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

Romain Jacob, Jonas Baechli, Reto Da Forno, Lothar Thiele
ETH Zurich
{rjacob, baechlij, rdaforno, thiele}@ethz.ch

Abstract

Synchronous Transmissions is a technology that combines energy efficiency and reliability for low-power wireless multi-hop networks. But using this technology to design network stacks is a complex task, in part due to the tight timing requirements on the execution of radio operations.

To facilitate the development of protocols based on Synchronous Transmissions, we developed *Baloo*, a flexible network stack design framework, which we present in this paper. We show that *Baloo* is flexible enough to implement a wide variety of network layer protocols, while introducing only limited memory and energy overhead. Most importantly *Baloo* makes Synchronous Transmissions accessible: The software is open source and well documented. We believe that *Baloo* will be an important enabler for a whole new class of Internet of Things applications leveraging the reliability, efficiency, and flexibility of Synchronous Transmissions.

Categories and Subject Descriptors

C.2.2 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Protocols*

General Terms

Design, Performance, Standardization

Keywords

Network protocol design, Network layer protocol, Middleware, Programming Interfaces

1 Introduction

Synchronous Transmissions (ST) is an increasingly used wireless communication technology for low-power multi-hop networks. Popularized by Glossy [23] in 2011, it has been proven to be highly reliable and energy efficient, as illustrated by the EWSN Dependability Competition [45], where all wining solutions in the past three years were based on ST [21, 31, 47].

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* [52]. The synchronization requirements vary from sub-μs to tens of μs, depending on the platform and modulation scheme [52].

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 [22].

Since Glossy [23], many flavours of ST have been proposed to improve performance in terms of reliability, latency, and energy consumption. To be more resilient to strong interference, Robust Flooding [31] is a primitive that modifies the RX-TX sequence from the original Glossy, whereas RedFixHop [20] uses hardware acknowledgements to minimize the number of retransmissions required. Instead, some primitives aim to minimize latency for specific traffic patterns. For example, Chaos [30] 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 [10]. Codecast [35] also targets many-to-many exchange for a larger amount of data. Pando [17] is another primitive focused on high throughput, which uses fountain code and packet pipelining for efficient data dissemination. To save energy, Syncast [36] tries to reduce the radio on time required compared to Glossy. Another recent proposal is Less is More (LiM) [53], a primitive that reduces energy consumption

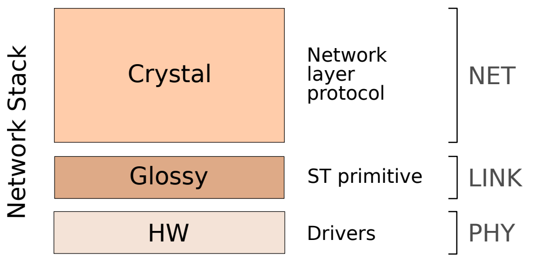

Figure 1. Crystal [27] is a typical example of network stack based on ST. *The implementation of the network layer protocol (Crystal) couples the interface to the underlying ST primitive (Glossy) and the protocol logic,* i.e.*, how long are the communication rounds, which radio channel is used, etc.*

using learning to avoid unnecessary retransmissions during flooding.

All these primitives share the same drawback: due to the very tight synchronization requirements, successful ST requires low-level control of timers and radio events. 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 remains 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) [22], 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 [48], a variation of LWB tailored to event-based data collection. Sleeping Beauty [44] was later proposed to minimize energy consumption for data collection scenarios with many redundant sensor nodes. Time-Triggered-Wireless (TTW) [28] was designed to minimize the end-to-end latency between communicating application tasks. Finally, Crystal [27] has been proposed as a network stack specialized for sporadic data collection. All these network stacks solely rely on Glossy as ST primitive.

The implementation of the network layer protocol (LWB, Sleeping Beauty and Crystal are openly available [1, 2, 6]) often couples the interface with Glossy (the underlying ST primitive) and the *protocol logic* (*i.e.*, when packets are scheduled, which nodes are sending and receiving, what the packets should contain, etc.). In principle however, the same protocol logic could leverage the benefits from *different* primitives. For example, an LWB network could start using Robust Flooding [31] in case of high interference, then later switch back to Glossy [23] for better time synchronization. If the nodes should be reprogrammed, the software update can be quickly disseminated using Pando [17]. Designing a modular network stack supporting multiple ST primitives adds a new level of complexity.

To facilitate the network stack design and let the same network layer protocol use different ST primitives, it makes sense to separate the concern of the timely execution of the primitives from the implementation of the protocol logic. One common approach to achieve such separation of concerns is to use a *middleware* as part of the network stack.

Challenges. 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 [37], "*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? This problem can be formulated by the following challenges:

C1 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.

C2 The middleware must enable the implementation of a large variety of network layer protocols.

C3 The middleware must enable one network layer protocol to use multiple ST primitives and switch between them at runtime.

While at the same time:

C4 The middleware must guarantee to respect the time synchronization requirements for ST (from sub-μs to tens of μs [52]).

Contributions. We have addressed these challenges with *Baloo*[1], a flexible design framework for low-power network stacks based on ST. *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 (see Fig. 2).
- We present the design of a middleware layer that meets the challenges **C1** to **C4**. This middleware is 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) [22], Sleeping Beauty [44], and Crystal [27].
- We illustrate the portability of *Baloo* by providing implementations for two platforms – the CC430 SoC [25] and the old but still heavily used TelosB mote [9].
- We demonstrate that *Baloo* induces only limited performance overhead (memory usage, radio duty cycle) compared to the original implementations.

This paper *is not* meant to detail all the inner mechanisms of *Baloo*, but rather present the core concepts of the framework. *Baloo* is open source and the complete technical documentation is available online [4].

[1] *Baloo* provides the "bare necessities" for a ST network stack design.

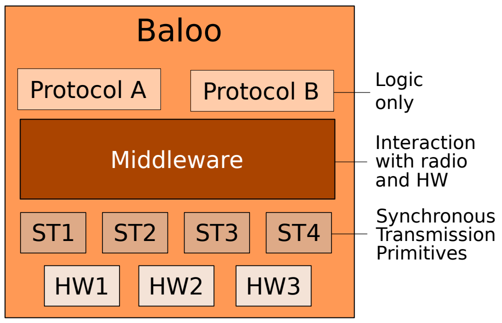

Figure 2. *Baloo* is a flexible design framework for network stacks based on Synchronous Transmissions (ST). It is based on a middleware layer that separates the concern of timely execution of ST primitives from the implementation of the protocol logic. *Thanks to this additional layer of abstraction, Baloo flexibly supports multiple ST primitives and significantly reduces the efforts required to implement network layer protocols compared to traditional stacks, like LWB [22] or Crystal [27].*

In the rest of this paper, we first present the general concepts of *Baloo* (Sec. 2), then discuss the design and implementation of the underlying middleware (Sec. 3). In Sec. 4, we briefly describe the set of advanced features provided by *Baloo* for designing network stacks, before the evaluation in Sec. 5, where the usability and performance of our proposal are discussed. Sec. 7 discusses requirements and limitations of *Baloo*, followed by a section on our lessons learned from this work (Sec. 8). We present some related works in Sec. 9 before drawing some conclusions and perspectives.

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 (Sec. 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 (Fig. 2) that isolates the network layer (where the protocol logic is implemented) from the lower layers (in particular the management of the radio and timers).

Why Synchronous Transmissions? ST is a wireless communication technique recognized to be reliable, fast, and energy efficient. ST primitives communicate using so-called *floods*, which realize an 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 [23] or Robust Flooding [31]) 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 [23] as default ST primitive, but it also supports other primitives, *e.g.*, Chaos [30]. In principle, *Baloo* is compatible with arbitrarily many other primitives (see Sec. 3.5), thus addressing **C3**.

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 work against achieving **C2**, 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 (Sec. 5 and 7).

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 [23], which is also used to synchronize the whole network.

Baloo is very flexible (**C1**); 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.

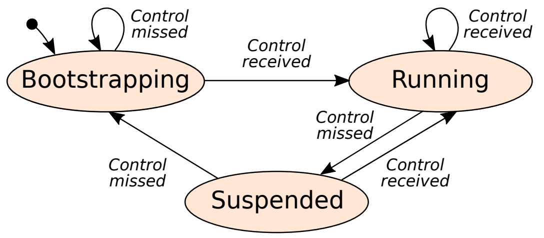

Figure 3. The middleware in *Baloo* implements a minimal state machine, sufficient to capture the desired behaviour of a node at physical layer. *A node either executes normally (Running), stays synchronized but does not participate to the communication rounds (Suspended), or continuously listens for an incoming control packet (Bootstrapping).*

- 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 management, state keeping, etc.) is implemented entirely within the callback functions[2]. The middleware interface is illustrated in Fig. 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.

Wrapping-up. These concepts form the core of *Baloo* and address the challenges of a flexible network stack design (**C1-C3**). Additional concepts are required to ensure that the timing requirements of ST are met (**C4**), which is the focus of the next section.

3 Implementing the Concepts

The previous section described the general concepts of *Baloo*, and discussed how they meet challenges **C1-C3**, presented in the Introduction. This section details how we implemented these concepts to address **C4** and complete the design of *Baloo*.

3.1 Contiki-NG as Operating System

Baloo relies on the availability of ST primitives (*e.g.*, Glossy [23], Chaos [30], etc.). Most openly available primitives use Contiki [18] as the underlying operating system, which made Contiki an obvious choice to implement *Baloo*. We have ported these primitives to the latest version of the OS: Contiki-NG [5] [3].

Contiki is a cooperative multi-threaded OS, tailored for resource-constrained devices in the Internet of Things. The middleware layer is implemented as the "master" protothread [19], 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 Sec. 3.3).

[2]The different callbacks are further described in Sec. 3.3.

[3]V4.2, released in November 2018

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 Fig. 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 Sec. 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 Sec. 4.1).

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 (**C1**). There are five different callbacks, which have specific purposes and are executed by the middleware at precise points in time, as illustrated in Fig. 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 Sec. 4.

3.4 Achieving Timeliness of Execution

The callback functions enable the flexible interaction between the network layer and the middleware, which is key to address **C1** and **C2**. However, this also inherently couples the two software components, thus challenging the timely execution of the middleware and compromising **C4**.

Indeed, the callbacks execute between communication slots or between rounds (see Fig. 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

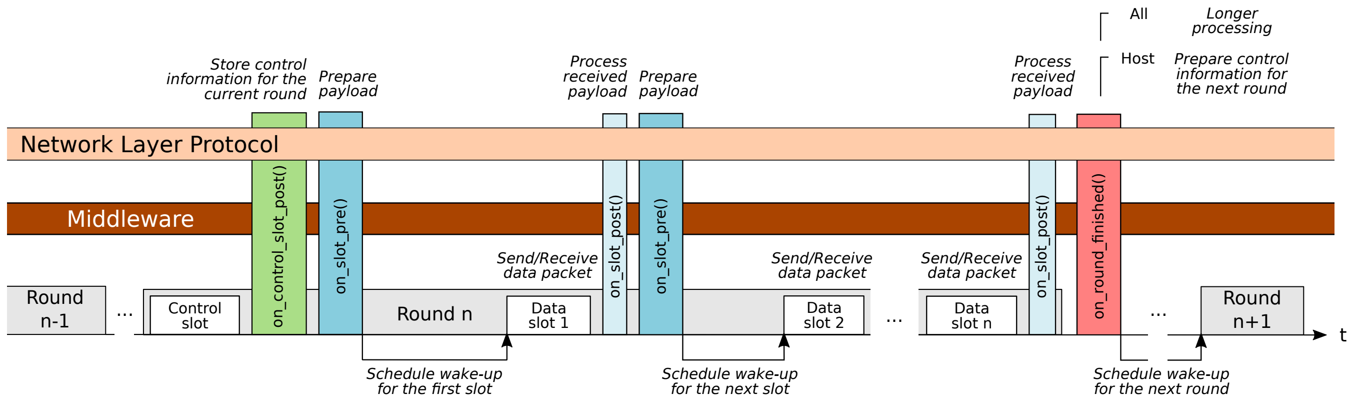

Figure 4. The protocol logic, *i.e.*, the handling of application payloads and the definition of the desired control parameters, is implemented in callback functions. These callbacks are triggered by the middleware before and after each slot and at the end of a round. The middleware schedules the wake-up of the radio core and executes the ST primitives.

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 (**C4**) *under the condition that the callbacks have enough time to complete their execution*.[4] 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 **C4** is met. Guidelines for setting such parameters (and in general: how-to use *Baloo*) are part of the online documentation [4].

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 (**C3**). 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 [52]).

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[5].

Using this approach, *Baloo* currently supports two ST primitives, Glossy [23] and Chaos [30], 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.

4 Advanced Features

In Sec. 2 and 3 we presented the general concepts of *Baloo* and how we implemented them to meet challenges **C1-C4**. 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 which have been used in previous protocols and proved themselves useful. We briefly present these features in this section.

4.1 Additional Functionalities

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.

[4]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 Sec. 4.1).

[5]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.

This feature is used *e.g.*, in Crystal [27].

Advanced state machine. The middleware in *Baloo* implements a minimal but sufficient state machine composed of three states (*Running*, *Suspended*, *Bootstrapping*; see Sec. 3.2 and Fig. 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 [22].

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() overruns).

4.2 Advanced Scheduling Functionality

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 [52].

Contention slots are used in many protocols, including LWB [22] and Crystal [27].

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 [27] 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 [28] or Crystal [27].

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 Sec. 4.1).

This feature is used *e.g.*, in Sleeping Beauty [44].

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 [27].

4.3 Control of 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 [27].

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

5 Experimental Evaluation

We presented *Baloo*, a flexible design framework for low-power network stacks based on ST; we now evaluate our proposal. First, we look into qualitative aspects. We argue that *Baloo* is indeed practical to use 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.

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 [27], Sleeping Beauty [44], and LWB [22]. 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* (see Sec. 4).
- 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 [4]. Naturally, our re-implementations of Crystal, Sleeping Beauty, and LWB are also available [4]. We intend to push *Baloo* to the public Contiki-NG repository [5], thus making it readily available to a large set of users.

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 Sec. 7). *Baloo* is readily available on two platforms, the CC430 SoC [25] and the TelosB mote [9]. Thanks to the abstraction provided by the middleware, the network layer protocol implementations using *Baloo* are *platform agnostic*. The exact same network layer software can be used to compile binaries for any target platform supported by *Baloo*. We argue that these elements, altogether, show the usability of *Baloo*.

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 [27], Sleeping Beauty [44], and LWB [22]. 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 [32] as it is the only public testbed featuring both CC430 SoC [25] and TelosB motes [9], 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 [22,27,44].

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 [27]).

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 [44].

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 1 to 4. Before discussing each of the metrics in more details, some comments are useful.

- Although the original code of Sleeping Beauty is openly available [1], 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 1 and 2 were read from Fig. 9 in [44].
- The original LWB paper presents results from an implementation on TelosB, but the available code from the authors is for the CC430 SoC [2]. 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 [27] 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 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[6], the sink only rarely receives packets in these rounds, thus resulting in poor PRR.

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 2.

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

The case of Crystal is interesting. We tried to reproduce some experiments from the original paper, and our results on the same platform (TelosB) show 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. It turns out that in our experiments

[6]Successful contention slots rely on capture effect, see Sec. 4

Table 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	100	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 2. Average radio duty cycle (DC) across all nodes but the data sink, expressed in percentages (%) [7]

	Crystal			Sleeping Beauty			LWB	
	$U = 0$	$U = 1$	$U = 20$	12.5%	25%	50%	$IPI = 30s$	$IPI = 4s$
Original paper	0.367	0.487	2.89	≈ 0.2	≈ 0.3	≈ 0.7	na	na
Native code	0.844	0.959	2.833	x	x	x	0.79	5.715
Baloo on TelosB	1.013	1.244	3.937	0.082	0.115	0.283	0.976	6.646
Baloo on CC430	0.355	0.494	2.64	0.07	0.106	0.27	0.914	6.007

Table 3. Estimate of the binary size of the network layer protocol code, expressed 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 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
Baloo/Native	0.58	1.06	0.55

on TelosB, Crystal consistently detects possible interference and (often needlessly) prolongs the communication rounds, thus increasing the DC. Assuming that the noise detection settings were correctly reported in the original paper [27] (threshold of −60 dBm in Sec. 6) this would indicate that the wireless environment is significantly different on Flocklab and on the private testbed used by the Crystal's authors. Interestingly, the original paper results are comparable to our CC430 SoC re-implementation. This could be explained by the fact that this radio uses the sub-GHz band, where interference is less present than on the 2.4 GHz band.

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 [44], 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. This remains an open question.

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 reports the difference between the total and "hello-world" binary sizes, a rough *estimate* of the memory required by the network layer protocol implementation[8].

Actually, the memory size of our re-implementations is comparable to that of the native codes. This 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 4 show that using *Baloo* can sig-

[7]For Sleeping Beauty, reported values exclude the bootstrapping phase.

[8]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.

nificantly 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.

To encourage ongoing efforts towards more transparency and repeatability in our field [13], we make all the data collected during our experiments publicly available [4], together with the post-processing artefacts we used to produce the results presented in this paper.

6 Putting *Baloo* to the Test

We are now using *Baloo* for our own research projects, starting with the 2019 EWSN Dependability Competition [7]. In this year's scenarios, each solution should perform well across a wide range of input parameters (*e.g.*, data rates or payload sizes), which demands the network stack to be adaptive. The flexibility of *Baloo* is very useful. 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 [23], one can efficiently collect acknowledgements from all destinations using Chaos [30].

We use the competition as an opportunity to practically evaluate the usability of *Baloo*: our competition solutions are developed by master students who did not have any prior knowledge of *Baloo*, nor ST. It will be interesting to see how well their solutions perform compared to "traditional" network stacks designed by experienced teams.

7 Requirements and Limitations

We argued that *Baloo* is a usable, flexible, and performant design framework (Sec. 5). To complete the description, we now detail the hardware and software requirements and discuss the portability and limitations of the framework.

7.1 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 [46] or the OpenMoteB [39] platforms would work as well. It would eventually be even more flexible, thanks to 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 that 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.

7.2 Portability

Baloo itself has limited hardware and software requirements (Sec. 7.1). 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 very low. A guide on "how-to-port" *Baloo* is included in the documentation [4].

7.3 Limitations

Baloo is a framework that facilitates the design of ST-based network stacks by providing some level of abstraction. We showed in Sec. 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.

This is a rather short list.

We presented in Sec. 4 a set of features that enrich the possibilities offered by *Baloo*. The feature set may not be complete, but it already offers a lot of options; and it can be extended in the future, if necessary. To the best of our knowledge, to date in the literature there does not exist a ST-based network layer protocol that is incompatible with *Baloo*. This is simply 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).

8 Discussion

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, these solutions should 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.

The availability and reusability of research artefacts is an increasingly important topic. Scientific societies start to incentivize reusable research; the ACM introduced in 2018 a badging system [8] which starts being used in leading conferences like CoNEXT 2018 [3]. We believe this is really important to keep in mind: if we want to see our research being used and impactful beyond academia, we must strive to make it more accessible. This is why *Baloo* is open source and accompanied with an extensive documentation of the code and how to use it [4].

9 Related Work

As mentioned in the introduction, the idea of middleware for Wireless Sensor Networks (WSN) has been around for more than a decade [14, 37, 43, 51]. 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 [38, 42] 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. [37] reviewed the literature focusing more on WSN: proposals include for example Impala [33] which explicitly address the problem of fault tolerance in mobile networks. Programming abstractions like TinyDB [34], RUNES [15], or TinyLIME [16] 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.*, [12, 50] for recent surveys). [26] 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) [29] and λ-MAC [41]. 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 [22,27,28,44,48,49], and almost no research has been conducted to propose a flexible design framework (*e.g.*, comparable to MLA [29]) but tailored to ST.

One notable exception is A^2 [10]. The approach of this work is very similar to ours; the authors try to facilitate the design of ST-based communication using a middleware component, which they called Synchrotron. 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 [23] or Chaos [30]) 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* is a natural next step towards a fully flexible and configurable network stack based on ST.

10 Conclusions

In this paper, we presented *Baloo*, a flexible design framework for low-power wireless network stacks based on ST. We illustrated its usability by re-implementing three well-known network stacks: the Low-power Wireless Bus (LWB) [22], Sleeping Beauty [44], and Crystal [27], and we showed that using *Baloo* induces only limited performance overhead in terms of radio duty cycle and memory usage. The code of *Baloo* is openly available and is accompanied by a detailed documentation of its features and how to use them [4]. Our re-implementations of Crystal, Sleeping Beauty, and LWB are also available [4]. We intend to push *Baloo* to the public Contiki-NG repository [5] in the near future. We argue that *Baloo* is a versatile and usable framework with limited overhead, hence effectively facilitating the design of performant ST-based network stacks. We believe it will be an important enabler for the development of future real-world applications leveraging state-of-the-art ST technology.

The usability of *Baloo* would be further increased by adding support for simulation in Cooja [40]. This is one of our priorities. Furthermore, an implementation of Glossy for the CC2538 SoC is available [24]. It would be interesting to port *Baloo* to this radio chip, which equips newer motes like the OpenMoteB [39]. As mentioned in the Related Work section, including A^2 [10] in the list of supported primitives would be a natural next step. In parallel to the development of *Baloo*, another team developed a ST primitive for Bluetooth 5 [11]. This could extends the usability of *Baloo* (and thereby LWB, Crystal, etc.) to Bluetooth technology.

As a side benefit, *Baloo* provides interesting perspectives for benchmarking protocols. For a given platform, protocol concepts can be compared more easily (as they are quick to implement), and more accurately (as they rely on the same implementation of the underlying ST primitives). Conversely, one can evaluate the impact of different platforms on the performances of a given network layer protocol. Benchmarking low-power communication protocols has recently gained some momentum in the community [13]; *Baloo* comes in timely and can be an interesting tool to facilitate benchmarking of solutions based on ST.

11 References

[1] Sleeping Beauty. https://github.com/csarkar/sleeping-beauty, Aug. 2016.

[2] Low-Power Wireless Bus (LWB). https://github.com/ETHZ-TEC/LWB, Sept. 2017.

[3] ACM CoNEXT 2018. https://conferences2.sigcomm.org/co-next/2018/#!/program, Oct. 2018.

[4] Baloo. http://www.romainjacob.net/research/baloo/, Dec. 2018.

[5] Contiki-NG. http://contiki-ng.org/, Nov. 2018.

[6] Crystal. https://github.com/d3s-trento/crystal, July 2018.

[7] EWSN 2019 Dependability Competition. http://ewsn2019.thss.tsinghua.edu.cn/competition-scenario.html, Feb. 2019.

[8] ACM. Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-badging, Apr. 2018.

[9] Advanticsys. MTM-CM5000-MSP 802.15.4 TelosB mote Module. https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html.

[10] B. Al Nahas, S. Duquennoy, and O. 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.

[11] B. Al Nahas, S. Duquennoy, and O. Landsiedel. Concurent Transmissions for Multi-hop Bluetooth 5. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, To appear.

[12] P. Bartolomeu, M. Alam, J. Ferreira, and J. Fonseca. Survey on low power real-time wireless MAC protocols. *Journal of Network and Computer Applications*, 75:293–316, Nov. 2016.

[13] C. A. Boano, S. Duquennoy, A. Förster, O. Gnawali, R. Jacob, H.-S. Kim, O. Landsiedel, R. Marfievici, L. Mottola, G. P. Picco, X. Vilajosana, T. Watteyne, and M. Zimmerling. IoTBench: Towards a Benchmark for Low-power Wireless Networking. In *1st Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench 2018)*, page 6, Apr. 2018.

[14] 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, Sept. 2007.

[15] 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, Sept. 2005.

[16] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming Wireless Sensor Networks with the TeenyLime Middleware. In R. Cerqueira and R. H. Campbell, editors, *Middleware 2007*, Lecture Notes in Computer Science, pages 429–449. Springer Berlin Heidelberg, 2007.

[17] W. Du, J. C. Liando, H. Zhang, and M. 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.

[18] 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, Nov. 2004.

[19] A. Dunkels, O. Schmidt, T. Voigt, and M. 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.

[20] 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, Nov. 2016.

[21] A. Escobar, F. Moreno, A. J. Cabrera, J. Garcia-Jimenez, F. J. Cruz, U. Ruiz, J. Klaue, A. Corona, D. Tati, and T. 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.

[22] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-power Wireless Bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 1–14, New York, NY, USA, 2012. ACM.

[23] 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, Apr. 2011.

[24] K. Hewage. Low-power Wireless Bus (LWB) for TI's CC2538 Soc for Contiki OS.: Kasunch/lwb-cc2538. https://github.com/kasunch/lwb-cc2538, July 2018.

[25] T. Instruments. CC430F6137 16-Bit Ultra-Low-Power MCU. http://www.ti.com/product/CC430F6137.

[26] 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.

[27] T. Istomin, M. Trobinger, A. L. Murphy, and G. P. 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.

[28] R. Jacob, L. Zhang, M. Zimmerling, J. Beutel, S. Chakraborty, and L. Thiele. TTW: A Time-Triggered-Wireless Design for CPS [Extended version]. *arXiv:1711.05581 [cs]*, Nov. 2017.

[29] K. Klues, G. Hackmann, O. Chipara, and C. 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.

[30] O. Landsiedel, F. Ferrari, and M. Zimmerling. Chaos: Versatile and Efficient All-to-all Data Sharing and In-network Processing at Scale. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 1:1–1:14, New York, NY, USA, 2013. ACM.

[31] R. Lim, R. Da Forno, F. Sutton, and L. 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.

[32] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, IPSN '13, pages 153–166, New York, NY, USA, 2013. ACM.

[33] T. Liu and M. 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.

[34] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, Mar. 2005.

[35] M. Mohammad and M. C. 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.

[36] M. Mohammad, M. Doddavenkatappa, and M. C. Chan. Improving Performance of Synchronous Transmission-Based Protocols Using Capture Effect over Multichannels. *ACM Trans. Sen. Netw.*, 13(2):10:1–10:26, Apr. 2017.

[37] L. Mottola and G. P. Picco. Middleware for wireless sensor networks: An outlook. *Journal of Internet Services and Applications*, 3(1):31–39, May 2012.

[38] M. Onderwater. An overview of centralised middleware components for sensor networks. *International Journal of Ad Hoc and Ubiquitous Computing*, 21(3):180–193, Jan. 2016.

[39] OpenMote. OpenMote B. http://www.openmote.com/product/openmote-b-single/.

[40] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pages 641–648, Nov. 2006.

[41] T. Parker, G. Halkes, M. Bezemer, and K. Langendoen. The λMAC Framework: Redefining MAC Protocols for Wireless Sensor Networks. *Wirel. Netw.*, 16(7):2013–2029, Oct. 2010.

[42] 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, Feb. 2016.

[43] K. Römer. Programming paradigms and middleware for sensor networks. *GI/ITG Workshop on Sensor Networks*, pages 49–54, 2004.

[44] 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, Oct. 2016.

[45] M. Schuß, C. A. Boano, M. Weber, and K. 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, 2017. Junction Publishing.

[46] N. Semiconductors. nRF52840. https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840.

[47] P. Sommer and Y.-A. 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.

[48] F. Sutton, R. Da Forno, D. Gschwend, T. Gsell, R. Lim, J. Beutel, and L. 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.

[49] 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.

[50] R. Teles Hermeto, A. Gallais, and F. 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, Dec. 2017.

[51] M.-M. Wang, J.-N. Cao, J. Li, and S. K. Dasi. Middleware for Wireless Sensor Networks: A Survey. *Journal of Computer Science and Technology*, 23(3):305–326, May 2008.

[52] 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, Oct. 2013.

[53] P. Zhang, A. Y. Gao, and O. 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.