



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Institut für
Technische Informatik und
Kommunikationsnetze

Low-Power Network Design: Work Hard, Play Hard - Data Collection

Semester Thesis

Anna-Brit Schaper

schapera@student.ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:

Romain Jacob

Reto Da Forno

Prof. Dr. Lothar Thiele

January 19, 2019

Acknowledgements

I would like to thank my supervisor Romain Jacob for his highly valued input and support throughout this project. Furthermore, I am grateful to Jan Müller for the collaborative and productive working relationship we had. This project has benefitted a great deal from many long and intensive discussions with these two individuals.

Finally, I would like to express my gratitude to Reto Da Forno for providing help on FlockLab and Baloo, Carlo Alberto Boano and Markus Schuss of TU Graz for their work and support on the EWSN Dependability Competition infrastructure, and Prof. Dr. Lothar Thiele for enabling me to work on this project.

Abstract

With the rapid expansion of the Internet of Things, an increasing number of applications relies on reliable, energy-efficient, and low-latency communication between small networked wireless devices. A large number of wireless communication protocols dealing with these challenges have been proposed in recent years. Their performance typically varies greatly with the application and environment they are used in. The EWSN Dependability Competition aims to achieve some comparability between solutions by evaluating a protocols under the same conditions. To encourage protocols that are as generally applicable as possible, the 2019 edition introduces some controlled variation to the evaluation conditions. Traditionally, designing and implementing a communication protocol that provides good performance in such a complex scenario required a deep understanding of the entire communication stack used by the nodes as well as a large amount of development time.

Our goal in this project is to show that this is no longer the case thanks to Baloo, a novel design framework for communication protocols based on Synchronous Transmissions. We demonstrate this by participating in the 2019 EWSN Dependability Competition with our own protocol based on Baloo, which was developed and implemented within just a few weeks by students with little prior experience in the field of wireless networks. Early testing during the preparation phase of the competition has indicated that our protocol should be able to deliver comparable performance to other competing solutions in terms of reliability, latency and energy consumption across some input conditions.

Contents

Acknowledgements	i
Abstract	ii
Abbreviations	v
1 Introduction	1
1.1 EWSN Dependability Competition	1
1.2 The Data Collection Scenario	2
1.3 Our Approach	3
2 Background	4
2.1 Synchronous Transmissions	4
2.2 Baloo	5
2.3 Approaches to Data Collection	7
3 Protocol	11
3.1 Scenario Breakdown	11
3.2 Setup Phase	16
3.3 Periodic Case	20
3.3.1 Schedule Initialization Phase	20
3.3.2 Normal Operation Phase	28
3.4 Aperiodic Case	34
3.5 Optimization Parameters	37
4 Evaluation	42
4.1 Protocol Performance	42
4.2 Usability of Baloo	45

CONTENTS	iv
5 Conclusion and Future Work	46
5.1 Conclusion	46
5.2 Future Work	46
Bibliography	48

Abbreviations

ACK acknowledgement

EEPROM Electrically Erasable Programmable Read-Only Memory

EWSN International Conference on Embedded Wireless Systems and Networks

GPIO general-purpose input/output

IoT Internet of Things

ISM industrial, scientific and medical

ISR interrupt service routine

RF radio frequency

NACK negative acknowledgement

ST Synchronous Transmissions

WSN wireless sensor network

Introduction

This chapter introduces the main motivations and goals behind the semester project. For the project, we competed in the 2019 EWSN Dependability Competition, on which we provide some background in Section 1.1. We competed in the data collection category, which is explained briefly in Section 1.2. Our basic approach to designing our competition protocol is explained in Section 1.3.

1.1 EWSN Dependability Competition

Low-power wireless sensor networks (WSNs) have been an active area of research over the past two decades. With the increasing penetration of the Internet of Things (IoT), interest in dependable wireless communication protocols has grown even further. A large number of protocols have been proposed in recent years. However comparing their performance is difficult. For a fair comparison, one would need to test them using the same

- network,
- environmental conditions,
- application and
- performance metrics.

The International Conference on Embedded Wireless Systems and Networks (EWSN) Dependability Competition [1], which has taken place annually since 2016, attempts to establish such comparability by benchmarking protocols under the same settings and comparable environmental conditions. Competing teams implement their protocols based on a description of the evaluation scenario. The performance of the implementation is measured in terms of reliability, latency and energy consumption.

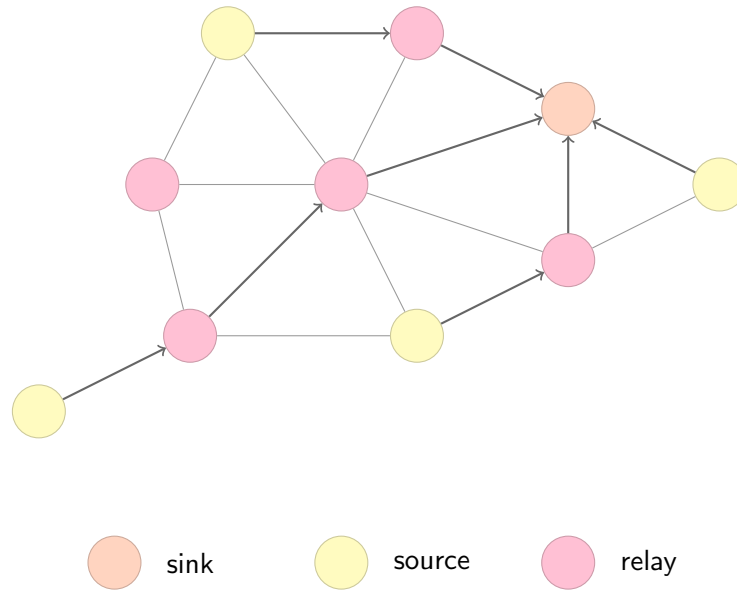


Figure 1.1: Illustration of the data collection scenario.

However, testing protocols under such specific known conditions has the drawback that the results are typically poor indicators for performance in real-life situations, as it encourages to overfitting to the evaluation scenario. To mitigate this, the 2019 edition of the Dependability Competition [2] will test competing solutions on a wide range of input parameters, whose exact values are unknown to the competitors prior to the evaluation.

1.2 The Data Collection Scenario

The 2019 EWSN Dependability Competition [2] considers two competition scenarios. The first is data collection, which is the scenario this project is focused on. The second is data dissemination.

In the data collection scenario, a fixed number of source nodes needs to transmit data to a single sink node, as illustrated by Figure 1.1. Further nodes in the network may act as relay nodes. More details on the exact conditions are given in Section 3.1. A real-world application of this scenario would be a network where a central controller node needs to monitor the conditions at a number of sensor nodes spread across a wireless network.

1.3 Our Approach

Our primary goal in this project was to design a protocol for the data collection scenario of the 2019 EWSN Dependability Competition [2] that performs as well as possible under a number of input parameters in terms of reliability, end-to-end latency and energy consumption.

The project's second objective was to test the usability of Baloo [3], a novel design framework for wireless communication protocols based on Synchronous Transmissions. Specifically, we wanted to find out whether Baloo would enable relative WSN-novices to design a low-power, wireless, multi-hop communication protocol with acceptable performance under a given set of conditions in a relatively short time frame of approximately 10 weeks.

The first step towards these goals was to familiarize ourselves with the Baloo, the concept of Synchronous Transmissions and the basic building blocks typically used in the design of collection protocols. An introduction to this technical background is presented in Chapter 2. We then proceeded to analyze the competition scenario in more detail and design a protocol tailored to it, as described in Chapter 3. Finally, we ran some initial tests using our protocol to get an idea of how it would perform in the final competition (see Chapter 4). While our final evaluation of our performance will depend on the results of the competition, which are not yet known to us, we draw some initial conclusions and give an outlook on potential improvements in Chapter 5.

Background

This chapter provides some background information on the technologies we are using in our protocol. Section 2.1 gives a short introduction to Synchronous Transmissions (ST) primitives with a focus on Glossy in particular. The Baloo design framework, briefly introduced in Section 2.2, is based on ST. In Section 2.3 we show a structure of some further techniques that are sometimes used in data collection scenarios.

2.1 Synchronous Transmissions

The term Synchronous Transmissions (ST) describes a collection of flooding-based communication techniques used in wireless networks. Flooding is a strategy commonly used in wired networks, where each node forwards every message it receives to all of its neighbors, with the exception of the neighbor it received the message from. In wireless networks, reliable flooding is more difficult to implement due to the hidden node problem, as illustrated by Figure 2.1. ST techniques avoid this problem by using tightly synchronized concurrent transmissions. Constructive interference and the capture effect enable the reception of such transmissions. The use of concurrent transmissions also reduces the time required to flood a network. Compared to solutions based on static routing, the use of flooding makes ST more flexible and robust against topology changes. The winning implementations of the previous editions of the EWSN Dependability Competition [1] were all based on ST [4][5][6], which demonstrates the high potential of ST techniques in reliable low-power wireless multi-hop communication.

Glossy [7] is a widely used ST primitive. It is also the primitive we are using in our protocol to profit from its great flexibility and out-of-the-box reliability (in the absence of jamming). The basic structure of a Glossy flood is shown in Figure 2.2 for a network of radius 3. Nodes which are 1 hop away from the initiator receive the message directly from the source. They then proceed to re-

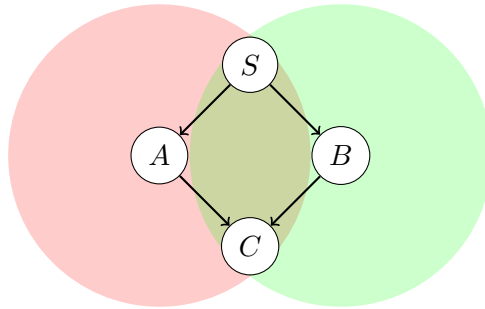


Figure 2.1: Illustration of the hidden node problem in wireless flooding. Node S initializes the flood by sending a packet to A and B . A and B then retransmit the packet. However, because they are positioned out of each other's ranges, they will not realize if their messages collide at C , which is in range of both A and B . Thus, C may never receive the packet. ST avoids this by ensuring A and B transmit the message synchronously, which allows C to receive the packet through constructive interference.

transmit the message, which reaches both the source and the nodes which are 2 hops away from it. Now, the nodes 2 hops away from the source can retransmit the message. This retransmission can be received by the nodes 1 and 3 hops away from the initiator. Under ideal conditions, all nodes in the network should now have received the message. However, to improve the reliability, each node retransmits the packet a set number of times, 3 times in this example. Thus, while the nodes 2 hops away from the initiator perform their first retransmission, the source also retransmits the message. The process continues this way until a maximum number of transmissions is reached at each node. The maximum number of transmissions introduces a tradeoff between energy and reliability.

2.2 Baloo

Baloo [3] is a novel design framework for ST-based communication protocols in wireless multi-hop networks. It provides a middleware which separates the design of network layer protocols from lower-level communication primitives. Baloo's goal is to reduce the complexity of designing ST-based wireless communication protocols while retaining enough flexibility to design well-performing solutions for a wide range of applications.

The framework organizes communication in *rounds*, which are further divided into communication *slots*, as illustrated by Figure 2.3. A dedicated *host* node dictates the scheduling and structure of these rounds by transmitting a *control packet* at the beginning of each round. The control packet is used by the lower layers of Baloo to provide the tight time synchronization required for the under-

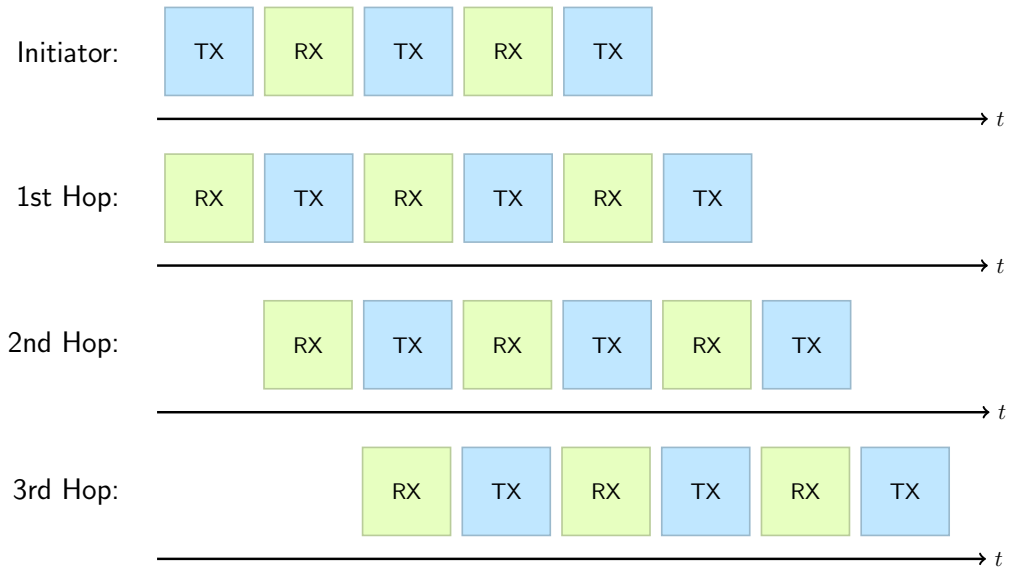


Figure 2.2: Simplified example of a Glossy flood in a network where the source node, i.e. the initiator of the flood, has radius 3. The number of transmissions is set to 3.

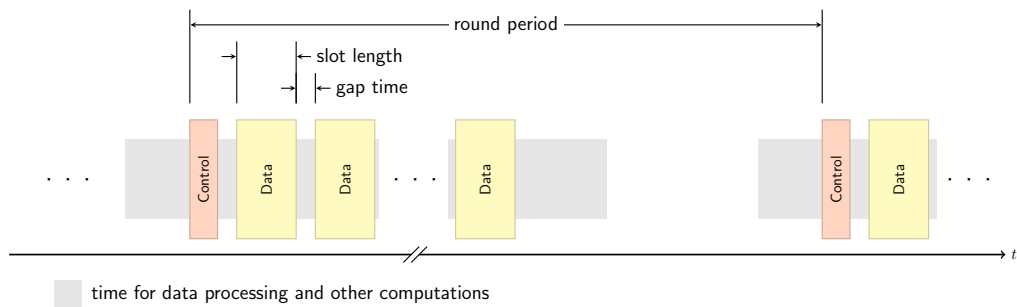


Figure 2.3: Overview of the basic structure of Baloo. Communication is organized in rounds. A round is divided into communication slots. Nodes can stay turned on for some additional processing around these slots, but go to sleep between rounds. The first slot in each round is reserved for a control packet sent by a dedicated host node.

lying ST primitive. The contents of the control packet may vary between rounds. The control packet may be used to distribute the following information:

- The *round period*, i.e. the period of time between the start of the current round and the start of the next round.
- The *schedule* for the round, i.e. how many slots it consists of and which nodes have permission to transmit in which slot. Each slot is either assigned to one specific node or left open for contention.
- The *ST primitive* used in the communication slots.
- Parameters required for the correct operation of the underlying ST primitive, namely
 - the *maximum payload size* to be transmitted per slot and
 - the *number of retransmissions* within the ST flood.
- The *slot length*, which needs to be sufficient to provide enough time for the underlying ST primitive to transmit the payload under the provided parameters.
- The *gap time*, i.e. the time between slots. This needs to provide enough time for a node to complete any processing it might need to perform between slots, such as preparing the payload.
- Any other user-defined information, contained in a fixed number of *user bytes*.

The host does not need to transmit all these parameters with each control packet. If the application and protocol design allows it, nodes can set and update them independently. In this case, it is the responsibility of the protocol designer to ensure the schedule and round configuration are consistent across the nodes. Reducing the size of the control packet and having the nodes operate as independently as possible can result in large energy savings. However, the control packet cannot be completely skipped as it is required to uphold time synchronization.

Nodes can use preconfigured time periods before and after rounds as well as between slots for processing data, preparing it for transmission performing any other computations that might be necessary for the application.

2.3 Approaches to Data Collection

Before starting to implement our own protocol, we investigated commonly used building blocks of existing collection protocols. We attempted to structure these

building blocks based on the communication layer they can be associated with. However, in WSNs communication protocols often cross layers. Thus, the result is only a rough structure, which is shown in Figure 2.4 for the network layer and Figure 2.5 for the link layer.

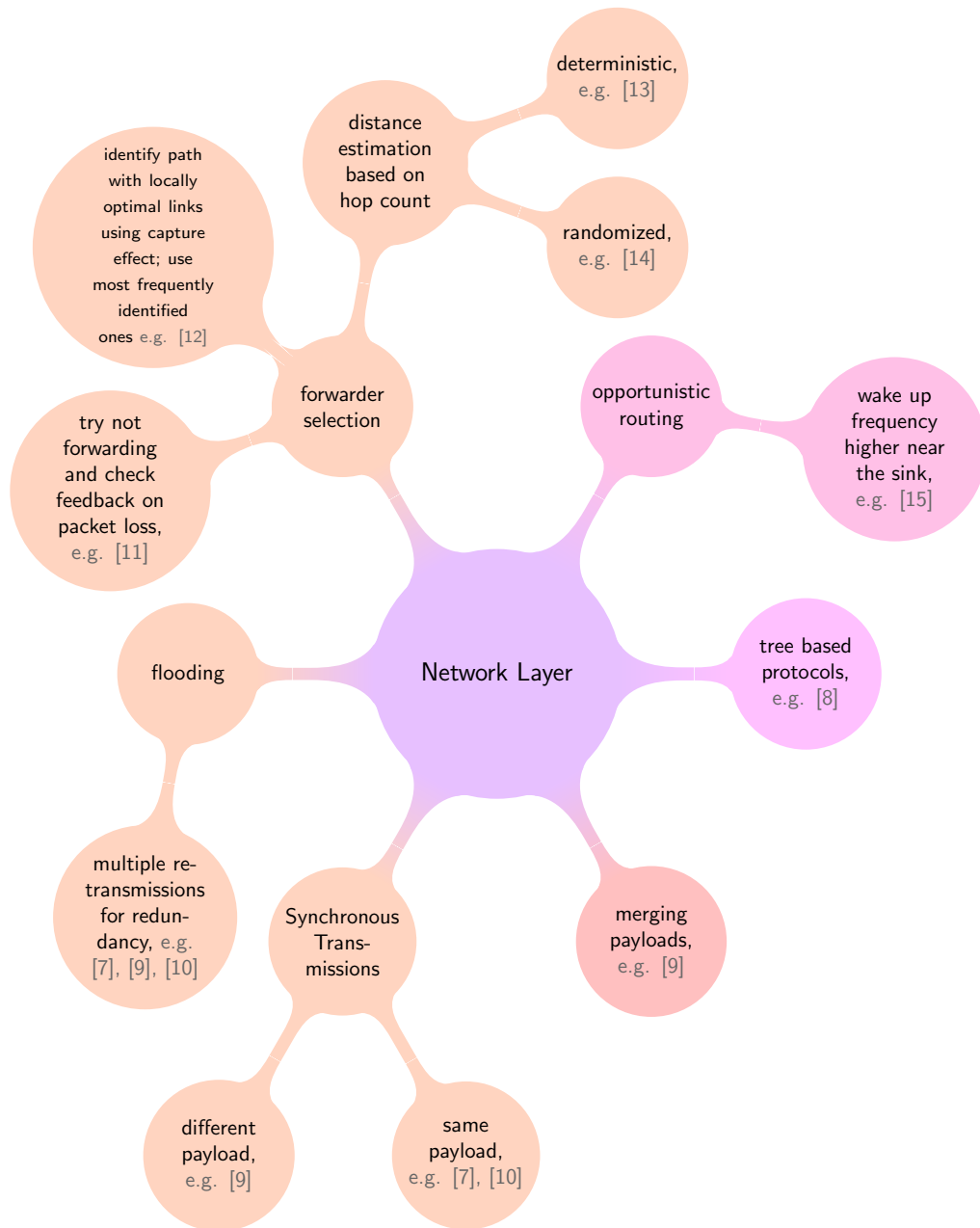


Figure 2.4: Overview of network layer techniques useful in wireless collection scenarios.



Figure 2.5: Overview of link layer techniques useful in wireless collection scenarios.

CHAPTER 3

Protocol

This chapter describes the data collection protocol we developed and the considerations that went into it. The first step in our design process was analyzing the scenario and breaking it down into smaller problems. We then proceeded to identify potential solutions to these problems and implement them on top of the Baloo middleware.

3.1 Scenario Breakdown

The EWSN 2019 Dependability Competition [16] measures protocol performance in terms of three metrics: *reliability*, *latency* and *energy*. The weights of the individual metrics are not known to the competing teams. However, in previous editions of the competition, the reliability metric carried most weight in the overall performance metric.

Reliability: The reliability metric is computed as

$$R = \frac{C}{E_S} \cdot \left(1 - \frac{K_S \cdot S}{E_S}\right) \cdot \left(1 - \frac{K_C \cdot O}{E_S}\right) \quad (3.1)$$

where:

- R = reliability
- C = number of correctly-received messages
- E_S = number of messages detected at the source node
- S = number of superfluous messages
- O = number of messages with causality errors
- $K_S = 2$ (constant)
- $K_C = 2$ (constant)

Superfluous messages are messages detected at the sink node which were not generated at the source. These also include duplicates of previously delivered

messages. Causality errors occur when a message is delivered at the sink before it was generated at the source.

Latency: The latency metric is a combines the mean and median end-to-end latency:

$$L = \frac{L_{\text{mean}} + L_{\text{median}}}{2} \quad (3.2)$$

where:

L = combined latency
 L_{mean} = mean latency
 L_{median} = median latency

Energy: The energy metric is simply the total energy consumption of an evaluation run, excluding the energy consumed during the setup time (explained below):

$$E = E_{\text{total}} - E_{\text{setup}} \quad (3.3)$$

where:

E = energy metric
 E_{total} = total energy consumed [J]
 E_{setup} = energy consumed during the setup time [J]

The data collection scenario defines specific conditions under which our protocol should perform well in those three metrics. We began our scenario analysis by identifying these conditions.

Node types and network topology: The competition testbed consists of 51 TelosB replica nodes. They are spread across multiple floors of an office building with varying density. In the data collection scenario, there are up to eight source nodes and one destination node. The remaining nodes may act as forwarders. Communication between the destination and sources may require multiple hops.

Data structure: The data to be delivered consists of messages of length 2 B to 64 B. The message content is generated randomly and is unique to each message, meaning duplicate messages can be identified by comparing the message contents.

Data generation and delivery: Source nodes generate messages and provide them in Electrically Erasable Programmable Read-Only Memory (EEPROM). A general-purpose input/output (GPIO) pin signals the availability of new data.

To deliver the data, it needs to be written to EEPROM at the destination node. The EEPROM read and write times increase with the message length. We determined experimentally that the maximum times for 64 B messages are below 7 ms for reading and 8 ms for writing. Messages are generated either periodically or aperiodically. In the periodic case, the period is provided as an input parameter. This period can take values between 5 s and 30 min. The actual period between message generations may be longer than the period provided as an input parameter by a few tens of milliseconds. A source node does not necessarily generate its messages at the same time as the other source nodes. In the aperiodic case, data can be generated at any time within the constraints of a minimum and maximum inter-arrival time which are provided as input parameters. There is a lower bound of 5 s on the minimum inter-arrival time.

Permissible frequency range: Only frequencies between 2400 MHz and 2483.5 MHz may be used for communication.

RF interference: Radio frequency (RF) interference may be present in the 2.4 GHz industrial, scientific and medical (ISM) band. Different probabilistic jamming patterns may be applied.

Setup time: At the beginning of an evaluation run, there is a 60 s period during which no data is generated. It is not considered in the calculation of the performance metrics. RF interference may be present during the setup time.

Besides developing a protocol that performs well under these conditions in terms of reliability, latency and energy, our goal in this project was to test the usability of Baloo [3] in the development of such a protocol. Thus, we need to base our protocol on the round-based structure imposed by the Baloo framework. To save energy in Baloo, it is desirable to have nodes determine the round schedule and configuration as independently from each other as possible. This means each node needs to know the values of the scenario parameters that are relevant to the protocol. This knowledge needs to be consistent across all nodes.

The scenario parameters can roughly be divided into three categories:

1. *Parameters whose values remain fixed across all evaluation runs*, such as the general traffic pattern of data collection, the sensor node model, the setup time or permissible frequency range. The protocol should first and foremost be tailored to these parameters, since the success of this will contribute to the success of every single evaluation run.
2. *Parameters whose values vary between evaluation runs*. There are multiple options for catering to the variation in these parameters. The more desirable option in terms of the achievable performance metrics is to opti-

Table 3.1: Overview of the input parameter bounds

	value		unit
	min	max	
<i>periodic & aperiodic case</i>			
sources	1	8	
payload size	2	64	B
<i>periodic case</i>			
period	5	1800	s
<i>aperiodic case</i>			
minimum inter-arrival time	5	∞	s
maximum inter-arrival time	5	∞	s

mize the protocol for every value the parameter takes. However, doing this for each of these parameters would lead to an extremely large complexity, which in turn would make the implementation error-prone and infeasible within the limited amount of space available for the code on the sensor nodes. An alternative option is to accommodate the worst case value of the parameter at each point in the protocol.

Within this group of parameters, a further significant distinction can be made:

- (a) *Parameters whose values are provided by binary patching*, i.e. whose values are injected into the protocol firmware by the testbed. This group consists of the periodicity of the data generation and the parameters listed in Table 3.1. Bounds for these parameters are known at compile time and even their values are known before the execution of the firmware. Early knowledge of these parameters still makes it comparatively easy to optimize the protocol for the values they might take.
 - (b) *Parameters whose values need to be learnt at runtime*, such as the time offset between the data generation schedules of the different nodes in the periodic case. Learning these parameters may carry an overhead, particularly in terms of the energy metric. This overhead needs to be weighed against the benefit of knowing the parameter value.
3. *Parameters whose values vary during an evaluation run*, such as the interference pattern or timing inconsistencies between the nodes. These parameters are the most difficult to cater for as this not only requires learning the parameter once, but also updating as it changes. Updates that are performed inconsistently across nodes or too infrequently can destabilize the protocol if the protocol relies on the knowledge of an up-to-date value

of the parameter. Ideally, the protocol should be able to recover from such inconsistencies. Robustness may also be improved by sending redundant update messages. In some cases another option may be to simply consider the worst case value of the parameter at each point during the evaluation run.

Using this classification, we can proceed to identify some parameters that inform the fundamental structure of our protocol.

First, we take a closer look at the Category 1 parameters. Clearly, we need to design a collection protocol, i.e. one where the source nodes can send data to the destination node. An important part of this is to determine how to ensure the destination actually receives the data. To find a good solution that not only takes into account the reliability, but also the energy and latency metrics, we rely on the knowledge of some other parameters. One such parameter is the setup time. We know that it does not count towards any of the metrics. Thus, we can save energy by beginning the protocol with a setup phase and using it to perform any initializations that do not rely on the generation of messages.

One parameter that sticks out from Category 2a is the periodicity of data generation. Good potential approaches to minimize the latency metric differ significantly between the periodic and aperiodic case. In the periodic case, we can find out at the beginning of the evaluation run exactly when each source generates new messages and schedule a data transmission directly after the data becomes available. In the aperiodic case, there is no such information we can rely on. Thus, we decided to split our protocol into two fundamental versions: one for the periodic case and one for the aperiodic case. Which version is executed is determined from the binary patching.

For good performance in the periodic case, we rely heavily on the knowledge of the exact data generation times for each source. The data generation period is a Category 2a parameter and can easily be incorporated into the protocol. However, to determine the exact times, we additionally need to rely on two Category 2b parameters: the time offsets between the nodes and the deviation of the actual data generation period from the corresponding input parameter provided during binary patching. Thus, we decided to further split the periodic protocol into a schedule initialization phase and a normal operation phase: In the schedule initialization phase, we learn the Category 2b parameter values and tailor the protocol in the subsequent normal operation phase to them. We decided not to rely on Category 2b parameter values in the aperiodic case. This allowed us to keep our protocol simple and general.

In conclusion, the initial analysis of the competition scenario lead us to structure the protocol design as shown in Figure 3.1. Section 3.3 goes into more detail on

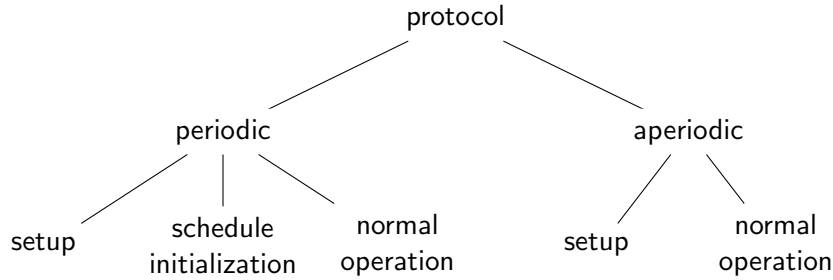


Figure 3.1: Breakdown of the protocol resulting from analysis of the competition scenario

the periodic protocol, Section 3.4 does the same for the aperiodic one. The setup phase is the same in both cases, it is described in Section 3.2.

3.2 Setup Phase

The first 60 s of each evaluation run, called the setup time, does not count towards any of the three performance metrics. Thus, we should ideally use this time to perform any protocol initializations which would otherwise carry an overhead in terms of energy, latency or reliability. However, we are limited by the fact that no new messages are generated during the setup time which means we cannot use it to learn anything new about the generation of messages. This leaves us with the following central goals for the first phase of our protocol, which we call the setup phase:

1. to achieve time synchronization between all nodes,
2. to establish a common state between the nodes and
3. to transition to a protocol phase which deals with the transmission of messages before the first message is generated.

Time synchronization in Baloo is achieved by means of the control packet. The host node, which transmits the control packet, is used as the base all other nodes synchronize to. Thus, it is important that the host node is chosen carefully and can be communicated with reliably. In our protocol, we use the sink node for the data collection as the host. This has several advantages over the use of other nodes. Primarily, the sink node is already involved in all data transmissions as a receiver anyway. Should we for instance decide to selectively turn off some nodes for transmissions by a given source to save energy, the host node would need to stay on anyway to receive the packet. Also, many design choices we make to improve the reliability of the data transmissions, such as the maximum number

of Glossy retransmissions, will also apply to the control packet. Furthermore, by selecting the sink to be the host, we can use the control packet to piggyback information that the sink needs to transmit to the sources, such as feedback on packet reception (ACKs and NACKs). Due to this choice, we will be using the terms host, destination and sink interchangeably for the remainder of this report.

To bootstrap, i.e. to synchronize with the host for the first time, nodes repeatedly turn on their radio in an attempt to receive a first control packet. Since RF interference may be present during the setup time, we cannot be sure this default bootstrap process will be successful: if the frequency channel the host sends the control packet on and the other nodes listen on is blocked by interference during the entire evaluation run, the nodes will never bootstrap. Thus, we decided to introduce a frequency hopping scheme, where the nodes periodically change the channel they attempt to communicate on. Since the nodes are not synchronized yet, care needs to be taken that each node eventually listens on the same channel the host transmits on. In a naïve approach, where all nodes hop to a different channel at the same period, nodes that turn on at a different time from the host may never bootstrap. We solved this issue by carefully choosing the period between channel hops at the host and at the other nodes.

For bootstrapping, the host needs to periodically transmit control packets. Since energy is of now concern during the setup time, we chose the period R_{Setup} to be as small as possible without having overlapping rounds, i.e. to be the same as the round duration plus some guard time. The host hops to a different channel in every round. This frequency is chosen from a static list known to all nodes based on the number of the round. The host keeps track of the round number by using a round counter r , which it initializes when it starts up. There is a defined number of channels the host can choose from. Thus, after $\# \text{ channels}$ rounds, the host will return to the initial frequency and start cycling through the channels again. In terms of actual time, the host sends a control packet on all used channels within a period of

$$\# \text{ channels} \cdot R_{\text{Setup}} \tag{3.4}$$

Thus, we can guarantee that the host and any other node will operate on the same channel for at least one round during this period if the other node changes its frequency based on the same static list of frequencies at a period greater or equal

$$\# \text{ channels} \cdot R_{\text{Setup}} + R_{\text{Setup}} \tag{3.5}$$

This is illustrated by Figure 3.2. If the hopping period for the regular nodes is chosen to be exactly $\# \text{ channels} \cdot R_{\text{Setup}} + R_{\text{Setup}}$, they will share a channel with the host periodically with this period, and the channel they share will differ with each period. We can make similar statements for larger periods. Since this period is quite small compared to the setup time, we can be sure that the nodes

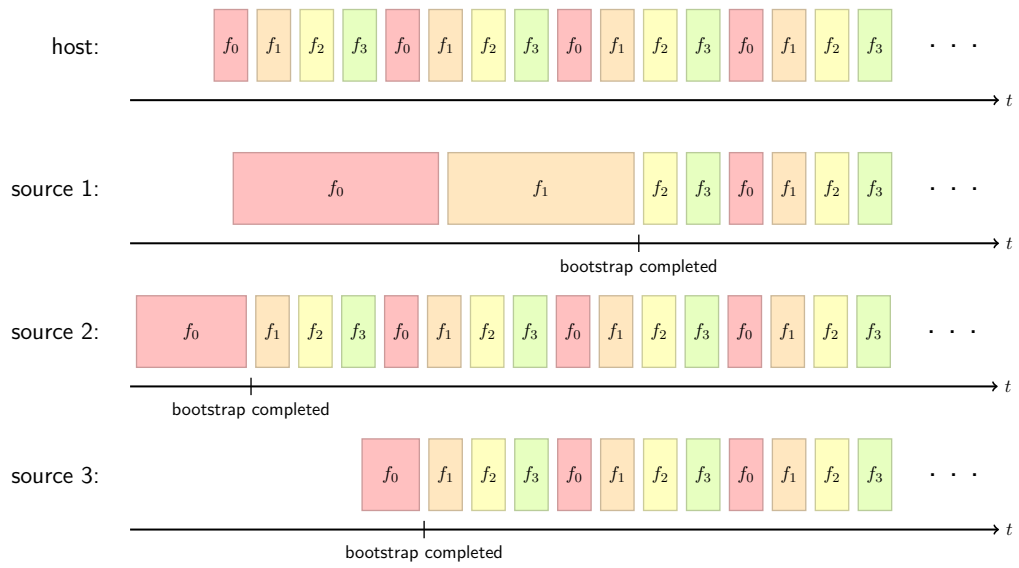


Figure 3.2: Example of the bootstrapping process under frequency hopping. The host changes frequency every round. Before receiving their first control packet, sources switch frequency with period $\# \text{ channels} \cdot R_{\text{Setup}} + R_{\text{Setup}}$. Afterwards, they follow the frequency hopping scheme of the host. Source 2 and 3 receive their first control packet once they first operate on the same channel as the host. Due to interference, source 1 misses the first control packet it could potentially have received, but then receives the packet once the host sends it on the next channel in the hopping sequence.

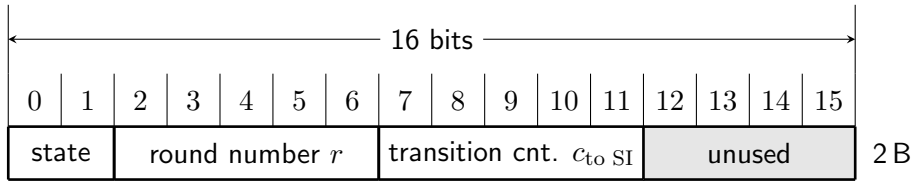


Figure 3.3: Structure of the control packets transmitted by the host/destination node during the setup phase. Contains the current state, the value of the round counter and a transition counter which is used to increase the likelihood of a successful synchronous transition to the next phase.

will share each channel with the host a number of times. Thus, it is highly likely that all nodes will receive the control packet at least once, which means they will likely be able to bootstrap well in advance of the end of the schedule init phase.

Once a source has bootstrapped, it follows the same frequency hopping scheme as the host. We have the host transmit its round counter r with the control, as shown in Figure 3.3. Because the limited space available in the control packet, the round counter will roll over fairly often. r is used to have some common state information between the nodes. If a node misses a control packet, it can update its counter independently and thus still operate on the correct frequency. In the implementation we have described so far, the host has no knowledge on how many sources have bootstrapped. To mitigate this slightly, we add one data slot for each of the up to eight source nodes to each round. Once a source has bootstrapped, it will send a “1” in its assigned data slot in each round until the end of the setup phase. Now, if the host receives a “1” in the data slot corresponding to a source, it knows this source has bootstrapped. This still gives the host only a very limited knowledge of the bootstrapping progress of the entire network, since there are many nodes beyond the source nodes. However, this is less problematic than it may seem. In the end, it only matters that the source nodes can successfully communicate with the sink. If the source nodes are able to bootstrap and let the host know they have done so, this likely means the communication between them will also work in the future. Other nodes which have not bootstrapped at this point are likely not essential parts these communication paths. Thus, it does not matter much whether or not they successfully bootstrap.

Once the host knows all sources have successfully bootstrapped, it can initialize the transition process to the next phase. It is crucial that all nodes know exactly when to switch to the next phase, since the round period and structure will change at this point. If a node changes these parameters too early or too late, it will no longer be synchronized with the other nodes and thus, it will not be able to communicate. Therefore, simply sending an instructing the nodes to

transition in the control packet of the round before the scheduled switch to the next phase is very risky. Instead, we use a transition counter $c_{to\ SI}$ which the host initializes to a given value as soon as it determines that all sources have bootstrapped. From this point on, it transmits its current counter value with the control packet, decreasing it by one in each round. Once the counter reaches zero, all nodes transition to the next phase. The advantage is that the nodes only need to receive a single counter value out of a number of transitions. Once they have received this value, they can proceed with the countdown independently.

3.3 Periodic Case

As alluded to in the initial analysis of the scenario, in the periodic case, we can reduce latency to a minimum by scheduling data transmissions as soon as possible after new data becomes available. Thus, after initialization, our protocol should result in a structure similar to the one shown in Figure 3.4. In order to be able to use this structure during normal operation of our protocol, we first need to find out when to schedule the data transmissions, i.e. when the sources generate new data. This is done during the schedule initialization phase.

3.3.1 Schedule Initialization Phase

The goal of the schedule initialization phase is to determine the exact times at which the nodes generate new data and use this information to develop a schedule where each source node can transmit its data as soon as possible after it becomes available. As described in Section 3.1, this requires knowledge of the following parameters for each source i :

- *The actual period T_i* which may be slightly larger than the period value T provided by binary patching. Even small differences between T and T_i can have a significant impact as they accumulate over time; the n th message will be delayed by

$$\delta t_i(n) = (n - 1) \cdot (T_i - T) \quad (3.6)$$

- *The data generation time offset Δt_i* which expresses that source nodes do not necessarily generate their first message immediately after the end of the setup time, at t_0 , but at a later time $t_i(1) = t_0 + \Delta t_i$. Since data generation is periodic, subsequent data will also be delayed by Δt_i ; the n th message at source i will be generated at

$$t_i(n) = t_0 + (n - 1) \cdot T_i + \Delta t_i \quad (3.7)$$

In practice, calculating the data generation time offset relative to the end of the setup time actually does not turn out to be a good idea. While we know

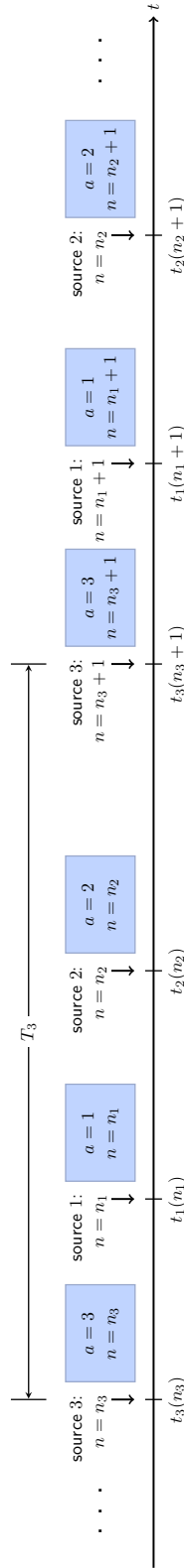


Figure 3.4: In the periodic case, new data is generated at source i each period T_i . After initialization, the transmission of a new message n_i should be scheduled as soon as possible after it becomes available at source i in a transmission slot that is reserved for source i .

the exact value of the setup time, we do not know at what point in time the nodes are in relation to the end of the setup time. We cannot assume that all nodes will start executing their firmware exactly at the beginning of the setup time. Thus, we only have a rough estimate of the end of the setup time in the timeframe of the nodes. If this estimate is too late, we might miss the first messages generated by the sources. Being too early, on the other hand, does not matter much. The offset value will be larger, but what ultimately counts is the absolute data generation time, which will remain the same since the reference time also changes. In our implementation, we choose the end of the protocol's setup phase as a reference time t_0 . By the measures described in Section 3.2 we ensure that the setup phase always ends before the end of the setup time. Thus, in the context of our protocol, we define the *offset* of a source i as

$$\Delta t_i = t_i(1) - t_0 \quad (3.8)$$

where:

$$\begin{aligned} t_i(n) &= \text{time the } n\text{th message becomes available at the source } i, n \in \mathbb{N} \\ t_0 &= \text{time of the end of the setup phase} \end{aligned}$$

Similarly, we define the *data period* of a source i as

$$T_i = t_i(n) - t_i(n - 1) \quad (3.9)$$

Note that we assume here that the data period is constant. In Section 3.3.2 we address the case where this assumption does not hold.

We have now determined which parameters we need to find. The next open questions are when and how to measure and communicate this information. Since we know nothing about the data generation timing of the sources at this point, it makes sense to periodically schedule communication rounds where sources which do have data available get a chance to send it to the destination node. To avoid collisions, we assign each node a time slot within the round during which it has the exclusive right to transmit one message. Figure 3.5 illustrates this setup and introduces some parameters we will use for learning the offsets and data periods.

We continue to use the global round counter r initialized in the setup phase (see Section 3.2). As in the setup phase, we use this round counter to set the frequency all nodes communicate on during the round. The host transmits the current value of the round counter with every control packet. The structure of the control packet for the schedule initialization phase is shown in Figure 3.6. Because of the limited number of bits available in the control packet, the round counter rolls over fairly often. This does not matter for the choice of communication channel, but is undesirable when using the counter as a measure of the time passed since the end of the setup phase, which we need for the offset calculation.

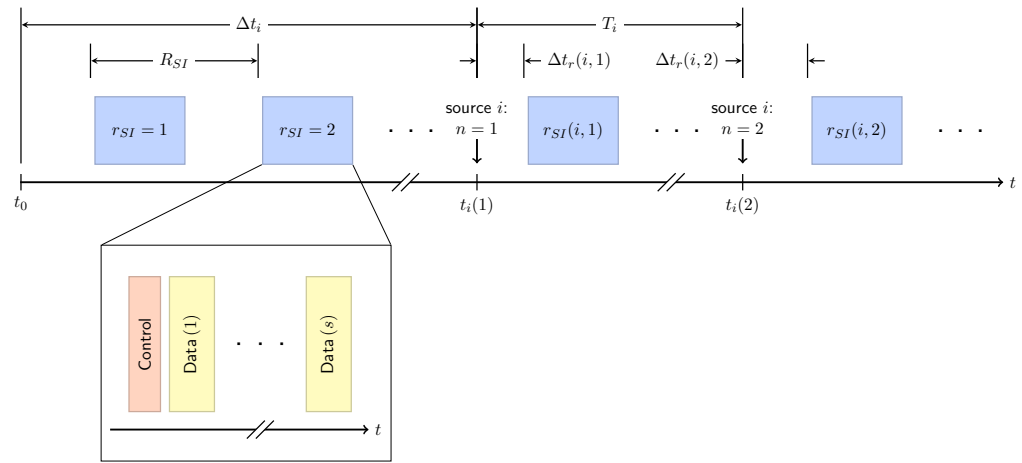


Figure 3.5: Overview of the basic structure of the schedule initialization phase. Baloo rounds are scheduled periodically with period R_{SI} and consist of a control slot followed by s data slots, where s is the number of source nodes. Each data slot is assigned to one source node. Each round is assigned a schedule initialization round number r_{SI} . An arrow labeled with a source index and a sequence number n indicates that new data with sequence number n becomes available at this source. $\Delta t_r(i, n)$ represents the period between the time $t_i(n)$ the n th packet becomes available at source i and the start of the subsequent round, which has the schedule initialization round number $r_{SI}(i, n)$. Δt_i shows the offset of source i relative to the end of the setup phase t_0 . T_i is the data period of source i .

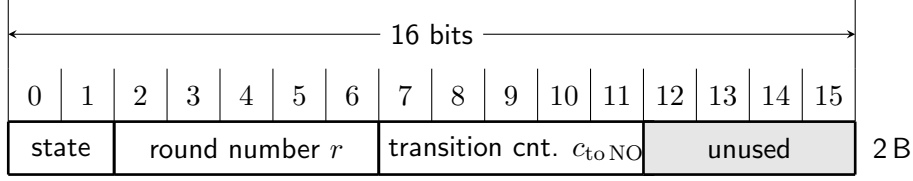


Figure 3.6: Structure of the control packets transmitted by the host/destination node during the schedule initialization phase. Contains the current state, the value of the round counter and a transition counter which is used to increase the likelihood of a successful synchronous transition to the normal operation phase.

For this purpose, we introduce a dedicated schedule initialization round counter r_{SI} which is updated independently by each node. It starts at 1 in the first round of the schedule initialization and is wide enough to not roll over during the entire duration of the schedule initialization phase.

The *round period* R_{SI} is a tuneable parameter. It must be longer than the time it takes to complete the communication and data processing associated with the round. On the other hand, it must be shorter than the data period. Otherwise, messages would be lost, since sources only have time to transmit one message per round. Between this lower and upper bound, there is a tradeoff between reliability and latency on the one hand and energy on the other. A shorter round period means each source can transmit the same data more often, improving reliability. It also reduces the expected time the source must wait to transmit new data. On the other hand, the larger number of retransmissions and the overhead associated with each additional round increase the energy consumption.

In our setup, for each message generated at source i with sequence number n , there will be a schedule initialization round $r_{SI}(i, n)$ (with an associated global round number $r(i, n)$) during which i transmits the packet for the first time. If we measure the time $\Delta t_r(i, n)$ that source i needs to wait from when message n becomes available until the beginning of round $r_{SI}(i, n)$, we can calculate the offset of source i as

$$\Delta t_i = r_{SI}(i, n) \cdot R_{SI} - \Delta t_r(i, n) \quad (3.10)$$

We call $\Delta t_r(i, n)$ the within-round offset. We can easily measure this by taking a timestamp in the interrupt service routine (ISR) triggered by the data becoming available and another in the pre-process of round $r_{SI}(i, n)$ and calculating the difference. If we know the within-round offsets and round numbers of the first transmission for two messages with sequence numbers n and m , $m > n$, we can also calculate the data period as

$$T_i = \frac{1}{m - n} \cdot ((r_{SI}(i, m) - r_{SI}(i, n)) \cdot R_{SI} + \Delta t_r(i, n) - \Delta t_r(i, m)) \quad (3.11)$$

Each node needs to know the data period and offset of all the source nodes.

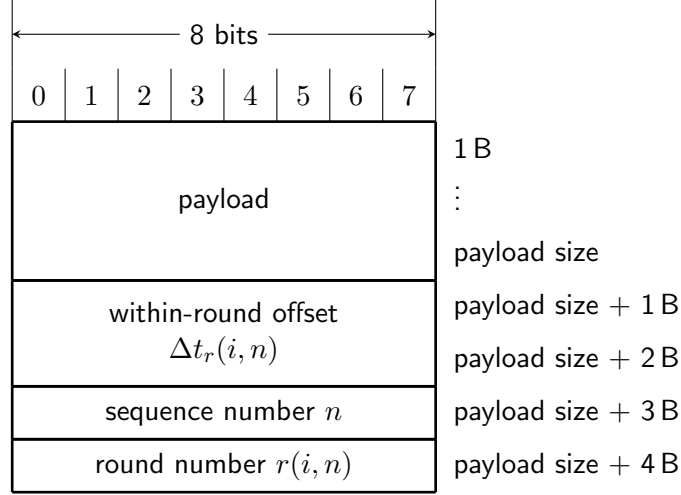


Figure 3.7: Structure of the data packets transmitted by source nodes in their assigned slots during the schedule initialization phase. Metadata which allows all other nodes to learn the data generation schedule of the transmitting node is piggybacked to the payload, which contains the message generated by the transmitting node.

To achieve this, each time a source i sends the message with sequence number n , we attach n , the within-round offset $\Delta t_r(i, n)$ and the global round counter value $r(i, n)$ of the first round of transmission of the message to the packet. The resulting packet structure is shown in Figure 3.7. A node can find $r_{SI}(i, n)$ by relating $r(i, n)$ to the values of the current global counter value and schedule initialization round counter it has stored. Once a node has received one such packets from a source i , it can independently calculate the offset of i . As soon as it receives another packet with a different sequence number, it can calculate the data period. We decided on this approach, since it allows us to reliably find the data period and offset as soon as possible while keeping the round structure constant across the entire phase. The timing information for each message generation is retransmitted as often as the message itself. If we tune the round period such that we have a high reliability for the messages, we will also have a high reliability on the timing information. Furthermore, we can calculate the parameter values from the timing information of any two messages. Thus, the protocol can remain stable even when all transmissions of a message are missed.

Once the exact data generation times of each source parameterized by the offset and data period are known at each node, these need to be translated into a Baloo-compatible schedule. As explained in more detail in Section 3.3.2, we will be using rounds which are dedicated to a specific source node to transmit the data to the destination during the normal operation phase. In each normal

operation round, the nodes need to know the following information:

- the *round assignee* of the current round
- the *period* until the next round

In our implementation, we provide Algorithm 1 to the normal operation phase as an interface to retrieve these parameters and update the underlying schedule. The algorithm uses the following parameters and data structures:

- The *maximum period* R_{\max} between two rounds we can have without getting synchronization issues or running into overflows. If the period between two subsequent rounds in the schedule is larger than this value, we need to schedule one or more empty rounds with just a control packet in between them to ensure we do not exceed the maximum period. A good value for the maximum period needs can be determined experimentally. It should not be much shorter than it needs to be to avoid wasting energy. We chose a value of 60 s.
- The *maximum duration of a normal operation round*. We need to ensure that the period between rounds is always larger than this round duration.
- The *next round assignee*.
- The *schedule*, an array containing the time period until each node should be scheduled next, relative to the time of the round of the next assignee. The period until the next round can be determined by finding the minimum value in the schedule. The node associated with that entry will be the assignee of the next round. After scheduling the next round, the value for each source in the schedule needs to be updated to reflect that we move forward in time by the extracted period.
- An array containing the *data period* for each source. When a node is scheduled in the next round, we increase the time until it should be scheduled again by its data period.

We can directly initialize the data period array with the values for the data periods we have found for all sources with the methods described above. Initializing the schedule and next assignee parameter is slightly more tricky and closely linked to the way in which we transition to the normal operation phase.

As with the transition from the setup to the schedule initialization phase described in Section 3.2, we need to ensure that all nodes change to normal operation in the same round. To this end, we introduce a *transition counter* $c_{to\ NO}$, which we attach to the control packet as shown in Figure 3.6. Once the host has obtained all data period and offset values, it initializes this counter to a

Algorithm 1 Pop schedule information

```

1. round_assignee  $\leftarrow$  next_assignee
2. period  $\leftarrow$  min(max_period, max(max_round_duration, min(schedule)))

3. if period is max_period then ▷ save assignee for next round
4.   next_assignee  $\leftarrow$  empty
5. else
6.   next_assignee  $\leftarrow$  argmin(schedule)
7. end if

8. for all sources in schedule do ▷ update schedule times
9.   schedule[source]  $\leftarrow$  schedule[source] - period
10. end for
11. schedule[next_assignee]  $\leftarrow$  schedule[next_assignee] + data_period[next_assignee]

12. return round_assignee, period

```

given value, which is a parameter of the protocol. All other nodes initialize their transition counter based on the counter value they receive in the control packet. Once initialized, the transition counter is decreased by one in each subsequent round. When the transition counter at a node reaches zero, the node transitions to the normal operation phase.

After a node has found all data period and offset values, it will continue to follow the normal operation round structure until the transition counter reaches zero. When using this round structure, we improve reliability by having a much shorter round period than data period and thus retransmitting each message across multiple rounds. There are no within-round retransmissions. This has the effect that we might run into reliability issues towards the end of the schedule initialization phase. If a new message is first transmitted in one of the last few rounds of the phase, its retransmissions will be limited by the rounds remaining in the phase. To mitigate this, we choose a threshold value c_{thr} for the transition counter; messages first transmitted in a round with a transition counter value below the threshold are retransmitted during the normal operation phase.

Thus, the first message scheduled in the normal operation phase for each source will be the first message generated after the round with transition counter c_{thr} . To initialize the schedule, we need to find the time at which each of these messages is generated, relative to the time of the last round of the schedule initialization round. Before including this value to the schedule, we add a small guard time t_g to ensure rounds are only started after the data becomes available. A good value for the guard time needs to be determined experimentally. In our

implementation, we use 20 ms. For source i , the schedule is thus initialized to

$$\Delta t_i + \left\lceil \frac{r_{SI,thr} \cdot R_{SI} - \Delta t_i}{T_i} \right\rceil \cdot R_{SI} - r_{SI,final} \cdot R_{SI} \quad (3.12)$$

where:

- Δt_i = offset of source i
- T_i = data period of source i
- R_{SI} = schedule initialization round period
- $r_{SI,thr}$ = schedule initialization round number of the round where $c_{to\ NO} = c_{thr}$
- $r_{SI,final}$ = schedule initialization round number of the round where $c_{to\ NO} = 0$

To initialize the next assignee (i.e., to find the assignee of the first normal operation round) and to determine the period between the last schedule initialization round and the first normal operation round, we run Algorithm 1 once, starting from Line 2. After running the algorithm, all schedule parameters are initialized and ready for use in the normal operation phase.

3.3.2 Normal Operation Phase

Typically, the normal operation phase is the phase the nodes operate in for most of the duration of the evaluation. Its performance relies on the setup performed in the setup phase and the schedule initialization phase.

During normal operation, as illustrated in Figure 3.8, a Baloo round is scheduled shortly after new data becomes available at a source. Each round has an assignee a , which corresponds to the source the data was generated at. The round is then reserved solely for the transmission of data from this source a to the destination. The data is transmitted in a designated data slot, which is scheduled after the control slot. In the subsequent slot, the destination sends a short packet, shown in Figure 3.9 which acknowledges the data if the destination received it (ACK), or does not acknowledge it (NACK) otherwise. At each node, this data and ACK/NACK slot pair is repeated until either of the following events terminates the round:

- The node has received (or, in the case of the destination, sent) an ACK.
- A maximum number of transmissions (i.e. data slots) $tx\#_{max}$ is reached.

The maximum on the number of transmissions is introduced for several reasons:

- It allows us to find an upper bound on the round duration, which we use to ensure subsequent rounds do not overlap.

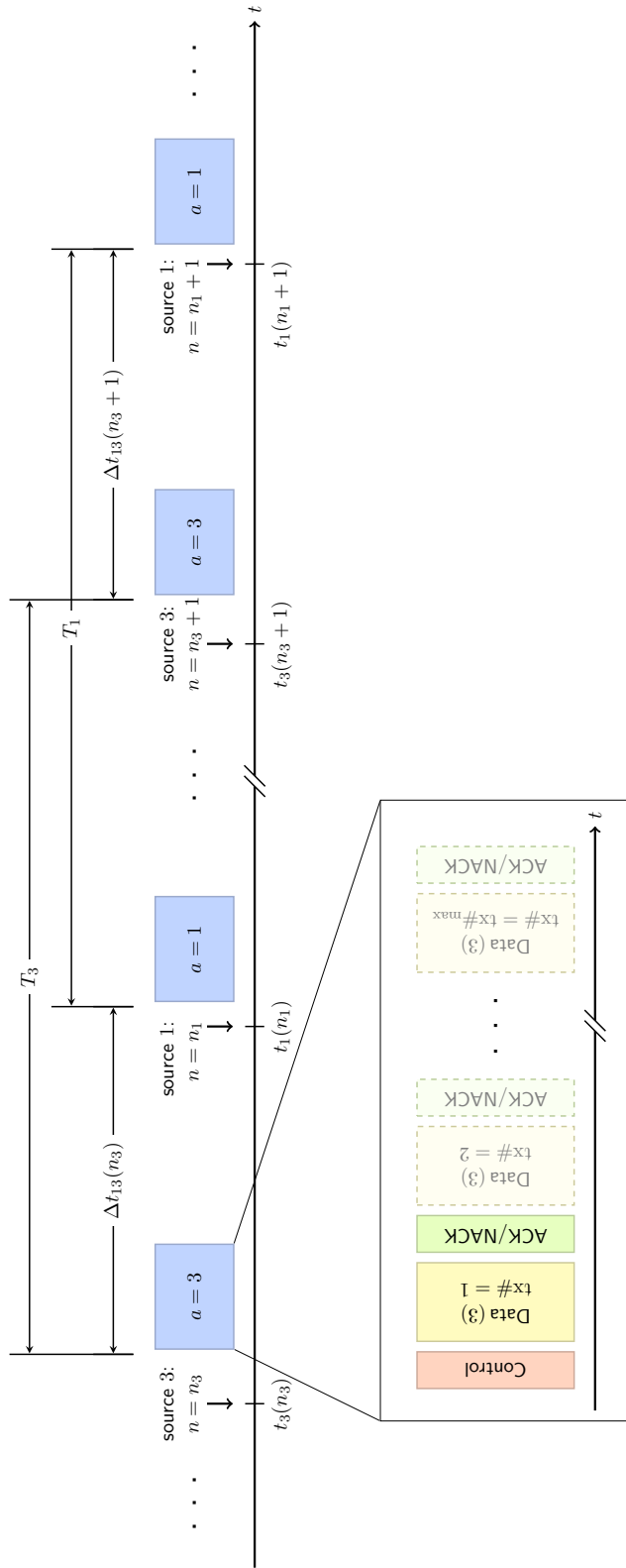


Figure 3.8: Overview of the basic structure of the normal operation phase. A Baloo round with assignee $a = i$ is scheduled as soon as possible after new data becomes available at source i . Besides the control slot, each round consists of up to tx\#_{\max} data and ACK/NACK slot pairs. During data slots, i transmits its new data. During ACK/NACK slots, the host informs the other nodes whether it has received the data from i . If a node receives an ACK, it immediately terminates the round. The host terminates the round once it has send an ACK.

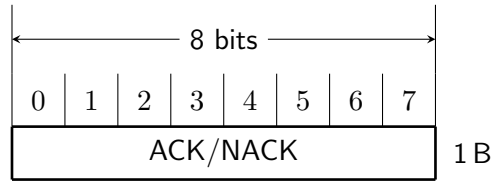


Figure 3.9: Structure of the packet used by the destination to inform nodes about whether or not it has successfully received a data packet. The destination sends the index of the round assignee if it has received the data (ACK) and a special NACK value otherwise.

- It limits the energy we waste in the case of heavy interference. If the data has not arrived after $\text{tx}\#_{\max}$ data slots, it will likely not get through at all any time soon. In this case, we give up on the packet instead of wasting energy on any more futile transmission attempts. One could consider trying to retransmit the packet in a later round. In our implementation, we have decided against this to avoid both making our protocol more complex and the impact on latency associated with the strategy.
- It limits the energy we waste in case an ACK gets lost. This can happen quite often, since the destination only transmits the ACK exactly once. Without a maximum number of retransmissions, any node which misses this one transmission of the ACK would stay in the round and continue to repeat the slot pairs for ever.

We chose $\text{tx}\#_{\max} = 4$, which gave a reasonable tradeoff between reliability on the one hand and energy and latency on the other.

To improve reliability under interference, we employ a frequency hopping strategy. For the control packet, we use the same approach based on the global round counter r as in the previous phases (see Section 3.2). In addition, we change the frequency channel after each slot pair. Remaining on the same channel for the ACK/NACK slot as the data slot has the advantage that, if the data was received, the ACK will likely be received as well. As mentioned before, receiving the ACK is important. Missing it means a node will remain awake for all $\text{tx}\#_{\max}$ slot pairs. For the first pair, we stay on the same channel as the control packet was transmitted on. This has the advantage that this channel is changing every round with a schedule already known to all nodes. The alternative would be to use the same frequency for the first slot pair in every round. This frequency could potentially be jammed for the entire duration of the evaluation. In this case, we would always waste the first slot pair and incur a large cost in terms of energy and latency. For the subsequent slot pairs, we choose a frequency out of a static list known to all nodes. The choice is based on the channel of the first slot pair and the number of the slot pair under consideration.

In each round, each node uses Algorithm 1, explained in Section 3.3.1, to schedule the next round as soon as possible after the next time data becomes available at a source. The initialization performed in the schedule initialization phase takes into account measurements of the offset Δt_i and the actual data period T_i of each source i . It also accounts for slight variations in the generation times of the data by introducing a guard time t_g . However, this guard time may not be enough to cover inaccuracies in the measurement of the actual data period or the case where this period changes slightly over time. This is because variations in the period lead to a change in offset that accumulates over time.

For instance, take the case where the actual data period is longer than the measured data period T_i by $\Delta T_{i,\text{error}}$. An example of this case is shown on the top of Figure 3.10. If the offset between the generation of message n at source i and the start of the subsequent round assigned to i is $\Delta t_r(i, n)$, then the corresponding offset for the generation of the next message $n + 1$ becomes

$$\Delta t_r(i, n + 1) = \Delta t_r(i, n) - \Delta T_{i,\text{error}} \quad (3.13)$$

The offset between the generation of some later message $n + m$ and its corresponding round decreases even further, it amounts to only

$$\Delta t_r(i, n + m) = \Delta t_r(i, n) - m \cdot \Delta T_{i,\text{error}} \quad (3.14)$$

At some point, this offset will become negative, which means the data will only become available after the round it is supposed to be sent in has started. Thus, the data needs to be sent one round later. The consequence is a latency that is almost as big as the data period. This is clearly undesirable.

One could attempt to mitigate this by purposely overestimating the data period. However, this has the disadvantage that the offset $\Delta t_r(i, n)$ will increase over time by the same principle, which will lead to unacceptable latencies in the long run. Instead, we use a more dynamic method, where the data period before the next round (and the next round only) is increased by a certain time Δt_{update} to $T_i + \Delta t_{\text{update}}$, if the offset $\Delta t_r(i, n)$ for a source falls below a certain threshold Δt_{thr} . This results of this drift compensation process are demonstrated in the lower part of Figure 3.10.

To achieve this, we need to know whether or not node i has undercut the threshold in its previous round, and this information needs to be consistent across all nodes by the time the next round for i is scheduled (i.e., in the round preceding the next round, assigned to some other source j). This is crucial to ensuring that the schedules kept at the nodes remain synchronized. As a first step, source i needs to measure whether it itself has undercut the threshold. To this end, we continue to have the source nodes keep track of their within-round offsets $\Delta t_r(i, n)$ in the normal operation phase. If $\Delta t_r(i, n)$ is below Δt_{thr} in

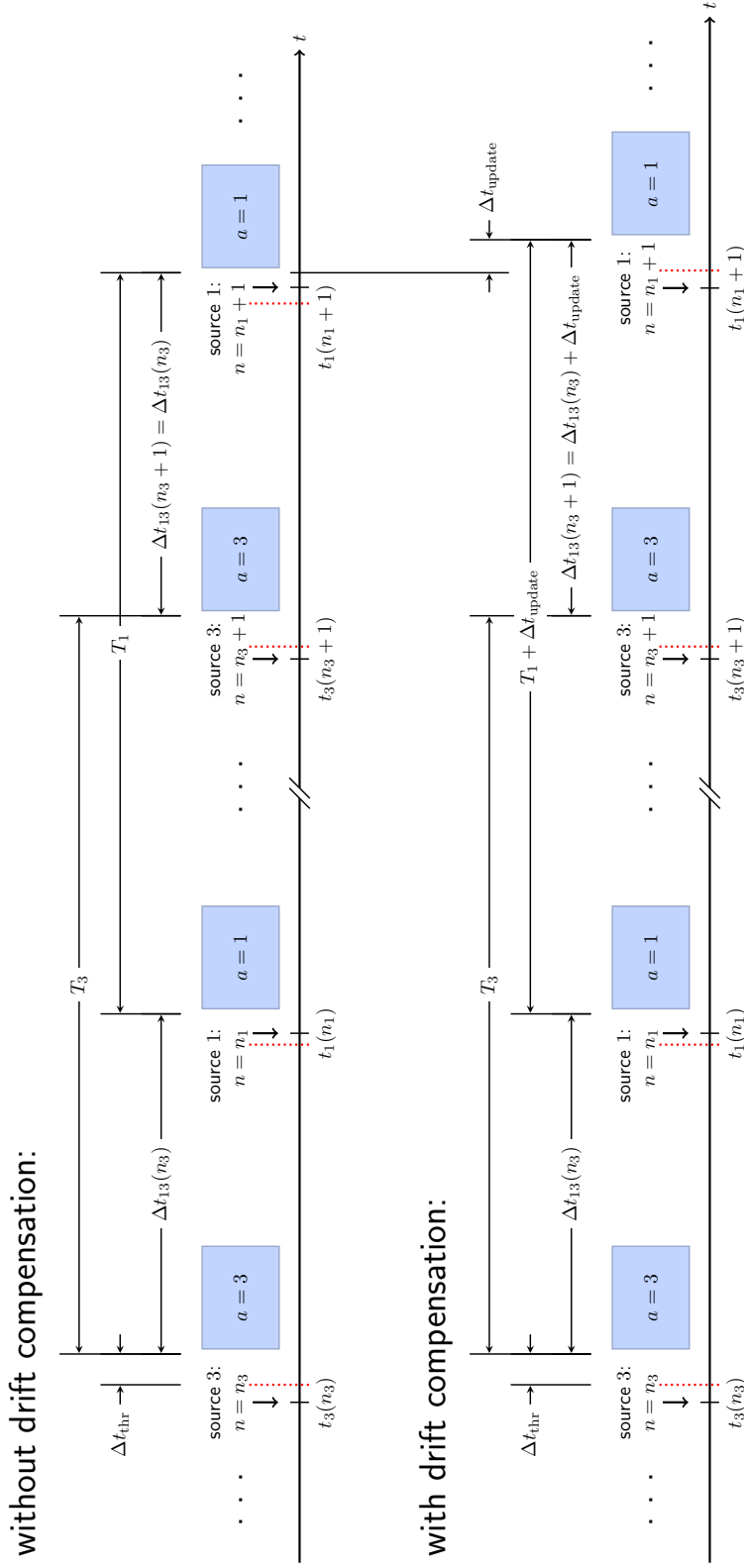


Figure 3.10: The top shows what happens if the data period T_i of a source i is underestimated and this error in measurement is not accounted for. In this case, the time between the data becoming available and the round it is transmitted in decreases in each round. Below this we show the effect of our drift compensation process. With this process, the data period before the next round assigned to a source i is increased by a certain time Δt_{update} to $T_i + \Delta t_{\text{update}}$, if the offset $\Delta t_r(i, n)$ for a source i falls below a certain threshold Δt_{thr} .

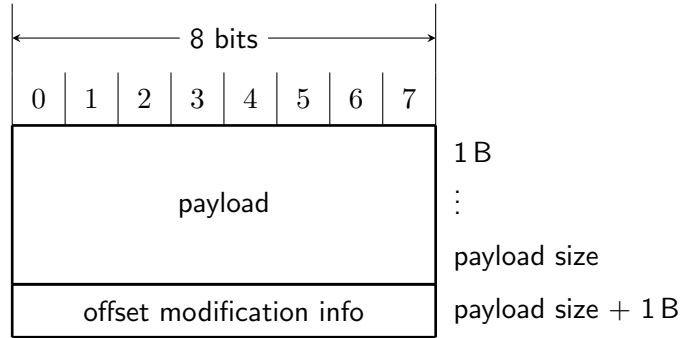


Figure 3.11: Structure of the data packet transmitted by source nodes in their assigned rounds during the normal operation phase. The payload contains the message generated by the transmitting node. If the message only became available closer to the start of the round than a certain threshold time Δt_{thr} by a certain offset, the source attaches this piggybacks this offset to the payload to notify the host that its schedule needs to be updated.

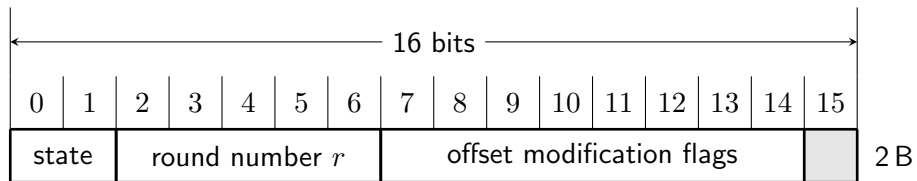


Figure 3.12: Structure of the control packets transmits by the host/destination during the normal operation phase. Contains the current state, the value of the round counter used for frequency hopping and a bit array with space for the maximum number of sources which contains flags that inform nodes if the schedule for the sources corresponding to the bits should be updated by a given value Δt_{update} known to all sources.

the round where i sends message n , i attaches the time difference between Δt_{thr} and $\Delta t_r(i, n)$ to the data packet it sends in the round, as shown in Figure 3.11. Because this packet can be retransmitted multiple times on different frequencies, we can be fairly certain that the host will receive this information. We even receive feedback in the form of an ACK.

Nevertheless, to ensure all nodes receive the update, we introduce an additional measure. With each control packet, the control packet sends a 1 B bit-array of offset modification flags. When the host receives the information that the threshold has been undercut by source i from a data packet transmitted by i , it sets the i th flag bit. By setting this bit, it informs all other nodes that the scheduled start time for the next round assigned to source i should be increased by Δt_{update} . Δt_{update} is a constant parameter known to all nodes. A good value for this needs to be found experimentally and is closely related to the size of the threshold time. We have chosen $\Delta t_{\text{thr}} = 35$ ms and $\Delta t_{\text{update}} = 4$ ms. These values could likely be set more tightly. However, we chose to rather incur a slightly higher latency in each round, than to run the risk of scheduling a round too early. Since there are typically multiple sources, the control packet will be transmitted multiple times on different frequencies, making it likely that it will be received by all nodes before the next round assigned to i needs to be scheduled. One could consider adding additional transmissions of the control packet to increase the reliability at the cost of energy. We decided against this, since we did not observe significant issues with the drift compensation mechanism and wanted to keep our protocol as simple as possible. In the round preceding the next round assigned to i , all nodes then increase the scheduled time of i by Δt_{update} . Also, the host resets the flag corresponding to i in the control packet of the round assigned to i . If i still undercuts the threshold in this round, i.e., if $\Delta t_r(i, n + 1) < \Delta t_{\text{thr}}$, the process is repeated. Otherwise, the round for message $n + 2$ of source i will be scheduled normally, i.e. one data period T_i after the round for message $n + 1$.

3.4 Aperiodic Case

The aperiodic case is fundamentally different from the periodic case, since there no way to determine in advance when new data will be generated. Thus, we have no real reference point for when to schedule transmissions with minimum latency and need to use an alternative scheme.

The most intuitive solution is to simply schedule transmissions periodically. This has the added benefit that the schedule does not need to be updated after the end of the setup phase, making it less likely for nodes to lose synchronization. The round period R between subsequent transmissions presents a tradeoff between latency and energy. A shorter round period means the latency will be lower on

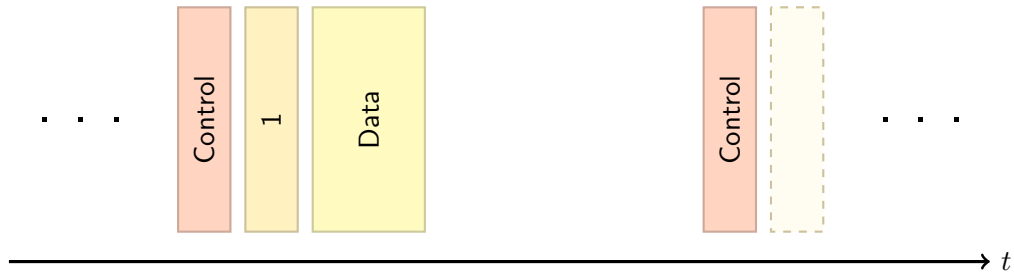


Figure 3.13: Overview of the round structure used in the aperiodic case. In the first depicted round, at least one source has unacknowledged data to transmit. In this case, the node sends a “1” in the flag slot following the control slot. It then proceeds its data during the subsequent data slot. In the second round of this example, no node has new data to transmit. Thus no flag packet is sent, and the nodes can go to sleep immediately after the flag slot. The control packet is used to acknowledge the last packet that was successfully received by the host.

average, but carries a larger overhead in terms of energy. We decided to choose the round period to be the maximum of 300 ms and the maximum duration of a transmission round. This choice had two main consequences for the further design of the protocol:

1. We decided to limit the number of transmissions per round to exactly one. Any nodes whose transmission is not received during a given round, can retransmit in the next round. The latency incurred should still be manageable due to the relatively short round period.
2. If there is no data to be transmitted during a round, nodes should realize this as soon as possible and go to sleep. Otherwise, the energy overhead incurred would be quite large.

From these considerations, we decided on the round structure depicted in Figure 3.13. The flag slot immediately following the control packet is a contention slot used to determine whether or not there is any data to be transmitted in the round. Any node that does have data available it has not yet received an ACK for, will send a short flag packet, shown in Figure 3.14, to indicate this. If a node has neither transmitted nor received a packet during the flag slot, skip the rest of the round and go to sleep until the next round to save energy. Otherwise, it proceeds to participate in the data slot. This slot is a contention slot as well. Any source that sent a flag packet in the previous slot may now transmit its data. To allow the host to determine which packet it has received, the source piggybacks its ID i as well as a sequence number n_i for the message onto the payload. The resulting packet structure is shown in Figure 3.15.

Since multiple source may transmit at the same time during the data slot, colli-

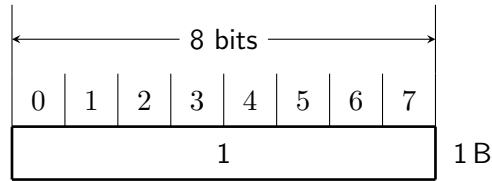


Figure 3.14: Structure of the flag packet send by source nodes in the aperiodic case to indicate that they have data to transmit.

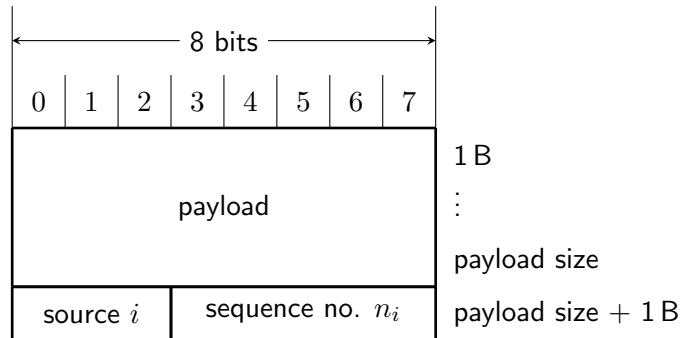


Figure 3.15: Structure of the data packet transmitted by source nodes in the aperiodic case. Piggybacked onto the payload is the ID of the transmitting source i and the sequence number n_i of the message in the payload.

sions may occur. However, since the number of sources is limited to eight in a fairly large network, we assume that the host will most likely still receive one of the packets due to the capture effect. Furthermore, the smaller the chosen round period, the less likely it is that all sources will attempt to transmit in the same round. Our experiments have shown that this reliance on the capture effect is justified.

The host needs to let sources know through an ACK which packets it has received, so the sources can decide whether or not they need to retransmit a given message. As explained before, we do not allow retransmissions during the same round, contrary to the periodic case. Therefore, there is no need for the host to acknowledge the data it has received before the start of the next round. Instead of sending a separate ACK packet, the host can thus simply piggyback the ACK for a given round onto the control packet of the next round, as shown in Figure 3.16. More specifically, the control packet always contains the source ID i and sequence number n_i of the last message the host has successfully received. Based on the information received in the control, sources can then decide whether or not to send the flag.

Sources store each new message in a buffer until it is acknowledged by the host.

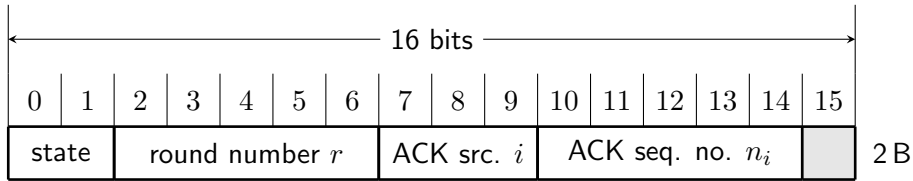


Figure 3.16: Structure of the control packets transmitted by the host/destination node in the aperiodic case. Contains the current state, the value of the round counter as well as the source ID i and sequence number n_i of the last packet the host has successfully received.

If a source has stored multiple messages, it always transmits the oldest message first. If the buffer is full and new data is generated, the source overwrites the oldest message in the buffer with the new data and proceeds to attempt to send the data that is the next oldest. This approach allows sources to get another attempt at transmit their data in case of heavy interference blocking transmissions across multiple data generations, while also limiting the latency we are willing to incur. A larger buffer may increase reliability, but it may also increase latency.

As in the periodic case, we use a frequency hopping scheme to improve reliability in the presence of interference. We continue using the same scheme based on a round counter r we introduced during the setup phase (see Section 3.2). Each round, nodes switch to a new frequency channel based on a static list of channels.

3.5 Optimization Parameters

In the following, we give an overview of a non-exhaustive selection of the parameters of our protocol that are easily tuneable to achieve different tradeoffs between the three performance metrics. The parameters are explained in more detail in the previous sections. Where applicable, we list a lower and upper bound on the parameter and give an indication which effect *increasing* the parameter may have on each of the performance metrics. We also provide the value we have chosen for each parameter based on our experiments.

Both Aperiodic & Periodic Case

Maximum round period: Maximum round period the protocol can deal with without losing synchronization between the nodes or running into overflows.

- *Lower bound:* round duration
- *Upper bound:* 65 535 ms (size of a 16-bit unsigned integer)

- *Reliability*: Decreases if nodes lose synchronization.
- *Energy*: Decreases.
- *Chosen value*: 60 s

Number of control frequency channels: The number of different frequency channels the control packet may be transmitted on.

- *Lower bound*: 1
- *Upper bound*: 15, i.e. the number of channels within the permissible frequency range
- *Reliability*: Typically increases. However, the bootstrapping phase may take longer, which may lead to a failure to bootstrap before the end of the setup time, resulting in a lower reliability.
- *Chosen value*: 8

Number of transmission in Glossy flood for control packets:

- *Reliability*: Increases.
- *Latency*: Increases.
- *Energy*: Increases.
- *Chosen value*: 5

Number of transmission in Glossy flood for data packets:

- *Reliability*: Increases.
- *Latency*: Increases.
- *Energy*: Increases.
- *Chosen value*: 4

Maximum number of hops in the network: Required to estimate the length of a Glossy flood.

- *Reliability*: Increases.
- *Latency*: Increases.
- *Energy*: Increases.

- *Chosen value:* 15 on the competition testbed, 5 on FlockLab

Initial transition countdown value in the setup phase:

- *Reliability:* Increases up to a point, as a higher value increases the likelihood that all nodes will receive a counter value before it reaches zero and will thus transition to the next phase in the same round. May decrease once the value gets too big, as this will extend the setup phase beyond the end of the setup time.
- *Latency:* No impact if the value is chosen small enough such that the setup phase ends before the end of the setup time. Increases otherwise.
- *Energy:* No impact if the value is chosen small enough such that the setup phase ends before the end of the setup time. Increases otherwise.
- *Chosen value:* $4 \cdot$ number of control frequency channels

Periodic Case

Schedule initialization round period:

- *Lower bound:* schedule initialization round duration
- *Upper bound:* $\min(\text{data period}, \text{maximum round period})$
- *Reliability:* Decreases.
- *Latency:* Increases.
- *Energy:* Decreases.
- *Chosen value:* $\max(300 \text{ ms}, \min(\text{max period}, \text{data period}/10))$

Initial transition countdown value in the schedule initialization phase:

- *Reliability:* Increases.
- *Latency:* Increases.
- *Energy:* Increases.
- *Chosen value:* $2 \cdot$ number of control frequency channels

Threshold transition countdown value for the last schedule initialization phase packet: Packets generated after this countdown value is reached are retransmitted during the normal operation phase.

- *Reliability*: Increases until the value reaches a point where the remaining rounds in the countdown take longer than one data period, then decreases.
- *Latency*: Increases.
- *Energy*: Increases.
- *Chosen value*: number of control frequency channels

Schedule guard time: Added to the schedule during schedule initialization to ensure rounds are scheduled after the data generation

- *Latency*: Increases. However, if the value is chosen too small, rounds might be scheduled too early for the data generation and the latency might increase to almost a data period.
- *Chosen value*: 20 ms

Maximum number of transmissions in normal operation: Corresponds to the maximum number of data and ACK/NACK slot pairs.

- *Lower bound*: 1
- *Upper bound*: such that $\# \text{sources} \cdot \text{round duration} < \text{data period}$
- *Reliability*: Increases.
- *Latency*: Increases.
- *Energy*: Increases.
- *Chosen value*: 4

Within-round offset threshold: Desired minimum period of time between new data becoming available and the start of the corresponding round during normal operation.

- *Latency*: Increases. However, if the value is chosen too small, rounds might be scheduled too early for the data generation and the latency might increase to almost one data period.
- *Chosen value*: 35 ms

Schedule update time: Time by which the next round for a source should be pushed back if the within-round offset falls below the threshold during normal operation.

- *Latency*: Increases. However, if the value is chosen too small, rounds might be scheduled too early for the data generation and the latency might increase to almost one data period.
- *Chosen value*: 4 ms

Aperiodic Case

Round period:

- *Lower bound*: round duration
- *Upper bound*: maximum round period
- *Reliability*: Decreases.
- *Latency*: Increases.
- *Energy*: Decreases.
- *Chosen value*: $\max(300 \text{ ms}, \text{round duration})$

Retransmission buffer size:

- *Reliability*: Increases.
- *Latency*: Increases.
- *Energy*: Increases.
- *Chosen value*: 3 messages

Evaluation

4.1 Protocol Performance

Unfortunately, the results of the 2019 EWSN Dependability Competition have not been made public by the time this report was written. Thus, at this point we can only make some rough estimations of our performance based on what we have observed during tests accompanying the development of our protocol.

When testing our protocol on the competition testbed [16] without jamming enabled, we were able to achieve reliability metrics of around 100% and average latencies of a few hundred ms. Most competing teams achieved reliability metrics of this magnitude in the case without interference. From observing the leaderboards during testing we expect our performance in terms of energy and latency will likely be well-balanced. While there were solutions which performed better in each of these two categories, no single solution on the leaderboard outperformed our protocol in both categories.

When adding interference to our tests, our reliability tended to drop quite significantly. This seems justified since our protocol does not use a very elaborate interference avoidance scheme and makes no provisions for retransmissions after a long time. In the periodic case, if there is heavy interference during just one round, we will lose a packet. This topic clearly warrants some further consideration, but for this edition of the competition we decided to focus mostly on the evaluation scenarios without heavy interference.

Due to its high utilization and limited availability, testing time on the competition testbed was limited. We used the FlockLab [17] testbed for some additional tests and debugging. As an added benefit, running the protocol on two different networks allowed us to get an impression of how well our protocol would generalize. Aside from a few minor changes, such as increasing the maximum hop count when moving from FlockLab to the competition testbed, we found that the protocol was able to perform its basic task highly reliably on both testbeds.

	payload size [B]	period [s]	total no. of packets sent	performance		
				reliability [%]	avg. latency [ms]	avg. current [mA]
<i>periodic case</i>						
	8	5	414	100	148	1.3
	8	30	65	100	584	0.9
	64	5	414	100	280	1.9
	64	30	65	100	508	1.1
<i>aperiodic case</i>						
	8		65	100	275	1.7
	64		65	100	302	1.7

Table 4.1: Performance achieved running the protocol for different parameters on the topology shown in Figure 4.1 on FlockLab for 8 min each.

However, with increased fine-tuning to either of the testbeds the performance on the other would likely suffer.

As an illustration of the protocol’s performance on FlockLab we provide the results of some tests in Table 4.1. All tests ran on the topology shown in Figure 4.1 for a duration of 8 min. One value that stands out is the high latency in the case of a 30s period. We choose the schedule initialization round period in relation to the data period. In this case, the round period is 3s long, resulting in a large latency on the first couple of messages. Because the total duration of the evaluation is quite short and not many packets are generated due to the large round period, the latency of the schedule initialization rounds has a particularly large impact on the average latency.

On the other hand, the 30s periodic case has the lowest current draw. In the aperiodic case, the energy is much higher, even though the same number of messages is transmitted in total. This is because we need to schedule many unused rounds to keep the latency at an acceptable level. This overhead becomes even clearer when comparing the aperiodic case to the 5s periodic scenario: even though more than 6 times as many messages were transmitted during the test, both the average latency and average current draw are lower in the periodic case.

Interestingly, the aperiodic evaluation run with 64 B payloads had approximately the same average current draw as the the run with the 8 B payloads. This indicates that our flag slot is functioning as planned; the overhead caused by the longer payload is only incurred in rounds where nodes actually transmit the message, which is a minority of the rounds. Thus, the impact of the larger payload on the total energy consumption is small.

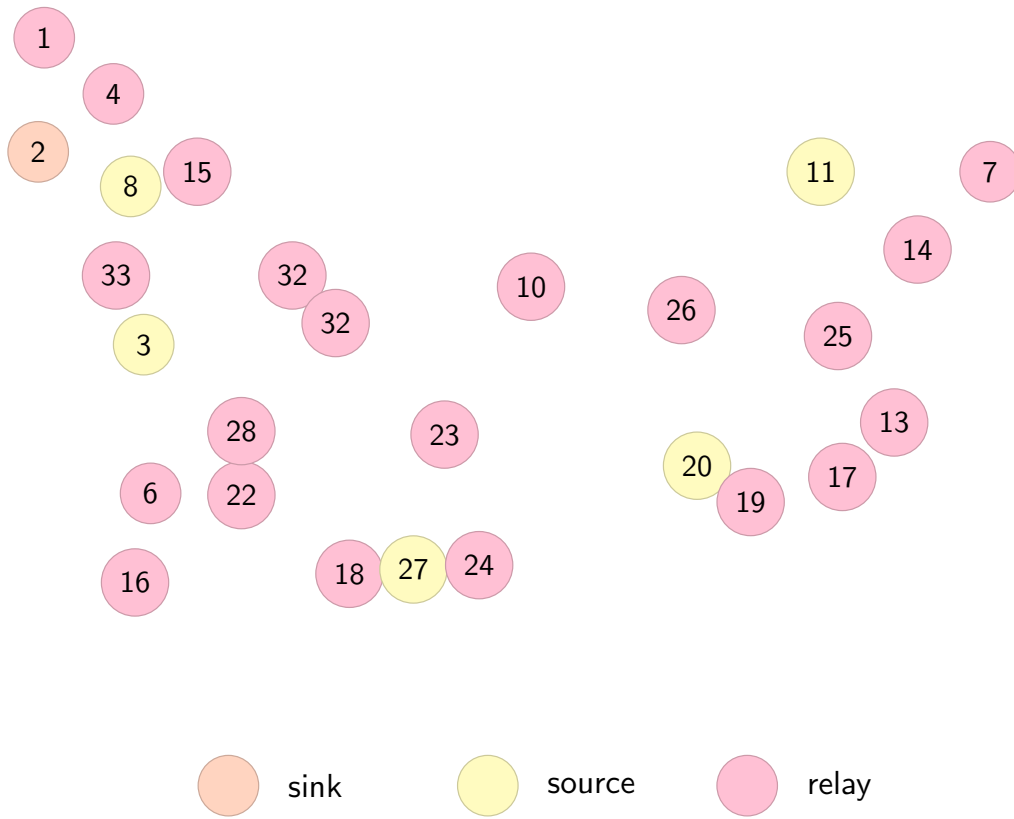


Figure 4.1: Evaluation topology on FlockLab.

4.2 Usability of Baloo

We found Baloo to be a highly valuable tool in designing our protocol. The framework saved us a large amount of time since we did not need to deal with the lower level aspects of ST, which would have been particularly time-consuming for someone with little experience with the underlying platform. Without Baloo, we would likely not have been able to implement a protocol of this degree of sophistication within the timeframe of the project.

However, there were some issues as well. Baloo is still fairly sparsely documented and it would have been difficult to navigate all its different features and use its full potential had we not been working with members of the group who developed it. Also, the rigid structure of Baloo can limit flexibility at times. For instance, Baloo does not provide the option to repeat a part of a round containing multiple data slots out of the box. It is only possible to repeat all slots since the beginning of the round, or a single slot. We were able to work around this issue, and Baloo could likely be easily updated to incorporate such a feature, but it demonstrates well how the framework needs to make tradeoffs between flexibility and complexity.

Conclusion and Future Work

5.1 Conclusion

In this project we have designed and implemented a wireless multi-hop communication protocol that is usable across a range of input conditions. We have demonstrated that Baloo makes it possible to do this in a short period of time even without much prior experience in the field of WSNs. We have seen some promising results when measuring the reliability, latency and energy-efficiency performance metrics of our protocol for a number of different input parameters during preliminary testing, particularly in the absence of strong RF interference. This leaves us optimistic that our protocol will deliver a performance at least comparable to that of other competing solutions across some evaluation scenarios.

5.2 Future Work

While our protocol already delivers good performance across a range of different input parameters, it could be improved further. By doing some more rigorous testing one could fine tune the optimization parameters a little further and reduce guard times where they are redundant.

From the preliminary tests it appears like the protocol performs a lot worse under interference. Thus, introducing in a more elaborate interference avoidance scheme, potentially based on adaptive frequency hopping, would likely be worthwhile. Along those lines, it might also help to introduce a mechanism whereby sources can stop transmissions for a while if they are experiencing heavy interference. The mechanism would need to include a way for nodes to store these messages and transmit them at a later point in time.

Another feature that did not make it into the final protocol but might lead to some performance improvements, particularly in terms of energy, is the concept

of forwarder selection. When forwarder selection is applied, nodes which do not lie on or near the shortest path between the host and a source node are turned off and do not participate in the communication, reducing the size of the network that needs to be flooded. We ultimately decided against using this concept due to the potential reliability cost, which we did not want to risk incurring without more extensive testing.

Bibliography

- [1] M. Schuß, C. A. Boano, M. Weber, and K. Roemer, “A competition to push the dependability of low-power wireless protocols to the edge,” in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, 2017.
- [2] “EWSN 2019 Dependability Competition - Detailed Information,” <http://ewsn2019.thss.tsinghua.edu.cn/competition-scenario.html>, 2018, accessed: 2019-01-18.
- [3] R. Jacob, J. Baechli, R. Da Forno, and L. Thiele, “Synchronous Transmissions made easy: Design your network stack with Baloo,” 02 2019.
- [4] 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*, ser. EWSN '18. USA: Junction Publishing, 2018, pp. 213–214.
- [5] 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*, ser. EWSN '17. USA: Junction Publishing, 2017, pp. 270–271.
- [6] J. Klaue, A. Corona, M. Kubisch, J. Garcia-Jimenez, and A. Escobar, “Competition: RedFixHop,” in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '16. USA: Junction Publishing, 2016, pp. 289–290.
- [7] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, “Efficient network flooding and time synchronization with Glossy,” in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE, 2011, pp. 73–84.
- [8] O. Gnawali, R. Fonseca, K. Jamieson, M. Kazandjieva, D. Moss, and P. Levis, “CTP: An efficient, robust, and reliable collection tree protocol for wireless sensor networks,” *ACM Transactions on Sensor Networks (TOSN)*, vol. 10, no. 1, p. 16, 2013.

- [9] 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*. ACM, 2013, p. 1.
- [10] 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*. ACM, 2012, pp. 1–14.
- [11] 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*. ACM, 2017, pp. 323–332.
- [12] D. Yuan, M. Riecker, and M. Hollick, “Making Glossy Networks Sparkle: Exploiting Concurrent Transmissions for Energy Efficient, Reliable, Ultra-low Latency Communication in Wireless Control Networks,” in *European Conference on Wireless Sensor Networks*. Springer, 2014, pp. 133–149.
- [13] D. Carlson, M. Chang, A. Terzis, Y. Chen, and O. Gnawali, “Forwarder selection in multi-transmitter networks,” in *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–10.
- [14] M. Brachmann, O. Landsiedel, and S. Santini, “Concurrent transmissions for communication protocols in the internet of things,” in *Local Computer Networks (LCN), 2016 IEEE 41st Conference on*. IEEE, 2016, pp. 406–414.
- [15] M. Cattani, A. Loukas, M. Zimmerling, M. Zuniga, and K. Langendoen, “Staffetta: Smart duty-cycling for opportunistic data collection,” in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. ACM, 2016, pp. 56–69.
- [16] “EWSN 2019 Dependability Competition Logistics Information, rev. 2,” https://iti-testbed.tugraz.at/static/upload/EWSN2019_DC_Logistics-2.pdf, 2018, accessed: 2019-01-18.
- [17] 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*. ACM, 2013, pp. 153–166.