in wireless embedded systems, e.g., power dissipation, program states or the control flow of a program. We develop methods to enable these measurements in a well synchronized way, while keeping instrumentation overhead small enough to only minimally alter the timing behavior of sensor network applications.

1.1 Challenges of Inspecting Wireless Embedded Systems

Challenges to inspection of wireless embedded system in the context of testbeds are related to time sensitive operations on resource constrained devices and distributed measurements.

Observability under scarce resources. Due to the little resources available in wireless embedded systems, only a small amount thereof can be allocated to debugging tasks on the device under test. Program instrumentation to record information about the program state consumes

additional processing time, and extra memory is needed to store the generated debug information. Extracting the necessary information during runtime is difficult, already in the setting of a pre-deployment testbed.

Instrumentation alters timing of execution. Observation of program behavior should be as non-intrusive as possible, in order not to alter program timing. For example in low-power MAC protocols, timing plays an important role in achieving power efficient communication. Inserting statements that log certain program activities are bound to influence the program timing. Such statements could store or update values in non-volatile memory for later extraction and inspection, or print out program state information on a serial line during execution. However, accessing flash memory, or formatting and transmitting information using `printf` statements introduce non-negligible delays. Care must be taken to avoid instrumentation statements in time-critical program parts. There is a trade-off between completeness of observation and the level of intrusiveness when instrumenting a program.

Distributed observations. Measurements relating to a distributed system—recorded at different physical locations—need to be consolidated into a global view in order to make sense of them. Putting measurements together requires either time synchronization or other means to globally order events by time of occurrence. The level of synchronization depends on the requirements of the measurements. On a radio message level, the correct order in a sequence of messages might be sufficient. If it comes to low-power MAC protocols, finer granularities are needed to investigate the interaction between nodes. Examples of such interactions are synchronized sleep and active states of radio transceivers, or interference. In Glossy, a flooding architecture that relies on constructive interference, packet transmissions on neighboring nodes need to be started with less than $0.5\ \mu\text{s}$ time difference to create constructive interference [FZTS11].

Increasing data volumes. Data generated by inspection needs to be acquired, collected and processed in order to make use of it. Testbeds typically provide an out-of-band channel to collect and disseminate data for every individual node. However, extending possibilities to observe and control embedded systems leads to an increase of possible data that can be extracted. When designing testbed architectures, meeting the required real-time properties and providing sufficient bandwidth and processing power is a challenging task.

1.2 State of the Art

As discussed in the following, three important problems remain unsolved by prior work related to inspection of wireless embedded systems:

1. A testbed architecture that supports multi-modal recording of power, serial communication and digital states.
2. A method to accurately synchronize measured data in a mixed indoor/outdoor environment without involving expensive cabling.
3. Support for network-wide, low-overhead tracing of control flow in wireless embedded systems.

Testbeds. As stated earlier, testing and verifying an implementation on a testbed is a fundamental step in the development cycle of a wireless embedded system. Current testbed designs focus on different aspects like relocatable testbeds [RHLG10], controlled mobility [JGMdDO10], distributed power measurements [HHP⁺08] or large scale networks with hundreds of devices [EAR⁺06]. Still, access to nodes is mainly provided through a serial port, enabling `printf` style debugging. While serial ports are sufficient for a number of long-term profiling tasks, logging serial data is highly intrusive and not suited for inspecting timing-sensitive code.

Time synchronization. Time synchronization is an essential service for many distributed systems. Its goal is to keep clocks progressing with a similar speed and offset at different locations. A prominent example is the network time protocol (NTP) to synchronize time in the Internet [MMBK10].

In the context of wireless embedded systems, the focus of time synchronization lies on scalability and energy efficiency [MKSL04]. Specifically, a time synchronization protocol in a wireless sensor network has to address following challenges: (i) the network topology might change due to changing channel quality, e.g., caused by moving nodes or changes in the environment. A protocol should foresee mechanisms to quickly adapt to topology changes. (ii) Time needs to be synchronized over multiple hops, possibly much farther than in the Internet (NTP considers a node at hop count (stratum) of 16 as disconnected). For example, a deployment on the Golden Gate Bridge exhibited a diameter of 46 hops [KPC⁺07]. (iii) The hardware is significantly less powerful than a general purpose computer. This affects clocks, radio communication and processing power. Accuracies reported by current state-of-the-art protocols are in the lower microsecond range for networks with a diameter of up to 30 hops [LSW14].

If sub-microseconds synchronization is required, methods based on Ethernet (e.g., PTP [ptp08]) or satellite communication (e.g., GPS) are commonly used, either relying on expensive cabling or energy demanding signal processing.

Tracing wireless embedded systems. Traces of program execution help to understand how a certain program state has been reached, providing important information to debug or optimize a system. In the context of this thesis, we discuss tracing approaches in two different settings: (i) in a deployment and (ii) a pre-deployment testbed setting.

Deployed systems cannot rely on any external hardware support—all data collection and processing needs to be done on the node itself. Recorded data is either stored on (non-volatile) memory for later extraction, or sent over the wireless network to a sink for online assessment. Clearly, these solutions have to cope with very little resources, in order not to overload the available communication network, but also to keep the energy consumption at a reasonable level. In this category, we find approaches that instrument at different levels of abstraction, e.g., function calls [LST15], basic blocks [SEZ10], or non-deterministic inputs [TSBE15]. Commonly, due to limited resources, these approaches can only trace the program for a short period of time, and also only selected parts of the program.

In a pre-deployment setting, external hardware offloads parts of the tracing. In addition, hardware debugging support can be leveraged, e.g., dedicated tracing ports or on-chip debuggers. The availability and functionality of such debugging features differs between microcontroller architectures and families. In the Minerva testbed [SK13], every node is connected to a debug board to interface with the node’s debugging module. Such solutions depend heavily on the feature set of a platform’s debugging module—there is no generic method to trace any microcontroller architecture. In the real-time domain, execution times are measured using GPIO lines [BMB10] for worst case execution time analysis.

1.3 Thesis Contributions and Road Map

Testbed infrastructure (Chapter 2). We design and implement FLOCKLAB, a new testbed architecture that allows for previously unattained level of detail in inspection of wireless embedded systems in a pre-deployment environment. Different to previous approaches, FLOCKLAB has the ability to acquire and control different modalities in combination throughout the entire testbed with virtually no impact on the devices under test, thus

increasing their controllability and observability.

As detailed in Chapter 2, FLOCKLAB consists of following building blocks:

- The key element in FLOCKLAB is a powerful *observer* platform that pairs with up to four different devices under test (*targets*). This design decision enables tight control of actions and accurate measurements on the attached targets, while providing sufficient processing power to handle possibly large data volumes.
- Every observer instance is capable of measuring and controlling different modalities: (i) By externally tracing digital GPIO lines, state on the devices under test can be extracted at a minimal overhead on the devices themselves. (ii) Control of GPIO lines enables timely controlled actions on a target. (iii) High resolution power measurements (up to 56 ksps) allow to put program state and energy consumption into perspective. (iv) Control of the target voltage facilitate studying the influence of different voltages, allowing e.g. to simulate battery depletion. (v) Serial communication between observer and target can be used for `printf` debugging.
- A publicly accessible web interface makes the FLOCKLAB deployment at ETH Zurich available to the research community. The deployment consists of up to 31 observer nodes.

We benchmark the services provided by FLOCKLAB, and we demonstrate its utility for testing, debugging, and evaluating wireless embedded systems through several real-world test cases.

Time of flight aware multi-hop time synchronization (Chapter 3). To study measurements at network scale, data needs to have a common time scale. In Chapter 2, we use NTP [MMBK10], the standard time synchronization protocol for the Internet to synchronize clocks, resulting in sufficient synchronization to study node interactions on a radio message level. However, depending on the phenomena that we want to observe, better synchronization is required.

In the area of wireless embedded systems, different multi-hop synchronization protocols have been proposed. In this chapter, we study the limits of two state-of-the-art protocols and propose the time of flight aware time synchronization protocol (TATS) to improve synchronization accuracy. Specifically, Chapter 3 contributes the following:

- We assess the impact of propagation delay on PulseSync [LSW14] and Glossy [FZTS11], two state-of-the-art time synchronization protocols that treat propagation delay as a negligible quantity. We

find that the errors introduced by this assumption are in a similar range as the overall synchronization error, therefore motivating the need for incorporating propagation delay compensation into a protocol design.

- Based on these insights, we design TATS, a new time synchronization protocol that combines following building blocks to further push the limits of time synchronization: (i) Time information is propagated using *fast flooding*. (ii) Without additional packets, propagation delays are *measured* and (iii) compensated during a flood. (iv) Consecutive synchronization points are combined in a linear regression to compute offset and speed of local clocks.

We show in testbed experiments that our new protocol outperforms Glossy and PulseSync by a factor of up to 6.9, while achieving sub-microsecond synchronization error over 22 hops.

Synchronized data acquisition system (Chapter 4). When instrumenting a program using low-overhead GPIO changing instructions, tracing these changes using an external monitoring device allows to observe system behavior at a very detailed level. Since such instructions impose only little impact on the performance and timing of the target, even time sensitive parts of a program can be instrumented and observed.

Recording such traces in a distributed system poses several challenges to the data acquisition system. In this chapter, we derive requirements for such a system. Based on these requirements, we design and implement FLOCKDAQ, an extension to FLOCKLAB to enable fine grained tracing in the testbed. As detailed in Chapter 4, the building blocks of FLOCKDAQ are the following:

- We design and implement a new data acquisition system, which is built around a field-programmable gate array (FPGA). This design decision facilitates fast and accurate capture of GPIO state changes and power samples.
- We introduce a combination of an open-loop controller and a feedback loop to digitally control the offset and the speed of the data acquisition system's internal clock. Input to this control mechanism is an external time pulse.
- To achieve network wide clock synchronization, we generate the time pulse on every observer using a protocol that combines Glossy's time synchronization with a jitter reduction filter.

In our evaluation, we find that this data acquisition system is able to capture state changes at the maximal rate emitted by state-of-the-art wireless sensor node platforms. In addition, FLOCKDAQ aligns concurrently recorded traces within $1\ \mu\text{s}$ with an empirical probability of 99.9%.

Testbed-assisted control flow tracing (Chapter 5). This chapter gives an answer to the question of *where* to place instrumentation code (*witnesses*) in a program in order to (i) faithfully reconstruct the control flow of a program after execution, while (ii) imposing only minimal runtime overhead on the program. We design an algorithm that can be used to automatically place instrumentation code into existing programs. The contributions of this chapter are the following:

- We introduce a new witness placing algorithm that is based on an elaborate static analysis of the program binary, which extracts the control flow graph of the program together with execution time information for each possible path in the graph. By exploiting this time information, we extend an existing non-time-aware placement algorithm to reduce the number of required witnesses substantially.
- We implement this algorithm in a tool that automatically instruments and replays MSP430 based microcontroller programs.

Testbed experiments involving several typical sensor network applications show reductions in runtime overhead of up to 38.3% when instrumenting the entire program binary. Overall, instrumentation for control flow tracing adds an overhead of 19% during execution. Performance metrics of a time sensitive application (Glossy [FZTS11]) show negligible sensitivity to our instrumentation method, i.e., we could not measure any significant difference between the original and an instrumented version.

Use case: Inspecting deployed systems by overhearing radio interference (Chapter 6). Since its first installation in the year 2012, the FLOCKLAB testbed has been widely used by people all around the world, both for educational purposes in lectures and for design and evaluations for scientific publications. In addition, it has been instrumental to various projects in the computer engineering group [ZFM⁺12, KBT12, FZMT12, FZMT13, ZFMT13, KBT13, SZDF⁺15, SBBT15, LMT16, ZMK⁺, SDFG⁺17]. In the last chapter of this thesis, we exemplarily show the benefits of our testbed infrastructure in a specific use case. In this project, we aim at counting wireless devices, in this case smartphones, passively and without intrusion.

In the particular case of smartphones, privacy is an issue because some messages sent by smartphones contain unique identifiers that

allow to create user profiles. We approach this problem by overhearing messages by means of signal strength measurements on a radio receiver that cannot decode these messages, i.e., we use a IEEE 802.15.4 receiver to overhear Wi-Fi transmissions. We design and implement `DEVCNT`, the first system that supports real-time counting of unmodified Wi-Fi enabled smartphones while preserving the privacy of the smartphone owners. `DEVCNT` consists of the following contributions:

- By using novel signal processing algorithms that execute on a multi-hop network of ZigBee devices, `DEVCNT` detects and counts active Wi-Fi scans performed by smartphones based on characteristic patterns in RSSI traces.
- Combining these counts with statistical information about the average active scanning rate, `DEVCNT` faithfully estimates the number of Wi-Fi enabled smartphones.

`DEVCNT` trades some fidelity in the smartphone count estimations for improved privacy. Results from controlled and real-world experiments show that `DEVCNT` provides estimates with an accuracy of up to 91 %.

2

FLOCKLAB: A Testbed for Tracing and Profiling of Wireless Embedded Systems

Testbeds play a key role in developing real-world wireless embedded systems by providing the facilities to debug and evaluate protocols and applications in a controlled, yet realistic distributed environment. This chapter establishes a fundamental testbed infrastructure for multi-modal inspection of wireless embedded systems. We will later on refine the data acquisition system of the testbed in Chapter 4.

A review of the spectrum of existing testbeds yields a long list: relocatable testbeds to study applications in the intended target environment [RHLG10], testbeds with robots for controlled mobility experiments [JGMdDO10], testbeds performing distributed power measurements [HHP⁺08], homogeneous testbeds with hundreds of devices [EAR⁺06], and emulation platforms [GES⁺04] and heterogeneous testbed federations [CPC⁺12] to assess large-scale services on thousands of nodes.

Despite this broad spectrum, the current practice of testbed-assisted development revolves around LED and `printf` debugging: developers use the nodes' on-board LEDs to observe conditions in the running program and `printf` statements to log diagnostic messages, performance counters, or program state over the serial port. However, it is well known that `printfs` alter the timing behavior and are therefore unsuitable for

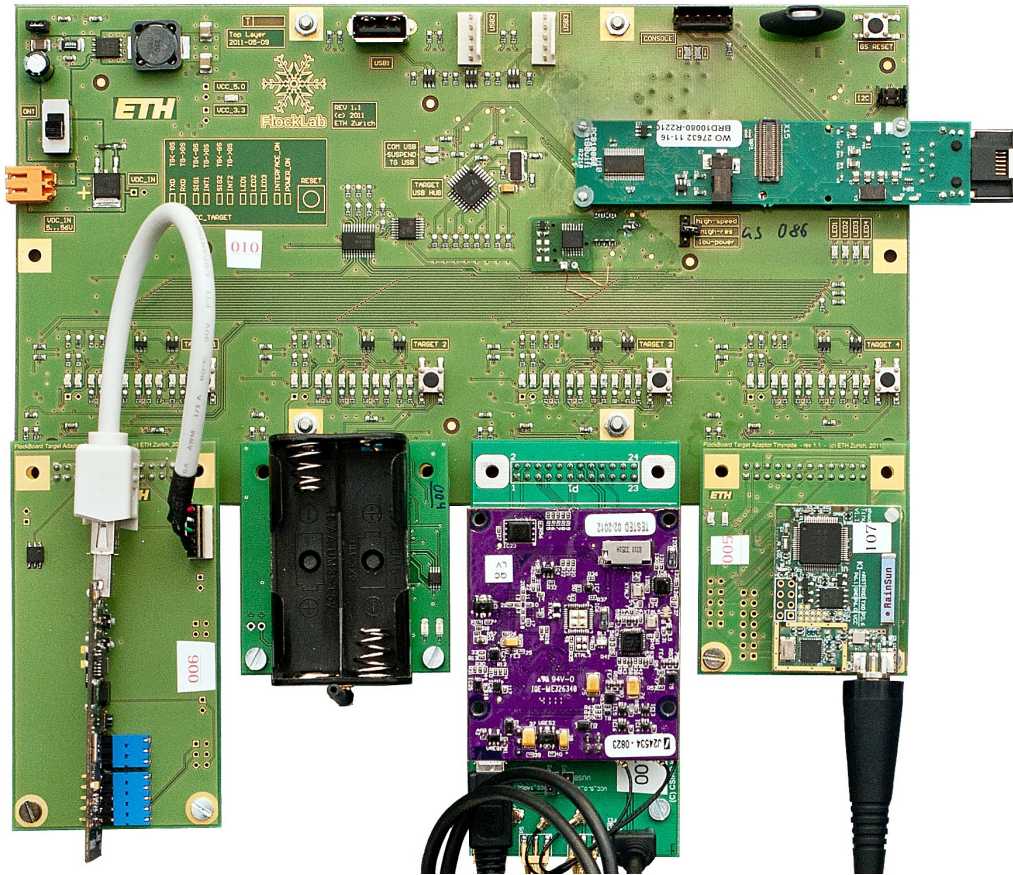


Figure 2.1: FLOCKLAB observer with TelosB, IRIS, Opal, and TinyNode connected via interface boards.

analyzing timing sensitive code such as radio drivers and MAC protocols. Perhaps one reason for the unchallenged popularity of these techniques is their ease of use [SSC10]. Another reason is that current testbeds allow access to the devices under test only through the serial port. As a result, developers are left with no other option than to use `printfs`, a means suitable for a number of long-term profiling tasks but cumbersome, highly intrusive, and unsuitable for detailed investigation of interactions among multiple devices, especially real-time issues.

The current solution for debugging low-level software and hardware interactions is a logic analyzer and a mixed-signal oscilloscope allowing to capture and trigger events of interest (e.g., changes in program state or packet transmissions) at high timing resolution. Different from `printfs`, setting digital GPIO pins on a node introduces a known delay of just a few clock cycles, which makes GPIO tracing a powerful tool for debugging timing sensitive code. The required equipment, however, limits the setup to a few nodes on a table, bearing little resemblance to a real multi-hop setting.

The main contribution of this chapter is FLOCKLAB, a testbed with services providing a previously unattained level of visibility into wireless embedded systems. FLOCKLAB’s novelty stems from the combined capability of tracing and actuating logical state changes at high level of detail, accurate timing information in the low microsecond range, and the possibility to profile and control power over the whole testbed. By coupling a powerful, stateful *observer* platform directly with every device under test, the *target*, FLOCKLAB decouples data acquisition and control from centralized data collection. FLOCKLAB leverages distributed target-observer pairs with deep local storage that are capable of capturing event and power traces of all targets locally, simultaneously, synchronously, and at high rates without sacrificing on timing accuracy or incurring data rate limitations of traditional backchannel-based testbeds [HKWW06, WASW05].

As such, FLOCKLAB combines the capability of a logic analyzer, power analyzer, serial data logger, and programmable power supply with network synchronization and deep local storage adjacent to each target—distributed across the entire testbed. FLOCKLAB also supports multiple target platforms, allowing for comparative analysis of applications and protocols on the same physical topology. It performs distributed power measurements at higher rate, resolution, and synchronization accuracy than prior testbeds. Users may apply power profiling and GPIO tracing against all targets to correlate power samples and logical events, or dynamically adjust the target supply voltage to emulate battery depletion effects. Section 2.1 details the services available in FLOCKLAB.

Section 2.2 presents the design of FLOCKLAB to meet the challenges that arise when providing these services. Based on our FLOCKLAB deployment at ETH Zurich, which consists of 30 observers in a mixed indoor/outdoor setting that host Opal, IRIS, TinyNode 184, and TelosB targets as shown in Figure 2.1, we benchmark FLOCKLAB’s performance in Section 2.3. We find, for instance, that FLOCKLAB can capture GPIO events reliably up to a rate of 10 kHz; it can timestamp distributed events and power samples with an average pairwise error below 40 μ s; and it measures power draw with an average error smaller than 0.4% over six orders of magnitude, while providing a highly stable and programmable supply voltage. We further demonstrate in Section 2.4 the utility of FLOCKLAB through various real-world test cases, including an experiment in which we take a detailed look into packet propagation and power draw during a Glossy network flood [FZTS11]. Gaining similar multi-modal insights at this level of detail would hardly be feasible with any prior testbed. We review related work in Section 2.5 and conclude in Section 2.6.

2.1 FlockLab Services

FLOCKLAB delivers new insights into wireless embedded systems by providing the following key services.

GPIO tracing. An observer can trace level changes of five target GPIO pins at a rate of up to 10 kHz. Setting a GPIO pin takes only 2–5 clock cycles on current target platforms, as listed in Table 2.1. Thus, using simple code instrumentation, this service allows for low-overhead tracing of events of interest; for example, a trace of packet exchanges may help to debug a MAC or routing protocol. Like a mixed-signal oscilloscope that can trigger on digital signals and capture on analog signals, it is also possible to couple GPIO tracing with GPIO actuation and power profiling using a callback mechanism: upon detecting a defined pin edge, an observer can set another GPIO pin or start measuring power.

GPIO actuation. An observer can set, clear, and toggle up to three target GPIO pins, one of which is the target’s reset pin, either periodically or at predefined times. This is useful, for example, to create controlled experiments by triggering some action on all targets at the same time, such as starting or stopping the nodes, turning on the radio, transmitting a packet, or freezing and logging a state variable.

Power profiling. An observer can sample the current draw of the target at a maximum frequency of 28 kHz when operating the ADC in high-resolution mode and up to 56 kHz when operating it in high-speed mode. FLOCKLAB defaults to the high-resolution mode since it provides a higher signal-to-noise ratio (SNR) than the high-speed mode, as further described in Section 2.2.4. Users specify time windows during which this service should be running. Resulting power traces can aid in developing energy-efficient applications and have also been used for conformance testing [WLT09] and failure diagnosis [KLL⁺10].

Adjustable supply voltage. An observer can dynamically adjust the target supply voltage between 1.8 V and 3.3 V in steps of 100 mV. To introduce repeatable voltage changes, users can select from a range of predefined charge/discharge curves or define their own voltage-time profiles. This can be used, for example, to study discharge-dependent behavior.

Serial I/O. Finally, an observer can read or inject data over the target’s serial port, which is a standard service available on almost any testbed. FLOCKLAB supports ASCII data, TOS messages, and SLIP datagrams, making it compliant with the serial communication available in state-of-the-art operating systems like TinyOS and Contiki.

FLOCKLAB allows a user to run any combination of the above services

Platform	Microcontroller	Speed	Cycles	Time
TelosB	MSP430 F1611	4 MHz	5	1,250 ns
TinyNode 184	MSP430 F2417	12 MHz	5	417 ns
Opal	ARM Cortex-M3	96 MHz	5	52 ns
IRIS	ATMega1281	8 MHz	2	250 ns

Table 2.1: Clock cycles and time needed to set a GPIO pin on selected FLOCKLAB targets. *The known, minimal delay of GPIO tracing allows for low-overhead debugging of timing sensitive code.*

simultaneously and synchronously on any subset of observers. FLOCKLAB *accurately timestamps* data acquired during a test *across all services and observers*, thus providing previously unattained insights into local and distributed system behavior both in detail and at scale. To the best of our knowledge, this makes FLOCKLAB unique in the spectrum of testbeds for wireless embedded systems.

2.2 FlockLab Architecture

Providing the above services presents several challenges to the design of FLOCKLAB. This section highlights these challenges and describes FLOCKLAB’s hardware and software architecture designed to solve them.

2.2.1 Challenges

- **Minimum disruption:** FLOCKLAB must not perturb the behavior of the system under test beyond the minimum necessary to obtain the desired measurements.
- **High accuracy and resolution:** FLOCKLAB needs to provide highly accurate power samples over a dynamic range that spans six orders of magnitude in current draw, from sleep currents of just $2\ \mu\text{A}$ on a TinyNode up to active currents on the order of 100 mA. The resolution of power measurements and event traces must approach or exceed 10 kHz to capture ephemeral radio events, such as clear channel assessments, which last only 100–200 μs .
- **Time synchronization:** FLOCKLAB must tightly time-synchronize the observers against a stable global clock, so as to precisely correlate events and power samples of one observer as well as across multiple observers. With sampling rates of at least 10 kHz, events and power samples must thus be timestamped with 50 μs accuracy or better.

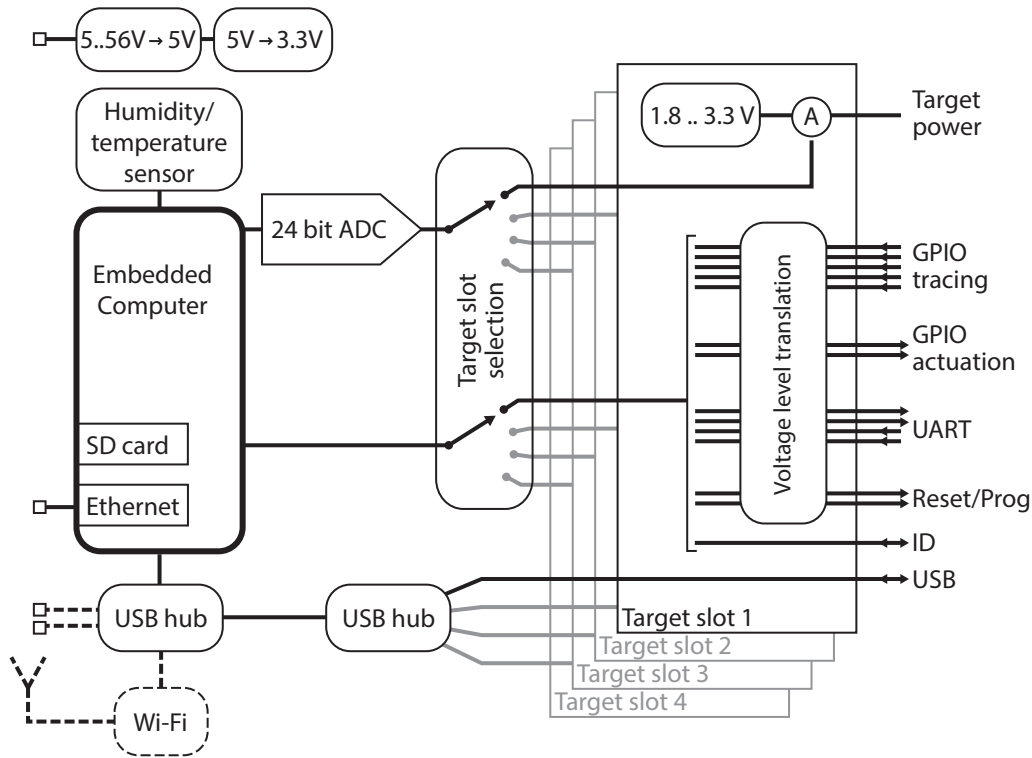


Figure 2.2: High-level schematic of the FLOCKLAB observer hardware.

- **Large data volume:** FLOCKLAB needs to cope with large data volumes that arise particularly during high-resolution power profiling. Samples should not be lost and be quickly processed to ensure complete and sound measurement data.
- **Platform support:** FLOCKLAB's hardware and software architecture must be designed in such a way that new platforms can be supported with little effort and cost.

2.2.2 Overview

FLOCKLAB consists of several distributed target-observer pairs and a set of servers. *Observers* are powerful platforms that can host up to four devices under test, the *targets*, connected through relatively simple *interface boards*. Observers implement all services available in FLOCKLAB in hardware or software. They connect to several backend servers responsible for coordinating their distributed and synchronized operation, for processing and storing collected results, and interacting with FLOCKLAB users.

2.2.3 Observer Hardware

The observer architecture, as depicted in Figure 2.2, is based on a custom-designed PCB assembly. The main processing unit is a Gumstix XL6P COM embedded computer, which is driven by a 624 MHz Marvell XScale PXA270 microprocessor and equipped with 128 MB SDRAM and 32 MB flash memory. We add an 8 GB SD card to cache test configurations, program images, and test results. Observers connect to FLOCKLAB servers preferably through the Ethernet expansion of the Gumstix; USB Wi-Fi adapters can be used if Ethernet proximity is lacking.

A switching regulator converts a 5–56 V DC input voltage to the 5 V on-board voltage required by the Gumstix. A linear regulator with low output noise further down-converts to the 3.3 V on-board voltage required by other components. The ADS1271, a 24-bit delta-sigma ADC, is used for power profiling as detailed in Section 2.2.4. Additionally, there are three USB connectors and a humidity/temperature sensor. In our deployment, described in Section 2.2.8, we use the readings of the latter to control a USB-powered fan on four outdoor observers to prevent humidity and overheating issues.

An observer provides four pin header connectors to attach targets through interface boards. The following main components are replicated for each connector: an LM3370 switching regulator to adjust the target supply voltage in the range of 1.8–3.3 V with 100 mV resolution; a MAX9923H current-sense amplifier for power measurements; five incoming and two outgoing GPIO lines to trace and actuate GPIO pins of the target; UART lines to read and inject data over the target’s serial port; lines to reset and program the target; an ID line to identify the interface board as further discussed in Section 2.2.6; and a USB port for USB-enabled targets and interface boards. Two 8-bit signal translators match the variable voltage of the target with the 3.3 V on-board voltage of the observer. Finally, an observer provides nine LEDs controlled by the GPIO and UART data lines for visual inspection.

Because of the limited number of GPIO pins on the Gumstix, we need to multiplex the available signal lines between the four targets. We achieve this by letting the Gumstix enable the two voltage level translators and the current-sense amplifier of the desired target and disable them for all other slots. The Gumstix can thus control one target at a time.

The cost of the complete observer PCB assembly amounts to a rough total of 1000 USD including manufacturing costs.

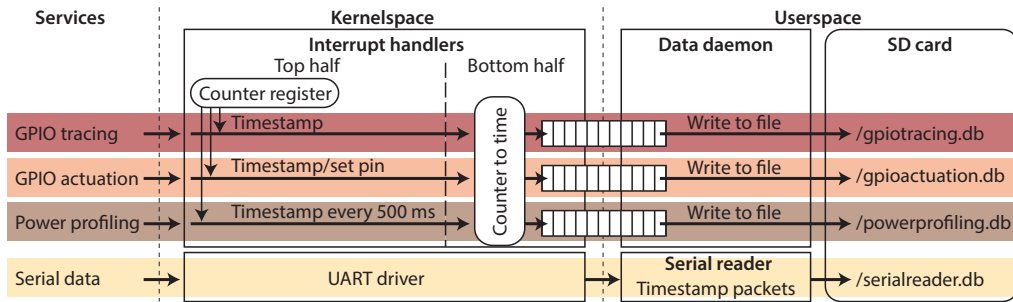


Figure 2.3: Processing of GPIO events, power samples, and serial data on an observer. *Timestamping occurs in the bottom half of an interrupt handler using a tick count taken in the top half, which increases precision and throughput.*

2.2.4 Measuring Power

To measure power, we put a small shunt resistor between the switching regulator and the target. The voltage across the resistor is proportional to the current draw of the target. We use a MAX9923H high-side current-sense amplifier to amplify the sense voltage by a gain of 100. The MAX9923H has low offset voltage and high gain accuracy, providing precise measurements also at low sense voltages. The output of the amplifier is then fed into an ADS1271 ADC, whose samples are fetched by the Gumstix over an SPI bus. Conversion into current is done in the FLOCKLAB backend based on the shunt resistance, the gain of the amplifier, and the reference voltage of the ADC.

The choice of the shunt resistor presents a tradeoff. Using a small resistor reduces the influence on the measurements, whereas using a large resistor gives a better SNR. Another important factor is the wide dynamic range of current draw. For instance, a TinyNode draws only $2\ \mu\text{A}$ in sleep mode, whereas an Opal draws as much as $49\ \text{mA}$ when both radios are turned on. To prepare FLOCKLAB for even higher current draws of future platforms, we want to support up to $160\ \text{mA}$. Based on these considerations, we decided to use a relatively small $150\ \text{m}\Omega$ shunt resistor, which still enables the high-gain amplifier to accurately measure low signal levels.

The ADC has a resolution of 24 bits, which gives a theoretical resolution of $10\ \text{nA}$ in current draw based on the specifications of shunt resistor, amplifier, and ADC. The ADC features two modes of operation that are interesting for FLOCKLAB, selectable by a jumper: high-speed and high-resolution. Using a $14.3\ \text{MHz}$ clock source, the ADC samples at $56\ \text{kHz}$ in high-speed mode and at $28\ \text{kHz}$ in high-resolution mode. FLOCKLAB defaults to the latter as it has a higher SNR of $109\ \text{dB}$, while still providing a sufficiently high sampling rate to capture short-lived radio

events.

2.2.5 Observer Software

Observers run OpenEmbedded Linux and use Chrony as an NTP [MMBK10] client to synchronize every 1–2 minute with the FLOCKLAB NTP server (see Section 2.2.7). This provides the basis for accurately timestamping GPIO events, power samples, and serial messages. Observers cache the timestamped data locally before uploading them to the FLOCKLAB database server, and have a collection of Python scripts that are used by the FLOCKLAB test management server to trigger scheduled actions such as starting and stopping a test, reprogramming a target, and setting the target supply voltage.

Data acquisition and timestamping. To gain access to hardware connected to the Gumstix—in our case the GPIO lines and the SPI bus which interfaces with the ADC—we implement data acquisition and timestamping as kernel modules. Kernel processes run with highest priority, which helps reduce processing delays and thus increase throughput.

As shown in Figure 2.3, data acquisition for GPIO tracing, GPIO actuation, and power profiling starts in interrupt handlers. Triggered by a hardware or timer interrupt, the top half of a handler serves the interrupt, reads a counter register to obtain the current time value, and requests that the bottom half of the handler be executed at some future time. The bottom half uses then the counter value to compute a precise Unix timestamp. This approach increases throughput and timestamp precision, because it minimizes the execution time of the top halves, enabling interrupt requests to be served at high rate and low jitter.

As for GPIO tracing and GPIO actuation, an observer timestamps single events. This is however different for power profiling. To reduce system load and memory consumption, we generate a timestamp only every 500 ms. Using the constant sampling rate of the ADC, the FLOCKLAB backend later interpolates the timestamps of single power samples.

Timestamping of serial messages is less critical since these are already affected by non-deterministic UART transfer delays [DEE03] and therefore should not be used to log data that require highly accurate timestamps. For this reason, we process and timestamp serial messages in userspace.

Data caching. When using FLOCKLAB’s power profiling service, the observers have to deal with enormous amounts of data, so efficient data handling is key. Motivated by this, we use a custom-built binary log file mechanism rather than a full-blown database system. As shown in

Figure 2.3, kernel FIFO queues are used for transferring acquired data from kernel to userspace, where a daemon receives the data and writes them into separate files on the SD card. Upon request from the FLOCKLAB test management server, an observer uploads accumulated data to the database server.

2.2.6 Supporting Diverse Target Platforms

FLOCKLAB possesses the flexibility to support diverse target platforms with little effort in terms of hardware and software. Every observer can host four targets of possibly different form factors, connectors, features, and tools required for installing program images. Key to this flexibility is the use of interface boards: simple PCB assemblies that interconnect the components on an observer (see Figure 2.2 and Section 2.2.3) with the corresponding components on the target.

Every platform requires its own custom-designed interface board, since there is no standardized connector or pin layout for wireless embedded devices. An interface board may also need to make provisions for different logic levels.

Additionally, FLOCKLAB imposes a few constraints on the design of an interface board. First, it needs to fit certain maximum dimensions and have an appropriate header connector. Second, the components on an interface board must work with one of the available power supplies: 3.3 V, 5.0 V, or the 1.8–3.3 V DC adjustable voltage. Third, an interface board must feature a serial ID chip that is compliant with the widely used DS2401, which is needed to automatically identify the mapping of target slots to interface boards.

Besides interface boards for TelosB, TinyNode, and Opal designed by us, external collaborators from IBM designed an interface board for IRIS, which also supports Mica2 and MicaZ due to pin-compatibility. We leverage these interface boards in our FLOCKLAB deployment at ETH Zurich to attach four different platforms to each observer, as shown in Figure 2.1.

On the software side, it is sufficient to port the reprogramming tool to the Gumstix to support a new platform. As for serial I/O, FLOCKLAB observers already support ASCII data, TOS messages, and SLIP datagrams. The target software requires no special measures, since embedded operating systems already provide functions for serial I/O and accessing GPIO pins, and power is measured by the observer.

2.2.7 Backend Infrastructure

Observers connect via Ethernet or Wi-Fi to a set of servers that provide all what it takes to make FLOCKLAB a testbed.

Time synchronization server. FLOCKLAB operates its own NTP server that synchronizes against another server on campus and a high-accuracy pulse per second (PPS) signal output by a GPS receiver, which provides a precise time reference. All observers synchronize against this NTP server.

Web server. Users interact with this server to schedule and configure their tests. Every user is allowed to reserve FLOCKLAB for a certain maximum duration and number of tests at a time. A test configuration consists of a single XML file to setup the services and one or more compiled binaries. A user can run a test as soon as possible or during some specified time slot, abort a running test, and fetch the results of successfully completed tests. If requested, a user receives email notifications about started and completed tests.

Test management server. This server is responsible for operations related to starting, running, and finalizing scheduled tests. If a test is about to start, it parses the configuration, prepares programmable images from the supplied binaries, and dispatches these data to the observers. While a test is running, it periodically queries the observers for results and stores them in a database. When a test has finished, it processes the raw data (e.g., interpolate timestamps, convert to current) and stores them in a compressed archive.

Database server. This server hosts a MySQL database, which stores test configurations, and user-specific data such as quotas and login information.

Monitoring server. Finally, we use Zabbix and Cacti to constantly monitor all server instances, networking components, and observers. In case of an abnormal situation, FLOCKLAB administrators are automatically informed via e-mail and/or SMS to ensure maximum uptime of the testbed.

2.2.8 Deployment

The FLOCKLAB deployment at ETH Zurich consists of 30 observers, each hosting a TelosB, IRIS, Opal, and TinyNode 184. As illustrated in Figure 2.4, 26 observers are deployed indoors across one floor in an office building, distributed in offices, hallways, and storerooms. Four observers are deployed outside, sitting on the roof of an adjacent building a few meters beneath the floor with the indoor observers.

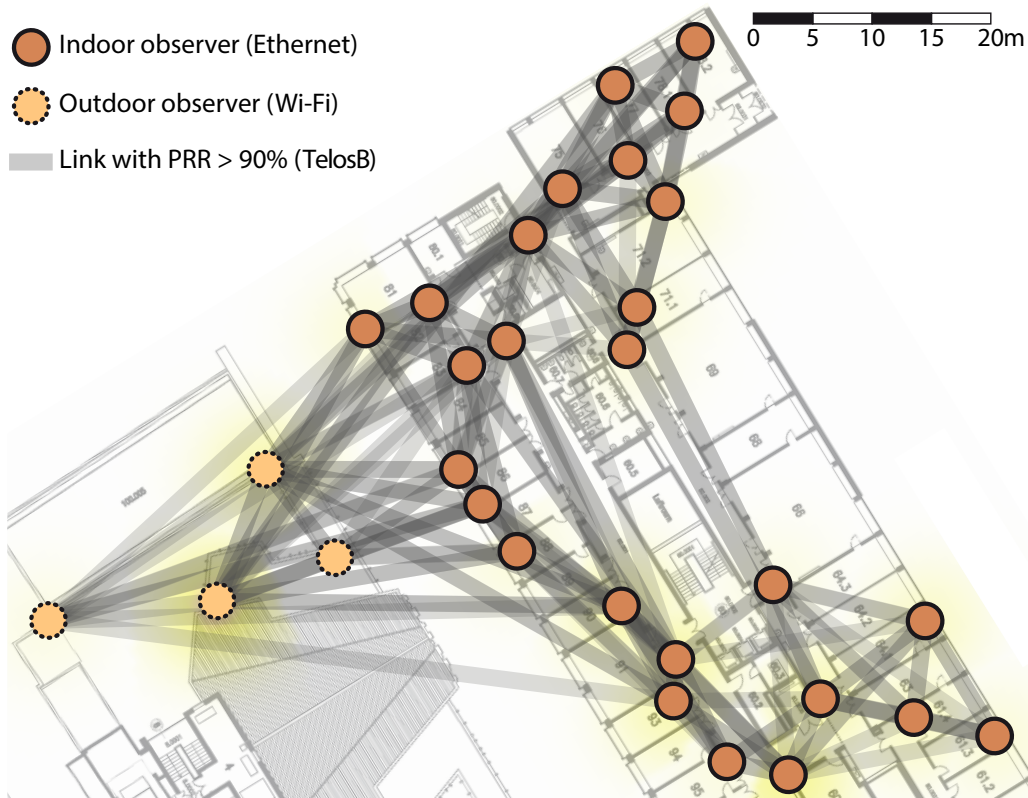


Figure 2.4: Layout of FLOCKLAB deployment including information about link qualities and noise.

All indoor observers connect via Ethernet, and have a light acrylic glass cover to protect against dust. To help the accuracy of NTP by reducing communication latency and jitter, they are all in the same LAN segment. The outdoor observers use Wi-Fi due to lack of Ethernet on the roof, and are housed in robust polycarbonate boxes with controlled ventilation to avert humidity and overheating problems.

During testbed idle times, the test management server runs an RSSI scanner on all target platforms, determining the noise level on all channels and frequency bands, and a test where targets broadcast 500 30-byte packets each and then report the number of packets they received from any other target, which gives an estimate of the link qualities in the testbed. This information is stored in the database and displayed on the FLOCKLAB website as an overlay on the deployment map as shown in Figure 2.4, giving users an idea as to what extent their tests may be affected by external interference (e.g., from co-located Wi-Fi) or limited connectivity.

2.3 Benchmarking FlockLab

Using our deployment, we benchmark in this section the accuracy and the limits of key FLOCKLAB services. We start by evaluating FLOCKLAB’s timing accuracy, which is fundamental to exploit the full potential of the GPIO and power profiling services, check the stability of the power supply and the accuracy of the power measurements, and finally determine the maximum rate for capturing GPIO events.

2.3.1 Timing Accuracy

2.3.1.1 GPIO Tracing and Actuation

Setup. We randomly select 7 Ethernet-connected observers, and put one Wi-Fi-connected observer indoors on a table. We evaluate GPIO tracing and actuation in two separate 1 hours tests. In the first test, we use a signal cable to connect a GPS clock to one GPIO pin of each observer. The GPS clock generates a PPS signal, and the observers timestamp the corresponding GPIO events. In the second test, we connect one GPIO pin of each observer to a Tektronix MSO4054B mixed-signal oscilloscope. All observers simultaneously toggle the pins every second, and the oscilloscope measures the actual timing of these events.

Pairwise timing error. We first measure the pairwise timing error between simultaneous GPIO events at different observers. This evaluates the alignment of GPIO traces collected by different observers and, for GPIO actuation, the precision with which simultaneous actions can be triggered.

Table 2.2 shows that the average pairwise error is smaller than $40 \mu\text{s}$ when using the 7 Ethernet-connected observers. If we add the Wi-Fi-connected observer, the error increases significantly due to higher and more variable delays in the exchange of NTP packets over Wi-Fi. The error is similar for GPIO tracing and actuation, as an observer executes similar operations when timestamping an event or setting a pin.

These results show that FLOCKLAB allows users to align GPIO traces and to set GPIO pins with an error as small as a few tens of microseconds when using the indoor observers. This high accuracy is more than sufficient to trace packet transmissions among targets, as we demonstrate in Section 2.4.5. The results also show that because of the higher NTP synchronization error over Wi-Fi, the outdoor observers are less suited for tests that require sub-millisecond timing accuracy.

Error on time intervals. Using data from the previous experiment, we also assess the error on time intervals. We compute for each observer the difference between timestamps of consecutive GPIO events and compare

GPIO service	7 Ethernet			7 Ethernet, 1 Wi-Fi		
	avg	85th	max	avg	85th	max
Tracing	36 μ s	69 μ s	255 μ s	166 μ s	527 μ s	1,161 μ s
Actuation	30 μ s	54 μ s	394 μ s	138 μ s	334 μ s	1,170 μ s

Table 2.2: Pairwise timing error of GPIO services. *The average error is smaller than 40 μ s with Ethernet observers.*

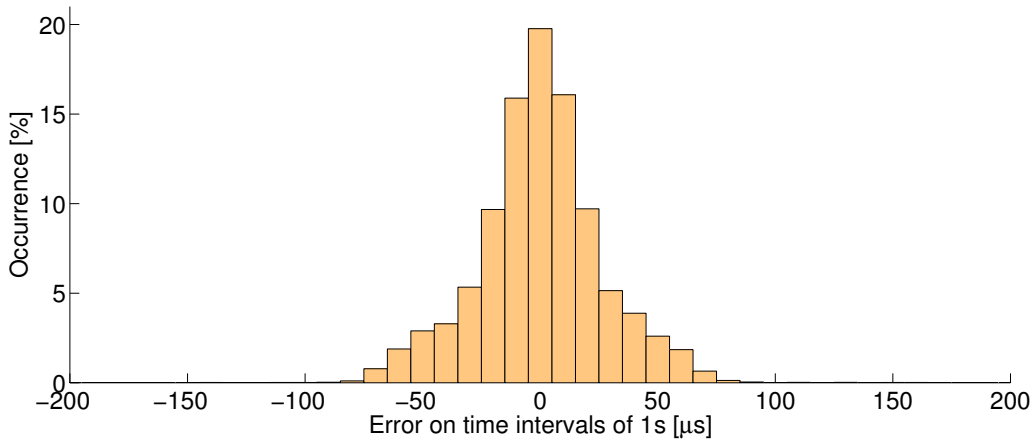


Figure 2.5: Distribution of the error on time intervals between GPIO events. *The average error is -0.011μ s.*

it to the PPS signal. In this way, we evaluate the precision with which an observer measures the interval between GPIO events.

Figure 2.5 shows the distribution of the error on time intervals, as measured by all 8 observers used in the experiment. We see that it approaches a normal distribution with a sample mean of -0.011μ s and a sample standard deviation of 27μ s. The average error is small because each timestamp is similarly affected by variable interrupt delays on an observer. We show in Section 2.4 that this precision allows to profile the radio activity or to measure the clock drift of a target.

2.3.1.2 Power Profiling

Setup. To evaluate the timing accuracy of the power profiling service, we run a 2 minute test on 6 TelosB targets attached to Ethernet-connected observers. One *transmitter* generates a 30-byte packet every 62.5 ms. The other 5 *receivers*, located in the transmission range of the transmitter, have their radios turned on and receive the packets. The corresponding

observers enable GPIO tracing and power profiling, measuring current¹ draws at 28 kHz.

When a start frame delimiter (SFD) interrupt signals the start of a packet reception, a receiver toggles a GPIO pin and turns on its three on-board LEDs. As shown in Figure 2.6(a), these operations generate a GPIO event and an increase in current draw from 22 mA to 34 mA. When the next SFD interrupt signals the end of a reception, each receiver turns off its LEDs and the current decreases accordingly. We consider these events as occurring at the same time, as we measure with an oscilloscope that the lag due to different time of flight and interrupt delays is smaller than 1 μ s. To compare power timestamps, we define that a *power event* occurs when the current rises above a *leds-on threshold* of 23 mA.

Timing error between GPIO and power events. We measure the timing error between GPIO and power events on the *same* observer by computing the interval between the GPIO and the respective power timestamp (i.e., between a vertical line and the corresponding circle in Figure 2.6(a)).

The solid line in Figure 2.6(b) shows the cumulative distribution of the timing error, which is 20 μ s on average and smaller than 29 μ s in 85 % of the cases. We see that the average error is close to half the power sampling period (17 μ s): power profiling has a lower resolution than GPIO tracing and most of the timing error comes from the random delay between a GPIO event and the following power sample.

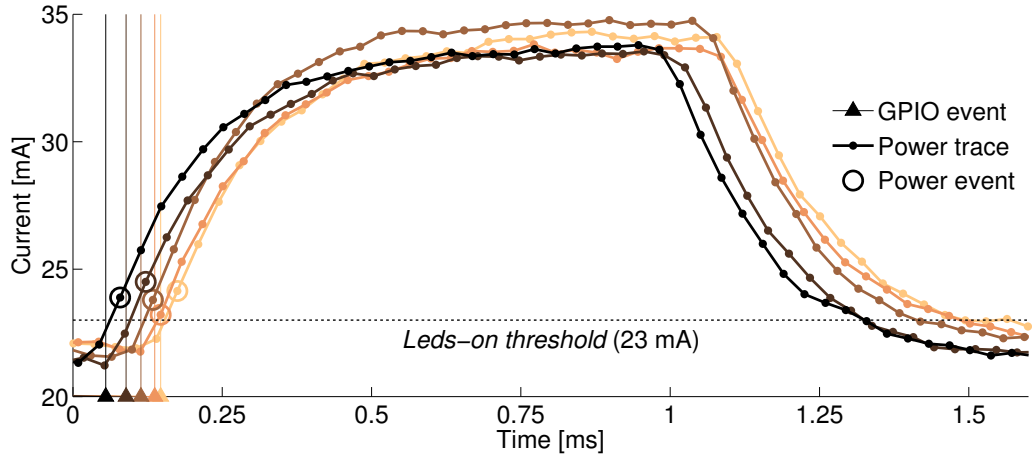
Pairwise timing error. We now look at the pairwise timing error between simultaneous power events on *different* observers (i.e., between two circles in Figure 2.6(a)).

The dashed line in Figure 2.6(b) shows the cumulative distribution of this pairwise timing error, averaging around 39 μ s with an 85th percentile below 68 μ s. The error is comparable to that of simultaneous GPIO events in Section 2.3.1.1, since the sources of time inaccuracies are similar. Figure 2.6(a) and the test case in Section 2.4.5 confirm that the precise alignment of power traces in FLOCKLAB allows to match the power draw of a target to packets exchanged with other targets.

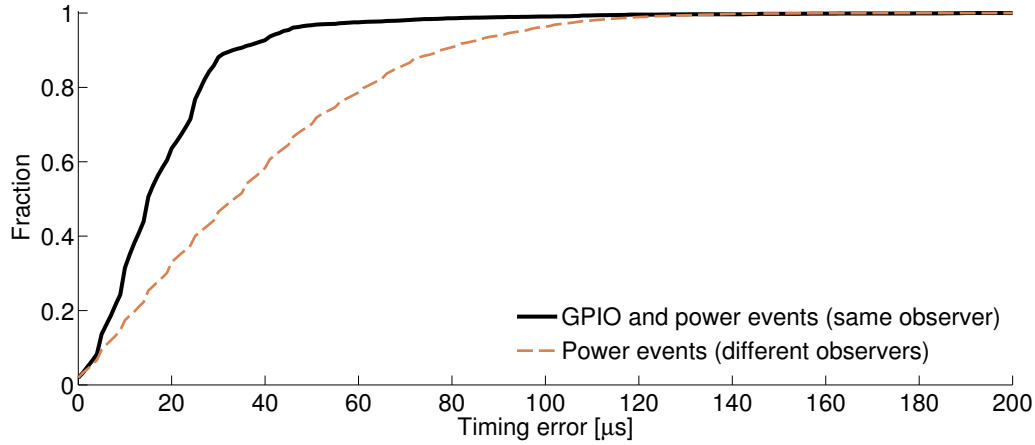
2.3.2 Power Accuracy

We use ad-hoc experiments to check whether an observer accurately measures the current draw of the target with only minimal impact on the stability of the target supply voltage.

¹We use power and current interchangeably in Secs. 2.3 and 2.4, because FLOCKLAB supplies a known, stable voltage (see Section 2.3.2) and thus power is directly proportional to current.



(a) Simultaneous GPIO and power events on 5 observers.

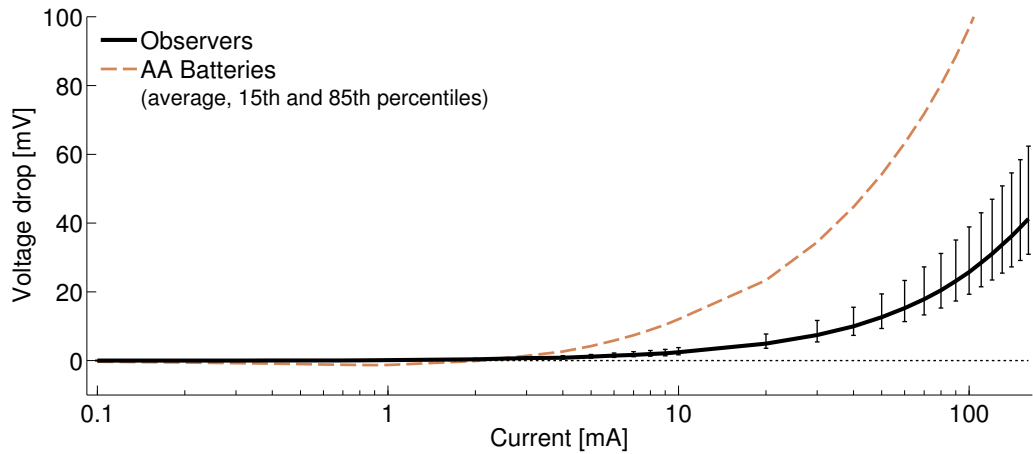


(b) Cumulative distribution of timing errors.

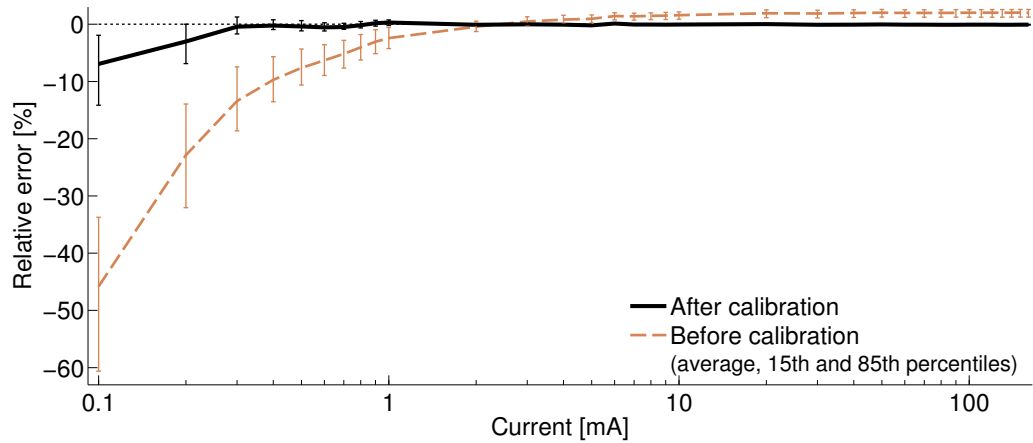
Figure 2.6: Timing errors of power profiling. *Observers timestamp simultaneous power events with an average pairwise timing error of 39 μs .*

Setup. We connect the target slot of an observer to a high-precision Agilent N6705A power analyzer, which acts as a target that draws predefined currents. The current draws cover the full dynamic range and proceed in a step-wise fashion as follows: from 0 mA to 1 mA in steps of 0.1 mA, from 1 mA to 10 mA in steps of 1 mA, and from 10 mA to 160 mA in steps of 10 mA; each of the 35 steps lasts 3 s. During the experiment, the observer supplies a nominal voltage of 3.3 V and records the current drawn by the power analyzer with a resolution of 14 kHz, while the power analyzer records the voltage supplied by the observer with a resolution of 24 kHz. We repeat the experiment 32 times, using the four target slots of eight randomly chosen observers.

Stability of power supply. We first look at the stability of the supply voltage. To this end, we measure the voltage drop as the difference



(a) The average voltage drop is small for typical target currents and less than 42 mV even when a target draws 160 mA.



(b) Calibration using linear regression reduces the average relative error on current draw to -0.39%.

Figure 2.7: Stability of the power supply and accuracy of the power measurements in FLOCKLAB.

between the zero-load voltage (i.e., when no current is drawn by the power analyzer) and the voltage supplied at a certain current draw. The solid line in Figure 2.7(a) shows that the average voltage drop is at most a few mV for typical current draws of our targets; for example, an Opal draws 49 mA when fully active, yielding an average voltage drop of 13 mV. The voltage drop of the other target platforms is even smaller, because they draw less current.

To put these numbers into perspective, we measure the voltage drop of two AA alkaline batteries, a typical power supply in real deployments. The dashed line in Figure 2.7(a) shows that their average voltage drop is higher than that of an observer, and is above 23 mV already at a current draw of 20 mA. We compute a linear fit between voltages and currents and find that a target sees an average resistance of 259 m Ω when connected to an observer, which is almost four times smaller than what a target would see with AA batteries (947 m Ω). The results show that power profiling with FLOCKLAB minimally affects the target supply voltage.

Accuracy of power measurements. Next, we evaluate the accuracy of power measurements by computing the relative error between the current draw measured by the observers and the current drawn by the power analyzer.

The dashed line in Figure 2.7(b) shows that FLOCKLAB underestimates at currents below 2 mA and slightly overestimates at higher currents. The relative error is particularly significant for low currents: static offset errors of the current-sense amplifier, manufacturing errors of the shunt resistor, and inaccuracies of the amplifier gain introduce a constant offset and a constant multiplication factor into the measurements. Motivated by this observation, we use linear regression to estimate these constants by comparing the measurements from the observers with those from the power analyzer, effectively calibrating FLOCKLAB's power profiling service.

For each observer and target slot, we repeat the experiment and correct the measured current draw by applying our calibration based on the constants computed from the previous experiment. The solid line in Figure 2.7(b) shows that the calibration reduces the relative error on current draw significantly, especially for currents below 1 mA. For currents between 0.1 mA and 160 mA, the accuracy of the power measurements increases by a factor of 6 after calibration.

Based on calibration parameters we computed for all target slots on all 30 observers, the FLOCKLAB test management server corrects the power measurements before delivering them to the user. We show in Section 2.4.3 that this results in accurate power measurements allowing to precisely measure the energy consumed by a target throughout a test.

Number of GPIO pins	Power profiling	Captured GPIO events	
		99 %	100 %
1	no	80 μ s	290 μ s
	yes	90 μ s	280 μ s
5 interleaved	no	20 μ s	80 μ s
	yes	30 μ s	90 μ s

Table 2.3: Minimum required interval between consecutive GPIO events to capture 99% and 100% of generated events. *An observer captures 99 % of events on one GPIO pin if they are at least 90 μ s apart.*

2.3.3 Limits in Capturing GPIO Events

The sampling rate of the ADC defines the interval between power samples. This is different for tracing GPIO events on an observer: the minimum required interval to reliably capture consecutive events depends on the interrupt delay and the execution time of the top half of the interrupt handler. We run experiments to determine this minimum interval.

Setup. We use all 30 TelosB targets and let them toggle GPIO pins with an increasing interval. Starting from 10 μ s, targets increase the interval in steps of 10 μ s up to 1 ms, and generate at each setting 100 GPIO events. We run four tests, each repeated ten times: two where they toggle a single pin and two where they toggle five pins interleaved. In both cases, we run one test with and one test without power profiling.

Minimum interval between GPIO events. For each interval, we compare the number of captured events with the number of generated events. Table 2.3 lists the minimum required interval to capture 99 % and 100 % of events. First, we see that FLOCKLAB captures events more reliably when they are interleaved on 5 GPIO pins. This is because every GPIO pin is mapped to a specific interrupt flag in the observer’s processor, and no new events can be captured until the respective flag is cleared.

We further observe that the minimum required interval to capture GPIO events increases by 10 μ s when the power profiling service is enabled. This service increases the load on an observer, leading to higher interrupt delays and thus to a lower probability that events are successfully captured. Finally, we note the significant difference between the minimum required intervals for capturing 99 % or 100 % of events, since sporadic activity on the observers (e.g., exchanging and processing NTP packets or storing measurement data into a file) may sometimes increase the interrupt delay, too.

The following section shows that FLOCKLAB’s GPIO tracing service

allows to accurately record MCU and radio activity, measure end-to-end packet delays, monitor the exchange of packets, and measure the clock drift of a target.

2.4 FlockLab in Action

After presenting the architecture of FLOCKLAB and evaluating its performance, we now demonstrate the utility of FLOCKLAB for testing, debugging, and evaluating wireless embedded systems through several real-world test cases.

2.4.1 Comparative Multi-Platform Analysis

One feature that sets FLOCKLAB apart from other testbeds is the possibility to test multiple platforms on the same physical topology. Comparative analyses of this type can provide valuable feedback, for example, to developers of communication protocols, because the characteristics of the underlying platform may affect the performance of these protocols considerably and in non-trivial ways.

In this test case we perform a comparative multi-platform analysis of the standard TinyOS data collection application. The application uses CTP [GFJ⁺09] on top of the LPL [PHC04] link layer to collect data from a set of nodes at a single sink. We use all 30 observers and all four targets available in FLOCKLAB: Opal, TinyNode, TelosB, and IRIS. The radios of these platforms differ in terms of frequency band, maximum transmit power, modulation scheme, and data rate. For each individual platform we let the nodes transmit at the highest power setting, and all nodes but the sink generate a packet every 5 s for a duration of 35 minutes.

We are interested in how each platform affects the trade-offs between energy consumption, data yield, and end-to-end latency. Since these key performance metrics are known to be influenced by the operational parameters of the link-layer protocol [ZFM⁺12], we further test for each platform six different LPL wake-up intervals: 20, 50, 100, 200, 500, and 1,000 ms. We thus expect to gain insights into the platform-dependent sensitivity of the system performance to changes in the LPL wake-up interval, too.

Without FLOCKLAB. Despite the lack of multiple platforms on other testbeds, it is not trivial to obtain reliable and unobtrusive measurements of the performance metrics we are interested in. Data yield can be measured quite straightforwardly based on the sequence numbers of received packets, but measuring energy and latency is more difficult.

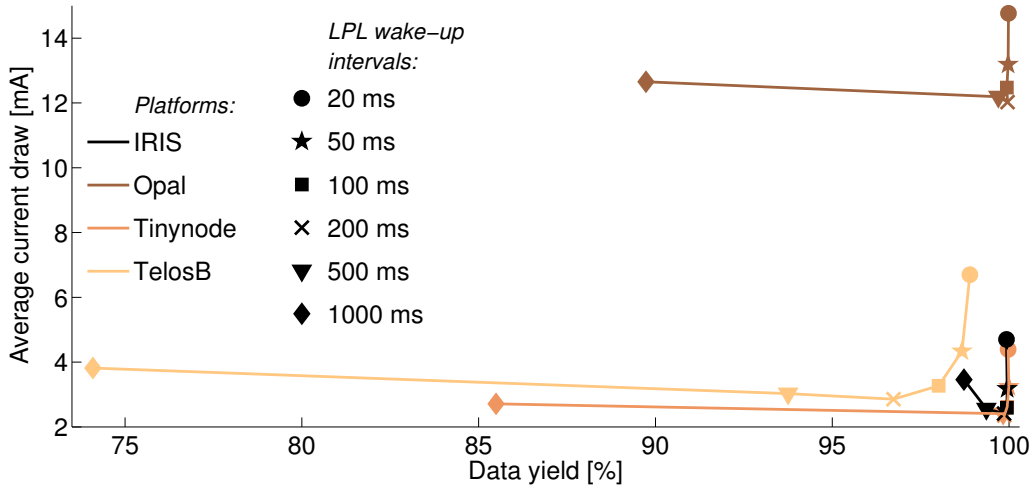
On testbeds that do not support power profiling, energy consumption must be estimated in software. For example, Energest [DOTH07] provides accurate energy estimations in Contiki but is also intrusive (see Section 2.4.3). Other operating systems like TinyOS lack a standard energy estimator. This increases the overhead to obtain energy estimates in the first place, may lead to incomparable results from different custom-built estimators, and generally encourages the use of radio duty cycle as a proxy for energy consumption, which may not be meaningful toward the total node energy budget.

One approach to measure the end-to-end latency is to log a message over the serial port when generating a packet at the source and another message when receiving a packet at the sink. However, serial logging alters the timing behavior of the application, and the resulting timestamps are inaccurate due to non-deterministic UART delays [DEE03]. Another approach is to run a dedicated time synchronization protocol such as FTSP [MKSL04] concurrently to the protocol under test and to timestamp packets at the source. But, as shown in [CKJL09], running multiple network protocols concurrently entails the risk of unanticipated interactions between protocols that can lead to performance losses or even failures. Furthermore, for some combinations of platforms and operating systems there may not be a synchronization protocol readily available.

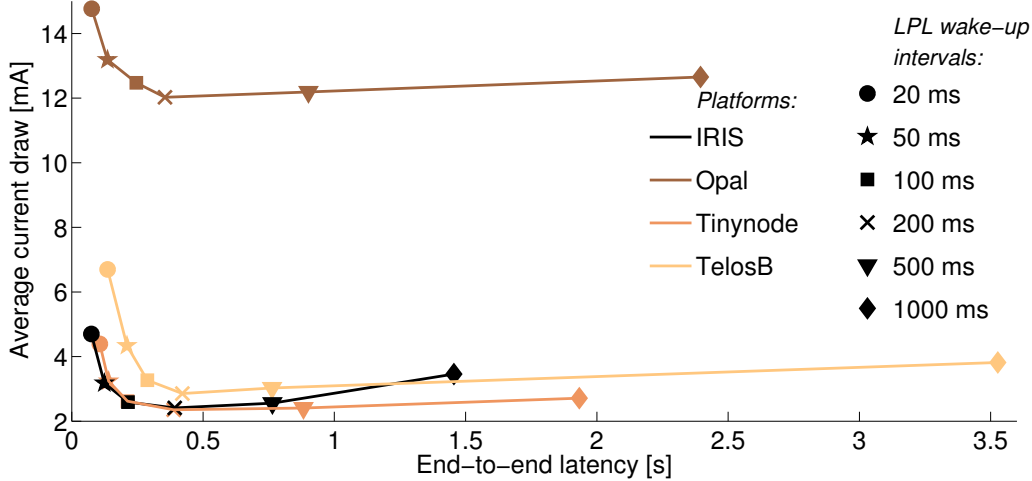
With FLOCKLAB. The power profiling service in FLOCKLAB provides non-intrusive current measurements for computing the energy consumption. The GPIO tracing service can be used to measure the end-to-end latency: the application toggles a GPIO pin when a source generates a packet and another GPIO pin when the sink receives a packet. Taking the interval between both events of the same packet, we obtain non-intrusive measurements of the end-to-end latency of received packets. The effort is limited to inserting two GPIO tracing statements in the application code and configuring the FLOCKLAB services in an XML file.

Figure 2.8(a) shows data yield and Figure 2.8(b) shows end-to-end latency against average current draw², for all platforms and LPL wake-up intervals. As expected, higher data yield and lower end-to-end latency can generally be achieved at the expense of higher average current draw. While this holds for all platforms, data yield and end-to-end latency are better with IRIS, Opal, and TinyNode than with TelosB, since the higher transmit power of the former platforms leads to shorter routing paths with CTP. Interestingly, IRIS is least sensitive to changes in the LPL wake-up interval, and all four platforms draw minimum current at 200 ms LPL wake-up interval, which is thus the most energy-efficient parameter

²The high current draw with Opal is due to a software issue that prevents the nodes from entering a low-power mode.



(a) Average current draw against data yield.



(b) Average current draw against end-to-end latency.

Figure 2.8: FLOCKLAB enables comparative performance analyses of the same application on multiple platforms. The plots show performance results from CTP running atop LPL for different LPL wake-up intervals and platforms, including IRIS (2.4 GHz, 3 dBm), Opal (868 MHz, 6 dBm), TinyNode (868 MHz, 12.5 dBm), and TelosB (2.4 GHz, 0 dBm).

setting for this particular topology and traffic load.

2.4.2 Finding and Fixing Bugs

GPIO tracing is also a very powerful debugging tool. We already found and fixed several bugs this way, and as a concrete example we describe next how we found and fixed a protocol misconfiguration that caused a poor performance during initial experiments of the previous test case.

Finding the bug. With LPL wake-up intervals of 500 ms and 1 s, the

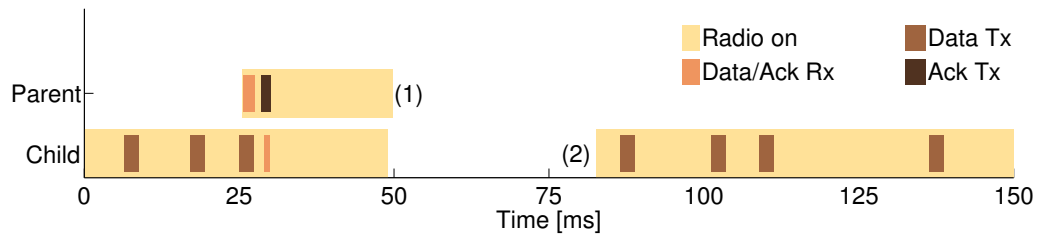


Figure 2.9: GPIO trace showing a misconfiguration of CTP and LPL on TinyNodes. After receiving a packet, the parent turns off the radio (1) before the child sends the next packet (2), causing packet loss due to queue overflows.

initial results from TinyNode and Opal nodes were significantly worse than expected in terms of data yield and end-to-end latency. All sources were seemingly affected, and we could not pinpoint specific nodes to debug with a logic analyzer. We thus instrumented the radio stacks to set different GPIO pins according to the current radio state (i.e., sleeping, active, receiving, or transmitting) and repeated the experiments with GPIO tracing enabled. Using `printfs` instead of GPIO, we would have run the risk of breaking the timing-sensitive operation of the radio driver and LPL.

After aligning the GPIO traces of all nodes, we noticed that nodes located farther away from the sink could communicate properly. The bug indeed affected mostly nodes close to the sink, which delivered only a small fraction of the many packets they had to forward. We decided to focus on these nodes and, by looking deeper into the transfers between a child and its parent, we found that children were transmitting at most one packet during an LPL wake-up interval, although they had multiple packets ready to be sent.

Fixing the bug. With the help of GPIO traces, we were also able to find and fix the cause of this bug. Figure 2.9 shows an example of the problem, based on GPIO traces collected from two TinyNodes. After a successful packet reception, the parent kept the radio on for a short time but went to sleep (1) before the child could transmit the next packet (2). As a result, children had to wait until the next regular wake-up of their parents before they could send the next packet, which caused severe data loss at long wake-up intervals.

This prompted us to check the configurations of CTP and LPL. Based on our settings, a TinyNode or Opal node kept the radio on for 20 ms after a reception, but its children transmitted additional packets only after 32 ms (the default CTP setting for generic platforms). We fixed this misconfiguration by changing the value of these parameters based on the

radios' data rate and experimental results. For example, on TinyNodes a parent keeps the radio on for 36 ms after a reception, and a child transmits the next packet after 10 ms.

2.4.3 Controlling and Profiling Applications

When evaluating applications like data collection it is often desirable not only to precisely measure performance figures but also to control nodes during an experiment, for example, to specify which nodes generate packets and when [ZFM⁺12], or to emulate failures by turning some nodes off during a certain interval [GF⁺09]. We now show that FLOCKLAB greatly helps control and profile typical data collection applications.

In this test case, we run the default data collection application of Contiki on TelosB targets, that is, Collect on top of ContikiMAC. The wake-up interval of the latter is 128 ms. We want one node to generate a packet every 2 s for 260 s, from $t = 30$ s to $t = 290$ s. We also want to measure the energy consumed by that node during these 260 s.

Controlling without FLOCKLAB. A common approach to control an experiment is to add some logic that, for example, starts and stops the generation of packets depending on the current time and the identifier of the node. This approach requires to recompile the application program for tests that need different parameterization, which is time-consuming. Most importantly, some form of in-band time synchronization is also needed if several nodes are to simultaneously start and stop generating packets, which can, however, degrade the performance of the application under test [CKJL09].

Controlling with FLOCKLAB. With GPIO actuation we can control the targets without employing an additional time synchronization within the application. In our test case, the observer connected to the node of interest sets a GPIO pin at $t = 30$ s and clears it at $t = 290$ s: the target starts and stops generating packets accordingly. Because the observers are time-synchronized, it is also possible to let multiple targets start and stop generating packets simultaneously. Moreover, we can test different generation patterns by simply modifying the GPIO actuation timings in the test configuration.

Profiling without FLOCKLAB. On testbeds without power profiling, the energy consumption of a node can be estimated in software. For example, Energest measures the time spent by the node in different states [DOTH07], which can be combined with the current draws in each state to estimate energy. This method is however intrusive, since it requires nodes to start and stop counters whenever they change state,

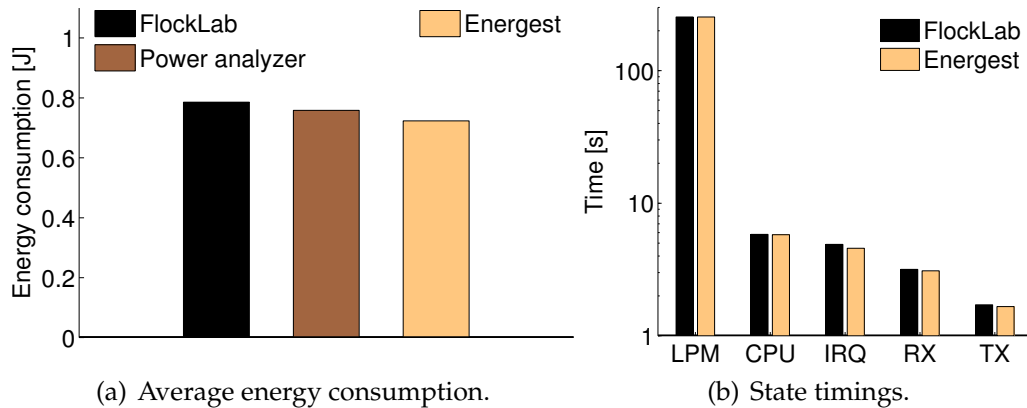


Figure 2.10: FLOCKLAB can be leveraged to obtain non-intrusive and highly accurate energy measurements.

and requires changes to existing code to be used on a different platform.

Profiling with FLOCKLAB. Power profiling allows to measure the energy consumption of any supported target in a completely non-intrusive fashion. GPIO tracing allows also to profile a target’s operation and measure state timings by setting GPIO pins according to the MCU and radio states.

In our test case, we enable power profiling between $t = 30$ s and $t = 290$ s. Figure 2.10(a) shows the energy consumption measured by FLOCKLAB, averaged over eight test repetitions, and compares it with measurements from a power analyzer attached to the target and with software-based estimations using Energest. The non-intrusive FLOCKLAB measurements are slightly more accurate than the estimations provided by Energest: the former measures an average energy consumption that is 3.6 % higher than the one measured with the power analyzer, while the latter underestimates it by 4.6 %. We also see from Figure 2.10(b) that the state timings measured with GPIO tracing correspond on average within 2.7 % to those reported by Energest. FLOCKLAB is however less intrusive than Energest; for example, we measure on a TelosB that Energest requires 11 and 21 MCU cycles to start and stop a counter, whereas only 5 cycles are required to set the level of a GPIO pin on a TelosB.

2.4.4 Measuring Clock Drift

When evaluating communication and time synchronization protocols, it is often desirable to measure how much the clock of a target drifts from the nominal frequency during a test. We now demonstrate that FLOCKLAB allows to run tests where targets experience different clock drifts (e.g., by using targets located outdoors) and to measure the actual drift during a

test accurately and minimally intrusive.

In this test case, we want to measure the clock drift of 30 TelosB targets during a 24 hours experiment. In particular, we are interested in comparing the drift of indoor and outdoor targets, and in relating the drift to the temperature measured by the targets' on-board sensors during the test.

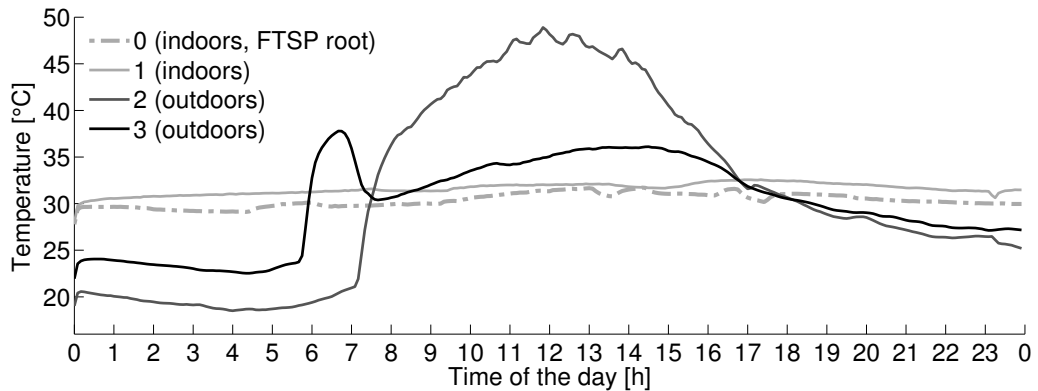
Without FLOCKLAB. A possible method to measure the clock drift is to employ FTSP, the default time synchronization protocol in TinyOS, as it periodically estimates how much the clock of a node drifts compared to the clock of a root [MKSL04]. As previously discussed, time synchronization protocols are however intrusive and may affect the behavior and the performance of the application under test.

With FLOCKLAB. With GPIO tracing we can measure the clock drift of a target in a simpler and less intrusive way, without the need of running a synchronization protocol on the target. In our test case, we instrument the application to toggle a GPIO pin every 0.5 s, and measure the clock drift by comparing the difference between consecutive GPIO timestamps with the nominal value of 0.5 s. We then average these drift values over intervals of 5 minute to limit the GPIO timing errors discussed in Section 2.3.1.1. To evaluate the accuracy of our measurements, we enable FTSP with a resynchronization interval of 3 s and use an indoor target as the root.

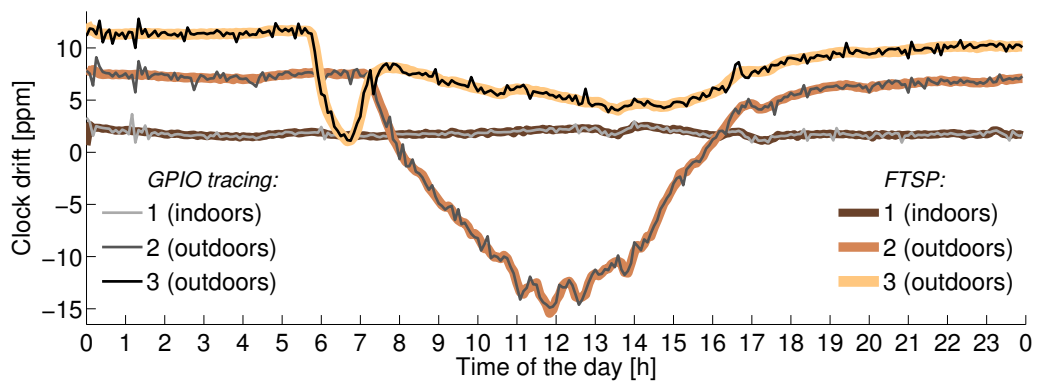
Figure 2.11(a) shows the temperature measured during the 24 hours by the FTSP root and three other targets, one located indoors and two outdoors. We notice that during daytime the outdoor targets experience significant (but different) temperature variations, while the indoor targets measure fairly constant temperatures. Figure 2.11(b) shows how the clocks of the three targets drift compared to the clock of the FTSP root, measured with GPIO tracing and by FTSP. As expected, we see that variations in temperature translate into variations in the targets' clock speed and thus into varying drift. We also notice that the drift measured with GPIO tracing corresponds to that estimated by FTSP: their difference is hardly noticeable in Figure 2.11(b) and averages 0.003 ppm. An observer performs such accurate drift measurements despite temperature variations affect also its clock speed, because it resynchronizes at least every 2 minute with the FLOCKLAB NTP server.

2.4.5 Multi-Modal Monitoring at Network Scale

The possibility of monitoring the activity of multiple targets while simultaneously measuring their current draws is invaluable for developers of low-power wireless applications. This allows, for example, to trace the exchange of packets among targets, to analyze in which states targets



(a) Temperature measured by four targets.



(b) Clock drift of three targets compared to the FTSP root, measured with GPIO tracing and by FTSP.

Figure 2.11: With FLOCKLAB it is possible to accurately measure clock drift on multiple targets during an experiment with minimal intrusiveness.

consume most energy, or to detect possible misbehaviors that may cause targets to reach undesired states or to draw more current than expected. Unlike previous testbeds, FLOCKLAB offers this possibility, and with a minimal effort from a user.

As a test case we use the Glossy flooding protocol, which lets an initiator flood a packet to all receivers within a few milliseconds [FZTS11]. We set the transmit power of 26 TelosB targets to -10 dBm and let Glossy flood a 30-byte packet every 24 ms, using different initiators in consecutive floods.

Without FLOCKLAB. With previous testbeds, the only possibility to monitor state transitions or packet exchanges is to instrument an application to store timestamps (e.g., into external flash memory) whenever an event of interest occurs. Nodes print these timestamps at the end of a test, and the testbed collects them from the serial ports. As mentioned before, this approach is very intrusive and provides

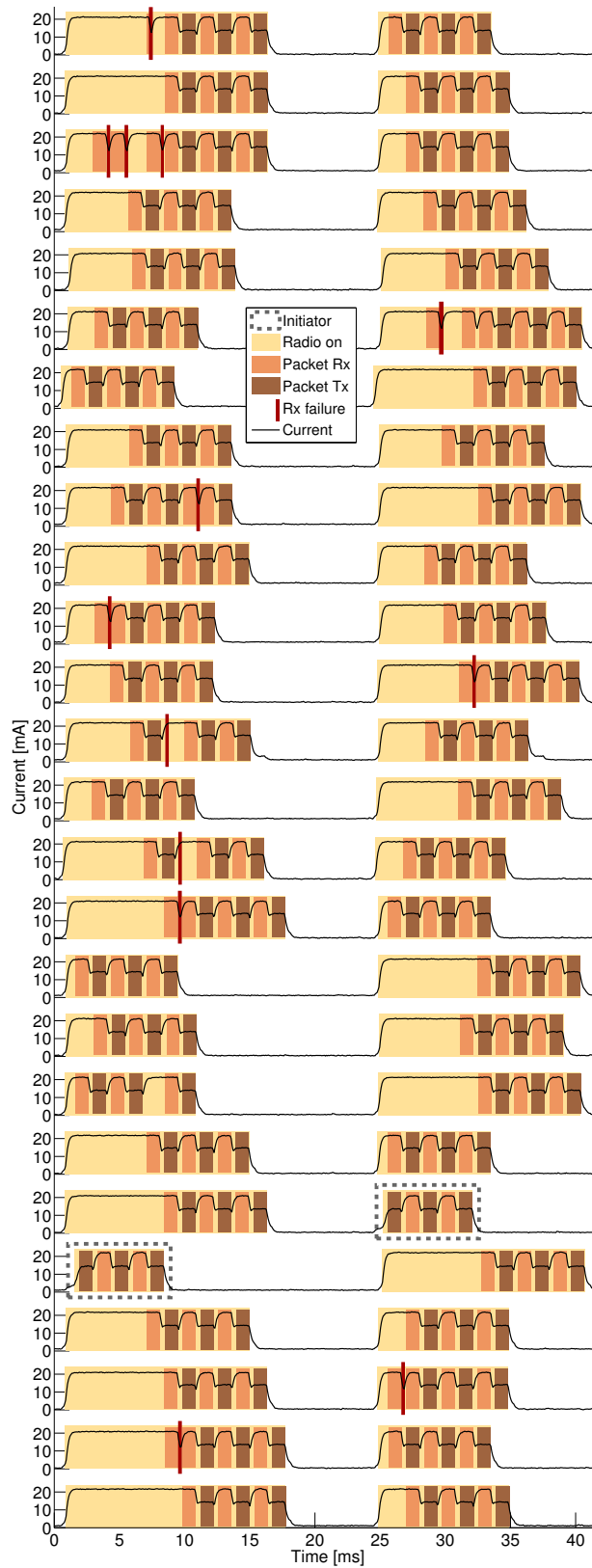


Figure 2.12: GPIO tracing and power profiling allow for monitoring the activity of multiple targets while simultaneously measuring their current draw.

meaningful results only if the nodes employ some form of in-band time synchronization. Alternatively, network simulators like Cooja [ODE⁺06] can be used to visualize the exchange of packets, but their channel and hardware models may not accurately reproduce what happens during an experiment on real devices. With either approach, however, no information about the instantaneous current draws of the nodes is available.

With FLOCKLAB. GPIO tracing allows to monitor the radio states of the targets and the exchange of packets among them with minimal intrusiveness. In our test case, we simply instrument Glossy to toggle four GPIO pins whenever the radio state changes: when it is turned on or off, when it starts or stops receiving or transmitting a packet, and when a packet reception fails (e.g., because of packet corruption). Together with GPIO tracing, we enable also power profiling to measure the current draw of the 26 targets.

Figure 2.12 shows a graphical representation of the radio states and the current draws of the 26 targets during two consecutive floods with different initiators, based on a short excerpt of the GPIO and power traces collected with FLOCKLAB. The timing accuracy of FLOCKLAB allows to precisely monitor how packets propagate in the network based on the reported radio states. For example, it is clearly visible that multiple targets transmit packets simultaneously, which is a peculiarity of Glossy. It is also possible to analyze the sets of targets transmitting or receiving at a certain time instant, and study how they are related to the network topology.

This type of visualization resembles that of the Cooja simulator, but with FLOCKLAB it is based on information collected during experiments on real devices and over real wireless links. FLOCKLAB provides also the current draws of the targets during the experiment. It is thus possible to correlate logical states and power samples and, for example, to measure the energy cost of different states. As expected, Figure 2.12 shows that targets draw most current when the radio is turned on, and in particular when they are receiving or waiting for a packet; transmissions are indeed cheaper due to the low transmit power used in the experiment.

To the best of our knowledge, FLOCKLAB is the first testbed that, among other features, provides the functionality of multiple logic analyzers and power analyzers—distributed and synchronized across the entire testbed. We maintain that developers of distributed applications and low-power wireless protocols can significantly benefit from such augmented debugging and testing capabilities.

Testbed	#Nodes	Supported Platforms	Sampling Rate	Resolution	Synchronization Error
PowerBench	28	TNode	5 kHz	12 bit	$\sim 1,000 \mu s$
SANDBed	28	MicaZ	40 kHz	16 bit	$\sim 10,000 \mu s$
w-iLab.t	200	TelosB	10 kHz	12 bit	unknown
FLOCKLAB	4×30	Opal, TinyNode IRIS, TelosB	28 kHz ^a 56 kHz ^b	24 bit	avg: $39 \mu s$ 85th%: $68 \mu s$

Table 2.4: Existing testbeds supporting distributed power measurements. FLOCKLAB is the only testbed with multiple platforms; it provides the highest resolution, and timestamps power samples with 20–200× better synchronization accuracy.

^ahigh-resolution mode

^bhigh-speed mode

2.5 Related Work

Sensor network testbeds. Departing from most of the early installations [EAR⁺06, WASW05], emerging testbeds are increasingly diverse and specialized: relocatable testbeds to evaluate applications in the intended target environment [RHLG10], testbeds with robots for controlled mobility experiments [JGMdDO10], and testbed federations to assess large-scale services [CPC⁺12]. FLOCKLAB, instead, aims to provide visibility into the distributed behavior of protocols and applications, to detect bugs and inefficiencies early in the development cycle. As such, to the best of our knowledge, FLOCKLAB is the first testbed with verified support for distributed, synchronized GPIO tracing and actuation coupled with high-resolution power profiling.

Closest to FLOCKLAB are PowerBench [HHP⁺08], SANDBed [HWM10], and w-iLab.t [BVJ⁺10]. As shown in Table 2.5, these testbeds also provide distributed power measurements at comparable or lower rates and resolutions. FLOCKLAB also achieves a better synchronization, allowing for a better alignment of power traces recorded at different nodes. Furthermore, FLOCKLAB supports four different platforms and future platforms can be added with little effort, whereas the other testbeds support only one platform. We note that w-iLab.t also seems to support GPIO-based services, but there exists no public information on the performance of these services in w-iLab.t.

DSN provides coarse network-wide power sensing by sampling the nodes' current draw every few minutes [DBK⁺07]. In addition, like MOTELAB [WASW05], DSN instruments one node with a high-precision multimeter. FLOCKLAB clearly exceeds the capabilities of these testbeds. However, the approach of coupling targets with powerful observers is

inspired by these and other systems [GES⁺04]. Like FLOCKLAB, many support multiple platforms [CPC⁺12, HKWW06], but, unlike FLOCKLAB, only serial I/O.

Power and energy estimation. Several sensor network simulators [BBV07, SHC⁺04] and emulators [LWG05] provide power or energy estimation capabilities. They mainly differ in the level of detail in which they model hardware components and program execution, and hence in the accuracy of their estimates. The basic approach consists of recording the time each hardware component spends in each power state, and combining these data with a calibrated power model of the target node.

Software-based online energy estimation follows the same approach, but performs time measurements on real nodes [DOTH07]. Different from simulation or emulation, intricate effects of interrupts and timers are automatically taken into account. Changes to existing code, overhead in terms of processing, memory, and code footprint, and lack of visibility into the instantaneous power draw are the downside of this approach.

By contrast, FLOCKLAB measures power non-intrusively on several platforms, enabling detailed profiling prior to deployment. Thus, FLOCKLAB has advantages especially in the early stages of development, whereas software-based estimation allows for energy profiling on larger testbeds.

Power and energy measurement. A number of methods exist for measuring rather than estimating power or energy. Some target external profiling [MMJ⁺05, THBR11], while others enable a node to measure its own consumption [DFPC08, JDCS07]. Different from FLOCKLAB, none of them addresses the challenge of synchronizing measurements across multiple nodes.

SPOT [JDCS07] uses a voltage-to-frequency converter to feed an energy counter that is read by the node. It achieves high accuracy across a large dynamic range, assuming a constant supply voltage. Aveksha adopts a similar approach to obtain power traces [THBR11]. The design in [TW08] measures also the supply voltage to accurately calculate energy as the voltage varies. iCount provides energy metering at nearly zero cost by counting the cycles of a node's switching regulator [DFPC08].

Quanto [FDLS08] builds on iCount to obtain the energy breakdown per programmer-defined activity, using regression models and causal activity tracking. Targeting high-performance sensing platforms, [SMK08] resolves energy usage at the level of processes and hardware components using a dedicated integrated circuit. By combining GPIO tracing with power profiling, also FLOCKLAB can be used to track network-wide activities and subsequently attribute costs to each activity.

Sensor network debugging. A wealth of research has been devoted to diagnosing and debugging wireless embedded systems. Existing approaches target failures caused by interactions among multiple nodes [KLA⁺08, RM09], network faults such as routing and node failures [LLL⁺08, RCK⁺05], or node-local bugs including data races and stack overflows [SEZ10, YSSW07].

Most of these systems feature a frontend that collects data about the running system and a backend that analyzes these data for possible failures. FLOCKLAB does not solve the latter, but it provides correlated power and event traces in a way that is nearly unobtrusive for the debugged application. This is in sharp contrast with many debugging techniques that perturb the timing behavior by adding debug statements, logging events into non-volatile memory, or transmitting debug messages in-band with application traffic. Because of this, FLOCKLAB can be highly effective in detecting failures due to time-critical interactions among multiple nodes, possibly by applying distributed assertions [RM09] or data mining techniques [KLA⁺08] on event traces. Moreover, power traces can be exploited for conformance testing [WLT09] and failure diagnosis [KLL⁺10]. For cycle-accurate debugging of a single node, however, other solutions may be more suitable.

For instance, Aveksha uses a custom-built debug board to interface with the on-chip debug module through JTAG [THBR11]. It provides breakpoints, watchpoints, and program counter polling for very detailed event tracing, and power measurements that can be correlated with events of interest. Aveksha is truly non-intrusive, except for breakpoints. However, the design is tied to MSP430-based platforms, and setting triggers correctly may require detailed knowledge of machine code and memory addresses. Instead, FLOCKLAB makes distributed event tracing as simple as LED and `printf` debugging, supports several platforms and MCUs, and facilitates the integration of new ones with little effort.

2.6 Summary

FLOCKLAB is the result of a multi-year effort to push beyond the capabilities of contemporary testbeds, providing the research community with a shared tool to study wireless embedded system in unprecedented detail. By providing new GPIO based tracing and actuation services and higher resolution power profiling than previous testbed approaches, FLOCKLAB allows to observe states and power dissipation on all nodes in the testbed with virtually no impact on devices under test. We presented the design of FLOCKLAB, benchmarked its performance, and demonstrated its utility

through real-world test cases.

FLOCKLAB has been in operation for 4 years now. During this time, more than 160 users from all over the world have used the testbed for their experiments. FLOCKLAB served educational purposes in several university classes within ETH Zurich and beyond, as well as for experiments that were part of more than 40 scientific publications.

3

Time-of-Flight Aware Time Synchronization for Wireless Embedded Systems

Distributed measurements, e.g., as performed in FLOCKLAB, need a common time scale in order to totally or partially order events. In this chapter, we explore and extend the limits of time synchronization in wireless multi-hop networks.

The requirements on time synchronization depend on the specific application area. For many applications, accuracies in the millisecond range is sufficient [CRM⁺08]. Other applications like distributed control in automation or distributed measurements, e.g., for network event analysis [RGD⁺15] or data acquisition during flight tests [MGHC07], require a higher degree of time synchronization in order to guarantee failure-free operation and faithful alignment of observations. In a testbed such as FLOCKLAB, tightly synchronized measurements are required to assess interaction between nodes below the level of single radio transmissions, e.g., sleep and active states of radio receivers or radio interference.

Established solutions to synchronize distributed systems with sub-microsecond precision are in general either based on satellite communication, such as the Global Positioning System (GPS), or wired infrastructure. Using GPS, it is possible to acquire very accurate timing, e.g., standard commodity L1-GPS receivers have an average timing error of 60 ns [u-b14]. For small and spatially limited deployments, or for locations without satellite reception, wired approaches are an alternative.

A prominent example is the precision time protocol [ptp08], which can leverage existing Ethernet infrastructure. However, the cost for integrating a GPS receiver is high, both economically and power-wise, and wired solutions for locations without exiting infrastructure have high initial cabling cost.

If no wired infrastructure or satellite reception is available or too costly, e.g., in a building without Ethernet infrastructure, time synchronization using a network of wireless embedded systems might be a viable solution. Current approaches achieve synchronization errors in the millisecond to microsecond range using inexpensive hardware, e.g., PulseSync has a worst case synchronization error of $19\ \mu\text{s}$ in a 30-hop line topology [LSW14].

Emerging embedded platforms for cyber-physical systems are more sophisticated than first generation wireless sensor network platforms, thereby providing higher clock rates and radio transceivers that are integrated into computation units. Examples of such integrated chips are the Texas Instruments CC430 or CC2538 series, combining an MSP430 microcontroller core with a sub-1 GHz radio transceiver or an ARM Cortex-M3 and an IEEE 802.15.4 compliant 2.4 GHz radio. Existing wireless time synchronization protocols can profit from this development in several ways. Faster system clocks result in higher time resolution of packet timestamps, while integration of MCU and radio core on one chip facilitates tighter control of the radio core.

In light of these developments, we revisit existing concepts with the aim to narrow the gap between wireless multi-hop time synchronization and its wired and GPS counterparts, and therefore bringing flexible and lower cost time synchronization to a wide set of applications that require sub-microsecond timing accuracy.

Challenges. To increase the coverage of a wireless network, nodes use intermittent hops to forward information to receivers that are not in communication range. When targeting sub-microsecond time accuracy, propagation delays are not negligible and need to be appropriately considered when exchanging time information. In addition, it has been shown that fast propagation of time information is essential, as error accumulation is proportional to the time spent in the network [LSW09, ZCH11]. Bringing both objectives together is a non-trivial task.

Contributions and road-map. We identify unequal propagation delays as an important aspect to further increase the accuracy of current time synchronization protocols. Based on the obtained insight, we propose the Time-of-Flight Aware Time Synchronization Protocol (TATS), a new protocol that compensates propagation delays on communication paths. TATS builds on existing flooding based synchronization approaches and

introduces propagation delay measurements *without* sending additional packets.

In summary, this chapter makes the following contributions:

- In Section 3.2, we assess the impact of propagation delays on two state-of-the-art synchronization protocols, namely Glossy [FZTS11] and PulseSync [LSW09]. We reveal a dependency between the minimal achievable global synchronization error and network topology, and thus motivate the need for propagation delay compensation.
- In Section 3.3, we design TATS, a multi-hop time-synchronization protocol that compensates the propagation delay experienced on communication paths with no additional packet complexity.
- We discuss implementation details of TATS on a recent hardware platform in Section 3.4.

We evaluate TATS in Section 3.5 and compare its performance against Glossy and PulseSync on FLOCKLAB (see Chapter 2). To show the impact of network topology on the global synchronization error, we use three different topologies: a short and a long 22-hop line topology, and a dynamically built distribution tree. Overall, TATS achieves an average synchronization error of $0.24 \mu\text{s}$ and a maximal synchronization error of $0.54 \mu\text{s}$, which is up to a factor of 6.9 better than its competition. To the best of our knowledge, we are the first to report of a *sub-microsecond* synchronization error over *tens* of hops using off-the-shelf wireless embedded nodes.

3.1 Related Work

This section summarizes publications closely related to the proposed TATS protocol. For an exhaustive survey on time synchronization protocols, the reader is referred to [SH15].

Wireless sensor networks. Table 3.1 gives an overview on synchronization accuracies reported for time synchronization protocols running on typical sensor node platforms. Direct comparison by numbers is not possible because evaluations are conducted under different circumstances. In addition, accuracies are sometimes reported as mean absolute error, i.e., the unsigned deviation, and sometimes as mean signed deviation. The latter generally results in lower values.

TPSN [GKS03] employs a two-way message exchange to measure the delay introduced by the communication stack. Although such

Table 3.1: Reported synchronization errors. *If not mentioned otherwise, error values are reported as magnitudes.*

Protocol	Hops	Interval	Avg	Max	Platform
TPSN	1	^{-b}	16.9 μ s	44 μ s	Mica
FTSP	6	30 s	2.3 μ s	14 μ s	Mica2
FTSP ^a	1	10 s	0.13 μ s ^c	n/a	Epic
RATS	11	30 s	2.7 μ s	26 μ s	Mica2
PulseSync	30	10 s	2.06 μ s	19 μ s	Opal
Glossy	8	^{-b}	0.4 μ s ^c	n/a	TelosB

^ausing virtual high resolution timer [SDS10]

^bsingle measurements, no linear regression

^cmean signed deviation, not mean absolute error

measurements include physical propagation delays of electromagnetic waves, other delays, e.g., jitter in radio interrupts, dominate the measurements. Different to our approach, TPSN creates a fixed hierarchical network structure and two messages are exchanged *per link* for a two-way delay measurement and time synchronization. In contrast, TATS builds on a dynamically built flooding tree and only one broadcast message per node to perform the same measurements.

By employing more sophisticated MAC-layer timestamping [MKSL04], the measured delay between sending and receiving a packet has a significantly narrower distribution and can be approximated by a constant value. This enables time synchronization using only unidirectional communication, e.g., FTSP [MKSL04], RATS [KDL⁺06] and PulseSync [LSW14] synchronize a network by flooding time information using broadcast messages. Communication patterns are less complex because flooding does not need a sophisticated routing tree. Different to TATS, performance of these protocols depends on the choice of a message delay calibration and on the distribution of propagation delays between individual nodes.

Glossy [FZTS11], a flooding architecture for wireless sensor networks based on concurrent transmissions implicitly provides network-wide synchronization. As such, common reference times can be computed on every node of the network. In contrast to TATS, Glossy does not foresee propagation delay compensation or drift compensation.

TATS builds on various concepts introduced by other time synchronization protocols. We use linear regression, as introduced by RBS [EGE02], to compute the offset and the speed of the local

clock relative to a reference clock. Same as RATS [KDL⁺06] and PulseSync [LSW14], TATS coordinates packet transmissions in order to achieve fast information propagation over several hops. By doing so, clock drift on forwarding nodes have a lower impact on time synchronization error. In [SDS10], the idea of high resolution, low-power clocks is introduced, which reduces the synchronization error. To accurately timestamp radio packets, TATS relies on a high frequency clock.

In summary, propagation delay has been treated as a negligible source of synchronization error in wireless sensor networks. In contrast to the mentioned work, we show that propagation delay plays a major role for sub-microsecond time synchronization accuracy and we propose TATS, a new time synchronization protocol that compensates for different propagation delays per link. TATS brings together the simplicity of unidirectional network flooding and the propagation delay awareness of two-way message exchange schemes.

High latency acoustic networks. While the need for compensation of propagation delays in air has been mostly neglected, it is more prominent in underwater networks [HYW⁺06]. In water, acoustic waves are used for communication. Compared to RF communication, the speed of acoustic waves is five orders of magnitudes slower. Due to different environmental influences such as multi-path effects or time dependent propagation characteristics, it is unclear whether such synchronization protocols can be directly applied to RF based communication. In addition, complex messaging hinders fast dissemination of time information over several hops. TSHL [SH⁺06] employs a two-phase approach to first estimate the local speed of the clock and then use a two-way message exchange to measure the propagation delay.

3.2 Impact of Propagation Delay

In this section we motivate that it is important to take propagation delay into account when designing a time synchronization protocol. We use the term *propagation delay* to refer to the duration a signal travels between the antennas of two communication partners. First, we quantify the propagation delay in wireless embedded systems and compare it against other errors present when synchronizing time in multi-hop networks. Then, we analyze the impact on PulseSync and Glossy, two state-of-the-art time synchronization protocols that treat propagation delay as a negligible quantity.

3.2.1 Time-of-Flight vs. Other Sources of Error

In order to assess the impact of propagation delays in wireless embedded systems, we put the error into perspective. Radio communication, based on electromagnetic waves, propagates at the speed of light, i.e., approximately 3×10^8 m/s. The indoor communication range of a typical wireless sensor node is 20–30 m [MEM11]. Accordingly, a message traveling between two nodes can experience a propagation delay of up to 100 ns.

To put this value into perspective, we consider time synchronization accuracies reported in literature as listed in Table 3.1. Although one cannot directly compare the protocols, as they are evaluated on different hardware platforms and using different settings and algorithms, we get a good picture of accuracies currently attained. As stated by [LSW09], synchronization error in a multi-hop network is a function of the network diameter. The more hops involved, the worse the accuracy. Depending on the algorithm, the error grows exponentially, linearly, or sub-linearly ($\sqrt{\text{diameter}}$) with the network diameter [LSW09]. To approximate the error introduced by each hop, we divide the average error by the number of hops. The lowest value for protocols in Table 3.1 results from the PulseSync protocol: $2.06 \mu\text{s}/30 = 67$ ns. Except for TPSN, all the listed protocols treat propagation delay as being constant.

We conclude that the impact of varying propagation delays is comparable to the error introduced by other effects like jitter when timestamping a packet or clock drifts between nodes. For outdoor deployments, the effect of propagation might be even more severe, as communication ranges are larger. Some deployments exhibit node distances of several hundreds to thousands of meters, e.g., on bridges [KPC⁺07] or in alpine environments [per].

3.2.2 Existing Multi-hop Time Synchronization Protocols

To assess the potential of propagation delay compensation, we investigate two recent protocols that are representative for the current state-of-the-art for time synchronization in wireless sensor networks. We identify shortcomings that prevent better accuracy just by increasing the frequency with which nodes re-synchronize. In the following, we use the term *message delay* to refer to the time between timestamping a message on the sender and the receiver. This delay also includes the propagation delay.

PulseSync [LSW14] builds on the insight that it is beneficial to forward time information as fast as possible through the network. To do that, the protocol floods *pulses* through the network. Each node sends exactly one message within each pulse after having received the message from its

predecessor. The initiating reference node embeds its current clock value into the message. All forwarding nodes update the time value by adding the message delay and the dwell time of the message. Here, the dwell time is the difference of the local clock values taken at receive-time and at send-time. The message delay for all pair-wise links is assumed to have the same normal distribution with a known mean value, i.e., the “differences in radio propagation times can be neglected in sensor networks” [LSW09]. The message delay is determined during a calibration phase.

Every message received serves as a sample point that relates the reference node’s time to a local clock value. The slope and the offset of each node’s local clock is then calculated using least squares linear regression over the last k sample points. PulseSync implements an optional drift compensation to reduce the error that is added by updating the time information in the packet on each node. This error stems from measuring the dwell time using local clocks that run at a slightly different speeds than other nodes.

To estimate the impact of calibrating the protocol with a single propagation delay parameter τ_c , we assume in the following a network where packets are perfectly timestamped (no jitter) with an arbitrarily accurate time resolution. Nodes have perfect clocks without drift and the message delay is equal to the propagation delay. Let us denote the number of hops a packet in a pulse travels from the reference node to node v as h_v , and the real accumulated propagation delay on this path from the reference node to v as τ_v . Since the constant message delay τ_c is added to the reference time at each hop, the error that results from imperfect knowledge of the propagation delay at node v is $h_v\tau_c - \tau_v$. The resulting global synchronization error \mathcal{G} , i.e., the maximal pairwise error across all nodes in the network is

$$\mathcal{G} = \max_v(h_v\tau_c - \tau_v) - \min_v(h_v\tau_c - \tau_v). \quad (3.1)$$

We see that the resulting global synchronization error heavily depends on the network topology, i.e., h_v and τ_v . An optimal parameter τ_c that minimizes \mathcal{G} can be found using linear programming. In general, it is difficult to find the optimal τ_c as this requires knowledge of all possible paths of the flood and the respective path delays. In addition, network structures change over time, e.g., due to mobility or changes in the environment, necessitating an adaption of τ_c . A necessary condition for the error to vanish completely is that all path delays τ_v are multiples of τ_c , which is very unlikely to be the case in a real wireless sensor network deployment.

Glossy [FZTS11] is a flooding mechanism based on concurrent transmissions that allows to disseminate messages in a multi-hop network

as fast as possible. To synchronize time, an *initiator node* starts a flood and embeds its clock value into the first packet. Every node that receives a packet immediately retransmits it, thereby effectively synchronizing packets sent in the same slot. With every transmission, a counter value c contained in the packet is incremented by one. With this information it is possible to calculate an estimate of the start time of the flood as

$$\hat{t}_{ref} = T_{c_0} - c_0 t_{slot}, \quad (3.2)$$

where T_{c_0} is the local time of the first received packet and c_0 the counter value contained in this packet. The propagation delay is contained in the t_{slot} value, as this is the interval between the start of a packet transmission with relay counter c and the start of the following packet transmission with relay counter $c+1$. Nodes estimate t_{slot} locally using packet timestamps. In [FZTS11], the authors assume that “ t_{slot} is a network-wide constant, since during a flood nodes never alter the packet length”.

There are two aspects where propagation delay plays a role: (i) the timestamp T_{c_0} is affected by different propagation delays, leading to a similar effect as for PulseSync if a constant propagation delay is assumed for the whole network; (ii) the slot time t_{slot} is strictly speaking not a network-wide constant, but rather depends on the immediate neighborhood of a node. To show this, we conduct following experiment. We let Glossy run in a setup as shown in Figure 3.1: three nodes are placed within different distances to each other. No communication is possible between the two outer nodes¹.

Communication link (I)-(F) experiences a longer delay than link (I)-(N). Node (I) starts a flood, while (F) and (N) participate in the flood. After receiving (I)’s message, (F) and (N) are transmitting the message concurrently. As (N)’s signal is received stronger at (I), due to capture effect [LF76], the packet sent by (F) has no impact on the timing at (I). After a while, we turn off (N).

The acquired slot estimates on individual nodes are shown in Table 3.2. When all nodes are participating in the flood, the estimates are similar. However, as we turn off the closer node (N), slot estimates become larger by approximately 150 ns. Such variations in t_{slot} have a large impact on the calculated reference time (3.2) because they are multiplied by the number of hops c_0 .

¹Communication channels are enforced by coaxial cables.

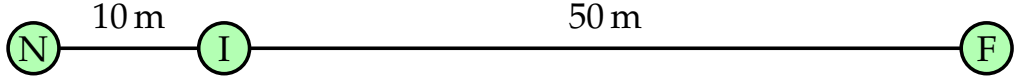


Figure 3.1: Experimental setup to show the influence of the capture effect on slot length measurements.

Table 3.2: Slot times estimated during Glossy floods.

	Initiator (I)	Node (F)	Node (N)
All nodes	516.78 μ s	516.78 μ s	516.78 μ s
Without (N)	516.93 μ s	516.95 μ s	-
Difference	0.15 μ s	0.17 μ s	-

3.2.3 The Need for Propagation Delay Compensation

Varying propagation delays can introduce per-hop errors as high as 100 ns for indoor deployments and are therefore relevant when aiming for sub-microsecond synchronization accuracy. State-of-the-art time synchronization protocols handle errors well that stem from clock drift and message delay jitter, by providing a fast flooding mechanism or combining several measurement points using linear regression, but lack the awareness for propagation delays.

3.3 Time-of-Flight Aware Time Synchronization

In this section, we describe TATS, our new protocol that combines per-link message delay compensation and fast flooding for highly accurate time synchronization. As seen in Section 3.2, synchronization accuracy suffers from unknown propagation delays between nodes. Therefore we want to compensate for this variation across the network, while keeping the advantages of state-of-the-art protocols, namely high synchronization accuracy due to fast dissemination, and low overhead due to flooding. Furthermore, the number of additional messages needed should be minimal.

We decompose the propagation delay compensation on a link into two steps: (i) estimating the delay on a link, and (ii) updating the time value contained in a message by adding the delay estimate. The message delay on a link can be estimated using *two-way delay* measurements, as depicted in Figure 3.2, also applied by TPSN [GKS03]. Two messages are exchanged per link: node 0 sends a packet to node 1 and remembers the timestamp T_0 . Upon reception, node 1 replies with a packet that contains

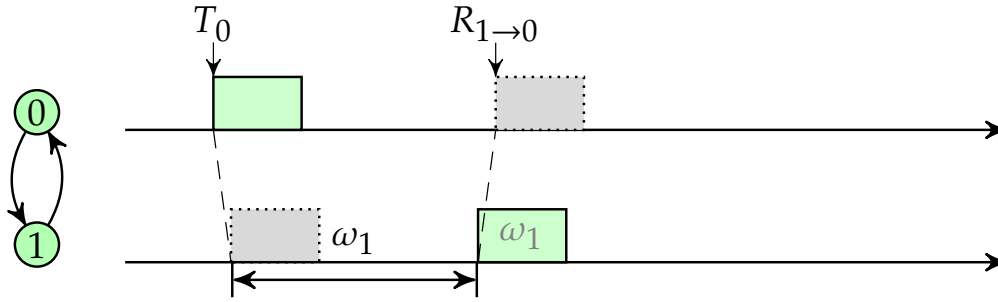


Figure 3.2: Two-way round-trip measurement.

the dwell time ω_1 , which is then used by node 0 to compute the two-way delay as $R_{1 \rightarrow 0} - T_0 - \omega_1$, where $R_{1 \rightarrow 0}$ is the reception time of the packet at node 0. The one-way delay is computed by dividing the two-way delay by two.

Adding *low overhead* message delay compensation to existing flooding based protocols is challenging for three reasons:

1. In contrast to flooding, where every node sends just one broadcast packet, the two-way delay measurement involves two packets *per link* and adds therefore considerable overhead.
2. After a message exchange, only the initiating node (node 0 in Figure 3.2) knows the delay. As floods are based on broadcasts and it is therefore unknown who will receive the packet, the delay estimate has to be compensated by the receiving node(s), hence the propagation delay knowledge is needed at the receiving node 1.
3. Two-way delay measurements are only feasible if links are bidirectional. Flooding does not have this restriction and therefore might use links for which message delays are not obtainable. Unidirectional links are very common in real deployments, e.g., [OC10] reports on a testbed where 46% of the links are unidirectional.

Next, in Section 3.3.1, we give an overview of our approach. In Section 3.3.2, we describe our method to measure message delays using broadcast packets, and finally, we introduce a heuristic that makes our protocol more resilient to non-symmetric links in Section 3.3.3.

3.3.1 Overview

The aim of TATS is to establish a global time on all nodes in the network that is synchronized to a reference node. The network is assumed to be short term stable, i.e., the mean propagation delay between nodes

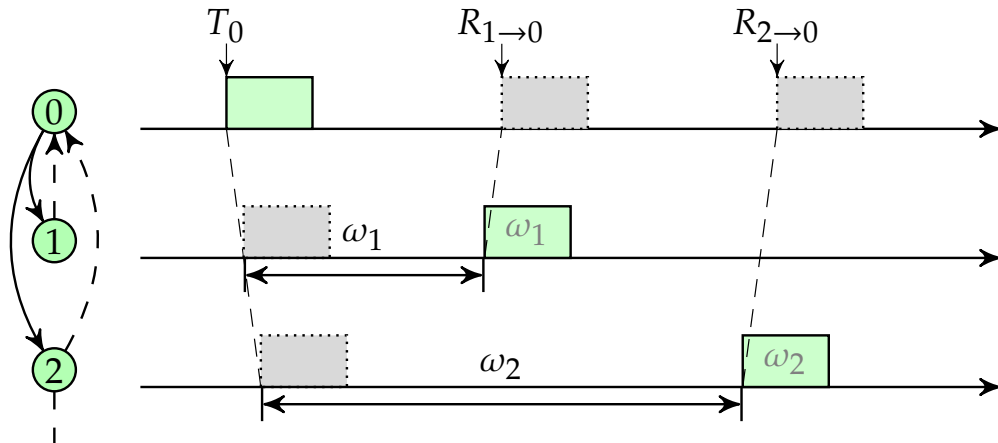


Figure 3.3: Round-trip measurements are based on time information embedded into time synchronization packets. A parent node 0 can acquire several measurements by listening to the packets that are transmitted by its children 1 and 2.

only changes slowly over time. This assumption is reasonable for static networks. We use PulseSync [LSW14] as a starting point and extend it with a propagation delay compensation: Similar as in PulseSync, messages containing the reference time are periodically flooded to all nodes, initiated by the reference node. Each node participates in the flood by (i) reading the reference time (ii) adding the message delay to it and, (iii) on transmission, adding the dwell time to the reference time. All communications are broadcasts and nodes transmit once for every flood, after a random and short timeout after receiving a packet. Each flood implicitly creates a routing tree, thereby defining a parent-child relation between nodes.

Received delay-compensated reference times are stored in a table together with the corresponding local reception times of packets. A node then performs a least squares linear regression on these value pairs to calculate the time offset and clock drift of the local clock relative to the reference clock.

Different to PulseSync, TATS applies individual message delays for each link in step (ii). Two-way message delays are measured by piggy-backing additional information onto regular synchronization packets. In this way, we add propagation delay compensation to PulseSync *without* sending additional packets. Next, we detail our approach for propagation delay measurements.

3.3.2 Propagation Delay Estimation

Let's consider Figure 3.3, which depicts a small part of a network consisting of three nodes. All links are bi-directional, i.e., communication is possible in both directions. Node 0 starts a flood by sending a broadcast packet containing the reference time. This packet is received by nodes 1 and 2. After a random timeout, each receiver updates and forwards the packet. As all transmissions are broadcasts, node 0 overhears the forwarded packets. This chain of actions resembles the same information flow as is needed to carry out a round-trip time measurement, as shown in Figure 3.2. To use this information flow for two-way measurements, messages need to contain the dwell time ω and the identifier of the parent. The latter is needed because communication is based on broadcast packets. Without that information, node 0 could not distinguish between messages of its child nodes and messages of other nodes. The one-way message delay between nodes v and w after flood k is computed as

$$\delta_{v \leftrightarrow w}^k = \frac{R_{w \rightarrow v} - T_v - \omega_w}{2}. \quad (3.3)$$

T_v and $R_{w \rightarrow v}$ are the timestamps taken at node v when node v sent its message and when it received a message from node w . The dwell time on w is denoted as ω_w . Because propagation delays are short term stable, a more accurate delay estimate $\bar{\delta}_{v \leftrightarrow w}^k$ can be obtained by averaging N consecutive measurements:

$$\bar{\delta}_{v \leftrightarrow w}^k = \frac{1}{N} \sum_{i=k-N+1}^k \delta_{v \leftrightarrow w}^i \quad (3.4)$$

In this way, parents can obtain message delay estimates for all links towards all their children. Every node keeps a number of most recent delay measurements in a table. As stated in Section 3.2, the estimates are needed on child nodes to compensate for propagation delays. To inform child nodes about message delays, parents embed the obtained average message delay into the time synchronization packet. The resulting packet format of TATS is shown in Table 3.3. A parent needs to forward estimates to potentially many children. As only a limited number of estimates fit into a synchronization message, parents select estimates to send in a round-robin fashion.

For every received time synchronization message in a flood, a child node w compensates the message delay by looking up the value belonging to the link $v \rightarrow w$ and adds that to the received reference time $G_{v \rightarrow w}$ to obtain the compensated reference time G_w :

Table 3.3: Structure of synchronization packets.

Name	Description
Sequence number ^a	Sequence number of flood
Reference time G^a	Global time
Node ID of parent	Parent ID in this flood
Dwell time ω	Elapsed time between receiving and sending
Node ID of measurement	Identifies the link of the measurement
Message delay $\bar{\delta}$	Average message delay measured by this node

^aSame as in PulseSync

$$G_w = G_{v \rightarrow w} + \bar{\delta}_{v \leftrightarrow w} \quad (3.5)$$

The proposed mechanism does not prevent collisions, e.g., node 1 and node 2 in Figure 3.3 could potentially send their packets at the same time during a flood, therefore render it impossible to perform a delay measurement. As timeouts are random, eventually, in a consecutive flood, a measurement will be possible. Because message delays are short term stable, a certain delay in measuring and distributing estimates is permitted and has a negligible effect on the performance.

3.3.3 Avoiding Unidirectional Links

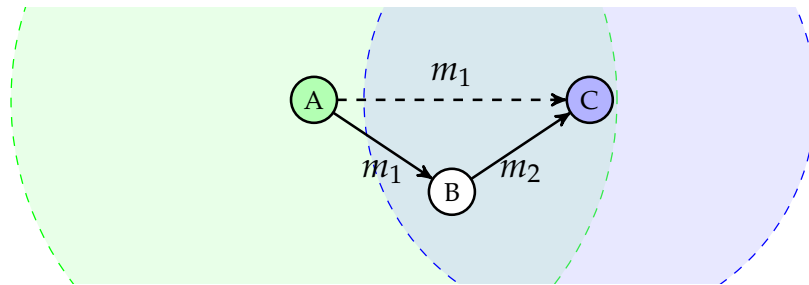


Figure 3.4: Unidirectional links prevent round-trip measurements. By introducing a short delay, intermediate nodes get the chance to forward the time information over bidirectional links.

The proposed mechanism for round-trip measurements requires that links can be used in both ways, otherwise message delay measurements are not possible. Possible reasons for unidirectional links are interference or weak signal powers close to a receiver's sensitivity threshold. In the situation illustrated in Figure 3.4, node C can hear node A, but communication in the opposite direction is impossible. Therefore, the propagation delay between A and C cannot be estimated. If there is an additional node B with bidirectional links to both A and C, route $A \rightarrow B \rightarrow C$ would allow for round-trip measurements on each link and consequently propagation delay compensation could be applied. We observe that in Figure 3.4, node C will eventually receive a message directly from A (m_1) and another one relayed over B (m_2). By ignoring the earlier message m_1 , we can establish the desired route.

TATS exploits this observation to reduce the number of unidirectional links used in a flood. Every time a packet is received over a link with unknown message delay, an additional waiting period is introduced before forwarding the message. If a messages from a neighbor with known message delay arrives during this period, the earlier message is ignored.

Our evaluation in Section 3.5.2 shows that this heuristic results in more round-trip measurements and less missing delay estimates.

3.4 Implementation

We implement TATS in Contiki OS [con] on a CC430 developer board to show its feasibility and to benchmark the performance.

Hardware platform. We use the Olimex MSP430-CCRF developer board [OLI13] as hardware platform for our implementation. This board features a low-power Texas Instruments CC430F5137 SoC, providing 32 kB of program memory and 4 kB of RAM. The chip integrates an MSP430 core and a CC1101 sub-1 GHz radio with configurable bit rate and radio modulation. The on-board printed PCB-antenna is used. A 26 MHz quartz oscillator provides the basis of a stable 13 MHz system and timer clock. The quartz has a nominal frequency deviation of ± 10 ppm and a temperature dependent deviation of ± 10 ppm over the specified range from -25 to 75°C . System time is stored in a 16-bit counter value and extended, on overflow, by incrementing an additional integer variable to a 64-bit timestamp.

Message timestamps. For propagation delay measurements, timestamps are taken on the sending and on the receiving nodes. Packet based radios like the CC1101 generate interrupts when a synchronization symbol is

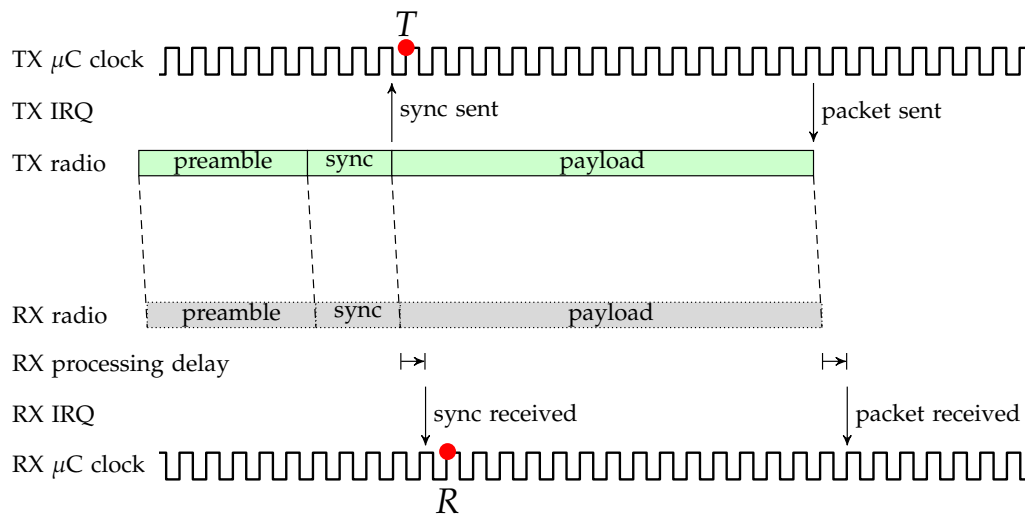


Figure 3.5: Timestamps for one message transmission. Timestamps T and R are inaccurate due to asynchronous clocks and uncertainties introduced with radio modulation.

detected. These interrupts occur both on the sender and on the receiver. The chain of events involved in message timestamping is shown in Figure 3.5. As soon as the sender has transmitted the synchronization symbol, an interrupt signal is generated. At the next rising edge of the sender's clock, the value of the timer register is stored as the timestamp T . On reception of the synchronization symbol, the receiver stores its timestamp R in a similar way.

Message timestamps are affected by jitter, which is caused by asynchronously running digital clocks and conversion between digital and analog domain when generating or decoding the radio signal. The smaller the jitter, the more accurate the resulting time synchronization. Therefore a fast clock is beneficial for synchronization.

We configure the radio to use GFSK modulation and a data rate of 250 kbps. The distribution of the message delay over a short distance, experimentally measured using two nodes on a desk and an external logic analyzer, is shown in Figure 3.6. The delay is normally distributed with a mean value of $13.68 \mu\text{s}$ and a standard deviation of 107 ns . Compared to other hardware platforms, this is a relatively low value (see Table 3.4). Lower jitter should potentially lead to lower synchronization error, provided the clock resolution for timestamps is sufficiently high. In our case, we can rely on a 77 ns clock resolution. We use the capability of the MSP430 to capture time values with dedicated capture registers, thus avoiding software interrupt delays when taking timestamps.

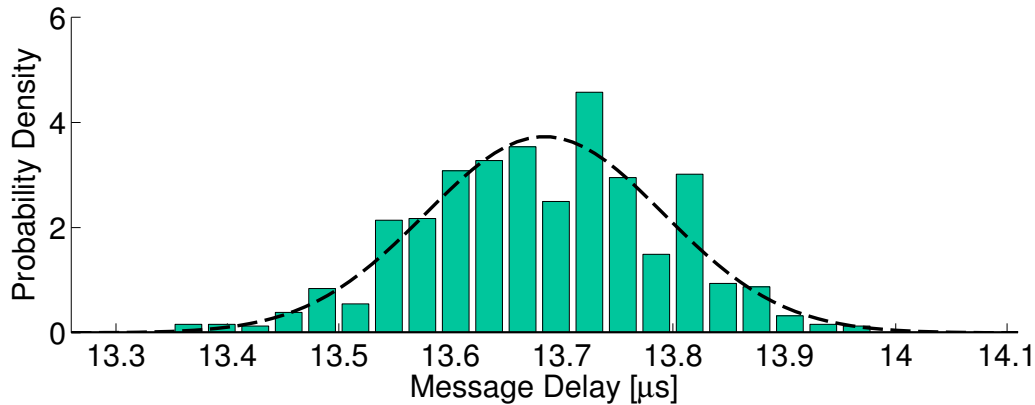


Figure 3.6: Message delay distribution of the CC430 radio on a single link. The dashed curve is a fitted normal distribution with a mean value of $13.68 \mu\text{s}$ and a standard deviation of 107 ns .

Table 3.4: Standard deviations for message delays on different platforms.

Platform	Standard deviation
TelosB [PSC05]	41 ns [SDS10]
Opal (AT86RF231) [J ⁺ 11]	180 ns [LSW14]
Mica2 [HC02]	$1.95 \mu\text{s}$ [SW09]
RF230 radio	370 ns [SDS10]
MSP430-CCRF	107 ns

3.5 Evaluation

In this section we evaluate TATS in FLOCKLAB (see Chapter 2). As TATS does not employ explicit two-way round-trip measurements, we will evaluate how quickly delays are measured by parents and forwarded to child nodes. In a second experiment, we do a head-to-head comparison of TATS against PulseSync and Glossy in different network structures. Our experiments reveal the following key findings:

- TATS quickly acquires message delay estimates solely based on network flooding.
- Despite unidirectional links, the acquired delay estimates allow to compensate propagation delay on 95 % of all involved links.
- In a 22-hop line topology TATS performs up to $6.9 \times$ better than PulseSync with respect to the average maximal synchronization

error and $3 \times$ better than Glossy and PulseSync on a shorter dynamic topology.

- In all settings, TATS's *maximal* synchronization error is clearly below 1 microsecond.

3.5.1 Experimental Setup

A common method to evaluate the synchronization accuracy of a protocol on real hardware is to put all the sensor nodes into a single broadcast domain and enforce a logical topology in software, i.e., only certain links are allowed to be used for communication. The accuracy is then measured by letting all nodes capture the time of a commonly received packet [LSW14, MKSL04]. Using a message as common reference is not preferable in our case as this message is also affected by propagation delays and would therefore reach different nodes at different time instances. Moreover, such a setting does not resemble well a real deployment, where nodes are scattered over a large area.

Therefore we choose a more realistic approach by letting nodes actually form a real multi-hop network. We run our tests on FLOCKLAB where 31 nodes are spread over an area of 75×35 meters in an office environment and also outdoors. The detailed layout of the testbed is shown in Figure 3.7. Since FLOCKLAB's time accuracy is not sufficient to accurately measure time in the sub-microsecond range, we equip six nodes with GPS receivers that generate an accurate reference pulse, i.e., a digital signal that has a low-high transition every second. This pulse is then connected to a GPIO pin of a node and timestamped using capture registers. The computed global timestamp of this event is then used to calculate the synchronization error. The employed LEA-6T GPS receivers provide timing accuracy with a root-mean-square error of 30 ns [u-b14].

For all experiments, we use a transmission power of 10 dBm and a radio frequency of 870 MHz.

3.5.2 Propagation of Message Delay Estimates

This experiment evaluates the feasibility of message delay measurements without additional packets, only based on flooding. In TATS, two-way delays are measured by parent nodes and then forwarded to child nodes. As described in Section 3.3.3, unidirectional links prevent two-way delay measurements, therefore TATS introduces a strategy to circumvent unidirectional links. If a synchronization packet is received over a link with unknown propagation delay, nodes wait for an additional waiting period for a synchronization packet on a link with known propagation

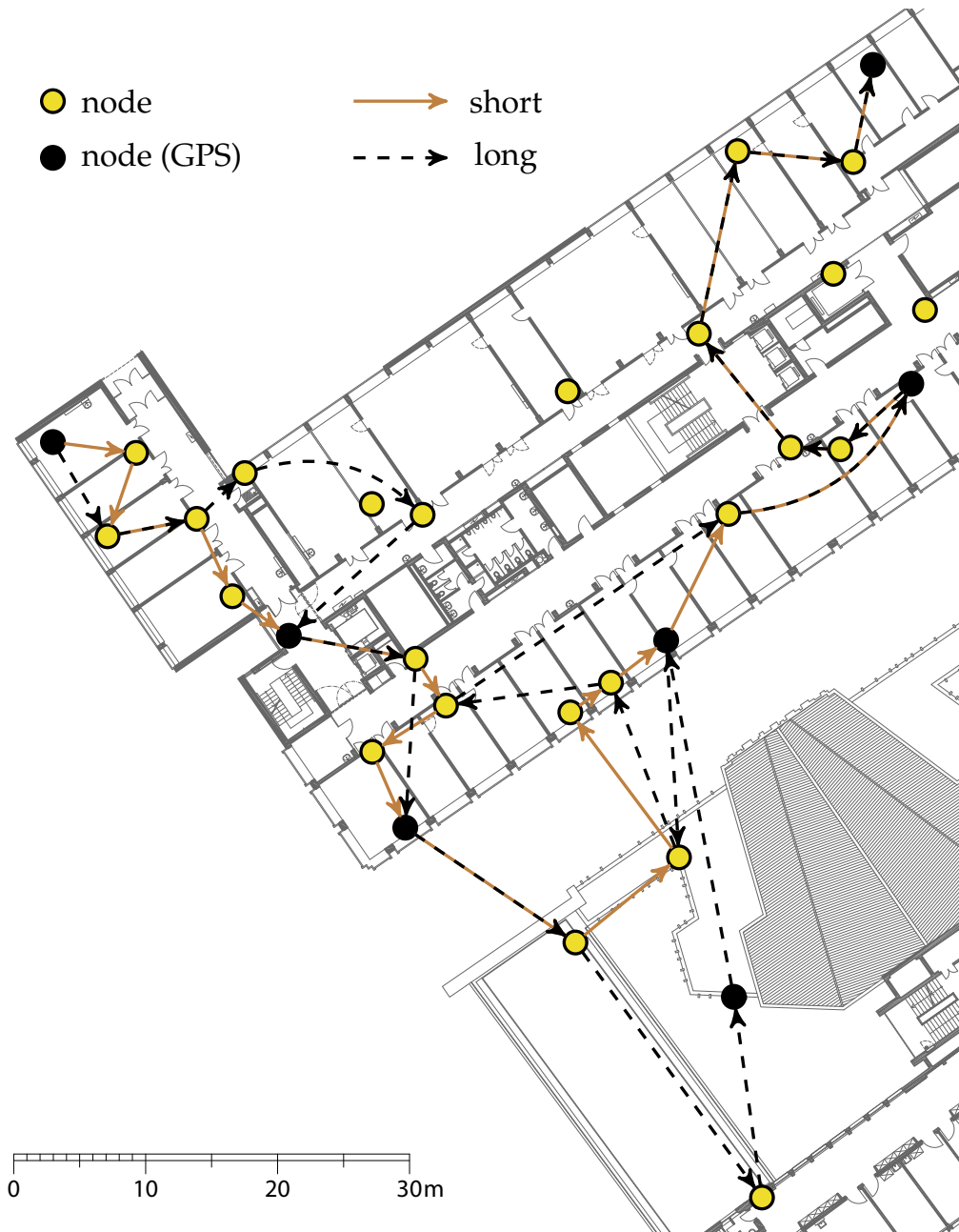


Figure 3.7: Testbed layout and software enforced path for the *short* and *long* line topology. Nodes in black are equipped with a high precision GPS receiver that generates a synchronized reference pulse. In the dynamic topology, all nodes participate.

delay. In this experiment, we quantify the impact of this strategy. We run TATS once without additional timeout (*immediate forwarding*) and once with a waiting period of 10 ms (*delayed forwarding*).

Setup. For each configuration, we let the synchronization protocol run

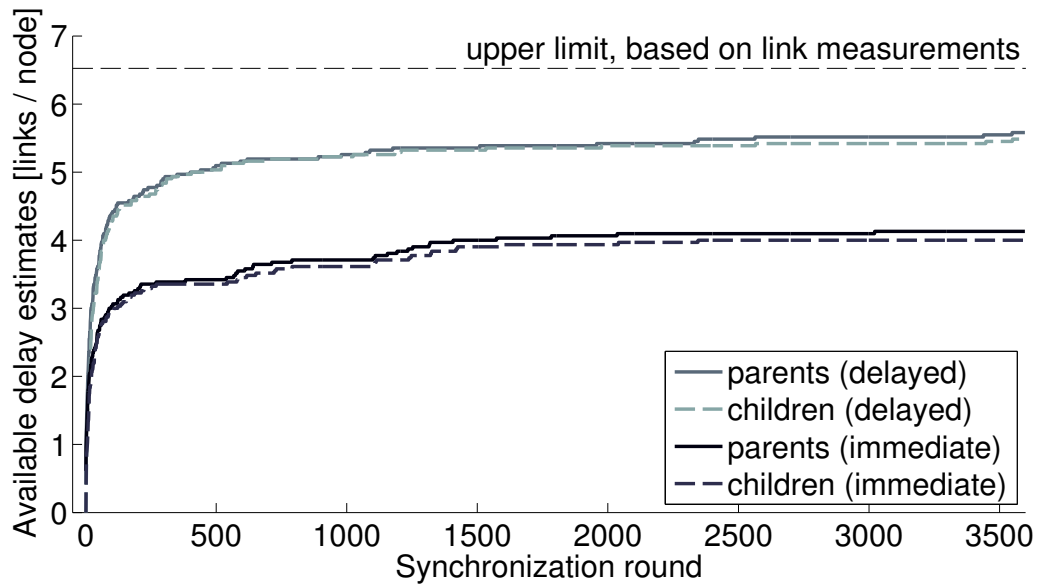


Figure 3.8: Available link delay estimates, on parents and on child nodes for TATS using immediate and delayed forwarding. *Measurements propagate quickly from parents to child nodes. Delayed forwarding achieves 37 % more measurements and covers 85.5 % of all possible links.*

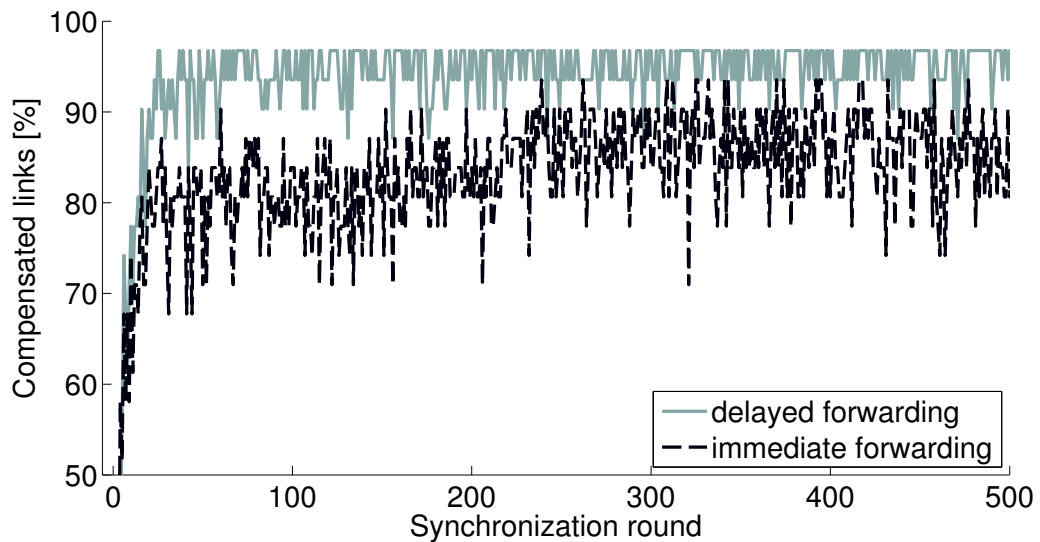


Figure 3.9: Percentage of compensated links during floods. *Both protocol variants quickly acquire delay measurements for relevant links, while delayed forwarding has more coverage and stabilizes at a level of 95 %.*

for one hour on all nodes in the testbed. The GPS node in the upper left corner in Figure 3.7 is used as reference node. We configure TATS to

have a synchronization period of 1 s. For every synchronization round, nodes report the number of message delays measured (as parent) and the number of delay measurements received (as child node). In addition, we count the number of unknown message delays when updating the global time in forwarded messages. From a regular link measurement on FLOCKLAB, we extract the number of available communication links to put our experiment into perspective. On average, we see 101 links between 31 nodes, 29.8 % are mostly unidirectional.

Results. Figure 3.8 shows the average number of estimated link delays per node, while Figure 3.9 presents the ratio of links that are compensated when running TATS. The latter uses a subset of all estimated link delays, i.e., those links that are part of the flooding tree. For both variants, measurements propagate quickly from parents to child nodes, which results in a very small difference in available delay estimates between parents and children. We find that delayed forwarding is beneficial and achieves 37 % more estimated links than immediate forwarding. Delayed forwarding leads to an increased coverage of links and also to less missing estimates while forwarding packets.

This experiment confirms the usefulness of delayed forwarding and shows that it is feasible to perform two-way delay measurements based on network flooding, even in the presence of unidirectional links.

3.5.3 Comparison to PulseSync and Glossy

In this experiment, we compare TATS against PulseSync and Glossy. To assess the synchronization accuracy, the *global synchronization error* is an important metric, i.e., the maximal pairwise difference between clock values of all node in the network. Technically this is not possible with our setup, as we would need a GPS receiver next to every node. A representative coverage of the network is attained by placing the GPS receivers evenly distributed. Instead of the global synchronization error, we measure the synchronization error relative to the reference node

$$\mathcal{G}_r = \max_{v \in V} (|t_r - t_v|). \quad (3.6)$$

Here, the set V contains all GPS nodes except the reference node r . The clock values t_r and t_v are timestamps of GPS pulses, converted to global time. For each test run, we report the average and the maximum synchronization error \mathcal{G}_r over the duration of the test.

Setup. We measure the accuracy in three different settings: a dynamically formed network and two different 22-hop line topologies. The dynamic network has a diameter of approximately 6 hops. Line topologies are enforced in software as shown in Figure 3.7: line topology *short* has a

Table 3.5: Accuracies measured for different protocols.

Setting	avg error	max error	PRR
TATS long	0.21 μ s	0.54 μ s	[0.96..1.00]
TATS short	0.23 μ s	0.46 μ s	[0.99..1.00]
TATS dynamic	0.24 μ s	0.54 μ s	[0.90..1.00]
PulseSync long	1.43 μ s	1.85 μ s	[0.98..1.00]
PulseSync short	0.93 μ s	1.31 μ s	[0.99..1.00]
PulseSync dynamic	0.75 μ s	1.23 μ s	[0.96..1.00]
Glossy dynamic	0.72 μ s	1.46 μ s	[1.00..1.00]

length of 182 m, while line topology *long* has a length of 283 m. This way, we evaluate the impact of different propagation delays and different network structures on TATS’s synchronization accuracy.

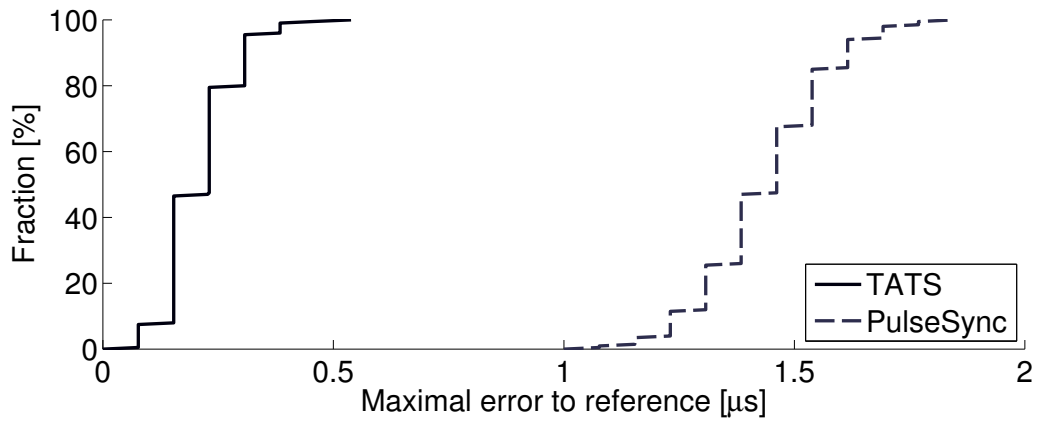
PulseSync is calibrated using a single message delay parameter [LSW14]. We averaged a total of 2014 measurements between two nodes to estimate this parameter. As we forward packets as fast as possible, we implement both PulseSync and TATS without drift compensation, as the effect of drift would be marginal. In case of larger clock drifts between nodes, caused e.g., by large temperature differences, drift could be compensated as described in [LSW14].

For a fair comparison, we perform the same linear regression as in TATS also for Glossy. As enforcing a real 22-hop line topology is not possible for concurrent transmissions in a setup with unrestricted communication, we compare Glossy only on the dynamic topology.

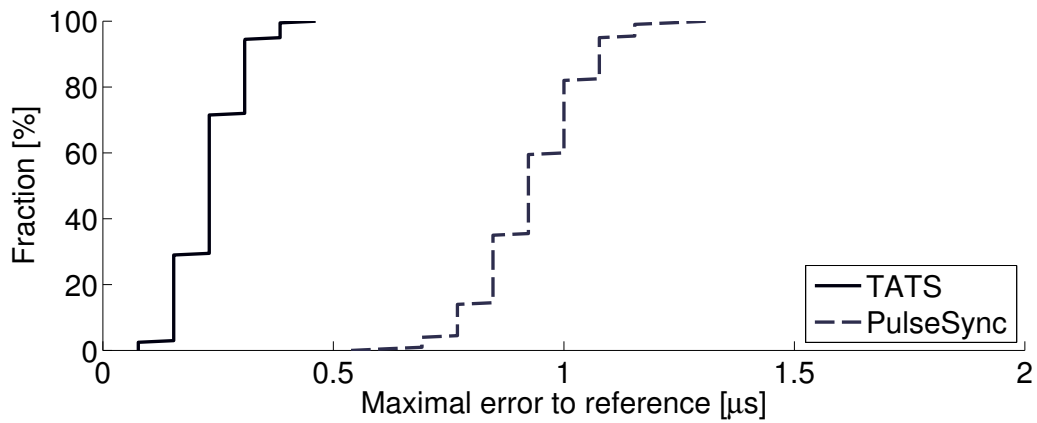
We configure all three protocols to use a synchronization interval of 1 s and a regression table of 80 samples. In total, seven different test runs are performed, each possible combination of protocol and topology once for a duration of one hour.

Results. Figure 3.10 shows the distribution of the maximal absolute time difference to the reference node over all synchronization rounds. The error distribution is stable for TATS on all three topologies, while PulseSync exhibits varying performance. In addition, the error is significantly higher than the one of TATS. Both effects can be attributed to the fact that a single point calibration of message delay cannot sufficiently represent the conditions in the whole network. Glossy exhibits a similar performance as PulseSync on the dynamic topology. Figure 3.11 shows the evolution over time. The error settles for all protocols after only a few synchronization rounds.

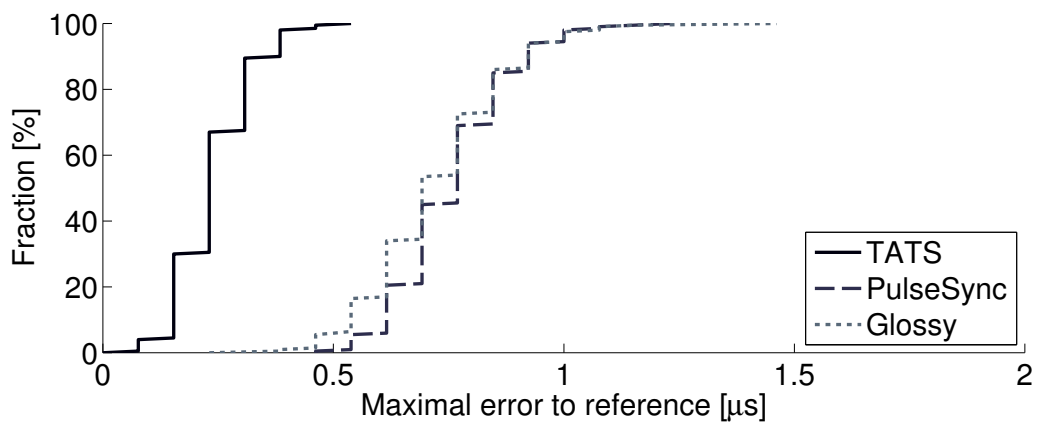
If we consider the average error of individual nodes in Figure 3.12, we



(a) Long topology

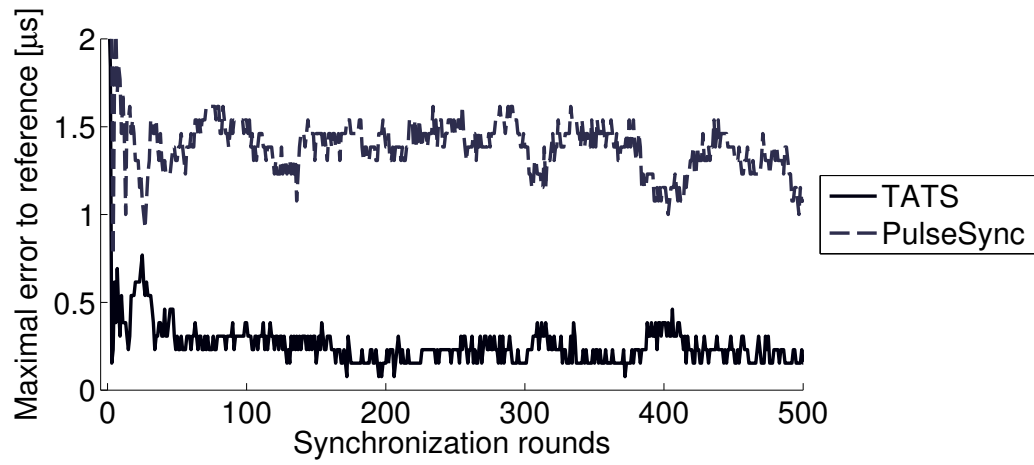


(b) Short topology

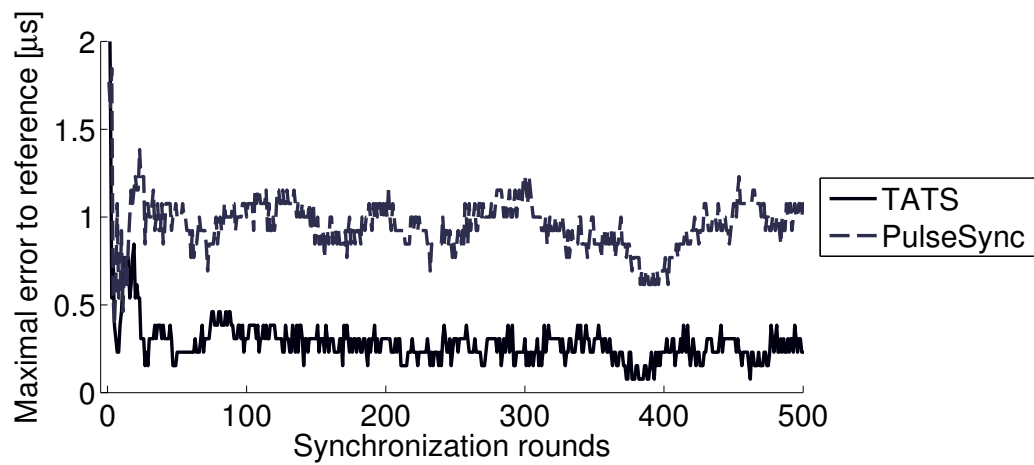


(c) Dynamic topology

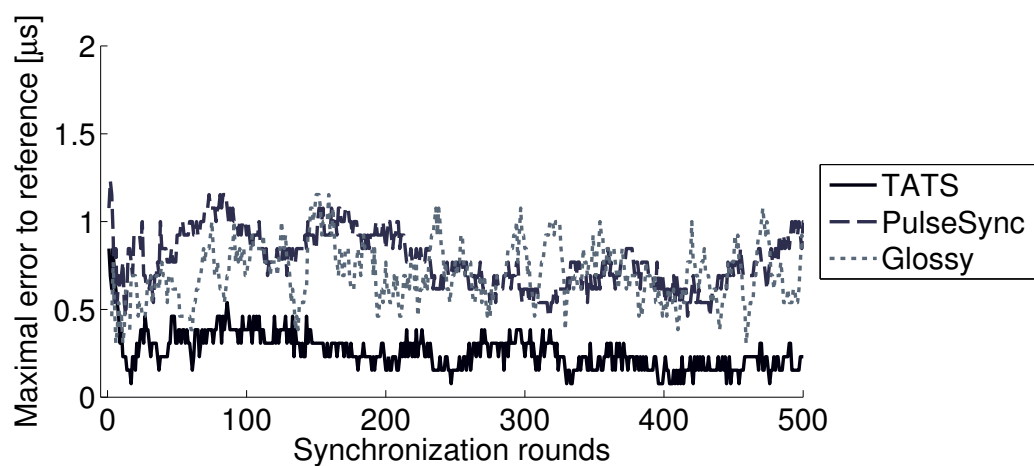
Figure 3.10: Cumulative distribution of synchronization errors, measured relative to the reference node. While PulseSync performs different on the three topologies, TATS can adapt and compensate for different propagation delays.



(a) Long topology



(b) Short topology



(c) Dynamic topology

Figure 3.11: Maximal synchronization error over time. For better visibility, only the first 500 rounds are shown.

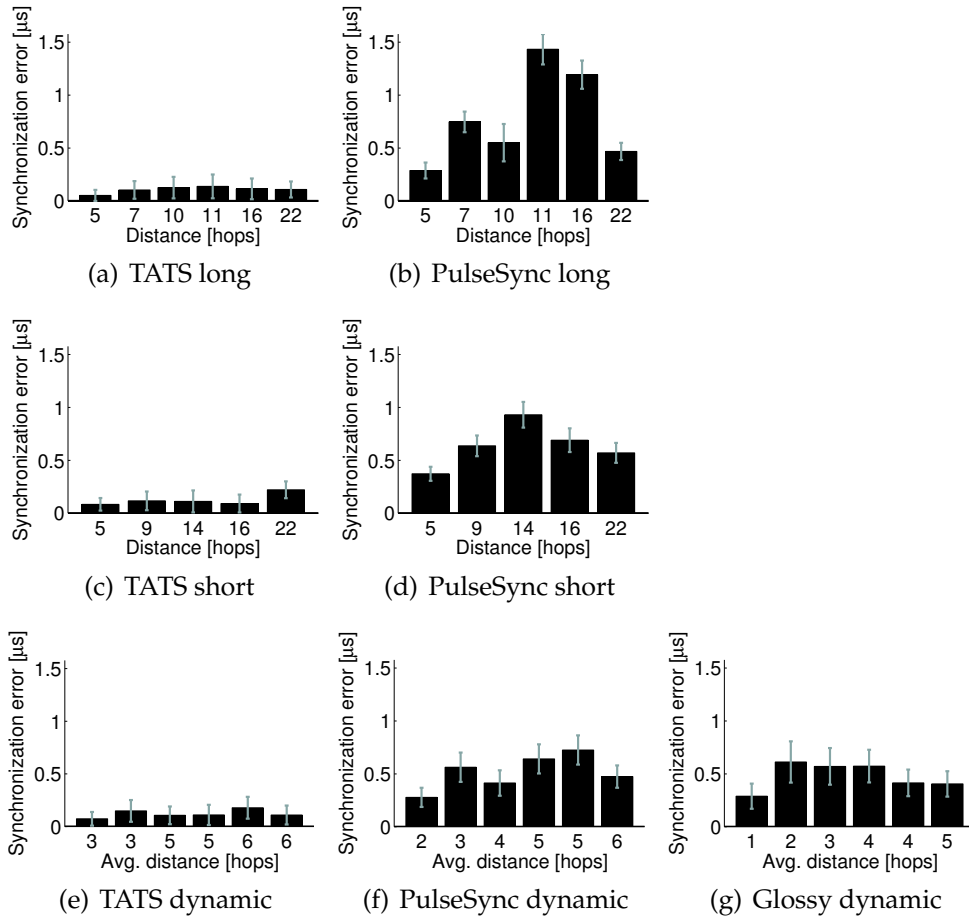


Figure 3.12: Average synchronization errors over time per node. Bars indicate standard deviation.

see that the error is evenly distributed for TATS on all topologies (a), (c), (e), while nodes running PulseSync are affected by more variance (b), (d), (f). The distribution for Glossy (g) is similar to PulseSync (f).

Key figures for all test runs are summarized in Table 3.5. TATS performs up to $6.9\times$ better than PulseSync with respect to average maximal synchronization error and $3\times$ better than Glossy and PulseSync on a dynamic topology with smaller diameter. In all settings, TATS's maximal synchronization error is below 1 microsecond. The lowest packet reception rate (PRR), i.e., the ratio between sent and received synchronization packets, is 0.9 over all test runs. Missing synchronization packets potentially lead to reduced accuracy, as less sampling points for the linear regression are available.

3.6 Summary

We have presented TATS, a new and highly precise time synchronization protocol for wireless embedded systems. TATS combines fast flooding and message delay compensation at similar message cost as existing protocols without delay compensation. Experiments on a testbed that resembles real deployment scenarios well with respect to node distances show that (i) TATS achieves up to $6.9\times$ better accuracy than state-of-the-art protocols, and (ii) can synchronize even networks with large diameters of up to 22 hops within sub-microsecond accuracy. This makes time synchronization using wireless sensor networks a viable option to wired or GPS-based high precision systems. More generally, TATS can be employed wherever precise clock synchronization is necessary. We explore in the next chapter the idea of using a wireless time synchronization protocol like TATS to date distributed events in FLOCKLAB.

4

Fine-Grained Tracing of Time Sensitive Behavior in Wireless Sensor Networks

While the currently available testbed services already cover many areas beyond the basic needs of wireless embedded application engineers, support still lacks for network-wide, fine-grained cycle-accurate event tracing to enable inspection of *time sensitive* system behavior. To illustrate this, we consider two examples:

1. Low-power MAC protocols require exact and coordinated timing of actions to ensure efficient operation, e.g., Glossy relies on constructive interference of concurrently transmitted packets, which requires transmissions of neighboring nodes to be aligned within $0.5 \mu s$ [FZTS11]. To observe and validate the interaction between different nodes in such a network, a distributed tracing mechanism must be minimally invasive and deliver the recorded trace of each individual node tightly time-synchronized with all other concurrent traces.
2. Control-flow tracing of programs allows for efficient debugging and to find potential failure causes [SEZ10]. In this case, every branch instruction in a program needs to be traced. The resulting enormous volume of tracing points necessitates an efficient and minimally intrusive trace recording system.

Existing testbed approaches are either too intrusive (e.g., using `printf` over a serial port), or not general enough to meet the diversity of available

node platforms (in system debugging [SK13],[THBR11]). More crucially, none of the available testbeds—including FLOCKLAB, as described in Chapter 2—is able to align traces with the required accuracy and cope with high peak event rates.

Contributions. To overcome the limitations of current testbeds, we introduce FLOCKDAQ, a new distributed data acquisition system that is capable of tracing mote application behavior at a high time resolution in a minimally invasive manner, tightly synchronized throughout all nodes in the testbed. The basis of our new system is the FLOCKLAB testbed architecture, which provides a distributed network of *observer* platforms that are used to stimulate and monitor the attached devices under test, the *targets*. We build on the idea of the existing GPIO tracing and actuation services, and extend their capabilities with respect to *sampling resolution, peak sampling rate, and time alignment of traces* by several orders of magnitude. By including short GPIO instructions into node applications, the program behavior can be traced in a minimally invasive manner. The design of the new data acquisition system consists of a field-programmable gate array (FPGA) and a CC430 SoC with RF core. The FPGA chip handles the timing sensitive data acquisition part, while the SoC is running a wireless time synchronization protocol to keep the system time of the FPGA chip on each observer synchronized. A testing environment to observe node interaction on a detailed level requires synchronization accuracy that rules out the commonly used network time protocol NTP [MMBK10]. For local networks, the precision time protocol PTP [ptp08] is a more accurate alternative, while GPS receivers provide accurate synchronization on a global scale. PTP requires special hardware support within the network infrastructure, while GPS receivers only provide accurate synchronization in places with good satellite reception. As we want to support both, indoor locations with possibly insufficient GPS satellite reception as well as outdoor locations with limited infrastructure support, we identify time synchronization using a low-power wireless multi-hop network, e.g., as described in Chapter 3, to be a viable solution. We integrate a synchronization algorithm based on Glossy [FZTS11] into FLOCKDAQ's data acquisition system, and show that the synchronization performance can be considerably improved by applying a jitter reduction filter.

Challenges. Depending on the node application, tracing an execution path with many conditional branches within a short time window requires an efficient data handling mechanism that can manage high peak data rates. To faithfully capture the interaction between different nodes in the network, measurements need to be accurately time-synchronized. The required combination of high peak sampling rate and accurate

time synchronization makes designing such a data acquisition system a challenging task.

Findings. Our system is designed to capture GPIO events with a time resolution of $0.1 \mu\text{s}$. According to the evaluation in Section 4.5, our prototype sustains a peak event rate of 10^8 events/s, while the maximal average event rate the system can handle is 2.85×10^5 events/s. Expressed in numbers of the TelosB node platform, this event rate allows to continuously trace a program of which one third of the instructions change GPIO states.¹ Typical low-power applications exhibit even less events to trace due to energy saving strategies that put the CPU to a sleep mode. Measurements in a 31-node network assisted by GPS precision timing show that FLOCKDAQ aligns concurrently recorded traces within $1 \mu\text{s}$ with an empirical probability of 99.9%.

In the following, we discuss related work in Section 4.1 and derive the requirements for a fine-grained, distributed trace recording system in Section 4.2. In Section 4.3, we give an overview of the system design and discuss the data acquisition system and the time synchronization mechanism in detail. We explain implementation specific details in Section 4.4, evaluate key properties of FLOCKDAQ in Section 4.5, and conclude the chapter in Section 4.6.

4.1 Related Work

Related to this chapter are hardware and software solutions that allow to trace embedded system behavior, both on a single entity and at network scale. In addition, we discuss time synchronization protocols for low-power wireless embedded systems and time synchronization using a 1-pulse-per-second (PPS) signal.

Tracing system behavior can be realized either on the target device itself, or using external hardware. Software solutions instrument program code at branch instructions and use efficient encoding to log control flow traces to flash memory [SEZ10]. Another instrumentation approach is pursued by Tardis [TSBE15], which rather logs non-deterministic program inputs as a trace for later replaying using a simulator. While software solutions are easily applicable and can provide very accurate information about the state of a node, the required resources on the target for processing and storing the traces render such an approach unsuitable to trace time sensitive behavior. Indeed, experiments with Tardis reveal that the CPU duty cycle of standard node applications can almost double when tracing is enabled [TSBE15].

¹Assuming a clock speed of 4 MHz.

Additional hardware can offload data processing from the target to an external observer platform, providing an out-of-band communication channel in addition. Two different data extraction methods are commonly applied in this context: On-chip debug interfaces or simple GPIO pins for binary state information. Aveksha [THBR11] uses a debug board extension to trace events of interest on a single target using the on-chip debug module of the MSP430 microcontroller. A low-cost and networked solution is provided by Minerva [SK13]. Tracing using on-chip debug interfaces is non-intrusive and expressive, but not easily portable between different microcontroller architectures. Monitoring GPIO pins is a more generic approach, at the expense of slightly higher intrusiveness caused by short GPIO instructions. GPIO pins can be traced at relatively high speeds, as shown in an FPGA-based logic analyzer design that is able to sample 8 GPIO pins on a single embedded system at a rate of 2×10^8 events/s [PPTDS10]. GPIO tracing in a distributed fashion is also a key element of FLOCKLAB. Different to FLOCKDAQ, the aforementioned single node monitoring solutions don't provide a consistent global view of a network. Available networked solutions are only conditionally suited for fine-grained distributed trace recordings due to their limited tracing rates and time synchronization accuracy, e.g., FLOCKLAB traces exhibit a maximal pairwise timing error of $255 \mu\text{s}$ and contain events at a maximal rate of 3.5 kHz for lossless traces. FLOCKDAQ is a distributed tracing solution that overcomes these limitations.

FLOCKDAQ implements a digital loop control algorithm to lock the FPGA-internal system clock to a PPS signal. Similar controllers have been used in the past, e.g., to reduce the jitter of a GPS PPS signal [GZF⁺07].

As further detailed in Section 4.2, measurements recorded with FLOCKDAQ have to be time-synchronized within $1 \mu\text{s}$, which we achieve by using a multi-hop time synchronization protocol. A popular time synchronization algorithm is employed by the flooding time synchronization protocol FTSP [MKSL04]. However, the achieved maximal synchronization error of less than $14 \mu\text{s}$ in a 6-hop network does not meet our requirements. Schmid et al. improve on the achieved accuracy of FTSP by introducing a high resolution clock [SDS10]. The Time-of-flight aware time synchronization protocol (TATS), as described in Chapter 3, has a maximal error of $0.54 \mu\text{s}$ over 22 hops. Glossy, a flooding architecture for wireless sensor networks that exploits constructive interference for fast network flooding, implicitly provides time synchronization [FZTS11]. On a TelosB, the reported average error over 8 hops is as low as $0.4 \mu\text{s}$, with a standard deviation of $4.8 \mu\text{s}$. In FLOCKDAQ, we port Glossy to a node platform that has two distinct properties that improve time synchronization: (i) as in [SDS10], a high

Table 4.1: Maximal event rates of different target platforms.

Node Platform	CPU clock frequency	Cycles per pin change	Peak event rate (events/s)
TelosB	4 MHz	5	0.8×10^6
Tinynode 184	12 MHz	5	2.4×10^6
IRIS	8 MHz	2	1.6×10^6
Opal	96 MHz	5	19.2×10^6

resolution system clock, and (ii) a radio chip with automatic RX/TX transceiver mode switching. Additionally to the baseline Glossy, we improve on the time synchronization variance by adding a jitter reduction filter. Although TATS provides better synchronization accuracy than Glossy, we decided in favor of Glossy due to its reliable and robust flooding architecture based on constructive interference.

4.2 Enabling Fine-Grained Tracing

In this section, we sketch the idea of tracing system behavior using GPIO pins as the monitoring interface, and we derive the requirements needed to actually enable fine-grained tracing of system behavior of low-power wireless sensor networks. For a more systematic approach to control flow tracing using GPIO pins, the reader is referred to Chapter 5.

We specify the system behavior of interest as the control path taken during a program execution on a set of nodes, annotated with time information. This information can be used in several ways to analyze code execution. Examples are (i) the quantification of code coverage for test applications, (ii) empirical determination of bounds for execution times of certain program parts, or (iii) the verification of system behavior against a given specification using exhaustive testing.

To get execution traces of programs, we instrument the program code using short *witness instructions* to emit pin state changes (*events*) at branches in the program flow. As these instructions are very short, we assume that these additional instructions only minimally affect the system behavior. We then reconstruct the taken program flow from the emitted sequence of pin level changes. This is feasible if the emitted GPIO trace is ordered by time and unambiguously mappable to a sequence of program executions. Ensuring unambiguous mapping by means of a limited number of pins possibly requires to encode individual witnesses using a sequence of pin changes.

What are the requirements for a trace recording system? The rate of the emitted GPIO level changes is limited by the maximal pin change rate of a target, which in turn depends on the target's MCU clock speed and the number of instructions needed to change a pin state as summarized in Table 4.1. This rate might be reached if programs have several conditional branches in a row, or if witnesses in the program code are encoded using sequences of pin changes. On the other hand, low-power wireless embedded systems are usually duty-cycled to save energy. There is no code execution during sleep states, and therefore the expected average tracing rate can be significantly lower than the peak event rate. These observations lead to the first requirement:

Requirement 1 A trace recorder needs to sustain for short periods of time a peak sampling rate that is able to capture the maximal pin level change rate of a target.

The time resolution of a trace should be sufficiently high to allow meaningful execution time measurements, thus our second requirement is:

Requirement 2 The trace of a single target must be ordered by time and exhibit a sub-microsecond time resolution.

To observe interactions between several nodes in a network, and to properly order these interactions relative to each other, time annotations within a trace must allow to sufficiently align it with traces of neighboring nodes. For instance, to measure and properly adjust timing properties like wake-up guard times of low-power MAC protocols, an alignment error in the range of the smallest controllable time quantity on a node would be preferable. Behavior is typically controlled by timers running from 32 kHz oscillators or a faster main system clock, which allows control at a resolution down to microseconds. This is also reflected in the timer architecture of TinyOS, which foresees time resolutions down to one microsecond [STG07]. We therefore phrase the last requirement as follows:

Requirement 3 The time synchronization error between observers of neighboring nodes should be in the microsecond range.

Next, we describe the architecture of FLOCKDAQ, and we explain how we address the given requirements.

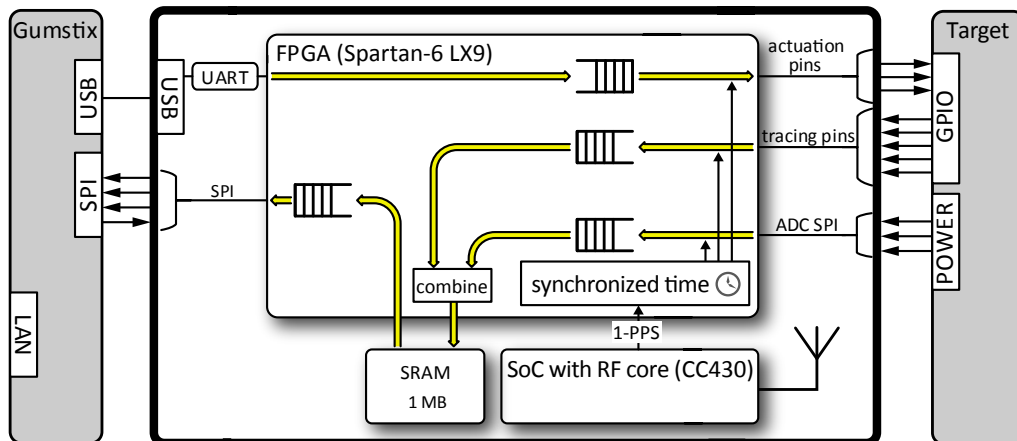


Figure 4.1: Overview of a single FLOCKDAQ observer.

4.3 Architecture

4.3.1 Overview

Figure 4.1 provides an overview on the system architecture of FLOCKDAQ. The newly designed system fits into the existing FLOCKLAB architecture and replaces the existing data acquisition part, which is originally running entirely on a Gumstix embedded computer.

The Gumstix features a 624 MHz Marvell XScale PXA270 microprocessor that runs OpenEmbedded Linux and is equipped with 128 MB SDRAM, 32 MB flash memory and an 8 GB SD card. All observers in FLOCKDAQ are connected over Ethernet or Wi-Fi (outdoor) to a backend infrastructure.

We trace a target in FLOCKDAQ by means of two different tracing interfaces, GPIO lines and an ADC, the former for digital state information, the later for power measurements. The GPIO interface is used in two directions, either controlled by the target or by the observer. The state of these interfaces are traced by the FLOCKDAQ board. Traces are annotated with a timestamp and forwarded to the Gumstix computer. We handle all time critical tasks on the FLOCKDAQ board, while test management and communication tasks are allocated to the Gumstix computer.

The FLOCKDAQ data acquisition architecture consists of a Spartan-6 FPGA chip, a static random-access memory (SRAM) and a CC430 SoC that combines an MSP430 microcontroller and a CC1101 radio transceiver. Functionally, the system has to process three different types of data streams: (i) GPIO actuation commands to control 3 GPIO pins, (ii) GPIO tracing on 5 pins and (iii) power profiling data. In total, 9 individual

streams need to be processed in parallel and with low time jitter. For this task, we employ an FPGA chip, which allows us to map the processing of each stream type to a dedicated hardware module. This design decision greatly facilitates deterministic, low jitter processing. In contrast to this paradigm, many existing testbed architectures, including FLOCKLAB, rely on sequential processing on a single processor.

We tackle the high peak sampling rate requirement by employing a hierarchical memory structure. Fast on-chip FIFO queues within the FPGA handle short bandwidth spikes, while SRAM memory is used to buffer larger amounts of sampled data, before we finally write the acquired traces to the serial peripheral interface (SPI) bus for further storage on an SD card on the Gumstix.

To put the measured information into a global time context, we need to keep the time on each observer synchronized with all the other observers. For this purpose, the internal time of the FPGA is disciplined by applying a PPS signal. The edge of such a pulse indicates the start of a new second. Typically, GPS receivers provide this kind of pulse for synchronization purposes. However, relying on GPS timing restricts the range of use to locations with good satellite reception, which rules out most indoor locations. To distribute a time pulse to all observers in the testbed, our design relies on a wireless time synchronization protocol based on Glossy [FZTS11], running on the SoC of the FLOCKDAQ board.

Next, we describe in Section 4.3.2 the clock control algorithm that keeps the internal FPGA-time locked to a PPS signal. Section 4.3.3 details the FPGA-design of our data acquisition system, Section 4.3.4 explains the configuration interface, and Section 4.3.5 describes how we accurately synchronize time on all observers in the testbed to a common reference.

4.3.2 Disciplined System Clock

In the following, we discuss time and clock related details of the FPGA design. The FPGA chip runs at a clock speed of 100 MHz. At the same time, this is also the maximally achievable sampling rate of GPIO states. To facilitate digital control, we derive a *system time* counter, running at a nominal frequency of 10 MHz, that is, a clock period of 100 ns. The speed of the system is adjustable by varying the number of FPGA clock cycles per system time clock period.

To keep the system time in line with the external reference PPS signal, we employ a clock control algorithm that combines an open-loop controller with a feedback control loop, as illustrated in Figure 4.2. The open-loop controller determines the actual local clock rate by averaging the number of FPGA clock cycles within N PPS periods, where $N = 8$ in our design. We denote the difference between this measured clock rate

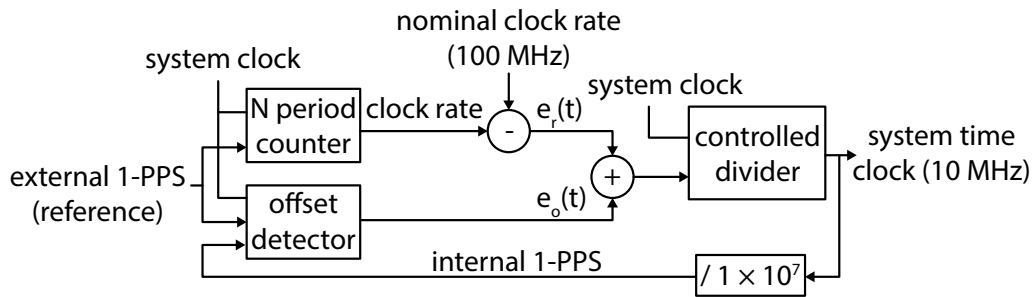


Figure 4.2: The clock control algorithm is a combination of an open-loop controller and a feedback control loop. The speed of the system time clock is controlled by a variable divider.

and the nominal clock rate as $e_r(t)$. The feedback control loop implements a P-controller, which corrects the offset of the system time relative to the external PPS signal. The offset $e_o(t)$ is measured using the offset detector in Figure 4.2. The sum of $e_r(t)$ and $e_o(t)$, expressed in system clock cycles, is given as input to a variable clock divider that generates the 2 MHz system time clock based on the 100 MHz FPGA system clock. The clock divider samples the system clock down by a variable factor $\beta \in \{9, 10, 11\}$. The factor is applied in a way, such that the total error is corrected evenly spread over the period of one second.

We evaluate this design in Section 4.5.2 and show that our controller keeps the internal system clock tightly synchronized to the PPS signal.

4.3.3 Data Acquisition

The data acquisition part of the FPGA-design captures the data generated by the target, that is, state changes of the target's GPIO lines and the power dissipation of the target, measured by the ADC. Each captured element has to be annotated with a precise timestamp and forwarded to the Gumstix computer.

The two data streams that need to be handled have different properties. GPIO state changes, which are triggered by single instructions on the target platform, occur at irregular time intervals and possibly exhibit high peak rates, e.g., an Opal node could potentially emit 19.2×10^6 state changes per second. Power measurements on the other hand follow a strict sampling interval and occupy less bandwidth, in our case up to 56 kbps. Our design takes these differences into consideration by applying different internal data representations. To maximize throughput, each acquired data element should occupy as little memory as possible. Space can be saved by not including the complete timestamp into each data

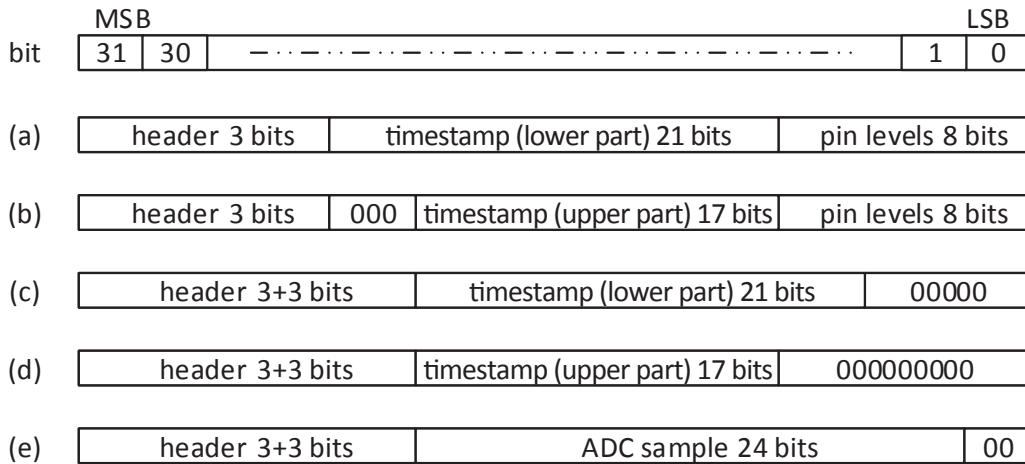


Figure 4.3: Data format of GPIO tracing and power profiling packets. All packets have a width of 32 bits and include a header of 3 bits. *Marker packets* (b) and (d) contain the upper 17 bits of the timestamp.

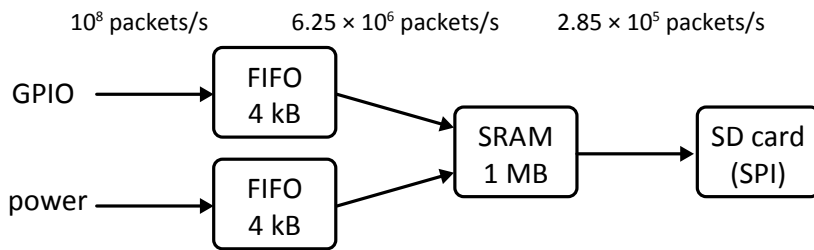


Figure 4.4: Data flow and memory structure of the data acquisition system. The SRAM serves as buffer for the SPI bus. The two data streams are merged when writing to the SRAM.

element, but rather stripping the most significant bits from the time value. As depicted in Figure 4.3, we introduce *marker elements* (b) and (d) into the data stream to mark the change of the significant bits in the stream. The complete time can be reconstructed in a later step, e.g., on a back-end server. For power profiling samples, the knowledge of the sampling rate even eliminates the need for storing the lower part of the timestamp, i.e., the essential data consists of an ADC sample and a header, as in (e) in Figure 4.3. Packet format (c) is needed to indicate the start and end of a power profiling trace, which can happen at arbitrary time instants. While our approach generates a low-rate base stream of metadata, it also greatly reduces the data volume for event bursts and power profiling.

The memory structure matches the input data stream (i.e., GPIO events and the power profiling), to the output, in our case an SPI bus running at a clock frequency of 12 MHz. As illustrated in Figure 4.4, the output

has a significantly lower peak bandwidth than the input. Therefore we resort to a hierarchical memory structure that provides both, high short term bandwidth on the input and enough buffer space to shape the data stream to match the slower output bandwidth. Small and fast FIFO queues can handle new data packets with every clock cycle, while the access time to the SRAM amounts to 16 cycles, i.e., a maximal packet rate of 6.25×10^6 packets/s.

The two input streams are merged before writing to the SRAM chip. To avoid starvation, power profiling data is prioritized because there is a tight and not saturating upper bound to the maximal data rate of this stream, which is less than 1 % of the SRAM memory bus bandwidth.

We empirically evaluate the throughput of the data acquisition with the help of an event generator in Section 4.5.1.

4.3.4 Configuration and Test Management

The FLOCKDAQ board can be configured over a UART interface, which is connected to the USB port of the Gumstix using a USB-to-serial converter. The choice of a dedicated configuration interface (in addition to the SPI bus) facilitates the software implementation on the Gumstix, as data acquisition and configuration can be handled independently. To configure the FPGA, commands are provided to start or stop a test, to set a mask for target pins to be traced, and to control power profiling. During a test, commands can be sent to set or clear 3 actuation pins on the target. These commands are kept in a FIFO queue on the FPGA and processed at the specified time instant. In order to ensure proper actuation timing, an actuation command has to be sent at least $70 \mu\text{s}$ in advance.

4.3.5 Time Synchronization for Distributed PPS

As described in Section 4.3.2, the internal system time on the FPGA is steered by applying an external PPS signal. In FLOCKDAQ, we leverage the fact that observers are placed within communication range of low-power wireless transceivers. Therefore we can employ a wireless mesh network built of such transceiver nodes to generate a synchronized PPS signal on all observers. In FLOCKDAQ, we use the CC1101 transceiver of the CC430 SoC for that purpose. A PPS signal of a single GPS receiver serves as reference for the initiator node in the network. The remaining nodes synchronize to the initiator by means of a time synchronization protocol.²

²Potential in-band interference with target nodes can be avoided by black-listing the frequency band/channel of the time synchronization protocol.

Glossy [FZTS11] is a flooding architecture for wireless sensor networks that exploits constructive interference for fast network flooding and implicit time synchronization. On a TelosB node it achieves an average time synchronization error below one microsecond and is therefore suitable for our purpose. In Glossy, every node estimates the start time of a flood, based on timestamps made from several packets within the flood. This estimate serves as a *reference time*. Due to redundant use of links and retransmissions, Glossy achieves a high reliability.

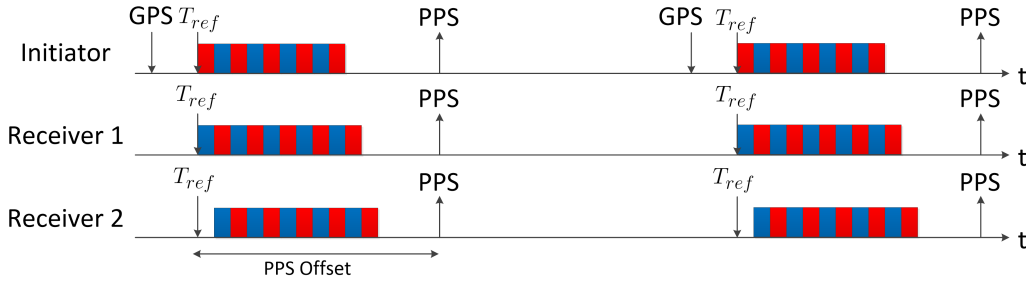


Figure 4.5: Generation of a synchronized PPS signal. An external GPS pulse triggers the start of a flood at the initiator. The PPS pulse is then emitted on all the nodes in the network based on the calculated reference time T_{ref} . This process is repeated every second.

Time synchronization in FLOCKDAQ is illustrated in Figure 4.5. The initiator starts a flood with every GPS pulse, i.e., there is flood happening every second. We configure the GPS receiver to emit the pulse slightly *before* the start of a new second in order to align the node generated pulses *with* the start of a second.

Due to different influences like measurement uncertainties or different propagation paths, the calculated reference time is affected by jitter. To reduce this jitter, we apply a heuristic that exploits the fact that the local clock, running from a quartz oscillator, is relatively stable during shorter periods of time. The intuition is to combine every new reference time $T_{ref,i}$ with the measurement $\hat{T}_{ref,i-1}$ from the previous flood and weight them according to some smoothing factor α :

$$\hat{T}_{ref,i} = \alpha T_{ref,i} + (1 - \alpha)(\hat{T}_{ref,i-1} + \bar{T}) \quad (4.1)$$

\bar{T} is the average interval between the last recent M reference times.

We evaluate the performance of the distributed synchronization pulse in Section 4.5.2 and quantify the impact of our heuristic.

4.4 Implementation

The hardware implementation of a FLOCKDAQ board is shown in Figure 4.6. It fits between a FLOCKLAB board and a Gumstix and therefore extends an existing FLOCKLAB observer in a modular fashion.

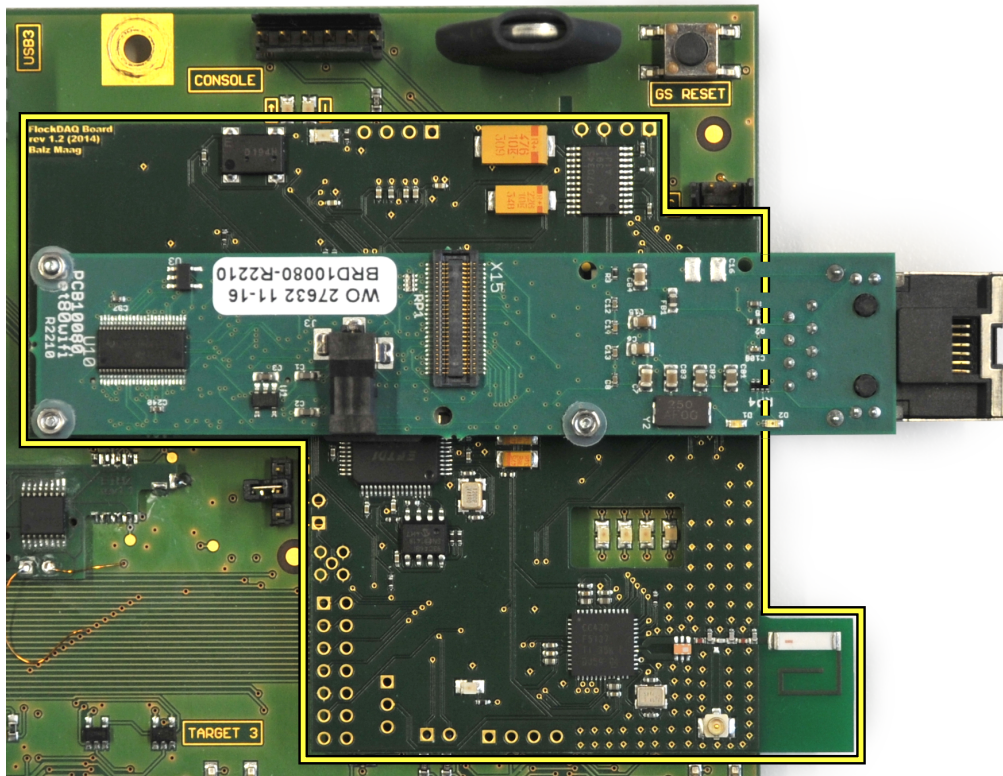


Figure 4.6: The FLOCKDAQ board fits between the FLOCKLAB board and the Gumstix. Visible on the lower right part is the CC430 with a chip antenna. The FPGA and the SRAM chip are on the bottom layer.

We implement the data acquisition part on an FPGA of the Xilinx Spartan-6 series. The design fits into a Spartan-6 LX9, which is second smallest member of that family, featuring 120 user I/Os and 9152 logic cells. The network time synchronization protocol runs on a Texas Instrument CC430F5137 SoC, featuring 32 kB of program memory and 4 kB of RAM. The chip integrates a sub-1 GHz radio with configurable bit rate and radio modulation. A 26 MHz quartz oscillator provides the basis of a stable 13 MHz system and timer clock. A 4-port USB-to-serial converter connects the debug and programming ports of the FPGA and the CC430 with the Gumstix computer.

On the CC430, we run Glossy on top of the Contiki OS [con]. As suggested by [FZTS11], we exploit the automatic RX/TX-transceiver mode

switch of the CC430 for accurate timing of concurrent transmissions in Glossy. Packets are sent using 250 kbps GFSK radio modulation in the 868 MHz frequency band.

4.5 Evaluation

In a first part of the evaluation, we focus on a single observer node of FLOCKDAQ. We measure peak and average throughput of the data acquisition system and assess the performance of the PPS-tracking algorithm on the FPGA. Next, we quantify the time synchronization error of the distributed time pulse for a setup of 31 observers in an office environment, using GPS receivers as ground truth. By running experiments with and without our jitter reduction algorithm, we show the beneficial impact of the algorithm. Finally, we assess the overall timing accuracy of FLOCKDAQ by using the wirelessly distributed time pulse as input to the PPS port of the FPGA.

4.5.1 Throughput

In this section, we quantify the throughput of the data acquisition system and compare it to the requirements given in Section 4.2.

As described in Section 4.3.3, the internal data path on the FPGA consists of several stages with different bandwidths. Here, we characterize the two maximal event rates that lead to a saturation of the first and second stage shown in Figure 4.4, that is, the FIFO queues and the SRAM. We empirically measure the number of events that can be processed without loss at event rates of 10^8 events/s and 6.25×10^6 events/s respectively. For this purpose, we connect an event generator to the tracing inputs of the FPGA and let the pin levels change at a constant rate. To detect the first lost packet, we compare received packets at the Gumstix with the generated events.

The results of this experiment are summarized in Table 4.2. The FIFO queue, which can store up to 1024 event packets, is saturated after 1070 events at a constant event rate of 10^8 events/s. While filling the FIFO, data packets are continuously removed and written to the SRAM. For the second stage, the SRAM, we generate events at a rate of 6.25×10^6 events/s. 270,000 data packets can be stored until the first packet is dropped. The SRAM is full within 40 ms. The maximal average data rate that the data acquisition system can handle is determined by the SPI bus, which is the slowest interface in the data path.

With a continuous throughput of 285,000 packets per second, FLOCKDAQ is able to trace programs with 1.48 % tracing instructions on

all target platforms in FLOCKLAB (see Table 4.1). The peak processing throughput of 10^8 events/s is high enough to meet the peak event rates of all available target platforms.

Table 4.2: Measured throughput burst sizes and maximum continuous rate.

Cycles between two events	Max. event burst size
1 (10 ns)	1070
16 (160 ns)	270000
350 (350 ns)	continuous

4.5.2 Timing

In this section, we first assess the performance of the clock control algorithm on the FPGA. Then, we proceed to the evaluation of the distributed PPS signal in a network of 31 observers and finally, we quantify the overall system performance of FLOCKDAQ in terms of time synchronization error between observers. In the experiments, we use one or several u-blox LEA-6T GPS receivers that provide an accurate PPS signal (RMS of 30 ns) [u-b14], either as a reference signal for the root node, or as ground truth.

Timing on a Single Observer. As described in Section 4.3.2, the implemented clock control algorithm on the FPGA seeks to correct the offset between the internal and the external PPS signal. The external signal is provided by a GPS receiver. To evaluate the performance of the algorithm, we measure the time difference between those two signals in system ticks (i.e., 10 ns) on a single FLOCKDAQ observer for a period of 5.5 hours.

The cumulative distribution function over all measurements is shown in Figure 4.7. In total, 19,713 offset measurements are made, with a 99th percentile of 40 ns, which corresponds to 4 FPGA clock ticks. Therefore, we conclude that our control algorithm keeps the system time on the FPGA within tight bounds if a proper external PPS signal is applied.

Network-Wide Time Synchronization. Next, we assess the accuracy of the distributed time pulse in a network setting of 31 nodes. This section focuses solely on the implementation on the CC430 SoC. The experiment is carried out on Olimex’s commercially available MSP430-CCRF development board. In total, we distribute 31 nodes as shown on the floor plan in Figure 4.8. 4 nodes are located outdoors while the remaining 27 are placed indoors in an office environment. We select

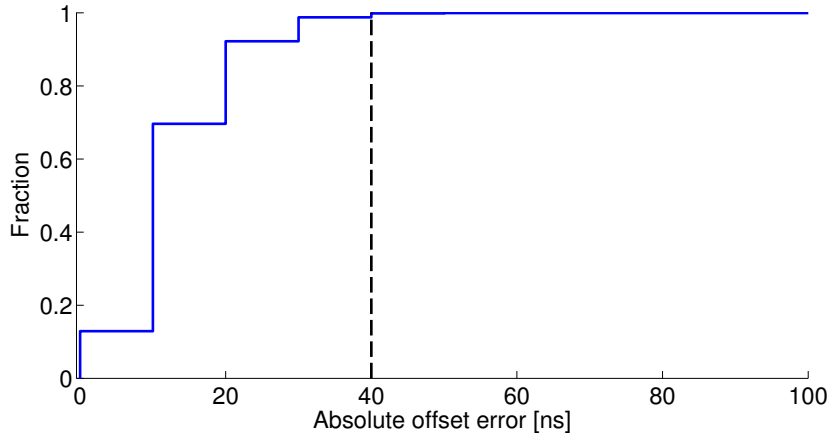


Figure 4.7: Cumulative distribution function of the absolute offset error between the internal and external PPS signal. The clock tracking algorithm keeps the offset error within ± 40 ns for 99 % of the time.

Table 4.3: Standard deviation and range of the error, measured in clock ticks (13 MHz), for all nodes, without and with jitter reduction heuristic.

Node	Glossy	Glossy with jitter reduction	Hop distance
1	4.05, [-25,82]	1.88, [-6,14]	4
2	2.15, [-11,17]	2.01, [-7,8]	2
3	1.94, [-7,7]	1.71, [-8,8]	1
4	3.25, [-23,23]	1.82, [-6,9]	2
5	3.62, [-22,32]	2.02, [-8,9]	4

a central node next to a window as initiator to keep hop distances short and to ensure a good satellite signal for the reference GPS receiver. On the nodes, we run two different versions of Glossy: a baseline implementation *without* jitter reduction, and a version *with* jitter reduction, as described in Section 4.3.5. We set the smoothing factor α to 0.1.

To assess the synchronization error, we equip 5 nodes with additional GPS receivers. On these nodes, the GPS PPS signal serves as ground truth. On every node, we locally measure for every Glossy flood the offset between the calculated reference time and the edge of the externally applied PPS signal. We then compare all offsets relative to the initiator node to get the synchronization error. To ensure a broad coverage of environmental conditions, such as closed office doors, working people and temperature variations, we combine measurements originating from various daytimes and weekdays into a total measurement duration of 6 hours. The results, summarized in Table 4.3, show that we are able to keep the standard deviation of the synchronization error below 5

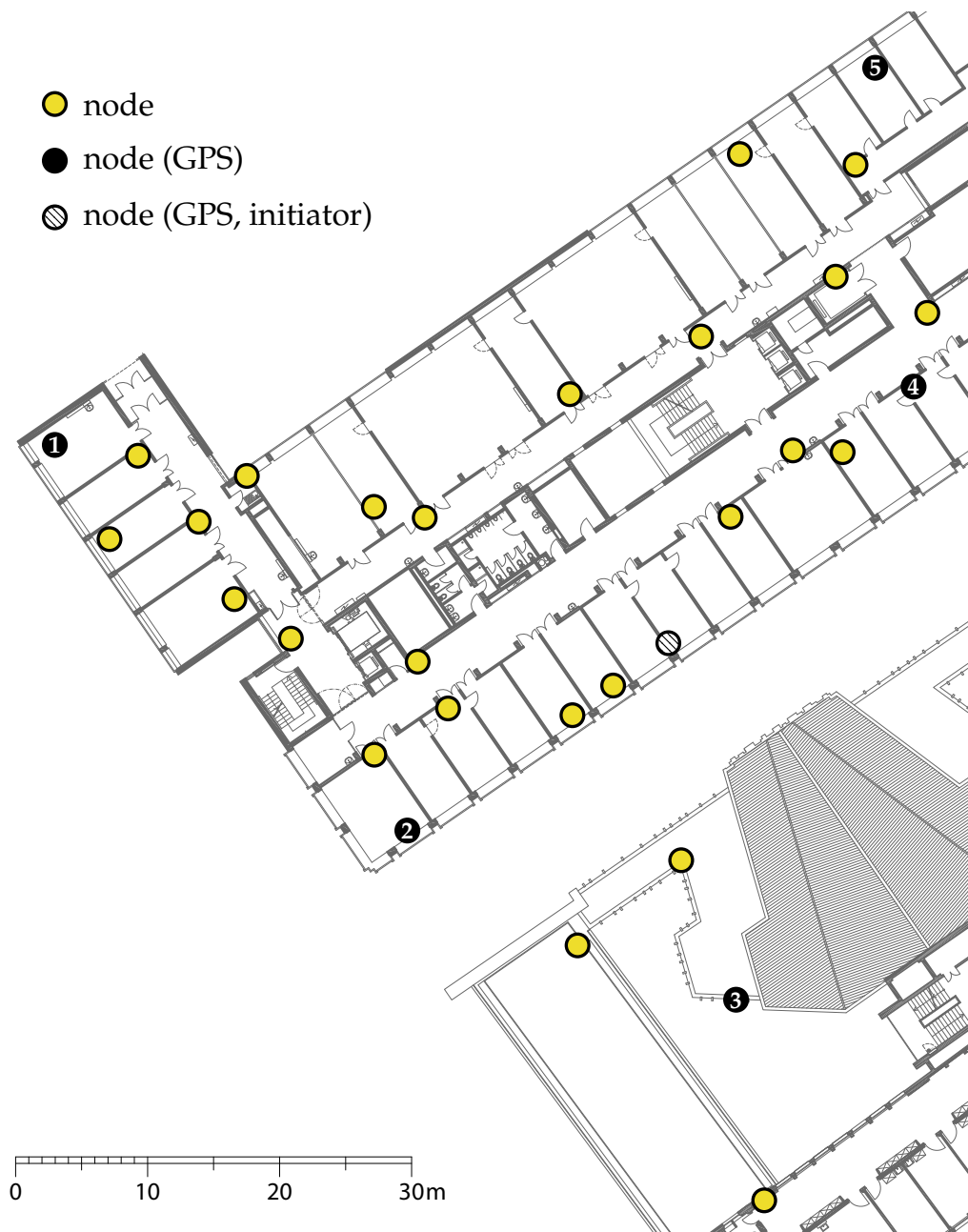


Figure 4.8: Layout of the network during evaluation of time synchronization.

clock ticks (385 ns) in the baseline implementation. Nodes that have a higher hop distance to the initiator exhibit a larger error. The outdoor node 3 has mostly a direct connection to the initiator and therefore the smallest error. The experimental data also shows a clear benefit of the jitter reduction heuristic. The maximal error as well as the standard deviation is considerably lower when jitter reduction is enabled. This effect is more prominent for nodes that are farther away from the initiator node. The

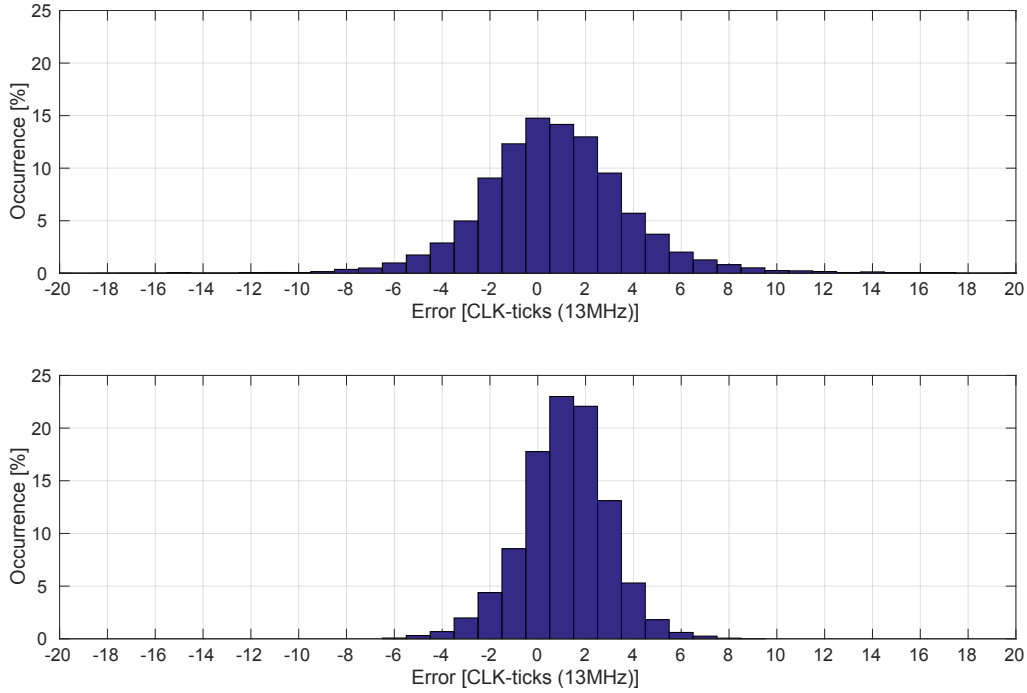


Figure 4.9: Distribution of synchronization error on node 4. The error distribution for the baseline implementation (top) is significantly broader than for the jitter reduced version (bottom).

error distributions for both implementations are exemplary illustrated for node 4 in Figure 4.9.

Based on our results, we conclude that our wireless PPS distribution infrastructure is well suited to synchronize observers in FLOCKDAQ with sub-microsecond timing error.

Overall Timing Accuracy of FLOCKDAQ. In the last experiment, we combine both the FPGA-design and the distributed time pulse to accurately trace GPIO events. The observers are placed in the same layout as in the previous experiment. Again, we equip the root observer and 5 other observers with a GPS receiver. The CC430 SoC on the root observer uses the PPS signal of the GPS as reference and distributes the pulse to all other observers using Glossy with jitter reduction. On all the GPS-equipped observers, we connect the PPS signal of the GPS to one of the tracing inputs of the FLOCKDAQ board. Then, we configure the data acquisition system to trace this pin for a duration of 1 hour. The data acquisition on the FPGA annotates every state change on the input pin using a globally synchronized timestamp. The difference between timestamps of different observers directly reflects the synchronization error. To evaluate the synchronization accuracy, we calculate for every

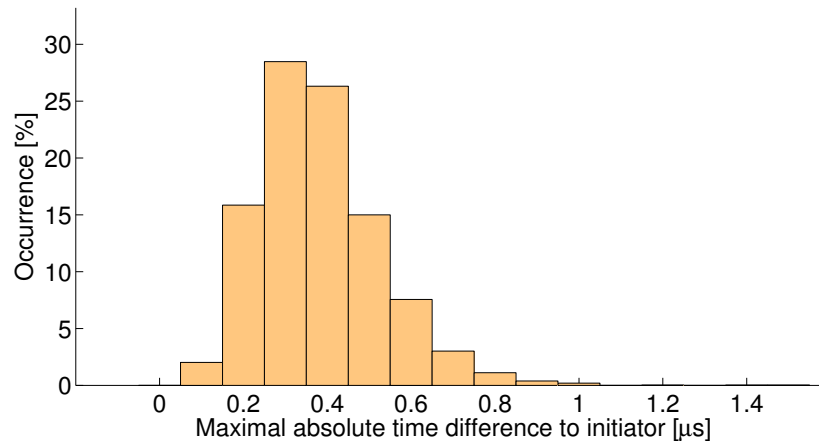


Figure 4.10: Absolute time synchronization error of FLOCKDAQ. The histogram shows the maximal error of 5 out of 31 observers, relative to the root observer.

pulse the maximal absolute error relative to the root observer. Figure 4.10 shows a histogram of the error. 99.9% of the errors are smaller or equal to $1 \mu\text{s}$; the maximal error is $1.5 \mu\text{s}$.

The reported error reflects the synchronization error between nodes on the edge of the network and the central root node. As the synchronization error in Glossy depends on hop distances [FZTS11], we expect synchronization between neighboring nodes to be even better. Overall, FLOCKDAQ provides distributed event tracing services with tightly synchronized time information.

4.6 Summary

We have presented FLOCKDAQ, a data acquisition system that allows to trace time sensitive system behavior of low-power wireless embedded systems in a fine-grained manner. By extending the FLOCKLAB architecture, as described in Chapter 2, with an accelerated data acquisition system based on an FPGA chip, FLOCKDAQ is able to capture state changes at the maximal rate emitted by any of the currently attached target platforms. FLOCKDAQ synchronizes traced data using a highly accurate wireless time synchronization protocol, thus enabling accurate monitoring of network interaction between all target nodes of the testbed, down to microsecond granularity.

5

Testbed Assisted Control Flow Tracing for Wireless Embedded Systems

While there have been many successful deployments of wireless embedded systems over the last ten years, building them is still a difficult task. There are several reasons for this: (i) the distributed nature and unreliable communication channels of wireless embedded systems makes it difficult to build precise models. (ii) Nodes do have a scarce energy budget, since batteries are heavy and expensive, and long periods of unattended lifetime are a prerequisite. Keeping costs and energy requirements low leads to hardware platforms that offer just enough memory and compute power for the task at hand [HNL08]. Due to the limited resources, systems regularly operate on the limits of the available computing, energy and communication capabilities. Therefore, the probability of misbehavior in terms of functional and non-functional properties is high. Besides careful design approaches, it appears that extensive testing and debugging is a major part of a successful design strategy [Woe10, OW10]. At the same time, methods to increase observability and controllability of executed programs have to cope with very little resources.

Methods applied for debugging software on a single node range from simple LED observations, over `printf` statements to in-system debuggers. Testbed infrastructure extends the observability of program execution to an entire network. All these debugging methods can give insight into particular parts of the running program, but lack the ability to

Table 5.1: Recent node platforms. The Cortex-M ETM module allows to extract program flow traces.

Name	Year	Architecture	HW tracing
OpenMote [VTWP15]	2015	Cortex-M3	-
panStamp NRG 2 [pan]	2015	MSP430	-
WandStem [TLF16]	2016	Cortex-M3	ETM
OpenMote+ [TVW16]	2016	Cortex-M4	ETM
Storm [AFC16]	2016	Cortex-M4	-

accurately trace the entire control flow of a program. As such, a program has to be (re-)instrumented every time for a specific goal. This is even true for in-system debuggers, since debugging interface bandwidths limit the extractable runtime information [SK13, THBR11].

The use of program flow tracing is not limited to debugging and failure diagnosis, but it is also applicable to program optimization or to collect software metrics like coverage [TH02, PY99]. In the area of general purpose computing, software-only methods exist to completely trace program executions [Lar99]. However, these methods have shown to be prohibitive when applied to wireless embedded systems [SEZ10]. Therefore, related approaches only trace a subset of the program, or instrument at a higher abstraction level, e.g., function calls [LST15]. Some microcontrollers include dedicated program flow tracing hardware inside the chip. The embedded trace macrocell (ETM) in selected ARM chips allows to extract a data stream of executed instructions. However, on-chip hardware debugging functionality comes at an additional cost in terms of die size and pin count [Fur00]. The majority of recent node platforms, exemplified in Table 5.1, does not support hardware assisted tracing. In the real-time domain, approaches exist to measure execution times for worst case execution time analysis by tracing GPIO lines [BMB10]. These solutions typically target more powerful processors, e.g., MIPS or PowerPC.

Contributions and road-map. In this chapter, we propose a novel hardware/software program flow tracing method that can be applied to trace the full program execution on instruction level in a pre-deployment testbed environment. Similar as [BMB10], our approach relies on an external monitoring device capable of observing GPIO state changes, e.g., a logic analyzer. FLOCKLAB (with FLOCKDAQ from Chapter 4) and testbeds with similar monitoring capabilities for GPIO states [BVJ⁺10, HSL10, PBMS14] allow to apply our approach to a large

number of nodes. In contrast to architecture specific hardware debugging facilities, our approach only requires some spare GPIO pins, which makes it applicable to virtually any node platform.

Inserting program statements for the purpose of tracing adds runtime overhead, i.e., additional CPU cycles. This overhead influences the program behavior and should therefore be minimal. We use information about execution time to reduce the runtime overhead of existing instrumentation approaches substantially. Timing information is extracted from the executable by means of an elaborate static analysis. We present an algorithm that reduces the number of recorded events while still being able to uniquely determine the executed program path.

In summary, this chapter makes the following contributions:

1. We design a method to trace program flow down to the instruction level using GPIO pin recording.
2. In Section 5.2, we present a new algorithm to reduce the number of emitted GPIO changes for tracing by exploiting time information.
3. In Section 5.3, we apply this algorithm to build a tool for MSP430 based platforms. It performs a static analysis of the program binary, adds instrumentation code and reconstructs traces from recorded GPIO events.

We experimentally show the influence of our new approach on different TinyOS and ContikiOS applications in a testbed of 31 nodes. The evaluation in Section 5.4 shows that our method adds an average runtime overhead of 19%. The use of time information reduces the runtime overhead by up to 38.3%. In addition, we find that instrumentation has no measurable influence on the reliability of Glossy [FZTS11], a timing sensitive flooding architecture that relies on constructive radio interference. Finally, in Section 5.5, we exemplify the usefulness of control flow tracing in two case studies.

5.1 Related Work

Tracing Program Execution. The problem of efficiently tracing and profiling program executions has been studied for several decades in the area of general purpose computing. One aspect of this problem is the question of where to best put witnesses (instrumentation code) in order to faithfully reconstruct the program flow [BL94]. Succeeding work found an efficient encoding of consecutive witnesses in program paths [BL96] and in whole programs [Lar99]. We build on findings in [BL94] and

extend these methods to make use of time information available when tracing embedded systems with an external observer. Since we can rely on external processing, our focus is to reduce the impact on the target system rather than to efficiently encode and compress traces. The aforementioned techniques cannot be directly applied to wireless embedded systems because of the scarce resources available on these devices.

Tracing program flow by monitoring GPIO lines has been done in the area of real-time systems [BMB10, WEE⁺08]. In particular, the pWCET tool instruments source code and traces program flow using this method [BCP03]. However, the aim of these approaches is to *measure* the execution time of a program. In contrast, our method is based on a timing model of the processor and *uses* execution time as a means to substantially reduce the induced tracing overhead.

Wireless embedded systems. Software solutions for tracing wireless embedded systems instrument program code with logging instructions and store the generated trace in flash memory or transmit it over radio or serial communication interfaces. In [SEZ10], instrumentation code is inserted at branch instructions and an efficient encoding is used to log control flow traces to flash memory. Similarly, in [WC13] the authors instrument each basic block of the program and use time information to compress the generated trace. In contrast to [WC13], our solution avoids putting witnesses on every single basic block, and we can also handle nested loops where the exact number of iterations is unknown at compile time.

Another instrumentation approach is pursued by Tardis [TSBE15], which rather logs non-deterministic program inputs. These inputs are then fed to a simulator when replaying the program execution. A combination of control flow and data tracing is employed by LibReplay [LST15]. By logging function call arguments, program execution at function level is logged for replay.

While software solutions can provide very accurate information about the state of a node, the resources required for processing and storing the traces render such an approach unsuitable to trace time sensitive behavior. Indeed, experiments with Tardis reveal that the CPU duty cycle of standard node applications can almost double when tracing is enabled [TSBE15]. In addition, software instrumentation potentially produces data streams that easily exceed the data produced by the application itself, rendering instrumentation of the whole program prohibitive if the trace needs to be stored on the node itself.

Hardware Assisted Tracing. Additional hardware offloads data processing from the node to an external observer platform, providing an out-of-band communication channel. Aveksha [THBR11] uses a debug

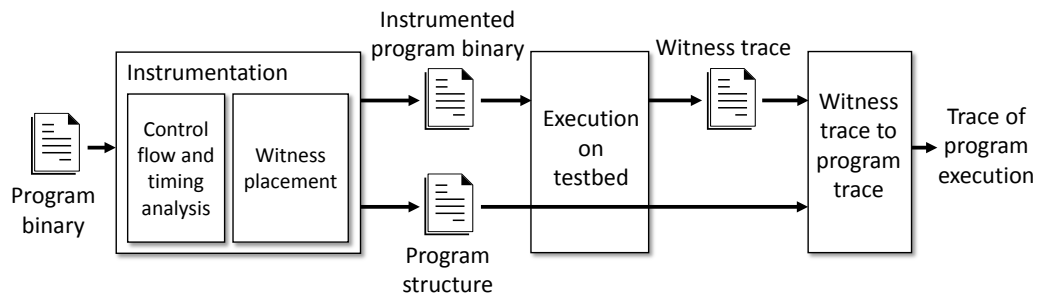


Figure 5.1: Overview of the tracing process.

board extension to trace events of interest on a single target node using the on-chip debug module of the MSP430 microcontroller. A low-cost and networked solution is provided by Minerva [SK13]. On-chip debug modules typically provide functions handy for interactive debugging (watch points, break points, reading and writing state information). Tracing the program flow by other means than polling the program counter is usually not possible. Reconstruction of the entire program flow might be possible if the polling interval is short enough. Recent processors include a special hardware tracing module to directly output trace information of the running program [ARM11, Int13]. However, typical sensor node platforms include lower end microcontrollers that do not include such features. Moreover, hardware debugging features are highly architecture dependent and therefore limited to specific node platforms.

Compared to existing hardware based tracing solutions, our approach is more portable and generally applicable since it only requires a few spare GPIO pins of a microcontroller instead of an architecture specific debug module or tracing module.

5.2 Control Flow Tracing

An overview of our approach to control flow tracing is given in Figure 5.1. We first statically instrument a program: the program binary is analyzed and witnesses are inserted at suitable locations. A witness is emitted whenever the control flow of the program passes its location. The instrumented binary is then loaded onto a set of real nodes. When the code is executed, we record the emitted witnesses. By combining the trace of witnesses with the extracted program structure, a trace of the program itself is reconstructed. In our case, witnesses are encoded into GPIO state changes and recorded using an external monitoring device.

Such an instrumentation has to fulfill the following requirements: (i) the emitted witness stream must be unambiguously mappable to the original program execution, and (ii) the runtime overhead due to added instructions must be kept to a minimum in order to preserve the original behavior of the program.

In the following, we first introduce in Section 5.2.1 the witness placement algorithm by Ball and Larus, which serves as baseline. Then, in Section 5.2.2, we improve the baseline instrumentation approach by making use of timestamps taken by the recording device. Finally, in Section 5.2.3 we show how we efficiently encode witnesses using a limited number of GPIO pins.

5.2.1 Ball and Larus

The algorithm by Ball and Larus [BL94] efficiently places witnesses onto edges in a given control flow graph $G = (V, E, W)$. Every procedure (e.g., function, interrupt handler) in a program is represented by a separate control flow graph. Basic blocks (groups of uninterrupted sequences of instructions) are associated with vertices $v \in V$, and directed edges $e \in E$ are transitions between basic blocks. Edges have annotated weights $w \in W$, representing the expected number of times each transition is taken during the execution of the program. Weights w can be obtained by profiling or by using a heuristic approach.

The algorithm optimizes the number of witnesses met during program execution, i.e., minimizing the total weight of all instrumented edges. Since finding the minimal-cost solution is NP complete [BL94], Ball and Larus propose to use a heuristic to find a good solution: first, a maximal spanning tree on G is built. Then, witnesses are put on all edges not in the spanning tree. This procedure guarantees that there is only one possible witness-free path between any two witnesses.

To follow the program flow in between different control flow graphs, *blocking* witnesses are introduced. Blocking witnesses are placed in the control flow graph on edges preceding call sites or exits of functions.

5.2.2 Exploiting Time Information

Logic analyzers or testbeds with GPIO tracing abilities do not only capture the states of the I/O pins, but also the time of the event, i.e., *when* a state changed. Figure 5.2 shows two excerpts of a control flow graph where we can leverage this knowledge to further reduce the overhead of instrumentation. On the left, we can measure the execution time between the two witnesses (black circles) to infer the number of loop iterations. This renders the crossed out witness superfluous. Similarly, in



Figure 5.2: By measuring the execution time, the control flow between the two witnesses (black circles) can be determined. This is possible e.g., for a simple loop (left) or for a graph with different execution times for each branch (right).

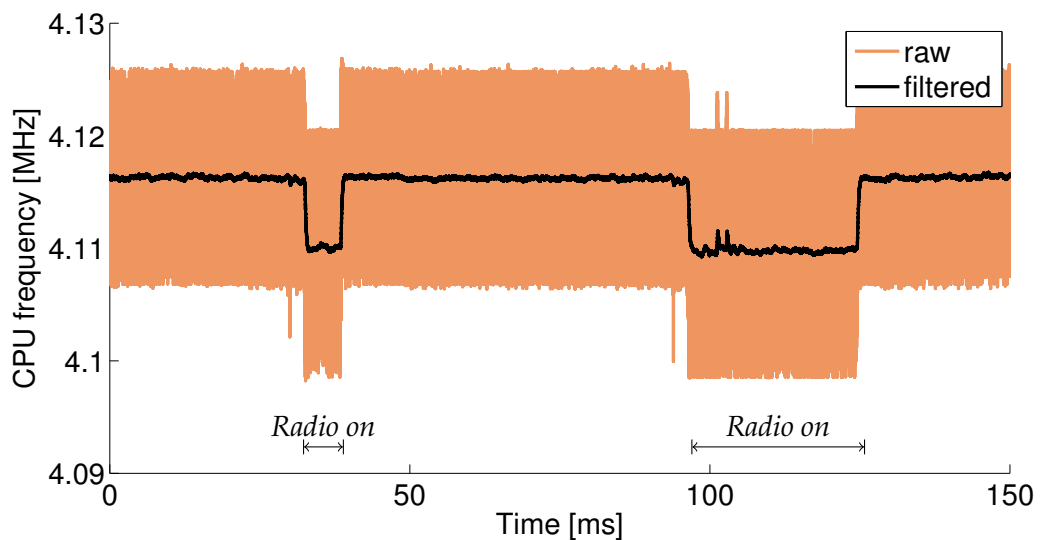


Figure 5.3: Excerpt of a CPU clock speed measurement on an MSP430. The clock frequency varies over time, in this example up to 30 kHz (0.7%).

the example on the right, if the execution times for the sequence $B - C - E$ is different from $B - D - E$, we can again infer the program flow from the time interval between the two remaining witnesses.

Both *time measurements* and *actual execution time* might be affected by uncertainties. Time measurements have a minimal time resolution, while execution times are affected by the stability of the processor's clock. Figure 5.3 shows a measurement of the CPU's main clock on a TelosB node while duty-cycling the radio transceiver. The resulting variation in power dissipation leads to changes in the supply voltage, and in consequence alters the speed of program execution. Uncertainties might also arise

from an inaccurate computation model, e.g., when neglecting effects of caches. Based on these observations, we conclude that time uncertainties need to be well incorporated in a method that infers execution flow based on time measurements.

5.2.2.1 Problem Definition and Modeling

Program model. To describe the problem more formally, we extend the given control flow graph to $G = (V, E, W, T, B)$. As before, G is a directed graph, and every procedure of the program is represented by a single instance of G . Vertices $v \in V$ represent basic blocks and edges $e \in E$ transitions between them. Weights $w(e) \in W$ are the expected number of times a transition is taken at runtime.

In addition, execution times of basic blocks are annotated to vertices as costs $t(v) \in T$, and every execution of the procedure corresponds to a path in G , i.e. a sequence of vertices and edges. To take uncertainties in the execution times into account, T actually provides execution time intervals. The execution time $t(v)$ of the basic block associated to v is in the interval $t(v) \in T(v) = [l(v), u(v)]$. With each vertex v , there is also associated a bound $b(v) \in B$ which bounds the number of times the program flow may pass the corresponding basic block, possibly infinity if there is no bound known. In other words, any (feasible) path has $b(v)$ or fewer occurrences of vertex v in its sequence.

We add two special vertices to G , an *ENTRY* vertex that has an edge e_{entry} to the entry of the procedure, and an *EXIT* vertex that has incoming edges $e_{\text{exit},i}$ from every returning vertex. These elements are added for the sake of modeling and do not materialize in any real instrumentation code. A witness set $E_{\text{witt}} \subseteq E$ contains all edges that can be observed during execution. Paths are sequences of edges and vertices in G . For the witnesses $e_i, e_j \in E_{\text{witt}}$, we denote the set of witness-free paths leading from e_i to e_j as $\text{path}_{i \rightarrow j}^*$, i.e., all paths that can reach the edge e_j when starting at e_i without passing another edge of the witness set.

Problem definition. Our goal is to find a set of edges $e \in E_{\text{witt}} \subseteq E$ that minimizes the expected runtime overhead $C(E_{\text{witt}})$, i.e., the total weight of the edges in the set

$$C(E_{\text{witt}}) = \sum_{e \in E_{\text{witt}}} w(e). \quad (5.1)$$

To ensure that we can safely reconstruct the program path taken, it is mandatory that all the witness-free paths between any pair of witnesses and *ENTRY* and *EXIT* have distinguishable execution times. Therefore, we have $\{e_{\text{entry}}, e_{\text{exit}}\} \subseteq E_{\text{witt}}$, and for every pair of $e_i, e_j \in E_{\text{witt}}$ and for every pair of disjoint witness-free paths $p, q \in \text{path}_{i \rightarrow j}^*$ between e_i and e_j it must

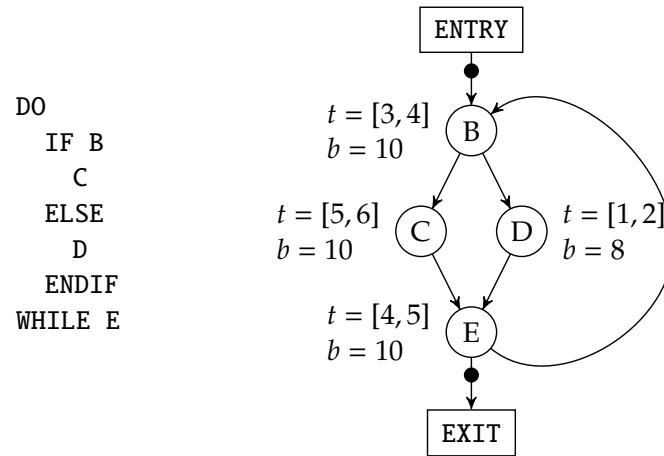


Figure 5.4: Example program with a while loop (left) and resulting annotated graph (right). Edge weights are omitted.

hold that

$$\max \left(\sum_{v \in p} l(v), \sum_{v \in q} l(v) \right) - \min \left(\sum_{v \in p} u(v), \sum_{v \in q} u(v) \right) > \delta. \quad (5.2)$$

This condition ensures that the total execution time of two different paths between witnesses differs by more than the measurement's time resolution δ . In other words, in an admissible witness set there are no two disjoint witness-free paths between any pair of witnesses whose execution times cannot be distinguished.

5.2.2.2 Approach

The problem has the same objective as the baseline approach in Section 5.2.1, namely to minimize the number of times a witness is encountered during runtime. However, the conditions to meet are more relaxed. We do not require to have only one single witness-free path between two witnesses. We even allow cycles in the path, as long as the resulting paths do not overlap in execution time. While this might help to reduce the runtime cost of instrumentation, it makes the problem computationally more difficult to solve because program loops and possibly overlapping cycles in the graph lead to an exponentially growing number of paths between two witnesses. In the example shown in Figure 5.4, there are two possible paths that can be taken within one loop iteration, leading to 2^n different possible paths of n iterations.

Because of these difficulties, we aim at finding a good heuristic rather than an optimal solution. The goal of the heuristic is to reduce the complexity of the problem while still being able to reduce the sum of the

Algorithm 1 Instrumentation**Input:** $G(V, E, W, T, B)$: Control Flow Graph, E_{in} : Initial feasible witness set**Output:** E_{out} : Reduced witness set

```

1:  $E_{out} \leftarrow E_{in}$ 
2: sort descending  $E_{out}$  according to  $W(E_{out})$ 
3: for all  $e \in E_{out}$  do
4:   if  $isAdmissible(G, E_{out} \setminus e)$  then
5:      $E_{out} \leftarrow E_{out} \setminus e$ 
6:   end if
7: end for

```

edge weights in the witness set. In the following, we apply two strategies for complexity reduction: (i) we consider only a subset of all edges in G to be eligible to bear witnesses, and (ii) we find a graph property that helps to quickly discern when paths between two witnesses are unlikely to be distinguishable.

5.2.2.3 Heuristic Overview

An overview on the heuristic is given in Algorithm 1. The key idea is to initially start with a feasible witness set $E_{in} \in E$. Then, we successively try to remove every witness in the set, starting from the one with the largest weight $w(e)$. A witness can be removed if the remaining witness set does still fulfill (5.2). This is verified with the function $isAdmissible$ in Algorithm 1. Since we only remove edges from the set, the total weight of edges in the set can only decrease. The total weight of E_{in} is therefore at the same time a safe upper bound on the resulting cost of the heuristic. We select E_{in} to be the edges determined by the baseline approach without time information.

This approach has some favorable properties: (i) it guarantees that the resulting solution is at least as good as the baseline solution, i.e., $C(E_{out}) \leq C(E_{in})$, (ii) it only needs to perform $|E_{in}|$ admission tests, which significantly reduces the search space, and (iii) for every admission test, only the subgraph that is affected by the removed witness has to be assessed, since we already start with a feasible witness set.

To see why only $|E_{in}|$ tests are necessary, let us consider the example in Figure 5.5. Suppose that we first try to remove e_1 from the witness set. If we cannot remove this witness, it means that there must be at least two paths connecting two other witnesses through e_1 with overlapping execution times. Let us call these other witnesses a and b , i.e., the overlapping paths start at a and end at b . e_1 cannot be removed as long as those witnesses exist. Suppose a is removed in a consecutive step of Algorithm 1 and e_1 revisited. All witness-free paths previously leading to

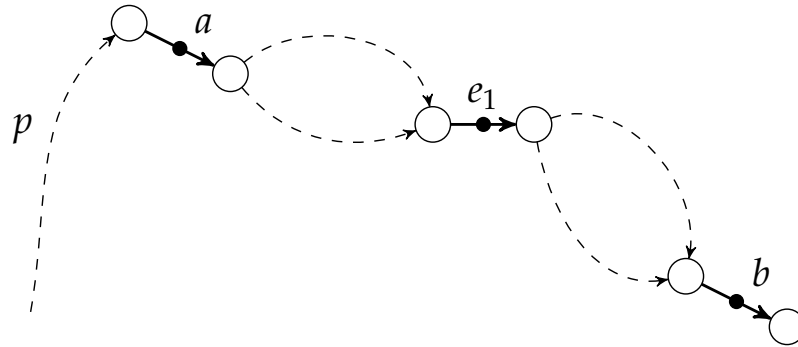


Figure 5.5: Possible initial witness set $\{a, b, e_1\}$. Dashed lines represent witness-free paths. Neither removing a nor b can make e_1 removable, if e_1 cannot be removed with a and b .

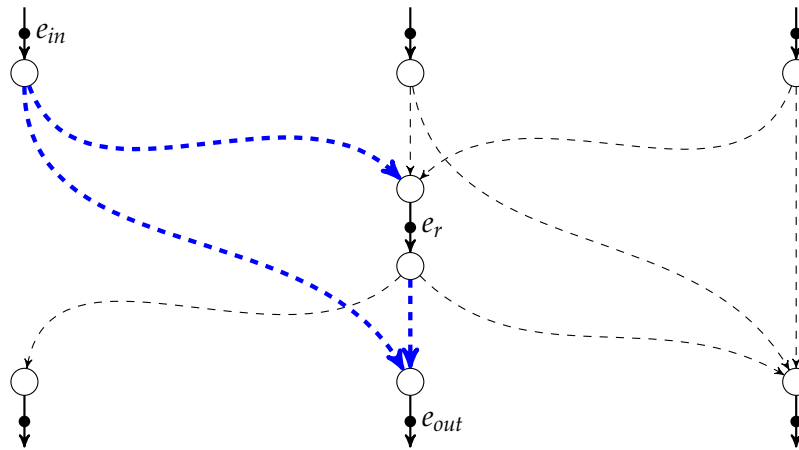


Figure 5.6: Subgraph of the control flow graph that is examined by the admission test when trying to remove the witness on e_r .

a reach now to e_1 . If we remove now e_1 , a path p that had led to a is now extended with all possible paths leading from a to b . Since we know that at least two paths from a to b overlap in time, two of the resulting paths must overlap as well, and therefore we must not remove e_1 . A similar argument can be made for when removing b . In summary, the admission test has to be done only once for each witness; a single iteration over all witnesses in the initial set E_{in} suffices.

5.2.2.4 Test for Non-Overlapping Paths in a Subgraph

Let us now focus on the *admission test*, which is performed by function *isAdmissible* in Algorithm 1. The purpose of this test is to check whether the remaining set of witnesses still fulfills the condition (5.2) on G .

Since we start with a feasible witness set, the test only needs to operate

on the subgraph SG affected by the witness e_r to be removed, i.e., including all vertices and edges that lay on a witness-free path between witnesses that can reach e_r , or that can be reached from e_r , as illustrated in Figure 5.6. In this example, the dashed paths need to be examined, other parts of the graph are not affected by e_r and are therefore omitted in the figure. As the uniqueness needs to be shown for each affected witness pair, we first reduce (for convenience) the subgraph SG . To this end, we choose one pair of witness edges e_{in}, e_{out} and remove all edges and nodes that are not reachable from e_{in} and cannot reach e_{out} , i.e., we keep only the bold dashed paths in Figure 5.6.

For each choice of e_{in}, e_{out} , all possible witness-free paths must be distinguishable in execution times for a feasible solution. For paths that do not contain any program loops, checking (5.2) is straightforward: the upper and lower range of a path can be calculated by summing up the costs of all vertices in the path. However, if a path contains loops, the number of different possible execution times to assess grows exponentially, as discussed in Section 5.2.2.2.

Our approach to mitigate this effect is as follows: we derive a necessary condition that needs to be fulfilled for unique execution times of paths containing loops. Graph structures that result from removing a witness and that do not fulfill this condition can be quickly rejected for admission.

Reasons for Non-unique Paths. As a prerequisite for the later discussion, we distinguish two reasons for non-unique paths:

- Two paths have exactly the same number of occurrences of each vertex, just in a different order. As a result, the accumulated execution times on each path are equal and therefore, the two paths cannot be distinguished. We say that two paths with the same number of occurrences of each vertex are equivalent.
- Two paths have different numbers of occurrences of each vertex, but the difference in (5.2) is not larger than δ .

In the following, we derive a condition that is necessary to exclude the existence of equivalent paths.

Equivalent Paths. The (reduced) control flow graph SG can first be decomposed into strongly connected components, i.e., into maximal subgraphs where there is a path in each direction between each pair of vertices of the graph. This is exemplified in Figure 5.7 (a). If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph (see Figure 5.7 (b)). In other words, the program flow follows the acyclic graph (never returns to previous subgraphs) but may do loops within each strongly connected component.

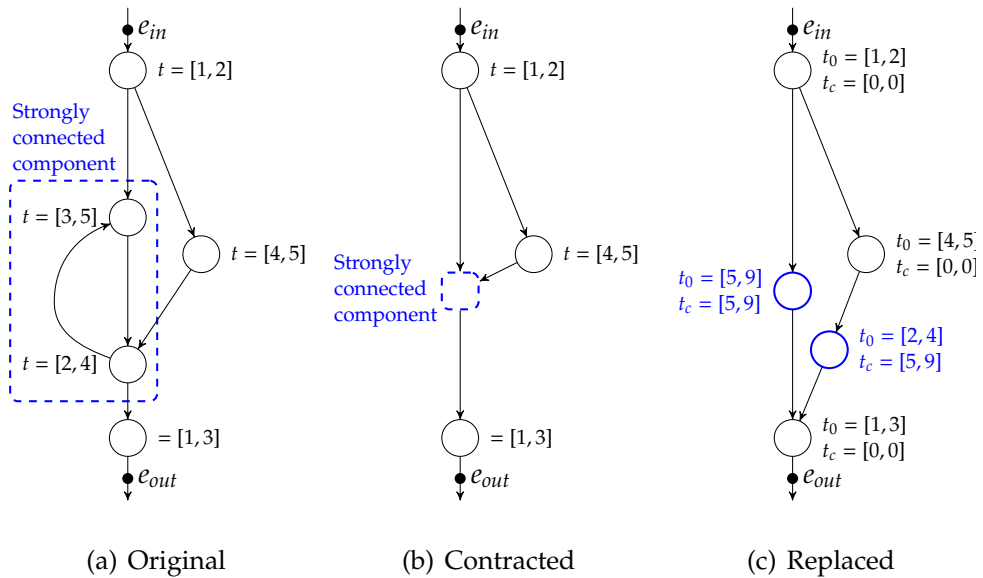


Figure 5.7: Example graph with one strongly connected component. Contracting the strongly connected component leads to a directed acyclic graph (b). The component can be replaced by an acyclic subgraph, leading to new vertices with execution times consisting of an additive part t_0 and a multiplicative part t_c (c).

The program path through the acyclic graph is unique in the number of occurrences of each vertex. Therefore, the only non-uniqueness can come from one of the strongly connected components.

Let us suppose that a strongly connected component contains a vertex v with at least two outgoing edges (v, i) and (v, j) where $b(i) \geq 1$ and $b(j) \geq 1$ is satisfied and with $b(v) > 1$. As defined in Section 5.2.2.1, $b(v)$ is the upper bound on the number of times v can be executed.

Then it may be possible that v is visited twice in a program path. The first time, the program path follows edge (v, i) and the second time it follows edge (v, j) . Now, there may exist another program path where the order of visit is reversed, i.e. at first (v, j) and then (v, i) . This can be seen as follows: The considered subgraph is a strongly connected component. Therefore, there exists a path from every vertex to every other vertex and therefore, there exists paths that connected i as well as j back to vertex v .

The reverse is true as well: if there does not exist such a vertex v in any strongly connected component, then there are no equivalent paths. This can be shown as follows: if there exist two equivalent paths, then there is a first vertex after which the two paths are different, but vertices are visited equally often. Let us call this vertex v and the two succeeding vertices i and j . As both edges can be taken, we have $b(i) \geq 1$ and $b(j) \geq 1$. Clearly, the node v needs to be visited at least twice as the both vertices

i and j are part of both paths (they need to be visited equally often). Therefore, $b(v) > 1$ is necessary.

In summary, if a strongly connected component contains a branch v , then under relatively general conditions ($b(i) \geq 1, b(j) \geq 1, b(v) > 1$) we can hardly exclude the existence of equivalent paths. In other words, a safe and not too restrictive assumption is that strongly connected components of the program graph should contain a single cycle only, without any internal branches. We can now apply this knowledge in our heuristic to early reject the removal of a witness. Strongly connected components in a graph can be found in linear time [Dij97].

Path Differences. Now we can address the problem of comparing path lengths in SG according to (5.2). As discussed in the previous section, we dismiss all subgraphs that contain strongly connected components with branches, i.e., the remaining subgraph contains only strongly connected component with single loops. For execution time calculation, we again use the decomposition of the reduced control flow graph into strongly connected components and the contraction of each component into a single vertex.

In terms of accumulated execution times, we have now a new control flow graph, e.g., as shown in Figure 5.7 (c), with the following properties:

- The new control flow graph has (as before) initial and final edges e_{in} and e_{out} .
- Each strongly connected component (SCC) is replaced by an acyclic subgraph with as many input and output edges as the product of the number of original inputs and outputs to the SCC. The acyclic subgraph has no internal edges. The vertex cost between one pair of input and output edges has two parts: The additive part $t_0(v)$ equals the sum of vertex costs (execution times) of the original program if the program path does not contain a cycle; the multiplicative part $t_c(v)$ equals the sum of execution times of a complete cycle execution. In other words, we have

$$t(v) = t_0(v) + n(v) \cdot t_c(v)$$

The factor $n(v)$ determines the number of complete cycle executions. This number can be bounded from the corresponding bounds of the original control flow graph.

- These acyclic subgraphs are connected by the rest of the control flow graph, i.e. all vertices that do not belong to a SCC. The vertex costs correspond to the original execution times of the control flow graph.

In summary, we now have an acyclic control flow graph $G = (V, E, W, T_0, T_c, B)$ where each vertex has a cost of the form $t(v) = t_0(v) + n(v) \cdot t_c(v)$ where $n(v) \leq b(v)$.

Now, testing (5.2) can be done as follows: the acyclic control flow graph is traversed in topological order. During the traversal, sets of intervals are maintained and updated. At join nodes, two sets are joined and if two intervals overlap, paths cannot be distinguished and the traversal can stop. If a set passes a vertex v , then we replace the set by a new one that contains for each interval I in the original set the intervals $I + t_0(v) + n(v) \cdot t_c(v)$ for all $n(v) \leq b(v)$. If an added interval overlaps with an existing one in the new set, the traversal stops as paths cannot be distinguished anymore.

Finally, we test whether the intervals in the resulting set (at e_{out}) are separated by at least δ , i.e., the distance between the upper bound of any interval and the lower bound of the subsequent interval is at least δ . If no paths are overlapping in execution time for any pair of e_{in}, e_{out} in SG , removing that particular witness from the initial witness set is admissible. We show in Section 5.4.1 that the off-line computational complexity of the admission test is sufficiently small for realistic programs.

5.2.3 Encoding of Witnesses

In order to instrument a program, we have to define an encoding that maps witness identifiers to GPIO state changes. We assume to have N pins that we can observe. The number of witnesses $|I|$ that can occur is $|E| - |V| + 1$ in the case of Ball and Larus, i.e., there is a witness on every edge in the control flow graph G but on those that are part of the spanning tree. In the case where $2^N > |I|$ each witness can be binary encoded with at most one state change per pin. Depending on the processor architecture, this can be managed by a single instruction if all pins belong to the same port on the microcontroller, e.g., using `XOR <IDValue>, <GPIO port>`. If there are more witnesses, we need an encoding that can represent witnesses using sequences of GPIO state changes with a minimal amount of CPU cycles. We employ two strategies to achieve this goal: (i) reuse witness identifiers, and (ii) encode identifiers that are used more often using cheaper codes.

Reusing Witness Identifiers. Witnesses in different control flow graphs can have the same identifiers, since the blocking witnesses allow us to unambiguously determine transitions between control flow graphs. Within a control flow graph, for every witness, every set of reachable witnesses must have unique identifiers. This problem can be modeled as finding a proper vertex coloring in an undirected graph where each

witness is a vertex. For every set of reachable witnesses, we add edges between all pairs of witnesses in the set. Finding a minimal number of unique identifiers is equal to finding a minimal number of colors for the vertex coloring problem.

Identifier Encoding. For the remainder, we assume a hardware platform that supports to set or change the state of all monitored pins at once. One state change is the smallest unit in terms of cost, and allows to represent 2^N different codes. Given a set of identifiers $m \in M$ and an expected occurrence probability $p(m)$, our goal is to find an encoding $C(m)$ that minimizes the total cost $R = \sum_m p(m) \cdot \text{len}(C(m))$. To minimize this cost, we use k-ary Huffman coding [Rom92] with $k = 2^N$.

Nested Witnesses due to Interrupts. Witnesses might be encoded using more than one single GPIO state change. When an interrupt occurs, witnesses emitted within that interrupt handler might separate an ongoing witness in the interrupted program and therefore alter the observed GPIO code of that witness. To avoid such ambiguities, we instrument every start of an interrupt handler with a globally unique GPIO code. Whenever a unique code is observed during trace reconstruction, the parts of the trace belonging to an interrupt can be safely separated from other parts.

5.3 Implementation

In this section, we present a tool that implements the heuristic algorithm described previously for MSP430 based platforms, such as the TelosB node.

The MSP430 series is a family of low-power microcontrollers, featuring a 16-bit RISC CPU [Tex06]. GPIO pins are organized in ports of 8 bits, which can be accessed by means of memory mapped I/O registers. The main clock of the CPU is sourced either by an external quartz oscillator or by an internal digitally controlled oscillator (DCO) with RC-type characteristic. The MSP430 architecture has no caches and no branch speculation, which makes execution timing deterministic. Each instruction takes a fixed number of CPU cycles to complete, depending on the addressing mode of the instruction. Sleep states disable one or several peripherals and clocks to save energy. Any sleep state disables the CPU clock.

We implement instrumentation and replay function in a Python based tool, which we also make available for download¹. Replay is the process

¹<https://github.com/rolim/msp430-tracing>

of reconstructing the program execution based on the recorded events. In the following, we describe the two functionalities in more detail, and we discuss selected design decisions.

5.3.1 Binary Instrumentation

The instrumentation process takes a program binary as input and adds the necessary instrumentation code. The instrumentation part works on the level of machine code instructions because our approach needs accurate information about execution timing. Instrumenting programs in high level programming languages is not an option, since the exact realization and timing highly depend on the compiler and the optimizations performed during compilation.

There are several steps involved in program instrumentation: *(i)* the program structure is analyzed and a control flow graph for each function or procedure is extracted. *(ii)* We then apply our heuristic algorithm to select edges that need to be instrumented with a witness. *(iii)* Instructions that change GPIO states are inserted into the program.

Program flow analysis. This step extracts the control flow graph for every function or procedure in the program. Each instruction of the executable is parsed and grouped into basic blocks, i.e., instructions that form a continuous program flow. The continuous flow is interrupted by (conditional) branch or call instructions. To connect basic blocks in the graph, all possible successors for such instructions need to be known. For jump instructions or conditional branches this is straightforward because the target address is contained in the instruction itself. More complicated are indirect branches, which often result from C-switch statements. Indirect branch instructions operate on addresses stored in registers, and therefore the calculation of the address needs to be understood for a general solution. However, we find that a template based approach works well if targeting only one specific compiler. Such a template is a set of rules that is applied to instructions preceding the indirect branch instruction to find the base address and the size of the jump table involved.

Interrupts and sleep states. Interrupts and sleep states need special treatment when tracing program execution since they influence the program flow. In low-power applications, it is not uncommon for microcontrollers to spend most of the time in a sleep state to preserve power. To properly trace the program execution, we have to keep track of the power state of the processor. Sleep states are exited by interrupts. On the other hand, interrupts might also occur while the processor is active. On the MSP430, the current low-power mode is configured by flags of

the status register. Before an interrupt handler is executed, the status register is pushed onto the stack. Thus, to track the low-power states of the processor, we insert additional witnesses at places where the program might enter or exit a low-power mode, i.e., at the start of an interrupt handler, or at instructions that modify the low-power mode flags of the status register. In addition, a witness at the beginning of an interrupt handler also encodes the last sleep state as stored in the status register on the stack.

Instrumentation code. Instructions that represent a witness should be cheap, both in terms of execution time and size. On the other hand, they must be correctly decodable, irrespective of the program flow and the current state of the GPIO pins. We considered two different possibilities to fulfill these requirements: (i) Every witness translates into a single *exclusive or* operation on the identifier and the current port state. (ii) Every witness consists of two instructions, one that resets the state of the GPIO port, and one that sets the value to the actual identifier of the witness. The first option requires to restore the status register of the CPU in case a succeeding instruction is influenced by a status flag (*zero, negative, ...*) because the *exclusive or* affects the status register. The second option has no functional impact on other instructions. We decided to use the first option, since it has lower overhead, and since we encountered the need for restoring the status register only very rarely.

Effects of instrumentation. Inserting additional instructions into an existing program might change addresses of existing instructions. Therefore, instructions relating to such addresses (e.g., function calls, branches, ...) need to be adapted. Some instructions need to be replaced by others, which is possibly the case for relative jump instructions. On the MSP430, these instructions can perform jumps to addresses that are located within 512 bytes of the jump instruction. Added instrumentation code might now lead to jumps that exceed this limit, and therefore require to replace a relative jump with an absolute one (which takes more cycles to execute). Changes in the program alter the execution timing of the program, and therefore also change the conditions that did hold during the placement of the witnesses.

We resolve this issue by iterating the instrumentation process as long as there are no more changes needed. In every iteration, we update the control flow graphs to reflect the altered timing behavior. We then re-run the witness placement heuristic on the updated control flow graphs. Now, this could lead to an oscillation or even an infinite loop, e.g., if placing an instruction makes an ambiguous path distinct again. To prevent this from happening, we only allow each iteration to add additional witnesses but not remove old ones.

5.3.2 Replay

During replay, we translate the recorded GPIO events to program addresses. A GPIO event is a tuple of time, pin number and pin state. First, we group the events that have the same time. Then we lookup the resulting witness identifier, which is encoded as described in Section 5.2.3. During replay, we keep track of the current program location. By following the path to the next recorded witness in the control flow graph, we can gradually reconstruct the control flow during execution. In case there are multiple possible paths between the current location and the next witness, we choose the path that has an execution time closest to the measured execution time. Our heuristic ensures that this is sufficient to remove path ambiguities.

5.4 Evaluation

We evaluate the impact of our tracing method on five different applications. All experiments run on 31 TelosB nodes on FLOCKLAB. We assess the impact of GPIO tracing with three increasing levels of information: (i) witness ID only, (ii) with timestamps of witnesses, and (iii) including upper bounds of loops. For every setting, we investigate the impact on program size and runtime overhead. Our experiments reveal the following key findings:

- Instrumentation with knowledge of upper bounds adds the lowest *static memory overhead* to the program binary, on average 31 % (6.8 kB). Compared to the witness-only case, upper bounds can reduce the number of non-blocking witnesses by 67 %.
- The *runtime overhead* in terms of CPU cycles to trace the entire program flow is between 14 % and 21 % (19 % average) using upper bounds. Compared to the baseline approach, we see reductions between 13.5 % and 38.3 %.
- Programs performed similarly with and without instrumentation. Glossy, which relies on exact timing of actions in order to generate constructive radio interference within the network, achieved at least 99.978 % data yield with any of the three instrumentation types.

5.4.1 Experimental Setup

To show the feasibility and the potential of GPIO tracing, we selected five example applications that are optimized for different target scenarios.

Table 5.2: Binary size and number of program elements for applications used in the evaluation.

Application name	Operating system	Size (bytes)	Functions	Basic blocks	Instructions
Blink	TinyOS	2564	19	259	875
MHO	TinyOS	24522	175	2728	8329
MHO LPL	TinyOS	26128	188	2967	8848
Dozer	TinyOS	33798	203	3347	11090
Glossy	ContikiOS	16128	125	1556	5395

Apart from the TinyOS *Blink* application, all candidates form a multi-hop network. *Multihop Oscilloscope (MHO)*, available from the TinyOS repository, is a general purpose data gathering application. It samples a sensor value every second and reports these values every five seconds to a base station. We choose to run this application with and without the low-power listening MAC protocol (LPL). In the LPL configuration, we set the wakeup interval to 512 ms. *Dozer* [BvRW07] is optimized for very low duty cycles and low data rates, generating a data packet on each node every 30 seconds. Finally, we include *Glossy* [FZTS11], which provides a fast and energy efficient flooding architecture. We use a flooding period of 2 seconds. This selection of example applications also covers two popular sensor network operating systems, namely TinyOS and ContikiOS. To illustrate the complexity level of each program, we list the program size and number of basic elements (functions, basic blocks and instructions) of each binary compiled for the TelosB platform in Table 5.2.

We instrument each application with three different levels of information to assess the benefits of additional information with respect to memory overhead and runtime overhead. First, we rely only on the recorded witness identifiers. Since this approach does not use any time information, there must be only one possible program path between two consecutive witnesses. This approach is the most expensive in terms of overhead. Then we make use of witness timestamps. As explained in Section 5.2.2, by taking execution times into account, we can remove witnesses from the previous approach, as long as paths between consecutive witnesses have distinguishable path lengths. Based on empirical measurements, we assume a time inaccuracy of 1%. We assume to have no information about loop upper bounds (i.e., the upper bound is infinity), and thus this approach cannot remove witnesses within loops. The two instrumentation methods so far do not require any specific runtime information about the program, and can therefore be applied

Table 5.3: Number of witnesses used in instrumentation, broken down in blocking and non-blocking witnesses for each application.

Application name	Blocking witnesses			Non-blocking witnesses		
	w/o time	time	time & bounds	w/o time	time	time & bounds
Blink	70	70	70	76	39	35
MHO	916	916	916	409	162	145
MHO LPL	1022	1022	1022	412	156	138
Dozer	1113	1113	1113	475	222	214
Glossy	390	390	390	438	258	184

without any further information about runtime execution. To further reduce the number of witnesses and therefore the runtime overhead of our tracing method, we add information about upper bounds of loops. For our experiments, we use a measurement based approach to get upper bounds. More specifically, we profile each application and count the maximal number of iterations of each loop. We then add a margin of 20% and use this value as upper bound. In the following, we refer to these three different variants as *w/o time*, *time* and *time & bounds*. The instrumentation process for any of the 15 binaries took less than 2 minutes each on a standard PC. As instrumentation adds witnesses at beginning of interrupt handlers (see Section 5.3.1), we adapt the time compensation code of Glossy in one of the handlers.

5.4.2 Static Overhead

The number of added witnesses for each instrumented application is shown in Table 5.3. A key figure is the number of non-blocking witnesses that remain to be instrumented. As described in Section 5.2, blocking witnesses are required to retrace the program flow between functions. Since we need this property in any of the three instrumentation variants, the timing aware approaches can only remove non-blocking witnesses. Figure 5.8 shows that time analysis cuts the number of non-blocking witnesses in half. Using loop upper bounds further reduces the number of witnesses. In the case of Multihop Oscilloscope LPL, we see a total reduction in non-blocking witnesses of 67%.

The static overhead in terms of program memory is of practical importance, since program memory is a scarce resource in wireless embedded systems. In Figure 5.9, we compare the size of the

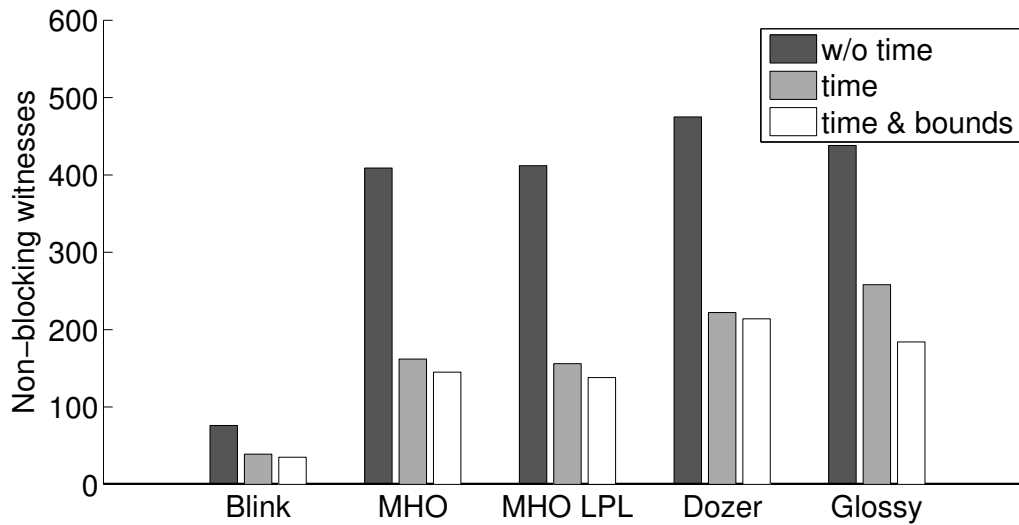


Figure 5.8: Number of non-blocking witnesses in each instrumented binary (last three columns of Table 5.3).

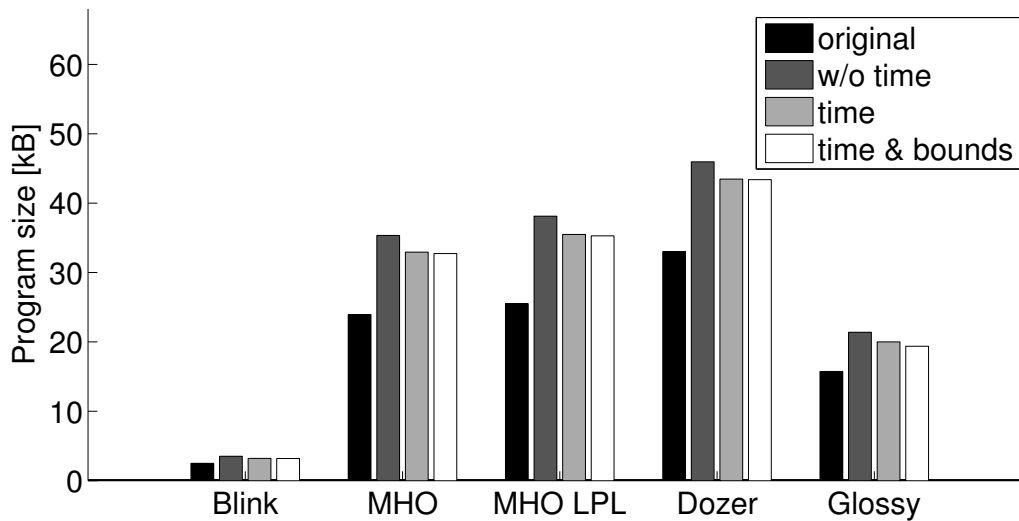


Figure 5.9: Program memory utilization for the original program and instrumented variants. No additional RAM is needed.

instrumented binary to the original one. We find that the memory overhead ranges from 23% to 49%. As expected, having timestamps and upper bounds each reduces the size of the instrumented binary. The most efficient variant (*time & bounds*) adds a memory overhead of at most 38% for all the investigated examples. While this is not a negligible amount of memory, it still allows us to fully instrument programs that use up to 72% of their total available flash memory, e.g., 34.7 kB on a TelosB node. Our tracing method does not require additional RAM.

Table 5.4: Reliability and radio duty cycle of Glossy.

Instrumentation type	Reliability	Radio duty-cycle
Unmodified	99.996 %	0.57 %
w/o time	99.986 %	0.60 %
Time	99.996 %	0.59 %
Time & bounds	99.978 %	0.59 %

5.4.3 Runtime Overhead

We use FLOCKLAB’s GPIO tracing service as described in Chapter 4 to record the states of the GPIOs during execution. FLOCKLAB allows to trace up to 5 GPIO pins on every node. We let every combination of application and instrumentation type run for 30 minutes on 31 TelosB nodes. Then, we replay all the traces and count the number of CPU cycles spent on emitting witnesses and the CPU cycles used for original instructions.

The runtime overhead is shown in Figure 5.10. The average overhead per experiment ranges from 14 % to 28 %. Similarly as for the static overhead, we see that the instrumentation method using loop upper bounds adds the lowest overhead (14 % to 21 %). The advantages of the upper bound approach are more pronounced because removing witnesses in a loop has a bigger impact on runtime overhead than on memory overhead. We also find that the runtime overhead (avg. 19 %) is smaller than the program memory overhead (avg. 31 %) would suggest. This is mainly due to instrumentation instructions on the MSP430 being larger than the average, but execution wise around average. Comparing *w/o time* to *time & bounds*, we see reductions between 13.5 % and 38.3 %.

Since Glossy requires tight synchronization between concurrent transmitters in order to achieve constructive interference [FZTS11], we verified the performance of all three Glossy runs by comparing the performance metrics *reliability* and *radio duty-cycle* to the unmodified program. As shown in Table 5.4, all three test runs exhibit a high reliability of at least 99.978 % and low radio duty-cycle. In addition, there are only slight variations between tests.

These experiments show that full control flow tracing in a testbed is feasible and adds a relatively low runtime overhead. Moreover, it is also suitable for tracing time sensitive applications.

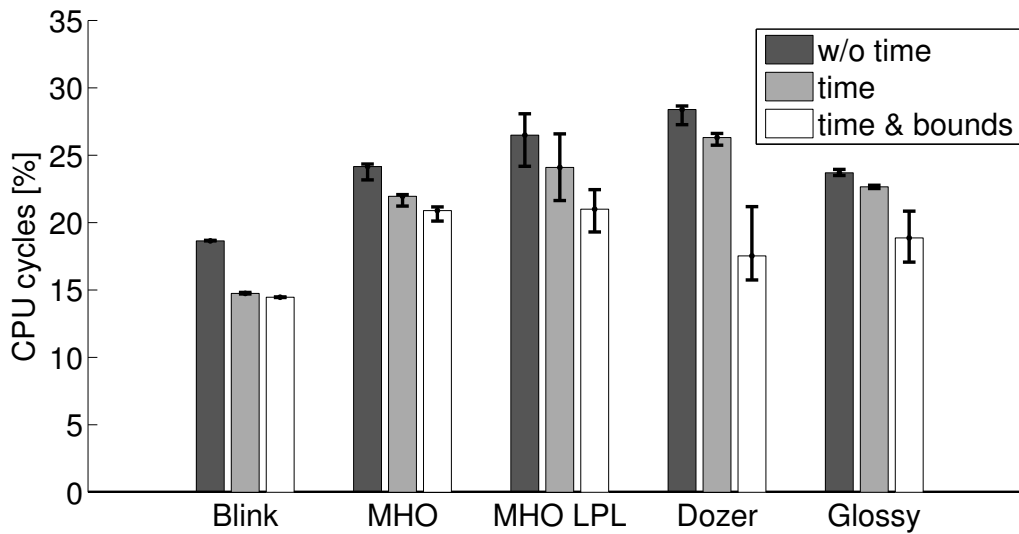


Figure 5.10: Runtime overhead caused by added instructions. Error bars indicate minimum and maximum.

5.5 Case Study

In the following, we give two examples that show how program traces can be leveraged to analyze program behavior in wireless embedded systems. We extract and analyze code coverage, and we use traces to inspect timing behavior.

5.5.1 Code Coverage

In this case study, we examine code coverage statistics that can be directly extracted from execution traces. Code coverage describes the amount of code that has been executed during runtime. It is used to measure the amount of program code that is covered by a set of tests. Test suites generally aim at maximizing their code coverage [PY99]. We calculate the code coverage as the percentage of distinct executed basic blocks during a program run.

In a distributed system, code coverage might differ between parts of the system, depending on the task set allocated on each node. Nodes with similar code coverage might take on similar roles in a network, e.g., acting as sink or leaf node, or relaying messages. Examining the diversity of code coverage might help to understand these systems better, and to tailor test cases to specific scenarios.

In the following, we analyze the traces obtained in Section 5.4 and calculate the overall code coverage and the distance of code coverage between nodes. We define the distance D between two sets of covered

Table 5.5: Code coverage of example applications.

Application name	Code coverage
Blink	66 %
Multihop Oscilloscope	65 %
Multihop Oscilloscope LPL	67 %
Dozer	62 %
Glossy	65 %

basic blocks Γ_i, Γ_j as the Hamming distance between the sets, i.e.,

$$D(\Gamma_i, \Gamma_j) = |\Gamma_i \cup \Gamma_j| - |\Gamma_i \cap \Gamma_j|.$$

Overall, we find that the test runs cover 62 % to 67 % of all basic blocks in the program. Specific numbers are given in Table 5.5. Interestingly, even a simple program like Blink does only cover two third of the entire program. Most of the uncovered parts of Blink are related to interrupt handlers (Timer A) that are never used during program execution, but still included in the binary. Pointers to uncovered program parts can help the developer to eliminate unused code in order to save on scarce program memory.

To analyze the variability in code coverage across the network, we calculate two coverage metrics from the traces: the average distance between sink node and the rest of the network, and the average pairwise distance between all nodes but the sink node. These metrics are shown in Table 5.5.1. We find that the sink node executes clearly a different set of basic blocks than the rest of the network. For Blink, all nodes have exactly the same code coverage, as would be expected, since the program runs independently on every single node. In the case of multi-hop applications, there seem to be two distinct cases that differ to each other by the amount of variation between the non-sink nodes. The applications running on top of topology based network protocols (MHO, MHO LPL, Dozer) exhibit larger variations than the network flooding based approach (Glossy). In the latter case, coverage differs only within 24 basic blocks among all non-sink nodes. This can be explained as follows: in topology based approaches, the executed code paths depend on the node's position in the topology, e.g., compared to a node close to the sink, a leaf node does not need to forward any messages. In Glossy, all nodes participate similarly in every flood, irrespective of their position in the network.

Control flow tracing allows to extract useful aggregated information from program executions. Code coverage can be used to reason about

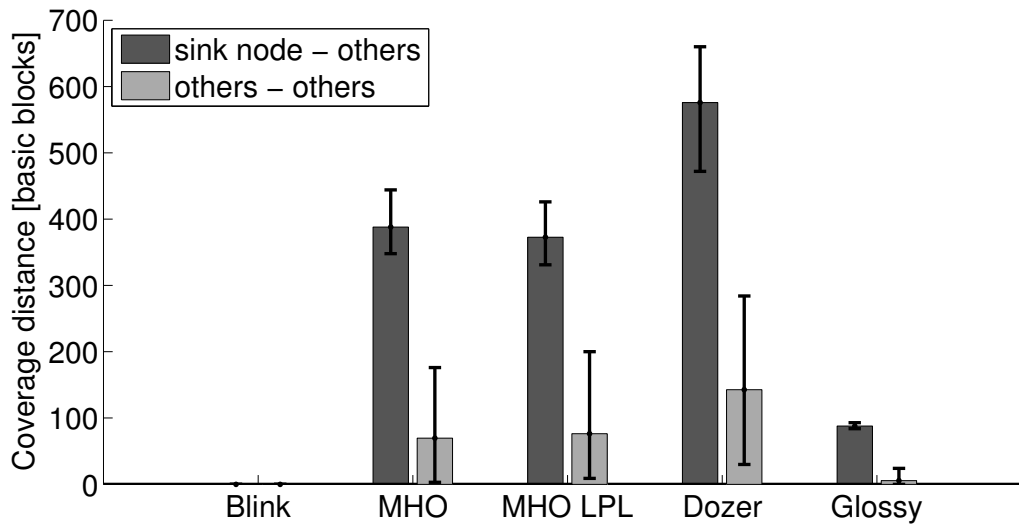


Figure 5.11: Average pairwise distance of code coverage between nodes in the experiment. Error bars indicate minimum and maximum.

program behavior in a network, or to find potentially unnecessary code, thus helping to reduce program size.

5.5.2 Inspecting Time Behavior

Knowing the time behavior of program tasks in embedded systems is important because erroneous or unwanted timing can degrade performance (e.g., higher energy usage, or lower throughput). Runtime traces can give pointers to problematic parts of a program.

In this section, we analyze the timing within a part of the boot sequence on a TelosB node. During our experiments for the evaluation of this chapter we realized that boot time is highly variable among nodes when running ContikiOS. We observed differences of several seconds. Inspection of traces show that most of the time is spent in the calibration routine of the digitally controlled oscillator (DCO). This oscillator has to be configured by software to run at the desired target frequency, 4 MHz in our case. The actual frequency of the DCO can be measured using the low frequency crystal oscillator on the TelosB as reference.

Figure 5.12 shows the actual DCO frequency (extracted from traces) during the calibration process on TinyOS and ContikiOS. The calibration process is very differently executed: TinyOS uses a binary search, achieving calibration within tens of milliseconds, while ContikiOS performs a linear sweep (steps are caused by overlapping frequency ranges within the sweep). An excerpt of the calibration routine is

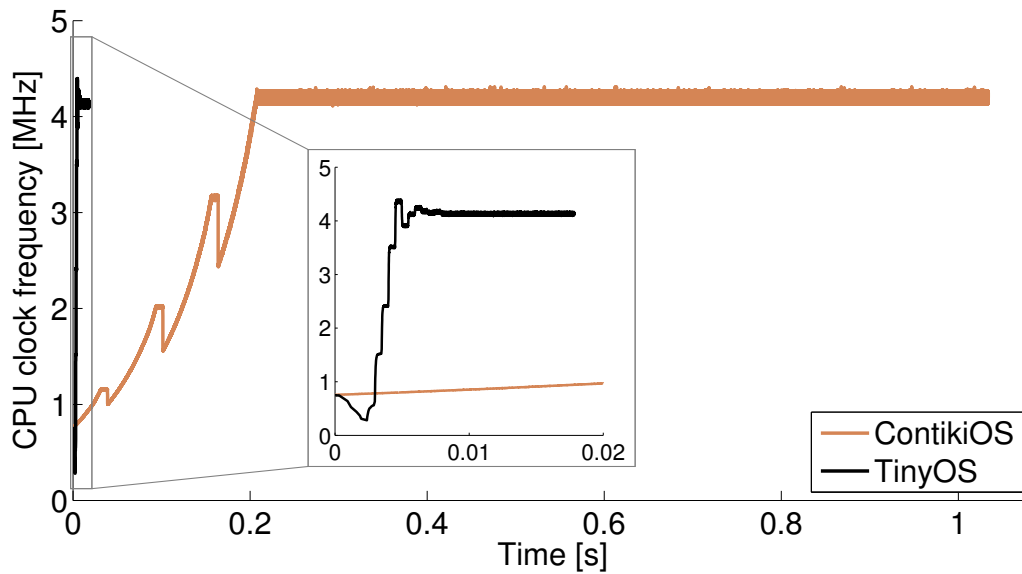


Figure 5.12: CPU speed profile at boot time for two different clock calibration algorithms.

provided in Listing 5.1². Apart from the clear difference in time needed to reach the target frequency, the calibration routine of ContikiOS exhibits a suspicious tail after reaching the target frequency. This tail turns out to be the main source of boot time variability among the nodes. By inspecting the traces, we find that this period of almost constant frequency is caused by the calibration routine oscillating around the target frequency. As can be seen in Listing 5.1, line 103, the calibration ends if the measured number of clock cycles *exactly* matches DELTA, which is the number of DCO cycles during one reference clock cycle at the target frequency. Due to the granularity of frequency control, it might take a long time until this condition is met, even though the actual frequency cannot be tuned closer to the target frequency. To remove this random behavior, we suggest to add a test for oscillation, or to replace the calibration routine with a similar approach as in TinyOS.

This example shows that the increased observability provided by control flow traces can greatly facilitate the analysis of embedded systems. Time behavior can be analyzed in detail without tailoring an instrumentation strategy to a specific goal, e.g., using counter registers and `printf` statements.

²<https://github.com/contiki-os/contiki>, October 7, 2016

Listing 5.1: Initial DCO calibration in ContikiOS.

```
95 while(1) {
96
97     while((CCTL2 & CCIFG) != CCIFG);
98     CCTL2 &= ~CCIFG;
99     compare = CCR2;
100    compare = compare - oldcapture;
101    oldcapture = CCR2;
102
103    if(DELTA == compare) {
104        break;
105    } else if(DELTA < compare) {
106        DCOCTL--;
107        if(DCOCTL == 0xFF) {
108            BCSCTL1--;
109        }
110    } else {
111        DCOCTL++;
112        if(DCOCTL == 0x00) {
113            BCSCTL1++;
114        }
115    }
116 }
117 }
```

5.6 Summary

We have presented a novel testbed based method that allows to completely trace the program flow of wireless embedded systems. By inserting instructions that encode witnesses of the program flow using GPIO state changes, the execution of a program can be observed down to instruction level. To this end, we designed a new algorithm that uses time information to reduce runtime overhead of instrumentation substantially, while still being able to uniquely determine the executed program path. Our experimental evaluation showed that our approach has an average runtime overhead of 19%. Compared to an approach without time analysis, using time information reduces the runtime overhead by up to 38.3%. This makes our approach also suitable to trace timing sensitive applications.

By enabling program tracing on a wide range of wireless embedded systems, we provide a tool that can serve as the starting point for new debugging methods, automated program verification or optimization.

6

Passive, Privacy-preserving Counting of Smartphones via ZigBee Interference

So far, we introduced a testbed architecture in Chapter 2, we covered the time synchronization aspect of distributed measurements (Chapter 3, Chapter 4) and developed a method to effectively trace the control flow in a testbed (Chapter 5). In this chapter, we make use of these building blocks and apply them in a project that aims at estimating the size of a crowd by analyzing interference generated by smartphones.

Smartphones are an integral part of our daily lives, and will be even more so in the future. Ericsson forecasts 5.6 billion smartphones around the world in 2019, accounting for 60 % of all mobile subscriptions [Eri14]. This proliferation is intriguing as it opens up the possibility to exploit smartphones for collecting statistically significant amounts of data about the way people behave and interact [BEM⁺13b]. In addition, nowadays almost every smartphone has Wi-Fi built in. Due to ever-increasing mobile data traffic [Eri14], users activate Wi-Fi on their smartphones to get fast and cheap connectivity to a Wi-Fi network whenever possible. To discover these offloading opportunities, smartphones actively scan for Wi-Fi access points (APs) by periodically sending out probe request frames, or *probes* for short.

Motivation. We seek to leverage the high proliferation of Wi-Fi enabled smartphones to determine their numbers in real-time based on the probes they emit. Given the high penetration of smartphones across the general

population, such counts are of great value in numerous applications. For example, they can be used to estimate the density of a crowd, which is an important feature in crowd management [HB]W05] to prevent disasters like the 2010 Love Parade stampede that killed 18 people [The10]; in retail, they provide insights into customer engagement and help improve in-store sales [Cis14]; and in a city, information about the number of pedestrians, cyclists, and motorists using particular road segments at any given time enables an intelligent transport system where traffic lights adapt to reduce travel times [The15].

Prior to the advent of the smartphone, video surveillance and image processing have been used to estimate the number of people in an area [HTWM04]. These solutions, however, need special-purpose powered equipment and impact privacy. Today, an alternative solution is to encourage people to run an application on their smartphones that periodically sends the GPS location to a server [WFMK⁺12]. While here users can opt out at any time, this approach is invasive in that it alters the smartphone software, and works only outdoors where GPS reception is possible.

A non-invasive approach (i.e., without modifying the smartphone) is to use existing Wi-Fi APs or to deploy dedicated Wi-Fi monitors to estimate people count by sniffing the unique MAC addresses of their smartphones, which are contained in clear text in every probe [Cis14, ME12]. From a privacy perspective, this is even worse than video surveillance because the owner, which can be unequivocally identified from the MAC address, has no means to notice that she is being tracked. Thus, lawyers, authorities, and the population take a skeptical position *despite the use of anonymization techniques such as MAC address hashing*. This could be witnessed, for example, by a public outcry and eventual ban of such a system in the City of London [The13], and unclear current law preventing the deployment of a smart traffic system in Copenhagen [The15].

Contribution. To address these issues, we introduce DEV_{CNT}, a system providing real-time estimates on the number of Wi-Fi enabled smartphones within an area in a non-invasive manner. DEV_{CNT} preserves *by design* the privacy of smartphone users, thus overcoming the concerns associated with prior approaches and fostering the rapid deployment of innovative applications.

To this end, DEV_{CNT} takes advantage of cross-technology interference in the 2.4 GHz band. As described in Section 6.1, the IEEE 802.11 standard prescribes that an *active scan* should involve sending a probe on each Wi-Fi channel. Since each Wi-Fi channel overlaps with at least one ZigBee channel, a ZigBee device can perceive a probe transmission as a short

increase in the received signal energy. DEV_{CNT} uses one or multiple battery-powered ZigBee devices to flexibly cover large areas. Every device periodically reports the number of active Wi-Fi scans it has seen to a sink. Based on the received scan counts, DEV_{CNT} estimates at the sink the number of Wi-Fi enabled smartphones within the range of each ZigBee device.¹

DEV_{CNT} works both indoors and outdoors, and provides a new estimate every few seconds. DEV_{CNT} is non-invasive and fully passive in that it neither modifies the software running on the smartphones, nor does it externally affect the smartphones' operation (as done in [ME12]). Because it is technically impossible for a ZigBee receiver to demodulate Wi-Fi frames, DEV_{CNT} cannot track individual smartphones nor identify their owners. Finally, by using low-power wireless battery-powered devices, DEV_{CNT} reduces costs and increases flexibility during system installation, maintenance, and removal compared with previous solutions that use video surveillance or Wi-Fi monitors.

Achieving these favorable properties while providing accurate smartphone counts is challenging for at least three reasons:

- Amplitude and time resolution of received signal strength (RSSI) information on a ZigBee receiver is coarse-grained. DEV_{CNT} must therefore cope with inaccuracies when characterizing Wi-Fi transmissions by their length and signal strength, which are the only features DEV_{CNT} is left with to detect scans on a device that cannot demodulate Wi-Fi.
- DEV_{CNT} cannot differentiate between individual smartphones because it cannot see their (unique) MAC addresses, probes from different smartphones may have the same length, and RSSI is a poor classifier due to mobility and multipath fading. While this preserves privacy, it also makes counting Wi-Fi enabled smartphones a difficult task.
- Sending all RSSI samples to a central sink for processing is prohibitive due to the limited bandwidth. Thus, as detailed in Section 6.2, DEV_{CNT} performs most of the required processing on the ZigBee devices. This, in turn, implies that each ZigBee device needs to multiplex its single microcontroller unit (MCU) between three tasks: (i) reading RSSI samples, (ii) processing samples to detect and count scans, and (iii) sending scan counts to the sink. DEV_{CNT} must temporally decouple (i) and (ii) from (iii) synchronously on all

¹Since this chapter deals primarily with the physical and media access layers of IEEE 802.15.4 and IEEE 802.11, we do not discern between these standards and the respective industrial alliances ZigBee and Wi-Fi.

devices to avoid distorting the smartphone count estimations due to self-interference, while simultaneously reducing the time needed for (iii) to have more time available for (i).

As discussed in Secs. 6.3 and 6.4, we tackle these challenges by designing novel signal processing, feature extraction, and classification algorithms. Given that these algorithms need to execute in real-time on devices with severely limited memory and compute power, our algorithms strike a balance between feasibility and optimality. The key insight we use to count smartphones without being able to distinguish them is that the active Wi-Fi scanning rate of a sizable population of smartphones follows a specific and typically narrow distribution. To enable temporal decoupling and fast all-to-one data collection, we use Glossy [FZTS11] to accurately time-synchronize the ZigBee devices and Chaos [LFZ13] as efficient communication support.

To demonstrate the feasibility of our design, we implement a `DEVCNT` prototype on the TelosB [PSC05] platform. We describe how we use `FLOCKLAB` in the development process in Section 6.5. In Section 6.6, we evaluate `DEVCNT` in controlled experiments with up to 31 smartphones from 4 different vendors running iOS or Android, and during a real-world test run in a large lecture hall with more than 100 students. Our results show the following:

- `DEVCNT` accurately detects more than 99% of Wi-Fi scans despite realistic interference from Bluetooth, ZigBee, and Wi-Fi traffic, the latter including TCP and UDP streams.
- `DEVCNT` detects scans with an accuracy above 90% even on ZigBee devices that are 50 m away from the phone.
- In a controlled experiment where the precise ground truth is available, `DEVCNT` estimates the number of Wi-Fi enabled smartphones with accuracies of up to 91%.
- In a real-world test run where the ground truth is extremely difficult to obtain, `DEVCNT`'s processing pipeline sustains signals from hundreds of Wi-Fi transmitters, thus providing meaningful smartphone counts that match the expectations.

Section 6.7 discusses trade-offs and limitations of `DEVCNT`, Section 6.8 reviews related work, and Section 6.9 concludes the chapter.

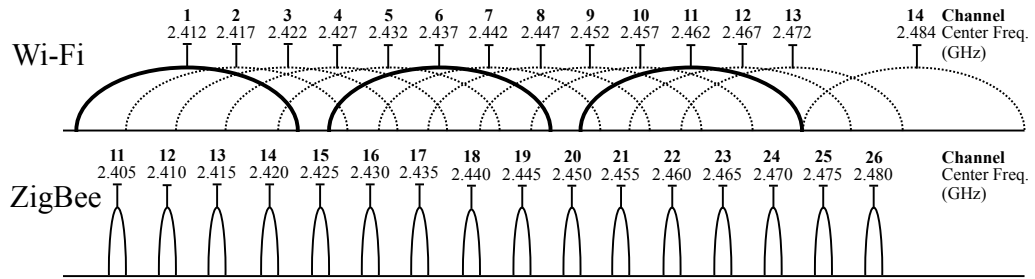


Figure 6.1: IEEE 802.11b/g (Wi-Fi) and IEEE 802.15.4 (ZigBee) channels in the 2.4 GHz industrial, scientific and medical (ISM) band.

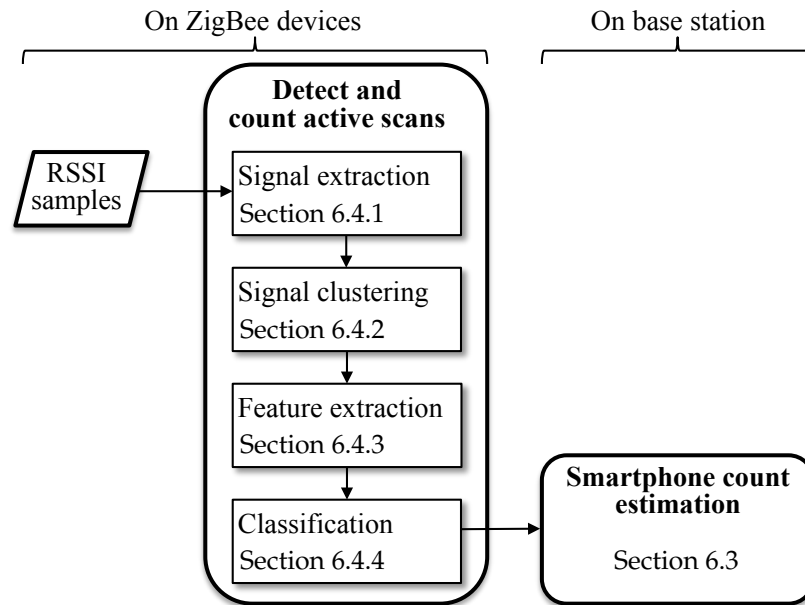


Figure 6.2: High-level view of DEVCNT. A multi-hop low-power wireless network of ZigBee devices continuously samples the received signal strength (RSSI), and uses a novel signal processing pipeline to detect and count active Wi-Fi scans performed by smartphones. Based on the scan counts periodically reported by the ZigBee devices, DEVCNT estimates on a base station the number of Wi-Fi enabled smartphones within the reception range of each ZigBee device.

6.1 Background and Terminology

DEV_{CNT} takes advantage of cross-technology interference between Wi-Fi and ZigBee in the 2.4 GHz band. IEEE 802.11 defines in total 14 channels, as shown in Figure 6.1. Each channel is 22 MHz wide and the center frequencies range from 2.412 to 2.484 GHz. Channel 14 is forbidden in most parts of the world, and channels 12 and 13 are typically not used

in North America due to FCC regulations. IEEE 802.15.4 defines in total 16 channels with a bandwidth of 2 MHz each and center frequencies between 2.405 and 2.480 GHz. Thus, ZigBee and Wi-Fi can interfere when operating in overlapping channels.

While often regarded as a major impediment to the performance of ZigBee networks [LPLT10], `DEV_CNT` takes advantage of this kind of interference. It uses *received signal strength (RSSI)* information available on commodity ZigBee devices to detect an interfering Wi-Fi transmission from a short-lived increase in the RSSI. According to the IEEE 802.15.4 standard, the RSSI value is an average over the last 8 *symbol periods* (128 μ s), and is typically updated on a per symbol basis, that is, every 16 μ s. Furthermore, many ZigBee radios allow to adjust their operating frequency with a certain granularity (e.g., 1 MHz on the CC2420 radio [Tex14]). As explained in Section 6.5, we use this feature in `DEV_CNT` to best align the operating frequency of the ZigBee devices with the center frequency of a Wi-Fi channel.

The majority of Wi-Fi networks operates in infrastructure mode, where *access points (APs)* manage all communications. A client, such as a laptop or a smartphone, needs to associate with an AP before it can use any network services. The IEEE 802.11 standard defines, among other things, a process called *active scanning*, whereby a client sends probe request frames (*probes*) to discover an AP. The standard prescribes that an active *scan* should involve broadcasting a probe on each channel, awaiting and processing possible responses from APs in between. Nevertheless, no further detailed specification of active scanning is provided in the IEEE 802.11 standard. As a result, different Wi-Fi drivers implement active scanning slightly differently. For example, we noticed that many drivers send multiple probes (mostly two) on each channel, as opposed to just one probe. Smartphones perform active scanning periodically every few tens of seconds [FMT⁺06], depending on the operating system and the current operating mode (e.g., whether the smartphone is actively used or in standby mode).

Although there is also a *passive scanning* process foreseen in the IEEE 802.11 standard, where clients passively listen for beacon frames from APs, mobile devices mostly rely on active scanning because it is faster. Typically, APs announce their presence by sending a beacon frame every 102.4 ms. A client must listen for at least that period on every channel to finish a passive scan, whereas probe requests are usually handled within a much shorter time. In the remainder of this chapter, we use the term *scan* to refer to active scans.

An IEEE 802.11b/g frame is preceded by a preamble that is at least 96 μ s long. This is above the 16 μ s symbol period of IEEE 802.15.4, so

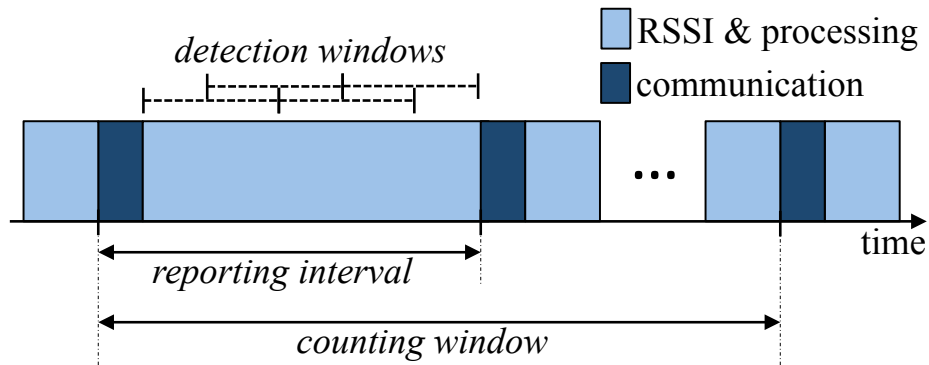


Figure 6.3: Illustration of how the activities of RSSI sampling, processing, and communication evolve over time in DEV_{CNT}. To avoid negative self-interference effects, RSSI sampling and processing are temporally decoupled from communication. The reporting interval determines the timeliness of smartphone counting in DEV_{CNT}, which is on the order of a few seconds.

Wi-Fi frames are, in principle, detectable from the RSSI samples of a ZigBee radio. After the preamble, a Wi-Fi probe contains in unencrypted form the client’s MAC address, the broadcast address or the SSID of a known Wi-Fi network of up to 32 bytes, the data rates supported by the client, and other (e.g., vendor-specific) information.

6.2 DEV_{CNT} Overview

We present DEV_{CNT}, the first system that provides real-time estimates on the number of Wi-Fi enabled smartphones in an area in a fully passive and non-invasive manner without revealing the identity or movement profile of smartphone users.

DEV_{CNT} counts the number of Wi-Fi enabled smartphones based on the probes they send while actively scanning for nearby Wi-Fi APs. Instead of directly eavesdropping on probes using a Wi-Fi capable receiver, DEV_{CNT} detects probes *indirectly* via their interference patterns at a ZigBee receiver. Since ZigBee radios cannot demodulate Wi-Fi probes, DEV_{CNT} does not see the unique MAC addresses contained therein and hence is unable to identify individual phones. As a result, DEV_{CNT} *preserves by design the privacy* of the smartphone users, which sharply differentiates DEV_{CNT} from prior art [Cis14, HTWM04, ME12]. Moreover, DEV_{CNT} is *fully passive* and *non-invasive*: It does not influence a smartphone’s normal operation, for example, by soliciting more probe transmissions [ME12], nor does it require modifications to the smartphones themselves, such as installing and running a dedicated application [WFMK⁺12].

Figure 6.2 provides a high-level view of DEV_{CNT}, while Figure 6.3 illustrates how the different activities in DEV_{CNT} evolve over time. DEV_{CNT} uses a multi-hop low-power wireless network of battery-powered ZigBee devices (*nodes*) deployed across the area of interest; a sink node is connected to a base station. Each node continuously samples the received signal energy by retrieving RSSI from the radio. Based on these RSSI samples, a node detects and counts the number of scans over a sequence of overlapping *detection windows*, as illustrated in Figure 6.3, using the four-stage signal processing pipeline shown in Figure 6.2. All nodes transmit the number of scans they detect within a regular periodic *reporting interval* to the sink. On the base station, DEV_{CNT} uses all scan counts received over a certain *counting window* (see Figure 6.3) to estimate the number of Wi-Fi enabled smartphones within the reception range of each node.

To avoid interference between ZigBee devices, which could adversely affect the smartphone count estimations, DEV_{CNT} decouples RSSI sampling and processing from communication over time. Temporal decoupling requires the devices be time-synchronized, which we achieve by letting the sink perform a Glossy network flood [FZTS11] at the beginning of every communication phase illustrated in Figure 6.3. In fact, the communication phases should be as short as possible to maximize the time the radio is available for RSSI sampling. We thus leverage Chaos as efficient communication support [LFZ13]. Chaos enables DEV_{CNT} to collect small amounts of data, such as a 1-byte scan count, from 100 nodes within less than 100 milliseconds [LFZ13], thus increasing the time available for RSSI sampling. Section 6.6 shows that DEV_{CNT} computes new estimates every few seconds based on up-to-date scan counts collected from ZigBee devices, enabling real-time crowd monitoring and analysis.

By designing and implementing a DEV_{CNT} prototype, we demonstrate that it is indeed possible to perform almost the entire processing *online* on resource-constrained devices. This includes in particular non-trivial signal processing, clustering, feature extraction, classification, and filtering algorithms to detect and count scans (see Figure 6.2), which has been considered too computationally demanding and hence impossible [ZXX⁺13].

Next, Section 6.3 details our approach to estimating the number of Wi-Fi enabled smartphones, and Section 6.4 describes how we detect and count active Wi-Fi scans on ZigBee devices.

6.3 Estimating Smartphone Counts

Counting smartphones without being able to identify them is difficult. Indeed, `DevCnt` cannot identify smartphones by their MAC addresses, since a ZigBee device cannot demodulate Wi-Fi frames. Identification through RSSI is extremely noisy due to mobility and environment dynamics [ME12]. Another option could be to use some form of fingerprinting to passively identify a smartphone based on device- and/or driver-specific variations in active scanning [FMT⁺06, LCTL00]. Unfortunately, such variations (e.g., in the scanning rate) can be extremely small, especially across smartphones from the same vendor running identical Wi-Fi drivers and operating systems, requiring observation periods of an hour or more [LCTL00]. Moreover, these techniques assume that consecutive scans of the same device can be grouped together by matching packet contents—however, `DevCnt` cannot access the contents of Wi-Fi packets.

From the discussion above, it is clear that the only viable option we are left with is to estimate the number of Wi-Fi enabled smartphones based on statistical information about their active scanning behavior. As discussed in Section 6.1, smartphones perform active Wi-Fi scans with a certain periodicity to quickly discover a nearby AP. Nevertheless, the interval between scans is not fixed and depends on several factors, including the smartphone vendor and model, the Wi-Fi driver, the operating system, the applications that are currently running, and whether the smartphone is in active or in standby mode. So a natural question that arises is whether statistical information about the frequency of active scans is indeed useful to accurately estimate the number of Wi-Fi enabled smartphones.

To answer this question, we analyze five different publicly available datasets collected by researchers from the Sapienza University of Rome, Italy [BEM⁺13a]. The datasets contain significant traces of Wi-Fi probes recorded with a commodity Wi-Fi card in monitor mode in diverse scenarios: at international events (Vatican1 and Vatican2), in a big mall (TheMall), at the central train station (TrainStation), and at one of the main entrances of Sapienza (University) [BEM⁺13b]. In addition, we analyze one dataset recorded by us in a (Lecture) hall. The traces range from a few hours to several weeks in length, containing between a few thousand and several million probes. They also differ as to whether the smartphones were likely mobile (University) or static (Lecture), whether the users were likely using their phone (TheMall, TrainStation) or just carrying it in standby mode in their pockets (University), and so on.

Figure 6.4 plots for each dataset the empirical probability distribution of the number of active scans per smartphone over an interval of 3 minutes. We see that despite the diversity of scenarios, all distributions

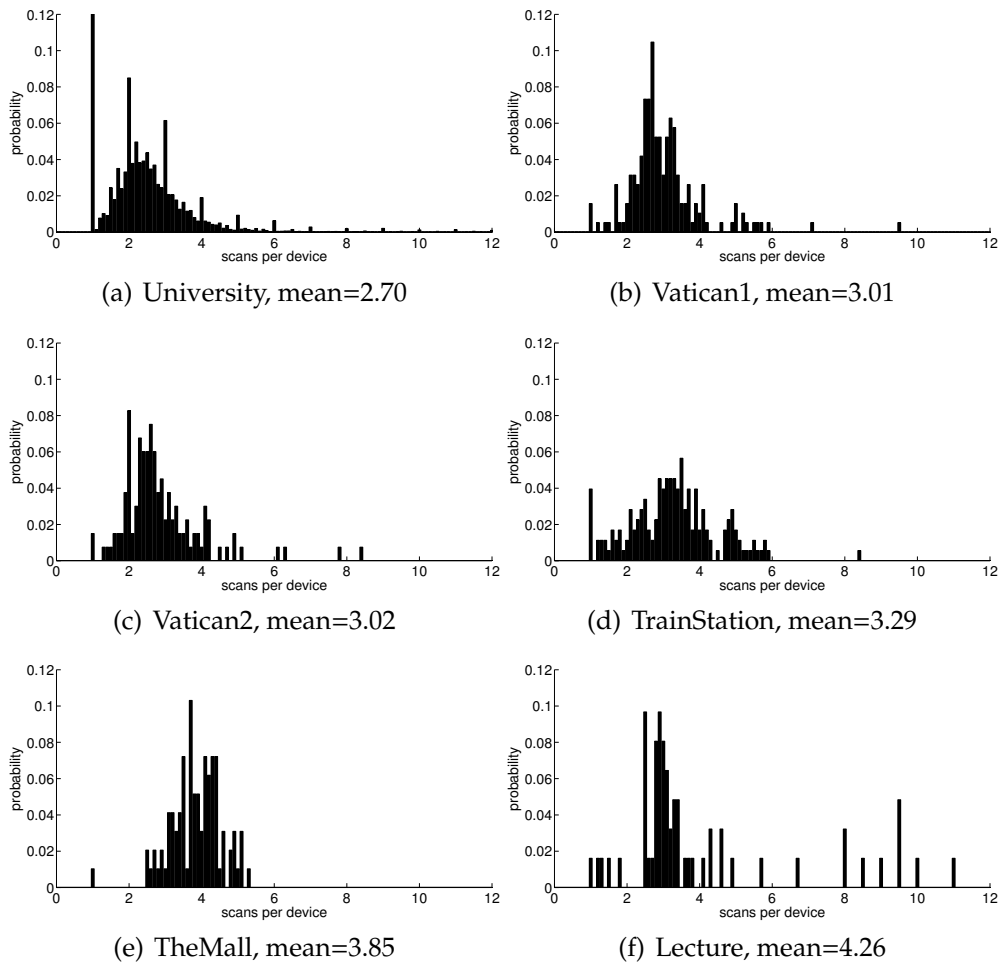


Figure 6.4: Empirical probability distribution of scan counts within a 3-minute time window for different real-world datasets containing massive Wi-Fi traces.

have a very similar shape and a mean of around 3 scans per device. Based on the scenario, we notice slight differences between the means. In the University trace, for example, smartphones are only for a short time in the vicinity of the Wi-Fi sniffer located at an entrance, and also likely in standby mode as they are carried in bags or pockets. As a result, the average number of scans seen from a device is a bit smaller (2.70). By contrast, in the TrainStation and TheMall traces, the smartphones are more static and actively used (e.g., while waiting for a train, taking a break, etc.), so the average number of scans per device is a bit higher (3.29–3.85).

These observations are encouraging in that the average number of active scans per device over some *counting window* is fairly stable despite several influencing factors. This raises the hope that we can use this figure to accurately estimate the number of Wi-Fi enabled smartphones, especially if the expected mobility and usage of smartphones are known,

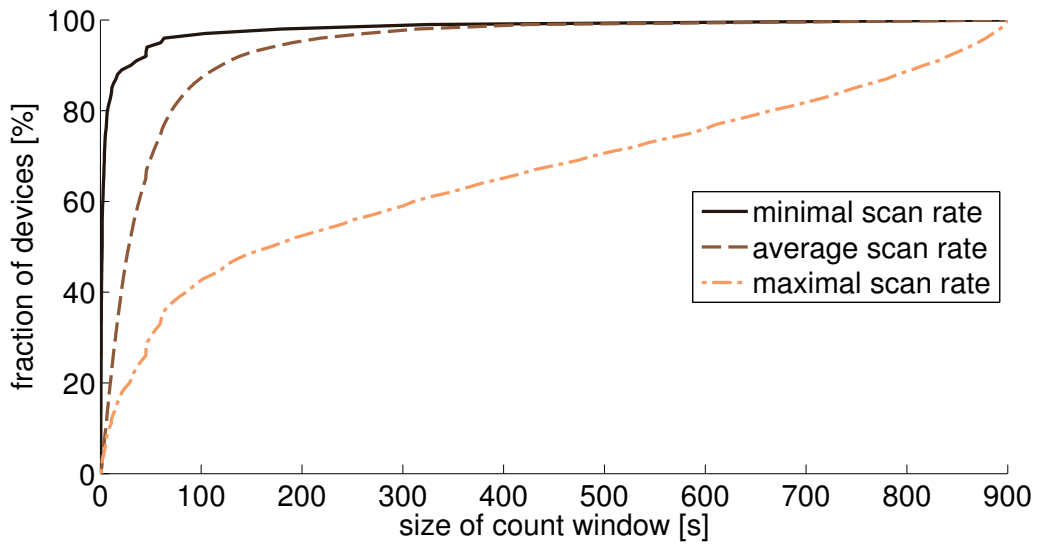


Figure 6.5: Fraction of devices that can be seen within a certain interval in the University Wi-Fi dataset. The empirical cumulative distribution function is shown for minimal, average and maximal scan rate of devices.

which is a valid assumption in the applications we target. Experiments in Section 6.6.2 show that DevCNT can estimate smartphone counts with an accuracy of up to 91%. Key to this performance is (i) the ability to accurately detect active scans on a ZigBee device, as described in the following section, and (ii) an appropriate size of the counting window, as discussed next.

Size of counting window. The counting window (see Figure 6.3) should be rather large to likely contain one or more active scans from a significant fraction of smartphones to provide *accurate* estimates, whereas it should be rather short to provide *timely* estimates. To study this trade-off, we use the University trace and plot in Figure 6.5 the fraction of devices that are covered with different counting window sizes. We assume a device to be out of range, once there is no activity for more than 15 minutes. As scan intervals are not necessarily constant, we display curves for minimal, average and maximal scan intervals of devices in the dataset. To capture scans of a larger fraction of smartphones, we have to choose a larger counting window. For example, using a counting window of 3 minutes, and considering average scan rates, the counting window is large enough to contain scans of 94% of the smartphones. The size of the counting window can be adjusted to match the accuracy and real-time requirements of the application.

6.4 Detecting and Counting Active Wi-Fi Scans

As mentioned in the previous section, `DEVCNT` relies on accurate scan counts to facilitate meaningful estimates on the number of Wi-Fi enabled smartphones in a given area. To this end, `DEVCNT` processes RSSI samples in *real-time* on the nodes using a novel four-stage pipeline, as shown in Figure 6.2.

In a first step, `DEVCNT` extracts interesting portions from a trace of RSSI samples, that is, short-lived periods of elevated signal strength, further referred to as *signals*.

Next, `DEVCNT` takes advantage of pertinent features of the active scanning function, including the periodicity with which probes are sent during an active scan. Specifically, `DEVCNT` (i) groups all signals observed over a fixed-length detection window together, and (ii) clusters the signals inside each group based on their length. The reasoning behind (ii) is that probes sent by a smartphone during an active scan have the same length; clustering ensures that signals that likely originate from the same smartphone are also considered together.

Afterward, `DEVCNT` checks whether a detection window contains signals from an active scan or not. To do so, it first computes a set of features for each cluster in that window. Then, it uses these features to classify each cluster as either "contains a scan" (Y) or "contains no scan" (N). If a detection window contains at least one (Y) cluster, `DEVCNT` considers that detection window as containing an active scan.

At the end of every reporting interval (see Figure 6.3), a node sums up the number of detection windows with an active scan seen in the current interval, and transmits this scan count (via Chaos) to the sink, which forwards it to the base station.

In the remainder of this section, we take a closer look at each of the four processing steps above. Experimental results in Section 6.6 show that due to our techniques `DEVCNT` detects and counts active scans with an accuracy above 99 % despite realistic interference from other wireless technologies.

6.4.1 Signal Extraction

A trace of RSSI samples is not very useful by itself. Rather, `DEVCNT` must first identify and extract parts of an RSSI trace that may (or may not) belong to a probe transmission. For this reason, `DEVCNT` needs to find on the fly the beginning and end of periods with elevated RSSI readings (signals), which is however a non-trivial task.

Figure 6.6 shows an example RSSI trace with three signals. As mentioned before, RSSI is an average over the last 8 symbol periods

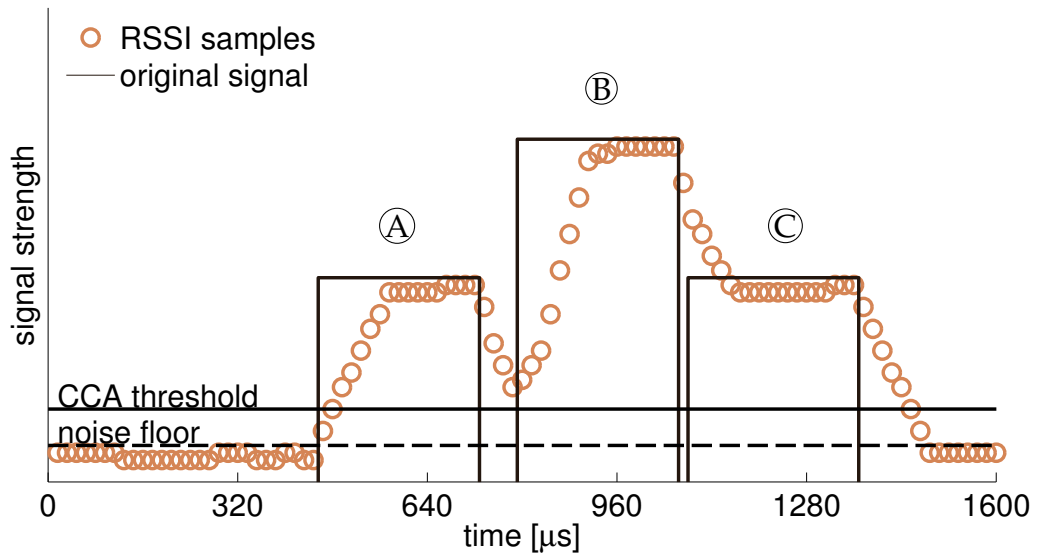


Figure 6.6: Example of three original signals and the corresponding trace of RSSI samples. The RSSI values are an average over the last 8 symbol periods. Thus, a naïve threshold-based approach cannot reliably discern close signals.

(128 μ s), which leads to an inherent smoothing effect. Therefore, as visible for signal (A), it takes 8 samples until the RSSI readings match the amplitude of the original signal. This complicates determining the beginning of a signal. Moreover, if the gap between two signals is less than 128 μ s, such as between signals (B) and (C), the RSSI values remain above the noise floor because they account for parts of either signal. Thus, a naïve threshold-based approach, which could be realized using the clear channel assessment (CCA) capability of a ZigBee radio, cannot reliably discern close signals.

Our solution to these problems is based on the observation that once a signal is present, the RSSI will plateau after some time and eventually start to fall again. Specifically, we abstract this trend as a sequence of states: *rising* \rightarrow *steady* \rightarrow *falling*. A node dynamically determines the current state by looking at the differences between RSSI samples. Based on this idea, we devise the state machine shown in Figure 6.7 to accurately identify the beginning and the end of signals.

The state machine operates on a sequence s_1, s_2, \dots, s_n of RSSI samples. The processing starts when the CCA pin set by the radio indicates an RSSI level above some threshold. Upon taking a transition, we read the next RSSI sample s_i . We remain in state *rising* as long as s_i is greater than the average of the two previous samples. Otherwise, we advance to an auxiliary state *first*, which serves to initialize two variables needed to determine the end of the signal: m , a moving average of the RSSI

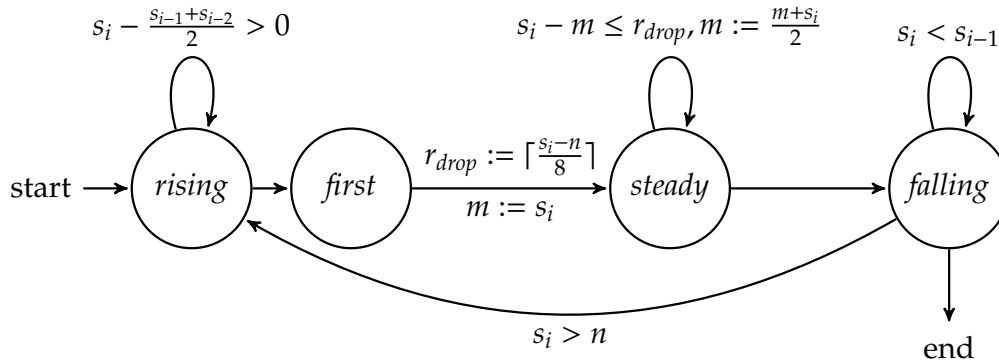


Figure 6.7: State machine implemented by a DEV_{CNT} device to determine the beginning and the end of a signal based on a trace of RSSI samples.

samples and r_{drop} , the *drop rate*. We set the drop rate dynamically, because strong signals cause a larger drop in the (averaged) RSSI values than weak signals. For example, in Figure 6.6 the drop rate of (B) is higher than the drop rates of (A) and (C). We set the drop rate r_{drop} to one eighth of the difference between the RSSI at the beginning of a plateau and the noise floor n . Once the original signal disappears, the RSSI will linearly drop by r_{drop} every symbol period due to the averaging. Based on both m and r_{drop} , we decide whether to stay in state *steady* or to move to state *falling*. The processing stops when the RSSI values fall below the noise floor.

In this way, we precisely identify on the fly the beginning and end of signals. To save memory and computational resources, we use this information to already filter out signals too short or too long to be a probe. Based on the University data set, which contains more than 7 million probes recorded over a period of 10 weeks, we plot in Figure 6.8 the cumulative distribution function (CDF) of probe air times. We find that 99.5% of probes are between 0.5 ms and 4 ms long. Thus, a DEV_{CNT} node only stores the average RSSI as well as the start and end times of signals that fall into this range.

6.4.2 Signal Clustering

An individual signal alone does not provide enough information to decide whether it is from a probe or not. Instead, we should look for relations among multiple signals in order to make this decision. To see why this is a sensible approach, we chart in Figure 6.9 a RSSI trace recorded with a TelosB during an active scan of a smartphone. Each signal corresponds to a probe, and signals with a similar amplitude correspond to probes that are sent on the same channel. We clearly notice, for example, a periodic pattern, which is however only apparent when looking at the entire group

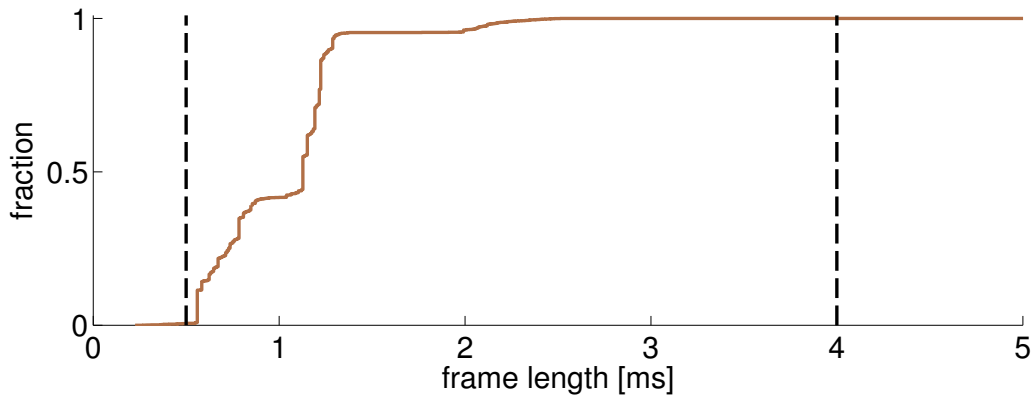


Figure 6.8: CDF of probe lengths of over 7 million probes in the University trace. More than 99 % of probes have a length between 0.5 ms and 4 ms.

of signals.

Ideally, such group formation distinguishes between signals from an active scan and other signals. As mentioned in Section 6.1 and visible from Figure 6.9, probes that belong to the same active scan have the same length and quickly follow each other. We exploit these properties by (i) grouping signals observed within a short *detection window*, and (ii) clustering the signals inside each group based on their length.

Size of detection window. The detection window should be large enough to contain sufficient signals from the *same* scan, but short enough so that it likely contains no signals from *different* scans. We find a good detection window size based on a 2-hour trace captured with a Wi-Fi receiver in a student lab. Due to the overlap of adjacent Wi-Fi channels, a ZigBee device often sees probes from 3 adjacent channels, as in Figure 6.9. In our trace, we find that in more than 99 % of the cases a scan across 3 adjacent channels takes less than 290 ms. We found very similar numbers also in other traces. Thus, we use detection windows that are $2 \times 290 = 580$ ms wide, and let them overlap by half of this size, as shown in Figure 6.3. This is because if we were to use contiguous detection windows, we would miss scans that cross detection window boundaries.

Clustering signals by length. To cluster the signals in a group, we first sort them by length. Then, we form clusters of signals so that signals in different clusters differ by more than a certain threshold. According to Nyquist's Theorem a sampling rate of $1/16 \mu\text{s}$ (i.e., the inverse of the IEEE 802.15.4 symbol period) allows to sample changes in the received signal strength at half of this rate. Thus, signal lengths that differ by less than 2 symbol periods ($32 \mu\text{s}$) are indistinguishable. To compensate for possible errors due to the averaging performed by a ZigBee radio, we add a slack

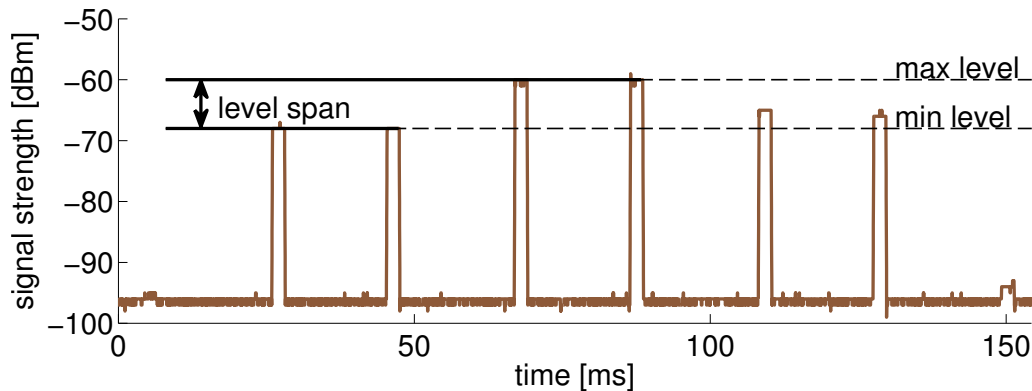


Figure 6.9: RSSI trace recorded with a TelosB during an active Wi-Fi scan of a nearby smartphone. Probes sent on adjacent channels exhibit a characteristic periodicity and difference in signal strength (level span) across channels.

Table 6.1: Explored signal features for signal cluster classification

Feature	Targeted aspect
Number of signals	Only consider clusters that contain a certain number of signals.
Number of different signal levels	Exploits variability introduced by sending frames in different channels.
Maximal difference of signal levels	Exploits variability introduced by sending frames in different channels.
Autocorrelation of RSSI trace	Periodicity of frames in a scan.
Distance to template scan	“Shape” of a scan pattern.

of 2 symbol periods and use an inter-cluster separation of $64 \mu\text{s}$ in signal length.

6.4.3 Feature Extraction

Next, DEV_{CNT} must decide whether a given cluster contains signals from a scan (Y) or not (N). To enable such classification, we require a set of features that (i) are expressive to reliably distinguish between (Y) and (N) clusters, and (ii) can be quickly computed on resource-constrained devices.

We explored various features (as listed in Table 6.1) and combinations thereof, and eventually settled on two features that could distinguish

clusters well in an offline classification setup. The first feature is based on the *autocorrelation*, and allows us to check for the presence of repeating patterns across the signals in a cluster. For example, when applied to the signals in Figure 6.9, this feature, f_p , can clearly identify a periodicity. The periodic pattern results because scans are a repeated sequence of probes sent in every channel. To distinguish between beacon frames sent by APs and active scans, we exclude the typical beacon frame period of 102.4 ms from our feature by only considering lags in the range [15, 95] ms.

The second feature, called *level span*, exploits that probes sent on adjacent Wi-Fi channels during a scan result in different signal levels at a ZigBee radio. As shown in Figure 6.9, we define the level span, f_l , as the difference between the highest and the lowest amplitude across all signals in a cluster to check for this kind of pattern. By applying a threshold on the product of both features, we can accurately distinguish between (Y) and (N) clusters, as shown in Figure 6.10 for data from an experiment with several interferers (see Section 6.6.1.1).

While the level span feature f_l can be quickly computed even on a resource-constrained platform, this does not hold for the periodicity feature f_p based on the autocorrelation, as also acknowledged by prior work [ZXX⁺13]. We explain in the following how we tackle this challenging problem in DEVCNT.

6.4.3.1 Sum of Autocorrelations

The periodicity feature f_p only cares about *when* signals in a cluster occur, and not about their amplitude. We thus represent a cluster as a discrete-time binary time series $\{x_i\}_1^w$, where x_i is 1 if and only if at time instant i a signal is present, and w is the size of a detection window. The autocorrelation ρ at lag τ is defined as

$$\rho(\tau) = \sum_{i=\tau+1}^w x_i x_{i-\tau}. \quad (6.1)$$

Checking whether a cluster exhibits a periodic pattern with a period in the interval $[a, b]$ entails computing the autocorrelation $\rho(\tau)$ using (6.1) for *each* lag $\tau \in \{a, a+1, \dots, b\}$. This, however, leads to prohibitive processing times on a resource-constrained platform.

The key insight we use to overcome this problem is that there is instead a way to efficiently compute the *sum* f_p of autocorrelations $\rho(\tau)$ over all lags τ in the interval $[a, b]$

$$f_p = \sum_{\tau=a}^b \rho(\tau). \quad (6.2)$$

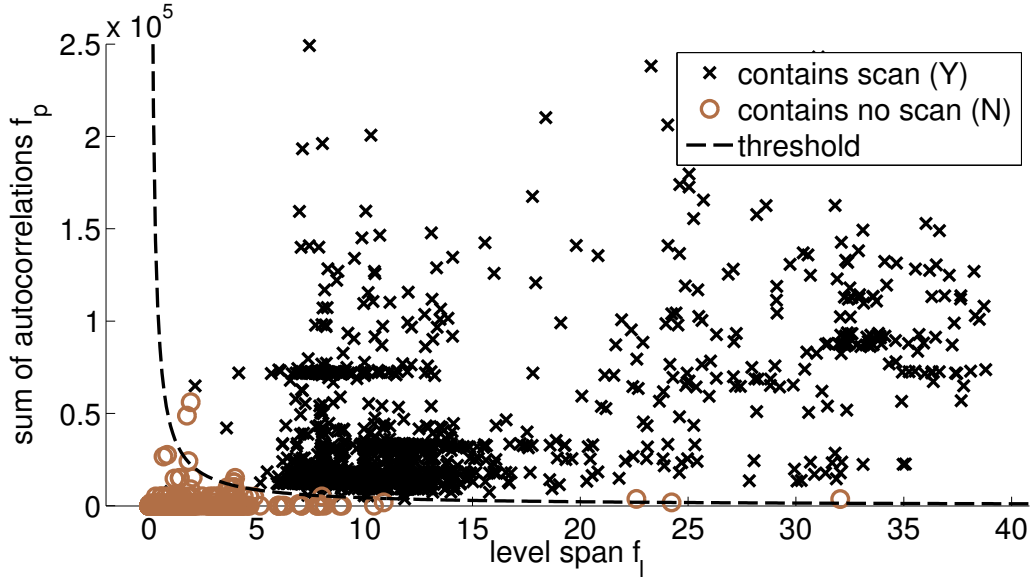


Figure 6.10: Illustration of classifying signal clusters into those containing an active scan (Y) and those containing no active scan (N), based on data from a real-world experiment with multiple interference sources. By applying a threshold on the product of the sum of autocorrelations feature f_p and the level span feature f_l , DEVCNT accurately classifies almost all clusters.

Intuitively, using the sum makes sense, because higher individual autocorrelations indicating periodicity result in a higher sum. While the inverse of this argument is not always true, empirical evidence from our real-world experiments shows that this approach is highly effective.

To efficiently compute the sum of autocorrelations feature f_p , we note that (6.2) can be transformed into

$$f_p = \sum_{j=1}^{w-a} \sum_{k=j-a}^{\min(w, j+b)} x_j x_k. \quad (6.3)$$

Crucially, (6.3) no longer iterates over individual lags τ : it essentially sums across the area that is bounded by a and b . Nevertheless, rather than summing up numerous "useless" 0's across the entire area, it is sufficient to only consider subareas containing 1's. The beginning and end of these subareas are precisely the x_i that mark the beginning and end of a period in which signals are present in a cluster. These observations materialize in an efficient algorithm for computing the sum of autocorrelations feature f_p . To this end, we define l as the increasingly ordered set of indexes of value changes in the binary vector X , that is, $l := \{i | x_i \neq x_{i+1}\}$. In the following, we use the notation l_s to refer to the element in l at position s .

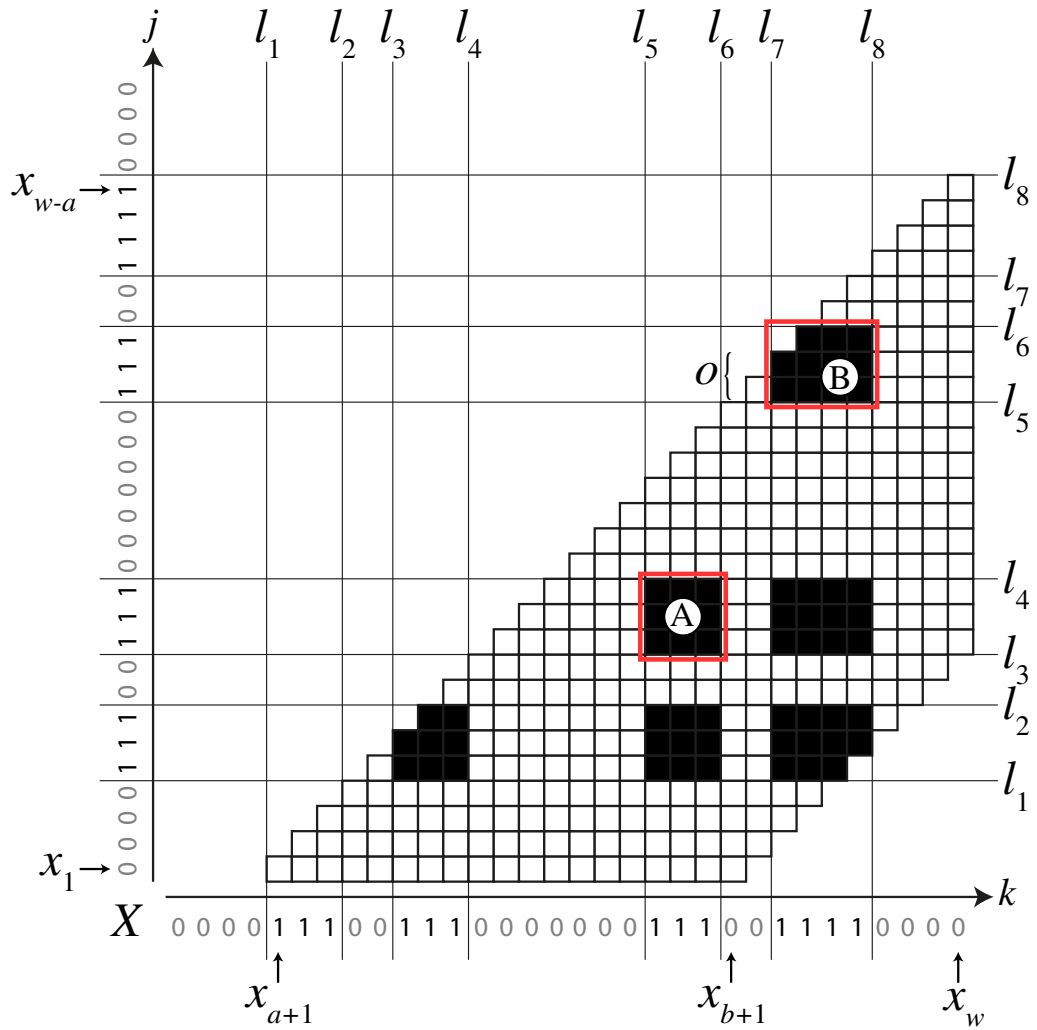


Figure 6.11: Sum of autocorrelations for an example binary vector X . The boxes represent all the $x_j x_k$ that are added up in (6.3). Black boxes are the elements that contribute to the sum. The area of adjacent black boxes can be calculated without iterating over every element therein.

Note that changes in X from 0 to 1 correspond to odd positions in l , while changes from 1 to 0 are at even positions.

Illustrative Example. We motivate our algorithm with the help of the example illustrated in Figure 6.11. Here, X contains four signals, which results in a set l of size 8, as shown on the vertical and the horizontal axes. The elements to be summed up, $x_j x_k$, are laid out in a 2-dimensional bitmap. Dark boxes indicate elements that contribute to the sum, that is, where both x_k and x_j are 1. The limits of the covered area are determined by $[a, b]$, the interval of the considered lags of the autocorrelation.

To compute the feature f_p , we add up the areas of (partial) black

rectangles formed by one or more adjacent black boxes. The area of the completely black rectangle $\textcircled{\text{A}}$ is the product of its length and width, that is, $\text{Area}(\textcircled{\text{A}}) = (l_6 - l_5)(l_4 - l_3)$. The area of partially black rectangles, such as $\textcircled{\text{B}}$, can also be calculated without iterating over every element by considering the offset o of the diagonal through the rectangle.

For our purpose, converting X into the set of value change indexes l reduces both space requirements and computational complexity: A detection window of 580 ms sampled with a resolution of 16 μs results in a binary vector X of 36250 bits (8.53 kB). According to our experiments, the number of signals in a detection window is rather small, and therefore the set l is significantly smaller than $|X|$. The reduced input size also reduces the complexity of the algorithm as we do not need to iterate over all samples in X , but only over the elements in l .

Algorithm and Pseudocode. Algorithm 2 calculates f_p . It iterates over both dimensions j and k using l , and sums up the rectangles delimited by the corners $(l_k, l_j)(l_{k+1}, l_{j+1})$. An odd position in l indicates the start of a signal, and an even position indicates the end of a signal. Thus, for every iteration, the index variables j and k are incremented by 2 to select the next rectangle. We distinguish three cases: (i) a fully contained rectangle (line 7), (ii) a rectangle partially outside on the lower end of k (line 11), and (iii) a rectangle partially outside the upper end of k (line 15). For (i) the number of elements is the product of the width and the height. For (ii) and (iii), the function `PartialRectangle()` shown in Algorithm 3 additionally uses the offset o to calculate the number of elements that fall into this partial rectangle. In each iteration, the contribution of the current rectangle is added to the final result f_p .

6.4.4 Classification

At the end of a detection window, each `DevCnt` node computes the two features above for each individual cluster and feeds them into a classification algorithm. If one or more clusters in a detection window are classified as containing signals from an active scan (Y), the node considers the whole detection window as containing a scan. As a result, it increments its local scan count, which it sends every reporting interval to the sink and then resets to zero.

Because fast processing is key in `DevCnt`, we opt for a computationally cheap decision tree classifier [DHS01]. For the same reason, instead of considering the two features separately, we use a threshold on their product for classification. We found this approach to be slightly more efficient in most of our tests without sacrificing classification accuracy. Therefore, the classification works on a decision tree with one branch

Algorithm 2 Compute the *Sum of Autocorrelations* Feature**Input:** l : indexes of level changes, a b : limits for lag**Output:** f_p : sum of autocorrelation between a and b

```

1:  $f_p \leftarrow 0$ 
2:  $j \leftarrow 1$ 
3: while  $j < |l|$  do
4:    $k \leftarrow 1$ 
5:   while  $k < |l|$  do
6:     if  $l_k \geq l_{j+1} + a - 1$  and  $l_{k+1} \leq l_j + b + 1$  then
7:        $f_p \leftarrow f_p + (l_{k+1} - l_k)(l_{j+1} - l_j)$ 
8:     else
9:       if  $l_k < l_{j+1} + a - 1$  and  $l_{k+1} > l_j + a$  then
10:         $o \leftarrow l_k - l_j - a + 1$ 
11:         $f_p \leftarrow f_p + \text{PartialRectangle}(l_{k+1} - l_k, l_{j+1} - l_j, o)$ 
12:      else
13:        if  $l_k > l_{j+1} + b$  and  $l_{k+1} > l_j + b + 1$  then
14:           $o \leftarrow l_{j+1} - l_{k+1} + b + 1$ 
15:           $f_p \leftarrow f_p + \text{PartialRectangle}(l_{k+1} - l_k, l_{j+1} - l_j, o)$ 
16:        end if
17:      end if
18:    end if
19:     $k \leftarrow k + 2$ 
20:  end while
21:   $j \leftarrow j + 2$ 
22: end while

```

Algorithm 3 $\text{PartialRectangle}(o, w_1, w_2)$ **Input:** w_1, w_2 : width and height of area, o : offset of diagonal**Output:** A : number of elements contained in area

```

1: if  $o > 0$  then
2:    $A_0 \leftarrow w_1 o$ 
3:    $d_1 \leftarrow w_1 - 1$ 
4: else
5:    $A_0 \leftarrow 0$ 
6:    $d_1 \leftarrow w_1 + o - 1$ 
7: end if
8:  $d_2 \leftarrow \max(1, w_1 - (w_2 - o))$ 
9:  $A \leftarrow A_0 + (d_1 + d_2)(d_1 - d_2 + 1)/2$ 

```

(i.e., one single *if*-statement) and incurs little runtime overhead. We use the `fitctree` function available in MATLAB to determine a threshold on the product of the two features, using a training set collected in a controlled experiment with several smartphones from different vendors and different interference sources, described in Section 6.6.1.1.

6.5 Implementation

With the help of the FlockLab testbed, we have implemented a `DevCnt` prototype on top of the Contiki operating system [DGV04]. Our prototype targets the TelosB platform, which features an 8 MHz MSP430 MCU, an IEEE 802.15.4-compliant 250 kbps low-power CC2420 radio, 10 kB of RAM, and 48 kB of program memory [PSC05]. The operating frequency of the radio can be programmed in steps of 1 MHz. We exploit this feature to tune the radio's operating frequency to the center frequency of a specific Wi-Fi channel.

Our `DevCnt` prototype uses a 580 ms detection window. Nodes report their scan counts with a reporting interval of 5 s, and the smartphone count estimations are based on a 3-minute counting window. At the end of each reporting interval, we allocate 122.5 ms for letting the sink first initiate a Glossy flood [FZTS11] to keep the nodes time-synchronized, and then collect a 1-byte scan count from each node using Chaos [LFZ13].

6.5.1 FLOCKLAB During the Development Process

In the following, we describe how we make use of FLOCKLAB services and other tools in order to support the development process of `DevCnt`. We do not provide a strict methodology on how to involve FLOCKLAB in the development process, but rather give some insights based on examples. More example use cases of FLOCKLAB are provided in Chapter 2, Section 2.4.

The `DevCnt` implementation can be divided into three tasks: (i) Implementation and verification the probe detection algorithm on a resource constrained device, (ii) integration and adaptation of Glossy and Chaos into a data collection service that suits the needs of `DevCnt`, and (iii) the combination of the probe detection part with the data collection into a working application.

Probe detection algorithm. Probe detection includes the processing chain described in Section 6.4. For design and implementation of this chain, we first record RSSI sample traces on a single node for later offline analysis. Based on these sample traces, we design the signal extraction step, and extract possible features to explore different machine learning algorithms. The chain is then implemented on the node. Crucial aspects of the implementation are the timing of the real-time signal extraction (see Section 6.4.1) and the overhead of the feature computation and probe classification (see Section 6.4.3). In this task, FLOCKLAB is used to verify a stable implementation under different interference patterns by exposing the prototype to an uncontrolled environment, such as an office building.

Orchestration of Glossy and Chaos. For the communication part of `DEVCNT`, we start off with two working base implementations for each of the two flooding primitives Glossy and Chaos. In `DEVCNT`, we combine both primitives to synchronize the network and to collect smartphone counts. One implementation difficulty stems from scheduling each flood consistently on all nodes. We have to deal with timer overflows, find good values for parameters like the number of retransmissions in a flood, or the duration of a flood.

For testing, we run the implementation in three different setups: (i) on a few nodes on the desk, (ii) on the Cooja simulator [ODE⁺06], and (iii) on `FLOCKLAB`. Testing on the desk provides a quick feasibility check of an implementation. In such a setup, data gathered from a serial port, a logic analyzer or an in-circuit debugger is used to inspect a test run. A simulator provides a good insight into the behavior of an application in a larger and arbitrary network topology. In later stages of the development process, a real multi-hop environment as in `FLOCKLAB` is used to make sure that the program is exposed to effects not accounted for in simulation (variations in clock speeds, multi-path fading of radio signals, dynamic environment, undocumented hardware features, hardware variations) nor on a small network on a desk (larger network, several hops, different node densities).

To debug scheduling issues, we find that `FLOCKLAB`'s GPIO tracing service is particularly useful. Due to the tight time synchronization of measurement data in `FLOCKLAB` (see Chapter 3 and Chapter 4), actions on different nodes can be accurately related to each other. We use Flooja [Büc14], a plugin to Cooja, to visualize and inspect measured data on `FLOCKLAB` (GPIO traces, power, and serial communication).

Verification of communication combined with probe detection. Once we have both, probe detection and communication implemented, we can start combining the two parts. Essential for reliable execution of `DEVCNT` is the isolation of communication and probe detection, since a node uses the same radio transceiver both for communication and as sensor to count probes. Therefore, we need to make sure that mode switches between communication and sensing work properly and timely, i.e., communication must reliably stop at the end of the communication interval, and probe detection must not extend signal acquisition into the communication phase.

For this last step towards the `DEVCNT` application, we mainly use TelosB nodes on a desk or in the testbed. To verify that communication and sensing do not overlap, we use again GPIO tracing in `FLOCKLAB`. To assess the overhead of our probe detection processing chain, we accumulate intervals of active CPU time on every node, and print out the overhead after every reporting interval. Timing and CPU overhead

can also be inspected by employing control flow tracing in FLOCKLAB, as presented in Chapter 5.

Summary. Based on our experience, the FLOCKLAB testbed infrastructure greatly enhances the development process of software for wireless embedded systems in various aspects by providing the necessary services and tools to inspect and observe different modalities on the actual hardware.

6.6 Evaluation

This section evaluates DEV_{CNT} in controlled experiments and a real-world trial. We start by investigating in Section 6.6.1 the accuracy with which DEV_{CNT} detects active Wi-Fi scans, with and without interference and depending on the distance between the smartphones and a ZigBee device. In Section 6.6.2, we assess the accuracy of DEV_{CNT}'s smartphone count estimations for different numbers of Wi-Fi enabled smartphones. Finally, in Section 6.6.3, we report on DEV_{CNT}'s performance during a short-term deployment in a large lecture hall.

6.6.1 Active Scan Detection Rate

We first evaluate the accuracy of scan detections, which is a key prerequisite to obtain accurate smartphone counts.

Setup. To avoid any bias in the measurements due to uncontrolled interference sources, we conduct these experiments in an environment where we verified with a spectrum analyzer that there is no interference in the 2.4 GHz ISM band. In particular, we conduct the indoor experiments in Section 6.6.1.1 and Section 6.6.1.2 in an underground garage, and the outdoor experiment in Section 6.6.1.2 in an open field.

We use six different smartphones that run three different versions of Android and iOS 7, as listed in Table 6.2. On Android phones, we install a dedicated application that triggers active Wi-Fi scans with a period of 20 s. Because a similar application is not available for iOS, we manually trigger active Wi-Fi scans on these two phones by retrieving the list of available APs.

We use one TelosB node connected to a laptop. The node reads out the RSSI register of the CC2420 radio at the IEEE 802.15.4 symbol rate of 62.5 kHz and logs them over the serial port. To obtain ground truth, we put the Wi-Fi card on the laptop in monitor mode and use Wireshark to log every observed Wi-Fi frame. Both the ZigBee radio and the Wi-Fi

Table 6.2: Smartphones and operating systems used in the experiments of Section 6.6.1.

Model	Operating system
Samsung Galaxy Nexus	Android 4.2.1
Samsung Galaxy S II	Android 4.1.2
Samsung Galaxy S3 Mini	Android 4.1.2
HTC Desire	Android 2.3.7
iPhone 4	iOS 7
iPhone 5	iOS 7

radio are tuned to an operating frequency of 2.422 GHz, corresponding to IEEE 802.11 channel 7.

Methodology and metric. We evaluate the performance of DEV_{CNT}'s signal processing pipeline (see Figure 6.2) in terms of *scan detection rate*, that is, the number of active Wi-Fi scans correctly detected by DEV_{CNT} from ZigBee RSSI traces over the number of active Wi-Fi scans in the Wireshark logs.

6.6.1.1 Impact of Interference

We first look at the robustness of the active scan detection rate against several typical interference sources in the 2.4 GHz band.

Setting. We consider five different interference settings in distinct 10-minute runs: (i) no interference, (ii) Wi-Fi TCP traffic, (iii) Wi-Fi UDP streaming, (iv) Bluetooth, and (v) ZigBee. We place the TelosB at a distance of 10 m from the smartphones. The interferers are 14 m and 10 m away from the TelosB and the smartphones, respectively.

For Wi-Fi settings (ii) and (iii), we associate a second laptop to an AP that also operates on channel 7. We generate TCP traffic by repeatedly accessing a web page with an HTTP client on this laptop. After each access, the HTTP client waits for a random interval between 1 and 5 s before it requests the next web page. We use Iperf to generate a UDP stream with a bit rate of 400 kbps, which is the rate of a typical Internet video stream.² We play music over a Bluetooth headset in setting (iv). In setting (v), we let another TelosB node transmit 30-byte packets with a random interval in [0.5, 1.5] s.

For every interference setting, we extract from the RSSI trace all sequences of the size of a detection window (580 ms) that contain a

²<https://support.google.com/youtube/answer/2853702>

Table 6.3: Scan detection performance with and without interference from various interference sources.

Interference setting	Scan detection rate (avg, std)
No interference	(100.0 %, 0.0 %)
Wi-Fi TCP traffic	(99.3 %, 1.3 %)
Wi-Fi UDP stream	(99.5 %, 0.9 %)
Bluetooth headset	(99.3 %, 1.7 %)
ZigBee device	(99.8 %, 0.6 %)

scan and label those sequences as class "contains a scan" (Y). To get a representative set of sequences that "contain no scan" (N), we extract the same number of sequences from random positions in the trace without a scan. For each sequence, we calculate the two features f_p and f_l used by our classifier (see Section 6.4.3). To assess the classification performance, we use 10-fold cross validation, each fold containing features from all five interference settings with similar frequency. In total, we evaluate 2784 sequences out of which 50 % contain an active Wi-Fi scan.

Results. Table 6.3 lists the scan detection rates achieved by DEV_{CNT} in the five interference settings. We see that DEV_{CNT} achieves an average accuracy above 99 % across the board. This shows that DEV_{CNT} reliably detects active scans despite interference from various common interference sources.

Despite this impressive accuracy, we note that Wi-Fi interference hardly presents a significant problem for DEV_{CNT} in a real deployment, because channel assignment in Wi-Fi production networks mostly focuses on a few non-overlapping channels [AJSS05]. Since probes are sent on each Wi-Fi channel during an active scan, tuning DEV_{CNT} to the center frequency of an unused Wi-Fi channel is therefore a viable option to reduce, or completely remove, the influence of Wi-Fi interference.

For the remaining experiments, we use the traces from this interference experiment to train our classifier, that is, to obtain the threshold on the product of the two features f_p and f_l .

6.6.1.2 Impact of Distance

Next, we study how the scan detection rate is affected by the distance between the smartphones and the DEV_{CNT} node.

Setting. We place the smartphones at different distances from the TelosB node. Outdoors, we check distances between 10 and 120 m; indoors, we are only able to go from 10 m up to 50 m due to the limited size of the

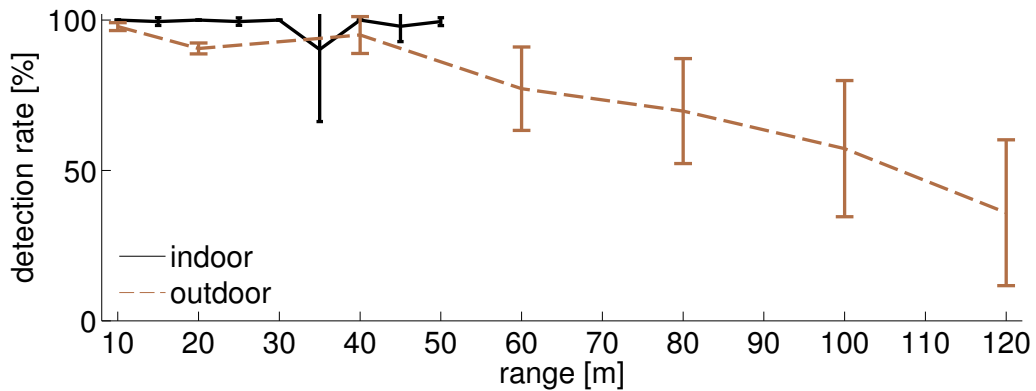


Figure 6.12: Average and standard deviation of scan detection rate in DEVCNT across six smartphones placed at different distances from a ZigBee device, measured both indoors and outdoors. DEVCNT can reliably detect active Wi-Fi scans from a smartphone that is approximately 50 m away.

underground garage. We place a second laptop running Wireshark next to the smartphones to capture all probes they emit, that is, the ground truth. We perform a 10-minute run at each distance.

Results. Figure 6.12 shows the average scan detection rate across all six smartphones as a function of their distance to the TelosB; error bars indicate the standard deviation. We see that the average scan detection rate is above 90 % up to a distance of 50 m, and shows very little variations between the different smartphones. The performance drop at 35 m in the indoor experiment is presumably due to multipath fading caused by the geometry of the underground garage. Beyond 50 m, the scan detection rate decreases steadily to 36 % at a distance of 120 m. We also note that the scan detection rate varies more between phones at larger distances: some smartphones have a larger Wi-Fi transmission range than others.

To further explain these results, we plot in Figure 6.13 the scan detection range against the received signal strength. We see a pronounced drop for signals below -85 dBm. This suggests that the scan detection rate largely depends on the signal levels.

In summary, we learn from these experiments that a single DEVCNT node is sufficient to reliably detect active scans from smartphones that roam about, for example, a large store. To cover areas that extend beyond 50 m, however, more DEVCNT nodes should be deployed to obtain accurate estimates.

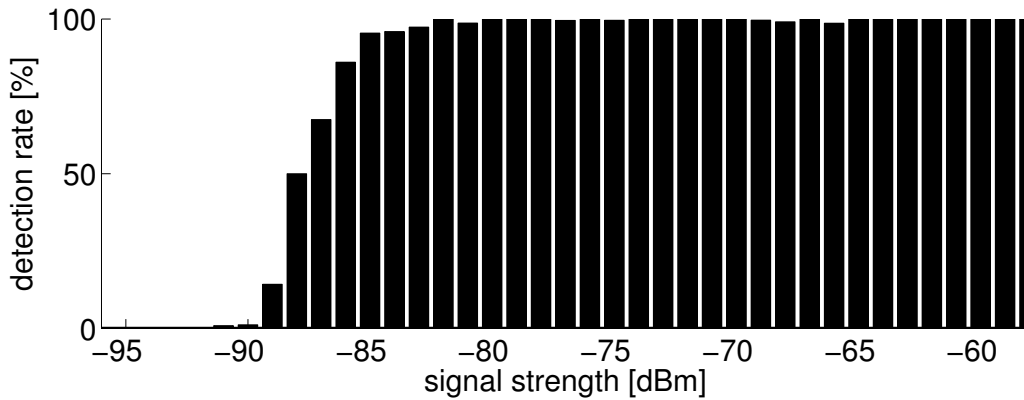


Figure 6.13: Average scan detection rate against received signal strength. *Active Wi-Fi scans with a received signal strength above -85 dBm are detected with a very high probability that is close to 100 %.*

6.6.2 Accuracy of Smartphone Count Estimations

In this experiment, we evaluate the accuracy of DEVCNT’s real-time smartphone count estimations.

Setting. We use again the underground garage to avoid any bias in our measurements due to uncontrolled interference. We place a TelosB node running DEVCNT in the middle of a 20×20 m area, and let it listen on Wi-Fi channel 6 to detect active scans. The node reports its scan counts every 5 seconds to a base station, where DEVCNT estimates the number of Wi-Fi enabled smartphones in the area. A laptop running Wireshark captures ground truth. In addition, we install one AP operating on channel 8 to mimic a realistic setup.

We use in total 31 smartphones, which run either iOS or Android. Including the smartphones from the previous experiments, there are 9 different models from 4 different vendors, representing a good mix of currently available smartphones. During the experiment, we change the number of smartphones inside the garage. Starting from 0, we add 10 phones after 15 min, another 10 after 30 min, and the remaining 11 after 45 min. Then, after 60 min, we start to actively use 10 of the 31 phones, unlocking the screen, scrolling through menus, or playing music. After 75 min, we stop using the phones. Finally, after 90 min, we start to remove phones: first a batch of 15 phones, and 15 min later the remaining 16 phones.

Meanwhile, DEVCNT estimates the number of Wi-Fi enabled smartphones every 5 seconds, based on the scan counts reported by the TelosB node. To this end, we use three different average scan rates for a 3-minute counting window: 2.70, 3.48, and 4.26. These correspond to the

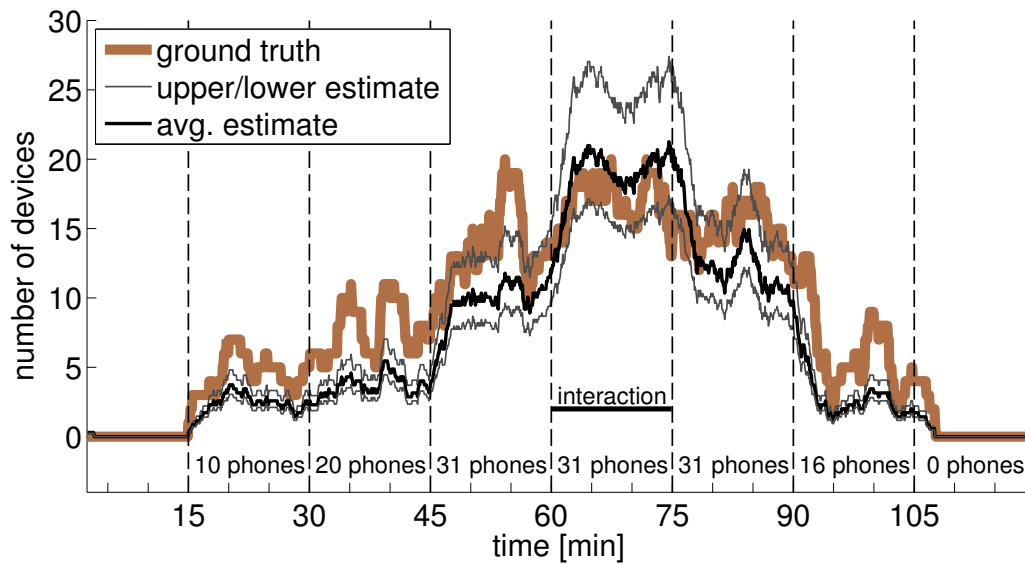


Figure 6.14: Estimated and real number of Wi-Fi enabled devices as smartphones are being added and removed over time. *DevCNT* provides accurate smartphone count estimates, achieving an average accuracy of up to 91 %.

lowest and highest average scan rates observed in the datasets shown in Figure 6.4 (3.48 is the average of 2.70 and 4.26). We compute the accuracy of the smartphone count estimations by comparing *DevCNT*'s estimates against ground truth obtained from the Wireshark logs when applying the same 3-minute counting window.

Results. Figure 6.14 plots *DevCNT*'s estimates and ground truth over time. We first note that the number of smartphones that are physically present inside the garage is roughly double the number of smartphones in the *ground truth*. We attribute this to the fact that several phones performed very few active scans in the experiment, with intervals much larger than the 3-minute counting window we use. In fact, 9 smartphones did issue less than 3 active scans during the whole experiment, predominately such with an Android version of 2.3.7 or lower. We could not expect this behavior based on our analysis of large real-world datasets in Section 6.3, as there is no information available on silent smartphones.

Nevertheless, we observe from Figure 6.14 that *DevCNT*'s estimates closely match ground truth as smartphones are being added and removed. When considering the average estimate, *DevCNT* achieves an accuracy of 68.9% throughout the entire 2-h experiment, which corresponds to an average absolute error of 3.0 smartphones. As mentioned earlier, we expect *DevCNT*'s estimates to be more accurate when the number of smartphones is higher. Our results confirm this expectation: Considering

the interval between 45 min and 75 min in which all 31 smartphones are present, and by taking into account the different activity patterns, `DEVCNT` achieves an average accuracy of 87.3 % (1.8 average absolute error) while all phones are in stand-by mode, and an average accuracy of 90.5 % (1.65 average absolute error) while 10 of the smartphones are active.

As one would expect, `DEVCNT` is less accurate than a Wi-Fi-based solution (such as [Cis14]), simply because it has less information at its disposal. In return, `DEVCNT` preserves by design the privacy of smartphone users, which is a strong asset when it comes to acceptance by law and the population [The13, The15]. Nevertheless, an accuracy of 70–90 % is sufficient for many applications we target, and comparable to what has been reported in the literature, for example, when counting smartphones using audio tones [KVC12] or when fingerprinting a Wi-Fi driver [FMT⁺06]. We thus conclude that `DEVCNT` provides accurate estimates on the number of Wi-Fi enabled smartphones if the mobility and usage profile of smartphones is known, which is a reasonable assumption in many applications [Cis14, HBJW05, The15].

Finally, we also logged performance counters throughout this experiment to study the processing overhead on the TelosB node. We find that the TelosB node was processing for only 1.7% of the time. This shows that our novel signal processing pipeline is amenable to an efficient implementation even on severely resource-constrained embedded devices.

6.6.3 Real-world Test Run

In a final real-world trial, we deploy `DEVCNT` in a lecture hall to show the applicability of the system in an uncontrolled environment. Such an environment presents a significant challenge for `DEVCNT` because of two reasons. First, in a larger room with many people carrying smartphones there are many more signals to process on the nodes and hence processing time might be high and reach the limits of the system. Second, the system is exposed to interference originating from different surrounding devices that might emit patterns that have not been considered in the training step of the classifier.

Setting. We install a multi-hop wireless network consisting of three `DEVCNT` nodes, a relay node, and a sink. We place two nodes inside the lecture hall, in the front and in the rear area; we put the third node outside at one of the two main entrances. The lecture hall has a size of 20 by 25 m. We observe APs on channels 1, 6, and 11. To minimize the interference from the APs, we let all `DEVCNT` nodes listen on Wi-Fi channel 8.

We let the system run from 11 AM to 1 PM, so we observe a morning

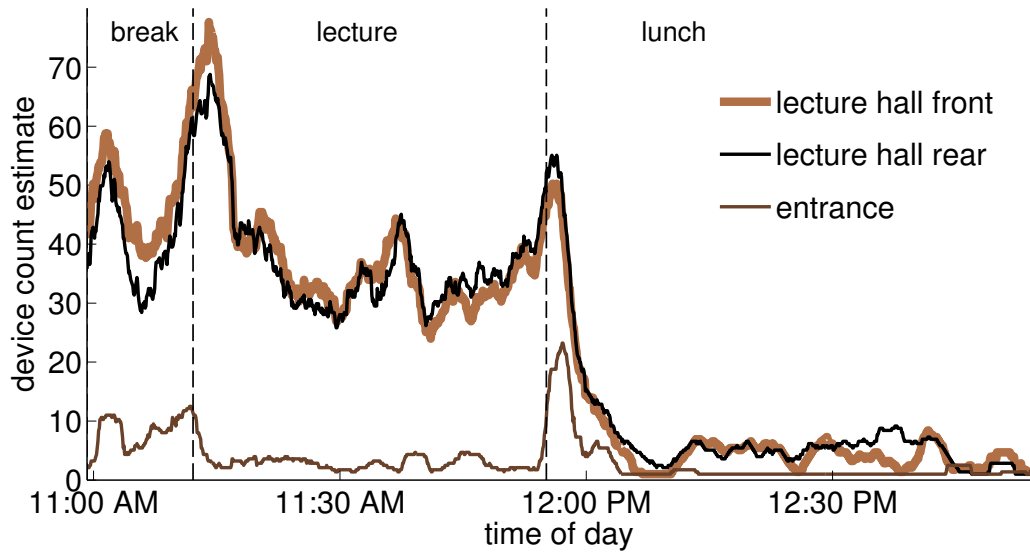


Figure 6.15: Estimated number of smartphones in a real world trial.

break, half a lecture, and a lunch break. At 11:30 AM we count by hand 111 students inside the lecture hall. While we also set up two laptops running Wireshark, we note that in this real-world setting *it is impossible to reliably determine the ground truth*. This is due to vastly different reception ranges of Wi-Fi and ZigBee radios: While a Wi-Fi receiver may be able to hear a probe from a weak sender (e.g., located in another room), probes from this sender on adjacent channels are often too weak to be heard by a ZigBee node at the same distance.

Results. Figure 6.15 shows the estimated smartphone counts over time for the 3 DEV_{CNT} nodes. Although we lack ground truth, we see that DEV_{CNT}'s estimates throughout the deployment closely match our expectations and visual on-site observations. For instance, during the initial break at about 11:05 AM there is a drop in the estimated smartphone counts, because a few students leave the lecture hall. The peaks at the beginning and at the end of the lecture are due to students using their phones more intensively, which leads to smartphones performing more active scans. Furthermore, as expected, DEV_{CNT}'s estimates remain fairly stable during the lecture, and afterward drop to numbers close to zero as almost all students leave for lunch. Looking at the node at the entrance, we see that it generally sees fewer smartphones, yet the periods where students enter or leave the hall before and after the lecture are clearly visible.

We further note that both nodes inside the lecture hall see about the same number of smartphones. This is in accordance with the findings in

Section 6.6.1.2: Since one `DEV_CNT` node can reliably detect devices within a radius of 50 m, a single node would have been sufficient to cover the entire lecture hall.

These results show that `DEV_CNT` can sustain signals from hundreds of Wi-Fi transmitters, as evident from our Wireshark logs, and delivers meaningful estimates in a real-world trial that resembles, for example, a retail or indoor concert setting.

6.7 Discussion

`DEV_CNT` estimates in real-time the number of Wi-Fi enabled smartphones within a given area. By detecting active Wi-Fi scans from RSSI traces, `DEV_CNT` obtains these counts in a fully-passive, non-invasive, and privacy-preserving manner.

Any system relying on externally observable properties for counting misses those smartphones that do not disclose these properties, and `DEV_CNT` is no exception. As such, like other solutions from academia [ME12] and industry [Cis14], `DEV_CNT` can only see phones that have Wi-Fi enabled and may count Wi-Fi transmitters other than phones. However, in the environments we target including open streets, shops, and train stations, the fraction of laptops and tablets is typically rather smaller.

`DEV_CNT` supports deployments across large areas through multi-hop communications. In those scenarios, multiple ZigBee devices may detect and then count the same smartphone. A practical approach to ameliorate this over-counting problem would be to carefully select the locations of the ZigBee devices so as to reduce areas of overlapping reception ranges. Another possibility would be to exploit the fact that `DEV_CNT` devices are time-synchronized, so an active scan that is detected by different devices at the same time likely originates from the same smartphone and could be accounted for only once. We intend to explore this idea in our future work.

`DEV_CNT` preserves the privacy of smartphone users as it cannot identify individual phones. While this is arguably a desirable property, it leaves `DEV_CNT` with no other option than to estimate the smartphone counts based on statistical information about the average active scan rate. `DEV_CNT` provides accurate estimates whenever the observed population of smartphones behaves according to the expectations, for example, in terms of their degree of mobility and how frequently the smartphones are being used. If phones behave sharply differently, however, `DEV_CNT`'s estimates may become less accurate. Nevertheless, we found in our tests

that phases of unusual behavior typically last for only a limited amount of time as visible, for example, in Figure 6.15 right after the break. In that sense, DEV_{CNT} is similar to participatory sensing approaches [WFMK⁺12], where the available GPS data fluctuate because smartphone users have full control over the application and are free to opt out at any time.

The flexibility of battery-powered nodes is not for free: To capture as many active scans as possible, all DEV_{CNT} nodes need to have their radio continuously turned on. In this case, a node powered by two AA batteries would last for a week, which is fine for short deployments (e.g., during a concert). One way to save energy would be to turn off the radio for extended periods of time when there is low Wi-Fi activity, such as during the night or outside of a shop's opening hours. We leave such energy considerations for future work.

Overall, DEV_{CNT} represents a new point in a multi-dimensional design space, trading some fidelity of the smartphone counts for full privacy of the smartphone users. Corresponding to this promising design point is a large number of application scenarios, ranging from crowd management [HBJW05] through public transport and event planning [CT10] to customer and visitor surveys [Cis14], where DEV_{CNT} could be highly beneficial.

6.8 Related work

Our work on DEV_{CNT} is related to prior efforts on leveraging the proliferation of smartphones for crowd counting and exploiting the interference between Wi-Fi and ZigBee.

Leveraging smartphones for crowd counting. Existing solutions employ different observable properties of a smartphone to count the number of people in an area or estimate the density of crowds. Such properties include audio tones [KVC12], GPS coordinates [WFMK⁺12], Bluetooth scans [WLBT14], and Wi-Fi probes [Cis14, ME12]. Conceptually, we can classify these solutions along three dimensions: privacy, invasiveness, and passiveness.

Both research [ME12] and commercial [Cis14] systems exist that directly eavesdrop on Wi-Fi probes, using existing APs and/or dedicated Wi-Fi monitors. Being able to demodulate and decode Wi-Fi frames, these systems can easily identify and track individual smartphones based on the unique MAC addresses embedded in each probe. Although anonymization techniques such as MAC address hashing are apparently used [Cis14], these systems may still be exploited (e.g., by an attacker) to compromise the privacy of the smartphone users, who possess no means

to “see” that they are being observed. Turning off Wi-Fi is therefore the only practical solution to guard against such impairment, but this may impact user experience [BEM⁺13b].

Another class of approaches requires to modify the smartphone itself, for example, by installing and running a dedicated application. The system presented in [KVC12] uses the built-in microphones to count smartphones by letting them exchange bit patterns encoded in audio tones. Others estimate crowd densities based on GPS data [WFMK⁺12] or the number of discovered devices by Bluetooth scans [WLBT14]. These approaches are invasive and rely on the voluntary and enduring cooperation of users to produce meaningful estimates. Finally, [ME12] shows that it is possible to solicit more probe transmissions from unmodified smartphones to improve tracking performance.

Unlike these prior works, *DEV_{CNT}* takes a fully-passive, non-invasive, and privacy-preserving counting approach. This approach relies on *DEV_{CNT}* taking advantage of interference between ZigBee and Wi-Fi, similar to other systems that are however designed for different purposes, as discussed next.

Exploiting interference between ZigBee and Wi-Fi. SoNIC classifies interference in the 2.4 GHz band into “Wi-Fi,” “microwave,” or “Bluetooth” based on RSSI information available on a ZigBee device [HRV⁺13]. SpeckSense and ZiFi exploit interference from Wi-Fi beacon frames, which are easier to detect than active scans because they exhibit a more rigid periodicity. SpeckSense processes RSSI information on a TelosB device in order to avoid Wi-Fi interference [IHV15]. ZiFi uses a built-in or external ZigBee radio to help a smartphone or laptop discover Wi-Fi APs in a more energy-efficient manner [ZXX⁺10]. Different from *DEV_{CNT}*, ZiFi benefits from ample resources of the host device compared to the limited memory and compute power of a low-power ZigBee mote. WizNet uses interference from probes, beacons, and other Wi-Fi traffic to monitor the spatio-temporal performance variations of Wi-Fi installations [ZXX⁺13]. Similar to *DEV_{CNT}*, WizNet uses, among other techniques, the discrete autocorrelation to identify probes from RSSI samples. However, unlike *DEV_{CNT}*, WizNet sends compressed RSSI traces to a more capable sink for computing the autocorrelation offline. *DEV_{CNT}* shows that active scan detection can indeed be performed online on mote-class devices, thereby reducing communication energy costs and bandwidth requirements by sending only the minimum amount of data to the sink.

6.9 Summary

We have presented `DEVCNT`, the first system that supports real-time counting of unmodified Wi-Fi enabled smartphones while preserving the privacy of the smartphone owners. Using novel signal processing algorithms that execute on a multi-hop network of ZigBee devices, `DEVCNT` detects and counts active Wi-Fi scans performed by smartphones based on characteristic patterns in RSSI traces. Combining these counts with statistical information about the average active scanning rate, `DEVCNT` faithfully estimates the number of Wi-Fi enabled smartphones. `DEVCNT` trades some fidelity in the smartphone count estimations for improved privacy. Results from controlled and real-world experiments show that `DEVCNT` provides estimates with an accuracy of up to 91%. We thus maintain that `DEVCNT` is a viable and promising solution for low-cost, real-time crowd counting in a broad spectrum of innovative applications.

7

Conclusions and Outlook

Wireless embedded systems have been around since more than a decade now. Ever since, people have been intrigued by the idea of cheap, long lasting, self-managing and self-healing networks of wireless sensor nodes, performing unattended sensing and actuation tasks. However, building such systems is difficult because of resource constraints (energy, processing, and memory), unreliable communication, or the distributed character of such systems.

To facilitate the development process of wireless embedded systems, testbed installations provide means to program and collect data from a network of sensor nodes. At the same time, testing and validating these systems on testbeds is notoriously impacted by limited observability and controllability. Initially, access to nodes in testbeds had been by serial port only, limiting debugging to inefficient printing of strings. Estimates of energy consumption, important to assess the energy efficiency, were done in software, or had been supported by low resolution measurements.

7.1 Contributions

To fill these gaps, we have made the following five main contributions in this thesis.

FLOCKLAB. We designed, implemented and deployed a new testbed that extends the state of current testbed architectures by means for multi-modal measurement and control abilities. FLOCKLAB can control and trace digital GPIO lines of a *target*, therefore enabling low-overhead

monitoring of devices under test. At the same time, services such as high resolution power profiling, serial logging, or target voltage control enrich the observability and controllability during a test run. These features, together with a wide range of different supported target platforms and public access using a web interface, have made FLOCKLAB popular in the research community. A high utilization of the testbed reflects the need for a multi-modal testing infrastructure.

Time synchronization. Given the need for accurately synchronized measurements in a testbed, we investigated limits of current clock synchronization protocols for wireless embedded systems. As a result, we presented TATS, a new synchronization protocol that combines propagation delay compensation and fast flooding to efficiently synchronize even large networks with tens of hops within sub-microsecond accuracy. We showed in testbed experiments that TATS outperforms two state-of-the-art time synchronization protocols (Glossy [FZTS11] and PulseSync [LSW14]) by up to $6.8\times$. In consequence, we proclaim that using a wireless multi-hop network to synchronize measurements in a distributed system is a viable alternative to wired (Ethernet) or GPS solutions.

FLOCKDAQ. To provision for extended tracing of GPIO events, and to incorporate better time synchronization into the testbed, we designed and implemented FLOCKDAQ. This data acquisition system, based on an FPGA paired with a wireless SoC, is capable of capturing GPIO events and power samples at the maximal rate of currently attached target nodes. In addition, all recorded data is annotated with a $1\ \mu\text{s}$ accurate time stamp. We achieve this by employing a hierarchical memory architecture and a wireless multi-hop time synchronization protocol.

Control flow tracing. To help developers instrument their code, we introduced a systematic approach that can automatically place tracing statements into a given program. Based on the extracted control flow of a program and execution time information, our new algorithm determines where to put tracing statements (witnesses) to faithfully determine the executed program path by processing the recorded witness IDs and time. By incorporating time information, we can substantially reduce the runtime overhead induced by the instrumentation. We showed the feasibility and usefulness of our approach by implementing the algorithm in a tool for MSP430 based platforms and showing the runtime overhead for various applications in FLOCKLAB.

DEVcnt. Finally, we demonstrated the benefits of a multi-modal testbed during the development process in the case of DEVcnt. DEVcnt tries to count the number of smartphones based on probe scans that are detected

on a ZigBee receiver. Since these receivers cannot decode Wi-Fi frames, the privacy of smartphones users is preserved. We developed signal processing methods to extract probe scans from RSSI readings and used machine learning techniques to distinguish probe scans from other traffic or interference.

7.2 Possible Future Directions

The contributions in this thesis significantly advance the possibilities to observe and control wireless embedded systems in a testbed. Besides further improvements in terms of observability, we envision also possible future research directions that are based on data gathered using testbeds.

Other modalities. An interesting path to explore is to augment a testbed by adding other measurement modalities. In particular, capturing activities in the wireless channel during an experiment could help to better put test results into context. Interference from non-decodable radio technologies can be made visible, and as such e.g. explain packet losses. A software defined radio or a spectrum analyzer could capture the baseband signal of overlapping transmissions, providing means to characterize the capture effect or assess the possibility of constructive interference. Besides enabling such measurements in a testbed, we see also the challenge to process and record such data in an efficient way.

Analyzing large data sets. Related to the increasing possibilities in observing embedded systems in a testbed is the problem of making use of that data. While log files containing strings might be quickly examined and processed using simple scripts, GPIO traces or control flow traces of several nodes in a network quickly grow to data volumes that require more processing power and methods to effectively extract the information of interest. Possible avenues to explore are:

1. For quick inspection, tools that facilitate browsing of different kinds of traces are required. An example of such a tool is Floopja [Büc14], which visualizes different measurements acquired in FLOCKLAB. Taking this approach further, we envision a tool to browse control flow traces captured on several nodes in a network.
2. The use of data mining techniques to automatically search for interesting patterns in testbed data seems to be a promising idea. Interesting patterns might be those that are very different to others, so called discords [KLF05]. Such techniques could assist a developer by pointing to rare and possible false behavior.

Applications for probe detection using ZigBee radios. Although not at the core of this thesis, we find that the interaction of different wireless technologies bears interesting research questions. Here, we pitch the idea of using a ZigBee receiver as wake-up radio for Wi-Fi access points. Due to the fact that ZigBee receivers are more energy efficient, they could be used to only turn on an access point if there is an actual client actively probing. This idea goes into a similar direction as wake-up radios for wireless sensor networks [SBBT15].

Bibliography

- [AFC16] M. P. Andersen, G. Fierro, and D. E. Culler. System design for a synergistic, low power mote/BLE embedded platform. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks (IPSN)*, 2016.
- [AJSS05] A. Akella, G. Judd, S. Seshan, and P. Steenkiste. Self-management in chaotic wireless deployments. In *Proceedings of the 11th International Conference on Mobile Computing and Networking (MobiCom)*, 2005.
- [ARM11] Embedded trace macrocell, architecture specification, 9 2011. Issue Q.
- [BBV07] A. Barberis, L. Barboni, and M. Valle. Evaluating energy consumption in wireless sensor networks applications. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, 2007.
- [BCP03] G. Bernat, A. Colin, and S. Petters. pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Technical report, University of York, 2003.
- [BEM⁺13a] M. V. Barbera, A. Epasto, A. Mei, S. Kosta, V. C. Perta, and J. Stefa. CRAWDAD data set sapienza/probe-requests (v. 2013-09-10). Downloaded from <http://crawdad.org/sapienza/probe-requests/>, September 2013.
- [BEM⁺13b] M. V. Barbera, A. Epasto, A. Mei, V. C. Perta, and J. Stefa. Signals from the crowd: Uncovering social relationships through smartphone probes. In *Proceedings of the Internet Measurement Conference (IMC)*, 2013.
- [BL94] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994.
- [BL96] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1996.

- [BMB10] A. Betts, N. Merriam, and G. Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.
- [Büc14] A. Büchel. Floopja: Visualisierung von FlockLab-Testbed-Daten. Semester project report, ETH Zurich, 2014.
- [BVJ⁺10] S. Bouckaert, W. Vandenberghe, B. Jooris, I. Moerman, and P. Demeester. The w-iLab.t testbed. In *Proceedings of the 6th ICST International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, 2010.
- [BvRW07] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: Ultra-low power data gathering in sensor networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.
- [Cis14] Cisco Systems. White paper: CMX Analytics, April 2014.
- [CKJL09] J. I. Choi, M. A. Kazandjieva, M. Jain, and P. Levis. The case for a network protocol isolation layer. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2009.
- [con] The Contiki operating system. <http://www.contiki-os.org/>.
- [CPC⁺12] 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*, 55(1), 2012.
- [CRM⁺08] K. Chebrolu, B. Raman, N. Mishra, P. K. Valiveti, and R. Kumar. Brimon: A sensor network system for railway bridge monitoring. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2008.
- [CT10] C. C. Cheong and R. To. Household interview surveys from 1997 to 2008—a decade of changing travel behaviours, May 2010. Online at <http://goo.gl/aw3WFV>.
- [DBK⁺07] M. Dyer, J. Beutel, T. Kalt, P. Oehen, L. Thiele, K. Martin, and P. Blum. Deployment support network: A toolkit for the development of WSNs. In *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN)*, 2007.
- [DEE03] M. Delvai, U. Eisenmann, and W. Elmenreich. Intelligent UART module for real-time applications. In *Proceedings of the 1st Workshop on Intelligent Solutions in Embedded Systems (WISES)*, 2003.

- [DFPC08] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN)*, 2008.
- [DGV04] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors (Emnets)*, 2004.
- [DHS01] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2001.
- [Dij97] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.
- [DOTH07] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the 4th IEEE Workshop on Embedded Networked Sensors (EmNets)*, 2007.
- [EAR⁺06] E. Ertin, A. Arora, R. Ramnath, V. Naik, S. Bapat, V. Kulathumani, M. Sridharan, H. Zhang, H. Cao, and M. Nesterenko. Kansei: A testbed for sensing at scale. In *Proceedings of the 5th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2006.
- [EGE02] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.
- [Eri14] Ericsson AB. Ericsson mobility report, June 2014.
- [FDLS08] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating systems design and implementation (OSDI)*, 2008.
- [FMT⁺06] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. Van Randwyk, and D. Sicker. Passive data link layer 802.11 wireless device driver fingerprinting. In *Proceedings of the 15th USENIX Security Symposium (SS)*, 2006.
- [Fur00] S. B. Furber. *ARM system-on-chip architecture*. Pearson Education, 2000.
- [FZMT12] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems, SenSys '12*, pages 1–14, New York, NY, USA, 2012. ACM.

- [FZMT13] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Virtual synchrony guarantees for cyber-physical systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 20–30, Sept 2013.
- [FZTS11] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with Glossy. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2011.
- [GES⁺04] L. Girod, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proceedings of the USENIX Annual Technical Conference*, 2004.
- [GFJ⁺09] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2009.
- [GKS03] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.
- [GZF⁺07] L. Gasparini, O. Zadedyurina, G. Fontana, D. Macii, A. Boni, and Y. Ofek. A digital circuit for jitter reduction of GPS-disciplined 1-pps synchronization signals. In *Advanced Methods for Uncertainty Estimation in Measurement, 2007 IEEE International Workshop on*, 2007.
- [HBJW05] D. Helbing, L. Buzna, A. Johansson, and T. Werner. Self-organized pedestrian crowd dynamics: Experiments, simulations, and design solutions. *Transportation Science*, 39(1):1–24, 2005.
- [HC02] J. L. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6), 2002.
- [HHP⁺08] I. Haratcherev, G. Halkes, T. Parker, O. Visser, and K. Langendoen. PowerBench: A scalable testbed infrastructure for benchmarking power consumption. In *Proceedings of the International Workshop on Sensor Network Engineering (IWSNE)*, 2008.
- [HKWW06] V. Handziski, A. Köpke, A. Willig, and A. Wolisz. TWIST: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proceedings of the 2nd ACM International Workshop on Multi-hop Ad Hoc Networks (REALMAN)*, 2006.
- [HNL08] M. Healy, T. Newe, and E. Lewis. Wireless sensor node hardware: A review. In *Sensors, 2008 IEEE*, 2008.

-
- [HRV⁺13] F. Hermans, O. Rensfelt, T. Voigt, E. Ngai, L.-A. Norden, and P. Gunningberg. SoNIC: Classifying interference in 802.15.4 sensor networks. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks (IPSN)*, 2013.
- [HSL10] W. Huangfu, L. Sun, and J. Liu. A high-accuracy nonintrusive networking testbed for wireless sensor networks. *EURASIP Journal on Wireless Communications and Networking*, 2010(1):1, 2010.
- [HTWM04] W. Hu, T. Tan, L. Wang, and S. Maybank. A survey on visual surveillance of object motion and behaviors. *IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews*, 34(3):334–352, 2004.
- [HWM10] A. Hergenröder, J. Wilke, and D. Meier. Distributed energy measurements in WSN testbeds with a sensor node management device (SNMD). In *Workshop Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS)*, 2010.
- [HYW⁺06] J. Heidemann, W. Ye, J. Wills, A. Syed, and Y. Li. Research challenges and applications for underwater sensor networking. In *Wireless Communications and Networking Conference (WCNC)*. IEEE, volume 1, 2006.
- [IHV15] V. Iyer, F. Hermans, and T. Voigt. Detecting and avoiding multiple sources of interference in the 2.4 GHz spectrum. In *Proceedings of the 12th International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2015.
- [Int13] Intel architecture, instruction set extensions programming, reference, 12 2013. 319433-017.
- [J⁺11] R. Jurdak et al. Opal: A multiradio platform for high throughput wireless sensor networks. *IEEE Embedded Systems Letters*, 3, 2011.
- [JDCS07] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.
- [JGMdDO10] A. Jiménez-González, J. Martínez-de Dios, and A. Ollero. An integrated testbed for heterogeneous mobile robots and other cooperating objects. In *Proceedings of the 23rd IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.
- [KBT12] M. Keller, J. Beutel, and L. Thiele. How was your journey?: Uncovering routing dynamics in deployed sensor networks with multi-hop network tomography. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems, SenSys '12*, pages 15–28, New York, NY, USA, 2012. ACM.

- [KBT13] M. Keller, J. Beutel, and L. Thiele. The problem bit. In *2013 IEEE International Conference on Distributed Computing in Sensor Systems*, pages 105–114, May 2013.
- [KDL⁺06] B. Kusy, P. Dutta, P. Levis, M. Maroti, A. Ledeczi, and D. Culler. Elapsed time on arrival: A simple and versatile primitive for canonical time synchronisation services. *Int. J. Ad Hoc Ubiquitous Comput.*, 1(4):239–251, July 2006.
- [KLA⁺08] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2008.
- [KLF05] E. Keogh, J. Lin, and A. Fu. Hot sax: efficiently finding the most unusual time series subsequence. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 8 pp.–, Nov 2005.
- [KLL⁺10] M. M. H. Khan, H. K. Le, M. LeMay, P. Moinzadeh, L. Wang, Y. Yang, D. K. Noh, T. Abdelzaher, C. A. Gunter, J. Han, and X. Jin. Diagnostic powertracing for sensor node failure analysis. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, 2010.
- [KPC⁺07] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.
- [KVC12] P. G. Kannan, S. P. Venkatagiri, and M. C. Chan. Low cost crowd counting using audio tones. In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2012.
- [Lar99] J. R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5):259–269, May 1999.
- [LCTL00] C. C. D. Loh, C. Y. Cho, C. P. Tan, and R. S. Lee. Identifying unique devices through wireless fingerprinting. In *Proceedings of the 1st ACM Conference on Wireless Network Security (WiSec)*, 2000.
- [LF76] K. Leentvaar and J. Flint. The capture effect in FM receivers. *Communications, IEEE Transactions on*, 24(5):531–539, May 1976.
- [LFZ13] O. Landsiedel, F. Ferrari, and M. Zimmerling. Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2013.
- [LLL⁺08] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong. Passive diagnosis for wireless sensor networks. In *ACM SenSys*, 2008.

-
- [LMT16] R. Lim, B. Maag, and L. Thiele. Time-of-flight aware time synchronization for wireless embedded systems. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, pages 149–158, USA, 2016. Junction Publishing.
- [LPLT10] C.-J. M. Liang, N. B. Priyantha, J. Liu, and A. Terzis. Surviving Wi-Fi interference in low power ZigBee networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.
- [LST15] O. Landsiedel, E. M. Schiller, and S. Tomaselli. LibReplay: Deterministic replay for bug hunting in sensor networks. In *Proceedings of the 12th European Conference on Wireless Sensor Networks (EWSN)*, 2015.
- [LSW09] C. Lenzen, P. Sommer, and R. Wattenhofer. Optimal clock synchronization in networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2009.
- [LSW14] C. Lenzen, P. Sommer, and R. Wattenhofer. PulseSync: An efficient and scalable clock synchronization protocol. *ACM/IEEE Transactions on Networking (TON)*, Mar 2014.
- [LWG05] O. Landsiedel, K. Wehrle, and S. Götz. Accurate prediction of power consumption in sensor networks. In *Proceedings of the 2nd Workshop on Embedded networked sensors (EmNets)*, 2005.
- [ME12] A. B. M. Musa and J. Eriksson. Tracking unmodified smartphones using Wi-Fi monitors. In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2012.
- [MEM11] MEMSIC. *TelosB Mote Platform*, 2011. Rev A.
- [MGHC07] H. Mach, E. Grim, O. Holmeide, and C. Calley. PTP enabled network for flight test data acquisition and recording. In *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS)*, 2007.
- [MKSL04] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *ACM SenSys*, 2004.
- [MMBK10] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, 2010.
- [MMJ⁺05] A. Milenkovic, M. Milenkovic, E. Jovanov, D. Hite, and D. Raskovic. An environment for runtime power monitoring of wireless sensor network platforms. In *Proceedings of the 37th Southeastern Symposium on System Theory (SSST)*, 2005.

- [OC10] J. Ortiz and D. Culler. Multichannel reliability assessment in real world WSNs. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, 2010.
- [ODE⁺06] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *Proceedings of the 7th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp)*, 2006.
- [OLI13] OLIMEX Ltd. *MSP430-CCRF development board: User's manual*, 2013. Revision C.
- [OW10] M. Okola and K. Whitehouse. Unit testing for wireless sensor networks. In *Proceedings of the on Software Engineering for Sensor Network Applications (SESENA)*, 2010.
- [pan] Panstamp NRG 2. <http://panstamp.com/>.
- [PBMS14] A. Pötsch, A. Berger, G. Möstl, and A. Springer. TWECIS: A testbed for wireless energy constrained industrial sensor actuator networks. In *Proceedings of the 19th International Conference on Emerging Technology and Factory Automation (ETFA)*, 2014.
- [per] The Permasense project. <http://www.permasense.ch>.
- [PHC04] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.
- [PPTDS10] N. Penneman, L. Perneel, M. Timmerman, and B. De Sutter. An FPGA-based real-time event sampler. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 5992, pages 364–371. 2010.
- [PSC05] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN)*, 2005.
- [ptp08] IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages c1–269, July 2008.
- [PY99] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, 1999.
- [RCK⁺05] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

- [RGD⁺15] J. Robert, J.-P. Georges, T. Divoux, P. Miramont, and B. Rmili. On the observability in switched Ethernet networks in the next generation of space launchers: Problem, challenges and recommendations. In *Proceedings of the 7th International Conference on Advances in Satellite and Space Communications (SPACOMM)*, 2015.
- [RHLG10] O. Rensfelt, F. Hermans, L.-Å. Larzon, and P. Gunningberg. Sensei-UU: A relocatable sensor network testbed. In *Proceedings of the 5th ACM International Workshop on Wireless network testbeds, experimental evaluation and characterization (WiNTECH)*, 2010.
- [RM09] K. Römer and J. Ma. Passive distributed assertions for sensor networks. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN)*, 2009.
- [Rom92] S. Roman. *Coding and information theory*, volume 134. Springer Science & Business Media, 1992.
- [SBBT15] F. Sutton, B. Buchli, J. Beutel, and L. Thiele. Zippy: On-demand network flooding. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys '15*, pages 45–58, New York, NY, USA, 2015. ACM.
- [SDFG⁺17] F. Sutton, R. Da Forno, D. Gschwend, T. Gsell, R. Lim, J. Beutel, and L. Thiele. A system design methodology for adaptive, responsive and energy-efficient wireless sensing. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, 2017.
- [SDS10] T. Schmid, P. Dutta, and M. B. Srivastava. High-resolution, low-power time synchronization an oxymoron no more. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, 2010.
- [SEZ10] V. Sundaram, P. Eugster, and X. Zhang. Efficient diagnostic tracing for wireless sensor networks. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2010.
- [SH⁺06] A. A. Syed, J. S. Heidemann, et al. Time synchronization for high latency acoustic networks. In *Proceedings of the 25th Conference on Computer Communications (INFOCOM)*, 2006.
- [SH15] A. R. Swain and R. Hansdah. A model for the classification and survey of clock synchronization protocols in WSNs. *Ad Hoc Netw.*, 27(C):219–241, April 2015.
- [SHC⁺04] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network

- applications. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.
- [SK13] P. Sommer and B. Kusy. Minerva: Distributed tracing and debugging in wireless sensor networks. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2013.
- [SMK08] T. Stathopoulos, D. McIntire, and W. J. Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN)*, 2008.
- [SSC10] R. Shea, M. Srivastava, and Y. Cho. Scoped identifiers for efficient bit aligned logging. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2010.
- [Sta14] J. A. Stankovic. Research directions for the internet of things. *IEEE Internet of Things Journal*, 1(1):3–9, Feb 2014.
- [STG07] C. Sharp, M. Turon, and D. Gay. Tinyos enhancement proposal 102: Timers. <http://www.tinyos.net/>, September 2007.
- [SW09] P. Sommer and R. Wattenhofer. Gradient clock synchronization in wireless sensor networks. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN)*, 2009.
- [SZDF⁺15] F. Sutton, M. Zimmerling, R. Da Forno, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel, and L. Thiele. Bolt: A stateful processor interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 267–280, New York, NY, USA, 2015. ACM.
- [Tex06] Texas Instruments. *MSP430x1xx Family User's Guide*, 2 2006. Rev. F.
- [Tex14] Texas Instruments. CC2420 datasheet, 2014.
- [TH02] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, 2002.
- [THBR11] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2011.

-
- [The10] The New York Times. Stampede at german music festival kills 18, July 2010. Online at <http://www.nytimes.com/2010/07/25/world/europe/25germany.html>.
- [The13] The Guardian. City of London Corporation wants 'spy bins' ditched, August 2013. Online at <http://www.theguardian.com/world/2013/aug/12/city-london-corporation-spy-bins>.
- [The15] The Local Denmark. Copenhagen to roll out new smart traffic system, February 2015. Online at <http://www.thelocal.dk/20150202/copenhagen-to-roll-out-new-smart-traffic-systems>.
- [TLF16] F. Terraneo, A. Leva, and W. Fornaciari. Demo: A high-performance, energy-efficient node for a wide range of WSN applications. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2016.
- [TSBE15] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster. Tardis: Software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN)*, 2015.
- [TVW16] P. Tuset-Peiró, X. Vilajosana, and T. Watteyne. OpenMote+: a range-agile multi-radio mote. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2016.
- [TW08] T. Trathnigg and R. Weiss. A runtime energy monitoring system for wireless sensor networks. In *Proceedings of the 3rd International Symposium on Wireless Pervasive Computing (ISWPC)*, 2008.
- [u-b14] u-blox. *LEA-6 data sheet*, 2014. R10.
- [VTWP15] X. Vilajosana, P. Tuset, T. Watteyne, and K. Pister. OpenMote: Open-source prototyping platform for the industrial IoT. In *Proceedings of the 7th International Conference on Ad Hoc Networks (AdHocHets)*, 2015.
- [WASW05] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Proceedings of the 4th ACM/IEEE International Symposium on Information Processing in Sensor Networks (IPSN)*, 2005.
- [WC13] L. Wan and Q. Cao. Towards instruction level record and replay of sensor network applications. In *Proceedings of the 21th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2013.

- [WEE⁺08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.
- [WFMK⁺12] M. Wirz, T. Franke, E. Mitleton-Kelly, D. Roggen, P. Lukowicz, and G. Tröster. CoenoSense: A framework for real-time detection and visualization of collective behaviors in human crowds by tracking mobile devices. In *Proceedings of the European Conference on Complex Systems (ECCS)*, 2012.
- [WLBT14] J. Weppner, P. Lukowicz, U. Blanke, and G. Tröster. Participatory Bluetooth scans serving as urban crowd probes. *IEEE Sensors Journal*, 14(12):4196–4206, 2014.
- [WLT09] M. Woehrle, K. Lampka, and L. Thiele. Exploiting timed automata for conformance testing of power measurements. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, 2009.
- [Woe10] M. Woehrle. *Testing of wireless sensor networks*. PhD thesis, ETH Zurich, 2010.
- [YSSW07] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.
- [ZCH11] Z. Zhong, P. Chen, and T. He. On-demand time synchronization with predictable accuracy. In *Proceedings of the 30th International Conference on Computer Communications (INFOCOM)*, 2011.
- [ZFM⁺12] M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele. pTunes: Runtime parameter adaptation for low-power MAC protocols. In *Proceedings of the 11th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2012.
- [ZFMT13] M. Zimmerling, F. Ferrari, L. Mottola, and L. Thiele. On modeling low-power wireless protocols based on synchronous packet transmissions. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 546–555, Aug 2013.
- [ZMK⁺] M. Zimmerling, L. Mottola, P. Kumar, F. Ferrari, and L. Thiele. Adaptive real-time communication for wireless cyber-physical

systems. To appear in *ACM Transactions on Cyber-Physical Systems*.

- [ZXX⁺10] R. Zhou, Y. Xiong, G. Xing, L. Sun, and J. Ma. Zifi: Wireless LAN discovery via ZigBee interference signatures. In *Proceedings of the 16th International Conference on Mobile Computing and Networking (MobiCom)*, 2010.
- [ZXX⁺13] R. Zhou, G. Xing, X. Xu, J. Wang, and L. Gu. WizNet: A ZigBee-based sensor system for distributed wireless LAN performance monitoring. In *International Conference on Pervasive Computing and Communications (PerCom)*, 2013.

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. **Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems.** In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks (IPSN)*. Philadelphia, Pennsylvania, USA, Apr 2013. (Chapter 2)

R. Lim, M. Zimmerling, and L. Thiele. **Passive, privacy-preserving real-time counting of unmodified smartphones via zigbee interference.** In *Proceedings of the 11th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*. Fortaleza, Brazil, Jun 2015. (Chapter 6)

R. Lim, B. Maag, B. Dissler, J. Beutel, and L. Thiele. **A testbed for fine-grained tracing of time sensitive behavior in wireless sensor networks.** In *Proceedings of the 40th IEEE Conference on Local Computer Networks, Workshops (SenseApp)*. Clearwater, FL, USA, Oct 2015. *Best paper*. (Chapter 4)

R. Lim, B. Maag, and L. Thiele. **Time-of-flight aware time synchronization for wireless embedded systems.** In *Proceedings of the 13th International Conference on Embedded Wireless Systems and Networks (EWSN)*. Graz, Austria, Feb 2016. (Chapter 3)

R. Lim and L. Thiele. **Testbed assisted control flow tracing for wireless embedded systems.** *Under Submission at the 14th International Conference on Embedded Wireless Systems and Networks (EWSN)* Uppsala, Sweden, Feb 2017. (Chapter 5)

The following list includes publications that are not part of this thesis.

M. Keller, M. Woehrle, R. Lim, J. Beutel, and L. Thiele. **Comparative performance analysis of the permadozer protocol in diverse deployments.** In *Proceedings of the 6th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp)*. Bonn, Germany, Oct 2011.

Y.-H. Chiang, M. Keller, R. Lim, P. Huang, and J. Beutel. **Poster abstract: Light-weight network health monitoring.** In *Proceedings of the 11th International Conference on Information Processing in Sensor Networks (IPSN)*. Beijing, China, Apr 2012.

L. Girard, J. Beutel, S. Gruber, J. Hunziker, R. Lim, and S. Weber. **A custom acoustic emission monitoring system for harsh environments: application to freezing-induced damage in alpine rock-walls.** In *Geoscientific Instrumentation, Methods and Data Systems*. Nov 2012.

R. Lim, C. Walser, F. Ferrari, M. Zimmerling, and J. Beutel. **Demo abstract: Distributed and synchronized measurements with flocklab.** In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems (SenSys)*. Toronto, Canada, Nov 2012.

R. Lim. **Poster abstract: Tracking smartphones using low-power sensor nodes.** In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys)*. Rome, Italy, Nov 2013.

M. Zimmerling, F. Ferrari, R. Lim, O. Saukh, F. Sutton, R. Da Forno, R. S. Schmidt, and M. A. Wyss. **Poster abstract: A reliable wireless nurse call system: Overview and pilot results from a summer camp for teenagers with duchenne muscular dystrophy.** In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys)*. Rome, Italy, Nov 2013.

S. Weber, J. Beutel, B. Buchli, S. Gruber, T. Gsell, R. Lim, P. Limpach, H. Raetzo, Z. Su, F. Sutton, C. Walser, and V. Wirz. **PermaSense L1-GPS for kinematic monitoring.** In *Book of Abstracts of the 4th European Conference on Permafrost*. Évora, Portugal, Jun 2014.

F. Sutton, R. Da Forno, R. Lim, M. Zimmerling, and L. Thiele.

Demonstration Abstract: Automatic Speech Recognition for Resource-Constrained Embedded Systems. In *Proceedings of the 13th International Conference on Information Processing in Sensor Networks (IPSN)*. Berlin, Germany, Apr 2014.

F. Sutton, R. Da Forno, M. Zimmerling, R. Lim, T. Gsell, F. Ferrari, J. Beutel, and L. Thiele. **Poster Abstract: Predictable Wireless Embedded Platforms.** In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN)*. Seattle, USA, Apr 2015.

F. Sutton, M. Zimmerling, R. Da Forno, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel, and L. Thiele. **Bolt: A stateful processor interconnect.** In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*. Seoul, South Korea, Nov 2015.

F. Sutton, M. Zimmerling, R. Da Forno, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel, and L. Thiele. **Demo: Building reliable wireless embedded platforms using the bolt processor interconnect.** In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*. Seoul, South Korea, Nov 2015.

F. Sutton, R. Da Forno, D. Gschwend, R. Lim, T. Gsell, J. Beutel, and L. Thiele. **Poster Abstract: A Heterogeneous System Architecture for Event-triggered Wireless Sensing.** In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks (IPSN)*. Vienna, Austria, Apr 2016.

