Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Real-time network functions for the Internet of Things

Semester Thesis

Fabian Walter
walterf@ethz.ch

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

Supervisors:
Romain Jacob
Prof. Dr. Lothar Thiele

June 26, 2017

Acknowledgements

I would like to thank Prof. Dr. Thiele and the TEC for enabling this thesis. I enjoyed the educational environment and was greatly supported in my learning process. Especially, I would thank my supervisor Romain Jacob. We had many interesting and helpful discussions and he supported me with great passion. Last but not least, I would also like to thank Reto Da Forno. He helped me a lot with his knowledge about LWB, DPP and FlockLab to implement and test the real-time network functions.

Abstract

The Internet of Things is evolving, more and more devices are getting connected, more data exchanged and increasingly complex tasks are handled by these networks. Especially, low-power wireless communication networks are widely used due to the flexibility and fast deployment with a battery. The networks must be at low cost and still guarantee a long battery life time and compute a complex task. The goal of this thesis is to evaluate the outsourcing of a potentially complex task, e.g. real-time scheduling, to another processor and enable a real-time functionality for the whole network. The goal is a proof of concepts of the scheduler outsourcing by evaluating the implementation at the system-level, combining hardware, scheduler and network protocol.

Many algorithms to support real-time functionalities are already developed, but the implementation for existing low-power systems with a single processor is difficult [1]. Low power systems with a single core are often resource limited in memory and computational power. Multiple processors can be used to overcome this problem, but challenges with the interprocessor communication arises. The Dual-Processor Platform (DPP) [2] enables a partition of the tasks between a very low-power communication processor (CP) and more powerful application processor (AP). The CP will be used to handle wireless communication based on the Low-power Wireless Bus (LWB) [3], which is a best-effort network protocol. The Blink scheduler [4] is used to enable the real-time functionality in the network. The potentially complex and memory demanding Blink scheduler is outsourced to the AP. Further, the LWB round structure is adapted, which leverages the DPP platform and allows the computation of the next schedule on the AP while the communication is ongoing on the CP. Therefore, the delay for outsourcing the scheduler is becoming neglectable when the wireless network communication has sufficient many slots per round. In the end, the limitations like additional interprocessor communication delays are analysed and potential improvements are evaluated.

Contents

List of Acronyms

AP	Application Processor
CP	Communication Processor
DPP	Dual-Processor Platform
FIFO	First-in First-out (buffer)
IoT	Internet of Things
IPI	Inter-Packet Interval (packet period)
LSB	Least Significant Bit
LWB	Low-power Wireless Bus
MSB	Most Significant Bit
OS	Operating System
PC	Personal Computer
RF	Radio Frequency
RTT	Round-Trip Time
s-ack	Stream acknowledgement
TEC	Computer engineering group (Lab from ETH Zurich)

List of Figures

List of Tables

Chapter 1

Introduction

The Internet of Things (IoT) is one of the technological trends of the last years. More and more devices are integrated into networks, exchange information and additional services are enabled by using the collected information. The usability of IoT is very versatile and therefore the requirements for such network are also very variable. The focus for this thesis is enabling a real-time functionality in a low-power wireless network by outsourcing the potentially complex scheduling task.

Challenges

Low energy consumption is often an important requirement. Battery operation in combination with wireless communication enables easier deployment and more flexibility, but has limited capacity. The financial resources might be limited and a low price is desirable as well. On the other hand, more and more complex tasks should be executed in these low-power and low-cost networks to enable better and broader services. Low-power, low-cost, and complex functionality are conflicting requirements. The major challenge is to find a energy and price efficient solution, which is still capable to perform complex tasks.

DPP enables new prospects

One possible solution is to use more than one processor. Multiple processors enable new prospects, but have also some challenges like the exchanging of information between the processors. Researchers from ETH Zürich developed a stateful process interconnection called BOLT [2] (see section 2.1). A very low-power processor with RF (Radio Frequency) is combined with a more powerful processor to benefit from the symbiosis. The resulting hardware is called the DPP.

Real-time network function on a DPP?

Real-time network functions can become very complex as the tasks must be completed within a strict deadline. If more tasks must be completed within the same deadline, an increased complexity must be handled without the increase of available time. The shorter the deadline, the bigger the computational complexity per time unit. A real-time network function can for example be the scheduling of task processing or packet sending. Scheduling more tasks/packets and scheduling faster both increase the complexity of the network functionality. The Blink scheduler provides real-time guarantees for wireless communication. An implementation of Blink on a low-power communication processor was realized by J. Acevedo [1]. However, it was limited to a handful of streams to schedule, because the memory and computational requirements were to high for such a low-power and low-price single processor platform. The DPP can be a way to overcome these limitations.

Thesis outline

Additionally to BOLT and Blink, a communication protocol is needed to enable a real-time network functionality. The implementation described later is strongly based on the LWB (Low-power Wireless Bus) [3], because it is energy efficient, simple and provides a good base for adaptation. A goal of the thesis is to show how the DPP can be leveraged in a low-power wireless network to enable a real-time functionality and gauge the potential benefits, like faster computation, energy saving or the parallel execution of tasks. This comes at the cost of latency for communication between the two processors and an increase of complexity for the message exchange. The additional inter-processor latency can be significantly reduced by adapting the LWB round structure. This allows a parallel computation of the scheduler on the AP while the wireless network communication is ongoing on the CP. The delay for outsourcing the scheduler is becoming neglectable when the wireless network communication is longer than the outsourcing delay. This is possible, because the wireless communication time is faster increasing with the increase of slots per round than the computation of the schedule is.

The steps towards the real-time functionality are described thereafter. A general overview about BOLT, Blink and LWB is given in chapter 2. The system design is explained in chapter 3. Chapter 4 is used to explain the implementation of the combination of BOLT, Blink and LWB on the DPP. The evaluation is made in chapter 5 and the thesis ends with a conclusion and a short discussion about the future work in chapter 6.

Chapter 2

Background and related work

Real-time functionality in a low-power wireless network requires multiple components. This chapter gives an overview about the technologies and methods this work is based on. First, the Dual-Processor Platform (DPP) with BOLT is explained to know what the hardware allows us to do. The second section is about the Blink scheduler and how the scheduler enables a real-time function by scheduling when data packets are sent. The final section in the chapter explains the basics of LWB and why it is a reliable and energy efficient protocol for low-power wireless network.

2.1 BOLT

BOLT is a stateful processor interconnetion that enables the communication between two arbitrary processors. This section further explains the hardware setup and is a summary of the publication about BOLT [2].

BOLT is a processor on its own that provides the functionality of exchanging data. BOLT is the interconnection between processor A and C (depicted in Fig. 2.1). Two virtual FIFO (First-In First-Out) queues are emulated in non-volatile memory to store the data packets. The output FIFO buffer of A is the Input buffer of C and vice-versa. Each processor has an BOLT API (Application programming interface) to control the message exchange with the following functions:

- *wtest*()**:** The write-test function returns true when the queue is not full and a write is possible. *wtest*() is handled by the Message Controller of BOLT.
- *write*()**:** The write function writes the message into the queue when not already full.

- *rtest*()**:** The test-read function returns true when the buffer is not empty and a message can be read. *rtest*() is handled by the Message Controller only.
- *read*()**:** The read function requests the oldest message from the input-buffer from the Message Controller. The data is afterwards read from the data line.
- *flush*()**:** The flush functions gives the Message Controller the task to empty the input-buffer, which can be used to prevent a buffer overflow.

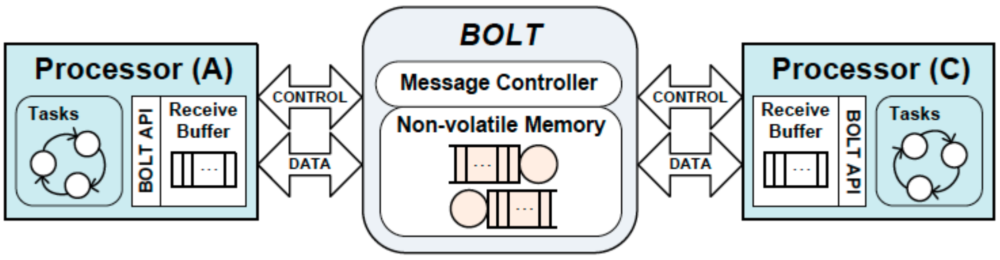


Figure 2.1: BOLT is connecting the two processors (A and C) with two FIFO queues.[2]

The whole communication can be completely decoupled in time as the communication is asynchronous and the two processors can have different clock speeds. Further more, a processor can write into the queue (when not full) even when the other processor is in sleep mode or busy. A processor can periodically check for a new message and sleep in between to reduce the power consumption. Additionally, a non-empty FIFO buffer is signalled with a control line to be high. This control line can also be used to trigger an interrupt when data is available.

BOLT is a general concept and the two processors can be chosen regarding the application specific requirements. The later used hardware with the major specification is shown in Tab. 3.1. The chosen processor combination has the following benefits:

- An extremely low power-processor with a radio to handle the wireless communication and some basic computations efficiently. This processor is called CP (Communication Processor).
- A less energy efficient processor with more computational power and memory resources. This processor is called AP (Application Processor). AP is used for the more complex tasks.

- The energy consumption can be minimised by letting the currently unused processor(s) sleep.
- AP enables the node to execute complex tasks, which would not be possible on the low-power CP.
- BOLT functions are time-bound and enable a deterministic communication.

More details about BOLT, an analysis of the timing and the proof of the functional correctness can be found in [2].

2.2 LWB (Low-power Wireless Bus)

LWB is a time-triggered best-effort communication protocol to handle wireless communication efficiently in low-power networks. LWB is stream-based and globally schedules the streams. Each stream represents a data packet, which is sent with a period defined by the IPI (Inter-Packet interval). The LWB was already implemented on a CC430 (with a MSP430 processor) node and the code is publicly available on GitHub[1]. The code on GitHub is the basic for the implementation explained in chapter 4 and is also used to understand the communication protocol. The informations about LWB are from [3].

2.2.1 LWB is Glossy-based

LWB is a based on Glossy [7]. Glossy is flooding-based and enables a multi-hop communication. If a node receives a packet, it checks the receiver of the packet in the header and has multiple options to do:

- If the destination-ID is the node-ID, put the message into the input buffer. The message can be used in the application routine of the node by reading the buffer.
- If the node is not the sole destination and the data packet has not had the maximum number of hops, send the message to all neighbours by broadcasting it.

It is called a flooding protocol, because the data packets "flood" the network in hopes and waves (see Fig. 2.2 (C)). The flooding mechanism is the opposite of routing, because data packets are sent to all neighbours. Glossy is highly reliable and flexible, because when a route is in the network, the packet will be received

[1] `www.github.com/ETHZ-TEC/LWB`

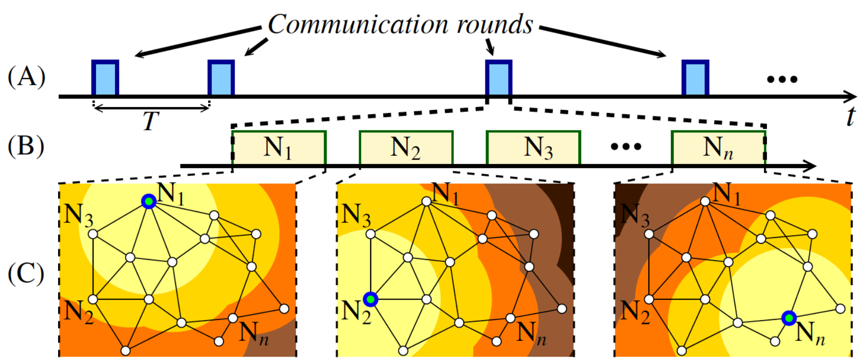


Figure 2.2: Time-triggered operation in LWB. Protocol operation is confined within communication rounds that repeat with a possibly varying round period T (A); each round consists of a possibly varying number of non-overlapping slots (B); each slot corresponds to a distinct Glossy flood (C)[3].

by the receiver. Additionally, the protocol makes it needless to send routing information and detect link or node failures. The nodes are also synchronised and therefore the packet reception rate is further increased, because the same data packet might be sent by multiple nodes at the same time and positive interference increases the Signal-to-noise ratio (SNR). Additionally, the protocol is less complex as no routing information have to be sent and stored. Glossy also allows multiple traffic pattern like one-to-one, one-to-many or many-to-one and many-to-many as eventually every packet is flooded in the whole network. LWB is as a consequence like a bus for the application running on the processor.

2.2.2 LWB round structure

LWB communication is based on a structure with rounds and phases. The communication rounds have different round periods. An example of multiple communication rounds is shown in Fig. 2.2 (A). A round has multiple data slots (depicted in Fig. 2.2 (B)). One data packet is sent in every slot and the data packet floods the whole network during this slot (see Fig. 2.2 (C)).

The communication is globally scheduled by the host node and distributed at the beginning of a round (see Fig. 2.3). The host node and (source) nodes can communicate with each other. The different phases are shown in Fig. 2.3.

The round can have different round periods and number of data slots. The nodes usually are in sleep mode from the end of round k until the beginning of the execution of the round *k* (see Fig. 2.4).

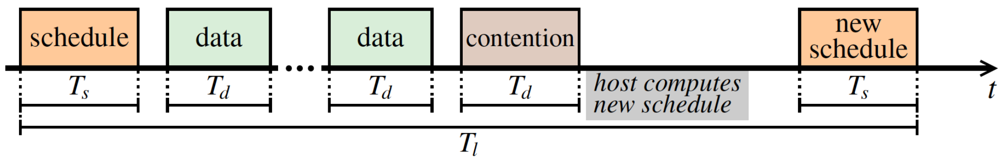


Figure 2.3: Every LWB communication round is separated in these phases. [3]

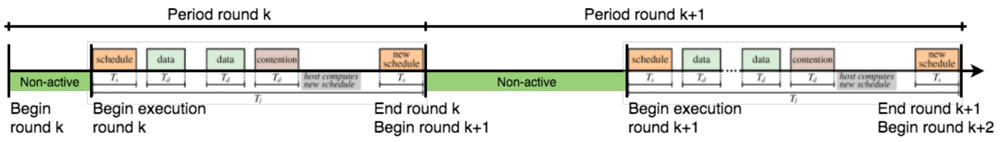


Figure 2.4: Multiple LWB rounds with different round periods (parts of the figure from [3]). The processor can go into a low-power mode when not in non-active phase.

The different phases and functionalities are:

- **Schedule:** The current schedule is sent by the host to allocate the data slots to the nodes. The schedule is also used to synchronise the whole network and give the round period.
- **Data:** The node checks if the current slot is assigned to its node-ID and sends its packet from the output buffer if so. Otherwise, the nodes follows the Glossy rules and helps to flood the packets.
- **Contention:** The contention slot is used to send stream requests and register new streams. It is random access and multiple stream requests from different node are possible to collide, but the Capture Effect[8] makes it very likely that one of the requests are received by the host.
- **Computation of new schedule:** The schedule is computed at the host by considering all registered streams and the potentially new arrived stream request.
- **New schedule:** The newly calculated schedule is sent at the end of the round. Afterwards, the node can go into sleep mode until the execution of the round starts.

The schedule of round k is sent at the end of round $k-1$ and at the beginning of round k. This redundancy is applied, because it is fundamental important to synchronise the network with the scheduler packet and an unreceived scheduler would lead to a node not participating in the round and goes in the unsynchronized state.

Energy consumption A node initialized into the unsynchronized state after start up or goes into the unsynchronized state after a schedule packet miss. The radio is kept on to get the next scheduler packet, which is sent at an unknown time. Such a wait for a scheduler is energy inefficient as the radio must be on until a maximum waiting time is over. Every schedule is sent twice to avoid such a scenario.

The maximum number of data slots per LWB round is a fixed parameter and the radio active time per round is independent of the round duration. Reducing the energy consumption is therefore achieved by setting the round periods as high as possible without violating the allocation streams according to the round times of the streams.

2.2.3 Limitations of LWB for real-time functionalities

The standard LWB scheduler is trading complexity, latency and low-power and schedules the streams periodically. Although, only the period of a stream is defined with the IPI (Inter-packet interval) and used for the scheduling of streams. If multiple data packets are available on a node, the earliest one in the FIFO output buffer is sent at the next slot assigned to the node. As an example (see Fig. 2.5): A stream r allows a high latency at the receiver and arrives at T_r. Another stream g has a smaller allowed latency and the stream arrives at T_g. The node gets a slot assigned at time T. The packet of the stream r is sent in a scenario, where $T_r < T_g < T$. The stream g has to be sent at the next assigned slot and might have a too high latency while the stream r could have afforded a longer latency. Therefore, the LWB by itself does not provide a real-time functionality and another scheduling technique is needed.

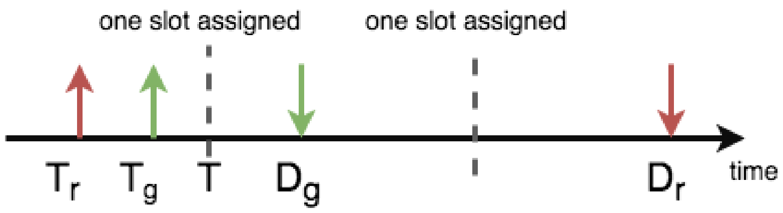


Figure 2.5: LWB is best-effort and does not guarantee real-time, because the scheduler does not consider the arrival time and deadline and the nodes send the data packet, which is arrived earlier.

2.3 Blink

The standard LWB scheduler is limited and does not enable a real-time functionality. Therefore, Blink is used to schedule the wireless communication of data packets and enable a real-time functionality with deadlines. Blink is also very adaptive to traffic demands changing over time and new streams can be requested. This section is based on the explaination and evaluation from [4].

2.3.1 Basic stream parameters

Every node can register multiple streams by sending a stream request to the scheduler. Each stream has the following properties that are used to schedule the stream:

- **IPI:** Inter-packet interval is the period of the stream.
- **Arrival-time:** The arrival-time defines when a packet is available to send. The arrival time can be defined as relative or absolute.
- **Deadline:** The deadline defines the time when the data packet must be received by the receiver. The deadline can be defined relative or absolute.
- **Node-ID:** The node-ID is globally unique and specifies each device.
- **Stream-ID:** The Stream-ID is given by the node itself. The Node-ID and the Stream-ID are both used to uniquely define a stream.

Mind that the scheduler does not have to care about which node is the receiver, because LWB is based on Glossy and just floods the packet with the receiver-ID in the header of the packet. The receiver(s) can just read the header and put the packet into the input buffer or flood it further.

2.3.2 Blink scheduler steps

The goal of Blink is to fulfil all deadlines and doing so as energy efficient as possible. The computation of the schedule is done in multiple phases, which are depicted in Fig. 2.6 and explained in the following subsections.

If not a new stream request arrived at the host, the first two steps can be bypassed as the synchronous busy period stays the same and no stream must be tested for admission. The information in the next sections is just for an overview and more detailed information can be found in [4].

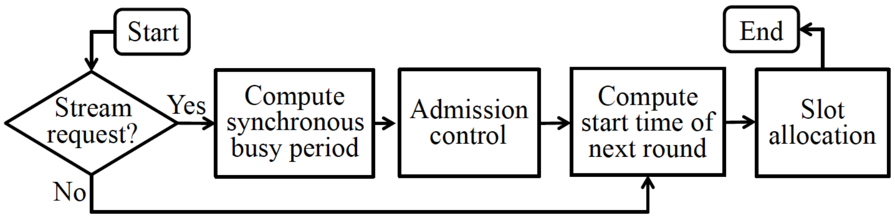

Start
Stream request?
Yes
No
Compute synchronous busy period
Admission control
Compute start time of next round
Slot allocation
End

Figure 2.6: The different steps to compute the next scheduler with Blink [4].

Compute synchronous busy period

The synchronous busy period is the time from when all streams get active in the same moment until no stream has to be handled. It defines the time a scheduler must look into the future to test if the requested stream does not lead to deadline misses. The synchronous busy time can be computed with the formulas 2.1 to 2.3

Variable definition: n is the number of tasks, C_i is the time to handle task i (considering Blink: it is one slot for every task/data packet) and T_i is the period (IPI) of task i.

$$T_b^{(0)} = \sum_{i=1}^{n} C_i \tag{2.1}$$

$$T_b^{(i+1)} = W(T_b^{(i)}) \tag{2.2}$$

$$W(t) = \sum_{i=1}^{n} \lceil \frac{t}{T_i} \rceil C_i \tag{2.3}$$

First, all streams are active and the time $T_b^{(0)}$ is calculated, which is the time to serve all streams once. Some streams might have become (multiple times) active again, because there period (IPI) is shorter than $T_b^{(i)}$. All these reactive streams must be handled again and $T_b^{(i)}$ gets updated until $T_b^{(i)} = T_b^{(i+1)}$. This time is the synchronous busy time. The synchronous busy time is in other words the time until no task has to be handled after all task arrived at the same time.[9]

Admission test

The admission test controls if all streams are still schedulable when the new stream request is accepted. It has to be checked that at every time until the synchronous busy time, all required streams can be allocated without a deadline miss. If the admission fails, the stream will be dropped.

Compute start of next round

The next round should start as late as possible, because the processor can sleep until the execution of the next round starts. Multiple ways to schedule the streams are possible. It was proven that a Lazy scheduler is optimal regarding the energy consumption in [4]. The Lazy scheduler allocates the streams as late as possible and the time before can be used to sleep by the processor. Although, the time can not be too late, because the future stream deadlines must be fulfilled as well.

Slot allocation

Allocating the slots is rather easy after the starting time is calculated. Slot allocation is done with EDF (earliest deadline first) prioritization. EDF is proven to be real-time optimal. Additionally, as many streams as possible are allocated. This means a stream is allocated if it is released and no more slots are assigned as the maximum slots per round. Not assigning a slot, when active streams are available, can make the next round start sooner and wastes resources.

Efficient software implementation

All the streams are located in a bucket queue with size twice the longest period. The streams are sorted by increasing deadline. The next deadline can efficiently found by looking at the first stream in the queue, which has the earliest deadline.

Chapter 3

System design

The Dual-Processor Platform (DPP) with BOLT as an interconnection enables different possibility to partition the tasks between the processors. This chapter explains why it is reasonable to outsource the scheduling task to a processor with more memory and computational resources. The outsourcing of the tasks makes it necessary to send the network information to the scheduler and the schedule from the scheduler back to the network. This chapter also discusses what information must be exchanged and when.

3.1 Task partitioning

BOLT enables a task partitioning between two processors with arbitrary different specifications. The TEC lab, from ETH Zürich, developed the DPP with a very general suitable CP and AP and connecting the two processors with BOLT. First, the available hardware is analysed and the possibilities of each processor are evaluated. As a second step, the requirements of the different tasks are observed. Afterwards, the information of the tasks and hardware is used to partition the individual tasks onto the two processors. It is important to consider and weight the introduced latency and complexity by the interprocessor communication with the benefits of the outsourcing.

3.1.1 Hardware specification

An overview about the possibilities of the three hardware units CP, AP and BOLT is given in this section. The overview is later used to evaluate which processor is most suitable for which tasks.

Communication Processor The specifications for the CP is shown in Tab. 3.1. The available memory resources of the processor can be limiting for memory hungry applications. The processor is actually a System-on-Chip (SoC) and has

	CP	AP
Processor Name	CC430F5147 SoC with RF	MSP432P401R
Max frequency	20 MHz	48 MHz
SRAM	4 kB	64 kB
Flash	32 kB	256 kB
Processor Typ	ARM 16 Bit Cortex M0 (RISC)	ARM 32 Bit Cortex M4F
Active energy consumption	160 pA/MHz	80 pA/MHz
Sleep mode	1 pA	25 nA
Timers	2	2
Additional features	System-on-Chip with low-power wireless communication, multiple low energy consumption levels	Floating point unit (FPU), multiple low energy consumption levels

Table 3.1: The used processor on the DPP board: CP and AP. Additional information like the different low-power modes between sleep and active mode can be found in the data sheet(CP[5] and AP[6]).

a radio frequency (RF) interface for wireless communication directly integrated. The chip has also multiple energy consumption levels between active and sleep mode and can switch during the rounds to safe energy.

Application Processor The specifications of the AP is shown in Tab. 3.1. The memory and computational resources are less limiting and the AP can be used for more complex tasks. The AP has also multiple energy consumption levels between the active and sleep mode and can switch during the rounds to safe energy.

BOLT processor BOLT has evaluated maximal delays for read only, write only and read&write. The timing for this implementation will be tested in chapter 5. The maximum size of a single message can be chosen by the programmer. As an example, each buffer can store 148 messages when the maximum message size is 128 B. The energy consumption when active is lower than the sleep energy consumption of the AP. Additional information about timing, energy consumption and the available slots depending on the maximum messages sizes can be found in [2].

3.1.2 Tasks overview

An overview about the different tasks, regarding how complex in the sense of computational and storage demand, is given in this section. The information is used to make the task partitioning in 3.1.3.

Network protocol and communication The network protocol of LWB is already implemented on the CC430 (see 2.2) and the computational and memory demand can be estimated directly from the code. Although some changes to the protocol are necessary (see chapter 4). The changes should not increase the complexity significantly. The protocol itself handles the preparation of the messages, sending via the RF interface, receiving messages and handle the received stream requests. The program has a linear flow and before every slot, the processor has to wait until the specific slot begins and sends/receives the packet at time specified by the scheduler. The following tasks must be handled by the host:

- **Sending the schedule:** The schedule must be received from the scheduler (AP), encoded and sent. The length of the schedule is upper-bounded by the maximum number of data slots per round.
- **Sending and receiving data:** The LWB protocol handles the communication and sends packets from a buffer when the communication slot is assigned to the host. Otherwise, the host participates in the flooding of received data following the Glossy rules.
- **Receiving stream requests in the contention slot:** At most one stream request is received during the contention slot. The message is short (IPI, start-time, deadline, Node-ID and Stream-ID). The stream request must be forwarded to the scheduler (AP) when scheduling in not done by the host (CP).

The source nodes have similar tasks:

- **Receiving the schedule:** The schedule must be received, stored and the times must be used to start sending/receiving data at the right times. The maximum storage needed is limited, because the slots per round are bounded.
- **Sending and receiving data:** The sending and receiving of the data is exactly the same as for the host.
- **Preparing and sending stream requests in contention slot:** The request are only a few bytes (IPI, start-time, deadline, Node-ID and Stream-ID) and can be encoded quickly.

Generally, the communication protocol needs only a low amount of memory and computational resources. The memory requirements are growing with the streams, which the node wants to send and receive. All the other network traffic is just received and forwarded. Therefore, even for a large network with many streams, the memory needed is limited. The computational resources are also limited, because the sending/receiving of a single data packets has always the same duration. This means if 10 times as much streams are sent, 10 times as much time is available for the sending of the data by the RF chip and the processor has more time to process the buffers. Additionally, the final protocol will be very similar to LWB, which is already implemented on the CC430 with very limited resources. Furthermore, the standard LWB does handle the network protocol and even a very basic scheduler on the host. The tasks should be executable on the CP (also a CC430).

Scheduling The different scheduling subtasks use different information from the network and need a different amount of computational and memory resources. The different subtasks are analysed in this section.

The Blink scheduler was already implemented on MATLAB® for simulation only. The code was analysed and the following perceptions can be made:

- The storage requirements for the streams is growing with the number of streams in the network.
- The allocation of the streams in the scheduled round is very efficient with the bucket queue. Especially, the computational demands are low. The accessing of memory to get the bucket queue information might become problematic for a big set of streams.
- Calculating the synchronous busy time is becoming more complex when the number of streams is increasing. The synchronous busy time is calculated with the formulas 2.1 to 2.3. The synchronous busy time increases in steps when a stream is added to the existing set of streams. The increase of the synchronous busy time depends on the existing set of streams and the new stream request. A prediction of the complexity is difficult to make, because the formula is iterative. Therefore, it is also difficult to upper-bound the complexity.[1]
- The admission test needs the synchronous busy time and checks if the new set of streams is still schedulable. The admission test has to check for deadline misses up to the synchronous busy time plus the maximum round

[1]The computation of the synchronous busy time can be influenced by setting the maximum utilisation lower.

period.[2] The complexity increases with the increase of the synchronous busy time, which is iteratively computed.

- The scheduler complexity is depending on the maximum round period, the maximum number of slots per round and the maximum number of streams in the network (maximum number of nodes in the network multiplied with the maximum number of streams per node). Generally, the allocation of the slots for a round can be done computationally efficient with the bucket queue.

The LWB code has already a very basic scheduler. The scheduler has additional functions to make the protocol more efficient and reliable.

- The scheduler also detects streams, which do not use there allocated communication slots, and removes these streams from the set of all active streams. Non-active streams can happen when for example a node's battery is empty or the node is moved out of the networks reachability. The implementation in LWB counts the unused slots in a row and deletes the stream when the number is above a specified threshold.
- The source nodes compete for a single contention slot per round and multiple source nodes might send a stream request to the host/scheduler. Multiple source nodes might not know if there stream request was received and if there stream request was accepted. Therefore, LWB sends a stream acknowledgement (s-ack) when a stream passes the admission tests and will be scheduled in the future rounds.

3.1.3 Partitioning

The goal of the partitioning is to find an allocation of the task to the processors, which allows a small latency for new streams, small IPI, short deadlines and is still suitable for a low-power wireless network.

RF Communication The RF communication tasks must be handled by the CP, because CP is a SoC with RF capabilities. The sending and receiving of packets are already implemented for the CC430 and it should not be necessary to adapt the low-level part.

[2]The maximum round period is used to limit the clock drift during waiting in the non-active part of the round.

Scheduling The schedule can become potentially very complex. Especially, the admission test and the calculation of the synchronous busy time with the formulas 2.1 to 2.3. The work done by J. Acevedo [1] shows how complex the scheduling can become for a low-power platform. He was able to schedule a handful of streams. He proposed techniques to improve the memory access and enable the scheduling of more streams, but was not able to test it. This limitations can be overcome by outsourcing the scheduling to the more powerful AP. The outsourcing leads to additional interprocessor communication. The scheduler (AP) has to send the schedule to the CP and is also responsible to send the s-ack to the CP, which sends the s-ack into the network.

The detection of non-active streams is a task with low computational demands, but the counters for the unused slots must be stored. The whole set of streams are stored on the AP. If the CP would detect the unused slots, CP must store a counter for each stream and send a message to the scheduler (AP) when one counter reached the threshold. Outsourcing the non-active stream detection would further relieve the CP and the AP can easily handle the counting directly where the streams are stored. The additional communication needed is regarded a small overhead compared with the CP must have to store information about all the streams in the network. Therefore, the outsourcing of the detection of non-active streams was chosen.

Communication Protocol (adapted LWB) The protocol was already implemented on a CC430 processor and the CP is capable to handle the protocol. The LWB protocol will be changed, but a drastic increase in complexity is not expected. The question is: Would it be possible and reasonable to outsource (parts of) the protocol to the AP and can it improve the overall performance? AP would be capable to execute the protocol faster, but the sending is time-triggered and an outsourcing would only lead to an increase in interprocessor communication, which can be slow. There is no benefit of outsourcing it, because the communication protocol is mostly only waiting for a slot to start and reading or writing buffers from/to the wireless communication chip. The read and writes would only take longer when the interprocessor communication is needed. Therefore, there is no benefit in outsourcing the communication protocol to the AP. The protocol is directly executed on the energy efficient CP.

Summary of partition The evaluation of task complexity and hardware capabilities showed the outsourcing of the scheduling is necessary and very reasonable. This was already expected at the beginning, but considering multiple possibilities was needed to find the drawbacks of the different choices, see the additional interprocessor communication demands (in section 3.2) and the potential problems of the implementation (in chapter 4). Generally, everything, which is considering the storage of streams, acceptance of new streams, deleting of non-active streams and the computation of the schedule is outsourced to the AP. Additionally, the outsourcing enables new prospects to use parallel execution, which can be used to reduce the minimum achievable round period or at least overcome the communication delays (see chapter 4 for the implementation and 5 for the evaluation).

3.2 Data exchange and overview of phases in a round

The outsourcing of the scheduling task makes it necessary to distinguish the CP and the AP from the host. CP_{Host} and AP_{Host} are now together the host and incorporate to deliver the host network functions. The CP sends the stream/schedule request to the AP and AP sends the computed schedule to CP.

The round structure is still the same as in LWB, but additional exchange of information is necessary. The round structure with the passing of messages is depicted in Fig. 3.1. The schedule and s-ack, which are sent at the beginning of round k, were computed in round $k-1$. The schedule of the current round and the s-ack for the stream request from the last round are sent from the CP and floods the network. Afterwards, the nodes and the host can send data when they get slots assigned by the scheduler. The contention slot is used to send stream requests. CP_{Host} has to forward the stream request and the AP_{Host} makes the admission test, updates the stream list, manages the bucket list, computes the schedule and sends the schedule to the CP_{Host}. The new schedule $(k+1)$ is send at the end of the round.

The theory about the LWB does not include a "Manage streams" phase in the LWB round (shown in blue in Fig. 3.1). All the slots and phases of the LWB round are strictly time-driven and there is no time for checking the state of a stream and putting messages into the output buffer for active streams, because this would delay the timing. Checking the state of the streams and putting messages into the output buffer is done in the "Manage streams" phase by all nodes. Additionally, the CP_{Host} uses this time to send a message with all unused communication slots, which are used to detect non active streams by the scheduler. The CP_{Host} can also directly send a host stream request in this phase.

It is fundamental to ensure the start of the next round is on time and the execution of the managing stream functions is finished, because the node will lose synchronisation with the network otherwise.

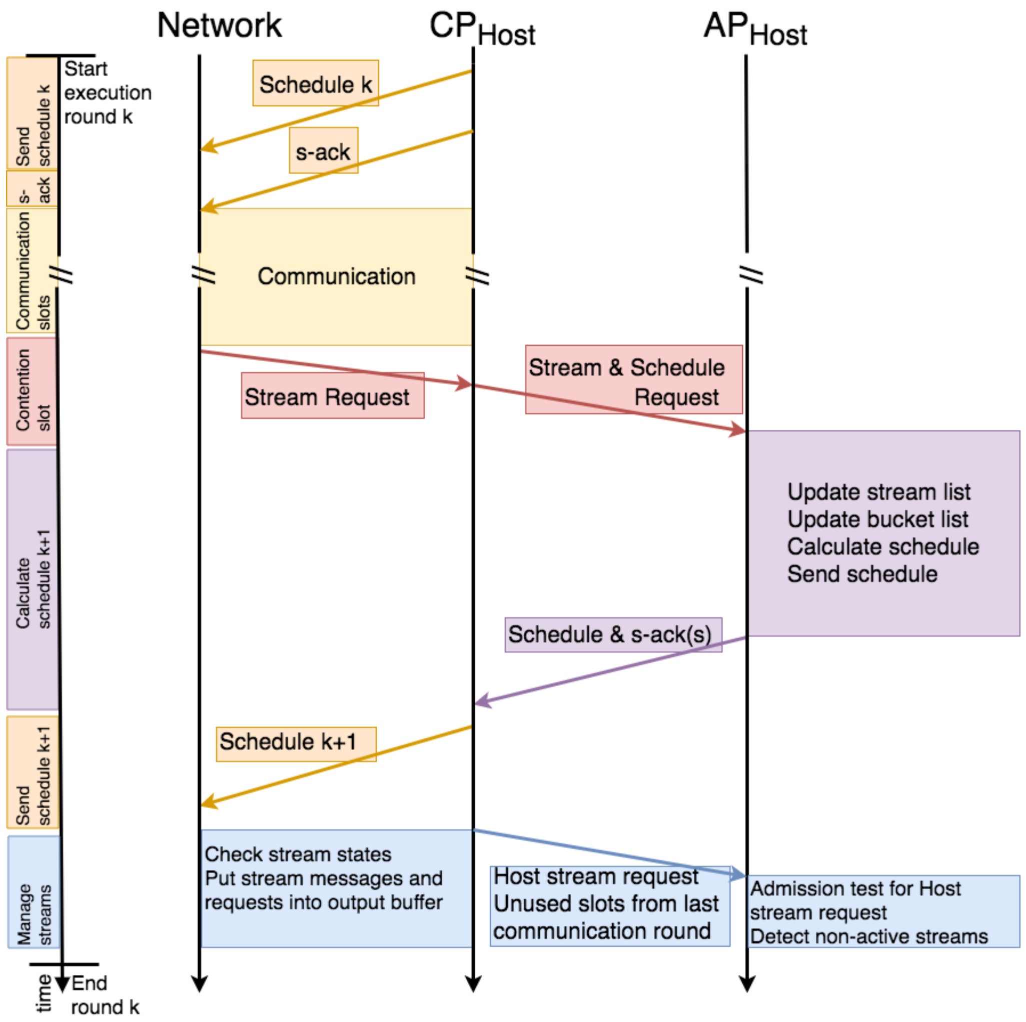


Figure 3.1: The outsourcing of the scheduler makes additional communication between CP and AP necessary.

Chapter 4

Implementation

After the partitioning and the evaluation of the interprocessor message exchange in the previous chapter, the tasks have to be implemented. Many functions are already available as for example the LWB code and the basic Blink for simulations on MATLAB®, but some fundamental changes are necessary. Each section in this chapter is considering the implementations for one processor. The scheduler is first implemented on a personal computer (PC) in MATLAB® to show the feasibility of the outsourcing and use the wide debug and testing possibilities. The schedule is sent from the PC with serial to the AP and the AP sends it to the CP via BOLT. The schedule/stream requests are sent from the scheduler to the CP via AP and BOLT. The setup with CP, BOLT, AP and PC is shown in Fig. 4.1.

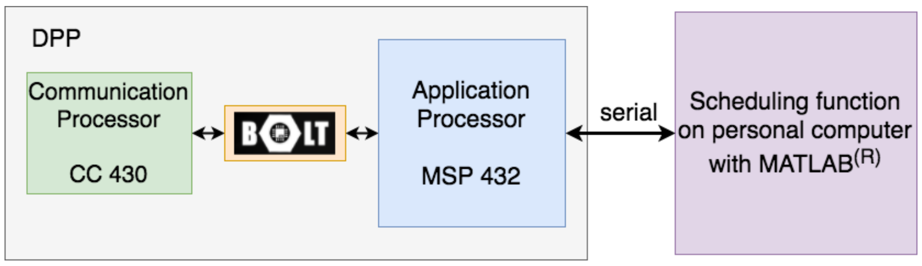


Figure 4.1: The scheduling function is further outsourced to a PC, which is connected with the AP per serial communication. This setup allows more debug possibilities compared to the direct implementation on the AP.

This chapter also focuses on the debugging and testing of specific functionalities. The existing code is distributed over tens of sources files and multiple thousand lines of code. Changing and adding code must be verified well before the next functionality is implemented, because searching a bug on multiple platforms with each tens of source files can be very tedious. The code is tested with some mini-software testbed or by setting up scenarios and observe the serial outputs.

Additionally, the LWB is time triggered and the executed commands between the phases can not have a longer execution time than defined in the configuration. The implementations in this chapters are tested with a standard setup, which has enough spare time. This enables the separation of checking the functionality and optimising the parameters (see also chapter 5 for the different parameters evaluated).

An overview about the implemented functions is given in the next sections and more details can be found in the project repository with the source files and comments. The most changes and implementations have been done for the CP code. The most important source files for the CP are:

- **core/net/scheduler/sched_outsource.c** The most scheduling related functions are implemented in this file. The functions are used to outsource the scheduler by sending the network information to the scheduler and receiving the schedule from the scheduler.
- **core/net/scheduler/lwb.c** The LWB round structure is implemented in this file. The functions and Macros for the wireless communication are implemented in this file and its header file. "wait_until" functions are used to wait until the next slot or LWB round phase start and used to trigger the functions to send or receive data. This file includes a separate function for the host and source nodes.
- **apps/sched_outsource/config.h** Most of the LWB protocol related definitions and parameters are defined in this file.
- **apps/sched_outsource/defines_outsource_sched.h** This header file was added to have a central header file with all definitions and parameters considering the outsourcing of the schedule. Many code structure were implemented to switch between different implementations by using defines and "#if". E.g. the LWB round structure can be changed by changing the boolean value of the define "CONTENTION_BEFORE_COMMUNICATION_AND_SACK". Other defines can turn of the detection of unused slots or enable a debug mode, which emulates stream requests without a connection to source nodes.

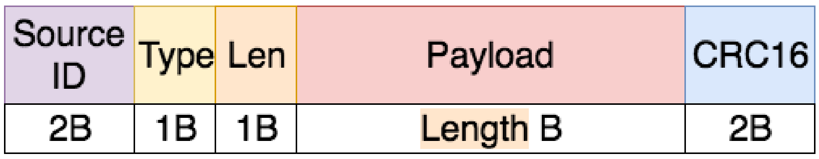
First, the information needed by the scheduler is analysed to define the communication packets and interfaces of each task. Afterwards, the changes in the LWB protocols are explained. The next section is about the tasks of the AP and afterwards, the changes on the Blink scheduler simulation on MATLAB® software is explained. Finally, all individual implementations are considered to find methods, which leverage the outsourcing of the scheduling task.

4.1 Data exchange

The data is exchanged over BOLT or the a serial connection. Both connections have different characteristics and need different packet structures, which are explained in this section.

4.1.1 Packets structure between CP and scheduler/ AP

CP and AP communicate over the highly reliable BOLT. A very short and efficient packet is implemented (shown in Fig. 4.2). The first field is the Source-ID of the sender. Currently, it is not checked, but the AP could also get data packet from another node in the network when additional functionalities are implemented. BOLT can not give an exact length of the packet[1] and a field with the payload length in bytes must be specified (shown as "Len" in the Fig. 4.2). The field "Type" is used to define the payload. Potential payload types can e.g. be a schedule packet or a stream request packet. The payload includes the actual data. The CRC16 is a cyclic redundancy check code with 16 bits to detect and correct bit errors. The CRC16 field is currently sent, but not used. It could be used when a less reliable interconnection is used or the serial communication has a non-zero bit error rate.



Source ID	Type	Len	Payload	CRC16
2B	1B	1B	Length B	2B

Figure 4.2: The packet structure for the communication between scheduler and the CP. The "Len" field is the payload length in bytes.

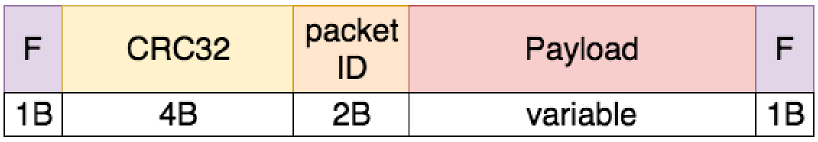
The maximum BOLT message size is a define in the AP and CP code. It is currently set to 256 Byte, because the overhead in decoding a message in multiple packets can be saved. If a packet should be greater than 256 Byte, a "packet-ID"[2] must be added. A packet over 256 Byte is rather unlikely, because scheduling packets are the longest packets and a scheduling packet must have over 80 allocated slots in a single round to reach the size of 256 Byte (see section 4.1.3).

[1]The BOLT processor runs on different clock speeds than the other two processors. BOLT signals the end of the message by changing a control line, which might be a few clock cycles slower than the processor reading.

[2]Additionally, the "Source-ID" or parts of the "Type" field could be reused for the "packet-ID".

4.1.2 Packets structure between AP and PC

The packet exchanged over the serial connection between the AP and the PC has the structure depicted in Fig. 4.3. The "F" is for the "Framing Byte", which is used to detect a beginning and the end of a message. If a "Framing Byte" is in the actual message, a "Escape Byte" is sent before the "Framing Byte". A "Escape Byte" in the message is encoded as two "Escape Bytes". The CRC32 is a cyclic redundancy check code with 32 bits. It can be used when the serial connection is unreliable to detect and correct bit errors. The CRC32 is currently not used. The "packet-ID" can be used to numerate multiple packet. The "Framing Byte" is defined as 127 (0x7F) and the "Escape Byte" as 126 (Ox80).



F	CRC32	packet ID	Payload	F
1B	4B	2B	variable	1B

Figure 4.3: The packet structure for the communication between AP and PC. F is a "Framing Byte" to define the beginning and end of a serial connection. A "Framing Byte" within the message is sent by adding a "Escape Byte" before.

4.1.3 Payload structure

This section describes what the payload encodes and why some design choices have been made. The general goal was to make the communication flexible, efficient in message size and also efficient for debugging.

Payload for packets between AP and PC The payload for the packet sent between the AP and the PC is the packet exchange between the CP and scheduler (shown in Fig. 4.2). The AP can get the payload out of the packet from the PC and send it directly via BOLT to the CP. Additional, the same packet structure can later be used when the AP performs the scheduler tasks.

Payload for packets between CP and scheduler The encoded data in the "payload" field depends on the "Type" field from Fig. 4.2. All the payload types are shown in Tab. 4.1 and explained in the next paragraphs.

HEADER_REQUEST_SCHEDULE The CP requests the next scheduler with payload shown in Fig. 4.4. The included $Time_{CP}$ is the system time of the CP when the message was sent. The time is not used for operation, but for debugging analysis of the evolution of the CP time. This message type is

Payload type	Value
HEADER_REQUEST_SCHEDULE	1 (0x01)
HEADER_STREAM_REQUEST	2 (0x02)
HEADER_SCHEDULE	3 (0x03)
HEADER_DELETE_STREAM	4 (0x04)
HEADER_UNUSED_SLOTS	16 (0x10)
HEADER_HOST_STREAM_REQUEST	18 (0x12)

Table 4.1: The different payload types for the communication between CP and scheduler.

sent when the contention slot was unused and no stream request arrived in the actual round.

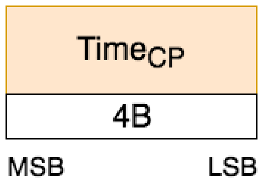


Figure 4.4: The payload for message type HEADER_REQUEST_SCHEDULE.

HEADER_STREAM_REQUEST The message with the type stream request sends the stream request from the contention slot to the scheduler, which makes the admission tests and eventually adds the stream to the list of active streams. Additionally, the message also requests the schedule. The payload is encoded with the Node-ID and Stream-ID from the requested stream. Additionally, the IPI, start time and deadline are in the payload. The start time was chosen to be relative to the beginning of the round in which the stream request was received. The deadline is relative to the start time and must be smaller than the IPI. A node can deregister its stream by sending a stream request with its node-ID and the stream-ID of the stream to deregister and set the IPI of the stream request to 0.

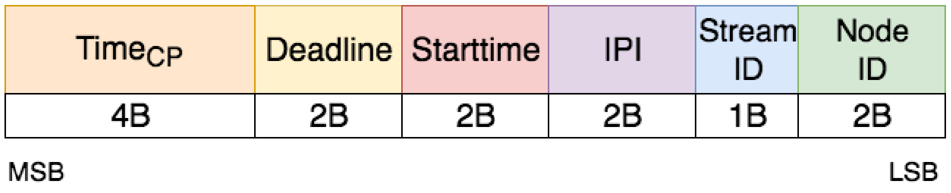


Figure 4.5: The payload for message type HEADER_STREAM_REQUEST.

HEADER_SCHEDULE The schedule packet can be the longest packet and includes multiple fields (see Fig. 4.6). The field "Round period" is the time between two round starts and is defined by the schedule. The field "NumAllocatedSlotS-Ack" includes the number of allocated slots, if a s-ack is included and if a host s-ack is included. The formula 4.1 describes the value in the "NumAllocatedSlotsS-Ack" field in pseudo code.

$$NumAllocatedSlotsS - Ack = bitshift(NumAllocatedSlots, 2) + bitshift(s\_ack\_depending, 1) + host\_s\_ack\_depending; \quad (4.1)$$

"NumAllocatedSlots" is the number of allocated slots in this round. "s_ack_depending" is 1, when the stream request from the last round was accepted and the "host_s_ack_depending" is 1 when the host stream request from the last round was accepted.

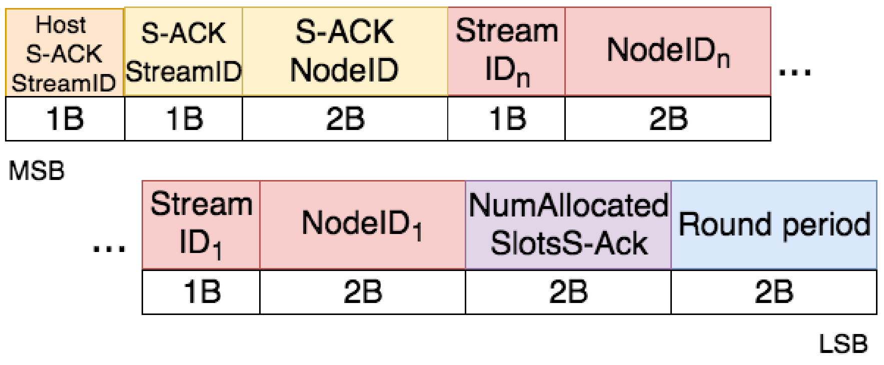


Figure 4.6: The payload for message type HEADER_SCHEDULE.

A slot is assigned to a stream and each stream is defined by the Node-ID of the source and the Stream-ID. The red fields (from Fig. 4.6) specify the stream, which is allowed to send a message in the slot defined by the index. The number of slots n is the value "NumAllocatedSlots" defined with the previous field. If "s_ack_depending" is 1, the yellow packet from Fig. 4.6 defines the s-ack to send in the round. This field is only sent, when a stream is acknowledged. The stream requests are acknowledged, because the contention slot is a random access channel and a node has to know if its request was accepted, because the node would otherwise just resend the request. The orange field from Fig. 4.6 is only sent, when a stream from the host is acknowledged. The host Node-ID is not needed to send, because there is only one host in the network and the host directly processes the host s-ack without sending it and sets its stream to active. The Tab. 4.2 shows how long each part of the schedule packet is. Currently, all messages are sent in a single packet. Therefore, it must be ensured the maximum BOLT message size and buffer size for the serial communication is at least as big as $(n * 3 + 14)B$. n is the the maximum number of slots per round.

Name	Length
packet header	$6B$
Round period	$2B$
NumAllocatedSlotsS-ack	$2B$
Allocated Slots	$n * 3B$
s-ack	$3B$
host s-ack	$1B$

Table 4.2: An overview of the fields and sizes of a schedule packet when both s-acks are sent and n slots are allocated.

HEADER_DELETE_STREAM The delete stream header is used when a stream is requested to be deleted/deregistered. The nodes could also send a stream request with IPI of 0, which means deleting of a stream. This function is currently only used to delete streams during testing and debugging phases.

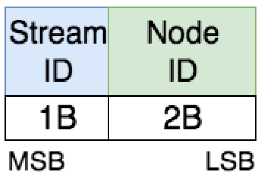


Figure 4.7: The payload for message type HEADER_DELETE_STREAM.

HEADER_UNUSED_SLOTS The message with the unused slots is sent to detect non-active streams. The payload is shown in Fig. 4.8. The scheduler knows which streams were assigned in which slot. Therefore, it is sufficient and more efficient to send only the slot index (starts at 0 for the first slot index). The message is sent without payload when all slots have been used. The payload length is defined in the packet header and is used to get the number of unused slots.

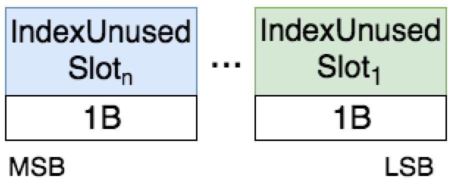


Figure 4.8: The payload for message type HEADER_UNUSED_SLOTS.

HEADER_HOST_STREAM_REQUEST The message with type host stream request is similar to the type with the "normal" stream requests, but does not request the computation of the next schedule and does not include the $Time_{CP}$. This message is sent when the host requests a new stream for itself. The host does not have to use the contention slot and can also send a separate message within the rounds. The host might have a centralized role in the network and has to send more streams than a normal source node and this message type enables a faster adoption to changing stream demands from the host.

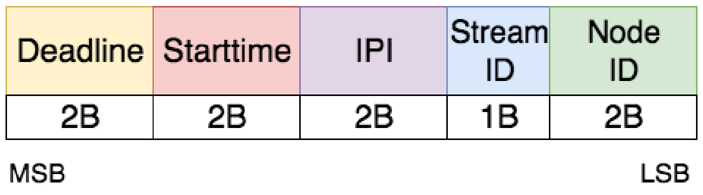


Figure 4.9: The payload for message type HEADER_HOST_STREAM_REQUEST.

4.2 Communication Processor

The basic functions are already implemented in the LWB code. The main protocol function is in the file "lwb.c" in the directory "/core/net". The functions for the different LWB round phases are called here. "wait_until" functions are called in between the phases and slots to enable the time-triggered execution. The changes and implementation of functions will be explained in the next sections. First, the changes to the handling of streams are explained and afterwards, the communication with the scheduler is described.

4.2.1 Enable scheduling of individual streams with start time and deadline

The standard LWB is assigning a communication slot to a node, but not to a specific stream. Scheduling nodes and not individual streams can destroy the real-time functionality (see the paragraph about the LWB limitations in section 2.2.3). Additionally, the LWB stream request do not include a start time and a deadline.

Stream request with start time and deadline The stream requests are defined in the struct "lwb_stream_req_t" in the file "/core/net/scheduler.h". The basic protocol has already an array of type *uint_8_t* and length "LWB_CONF_-STREAM_EXTRA_DATA_LEN". Defines in "defines_outsource_sched.h" set the extra data length to 4 (bytes) and specify which index of the array is used for which byte of the start time and deadline. Additionally, the functions to handle stream request were updated to read the extra data from the stream request and encode it for the scheduler.

Slots are defined by Node-ID and Stream-ID The original LWB does assign communication slots to a Node-ID. The slots must be assigned to a specific stream to ensure the "right" stream is sent. This means the slot must also be specified by the stream-ID, which adds another one byte to every assigned slot. The slots assignment would be 2 bytes for the Node-ID and 1 byte for the Stream-ID. It was not clear if the increased message size can be supported, because the sending of the schedule needs more time and memory. Therefore two different implementations are made, which can be selected with the defines in "defines_outsource_sched.h":

- The define "SCHED_SLOT_SIZE" is 2 and the slot is defined by a 2 byte variable. This is the same as in the standard LWB. The node-ID and stream-ID are combined into one stream-node-ID, where the stream-ID is bit shifted and added to the node-ID. Another define will specify the bit shift and the bit masks to get the node-ID and stream-ID. Therefore, the 16 bits of the stream-node-ID can be flexible assigned. A network with many nodes can assign a higher range of the 16 bits for the node-ID and the stream-ID range is limited. The standard is 8 bit for the stream-ID and node-ID each. This allows 256 different stream-IDs and 256 different node-IDs.
- The define "SCHED_SLOT_SIZE" is 4 and the slots are defined with a 4 byte variable. The bit shift define can be set to 16 (and the bit masks accordingly) and the whole range of the node-IDs (up to 2^{16}) and stream-IDs (up to 2^8 per node) can be used.[3]

The bit shifts and bit masks defines are now used in the "lwb.c" code to get the stream-ID for which the slot is allocated.

Separate output buffers for streams The standard LWB code has one FIFO output buffer for the messages to send. The knowledge of the stream-ID cannot be used to find the allocated stream when all the data packets are in the same output buffer. A possibility was to also add the stream-ID to the packets in the output buffer, but it would be necessary to search in the buffer and a single stream could occupy the whole buffer with multiple messages. Therefore, multiple FIFO-buffers are implemented.

[3]REMINDER: This was not completely tested and can lead to timing problems.

The function call "lwb_out_buffer_get(glossy_payload.raw_data)" in "lwb.c" stores the first message in the output buffer at the location of "glossy_payload.raw-_data". This function was changed and has also the stream-ID as parameter. The function is defined in "lwb.c" and it uses the FIFO buffer functionality from "fifo.c". The FIFO functions need a pointer to the memory location, where the buffer is stored. An additional layer is added to get the pointer, which depends on the stream-ID. These functions are in the source file "apps/sched_outsource/-stream_centered_output_buffer.c". The functions to set up the buffer(s), register new streams (in function "lwb_stream_add") , bind new streams to a output buffer and remove the binding with the buffer when a stream is deleted is also implemented in the same source file.

Sending streams The functionalities must also be tested and therefore the file "apps/sched_outsource/lwb-test.c" is used.

The state of stream is tested with "lwb_stream_get_state(node-id)". If the state is non-active (means stream request not sent or no s-ack received), the stream request is sent (again). The node can put a stream request into the output buffer with the function "lwb_request_stream(&my_stream, 0)", where "my_stream" is a stream request defined in the file and 0 is the priority of the stream request.[4]

If a stream is active, it puts a dummy data packet into the output buffer of the stream with the function "lwb_send_pkt(receiver_node_id, stream_id, &data, payload_size)". All the nodes have a serial debug output when a packet arrives with the receiver-ID equal to the node-ID and the sending and receiving of the data can be tested. Multiple streams have been registered on the same node to test the functionality of the output buffer, the sending and also the receiving of packets.

4.2.2 Send the Data to the scheduler

Outsourcing of the scheduler function makes it necessary to exchange data between the network and the scheduler. The Fig. 3.1 gives an overview about the data exchange when the scheduler is located at the AP. The AP is currently used to forward the packets from CP to the PC and vice versa. The whole outsourcing on the AP and in MATLAB® is summarized in the violet box (in Fig. 3.1).

[4]LWB also allows to have high "priority" streams requests, which are sent in the next contention slot or if earlier, in the next communication slot assigned to the node ("piggybacking"). "Piggybacking" is not implemented for the outsourced scheduler and the priority of 1 is not allowed when real-time scheduling is executed, because stream request can use data slots and the actually allocated stream will not be sent in this round, which can leads to deadline misses.

Stream requests The stream requests are received in the contention slot (see section 4.1 for the packet structure or the source file "lwb.c"). The received packet is given to the function "lwb_sched_proc_src", which sends the stream request to the host with the packet structure defined in Fig. 4.5.

Schedule request The schedule request is only sent when no stream request is sent, because the stream request also triggers the computation and sending of the next schedule. The schedule request is called after the contention slot (see "lwb.c"). The function to send a stream request is called "send_schedule_request()" and sends the stream request packet (see section 4.1 for the packet structure).

Host stream requests The host is scheduling all streams in the standard LWB and the host just added streams for itself. The outsourcing makes it necessary for the host to send a stream request as well. Two different solutions have been considered: A separate host stream request message and combining the stream request with a normal stream request from the contention slot.

- **Combine stream request with normal stream requests:** The first implementation for the host stream request was just to send the host stream request during the normal contention slot, because most of the stream request functions for the normal nodes can be reused. Although, it is problematic to send multiple stream requests at once, because the time to compute the admissions tests is increasing, which can lead to timing problems with the time-triggered LWB rounds. The host has a centralized role and it is likely the host has to send more streams than the normal source node. Therefore, the host got a higher priority and if the host has a stream request, the normal stream request from the contention slot is ignored. Unfortunately, the used functions for the source-node stream requests have not been developed for the host and the implementation failed when a s-ack for a host stream was received. The error was evaluated in the clearing of the bit in the variable "lwb_pending_requests", which is used to keep track of the unacknowledged streams. An elegant and easy solution was not available. The additional effort for a separate host stream request was regarded smaller and the sending of separate host stream request also enable to request two streams per round (see paragraph below).
- **Separate host stream request:** Therefore, the solution with an individual stream request was implemented. The function to send a stream request is "lwb_stream_add(&my_stream_request)"[5], which is changed to send a host stream request in a packet structure explained in section 4.1.

[5]my_streaam_request is from data type "lwb_stream_request" and includes all information for a stream request.

This function can be called during the "Manage streams" phase (see Fig. 3.1) and sends the message directly to BOLT.

Unused slots The unused slots are needed to detect non-active streams and delete these, because non-active streams can happen in a low-power wireless network and waste communication resources.

The current slot index is added to the array "unused_slots" when a data slot was unused in "lwb.c". The variable "index_unused_slots" is also increased every time a slot is unused (in the communication phase in "lwb.c"). The array is sent at the end of the round in the packet structure explained in section 4.1).

4.2.3 Receive the Data from the scheduler

The outsourced scheduler has to send data into the network like the schedule and s-acks. The s-acks and the schedule are sent in one packet, which makes the encoding of the packet easier and safes the overhead of sending/receiving two packets. The packet structure is shown in Fig. 4.4.

The receiving is implemented with a BOLT interrupt, because the time needed for the outsourcing is variable. The interrupt is defined in the function "handle_bolt_interrupt" from file "sched-outsource.c". The interrupt should not be executed during the communication of the network, because the host would miss the timing for sending and receiving. The interrupt must be disabled during the network communication phases. The interrupt is activated at the end of the function "lwb_sched_compute(...)", which is after the communication slots, and deactivated after the execution of the interrupt routine.

4.3 Application Processor

The final task of the AP is to calculate the schedule. This functionality was not implemented, yet, because outsourcing the scheduler to MATLAB® on a PC enables a wider range of debug possibilities like manually set up a set of streams and change parameters fast (see section 4.4). The AP is currently used for the communication between the CP and the scheduler running in MATLAB® on the PC.

The message exchange between PC and AP is serial and the message exchange between AP and CP is over BOLT. The overall data exchange between CP and PC via BOLT, AP and serial is considered in this section.

4.3.1 Communication from PC to CP via AP and BOLT

As a first step, the communication from PC to AP via serial and from AP to CP via BOLT is explained.

The standard LWB code for the DPP has already implemented a console-like function on the AP. The function "console" in the source file "console.c" compares the input with predefined function names. The implemented functions are read and write BOLT and get status informations from the system. Whenever a character (single byte) arrives via the serial port, an interrupt is triggered, the character is added to an array and the array is compared to the predefined function names.

The interrupt functionality was kept for the serial communication and the function in "console.c" was changed to handle the new communication. The packet structure from Fig. 4.3 with the "Framing Byte" and "Escape Byte" is used to detect the start and end of messages. All the bytes from the serial port are read individually and the received message is sent via BOLT after the "Framing Byte" is received. The serial connection has a Baud-rate of 115200 $symbols/s$. The duration of the serial write is evaluated in section 5.1.2.

A first experiment was made to control the timing and functionality of reading a message from BOLT. The time was measured after the message was written to BOLT by the AP and until the message was read by the CP. The results are shown in Fig. 4.10 (orange line). The duration was very constant and other measurements in Fig. 4.11 and Fig. 4.12 (orange line in all figures) show the same results. The CP reacts fast to the available message and reads the message. The delay is very predictable and less than 2 ms. The main reasons are the deterministic operation of BOLT and the CP is interrupt triggered to read BOLT messages. Therefore, no changes to the first implementation is necessary.

4.3.2 Communication from CP to PC via AP and BOLT

The first implementation of the communication was in the main routine of the AP. The Macro "BOLT_DATA_AVAILABLE" is used to check whether a message is in the BOLT input queue. If a message is available, the message is read and send via serial to the PC. The time was measured in one of the firsts tests and showed the duration of AP detecting a message in the BOLT queue and reading the message. The results are shown in Fig. 4.10 (blue line). The message size is slightly varying. The measurement was difficult to analyse and therefore an observation with a constant message size (CP is always sending a schedule

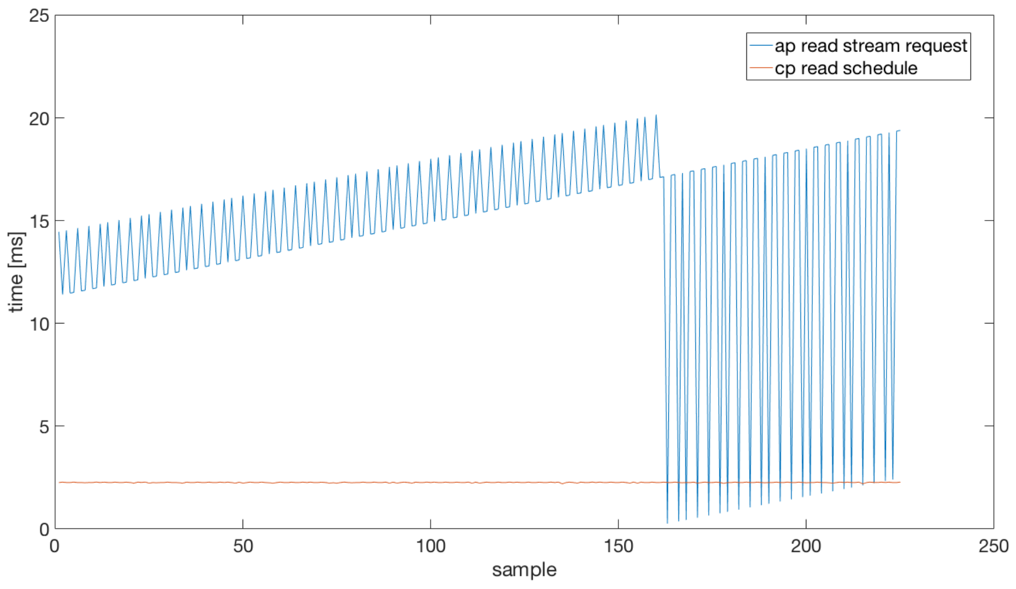


Figure 4.10: The time for detecting a message in the BOLT queue and reading the message for the CP and AP. The message size for the communication from AP to CP was constant. The message size from CP to AP was not constant and AP used polling to read BOLT. The CP used an interrupt to read the message and the reaction was very predictable and fast.

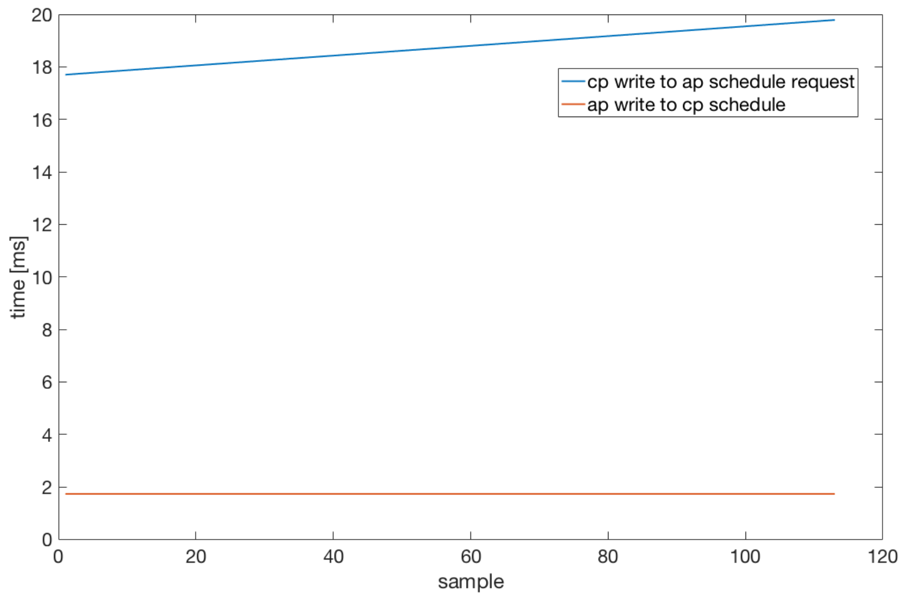


Figure 4.11: The time for detecting a message in the BOLT queue and reading the message for the CP and AP. The message size of the both communication was constant and the AP used polling to read BOLT. The CP used an interrupt to read the message.

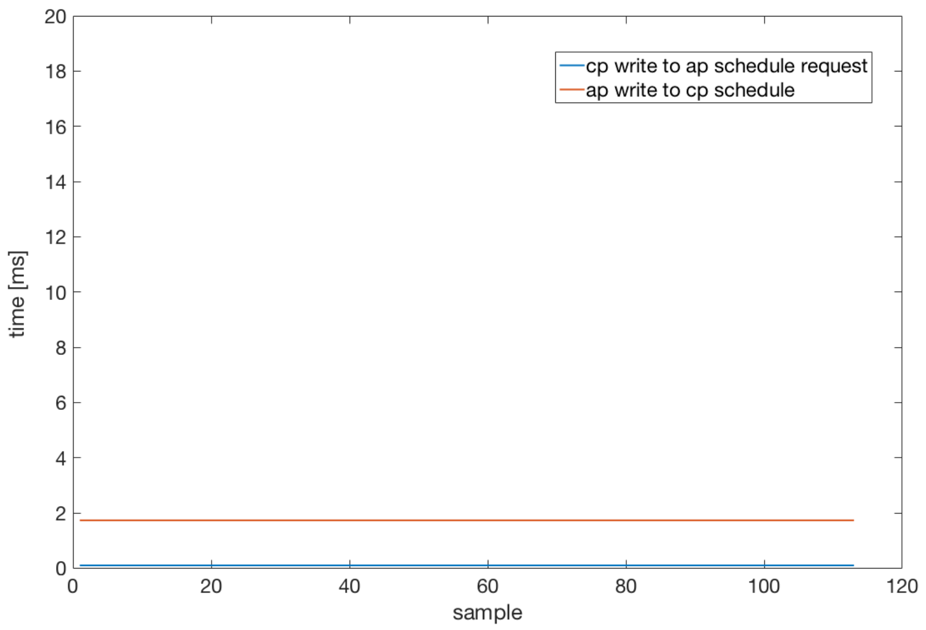


Figure 4.12: The time for detecting a message in the BOLT queue and reading the message for the CP and AP. The message size for both communications were constant and the AP and CP used an interrupt to read from BOLT.

request and no stream request) was done and the result is shown in Fig. 4.11. There is still the linear increase in the duration. The execution time of the main function is roughly $20ms$ (see Fig. 4.10 when the duration toggles from close to $0ms$ and $20ms$). Most likely the duration will drop to nearly $0ms$ when close to $20ms$ as well, because this is the difference when the main function is started directly after the BOLT message is written and the BOLT message is written directly after the main function was executed.

The time duration has a very large range of possible times and it is not the perfect implementation as we want a predictable and tight upper-bound. The results in Fig. 4.10 and Fig. 4.11 for the CP reading suggest an improvement when the AP also uses an interrupt to read the BOLT messages. The interrupt was implemented in the source file "main.c" at the end and the interrupt is called "isr_bolt_interrupt()". The interrupt is enabled with the Macro "MAP_INTERRUPT_enable_INTERUPT(INT_PORT1)" and setting the "INT_PORT1" port to the name of the interrupt in the file "msp432_startup_ccs.c". The result is shown in Fig. 4.12. The reading of the AP is now even faster than the reading of the CP, because the AP has a faster clock and is a more powerful processor.

Every function executed by the AP is now interrupt driven. This allows to set the AP into a low-power mode after an interrupt ends and wake up when an interrupt is triggered. The low-power mode after an interrupt is enabled with the Macro "PM_ENABLE_SLEEP_ON_ISR_EXIT". The values in Fig. 4.12 are already measured when the AP goes into sleep mode after an interrupt. In summary, the interrupt implementation made the exchanging of the messages more predictable, faster and more energy efficient.

A more detailed evaluation of the upper-bound of the communication can be found in chapter 5.

4.4 Scheduler (MATLAB® on PC)

The scheduling functions are implemented in MATLAB® first, because it allows to directly change parameters without recompiling the code, a lot of debug output is possible without significantly effecting the critical timing, store an increased amount of logging data and directly modify the stored set of streams to test a specific scenario.

The Blink scheduler was already implemented in MATLAB® for simulation purpose only. The three different scheduling techniques "greedy", "naive lazy"

Field name	Stream 1	Stream 2
Periods (IPI)	8	10
Start-times	3	5
IDs	1	2
Deadlines	7	8
stream_node_ids	3000011	2000011
unused_slots	0	0

Table 4.3: The set of all active streams is stored in the "streams" struct with the shown fields and values for two example streams.

and "lazy" are available. The "lazy" technique achieved the best performance (see [4]) and was therefore used. The implemented scheduler is able to handle a time unit with an integer data type. The time unit can be defined to be any (scaled) time unit suitable for the application. The implemented Blink scheduer was only capable in setting up a set of streams and calculate the schedule for a given time. It was not possible to change the set of streams during the execution. This means additional functionalities are needed.

The main structure of the code is the "main" function, which decodes the input messages from the CP and calls the responsible function to handle the input. Afterwards, the "main" function initiatives the computation of the schedule and the sending of the schedule. The scheduler needs the information from the network and sends the schedule back to the network (CP) with the functions described in the section about the serial and packets functions. The other functions to handle the stream requests, update the bucket queue and detect non-active streams are explained in the next sections. Additionally, functions were implemented in MATLAB® to analyse the timing, which are explained in section 4.4.9.

4.4.1 Data structures

The two most important data structures in the MATLAB® code are the active streams and the bucket queue explained in the next paragraphs.

Active streams The struct "streams" includes all active stream with the information depicted in Tab. 4.3.

The example in Tab. 4.3 shows two streams. The first stream has ID 1, a period of 8 time units, a start time of 3 time units, a deadline of 7 time units, a stream-ID of 3, a node-ID of 11 and has used its last allocated data slot. The stream-ID and node-ID are combined into the stream_node_ids by adding up the stream-Id*1'000'000 and the node-ID.

IDs	Start-times	Deadlines	IPI	Stream-node-IDs
1	11	15	8	3000011
2	15	18	10	2000011

Table 4.4: The bucket queue implemented as a matrix with two example streams.

Bucket queue The bucket queue is used to find the stream with the earliest deadline. The matrix has the following structure depicted in Tab. 4.4. The IDs are only in relation when the stream request arrived and are continuous counted up. The ID can be used to relate a bucket queue entry with the set of active streams (in the struct ”streams”). The start-times and deadlines are absolute times. The start-times and deadlines are updated whenever a stream was executed and set to the next start-time and deadline by adding the IPI to the current times. The streams in the bucket queue from Tab. 4.4 are the same as shown in the active streams in Tab. 4.3 after each stream was executed once. The stream-node-ID makes it easier to get the stream-ID and node-ID without using the ID and searching the ID in ”streams”. The bucket queue is used to find the task with the next deadline and therefore, the queue is ordered by increasing deadline.

Possible implementation of bucket queue on an embedded processor The storage demand in MATLAB$^{®}$ can be neglected, because a current PC has enough resources to store the set of active streams and the bucket queue separate. Although, an embedded processor like the AP has limitations regarding memory. The bucket queue and the active streams stored in the struct ”streams” have many shared information. It would be sufficient to store all information in the bucket queue by adding the field ”unused_slots” to the bucket queue and remove the ”IDs” column in the bucket queue. Although, it would not make it possible to show all streams in a readable view as with the ”streams” struct during testing.

4.4.2 Main function

The main function calls the ”serial_read”, which returns the serial message as an array. The array is read with ”packet_read” (regarding the packet structure in Fig. 4.2 and 4.3). Afterwards, the header of the packet is checked and the type of the payload (see Tab. 4.1 for all payload types) is determined and the appropriate function is called to handle the input. The main function also saves logging information and makes debug print-outs to improve the debugging process.

4.4.3 Serial functions

The functions to handle serial communications in MATLAB® are in the directory "serial_communication" and are called "serial_write" and "serial_read". Both functions handle the communication protocol based on the "Framing Byte" and "Escape Byte" described in section 4.1.2. The function "serial_read" reads byte by byte of the message with the MATLAB® function "fread"[6] and parses the message. The function "serial_write" returns an array with the "Framing Bytes" and "Escape Bytes". The array is then written to the serial port with the MATLAB® function "fwrite"[7].

4.4.4 Packet functions

The functions to decode and encode the packets in MATLAB® are in the directory "serial_communication" and are called "packet_read" and "packet_write". The functions encode and decode the packets with the structure descriebed in section 4.1.1. The main function stores the last 10 sent and received packets, which enables backtracking when an error occurred.

4.4.5 Stream requests

The stream request is first checked if it is a new request or wants to update an existing stream with the same stream-node-ID. The stream-node-ID is the node-ID plus the stream-ID times 100′000. This combination is used to only search for one id instead of two and the overhead is seen as small, because MATLAB® stores every variable with type "double"[8]. The function is implemented in "newStream-Request.m". First, the stream-node-ID of the stream requested is searched in the set of active streams and the index of the stream is returned. If the index is not found, it is a new stream request and else it is a stream to update.

New stream request The received stream request is a new stream, because there is no existing stream with the same stream-node-ID. The requested stream is checked to have an IPI bigger than 0, because new streams with an IPI of 0 (or smaller) are ignored. Afterwards, a temporal set of all the existing streams and the new stream is made and the admission test checks if the temporal set is still schedulable. If the temporal set is schedulable, the existing set is replaced with the temporal set and a s-ack is added to the pending array. The s-ack will be sent with the next schedule. If the admission test fails, the new stream request is ignored.

[6] `ch.mathworks.com/help/matlab/ref/serial.fread.html`

[7] `ch.mathworks.com/help/matlab/ref/serial.fwrite.html`

[8] Floating point with a total length of 64 bits, 52 bits precision, 10 bits exponent and 1 bit for the sign.

Update existing stream request The received stream request is a request to update an existing stream. If the IPI of the stream is set to 0, the stream will be deleted from the set. Otherwise, a temporal set of streams with the updated stream is made and an admission test is executed. If the admission test is positive, the existing set of streams is replaced with the temporal set. If the admission test fails, the stream update is ignored. The stream will still exist with the old parameters.[9]

4.4.6 Bucket queue

At the beginning of the execution of the main function, the function "init_bucket_queue" queue is called, which sets up the bucket queue with all the streams stored in the struct "streams". Normally, the "streams" should be empty at the start of the scheduler, but this mechanism allows to add arbitrary streams before the start of the network and test specific scenarios.

The updating of the bucket queue is done by the function "manage_bucket_queue". Depending on the results from the admission test from the stream request, a stream will be added, deleted or updated in the bucket queue.

4.4.7 Detect non-active streams

The message with the payload (shown in Fig. 4.8) is used to get the streams, which have not used the allocated slots in the previous sent schedule. The "unused_slots" field in the "streams" struct is increased for these streams. If the value of the unused slots in a row is above a specified threshold (defined by "threshold_delete_unactive" and currently set to 3), the stream will be removed from the active streams and the bucket queue with the functions "stream_requests" and "manage_bucket_queue" by setting the IPI to 0. The "unused_slots" field in the "streams" struct from all streams, which were allocated in the last schedule and are not in the "unused_slots" message, are set to 0.

4.4.8 Host stream requests

The host stream requests are handled like the normal stream requests explained in section 4.4.5. The only difference is the different way of sending the host s-ack in the next schedule packet (shown in Fig. 4.6) and the schedule is not computed after the host stream request.

[9]This is a design choice made, because the stream can still send the current amount of data. Another option would be to delete a stream when the updating fails.

4.4.9 Timing analysis

The functions in the directory "experiment_evaluation" are used to plot the observed times from the experiment.

safe_data_timing_experiment.m This function is used to plot the execution time of one schedule/stream request in MATLAB® like Fig. 5.5.

analyze_timing.m The times for the serial communication, the BOLT delays, time from the start of a round until the schedule is received and the communication delays on the DPP are all plotted with this function. See Fig. 4.12 for an example. The output is often used for the function "plot_compare_slots_per_round" (see below) to compare the timing for different number of slots per round.

plot_compare_slots_per_round.m This function takes the values from "analyze_timing" as input and produces figures to compare the timing of the same delay with different number of slots per round. See Fig. 5.4 or 5.7 for examples.

4.5 Leveraging the outsourced scheduler on the DPP

Many real-time application require small deadlines and IPIs. Additionally, an IoT device should be as energy efficient as possible. The longer a device can stay in a low-power mode, the less energy it consumes. Smaller IPIs and longer sleep time for the CP can be achieved by reducing the round length (see T_l in Fig. 2.3), which is the time for the active execution of the network protocol (sending schedule, s-ack, data, stream requests, computing of schedule and managing streams functions, see Fig. 2.4). Reducing the round length has two positive effects:

- If the round period stays the same, the CP can sleep longer, because the processor is active only during the round length (see Fig. 2.4).
- The round times can be reduced until the round time is equal to the round length. A shorter round length therefore allows smaller deadlines and IPIs of the streams.

Outsourced scheduler is used to speed up computation and enable more streams The original LWB round structure was constructed for a sequential execution on a single processor. The round structure is shown in Fig. 4.13 (a). The outsourced scheduler (AP or PC) is used to speed up the computation and enable more streams, because more memory is available. The whole LWB round is executed sequentially.

Is it possible to run the scheduler in parallel to the communication protocol? The DPP allows to escape the sequential execution, because the network protocol and the scheduler run on different processors. It is evaluated if a parallel execution of network protocol and scheduler is possible. The parallel execution could reduce the round length. First, the data dependencies within a LWB round are considered to find possibilities to reorder the round structure and enable parallel execution:

- It is necessary to send the schedule k at the beginning of round k and the schedule $k + 1$ at the end of the round. The schedule at the beginning of the round is used to synchronise the network again after the potentially long waiting time since the end of the last round. The processors are in the sleep mode until the execution of the next round starts. The clock drifts of the nodes in the network are integrating and long round period cause large offset in the time for each node. A large time offset is not acceptable, because the LWB rounds are time-triggered and must be synchronised. The schedule at the end of the round is used to know when the communication of the next round starts. The processors can go into sleep mode until the start of the communication of the next round. Therefore, the schedules must be sent at the beginning and end of the rounds.
- The contention slot must be before the computation of the next schedule to ensure a fast adaptation to new streams. The calculation of the scheduler needs the information of the contention slot to compute the next schedule (see Fig. 4.13 (a) for the data dependency).
- The results of the calculation of the next schedule must be available for the the sending of the schedule, which is for the next round (see Fig. 4.13 (a) for the data dependency).
- The s-ack and communication slots are defined by the schedule from the previous round. The current implementation sets a stream active directly when the s-acks is received. Although, data is put into the output buffer if the stream is active in the "Manage streams" phase. Therefore, the s-ack slot must only be before the "Managing streams" phase. There is no data dependency between the computation of the schedule in round k and the s-ack and communication slots in round k.

The schedule can be computed in parallel to the communication slots and the s-ack slots, because there is no data dependency and the communication part is handled on the CP, while the scheduling task is outsourced (see Fig. 4.13 (b) for the adapted round structure). Therefore, the round length can be reduced, which enables a more energy efficient execution and smaller deadlines.

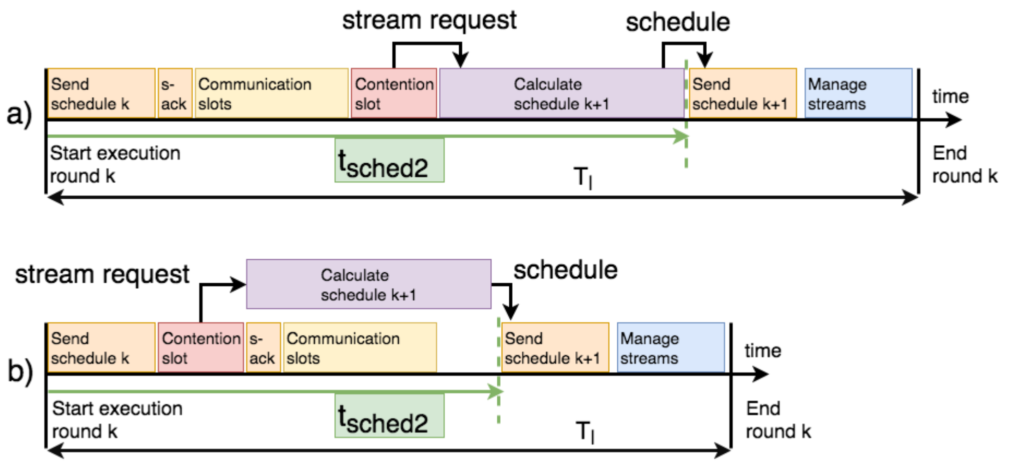


Figure 4.13: (a) The standard LWB round structure executed on a single processor is sequential. (b) The adapted LWB round structure for the DPP enables parallel computation of the schedule during the communication of the network. The parallel execution reduces the round length (T_l).

4.6 Optimise the round duration

The time t_{sched2} was introduced in Fig 4.13. It represents the time at which the schedule from the next round is sent, with respect to the start time of the current round. t_{sched2} is a fixed parameter for all rounds and independent of the number of allocated data slots per round. It is important that the communication slots have finished and the outsourced schedule arrived back from the scheduler. Of course, the goal is to have a round length as small as possible and the parameter t_{sched2} should be a close upper-bound of the communication slot duration and the time needed to compute the outsourced schedule. This section proposes a model of the different parameters of a LWB round, which is used for the implementation evaluation in chapter 5.

4.6.1 Outsourcing the schedule

The second schedule can not be sent before the schedule was arrived from the outsourced scheduler (see Fig. 4.14). This induces a lower-bound on t_{sched2} (see formula 4.2).

$$t_{sched2} \geq (T_{sched} + T_{gap}) + (T_{cont} + T_{gap}) + t_{sched\_outsource\_compute} \tag{4.2}$$

The delays for the outsourcing of the schedule depends on the maximum data slots per round, if a stream request was sent and on the serial communication. Therefore, the timing of the different phases is looked at in details in chapter 5.

Parameter	Value
t_{sched}	$18ms$
t_{data}	$12ms$
t_{cont}	$12ms$
t_{round}	$394ms$
t_{gap}	$4ms$
data packet size	$32B$
slots per round	20

Table 4.5: The different standard parameters for the LWB round structures. The values are from [3] except the duration of the contention slot was increased to the same value as the contention slot (see Fig. 4.14 a graphical view of the parameter names).

4.6.2 LWB communication phase

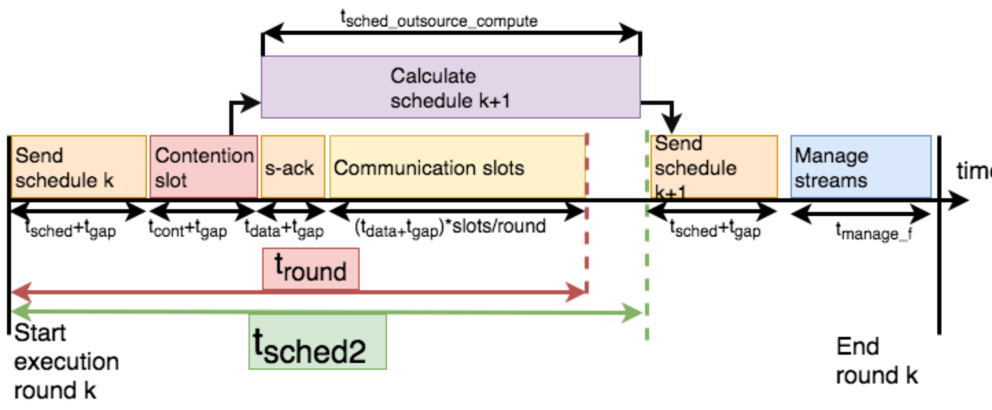


Figure 4.14: The adapted LWB round structure with the parameter names of the different phases and slots.

The length of communication operations, like sending the schedule, the contention slot, the data slots and the gap time between two slots, is fixed and reported in Tab. 4.5. The used parameters are also shown graphically in Fig. 4.14[10]. All the values are the standard values from LWB except the duration of the contention slot was increased to the duration of a data slot[11]. The time t_{gap} is the gap between two slots. The value t_{sched2} is analysed in chapter 5.

The value t_{round} can be calculated with Formula 4.3. The duration of the

[10]Mind: The values are printed to the serial debug output at start up and are rounded.

[11]The contention slot was moved before the s-ack and data slots. This makes the calculation of the starting times for all following slots complex. Setting the contention slot to the same length as the data slots allows to use the standard LWB Macro to get the start times.

s-ack slot is the same as for a normal data slot (t_{data}), which is also the reason to multiple the duration of data slot (t_{data}) with the amount of slots per round (Max_Data_Slots) plus 1.

$$t_{sched2} \geq t_{round} = (t_{sched} + t_{gap}) + (t_{cont} + t_{gap}) + (Max\_Data\_Slots + 1) * (t_{data} + t_{gap}) \quad (4.3)$$

The time until the second schedule is sent from the beginning of the round (t_{sched2}) must be greater or equal to t_{round}, because the next schedule can not be sent before the communication slots are over.

4.6.3 Summary for constrains on t_{sched2}

The two sections before explained the two constrains on the t_{sched2} parameter. Both constrains are combined in formula 4.4.

$$t_{sche2} \geq (T_{sched} + T_{gap}) + (T_{cont} + T_{gap}) + max((Max\_Data\_Slots + 1) * (T_{data} + T_{gap}), \quad t_{sched\_outsource\_compute}) \quad (4.4)$$

There is a fixed part of the minimal value of t_{sched2}, which is independent of the number of data slots per round: $(T_{sched} + T_{gap}) + (T_{cont} + T_{gap})$. The other part is the maximum of the duration for the outsourcing of the schedule and the duration of the communication slots. A model for the variable part is evaluated:

- The communication time is the duration of a s-ack slot and the number of slots per round multiplied with the duration of a data slot. t_{sched2} is a fixed parameter for every round. Therefore, the maximum duration of the communication slots must be considered for every round. The maximal duration for the communication slots is: $(n + 1) * t_{data}$, where n is the maximum number of slots per round. The duration is linearly increasing with the number of slots per round and is shown in Fig. 4.15 in blue.
- The duration for outsourcing the scheduling tasks is depending on the maximum number of slots per round as well, because the admission test, the scheduling of the streams and the message size depend on the number of slots per round. It is likely to have a bigger offset than the communication slots duration, because the request of the schedule and sending the schedule has a relatively high latency (see chapter 5). If the increase of the duration with the number of slots per round is not as dramatic as for the communication slots duration, the duration depending on the slots per round could look like the green line in Fig. 4.15.

Assuming the scheduling duration increases slower with the number of slots per round than the communication duration, the two delays evolve as represented in Fig. 4.15. This implies that beyond a given number of slots per round n', the outsourcing of the schedule virtually costs 'no time' (see Fig. 4.15 for point n').

Numerical values for those delays and number of slots per round are investigated in chapter 5.

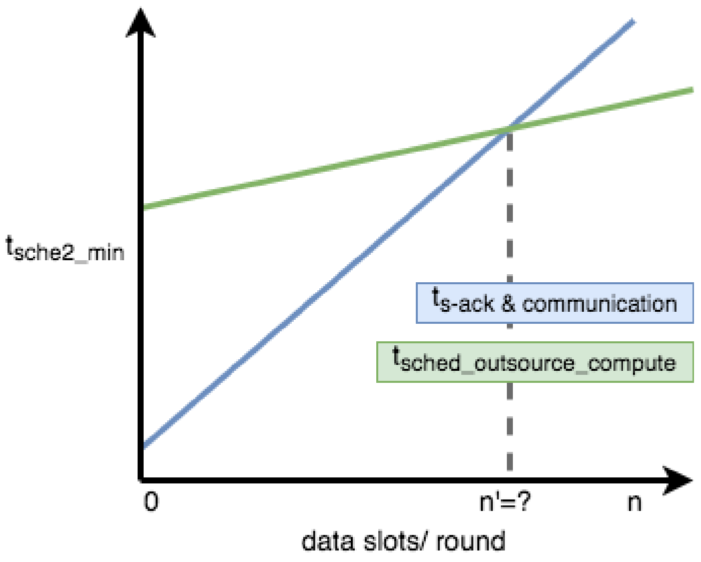


Figure 4.15: The model for the minimum of t_{sched2}. t_{sched2_min} is the maximum of the delay for the communication slots (blue) and the outsourcing of the schedule (green).

The delays are analysed for different number of slots per round in chapter 5 and tried to verify if there is a number of slots per round above the outsourcing of the scheduler is neglectable (see Fig. 4.15 point n').

Chapter 5

Tests and Evaluation

This chapter presents the system design and implementation evaluation. It has several objectives:

- Measuring the various system delays having an impact on the implementation (see Fig. 4.14 for the parameters for the delays and times like t_{sched2} and section 4.6 for the explanation of the delays).
- Investigating the dependency between the schedule computation time and the numbers of slots per round.
- Validating the correct implementation of the outsourced computation on a real-life network environment like the wireless network testbed Flocklab [10].

5.1 Evaluate parameters for different number of slots per round

Minimising the round length, which is the time from the start of the execution of round k until the end of the round k (see Fig. 5.1), is the goal of this section. Therefore, the duration of all LWB round phases and the time for outsourcing the scheduler must be upper-bounded. Additionally, the parameter t_{sched2} must fulfil the formula 4.4. The overall structure with the evaluated delays is shown in Fig. 5.1. It is particular difficult to upper-bound the delay for outsourcing the schedule computation to the PC over serial (tagged as 2a in Fig. 5.1). Finding an analytical value for t_{sched2} is very difficult, because the computation of the schedule and the serial communication is handled by a PC with a non-real-time OS. Therefore, the parameters are evaluated with multiple observations over $30min$. This section describe how the experiments are realized to estimate the upper-bound for t_{sched2}.

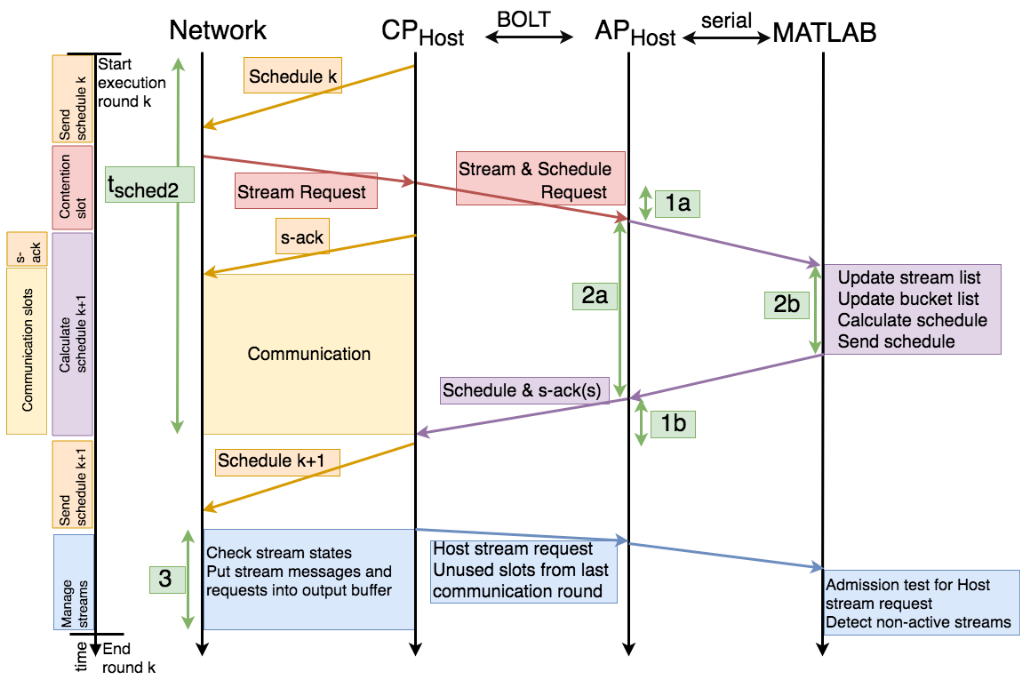


Figure 5.1: The overall setup when the scheduler is outsourced to MATLAB®. The relation from communication slot together with s-ack (yellow and orange) to the calculate schedule block (violet) is not fixed and depends on the number of data slots per round.

The times were measured by toggling pins on the CP and AP. Each processor has two pins, which can be toggled in the code by calling the Macro "PIN_XOR(pin_name)". The pins are analysed with a Logic Analyser, which exported the time of the pin changes. The function "experiment_evaluation/analyze_timing.m" reads the timing file and generates the following plots in Fig. 5.2 and 5.3 and produces the input for the function "plot_compare_slots_per_round", which produces the Fig. 5.4, 5.5,5.7 and 5.9.

The scheduler is executed with MATLAB® on a PC [1]. MATLAB® was the only application actively running. Other applications like the anti-virus software and some OS managing functions were still running in the background. The energy saving functions were disabled.

The goal is to find the upper-bounds and therefore a setup for the most extreme scenario is used. A new stream request arrives in every round and no streams are deleted. The message from the CP to AP are therefore always the maximum size of a stream request. After an initial phase, the schedule message from the AP is often a schedule with all slots assigned, because streams are added until the admission test fails and the network utilisation is very high. A set of 128 pseudo random streams are emulated with an IPI[2] between 4 and 9 s and a relative deadline of $(IPI - 2)$ s with the following pseudo-c code:

```
static uint_16 random_stream_request_counter=0;
void emulate_stream_request(){

    lwb_stream_req_t random_stream_request;

    random_stream_request.id= 147;
    random_stream_request.stream_id= random_stream_request_counter%128;
    random_stream_request.ipi= (random_stream_request_counter)%5+4;
    random_stream_request.start_offset= 2;
    random_stream_request.deadline= (random_stream_request_counter)%5+2;

    //send emulated stream request to scheduler
    lwb_sched_proc_srq(&random_stream_request);

    random_stream_request_counter= (random_stream_request_counter+1)%1024;
    DEBUG_PRINT_INFO("emulated stream request sent");

}
```

[1]A MacBook pro 15' with an Intel Core i7 processor (quad-core with 2.3 GHz) and 16 GB RAM was used.

[2]REMINDER: The current implementation of the Blink scheduler can schedule integer time units. The used time unit is chosen to be one second. Other values like 500ms are also possible.

Maximum slots per round	After initialisation	After 30 min.
5	25	26
10	50	51
15	75	78
20	100	103

Table 5.1: The number of active streams is depending on the maximum slots per round. The scheduler is initialised with an empty set of streams and a stream requests is arriving in every round. The number of active streams is quickly growing to the shown value "After initialisation" and afterwards only a few more streams got accepted until the final value "After 30 *min*". Therefore, the utilisation was constantly high and the computation of the schedule was made complex.

After 128 streams with a different stream-ID have been requested, every stream gets again requested with a potentially different IPI. Therefore, the scheduler has to make a full admission test in every round and test if the stream is updated or added. The tests were done with 5, 10, 15 and 20 slots per round. The tests are all done with the same stream set (see pseudo-c code). Nearly all the streams have been accepted in the initialisation phase. Afterwards, the number of streams is more or less constant with a small increase. The number of active streams after the initialisation (phase as long as nearly all streams are accepted) and after 30 *min* is shown in Tab. 5.1. The 128 different streams are a big enough stream set, because even for 20 slots per round at least 25 streams have not been accepted. The utilisation was oscillating around 90%[3] after the initialisation.

5.1.1 Communication between CP and AP

The BOLT communication delays were already considered for the implementation of the functions. Fig. 4.12 shows a very constant duration for the reading of BOLT messages. Additional tests are evaluated in this section to find an upper-bound for the BOLT read & write delays together.

CP writes stream request to BOLT and AP reads The delay for the CP writing a schedule/stream request to the AP via BOLT and AP reading it is shown in the overall setup Fig. 5.1 (green 1a). The duration of exchanging a stream request from CP to AP via BOLT is shown in Fig. 5.2 (blue).

[3] $Utilisation = \frac{1}{max\_num\_slots\_per\_round} * \sum_{i=1}^{n} \frac{1}{IPI_i}$, the IPI of stream i is in the same time unit as the Blink scheduler is running (in seconds).

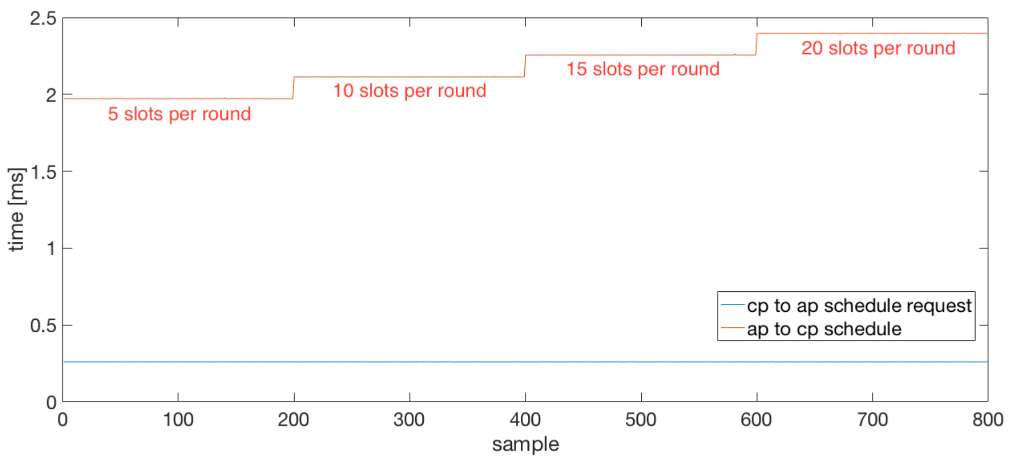


Figure 5.2: The BOLT communication delay is measured for different number of slots per round (see red labels). (Blue): Writing a stream request by CP and AP reading it (see Fig. 5.1 (1a)). (Orange): AP writing a schedule and CP reading it (see Fig. 5.1 (1b)). The delay for the schedule depends on the schedule size, which depends on the number of slots per round.

The writing and reading of the schedule request from CP to AP via BOLT has a very low variance and is independent of the number of slots per round, because the stream request message size is constant and independent of the number of slots per round. The delay is less than 0.265 *ms* for all observed rounds.

AP writes the schedule to BOLT and CP reads The delay for BOLT communication from AP to CP is shown in the overall setup Fig. 5.1 (green 1b). The duration of the schedule exchange over BOLT is shown in Fig. 5.2 (orange) for different number of slots per round.

The time is clearly depending on the number of slots per round, because the schedule message size is increasing with the number of slots per round. The first 200 samples are with 5 slots per round, the second 200 samples for 10 slots per round, the third 200 samples for 15 slots per round and the last 200 samples are for 20 slots per round. The delay is low (the maximum is below 2.5 *ms*) and has low variance. These are the expected properties of BOLT communication.

5.1.2 Delay for the communication between AP and scheduler including computing the schedule

The communication delay to the scheduler is also evaluated to find the upper-bound of the delays. The round-trip time (RTT) of the serial communication with the computation of the schedule on the PC was measured together (marked green in Fig. 5.1 (2a)). The AP toggles a pin when a packet per serial is written and received.

Variance of the serial communication The first measurement investigated the predictability of serial communication delays. The results for the RTT are shown in Fig. 5.3. To remove as much noise as possible an empty schedule was sent, without any schedule computation. The pure scheduler execution on MATLAB® was measured with the functions "tic" and "toc"[4] from MATLAB® and was below $2ms$. The experiment showed that the serial communication variance is very high (between 111 ms and 154 ms).

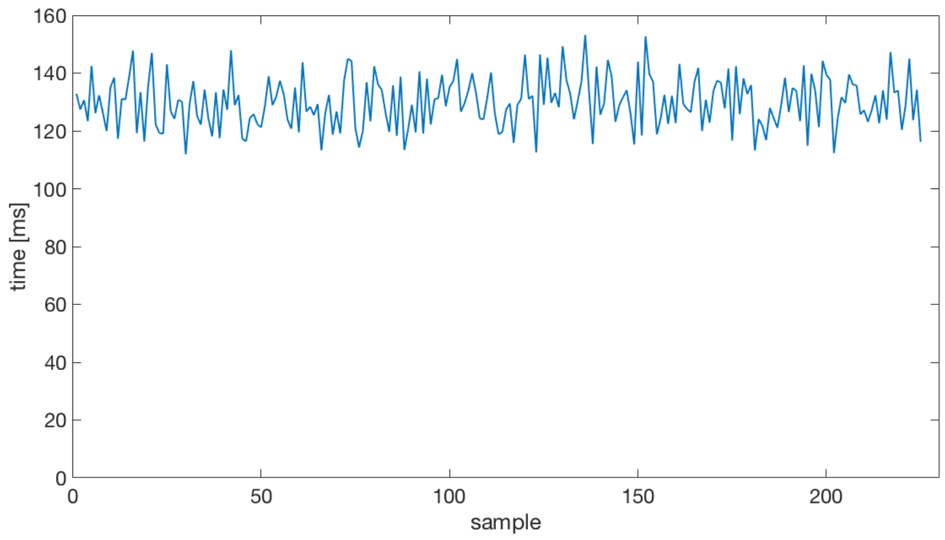


Figure 5.3: RTT of the serial communication with sending an empty schedule. The delay has a high variance (between $111.897ms$ and $153.1714ms$).

The reasons can be the complex Operation System (OS) running on the PC, which assigns the CPU to the processes dynamically, the serial communication is handled by the OS as well and many processes are running in the background. The variance is very high and therefore, a more detailed evaluation with at least 1800 rounds was done (i.e., a 30 min-long experiment). The more detailed evaluation of the upper-bound of the delay of the scheduler and the communication is found in the next paragraph.

Upper-bound for the serial communication and schedule tasks The delays from Fig. 5.3 have a high variance. It is difficult to find a upper-bound

[4] `ch.mathworks.com/help/matlab/ref/tic.html` and `ch.mathworks.com/help/matlab/ref/toc.html`

for delay with such a high variance. It was tried to measure the extreme situation with a stream request in every round and many schedule with the maximum size (when the maximum number of slots per round are assigned). This setup ensures the maximum communication delay, because the messages are as large as possible. Additionally, the computation of the admission test for a new stream and the computation of the schedule is made complex by requesting a new stream in every round, leading to an increasing amount of active streams in every round. The result for different number of slots per round is shown in Fig. 5.4.

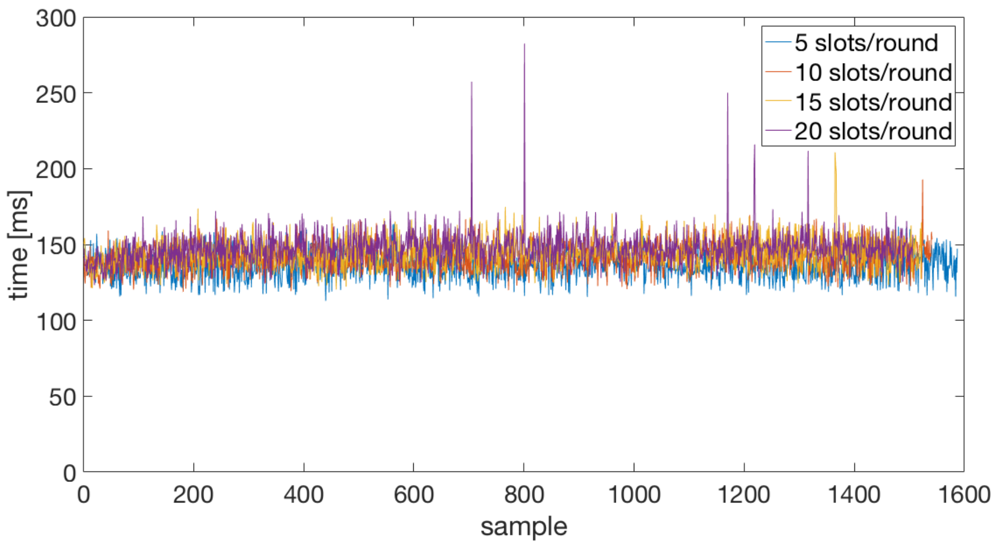


Figure 5.4: The time for the serial communication and computation of the schedule is shown for different number of slots per round. Computing the schedule includes an admission test for a stream request. The delay depends on the number of slots per round. The delays is not well predictable, because the delay has a high variance and peaks.

The variance of the RTT is large (values between 107 ms and 290 ms) and peaks happen for 10, 15 and 20 slots per round. The value is weakly depending on the number of slots per round. An evaluation of the delay, the variance and the peaks is difficult, and therefore the time to compute the schedule is considered in the next paragraph.

Observe the computation time of the schedule The scheduler execution time was measured in MATLAB® to find the source of the high variance and peaks in Fig. 5.4. It is important to know the reason for the peaks and the variance and if the implementation of the scheduler on an embedded system like the AP would make it more predictable. The predictability is desirable, because it allows to make a tight upper-bound of the delays, which are used to specify t_{sched2} (see section 4.6.3) and the round length.

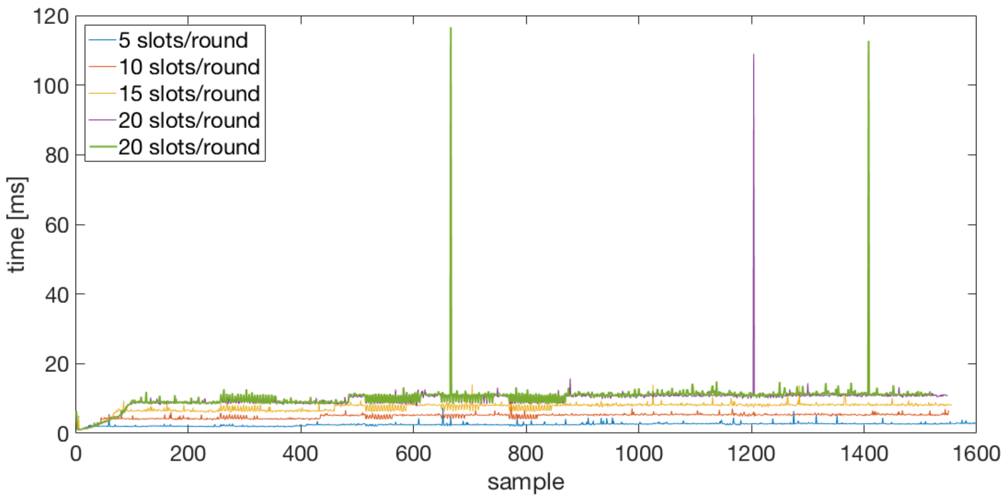


Figure 5.5: Time for the computation of the schedule with an admission test for a stream request with different number of slots per round. The (violet) and (green) observations are two identical test with the same parameters and stream requests. The peaks are at different rounds and therefore, the peaks are caused by the OS and not by a complex stream request.

The measured schedule computation delay can be seen in the overall time diagram in Fig. 5.1 (2b in green). The scheduler execution time is shown in Fig. 5.5 for different number of slots per round. The first four observations (blue, yellow, orange and violet lines in Fig. 5.5) are from the same test as the data for the Fig. 5.4. Multiple point can be seen by considering Fig. 5.4 and Fig. 5.5:

- **What is causing the peaks:** Not all peaks in Fig. 5.4 are caused by the peaks of the computation of the scheduler (see Fig. 5.5). The serial communication and the computation of the schedule cause peaks. The peaks caused by the serial computation are irrelevant when the scheduler is implemented on the AP. The outliers of the schedule computations are problematic for an implementation on the AP (see below).
- **What is causing the variance and most of the delay:** If the peaks are neglected, the computation duration is shorter than 15 ms and the delay is very depending on the number of slots per round. This means the serial communication adds a significant offset, because the minimum value for the serial RTT with the computation is over 107 ms and the maximum value for the computation is below 15 ms (when the outliers are ignored). The variance is also caused by the serial communication, because the time

to compute the schedule is within a small range (between 5 *ms* and 15 *ms* for 20 slots per round when the outliers are not considered). The variance and the delay caused by the serial communication are avoided when the scheduler is implemented on the AP.

- **What is causing the outliers in the schedule computation:** The outliers of the tests are only observed for 20 slots per round. If the outliers are caused by a very complex scheduling decision, the outliers for the computation on an embedded system would be similar as on the PC. The exact same experiment (same parameters and order of stream requests) was done twice (see green and violet lines in Fig. 5.5). The peaks happen at different rounds and therefore the peaks are caused by the non-real-time OS. The implementation on the AP must be more predictable than the implementation on the PC.

5.1.3 Handling of host stream requests and detection of non-active streams

Every node manages its streams between receiving the next schedule and the start of the communication of the next round (see Fig. 5.1 number 3 in green). The host also sends the unused slots in the "Manage streams" phase and can send a host stream request. The maximum execution delay of the managing functions executed after sending the schedule also increases the minimum possible round time, because the CP is busy and cannot communicate.

Source nodes Every source node checks the state of the streams and adds data to the output buffer if the stream is active. Additionally, some debug output is made in this phase. The host does this as well and additionally has other tasks like the host stream request and sending the unused slots. Therefore, it is sufficient to test the timing for the host.

CP Host The CP from the host makes debug outputs, has to send the unused slots, potentially send a host stream request, test the states of all its streams and add data to the output buffer. The time was measured between the end of sending the new schedule and when the host went into sleep mode after the "Manage streams" phase. The result is shown in Fig. 5.6. The test was done with a unused slot message of 20 slots.

The slight increase of the time is the time needed to read a message from the input buffer, because another host started to send data at this moment. Generally, the delay is very constant a upper-bound of 50 *ms* is estimated.

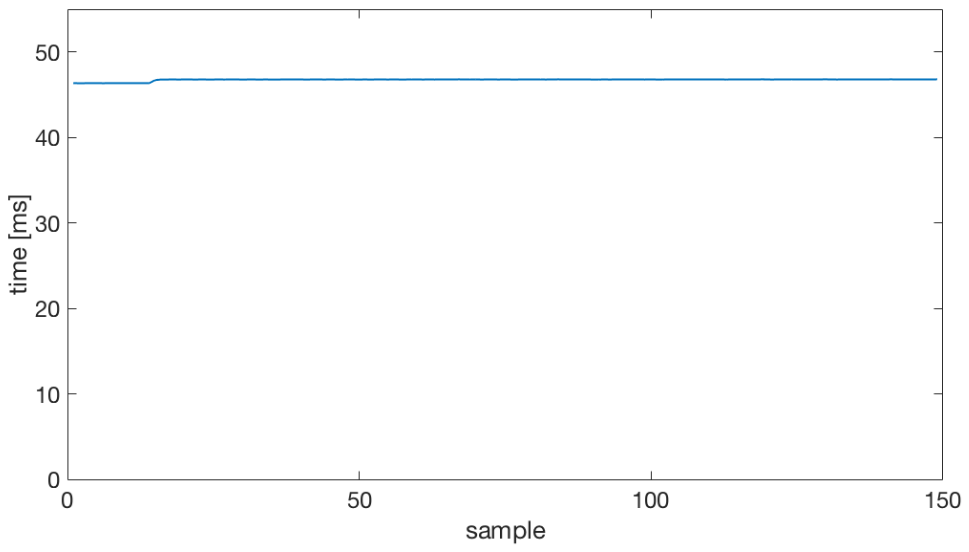


Figure 5.6: The time needed by the Host CP to make debug outputs, send the unused slots (here 20 slots), a host stream request, test the state of one stream and put a message for this stream into the output buffer. The time is very constant and the slight increase after 20 rounds caused by the host is reading a data package from the input buffer. The data packet was not sent before by another node.

Scheduler The scheduler has to search for non-active streams with the unused slots message from the host CP and potentially has to check a host stream request. The scheduler must not be finished at the end of the LWB round, because the scheduler is not getting new commands before the contention slot of the next round. It would be ideal when the checking for unused slots and the host stream request is handled before the stream request, which arrives in the contention slot in the next round. Otherwise, the computation of the next schedule can be delayed. The detection of the non-active streams is very fast, because it is only handling counters and deleting elements in the bucket queue. The admissions tests for host stream request is more complicated. Although, the admission test is less complex than the admission test with computing the schedule, which is shown in in Fig. 5.5. The available time is the time for managing the streams, sending a schedule and the duration of the contention slot including the gap between is at least $84ms$ ($50ms + 18ms + 4ms + 12ms = 84ms$). The duration for the schedule computation had a few peaks for 15 and 20 slots per round (see Fig. 5.5). The maximum observed delays are 120 ms. The computation of the next schedule is prolonged in the next round if such a peak happens for a host request. This additional delay will be considered for the determination of the parameter t_{sched2} at the end of section 5.1.4.

5.1.4 Minimizing t_{sched2}

The minimal value for t_{sched2} is the maximum duration of the data communication slots and the duration for outsourcing the schedule tasks. All the delays, which sum up to t_{sched2}, are in Formula 4.4. Especially, the serial communication between AP and scheduler and the computation of the schedule has a high variance (see Fig. 5.1 (green 2a)). The time between the start of the execution of the round until the outsourced schedule is received and ready to be sent by CP_{Host}, depending on the number of slots per rounds, was measured in an experiment (see t_{sched2} in Fig. 5.1). The results are shown in Fig. 5.7 for different number of slots per round.

We observe two different patterns depending on the number of slots per round. The duration for 5 slots per round (blue) is very jittery. The other delays (red for 10 slots per round, yellow for 15 slots per round and violet for 20 slots per round) are very regular. This can be explained easily: with only 5 slots per round, the time to compute the schedule dominates, and we observe a jitter delay due to the serial communication between AP and the PC. For 10 slots and more, the duration of network communication becomes longer than the scheduling. The values change between two points, which are the times for the maximum number of slots per round in the communication phase and when one slot is free. The tested scenario was with pseudo random streams, which leaded to rounds with one free slot. The durations for a maximum number of 20 slots

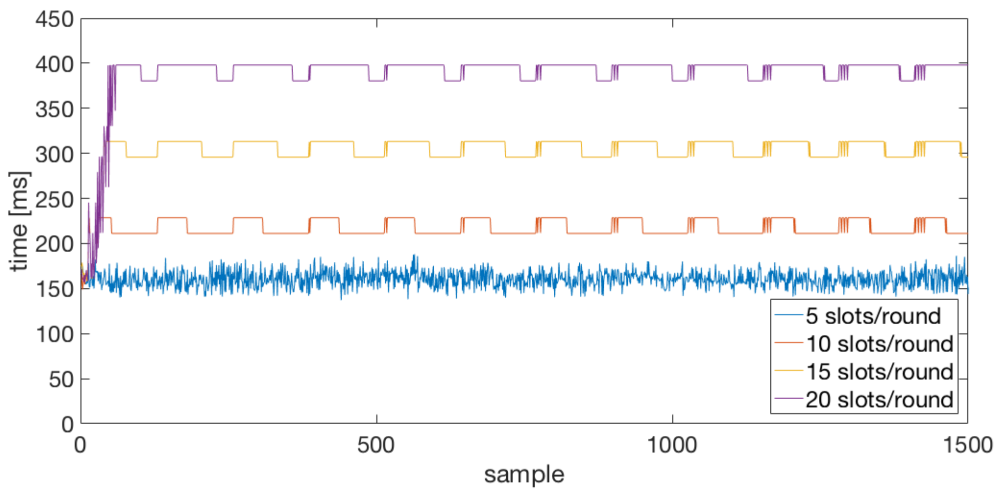


Figure 5.7: The time between the start of the round and when the schedule is stored at the CP. This time is the lower bound for the parameter t_{sched2}.

per round is close to $400ms$, which is the time for 20 allocated data slots, or close to $390ms$, which is the time for 19 slots allocated. In conclusion, the delay for outsourcing the scheduler is neglectable for more than 10 slots per round.

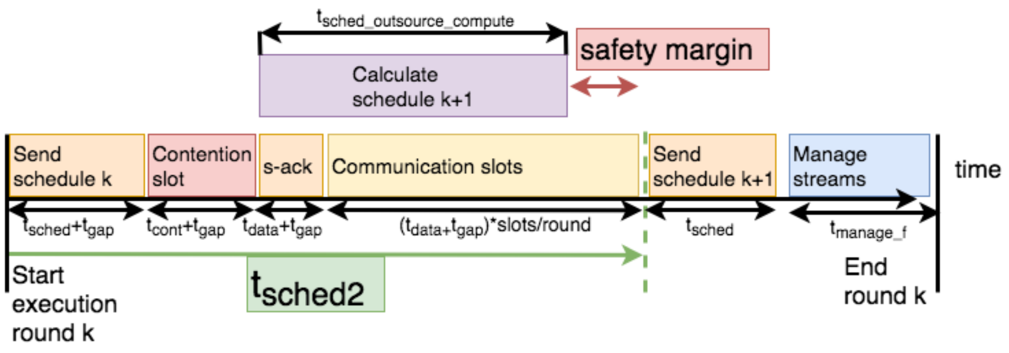


Figure 5.8: Time diagram when the communication slots are longer than the delay for the outsourcing of the scheduler.

The parameter t_{sched2} must fulfil the formula 4.4. The following points are important for evaluating t_{sched2}:

- t_{sched2} is depending on the number of slots per round.
- The value for t_{sched2} must be larger than all the samples in Fig. 5.7, which include the network communication and the delay for outsourcing the schedule.

- The duration for the network communication is predictable, because LWB is time triggered.
- The duration for the outsourcing of the schedule is difficult to predict, because a non-real-time OS is handling the serial communication and the computation of the schedule (see Fig. 5.1 (2a in green) for the duration in the overview and Fig. 5.4 for the values of the delay). It can not be guaranteed to have observed the highest peak values by observing the system for a bounded time. Therefore, a safety margin must be included for the outsourcing of the schedule.
- If the network communication is significantly longer than the outsourcing of the scheduler, a safety margin is already included (see illustration of the "included" safety margin in Fig. 5.8 in red).

The "included" safety margin in Fig. 5.8 can be estimated by observing the time, which the schedule message is in the BOLT buffer after the AP wrote the schedule to BOLT (see Fig. 5.1 (1b in green)). The AP writes the schedule to BOLT as soon as the schedule arrives via serial. The CP reads the schedule as soon as the BOLT message is available and the network communication is over. If CP does not have to wait until the end of the network communication, the time for AP writing to BOLT and CP reading the message was observed in Fig. 5.2 (in red). The maximum value is below 2.5 *ms*. Therefore, the time, which the schedule message is in the BOLT buffer, can be used to estimate the included safety margin and is shown in Fig. 5.9.

The estimated values together with the included safety margin of how long the message is in the BOLT queue are shown in Tab. 5.2. The additional safety margin for 5 slots per round was chosen to be 42 *ms*, because there is no "included" safety margin as the outsourcing delay is longer than the network communication. The setup with 10 slots per round does already include a safety margin of at least 8 *ms* and another 12 *ms* are added. The safety margin is regarded enough, because the outsourcing of the schedule had only one outlier (see Fig. 5.9), which was in a round with only 9 slots per round and the maximum slots per round of 10 slots would add an additional 16 *ms* to the network communication delay (see Tab. 4.5). The setups with 15 or 20 slots per round have already included more than 50 *ms* as safety margin and the values are only rounded up. This safety margin also covers a high peak in the admission test of the host stream request in the "Managing streams" phase, which can prolong the execution of the scheduler (see last paragraph in section 5.1.3).

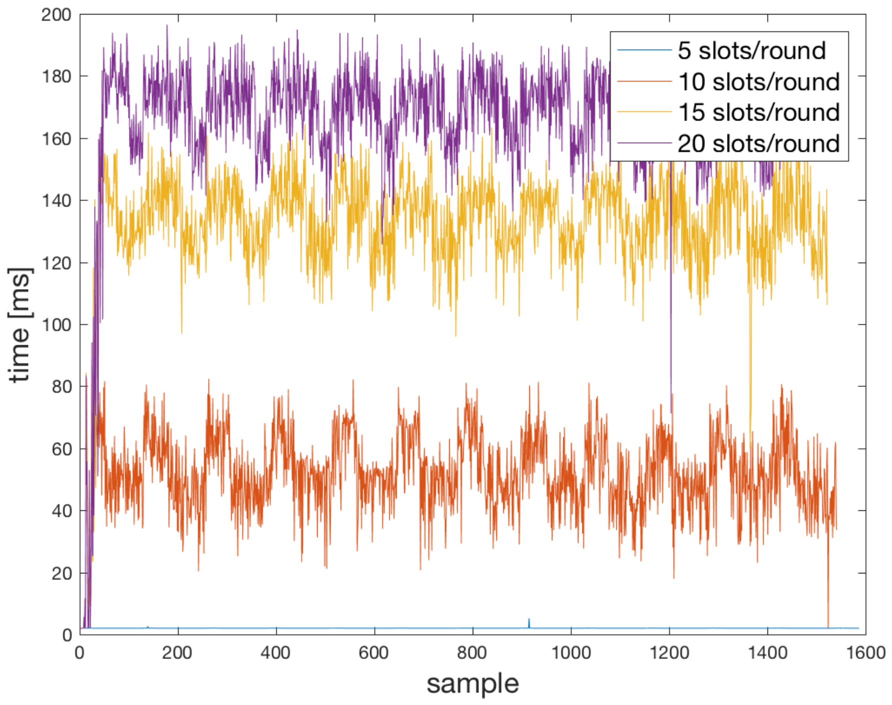


Figure 5.9: The duration for AP writing to BOLT and CP reading the schedule from BOLT during the execution. CP can only read BOLT after the communication slots are over. Therefore, the values can be used to estimate how much longer the communication slots are compared to the delay for outsourcing the schedule, which is the "included" safety margin for the unpredictable outsourcing of the schedule.

Slots/ round	Largest value t_{sched2} from tests	Minimal margin BOLT read	t_{sched2} chosen
5	$188ms$	$< 0ms$	$230ms$
10	$228.6ms$	$8ms$	$240ms$
15	$313.3ms$	$> 50ms$	$315ms$
20	$398ms$	$> 80ms$	$400ms$

Table 5.2: Values for t_{sched2} for different slots per round is estimated by including a safety margin for the unpredictable delay for the outsourced scheduler. The "included" safety margin (minimal value from Fig. 5.9) is considered for the estimation.

slots per round	t_{sched2}	Minimal round period
5	$230ms$	$302ms$
10	$240ms$	$312ms$
15	$315ms$	$387ms$
20	$400ms$	$472ms$

Table 5.3: The minimal round period possible depending on the number of slots per round.

5.1.5 Minimal round period possible

The minimal possible round period is the round length, because this is the scenario when the next round directly starts after the end of the previous round. The round length is the duration of the execution of all LWB phases (see T_l in Fig. 4.13 (b)). The round length must be larger than t_{sched2}, the time to sending the next schedule and finish the manage functions at the end of the round like sending the unused slots, host stream request, updating the stream states and put messages into the output buffers (see Fig. 5.8). t_{sched2} is evaluated in section 5.1.4 and in Tab. 5.2 are reasonable values depending on the number of slots per round. Sending a schedule (t_{sched}) takes $22ms$ (see Tab. 4.5 for the parameters used). The time t_{manage_f} from Fig. 5.8 was evaluated in section 5.1.3 and is below $50ms$. Therefore, the minimal round period is $t_{sched2} + 22ms + 50ms$ and is shown in Tab. 5.3 for different number of slots per round.

5.2 Flocklab

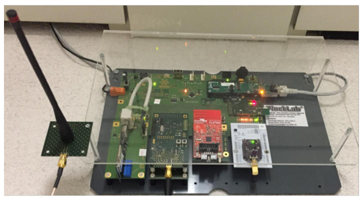
The previous validations of the real-time functionalities has been done with only the host and all the stream requests were emulated. Additionally, tests have been made with a source node and the host sending multiple streams to each other. Every aspect like timing and small subfunctions have been tested with this setup. The goal was to verify small pieces of the implementation. The working subfunctions are a sign for the overall functionality, but a more realistic scenario with multiple nodes and streams is necessary to really conclude the overall functionality. Therefore, the implementation was tested on FlockLab [10].

FlockLab is a testbed with more than 25 DPPs. The testbed is configured with an xml-file[5], which includes the parameters for the observers (e.g. observe serial output and toggling of pins), a list for the used nodes and the binaries of the processors used. An xml-file can be found in the repository

[5]`www.flocklab.ethz.ch/wiki/wiki/Public/Man/XmlConfig`

(LWB_CP_CC430/tools/flocklab). The code must be compiled without specifying the "NODE_ID", because FlockLab inserts the "node_id" from the nodes used in the network.

The test application running is in the file "lwb-test.c". Every node tries to add a stream with the IPI of "node_id" modulo 10 plus 3 ($node\_id\%10 + 3$). This allows to quickly observe if the IPI of the stream was correctly scheduled, the schedule received and the packet sent at the right time. The receiver of the packets is always the host. The debug output from the host can be read by the PC on the desk. See Fig. 5.10 for the used setup.



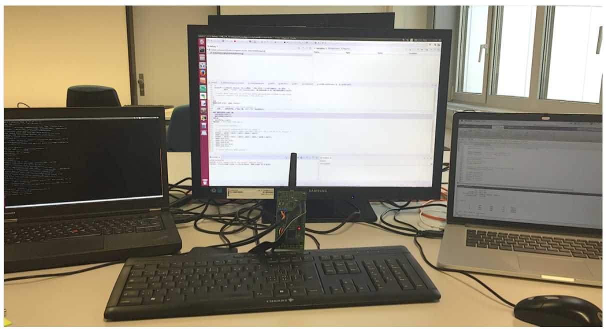


Figure 5.10: (Left): One of the used FlockLab nodes with the DPP during a network test. (Right): The setup on the working desk with a notebook to flash the program and observe the serial debug output from the host CP, the DPP host node, which is connected via serial to the PC, and the PC is running the scheduler.

The FlockLab web-interface has a preview of the results, which shows the value of the LED on the DPP. The LED is on when the RF module is on. It is useful to get a first impression, if the node is synchronised and participated in the flooding. The LED diagram for multiple FlockLab nodes over time is shown in Fig. 5.11. All the nodes are synchronised, participate in the flooding and receive the second scheduler after t_{sched2} (0.5 s).

Every nodes CP is sending debug output via the serial port. The serial debug outputs of all FlockLab nodes are captured by the FlockLab testbed. The debug outputs of the nodes and the host are analysed and it is checked if the stream request were sent and received, the s-ack were sent and received and the data is sent and received with the correct IPI. The serial debug output from the node with node-ID 10 is shown in Fig. 5.12. The node is sending a stream request, gets the s-ack and later sends the packets with the correct IPI (3 s, because the IPI is node-ID % 10 + 3).

Fig. 5.13 is showing a snippet of the serial debug output from the host. The

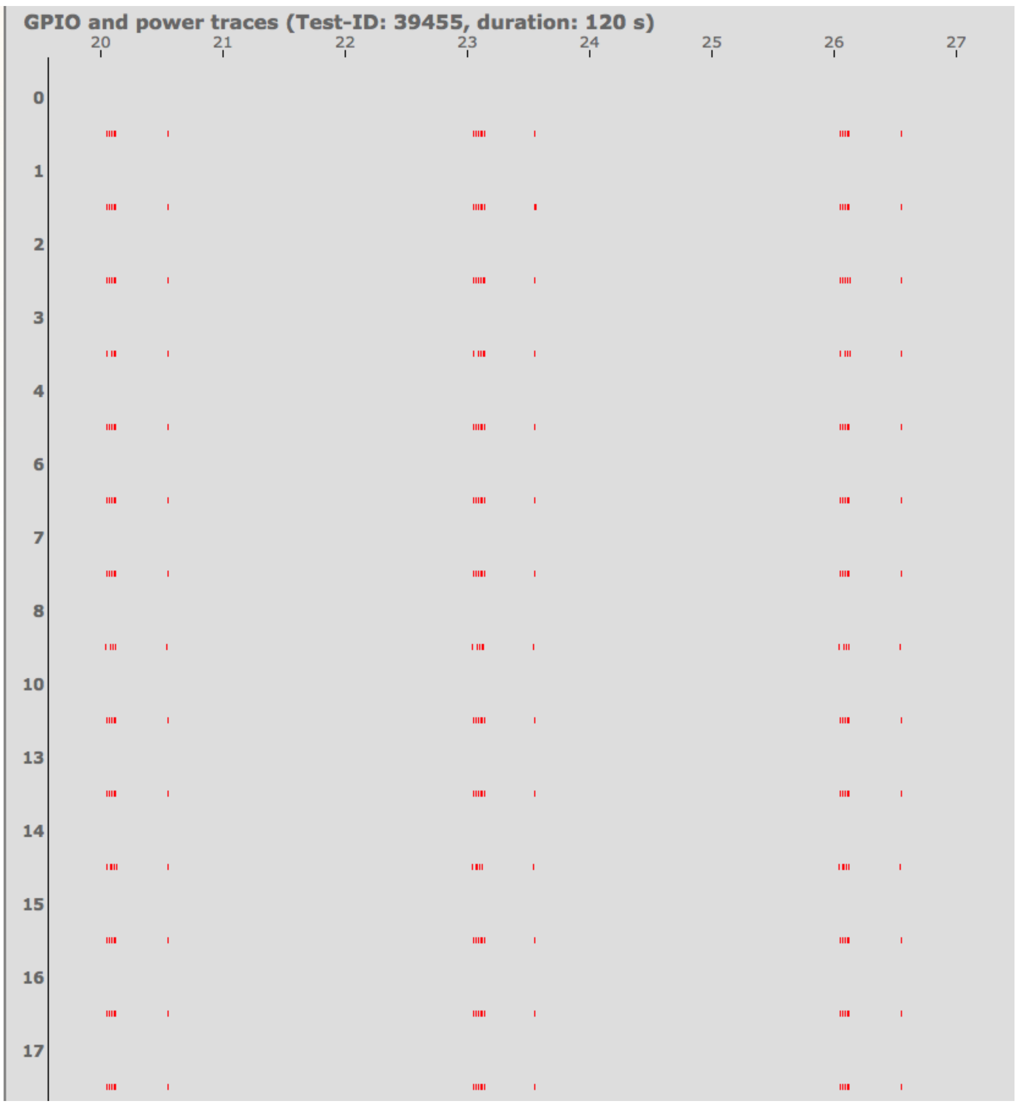
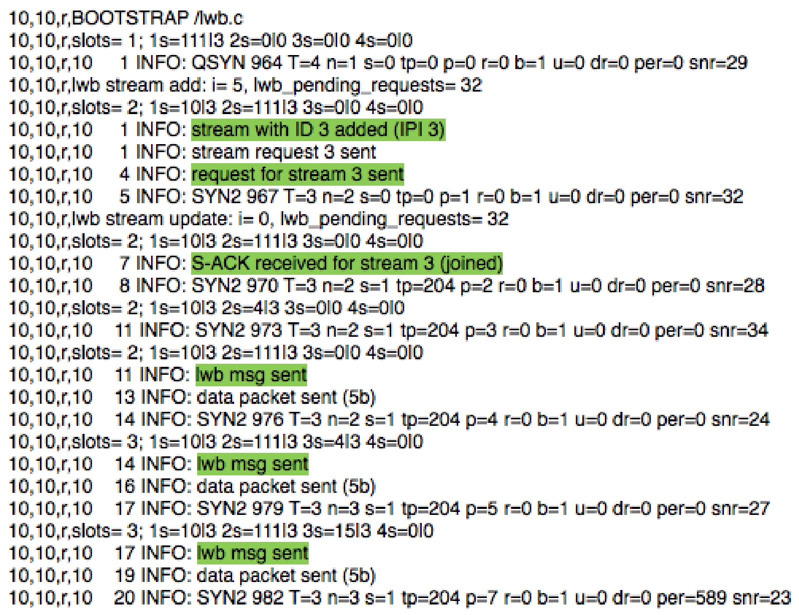


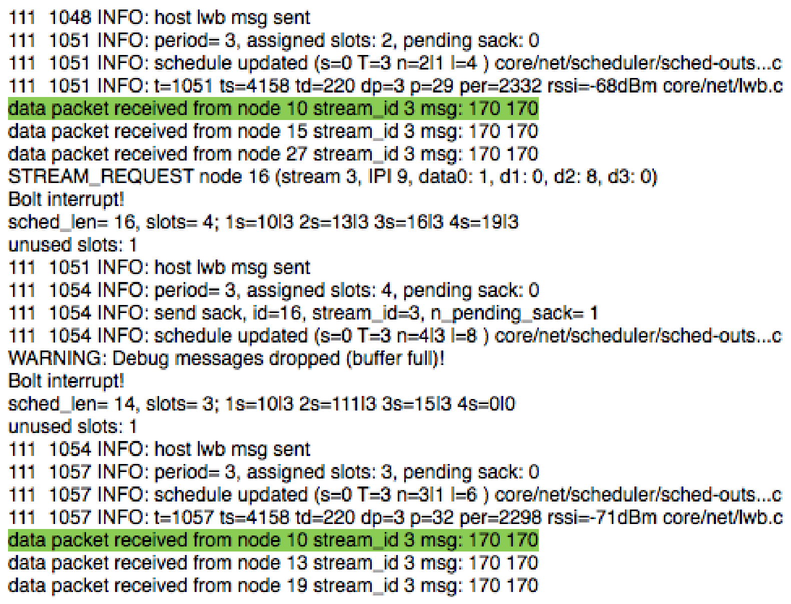
Figure 5.11: The value of the LED on different FlockLab nodes over time. The red line means the LED is on, which signals the RF module is on. The single on-impulse 0.5 s (is the used t_{sched2}) after the first impulse is signalling the receiving of the second schedule. All the nodes are synchronised and participate in the flooding.



```
10,10,r,BOOTSTRAP /lwb.c
10,10,r,slots= 1; 1s=111|3 2s=0|0 3s=0|0 4s=0|0
10,10,r,10     1 INFO: QSYN 964 T=4 n=1 s=0 tp=0 p=0 r=0 b=1 u=0 dr=0 per=0 snr=29
10,10,r,lwb stream add: i= 5, lwb_pending_requests= 32
10,10,r,slots= 2; 1s=10|3 2s=111|3 3s=0|0 4s=0|0
10,10,r,10     1 INFO: stream with ID 3 added (IPI 3)
10,10,r,10     1 INFO: stream request 3 sent
10,10,r,10     4 INFO: request for stream 3 sent
10,10,r,10     5 INFO: SYN2 967 T=3 n=2 s=0 tp=0 p=1 r=0 b=1 u=0 dr=0 per=0 snr=32
10,10,r,lwb stream update: i= 0, lwb_pending_requests= 32
10,10,r,slots= 2; 1s=10|3 2s=111|3 3s=0|0 4s=0|0
10,10,r,10     7 INFO: S-ACK received for stream 3 (joined)
10,10,r,10     8 INFO: SYN2 970 T=3 n=2 s=1 tp=204 p=2 r=0 b=1 u=0 dr=0 per=0 snr=28
10,10,r,slots= 2; 1s=10|3 2s=4|3 3s=0|0 4s=0|0
10,10,r,10    11 INFO: SYN2 973 T=3 n=2 s=1 tp=204 p=3 r=0 b=1 u=0 dr=0 per=0 snr=34
10,10,r,slots= 2; 1s=10|3 2s=111|3 3s=0|0 4s=0|0
10,10,r,10    11 INFO: lwb msg sent
10,10,r,10    13 INFO: data packet sent (5b)
10,10,r,10    14 INFO: SYN2 976 T=3 n=2 s=1 tp=204 p=4 r=0 b=1 u=0 dr=0 per=0 snr=24
10,10,r,slots= 3; 1s=10|3 2s=111|3 3s=4|3 4s=0|0
10,10,r,10    14 INFO: lwb msg sent
10,10,r,10    16 INFO: data packet sent (5b)
10,10,r,10    17 INFO: SYN2 979 T=3 n=3 s=1 tp=204 p=5 r=0 b=1 u=0 dr=0 per=0 snr=27
10,10,r,slots= 3; 1s=10|3 2s=111|3 3s=15|3 4s=0|0
10,10,r,10    17 INFO: lwb msg sent
10,10,r,10    19 INFO: data packet sent (5b)
10,10,r,10    20 INFO: SYN2 982 T=3 n=3 s=1 tp=204 p=7 r=0 b=1 u=0 dr=0 per=589 snr=23
```

Figure 5.12: The CP of the node with node-ID 10 debug output from the FlockLab test. The node is requesting a stream, gets the s-ack and is sending the data with the correct IPI of 3 s (important steps marked in green). The node worked as expected.

host is receiving multiple data packets from different nodes. Every node sent a stream request, the stream request was received and acknowledged, the stream was scheduled, the schedule was received and the packet was sent. Therefore, the network worked as expected.



```
111  1048 INFO: host lwb msg sent
111  1051 INFO: period= 3, assigned slots: 2, pending sack: 0
111  1051 INFO: schedule updated (s=0 T=3 n=2|1 l=4 ) core/net/scheduler/sched-outs...c
111  1051 INFO: t=1051 ts=4158 td=220 dp=3 p=29 per=2332 rssi=-68dBm core/net/lwb.c
data packet received from node 10 stream_id 3 msg: 170 170
data packet received from node 15 stream_id 3 msg: 170 170
data packet received from node 27 stream_id 3 msg: 170 170
STREAM_REQUEST node 16 (stream 3, IPI 9, data0: 1, d1: 0, d2: 8, d3: 0)
Bolt interrupt!
sched_len= 16, slots= 4; 1s=10|3 2s=13|3 3s=16|3 4s=19|3
unused slots: 1
111  1051 INFO: host lwb msg sent
111  1054 INFO: period= 3, assigned slots: 4, pending sack: 0
111  1054 INFO: send sack, id=16, stream_id=3, n_pending_sack= 1
111  1054 INFO: schedule updated (s=0 T=3 n=4|3 l=8 ) core/net/scheduler/sched-outs...c
WARNING: Debug messages dropped (buffer full)!
Bolt interrupt!
sched_len= 14, slots= 3; 1s=10|3 2s=111|3 3s=15|3 4s=0|0
unused slots: 1
111  1054 INFO: host lwb msg sent
111  1057 INFO: period= 3, assigned slots: 3, pending sack: 0
111  1057 INFO: schedule updated (s=0 T=3 n=3|1 l=6 ) core/net/scheduler/sched-outs...c
111  1057 INFO: t=1057 ts=4158 td=220 dp=3 p=32 per=2298 rssi=-71dBm core/net/lwb.c
data packet received from node 10 stream_id 3 msg: 170 170
data packet received from node 13 stream_id 3 msg: 170 170
data packet received from node 19 stream_id 3 msg: 170 170
```

Figure 5.13: The host CP debug output from the FlockLab test. The host is receiving multiple packets. E.g. packets from node 10 are received every 3 s, which is the IPI of the stream from node 10 (marked green). Receiving a packet at the right time guarantees the functionality of all previous necessary steps like synchronisation, sending a stream request, receiving a s-ack and sending the data regarding the schedule.

Chapter 6

Conclusion and future work

The goal was to enable a real-time functionality for the IoT or more precisely for a low-power network. Real-time scheduling of wireless communication was chosen as an example for a real-time network functionality. The DPP is used to enable the outsourcing of the complex scheduling task. A network protocol strongly based on LWB was implemented on the CP and BOLT was used to enable the interprocessor communication and outsource the Blink real-time scheduler on the AP. In a first step, the Blink scheduler functions were implemented in MATLAB® on a PC, which is connected to the AP via serial communication. The outsourcing of the scheduler to MATLAB® enabled better debug possibilities and testing. The conclusion and future work will be discussed in this chapter.

6.1 Conclusion

The FlockLab tests and the tests with a small network on the desk show the functionality of the outsourcing. The DPP and the outsourcing caused additional delays, but the outsourcing enabled a reordering of the LWB round structure, which allows the parallel computation of the schedule on the AP while the network communication is executed on the CP. The delay overhead of outsourcing the scheduler is decreasing for an increasing number of slots per round. If the maximum number of slots per round is larger than 10, the overhead for the outsourced real-time scheduler can be neglected, because the time for the network communication is longer than the time for outsourcing the schedule. Therefore, the real-time functionality is enabled by a very small overhead of the sending of the stream/schedule request and receiving the schedule.

6.2 Future work

The final goal is running the whole functionality on the DPP. Blink must be implemented on the AP for the DPP to enable the real-time functionality on its own. The implementation of Blink on the AP should be possible as the AP is more powerful than a CC430 and has with 64kB SRAM and 256KB Flash definitely more memory available than the implementation on the CC430 (see [1] for a limited Blink implementation on a CC430). Additionally, Blink was also implemented on a ARM Cortex M4 (same microcontroller family as the AP, but with 72 MHz clock frquency instead of 48 MHz) by [4]. They were able to execute Blink in less than 30 ms for a comparable stream set as the tests done in chapter 5. Furthermore, running the scheduler on the AP allows to dismiss the serial communication, which makes the execution faster and more predictable.

Bibliography

[1] J. Acevedo, “Real-time scheduling on resource-constrained embedded systems,” Master’s thesis, TU Dresden, 2016.

[2] F. Sutton, M. Zimmerling, R. Da Forno, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel, and L. Thiele, “Bolt: A Stateful Processor Interconnect,” in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. New York, NY: ACM, 2015, pp. 267–280.

[3] 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*. New York: ACM, 2012.

[4] M. Zimmerling, L. Mottola, P. Kumar, F. Ferrari, and L. Thiele, “Adaptive real-time communication for wireless cyber-physical systems,” *ACM Trans. Cyber-Phys. Syst.*, vol. 1, no. 2, pp. 8:1–8:29, Feb. 2017. [Online]. Available: http://doi.acm.org/10.1145/3012005

[5] T. I. Inc., “Datasheet cc430f5147,” 2013, [Online; accessed 6-June-2017]. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc430f5147.pdf

[6] T. I. Inc., “Datasheet msp432p401r,” 2017, [Online; accessed 6-June-2017]. [Online]. Available: http://www.ti.com/lit/ds/symlink/msp432p401r.pdf

[7] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, “Efficient Network Flooding and Time Synchronization with Glossy,” in *10th International Conference on Information Processing in Sensor Networks (IPSN), 2011 : 12 - 14 April 2011, Chicago, IL, USA*. Piscataway, NJ: IEEE, 2011, pp. 73–84.

[8] K. Leentvaar and J. Flint, “The capture effect in fm receivers,” *IEEE Transactions on Communications*, vol. 24, no. 5, pp. 531–539, May 1976.

[9] M. Spuri, “Holistic Analysis for Deadline Scheduled Real-Time Distributed Systems,” INRIA, Research Report RR-2873, 1996, projet REFLECS. [Online]. Available: https://hal.inria.fr/inria-00073818

[10] 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*, ser. IPSN

'13. New York, NY, USA: ACM, 2013, pp. 153–166. [Online]. Available: http://doi.acm.org/10.1145/2461381.2461402

[11] R. Jscob, M. Zimmerling, P. Huang, and L. Thiele, "Towards Real-time Wireless Cyber-physical Systems," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, S. Altmeyer, Ed. Piscataway, NJ: IEEE, 2016, pp. 7–9.

[12] R. Jacob, M. Zimmerling, P. Huang, J. Beutel, and L. Thiele, "End-to-end Real-time Guarantees in Wireless Cyber-physical Systems," in *Proceedings of 2016 IEEE Real-Time Systems Symposium (RTSS)*. Piscataway, NJ: IEEE, 2016, pp. 167–178.