Diss. ETH No. XX

Leveraging Synchronous Transmissions for the Design of Real-time Wireless Cyber-Physical Systems

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich
(Dr. sc. ETH Zurich)

presented by
Romain Jacob

born on 03.12.1990
citizen of France

submitted to
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Thiemo Voigt, co-examiner
Prof. Dr. Martina Maggio, co-examiner

November 2019

B

Contents

1

Introduction

Cyber-Physical Systems (CPS) are understood as systems where "*physical and software components are deeply intertwined, each operating on different spatial and temporal scales, exhibiting multiple and distinct behavioral modalities, and interacting with each other in a myriad of ways that change with context*" [138]. The domains of application of CPS are very diverse: *e.g.*, robotics, distributed monitoring, process control, power-grid management [171, 113, 148].

It is important to realize that the design of CPS encompasses three main aspects, mapping to as many research fields, with their own purpose and goals: The *embedded hardware design* aims to extend the amount of computational resources available (*e.g.*, processing power, memory, sensors and actuators) while limiting the cost, form factor, and energy consumption of a device. The *communication*, either wired or wireless, aims to transmit messages between distributed devices efficiently; that is, quickly and using little energy. Finally, the *distributed system design* realizes the implementation of the CPS functions, such as *e.g.*, remote monitoring and control of distributed processes.

The goal of the overall design is to reliably provide the specified CPS functions. Achieving this goal relies on hardware and communication; however reaching "perfect" communication, such as 100% packet reception rate, is not a goal in itself; it is merely a mean to an end. What truly matters is to fulfill the system functionality. Typically, CPS design aims to provide end-to-end performance guarantees, such as meeting hard deadlines between the execution of distributed tasks; for example, between the start of a sensing task to the end of the corresponding actuation tasks – Figure 1.1. Meeting such deadlines is called *providing real-time guarantees*.

The potential benefits of wireless communication for CPS applications are well-known and include *e.g.*, simpler deployment and maintenance, cheaper operational costs, lighter weight [119]. Furthermore, wireless is the only

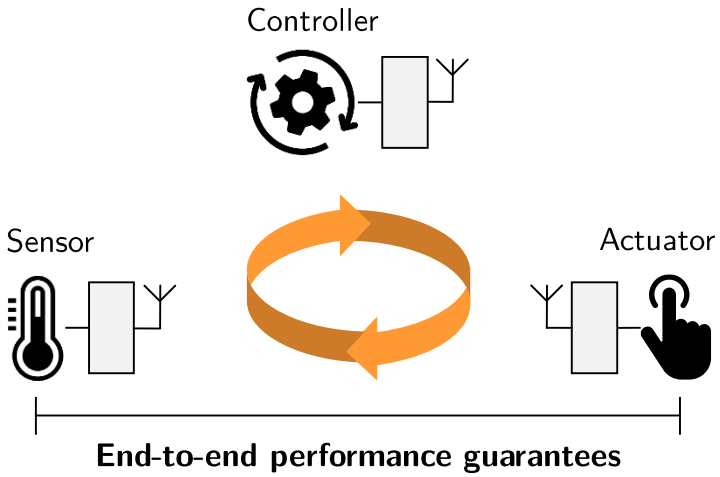


Figure 1.1 The primer objective of CPS design is to provide end-to-end performance guarantees for distributed applications. In this dissertation, we consider synchronous transmissions, a recent development in low-power wireless communication, and attempt to leverage the technique to provide real-time guarantees in wireless CPS.

viable option in application domains including highly mobile nodes, such as an automated warehouse with transport robots [84] or teams of drones [132]. However, CPS applications have challenging performance requirements [18], which are hard to fulfill with a wireless design.

1.1 Requirements of Wireless CPS

CPS applications are subject to different types of requirements, such as the specified end-to-end latency, bandwidth, or number of devices; the precise performance level for these requirements would depends on the application context. Generally, CPS requirements belong to one of the following classes:

Reliability A large ratio of messages are successfully transmitted wirelessly.

Adaptability The system adapts to runtime changes in resource demands.

Mobility The system supports mobile devices.

Timeliness Applications meet their deadlines, which are often specified end-to-end. Depending on the class of systems, deadlines can be either soft or hard [45].

Efficiency The system supports short end-to-end latency, scales in terms of system size, and optimizes its energy consumption and bandwidth utilization.

These requirements are mutually conflicting. For example, reducing the energy consumption is typically achieved by keeping the radio turned off whenever

possible. However, this directly conflicts with *Adaptability*, as the system cannot adapt reliably without exchanging some extra messages. In general, there is a price to pay in terms of *Efficiency* for meeting any of the other requirements. Hence, designing CPS consists in exploring the design space for relevant trade-offs; that is, the design optimizes the overall system *Efficiency* while meeting other application requirements.

1.2 Traditional Wireless Networking

Low-power wireless communication is a mature field of research, heavily studied for more than two decades. A large part of the research focused on wireless sensor networks, where low power consumption is a key requirement to enable long-term operation of the deployed networks, with specifications up to multiple years of operation on small batteries. Many successful applications and deployments include monitoring of soils [80], permafrost [193], buildings [49], or wildlife [46, 205].

In these scenarios, the distributed application often remains simple (*e.g.*, collect sensor readings). The main challenge is to reliably aggregate or disseminate messages across a multi-hop network. *Single-hop* communication refers to the case where a source node is in communication range from its destination. This is a rather simple case, but the deployed networks often span large areas whereas low-power radios can typically communicate in the range of tens of meters. Thus, *multi-hop* communication is required, whereby a source node must rely on other nodes in the network to forward its messages, hop after hop, until the destination is reached.This is the same principle as in the children's game known as Chinese whispers [197]; if you ever played, you know that the original message hardly ever reaches the end of the chain successfully.

Multi-hop communication is a collaborative task for which the nodes must be coordinated. Indeed, if a node transmits a message while another wireless communication is ongoing, the transmissions will interfere and they may both fail. Furthermore, the radio frequency bands used for wireless communication cannot be isolated. Other networks are potentially exchanging messages on the same frequencies, which generates external interference and triggers packet losses. As a result, traditional multi-hop communication requires complex mechanisms for coordinating the nodes, scheduling the different transmissions to forward all messages throughout the network, and retransmitting messages that have been lost (*e.g.*, due to external interference). The complexity is further increased in mobile scenarios, where the set of neighboring nodes (which may relay a node's messages) changes frequently. The traditional wireless networking approach performs multi-hop communication by carefully planning a sequence of unicasts (*i.e.*, one-hop transmissions), usually performed along one or a few of the shortest paths possible between a message source and

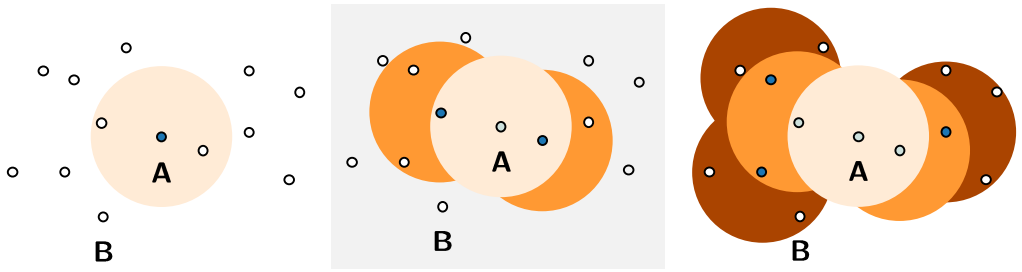
its destination [191, 103, 133]. Intuitively, this is efficient because only the necessary nodes are involved in relaying a message.

In practice however, multi-hop wireless network are sensitive to topology changes, external interference, and traffic congestion. These limit the reliability of communication, which has been a major obstacle to the utilization of wireless technology in CPS: for a long time, it has been considered impossible to provide the required level of reliability using wireless [170]. Synchronous transmissions have fundamentally changed that.

1.3 Synchronous Transmissions

Synchronous transmissions (ST), also referred to as concurrent transmissions, is a technique consisting in letting multiple nodes transmit a message at the "same time" (hence the name of *synchronous* transmissions). A destination node can successfully receive (one of these) synchronous transmissions thanks to two effects taking place at the physical layer: constructive interference and the capture effect [201, 68]. In a nutshell, ST is likely to be successful if the incoming messages arrive at the receiving node's antenna within a small time offset (in the range of a few symbol periods – a few tens of μs – depending on the physical layer and the effect considered). ST has been shown to work both analytically [198], empirically [70], and on different physical layers, such as IEEE 802.15.4[202], Bluetooth [21], and LoRa [194].

The use of ST in low-power communication, pioneered by Glossy [70] in 2011, has triggered a paradigm shift in the low-power wireless community: ST can be leveraged to implement efficient broadcast in a multi-hop network using network-wide flooding (Figure 1.2). The flooding procedure implemented by Glossy is illustrated in Figure 1.3. A first node initiates the flooding process. The 1-hop neighbors of the initiator receive the message and synchronously broadcast this same message in the next time step, which is then received by the initiator's 2-hop neighbors with high probability, thanks to ST. The process repeats following the same logic: a node that receives a packet broadcasts it again in the next time slot. Each node in the network transmits each packet up to N times, after which the flood terminates. It has been shown in a wide range of scenario that, with $N = 3$, Glossy achieves a reliability above 99.99% [70]; that is, 99.9% of the floods are successfully received by nodes in the network. With $N = 5$, the average reliability reaches 99,999% [70]. Glossy achieves such high reliability by leveraging spatio-temporal redundancy. Packets are transmitted along all possible paths; in other words, they are implicitly routed everywhere, and therefore avoid interference sources localized in space. In addition, having each node transmitting N times creates temporal redundancy, thereby avoiding interference sources localized in time. Moreover, the predictability of the operation timing in ST-based flooding can be leveraged



Node **A** initiates the flood. The 1-hop neighbors of **A** synchronously repeat. Node **B** receives the message.

Figure 1.2 Flooding process for a message sent from node **A** to node **B**

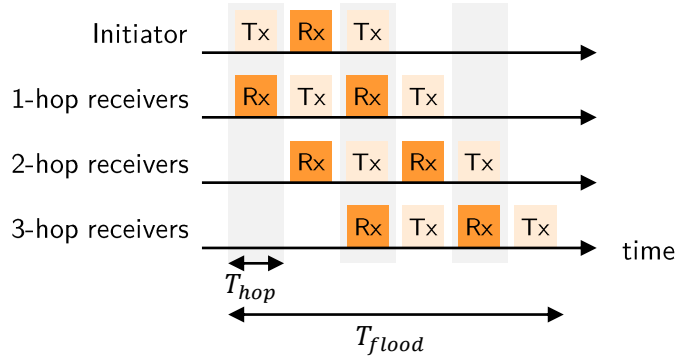


Figure 1.3 Glossy operation in a 3-hop network with 2 transmissions per node (N)

to perform distributed time synchronization. Glossy demonstrated that sub-μs synchronization accuracy can be achieved in a multi-hop network composed of tens to hundreds of nodes [70]. Since Glossy, other flooding strategies have been proposed [114, 26, 120], but the overall principle remains the same.

The key benefit of ST is that, thanks to the provided multi-hop broadcast primitive, the overall communication design can be dramatically simplified. Essentially, one can abstract the underlying multi-hop topology as a *virtual single-hop network*, which can scheduled like a shared bus: any node can send a message to any other node(s) in the network in bounded time. The only requirement is that no other node is using the "bus" at the same time. This design, first proposed with the Low-power Wireless Bus protocol [72], has been adapted into many different flavors (*e.g.*, [95, 20, 155, 26]) with always a similar concept (Figure 1.4): communication is organized in rounds, between which nodes keep their radio turned off to save energy. Each round is composed of time slots, which are assigned to certain nodes for communication. In each of these slots, nodes execute a flooding primitive (*e.g.*, Glossy) thereby preforming a one-to-all communication. Consequently, the complexity of performing reliable multi-hop communication (described in Section 1.2) is significantly relaxed. Thanks to ST, multi-hop communication is reduced to the scheduling of a single shared resource, a well-understood and relatively easy problem [45].

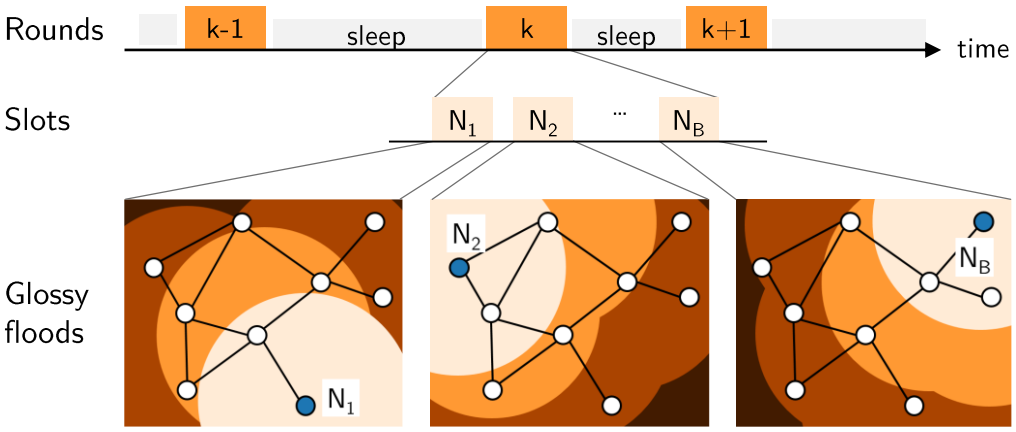


Figure 1.4 Thanks to synchronous-transmissions based flooding, a multi-hop network can be abstracted and scheduled like a shared bus. Communication is organized in rounds, composed of time slots; in each time slot, a node initiates a flood which allows to send a message to any other node(s) in bounded time. This mimics the operation of classical field bus, but with a wireless design.

A priori, flooding seems to be a wasteful approach: every message sent by any node will be received and forwarded by every other node in the network. However, the simplicity and reliability of the approach actually pays off. (i) Since the flooding logic is simple, it requires little communication overhead for the coordination of the network; nodes mostly send application data. (ii) The spatio-temporal redundancy embedded in the flooding process makes it very reliable; once a flood is completed, there is hardly ever a need to further retransmit a message in a subsequent flood. (iii) Finally, since multiple nodes can transmit simultaneously, the flooding process completes quickly; very close to the theoretically optimal speed [70].

Thus, with flooding approaches based on ST, the energy cost of sending one byte of data is relatively high (since this byte will be retransmitted by all the nodes), but the *overall cost* for communication remains relatively small, thanks to the limited protocol overhead and the absence of need for further retransmissions. The energy efficiency and reliability of ST-based flooding has been demonstrated in many research contributions (*e.g.*, [70, 111, 89]) and showcased in the EWSN Dependability Competitions [160], where all wining solutions in the past four years (2016 to 2019) were based on ST [67, 167, 114, 69, 120].

The downside of ST is that it is difficult to use in more complex system designs, such as those envisioned for wireless CPS [18]. The difficulty stems from the tight timing requirements for successful ST: to be received reliably, transmissions must be initiated by the different nodes within few μs. Practically, this implies that the runtime execution of a node is governed by the communication protocol, which makes the implementation of advanced distributed tasks complex and error prone. As a consequence, ST has thus far been mainly used for academic endeavors and mostly in wireless

sensor network scenarios where the application tasks are typically simple and non-critical. Collecting a new sensor reading is a task that can usually tolerate being delayed by a few milliseconds while communication is ongoing. This is generally not acceptable for wireless CPS.

1.4 The Dual-Processor Platform

In CPS, each device must perform application and communication tasks in order to fulfill the overall system functions; this poses the challenge of interference between tasks which contend for processor execution time. This interference problem can be mitigated by a new breed of embedded platforms featuring multiple processing cores, such as the NXP LPC541XX [140] or the VF3xxR [141]. On the one hand, this helps because applications and communication tasks can be processed in parallel, but on the other hand, it creates contention for the access to the resources shared between the cores. Efficient scheduling of multi-core platforms is a complex problem and a research field of its own.

Instead of resolving contention by scheduling, another approach proposed in the literature attempts to *prevent interference by design*. This principle, soberly called the Dual-Processor Platform (*DPP* – [35]), consists in linking two processors with a processor interconnect called *Bolt* (Figure 1.5). *Bolt* [174] provides predictable asynchronous message passing between two arbitrary processors while decoupling these processors with respect to time, power, and clock domains. The lower part of Figure 1.5 shows a conceptual view of the *DPP*, including two message queues with first-in-first-out (FIFO) semantics, one for each direction, which are the only communication channels between the interconnected processors. The guiding principle of *Bolt* design is to limit the interference between the interconnected processors as much as possible, then to provide formally verified bounds on the unavoidable interference remaining. Concretely, this means that the *Bolt* API functions, used by the processors to exchange messages, have hard latency bounds. The upper part of Figure 1.5 shows an early prototype of a *DPP*. Section 1.B illustrates other *DPP* designs, integrating the concept into smaller form factors and using different processors and targeting different application scenarios.

The *DPP* concept provides a predictable architecture for wireless CPS nodes. By entirely dedicating one processor to the application tasks and another one to wireless communication, we can decouple the timing of communication from the timing of the applications, and therefore facilitate the integration of ST in a CPS design. Furthermore, this helps optimizing performance: each processor can be customized for the specific operations it has to perform. The division of labor fosters specialization, thereby reducing the overall energy consumption and execution time; *i.e.*, maximizing *Efficiency*.

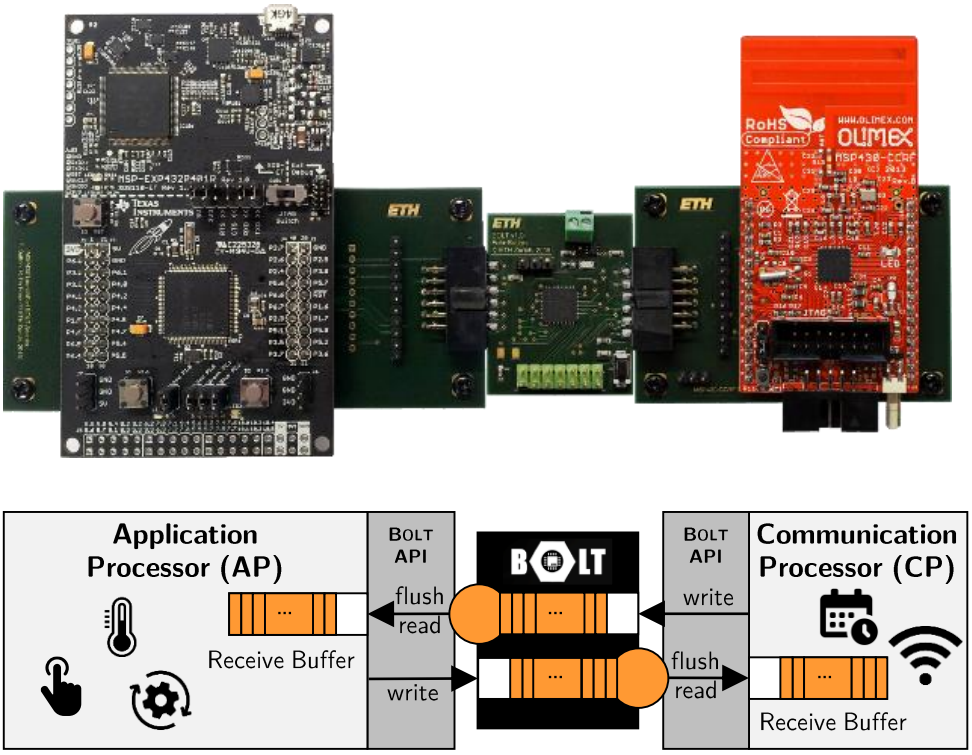


Figure 1.5 Top: Example of a custom-built heterogeneous *DPP*. *Bolt (in the middle) interconnects a powerful application processor (TI MSP432 [181]) on the left with a state-of-the-art communication processor (TI CC430 [180]) on the right.* Bottom: Conceptual view of the *Bolt* processor interconnect. *Using the Bolt's API functions (*`write`*,* `read`*, and* `flush`*), the processors dedicated to application (AP) and communication (CP) can asynchronously exchange messages with predictable latency, while otherwise executing independently from one another.*

1.5 Performance Evaluation in Networking

Over the past decade, low-power wireless communication has made significant progress, which are not limited to ST. The overall level of performance has increased, and it is now common to see reports of packet reception rates above 99% [70, 63, 64, 95].

The more extreme the performance level, the more critical it becomes to confidently assess the performance. Higher levels of confidence become necessary to argue about the differences in protocol design and quantify their performance trade-offs. Obviously, this is import for science as it allow to compare competing approaches. But it is also important for industry: these new and promising technologies will never be adopted unless we can back our performance claims confidently. In other words, others must be able to reproduce our experiments.

In the context of wireless networking, reproducible performance evaluation is made particularly challenging by the inherent variability of the experimental

conditions: the uncontrollable dynamics of real-world networks [127, 44] and the unsteady performance of hardware and software components [124, 37] can cause a large variability in the experimental conditions, which makes it hard to quantitatively compare different solutions [28].

This reproducibility challenge (sometimes even referred to a "crisis"[30]) touches all scientific fields, and recently received significant attention in computer science [52, 156, 29]. Yet, how to practically design and execute performance evaluation experiments for wireless protocols remains a largely open question which is being debated by the community [38]. The lack of a standard for evaluating performance prevents a clear comparison of the different approaches, and therefore hinders the adoption of the technology.

If everyone claims to be the best, it is difficult to trust anyone.

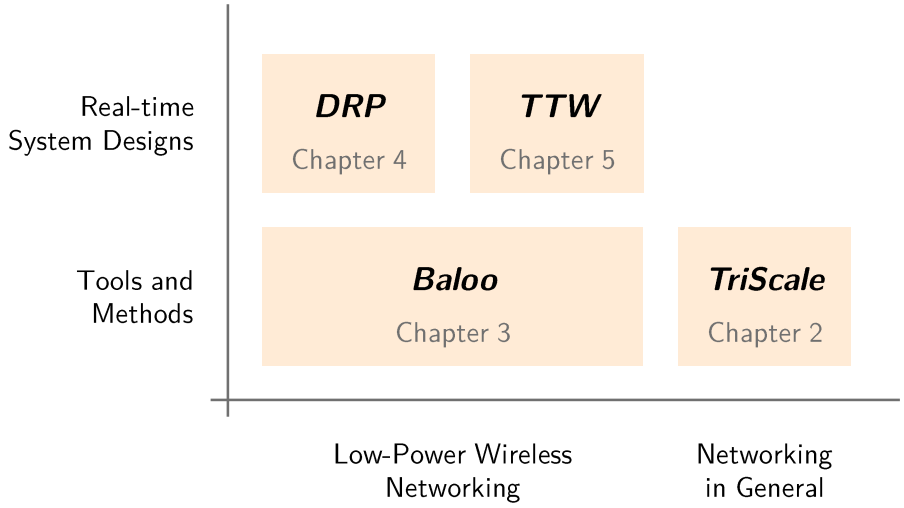


Figure 1.6 Overview of the chapters and contributions of this dissertation.

1.6 Thesis Outline and Contributions

In this dissertation, we attempt to leverage recent advances in the domain of low-power wireless communication, in particular synchronous transmissions, in order to design wireless CPS providing end-to-end real-time guarantees. Figure 1.6 provides an overview of the contributions of the dissertation, which are summarized below.

- We work towards more rigorous and reproducible experimental networking research. For the first time, we go beyond simple guidelines and propose a concrete methodology for designing networking experiments and analyzing their data. We leverage this methodology to propose the first formalized definition of reproducibility for networking experiments. We implemented our methodology in a framework called *TriScale*, a first-of-its-kind tool that assists researchers by streamlining the design process and automating the data analysis (Chapter 2).
- We propose and implement *Baloo*, a design framework for network stacks based on synchronous transmissions (ST). *Baloo* significantly lowers the entry barrier for harnessing the efficiency, reliability and mobility support of ST: users implement their protocol through a simple yet flexible API while *Baloo* handles all the complex low-level operations based on the users' inputs (Chapter 3).
- We demonstrate for the first time that end-to-end real-time guarantees can be obtained in wireless CPS by leveraging the efficiency and reliability of synchronous transmissions. We propose and implement wireless real-time protocols for two different design objectives.
  - The Distributed Real-time Protocol (*DRP*) uses contracts to maximize the flexibility of execution between application tasks (Chapter 4).
  - Time-Triggered Wireless (*TTW*) statically co-schedules all task executions and message transfers to minimize end-to-end latency (Chapter 5).

1.A "Reproducible" Dissertation

As discussed above, one of the contribution of this dissertation attempts to foster reproducible experimental practices in networking research (Chapter 2). In line with this idea, we take a step forward in that direction and attempt to make this entire dissertation "reproducible".

- Each chapter includes an "Artifacts and Links" appendix. As the name suggests, the reader will find there various links to supplementary materials related to the corresponding chapter.
- In particular, whenever possible, we make publicly available all the data presented in this dissertation, in both raw and processed form. We often provide links to digital notebooks, which allow data visualization in a web browser without requiring any file download.
- All the plots in this dissertation are "clickable"; that is, the plots are hyperlinks pointing to dynamic visualizations which let the reader explore the data (*e.g.*, zooming-in and -out in the plot, toggle visibility of individual traces, etc.). If you are reading a printed version of this document, you can find the corresponding link addresses in the "Artifacts and Links" appendices.
- The source files of this dissertation (this document) are themselves publicly available. TEX source files and figures are published on GitHub under the Creative Commons CC-BY-4.0 license.

1.B Dual-Processor Platforms

This appendix illustrates various *DPP* designs developed by the Computer Engineering Group at ETH Zurich. These research prototypes implement the *DPP* concepts described in Section 1.4, using different processors and targeting different application scenarios. Some of these designs are used in the real-world data collection deployments reported *e.g.*, in [193, 128].

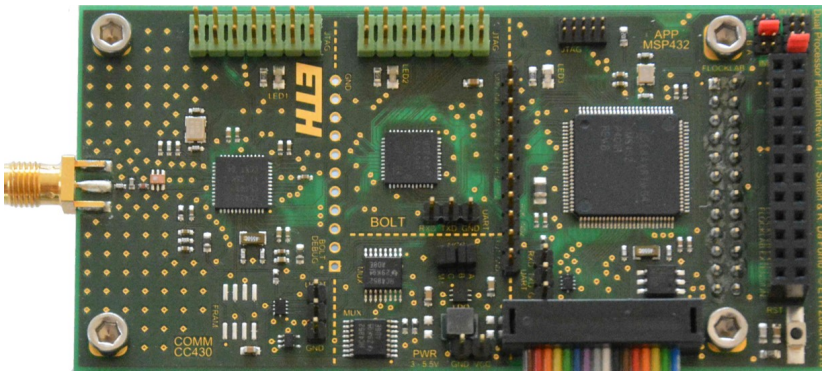


Figure 1.7 Integrated design of the same *DPP* as in Figure 1.5, featuring a TI MSP432 [181] as application processor (right) and a TI CC430 SoC [180] as communication processor (left). *Bolt* sits in the middle, implemented on a TI MSP430 core featuring 64k FRAM [182].

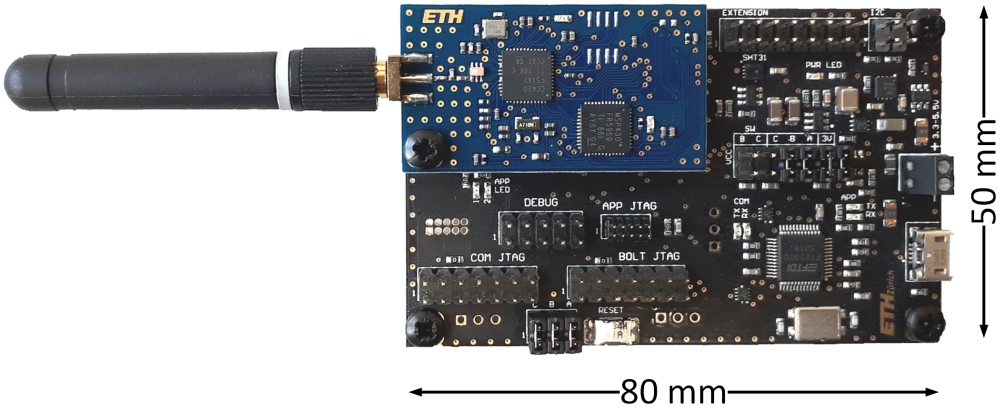


Figure 1.8 Redesign of the *DPP* concept into two separated boards, called the *DPP2* [35]. The lower board (black), called "application development board" hosts the application processor (in this case, a TI MSP432 [181]) as well as all the I/O, power management, debugging, and other support functions. The upper board (blue) is called the "communication board" and hosts the communication processor (in this case, a TI CC430 SoC [180]) and the *Bolt* interconnect. This platform has been used for a prototype wireless CPS presented in [123, 122] and discussed further in Chapter 5.

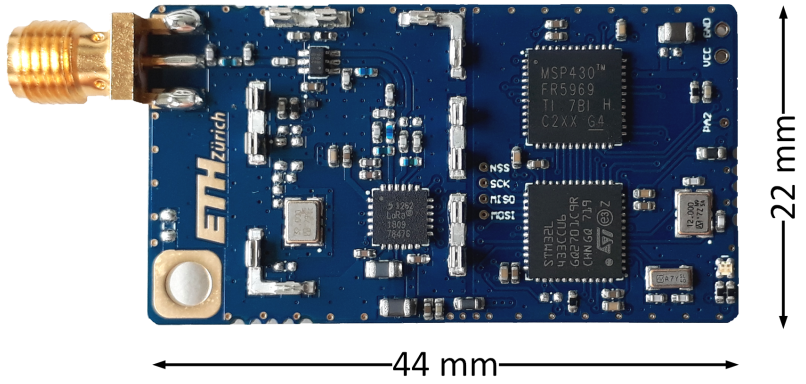


Figure 1.9 A different *DPP2* "communication board" featuring a STM32L4 ARM core and Semtech latest generation long-range low-power LoRa transceiver with up to +22 dB m transmit power at 868 MHz (Semtech SX1262) [163].

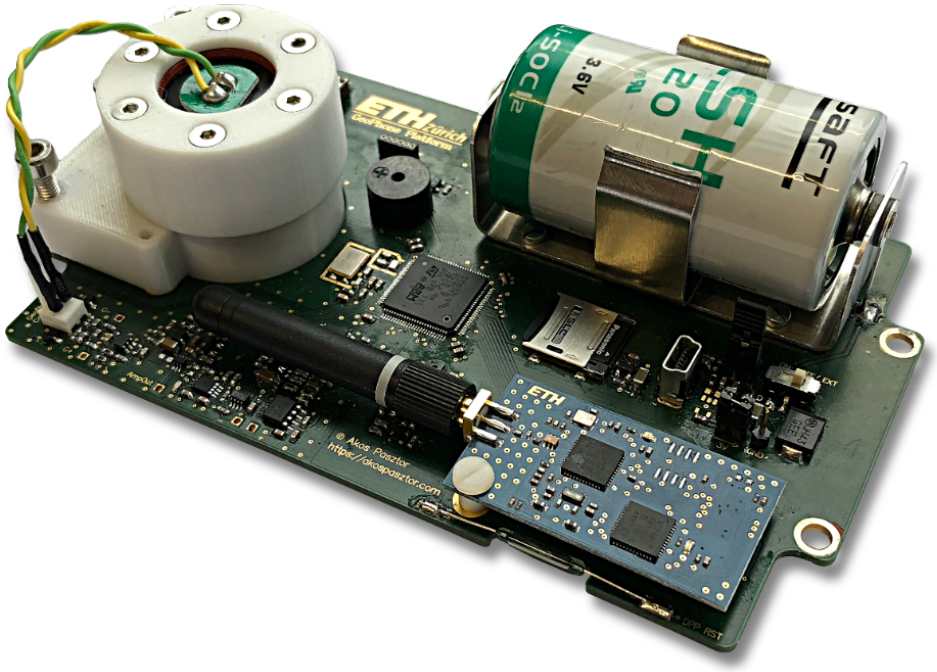


Figure 1.10 Battery-powered geophone sensor node based on the *DPP2* design. The lower "application development board" includes a geophone sensor, an analog low-power threshold-based wake-up circuit, and an ARM-Cortex M4 [22] as application processor. On top sits the same "communication board" as in Figure 1.8. This design is currently deployed on the Hörnligrat of the Matterhorn [128].

2

TriScale: A Framework Supporting Reproducible Networking Experiments

The design of a system is not truly completed until the system's performance has been evaluated. This evaluation must be performed in a way that can be reproduced by others to build confidence in the performance claims. In this dissertation, we study the design of wireless CPS and we therefore are directly concerned by the challenge of performing reproducible performance evaluation of networking protocols. Thus, in this chapter, we tackle the challenge of reproducibility in experimental networking research in general, beyond the sole context of low-power wireless networking (the main focus of this dissertation).

Achieving reproducibility in networking experiments requires a concrete methodology, which is currently missing. The design and data analysis of experiments raise questions such as: How many runs to perform? How to account for the variability of networking experiments? Despite the best intentions, researchers often answer these questions differently, which impairs the reproducibility of the entire evaluation. Moreover, it is currently unclear how to formalize reproducibility, let alone assess whether performance evaluation results are "reproducible" or not.

Claim. We contribute to make experimental networking research more rigorous and more reproducible. For the first time, we go beyond simple guidelines and propose a concrete methodology for designing networking experiments and analyzing their data. We leverage this methodology to propose the first formalized definition of reproducibility for networking experiments. Finally, we implement our methodology in a framework called *TriScale*, a first-of-its-kind tool that assists researchers by streamlining the design process and automating the data analysis.

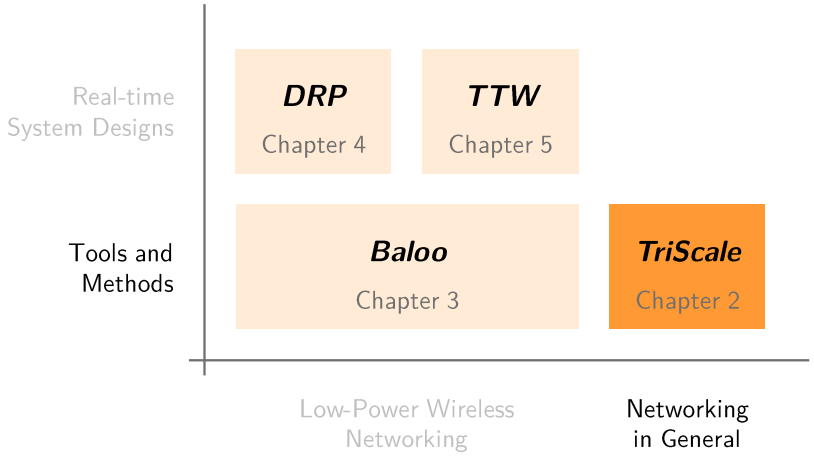


Figure 2.1 This chapter presents *TriScale*, a framework supporting reproducible evaluations in networking research in general, beyond the sole context of low-power wireless.

The material from this chapter builds upon the work from Antonios Koskinas [108]. It relates to the following publications.

Towards a Methodology for Experimental Evaluation in Low-Power Wireless Networking
Romain Jacob, Carlo Alberto Boano, Usman Raza, Marco Zimmerling, Lothar Thiele
CPS-IoTBench 2019. Montréal, Canada (April 2019)

TriScale: A Framework Supporting Reproducible Networking Evaluations
Romain Jacob, Marco Zimmerling, Carlo Alberto Boano,
Laurent Vanbever, Lothar Thiele
(*Under submission*) NSDI 2020. Santa Clara, CA, USA (February 2020)

2.1 Problem Setting

The ability of reproducing experimental results is broadly considered a prerequisite for establishing a scientific claim. In networking research, reproducibility[1] is a well-known issue due to the inherent variability of the experimental conditions. On the one hand, the uncontrollable dynamics of real-world networks [127, 44] and the unsteady performance of hardware and software components [124, 37] can cause a large variability in the experimental conditions, which makes it hard to quantitatively compare different solutions [28]. On the other hand, differences in the methodology used to design the experiment, analyze the data, and reason about the results impair the ability to

[1] A variety of terminologies is used to define the different aspects of reproducible research [146, 32]. In this dissertation, we refer to *reproducibility* as the ability of different scientists to follow the steps described in published work using the *same tools* and obtain the *same results* within the margins of experimental error. This corresponds to ACM's definition of *replicability* [15].

reproduce and critically assess the validity of claims made by other scientists. Without reproducibility, any performance comparison is debatable, at best.

Key Research Questions

Question 1 Can we improve the reproducibility of performance claims of networking experiments?

Question 2 Can we compare protocol performance in a statistically sound manner? In particular, how can we account for the inherent variability of the experimental conditions?

The problem. The community has relentlessly developed testbeds [139] and data collection frameworks [199] to minimize the variability in experimental conditions. In comparison, only a few works targeted the design of a methodology for networking evaluations [109]. In this regard, the literature is limited to best practices and generic guidelines [29, 156, 129].

Experimental networking research suffers from the lack of a *systematic methodology* specifying (i) how to concretely design an experiment, and (ii) how to analyze and report its results. Scientists are left with many open questions *before* carrying out an experiment (*e.g.*, How many runs? How long should they be? When should they run?) and *after* the data has been collected (*e.g.*, How to analyze and summarize the data in a concise yet accurate way?).

These open questions lead scientists to design similar experiments in different ways, which hinders comparability even when using the same tools [38]. Furthermore, the lack of a common methodology for data analysis hampers the ability to recreate results even when all raw data are available (as known as *computational reproducibility* [116]), a standard we argue that any scientific contribution should meet. Finally, it is unclear how to concretely define reproducibility and assess whether a networking experiment is indeed "reproducible". When reproducibility is considered a prerequisite to any scientific claim, this question cannot be left unanswered.

The challenge. We identify a set of requirements that an experimental methodology should meet in order to address this reproducibility problem.

Generality The methodology is applicable to a wide range of metrics, evaluation scenarios (both emulated and real-world settings), as well as network types (both wired and wireless).

Conciseness The methodology describes the experiment design and data analysis in a concise yet unambiguous way. This enables computational reproducibility while minimizing the use of highly-treasured space in research papers.

Robustness The methodology accounts for the intrinsic variability of networking experiments. The uncertainty is quantified and the analysis does not presume the nature of the raw data distribution (*e.g.*, no assumption of normality). The uncertainty quantifies the variation in performance that we expect to observe shall the experiments be repeated; in other words, the statistics used are not only descriptive but also predictive.

Rationality The methodology rationalizes the experiment design and helps answering questions such as: How many runs? How long should they be? When should they run? This allows to minimize the number of experiments while collecting enough data to meet the desired level of confidence.

Our solution. This chapter presents *TriScale*, a framework that meets all the above requirements for supporting researchers in the design and analysis of networking experiments. We make the following contributions:

- We propose a methodology for designing networking experiments and analyze their data. This methodology is grounded on non-parametric statistics (Section 2.3), which provides performance reports that are both robust and clear. Our methodology rationalizes the experimental design process and quantifies the reproducibility of an experiment.
- We implement this methodology in *TriScale*, a framework that assists its user in designing experiments and automating data analysis (Section 2.4).
- As a case study, we use *TriScale* to compare congestion-control schemes using the Pantheon data collection framework [199] and show how *TriScale* improves on the legibility and confidence in the results (Section 2.5). Additional examples using low-power wireless protocols running on the FlockLab testbed [115] illustrate the generality of the framework. Finally, we showcase the scalability of *TriScale*: the data analysis completes within seconds for millions of data points (Section 2.6).
- Finally, we make *TriScale* openly available (Section 2.A) for the networking community to use, extend, and build upon.

TriScale does not "solve" the entire reproducibility problem; in particular, it does not handle the data collection (see discussion in Section 2.7). *TriScale* does however fill a critical gap towards reproducible networking evaluations by providing a consistent methodology for the design of networking experiments and the analysis of their data.

2.2 Overview of *TriScale*

This section provides a high-level description of *TriScale*. First, we illustrate how *TriScale* clarifies the interpretation of the results of networking experiments

with a concrete example (Section 2.2.1), then we present the core principles of *TriScale* and introduce the structure of the framework (Section 2.2.2).

2.2.1 Shortcomings in the Data Analysis

Let us assume you are a networking researcher discovering the field of congestion control and trying to understand the strengths and weaknesses of the state-of-the-art. Luckily, the community has developed useful tools such as Pantheon [199], a data collection framework that facilitates comparisons between different schemes.

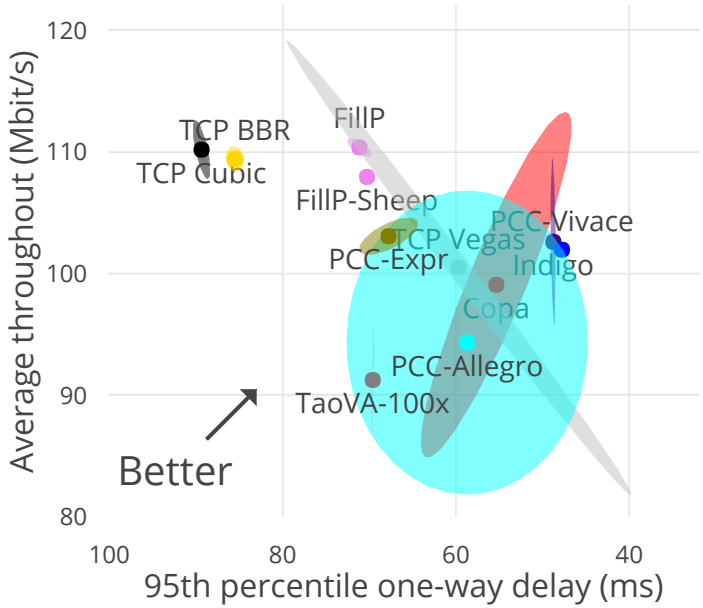
You are especially interested in comparing the average throughput and one-way delay of long-running full-throttle flows, *i.e.*, stable flows whose only throttling/limiting factor is the congestion control. You start with one flow and evaluate performance using the MahiMahi [137] emulator (integrated in Pantheon), following the same settings as in the original paper [199], *i.e.*, 10 runs of 30 seconds each. You collect data for the 17 schemes available at the time of your experiment.

Pantheon focuses on collecting data, not on their interpretation. Yet, the interpretation is not trivial. Consider for example the data shown in Figure 2.2a (reproduced from [199]). Multiple questions arise:

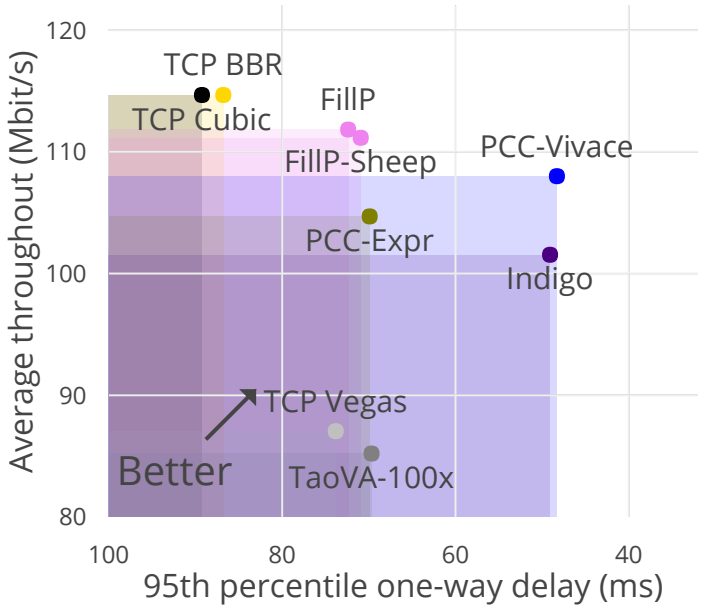
- Can the schemes be compared? It appears that *Vegas* performs better than *e.g.*, *TaoVA-100x*. However, the ellipses capture the variability of results across multiple runs: more precisely, they represent the $(1-\sigma)$ variation across runs. What can you then conclude about the actual performance of these schemes? Can you conclude anything when ellipses are overlapping? For example, can you say that *Vegas* performs better than *PCC-Expr*?
- What is the confidence in the comparison? Intuitively, the results of *e.g.*, *PCC-Allegro*, which has a large variability, are "less trustworthy" than *e.g.*, *FillP-Sheep*, for which you cannot even see the ellipse on the graph. But can you quantify the confidence in the result?
- Is a runtime of 30 seconds (the default setting) really long enough to capture the long-running performance of the various schemes?

These questions relate to the *Robustness* and *Rationality* requirements mentioned in Section 2.1. The data analysis shown in Figure 2.2a leaves these questions unanswered. Worse, the analysis may suggest wrong interpretations: the ellipses are a two-dimensional representation of the standard deviation across the runs, which suggests that one expects about 68% of the data points to fall in that region. However, this is correct **only if** the underlying distribution is normal, which is hardly ever true (Section 2.3).

Let us now compare the *same raw data*, but analyzed using *TriScale* (Figure 2.2b). The points in the plot represent *TriScale*'s Key Performance



(a) Data analysis and visualization reproduced from [199]. The dots represent the mean performance of the runs; the ellipses represent the $(1-\sigma)$ variation across runs.



(b) Data analysis and visualization produced by *TriScale*. The dots represent the KPIs of each scheme. Shaded areas represent dominance regions: *scheme* A performs better than *scheme* B if the KPI of B lies in the dominance region of A.

Figure 2.2 The same data may be analyzed in different ways. Figure 2.2b illustrates the output of a data analysis performed by *TriScale*. Compared with Figure 2.2a, the interpretation of the results is more intuitive: The performance of each scheme is reduced to a single point (*TriScale*'s KPIs), which makes the comparison between the schemes unambiguous. *TriScale*'s KPIs are not arbitrary: they are robust statistics estimating, with a given confidence level, the expected performance if the experiment was repeated (see Section 2.4.2). As such, the KPIs inherently account for the variability in the results. *Experiment settings: 1 flow, 10 runs, 30 s runtime, emulated network (sample data from the congestion-control case study Section 2.5).*

Indicators (KPIs). A KPI is defined as the estimate of a given percentile of one performance metric distribution. For example: with 10 runs, we collect 10 samples of the performance metric "average throughput". Based on these 10 samples, we can *estimate* some properties of the underlying distribution of "average throughput" (*i.e.*, the *unknown* distribution one would obtain with infinitely many samples). *TriScale*'s KPIs estimate percentiles of that underlying distribution with a certain confidence. In Figure 2.2b, the "average throughput" KPI is defined as the estimate of the 25th percentile with a 75% confidence level; in other words, the KPI has a 75% probability to correctly estimate the 25th percentile of the distribution (more details in Section 2.4.2).

Since KPIs are individual points, they unambiguously compare different schemes. *TriScale* shows, for example, that *Vegas* is not generally better than *TaoVA-100x*: each performs better in either delay or throughput (Figure 2.2b). Furthermore, *PCC-Expr* performs strictly better than *Vegas*, whereas Figure 2.2a suggests the opposite. The KPIs in Figure 2.2b can be interpreted as follows: *with a 75% probability, 75% of the runs will yield a performance at least as good as the KPI value* (*i.e.*, higher throughput and lower delay).

Observe that *Copa* and *PCC-Allegro* are no longer present in Figure 2.2b. Indeed, *TriScale* first verifies whether the schemes have "converged"; that is, it checks whether the performance metrics have reached stable values (Section 2.4.5). These two schemes eventually converge, but it often takes more than 30 seconds. In Figure 2.2b, *the KPIs are representative of the "long-running" performance* (*i.e.*, the performance expected if the scheme would run "forever") with a 95% probability.

Conclusion. Tools like Pantheon [199] support data collection, but leave to the researcher to design the experiments and analyze the data, leading to ambiguous interpretations and non-reproducible results. *TriScale* aims to fill this gap.

2.2.2 Methodological Core Principles

We now introduce how *TriScale* supports the design and analysis of networking experiments. The structure and inputs/outputs of the *TriScale* framework are illustrated in Figure 2.3.

Experiment design. *TriScale* achieves *Rationality* (Section 2.1) by formalizing the evaluation definition and by streamlining the design of experiments. The design phase starts with the definition of the evaluation objectives: for each performance dimension, the user defines the metric, the convergence requirements, a KPI, and a variability score. From these inputs, *TriScale* returns the minimal number of runs ($\#runs$) and series ($\#series$) necessary to compute the chosen KPIs and variability scores; that is, *how many runs to perform*. Using

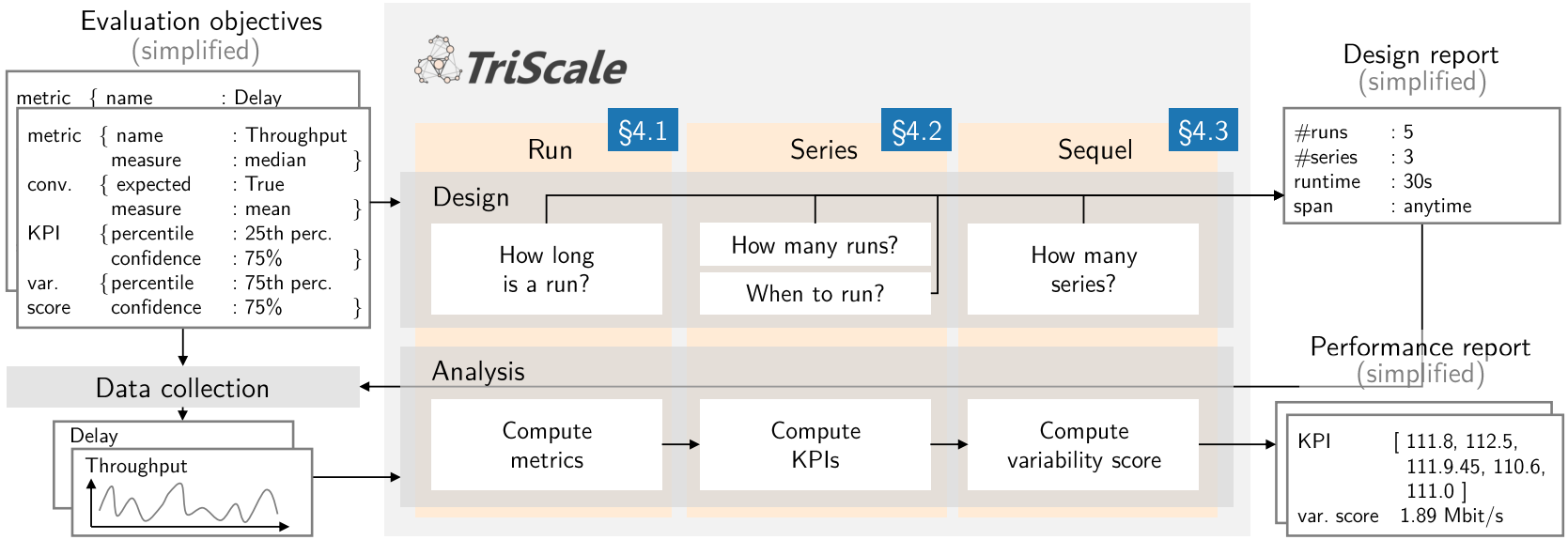


Figure 2.3 Overview of *TriScale*. *TriScale is a framework supporting the design and data analysis of networking experiments. TriScale assists the user in the design phase with a systematic methodology to answer important experiment design questions such as "How many runs?" and "How long should the runs be?". After the data has been collected, TriScale supports the user by automating the data analysis. The framework implements robust statistics that handle the intrinsic variability of experimental networking data and return expressive performance reports as well as a variability score.*

the data from a few test runs, *TriScale* can assess whether the length of a run (the *runtime*) is suitable; *i.e.*, *how long a run should be*. Finally, *TriScale* uses network profiling information to avoid time-dependent bias in the experiments; *i.e.*, it tells *when the experiment should be carried out* (the *span*).

In the congestion-control example presented previously, the evaluation objectives are the following. The metrics' measures for throughput and delay are the median and 95th percentile, respectively. The KPIs are chosen as the 25th and 75th percentiles with 75% confidence, for which *TriScale* returns a minimum of 5 runs. Convergence is expected and initial tests reveal that *PCC-Allegro* and *Copa* almost never converge within 30 s (see Section 2.5). As experiments are carried out in emulation, there is no time dependency and therefore it does not matter when the experiment is performed (*i.e.*, *span: anytime*).

Data analysis. *TriScale* achieves *Robustness* (Section 2.1) by applying carefully-chosen statistical methods, verifying that their hypotheses hold for the collected data, and automating the computations. In particular, once the experiment has been designed and the data collected, the raw data is passed to *TriScale* for analysis. The analysis is divided into three timescales: *runs*, *series*, and *sequels* (hence the name of *TriScale*):
A **run** is one execution of the evaluation scenario (*e.g.*, a 30 s execution of one congestion-control scheme). The raw data from one run are analyzed to compute the performance metrics defined in the evaluation objectives. This timescale leads to one number per run and per metric (Section 2.4.1).
A **series** is a set of runs performed closely in time (*e.g.*, in the same day). Multiple runs allow to account for the inherent variability in the experiments. This timescale leads to one number per series and per metric, *i.e.*, *TriScale*'s KPIs (Section 2.4.2).
A **sequel** is a repetition of a series, performed at a later point in time (*e.g.*, a week, a month, or a year later). *TriScale* uses sequels (*i.e.*, a set of series) to compute a variability score which captures the long-term variability of the KPIs. This timescale leads to one number per metric (Section 2.4.3).

In the previous example, *TriScale* returns a pair of KPIs per scheme, which allow to unambiguous compare the different schemes with a quantified confidence (in this case, 75% – see Figure 2.2b).

2.3 Backgrounds on Statistics

This section briefly discusses some background on statistics which is relevant to performance evaluation.

Descriptive and predictive statistics. A statistic is a number computed from a data set using a mathematical formula. A statistic can always be

calculated and provides a factual description of the underlying data. This is referred to as a *descriptive statistic*. However, certain statistics have also some *inference* power; that is, based on the collected data, one may infer the shape of the underlying data distribution, which is unknown. These are referred to as *predictive statistics*.

Predictions are always uncertain and often rely on certain hypotheses. If the hypotheses hold for the collected data, then the statistic estimates some property of the underlying distribution (*e.g.*, mean, median, etc.) with a quantifiable level of confidence. One can then predict the expected values of data samples that have not been collected. One common hypothesis for predictive statistics is that the collected data is *independent and identically distributed* (*i.i.d.*); informally, this means that the underlying distribution of the data does not change and that successive data samples are not correlated. It is also common to presume the *nature of the data distribution* (*e.g.*, a normal or a Poisson distribution), which allows to make "better" predictions with less data. It is paramount to keep in mind the hypotheses underlying a statistical prediction.

Example 2.1. One can compute the mean μ and standard deviation σ of a data sample. If the underlying data distribution is normal (the hypothesis), then we can infer that about 68% of all data points (the distribution) will be contained in $\mu \pm \sigma$. However, **if the distribution is not normal, μ and σ are only descriptive statistics**; *i.e.*, they do not predict anything about the underlying data distribution.

Statistical methods. Many common statistical methods assume Gaussian distributions (*i.e.*, normally distributed data). However, literature reports that experimental data is rarely normal [124, 158] and hence recommends using *non-parametric statistics*; *i.e.*, statistics that do not make any assumption on the nature of probability distributions. Furthermore, it is important to consider *robust statistics*, *i.e.*, statistics that are not overly skewed by outliers (common in networking data). There are two main classes of statistical approaches: hypothesis testing and estimation. *Hypothesis testing* consists in formulating a so-called null hypothesis, that the test aims to reject. Based on the collected data, one computes the probability, called the p-value, that the null hypothesis is correct. If the p-value is sufficiently low, the null hypothesis is rejected and considered proven incorrect. *Estimation* consists in computing confidence intervals (CIs) for a given parameter (*e.g.*, the median of a distribution). A CI is always associated a confidence level (*e.g.*, a 95% CI) which is the probability that the interval includes the true value of the parameter. For example, $[a, b]$ is a 95% CI for the median if the true median value is between a and b with 95% probability (or better).

CIs are more legible than p-values: "*CIs provide a mechanism for making statistical inferences that give information in units with practical meaning*" [58].

Furthermore, the level of confidence of an estimation only depends on the sample size. In other words, estimations can be used to guide the experimental design. By setting the desired level of confidence, one defines the (minimal) number of samples required. This is a key property that *TriScale* leverages.

Reproducibility is a predictive statistic. Informally, reproducibility is the principle that the "same experiment" leads to the "same results". Thus, assessing reproducibility entails predicting that future data (*i.e.*, the results of a newly-performed experiment) will be the same as the known data (*i.e.*, the results of previously conducted experiments): this is a prediction. Thus, assessing reproducibility requires making certain hypotheses on the data. It is hence crucial to *(i)* choose statistics with hypotheses compatible with actual networking data, and to *(ii)* verify that the hypotheses do hold for the data that one collects. To this end, *TriScale* makes use of non-parametric statistics and verifies that their hypotheses hold for the collected samples.

2.4 Designing *TriScale*

In this section, we first describe the data analysis performed by *TriScale* and how the analysis procedure is linked to the design of experiments (Section 2.4.1 to Section 2.4.3). We then illustrate how the formalism introduced by *TriScale* allows to unambiguously describe an entire performance evaluation with only a handful of parameters (Section 2.4.4). Thereafter, we detail the robust and non-parametric statistical methods used by *TriScale* (Section 2.4.5), and discuss how the framework assists a user in deciding the required time span for a series of runs (Section 2.4.6). We finally show how *TriScale* helps assessing the reproducibility of experiments by computing a variability score (Section 2.4.7).

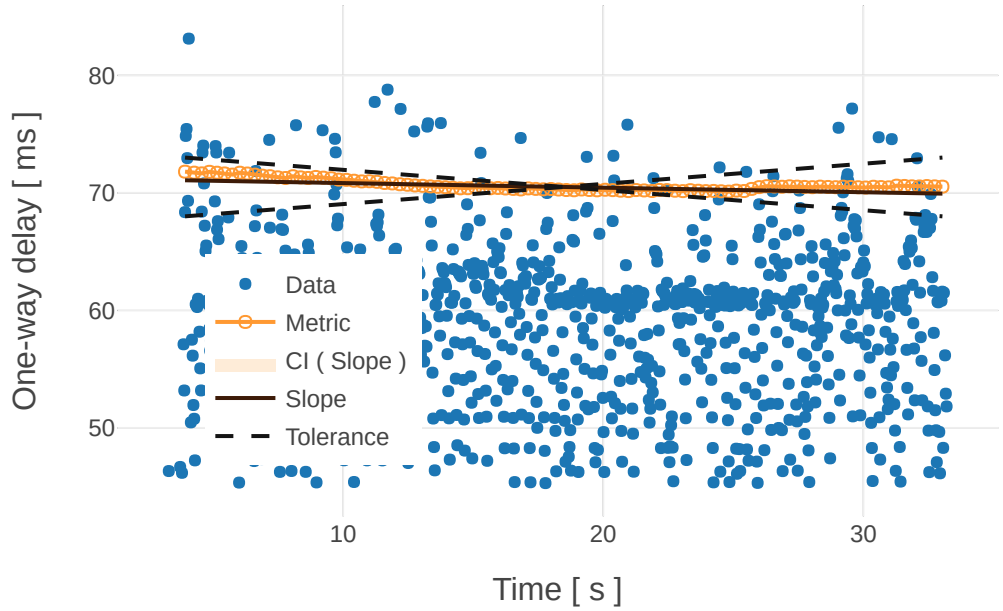
2.4.1 Runs and Metrics

A *TriScale*'s metric evaluates a performance dimension across one run. For example, a metric may be the average throughput achieved by a congestion-control scheme over 30 s runtime of a full-throttle flow. Computing a metric takes the following inputs.

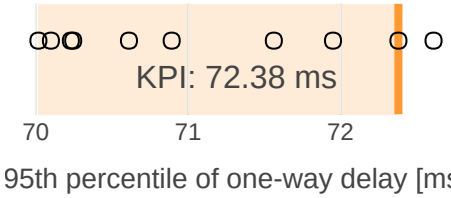
Inputs.

- The metric *measure*; *e.g.*, mean, maximum, etc.
- The *convergence* requirements
  $\{$ expected : true/false,
  confidence : C (*default: 95%*),
  tolerance : t (*default: 1%*) $\}$
- The raw data of the run.

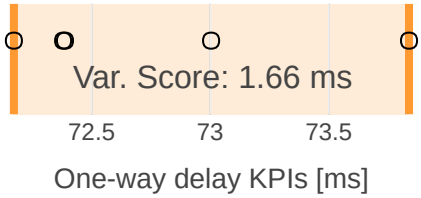
In its current implementation, *TriScale* supports only percentiles as measures.



(a) Raw data (one-way delay) and metric data (95th percentile).



(b) Runs' metric data and corresponding KPI value.



(c) Series' KPI data and corresponding variability score.

Figure 2.4 Example plots produced by *TriScale* during the data analysis. Figure 2.4a: computation of the metric (95th percentile on one-way delay) with convergence test (confidence 95%, tolerance 1%). Figure 2.4b: computation of the KPI (75th percentile with 75% confidence). Figure 2.4c: computation of the variability score (25-75th percentiles with 75% confidence). *Sample data from the case study (Section 2.5) for the FillP congestion-control scheme.*

This can be easily extended to offer a catalog of common measures for networking experiments (mean, max, fairness index, etc.).

Procedure. If the run is expected to convergence, *TriScale* starts by performing a convergence test. The purpose of the convergence test is to assess whether the metric measure has reached a "stable" value by the end of the run, and therefore if it is a good estimate of the "long-running" performance.

To test this, *TriScale* divides the raw data in 200 chunks. First, it considers the first 100 chunks (*i.e.*, the first half of the data) and computes a first value of the measure. Then, *TriScale* adds one chunk of data and recomputes a new measure value (*i.e.*, using the first 101 chunks). The process repeats until all chunks are used, leading to a set of 100 measure values. *TriScale* performs its convergence

test (detailed in Section 2.4.5) on the measure data. This procedure (i) tests what we are interested in (*i.e.*, the convergence of the *measure*, not of the protocol in general) and (ii) smooths the effects of ramp-up time by computing the first measure with already half of the run data. If the test is passed, *TriScale* returns the median of the measure data as the metric measure for the run.

Remark 1. The choice of 200 chunks is arbitrary. We choose this value as it is practical and leads to 100 measure data samples. Empirically, we found this is enough to reliably test for the metric convergence.

Remark 2. If there are less than 200 raw data points, *TriScale* reduces the number of chunks to the number of data points. *TriScale* (arbitrarily) sets a minimum of 20 data points for a convergence test.

Outputs.
- The result of the convergence test (if performed),
- The metric value for the run,
- Textual logs; plot of the input data and metric (Figure 2.4a).

Link to the experiment design. The computation of *TriScale* metrics is linked to the definition of the *runtime*; *i.e.*, how long a run should be. If the evaluation scenario is terminating (*e.g.*, transmit 1 MB through a link), the runtime must be long enough to complete the task. If the evaluation is "long-running" (*e.g.*, one-way delay in a full-throttle flow), the runtime must be long enough for the metric (the one-way delay) to converge (convergence test details in Section 2.4.5). *TriScale* can analyze preliminary experiments to estimate the required runtime: by performing increasing long runs and test for convergence (illustrated in Section 2.5).

2.4.2 Series and KPIs

TriScale's key performance indicators (KPIs) evaluate performance dimensions across a series of runs. Performing multiple runs allows to mitigate the inherent variability of the experimental conditions. KPIs capture this variability by estimating percentiles of the (unknown) metric distributions. Concretely, a *TriScale* KPI is a one-sided CI of one percentile; *e.g.*, a lower-bound for the 75th percentile of the delay metric estimated with a 75% confidence level. Computing a KPI takes the following inputs.

Inputs.
- The KPI definition
  $\{$ percentile : p
  confidence : C $\}$
- The metric data (computed from a series of runs).

Procedure. To compute the KPI (*i.e.*, to compute a CI for a given percentile), *TriScale* uses the Thompson's method (Section 2.4.5), which requires the

input data to be *i.i.d.*. Thus, *TriScale* starts by performing an independence test (Section 2.4.5) on the metric data before computing the KPI.

Outputs.
- The result of the independence test,
- The KPI value for the series of runs,
- Textual logs; plot of the metric data and corresponding KPI (Figure 2.4b).

Link to the experiment design. The computation of *TriScale* KPIs is linked to the definition of the number of runs in a series ($\#$ *runs*) and the series time span (*span*). The minimal number of runs in a series directly follows from the definition of the KPI; *i.e.*, the percentile to estimate p and the desired confidence level C. The series time span refers to the time interval used for scheduling the runs in a series (*i.e.*, when to run the experiment). This is important because networks often feature time-dependent conditions; for example, there may be systematically more cross-traffic during daytime than nighttime. Failing to account for such dependencies may bias the results and yield wrong conclusions. *TriScale* helps the experimenter handling this problem with a dedicated analysis module called "network profiling" (described in Section 2.4.6).

2.4.3 Sequels and Variability Score

TriScale's variability score evaluates the variations of KPI values across repetitions of the experiment (*i.e.*, a series of run), which we call sequels. Performing sequels allows to detect long-term variations in the KPI values which ultimately quantify the reproducibility of the experiment. Concretely, a variability score is a two-sided CI for a symmetric pair of percentiles, *e.g.*, the 75% CI for the 25-75th percentiles of the delay KPI. Computing a variability score takes the following inputs.

Inputs.
- The variability score definition
  $\{$ percentile : p (or 1-p)
  confidence : C $\}$
- The KPI values of each sequel.

Procedure. The procedure is the same as for the KPI: The Thompson's method requires the input data to be *i.i.d.* (Section 2.4.5), thus *TriScale* performs an independence test on the KPI data before computing the variability score.

Outputs.
- The result of the independence test,
- The variability score value for the entire sequels,
- Textual logs; plot of the KPI data and corresponding variability score (Figure 2.4c).

Link to the experiment design. The computation of the variability score is linked to the definition of the number of series (*#series*). The minimal number of series directly follows from the definition of the variability score; *i.e.*, the percentile to estimate p and the desired confidence level C.

2.4.4 Formalism Brings Conciseness

TriScale formalizes the definition of the evaluation objectives. For each performance dimension, the experimenter defines a metric and convergence requirements (Section 2.4.1), a KPI (Section 2.4.2), and a variability score (Section 2.4.3). *TriScale* links these objectives with the experiment design, resulting in four additional parameters: the number of runs per series (*# runs*), the number of series (*# series*), the length of a run (*runtime*), and the time span of a series (*span*).

Thanks to this formalism, *TriScale* meets the *Conciseness* requirement: Altogether, these 12 parameters are sufficient to *formally describe the entire performance evaluation* such that it can (eventually) be reproduced. In particular, since the data analysis in *TriScale* is automated and deterministic, documenting these parameters guarantees computational reproducibility (the ability to recreate the results when all raw data are available [116]).

Table 2.1 shows a few examples of concrete parameter settings for typical networking evaluation objectives. For example, evaluating the latency of a real-time protocol requires high confidence levels for extreme percentiles. This very quickly increases the number of runs that one must perform:
- at least 90 runs for estimating the 95th percentile with 99% confidence;
- at least 299 runs for estimating the 99th percentile with 95% confidence.

This illustrates that it is "easier" to increase the confidence level of an estimate than to estimate a more extreme percentile with the same confidence level. Note that both *#runs* and *#series* are only derived based on the definition of the KPI and variability score; these parameters are not influenced by the runtime or the time span of an experiment.

The second use case in Table 2.1 (bottom rows) illustrates two different perspectives on "averages", using delay as an example:
- If the metric is the median and the KPI the 90th percentile, one can conclude that 90% of the runs have a median delay equal or better than the KPI value.
- If the metric is the 90th percentile and the KPI the median, one can conclude that, in half of the runs, the 90th percentile of the delay in the run be equal or better than the KPI.

Both are "averages" but with different meanings and different requirements in terms of number of runs.

Table 2.1 Exemplary evaluation parameters of typical networking use cases. * *TriScale returns the minimal number of runs (#runs) and series (#series) based on the definition of KPI and variability score, respectively.*

Use case	**Evaluation Objectives**								**Experiment Design**			
	Metric	Convergence			KPI		Var. Score					
	Measure	Exp.	Conf.	Tol.	Perc.	Conf.	Perc.	Conf.	#runs*	#series*	runtime	span
Latency of real-time protocol	max	True	95%	1%	95	95%	median	75%	59	3	Depend on networks and protocols	
					95	99%	75	75%	90	5		
					99	95%	median	90%	299	5		
Average delay	median	False	–	–	90	95%	median	90%	29	5		
	90th perc.				median	95%	median	90%	5	5		

2.4.5 Statistics in *TriScale*

TriScale uses carefully chosen statistical methods. As discussed in Section 2.3, networking performance evaluations should focus on statistics that are both *robust* (*i.e.*, that can tolerate outliers) and *non-parametric* (*i.e.*, that do not make any assumption on the nature of the data distribution). This section describes the three statistical methods used in *TriScale*. We first present the convergence test used in the computation of metrics (Section 2.4.1); This test is based on the Theil-Sen linear regression [183, 164]. We then introduce the computation of confidence intervals using Thompson's method [185], which requires the data to be *i.i.d.*. Thus, to verify this assumption, *TriScale* integrates an independence test that we present last.

Convergence test. When an evaluation aims to estimate the "long-running" performance (*i.e.*, the expected performance if the run would run "forever"), one must verify whether the runs are long enough to produce reliable estimates.

To verify this, *TriScale* implements a convergence test based on the Theil-Sen linear regression [183, 164]. The approach computes the slope of the regression line as the median of all slopes between paired values. A C% confidence interval (CI) for the slope is defined as the interval containing the middle C% of slopes between single pairs. *TriScale* convergence test is passed if the C% CI for the regression is included in the tolerance value ($\pm t$%). To test the convergence of a run, *TriScale* uses the confidence C and tolerance t parameters specified in the evaluation objectives (Section 2.2); C and t are set to 95% and 1% by default.

Such a test is sensitive to the scale of the input data. To remove this dependency, *TriScale* first maps the data to $[-1, 1]$ using a linear transformation then performs the convergence test on the scaled data. Hence, the convergence test becomes dimensionless and the same tolerance value can be used for different evaluations without introducing bias. An example of the Theil-Sen slope (brown, solid), its CI (light orange, solid), and tolerance (black, dotted) is shown in Figure 2.4a.

Confidence Intervals. *TriScale* defines KPIs (Section 2.4.2) and variability scores (Section 2.4.3) based on CIs for distribution percentiles, which can be computed using Thompson's method [185], a robust and non-parametric approach.

Let us denote by P_p, the p-th percentile of a distribution and $\mathbb{P}(X)$ the probability of an event X. By definition, every data sample x is smaller than P_p with probability p (and larger with probability $1 - p$). For a sorted list of *i.i.d.* samples x_i (where $i = 1..N$), the probability that P_p lies between two consecutive samples follows the binomial distribution [185]:

$$\mathbb{P}(x_k \leq P_p \leq x_{k+1}) = \binom{N}{k} p^k (1-p)^{N-k}, \quad k = 0..N \tag{2.1}$$

where we assume $x_0 \to -\infty$ and $x_{N+1} \to +\infty$. From this result, it follows that the probability of P_p to be larger than any sample x_m (where $1 \leq m < N/2$) can be computed as:

$$\begin{aligned} \mathbb{P}(x_m \leq P_p) &= \mathbb{P}(x_{N-m+1} \geq P_{1-p}) \\ &= 1 - \sum_{k=0}^{m-1} \binom{N}{k} p^k (1-p)^{N-k} \end{aligned} \tag{2.2}$$

Equation (2.2) provides the upper- and lower-bound required for computing of CIs. See [158] for more details.

This approach provides robust estimates for distribution percentiles and *does not make any assumption on the nature of the underlying distribution*. It does, however, require that the data samples are *i.i.d.*. *TriScale* checks whether this hypothesis holds using an independence test, described below.

Independence test. Estimating the percentile of a distribution requires often (if not always) that the samples are *i.i.d.* (Section 2.3); this is also the case for Thompson's method [185]. *TriScale* implements an empirical independence test to verify whether the *i.i.d.* assumption holds. This independence test is applied to the metric data (resp. KPI data) before the computation of a KPI (resp. a variability score). This poses the particular challenge that the number of data samples may be very small (*e.g.*, 3 or 5 KPI values). *TriScale*'s independence test must therefore not be too strict.

The test is divided in two steps. First, *TriScale* tests whether the data appear *weakly stationary* (*i.e.*, no trend and constant autocorrelation structure [43]). *TriScale* verifies this empirically using its convergence test with a confidence of 50% and tolerance of 10%; these "loose" parameters are used to compensate for (very) small sample sizes. Second, *TriScale* computes the *sample autocorrelation coefficients*, denoted by $\widehat{\rho_k}$, which measure the linear dependency between values of a weakly stationary data series. A series of size N is *i.i.d.* with 95% probability if $|\widehat{\rho_k}| \leq 1.95/\sqrt{N}$ for $k \geq 1$ [43].

What if the tests fail? The experimenter is responsible for designing the evaluation in such a way that the collected data will (likely) pass the tests. *TriScale* facilitates this by guiding the choice of runtime to pass the convergence test and informing about any network time dependencies (Section 2.4.6) to pass the independence test. Yet, the data may still be correlated or unstable, leading to failing tests (see examples in Section 2.5). Even in such cases, the data still contain useful information. *TriScale* metrics, KPIs, or variability scores can be computed, however since the corresponding hypotheses do not hold, the statistics are *only descriptive* (Section 2.3); they do not predict the expected performance, and in particular they cannot (and should not be used to) assess the reproducibility of the experiment.

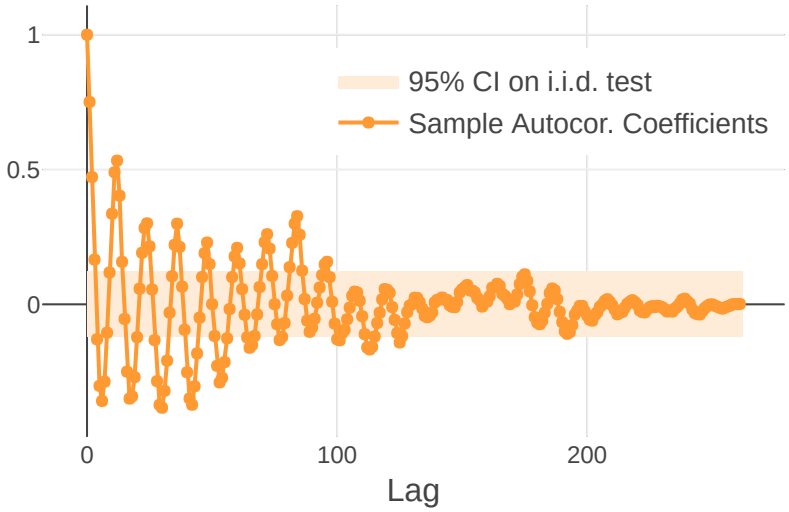


Figure 2.5 Autocorrelation plot for the wireless link quality on FlockLab, based on the raw data collected by the testbed maintainers [99] (data from August 2019). *The dataset contains one test every two hours. The first peak at lag 12 (*i.e.*, 24h) reveals the daily seasonal component. The data also show another at lag 84; which corresponds to one week. Indeed, there is less interference in the weekends than on weekdays: this creates a weekly seasonal component.*

2.4.6 Network Profiling

TriScale assists the user in deciding on the time span for a series of runs, *i.e.*, when should the run be performed in a series. This is important to avoid biasing the evaluation results with time dependencies in the experimental conditions. Indeed, it is common for real-world networks to exhibit periodic patterns. For example, there may be a lot more cross-traffic (*i.e.*, interference) at specific times. In the statistics literature, these patterns are called *seasonal components*. Neglecting these may result in biased experiments leading to wrong conclusions, as illustrated in the case study below.

Case study: low-power wireless. We run a simple evaluation of Glossy [70], a low-power wireless protocol based on synchronous transmissions (Section 1.3). Glossy includes as parameter the number of retransmissions of a packet, called N. We investigate the impact of two values of N on the reliability of Glossy, measured as the packet reception ratio (PRR). We define our KPI as the median with 95% confidence level. Refer to Section 2.A for the complete case study.

We collect data using the FlockLab testbed [115]. This testbed is located in an office building, where we expect more interference during daytime than nighttime. Thus, we schedule a series of 24 runs randomly within one day.
- $N = 1$ leads to a PRR of 88%,
- $N = 2$ leads to a PRR of 84%.

In other words, it appears that doing two retransmissions (instead of one) reduces reliability.

The experiment leads to this (incorrect) conclusion because we have neglected a second seasonal component of the FlockLab testbed: there is a weekly time dependence, revealed by Figure 2.5. To account for this dependency, one must schedule runs with a span of at least one week. When comparing again the performance of Glossy but with tests spanning over a week
- $N = 1$ leads to a PRR of 80%,
- $N = 2$ leads to a PRR of 88%,

which better matches our knowledge about the performance of Glossy.

Conclusion. This simple example illustrates that using a high confidence level is not enough to avoid drawing wrong conclusions due to the variability in the experimental conditions. On a real network, short-term variations are unpredictable and (often) unavoidable. This is why it is important to perform multiple runs in a series: it increases the chances to do the experiment in the whole range of favorable to unfavorable conditions.

However, we illustrated that systematic patterns are also present. In other words, there are times where there is consistently more or less interference. Knowing about these dependencies is important to ensure fairness in the comparison between protocols, and enable reproducibility of the evaluations. The series span must be long enough such that it does not matter when the series actually starts (*e.g.*, a weekend or a weekday)

TriScale integrates a network profiling function that analyzes link quality data (such as those available at [99]) and searches for seasonal components in the link quality data. This helps the experimenter detecting (sometimes unexpected) time dependencies, thus choosing a suitable time span for series of runs.

2.4.7 Assessing Reproducibility

Reproducibility refers to the ability of obtaining "the same" results when performing "the same" experiment. In statistics, such property can be investigated using *equivalence testing* [110], which checks whether the values of some parameter of interest (*e.g.*, the median) obtained for different samples are sufficiently close to be considered "the same". Unfortunately, there is no general way to define "sufficiently close"; one must define in advance a threshold for the equivalence test based on expertise. Then, how to assess reproducibility of networking experiments? How to design a "reproducibility test" that fairly adapts to different networking contexts and different metrics? After some failed attempts, we conclude that defining a generic threshold for equivalence testing in networking might not be possible. But it may not be necessary.

We argue that the most important is to confidently estimate the variability of the results, which *TriScale* computes with its variability score (Section 2.4.3). This score *quantifies reproducibility*: the larger the score, the less reproducible the results are. Shall a binary cut between "reproducible" and "not reproducible"

be desired, a threshold can be set based on the variability score; *e.g.*, "Results are said reproducible when the variability score is less than 20 Mbit/s.". Such a threshold can only be context-specific; thus, deciding on threshold values relates more to benchmarking and therefore goes beyond the scope of *TriScale* (see discussion in Section 2.7).

2.5 *TriScale* in Action

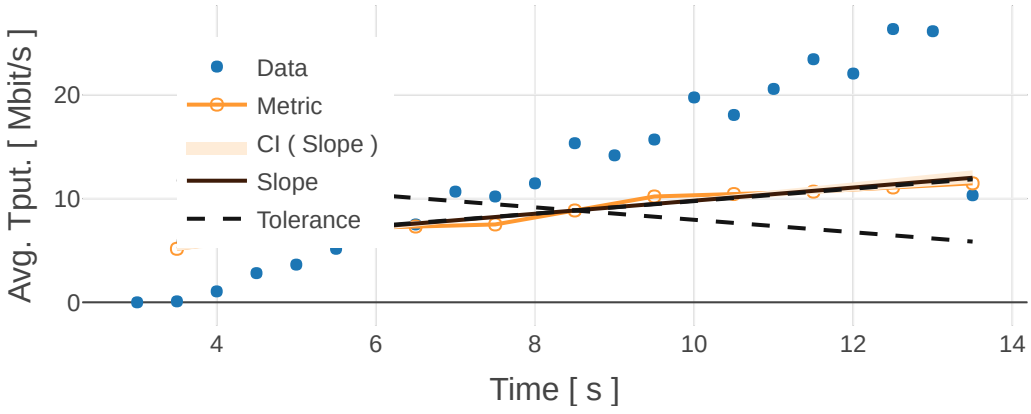
This section continues the case study introduced in Section 2.2.1. We compare the performance of 17 congestion-control schemes using Pantheon [199]. We evaluate the throughput and one-way delay of long-running full-throttle flows, *i.e.*, stable flows whose only throttling/limiting factor is the congestion control. For a fair comparison between the schemes, we use the MahiMahi emulator [137] (integrated in Pantheon). We focus on a single flow scenario and use the calibrated path from AWS California to Mexico.[2] The complete case study is available as complementary materials (Section 2.A); we present here only a fraction of it and focus on showcasing how *TriScale* avoids certain shortcomings in the experiment design and analysis. Finally, we illustrate how to quantify performance variability: a prerequisite for assessing reproducibility (Section 2.4.7).

Convergence time. The first step in the design of an evaluation is to decide how long the runs should be. Since all schemes are different, it is hard to know a priori the minimum runtime for which the various schemes actually converge.

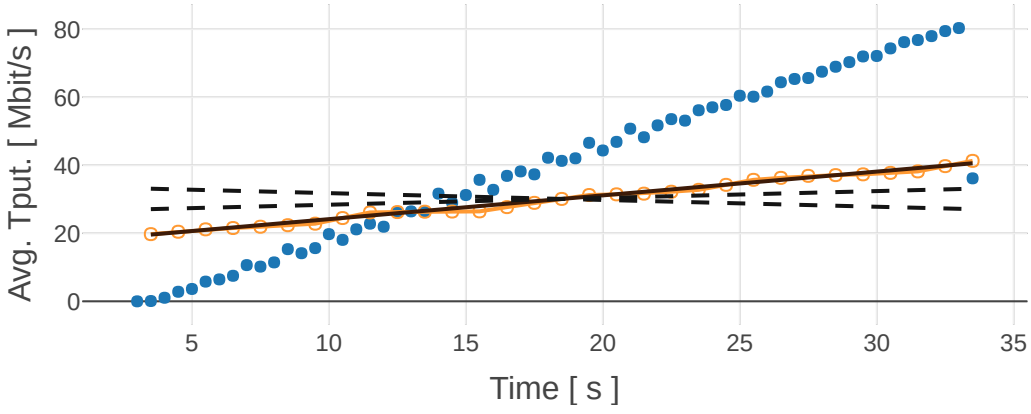
We test runtimes from 10 to 60 s and check whether the 17 congestion-control schemes pass *TriScale*'s convergence test (Section 2.4.5). With a runtime of 30 s, only twelve schemes often pass the test; others (*Verus*, *PCC-Allegro*, *Copa*, and *QUIC Cubic*) converge in less than half the runs. Furthermore, *LEDBAT* never pass the test, even with a runtime of 60 s. The reason for this is shown in Figure 2.6: the inner working of the protocol causes the throughput to ramp-up in the first 38 s of runtime and then converge to about 92 Mbit/s. If one uses 30 s runtime without checking for convergence, the computed average throughput is about 40 Mbit/s, which is a wrong estimation of *LEDBAT* "long-running" throughput. By performing the convergence test, *TriScale* hints the experimenter about the need to either increase the runtime, or prune the start-up time in the raw data.

Independence tests. The computation of *TriScale*'s KPIs and variability scores requires the samples to be *i.i.d.* (Section 2.4.5). However, the number of runs and series performed in an evaluation tends to be small (as experiments are both time- and resource-consuming), which limits the

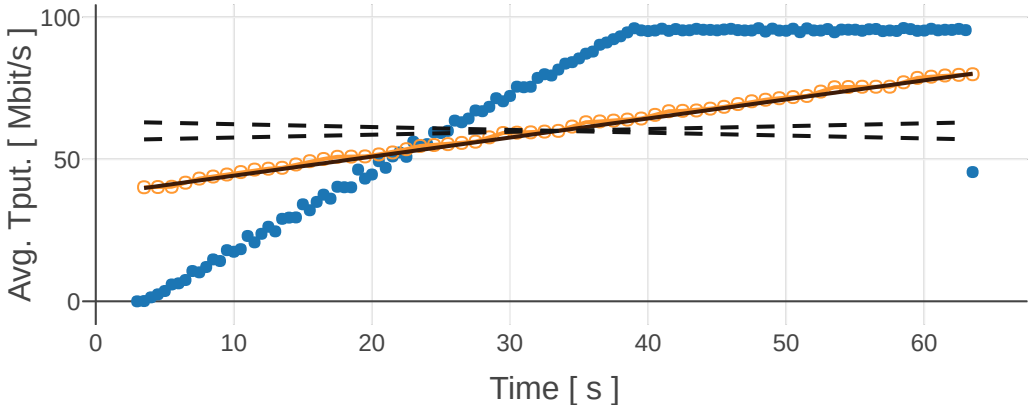
[2] pantheon.stanford.edu/result/6539/



(a) 10 s runtime



(b) 30 s runtime



(c) 60 s runtime

Figure 2.6 Egress throughput of *LEDBAT* in MahiMahi, calibrated to the real path from AWS California to Mexico [199]. *A runtime of 30 s is clearly not sufficient for LEDBAT's throughput to converge (Figure 2.6b). The scheme does converge eventually (Figure 2.6c), but even with 60 s runtime, TriScale's convergence test fails: the impact of the start-up phase is too important. Two possible solutions are to (i) increase the runtime or (ii) prune the start-up time from the raw data.*

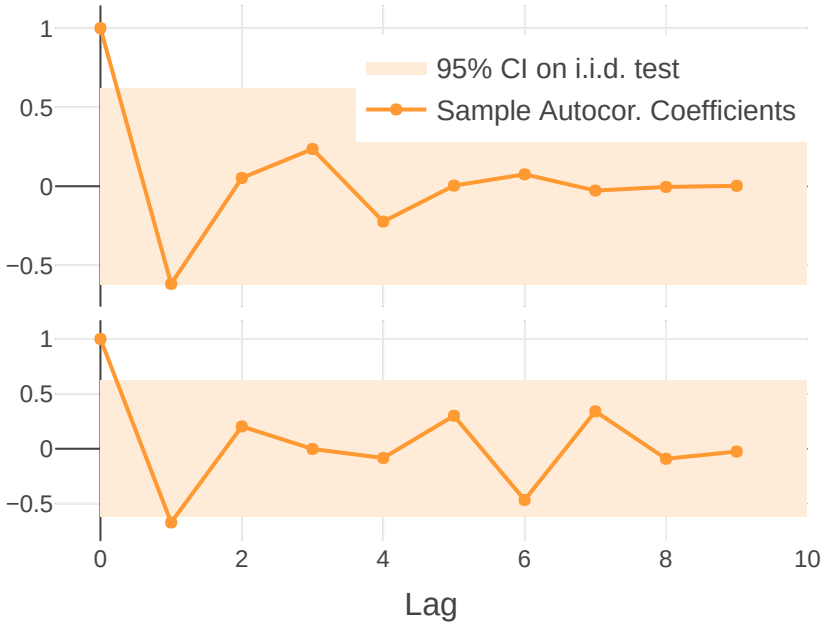


Figure 2.7 Autocorrelation coefficients for two exemplary series of *WebRTC*. *The upper series passes the autocorrelation test, whereas the lower series does not: this is an artifact induced by the small number of samples (here: 10 samples).*

significance of the independence test. Figure 2.7 illustrates this problem: it shows the autocorrelation plot for two series for *WebRTC*. The autocorrelation coefficients must be in the shaded gray area for the test to pass. In this case, the upper series passes the test, whereas the lower one does not. However, there is no clear difference in the correlation structure of the two series: the lower series does not seem significantly more correlated than the upper one. All other series of *WebRTC* pass the independence test, which hints that the failed series is merely an artifact induced by the small number of runs in the series (in this example, 10 runs per series). In such cases, it is important that the experimenter critically assesses *TriScale*'s results and increase (when necessary) the number of runs or series to improve the significance of results – or overrule the test (discussion in Section 2.7).

Evaluation in emulation. Using the MahiMahi emulator for the evaluation is expected to be the most favorable setting to test the reproducibility of the congestion-control schemes, since it allows to recreate the exact same test conditions and therefore improves the comparability between the runs. This however has an unexpected side-effect: while they appear to have a very stable behavior, *TCP BBR* and *TCP Cubic* would always fail the independence test. Actually, these two schemes are designed to use all the available bandwidth and, since MahiMahi artificially sets the latter to a fixed value, the two schemes always reach the exact same throughput. This leads to the exact same metric values; in other words, the throughput is *perfectly correlated* across runs. Naturally, this is an artifact of the experiment: the independence test will always

fail if all data samples have the same value.

TriScale always computes the KPIs and variability scores, even if the independence test fails. The experimenter is responsible to judge whether there is indeed true correlation in the data, or if one can overrule the test result and proceed with the analysis. In the example of *TCP BBR* and *TCP Cubic*, one can proceed.

KPIs. We illustrated in Section 2.2.1 how *TriScale*'s KPIs allow to unambiguously compare the performance of different schemes. Figure 2.2b shows the KPI for the average throughput (resp. one-way delay), defined as the estimate of the 25th (resp. 75th) percentile with a 75% confidence level for 10 runs with 30 s runtime. We use 30 s to compare with the Pantheon results shown in Figure 2.2a). However, five schemes fail to converge sufficiently often and thus do not appear in Figure 2.2b.

Variability scores. Although *TriScale*'s KPIs unambiguously compare the performance of diverse schemes, they only consider one series of runs, which does not indicate how reproducible the results actually are. *TriScale* investigates reproducibility using sequels and quantifies the expected variability in the KPI values with a variability score (Section 2.4.3). In this case study, we define the variability score as the difference between the 75th and 25th percentiles, estimated with 75% confidence. We compute the variability scores for our two performance dimensions (average throughput and one-way delay – Figure 2.8). The scores can be interpreted as follows: with 75% probability, the variability scores (orange bars) give the magnitude of variation expected (shall one perform infinitely many series) in the middle 50% of KPI values. Hence the variability score quantifies reproducibility: the larger the score, the less reproducible the results are.

Remark 3. *TriScale*'s variability scores are absolute values with units (*e.g.*, in Mbit/s). Arguably, it may be useful to use relative scores (in percentages) to compare the scores of different protocols.

Conclusion. This case study only considers emulation and one emulated path. As such, it does not aim to fully evaluate the performance of the different congestion-control schemes. Rather, it illustrates how *TriScale* may be used for an actual performance evaluation and the importance of carefully choosing the parameters of an experiment; such as the runtime (Figure 2.6). We highlight two important takeaways:

- It is important to critically consider *TriScale*'s results: the tests are intentionally conservatives to limit the risk of false positives (*e.g.*, not detecting correlation in the data),
- It is useful to collect more samples than strictly necessary: it improves the significance of the tests and therefore limits the risk of false negatives.

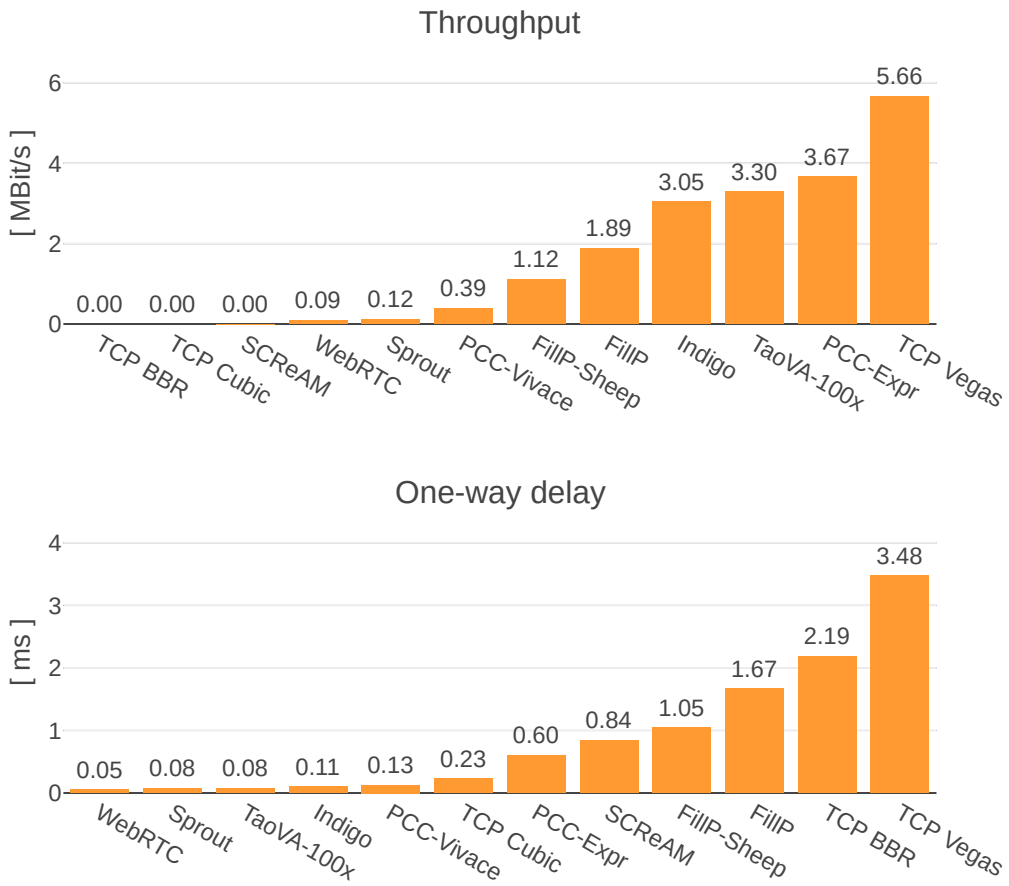


Figure 2.8 Variability scores computed by *TriScale* for the performance dimensions throughput and delay. *In this example, the variability scores are computed as the 25th to 75th percentile interval estimated with 75% confidence. From the variability scores, the user gets a quantification, with a 75% probability, of the range of variation in the KPI values for 50% of the series. The variability scores hence quantify reproducibility: the larger the scores, the less reproducible the results are.*

2.6 Implementation and Scalability of *TriScale*

2.6.1 Python Package

We implement *TriScale* in Python (≈1000 lines of code) and make it open source (Section 2.A). *TriScale*'s API contains one function for each timescale of the data analysis; *i.e.*, the computation of metrics, KPIs, and variability scores. Docstrings contain detailed information about the functions usage. Our implementation relies on standard scientific packages such as NumPy [5], Pandas [8], SciPy [9]. As the use of non-parametric statistics is not (yet) widespread, we had to implement some of the statistics used by *TriScale* (in particular the computation of CI using Thompson's method). We hope to see these functions integrated in a future release of SciPy.

Table 2.2 Scalability evaluation. *TriScale data analysis is fast and scales well with increasing input sizes. The most time-consuming element is the convergence test (Section 2.4.5) which is performed before the computation of metrics. Still, it generally takes less than one second for inputs (*i.e.*, the number of raw measurements in a run) of up to one million data points.*

Computation of	**Input size**	**Execution time** (approx.)
Metrics	1000	20 ms
	10'000	50 ms
	1'000'000	1 s
KPIs & Variability scores	100	10 ms
	1000	100 ms

It is important to produce useful visualizations to support the experimenter. Thus, we paid a particular attention to the plotting functions in *TriScale*. *TriScale* uses Plotly [7] to create interactive plots: one can zoom in and out in the plots, toggle the visibility of individual traces, read data point values on hover, etc. All the plots in this chapter are produced using *TriScale* and are "clickable": figures are hyperlinks leading to dynamic versions of the plots.

2.6.2 Scalability of *TriScale* Data Analysis

We evaluate the scalability of *TriScale* with respect to computation time; *i.e.*, how does the data analysis time scales with increasing input sizes. We only consider the time required for performing computations; other outputs such as logs and plots (*e.g.*, Figure 2.4a) are excluded. The complete scalability evaluation (including data, plots, and discussions) is available as complementary materials (Section 2.A). Generally, the computation time for the data analysis in *TriScale* scales linearly with the input size (Table 2.2): it is fast (less than 1 s for one million data points on a commodity laptop) and overall negligible compared to the data collection time.

2.7 Discussion, Limitations, and Future Work

Data collection. *TriScale* is not responsible for the execution of networking experiments, *i.e.*, it does not perform the data collection (Section 2.2). Frameworks specialized in data collection, such as Pantheon [199], already exist and *TriScale* can be integrated into these frameworks to create a fully-automated experimentation chain. Other examples include low-power wireless testbeds [160, 159, 115] and networking facilities [31, 62, 139], which could be combined with *TriScale* to build full-fledged benchmarking infrastructures [38].

Human-in-the-loop. *TriScale* automates the data analysis and implements tests that verify whether the required hypotheses hold. However, these tests are not perfect: the confidence will never be 100%. Moreover, the significance of such tests is always low when the sample sizes are small; *e.g.*, the independence test may flag correlated data when "correlation" is only an unlucky random variation (Figure 2.7). *TriScale* raises flags to avoid missing clear issues (*e.g.*, *LEDBAT* convergence time – Figure 2.6), but the experimenter must always critically assess *TriScale* results and potentially overrule them; *e.g.*, neglecting the correlation of perfect throughput for *TCP BBR* and *TCP Cubic*.

Ranking solutions. *TriScale* compares performance, but it does not rank. The results of a networking evaluation are always relative to a specific network and evaluation scenario. It is not trivial to generalize and claim that a solution A is better than a solution B. This problem relates to benchmarking and multi-objective optimization, which goes beyond the scope of *TriScale*.

Community guidelines. *TriScale* formalizes evaluation objectives (Section 2.4.4), but it does not dictate which parameters to use. Similarly, *TriScale* quantifies the variability of an experiment (Section 2.4.3), but it does not conclude whether the experiment is reproducible (Section 2.4.7). *TriScale* provides a framework to describe evaluations and analyze the data in a consistent and statistically sound manner. It is now up to the networking communities to set their own standards, parameters to use, and acceptable requirements; similar to what is already done in other disciplines [78]).

2.8 Related Work

The reproducibility of experiments and comparability of results are cornerstones of the scientific method. In recent years, several studies have highlighted the inability of scientists from various disciplines to reproduce their own experimental results [30, 145], often due to sloppy research protocols and faulty statistical analysis [39, 37, 158]. This problem has also been recognized within computer science [52, 187], where experiments are seldom reproducible and artifacts rarely shared.

Promoting reproducibility. To address this "reproducibility crisis" [30], several efforts aiming to incentivize a rigorous experimentation have gained momentum in computer science, including *e.g.*, ACM's badging system for publications [15]. Especially in the networking community – challenged by the need to carry out experiments on dynamic and uncontrollable conditions [44, 127] – several workshops [28, 81, 41], surveys [76], guidelines [29, 156, 109, 129], as well as teaching activities [200] have raised awareness on the reproducibility problem and promoted better experimentation practices. This large body of work mostly offers *qualitative* statements on how an

experiment should be performed and documented. Such qualitative statements emphasize for example the need to carefully choose when and how often to sample data [29], or suggest which methodology to adopt during performance evaluations [109]. However, there is no guarantee that following these recommendations leads to reproducible results, nor is there a concrete way to assess whether an experiment can be considered reproducible.

None of the existing works provide scientists with *quantitative* answers about how to concretely perform an experiment, *e.g.*, how many runs should be completed and how long should they be. *TriScale* fills this gap by providing quantitative answers to these questions with an experimental methodology grounded on robust non-parametric statistics. *TriScale* also allows to assess and compare the reproducibility of experimental results by computing unambiguous performance indicators and variability scores.

Supporting reproducibility. A large number of experimental facilities and tools have been developed in recent years to aid scientists and practitioners in carrying out reproducible networking studies [139]. Testbeds such as EmuLab [195] and FlexLab [150], as well as emulation tools such as MiniNet [86] and MahiMahi [137], enable the creation of artificial network conditions using a given specification or passively-observed traffic. Emulated conditions offer a more controlled environment than experiments faced with real-world traffic (*e.g.*, by transmitting data over the Internet [51, 34], cloud [62, 40], or wireless interfaces [16, 79, 125]). Still, they suffer from performance variability caused by the underlying hardware and software components, which hampers reproducibility [124]. To overcome these problems, several solutions have been proposed [65]: *e.g.*, revisiting operating system libraries [177], using virtualization [86, 102, 107], adaptable profiles [151], and fault patterns [1]. Other tools have been developed to support mobility experiments [50, 31], maximize the repeatability of interference generation [161], and enable researchers to consistently evaluate congestion control schemes or transport protocols [199]. Other works model the execution of experiments, and uses such models to quantify the similarity between different runs [165, 74].

While all aforementioned tools aim to improve reproducibility *during* the experiments, *TriScale* assists researchers *before* and *after* their execution. It does so by informing about the number and length of runs necessary to obtain a sufficient statistical significance, as well as by computing a score quantifying the variability of the results. Hence, *TriScale* complements the existing body of literature promoting and enhancing reproducibility in networking research.

2.9 Summary

Establishing a consistent methodology for the design of networking experiments and the analysis of their data is a crucial step towards a more rigorous and reproducible scientific activity. This chapter presented *TriScale*, the first concrete proposal in that direction.

TriScale implements a methodology grounded on non-parametric statistics into a framework that assists scientists in designing experiments and automating the data analysis. *TriScale* ultimately improves the legibility of results and helps quantifying the reproducibility of experiments, as highlighted in the case studies presented throughout the chapter. We expect *TriScale*'s open availability to actively encourage its use by the networking community and promote better experimentation practices in the short term. The quest towards highly-reproducible networking experiments remains open, but we believe that *TriScale* represents an important stepping stone towards an accepted standard for experimental evaluations in networking.

2.A Appendix – Artifacts and Links

2.A.1 Related Publications

Towards a Methodology for Experimental Evaluation in Low-Power Wireless Networking
Romain Jacob, Carlo Alberto Boano, Usman Raza, Marco Zimmerling, Lothar Thiele
CPS-IoTBench 2019. Montréal, Canada (April 2019)

Paper	10.3929/ethz-b-000325096
Presentation	10.3929/ethz-b-000349885
Video	youtu.be/XEwCqmU9Zzo

TriScale: A Framework Supporting Reproducible Networking Evaluations
Romain Jacob, Marco Zimmerling, Carlo Alberto Boano, Laurent Vanbever, Lothar Thiele
(*Under submission*) NSDI 2020. Santa Clara, CA, USA (February 2020)

Paper	10.5281/zenodo.3464274

Is low-power wireless networking a reproducible science?
Antonios Koskinas
Semester Thesis. ETH Zurich (January 2019)

Thesis	10.3929/ethz-b-000324251

2.A.2 Complementary Materials

Complementary materials for this chapters are available on GitHub, together with the dissertation source files. For all links below, replace `<root>` by "github.com/romain-jacob/doctoral-thesis/blob/master"

TEX sources	`<root>`/20_TriScale/
Figures	
— Static	`<root>`/20_TriScale/Figures/
— Dynamic	`<root>`/notebooks/triscale_plots.ipynb
Case studies	
— Congestion control	`<root>`/notebooks/triscale_pantheon.ipynb
— Low-power Wireless	`<root>`/notebooks/triscale_flocklab.ipynb
— Scalability	`<root>`/notebooks/triscale_scalability.ipynb
TriScale source code	10.5281/zenodo.3451417
Experiment data	10.5281/zenodo.3451417

3

Synchronous Transmissions Made Easy: Design Your Network Stack with *Baloo*

In the previous chapter, we discussed how to design and analyze networking experiments in general (Chapter 2). In the rest of this dissertation, we will focus on low-power wireless networking, and more specifically on a technique called *synchronous transmissions*.

Synchronous transmissions (ST) refers to a wireless approach for broadcasting messages in a multi-hop network using flooding. This is made efficient by letting multiple transmitters send the *same packet* at the *same time*; hence the name of *synchronous* transmissions.[1] ST has been proven highly reliable and energy efficient for low-power wireless networks. Furthermore, ST supports mobility by design thanks to the stateless logic of flooding-based communication.

Unfortunately, it is difficult to guarantee that multiple nodes actually send at the "same time". The required precision on synchronization depends, *e.g.*, on the physical layer speed, the radio modulation speed or the encoding scheme. For typical low-power wireless motes available today, the synchronization must be in the order of μs for ST to work reliably. Achieving such time synchronization requires to precisely control the timing of radio operations, which involves careful timer settings and interrupt handling. The integration of such "low-level software" within a entire network stack is challenging. Consequently, the adoption and development of ST-based network stacks has been hindered by the lack of usable and flexible design tools.

Thus, in this chapter, we study the feasibility of a design tool that would facilitate the development of network stacks based on ST (Figure 3.1); typically, such a tool would be useful for implementing our real-time protocol stacks (see Chapter 4 and Chapter 5).

[1] The name *concurrent transmissions* is also found in the literature.

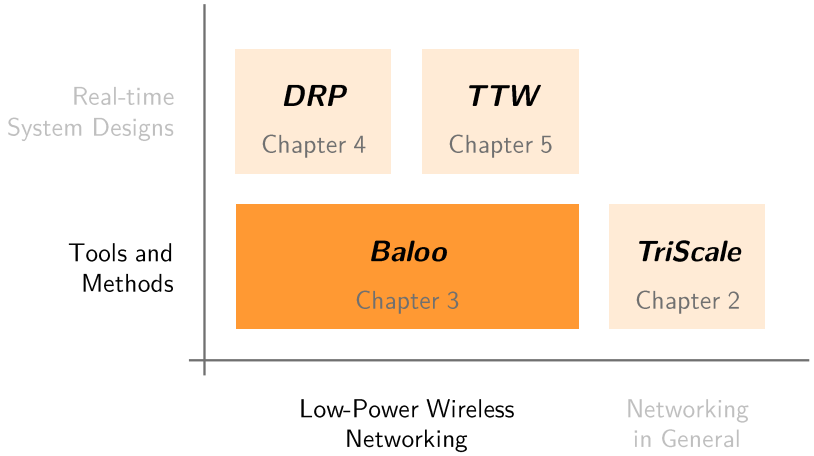


Figure 3.1 Positioning of this chapter in the dissertation. *This chapter presents Baloo, a design framework facilitating the implementation of low-power wireless networking protocols based on synchronous transmissions.*

Claim. We propose and implement *Baloo*, a design framework for network stacks based on synchronous transmissions. *Baloo* significantly lowers the entry barrier for harnessing the efficiency, reliability and mobility support of synchronous transmissions: users implement their protocol through a simple yet flexible API while *Baloo* handles all the complex low-level operations based on the users' inputs.

Baloo is flexible enough to implement a wide variety of network layer protocols, with only limited memory and energy overhead.

The material from this chapter builds upon the work from Jonas Bächli [25]. It relates to the following publication.

Synchronous Transmissions Made Easy: Design Your Network Stack with Baloo
Romain Jacob, Jonas Bächli, Reto Da Forno, Lothar Thiele
EWSN 2019. Beijung, China (February 2019)

3.1 Prolem Setting

Synchronous Transmissions (ST) is an increasingly used wireless communication technology for low-power multi-hop networks. Popularized by Glossy [70] in 2011, it has been proven to be highly reliable and energy efficient, as illustrated by the EWSN Dependability Competition [160], where all wining solutions were based on ST [67, 167, 114, 69, 120] in the past four years (2016 to 2019).

A *ST primitive* refers to a protocol that efficiently realizes broadcast (*i.e.*, any-to-all communication) in bounded time, usually relying on *flooding*. Flooding is a communication strategy that realizes broadcast by having all receivers of a packet retransmit this same packet to all their neighbours; the packet is thus

"flooded" through the whole network. ST makes flooding energy and time efficient by letting multiple wireless nodes transmit the packet *synchronously*, hence the name of *Synchronous Transmissions*. The successful reception of the packet can be achieved if the transmitters are tightly synchronized, thanks to *constructive interference* and the *capture effect* [201]. The synchronization requirements vary from sub-μs to tens of μs, depending on the platform and modulation scheme [201]. Such a broadcast primitive simplifies the design of network layer protocols: The underlying multi-hop network can be abstracted as a *virtual single-hop network* and thus be scheduled like a shared bus [72]. One may refer to Chapter 1 for more details on ST.

Since Glossy [70], many flavours of ST primitives have been proposed to improve performance in terms of reliability, latency, and energy consumption. To be more resilient to strong interference, Robust Flooding [114] is a primitive that modifies the RX-TX sequence from the original Glossy, whereas RedFixHop [66] uses hardware acknowledgements to minimize the number of retransmissions required. Instead, some primitives aim to minimize latency for specific traffic patterns. For example, Chaos [111] lets all nodes modify the packet being flooded to quickly aggregate information (*e.g.*, the max value of all sensor readings) or efficiently perform all-to-all data sharing to achieve distributed consensus [20]. Codecast [130] also targets many-to-many exchange for a larger amount of data. Pando [59] is another primitive focused on high throughput, which uses fountain code and packet pipelining for efficient data dissemination. Syncast [131] aims to reduce the radio on time required compared to Glossy to save energy. Finally, Less is More (LiM) [206] is a primitive that reduces energy consumption using learning to avoid unnecessary retransmissions during flooding.

All these primitives share the same drawback: Successful ST requires low-level control of timers and radio events in order to meet ST tight synchronization requirements (the order of μs). This degree of accuracy is difficult to achieve as it requires a detailed knowledge of the underlying hardware, low-level control of the radio operations, and a very careful management of software delays.

As a result, designing a network stack based on ST is a complex and time consuming task, for which only few solutions have been proposed. One of the first was the Low-power Wireless Bus (LWB) [72], which tries to flexibly support all kinds of traffic patterns in a balanced trade-off between latency and energy consumption. The same group designed eLWB [173], a variation of LWB tailored to event-based data collection. Sleeping Beauty [155] was later proposed to minimize energy consumption for data collection scenarios with many redundant sensor nodes. Time-Triggered-Wireless (TTW – Chapter 5) [100] was designed to minimize the end-to-end latency between communicating application tasks. Finally, Crystal [95] has been proposed as a network stack specialized for sporadic data collection. All these network stacks solely rely on Glossy as ST primitive. In principle however, the same protocol logic could benefit

from *multiple* primitives. For example, an LWB network could use Robust Flooding [114] in case of high interference, then revert to Glossy [70] for better time synchronization. If nodes need reprogramming, the software update can be quickly disseminated using Pando [59]. Designing a modular network stack supporting multiple ST primitives adds a new level of complexity.

Key Research Questions

Question 1 Can we facilitate the design of wireless network stacks based on Synchronous Transmission?

Question 2 Can we implement flexible and adaptive protocols, potentially leveraging multiple ST primitives, while guaranteeing that the timing requirements of ST are met?

The problem. To facilitate the network stack design (**Question 1**), a natural idea is to separate the concern of the timely execution of the primitives from the implementation of the protocol logic. One way to achieve such separation of concerns is to use a *middleware* as part of the network stack.

The idea of a middleware for Wireless Sensor Networks (WSN) is not new, and the main challenge in such an endeavour is well-known. As phrased by Mottola and Picco [134], "*striking a balance between flexibility and complexity in providing access to low-level features is probably one of the toughest, yet most important, problems in WSN middleware*".

The design of a middleware for ST is particularly challenging. Indeed, meeting the tight timing requirements for ST is directly conflicting with the concept of abstraction of a middleware: How to guarantee that the network layer does not hinder the timing accuracy for ST if it is itself unaware of the execution of the primitives? That is **Question 2**.

The challenge. A middleware for ST should meet the following requirements.

Usability The middleware must realize a well-defined interface enabling runtime control from the network layer (which implements the protocol logic) over the execution of the underlying ST primitives.

Generality The middleware must enable the implementation of a large variety of network layer protocols.

Versatility The middleware must enable one network layer protocol to use multiple ST primitives and switch between them at runtime.

Synchronicity The middleware must guarantee to respect the time synchronization requirements for ST (from sub-μs to tens of μs [201]).

Our solution. To address these challenges, we have designed *Baloo*, a flexible design framework for low-power network stacks based on ST.[2] *Baloo* provides a large set of features enabling performant protocol designs, while abstracting away low-level hardware management such as interrupt handling and radio core control. In summary:

- We propose *Baloo*, a flexible design framework for low-power wireless network stacks based on ST, illustrated in Figure 3.2.
- We present the design of a middleware layer that meets all our requirements. This middleware forms the core component of *Baloo*.
- We showcase the usability of *Baloo* by re-implementing three well-known network stacks using ST: the Low-power Wireless Bus (LWB) [72], Sleeping Beauty [155], and Crystal [95].
- We illustrate the portability of *Baloo* by providing implementations for two platforms – the CC430 SoC [180] and the old but still heavily used TelosB mote [17].
- We demonstrate that *Baloo* induces only limited performance overhead (memory usage, radio duty cycle) compared to the original implementations.

This chapter *is not* meant to cover all details and inner mechanisms of *Baloo*, but mainly presents the core concepts of the framework. *Baloo* is open source and the complete technical documentation is available online (Section 3.A).

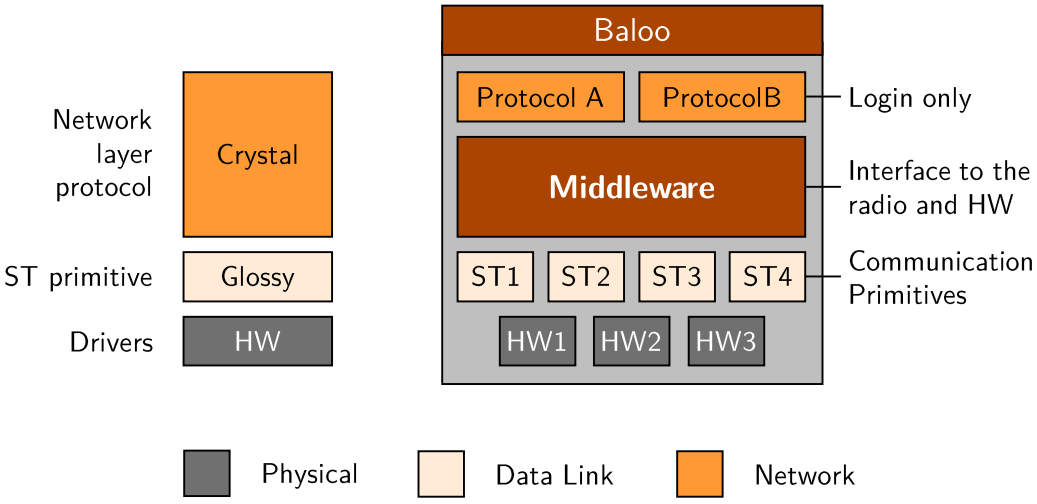
3.2 Overview of *Baloo*

This section presents an overview of the concepts of *Baloo*. The implementation of these concepts using a middleware will be described in the next section (Section 3.3).

Baloo is a flexible framework designed to harness the potential of the Synchronous Transmissions (ST) technology and make it more accessible. *Baloo* uses Time Division Multiple Access (TDMA) rounds made of communication slots. A ST primitive is executed in each slot. All necessarily control information is sent by a central node in the first slot of each round. The core of *Baloo* is made of a middleware layer (see Figure 3.2) which isolates the network layer from the lower layers. Concretely, this separates the management of the radio and timer settings from the implementation of the protocol logic.

Why Synchronous Transmissions? As discussed in Section 3.1, ST is a wireless communication technique known to be reliable, fast, and energy efficient. ST primitives communicate using so-called *floods*, which realize an

[2] The framework provides the "bare necessities" for the design and implementation of ST-based network stacks; so we called it *Baloo*.



(a) The implementation of the network layer protocol (Crystal) couples the interface to the underlying ST primitive (Glossy) and the protocol logic, *i.e.*, how long are the communication rounds, which radio channel is used, etc.

(b) Thanks to its additional middleware layer, *Baloo* flexibly supports multiple ST primitives and significantly reduces the efforts required to implement network layer protocols compared to traditional stacks, like LWB [72] or Crystal [95].

Figure 3.2 Crystal [95] is a typical example of network stack based on ST (Figure 3.2a). Conversely, *Baloo* is a flexible design framework. It is based on a middleware layer that separates the concern of timely execution of ST primitives from the implementation of the protocol logic (Figure 3.2b).

any-to-all communication. Thus, ST seamlessly supports multiple types of transmission patterns (*i.e.*, unicast, multicast, broadcast). As a result, ST enables to abstract away the complexity of a multi-hop mesh into a *virtual single-hop network*. Furthermore, some ST primitives (*e.g.*, Glossy [70] or Robust Flooding [114]) provide tight bounds on the completion time of a flood, given the payload size and network diameter.

This makes ST particularly suited for a time-triggered communication scheme. Within one bounded time slot, one can schedule a communication from one to any (set of) node(s) in the network, which greatly simplifies the design of a network layer protocol.

Baloo uses Glossy [70] as default ST primitive, but it also supports other primitives, *e.g.*, Chaos [111]. In principle, *Baloo* is compatible with arbitrarily many other primitives (see Section 3.3.5), thus addressing *Versatility*.

Round-based Design. To maximize the benefits of ST, *Baloo* organizes communication in TDMA rounds, with dedicated time slots assigned to specific nodes which are then allowed to initiate a transmission in this slot. The first

slot in each round is assigned to a central node, called the *host*, to send some control information (see below). This *control slot* is then followed by arbitrarily many *data slots*. Nodes turn their radio off between rounds to save energy.

While this framework may look restrictive and hinder *Generality*, such round-based design is in fact very generic, and compatible with many (if not all) ST-based network stacks proposed so far in the literature. The flexibility and limitations of *Baloo* will be discussed in the evaluation (Section 3.5 and 3.6).

Control Information. In *Baloo*, the control packet, sent at the beginning of each round, plays a key role. It is constructed such that if a node receives a control packet from the host, this nodes knows exactly

- how to execute the current communication round, and
- when to wake up for the next round.

Thus, the control packet contains both *schedule information* (*e.g.*, the slot assignment for the round or the time interval before the next round) and *configuration parameters*, like the length of the slots or the number of retransmissions. The control packet is broadcast using Glossy [70], which is also used to synchronize the whole network.

Baloo is very flexible (*Usability*, *Generality*); both schedule and configuration can be updated at anytime by the host and the whole network adapts to follow the instructions. This poses the problem of a node not correctly decoding a control packet, thus having possibly outdated control information.

Consequently, *Baloo* adopts the following fail-safe mechanism: *a node does not participates in a communication round unless it correctly decodes the control packet*. This guarantees that, even in case of packet losses, a node will never disturb the execution of the rest of the network.

A Middleware to Provide the Right Level of Abstraction. The main challenge in the design of *Baloo* is the definition of an interface that isolates the management of the radio (*i.e.*, running the ST communication primitives) from the implementation of the protocol logic at the network layer. *Baloo* realizes this interface using a middleware layer that is responsible for the following tasks:

- The middleware organizes the timers and controls the radio operations (*i.e.*, it executes the ST primitives).
- The middleware manages the communication round operations according to the control information received from the host.
- The middleware executes callback functions, which are used to interact with the application running above the network layer (*i.e.*, passing packet payload and implementing the protocol logic).

The middleware is a *fixed piece of software* which can be configured but neither accessed nor modified by the network layer. The protocol logic (payload

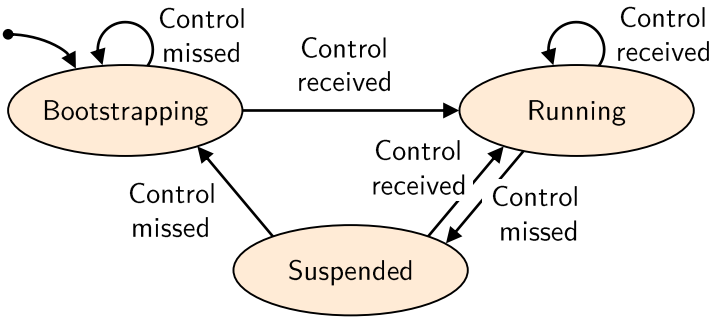


Figure 3.3 The middleware in *Baloo* implements a minimal state machine, sufficient to capture the desired behaviour of a node at physical layer. *A node either executes normally (Running), stays synchronized but does not participate to the communication rounds (Suspended), or continuously listens for an incoming control packet (Bootstrapping).*

management, state keeping, etc.) is implemented entirely within the callback functions.[3] The middleware interface is illustrated in Figure 3.4.

With this approach, all low-level programming complexities are managed by the middleware and let the network designer focus on the main task: design the protocol logic of the network layer.

These concepts form the core of *Baloo* and address the *Usability*, *Generality*, and *Versatility* requirements of a flexible network stack design. Additional concepts are required to ensure that the timing requirements of ST are met (*Synchronicity*): this is the focus of the next section.

3.3 Implementing the Concepts

The previous section described the general concepts of *Baloo*. In this section, we present how we implemented these concepts to meet the *Synchronicity* requirement and complete the design of *Baloo*.

3.3.1 Contiki as Operating System

Baloo relies on the availability of ST primitives (*e.g.*, Glossy [70], Chaos [111], etc.). Most openly available primitives use Contiki [60] as the underlying operating system, which made Contiki an obvious choice to implement *Baloo*. We have ported these primitives to the new version of the OS: Contiki-NG [11].[4]

Contiki is a cooperative multi-threaded OS, tailored for resource-constrained

[3] The different callbacks are further described in Section 3.3.3.

[4] v4.2, released in November 2018

devices in the Internet of Things. The middleware layer is implemented as the "master" protothread [61], where most of the program is executed. The middleware implements the communication rounds, controls the radio operations, and executes the callback functions in which the network protocol logic is implemented (see Section 3.3.3).

3.3.2 Minimal State Machine

For generality and simplicity, the middleware implements only a minimal state machine. A node is either in *Bootstrapping*, *Suspended*, or *Running* state. State transitions result from (un)successful receptions of control packets (see Figure 3.3). When bootstrapping, a node continuously listens for a control packet. In the *Suspended* state, a node does not participate in the round and will sleep until the next round.

As described in Section 3.2, a node may participate in a round (*i.e.*, be in the *Running* state) if and only if it correctly receives the control packet at the beginning of the round. The default behaviour of *Baloo* is that a node suspends itself if it misses a control packet, and goes back to Bootstrapping if it misses two in a row. A node exits the *Bootstrapping* state whenever it receives a control packet containing both scheduling and configuration information, *i.e.*, when a node knows with certainty how it is expected to operate. If necessary, the network layer protocol can extend this minimal state machine using one of the callback functions (see Section 3.4.1).

3.3.3 Middleware Callback Functions

Baloo uses callback functions to implement the network layer protocol logic. This is how the network layer interacts with the middleware at runtime (*Usability*). There are five different callbacks, which have specific purposes and are executed by the middleware at precise points in time, as illustrated in Figure 3.4.

`on_control_slot_post()`
: is executed at the end of the control slot. It is used to process the received control information and prepare for the round.

`on_slot_pre()`
: is executed before each data slot. It is used to pass the payload to send to the middleware, if any.

`on_slot_post()`
: is executed at the end of each data slot. It is used to process the received payload, if any.

`on_round_finished()`
: is executed at the end of the round. It is used to do more time consuming state management or data processing.

`on_bootstrap_timeout()`
: is executed when a node fails to bootstrap (*i.e.*, it has listened for some time without receiving any control packet). This callback allows nodes to go to sleep and retry bootstrapping later, in order to save energy.

These callback functions are also used to implement more advanced features of *Baloo* (*e.g.*, skipping or repeating a slot), which are briefly presented in Section 3.4.

3.3.4 Achieving Timeliness of Execution

The callback functions enable flexible interactions between the network layer and the middleware. While this is key to address *Usability* and *Generality*, it also inherently couples the two software components, thus challenging the timely execution of the middleware and compromising *Synchronicity*.

Indeed, the callbacks execute between communication slots or between rounds (see Figure 3.4), which must start synchronously on all nodes to permit successful ST. The middleware could interrupt an overrunning callback to ensure synchronicity, but that is not desirable. In general, an interrupted callback would have to be considered as a failure by the network layer; then successful ST at the lower layer would not really matter anyway.

To mitigate this problem, the middleware *monitors* the execution time of the callbacks. If a callback overruns and the middleware cannot guarantee the timely execution of the next slot, this slot is skipped (*i.e.*, the node does not participate at all in this slot) and a notification event is sent to the network layer.

With this approach, *Baloo* can guarantee to respect the timing requirement for ST *under the condition that the callbacks have enough time to complete their execution*.[5] To satisfy this condition, the available time between slots for the execution of the callbacks is controlled by a dedicated configuration parameter: the *gap time*. Since callbacks implement the network layer protocol logic, it can only be the responsibility of the network designer to set suitable gap times such that the *Synchronicity* requirement is met. Guidelines for setting such parameters (and in general: how-to use *Baloo*) are part of the online documentation (Section 3.A).

[5] This default strategy may lead to a starvation problem if a callback "never" returns, *e.g.*, if it relies on another software sitting at higher layers. One advanced feature lets the middleware interrupt overrunning callbacks (see Section 3.4.1).

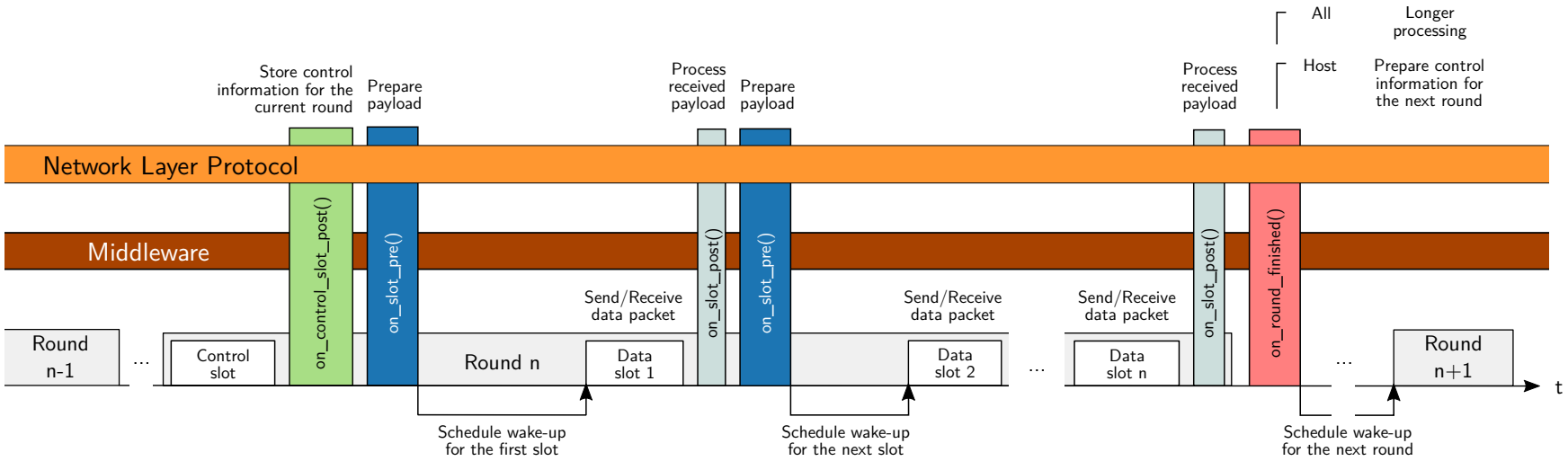


Figure 3.4 The protocol logic, *i.e.*, the handling of application payloads and the definition of the desired control parameters, is implemented in callback functions. These callbacks are triggered by the middleware before and after each slot and at the end of a round. The middleware schedules the wake-up of the radio core and executes the ST primitives.

3.3.5 Supporting Multiple ST Primitives

Compared to previously proposed low-power network stacks, one key difference of *Baloo* is that it flexibly supports multiple ST primitives (*Versatility*). This is difficult given the nature of ST, which requires tight timing of radio events (from sub-μs to tens of μs depending on the primitive [201]).

In practice, achieving such synchronization requires a direct monitoring of hardware timers and a custom implementation of the associated Interrupt Service Routines (ISR) for each ST primitive executed by the middleware in *Baloo*. However, there cannot be multiple implementation of the same ISR. Thus, supporting multiple ST primitives in the same stack requires to extract the interrupt management from the ST code, which becomes a shared software component between different primitives.

Technically, we implemented this using a renaming trick. Indeed, each ST primitive has its own ISR implementation for the radio timer, but *Baloo* never uses more than one ST primitive at the same time (*i.e.*, one per data slot). The middleware must only execute the instructions of the ISR from the currently running primitive. Thus, we can encapsulate the ISR of each primitive into a dedicated (*i.e.*, unique) function and implement the radio timer ISR as a simple "switch" function. A global variable keeps track of the currently running primitive; whenever the radio timer fires, the corresponding primitive "ISR function" is executed.

The only difference with the original primitives' implementation is an additional software delay between the radio interrupt and the execution of the ISR's instructions (the few ticks of delay to execute the switch). This adds a negligible synchronization error due to differences in clock speed across different nodes.[6]

Using this approach, *Baloo* currently supports two ST primitives, Glossy [70] and Chaos [111], as well as a classical strobing communication primitive: one node transmits its packet, multiple times, while all other nodes only listen. Practically, there is no limitation on the number of primitives that *Baloo* can support, apart from the available memory.

3.4 Advanced Functionalities

In Section 3.2 and 3.3 we presented the general concepts of *Baloo* and how we implemented them to meet the requirements presented in Section 3.1. To further extend the variety of network layer protocols that can be implemented using *Baloo*, we have enriched the framework with various features, most of

[6] Assuming an absolute clock drift of 100 ppm between two nodes (which is pessimistic), the error introduced is ~ 0.24 picosecond per tick of delay.

which have been used in previous protocols and proved themselves useful. We briefly present these features in this section.

3.4.1 High-level Functions

Detection of interference Low-power wireless networks often suffer from interference. Multiple strategies have been proposed to escape and/or mitigate its effects.

Baloo allows to monitor the power level on the channel being used during a slot. This information can be used to detect potential interference and react accordingly.

This feature is used *e.g.*, in Crystal [95].

Advanced state machine The middleware in *Baloo* implements a minimal but sufficient state machine composed of three states (*Running*, *Suspended*, *Bootstrapping*; see Section 3.3.2 and Figure 3.3).

Baloo lets the network layer protocol implement a more advanced state machine. The return value of the `on_control_slot_post()` callback is used to inform the middleware of the desired behaviour for the node, *i.e.*, whether it should be in the *Running*, *Suspended*, or *Bootstrapping* state for the coming round.

This feature is used *e.g.*, in LWB [72].

Starvation protection Skipping slots due to overrunning callbacks may lead to starvation problems. The middleware behaviour can be modified to interrupt these overruns. If and when this occurs, the interrupted node will suspend its operation for the coming slot (or the complete round, in case the `on_control_slot_post()` over-runs).

At the time of writing, this feature is **not included** in the publicly available implementation of *Baloo*.

3.4.2 Scheduling Features

Contention slots In a contention slot, all nodes are allowed to transmit their own packet; they "contend" for access to the wireless medium. The successful reception of one of the packets remains possible due to the capture effect [201].

Contention slots are used in many protocols, including LWB [72] and Crystal [95].

Per-slot configuration By default, the same configuration parameters are used for all slots in the same round. *Baloo* lets the network layer specify some configuration on a per-slot basis. These optional parameters are

sent by the host as part of the control packet.

Crystal [95] for example uses different number of retransmissions for data and acknowledgement packets (the latter are retransmitted more often).

Static schedule and configuration In many network layer protocols, the scheduling policy is static: either the control information remains the same or it changes according to some offline algorithm.

In such cases, *Baloo* can spare the overhead of sending redundant information in the control packet, thus saving time and energy. All nodes are then responsible to locally update their control information.

Static schedules are used *e.g.*, in TTW [100] or Crystal [95].

Skipping slots and rounds It is sometimes useful that some nodes do not participate in certain slots, or even skip complete rounds, *e.g.*, to save energy, or to improve performance in very dense networks.

Baloo lets the network layer trigger slot skipping using the return value of the `on_slot_pre()` callback. To skip an entire round, one can return *Suspended* in the `on_control_slot_post()` callback (see Section 3.4.1).

This feature is used *e.g.*, in Sleeping Beauty [155].

Repeating slots or rounds On the contrary, it may be useful to repeat the execution of specific slots, or even entire communication rounds (*e.g.*, when the number of slots required in a round is dynamic) or to retransmit lost packets.

Baloo lets the network layer trigger slot and round repeat using the return value of the `on_slot_post()` callback.

- If the slot repeat flag is received, the middleware re-executes the same slot.
- If the round repeat flag is received, the middleware immediately restarts executing from the first slot of the round.

This feature is used *e.g.*, in Crystal [95].

3.4.3 Radio Settings

Radio channel setting *Baloo* lets the network designer select the radio frequency channel for each slot. This can be useful to proactively or reactively hop between channels in case of interference.

This feature is used *e.g.*, in Crystal [95].

Transmit power setting *Baloo* lets the network designer set the desired transmit power, possibly changing between each communication slot.

3.5 Performance Evaluation

This section presents an evaluation of *Baloo*. We look first into qualitative aspects. We argue that *Baloo* is indeed usable and validate the premise that it makes it easy to design a network stack based on ST. We then discuss quantitative aspects by looking at the performance overhead of using *Baloo* compared to original implementations.

3.5.1 Qualitative Evaluation

The evaluation of software usability is a challenging task that suffers from almost unavoidable bias. To support our claim that *Baloo* is indeed easy to use, we used it ourselves to perform one of the most time consuming task in experimental research: the re-implementation of someone else's protocol. We re-implemented the protocol logic of three network stacks: Crystal [95], Sleeping Beauty [155], and LWB [72]. We chose these protocols because:

- They are well-known solutions from the literature, considering different types of scenario.
- Together, they use most of the features offered by *Baloo*.
- The authors' source code is publicly available.

It is fair to say that the fact that we can use our own software brings only little evidence of the usability of *Baloo*. Indeed, its usability will be ultimately demonstrated if and when other people start using it to implement their own protocols. To facilitate this, the code of *Baloo* is openly available, including demo applications, and is accompanied by a detailed documentation of its features and how to use them. Naturally, our re-implementations of Crystal, Sleeping Beauty, and LWB are also available (Section 3.A).

In addition, we used the 2019 EWSN Dependability Competition [12] as a case study. In this competition, networking solutions must perform well across a wide range of input parameters (*e.g.*, data rates or payload sizes), which demands the network stack to be adaptive. This is a perfect application for *Baloo*: by design, the middleware takes care of adjusting the timing of operations (*i.e.*, when the ST primitives should be executed) based on the application parameters (*e.g.*, the payload size). Furthermore, one can leverage the availability of different primitives. For example, after a data packet has been sent using Glossy [70], one can efficiently collect acknowledgements from all destinations using Chaos [111].

We tested the *Usability* of *Baloo* by having master students (who did not have any prior knowledge of *Baloo*, nor ST) competing using the framework [136]. Naturally, they did not beat teams of experienced researchers, but they did perform well. As they put it themselves in their report:

> "We had not much prior experience in WSN protocol design. While the result is not perfect, we managed implement our protocol within 8 weeks of part-time work. We would not have been able to do that without *Baloo*." [157, 135]

Finally, another important qualitative aspect of *Baloo* is its portability. The underlying middleware has been designed to minimize the software parts that are platform-dependent, and those have been isolated as much as possible. Essentially, the platform-dependent part is limited to the hardware timer interface and the radio drivers (further discussed in Section 3.6). *Baloo* is readily available on two platforms, the CC430 SoC [180] and the TelosB mote [17]. Thanks to the abstraction provided by the middleware, the network layer protocol implementations using *Baloo* are *platform agnostic*. In other words, the same network layer implementation can be used to compile binaries for any platform supported by *Baloo*. We argue that these elements, altogether, show the usability of *Baloo*.

3.5.2 Quantitative Evaluation

Abstraction and flexibility usually impact quantitative performance metrics. In this section, we evaluate the performance overhead of *Baloo* along four metrics: the packet reception rate (PRR), the radio duty cycle (DC), the binary size, and the number of lines of code.

We performed this evaluation using our three re-implementations of Crystal [95], Sleeping Beauty [155], and LWB [72]. It is important to clarify the objective of the experiments we conducted: the goal is to evaluate the *performance overhead* of using *Baloo* compared to native implementations; not to evaluate the actual protocol performances.

Experimental Setup. All our experiments were conducted on Flocklab [115] as it is the only public testbed featuring both CC430 SoC [180] and TelosB motes [17], the two platforms for which *Baloo* is available. All tests ran for one hour on 26 nodes, leading to tens of thousand data packets exchanged for each protocol. As much as possible, we designed the experiments to match those from the original protocol papers [95, 155, 72].

Crystal We ran tests varying the number of source nodes U that have a packet to transmit in each round. We used $U = \{0, 1, 20\}$. All other parameters were set according to the author's paper (first row of Table 2 in [95]).

Sleeping Beauty We ran tests varying the percentage of nodes that that have a packet to transmit in each round. We used 12.5%, 25%, and 50% of the available nodes. All other parameters were set according to the author's paper [155].

LWB We ran tests varying the inter-packet interval IPI of the data stream registered by each node in the network. We used $IPI = \{4\,\text{s}, 30\,\text{s}\}$ and the dynamic scheduler aiming to minimize energy consumption.

In each case, we compared: *(i)* the results reported in the original papers, *(ii)* the results we obtained by running the publicly available code, and *(iii)* our re-implementations using *Baloo* on both the TelosB motes and *(iv)* the CC430 SoC. All results are summarized in Tables 3.1 to 3.4. Before discussing each metrics in details, some comments are useful.

- Although the original code of Sleeping Beauty is openly available [10], we failed to run the protocol successfully. More precisely, the observed behaviour was quite different from the paper description and led to inexplicably poor results. It did not seem fair to present these as a truthful measure of the protocol performance. Thus, we do not report any results for the native code for Sleeping Beauty.
- The original Sleeping Beauty paper does not report exact values for PRR and duty cycle. The values from Table 3.1 and 3.2 were read from Fig. 9 in [155].
- The original LWB paper presents results from an implementation on TelosB, but the available code from the authors is for the CC430 SoC [71]. Thus, we compare our re-implementations with the native code, but not with the original paper results.

Packet Reception Rate (PRR). We consider the end-to-end PRR: the percentage of the packets generated by the application at the source nodes that have reached their intended destination. We count a packet as lost *only if none of its transmissions* has been successfully received at the destination. In Crystal [95] for example, data packets may be lost and successfully received later; that does not impact the PRR.

We do not expect any overhead in term of reliability from using *Baloo*, as this metric essentially depends on the underlying ST primitive and the network layer protocol logic; two elements not modified by the framework. This metric mostly verifies that our re-implementations "work". The results in Table 3.1 indeed show similiar PRR for all implementations, with one notable exception.

Baloo on CC430 for Crystal with $U = 20$ performs poorly. After closer investigation, it appears that the success rate of the capture effect on the CC430 SoC is much lower than on the TelosB (presumably due to the different modulation schemes used by the radio: 2-FSK and O-QPSK respectively). When $U = 20$, it is highly probable that all the one-hop neighbours of the sink are selected source nodes that generate a packet in a round (20 out of 25 nodes available on Flocklab). As Crystal transmits all its data packets using contention slots,[7] the sink only rarely receives packets in these rounds, thus resulting in poor PRR.

[7] Successful contention slots rely on capture effect, see Section 3.4

Table 3.1 End-to-end packet reception rate (PRR), expressed in percentage (%)

	Crystal			Sleeping Beauty			LWB	
	$U = 0$	$U = 1$	$U = 20$	12.5%	25%	50%	$IPI = 30s$	$IPI = 4s$
Original paper	na	100	100	≈ 99	≈ 99	≈ 99	na	na
Native code	na	100	99.91	x	x	x	99.79	99.56
Baloo on TelosB	na	100	100	99.43	99.75	99.04	100	99.92
Baloo on CC430	na	100	81.36	99.48	97.39	98.72	99.81	99.85

Table 3.2 Average radio duty cycle (DC) across all nodes but the data sink, expressed in percentages (%)
For Sleeping Beauty, the reported values exclude the bootstrapping phase.

	Crystal			Sleeping Beauty			LWB	
	$U = 0$	$U = 1$	$U = 20$	12.5%	25%	50%	$IPI = 30s$	$IPI = 4s$
Original paper	0.367	0.487	2.890	≈ 0.2	≈ 0.3	≈ 0.7	na	na
Native code	0.321	0.440	2.115	x	x	x	0.79	5.715
Baloo on TelosB	0.362	0.493	2.703	0.082	0.115	0.283	0.976	6.646
Baloo on CC430	0.355	0.494	2.640	0.070	0.106	0.270	0.914	6.007

Table 3.3 Estimate of the binary size of the network layer protocol code (in kB)

	Crystal	Sleeping Beauty	LWB
Native code	16.87	17.47	19.11
Baloo on TelosB	16.36	20.71	19.7
Baloo on CC430	16.38	19.3	19.18

Table 3.4 Estimate of the number of lines of code in the implementation of the network layer protocol

	Crystal	Sleeping Beauty	LWB
Native code	931	751	1881
Baloo	539	797	1029
ratio	0.58	1.06	0.55

Radio Duty Cycle (DC). The radio duty cycle (DC) is expected to reveal more of the actual overhead induced by *Baloo*. Our results are summarized in Table 3.2.

The LWB experiment perfectly matches our expectations: the DC are comparable, with a slight increase for *Baloo* (5 to 15% more compared to the native code), which is due to the cost of sending more information in the control packet.

In the original *Baloo* paper [96], we reported surprising results regarding Crystal: Our results on the same platform (TelosB) showed significantly higher DC both for the native code and our re-implementation (from 50% to 100% increase) whereas on the CC430 SoC, results are more comparable. This was due to a misconfiguration of the clear channel assessment (CCA) threshold value: there is an offset of approximately $45dB$ between the value set in the register and the actual sensitivity of the CCA pin.[8] Consequently, when setting -60 dBm (the value suggested by the Crystal authors [95]), we actually obtained a CCA sensitivity of about -105 dBm, which is lower than the noise floor. Hence, Crystal consistently detected possible interference and (often needlessly) prolonged the communication rounds, thus artificially increasing the DC. We have re-run these experiments with the correct CCA setting (*i.e.*, -60 dBm) and validate that, as expected, the DC is slightly increased by *Baloo* (Table 3.2).

The results for Sleeping Beauty are more surprising. In spite of the overhead induced by *Baloo*, our re-implementation achieves about 2.5x reduction in DC. It is unclear what can be the source of such difference. As we based our re-implementation only on the original paper description [155], one possible explanation is that we might lack some of the original protocol features, that would induce more radio on time. However, the good results we obtain with our re-implementation would question the usefulness of such features.

Binary size. The binary file size is another metric where we expect *Baloo* to induce some overhead, as the middleware introduces additional files, types and features that are not always necessary for all protocols.

Since the protocol implementations we looked at are based on different versions of the Contiki OS, we tried to evaluate the actual size of the *network layer protocol* only by deducing the memory required for the OS. The OS memory requirements were obtained by looking at the size of a minimal "hello-world" application. Table 3.3 reports the difference between the total and "hello-world" binary sizes, a rough *estimate* of the memory required by the network layer protocol implementation.[9]

[8] [179]: RSSI / Energy Detection, page 48

[9] More advanced metrics could be used, *e.g.*, summing the size of relevant functions in the object file. We chose to used a very simple approach because our goal is only to give an estimate of the impact of *Baloo* on the memory requirements.

Actually, the memory size of our re-implementations is comparable to that of the native codes. Likely, t his is due to the configurable nature of the framework. Many features are available, but the protocol designer flexibly selects which are required, thus limiting the size of the compiled code. Furthermore, the structure imposed by the framework may lead to a more concise implementation, as discussed next.

Lines of Code. The last metric we considered is the number of lines of code that is part of the *network layer protocol* (*i.e.*, for the *Baloo* re-implementations, only the callbacks and custom functions; not the middleware code). This is arguably a rough metric, for at least two reasons: *(i)* none of the implementations aimed to minimise its code size; *(ii)* in the original implementations, it is not easy to isolate the code implementing the protocol logic from the interface with the lower layers (precisely, this is one of the differences with *Baloo*). Still, the number of lines of code provides some insights on the potential benefits of *Baloo* in terms of usability.

The results in Table 3.4 show that using *Baloo* can significantly reduce the amount of code required to implement some network layer protocol logic (up to 45% reduction for LWB). More importantly, the protocol implementations in *Baloo do not contain* any timer setting or register accesses, as these are handled directly by the middleware.

Summary. Ultimately, our quantitative evaluation shows through a few examples that implementations using *Baloo* perform well and that the framework induces only limited (if any) overhead in terms of radio duty cycle and binary size.

3.6 Discussion and Limitations

We argued that *Baloo* is a usable, flexible, and performant design framework (Section 3.5). To complete the description, we now detail hardware and software requirements and discuss the portability and limitations of the framework.

3.6.1 Requirements

Hardware requirements. The only strict hardware requirement of *Baloo* is one dedicated Capture Compare Register (as required by any time-triggered protocol). The actual timer frequency is not important; a standard 32 kHz clock is already fast enough. This timer is used to schedule the communication slots, wake-up times, and callback executions.

Both supported platforms feature an MSP430 CPU, but this is not a constraint.

An ARM core like the ones embedded on the nRF52840 [162] or the OpenMoteB [6] platforms would work as well. It would eventually be even more flexible given the support for interrupt priorities.

Software requirements. *Baloo* requires a software-extended timer implementation to enable the scheduling of firing epochs further than one roll-over of the timer. This is (surprisingly) not part of Contiki by default, but it is a rather minor extension. Some features of *Baloo* rely on radio functions (*e.g.*, the noise detection); these features are obviously platform-dependent. The rest of the platform-dependent software in *Baloo* is a mapping between ST primitive functions and generic macros used by the middleware.

3.6.2 Portability

Baloo itself has limited hardware and software requirements (Section 3.6.1). The main constraint comes from the availability of ST primitives, which are notoriously difficult to implement; but this is independent of the framework. Assuming ST primitives are available, the requirements and efforts to port *Baloo* to a new platform are limited.

We implemented *Baloo* using Contiki (see Section 3.3), which turned out having pros and cons. On the one hand, it facilitates the port of *Baloo* to other platforms *that already run Contiki*. On the other hands, it makes *Baloo* harder to port to *other platforms*, as it requires to port the Contiki OS first. It has been a limitation in some later projects (see Section 3.7).

Furthermore, *Baloo* does not require much of the complex machinery of a full-fledge operating system. Thus, a bare-metal implementation of *Baloo* could bring multiple benefits: increased reliability (as there is no interference from the OS), lighter weight, and simpler to port on any platform (as there is no need to port an entire OS first).

At the time of writing, the only known publicly available implementation of *Baloo* uses Contiki. Ongoing development efforts are discussed in Section 3.7.

3.6.3 Limitations

Baloo is a framework that facilitates the design of ST-based network stacks by providing some level of abstraction. We showed in Section 3.5 that this abstraction has only a moderate impact on performance. However, abstraction also limits the design freedom, and this also applies to *Baloo*. We honestly tried to think of sensible design concepts that are incompatible with the framework, while they would be technically possible to implement:

- *Baloo* does not support multiple hosts (*e.g.*, for redundancy purposes).

- *Baloo* cannot start primitives at different times on different nodes (*e.g.*, to save energy).
- *Baloo* cannot execute different ST primitives on different nodes during the same data slots.

Section 3.4 presented a set of features offered by the *Baloo* framework. The feature set may not be complete, but (i) it already offers a lot of options, and (ii) it can be extended in the future, if necessary. To the best of our knowledge, to date, there is no ST-based network layer protocol in the literature that is incompatible with *Baloo*. Likely, this is because what *Baloo* cannot do is either hard to do in general (*e.g.*, supporting redundant hosts) or are complex optimizations with uncertain benefits (*e.g.*, starting primitives with time offsets).

3.6.4 Lessons Learned

During this work, we have learned a few lessons that might be worth sharing.

Re-implementing protocols. Re-implementing a complete protocol (solely) based on the description from a research paper is very difficult, if not impossible. Many implementation details and design choices are omitted, for good reasons: research papers rather focus on novel concepts and ideas. Without a detailed technical documentation, a large part of the engineering is lost, and it becomes very hard to fairly compare two implementations of the same protocol.

Running protocols. Publishing code does not mean it is (re)usable. Our experience with Sleeping Beauty has been a perfect example of that: even with the code freely available and quite some experience with testbed experiments, we were not able to successfully run the protocol on Flocklab. More generally usefulness of publishing code is greatly reduced (if not voided) without proper instructions and documentation.

The point here is not to say that every research work *must* openly release code together with an extensive documentation. However, *if* one claims his or her research is providing practical solutions to concrete problems, then these solutions must be made available. When such a solution is a piece of software (*e.g.*, a network stack for low-power wireless), the (re)usability of the software is at least as important as the research paper presenting the underlying concepts.

3.7 Leveraging *Baloo*

One important motivation for working on *Baloo* was to leverage the tool for our other research projects, and *Baloo* has proved itself useful indeed. Within the time-frame of this thesis:

- Master students used *Baloo* to participate in the 2019 EWSN Dependability Competition (Section 3.5.1).
- We used *Baloo* to implement and test the Time-Triggered Wireless protocol (Chapter 5).
- We used *Baloo* to implement a generic firmware for collecting link quality data in wireless networks. We run this firmware on the FlockLab testbed multiple times per day and publish the newly collected data every month [97].

Contrarily to our original plans [96], we decided not to pursue the integration of *Baloo* within Contiki-NG. The main reason is that *Baloo* barely uses any feature from the OS itself, which is more focused on offering standardized protocol implementations for the IPv6 stack. Ultimately, this hinders the portability of *Baloo*, as one needs to port the Contiki OS first.

The development of the *Baloo* framework continues. In middle-term, we plan a new release of the framework (including improved features related to Chaos [111]), a port to the SX1262 platform [163] using FreeRTOS [4], and a bare-metal port to the nRF52840 platform [162].

3.8 Related Work

As mentioned in the introduction, the idea of middleware for Wireless Sensor Networks (WSN) has been around for more than a decade [152, 47, 190, 134]. These papers generally agree on the needs and challenges for WSN middlewares. Yet, there have been relatively few proposals to address these challenges. Recent surveys [149, 143] provide an overview of the middleware literature in the wider context of the Internet of Things, which covers all layers from local devices to cloud services. [134] reviewed the literature focusing more on WSN: proposals include for example Impala [117] which explicitly address the problem of fault tolerance in mobile networks. Programming abstractions like TinyDB [121], RUNES [53], or TinyLIME [54] have also been proposed. However, in these works, the level of abstraction is either higher or lower than the network layer.

In the past two decades, countless wireless MAC protocols have been proposed (see *e.g.*, [178, 33] for recent surveys). [94] surveyed and classified Wireless MAC protocols according to their programmability *scope* (what elements of the MAC layer are programmable) and *level* (the granularity at which the protocol logic can be programmed). The authors classify protocols as either monolithic, parametric or modular. In the context of IEEE 802.15.4 networks, modular protocols include *e.g.*, the MAC Layer Architecture (MLA) [104] and λ-MAC [144]. In contrast, *Baloo* would be classified as parametric, as it allows "parameter tuning through interfaces". Using Synchronous Transmissions (ST) interestingly changes the way network stacks can or should be designed. So far, only few network stacks using ST have been proposed [72, 95, 155, 173,

100, 175], and almost no research has been conducted to propose a flexible design framework (*e.g.*, comparable to MLA [104]) but tailored to ST. Two noteworthy exceptions are A^2 [20] and Atomic-SDN [26].

A^2 aims to facilitate the design of ST-based communication using a middleware component, called Synchrotron. However, A^2 is not a network stack, it is a *generic ST primitive*. *Baloo* and A^2 actually complement each other perfectly: *Baloo* facilitates the design of network layer protocols, but it requires to have ST primitives (*e.g.*, Glossy [70] or Chaos [111]) available, which are typically hard to implement. In turn, A^2 facilitates the design of such ST primitives. The support of A^2 within *Baloo* would be a natural next step towards a fully flexible and configurable network stack based on ST.

Atomic-SDN [26] is another work sharing this idea of a fully configurable network stack. *Baloo* and Atomic-SDN are very similar pieces of software, which have been developed in parallel. Compared to *Baloo*, Atomic-SDN is slightly more specialized: it pre-defines top-level functions (collection, configuration, reaction, and association) with a given implementation, which the user can schedule on-demand. By contrast, *Baloo* leaves the user access the various ST primitives and compose them freely to implement the desired protocol logic. Another key difference is precisely that *Baloo* lets the user pick-choose-and-combine the primitives to use, whereas Atomic-SDN is restricted to only one primitive (in the current implementation). Atomic-SDN uses a back-to-back transmissions schemes ([27, 114]) instead of traditional Glossy floods [70].

3.9 Summary

This chapter presented *Baloo*, a flexible design framework for low-power wireless network stacks based on ST. We illustrated its *Usability* and *Generality* by re-implementing three well-known network stacks: the Low-power Wireless Bus (LWB) [72], Sleeping Beauty [155], and Crystal [95], and we showed that using *Baloo* induces only limited performance overhead in terms of radio duty cycle and memory usage. *Baloo* supports the use of multiple ST primitives within the same network stack (*Versatility*) while guaranteeing that the timing requirements for ST are met (*Synchronicity*).

The key concept of *Baloo* is its clean API, based on callback functions, which let the users focus on implementing the protocol logic without worrying about low-level radio control (interrupt handling, timer settings, etc.). The API is generic and supports the different communication primitives. Through this API, multiple primitives can be used within the same network stack without additional complexity for the users.

The code of *Baloo* is openly available and is accompanied by a detailed documentation of its features and how to use them (Section 3.A). Our re-implementations of Crystal, Sleeping Beauty, and LWB are also available. We believe *Baloo* will be an important enabler for the development of future real-world applications leveraging state-of-the-art ST technology.

3.A Appendix – Artifacts and Links

3.A.1 Related Publications

Synchronous Transmissions Made Easy: Design Your Network Stack with Baloo
Romain Jacob, Jonas Bächli, Reto Da Forno, Lothar Thiele
EWSN 2019. Beijung, China (February 2019)

Paper	10.3929/ethz-b-000324254
Presentation	10.3929/ethz-b-000328814

Creating a Flexible Middleware for Low-Power Flooding Protocols
Jonas Bächli
Master Thesis. ETH Zurich (June 2018)

Thesis	10.3929/ethz-b-000270388

3.A.2 Complementary Materials

Complementary materials for this chapters are available on GitHub, together with the dissertation source files. For all links below, replace `<root>` by "github.com/romain-jacob/doctoral-thesis/blob/master"

TEX sources	`<root>`/30_Baloo/
Figures	`<root>`/30_Baloo/Figures/
Webpage	romainjacob.net/baloo
Baloo source code	
— Documentation	GitHub Wiki
— Latest release	10.5281/zenodo.3510171
— "This-version" release	10.5281/zenodo.3530632
Experiment data	
— Latest release	10.5281/zenodo.3510198
— "This-version" release	10.5281/zenodo.3510214

4

DRP: End-to-end Real-time Guarantees in Wireless Cyber-Physical Systems

In the previous chapter, we presented *Baloo*, a design framework for network stacks based on synchronous transmissions (ST). The next two chapters of this dissertation focus on leveraging ST for real-time applications. In particular, we investigate the feasibility of providing end-to-end real-time guarantees in wireless cyber-physical systems (CPS).

In CPS, the communication among the sensing, actuating, and computing elements is often subject to hard real-time constraints. In the embedded domain, real-time scheduling of dynamic applications has been extensively studied; real-time communication between wireless network interfaces is well studied as well. Yet, the design of an entire system providing end-to-end real-time guarantees between distributed applications connected through a multi-hop wireless network remains an unsolved problem.

Indeed, providing end-to-end guarantees requires to jointly consider the scheduling of distributed applications *and* the wireless communication protocol; whereas these are typically designed independently from each other, by people with different expertise. Instead, we argue for a global design considering the complete message transmission chain: peripheral buses, memory accesses, networking interfaces, and the wireless communication protocol.

In this chapter, we propose to leverage the unique properties of ST for designing reliable and efficient real-time wireless CPS. ST abstracts the complexity of a multi-hop wireless network into a virtual "wireless bus" (Chapter 1) which can then be scheduled similarly as a regular field bus. Hence, traditional scheduling techniques can be applied to both the wireless network and the distributed applications, which facilitates designs of global real-time systems.

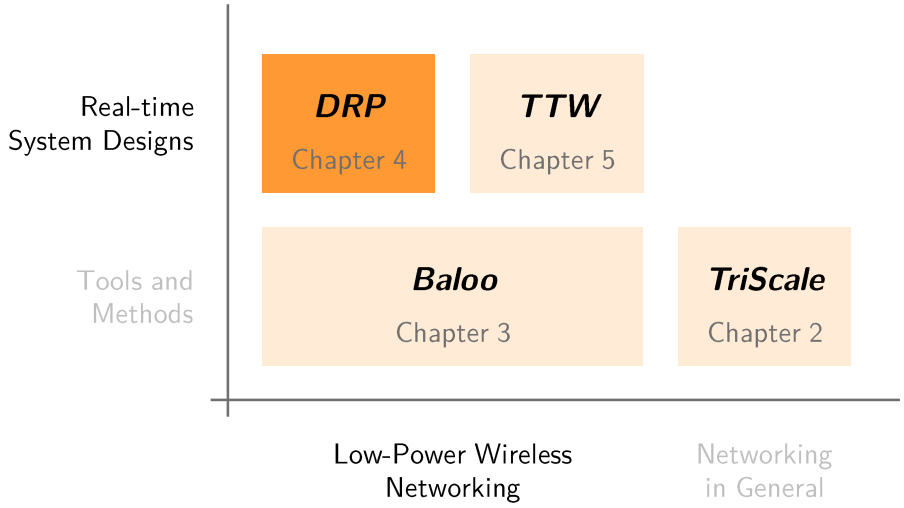


Figure 4.1 This chapter presents the Distributed Real-time Protocol (*DRP*). *DRP provides end-to-end real-time guarantees using runtime contracts, aiming to maximize the flexibility of execution between distributed tasks.*

Claim. We demonstrate for the first time that end-to-end real-time guarantees can be obtained in low-power wireless networks by leveraging the efficiency and reliability of synchronous transmissions (ST). In particular, this chapter presents the Distributed Real-time Protocol (*DRP*), a design using contracts to maximize the flexibility of execution between distributed tasks.

The material from this chapter builds upon the work from Fabian Walter [188] and Andreas Biri [36]. It relates to the following publication.

End-to-End Real-Time Guarantees in Wireless Cyber-Physical Systems
Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel, Lothar Thiele
RTSS 2016. Porto, Portugal (December 2016)

4.1 Problem Setting

Cyber-physical systems (CPS) tightly integrate components for sensing, actuating, and computing into distributed feedback loops to directly control physical processes [148]. The successful deployment of CPS technology is widely recognized as a grand challenge to solving a number of societal problems in domains ranging from healthcare to industrial automation. To reach into new areas and realize systems with unprecedented capabilities, there is a trend toward increasingly smaller and autonomously powered CPS devices that exchange data through a low-power wireless communication substrate. As many CPS applications are mission-critical and physical processes evolve as a function of time, the communication among the sensing, actuating, and computing elements is often subject to real-time requirements, for example, to guarantee stability of the feedback loops [170].

These real-time requirements are often specified from an end-to-end application perspective. For example, a control engineer may require that sensor readings taken at time t are available for computing the control law at $t + D$, where the relative deadline D is derived from the application requirements; *e.g.*, the maximum tolerable delay between a sensing and a control task, where these tasks are typically executed on physically distributed devices.

Meeting end-to-end deadlines is non-trivial because data transfers between application tasks involves multiple other tasks (*e.g.*, operating system, networking protocols) and shared resources (*e.g.*, memories, system buses, wireless medium). The entire transmission chain of the data throughout the system must be taken into account to enable end-to-end real-time guarantees.

Key Research Questions

Question 1 Can we provide end-to-end real-time guarantees between distributed applications in a wireless CPS?

Question 2 Can we do so while preserving runtime adaptability and flexibility in the timing of task executions?

The problem. Enabling real-time communication between network interfaces of sources and destinations in a low-power wireless network has been studied for more than a decade [118, 169, 87]. Today, standards such as WirelessHART [91] and ISA100.11a [93] for control applications in the process industries already exist [192], and considerable progress in real-time transmission scheduling and end-to-end delay analysis for WirelessHART networks has been made [153, 154].

Unfortunately, wireless real-time protocols such as WirelessHART [91] or Blink [208] only provide guarantees for message transmissions between *network interfaces*. These protocols do not handle the application schedules; at the source, the message release is typically assumed periodic; at the destination, nothing guarantees that the application will actually process the message in time.

Providing end-to-end guarantees between *distributed application* (**Question 1**) demands to combine a wireless real-time protocol with the rest of the system; *i.e.*, accounting for the application schedules and handling interference on shared resources.

The challenge. To support a broad spectrum of CPS applications, a solution to this problem should fulfill the following requirements.

Timeliness All messages received by the destination application meet their end-to-end deadlines.

Reliability All messages received at the wireless network interface are success-

fully delivered to their destination application (*i.e.*, no buffer overflows).

Adaptability The system adapts to dynamic changes in traffic requirements at runtime.

Composability Existing hardware and software components can be freely composed to satisfy specific application's needs, without altering the properties of the integrated parts.

Efficiency The solution scales to large system sizes and operates efficiently with regard to limited resources such as energy, wireless bandwidth, computing capacity, and memory.

A major challenge in meeting these requirements is to funnel messages in real-time through tasks that run concurrently and access shared resources. Interference on such resources can delay tasks and communication arbitrarily, therefore hampering *Timeliness*, *Reliability*, and *Composability*.

Our solution. In this chapter, we present a real-time wireless CPS that tackles interference on shared resources by defining (minimal) constraints on the application schedules. This is achieved by combining a predictable device architecture with a real-time scheduler for the entire system.

Predictable device architecture We use the Dual-Processor Platform (*DPP*) concept, presented in Introduction (Chapter 1). The *DPP* dedicates a communication processor (CP) exclusively to the real-time network protocol and executes all other tasks on an application processor (AP). The *DPP* is based on the *Bolt* interconnect [174], which decouples two processors in the time, power, and clock domains, while allowing them to asynchronously exchange messages within predictable time bounds.[1]

Thus, on each device, we decouple the communication and application tasks, which can be independently invoked in an event- or time-triggered fashion. The *DPP* concept guarantees the faithfulness of the network interface (*Reliability*), supports *Composability*, and leverages the recent trend toward ultra low-power multi-processor architectures, which can be chosen individually to match the needs of the application and the networking protocol respectively (*Efficiency*).

Real-time scheduler We design the Distributed Real-time Protocol (*DRP*), a scheduler that provably guarantees that all messages received at the application interfaces meet their end-to-end deadlines (*Timeliness*) and that message buffers along the data transfers do not overflow (*Reliability*).

To accomplish this while being adaptive to unpredictable changes (*Adaptability*), *DRP* dynamically establishes at runtime a set of contracts based on the current traffic demands in the system. A contract determines

[1] Refer to the Introduction (Section 1.4) for more details.

the mutual obligations in terms of (i) minimum service provided, and (ii) maximum demand generated between the networking protocol and an application. *DRP* contracts define time bounds that can be analyzed to ensure that end-to-end deadlines are met, while preserving flexibility in the timing of distributed task executions (**Question 2**).

The contributions of this chapter are summarized below.

- We design *DRP*, a wireless CPS system that provably provides end-to-end real-time guarantees between distributed applications. *DRP* does so by harnessing the benefits of synchronous transmissions (Section 1.3) and building upon the Blink real-time scheduler [208] and the Dual-Processor Platform architecture (Section 1.4).
- We simulate *DRP* execution to demonstrate that the provided bounds are both safe and tight.
- We implement *DRP* on embedded hardware and showcase that the protocol works as expected.
- We make our implementation of *DRP* publicly available, which includes the Blink scheduler for LWB [72].

4.2 System Model

Let $\mathcal{F}$ be the set of real-time message *flows* in the system. The message release of each flow is *sporadic with jitter*; *i.e.*, each flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$ is defined by a *source application* running on *source node* n_i^s that *releases* messages with a *minimum message interval* T_i and *jitter* J_i ($J_i < T_i$), such that the time span of n successive messages is never smaller than $(n-1)\times T_i - J_i$ for any n. Every message released at n_i^s should be delivered to the application running on *destination node* n_i^d within the same *relative end-to-end deadline* $\mathbf{D}_i$.

The system model is illustrated in Figure 4.2: a set of *nodes* $\mathcal{N}$ exchange messages over a wireless multi-hop network; messages sent from a source node to a destination node are possibly relayed by multiple other nodes. A logically global network manager, called the *host*, arbitrates access to the network. Physically, the host may be one of the nodes. The source and destination applications of a flow F_i run on physically distributed nodes n_i^s and n_i^d. Nodes can send to and receive messages from all other nodes in the system.

Problem statement. The problem is to design a wireless CPS that fulfills all the requirements presented in Section 4.1 such that, for every message of every flow $F_i \in \mathcal{F}$ released at the source node n_i^s, if it is successfully transmitted by the wireless network, then it is delivered to the destination application running on node n_i^d within the flow end-to-end deadline $\mathbf{D}_i$.

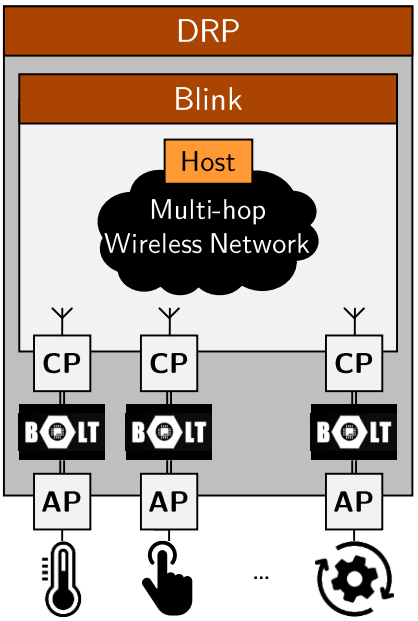


Figure 4.2 System model of the Distributed Real-time Protocol (*DRP*). *A set $\mathcal{N}$ of DPP nodes are forming a wireless CPS. The communication processors (CPs) run the Blink real-time protocol [] to exchange messages across a multi-hop wireless network. The CPs forward and receive messages from their application processors (APs) through Bolt []. DRP is a global scheduler that arbitrates end-to-end communications between the APs. On each AP, the application tasks can be scheduled freely (*e.g., *using a polling server, rate monotonic, etc.) as long as the resulting schedule satisfies the DRP contracts (Section 4.4). The host is a (logically) global network manager which arbitrates the access to the shared wireless medium;* i.e., *it runs the Blink and DRP schedulers. Physically, one of the nodes plays the role of the host.*

Application use case. Consider an acoustic wireless sensor network, such as those used to monitor permafrost in high alpine regions [193, 128]. Rock cracks are unpredictable events; however when one such event does happen, data must be collected and forwarded rapidly to a sink node for processing. For an early-warning system, it is crucial that this happens in real-time. Such an application perfectly matches our system model and motivates our problem statement.

4.3 Overview of *DRP*

This chapter presents the Distributed Real-time Protocol (*DRP*), a solution to provide end-to-end real-time guarantees between distributed applications. Before delving into details, this section provides an overview of *DRP*'s principles.

The system model of *DRP* divides the end-to-end communication between local and wireless parts (Figure 4.2):

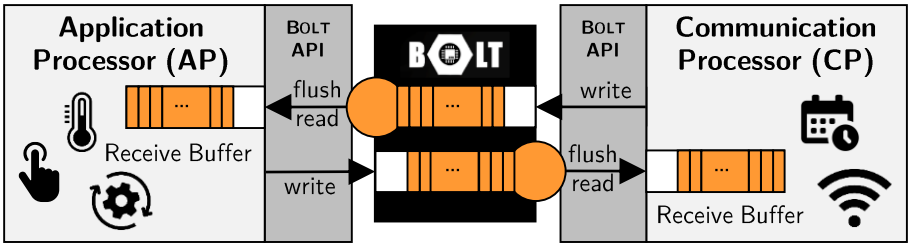


Figure 4.3 Conceptual view of the DPP, based one the *Bolt* processor interconnect. *Using functions* `write`, `read`, *and* `flush`, *the application* (AP) *and communication* (CP) *processors can asynchronously exchange messages with predictable latency. The* AP *executes application tasks (*e.g., *sensing, actuation, control, etc.) while the* CP *is dedicated to radio communication.*

$AP \leftrightarrow CP$ Applications run on dedicated application processors (APs) which are isolated from the rest of the network by their attached communication processor (CP). Local communication between APs and CPs takes place over the *Bolt* interconnect [174], which provides asynchronous message passing with bounded delays. This device architecture, called the Dual-Processor Platform (*DPP*), is illustrated in Figure 4.3 (more details in Chapter 1).

$CP \leftrightarrow CP$ The CPs exchange messages over a multi-hop wireless network using the Blink real-time protocol [208]. Blink is adaptive to dynamic changes in traffic demands, energy efficient, and delivers messages in real-time.

The *DPP* and Blink are key building blocks to fulfill the *Reliability*, *Adaptability*, *Composability*, and *Efficiency* requirements. However, two major issues remain in order to achieve *Timeliness*.

First, the communication between APs and CPs cannot be completely asynchronous: to guarantee end-to-end deadlines, both processors must look for incoming messages with some minimal rate. Second, Blink assumes a periodic release of messages at the network interfaces (*i.e.*, the CPs); since our flow model is not periodic but sporadic with jitter (Section 4.2), messages may be delayed in CPs buffer until they can be transmitted over the network.

DRP strikes a balance between *Composability* and *Efficiency*; that is, between

- decoupling the execution of APs, CPs, and Blink, and
- supporting short the end-to-end deadlines between the APs.

The idea behind *DRP* is to split the responsibility of meeting end-to-end deadlines between (i) the source node n_i^s and Blink, and (ii) the destination node n_i^d; If the source does not write too many messages, Blink guarantees every message will meet a given network deadline D, in turns, the destination commits to read its *Bolt* queue sufficiently often to meet the flow's end-to-end

deadlines $\mathbf{D}$.

DRP formalizes these "commitments" into *contracts* between the different entities. The challenge is to define, given the current network state and an end-to-end deadline $\mathbf{D}$ to satisfy, what must be (i) the network deadline D requested to Blink and (ii) the minimal reading rate at the destination node. The goal is to make these contracts minimally restrictive, such that APs, CPs, and Blink can operate as much as possible independently from each other (*Composability*).

4.4 Designing *DRP*

We now detail the three building blocks of our solution: We first describe how APs and CPs exchange messages through *Bolt* (Section 4.4.1), then we outline the operation of the Blink wireless real-time protocol (Section 4.4.2), and finally we present the detailed design of *DRP* (Section 4.4.3).

4.4.1 Bolt Processor Interconnect

Bolt [174] provides predictable asynchronous message passing between two arbitrary processors, and hence decouples the processors with respect to time, power, and clock domains. Concrete realizations of Dual-Processor Platforms (*DPP*) based on *Bolt* are depicted in Section 1.B.

Figure 4.3 shows a conceptual view of the DPP two message queues with first-in-first-out (FIFO) semantics, one for each direction, form the core of *Bolt*. *Bolt* allows for concurrent `read` and `write` operations by AP and CP on both queues.

Bolt API includes three functions (Table 4.1). The `write` function appends a message to the end of the outgoing queue, whereas `read` reads and removes the first message from the incoming queue. Calling `flush` results in a sequence of `read` operations until the incoming message queue is empty. The implementation of `flush` is peculiar. As *Bolt* allows for concurrent `read` and `write` operations, in theory, a `flush` may result in an infinite sequence of `read` operations. To prevent this, the number of `read` during a `flush` is upper-bounded by f_{max}. f_{max} is set to the number of messages that fit into one *Bolt* queue, denoted by S_{Bolt},

$$f_{max} = S_{Bolt} \tag{4.1}$$

Thus, a `flush` terminates when the incoming queue is found empty or when f_{max} messages have been read out.

By design, *Bolt* API features predictable execution times, independently of the interconnected processors [174]. We denote by C_w, C_r, and C_f the worst-case execution times (WCETs) of `write`, `read`, and `flush`.

Table 4.1 *Bolt* application programming interface (API)

Function	Description	WCET
`write`	Append a message to outgoing queue	C_w
`read`	Read and remove the first message from incoming queue	C_r
`flush`	Perform up to f_{max} `read` operations, or until incoming queue is empty	$C_f = f_{max} * C_r$

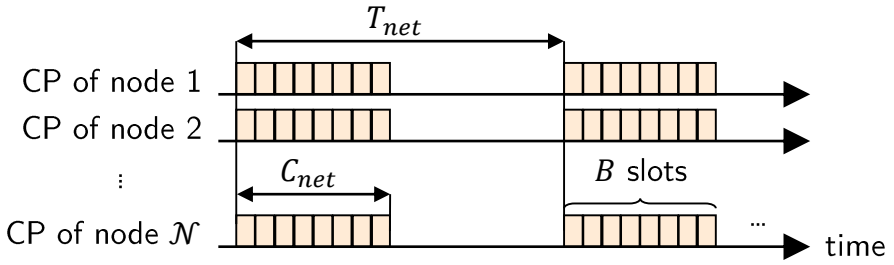


Figure 4.4 Operations in Blink are globally time-triggered. *Communication occurs in rounds of equal duration C_{net}. Each round consists of a sequence of up to B_{max} exclusive time slots, each of which serves to send one message using Glossy floods [70]. The time interval between two consecutive rounds (T_{net}) may vary. During a round, the CP of all nodes in the system participate in the flood.*

4.4.2 Blink Wireless Real-time Protocol

In Blink [208], wireless multi-hop communication is globally time-triggered and occurs in rounds of equal duration C_{net} (Figure 4.4). Each round serves to send up to B_{max} messages within exclusive time slots. In each time slot a message is sent from a given CP to all other CPs using Glossy floods, which deliver packets with a probability above 99.9 % [70].[2] The interval between the start of consecutive rounds, denoted by T_{net}, is determined by the host at runtime based on current real-time traffic demands. T_{net}^{min} and T_{net}^{max} are implementation-specific bounds on T_{net}. Blink defines T_{net} and the assignment of messages to rounds such that the number of rounds is minimized and all messages meet their *network deadline* D_i between network interfaces (*i.e.*, the CPs).

Blink communication rounds are atomic: during a round, all CPs are busy executing Blink; other tasks (*e.g.*, exchanging messages through *Bolt*) can only be executed between rounds. Furthermore, Blink assumes constrained deadlines ($D_i \leq T_i$). Thus, the network deadline D must be larger than the minimal

[2]The principle of round-based communication using Glossy floods was introduced in the Low-power Wireless Bus (LWB) [72]. The concept was later adapted in many different flavors (see the introduction of Chapter 3). Blink is a real-time scheduler for LWB.

round interval and smaller than the flow period T

$$T_{net}^{min} \leq T_{net} \leq D \leq T \tag{4.2}$$

Blink expects periodic message arrivals with a known initial phase for the first packet. We refer to this as the expected arrival pattern. For any message matching the expected arrival pattern, Blink guarantees that, if the message is successfully received at the destination CP, the message meets its relative network deadline D.

However, we must consider the complete system: (i) the message release from the APs is sporadic with jitter, and (ii) APs and CPs operate independently (*Composability*). Thus, there is a mismatch between the periodic arrival pattern assumed by Blink and the actual message arrival at the CPs.

4.4.3 *DRP*: Distributed Real-time Protocol

Blink provides real-time guarantees between the network interfaces (*i.e.*, the CPs) assuming periodic message release. *DRP* handles the mismatch between the Blink assumptions and the actual message arrival at the CPs by (i) letting the host *assume* that messages are indeed released periodically at the CPs. Blink's communication schedule is computed based on the expected arrival pattern and using an arbitrary initial phase for the flows. (ii) analyzing the maximal mismatch between the actual and expected arrival patterns.

This upper-bound represents the maximum extra-delay that a message can suffer before it is scheduled for communication by Blink. Then, the delay bounds for communication over *Bolt* and over the wireless network can be combined into a worst-case latency analysis which connects the system parameters (such as the network deadlines and flushing rate of *Bolt*) with the expected message latency. *DRP* reverts these relations to define values for the system parameters that guarantee to meet the specified end-to-end deadlines. *DRP* enforces such parameter values using contracts, which are agreed upon at runtime every time a new message flow is registered.

Contracts. *DRP* contracts are key to fulfill the *Timeliness* and *Reliability* requirements. Concretely, these contracts

- avoid overflows of message buffers (*e.g.*, the *Bolt* queues) at the source and destination nodes, thus preventing message losses;
- ensure that messages are handled "fast enough" between the network (*i.e.*, CPs) and the application (*i.e.*, APs) interfaces by the source and destination nodes, such that all messages meet their end-to-end deadlines.

To avoid overflows, *DRP* defines maximum time intervals between two `flush` operations of *Bolt* by the CPs and APs, denoted by T_f^s and T_f^d respectively. T_f^s

is statically set for all CPs in order not to constrain the achievable end-to-end deadline. Conversely, T_f^d is adjusted dynamically by the destination nodes upon registration of a new flow.

Providing end-to-end guarantees entails that *DRP* decides on the distribution of responsibilities among the source node, Blink, and the destination node of a flow F_i with regard to meeting the end-to-end deadline $\mathbf{D}_i$. To this end, *DRP* uses the *deadline ratio* $r \in (0, 1)$, a global parameter chosen at design time. The joint responsibility of the source and Blink is a function of the source flushing interval T_f^s and the flow's network deadline D_i (computed by *DRP*–Figure 4.5). They are responsible for meeting a fraction r of the end-to-end deadline

$$f(T_f^s, D_i) \ \leq \ r * \mathbf{D}_i \tag{4.3}$$

The remaining part of the end-to-end deadline defines the responsibility of the destination, which is a function of its flushing interval T_f^d

$$g(T_f^d) \ \leq \ (1 - r) * \mathbf{D}_i \tag{4.4}$$

In Section 4.5, we derive concrete expressions for the functions f and g, and we specify how *DRP* computes D_i and T_f^d. In Section 4.7 we illustrate how the choice of the deadline ratio r influences the achievable bandwidth and end-to-end guarantees of our wireless CPs system.

For each newly admitted flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i) \in \mathcal{F}$, *DRP* dynamically establishes two contracts.

- **Source ↔ Blink** F_i's source application, which runs on AP_s at node n_i^s, agrees to write no more messages than specified by the minimum message interval T_i and the jitter J_i. The attached CP_s prevents overflows of *Bolt* and its local message buffer. In turn, Blink agrees to serve flow F_i such that any message matching the expected arrival of F_i meets its network deadline D_i.
- **Blink ↔ Destination** Blink agrees to deliver no more messages than specified by T_i. In turn, CP_d and AP_d agree to read out all delivered messages such that overflows of *Bolt* and CP_d's local buffer are prevented and all messages meet F_i's end-to-end deadline $\mathbf{D}_i$.

For any flow, if both contracts are fulfilled, all messages that are successfully delivered by Blink will meet their end-to-end deadline. In practice, the contracts fulfillment is guaranteed by a set of *admission tests*, which are performed in sequence upon registration of a new flow, as described next.

Flow registration. Figure 4.5 shows the full procedure for registering a new flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$ in *DRP*. The flow's source application running on AP_s first computes the network deadline D_i (Section 4.5) before it writes the request to the attached CP_s through *Bolt*. CP_s uses its admission test to check whether it could still prevent overflows of *Bolt* and its local memory

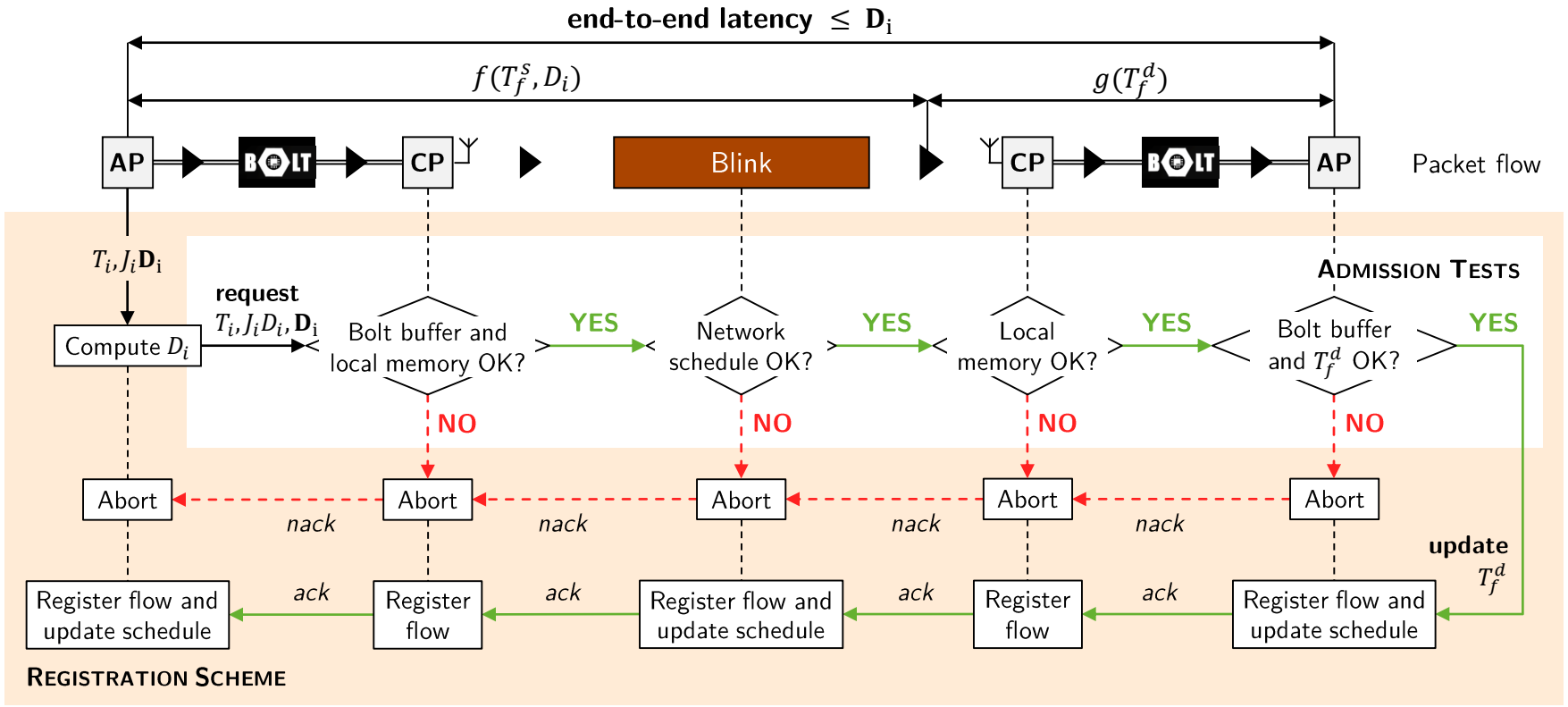


Figure 4.5 Steps and components involved when registering a new flow in *DRP*. *Given a request for a new flow* $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$*, the source application running on* AP_s *at node* n_i^s *computes the flow's network deadline* D_i*. Then, all components check one after the other using specific admission tests whether they can admit the new flow. DRP registers a new flow only if all admission tests succeed, which eventually triggers changes in the runtime operation (the schedule) of Blink as well as of the source and destination application processors* AP_s *and* AP_d*.*

if F_i were present. If so, CP_s forwards the request to the host, which checks the schedulability using Blink's admission test [208]. If Blink admits the flow, the destination node's CP_d and AP_d check whether they can prevent overflows of CP_d's local memory and *Bolt*, respectively. Moreover, AP_d re-computes its required flushing interval T_f^d and checks using mainstream schedulability analysis [45] whether it can support this new load (in addition to the load incurred by other tasks running on AP_d). *DRP* registers a flow only if all admission tests succeed, which then triggers changes in the runtime operation of AP_s, Blink, and AP_d.

Flow requests and acknowledgments are sent via dedicated control flows, which are registered by default at bootstrapping for each node in the network.

***DRP* procedure.** Figure 4.6 summarizes all the inputs and outputs of *DRP*. Hardware parameters (related to *Bolt*) and design parameters (*i.e.*, the length of a communication round C_{net}, the deadline ratio r, and the number of slots per round B_{max}) are constants known at compile time. The application's real-time communication requirements may change at runtime as new flows are requested and existing flows are removed. *DRP* determines T_f^s statically, while all other outputs are dynamically computed whenever the set of flows changes, according to the procedure illustrated in Figure 4.5.

4.5 Concrete Realization of *DRP*

This section discusses how to concretely implement *DRP*'s concepts. In particular, one needs to define (i) the fixed flushing interval T_f^s of the CPs (Section 4.5.1), and (ii) how to dynamically compute the network deadline D_i of a flow F_i and the flushing interval T_f^d of each AP (Section 4.5.2).

Then, a worst-case buffer analysis (Section 4.5.3) will allow to formulate admission tests (Section 4.5.4), one for APs and one for CPs. The success of all admission tests guarantees that both contracts **Source** $\leftrightarrow$ **Blink** and **Blink** $\leftrightarrow$ **Destination** can be satisfied by *DRP*.

4.5.1 Setting CPs' Flushing Interval

To guarantee that all CPs fulfill their share of the contracts (*i.e.*, prevent buffer overflows), we conceive a time-triggered approach to schedule all tasks of CPs. It consists of (*i*) setting the flushing interval T_f^s of all CPs to the same constant value, and (*ii*) letting the round interval T_{net} be a multiple of T_f^s. As discussed in Section 4.4, T_f^s should not to constrain the achievable deadline: thus, we aim to set it as short as possible. CPs have three tasks to perform

- flushing *Bolt* before each communication round,

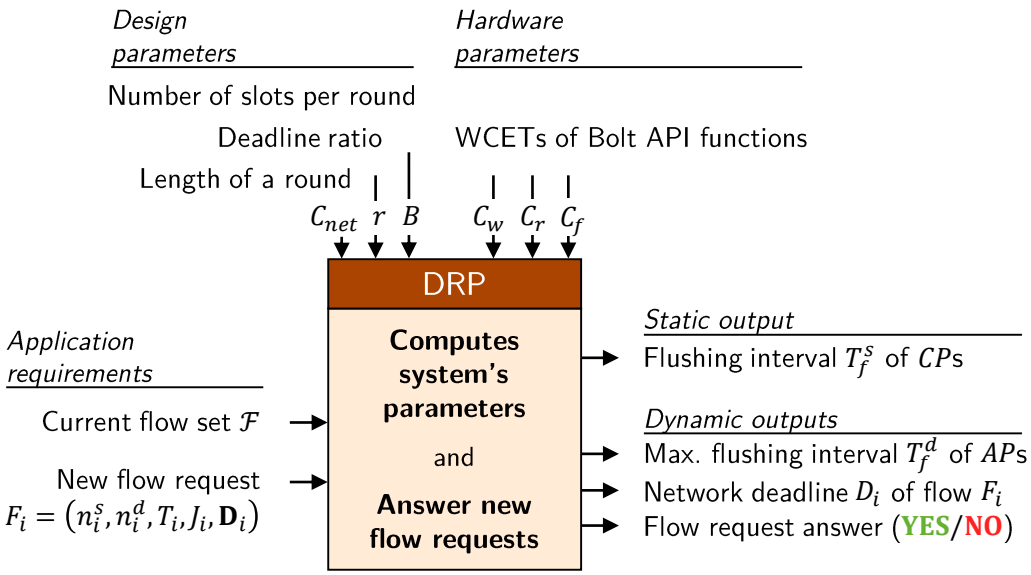


Figure 4.6 Inputs and outputs of *DRP*. *Hardware and design parameters are fixed at design time, while the application requirements may change at runtime. DRP statically computes the flushing interval of CPs; all other outputs are dynamically computed whenever the flow set $\mathcal{F}$ changes.*

- participating in the communication during the rounds,
- writing all received messages into *Bolt* after the rounds.

Performing those tasks altogether takes $C_{CP} + C_{net}$ time units, where $C_{CP} = C_f + B_{max} * C_w$, and B_{max} denotes the number of time slots in one round. Hence, $C_{CP} + C_{net}$ is the smallest admissible round interval (otherwise CPs' task set is not schedulable). Thus we set for all CPs in the system,

$$T_f^s = C_{CP} + C_{net} \tag{4.5}$$

and we let the round interval be a multiple of T_f^s. In other words, for $k \in \mathbb{N}$, $k > 0$,

$$T_{net} = k * T_f^s \tag{4.6}$$

For a given C_{net}, a larger k entails less available bandwidth but also lower energy consumption. Blink is designed to dynamically adjust k to match the bandwidth requirements and save energy [208].

4.5.2 Computing Network Deadlines & APs' Flushing Interval

Having fixed CPs' flushing interval, we now turn to the problem of dynamically computing the network deadline D_i of flow F_i and the flushing interval T_f^d of F_i's destination AP_d, such that the end-to-end deadline $\mathbf{D}_i$ is met. To this end, we need to define expressions for the functions f and g (introduced in

Section 4.4), and derive values for D_i and T_f^d such that equations (4.3) and (4.4) are satisfied.

Theorem 4.1. *For any flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$, and given the duration of communication rounds C_{net}, functions f and g are upper-bounded as follows*

$$f(T_f^s, D_i) \leq T_i + D_i + \overline{J_i} + \delta_f^{const} \tag{4.7}$$

$$g(T_f^d) \leq T_f^d(n_i^d) + \delta_g^{const} \tag{4.8}$$

where δ_f^{const} and δ_f^{const} are constant delays that depend on the WCETs of the Bolt API functions, on the maximum number of messages B_{max} that can be served by Blink in one round, and on the fixed flushing interval T_f^s of CPs,

$$\delta_f^{const} = C_w + C_f + T_f^s \tag{4.9}$$

$$\delta_g^{const} = B_{max} * C_w - (B_{max} - 1) * C_r + C_f \tag{4.10}$$

$$\overline{J_i} = \left\lfloor (J_i + C_f - C_r)/T_f^s \right\rfloor \cdot T_f^s \tag{4.11}$$

Proof. Function f is the time between when a message is written into *Bolt* by the source AP_s and when the communication round in which the message is sent by Blink ends (*i.e.*, when the message is available at the destination CP_d). This is the sum of two delays: δ_{source}, the time until the message is available for communication at the source CP_s; and $\delta_{network}$, the time until the message is shipped over the network to CP_d.

Similarly, function g is the time between when a packet is available at the destination CP_d and the end of the `flush` operation that reads the message out of *Bolt* at the destination AP_d (*i.e.*, when the message can be processed by the destination application). We refer to this delay as δ_{dest}.

Hence, the expressions for functions f and g in (4.7) and (4.8) directly follow from the delays expression derived in Lemmas 4.4, 4.5, and 4.6 (Section 4.B). ■

We use Theorem 4.1 to express conditions on D_i and T_f^d such that (4.3) and (4.4) are satisfied. In particular, it is sufficient that for any flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i) \in \mathcal{F}$

$$\begin{aligned} T_i + D_i + \overline{J_i} &\leq r * \mathbf{D}_i - \delta_f^{const} \\ \Rightarrow \quad T_i &\leq r * \mathbf{D}_i - \delta_f^{const} - D_i - \overline{J_i} \end{aligned} \tag{4.12}$$

and

$$T_f^d(n_i^d) \leq (1 - r) * \mathbf{D}_i - \delta_g^{const} \tag{4.13}$$

As low-power wireless networks typically feature limited bandwidth,[3] it makes sense to choose the network deadline D_i as large as possible in order to

[3] Low-power radio bit rates are often limited to 256 kbps; the latest version of Bluetooth (Bluetooth 5) supports up to 2 Mbit/s.

increase the schedulability of flows in the network. However, Blink only supports constrained deadlines ($D \leq T$) and deadlines must be multiples of the round length ($D = 0 \mod T_{net}$). Furthermore, a network deadline cannot be smaller than T_{net}^{min} (see Section 4.4.2). Hence, for any flow F_i, it must hold that

$$T_{net}^{min} \leq D_i \leq T_i \tag{4.14}$$

$$D_i = 0 \mod T_{net} \tag{4.15}$$

Finally, to satisfy all contracts in the system, (4.12), (4.13) and (4.14) must hold for all flows $F_i \in \mathcal{F}$. Hence, the values for D_i and T_f^d computed dynamically at runtime must satisfy for any flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i) \in \mathcal{F}$ and any $n \in \mathcal{N}$

$$D_i = \min\left(T_i\ , r * \mathbf{D}_i - \delta_f^{const} - T_i - \overline{J_i}\right) - (D_i \mod T_{net}) \tag{4.16}$$

$$T_f^d(n) \leq \min_{F_j \in \mathcal{F}, n = n_j^d} \left((1-r) * \mathbf{D}_j - \delta_g^{const}\right) \tag{4.17}$$

If using (4.16) leads to a violation of the constraint in (4.14) or if T_f^d results in a load that AP at node n cannot handle, *DRP* rejects the flow since the two contracts cannot be guaranteed.

4.5.3 Worst-case Buffer Analysis

Satisfying all contracts entails preventing overflows of message buffers in the system. Specifically, as shown in Figure 4.5,

- APs are responsible for ensuring that the incoming *Bolt* queues do not overflow, and
- CPs are responsible for ensuring that their local message buffers and the outgoing *Bolt* queues do not overflow.

To formulate the admission tests for APs and CPs, we first need the worst-case buffer sizes (*i.e.*, maximum number of messages in a buffer) induced by a given flow set $\mathcal{F}$. For ease of exposition, we make the following hypothesis.

Hypothesis 1. For a given flow set $\mathcal{F}$, an AP (resp. CP) never writes more messages into *Bolt* than can be flushed by CP (resp. AP) in one `flush` operation in the time span between two `flush`.

This hypothesis implies that the *Bolt* queues are always empty at the end of a `flush` operation. We prove at the end of this section that our admission tests effectively guarantee that Hypothesis 1 is always verified.

Lemma 4.1. *Given a flow set $\mathcal{F}$, the buffer size of the outgoing Bolt queue of node $n \in \mathcal{N}$, $B_{Bolt,out}(n)$, is upper-bounded,*

$$B_{Bolt,out}(n) \leq \sum_{F_i \in \mathcal{F},\, n = n_i^s} \left\lceil \frac{T_f^s + C_w + C_r + J_i}{T_i} \right\rceil \tag{4.18}$$

Proof. According to the **Source** $\leftrightarrow$ **Blink** contract, AP_s at node n does not write more than one message every T_i with jitter J_i into the outgoing *Bolt* queue. Based on Hypothesis 1, the buffer size is bounded by the number of messages that can be written by AP_s during the maximum time a message can stay inside the queue, which is $\Delta = T_f^s + C_w + C_r$ (see Figure 4.13). The maximum number of messages that can be written by AP_s within any time interval Δ is $\lceil (\Delta + J_i)/T_i \rceil$ for each flow F_i sourced by n. ■

The worst-case buffer size of a CP depends on (i) the maximum time a message can stay in CP's local memory awaiting to be served by Blink, and (ii) the number of messages that can be sent within one round to a node.

Lemma 4.2. *Given a flow set $\mathcal{F}$, the buffer size of CP's internal memory of node $n \in \mathcal{N}$, $B_{CP}(n)$, is upper-bounded,*

$$B_{CP}(n) \leq \sum_{\substack{F_i \in \mathcal{F}, \\ n = n_i^s}} 1 + \left\lceil \frac{D_i + \overline{J_i} + C_f}{T_i} \right\rceil + \sum_{\substack{F_i \in \mathcal{F}, \\ n = n_i^d}} 1 \tag{4.19}$$

Proof. On the source side, we make the conservative assumption that all messages read out during a `flush` occupy memory in CP_s from the beginning of the `flush`. Hence, the maximum waiting time in CP_s for a message until it is served by Blink is $\delta_{network} + C_f$ (see Lemma 4.5 – Section 4.B). The number of messages in CP_s due to the source is upper-bounded by the maximum number of messages AP_s can write during this time interval, given by $\lceil \delta_{network} + C_f)/T_i \rceil$. Using Lemma 4.5, this is at most $1 + \left\lceil (D_i + \overline{J_i} + C_f)/T_i \right\rceil$ per outgoing flow.

One the destination side, during a round, CP_d may receive several messages, which it immediately writes into *Bolt* after the round. However, Blink expects one packet every T_i from each flow, which it serves within D_i. As $D_i \leq T_i$, Blink never schedules more than one packet per round for each flow. Thus, the maximum number of messages in CP_d due to the destination is 1 packet per incoming flow. ■

Lemma 4.3. *Given a flow set $\mathcal{F}$, the buffer size of the incoming Bolt queue of node $n \in \mathcal{N}$, $B_{\textit{Bolt,in}}(n)$, is upper-bounded,*

$$B_{\textit{Bolt,in}}(n) \;\leq\; \sum_{\substack{F_i \in \mathcal{F}, \\ n = n_i^d}} \left\lceil \frac{T_f^d(n) + C_w + C_r + D_i}{T_i} \right\rceil \tag{4.20}$$

Proof. As specified in the **Source** $\leftrightarrow$ **Blink** contract, Blink delivers packets from any flow F_i before the network deadline D_i (Section 4.4). Therefore, Blink delivers at most one packet every T_i time units, with a jitter equal to D_i, which are written into *Bolt* immediately after the round.

Based on Hypothesis 1, the buffer constraint of the incoming *Bolt* queue is bounded by the number of packets that can be written by CP_d during the maximum elapsed time before a packet is read out by AP_d. As in the proof of Lemma 4.1, there are at most $\left\lceil (T_f^d(n) + C_w + C_r + D_i)/T_i \right\rceil$ such messages from each flow F_i that has node n as destination. ■

4.5.4 Admission Tests

We now combine the above results and formulate the admission tests for CPs and APs, which form the cornerstone of *DRP*'s registration mechanism described in Section 4.4. We further show that the computation complexity of the admission tests is not only small but *constant*, and hence supports the requirements of *Adaptability* and *Efficiency*.

Let F_j be the flow for which a request has been issued, and $\mathcal{F}_{new} = \mathcal{F} \cup \{F_j\}$. The CP of node n is responsible for preventing overflows of its local memory (of size S_{CP}) and of the outgoing *Bolt* queue of node n (of size S_{Bolt}).

Theorem 4.2 (Admission Test of CP)**.** *If*

$$S_{Bolt} \geq \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^s}} \left\lceil \frac{T_f^s + C_w + C_r + J_i}{T_i} \right\rceil$$

$$\text{and} \qquad S_{CP} \geq \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^s}} 1 + \left\lceil \frac{D_i + \overline{J_i} + C_f}{T_i} \right\rceil + \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^d}} 1$$

then the requested flow F_j can be safely admitted by CP.

Proof. Immediate from Lemmas 4.1 and 4.2. ■

The AP of node n is responsible for preventing overflows of the incoming *Bolt* queue (of size S_{Bolt}) and for guaranteeing its share of the end-to-end deadline.

Theorem 4.3 (Admission Test of AP)**.** *If there exists $T_f^d(n)$ such that*

$$T_f^d(n) \leq \min_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^d}} \left((1 - r) * \mathbf{D}_i - \delta_g^{const} \right)$$

$$\text{and} \qquad S_{Bolt} \geq \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^d}} \left\lceil \frac{T_f^d(n) + C_w + C_r + D_i}{T_i} \right\rceil$$

then the requested flow F_j can be safely admitted by AP.

Proof. Immediate from Lemma 4.3 and equation (4.17). ■

Finally, we verify that Hypothesis 1 holds, showing the validity of our buffer analysis. From (4.1) we have $f_{max} = S_{Bolt}$. Thus, by performing the admission tests at runtime, it follows from Theorems 4.2 and 4.3 and Lemmas 4.1 and 4.3 that f_{max} is always bigger than the filling level of any *Bolt* queue, which entails Hypothesis 1 is true.

4.6 Implementating *DRP*

This dissertation aims to provide concrete solutions for wireless CPS, not only theoretical concepts. It is therefore important to implement the concepts and evaluate their performance when running on real hardware. This section presents some important design choices we made for our implementation of *DRP*. The performance evaluation is presented in Section 4.7.

DRP extends Blink [208], which is a real-time scheduler for LWB [72]. Thus, we start our implementation of *DRP* from an existing implementation of LWB [71]. There are two main functionality to implement for running *DRP*:

- compute Blink schedules, and
- perform *DRP* admission tests.

DRP leverages the *DPP* design by having all the computations performed by the AP while the CP only runs LWB. From an implementation stand-point, the challenge is that the AP does computations but the CP needs the results (*e.g.*, the LWB schedules); the communication over *Bolt* between AP and CP must happen in a way that does not interrupt or delay LWB operations. Moreover, the memory and computational requirements of the protocol must be compatible with the (typically limited) capacity of embedded hardware.

Task distribution. On all nodes, the computations related to *DRP* admission tests are performed by the AP, including CP's buffer checks (Theorem 4.2). These tests only require minimal state keeping, which can be easily delegated to the AP. This has two benefits: (i) it limits the modification to the CP firmware (*i.e.*, LWB) and (ii) it improves performance (the admission test is more efficiently computed by the AP than by the CP).

Round structure. Our implementation maintains the original LWB round structure, only removing the contention slot (Figure 4.7). This slot becomes redundant in *DRP* since each node bootstraps with registered control flows, which are used for further *DRP* requests and flow registrations (Section 4.4).

Schedule fetch. The host CP fetches the schedule for the LWB round $i+1$ at the end of round i (Figure 4.7). CP requests the schedule to AP, which maintains the schedule ready and writes it to *Bolt* whenever the request comes.

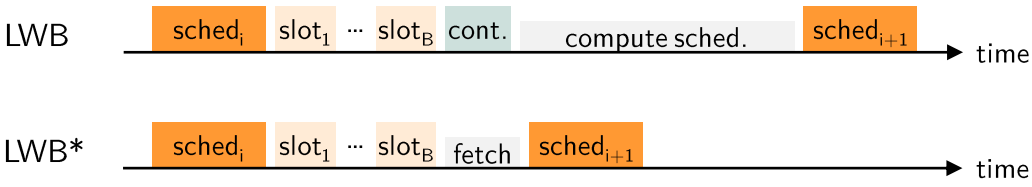


Figure 4.7 The original LWB (Top) and the modified round used in *DRP* (Bottom). *DRP removes the contention slots, made redundant by the presence of dedicated control flows. Furthermore, the time span before sending the next round's schedule can be significantly reduced: indeed in DRP, the host CP can quickly fetch the schedule from the AP, which has done the computation in parallel.*

Communication model. The communication between AP and CP over *Bolt* can be based on polling or interrupts. When using polling, processors asynchronously check the status of their incoming *Bolt* queue to check whether a message is present. Conversely, an interrupt can be generated at the receiving processor whenever a *Bolt* `write` operation is completed.

AP to CP When AP writes to *Bolt*, we trigger an interrupt on the CP and read out the message immediately. Since *DRP* flow model is sporadic and new packets are expected infrequently (Section 4.2), using interrupt is fast, energy efficient, and avoids building up the *Bolt* queue.

However, during LWB rounds, it is paramount that CP operations are not delayed or disturbed. Thus, during the rounds, interrupts are disabled and (potential) new messages written by AP are read out after the round.

CP to AP We assume that the AP on the host node is dedicated to its role of host (*i.e.*, there are no other application tasks running of that AP). Thus, we use interrupts, which allows for fast reaction times from AP to CP's requests (*e.g.*, when fetching the next round's schedule). When CP receives a *DRP* request, the AP immediately reads out the request, stores it in a request queue (Figure 4.9), then processes incoming requests whenever possible (Figure 4.8)

For the other nodes, the decision of using polling- or interrupt-based communication depends on the timing requirements of the application. Our implementation supports both modes. For simplicity, we use interrupts in our evaluation (Section 4.7).

Host AP state-machine. The host AP is responsible to compute the Blink schedules and to perform the admission tests for incoming *DRP* requests. As processing *DRP* requests takes a variable (and possibly long) time, priority is given to the schedule computation: as soon as the schedule for the round i has been fetched by the CP, AP computes the schedule of round $i + 1$.

Once the schedule is ready, AP continues with the processing of *DRP* requests. After each processed request, if the flow is admitted, AP recomputes a schedule

for round $i+1$ taking the new flow into account. The procedure repeats until the schedule is fetched by CP or when all requests have been processed. The host AP state-machine is illustrated in Figure 4.8.

Alternating buffers. As described above, the host CP fetches the next round schedule from its AP, which sends a valid schedule immediately. However, this conflicts with the admission of new flow requests: the AP may be processing a request (including the computation of a new schedule) when CP requests the schedule.

We handle this situation using an alternating buffer for the next round schedule (Figure 4.9). One buffer is used to store a valid schedule for the next round, which the AP computes first. After each *DRP* request is processed, the newly computed schedule is written into the other buffer. The process repeats until all requests are processed.

Hence, even if CP fetches while AP is writing a schedule in one buffer, the other buffer still contains a valid schedule, which can be immediately sent to CP over *Bolt*. This guarantees a fast transmission of the schedule from AP to CP while avoiding memory corruption on the AP.

Remark 4. Alternative design choices and their respective benefits are further discussed in Andreas Biri's Master Thesis [36].

4.7 Performance Evaluation

After detailing the design (Section 4.4) and implementation (Section 4.6) of *DRP*, we now evaluate the performance of the system. We consider three different performance aspects.

- First, we derive the theoretical optimal performances achievable by *DRP*, based on the system model (Section 4.7.1).
- Then, we first use simulation to demonstrate the tightness of the worst-case analysis underlying *DRP*'s design: we show that end-to-end message latency reaches up to 97% of the analytic bounds (Section 4.7.2).
- Finally, we showcase that our *DRP* implementation performs as expected: all messages successfully transmitted through the wireless network do meet their end-to-end deadline. Furthermore, we illustrate that on a "real" network, messages typically experience latency much shorter than their end-to-end deadline (Section 4.7.3).

Remark 5. The *TriScale* framework, introduced in Chapter 2, would be beneficial for the design and analysis of *DRP*'s performance evaluation. However, the evaluation described below is anterior to the work we have done on *TriScale*, and thus does not use the framework.

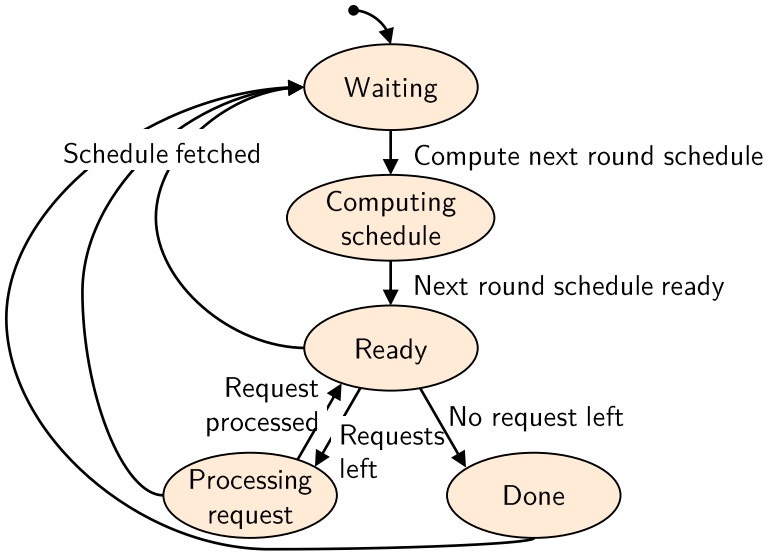


Figure 4.8 State-machine of the host AP. *AP first computes a valid schedule for the next round. Then, the AP processes queued DRP requests, one at a time, and recomputes a (new) schedule accounting for the newly admitted flow.*

4.7.1 Performance Model

The admission tests for AP and CP (which ensure that all contracts are satisfied after the admission of a new flow) critically depend on global parameters: the duration of a round C_{net} and the deadline ratio r.

In this section, we analyze the influence of these parameters on the achievable performance of *DRP* in terms of responsiveness (*i.e.*, the minimal admissible end-to-end deadline) and bandwidth.

Responsiveness: Minimal Admissible End-to-end Deadline. Let us assume that the duration of communication rounds C_{net} is given. *DRP* handles messages between application interfaces (*i.e.*, the APs) and constrains the destination AP_d to flush *Bolt* (at least) every T_f^d. Naturally, there exists a lower bound on the admissible T_f^d; let us refer to this bound as $T_{f,min}^d$. Given these parameters, we are interested in the minimal admissible end-to-end deadline $\mathbf{D}_{min}$, or in other words, the maximal responsiveness of the protocol.

From the previous remark on T_f^d and eq. (4.17) it follows

$$\begin{aligned} T_{f,min}^d \;&\leq\; T_f^d \leq \left((1-r) * \mathbf{D} - \delta_g^{const}\right) \\ \Rightarrow \quad \mathbf{D} \;&\geq\; \frac{T_{f,min}^d + \delta_g^{const}}{(1-r)} \end{aligned} \tag{4.21}$$

From (4.12) we also have

$$T + D + \overline{J} \;\leq\; r * \mathbf{D} - \delta_f^{const}$$

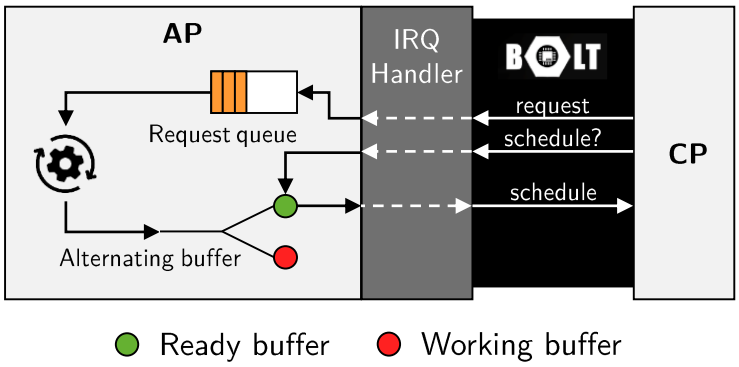


Figure 4.9 Request queue and alternating buffer on the host AP. *Incoming messages from CP trigger an interrupt and are handled immediately. DRP requests are put in a dedicated queue for later processing (Figure 4.8). When CP writes a schedule request, the interrupt handler fetches the most recent schedule from the "Ready buffer" and write it to Bolt.*

$$\Rightarrow \quad \mathbf{D} \geq \frac{T + D + \overline{J} + \delta_f^{const}}{r}$$

We look for the minimal expression of the right-hand side term. (4.14) : $T_{net}^{min} \leq D_i \leq T_i$ yields $T_{min} = D_{min} = T_{net}^{min}$. Moreover, combining (4.5) and (4.6) entails $T_{net}^{min} = T_f^s = C_{net} + C_{CP}$. Hence T_{min} is fixed given C_{net}. Finally, in the best case, there is no (or small) jitter (*i.e.*, $\overline{J} = 0$), and we obtain

$$\Rightarrow \quad \mathbf{D} \geq \frac{2T_{min} + \delta_f^{const}}{r} \tag{4.22}$$

(4.21) and (4.22) define two lower bounds on the minimal admissible end-to-end deadline $\mathbf{D}_{min}$ induced by the contracts. Combining them, it follows that

$$\mathbf{D}_{min} = \min_r \left(\frac{T_{f,min}^d + \delta_g^{const}}{(1-r)}, \frac{2T_{min} + \delta_f^{const}}{r} \right)$$

The minimal value $\mathbf{D}_{min}$ is reached for

$$r_{opt} = (2T_{min} + \delta_f^{const})/(T_{f,min}^d + \delta_g^{const} + 2T_{min} + \delta_f^{const}) \tag{4.23}$$

and it yields

$$\mathbf{D}_{min} = T_{f,min}^d + 2T_{min} + \delta_f^{const} + \delta_g^{const} \tag{4.24}$$

Using the parameters in Table 4.2 from real-world prototypes, if $C_{net} = 1\,\text{s}$ and $T_{f,min}^d = 0.1\,\text{s}$, the minimal end-to-end deadline that can be supported is $\mathbf{D}_{min} = 3.43\,\text{s}$, with $r = r_{opt} = 0.95$, and the minimum message interval $T = T_{min} = 1.074\,\text{s}$. This case is illustrated in Figure 4.10.

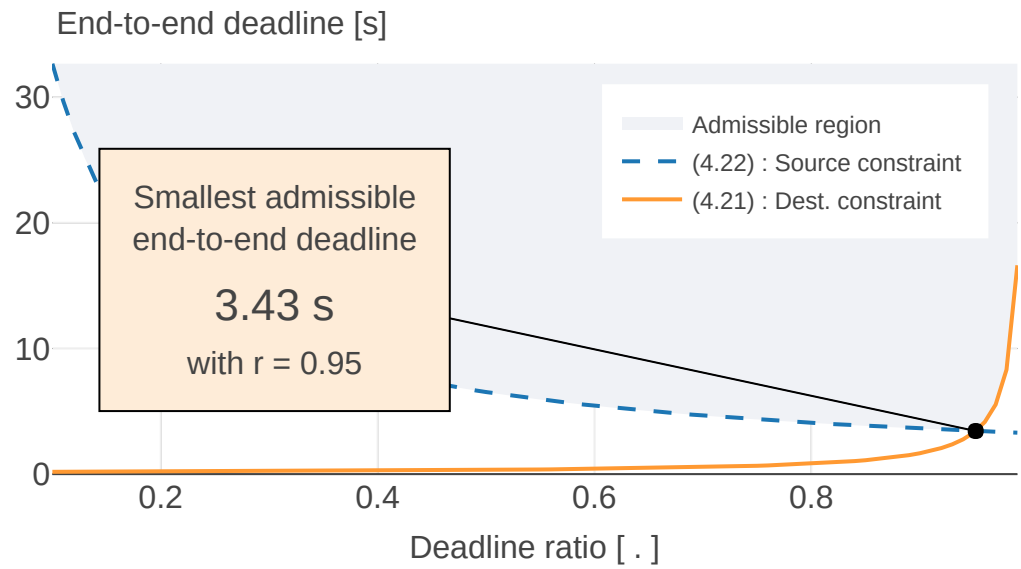


Figure 4.10 The smallest admissible end-to-end deadline for $C_{net} = 1\,\text{s}$ and $T^d_{f,min} = 0.1\,\text{s}$ is $\mathbf{D}_{min} = 3.43\,\text{s}$. *Equations* (4.21) *and* (4.22) *each define a feasible region for* $(r, \mathbf{D})$ *tuples. The intersection defines the admissible region.*

Bandwidth: Maximal Duration of Communication Rounds. Conversely, let us now assume that the minimal end-to-end deadline to be supported is given by $\mathbf{D}$, and consider the same assumption on T^d_f. The maximal bandwidth achievable by Blink is B_{max}/T^{min}_{net} pkt/s/ s. The round length C_{net} is a linear function of the number of packets per round B_{max} (*i.e.*, a constant time per packet plus some overhead), and (4.6) : $T^{min}_{net} = C_{net} + C_{CP}$. Hence, the maximal bandwidth actually grows with C_{net}. Thus, we now investigate the maximal admissible duration of communication rounds C_{net} that yields the maximum available network bandwidth.

From (4.12) we have $T + D + \overline{J} \leq r * \mathbf{D} - \delta^{const}_f$, and, as previously, $T^{min}_{net} = C_{net} + C_{CP} \leq D \leq T$. We get

$$C_{net} \leq \frac{1}{2}(r * \mathbf{D} - \delta^{const}_f) - C_{CP} \tag{4.25}$$

From (4.21), given $\mathbf{D}$ and $T^d_{f,min}$, the maximal admissible value for r is $r_{max} = 1 - (T^d_{f,min} + \delta^{const}_g)/\mathbf{D}$, and finally

$$C_{net} \leq \frac{1}{2}\left(\mathbf{D} - \delta^{const}_f - \delta^{const}_g - T^d_{f,min}\right) - C_{CP} \tag{4.26}$$

Using the parameters from Table 4.2, if we need to satisfy end-to-end deadlines of $\mathbf{D} = 10\,\text{s}$ and $T^d_{f,min} = 3\,\text{s}$, the maximal round length that can be supported is $C_{net} = 2.82\,\text{s}$, with $r = r_{max} = 0.69$, and the minimum message interval

Table 4.2 Simulation parameters. *The Bolt API execution times are formally proven bounds for the given hardware [174]. The number of slots per round B_{max} is defined as the number of slots that "fit" into a round of length C_{net} given the packet size.*

	Parameter	Symbol	Value
Bolt	WCET of `write`	C_w	$116\,\mu$s
	WCET of `read`	C_r	$112\,\mu$s
	WCET of `flush`	C_f	$684\,$ms
Blink	Round length	C_{net}	$1\,$s
	Packet size	L	32 bytes
	Max number of slots in one round	B_{max}	46
DRP	Number of nodes	.	20
	Deadline ratio	r	0.5
	Flushing interval of CP	T_f^s	$1.074\,$s

$T = C_{net} + C_{CP} = 2.89\,$s. That upper-bound also yields the maximal achievable network bandwidth.

Effect of Deadline Ratio on System Performance.

We presented earlier that given C_{net} and $T^d_{f,min}$, there is an optimal value for r that minimizes the admissible end-to-end deadline $\mathbf{D}$. If one tolerates "larger" deadlines, r can be increased to allow for a bigger round length C_{net} (see (4.25)), which increases the maximal network bandwidth.

However, (4.17) yields $T^d_f \leq (1-r) * \mathbf{D} - \delta_g^{const}$. Hence, the bigger r is the smaller T^d_f must be, which may results in more flows rejected by the destination application. On the contrary, if r is set to its minimal value $r_{min} = (2 * T_{min} + \delta_g^{const}) / \mathbf{D}$ (obtained from eq. (4.22)), it yields $T = D = T_{min} = 1.074\,$s and $\overline{J} = 0\,$s. In other words, the maximal admissible jitter (obtained from (4.11)) is $J < T^s_f + C_r - C_f \approx 0.390\,$s.

How to set the parameters for *DRP* depends on the application. For instance, if one consider an acoustic sensing scenario, responsiveness is usually quite critical, and the sensors (*i.e.*, the APs) should spend most of their time on sensing, not being busy with flushing *Bolt*. Thus, we want to support a rather small $\mathbf{D}_{min}$ while having a strong constraint on $T^d_{f,min}$. This will come at the cost of a "small" network bandwidth.

4.7.2 Simulated Worst-Case Performance

In Section 4.7.1, we derived the optimal performance achievable according to our *DRP*'s model. However, this model is based on a worst-case analysis of message latency throughout the system. Because such an analysis is inherently

pessimistic, it is important to estimate how pessimist the analysis is. In other words, how tight are the latency bounds given by the model? In this section, we investigate this question using a discrete event simulation.

Procedure. We simulate the run-time behavior of *DRP* using the values and parameters from our implementations (Table 4.2). The simulation framework tracks the latency of each individual message through the entire system, *i.e.*, all APs, CPs, *Bolt* and the wireless communication network. Concretely, the simulation is implemented using Matlab scripts (openly available – Section 4.A).

Blink computes the round schedules assuming that the first message of each flow is available for communication at $t = 0\,\text{s}$. The actual epoch at which the APs write the first packet of each flow is randomized between $0\,\text{s}$ and the flow's minimal message interval T; subsequent packets are sent with period T. The random seed is fixed for reproduciblility.

Scenario. Node 1 acts as the sink and communicates with all other nodes in the network. As described in Section 4.4, *DRP* is initialized with a set of control flows $\mathcal{F}_{control}$, which is necessary in order to register subsequent flows

$$\mathcal{F}_{control} = \left\{ \begin{matrix} (1\,,\,\text{n}\,,\,T = 10\,\text{s}\;\,,\,J = 0\,\text{s}\;\,,\,\mathbf{D} = 30\,\text{s}) \\ (\text{n}\,,\,1\,,\,T = 10\,\text{s}\;\,,\,J = 0\,\text{s}\;\,,\,\mathbf{D} = 30\,\text{s}) \end{matrix} \right\}$$

for $n \in (2..20)$. In practice, such flows can also be used to send low-priority data (*e.g.*, status data) regularly to the sink.

An event from the environment (*e.g.*, a rock crack [128]) is co-detected by nodes 2 to 5, which consequently emit a request for a new flow to the sink node. In order to transfer the event data as fast as possible, the message interval is chosen as small as possible (*i.e.*, equal to T_f^s, the flushing interval of CP – Refer to (4.5), (4.6) and (4.14)),

$$\mathcal{F}_{new} = \Big\{ (\text{n}\,,\,1\,,\,T = 1.074\,\text{s}\;\,,\,J = 0\,\text{s}\;\,,\,\mathbf{D} = 10\,\text{s}) \Big\}$$

for $n \in (2..5)$. We record the end-to-end latency of all packets during two minutes, during which about 900 messages are transmitted through the system.

Results. Figure 4.11 shows the distribution of end-to-end latency of messages, shown as percentage of the analytical worst-case latency (given by Theorem 4.1). We see that a few messages indeed experience a latency up to 97 % of the analytic worst-case bound. The simulation also indicates that, in many cases, the worst-case buffer sizes of CP and *Bolt* are reached. Overall, these results support our analysis of *DRP*. They show that our worst-case bounds are tight; therefore, we can conclude that the performance derived using *DRP*'s model (Section 4.7.1) is representative of the performance that can be truly guaranteed by the system.

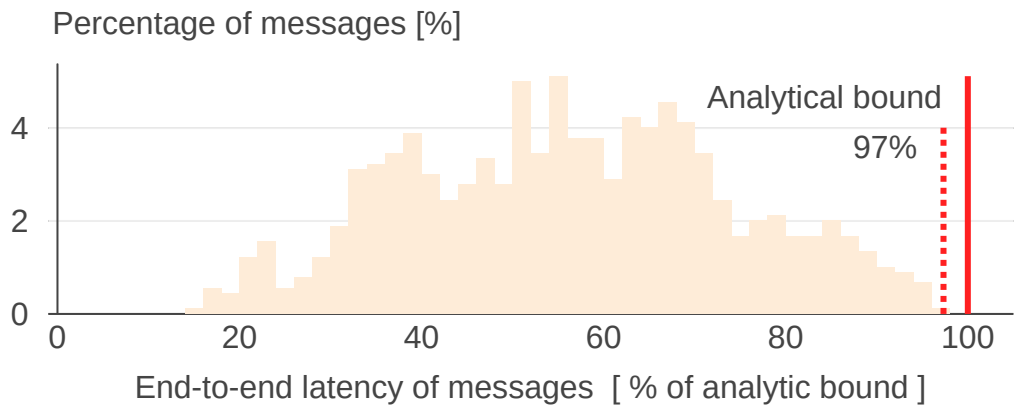


Figure 4.11 Distribution of end-to-end latency of messages, shown as percentage of the analytical worst-case latency. *Some messages experience a latency very close to their worst-case bound (97 %), which demonstrates the tightness of the analysis.*

Table 4.3 Flow sets used in the end-to-end latency evaluation of *DRP*. *The Blink utilization is computed assuming a strictly periodic release of messages.*

Flow label	**Blink utilization**	**Flow set (period, deadline)**
Control flows	na	20 × (15, 60)
Low	41%	Control flows; (10, 58); (10, 58); (11, 59); (14, 52); (15, 53); (16, 54); (16, 54); (19, 57)
Medium	60%	Control flows; (5, 53); (5, 54); (6, 54); (7, 50); (7, 55); (8, 56); (9, 57); (9, 58)
High	92%	Control flows; (3, 55); (3, 56); (3, 56); (3, 56); (3, 59); (3, 59); (4, 51); (4, 53); (4, 55); (4, 57)

4.7.3 Real-World Performance

We now consider the performance of our implementation of *DRP* on embedded hardware: We use the first-generation DPP, which features a TI MSP432P401R as AP and a TI CC430F5147 as CP (Section 1.B). The software is based on the publicly available implementation of LWB [71]; it is written in C and uses Contiki 2.7 [2] as operating system. The implementation of Blink on the AP is built upon [14]. We discuss our implementation performance in terms of memory usage, computation workload, and message latency.

Memory usage. *DRP* requires both AP and CP to store some state information related to the currently running flows, as well message queues and buffers. The available RAM on both processors is shown in Table 4.4.

The 64 kB of the AP are largely sufficient; it would supports hundreds of flows. The CP is more limited: With a payload size of 32 bytes, CP is capped to a maximum for 40 flows (Table 4.4). For a regular node, this will likely be

Table 4.4 Memory available and required for our implementation of *DRP*. *The difference between "total" and "available" memory corresponds to the memory taken by the firmware only.* L *is the message payload.*

Memory [bytes]	**MSP432P401R** AP	**CC430F5147** CP
Total RAM	64k	4k
Available RAM	45k	2.4k
Memory required (per flow)	45	$23+L$

sufficient for most applications; however, on the host node, this seriously limits the scalability of the system.

One possible solution is to use the embedded external memory (128 kB); this would solve the memory limitation issue, but it may also introduce additional delays, which are currently not accounted for. Or we could use another processor as CP with more than 4 kB of RAM.

Computation. The most extensive computations in *DRP* are the computations of the Blink schedules and admission tests. The evaluation of these computations on embedded hardware is discussed in depth in [208].

In addition to Blink computations, the APs must perform *DRP* admission tests. These are simple operations (Theorem 4.2 and 4.3) which can be implemented efficiently. In our experiments, an admission test takes typically around 30 ms to complete (maximum observed execution time: 130 ms).

DRP admission tests are performed only once per flow (when a new flow is requested). Thus, we can conclude that the computational workload induced by *DRP* (in addition to Blink) is negligible.

End-to-end latency. We investigate the experienced end-to-end latency of messages. We use a network of 10 source nodes and one host, and run experiments on the FlockLab testbed [115]. In addition to the control flows, each source node request a data flow toward the host with a pseudo-random period and end-to-end deadline. Once the flow is admitted, the source APs release new packets periodically. The different flow sets used are listed in Table 4.3. *DRP* is configured with a deadline ratio $r = 0.5$, a round length $C_{net} = 1\,\text{s}$, and a maximum of $B_{max} = 5$ slots per round.

The results are summarized in Table 4.5 and Figure 4.12, reporting data from one run for each flow set. The first observation is that all messages that are successfully transmitted over the wireless network do meet their end-to-end deadline. However, compared to the simulation experiment (Section 4.7.2), we do not encounter so much analytical corner cases: the observed latency is often much smaller than the analytical upper-bound (Figure 4.12). On the other hand, this means that the actual runtime performance is better that what is

Table 4.5 Experienced latency of messages, expressed as percentage of the flow's end-to-end deadline. *The tightness correspond to the ratio of the experienced latency with the analytical upper-bound given by Theorem 4.1.*

	Low	**Median**	**High**
Median latency [%]	14 %	7 %	4 %
Maximum latency [%]	58 %	68 %	41 %
Tightness [%]	60 %	89 %	70 %

guaranteed: even with large end-to-end deadlines (around 60 s), the experience message latency is most of the time between 5 s to 15 s.

Such "short" average latency can be explained by the nature of the flow set. The *network* deadline, enforced by Blink, must be smaller than the flow period Section 4.4. Since the period are small compared to the *end-to-end deadlines*, these end-to-end deadlines do not constraint the *DRP* contracts: the experience latency correlates with the flow period.

It is interesting to observe that the average end-to-end latency is smaller for the high utilization flow set than for the low and middle ones. Again, this is due to the flow set. To meet the network deadline, Blink schedule at least one round per period. Thus, a flow set with shorter period results in more frequent rounds. Once a round is scheduled, it is filled with any message ready for transmission. Thus, flows are opportunistically served earlier than necessary to meet their end-to-end deadline. Conclusion: having a flow with a small period reduces the average latency experienced by all flows in the system. This is an interesting (and unforeseen) consequence of *DRP* mechanism.

Conclusions. The performance evaluation presented in this section validates the design of *DRP* and our implementation: we showcased that we can run a wireless CPS that meet end-to-end deadlines between distributed applications (Section 4.7.2 and 4.7.3). In addition, we derived the theoretical optimal performance achievable based on *DRP*'s model (Section 4.7.1).

A more thorough investigation of the actual system performance across different scenarios and environments remains to be performed. For such a performance evaluation, using *TriScale* (Chapter 2) would be natural.

Chronologically, *TriScale* is the last piece of work of this dissertation. In hindsight, our evaluation of *DRP* appears a bit naive and simple. Still, we argue that it successfully demonstrates the soundness of *DRP*'s design.

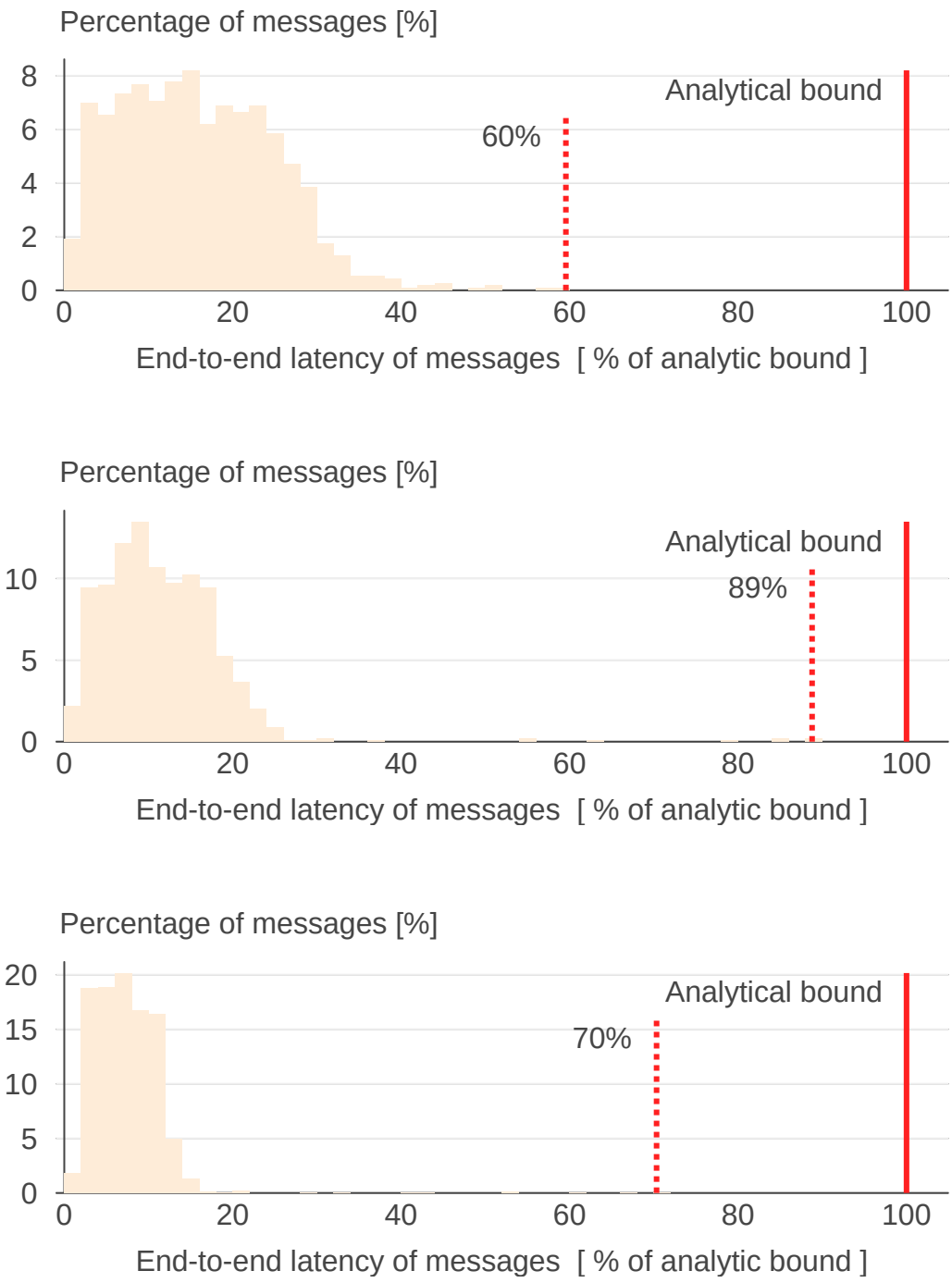


Figure 4.12 Distribution of end-to-end latency of messages, shown as percentage of the analytical worst-case latency for a *DRP* run using different flow sets (Table 4.3). *Top – Low utilization. Middle – Medium utilization. Bottom – High utilization.*

4.8 Related Work

Providing end-to-end guarantees in distributed networked systems has a long history in the context of the Internet. Notable developments are the resource reservation protocol (RSVP) that combines flow specification, resource reservation, admission control, and packet scheduling to achieve end-to-end quality of service (QoS) [204]. Network calculus [57] provides some of the necessary theoretical concepts to determine bounds on buffer sizes and delay in communication networks. Extension toward hard real-time computing and communication systems is known as real-time calculus [184]. The analysis of distributed hard real-time systems also has a long history [186], and so do compositional analysis frameworks, such as MAST [82], SymTA/S [88] and MPA [189].

Early works on real-time communication in sensor networks consider classical non-deterministic routing protocols [118, 169, 87], thus providing only soft guarantees. Stankovic et al. [169] even argue that specific message delivery orderings, such as those useful to apply established dependability techniques [73], are impossible to guarantee in a multi-hop low-power wireless network. More recently, standards like WirelessHART [91] have been analyzed to provide communication guarantees [154, 153]. But [153] is based on NP-hard multiprocessor scheduling and requires a global network view, which limits its adaptability to dynamic changes in the system [19]. It is however possible to integrate the wireless protocol with the rest of the system to avoid interference by jointly schedule transmissions in the network and all other tasks in the system, as we demonstrate in Chapter 5. Other wireless real-time protocols have been described recently [142, 192]. However, the integration of these protocols into a methodology to provide end-to-end real-time guarantees between *application interfaces* is unsolved.

Recently, a game-changing approach to wireless multi-hop communication using synchronous transmissions has been described [70, 72, 207]. It avoids the computation of multi-hop routing paths and per-node communication schedules based on, for example, neighbor lists and link qualities, because the protocol logic is independent of such volatile network state. Experiments on several large-scale testbeds show that the approach is highly adaptive and achieves an end-to-end packet reliability higher than 99.9 % [70, 72]. Furthermore, the few packet losses can be considered statistically independent [207], which eases the design of CPS controllers that can deal with intermittent observations [166].

4.9 Summary

In this chapter, we presented the Distributed Real-time Protocol (*DRP*), a global system design that provides end-to-end real-time guarantees between interfaces

of distributed applications in wireless CPS. *DRP* meets the requirements of *Timeliness*, *Reliability*, *Adaptability*, and *Composability*. However, since *DRP* guarantees relies on worst-case analysis, the system's *Efficiency* is inherently limited; still, we demonstrated that our analysis is tight (Section 4.7) which shows that *DRP* is not overly pessimistic.

The key concept of *DRP* is to (i) physically decouple the communication protocol from the application tasks (each running on dedicated communication and application processors), and (ii) guarantee the timeliness of message transmissions throughout the system using minimally restrictive contracts between the different entities.

We implemented and ran a proof-of-concept implementation of *DRP* on embedded hardware. The firmware source code as well as our *DRP* simulation framework are openly available (Section 4.A). *DRP* appears to be a promising solution for low-rate applications, such as smart homes, where coexists multiple context-specific "applications" (*e.g.*, fridge, air-conditioning, lightning) which would particularly benefit from being scheduled independently from each other while being able to communicate in real-time.

4.A Appendix – Artifacts and Links

4.A.1 Related Publications

End-to-End Real-Time Guarantees in Wireless Cyber-Physical Systems
Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel, Lothar Thiele
RTSS 2016. Porto, Portugal (December 2016)

Paper 10.3929/ethz-a-010881673

Real-time network functions for the Internet of Things
Fabian Walter
Semester Thesis. ETH Zurich (June 2017)

Thesis 10.3929/ethz-b-000234920

Unleashing the potential of Real-time Internet of Things
Andreas Biri
Semester Thesis. ETH Zurich (December 2017)

Thesis 10.3929/ethz-b-000234913

4.A.2 Complementary Materials

Complementary materials for this chapters are available on GitHub, together with the dissertation source files. For all links below, replace `<root>` by "github.com/romain-jacob/doctoral-thesis/blob/master"

TEX sources `<root>`/40_DRP/

Figures
— Static `<root>`/40_DRP/Figures/
— Dynamic `<root>`/notebooks/drp_plots.ipynb

Experiment data
— Latest release 10.5281/zenodo.3530757
— "This-version" release 10.5281/zenodo.3530758

4.B Appendix – Worst-case Latency Analysis

Worst-case analysis of the source delay.

Definition 4.1 (Source delay – δ_{source})**.** The source delay is the elapsed time from a packet being written in *Bolt* by the source AP_s until the end of the `flush` operation where it is read out of *Bolt* by the source CP_s. For a flow F_i, it is denoted by $\delta_{source,i}$.

Lemma 4.4. *For any flow F_i, the source delay is upper-bounded by*

$$\delta_{source,i} \leq C_w + T_f^s + C_f \tag{4.27}$$

Proof. Let us recall that a `flush` is a sequence of `read` operations. When the *Bolt* queue is found empty, the `flush` is terminated and no other `read` is performed until the next `flush` (refer to 4.4.1 for details). Therefore, if the *Bolt* queue is empty and a `write` operation terminates just after a `flush` is triggered, that `flush` immediately terminates and the packet is delayed until to the end of the next `flush`. Possible jitter on the `write` operation pattern does not have any influence on the worst-case for $\delta_{source,i}$. This worst-case scenario for the source delay is illustrated on Fig. 4.13. ■

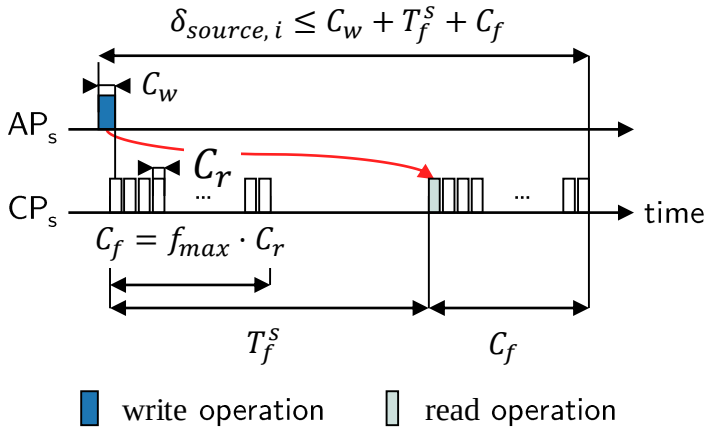


Figure 4.13 Worst-case analysis of the source delay. *A packet is written as early as possible such that it misses a* `flush` *and must wait until the next one.*

Worst-case analysis of the network delay.

Definition 4.2 (Network delay – $\delta_{network}$)**.** The network delay is the elapsed time from a packet being available for communication at the source CP_s until the end of the communication round where it is served by the wireless protocol (i.e., when it is available at the destination CP_d). For a flow F_i, it is denoted by $\delta_{network,i}$.

Lemma 4.5. *For any flow F_i, the network delay is upper-bounded by*

$$\delta_{network,i} \leq T_i + D_i + \left\lfloor \frac{J_i + C_f - C_r}{T_f^s} \right\rfloor \cdot T_f^s \tag{4.28}$$

Proof. As presented in Section 4.4.2, Blink guarantees that every packet matching the *expected arrival* is served in a round that terminates before the network deadline D_i. Hence, the *delay of an expected packet* is no more than D_i.

However, the actual arrival of packets at the source CP_s does not match the expected arrival in general, but results from `flush` operations, which occur every T_f^s time unit. Hence, a packet may arrive *earlier* than the next expected packet. That mismatch between the two arrival times (actual and expected) adds up with the delay of the expected packet (i.e., D_i).

Let us consider first that the flow F_i has no jitter (*i.e.*, $J_i = 0$) and let m be the mismatch between actual and expected arrival time at CP_s. m cannot be larger than the flow's minimum message interval T_i

$$m \leq T_i$$

The intuition is given with Figure 4.14. See the caption for details.

Now, if flow F_i has also jitter J_i, this may entail a bigger mismatch. Actual "arrival" of packets (*i.e.*, the epoch when a packet is available for communication at the source CP_s, according to the definition of the network delay) can occur only every T_f^s (*i.e.*, at the end of one `flush` operation). Therefore, one can see that jitter may induce an extra delay, or mismatch, of roughly $\left\lfloor J_i/T_f^s \right\rfloor \cdot T_f^s$. A more precise analysis of the flushing dynamics (see Figure 4.15 for details) entails that, overall, the worst-case mismatch m is bounded by

$$m \leq T_i + \left\lfloor \frac{J_i + C_f - C_r}{T_f^s} \right\rfloor \cdot T_f^s \tag{4.29}$$

and finally,

$$\begin{aligned} \delta_{network,i} &\leq D_i + m \\ \delta_{network,i} &\leq T_i + D_i + \left\lfloor \frac{J_i + C_f - C_r}{T_f^s} \right\rfloor \cdot T_f^s \end{aligned}$$

■

Worst-case analysis of the destination delay.

Definition 4.3 (Destination delay – δ_{dest})**.** The destination delay is the elapsed time from a packet being available at the destination CP_d until the end of the `flush` operation where it is read out of *Bolt* by the destination AP_d (i.e., when it is available for the application). For a flow F_i, it is denoted by $\delta_{dest,i}$.

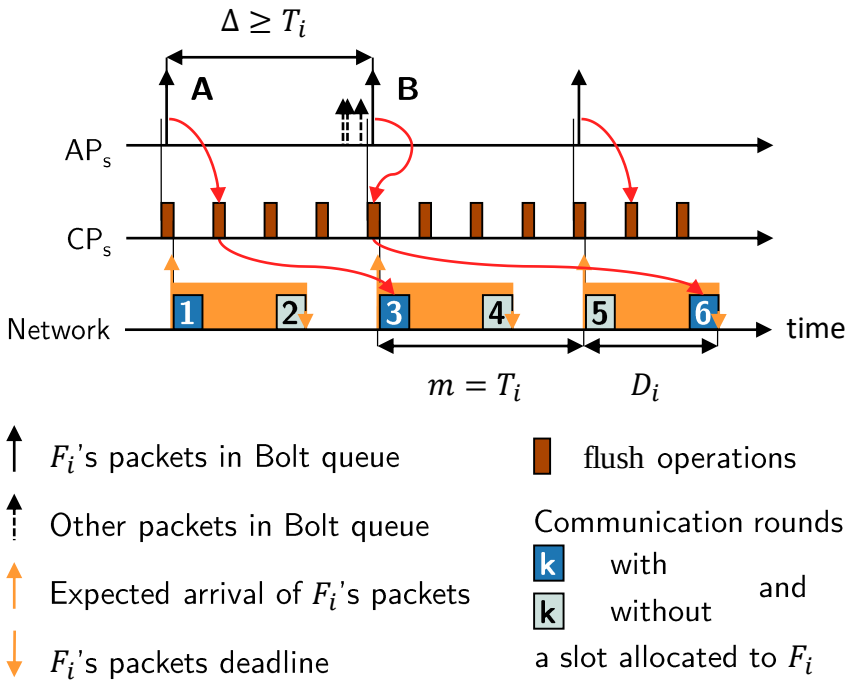


Figure 4.14 Worst-case analysis of the network delay without jitter. *Because of the Bolt queue being empty, packet **A** misses the first flush operation (similarly as in Figure 4.13), hence the slot allocated to F_i in round **1** is wasted. Due to packets released from other flows in the meantime, packet **B** is flushed directly in the operation preceding round **3**, in which flow F_i is allocated a new slot. However, as packet **A** is still in queue, packet **B** is not served right away but is delayed until the next allocated slot (*i.e., *in round **6**). This creates a mismatch of T_i for packet **B**. Furthermore, the mismatch cannot get bigger; assume **B** were to be available at CP_s earlier (*i.e., *one flush operation before, at least), because the time interval between **A** and **B** must be at least T_i, **A** would arrive earlier as well. Hence, **A** would not miss the slot in round **1**, **B** would be served in round **3**, and thus it would yield a smaller mismatch for packet **B**.*

Lemma 4.6. *For any flow F_i, the destination delay is upper-bounded by*

$$\delta_{dest,i} \leq B_{max} * C_w - (B_{max} - 1) * C_r + T_f^d + C_f \tag{4.30}$$

Proof. The situation is similar as for the source delay, except that CP_d writes every T_{net} time unit (*i.e.*, after each round) all the packets it received during the last round, which can be as many as B_{max} packets. The maximal delay for a packet occurs when it is written too late to be read out during an ongoing `flush` and must wait for the next one.

A careful analysis of the *Bolt* dynamics shows that the `read` operation is slightly shorter than `write` [174] (i.e., $C_r < C_w$, see Table 4.2). Hence, the more packets are written at once by CP_d, the later a `flush` can start and still miss the last written packet. The worst-case is illustrated on Fig. 4.16. ■

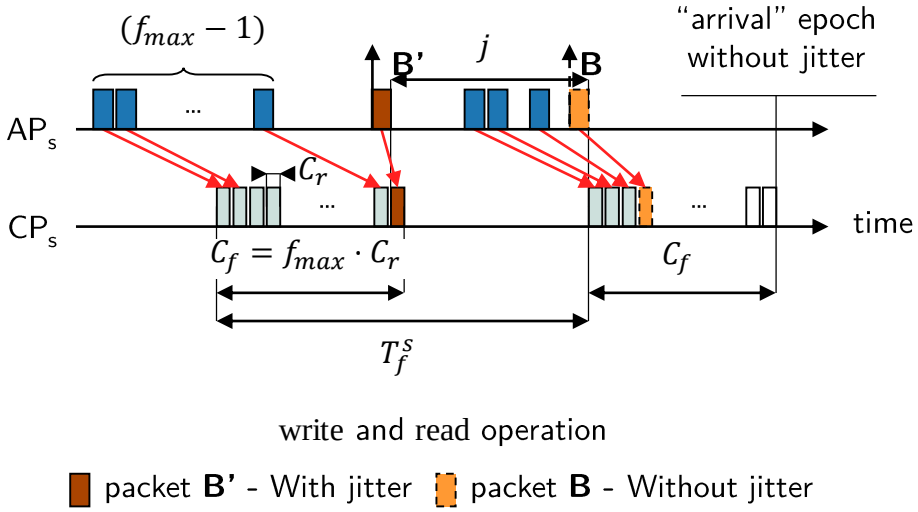


Figure 4.15 Influence of jitter on the network delay. *Let us have a closer look at packet **B** from the previous figure, positioned as early as possible (*i.e., *if it were earlier, so would be **A**, which would then not miss its slot in round **1**). Due to jitter, **B** is released earlier, say by a amount j. This can yield packet **B'** (**B** with jitter) to be read out in a previous* `flush` *operation. In the worst-case, packet **B'** is read out one operation earlier as soon as j is bigger than $T_f^s - C_f + C_r$, which increases the mismatch m by T_f^s. Similarly, m increases by $k \cdot T_f^s$ when j reached $k \cdot T_f^s - C_f + C_r$, which yields $k = \left\lfloor \frac{j + C_f - C_r}{T_f^s} \right\rfloor$ and concludes to equation* (4.29).

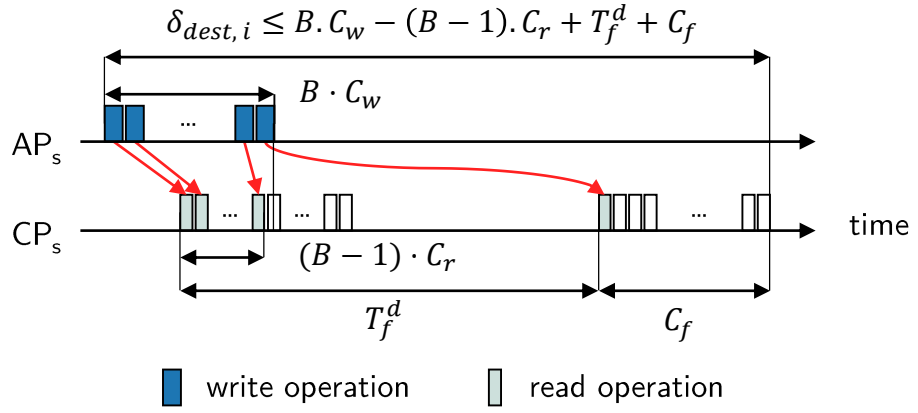


Figure 4.16 Worst-case analysis of the destination delay. *A packet is written as early as possible such that it misses a* `flush` *and must wait until the next one.*

5

TTW: A Time-Triggered Design for Wireless Cyber-Physical Systems

We revisit the challenge addressed in the previous chapter: Providing end-to-end real-time guarantees in wireless cyber-physical systems (CPS). With the design of *DRP* (Chapter 4), we demonstrated that, by leveraging synchronous transmissions (ST), it is possible to meet end-to-end deadlines between distributed tasks communicating through a multi-hop wireless network. The principle of *DRP* is to keep the tasks as independent as possible; *i.e.*, constraining their schedule as little as necessary to provide end-to-end guarantees.

Because of that maximal-flexibility principle, the guarantees that can be provided by *DRP* are rather "slow": the minimal end-to-end deadline supported by the protocol is more than two times as large as a communication round (Section 4.7). Furthermore, there is large jitter between successive task executions and message transmissions. This does not comply well with the requirements of industrial CPS applications, which often require short delays (the order of ms) and benefit from negligible jitter.

Thus, in this chapter we change the design objective: Instead of focusing on flexibility, we aim for minimizing latency and jitter in the system execution.

Claim. We demonstrate for that end-to-end real-time guarantees can be obtained in low-power wireless networks by leveraging the efficiency and reliability of synchronous transmissions. In particular, this chapter presents Time-Triggered Wireless (*TTW*), a design that statically co-schedules all task executions and message transfers to minimize end-to-end latency and jitter.

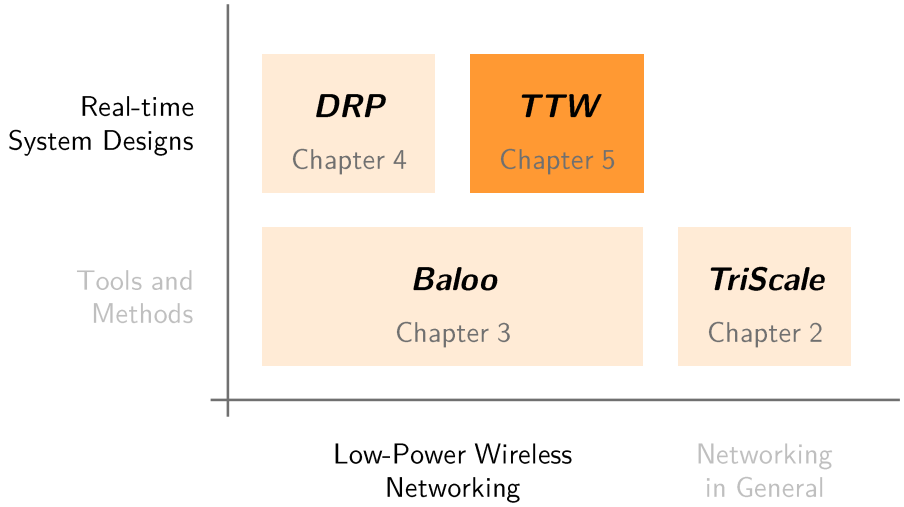


Figure 5.1 This chapter presents the Time-Triggered Wireless (*TTW*), a wireless CPS design that statically co-schedules all task executions and message transfers to minimize end-to-end latency and jitter.

The material from this chapter relates to the following publication.

TTW: A Time-Triggered Wireless design for CPS
Romain Jacob, Licong Zhang, Marco Zimmerling, Jan Beutel, Samarjit Chakraborty, Lothar Thiele
DATE 2018. Dresden, Germany (March 2018)

5.1 Problem Setting

CPS are understood as systems where *"physical and software components are deeply intertwined, each operating on different spatial and temporal scales, exhibiting multiple and distinct behavioral modalities, and interacting with each other in a myriad of ways that change with context"* [138]. Application domains include, *e.g.*, robotics, distributed monitoring, process control, power-grid management. CPS combine physical processes, sensing, online computation, communication, and actuation in a single distributed control system.

To support CPS, industry has been widely relying on wired field buses such as CAN, FlexRay, ARINC 429, AFDX, with good reasons. They combine functional and non-functional predictability with appropriate bandwidth, message delay, and fault-tolerance. Yet, several of the above application domains would benefit from wireless communication for its ease of installation, logical and spatial reconfigurability, and flexibility; in other domains, such as mobile scenarios, wireless is simply the only viable option.

One major obstacle in using wireless communication for CPS has been the reliability of packet transmission. However, in the last decade, several low-power protocols featuring very low packet loss rate have been proposed and standardized (*e.g.*, TSCH [192]). Another technique, called synchronous transmissions (ST– Section 1.3), has been proven to be highly reliable and energy efficient. The most striking evidence of ST benefits has been the EWSN Dependability Competition [160], where wireless protocols are tested in high-interference environments: all wining solutions in the last four years (2016 to 2019) were based on ST [67, 167, 114, 69, 120].

Despite these achievements, wireless CPS solutions capable of providing end-to-end guarantees for distributed applications are still missing. It is challenging to concurrently meet all the requirements of typical industrial applications [18]: reliability, timing predictability, low end-to-end latency at the application level,[1] energy efficiency, and quick runtime adaptability to different modes of operation.

Key Research Questions

Question 1 Can we provide end-to-end real-time guarantees between distributed applications in wireless CPS?

Question 2 How can we minimize latency and jitter in the application execution while retaining some level of runtime adaptability?

The problem. To understand the challenges of wireless CPS, it is helpful to highlight the fundamental difference between a field bus and a wireless network. In a field bus, whenever a node is not transmitting, it can idly listen for incoming messages. Upon request from a central host, each node can wake-up and react quickly. For a low-power wireless node, the major part of the energy is consumed by its radio. Therefore, energy efficiency requires to turn the radio off whenever possible to support long autonomous operation without an external power source. Since nodes are unreachable until they wake up, they require overlapping wake-up time intervals to communicate.

This observation often results in wireless system designs that minimize energy consumption by using communication rounds, *i.e.*, time intervals where all nodes wake-up, exchange messages, then turn off their radio [91, 192, 72, 96]. Scheduling policies define when the rounds take place (*i.e.*, when to wake up) and which nodes are allowed to send messages during the round. Moreover, CPS do not only exchange messages, they also execute tasks (*e.g.*, sensing or actuation). Typically, the system requirements are specified end-to-end, *i.e.*, between distributed tasks exchanging messages. One option to meet such end-to-end requirements (**Question 1**) is to co-schedule the execution of tasks and the transmission of messages, as proposed in the literature for

[1] Range of 10-500 ms delay for a distributed closed-loop control system [18].

wired architectures [13, 56, 23]. However, these schedules result from complex optimization problems which are difficult to solve online, even more so in a low-power setting. Thus, schedules are often pre-computed offline, which restricts the runtime adaptability of the resulting system (**Question 2**).

The challenge. To support wireless CPS applications in an industrial context, a solution to this problem should fulfill the following requirements.

Timeliness All distributed applications meet their end-to-end deadlines.

Reliability A large ratio of messages are successfully transmitted over wireless and conflict-free communication is guaranteed between the system's nodes.

Adaptability The system adapts to dynamic changes at runtime.

Mobility The system supports mobile devices.

Efficiency The system supports short end-to-end latency (in the ms range), scales to medium-to-large system sizes, and optimizes its energy consumption and bandwidth utilization.

Our solution. In this chapter, we propose a solution to the industrial wireless CPS problem that fulfill these requirements. We do so by combining co-scheduling techniques, inspired from the wired literature, with a ST-based wireless system design using communication rounds.

ST provides highly reliable wireless communication (*Reliability*) and inherent support for *Mobility*. A round-based design allows to minimize the energy consumed for communication, which is a large part of the total energy budget of a low-power system (*Efficiency*). The co-scheduling approach results in highly optimized schedules (*Efficiency*) which guarantee to meet the application deadlines (*Timeliness*). Our system provides some runtime *Adaptability* by switching between multiple pre-computed operation modes, a well-known concept in the wired literature [77].

The main challenge in realizing such a system is to integrate the allocation of messages to communication rounds (which is similar to a bin-packing problem [196]) with a co-scheduling approach (which typically solves a MILP [24] or a SMT [172, 55, 90] formulation).

The contributions of this chapter are summarized below.

- We present Time-Triggered Wireless (*TTW*), a low-power wireless CPS that meets the common requirements of industrial applications.

- We formulate a joint optimization problem for co-scheduling distributed tasks, messages, and communication rounds that guarantees to meet application deadlines, minimize the energy consumed for wireless communication, and ensures safety in terms of conflict-free communication, even under packet loss.

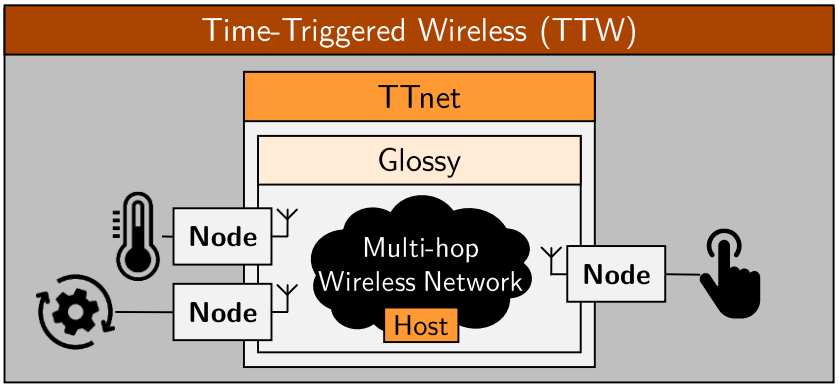


Figure 5.2 Overview of *TTW*. *Nodes execute distributed CPS applications. They communicate over a multi-hop wireless network using Glossy floods [70]. Communication is organized in TTnet rounds (Figure 5.3) which are controlled by a central host (running on one of the nodes). TTW is a global scheduler for the entire system: it co-schedules the execution time of all tasks and messages in order to reduce the end-to-end latency of applications and meet short deadlines.*

- We provide a methodology that efficiently solves this optimization problem, known to be NP-hard [101].
- Using time and energy models, we quantify the benefits of rounds to minimize energy, and we derive the minimum end-to-end latency achievable.
- We implement *TTW* on embedded hardware and demonstrate that the system is suited for fast feedback control applications.

5.2 Overview of *TTW*

We first present *TTnet*, a network stack based on synchronous transmissions (ST), which serves as communication backbone for our solution (Section 5.2.1). We then introduce the concepts of *TTW* (Section 5.2.2), a system-wide scheduler built atop *TTnet* to realize a wireless CPS solution meeting the requirements described above.

5.2.1 The *TTnet* communication backbone

We consider a set of nodes connected by a wireless multi-hop network (Figure 5.2). Each node is a low-power embedded device, typically battery-powered, with limited computational resources such as memory or processing power. These devices collectively implement distributed applications (*e.g.*, closed-loop control). These applications are composed of multiple tasks and messages; the tasks are executed locally by the nodes; the messages are exchanged over the multi-hop wireless network. In low-power wireless CPS, a significant part of the

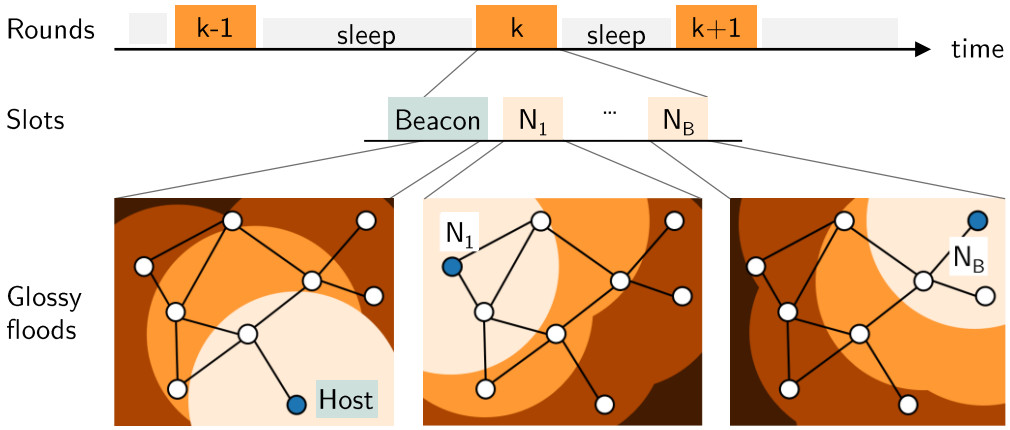


Figure 5.3 The *TTnet* network stack. *TTnet organizes communication in time-triggered rounds, between which nodes turn their radio off to save energy. The rounds are composed of (up to) B communication slots preceded by a beacon, sent by the host to distribute runtime control information. In each slot, an one-to-all communication is realized using Glossy [70].*

energy is consumed by wireless communication. Thus, to minimize the energy consumption, we group messages into communication rounds, *i.e.*, time intervals where all nodes turn their radio on and communicate. Each round is composed of dedicated time slots where nodes communicates using Glossy, a flooding protocol which delivers packets with a probability above 99.9 % [70]. The system is controlled centrally by a node called the host, which sends commands at the beginning of each round in a special slot called beacon. Physically, one of the nodes plays the role of the host. We call this network stack *TTnet* (Figure 5.3).

TTnet concept of round-based communication using Glossy floods is inspired by the Low-power Wireless Bus (LWB) [72]; this design has several benefits:

- It is based on Glossy, which has been proven to be highly reliable and energy efficient ([160, 114, 68]).
- Glossy provides sub-microsecond time synchronization across the network [70], which is instrumental to achieve *Timeliness* and *Reliability*.
- The flooding process in Glossy is independent of the network state; thus it creates a virtual single-hop network where each node can communicate with every other node in bounded time. As a result, network stacks like LWB or *TTnet* can be scheduled like a shared bus.
- As messages are flooded in the entire network, unicast multicast and broadcast are equivalent: for a given payload, the transmission time only depends on the network diameter (the maximal hop distance between nodes).
- Thanks to its stateless flooding logic, Glossy inherently supports *Mobility*.

Despite these benefits, LWB or *TTnet* alone cannot meet all the requirements of wireless CPS. In particular, one must account for the scheduling of

distributed tasks in order to provide end-to-end timing guarantees (*Timeliness*). This motivates the design of *TTW*, a real-time scheduler for the *TTnet* stack.

5.2.2 Building-up *TTW*

The *TTnet* is the communication backbone of *TTW*. Building on that structure, we design *TTW*, a real-time scheduler for the entire wireless CPS.

TTW is based on four key concepts.

Global co-scheduling In order to minimize the achievable end-to-end latency, *TTW* co-schedules the task executions and message transmissions, similarly to the state-of-the-art for wired protocols (*e.g.*, [56, 203]). Moreover, the round-based design of *TTnet* demands to integrates the communication rounds to the schedules. The allocation of messages to communication rounds is similar to a bin-packing problem [196]. Combining pin-packing with traditional task-and-message co-scheduling approaches is non trivial (Section 5.4).

The resulting problem is a complex optimization that cannot be solved online, even less in a low-power setting. Therefore, *TTW* statically synthesizes the schedule of all tasks, messages, and rounds to meet real-time constraints, minimize end-to-end latency, and minimize the energy consumed for communication. The schedule is synthesized by solving an MILP (mixed integer linear programming) formulation.

Static schedules Since *TTW* relies on static scheduling, we distribute the schedules at deployment time to limit the communication overhead at runtime, thus optimizing energy efficiency. Each node stores its own schedule information, thereby trading-off memory utilization with energy consumption; this significantly improves the *Efficiency* of the system.

Multiple operation modes The obvious drawback of using static schedules is that the system always execute the same schedule, compromising *Adaptability*. *TTW* mitigates this problem by using the traditional concept of operation modes [77]: Multiple schedules are computed offline and stored in the nodes' memory. The system can switch at runtime between different modes, thereby recovering some degree of *Adaptability*.

Runtime control At the beginning of each round, the host sends a beacon, which is used to control the system execution at runtime. A beacon contains the current round id, the mode id, and a trigger bit used in the mode change procedure (Section 5.6).

Thanks to the distributed schedule information, it is sufficient for any node to receive a single beacon to retrieve the system state (*i.e.*, the phase of the schedule given by the round id) and therefore know (i) which message to send in which slot, and (ii) when to wake up for the next communication

round. If a node does not receive the beacon, it does not participate in the round. Hence, even if nodes miss some control information, they do not initiate a communication in a slot allocated to another node, thus guaranteeing conflict-free communication (*Reliability*).

By globally optimizing the entire system schedule, *TTW* can meet tight end-to-end deadlines (tens of ms) while minimizing the energy spent for wireless communication, thus addressing the *Timeliness* and *Efficiency* requirements. The runtime control based on beacons provides *Reliability*, while switching between multiple operation modes at runtime offers some degree of *Adaptability*. Finally, *Mobility* is supported by design thanks to the stateless logic of Glossy [70], the underlying communication primitive used by *TTW*.

Remark 6. *TTW* combines offline scheduling and online decisions whereas *DRP* (Chapter 4), by contrast, does everything online. Hence, *TTW* trades the flexibility of execution of the distributed application tasks for short latency and fast mode changes.

5.3 System Model and Scheduling Problem

Nodes. We denote by $\mathcal{N}$ the set of *nodes* in the system. Nodes implement distributed applications, composed of multiple tasks to execute and messages to exchange. A node is considered capable of performing one task execution and one message transmission simultaneously; this is supported by state-of-the-art wireless CPS platforms featuring two cores, such as the NXP LPC541XX [140], VF3xxR [141], or more generally any platform following the Dual-Processor Platform concept (Section 1.4).

Applications. We denote by $\mathcal{A}$ the set of *applications* in the system. Each distributed application is composed of *tasks* and *messages* connected by precedence constraints described by a directed acyclic graph, where vertices and edges represent tasks and messages, respectively. We denote by $a.\mathbb{P}$ the *precedence graph* of application a (Figure 5.4). Each application executes at a periodic interval $a.p$, called the *period*. An application execution is completed when all tasks in $\mathbb{P}$ have been executed. All tasks and messages in $a.\mathbb{P}$ share the same period, $a.p$. Applications are subject to real-time constraints: The application *relative deadline*, denoted by $a.d$ represents the maximum tolerable *end-to-end delay* to complete the execution. The deadline is arbitrary (*i.e.*, it has no relation with the period $a.p$). Certain critical applications may require to keep the same schedule (*e.g.*, same task offsets) when switching between operation modes. We call these *persistent applications* and denote their set by $\mathcal{A}_P$; $\mathcal{A}_P \subset \mathcal{A}$. In summary, an application a is characterized by

$$a = \{ \quad a.p \quad - \quad \text{period}$$

$$\begin{array}{rcll} & a.d & - & \text{end-to-end deadline} \\ & a.\mathbb{P} & - & \text{precedence graph} \quad \} \end{array}$$

Tasks. We denote by $\mathcal{T}$ the set of tasks. A node executes at most one task at any point in time and we consider non-preemptive task scheduling. Each task τ is mapped to a given node $\tau.map$, on which it executes within a WCET (worst-case execution time) $\tau.e$. The *task offset* $\tau.o$ represents the start of the task execution, relative to the beginning of the application execution. A task can have an arbitrary number of preceding messages, *i.e.*, messages that must be received before the task can start. $\tau.prec$ denotes the set of preceding message ids. Within one application, each task is unique; however, the same task may belong to multiple applications (*e.g.*, the same sensing task may source different feedback loops). If so, we consider that these applications have the same period. In summary, a task τ is characterized by

$$\begin{array}{rcrcll} \tau & = & \{ & \tau.o & - & \text{offset} \\ & & & \tau.map & - & \text{mapping} \\ & & & \tau.e & - & \text{WCET} \\ & & & \tau.prec & - & \text{preceding message set} \\ & & & \tau.p & - & \text{period (equal to } a.p) \quad \} \end{array}$$

Messages. We denote by $\mathcal{M}$ the set of messages. Every message m has at least one preceding task, *i.e.*, a task that needs to finish before the message can be transmitted. The set of preceding task ids is denoted by $m.prec$. The *message offset* $m.o$, relative to the beginning of the application execution, represents the earliest time the message m can be allocated to a round for transmission, *i.e.*, after all preceding tasks are completed. The *message deadline* $m.d$, relative to the message offset, represents the latest time when the message transmission must be completed, *i.e.*, the earliest offset of successor tasks. All messages share the same maximal payload L_{max}. Messages are not necessarily unique, *i.e.*, multiple edges of $a.\mathbb{P}$ can be labeled with the same message m, which captures the case of multicast or broadcast (Figure 5.4). If the same message belongs to multiple applications, we consider that these applications have the same period. In summary, a message m is characterized by

$$\begin{array}{rcrcll} m & = & \{ & m.o & - & \text{offset} \\ & & & m.d & - & \text{deadline} \\ & & & m.prec & - & \text{preceding task set} \\ & & & m.p & - & \text{period (equal to } a.p) \quad \} \end{array}$$

Operation modes. We denote by $\mathcal{O}$ the set of operation modes (also simply called "modes"). These modes represent mutually exclusive phases of the

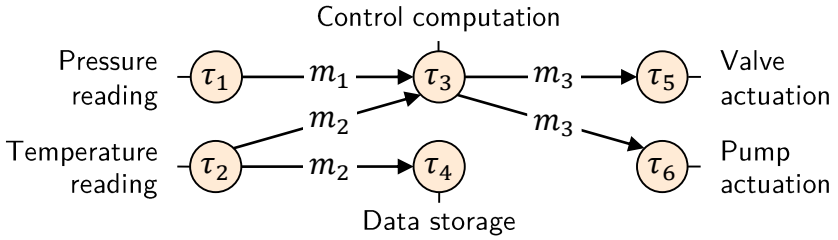


Figure 5.4 An example application and its precedence graph $\mathbb{P}$. *The execution starts with sensor readings – either τ_1 or τ_2. After both have been received by the controller, actuation values are computed (τ_3), multicast to the actuators (m_3), and applied (τ_5 and τ_6).*

system, *e.g.*, boostrapping, normal, and emergency modes; each having its own schedule. Each mode has a unique *priority* $M.prio$, used for scheduling purposes (Section 5.5). A mode M is characterized by

$$M = \{ \quad a_i\,,\ a_j\,,\ \ldots \quad - \quad \text{applications to execute} \\ M.prio \quad - \quad \text{priority} \quad \}$$

We write $a \in M$ to denote that a is executing in mode M. When unambiguous, we use M to denote the set of applications executing in mode M. The mode *hyperperiod* LCM is the least common multiple of the mode's applications. Possible transitions between modes at runtime are described with the *mode graph* $\mathbb{M}$ (Figure 5.7). The mode graph is undirected; a transition from M_i to M_j implies that it is possible to transition from M_j to M_i as well.

Rounds. The schedule of a mode M contains R_M *communication rounds* r. Rounds are *atomic*; that is, they cannot be interrupted. Therefore, the ordering of messages within one round does not matter.[2] Each round r is composed of B_r slots (with a maximum of B_{max}), each allocated to a unique message m. This results in a round length $T_r = T_o + B_r * T_{slot}$, where T_{slot} is the length of one communication slot, and T_o is the constant time overhead per round. The round *starting time* $r.t$ is the start of the round relative to the beginning of the mode hyperperiod. The *allocation vector* $r.[B_r]$ is a vector of size B_r containing the ids of the messages allocated to the slots. $r.B_s$ denotes the allocation of the s-th slot. In summary, a round r is characterized by

$$r = \{ \quad r.t \quad - \quad \text{starting time} \\ r.[B] \quad - \quad \text{allocation vector} \quad \}$$

Scheduling problem. We consider that all modes, applications, task mappings and WCETs are given. For a given mode M, the remaining variables define the

[2] *TTW* could be extended to account for the relative ordering of messages in a round. In theory, this would allow to further reduce the achievable latency of applications at the cost of a more complex synthesis problem to solve.

Table 5.1 Inputs and outputs of the scheduling problem solved by *TTW*

Inputs	$\mathcal{N}$	Set of nodes in the system
	$\mathcal{A}$, $\mathcal{A}_P$	Set of applications and persistent applications (including periods, deadlines, and precedence graphs)
	$\mathcal{O}$	Set of operation modes (including mode priorities)
	$\mathbb{M}$	Mode graph
	$\tau.p$, $m.p$	Task and message periods, inherited from the application
	$\tau.map$	Mappings of tasks to nodes
	$\tau.e$	Task worst-case execution time (WCET) given $\tau.map$
	B_{max}	Maximum number of slots per round
	L_{max}	Maximum message payload size
	H	Network diameter (in number of hops)
	N	Number of transmissions in a Glossy flood [70]
Outputs	$\tau.o$	Task offsets
	$m.o$, $m.d$	Message offsets and deadlines
	$r.t$, $r.[B_{max}]$	Round starting times and allocation vectors

mode schedule, denoted by $Sched(M)$:

$$Sched(M) = \left\{ \begin{array}{l} \tau.o,\, m.o,\, m.d \\ r_k.t,\, r_k.[B_{max}] \end{array} \middle| \begin{array}{l} \forall\ a \in M,\ (\tau, m) \in a.\mathbb{P} \\ \forall\ k \in [1,\, R_M] \end{array} \right\}$$

A schedule for mode M is said to be *valid* if all applications executing in M meet their end-to-end deadlines. The scheduling problem to solve consists in deriving valid schedules for all operation modes in $\mathcal{O}$ such that

(O1) The number of communication rounds is minimized, thereby minimizing the energy consumed for wireless communication.

(O2) All persistent applications $\mathcal{A}_P \subset \mathcal{A}$ seamlessly switch between modes; *i.e.*, their schedule remains the same over a mode change.

All inputs and output of the problem are summarized in Table 5.1.

Application use case. Consider the control of physical systems demanding update rates in the order of tens of ms, which are common in an industrial context [18]. The classical proof-of-concept application is the closed-loop control of an inverted pendulum [42]. With *TTW*, one can use low-power wireless technology to remotely control multiple such pendulums, as demonstrated in [123]. Different *TTW* operation modes may correspond to different control tasks: *e.g.*, solely stabilizing the pendulums or synchronizing their positions [122]. Furthermore, thanks to the stateless logic of *TTnet* (inherited from Glossy [70]), *TTW* inherently supports *Mobility*.[3]

Roadmap. The rest of this chapter presents how *TTW* solves the scheduling problem described above. In Section 5.4, we presents how to synthesize a

[3]Mobility experiment (1min): youtu.be/19xPHjnobkY

valid schedule for a single mode such that the number of communication round used is minimized **(O1)**. Then, in Section 5.5, we address the extension to the multi-mode case, and in particular how to allow applications to keep the same schedules in different modes **(O2)**. The subsequent sections discuss our implementation of *TTW* and its performance evaluation.

5.4 Single Mode Schedule Synthesis

TTW statically synthesizes the schedule of all tasks, messages, and communication rounds to meet real-time constraints by solving a MILP formulation. This section presents how to solve it efficiently and ensure that the resulting schedule minimizes the number of communication rounds **(O1)**.

The schedule of a mode M is computed for one hyperperiod, after which it repeats itself. To minimize the number of rounds used while handling computational complexity, we solve the problem sequentially, as described in Algorithm 1. Each formulation considers a fixed number of rounds R_M to be scheduled, starting with $R_M = 0$. The number of rounds is incremented until a feasible solution is found, or until the maximum number of rounds R_{max} (the number of rounds that can "fit" into one hyperperiod) is reached. Thus, Algorithm 1 guarantees by construction that if the problem is feasible, the synthesized schedule is optimal in terms of number of rounds used.

Algorithm 1 Pseudo-code of the single-mode schedule synthesis

Input: mode M, $a \in M$, $\tau.map$, $\tau.e$, B_{max}, T_o
Output: $Sched(M)$

```
LCM ← hyperperiod(M)
R_max ← floor(LCM/T_o)
R_M ← 0
while R_M ≤ R_max do
    formulate the MILP for mode M using R_M rounds
    [ Sched(M), feasible ] = solve( MILP )
    if feasible then return Sched(M)
    end if
    R_M ← R_M + 1
end while
return 'Problem infeasible'
```

The MILP formulation contains a set of classical scheduling constraints: The precedence constraints between tasks and messages must be respected; Applications end-to-end deadlines must be satisfied; Nodes process at most one task simultaneously; Communication rounds must not overlap; Rounds must not be allocated more then B_{max} messages. These constraints can be easily formulated using our system model (full formulation in Section 5.A). However, one must also guarantee that the allocation of messages to rounds is valid, *i.e.*,

(C1) Messages must be served in rounds that start after their release time.

(C2) Messages must be served in rounds that finish before their deadline.

In other word, we must integrate the bin-packing problem of messages to rounds within the MILP formulation. This is non-trivial and a major difference with the existing approaches for wired architectures (*e.g.*, [56]).

To address this challenge, we first formulate the constraints **(C1)** and **(C2)** using *arrival*, *demand*, and *service* functions, af df and sf, using network calculus [112]. Those functions count the number of message instances released, with passed deadlines, and served since the beginning of the hyperperiod, respectively. Those three functions are illustrated in Figure 5.5. It must hold that

$$\forall\, m_i \in \mathcal{M},\ \forall\, t, \qquad df_i(t) \leq sf_i(t) \leq af_i(t) \tag{5.1}$$

$$\text{with,} \qquad af_i :\ t \longmapsto \left\lfloor \frac{t - m_i.o}{m_i.p} \right\rfloor + 1 \tag{5.2}$$

$$\text{and,} \qquad df_i :\ t \longmapsto \left\lceil \frac{t - m_i.o - m_i.d}{m_i.p} \right\rceil \tag{5.3}$$

However, as the service function stays constant between the rounds, we can formulate **(C1)** and **(C2)** as follows
$\forall\, m_i \in \mathcal{M},\ \forall\, j \in [1..R_M]$,

$$\textbf{(C1)} \quad : \qquad sf_i(r_j.t + T_r) \ \leq\ af_i(r_j.t) \tag{5.4}$$

$$\textbf{(C2)} \quad : \qquad sf_i(r_j.t) \ \geq\ df_i(r_j.t + T_r) \tag{5.5}$$

The arrival and demand functions are step functions. They cannot be used directly in an MILP formulation, however

$$\forall\, k \in \mathbb{N}, \qquad af_i(t) = k \quad \Leftrightarrow \quad 0 \leq t - m_i.o - (k-1)m_i.p < m_i.p \tag{5.6}$$

$$\text{and} \qquad df_i(t) = k \quad \Leftrightarrow \quad 0 < t - m_i.o - m_i.d - (k-1)m_i.p \leq m_i.p \tag{5.7}$$

For each message $m_i \in \mathcal{M}$ and each round r_j, $j \in [1..R_M]$, we introduce two integer variables k_{ij}^a and k_{ij}^d that we constraint to take the values of af and df at the time points of interest (respectively $r_j.t$ and $r_j.t + T_{r_j}$). That is,

$$0 \leq r_j.t - m_i.o - (k_{ij}^a - 1)m_i.p < m_i.p \tag{5.8}$$

$$0 < r_j.t + T_{r_j} - m_i.o - m_i.d - (k_{ij}^d - 1)m_i.p \leq m_i.p \tag{5.9}$$

$$\text{Thus,} \quad (5.8) \quad \Leftrightarrow \quad af_i(r_j.t) = k_{ij}^a$$

$$(5.9) \quad \Leftrightarrow \quad df_i(r_j.t + T_{r_j}) = k_{ij}^d$$

Finally, we must express the service function sf, which counts the number of message instances served *at the end* of each round. Remember that $r_k.B_s$

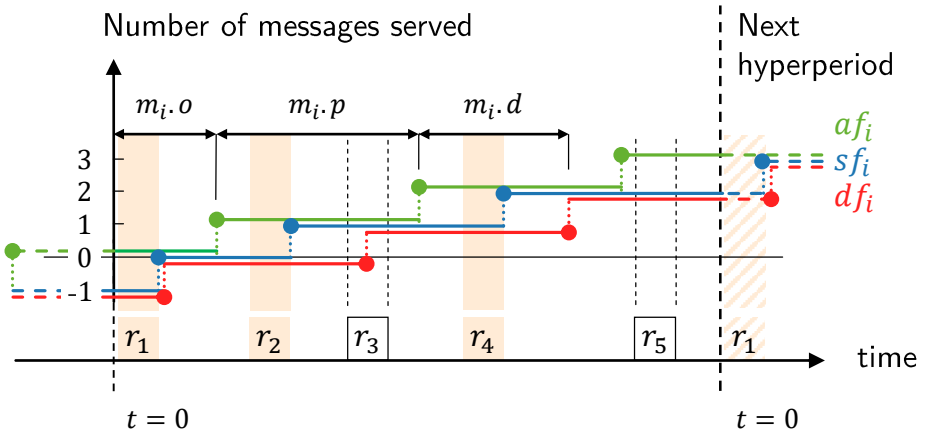


Figure 5.5 Representation of arrival, demand, and service functions of message m_i. The lower part shows the five round, r_1 to r_5, scheduled for the hyperperiod. m_i *is allocated a slot in the colored rounds,* i.e., r_1, r_2, *and* r_4. *The allocation of* m_i *to* r_3 *instead of* r_2 *would be invalid, as* r_3 *does not finish before the message deadline,* i.e., *it violates* **(C2)**. *However, the allocation of* m_i *to* r_5 *instead of* r_1 *would be valid and result in* $r_0.B_i = 0$.

denotes the allocation of the s-th slot of r_k (*i.e.*, the id of the message allocated to the slot). For any time $t \in [\, r_j.t+T_{r_j} \,;\, r_{j+1}.t+T_{r_j} \,[$, the number of instances of message m_i served is

$$\sum_{k=1}^{j} \sum_{s=1}^{B} r_k.B_s \quad s.t.\ B_s = i$$

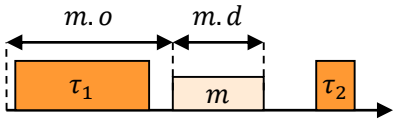
It may be that $m.o+m.d > m.p$, resulting in $df(0) = -1$ (Equation (5.3)), like it is the case in Figure 5.5. This "means" that a message released at the each of one hyperperiod will have its deadline in the *next* hyperperiod. To account for this situation, we introduce, for each message m_i, a variable $r_0.B_i$ set to the number of such "leftover" message instances at $t = 0$. Finally, for each message $m_i \in \mathcal{M}$, and $t \in [\, r_j.t + T_{r_j} \,;\, r_{j+1}.t + T_{r_j} \,[$,

$$sf_i : t \longmapsto \sum_{\substack{k=1 \\ s.t.\ r_k.t+T_{r_k} < t}}^{j} \sum_{\substack{s=1 \\ s.t.\ B_s=i}}^{B} r_k.B_s - r_0.B_i \tag{5.10}$$

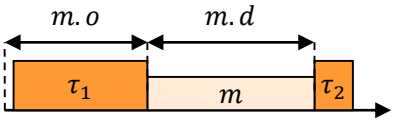
Ultimately, **(C1)** and **(C2)** can be formulated as MILP constraints using Equations (5.8) and (5.9), and the following two equations:

$$(5.4) \quad \Leftrightarrow \quad \sum_{k=1}^{j} \sum_{\substack{s=1 \\ s.t.\ B_s=i}}^{B} r_k.B_s - r_0.B_i \ \leq\ k_{ij}^{a} \tag{5.11}$$

$$(5.5) \quad \Leftrightarrow \quad \sum_{k=1}^{j-1} \sum_{\substack{s=1 \\ s.t.\ B_s=i}}^{B} r_k.B_s - r_0.B_i \ \geq\ k_{ij}^{d} \tag{5.12}$$



(a) Example schedule without deadline maximization.



(b) Example schedule with deadline maximization

Figure 5.6 Illustration of the impact of the message deadlines maximization. *If Figure 5.6a is a valid schedule, then Figure 5.6b is also valid, but it relaxes the constraints on other modes which also contain message m. Maximizing the message deadlines improves the schedulability of the multi-mode problem (Section 5.5).*

Objective function. Within our scheduling framework, the MILP does not need to optimize any objective function. Indeed, we mainly want to minimize of the number of rounds R used in the schedule, which is achieved by incrementally increasing the number of rounds until a valid schedule is found (Algorithm 1).

However, when considering the multi-mode case (Section 5.5), it is beneficial to maximize the message deadlines, as illustrated in Figure 5.6. In a nutshell, it relaxes the constraints that are inherited between different modes, and therefore improve the schedulability of the whole problem. Concretely, the deadline maximization is achieved by setting the following objective to the MILP solver

$$obj = \sum_{m_i \in \mathcal{M}} m_i.d \tag{5.13}$$

5.5 Synthesis of Compatible Multi-Mode Schedules

TTW statically co-schedules all tasks and messages in order to satisfy tight deadline constraints (Section 5.4). To preserve a certain degree of adaptability at runtime, we support multiple operation modes. Doing so requires to ensure predictable mode switches; that is, applications always meet their end-to-end deadlines **(O1)** and the persistent applications have compatible schedules in different modes **(O2)**.

The multi-mode case is essentially a multi-objective problem. One could decide to minimize the overall number of rounds used (*i.e.*, the sum of rounds in all the modes); however, it might also be interesting to optimize the "most common mode"; this is, the mode in which the system operates most of the time. Ultimately, one must weight the different modes to define a globally optimal solution.

Instead of solving the entire multi-mode problem at once, which would have scalability issues, we solve the problem sequentially: one mode at a time, in order of increasing priority. However, ensuring schedule compatibility between the different modes **(O2)** creates dependencies, as illustrated below.

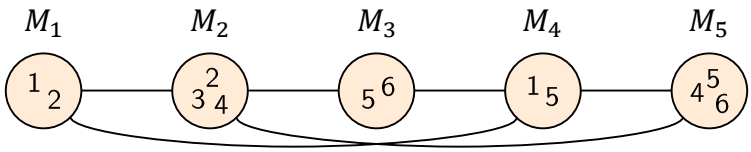
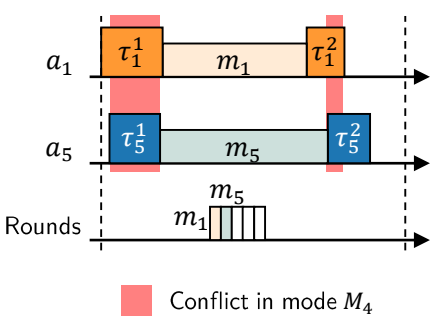
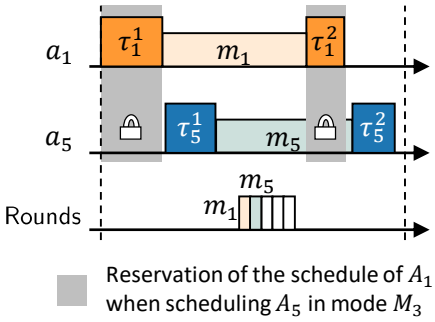


Figure 5.7 The mode graph $\mathbb{M}$ discussed in Examples 5.1 and 5.2. *Five modes are represented by circles, the possible transitions between modes as arcs. Six applications* a_1 *to* a_6 *are specified. The specification is shown with the numbers in the circles;* e.g., $S_1 = \{a_1, a_2\}$. *Mode* M_i *has priority* i.



(a) a_5 is scheduled in mode M_3 without considering the previously computed schedule of a_1, which leads to a conflict in mode M_4.



(b) a_5 is scheduled in mode M_3 considering the schedule of a_1 as *reserved*. Thus, a compatible schedule for a_5 is computed, which prevents conflicts due to schedule inheritance in mode M_4.

Figure 5.8 Representations of the schedule of applications a_1 and a_5 from Example 5.1. For the sake of illustration, we consider that all tasks are mapped to the same node. a_1 and a_5 are scheduled respectively in mode M_1 and M_3, and must both be inherited in mode M_4. In Figure 5.8a, overlapping task schedules result in a conflict, while in Figure 5.8b, it was prevented by reserving a_1's schedule. The situation is different for the messages: overlapping message schedules is no issue, it simply represents a time interval where both messages can be served during the same round, as shown in this example.

Example 5.1. Let us consider the mode graph in Figure 5.7 and assume that all applications are persistent. The modes are scheduled sequentially, starting with the highest-priority mode M_1, which is freely scheduled. When mode M_2 is scheduled, the schedule for application a_2 is inherited from mode M_1 **(O2)** and the schedules for applications a_3 and a_4 are synthesized without constraints. In M_3, the specified applications, a_5 and a_6, are new and can be scheduled without constraints. Then, in mode M_4, the specified applications, a_1 and a_5, have both been previously scheduled and thus must be inherited **(O2)**. However, as mode M_3 has been scheduled without constraint, the schedule synthesized for a_5 may be non-compatible with that of a_1 from mode M_1. This leads to a conflict in M_4 and thus renders the sequential synthesis of the multi-mode problem unfeasible (illustrated in Figure 5.8).

As illustrated in Example 5.1, it may be necessary to "reserve the space" of

previously scheduled applications (*i.e.*, in previous modes) in order to avoid schedule conflicts. The simplest approach is to reserve the space of *all previous scheduled applications*. This is definitely safe but often pessimistic, as there may not be risk of conflicts for certain applications.

In this section, we derive the set of schedule reservations that is necessary and sufficient to prevent inheritance conflicts. We first formalize the continuity constraints that we want to satisfy **(O2)** (Section 5.5.1). Then, we characterize conflicting modes and formalize how continuity constraints may lead to conflicts (Section 5.5.2). Finally, we derive the minimally restrictive reservations that are necessary and sufficient to prevent conflicts while satisfying **(O2)** (Section 5.5.3).

5.5.1 Continuity Constraints

The schedule synthesis returns the application schedules, *i.e.*, the task and message offsets and the message deadlines; and the round schedules, *i.e.*, the round starting times and the allocation vector. We abstract an application schedule with a scheduling function s as follows.

Definition 5.1 (Scheduling function)**.** The scheduling function s is defined over the set of applications $\mathcal{A}$ and returns, for a given application a, all the parameters characterizing the schedule of application a. The schedule of an application a is denoted by $s(a)$. The scheduling function is extended to sets of applications as follows.

$$\forall\, S \subset \mathcal{A}\ , \quad s(S) = \bigcup_{a \in S} s(a)$$

$s_M(a)$ denotes the schedule of application a in mode M.

All persistent applications $a \in \mathcal{A}_P$ are subject to continuity constraints, formalized as follows.

Definition 5.2 (Continuity constraint)**.**

$\forall\, a \in \mathcal{A}_P,\ \forall\, (M_i, M_j) \in \mathcal{O}^2$,

$$a \in M_i \ \wedge\ a \in M_j \ \wedge\ \mathbb{M}(M_i, M_j) = 1 \quad \Rightarrow \quad s_{M_i}(a) = s_{M_j}(a) \qquad (5.14)$$

In other words, an executing application must keep the same schedule, regardless of mode changes.

Definition 5.3 (Schedule domains)**.** The schedule domains of an application are the (possibly multiple) subsets of modes in which the application schedule must remain the same.

Corollary 1. *Two modes M_i and M_j belong to the same schedule domain of an application $a \in \mathcal{A}_P$ if and only if*

- *a is scheduled in both modes,* i.e., $a \in M_i \ \wedge \ a \in M_j$, *and*
- *There is a possible transition between the two modes,* i.e., $\mathbb{M}(M_i, M_j) = 1$.

Proof. Multiple modes belong to the same schedule domain because of a continuity constraint. The formalization of the schedule domains directly follows from Definition 5.2 ■

The mode graph can be analyzed to extract the schedule domains of any application. A simple approach entails considering the sub-graph $\mathbb{G}_A$ from $\mathbb{M}$, *i.e.*, where one keeps only the modes in which application a is specified. Every connected component of $\mathbb{G}_A$ is a schedule domain of a.

Hypothesis 2. We consider in the rest of this chapter that (i) all applications are persistent, and (ii) applications have a single scheduling domain.

Hypothesis 2 induces no loss of generality. Indeed, non-persistent applications present in multiple modes can be replaced by distinct applications, with the same parameters, executing in one mode each. Similarly, persistent applications different scheduling domains can be replicated into different applications having one domain each. This is illustrated in the following example.

Example 5.2. Consider again the mode graph in Figure 5.7. Application a_6 has two distinct application domains, $\{M_3\}$ and $\{M_5\}$. It can be equivalently modeled as two distinct applications $a_{6.3}$ and $a_{6.6}$ executing in in M_3 and M_6 respectively.

On the contrary, a_1 has only one schedule domain, $\{M_1, M_4\}$. If a_1 is not persistent, the continuity constraint does not apply (Definition 5.2). Thus, a_1 can be equivalently modeled as two distinct applications $a_{1.1}$ and $a_{1.4}$ executing in M_1 and M_4 respectively.

5.5.2 Characterization of Conflicts

As illustrated in Example 5.1, the continuity constraint may lead to conflicts, leading to the failure of the multi-mode schedule synthesis problem, while a solution could exist. In particular, if a given mode M belongs to the schedule domains of two different applications which have been independently scheduled in higher priority modes, there is a risk of conflict, *i.e.*, the two inherited schedules may be non-compatible. This section formalizes the notions of (virtual) legacies and conflicting modes. $\overline{X}$ denotes the complement of X; *i.e.*, $\overline{X} = \mathcal{A} \setminus X$. For each mode M_i, we define four sets of applications.

Known applications are the applications previously scheduled in higher priority modes. The set of known applications of mode M_i is denoted K_i:

$$K_i = \cup_{j=1}^{i-1} M_j \tag{5.15}$$

Free applications are the newly scheduled applications in mode M_i, *i.e.*, no higher priority mode belongs to the schedule domain of these applications. The set of free applications of mode M_i is denoted F_i:

$$F_i = M_i \cap (\mathcal{A} \setminus K_i) = M_i \cap \overline{K_i} \tag{5.16}$$

Legacy applications are the applications previously scheduled in higher priority modes which must be scheduled in mode M_i. Since we assume a single scheduling domain (Hypothesis 2), M_i necessarily belongs to the same schedule domain as these higher priority modes and the legacy application schedules must be inherited. The set of legacy applications of mode M_i is denoted L_i:

$$L_i = M_i \cap K_i \tag{5.17}$$

Finally, **virtual legacy applications** are the applications previously scheduled in higher priority modes which are not scheduled in mode M_i. The set of virtual legacy applications of mode M_i is denoted VL_i:

$$VL_i = (\mathcal{A} \setminus M_i) \cap K_i = \overline{M_i} \cap K_i \tag{5.18}$$

The virtual legacy applications of M_i are not executed in M_i; they simply have been scheduled in higher-priority modes. As illustrated in Example 5.1, it may be necessary to "reserve the space" of some of these virtual legacy applications in order to avoid future inheritance conflicts.

The schedule of two applications A and B are said in conflict when two tasks from A and B respectively are mapped to the same node and are scheduled during overlapping time intervals. We denote by $s(A) \cap s(B) \neq \emptyset$ the property that "A and B are in conflict".

Definition 5.4 (Conflict-free)**.** A set of applications S is said to be conflict-free when there is no conflict between the schedules of the applications in S. We denote by $CF(S)$ the property that S is conflict-free, and $\overline{CF(S)}$ denotes that the set S is in conflict. Formally,

$$CF(S) \quad \Leftrightarrow \quad \bigcap_{A \in S} s(A) = \emptyset$$

A mode is said to be conflict-free if its legacy applications are conflict-free. In other words, $\forall M_i \in \mathcal{O}$,

$$CF(M_i) \quad \Leftrightarrow \quad CF(L_i)$$

The schedule $Sched(M_i)$ of mode M_i is valid only if $CF(M_i)$.

Corollary 2. *A valid schedule for a mode $M_i \in \mathcal{O}$ can only exist if the virtual legacy applications of M_i are conflict-free; that is,*

$$CF(L_i) \;\Leftarrow\; \text{“Sched}(M_i)\text{ is feasible”}$$

Proof. Using Example 5.1 as a counter-example, $\overline{CF(L_4)}$ makes it impossible to derive a valid schedule for M_4. ■

5.5.3 Minimal Inheritance Constraints

The single-mode schedule synthesis algorithm (Algorithm 1) is complete: if the problem is feasible, a valid schedule is found. In particular, the scheduled mode is conflict-free; *i.e.*, $CF(M_i)$. Certain applications are subject to continuity constraints (Section 5.5.1), which are satisfied by fixing the schedules of legacy applications L_i in the MILP formulation for M_i. However, this can lead to a feasible schedule only if $CF(L_i)$ (Corollary 2).

Example 5.1 illustrated that inheriting legacy applications is not sufficient to prevent conflicts. Thus, we now derive the subset of the virtual legacy applications VL_i of a M_i that is necessary and sufficient to reserve in order to guarantee the absence of conflict due to continuity constraints. In other words, the objective is that for any mode M_i,

$$\forall\, k \in [1..i-1],\ \text{“}Sched(M_k)\text{ is feasible”} \quad \Rightarrow \quad CF(L_i) \tag{5.19}$$

First, we formalize the constraints on the $Sched()$ function such that continuity constraints are enforced and conflicts are prevented.

$$\begin{aligned} Sched : \mathcal{O} \;&\longmapsto\; Sched(M) \\ s.t.\quad & CF(M_i) \\ & \forall\, a \in L_i \cap M_j,\ j < i,\ s_{M_i}(a) = s_{M_j}(a) \\ & \forall\, a \in F_i,\ s(a) \;\cap\; s(\widetilde{VL_i^{\,a}}) = \emptyset \end{aligned} \tag{5.20}$$

In Equation (5.20), the first constraint $CF(M_i)$ is necessary for the schedule to be valid. The second enforces the continuity constraints of applications. Finally the third constraint aims to enforce Equation (5.19). The idea is that the newly scheduled application in mode M_i, *i.e.*, $a \in F_i$, should be compatible with the schedules of some virtual legacies. The objective is to derive the minimal sets $\widetilde{VL_i^{\,a}}$ for any $a \in F_i$ such that condition Equation (5.19) is satisfied.

Theorem 5.1 (Minimal virtual legacy sets)**.** *For any mode M_i and any application a in F_i, the minimal set of virtual legacy applications $\widetilde{VL_i^{\,a}}$ necessary and sufficient to satisfy* (5.19) *is given by*

$$\widetilde{VL_i^{\,a}} = \{X \in VL_i \mid \exists\, j > i, \quad a \in L_j \;\wedge\; X \in L_j\} \tag{5.21}$$

Proof. We first prove that virtual legacy sets as defined in (5.21) are sufficient to satisfy (5.19). This is done by recurrence.

For the highest priority mode M_1, by definition, $L_1 = \emptyset$, thus $CF(L_1)$. Let us assume that for any $k \in [1..i]$, $Sched(M_k)$ is feasible in the sense of (5.20). This induces that $CF(S_k)$, hence $CF(L_k)$ for any $k \in [1..i]$. Let us finally assume that L_{i+1} is *not* conflict-free; that is,

$$\overline{CF(L_{i+1})} \quad \Leftrightarrow \quad \bigcap_{A \in L_{i+1}} s(A) \neq \emptyset \tag{5.22}$$

Therefore,

$$(5.22) \quad \Rightarrow \quad \exists\, (A,B) \in {L_{i+1}}^2,\ s(A) \cap s(B) \neq \emptyset \tag{5.23}$$

$$\Rightarrow \quad \begin{cases} \exists!\, M_a,\ A \in F_a\ \wedge\ a < i+1 \\ \exists!\, M_b,\ B \in F_b\ \wedge\ b < i+1 \end{cases} \tag{5.24}$$

where $\exists!$ means "there exists a unique". Without loss of generality, we consider $a \leq b$. If $a = b$, then $S_a = S_b$ and $CF(S_a) \equiv CF(S_b)$. Therefore,

$$\bigcap_{A \in S_a = S_b} s(A) = \emptyset \quad \text{in particular} \tag{5.25}$$

$$\Rightarrow \quad s(A) \cap s(B) = \emptyset \quad \text{which contradicts (5.23)} \tag{5.26}$$

$$\Rightarrow \quad a < b \tag{5.27}$$

In other words, mode M_a has higher priority than mode M_b. Therefore, a belongs either to L_b or VL_b by definition of those sets. By hypothesis, $CF(S_b)$ and (5.24) : $B \in F_b$, thus

$$A \in L_b \quad \Rightarrow \quad s(A) \cap s(B) = \emptyset$$

which contradicts (5.23). Hence necessarily, $A \in VL_b$. Furthermore,

$$\left.\begin{array}{l} (5.24) : i+1 > b \\ (5.23) : A \in L_{i+1} \\ (5.23) : B \in L_{i+1} \end{array}\right\} \text{ Taking } b = i \text{ and } j = i+1, \quad (5.21)\ :\ A \in \widetilde{VL_b}^B \tag{5.28}$$

By hypothesis, $Sched(M_b)$ is feasible, thus $s(B) \cap s(\widetilde{VL_b}^B) = \emptyset$, which yields $s(B) \cap s(A) = \emptyset$ and contradicts (5.23) again. Therefore, the recurrence hypothesis is necessarily false. Hence, if for any $k \in [1..i]$, $Sched(M_k)$ is feasible in the sense of (5.20), then $CF(L_{i+1})$. By recurrence, we can conclude that the *virtual legacy sets as defined by* (5.21) *are sufficient* to satisfy (5.19).

We now prove they are also necessary. Let us consider smaller virtual legacy sets than defined by (5.21), *i.e.*, $\exists\, i \in [1..M],\ a \in F_i,\ \ \widehat{VL_i}^A \subsetneq \widetilde{VL_i}^A$. Let us further assume that $Sched()$ is redefined to replace $\widetilde{VL}$ by $\widehat{VL}$. By hypothesis,

$$\exists\, X \in \mathcal{A},\ X \in \widetilde{VL_i}^A\ \wedge\ X \notin \widehat{VL_i}^A \tag{5.29}$$

Furthermore,

$$X \in \widetilde{VL_i}^A \quad \Rightarrow \quad X \in VL_i \quad \Rightarrow \quad X \notin S_i \tag{5.30}$$

$$X \in \widetilde{VL_i}^A \quad \Rightarrow \quad \exists\, j > i, \quad a \in L_j \;\wedge\; X \in L_j \tag{5.31}$$

Assuming that $Sched(M_i)$ is feasible, the resulting schedule guarantees that

$$\begin{aligned} & CF(M_i) \\ \text{and} \qquad & \forall\, a \in F_i,\; s(a) \cap s(\widehat{VL_i}^A) = \emptyset \end{aligned} \tag{5.32}$$

However, (5.29) : $X \notin \widehat{VL_i}^A$ and (5.30) : $X \notin S_i$. Hence the schedule $s(a)$ may be synthesized such that $s(A) \cap s(X) \neq \emptyset$. According to (5.31) : $(A, X) \in {L_j}^2$, thus this induces a conflict in mode M_j. Hence, we can conclude that no sets $\widehat{VL}$ smaller that $\widetilde{VL}$ are sufficient to satisfy (5.19).

Therefore, one concludes that the virtual legacy sets $\widetilde{VL}$ as defined in (5.21) are both necessary and sufficient for the schedule synthesis method to satisfy (5.19), *i.e.*, to guarantee that legacy applications schedule inheritance does not lead to conflicts in lower-priority modes. In other words, $\widetilde{VL}$ from (5.21) define the *minimally restrictive constraints sets* such that $Sched()$ as defined in (5.20) satisfies (5.19). ■

5.6 Changing Mode at Runtime

TTW's *Adaptability* relies on switching between different operation modes at runtime. The multi-mode schedule synthesis procedure described in the previous section guarantees that the computed schedules are compatible; that is, persistent applications have the same schedule in different modes (Section 5.5). In this section, we describe the procedure implemented in *TTW* to perform the mode changes at runtime.

TTW controls mode changes using the beacons, sent by the host at the beginning of each round. A beacon contains three elements: the current round id, a mode id, and a trigger bit TB. Modes and rounds have unique ids with a known mapping of the rounds to the modes. By default, a beacon includes the current mode id and the trigger bit is 0. A mode change happens in two phases: First, the change is announced by a beacon including the new mode id (instead of the current one). In a later round, the TB is set to 1, which triggers the mode change; the new mode starts at the end of the round where the TB is set. This two step procedure (illustrated in Figure 5.9) lets the nodes prepare for an upcoming mode change.[4] Let us denote a beacon by b and

[4] *E.g.*, stopping the execution of applications that will be discontinued in the new mode.

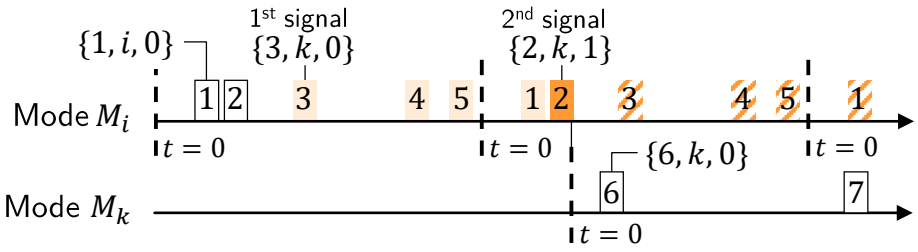


Figure 5.9 Example of mode change from mode M_i to M_k. *The dashed lines show the start of the mode hyperperiods. Each numbered box represents a communication round with its id. The content of some of the host beacons is shown next to the corresponding rounds. In r_3, the host sends the first signal to change to mode M_k. During this transition phase, the rounds are lightly colored. In the dark colored round, the host sets $TB = 1$. This is the second signal – directly after this round, mode M_k starts executing. Dashed rounds are not executed.*

assume $b = \{j, i, 0\}$. If round r_j (the round with $id = j$) is mapped to mode M_i, no mode change is announced: the next round is round r_{j+1} (in the cyclic sequence of rounds associated to mode M_i). When $b = \{j, k, 0\}$ is sent, the mode change from mode M_i to M_k is announced, but the next round remains based on mode M_i schedule. Finally, $b = \{j, k, 1\}$ is sent, which triggers the mode change: the next round is the first round of mode M_k.

Many mode change protocols have been proposed in the literature (see [48] for an overview). These protocols define the expected behavior of applications in the old and new modes; for example, they specify whether the running applications should be left enough time to complete, or be interrupted by the mode change. *TTW* does not define a specific mode change protocol. Instead, it provides a general strategy to execute mode changes and lets the user define the desired mode change protocol and implement it at the application level.

Example 5.3. Let us consider again the feedback control application from [122]. In this system, we implement a simple mode change protocol: all applications keep running until they are interrupted by a mode change. When a mode change is announced, a counter (initialized to 10) is appended to the beacon, and decremented with every round. The mode change is triggered (*i.e.*, TB is set to 1) when the countdown reaches zero. With this approach, it is sufficient for any node to receive only one out of ten beacons to change mode at the correct time. Since the probability of reception of a packet is expected to be above 99% [70], receiving at least one out of ten beacons will happen with very high probability . Thus, this mode change protocol guarantees with very high probability that all the nodes in the system are always executing in the same mode, even in the occurrence of sporadic packet losses; this guarantees conflict-free communication across mode changes.

Table 5.2 Worst-case execution time of *Bolt* `read` and `write` functions

Payload [bytes]	`read` [μs]	`write` [μs]
L	$102+2\times L$	$108+2\times L$
8	118	124
16	134	140
64	230	236

5.7 Implementing *TTW*

TTW is a wireless CPS design composed of two main building block: (i) the system's communication backbone, which we call *TTnet* (Section 5.2), and (ii) the *TTW* scheduler, which synthesizes the timing of execution of all tasks, messages, and communication rounds (Sections 5.4 and 5.5). In this section, we describe how we implement *TTnet* and the *TTW* scheduler to obtain a fully functional *TTW* system. The performance of the implementation is discussed in Sections 5.8 and 5.9.

5.7.1 Implementation of *TTnet*

We implement *TTnet* on embedded platforms built using the Dual-Processor Platform (*DPP*) concept. The *DPP* links two arbitrary processors with a processor interconnect called *Bolt* [174], which provides predictable asynchronous message passing between the two processors using message queues with first-in-first-out (FIFO) semantics, one for each direction.[5] The *DPP* architecture is compatible with *TTW*'s system model, which assumes that a node is capable of performing one task execution and one message transmission simultaneously (Section 5.3): We dedicate one processor (a TI MSP432 [181]) to the execution of tasks and another one (a TI CC430 SoC [180]) to wireless communication. The *DPP* platform used for our implementation is illustrated in Figure 1.8.

To implement the *TTnet* network stack, we leverage the *Baloo* design framework (introduced in Chapter 3). In particular, we use the "static configuration" mode of *Baloo* (Section 3.4): *TTW* produces scheduling tables offline, which can be loaded in the nodes' memory to limit the runtime communication overhead (*Efficiency*). The *TTnet* beacons (Section 5.2) are sent as *Baloo* control packets, using the customizable "user-bytes" field.[6]

To synthesize the schedules, one must know how long a communication round lasts. For this purpose, the predictability of the *Baloo* framework is very beneficial: based on the implementation parameters, one can derive a precise

[5] Refer to the Introduction chapter for more details about the *DPP* concept.
[6] github.com/ETHZ-TEC/Baloo/wiki/Baloo-control-packet

estimate of the maximum possible length of a round.[7] This model must account for the time to read and write messages over *Bolt*, the *DPP* processor interconnect. The *Bolt* API functions have formally verified semantics and bounded execution times [174]. In our implementation, the two processors read and write over *Bolt* using the maximally supported SPI frequency of 4 MHz, leading to the worst-case execution times shown in Table 5.2. On the application side, we include the time to read and write messages within the WCET of tasks; in other words, we consider that a task "starts" when the processor initiates the *Bolt* `read` function, and terminates once the *Bolt* `write` function is completed.[8] On the communication side, incoming messages are read from *Bolt* before the rounds. This is performed during the so-called "pre-process", which is scheduled before each *Baloo* round.[9] The messages received from the wireless network are written over *Bolt* directly after the communication slot, in *Baloo*'s `on_slot_post()` callback function (Section 3.3). Our implementation of *TTnet* is available in the *Baloo* repository [98].

5.7.2 Implementation of the *TTW* Scheduler

We implement the *TTW* scheduler using Matlab [126] and we use the Gurobi solver [83] for synthesizing the schedules. All scripts are publicly available (Section 5.A).[10] The details of the MILP formulation can be found in Appendix (Section 5.A). In this section, we describe how to run the *TTW* scheduler.

There are four high-level communication parameters that the user must specify in the `multimode_main.m` file:

- L The message payload size (in bytes)
- B_{max} The maximal number of slots per round
- N The number of message transmissions in a Glossy flood [70]
- H The estimated network diameter (in number of hops)

The `loadRoundModel.m` file contains the parameters of the *TTnet* implementation (Section 5.7.1). Combined with the high-level parameters described above, this allows the scheduler to compute the length of communication rounds.[7] Finally, the user must specify the system's operation modes, applications, tasks, and messages, which is done by filling cell arrays in the `system_configuration.m` and `mode_configuration.m` files.

The scheduler supports user-defined constraints on the task and message offsets.

[7] The *TTnet* model and its evaluation are presented in Section 5.9.

[8] Assuming that the task has some input and output messages.

[9] github.com/ETHZ-TEC/Baloo/wiki/Baloo-pre-post-processes

[10] Although Matlab and Gurobi are commercial software, free academic and/or student licenses are currently available from the software vendors.

```
CustomConstaints = { ...
    { {{'T_loc_stab1', 1},{'T_loc_stab2', -1}}, '=', 0}, ...
    { {{'T_loc_stab1', 1},{'T_loc_stab3', -1}}, '=', 0}, ...
    { {{'T_loc_stab1', 1},{'T_loc_stab4', -1}}, '=', 0}, ...
    { {{'T_loc_stab1', 1},{'T_loc_stab5', -1}}, '=', 0}, ...
    };
```

Figure 5.10 Example of specification of user-defined constraints. *The generic formulation let the user define any linear constraint between the offsets of tasks and messages in the system. In this example, the constraints force all tasks* `T_loc_stabX` *to have the same offset; in other words, these tasks will execute simultaneously.*

These constraints must have the following form

$$\sum_k \alpha_k * id_k.o \quad `sign' \quad \beta \tag{5.33}$$

where id is the unique identifier of a task or a message, α is a constant multiplier, $sign$ can be specified as '$=$' or '$<$', and β is the right-hand side term of the constraint. The constraint may contain an arbitrary number of left-hand terms, denoted here by k. These constraints are specified as tuples of the form $\{\{id_k, \alpha_k\}_k, sign, \beta\}$, as shown in Figure 5.10; in that example, the user-defined constraints force some tasks to have the same offset; in other words, these tasks will execute simultaneously. The user-defined constraints are automatically added to the MILP formulation; if the problem is feasible, the schedule is guaranteed to satisfy them. Once all inputs have been specified, the schedules for all operation modes are synthesized and displayed by running the `multimode_main.m` script.

5.8 Performance of the *TTW* Scheduler

The following two sections present the performance evaluation of our *TTW* implementation, presented in Section 5.7. We first evaluate the performance of the scheduler. In particular, we illustrate the benefits of the minimal inheritance strategy presented in Section 5.5 and we show that the complexity of the schedule synthesis is tractable.

5.8.1 Benefits of Minimal Inheritance

Every round introduces some overhead (mainly from sending the beacon), which consumes energy. To reduce the energy consumption, *TTW* aims to minimize the number of rounds **(O1)**. The schedule synthesis for a single mode is optimal in this respect; that is, the procedure guarantees that the schedule minimizes the number of communication rounds (Section 5.4).

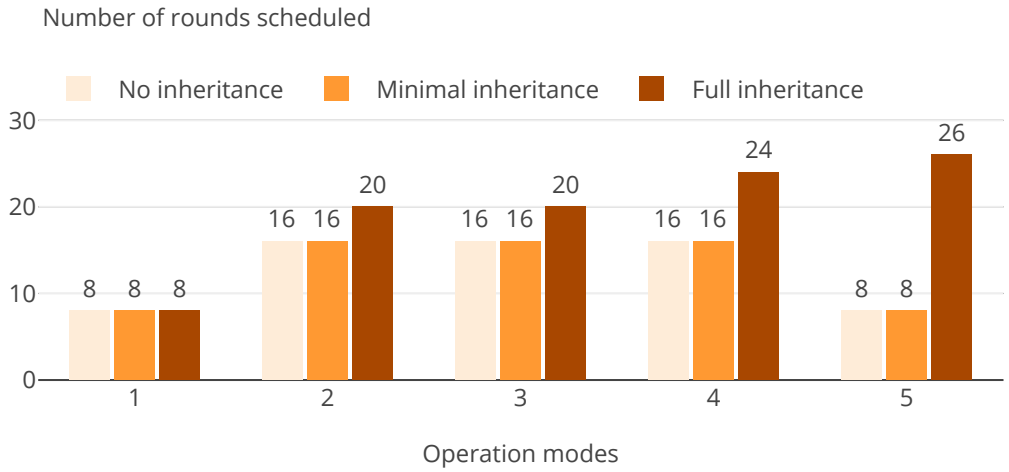


Figure 5.11 Number of rounds in the different modes' schedule, depending on the inheritance approach considered. *We consider the number of rounds scheduled over 80 s, which is the least common multiple of the modes' hyperperiod.*

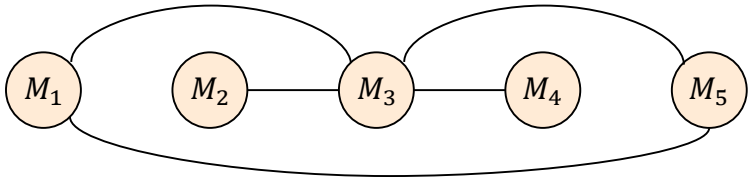


Figure 5.12 The mode graph $\mathbb{M}$' used in the inheritance evaluation scenario. Mode M_i has priority i. The applications executing in each mode are listed in Section 5.B.

The second objective of the scheduler is to allow persistent applications to keep the same schedule in different operation modes **(O2)**. This creates additional constraints that break the optimality guarantee: in other words, the schedule of mode M_j, when constrained to be compatible with mode M_i, may contain more rounds that required to schedule the mode M_j alone. A naive solution to meet **(O2)** is to completely "reserve the space" of previously scheduled modes. This is equivalent to consider that all applications executing in mode M_i are also executing in M_j, even if it is not actually the case. We call this the *full inheritance* approach. This approach does guarantee compatibility but it is very pessimistic: it leads to an excessive increase of the number of rounds and find problems to be non-schedulable when they may in fact be feasible.

In Section 5.5, we derived the minimal set of constraints that are necessary to guarantee the compatibility between modes **(O2)**, which we refer to as *minimal inheritance*. We now illustrate with a simple example that the minimal inheritance does not overly increase the number of communication round required and performs much better than the full inheritance. We consider the following configuration (fully detailed in Section 5.B): The system is composed of 13 nodes, running 15 different applications including 45 tasks

and 30 messages. The periods and deadlines vary between 10 and 80 s. The applications are executing in 5 different modes connected by the mode graph shown in Figure 5.12. Finally, all applications are considered persistent. We synthesize the schedule for the 5 modes while considering (i) no inheritance,[11] (ii) our minimal inheritance approach, and (iii) the naive full inheritance approach. The results are shown in Figure 5.11.

One important observation is that the number of rounds steadily increases with the full inheritance approach, which is expected: the full inheritance assumes that all previously scheduled applications are still executing. Thus, the number of applications to schedule only increases, and so does the number of round required. Ultimately, this not only wastes energy, it also limits scalability with the number of operation modes. In comparison, our minimal inheritance approach performs much better: Since only the required constraints are included, the minimal inheritance does not suffer from the scalability issue mentioned above. In this example, the minimal inheritance performs optimally (*i.e.*, it does not schedule more rounds that the minimum, captured by the "no inheritance" case); but note that this is **not true in general**, it simply happens to be the case in this example.

Conclusion. The minimal inheritance approach derived in Section 5.5 efficiently addresses the challenge of synthesizing compatible schedules **(O2)** while minimizing the energy impact in terms of number of rounds scheduled **(O1)**. This approach does increase the complexity of the synthesis formulation; however, it is implemented in our *TTW* scheduler, it induces no overhead for the user, and it does not affect the synthesis solving time, as discussed below.

5.8.2 Offline Solving Time

We computational complexity of the schedule synthesis is made tractable by *TTW*'s sequential approach: modes are scheduled individually, in order of priority (Section 5.5) and for each mode, the number of rounds to schedule is kept fixed then incremented until a solution is found (Section 5.4).

For the evaluation scenario described above, the solving time for one mode grows up to ten minutes (Table 5.3) and is generally correlated with the complexity of the mode to schedule: mode M_3 and M_4 contains the most applications (Section 5.B), leading to more constraints in the formulation. Furthermore, we note that the minimal inheritance strategy does not increase the overall solving time compared to "no inheritance". The intuition is that, by reserving some applications' schedule, we fix the value of some of the problem variables, thereby reducing the complexity of the problem. However, as shown

[11] Considering no inheritance is equivalent to set all applications as non-persistent. In other words, there are no constraints between the different modes and the individual mode schedules are guaranteed to be optimal in terms of number of rounds (Section 5.4).

Table 5.3 Approximate solving time for the different modes of the inheritance evaluation (Section 5.8). *Time expressed in seconds; all computation performed on a commodity laptop.*

	M_1	M_2	M_3	M_4	M_5
No inheritance	7	≈ 0	184	536	≈ 0
Minimal inheritance	6	≈ 0	114	438	≈ 0
Full inheritance	3	79	65	231	578

by the full inheritance approach, if too many variables are fixed, the resulting problem might become harder to solve: more communication rounds become required, which increases the number of variables and thus the complexity.

Conclusion. The evaluation scenario is simple but representative of a middle-sized CPS. Our evaluation shows that the computational complexity may grow to the scale of minutes for challenging modes, which remains perfectly tractable for a task that needs to be performed only once and before deployment.

5.9 Performance of *TTnet*

After the evaluation of the *TTW* scheduler (Section 5.8), we now consider the performance of our *TTnet* implementation, described in Section 5.7.1.

5.9.1 Memory Utilization

First, we consider the memory utilization induced by storing the scheduling tables in the nodes' memory. For a given operation mode, the entire schedule contains the task offsets, the message offsets and deadlines, the round starting times, and the allocations of messages to rounds (Table 5.1). In addition, nodes must know the task periods and the mode hyperperiod to compute the absolute start time of the tasks and rounds.

Since we dedicate the execution of tasks and the wireless communication to different processors (Section 5.7), the memory cost for storing the schedule can be splitted. On the application side, we must store the task offsets and periods, which are required to know when to execute the tasks; *i.e.*, 2 variables per task. On the communication side, we must store the mode hyperperiod and the rounds information, *i.e.*, the offset and allocation of the rounds scheduled within the mode's hyperperiod; *i.e.*, $(B_{max} + 1)$ variables per round. The message deadlines are not required at runtime and do not need being stored. As a result, we can generally estimate that the scheduling tables represent tens to hundreds of variables per mode for each processor.

Conclusion. Our application and communication processors feature 64 kB [181] and 4 kB [180] of RAM, respectively. Thus, considering an average size of two bytes per variable, storing the scheduling tables represent a significant overhead and limits the scalability of the system, in particular on the communication processor. This limitation would be significantly relaxed with newer platforms, which commonly feature 256 kB of RAM [162].

5.9.2 *TTnet* Model

As discussed in Section 5.7, we implement *TTnet* using *Baloo*, which allows to derive a precise model of (i) the execution time of a communication round and (ii) the time spent with the radio turned on, which correlates with the energy consumed for communication. Estimating the communication time is necessary to synthesize the schedules since the scheduler must know how long the rounds last. This model should be as tight as possible not to "waste" time and thus minimize the end-to-end deadlines schedulable by *TTW*, but it must be a safe upper-bound in order to prevent deadline misses. This section presents our *TTnet* model and derives the theoretically achievable performance in terms of minimal message latency and the energy savings expected from using rounds.

Let $a.\delta$ denote the latency of an application a. This latency represents the delay for a complete execution of a; that is, the completion of all tasks in $a.\mathbb{P}$. Let $a.c$ be a *chain* in $a.\mathbb{P}$. A chain is defined as a path of $a.\mathbb{P}$ starting with a task without predecessor and ending with a task without successor.[12] The minimum achievable latency for a single message in *TTW* is the length of a round composed of only one slot, denoted $T_r(L,1)$ where L is the payload size. Thus $a.\delta$ is lower-bounded by

$$a.\delta \geq \max_{a.c \in a.\mathbb{P}} \left(\sum_{\tau \in a.c} \tau.e + \sum_{m \in a.c} T_r(L,1) \right) \tag{5.34}$$

Remark 7. By comparison, the best possible guarantee for the latency of a single message provided by *DRP* (Chapter 4) is of the order of $2 * T_r(L, B_{max})$. Since $T_r(L, B_{max}) \approx B_{max} * T_r(L,1)$, *TTW* reduces the minimal guarantee on message latency by a factor of approximately $2 * B_{max}$. For a relatively small number of slots per round, such as $B_{max} = 5$, this represents an order of magnitude improvement. This difference stems from the loose coupling between the task and message schedules in *DRP*, whereas *TTW* statically schedules all tasks and messages.

A round T_r is composed of up to $(B_{max} + 1)$ slots in which Glossy floods [70] are executed. An entire slot completes in time T_{slot}, decomposed into

$$T_{slot} = T_{wake-up} + T_{start} + T_{flood} + T_{gap} \tag{5.35}$$

[12] For example, (τ_2, m_2, τ_4) is a chain of $\mathbb{P}$ in Figure 5.4.

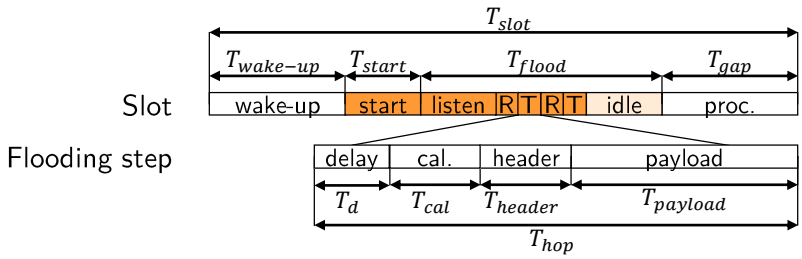


Figure 5.13 Break-down of a communication round. *At the slot level, the colored boxes identify phases where the radio is on. In the "idle" phase, the radio is turned off in practice, but this idle time depends on each node's distance to the initiator. To estimate the energy saving of rounds (Figure 5.15), we assume that the radio stays on for the whole time of T_{flood}, as specified in Equation* (5.40).

The composition of a slot in our implementation is detailed in Figure 5.13. First, all nodes wake up ($T_{wake-up}$) and switch on their radio (T_{start}). Then the message flood starts. We denote by T_{hop} the time required for one protocol step, *i.e.*, a one-hop transmission. The total length of the flood is

$$T_{flood} = (H + 2N - 1) * T_{hop} \tag{5.36}$$

with H the network diameter and N the number of times each node transmits each packet.[13] T_{hop} is itself divided into

$$T_{hop} = T_d + T_{cal} + T_{header} + T_{payload} \tag{5.37}$$

where T_d is a radio delay, and T_{cal}, T_{header} and $T_{payload}$ are the transmission times of the clock calibration message, the Glossy header and the message payload, respectively. With a bit rate of R_{bit}, the transmission of L bytes takes

$$T(L) = 8L/R_{bit} \tag{5.38}$$

Once the flood is completed, some gap time T_{gap} is necessary to process the received packet. This time is used (among other things) to execute *Baloo*'s `on_slot_post()` callback, where the received messages are written into *Bolt*. We divide T_{slot} into T^{on} and T^{off}, which denote the time spent with radio on and off, respectively.

$$T^{off} = T_{wake-up} + T_{gap} \tag{5.39}$$

$$T^{on}(L) = T_{start} + (H + 2N - 1) * (T_d + 8(L_{cal} + L_{header} + L)/R_{bit}) \tag{5.40}$$

$$T_{slot}(L) = T^{off} + T^{on}(L) \tag{5.41}$$

$$T_r(L) = T_{slot}(L_{beacon}) + B * T_{slot}(L) + T_{preprocess} \tag{5.42}$$

Sending beacons is necessary to let the nodes know about the current state of the system; *i.e.*, which mode is executing and "how far" is the system in the

[13] Glossy achieves more than 99.9% packet reception rate using $N = 2$ [70].

Table 5.4 *TriScale* parameters for the experimental validation of *TTnet*'s model

	Evaluation Objectives					
Performance dimension	Metric	Convergence	KPI		Var.Score	
Round length – T_r	max	False	95th	95%	median	75%
Energy savings – E	median	False	5th	95%	median	75%

scheduling table. Without that information, it is impossible for a failing node to recover and resume its normal operation. Moreover, beacons prevent message collisions by guaranteeing that the nodes always know the system's state when a round starts. In a design *without* round, each message transmission should be preceded by its own beacon to provide the same guarantees. Thus, the transmission time for B messages of size L, denoted $T_{wo/r}(L)$, would take

$$T_{wo/r}(L) = B * (T_{slot}(L_{beacon}) + T_{slot}(L)) \tag{5.43}$$

We can then derive the relative energy savings E granted by using a round-based design, which we compute as $E = (T^{on}_{wo/r} - T^{on}_{r})/T^{on}_{wo/r}$.

The complete *TTnet* model is available in Appendix (Section 5.A). We use this model to compute the round length T_r and the energy savings E for different values of number of slots per rounds (B), message payload size (L), network diameter H, and number of transmissions in Glossy floods (N). Selected results are shown in Figures 5.14 and 5.15. For example, with N set to 2, it takes less than 100 ms to complete a 10-slot round sending 16-bytes messages over a 4-hop network (Figure 5.14, top).

5.9.3 Model Validation

We now evaluate the runtime execution of our implementation and aim to validate our *TTnet* model. In particular, it is important that the round length model gives safe upper-bounds since the *TTW* scheduler relies on the model to schedule messages and tasks: if a round overruns, this may delay the execution of subsequent tasks and cause deadline misses. We test our *TTnet* implementation for different number of slots per round B and payload size L, we measure the round length and radio-on time experienced by the different nodes in the network, and we compare the results with the *TTnet* model.

Evaluation scenario. We program the network to execute, one round with B slots, followed by B rounds with one slot. For each of these rounds, we collect the round length and the radio-on time. Both values are measured in software (*i.e.*, the measurement is implemented in the firmware) and use a 32 kHz timer, leading to a measurement accuracy of about 30 μs.

Experiment design. We design the evaluation using the *TriScale* framework (introduced in Chapter 2). The evaluation parameters are listed in Table 5.4.

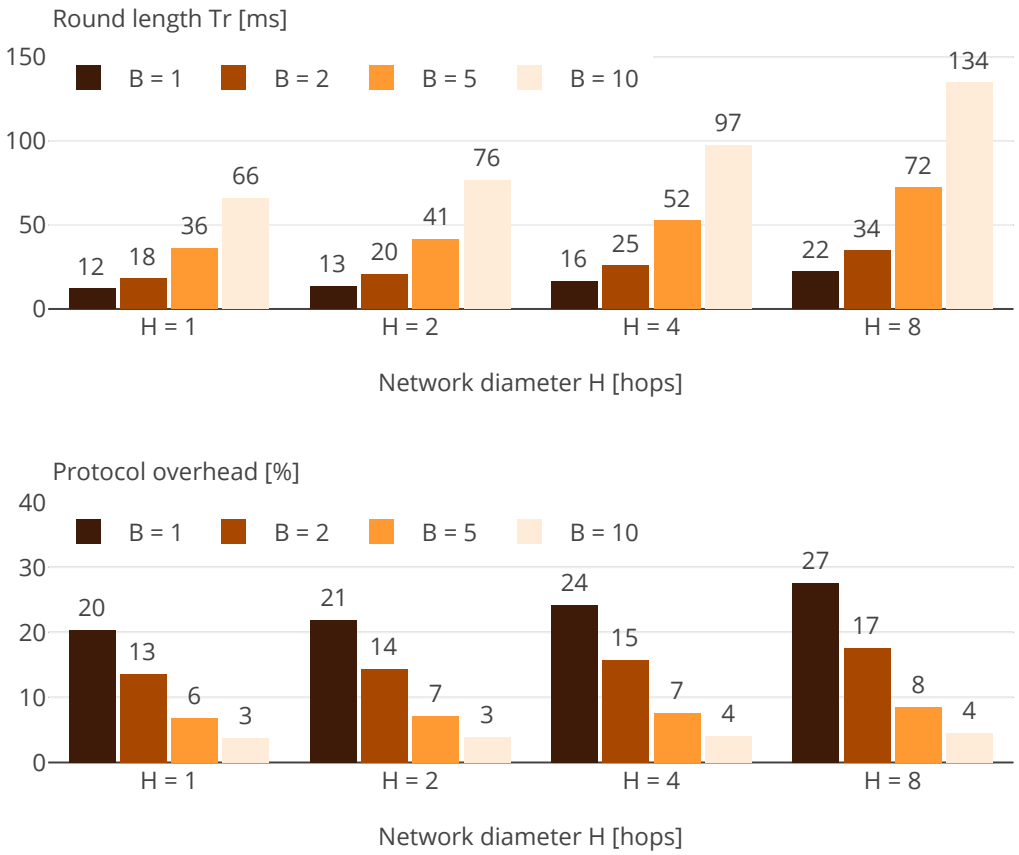


Figure 5.14 Example values of round length (top) and protocol overhead (bottom) computed using the *TTnet* model (Section 5.9.2). The protocol overhead is computed as the percentage of time spent to send the beacons relative to the overall communication time for a round containing B slots. Payload is set to 16 bytes and we use N =2 transmissions in the Glossy floods [70].

Our evaluation scenario is terminating (there is a finite task to accomplish); thus there is no need to test for convergence. The round length evaluation aims to validate that the *TTnet* model is a safe upper-bound; thus, we use the maximum measured round length across all nodes as metric for a run. For the same reason, we choose a large KPI (95th percentile) and a high confidence level (95%), which leads to a minimal number of 59 runs per series. To investigate the reproducibility of the results, we choose the median and a 75% level of confidence for the variability scores, leading to a minimal number or 3 series. To evaluate the average savings provided by using communication rounds, we use the median values across nodes as metric. We perform the evaluation on FlockLab [3], an indoor testbed located in an office building. It has been shown that the experimental conditions on FlockLab exhibits weekly seasonal components (Section 2.4.6); therefore, to avoid biasing our evaluation, we perform our series of runs using a span of one week, during which we schedule randomly 60 runs per set of parameters. We test our *TTnet* implementation

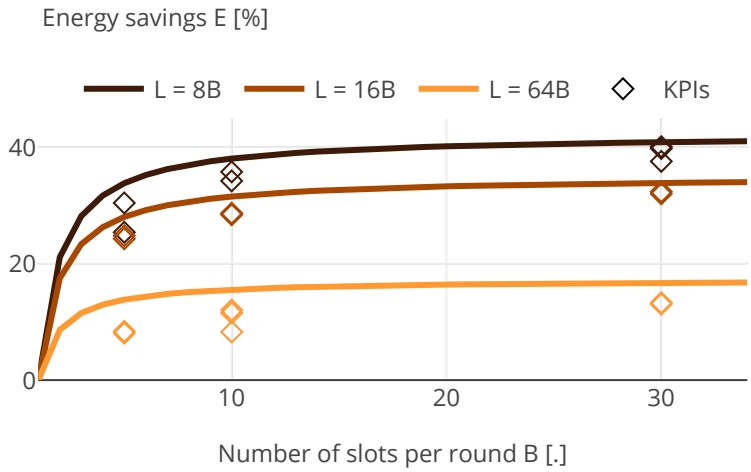


Figure 5.15 Relative radio-on time savings by using rounds compared to single messages. *The energy savings induced by a round-based design grow with the number of number of slots per round (X-axis). Conversely, these savings become less significant as the payload size increases (lighter colors). The diamonds show our evaluation KPIs and thus estimate, with a probability of 95%, the average energy savings expected in 95% of the test runs.* $H = 4$, $N = 2$.

using 5, 10, and 30 slots per round, and payloads of 8, 16, and 64 bytes. The three series of tests were performed between May and October 2019. The KPI values from our evaluation are listed in Table 5.5.

Remark 8. Observe that certain values in Table 5.5 are reported with a * or † symbol. The * marks series where *TriScale* independence test fails; this indicates that the metric data do not appear to be *i.i.d.* and therefore the KPI value loose its predictive power (*i.e.*, it does not allow to infer what is the expected performance). However, in our evaluation, the autocorrelation plots show no significant differences between the series that passes the test and those that do not (data available in Section 5.A). Moreover, the round length KPI values are almost the same in all series. Together, these two facts increase our confidence in the results and suggests that the reported KPIs are robust estimates of the expected performance.
The † marks series where we could not collect enough data in order to compute the KPIs. This appended in Series 3 due to construction work taking place in the FlockLab building, where many test runs were lost due to sporadic power outages. In both cases, the table shows the maximum round length or minimum energy savings metric values obtained across all the runs in the series.

Results – Round length. The results for the round length are extremely stables (Table 5.5): the differences of KPI values between series are at most one time tick ($\approx 30\,\mu$s), which is our measurement accuracy. Concretely, this means that, in all series, the largest round length measured by any node is essentially the same.

Furthermore, the KPI values are (i) very close to and (ii) consistently lower than the model. By definitions of the KPI, we can estimate with 95% probability that (at least) 95% of runs will yield a maximal round length smaller than the KPI value, and thus smaller than the model value. Figure 5.16 (top) shows the distribution of the round length measurements from all the nodes collected during one series of 60 runs. We observe that the distribution is narrow (less than 300 μs of spread), which is expected. Indeed, the *TTnet* rounds are fully time-triggered; thus, the measurement differences between nodes mainly come from the difference in execution time of *Baloo*'s end-of-round operations, which is expected to be small.

In our entire evaluation, there was one case where a node reported a value (77.85 ms) larger than the model (77.52 ms). This concerned only one node: in this run[14] the second highest value reported (77.12 ms) was smaller than the model value. It is hard to know a posteriori what may have cause this. However, we argue that this one overshoot is more likely imputable to some sporadic hardware delay than due to a miscalibration of the model.

Results – Energy savings. The energy savings results show more fluctuations than the round length, which is not surprising: (i) the energy model is less precise and (ii) the dynamic interference conditions affect the radio-on time, as nodes may need to keep their radio on for a longer share of T_{flood}. Figure 5.15 shows the model and our energy savings KPIs together. Figure 5.16 (bottom) shows the distribution of radio-on time measurements from all the nodes, collected in one series of 60 runs: nodes experience significant differences in radio-on time during a round. This is expected since nodes terminate a flood as soon as they have transmitter a packet N times, which happens earlier for nodes that are closer to the initiator,

Overall, the energy savings come from the "distribution" of the overhead from sending beacons between the slots. Thus, the more slots (increasing B) and the smaller the slots (decreasing L), the more radio-on time is spared by using rounds. For a payload of 16 bytes, we obtain an average energy savings of about 30% with only 10 slots per round.

Conclusion. We validated the tightness and safeness of *TTnet* round length model, which was found to be an upper-bound of the effective round length for all but one in about 14k measurements collected. Furthermore, we showcased that, even with small beacons (2 bytes in our implementation), a round-based design yields significant reduction of radio-on time, and therefore helps minimizing the overall energy consumption (*Efficiency*).

[14] FlockLab test number 66992; data available in Section 5.A.

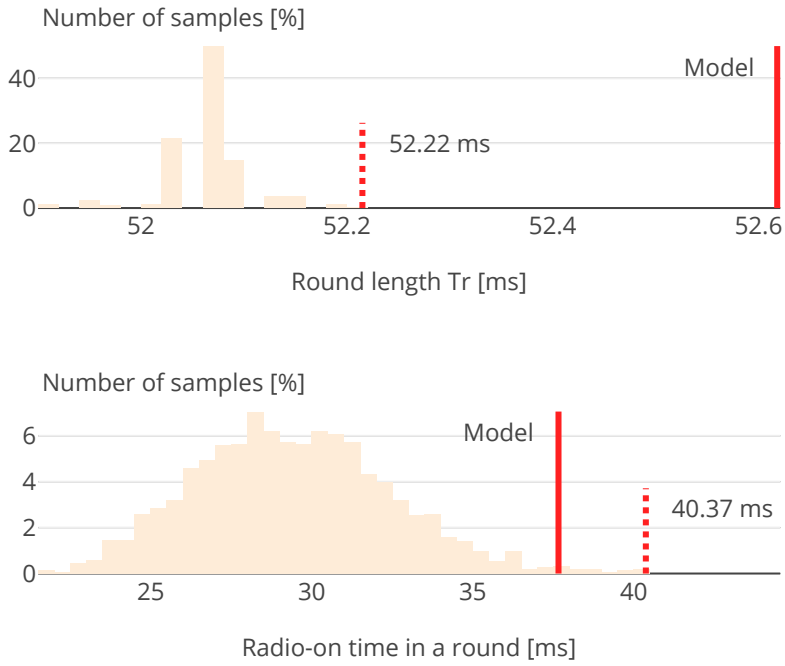


Figure 5.16 Distributions of round length (top) and radio-on time (bottom) measurements from all the nodes, collected in one series of 60 runs (Serie 2, $L = 16$, $B = 5$). *While the distribution of round length is very narrow, the radio-on time exhibits a much larger spread. We can see that the TTnet model provides a generally overestimate the radio-on time, which is expected since it assumes that nodes keep their radio on for the entirety of T_{flood} (Equation* (5.40)*), which is not the case in practice: nodes turn the radio off when they have transmitted a packet N times.*

Table 5.5 KPIs from the performance evaluation described in Section 5.9 and corresponding model values for the *TTnet* round length T_r and energy savings E; other settings: $H = 4$ and $N = 2$. *The value marked in bold corresponds to the one case where the round length KPI is larger than the model value.* * *marks series where TriScale independence test failed.* † *marks series without enough samples for computing the KPIs. In these two cases, reported values are the maximum round length or minimum energy savings metric values for all the runs in the series.*

Payload	**Slots per round**	**Round length** T_r [ms]				**Energy savings** E [%]			
L [bytes]	B [.]	Series 1	Series 2	Series 3	Model	Series 1	Series 2	Series 3	Model
8	5	42.3*	42.3	42.3†	42.52	25	30	29†	34
	10	**77.85**	77.24	77.24†	77.52	34	36	36†	38
	30	217.01*	217.01	217.01*	217.52	37	40*	39*	41
16	5	55.22	52.22	52.22†	52.52	25*	24	22†	28
	10	97.05*	97.08*	97.08†	97.52	28	29*	30†	32
	30	276.52*	276.52*	276.49†	277.52	31	32*	32†	34
64	5	104.77	104.74	104.74†	105.02	8*	8*	9†	14
	10	202.09*	202.12	202.09	202.52	8	12	12	16
	30	591.49	591.52	591.49†	592.52	13*	13*	14†	17

5.10 Discussion, Limitations, and Future Work

Debugging the scheduler is hard. When a system configuration is found unfeasible by the *TTW* scheduler, it is not easy to identify which of the constraints are responsible. Our scheduler implementation would benefit from better built-in support to turn the IIS (irreducible constraint set, returned by the Gurobi solver) into a legible output for the user.

The multi-mode scheduling strategy is not complete. Even with our minimal inheritance approach (Section 5.5), the *TTW* scheduler may find a problem unfeasible even though a solution exists. This is due to the sequential synthesis of schedules, which is a generally under-defined problem: the choices made when synthesizing a mode's schedule may make lower-priority modes unfeasible. Completeness (the guarantee to find a solution if one exists) can be obtained by synthesizing all schedules within a single MILP formulation; however, the complexity scales exponentially with the number of variables [101], which limits the practicality of this "one-shot" approach. Another alternative would be to leverage the IIS information to iterate on previously scheduled modes. This is not trivial and, in particular, the scheduler must be careful not to search forever shall the system configuration would be truly unfeasible.

In practice though, the lack of completeness is not a critical problem. If one mode M is found unfeasible, a practical work-around is to add the conflicting applications (derived from the IIS) to the specification of M. This triggers the minimal inheritance mechanism and prevents conflicts in mode M, at a cost of an increase in utilization.

Integration of *TTnet* with the *TTW* scheduler. We presented in Section 5.7 our implementation of *TTnet* and the offline *TTW* scheduler. To obtain a full-fledged *TTW* implementation, one needs to integrate these two pieces. Concretely, that means designing a pipeline that turns the outputs of the *TTW* scheduler into individual nodes' scheduling tables, which can be then patched in the *TTnet* firmware. There is no technical limitation in realizing this; it could not be done in time before the completion of this dissertation, but we intend to make this happen in the near future.

Showcasing the full feature set of *TTW*. A running implementation of the *TTW* concept, including switching between operation modes, has been presented in the 2019 IPSN Demo Session [122]. This demo ran a more static and rigid software than the implementation presented in this chapter. Furthermore, all applications were considered non persistent; in other words, applications were re-starting from scratch with every mode change. A demonstration of the full feature set of *TTW* remains to be done.

Porting to other platforms. Our implementation of *TTnet* currently supports the TI CC430 SoC [180], which is not a commonly used platform and features

only little memory. Porting *TTnet* to newer platforms would be beneficial, and this is made simple by the use of the *Baloo* framework: whenever *Baloo* becomes available for a new platform, so does *TTnet*. In particular, a port to the nRF52840 Dongle [162] is envisioned, which would provide a physical layer (up to) 8x faster (and therefore, support for shorter end-to-end deadlines) and larger memory, which would facilitate the storing of scheduling tables.

Adaptability by reprogramming. The main limitation of *TTW* is that the schedules are static: the system executes pre-computed scheduling tables stored in the nodes' memory, with runtime adaptability limited to switching between different operating modes. However in principle, *Adaptability* could be improved by specifying a "reprogramming" mode in which new scheduling tables could be disseminated to the network to perform some sort of over-the-air reprogramming [85]. Doing this reliably is challenging and would be an interesting extension of this work.

5.11 Related Work

Various high reliability protocols have been proposed for low-power multi-hop wireless network, like TSCH [192], WirelessHART [91] or LWB [72]. Blink [208] was proposed as a real-time scheduling extension for protocols based on synchronous transmissions. Despite their respective benefits, all these protocols consider only network resources. They do not take into account the scheduling of distributed tasks on the computation resources, and therefore they hardly support end-to-end deadlines as commonly required for CPS applications [18]. In Chapter 4, we proposed *DRP* a protocol that provides such end-to-end guarantees, but couples tasks and messages as loosely as possible, aiming for efficient support of sporadic or event-triggered applications. This results in high worst-case latency and is thus not suitable for demanding CPS applications [18]. This observation points toward a fully time-triggered system where tasks and messages are co-scheduled.

In the wired domain, much work has been done on time-triggered architecture, like TTP [105], the static-segment of FlexRay [75], or TTEthernet [106]. Many recent works use SMT- of MILP-based methods to synthesize and/or analyze static (co-)schedules for those architectures [172, 56, 23, 176, 203]. However, these approaches assume that a message can be scheduled at any time. While being a perfectly valid hypothesis for a wired system, this assumption is not compatible with the use of communication rounds in a wireless setting. As shown in Section 5.9, using rounds significantly reduces the energy consumed for communication, but it makes the schedule synthesis more complex (Section 5.4).

5.12 Summary

In this chapter, we presented Time-Triggered Wireless (*TTW*), a time-triggered design for wireless CPS. *TTW* provides end-to-end real-time guarantees by statically co-scheduling all tasks and messages in the system, which is performed offline by resolving a MILP formulation. This approach is inspired by similar work in the wired domain, in particular the real-time scheduling of FlexRay buses. Compared to *DRP* (Chapter 4), *TTW*'s static schedules allow to meet shorter end-to-end deadlines *Efficiency* at the cost of a lesser (*Adaptability*); indeed, *TTW*'s runtime adaptability is limited to switching between pre-defined operation modes.

The main challenge in the *TTW* design is that, with wireless communication, it is highly beneficial in terms of energy to send messages in rounds. Thus, the assignment of messages to round (similar to a bin-packing problem) must be combined to the traditional co-scheduling approaches, which is non-trivial.

We solved this problem and implemented a multi-mode scheduler that allows critical applications to seamlessly switch between modes while minimizing the energy consumption spent for wireless communication. We further implemented a predictable network stack, called *TTnet*. Together, these two pieces from *TTW*, a publicly available (Section 5.A) real-time wireless CPS design.

5.A Appendix – Artifacts and Links

5.A.1 Related Publications

TTW: A Time-Triggered Wireless design for CPS
Romain Jacob, Licong Zhang, Marco Zimmerling, Jan Beutel, Samarjit Chakraborty, Lothar Thiele
DATE 2018. Dresden, Germany (March 2018)

Paper	10.23919/DATE.2018.8342127
Extended paper	arxiv.org/abs/1711.05581v2
Poster	10.3929/ethz-b-000375611

5.A.2 Complementary Materials

Complementary materials for this chapters are available on GitHub, together with the dissertation source files. For all links below, replace `<root>` by "github.com/romain-jacob/doctoral-thesis/blob/master"

TEX sources `<root>/50_TTW/`

Figures
- Static `<root>/50_TTW/Figures/`
- Dynamic `<root>/notebooks/ttw_plots.ipynb`

TTnet source code (examples/baloo-ttnet)
- Latest release 10.5281/zenodo.3510171
- "This-version" release 10.5281/zenodo.3530632

TTW Scheduler (sources and documentation)
- Latest release 10.5281/zenodo.3530665
- "This-version" release 10.5281/zenodo.3530666

Experiment data
- Repository 10.5281/zenodo.3530721
- Notebook (online visualization)

5.B System Configuration – Scheduler Evaluation

This appendix details the system configuration used for the performance evaluation of *TTW*'s scheduler (Section 5.8). The configuration consists of 15 different applications involving 45 tasks and 30 messages, executing in 5 different modes (Table 5.7). All applications are considered persistent, and the precedence graphs always contain a single chain. The tasks are mapped to 13 different nodes (Table 5.6), with a WCET set to 1 ms. The mode graph is shown in Figure 5.12.

Table 5.6 Task mapping for *TTW*'s scheduler evaluation (Section 5.8).

Node	Mapped tasks					
Node 1	$T1$	$T3$	$T4$	$T6$	$T25$	$T27$
Node 2	$T7$	$T9$	$T10$	$T12$		
Node 3	$T13$	$T15$	$T16$	$T18$		
Node 4	$T22$	$T24$				
Node 5	$T28$	$T30$				
Node 6	$T2$	$T8$	$T14$			
Node 7	$T5$	$T11$	$T17$	$T23$	$T26$	$T29$
Node 8	$T31$	$T33$	$T34$	$T36$		
Node 9	$T37$	$T39$	$T40$	$T42$		
Node 10	$T52$	$T54$				
Node 11	$T55$	$T57$				
Node 12	$T32$	$T38$	$T56$			
Node 13	$T35$	$T41$	$T53$			

Table 5.7 System configuration for *TTW*'s scheduler evaluation (Section 5.8).

Mode	Application	Period (s)	Deadline (s)	Chain				
M_1	$A1$	20	20	$T1$	$M1$	$T2$	$M2$	$T3$
	$A3$	20	10	$T7$	$M5$	$T8$	$M6$	$T9$
	$A4$	20	20	$T10$	$M7$	$T11$	$M8$	$T12$
	$A8$	40	40	$T22$	$M15$	$T23$	$M16$	$T24$
	$A10$	80	80	$T28$	$M19$	$T29$	$M20$	$T30$
M_2	$A1$	20	20	$T1$	$M1$	$T2$	$M2$	$T3$
	$A3$	10	20	$T7$	$M5$	$T8$	$M6$	$T9$
	$A4$	20	20	$T10$	$M7$	$T11$	$M8$	$T12$
	$A6$	10	10	$T16$	$M11$	$T17$	$M12$	$T18$
M_3	$A3$	20	10	$T7$	$M5$	$T8$	$M6$	$T9$
	$A9$	80	80	$T25$	$M17$	$T26$	$M18$	$T27$
	$A10$	80	80	$T28$	$M19$	$T29$	$M20$	$T30$
	$A11$	20	20	$T31$	$M21$	$T32$	$M22$	$T33$
	$A14$	10	10	$T40$	$M27$	$T41$	$M28$	$T42$
	$A18$	40	40	$T52$	$M35$	$T53$	$M36$	$T54$
M_4	$A2$	20	20	$T4$	$M3$	$T5$	$M4$	$T6$
	$A3$	20	10	$T7$	$M5$	$T8$	$M6$	$T9$
	$A5$	20	10	$T13$	$M9$	$T14$	$M10$	$T15$
	$A6$	10	10	$T16$	$M11$	$T17$	$M12$	$T18$
	$A9$	80	80	$T25$	$M17$	$T26$	$M18$	$T27$
	$A12$	20	20	$T34$	$M23$	$T35$	$M24$	$T36$
	$A19$	80	40	$T55$	$M37$	$T56$	$M38$	$T57$
M_5	$A2$	20	20	$T4$	$M3$	$T5$	$M4$	$T6$
	$A4$	20	20	$T10$	$M7$	$T11$	$M8$	$T12$
	$A12$	20	20	$T34$	$M23$	$T35$	$M24$	$T36$
	$A13$	20	20	$T37$	$M25$	$T38$	$M26$	$T39$

6

Conclusions and Outlook

Cyber-Physical Systems (CPS) are believed to be the vector of the next computing revolution [148]. In recent years, the buzz words have changed: it is now all about the "Internet of Things" or "Industry 4.0" – different names, same concept. We are envisioning a world where devices with computing and wireless communication capabilities are deeply embedded into our lives, in our environments, and even within ourselves. However, the current state of technology is not quite yet able to realize these grand visions. In this dissertation, we considered certain challenges related to communication in ad-hoc networks; or as we say today: "on the edge". One important challenge is to support energy efficient and reliable communication within a network of mobile nodes, such as teams of robots or drones. It was recently shown that the technique referred to as *synchronous transmissions* (ST) has a lot of potential to tackle this challenge. Indeed, ST can be leveraged to perform network-wide flooding in a stateless fashion; that is, packets are sent to all nodes in the network, in a way that does not depend on the current topology. And since the topology does not matter, supporting mobility comes essentially for free.

Despite being a potential enabler for a whole new class of mobile CPS applications, ST has not yet been used much outside academia. We identify at least three reasons for that:

- ST requires a very precise control of the hardware; in particular, the timing of radio operations is critical (Section 1.3). This makes ST complex to integrate into larger systems. As there is a lack of tools and methods to facilitate this integration, using ST requires an expert knowledge that the typical system designer does not have.
- It is easier to believe what we can see, and the efficiency and reliability of ST often come across as too-good-to-be-true. While the academic community is now generally convinced, industry demands some proof-of-concepts showing that ST does fulfill its promises, when put to the test.

- Finally, it is paramount to confidently evaluate and estimate expected performance. The challenge of reproduciblility in networking experiments is often debated, and there seems to be a general lack of trust in the results presented in academic papers. It is not saying that researchers lie about their experiment results, but would the system perform as well in another scenario? Or in another network? Or with other parameters? The inability of answering these questions is problematic.

6.1 Contributions

This dissertation makes three main contributions to foster the adoption of ST.

Baloo We designed *Baloo*, a framework facilitating the implementation of network stacks based on ST. It is meant as a tool for system designers to test and experiment with ST without having to be versed into the details of radio management: users implement their protocol through a simple yet flexible API while *Baloo* handles all the complex low-level operations based on the users' inputs.

DRP* and *TTW We proposed two different system designs – the Distributed Real-time Protocol (*DRP*) and Time-Triggered Wireless (*TTW*) – which demonstrate for the first time that end-to-end real-time guarantees can be obtained in CPS using low-power wireless technology by leveraging ST. Our systems explore different parts of the design space: *DRP* uses contracts to maximize the flexibility of execution between distributed tasks, whereas *TTW* statically co-schedules all task executions and message transfers to minimize end-to-end latency. In particular, together with project partners, we used the *TTW* design to implement the first demonstrator of feedback-control of fast physical systems (multiple inverted pendulums) across a multi-hop network using low-power wireless technology [122].

TriScale With the design of *TriScale*, we worked towards more rigorous and reproducible experimental networking research. For the first time, we went beyond simple guidelines and proposed a concrete methodology for designing networking experiments and analyzing their data, which allowed us to proposed the first formalized definition of reproducibility in the context of networking experiments. The driving principle of *TriScale* is to rationalize the evaluation process: *i.e.*, justify what is required to do and what data to collect in order to support one's performance claims.

6.2 Future Developments

Benchmarking Wireless Protocols. Our work on *TriScale* stemmed from discussions in the low-power wireless community regarding the need for a benchmark to compare networking protocols [38]. As we were reflecting on how to design such a benchmark, the need for a more rigorous experimental methodology became obvious; a need that *TriScale* tries to fill. We can now return to our initial objectives and attempt to realize the vision of IoTBench: a benchmark to thoroughly and confidently compare the performance of wireless networking protocols [92].

Going further with ST. With the design of *Baloo*, we attempted to make ST more accessible; an attempt that appears to be successful. Less than a year after the initial paper, the first independent studies using *Baloo* have been published [168]. There are many opportunities for future developments of the framework; the most natural being the port of *Baloo* to other platforms. It has been shown that the principle of ST also work on other physical layers that IEEE802.15.4 (*e.g.*, Bluetooth [21] and LoRa [194]). To investigate this further, a port to the LoRa-compatible SemTech SX1262 chip [163] is currently under development. Another port to the Bluetooth-compatible nRF52840 Dongle [162] is planned in a near future. These would allow to experiment with ST-based networking on different physical layers and, by leveraging (hopefully upcoming) wireless protocol benchmarks, we would be able to objectively compare the performance trade-offs of these different technologies in a wide range of scenarios and applications.

Dependable networking. One important limitation of the system designs presented in this dissertation is the reliance on a central authority, which we call *host*, in order to coordinate communication within the network. This creates a single point of failure: if the host should fail (or be jammed), the entire network would stop its operation. For any safety-critical applications, this is not acceptable. It is therefore important to work on system designs that would "distribute the responsibility" of the host. Recent contributions provide consensus primitives in low-power networks [168, 21, 147], an important piece for fault-tolerance in distributed systems. However these works still rely on a central authority for elementary network functions, such as time synchronization. More efforts are required to designed truly dependable wireless networks.

6.3 Affirming Oneself as a Researcher

Pursuing a doctorate degree is not only about writing a dissertation: it is a fundamental training step towards affirming oneself as a researcher. The shaping of my own researcher identify has been strongly influenced by this statement[1]

"*If I am going to "make it" in science, it has to be on terms I can live with.*"

I cannot agree more and this idea now impacts many aspects of my work. For me, this translates into trying to follow the principles generally referred to as "Open Science". I hope this can be perceived in this dissertation. In practice, this implies for example favoring open and free software over commercial tools; this is why the earlier works from the dissertation used Matlab, while the later ones are based on Python. Moreover, I try to systematically release data and code with all publications, aiming to make any plot and experiment reproducible by others, which I believe should be a standard in science.

I recently wrote down my own objectives and expectations regarding the way I intend to do research; this has materialized into a "Pledge to Open Science" which is publicly available on my personal webpage.[2] I will do my best to live and work by this principles, because I believe this is the right thing to do. We shall see if that will be good enough to "make it".

[1] Erin McKiernan. 10.6084/m9.figshare.954994
[2] www.romainjacob.net/pledge-to-open-science

Bibliography

[1] –. Angainor – Reproducible Evaluation and Fault Injection of Large-Scale Distributed Systems. [Online] - Last access: 2019-9-19.

[2] –. Contiki: The Open Source Operating System for the Internet of Things. http://contiki-os.org/.

[3] –. FlockLab. https://gitlab.ethz.ch/tec/public/flocklab/wikis/home. [Online] - Last accessed: 2019-10-11.

[4] –. FreeRTOS - Market leading RTOS for embedded systems with Internet of Things extensions. https://www.freertos.org/.

[5] –. NumPy: The Fundamental Package for Scientific Computing with Python. [Online] - Last access: 2019-9-19.

[6] –. OpenMote B. http://www.openmote.com/product/openmote-b-single/.

[7] –. Plotly: Modern Analytic Apps for the Enterprise. https://plot.ly. [Online] - Last access: 2019-9-19.

[8] –. Python Data Analysis Library. https://pandas.pydata.org/. [Online] - Last access: 2019-9-19.

[9] –. SciPy: Open source scientific tools for Python. https://scipy.org/. [Online] - Last access: 2019-9-19.

[10] –. Sleeping Beauty. https://github.com/csarkar/sleeping-beauty, August 2016.

[11] –. Contiki-NG. http://contiki-ng.org/, November 2018.

[12] –. EWSN 2019 Dependability Competition. http://ewsn2019.thss.tsinghua.edu.cn/competition-scenario.html, February 2019.

[13] Tarek F Abdelzaher and Kang G Shin. Combined task and message scheduling in distributed real-time systems. *IEEE Transactions on parallel and distributed systems*, 10(11):1179–1191, 1999.

[14] Javier Acevedo. *Real-Time Scheduling on Resource-Constrained Embedded Systems*. Master Thesis, TU Dresden, September 2016.

[15] ACM. Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-badging, April 2018.

[16] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne. FIT IoT-LAB: A large scale open experimental IoT testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 459–464, December 2015. doi:10.1109/WF-IoT.2015.7389098.

[17] Advanticsys. MTM-CM5000-MSP 802.15.4 TelosB mote Module. https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html. [Online] - Last accessed: 2019-10-11.

[18] Johan Åkerberg, Mikael Gidlund, and Mats Björkman. Future research challenges in wireless sensor and actuator networks targeting industrial automation. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 410–415, July 2011. doi:10.1109/INDIN.2011.6034912.

[19] Johan Åkerberg, Mikael Gidlund, Frank Reichenbach, and Mats Björkman. Measurements on an industrial wireless HART network supporting PROFIsafe: A case study. In *ETFA2011*, pages 1–8, September 2011. doi:10.1109/ETFA.2011.6059011.

[20] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. Network-wide Consensus Utilizing the Capture Effect in Low-power Wireless Networks. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, pages 1:1–1:14, New York, NY, USA, 2017. ACM. doi:10.1145/3131672.3131685.

[21] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. Concurrent Transmissions for Multi-Hop Bluetooth 5. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, page 12, February 2019.

[22] ARM. Cortex-M4. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4.

[23] Mohammad Ashjaei, Nima Khalilzad, Saad Mubeen, Moris Behnam, Ingo Sander, Luis Almeida, and Thomas Nolte. Designing end-to-end resource reservations in predictable distributed embedded systems. *Real-Time Systems*, pages 1–41, 2017.

[24] Akramul Azim. *Scheduling of Overload-Tolerant Computation and Multi-Mode Communication in Real-Time Systems*. Doctoral Thesis, University of Waterloo, December 2014.

[25] Jonas Bächli. *Creating a Flexible Middleware for Low-Power Flooding Protocols*. Master Thesis, ETH Zurich, June 2018. doi:10.3929/ethz-b-000270388.

[26] Michael Baddeley, Usman Raza, Aleksandar Stanoev, George Oikonomou, Reza Nejabati, Mahesh Sooriyabandara, and Dimitra Simeonidou. Atomic-SDN: Is Synchronous Flooding the Solution to Software-Defined Networking in IoT? *IEEE Access*, 7:96019–96034, 2019. doi:10.1109/ACCESS.2019.2920100.

[27] Michael Baddeley, Aleksandar Stanoev, Usman Raza, Yichao Jin, and Mahesh Sooriyabandara. Competition: Adaptive Software Defined Scheduling of Low Power Wireless Networks. In *16th International Conference on Embedded Wireless Systems and Networks (EWSN 2019)*, page 3, February 2019.

[28] Vaibhav Bajpai, Olivier Bonaventure, Kimberly Claffy, and Daniel Karrenberg. Encouraging Reproducibility in Scientific Research of the Internet (Dagstuhl Seminar 18412). *Dagstuhl Reports*, 8(10):41–62, 2019. doi:10.4230/DagRep.8.10.41.

[29] Vaibhav Bajpai, Anna Brunstrom, Anja Feldmann, Wolfgang Kellerer, Aiko Pras, Henning Schulzrinne, Georgios Smaragdakis, Matthias Wählisch, and Klaus Wehrle. The Dagstuhl Beginners Guide to Reproducibility for Experimental Networking Research. *SIGCOMM Comput. Commun. Rev.*, 49(1):24–30, January 2019.

[30] Monya Baker. Is There a Reproducibility Crisis? *Nature News*, 533(7604):452–454, May 2016.

[31] Arijit Banerjee, Junguk Cho, Eric Eide, Jonathon Duerig, Binh Nguyen, Robert Ricci, Jacobus Van der Merwe, Kirk Webb, and Gary Wong. PhantomNet: Research Infrastructure for Mobile Networking, Cloud Computing and Software-Defined Networking. *GetMobile: Mobile Comp. and Comm.*, 19(2):28–33, August 2015. doi:10.1145/2817761.2817772.

[32] Lorena A. Barba. Terminologies for Reproducible Research. *arXiv:1802.03311 [cs]*, February 2018. arXiv:1802.03311.

[33] Paulo Bartolomeu, Muhammad Alam, Joaquim Ferreira, and José Fonseca. Survey on low power real-time wireless MAC protocols. *Journal of Network and Computer Applications*, 75:293–316, November 2016. doi:10.1016/j.jnca.2016.09.004.

[34] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, March 2014. doi:10.1016/j.bjp.2013.12.037.

[35] Jan Beutel, Roman Trüb, Reto Da Forno, Markus Wegmann, Tonio Gsell, Romain Jacob, Michael Keller, Felix Sutton, and Lothar Thiele. The Dual Processor Platform Architecture: Demo Abstract. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN '19, pages 335–336, New York, NY, USA, 2019. ACM. doi:10.1145/3302506.3312481.

[36] Andreas Biri. *Unleashing the Potential of Real-Time Internet of Things*. Semester Thesis, ETH Zurich, December 2017. doi:10.3929/ethz-b-000234913.

[37] Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Nathan Keynes, Nathaniel Nystrom, and Andreas Zeller. The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations. *ACM Trans. Program. Lang. Syst.*, 38(4):15:1–15:20, October 2016. doi:10.1145/2983574.

[38] Carlo A. Boano, Simon Duquennoy, Anna Förster, Omprakash Gnawali, Romain Jacob, Hyung-Sin Kim, Olaf Landsiedel, Ramona Marfievici, Luca Mottola, Gian Pietro Picco, Xavier Vilajosana, Thomas Watteyne, and Marco Zimmerling. IoTBench: Towards a Benchmark for Low-power Wireless Networking. In *1st Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench 2018)*, April 2018. doi:10.3929/ethz-b-000256517.

[39] Ronald F. Boisvert. Incentivizing Reproducibility. *Commun. ACM*, 59(10):5–5, September 2016. doi:10.1145/2994031.

[40] Raphaël Bolze et al. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[41] Olivier Bonaventure, Luigi Iannone, and Damien Saucez, editors. *Proceedings of the International ACM SIGCOMM Reproducibility Workshop (Reproducibility'17)*. ACM, Los Angeles, CA, USA, August 2017.

[42] Olfa Boubaker. The inverted pendulum: A fundamental benchmark in control theory and robotics. In *International Conference on Education and E-Learning Innovations*, pages 1–6, July 2012. doi:10.1109/ICEELI.2012.6360606.

[43] Peter J. Brockwell, Richard A. Davis, and Stephen E. Fienberg. *Time Series: Theory and Methods: Theory and Methods*. Springer Science & Business Media, 1991. doi:10.1007/978-1-4419-0320-4.

[44] Ryan Burchfield, Ehsan Nourbakhsh, Jeff Dix, Kunal Sahu, S. Venkatesan, and Ravi Prakash. RF in the Jungle: Effect of Environment Assumptions on Wireless Experiment Repeatability. In *Proceedings of the International Conference on Communications (ICC)*, pages 1–6. IEEE, June 2009.

[45] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Science & Business Media, September 2011.

[46] Björn Cassens, Markus Hartmann, Thorstan Nowak, Niklas Duda, Jörn Thielecke, Alexander Kölpin, and Rüdiger Kapitza. Bursting: Increasing Energy Efficiency of Erasure-Coded Data in Animal-Borne Sensor Networks. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, EWSN '19, pages 59–70, USA, 2019. Junction Publishing.

[47] I. Chatzigiannakis, G. Mylonas, and S. Nikoletseas. 50 ways to build your application: A survey of middleware and systems for Wireless Sensor Networks. In *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*, pages 466–473, September 2007. doi:10.1109/EFTA.2007.4416805.

[48] T. Chen and L. T. X. Phan. SafeMC: A System for the Design and Evaluation of Mode-Change Protocols. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–116, April 2018. doi:10.1109/RTAS.2018.00021.

[49] K. Chintalapudi, T. Fu, J. Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Monitoring civil structures with a wireless sensor network. *IEEE Internet Computing*, 10(2):26–34, March 2006. doi:10.1109/MIC.2006.38.

[50] Junguk Cho, Jonathan Duerig, Eric Eide, Binh Nguyen, Robert Ricci, Aisha Syed, Jacobus Van der Merwe, Kirk Webb, and Gary Wong. Repeatable mobile networking research with phantomNet: Demo. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking - MobiCom '16*, pages 489–490, New York City, New York, 2016. ACM Press. doi:10.1145/2973750.2985616.

[51] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-coverage Services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003. doi:10.1145/956993.956995.

[52] Christian Collberg, Todd Proebsting, and Alex M. Warren. Repeatability and Benefaction in Computer Systems Research. Technical Report TR 14–04, University of Arizona, February 2015.

[53] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The RUNES middleware: A reconfigurable component-based approach to networked embedded systems. In *2005 IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications*, volume 2, pages 806–810 Vol. 2, September 2005. doi:10.1109/PIMRC.2005.1651554.

[54] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. Programming Wireless Sensor Networks with the TeenyLime Middleware. In Renato Cerqueira and Roy H. Campbell, editors, *Middleware 2007*, Lecture Notes in Computer Science, pages 429–449. Springer Berlin Heidelberg, 2007.

[55] Silviu S. Craciunas and Ramon Serna Oliver. SMT-based Task- and Network-level Static Schedule Generation for Time-Triggered Networked Systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 45:45–45:54, New York, NY, USA, 2014. ACM. doi:10.1145/2659787.2659812.

[56] Silviu S. Craciunas and Ramon Serna Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Real-Time Systems*, 2016.

[57] R. L. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991. doi:10.1109/18.61109.

[58] Geoff Cumming and Sue Finch. A Primer on the Understanding, Use, and Calculation of Confidence Intervals that are Based on Central and Noncentral Distributions. *Educational and Psychological Measurement*, 61(4):532–574, August 2001.

[59] Wan Du, Jansen Christian Liando, Huanle Zhang, and Mo Li. When Pipelines Meet Fountain: Fast Data Dissemination in Wireless Sensor Networks. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 365–378, New York, NY, USA, 2015. ACM. doi:10.1145/2809695.2809721.

[60] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, November 2004. doi:10.1109/LCN.2004.38.

[61] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM. doi:10.1145/1182807.1182811.

[62] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pages 1–14, 2019.

[63] Simon Duquennoy, Beshr Al Nahas, Olaf Landsiedel, and Thomas Watteyne. Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 337–350, New York, NY, USA, 2015. ACM. doi:10.1145/2809695.2809714.

[64] Simon Duquennoy, Joakim Eriksson, and Thiemo Voigt. Five-Nines Reliable Downward Routing in RPL. *arXiv:1710.02324 [cs]*, October 2017. arXiv:1710.02324.

[65] Sarah Edwards, Xuan Liu, and Niky Riga. Creating Repeatable Computer Science and Networking Experiments on Shared, Public Testbeds. *SIGOPS Oper. Syst. Rev.*, 49(1):90–99, January 2015.

[66] A. Escobar, F. J. Cruz, J. Garcia-Jimenez, J. Klaue, and A. Corona. RedFixHop with channel hopping: Reliable ultra-low-latency network flooding. In *2016 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–4, November 2016. doi:10.1109/DCIS.2016.7845367.

[67] Antonio Escobar, Fernando Moreno, Antonio J. Cabrera, Javier Garcia-Jimenez, Francisco J. Cruz, Unami Ruiz, Jirka Klaue, Angel Corona, Divya Tati, and Thomas Meyerhoff. Competition: BigBangBus. In *Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks*, EWSN '18, pages 213–214, USA, 2018. Junction Publishing.

[68] Antonio Escobar-Molero. Improving reliability and latency of Wireless Sensor Networks using Concurrent Transmissions. *at - Automatisierungstechnik*, 67(1):42–50, 2019. doi:10.1515/auto-2018-0064.

[69] Antonio Escobar-Molero, Javier Garcia-Jimenez, Jirka Klaue, Fernando Moreno-Cruz, Borja Saez, Francisco J Cruz, Unai Ruiz, and Angel Corona. Competition: RedNodeBus, Stretching out the Preamble. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, page 2, February 2019.

[70] 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*, pages 73–84, April 2011.

[71] Federico Ferrari, Marco Zimmerling, and Reto Da Forno. Low-Power Wireless Bus (LWB). https://github.com/ETHZ-TEC/LWB, September 2017.

[72] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar 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. doi:10.1145/2426656.2426658.

[73] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Virtual Synchrony Guarantees for Cyber-physical Systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 20–30, September 2013. doi:10.1109/SRDS.2013.11.

[74] Daniel C. Ferreira, Félix Iglesias Vázquez, Gernot Vormayr, Maximilian Bachl, and Tanja Zseby. A Meta-Analysis Approach for Feature Selection in Network Traffic Research. In *Proceedings of the International ACM SIGCOMM Reproducibility Workshop (Reproducibility'17)*, pages 17–20, Los Angeles, CA, USA, August 2017. ACM.

[75] FlexRay. ISO 17458-1:2013–Road vehicles–FlexRay communications system–Part 1: General information and use case definition. Standard, International Organization for Standardization (ISO), Geneva, Switzerland, 2013.

[76] Matthias Flittner, Mohamed Naoufal Mahfoudi, Damien Saucez, Matthias Wählisch, Luigi Iannone, Vaibhav Bajpai, and Alex Afanasyev. A Survey on Artifacts from CoNEXT, ICN, IMC, and SIGCOMM Conferences in 2017. *SIGCOMM Comput. Commun. Rev.*, 48(1):75–80, April 2018. doi:10.1145/3211852.3211864.

[77] Gerhard Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE transactions on information and systems*, 76(11):1333–1340, 1993.

[78] C. Galán, M. Smith, M. Thibaudon, G. Frenguelli, J. Oteros, R. Gehrig, U. Berger, B. Clot, R. Brandao, and EAS QC Working Group. Pollen monitoring: Minimum requirements and reproducibility of analysis. *Aerobiologia*, 30(4):385–395, December 2014. doi:10.1007/s10453-014-9335-5.

[79] Sachin Ganu, Haris Kremo, Richard Howard, and Ivan Seskar. Addressing Repeatability in Wireless Experiments Using ORBIT Testbed. In *Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM)*, pages 153–160, Trento, Italy, February 2005. IEEE Computer Society.

[80] Kai Geissdoerfer, Brano Kusy, Raja Jurdak, and Marco Zimmerling. Getting More Out of Energy-harvesting Systems: Energy Management under Time-varying Utility with PREAcT. In *2019 18th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 109–120, April 2019. doi:10.1145/3302506.3310393.

[81] Omprakash Gnawali, Marco Zimmerling, and Sebastian Trimpe, editors. *Proceedings of the 1st International Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench)*. IEEE, Porto, Portugal, April 2018.

[82] M. Gonzalez Harbour, J.J. Gutierrez Garcia, J.C. Palencia Gutierrez, and J.M. Drake Moyano. MAST: Modeling and analysis suite for real time applications. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 125–134, June 2001. doi:10.1109/EMRTS.2001.934015.

[83] Gurobi Optimization. Gurobi - The fastest solver. https://www.gurobi.com/.

[84] Martin Haegele. Logistics drives 39% increase in professional service robot sales. https://ifr.org/post/logistics-drives-39-increase-in-professional-service-robot-sales, November 2018.

[85] Andrew Hagedorn, David Starobinski, and Ari Trachtenberg. Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes. In *2008 International Conference on Information Processing in Sensor Networks (Ipsn 2008)*, pages 457–466, April 2008. doi:10.1109/IPSN.2008.9.

[86] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-Based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 253–264, Nice, France, December 2012. ACM.

[87] Tian He, J.A. Stankovic, Chenyang Lu, and T. Abdelzaher. SPEED: A stateless protocol for real-time communication in sensor networks. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 46–55, May 2003. doi:10.1109/ICDCS.2003.1203451.

[88] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2):148–166, March 2005. doi:10.1049/ip-cdt:20045088.

[89] Carsten Herrmann, Fabian Mager, and Marco Zimmerling. Mixer: Efficient Many-to-All Broadcast in Dynamic Wireless Mesh Networks. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, pages 145–158, New York, NY, USA, 2018. ACM. doi:10.1145/3274783.3274849.

[90] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll. Static scheduling of a Time-Triggered Network-on-Chip based on SMT solving. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 509–514, March 2012.

[91] International Electrotechnical Commission (IEC). Industrial networks - Wireless communication network and communication profiles - WirelessHART. https://webstore.iec.ch/publication/24433.

[92] IoT Benchmarks Initiative. IoTBench. https://www.iotbench.ethz.ch/.

[93] ISA100. Wireless Compliance Institute. http://www.isa100wci.org/, 2009.

[94] P. H. Isolani, M. Claeys, C. Donato, L. Z. Granville, and S. Latré. A Survey on the Programmability of Wireless MAC Protocols. *IEEE Communications Surveys Tutorials*, pages 1–1, 2018. doi:10.1109/COMST.2018.2881761.

[95] Timofei Istomin, Matteo Trobinger, Amy L. Murphy, and Gian Pietro Picco. Interference-resilient Ultra-low Power Aperiodic Data Collection. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '18, pages 84–95, Piscataway, NJ, USA, 2018. IEEE Press. doi:10.1109/IPSN.2018.00015.

[96] Romain Jacob, Jonas Bächli, Reto Da Forno, and Lothar Thiele. Synchronous Transmissions Made Easy: Design Your Network Stack with Baloo. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, 2019.

[97] Romain Jacob, Reto Da Forno, Roman Trüb, Andreas Biri, and Lothar Thiele. Wireless Link Quality Estimation on FlockLab - and Beyond, July 2019. doi:10.5281/zenodo.3354717.

[98] Romain Jacob, Reto Da Forno, and Jonas Bächli. Baloo: Latest release. Zenodo, October 2019. doi:10.5281/zenodo.3510171.

[99] Romain Jacob, Reto Da Forno, Roman Trüb, Andreas Biri, and Lothar Thiele. Dataset: Wireless Link Quality Estimation on FlockLab – and Beyond. In *Proceedings of the 2nd International Workshop on Data Acquisition to Analysis (DATA)*, New York, NY, USA, November 2019. ACM. doi:10.3929/ethz-b-000355846.

[100] Romain Jacob, Licong Zhang, Marco Zimmerling, Jan Beutel, Samarjit Chakraborty, and Lothar Thiele. TTW: A Time-Triggered-Wireless Design for CPS [Extended version]. *arXiv:1711.05581 [cs]*, November 2017. arXiv:1711.05581.

[101] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.

[102] Pravein Govindan Kannan, Ahmad Soltani, Mun Choon Chan, and Ee-Chien Chang. BNV: Enabling Scalable Network Experimentation throughBare-metal Network Virtualization. In *Proceedings of the 11th USENIX Conference on Cyber Security Experimentation and Test (CSET)*. USENIX Association, August 2018.

[103] H. S. Kim, J. Ko, D. E. Culler, and J. Paek. Challenging the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL): A Survey. *IEEE Communications Surveys Tutorials*, 19(4):2502–2525, Fourthquarter 2017. doi:10.1109/COMST.2017.2751617.

[104] Kevin Klues, Gregory Hackmann, Octav Chipara, and Chenyang Lu. A Component-based Architecture for Power-efficient Media Access Control in Wireless Sensor Networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 59–72, New York, NY, USA, 2007. ACM. doi:10.1145/1322263.1322270.

[105] H. Kopetz and G. Grunsteidl. TTP - A time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 524–533, June 1993. doi:10.1109/FTCS.1993.627355.

[106] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The Time-Triggered Ethernet (TTE) Design. *Proc. of the IEEE ISORC*, 2005.

[107] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 203–216, 2014.

[108] Antonios Koskinas. *Is Low-Power Wireless Networking a Reproducible Science?* Semester Thesis, ETH Zurich, January 2019. doi:10.3929/ethz-b-000324251.

[109] K. Kritsis, G. Z. Papadopoulos, A. Gallais, P. Chatzimisios, and F. Théoleyre. A Tutorial on Performance Evaluation and Validation Methodology for Low-Power and Lossy Networks. *IEEE Communications Surveys Tutorials*, pages 1–1, 2018. doi:10.1109/COMST.2018.2820810.

[110] Daniël Lakens. Equivalence Tests: A Practical Primer for *t* Tests, Correlations, and Meta-Analyses. *Social Psychological and Personality Science*, 8(4):355–362, May 2017. doi:10.1177/1948550617697177.

[111] Olaf Landsiedel, Federico Ferrari, and Marco 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 '13, pages 1:1–1:14, New York, NY, USA, 2013. ACM. doi:10.1145/2517351.2517358.

[112] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050. Springer Science & Business Media, 2001.

[113] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008. doi:10.1109/ISORC.2008.25.

[114] Roman Lim, Reto Da Forno, Felix Sutton, and Lothar Thiele. Competition: Robust Flooding Using Back-to-Back Synchronous Transmissions with Channel-Hopping. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, EWSN '17, pages 270–271, USA, 2017. Junction Publishing.

[115] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan 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 '13, pages 153–166, New York, NY, USA, April 2013. ACM. doi:10.1145/2461381.2461402.

[116] David M. Liu and Matthew J. Salganik. Successes and Struggles with Computational Reproducibility: Lessons from the Fragile Families Challenge. Technical report, OSF.io, March 2019.

[117] Ting Liu and Margaret Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 107–118, New York, NY, USA, 2003. ACM. doi:10.1145/781498.781516.

[118] Chenyang Lu, B.M. Blum, T.F. Abdelzaher, J.A. Stankovic, and Tian He. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–66, September 2002. doi:10.1109/RTTAS.2002.1137381.

[119] M. Luvisotto, Z. Pang, and D. Dzung. Ultra High Performance Wireless Control for Critical Applications: Challenges and Directions. *IEEE Transactions on Industrial Informatics*, 13(3):1448–1459, June 2017. doi:10.1109/TII.2016.2617459.

[120] Xiaoyuan Ma, Peilin Zhang, Ye Liu, Xin Li, Weisheng Tang, Pei Tian, Jianming Wei, Lei Shu, and Oliver Theel. Competition: Using DeCoT+ to Collect Data under Interference. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, page 2, February 2019.

[121] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005. doi:10.1145/1061318.1061322.

[122] Fabian Mager, Dominik Baumann, Romain Jacob, Lothar Thiele, Sebastian Trimpe, and Marco Zimmerling. Demo Abstract: Fast Feedback Control and Coordination with Mode Changes for Wireless Cyber-Physical Systems. *Proceedings of the 18th International Conference on Information Processing in Sensor Networks - IPSN '19*, pages 340–341, 2019. arXiv:1906.05554, doi:10.1145/3302506.3312483.

[123] Fabian Mager, Dominik Baumann, Romain Jacob, Lothar Thiele, Sebastian Trimpe, and Marco Zimmerling. Feedback Control Goes Wireless: Guaranteed Stability over Low-power Multi-hop Networks. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '19, pages 97–108, Montreal, Quebec, Canada, 2019. ACM. doi:10.1145/3302509.3311046.

[124] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *Proceedings of the 13th International USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 409–425, Carlsbad, CA, USA, October 2018. USENIX Association.

[125] Abdelbassat Massouri, Leonardo Cardoso, Benjamin Guillon, Florin Hutu, Guillaume Villemaud, Tanguy Risset, and Jean-Marie Gorce. CorteXlab: An open FPGA-based Facility for Testing SDR and Cognitive Radio Networks in a Reproducible Environment. In *Proceedings of the International Conference on Computer Communications (INFOCOM) Workshops*, pages 103–104, San Francisco, CA, USA, April 2014. IEEE.

[126] MathWorks. MATLAB - MathWorks. https://www.mathworks.com/products/matlab.html.

[127] Miguel Matos. Towards Reproducible Evaluation of Large-Scale Distributed Systems. In *Proceedings of the International Workshop on Advanced Tools, Programming Languages, and Platforms for Implementing and Evaluating Algorithms for Distributed Systems (ApPLIED)*, pages 5–7, Egham, United Kingdom, July 2018. ACM. doi:10.1145/3231104.3231113.

[128] Matthias Meyer, Timo Farei-Campagna, Akos Pasztor, Reto Da Forno, Tonio Gsell, Jérome Faillettaz, Andreas Vieli, Samuel Weber, Jan Beutel, and Lothar Thiele. Event-triggered Natural Hazard Monitoring with Convolutional Neural Networks on the Edge. In *2019 18th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 73–84, April 2019. doi:10.1145/3302506.3310390.

[129] Micro Focus. Seven Ways to Fail. Technical Report Brochure on Application Development, Test, and Delivery, –, March 2018.

[130] Mobashir Mohammad and Mun Choon Chan. Codecast: Supporting Data Driven In-network Processing for Low-power Wireless Sensor Networks. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '18, pages 72–83, Piscataway, NJ, USA, 2018. IEEE Press. doi:10.1109/IPSN.2018.00014.

[131] Mobashir Mohammad, Manjunath Doddavenkatappa, and Mun Choon Chan. Improving Performance of Synchronous Transmission-Based Protocols Using Capture Effect over Multichannels. *ACM Trans. Sen. Netw.*, 13(2):10:1–10:26, April 2017. doi:10.1145/3043790.

[132] Luca Mottola, Mattia Moretta, Kamin Whitehouse, and Carlo Ghezzi. Team-level Programming of Drone Sensor Networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 177–190, New York, NY, USA, 2014. ACM. doi:10.1145/2668332.2668353.

[133] Luca Mottola and Gian Pietro Picco. MUSTER: Adaptive Energy-Aware Multisink Routing in Wireless Sensor Networks. *IEEE Transactions on Mobile Computing*, 10(12):1694–1709, December 2011. doi:10.1109/TMC.2010.250.

[134] Luca Mottola and Gian Pietro Picco. Middleware for wireless sensor networks: An outlook. *Journal of Internet Services and Applications*, 3(1):31–39, May 2012. doi:10.1007/s13174-011-0046-7.

[135] Jan Mueller. *Low-Power Network Design: Work Hard, Play Hard*. Semester Thesis, ETH Zurich, January 2019. doi:10.3929/ethz-b-000324247.

[136] Jan Mueller, Anna-Brit Schaper, Romain Jacob, and Reto Da Forno. Competition: Keep it Simple, Let Flooding Shine. In *16th International Conference on Embedded Wireless Systems and Networks (EWSN 2019)*, February 2019. doi:10.3929/ethz-b-000325870.

[137] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of the International USENIX Annual Technical Conference (ATC)*, pages 417–429, Santa Clara, CA, USA, July 2015. USENIX Association.

[138] NSF. *Cyber Physical Systems – Nsf10515*. NSF, 2010.

[139] Lucas Nussbaum. Testbeds Support for Reproducible Research. In *Proceedings of the International ACM SIGCOMM Reproducibility Workshop (Reproducibility'17)*, Reproducibility '17, pages 24–26, Los Angeles, CA, USA, August 2017. ACM. doi:10.1145/3097766.3097773.

[140] NXP. LPC541XX. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc54000-cortex-m4-/low-power-microcontrollers-mcus-based-on-arm-cortex-m4-cores-with-optional-cortex-m0-plus-co-processor:LPC541XX.

[141] NXP. VF3xxR. https://www.nxp.com/products/processors-and-microcontrollers/legacy-mcu-mpus/vfxxx-controller/r-series/32-bit-devices-for-advanced-connected-radio-entry-level-infotainment-and-digital-instrument-cluster-applications.:VF3xxR.

[142] Tony O'donovan, James Brown, Felix Büsching, Alberto Cardoso, José Cecílio, Jose Do Ó, Pedro Furtado, Paulo Gil, Anja Jugel, Wolf-Bastian Pöttner, Utz Roedig, Jorge Sá Silva, Ricardo Silva, Cormac J. Sreenan, Vasos Vassiliou, Thiemo Voigt, Lars Wolf, and Zinon Zinonos. The GINSENG System for Wireless Monitoring and Control: Design and Deployment Experiences. *ACM Trans. Sen. Netw.*, 10(1):4:1–4:40, December 2013. doi:10.1145/2529975.

[143] Martijn Onderwater. An overview of centralised middleware components for sensor networks. *International Journal of Ad Hoc and Ubiquitous Computing*, 21(3):180–193, January 2016. doi:10.1504/IJAHUC.2016.075378.

[144] Tom Parker, Gertjan Halkes, Maarten Bezemer, and Koen Langendoen. The λMAC Framework: Redefining MAC Protocols for Wireless Sensor Networks. *Wirel. Netw.*, 16(7):2013–2029, October 2010. doi:10.1007/s11276-010-0241-7.

[145] Roger Peng. The Reproducibility Crisis in Science: A Statistical Counterattack. *Significance*, 12(3):30–32, June 2015.

[146] Hans E. Plesser. Reproducibility vs. Replicability: A Brief History of a Confused Terminology. *Frontiers in Neuroinformatics*, 11(76):1–4, January 2018.

[147] Valentin Poirot, Beshr Al Nahas, and Olaf Landsiedel. Paxos Made Wireless: Consensus in the Air. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, EWSN '19, pages 1–12, USA, 2019. Junction Publishing.

[148] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*, pages 731–736, June 2010. doi:10.1145/1837274.1837461.

[149] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(1):70–95, February 2016. doi:10.1109/JIOT.2015.2498900.

[150] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera, and Jay Lepreau. The Flexlab Approach to Realistic Evaluation of Networked Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 15–15, Cambridge, MA, USA, April 2007. USENIX Association.

[151] Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb, Jonathon Duerig, Keith Downie, and Mike Hibler. Apt: A Platform for Repeatable Research in Computer Science. *SIGOPS Oper. Syst. Rev.*, 49(1):100–107, January 2015. doi:10.1145/2723872.2723885.

[152] Kay Römer. Programming paradigms and middleware for sensor networks. *GI/ITG Workshop on Sensor Networks*, pages 49–54, 2004.

[153] Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. Real-Time Scheduling for WirelessHART Networks. In *2010 31st IEEE Real-Time Systems Symposium*, pages 150–159, November 2010. doi:10.1109/RTSS.2010.41.

[154] Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. End-to-End Communication Delay Analysis in Industrial Wireless Networks. *IEEE Transactions on Computers*, 64(5):1361–1374, May 2015. doi:10.1109/TC.2014.2322609.

[155] C. Sarkar, R. V. Prasad, R. T. Rajan, and K. Langendoen. Sleeping Beauty: Efficient Communication for Node Scheduling. In *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 56–64, October 2016. doi:10.1109/MASS.2016.018.

[156] Damien Saucez and Luigi Iannone. Thoughts and Recommendations from the ACM SIGCOMM 2017 Reproducibility Workshop. *SIGCOMM Comput. Commun. Rev.*, 48(1):70–74, April 2018. doi:10.1145/3211852.3211863.

[157] Anna-Brit Schaper. *Low-Power Network Design: Work Hard, Play Hard: Data Collection*. Semester Thesis, ETH Zurich, January 2019. doi:10.3929/ethz-b-000324250.

[158] H. Schmid and A. Huber. Measuring a Small Number of Samples, and the 3σ Fallacy: Shedding Light on Confidence and Error Intervals. *IEEE Solid-State Circuits Magazine*, 6(2):52–58, June 2014. doi:10.1109/MSSC.2014.2313714.

[159] Markus Schuß, Carlo Alberto Boano, and Kay Römer. Moving Beyond Competitions: Extending D-Cube to Seamlessly Benchmark Low-Power Wireless Systems. In *Proceedings of the 1st International Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench)*, page 6. IEEE, April 2018. doi:10.1109/CPSBench.2018.00012.

[160] Markus Schuß, Carlo Alberto Boano, Manuel Weber, and Kay Römer. A Competition to Push the Dependability of Low-Power Wireless Protocols to the Edge. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, EWSN '17, pages 54–65, USA, February 2017. Junction Publishing.

[161] Markus Schuß, Carlo Alberto Boano, Manuel Weber, Matthias Schulz, Matthias Hollick, and Kay Römer. JamLab-NG: Benchmarking Low-Power Wireless Protocols under Controllable and Repeatable Wi-Fi Interference. In *Proceedings of the 16th International Conference on Embedded Wireless Systems and Networks (EWSN)*, pages 83–94, Beijing, China, February 2019. Junction Publishing.

[162] Nordic Semiconductors. nRF52840. https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840. [Online] - Last accessed: 2019-10-11.

[163] Semtech. SX1262. Long Range Low Power LoRa. https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1262.

[164] Pranab Kumar Sen. Estimates of the Regression Coefficient Based on Kendall's Tau. *Journal of the American Statistical Association*, 63(324):1379–1389, December 1968. doi:10.1080/01621459.1968.10480934.

[165] Swati Sharma, Alefiya Hussain, and Huzur Saran. Towards Repeatability and Verifiability in Networking Experiments: A Stochastic Framework. *Journal of Network and Computer Applications*, 81(1):12–23, March 2017.

[166] Bruno Sinopoli, Luca Schenato, Massimo Franceschetti, Kameshwar Poolla, Michael I. Jordan, and Shankar S. Sastry. Kalman Filtering with Intermittent Observations. *IEEE Transactions on Automatic Control*, 49(9):1453–1464, 2004.

[167] Philipp Sommer and Yvonne-Anne Pignolet. Competition: Dependable Network Flooding Using Glossy with Channel-Hopping. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, EWSN '16, pages 303–303, USA, 2016. Junction Publishing.

[168] Alberto Spina, Michael Breza, Naranker Dulay, and Julie McCann. XPC: Fast and Reliable Synchronous Transmission Protocols for 2-Phase Commit and 3-Phase Commit. *arXiv:1910.09941 [cs]*, October 2019. arXiv:1910.09941.

[169] J.A. Stankovic, T.E. Abdelzaher, Chenyang Lu, Lui Sha, and J.C. Hou. Real-time communication and coordination in embedded sensor networks. *Proceedings of the IEEE*, 91(7):1002–1022, July 2003. doi:10.1109/JPROC.2003.814620.

[170] John Stankovic, Insup Lee, Aloysius Mok, and Raj Rajkumar. Opportunities and obligations for physical computing systems. *Departmental Papers (CIS)*, November 2005.

[171] John A. Stankovic. When Sensor and Actuator Networks Cover the World. *ETRI Journal*, 30(5):627–633, 2008. doi:10.4218/etrij.08.1308.0099.

[172] Wilfried Steiner. An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 375–384. IEEE, 2010.

[173] Felix Sutton, Reto Da Forno, David Gschwend, Tonio Gsell, Roman Lim, Jan Beutel, and Lothar Thiele. The Design of a Responsive and Energy-efficient Event-triggered Wireless Sensing System. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, EWSN '17, pages 144–155, USA, 2017. Junction Publishing.

[174] Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell, Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele. Bolt: A Stateful Processor Interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 267–280, New York, NY, USA, 2015. ACM. doi:10.1145/2809695.2809706.

[175] M. Suzuki, Y. Yamashita, and H. Morikawa. Low-Power, End-to-End Reliable Collection Using Glossy for Wireless Sensor Networks. In *2013 IEEE 77th Vehicular Technology Conference (VTC Spring)*, pages 1–5, June 2013. doi:10.1109/VTCSpring.2013.6692624.

[176] Domitian Tamas-Selicean, Paul Pop, and Wilfried Steiner. Synthesis of communication schedules for TTEthernet-based mixed-criticality systems. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 473–482. ACM, 2012.

[177] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. In *Proceedings of the 9th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, CoNEXT '13, pages 217–228, New York, NY, USA, December 2013. ACM. doi:10.1145/2535372.2535374.

[178] Rodrigo Teles Hermeto, Antoine Gallais, and Fabrice Theoleyre. Scheduling for IEEE802.15.4-TSCH and slow channel hopping MAC in low power industrial wireless networks: A survey. *Computer Communications*, 114:84–105, December 2017. doi:10.1016/j.comcom.2017.10.004.

[179] Texas Instruments. CC2420 Datasheet.

[180] Texas Instruments. CC430F6137 16-Bit Ultra-Low-Power MCU. http://www.ti.com/product/CC430F6137. [Online] - Last accessed: 2019-10-11.

[181] Texas Instruments. MSP-EXP432P401R SimpleLink™. http://www.ti.com/tool/MSP-EXP432P401R.

[182] Texas Instruments. MSP430FR5969. http://www.ti.com/product/MSP430FR5969.

[183] Henri Theil. A Rank-Invariant Method of Linear and Polynomial Regression Analysis. In Baldev Raj and Johan Koerts, editors, *Henri Theil's Contributions to Economics and Econometrics: Econometric Theory and Methodology*, Advanced Studies in Theoretical and Applied Econometrics, pages 345–381. Springer Netherlands, Dordrecht, 1992. doi:10.1007/978-94-011-2546-8_20.

[184] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time Calculus for Scheduling Hard Real-time Systems. In *Proc. of IEEE ISCAS*, 2000.

[185] William R. Thompson. On Confidence Ranges for the Median and Other Expectation Distributions for Populations of Unknown Distribution Form. *The Annals of Mathematical Statistics*, 7(3):122–128, 1936.

[186] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40, 1994.

[187] Jan Vitek and Tomas Kalibera. Repeatability, Reproducibility and Rigor in Systems Research. In *Proceedings of the 9th International Conference on Embedded Software (EMSOFT)*, pages 33–38. ACM, October 2011.

[188] Fabian Walter. *Real-Time Network Functions for the Internet of Things*. Semester Thesis, ETH Zurich, June 2017. doi:10.3929/ethz-b-000234920.

[189] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: A case study. *Int. Journal on Software Tools for Technology Transfer*, 8(6), 2006.

[190] Miao-Miao Wang, Jian-Nong Cao, Jing Li, and Sajal K. Dasi. Middleware for Wireless Sensor Networks: A Survey. *Journal of Computer Science and Technology*, 23(3):305–326, May 2008. doi:10.1007/s11390-008-9135-x.

[191] T. Watteyne, V. Handziski, X. Vilajosana, S. Duquennoy, O. Hahm, E. Baccelli, and A. Wolisz. Industrial Wireless IP-Based Cyber-Physical Systems. *Proceedings of the IEEE*, 104(5):1025–1038, May 2016. doi:10.1109/JPROC.2015.2509186.

[192] T. Watteyne, P. Tuset-Peiro, X. Vilajosana, S. Pollin, and B. Krishnamachari. Teaching Communication Technologies and Standards for the Industrial IoT? Use 6TiSCH! *IEEE Communications Magazine*, 55(5):132–137, May 2017. doi:10.1109/MCOM.2017.1700013.

[193] Samuel Weber, Jan Beutel, Reto Da Forno, Alain Geiger, Stephan Gruber, Tonio Gsell, Andreas Hasler, Matthias Keller, Roman Lim, Philippe Limpach, Matthias Meyer, Igor Talzi, Lothar Thiele, Christian Tschudin, Andreas Vieli, Daniel Vonder Mühll, and Mustafa Yücel. A decade of detailed observations (2008–2018) in steep bedrock permafrost at the Matterhorn Hörnligrat (Zermatt, CH). *Earth System Science Data*, 11(3):1203–1237, August 2019. doi:https://doi.org/10.5194/essd-11-1203-2019.

[194] Markus Wegmann. *Reliable 3rd Generation Data Collection*. Master Thesis, ETH Zurich, 2018.

[195] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, December 2002.

[196] Wikipedia. Bin packing problem. *Wikipedia*, October 2019. Page Version ID: 921684516.

[197] Wikipedia. Chinese whispers. *Wikipedia*, October 2019. Page Version ID: 920368233.

[198] Matthias Wilhelm, Vincent Lenders, and Jens B. Schmitt. On the Reception of Concurrent Transmissions in Wireless Sensor Networks. *IEEE Transactions on Wireless Communications*, 13(12):6756–6767, December 2014. doi:10.1109/TWC.2014.2349896.

[199] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: The Training Ground for Internet Congestion-control Research. In *Proceedings of the International USENIX Annual Technical Conference (ATC)*, pages 731–743, Boston, MA, USA, July 2018. USENIX Association.

[200] Lisa Yan and Nick McKeown. Learning Networking by Reproducing Research Results. *SIGCOMM Comput. Commun. Rev.*, 47(2):19–26, May 2017. doi:10.1145/3089262.3089266.

[201] D. Yuan and M. Hollick. Let's talk together: Understanding concurrent transmission in wireless sensor networks. In *38th Annual IEEE Conference on Local Computer Networks*, pages 219–227, October 2013. doi:10.1109/LCN.2013.6761237.

[202] Dingwen Yuan, Michael Riecker, and Matthias Hollick. Making 'Glossy' Networks Sparkle: Exploiting Concurrent Transmissions for Energy Efficient, Reliable, Ultra-Low Latency Communication in Wireless Control Networks. In Bhaskar Krishnamachari, Amy L. Murphy, and Niki Trigoni, editors, *Wireless Sensor Networks*, Lecture Notes in Computer Science, pages 133–149. Springer International Publishing, 2014.

[203] Licong Zhang, Dip Goswami, Reinhard Schneider, and Samarjit Chakraborty. Task- and network-level schedule co-synthesis of Ethernet-based time-triggered systems. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 119–124, January 2014. doi:10.1109/ASPDAC.2014.6742876.

[204] Lixia Zhang, Stephen Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7(5), 1993.

[205] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware Design Experiences in ZebraNet. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 227–238, New York, NY, USA, 2004. ACM. doi:10.1145/1031495.1031522.

[206] Peilin Zhang, Alex Yuan Gao, and Oliver Theel. Less is More: Learning More with Concurrent Transmissions for Energy-Efficient Flooding. In *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous 2017, pages 323–332, New York, NY, USA, 2017. ACM. doi:10.1145/3144457.3144482.

[207] Marco Zimmerling, Federico Ferrari, Luca Mottola, and Lothar Thiele. On Modeling Low-power Wireless Protocols Based On Synchronous Packet Transmissions. In *Proc. of IEEE MASCOTS*, 2013.

[208] Marco Zimmerling, Luca Mottola, Pratyush Kumar, Federico Ferrari, and Lothar Thiele. Adaptive Real-Time Communication for Wireless Cyber-Physical Systems. *ACM Trans. Cyber-Phys. Syst.*, 1(2):8:1–8:29, February 2017. doi:10.1145/3012005.