Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Unleashing the potential of Real-time Internet of Things

Semester Thesis

Andreas Biri

abiri@ethz.ch

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

Supervisors:
Romain Jacob
Prof. Dr. Lothar Thiele

24th December 2017

Acknowledgements

I would like to express my gratitude to Romain Jacob for his supervision and guidance during this thesis. His strong personal dedication to his work and willingness to share his professional opinion as a researcher illuminated and inspired me and greatly helped to shape the project. With his wealth of pedagogic knowledge and readiness to discuss and refine new concepts, our meetings always resulted in interesting and constructive discussions and gave me invaluable inputs. His supervision provided me with highly-useful tools and concepts as a researcher and greatly improved and formalised my scientific reasoning. I hope that this thesis founded on his previous works will prove helpful and expand the spread of his innovative research.
Furthermore, I am grateful to Reto Da Forno for taking the time to introduce me to the existing code structures and helping me with the engineering on the physical platform.

I would like to thank Prof. Dr. Lothar Thiele for enabling me to conduct this semester thesis in his group and therefore allowing me to work on such an interesting topic. The supporting staff and all other members at TEC offered me a truly enjoyable and productive working environment.

Abstract

With the recent surge in the interest for the *Internet of Things* (IoT) and an increased deployment of *cyber-physical systems* (CPS) in commercial and industrial applications, distributed systems have gained a significance influence on modern civilization and are performing increasingly complex tasks. Building such platforms in a reliable manner is challenging, as they include concurrent tasks on the application and the communication layers. As the majority of such devices features a single processor, tasked with both communicating over the network as well as sensing and computing, real-time scheduling conflicts arise as the resource separation of the applications in software is difficult to manage.
To achieve such independence, we propose a platform consisting of dedicated *application* (AP) and *communication* (CP) processors which are completely decoupled in terms of resource access, clock speeds and power management using BOLT [1]. Leveraging this hardware separation, we then use the *Distributed Real-time Protocol* (DRP) [2] to provably provide *end-to-end* real-time guarantees for the communication between distributed applications over a multi-hop wireless network. By establishing a set of contracts at run-time, DRP ensures that all messages reaching their destination meet their hard deadline. To demonstrate this, we implement the BLINK scheduler [3] directly on the AP and adapt the LWB [4] round structure to use DRP as a *control layer* protocol. We show that our system is capable of supporting several hundred simultaneous streams and can respond to requests in maximally 3 stream periods over up to 10 hops.

Contents

Chapter 1

Introduction

"We have a deep need and desire to connect. Everything in the history of communication technology suggests we will take advantage of every opportunity to connect more richly and deeply."
— Peter Morville [5]

With the commercialisation of the Internet at the turn of the century, *digitalization* and *the cloud* became buzzwords well-known to anyone reading the news [6]. While in the past, existing providers mainly focused on centralized, outsourced offers, recent developments in embedded systems and wireless communication allowed companies to bring their services closer to the end-user. Suddenly, everything ranging from your TV, the lights in your office up to the industrial controller regulating the heating of your building is inter-connected and directly accessible. An entirely new domain, the so-called *Internet of Things*, has been created; software companies like Google have started developing dedicated operating systems and giants like Amazon and Verizon are contending for startups in the field to position themselves for the expected coming surge in applications and customers [7, 8]. In the past decade, this new concept awoke the interest of the general public and kept on growing (see Figure 1.1).

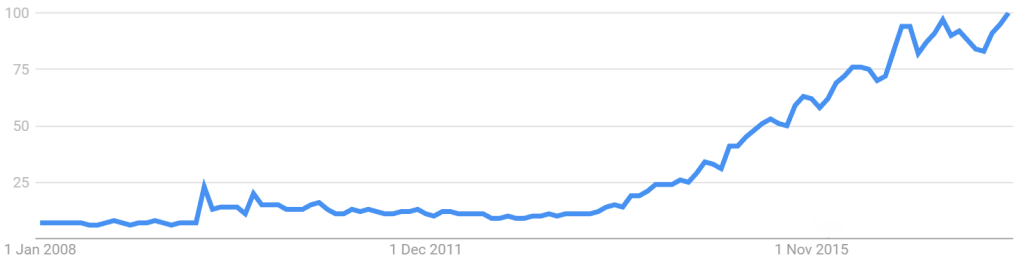

Figure 1.1: Last month showed more interest (counted in Google search queries) in IoT than ever before [9] (Scale is relative to its maximum popularity).

1.1 Cyber-physical systems

The Internet of Things (IoT) falls under the broader category of *cyber-physical systems* (CPS). These "smart" systems generally describe the integration of interacting networks, computational components and physical processes [10]. Applications range from traffic flow management and electric power generation & delivery to emergency response systems and personalized health care. Especially in industrial environments, an input or feedback loop is a central part of the concept and requires precise timing and bounded delays. Therefore, such requirements must be tightly integrated into the general design layout. Due to the devices distributed nature, centralized controlling is often difficult, inefficient and not well scalable. In general, one can therefore define the following main properties under consideration for a system's performance:

- **Scalability:** The behaviour under an increasing number of participants or computational load is a central aspect for any network-related tasks. To leverage the distributed setting, nodes need to predominantly work in their close surroundings and minimize far-reaching effects.
- **Connectivity:** As already described in the name IoT itself, devices must be able to interact with each other. Important characteristics hereby are the number of neighbouring nodes, the transmission range, link capacity and resilience to failures.
- **Power consumption:** As CPS nodes often operate independently of a stable power supply, supplementing techniques such as energy harvesting and batteries have to be considered. By leveraging low-power components, the period of application can be extended.
- **Computation power:** The load on the processor units can directly affect timing behaviour and response times and should therefore be distributed and in reasonable relation to the hardware properties.
- **Costs:** Depending on the field of application and budgetary considerations, implementation costs strongly affect the spectrum of solutions.
- **Reliability:** The ability of the system to cope with erroneous messages, network or command outage and delays in its communications is critical. As external events can influence devices and alter their environment, it is important to anticipate and mitigate damaging effects.

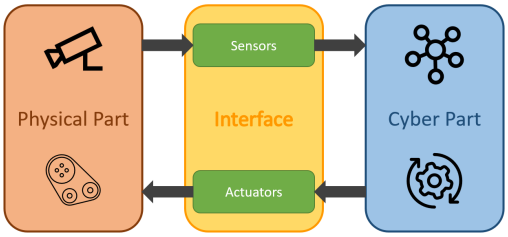

Figure 1.2: A CPS consists both of a a *physical* domain representing real world processes and a *cyber* part which includes transmission and processing of signals.

1.2 Motivation

> *"Reliability will be key. If such systems prove to be unreliable, people will leave in droves. So that's a primary requirement."* — Vinton Cerf, *Chief Internet Evangelist* at Google [5]

As mentioned in section 1.1, *reliability* is one of the main concerns when designing a cyber-physical system. CPS strongly depend on timely delivery of packets as a signal for an industrial machine might be mission-critical and useless if it arrives after the point in time where a certain action should have been performed. Therefore a well-designed communication scheme is of great importance.
Real-time network functions such as scheduling are complex and the communication between sensing, actuation and computing elements often needs to comply with given hard real-time constraints [2]. Guarantees can only be achieved by designing a system consisting of both hardware and software which allows to limit disruptions such as dependencies between components. Furthermore, it should offer optimal performance to keep requirements well within feasible limits.

A first complete solution to this problem was proposed by Jacob et al. [11]. Employing the *BLINK* scheduler [3] on the network level to provide real-time guarantees for wireless communication and leveraging the processor interconnect *BOLT* [1] as hardware to separate the communication and application aspects, the *Distributed Real-time Protocol* (DRP) provides *end-to-end* real-time guarantees between interfaces of distributed applications and prevents buffer overflows along the transmission chain.
Paired with LWB [4] as a transport protocol on multi-hop wireless networks, DRP offers an API to distributed applications and relieves single nodes from the complexity of transmission within given time bounds.

1.3 Goals

This thesis aims to develop an implementation of DRP and deliver such guarantees using the methods described in the paper [2]. The entire functionality should reside on the nodes themselves without dependencies on external infrastructure and should allow dynamic registration and scheduling of real-world traffic. Building on previous work from Walter [22] and an existing hardware platform called *Dual-Processor Platform*, we designed and built a system which minimizes dependencies between hardware and software components and offers an adaptable platform for *end-to-end* communication in a distributed network.

The rest of this document is structured as follows: Chapter 2 *Background* offers an overview of the underlying protocols and hardware components which were employed during development. In chapter 3 *System design*, we present the system design of our DRP implementation and detail influencing considerations. Chapter 4 *Implementation* sketches the final product and gives an insight into its technical aspects. Chapter 5 *Evaluation* describes how the system performs under testing and which further experiments could be performed to ready the implementation for practical applications. In chapter 6 *Conclusion & future work*, we present our findings and show possible extensions of the concept.

Chapter 2

Background

In distributed networks, applications situated on different nodes aim to exchange messages between each other. The *end-to-end* performance of the system depends on the entire transmission chain, encompassing the application on the source node, the communication on the distributed network and the timely reception and reading of the internal queues at the receiver. In order to achieve a reliable and assured service, we require robust components with bounded behaviour on all levels:

- **Physical layer:** The *Low-power Wireless Bus* (LWB) delivers messages using energy-efficient flooding by applying *Glossy*. This stands in stark contrast to traditional routing which directly or indirectly depends on the links and number of hops between source and destination,.
- **Network layer:** Using TDMA to separate communication in the time domain and only a small contention slot for non-deterministic access, we then employ *BLINK* as a reliable and highly adaptive scheduler which considers the hard real-time constraints of various streams.
- **Hardware:** To allow for concurrent and independent transmission and application tasks such as sensing or calculations, we deploy the *Dual-Processor Platform* (DPP) with separate processors for each aspect. Leveraging *BOLT* to decouple the processors, we can design the communication and application sub-systems independently and control how they interact.
- **Control plane:** To guarantee correct and timely executions on the hardware and networking levels, the *Distributed Real-time Protocol* (DRP) acts as an additional layer and interconnects the different parts. By defining timing behaviour such as network deadlines and update intervals, it manages the underlying components and offers a simple API to the end-user.

This chapter presents these four fundamental components - LWB, BLINK, the DPP and DRP - in more details and presents how they are interconnected.

2.1 Low-power Wireless Bus

The *Low-power Wireless Bus* (LWB) is a communication scheme which turns a multi-hop network into an infrastructure similar to a shared bus [4]. All nodes are potential receivers and can send data in one-to-one or one-to-many broadcasts at the same costs. The transmission pattern of LWB is based on *Glossy*, a flooding protocol devoid of topology-dependent state which enables multi-hop communication without routing. The concept was extended for improved all-to-all communication in *CHAOS* [12].
LWB uses the concepts of real-time streams to request and distribute transmission opportunities for interested nodes. Such streams contain information about the *inter-packet interval* (IPI) and starting time as well as intended recipients. All transmissions in LWB occur during given slots which are detailed by a schedule computed and communicated at run-time. The allocated data slots use *time-division multiple access* (TDMA) to guarantee an uncontended channel for messages and therefore do not interfere with each other.

2.1.1 Glossy

The underlying transport scheme of LWB is *Glossy* [13]. In stark contrast to traditional link-state based routing protocols, Glossy transmits packets without considering surrounding neighbours lists or link parameters. It does so by globally scheduling synchronous floods through a logically centralized controller, also called *host*. The broadcasts are precisely synchronized in time to guarantee simultaneous transmission of the same data packet. As nodes again broadcast and therefore relay any message they received, the packets of different senders will be superimposed. Due to constructive interference, the resulting signal-to-noise ratio (SNR) is still strong enough to be received successfully by neighbouring nodes and the packet can continue to propagate through the system. This allows for low-power transmission on the nodes despite agnostic flooding.
Glossy defines a very simple communication behaviour for all nodes: Whenever a device receives a given message for the first time, it chooses between two options. If the node itself is the single intended target, it will transfer the packet to its internal input queue. In any other case, it will broadcast the packet and therefore flood the network so that the packet further propagates to its intended recipients (see Figure 2.1).

This method offers two properties which are central for LWB: Due to the synchronized transmission, Glossy provides accurate global time synchronization which can be leveraged to schedule the communication rounds and clearly define round durations and wake-up times.

Furthermore, it is by nature independent of the volatile network state. Because transmission is independent of the node's neighbours, communication is much more resilient to interference, node failures and mobility. Thanks to this property, LWB was shown to outperform comparable schemes and provide a reliability higher than 99.9% [13].

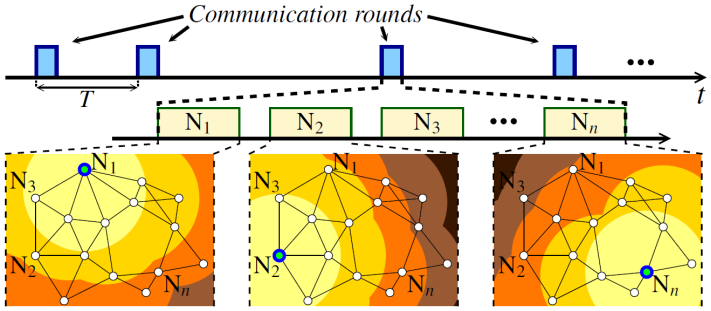

Figure 2.1: A packet, originating from a single node, travels throughout the network. Notice that the intervals between the communication rounds depends on the scheduler and might vary. [13]

2.1.2 Round structure & scheduler

LWB transmissions are structured into rounds of clearly defined segments. The execution of each segment is time-triggered and synchronized throughout the network by leveraging Glossy's accurate time synchronization.
Rounds can be split into two periods: during the *active* phase (the *communication round* shown in Figure 2.1) at the beginning of the round, nodes exchange messages such as data packets and scheduling information. Afterwards, the nodes determine their own operations during the following *inactive* phase and are allowed to sleep. The length of the active phase is fixed and depends on the maximum number of data slots which are allowed to be scheduled. Depending on the calculations of the host, the start of the next round will be delayed as much as possible to minimize energy consumption.

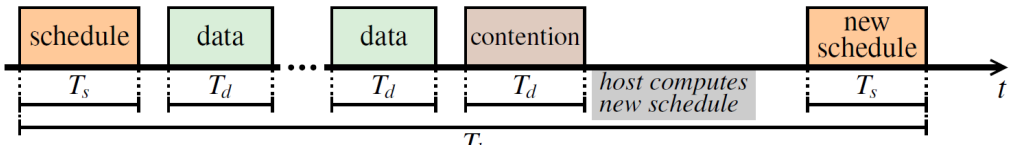

Figure 2.2: During a communication round, the schedule is sent twice to prevent energy-inefficient listening operations during failures [13]. Notice that a long schedule computation on the host prevents the entire network from going into sleep mode and conserving energy.

A communication round consists of the following segments (see Figure 2.2):

- **Schedule S_i:** The first schedule received during a communication round is a repetition of the schedule sent at the end of the previous round. It is primarily used for better synchronization and resilience to interference, as a node is only allowed to send during a round if it has successfully received the corresponding schedule [4].
- **Data:** In each slot, the node specified by the schedule is allowed to trigger a flood of the network with its own package. Packets will then be transmitted according to Glossy (see section 2.1.1 *Glossy*).
- **Contention:** The contention slot is only used under non steady-state conditions. It is the sole slot which is not clearly allocated to any specific node and offers random access. Nodes which intend to register a new stream can try to do so during this period; however, no assurances are given from the host that it will receive it [14].[1]
- **Schedule computation:** During this segment, only the host is active and computes the schedule for the next round, based on the state from previous rounds and possibly information gathered during the contention slot.[2]
- **Schedule S_{i+1}:** The new schedule, including the start of the next round, is announced by the host.

2.2 BLINK

LWB provides highly reliable communication but does not include the notion of real-time traffic. Furthermore, its standard scheduler is limited in functionality and therefore offers room for improvement. BLINK is a real-time scheduler which is adaptive to changes in network state and application requirements [3]. Unlike previous approaches [15, 16], it is agnostic to the current network state as it relies on LWB [4] for providing the underlying communication scheme. Its global view on the real-time traffic requirements of requested streams allows for centralized control while still meeting all hard real-time constraints.

BLINK employs *Earliest-Deadline-First* (EDF) [17] strategies to both guarantee optimal energy consumption and allow all admitted streams to provably meet their deadlines. With its online schedule computation based on the traffic metrics of source nodes, it is highly adaptive to dynamic changes and can quickly react to changing network states and node failures.

[1] We will see that in DRP, contention is redundant and therefore excluded from the round.
[2] In the implementation of DRP, the schedule is already pre-computed and is simply fetched.

2.2.1 Concept

In stark contrast to the standard LWB scheduler which does not take real-time traffic requirements into considerations and only tries to minimize energy consumption [4], BLINK verifies viable scheduling before admitting a stream [3]. The calculations are based on stream requests from the source nodes detailing their behaviour and include the following parameters:

- **Inter-packet interval (IPI):** This interval specifies the elapsed time between the periodic release of two packets at the source. The source stream must have at least one opportunity to transmit during the IPI.
- **Deadline:** As CPS contain hard real-time constraints to deliver packets, such requirements must be modelled. BLINK assumes that the relative deadline is always smaller or equals to the IPI so that one stream only has at most one packet available at any time [17].
- **Start time:** This parameter defines the release time of the earliest packet and indicates the beginning of the stream.

Scheduling principle

As Glossy-based flows allow the network to appear like a one-hop neighbourhood [13] for the scheduler, the transmission durations are independent of the actual receivers and can be regarded as *atomic* [3]. Therefore, BLINK simply schedules according to EDF [17] and always allocates the stream with the nearest deadline.

Starting a round induces fixed costs such as the two schedules which have to be sent regardless of the number of scheduled slots (see Figure 2.2). Therefore, this additional cost is minimized if it is spread over a maximal number of data packet transmissions. This is achieved by delaying the start of the next round as much as possible while preventing any deadline misses. As the earliest packet deadline requires a communication round and compels the host to send schedules in either case, it is energy optimal to additionally schedule as many streams as possible. Note that it might be possible that such a full round could have been achieved before as enough streams were already released; however, this could potentially lead to reduced efficiency in the long run [3].

BLINK leverages efficient data structures using priority queues to reduce computational and memory demands on the executing hardware. This is necessary, as previous works did not apply EDF despite its proved realtime-optimality due to a large run-time overhead [3]. As CPS often consist of embedded systems with limited resources, an inefficient implementation reduces the algorithm's benefits.

2.2.2 Construction

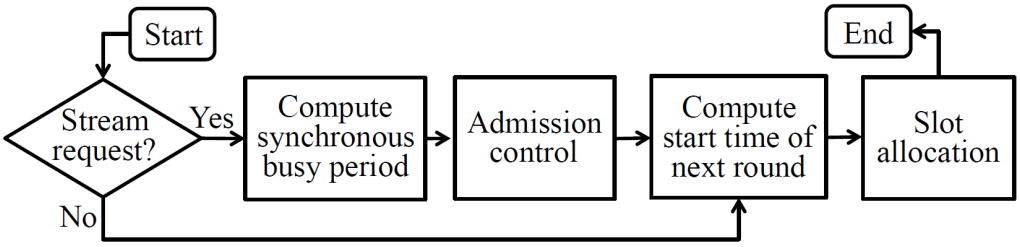

Figure 2.3: BLINK uses a chain of algorithms to process requests and schedule them according to EDF [3].

Every stream request will first pass a test called *admission control* which guarantees that admitting the additional stream does not result in missed deadlines in the future, i.e. that the new set of streams will never require a larger bandwidth than available. After the request has been processed, BLINK computes the start time of the next round; as described above, this should be delayed as much as possible to preserve energy while still allowing all streams to meet their deadlines. When this time is known, the algorithm can check which streams already possess released packets at that moment and will start allocating slots according to their deadline. In case the system is in steady-state and no stream requests arrive, the first two procedures can be skipped as seen in Figure 2.3.

Synchronous busy period

The admission control guarantees that including the new stream will at no point in time lead to a time interval where the required bandwidth exceeds the available one. One approach to achieve this is by artificially creating an interval of maximum demand by releasing all streams at the same time. It can be shown that if such an extreme situation does not result in a missed deadline, then it is safe to allow the request [3].
Furthermore, we can bound the time horizon which we have to check into the future by determining the so-called *synchronous busy period*. This duration spans from the point in time where the maximum demand is created to the first idle period where no more streams have a pending packet. If no violation of any requirements occurred up to this point, we can conclude that the stream set is schedulable; the worst situation one could experience afterwards would be another maximum demand. As this case is simply a repetition of the pattern which was just checked, the possibility of a missed deadline is eliminated even beyond this time interval.

Admission control

After having determined the maximal length of time which needs to be checked to assure that a given stream request can be handled by the system, we then start the actual testing. Following the exact same considerations as for the synchronous busy period, we do so by deliberately generating maximal demand at the start of the sequence. By continuously proceeding according to the allocation rules of EDF [17], we schedule streams as long as no deadline miss occurs. If the scheduling succeeds up to the end of the synchronous busy period, the request is accepted and will be included in the following computations. If however the additional requirements would result in a violation of previous guarantees and the test should fail, the request will be rejected.

Computation of the start time

While *continuous scheduling* (CS), starting one round right after its predecessor, and *greedy scheduling* (GS), starting a round as soon as any packet can be sent, are realtime-optimal and never miss deadlines as long as the stream set is schedulable, they can result in poor energy efficiency. This comes from the fact that in some situations, it would be favourable to further delay the start of the round and fully utilize all data slots. This intuitive idea of delaying the next round "as long as we possibly can" is the underlying reasoning of the realtime-*and* energy-optimal *lazy scheduling* (LS) [3].
It is obvious that there are limits as to how much delay is feasible until we might artificially create an interval of excessive bandwidth requirements which would result in missed deadlines. LS uses the notion of *future demand* to forecast the required bandwidth and deduces the resulting *minimal slack* time before the next LWB round must be scheduled.

Slot Allocation

As previously argued, maximizing the number of used slots inside a given round spreads the energy overhead of the schedules over the most goodput and therefore results in maximum energy efficiency. As delaying a stream even if it were already available can only result in equal or even worse schedulability, we will always try to maximize the number of streams per round [3].
The scheduling itself then happens according to the afore mentioned *earliest-deadline-first* (EDF) principle by always prioritizing the available packet with the earliest absolute deadline. As EDF is provably realtime-optimal [17] and computes the schedule during run-time while integrating changes on the fly, it is perfectly suited for our task [2].

2.3 Dual-Processor Platform & BOLT

Up to now now, we mainly considered the networking aspect. However, for the end-user, it is primarily the application tasks such as sensing and computation which determine usability. In the original LWB paper [4], the authors recommend using the time in-between two communication rounds to execute processes which are not networking related.
DRP does not rely on a separation in time, but leverages the *Dual-Processor Platform* (DPP) to completely eliminate dependencies between application and networking tasks and separate them spatially by using two dedicated processors.

2.3.1 DPP

Traditional platforms with a system-on-chip offering a single processor struggle to achieve the high timing precision and availability of requested resources to execute both communication and computation in a way which satisfies requirements. This feat is challenging, as application tasks are often event-triggered and require access to shared resources [1]. As most real-time wireless systems rely on TDMA to communicate with a deterministic channel access to guarantee throughput, communication occurs at clearly defined time slots and interrupts other tasks such as sensing. With a single core, no set of processes can truly run concurrently and requires a threaded environment which introduces timing uncertainties.
The *Dual-Processor Platform* (DPP) solves this inherent predicament of single-core systems by introducing two processors and a deterministic interface in-between.[3] Each processor has its clearly defined task set and is capable of operating independently by only asynchronously exchanging messages with its counterpart.

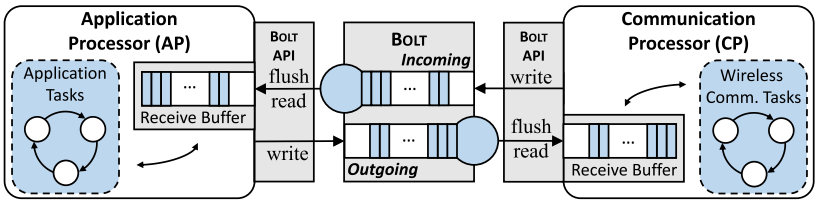

Figure 2.4: On the DPP, BOLT [1] interconnects the *application* processor and *communication* processor and uses its internal memory to simulate FIFO queues to decouple the message passing [2].

[3]The technical specifications of the below-mentioned processors can be found in section 7.1 *Technical specifications*.

Communication Processor

The *communication processor* (CP) is responsible for all communication procedures and includes the entire LWB processing stack. To preserve energy, it is primarily run in low-power mode during the inactive phase.
The *CC430* [26] is a system-on-chip providing a low-power 16bit processor which offers limited memory and computational capabilities. Through its integrated RF transceiver in the sub-1-GHz range, it can transmit up to 500'000 symbols per second while offering minimal energy consumption when idle.

Application Processor

The *application processor* (AP) handles the actual processes the embedded system is designed for. While only periodically polling its incoming queue to read packets from the network, it can sense and compute without interruption and therefore offers the ideal platform for time-critical applications such as feedback control of industrial processes and safety systems. For BLINK, the host leverages the extensive computational power to calculate the schedule and manage its state (see section 3.3.1 *Task distribution*).
Predominantly designed for application tasks, the *MSP432* [27] offers much more powerful hardware to store more than ten times as much data as the CP, execute tasks with an increased rate and process floating points with a dedicated FPU.

2.3.2 BOLT

BOLT serves as an interface to separate the two previous processors in time, power and clock domains [1]. A stateful processor itself, it offers its internal memory to emulate two virtual *First-In-First-Out* (FIFO) queues for the communication between AP and CP in non-volatile memory.
As long as the queue is not yet filled, a processor can write into the queue in a non-blocking manner and pass messages on to its counterpart while the latter does not have to immediately accept it and might even be in sleep mode. Both parties can choose when reading and writing best fits into their individual schedules and only have to take their previously guaranteed flush times into account.

While BOLT functions are prompt and have predictable timing characteristics [1], they further allow the processors to choose how to access the queues individually. Whereas interrupt-driven delivery can decrease the response time for time-insensitive tasks such as the stream request processing of a BLINK host, concepts such as the atomic slot of LWB require undisturbed execution and therefore strongly favour polling to read outgoing packets at the CP.

2.4 Distributed Real-time Protocol

2.4.1 Concept

In contrast to the afore mentioned concepts which target specific parts of the transmission chain between distributed applications, the *Distributed Real-time Protocol* (DRP) builds upon all previously discussed techniques. It offers a simple API for applications and provably guarantees to meet real-time traffic requirements and deliver packets to their recipients. While BOLT decouples networking from application, DRP again sees to correct functioning so that AP and CP act according to mediated contracts [2]. These contracts have to be accepted by all involved parties (source, network scheduler & destination) and contain the following parameters:

- **Source & Destination node:** For the standard BLINK protocol, only the node requesting resources is relevant. In contrast, DRP only allows one-to-one communication to guarantee that all parties can actually check and fulfil the requirements and therefore requires destinations to be explicitly defined.
- **Minimum message interval T_i:** This metric is very similar to the *inter-packet interval* of BLINK and LWB. However, we lessen the restriction on a strictly periodic appearance of packets and simply define a *minimal* interval between packets which might possibly be larger in practice.
- **Jitter J_i:** DRP simulates that the message release is sporadic with jitter. This way, BLINK can still assume a periodic release and does not depend on the initial phase or start time [2].
- **End-to-end deadline D_i:** This deadline extends the concept of BLINK's deadline to include the entire transmission chain from the API on the source AP to the API on the destination AP.

2.4.2 Device level

DRP leverages the DPP to reserve a dedicated processor (CP) exclusively to network related tasks and another, more powerful processor (AP) for the remaining work. This separation avoids possible access conflicts on shared resources and prevents an unpredictability of the involved execution times. The included BOLT interface therefore decouples communication and application and allows the system designer to customize the platform and compose it according to specific application requirements.

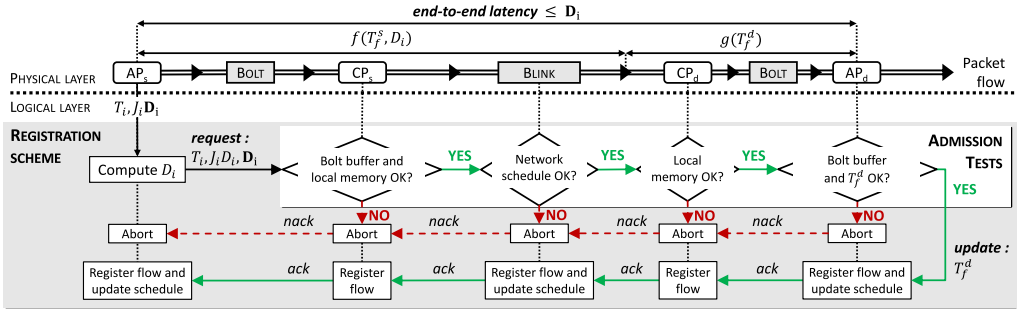

Figure 2.5: Before a request is accepted, it has to traverse multiple stages on source, network scheduler and destination [2].

2.4.3 System level

By building an additional layer on top of an adaptable scheduler such as BLINK and utilizing the reliability and simplicity of LWB, DRP provably guarantees that real-time traffic requirements are met for all messages received at the application interface. Source nodes use predefined *control flows* which are dedicated one-to-one streams to the host to send new stream requests and receive responses. As those flows provide assigned time intervals for nodes to communicate with the host, this renders random access methods superfluous. Therefore, DRP does not include the contention slot of the original LWB round [4] anymore.

Figure 2.5 gives an overview over the different steps of a stream request, starting at the source node and ending at the destination. It is important to notice that for DRP, the host **must** be one of the directly involved parties, i.e. either the source or the destination of any given stream. The DRP concepts may possible extend to any type of traffic as the network scheduler is logically independent from the stream, but the currently provided guarantees [2] hold only for host-to-source and source-to-host streams.

A contract contains information which needs to be checked on the source and the destination node as well as by the network scheduler:

- **Buffers:** Each processor in the chain verifies that enough space is available in its internal memory and that it is capable of flushing the BOLT queue sufficiently often to prevent congestion. These tests guarantee that no message buffers along the transmission chain overflow at any point in time and no packets are lost during transmission. Furthermore, the minimal flush time assures that packets are available for the application when promised and do not still reside in a buffer. This could otherwise cause significant differences, as a complete flush of the BOLT queue takes multiple milliseconds [1].

- **Network:** The admission test in BLINK on the network scheduler assures that all network deadlines are met and energy consumption is minimized.[4] By having a global view on the requirements of the network, such a scheduler could even enforces policies such as limited bandwidth for single nodes or minimal deadline requirements to lessen the stress on the system.

After the destination has accepted the new stream and agreed to reserving corresponding resources, a *DRP Response* packet will be sent in the opposite direction. This message again uses allocated control flows and notifies all entities in the processing chain of the result.
If one of the entities conducting the tests realizes that it cannot fulfil the requirements detailed in the request, a NACK (*Negative-Acknowledgement*) will be returned via a *DRP Response* and no further parties along the path are contacted.

The *end-to-end* deadline consists of the *network* deadline and the sum of the flush intervals of the different buffers. A successfully requested stream must guarantee that the requested deadline is larger than the sum of the upper bound on the latency of all sub components. With the tests on the buffers and on the network, we verify that sufficient capacity is available before scheduling a flow. Therefore, we can guarantee that in case a packet does arrive at the destination API, it will have met its deadline and is received on time.[5]

[4]Due to the concept of checking an artificial period of maximal demand (see section 2.2 *BLINK*), it might be that streams are rejected which could in practice be scheduled. However, such a conservative approach guarantees that no false positives might occur which are accepted despite an unschedulability of the resulting stream set.

[5]No protocol can absolutely guarantee that packets will be delivered successfully, as environmental factors can prevent physical communication even under the most robust schemes.

Chapter 3

System design

In the following sections, we specify the problem setting and detail how the final system was designed to optimize performance while still providing a modifiable platform. This chapter further explains the reasoning behind our design decisions and the considerations under which they were taken.

3.1 Problem setting

The primary focus of DRP lies in guaranteeing that packets are delivered from one endpoint to another according to real-time traffic requirements. As described in section 1.2 *Motivation*, CPS require such temporally bounded delivery for feedback loop stability and to describe the behaviour of physical processors with maximal precision [2].

In a general setting such as for *wireless sensor networks* [18], the network consists of multiple *source* nodes and a centralized controller on a *host* node. While former are primarily occupied with gathering sensor data and relaying messages, the *network manager* orchestrates communication and often features improved hardware for more extensive calculations and post-processing. Furthermore, the host can act as a gateway for the network and is often connected to a backbone infrastructure for accessibility over the Internet.

To coordinate transmissions and mediate access to the wireless bus, each node notifies the controller of its needs using a *Request* packet. This message contains information about the stream such as source & destination node, the *minimum message interval*, the maximal amount of jitter which might occur between releases as well as the *end-to-end* deadline relative to the release of a packet. After processing the request, the host then returns a *Response* packet, informing the node about its decision on whether it will integrate the stream and schedule it in future rounds.

3.2 Design considerations

One of the core principles of DRP is to decouple the individual components and allow them to schedule and execute tasks independently. As previous work already guarantees reliable communication (section 2.1 *Low-power Wireless Bus*) and provides suitable hardware (section 2.3 *Dual-Processor Platform & BOLT*), this thesis focuses on the *implementation* of the DRP protocol itself and aims to:

i) leverage the decoupled processors to simultaneously communicate and execute the protocol & application tasks.

ii) at any time provide an up-to-date schedule including recent requests in a time-efficient manner.

Our design completely decouples the round management and the schedule computation and therefore frees the application processor from any synchronisation constraints with other parties. By leveraging the DPP to concurrently process streams & compute schedules on the AP and communicate with other nodes over the network using the CP, we optimally support the core ideas of DRP.

3.3 System overview

3.3.1 Task distribution

The DPP was designed to specialize processors and allow them to exclusively focus on their respective primary purpose. By respecting this separation and using it as a design principle, we split the range of operations of nodes into two parts. The entire application-specific functionality, including the DRP-related functions, has been concentrated on the AP to reduce implementation complexity and dependencies on other processors. This left the CP with the simple task of transmitting packets without any packet inspection or other extensive analysis.

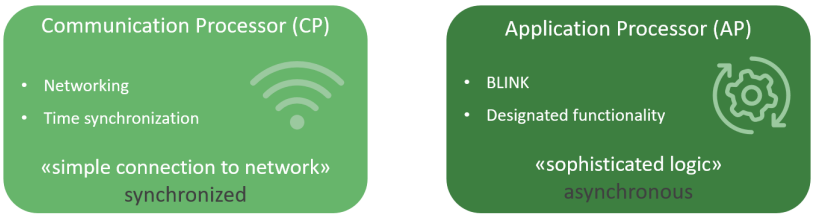

Figure 3.1: CP and AP both have separate responsibilities and are not synchronized with each other; the CP however is strictly timed by the LWB round.

There are three major processing steps for any DRP request (see Figure 2.5):

- **Source test:** The node which intends to send a request first makes sure that its internal (BOLT) queues have enough memory left to receive additional packets. Additionally, it checks that *both* involved processors, AP as well as CP, are able to handle the increased traffic.
- **Network test:** The network scheduler verifies that the traffic requirements comply with given restrictions [2] and that the network can manage further traffic. Note that this part is always executed on the host node, as it runs the scheduler locally.
- **Destination test:** Similar to the first test, the recipient of the stream ensures that no packets will be dropped internally due to memory overflows and that necessary flushing times are respected.

Following our design paradigm, these verifications along with all other DRP functionality are executed on the AP of either a source or host node. Notice that during the first and third test, the AP also examines the admissibility in regard to CP performance. As those tests are logically independent of the execution location and only require state which is available on the AP, the AP also assumes this role for ease of implementation. This further reduces the required changes in the code base of the CP and preserves the resemblance to the legacy code for improved maintainability.

Application processor

The processes which are running on the AP depend on the role of the node in the network.[1] For a *source* node, the primary intention of the application processor is to fulfil its purpose inside a larger system and e.g. monitor the environment or process data. While the specifics depend on the implementation, each such node will have to be able to request DRP streams and process responses from the host whenever it intends to change its traffic requirements.

A *host* node on the other hand is primarily focused on handling data flows inside the network and processing DRP requests. The host's AP provides the CP with the schedule upon query and can read & handle DRP-related packets from BOLT or generate new requests itself.

[1] We will see in Chapter 4 *Implementation* that the role of the node is defined by the node ID. This ID identifies the node and is compared against a hard-coded *Host* ID.

Communication processor

The CP contains only a low level of sophistication. Its primary function is to send & receive packets on the network and relay them to the BOLT interface. It therefore acts as an intermediary between AP and the network and connects the application to other nodes. The processor is further responsible for round management (behaving according to the timing specifications of LWB) and therefore stores the schedule. The saved schedule is also used on the host node for sending feedback about the activity of other scheduled nodes to the controller (see treatment of unused slots in [22]).

It is important to note that apart from sending & receiving the schedule, the functionality of the CP does not differ for source and host node. This demonstrates once more that the CP simply serves as an interface and does not contain state or functionality depending on the role of the node in the bigger picture.

3.3.2 Round structure

The round layout for communication adopts its general structure from the original LWB [4]. As discussed in section 2.4 *Distributed Real-time Protocol*, the usage of control flows to send DRP requests and responses renders both the *contention* and the *stream acknowledge* slot redundant. Source nodes now possess a deterministic way of interacting with the network scheduler and therefore do not require such random access methods anymore. Hence, the round simply consists of a transmission of the schedule, subsequent data slots and the distribution of the schedule for the next round as shown in Figure 3.2.

This new round structure is only directly influencing the communication processor. The highly time sensitive nature of Glossy [13] requires tight synchronization between CPs and therefore requires precisely controlled and deterministic executions.

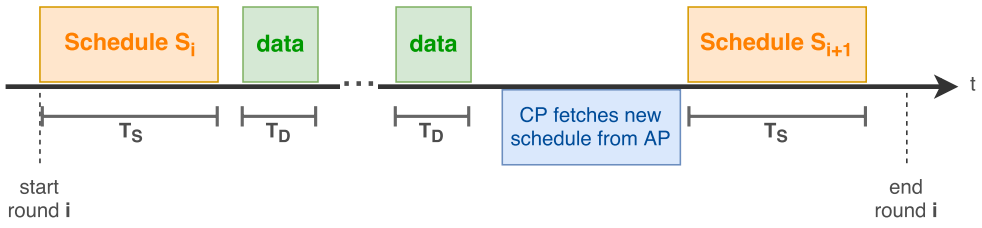

Figure 3.2: Compared to Figure 2.2, the contention slot was discarded and the schedule computation time was reduced by simply fetching a new schedule from the AP instead of calculating one from scratch.

The behaviour of the AP on the other hand has been designed to be as oblivious to the external network as possible. This decouples the execution on the processors and makes the design much simpler and less error-prone. The application processor does *not* have to follow a strictly timed or fixed schedule, but runs independently of other entities and simply reacts to messages received from the CP.

While there are no messages in the incoming BOLT queue, the host's AP processes its internal request queue & calculates an updated schedule after dispatching each request.[2] In the case of incoming messages from the CP, BOLT will trigger an interrupt on the AP, whose handler quickly returns with the requested result. Because the cycle on the AP runs independent of the network, the tasks on the AP are not relying on tight synchronization with any other, possibly time-triggered process and can therefore run irrespective of the state of other nodes. Furthermore, it is oblivious to the point in time when the schedule is fetched; this trait can be exploited as we will see in section 3.4.2.

A central feature is the pre-computation of a valid schedule. This guarantees that a schedule request via interrupt can be handled immediately by the AP. Furthermore, the time required to fetch a schedule from the AP is completely independent of any other parameters such as the size of the data structure, the number of streams or the actual traffic parameters. Even though those factors strongly influence the time required to calculate a schedule, all calculations are performed before the request was received and do not depend on the actual timing of the interrupt. This further allows complete decoupling of CP and AP, as both can process their tasks independently and simply communicate when required. As this happens over *BOLT* with a non-blocking write, the CP must not wait for the response and can continue normal execution.
Reducing the effective schedule computation (or rather fetch) time for the CP achieves both increased energy efficiency and additional network bandwidth. As source nodes need to wait for the schedule and cannot conduct other operations such as sleeping, waiting for a single point in the network otherwise reduces the scalability of the system. Furthermore, for a given fixed round length [4], decreasing the schedule computation time improves the available bandwidth by increasing the maximal number of data slots schedulable in one round [3].

3.3.3 Inter-process communication

BOLT provides both options to either deliver packets using interrupts or let the connected processors poll the queues at their leisure. Interrupts provide the fastest access time [22] and only require resources when a message is actually

[2] See section 4.3.2 *State machine* for a more detailed description of the involved state machine.

present. However, one needs to be careful not to disrupt time-critical phases such as the active transmission and reception between nodes. Therefore, care has to be taken when designing the message passing between the two processors.

CP → AP

The AP on a source node is not obligated to read the BOLT queue at given points in time. As a destination node of a stream, it merely guaranteed to flush the queue at a given minimal frequency in its DRP contracts. However, it is not further bound in its reaction time and can therefore decide individually when it is best suited to poll the incoming BOLT queue; this can be scheduled to limit the disturbance on its application process. Therefore, interrupt-driven delivery is not required in such a case, but still offers a viable option for a system designer.

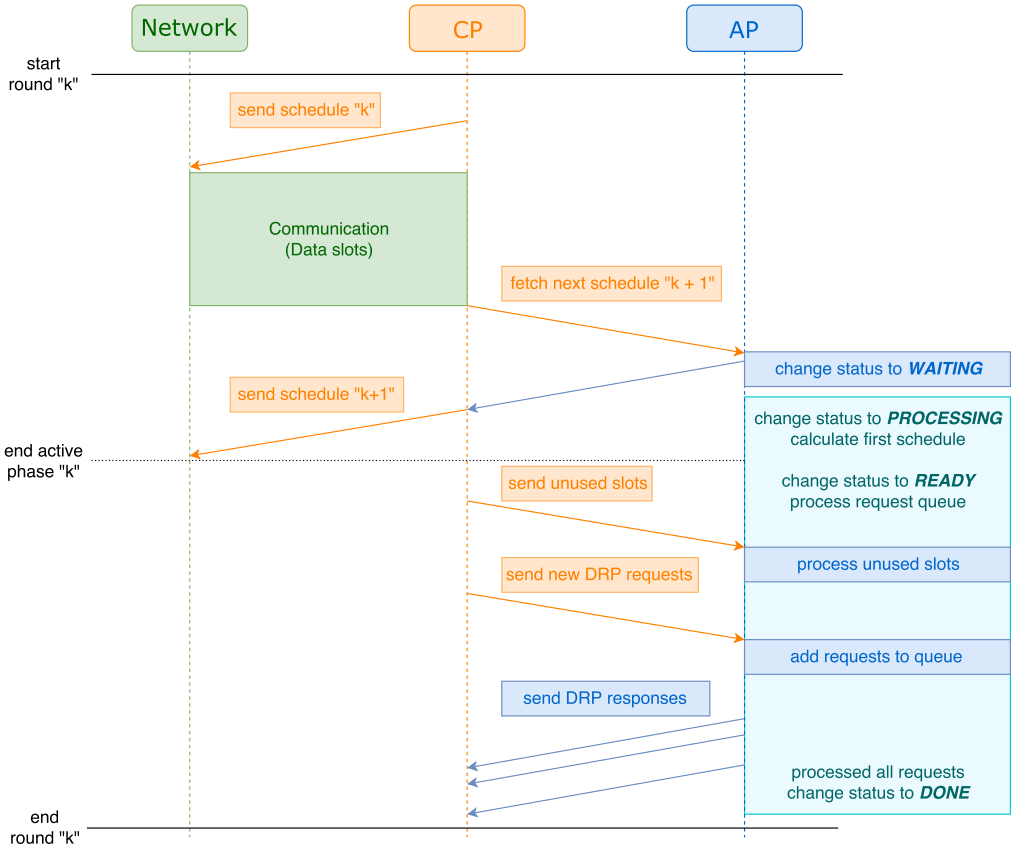

Figure 3.3: For the standard implementation, the schedule is fetched at the usual LWB location. As the absolute point in time of the request is not relevant for the AP, this can be adapted to specific application requirements.

For the AP on a host node, we leverage interrupts to achieve a schedule fetch requiring minimal latency. As previously mentioned, the schedule is pre-computed and simply has to be fetched. This interrupt routine returns promptly and only temporarily halts normal AP executions. However, it is important that the design preserves a correct internal state and prevents the interrupt routine from directly accessing temporal data and corrupting the data structures.
Our design achieves this using an alternating buffer which stores the most recent schedule. While such a structure allows interrupts at any time without any impact on the current calculations, it also serves as an indirection layer to prevent direct (and possibly violating) access to the underlying data structures. Furthermore, as the schedule is directly available in the AP's cache and does not require any further computations, the look-up is highly efficient [27].

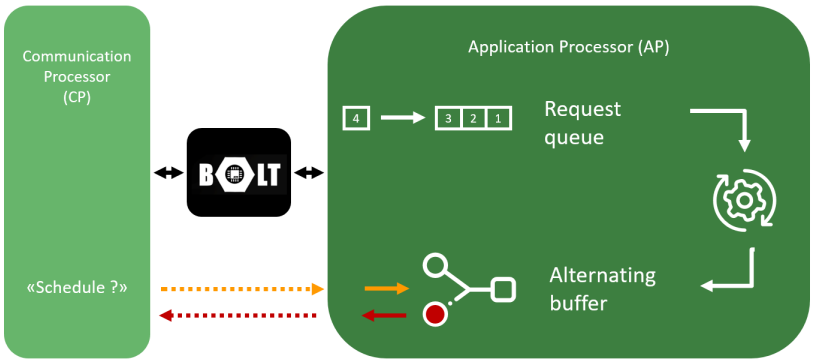

Figure 3.4: The Host's AP stores incoming DRP requests in a request queue and processes them according to its independent state machine. After the request has been dispatched, the updated schedule is written into an alternating buffer. This buffer can directly answer a subsequent schedule fetch request from the CP.

A similar concept has been chosen for processing DRP requests. An incoming interrupt simply adds the request to a request queue to keep the duration of interrupt routines to a minimum. This method always keeps the incoming BOLT queues flushed and leaves the timing of the dispatch up to the AP. The AP can then process them one after the other during its normal application loop, respecting their order of arrival and preserving the guarantee that a valid schedule will be available at any time (see Figure 3.3).

AP → CP

As the CP's communication is highly time-sensitive and does not tolerate any delay, it is of outmost importance to prohibit any interruption during the active phase. Therefore, interrupts are disabled during the transmission of the schedule and while data slots are scheduled.

During the inactive phase of the round (see section 2.1.2), the CP is in sleep mode and wakes up to read pending messages out of the outgoing BOLT queue. This guarantees that the processor is only awake when data is available (improving energy-efficiency) and that the BOLT queue is empty at the beginning of the round, as all messages have been read as soon as they arrived.

If the schedule is fetched at its standard location during the active phase (see Figure 3.3), it is important to be aware of the fact that the BOLT queue might already be partially filled. As the AP can produce new messages while the CP is communicating, the response to the schedule request might not be the only packet in the queue. Therefore, for the subsequent schedule fetch, the CP needs to send the request and immediately start flushing the BOLT queue. In the case of numerous messages, fetching the schedule may involve a limited delay larger than the round-trip time itself.

As the host is mainly concerned with *receiving* data, the possible number of outgoing messages is highly limited and primarily consist of responses to DRP requests. Those responses are upper-bounded by the maximal number of slots in the previous round and the number of scheduled control flows, limiting the actual delays. Furthermore, requests are usually processed directly after receiving them in the inactive phase and are in practise mostly already dispatched. Nevertheless, if fast responses are of prime importance, the schedule can already be fetched before the start of the round where the BOLT queue is constantly read and therefore cannot build up queuing delays.

3.4 Design choices

During the project, various choices had to be considered and evaluated. In this section, we highlight the underlying reasons behind some of the mechanisms mentioned above and show how the protocol can be adapted for specific installations by a system designer.

3.4.1 Task distribution

During the processing of a DRP request on the source and destination nodes, both AP *and* CP are involved according to the original paper [2]. However, this separation comes into conflict with the principal idea of the DPP that the CP's *sole* purpose is communication. As DRP is a control protocol and therefore resides in the application layer, this task is better suited for the AP due to its superior processing power and memory. Performing the test on the CP would require application-layer dependent state and state management which shifts the

focus of the processor and might corrupt the tight time synchronization required on that platform.
Therefore, we decided to directly calculate *all* test on the AP as proposed by the DPP. As the AP must already keep state for the DRP protocol and requires the exact same amount of memory for both CP and AP tests, this additional test adds only a negligible overhead. Furthermore, as the host is always one of the involved party in any stream and the host AP's primary function is the execution of DRP protocol in any case, such checks are precisely what the application processor is designed for. Due to its reduced energy consumption during execution compared to the CP, such a task distribution also minimizes the required power (see section 7.1 *Technical specifications*).

3.4.2 Round structure

Due to the basic principle of DRP to rely on control flows instead of contention access to send requests, the contention and stream-acknowledge slots of the original LWB are redundant. This further implies that the schedule sent at the end of the round does not depend on a stream request sent during the round and can therefore already be pre-computed. As the maximal number of new stream requests is directly proportional to the number of slots in a round, the total processing scales with the round length. To prohibit such a coupling, the straight-forward approach to dispatch requests as soon as they arrive (i.e. during the active phase) was dismissed.

Fetching the schedule from the AP occurs via an interrupt as described in section 3.3.3 *Inter-process communication*. This does not require synchronous clocks on both processors and allows the AP to process as many requests as possible and then immediately and timely deliver the required schedule to the CP when it needs it. The AP will write all arriving requests into a circular request buffer and leave the interrupt routine. If the request queue is not empty after the interrupt routine, the AP will start processing requests and calculates the new schedule using the altered stream set. After the calculation, it will write the resulting updated schedule into an alternating buffer; this procedure prevents any corruption of the internal state. Such an incident might otherwise happen in cases where the AP writes into its internal buffer during processing and therefore temporary alters the properties of the streams while an interrupt routine for fetching the schedule arrives and has to be served based on inconsistent data.

The pre-computation renders the design independent of the actual point in time where the schedule will be fetched by the CP. After a schedule fetch request has been served at the AP, the next schedule is directly computed and updated in the buffer. Therefore, the AP ensures that the CP is always able to fetch a valid schedule, even if it might not yet have been able to process all recent requests.

This makes it possible for the CP to freely choose the exact moment when it would like to receive the schedule of the next round:

- **Start of round:** As the schedule will never include requests which are received during the present round as discussed above, all the necessary information for the computation of schedule S_{i+1} is already available at the start of round i. Hence, it is possible to fetch the schedule immediately before the round starts as visualized in Figure 3.5. Because the schedule is at hand on the CP, it can be sent directly without waiting for a packet from the AP. This reduces the time of the active phase for the entire network and can therefore improve energy efficiency. Furthermore, it is possible to send only a single schedule at the beginning which already includes the current round duration so that the second schedule (see Figure 3.2) can be omitted. This further enhances the energy per data bit and allows a higher data rate for a given active phase length [3]. Such a condensation does however come at the cost of reduced reliability, as the redundancy of the original design (Figure 2.2) was stripped away.
- **End of round:** The distribution of DRP request occurrences is also a crucial factor. If one expects burst behaviour with many requests arriving simultaneously and demands to process them as quickly as possible, it might be favourable to fetch the schedule at the standard implementation (see Figure 3.3). The delayed fetch gives the AP more time to process newly arrived requests concurrently with the ongoing communication and will therefore result in a schedule which already includes more recent information. This additional computation time increases linearly with the maximal number of slots per round. In case the active phase is severely restricted, the gain would be negligible and the first option might be preferable due to its improved efficiency.

The indifference of the AP to the point in time when the fetch occurs allows for a large amount of flexibility for a system designer employing DRP, as he can choose whether to prioritise small active round lengths delivering high efficiency or more up-to-date information by providing longer computation time. If the host should perform further services than merely as a network scheduler, it is advised to use the first option (fetch before the round), as this guarantees that the BOLT queue is empty and the requested schedule can be received without further delays. Otherwise, one would have to delay the entire active round period by a complete BOLT flush time as the interrupt handling of the Bolt queue on the CP side is disabled during communication. This can lead to an accumulated queue which might contain other messages such as data packets.[3]

[3] In the current implementation, we decided to use the general case of a limited host which only serves as a network scheduler and fetches the schedule at the end of the round. This only required minor changes as it conforms with the existing LWB implementation.

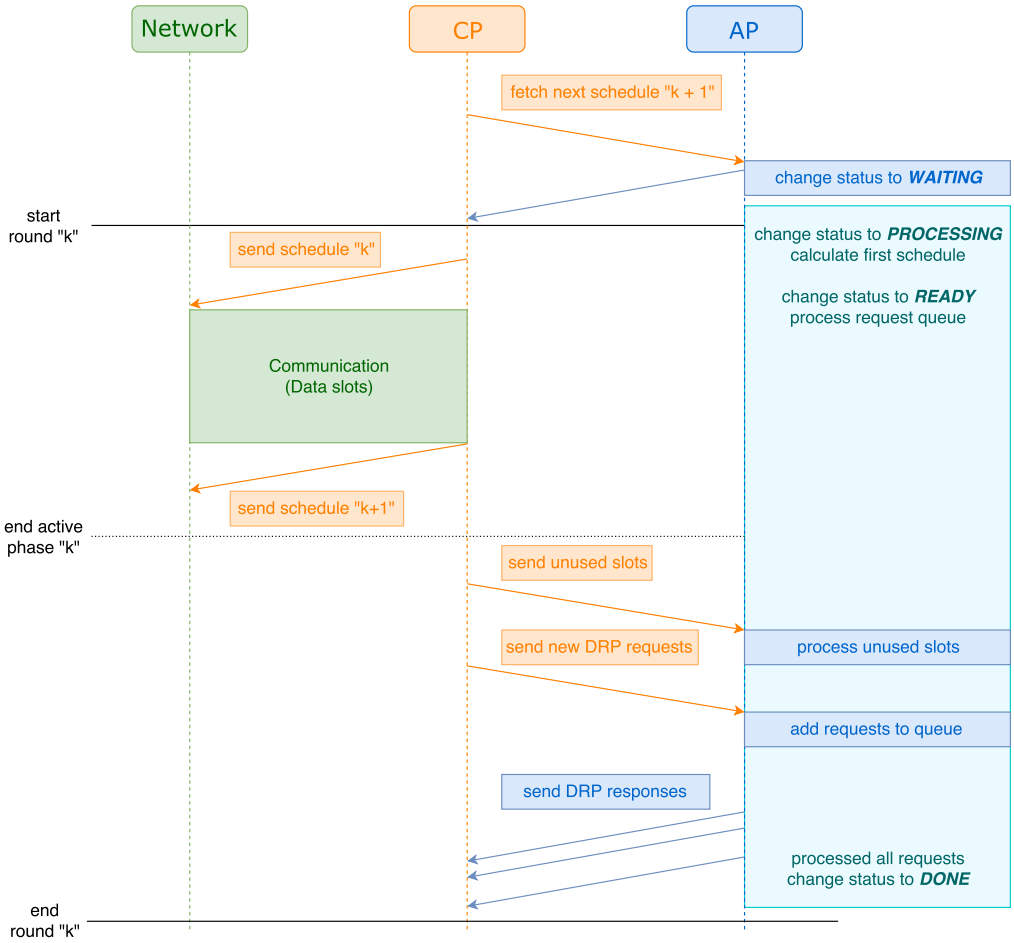

Figure 3.5: If energy consumption is an absolute priority, the schedule can also be fetched at the start of the round as opposed to at its end (compare with Figure 3.3). However, this might result in delayed request processing, as the AP has less time in-between receiving new requests and the fetch of the next schedule.

3.4.3 Interrupt handling

DRP requires flushing at regular intervals and determines such minimal periods during contract mediation. In our design, we opted for interrupt-driven message passing on the CP and the host AP, as both entities are exclusively used for executing the DRP protocol and intend to primarily limit energy consumption otherwise.

The **communication processor** is solely responsible for sending and receiving messages over the network. As it must not be disturbed during such actions, neither interrupts nor polling must occur during this period. Outside of the active phase, it would in principle be possible to employ a polling mechanism. As such constant "busy polling" prevents the system from going into sleep mode and would therefore force it to remain awake even during the entire inactive phase, such an implementation is extremely inefficient. Therefore, we chose to simply disable interrupts during critical periods, i.e. during communication, and otherwise enable them for quick and efficient flushing of the incoming queue.

The **application processor on the host** is currently only tasked with scheduling and exclusively reserved for DRP. As a timely and efficient interaction with the CP is therefore its main goal, interrupts provide an optimal way of immediately reacting to requests. In principle however, the AP could also perform other tasks; nevertheless, as schedule computation must always remain a priority, the schedule fetch would still have to be handled by means of an interrupt routine.

The **application processor on a source node** should in general use polling to limit interference between the processing of DRP packets and its normal functionality. As its tasks are very application-dependent and therefore require adaptations to the specific circumstances and scheduling requirements, the internal scheduling of the AP is not controlled by DRP. The control logic of DRP does not depend on either interrupt or polling and timing behaviour has to be examined and guaranteed in regard to the specific application.[4]

[4] At present, the implementation provides both code for polling and interrupt-based reading. For the proof-of-concept, we employed interrupt-driven delivery for all APs to simplify the code structure.

Chapter 4

Implementation

While the abstract system design has been detailed in the previous chapter, the following sections present the proof-of-concept implementation on physical hardware and demonstrate some of the practical aspects of DRP.
First, we identify the tools and platforms used to build an implementation which allows us to achieve the aims of DRP. The later sections treat the communication scheme and the internal components such as the state machine and employed data structures.

4.1 Means & methods

To implement the design on physical hardware and build a proof-of-concept system, we employed the *Dual-Processor Platform* developed at the *TEC* group of ETH Zurich [19]. The first-generation DPP features a *TI MSP432P401R* as an *application* processor and a *TI CC430F5147* for the *communication* processor. The platform uses *Contiki 2.7* [20] as an operating system and was forged from the *LWB-DPP* branch on Github [21]. The programming code has been written in the `C programming language` and builds upon previous work by Walter [22] for communication aspects and Acevedo [23] for the BLINK implementation.

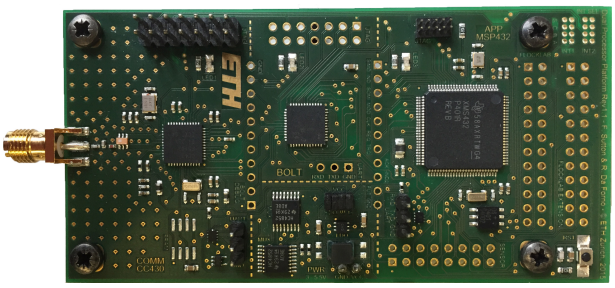

Figure 4.1: The DPP consists of a CP (left), BOLT (middle) and an AP (right) which are all placed on the same circuit board.

For the networking aspects, we used the *FlockLab* testbed [24] run by the *TEC* group [19] which consists of 30 observers and can be accessed over the Internet [25]. It provides GPIO tracing and actuation tools and offers highly precise timing information in the low microsecond range [24].

4.2 Communication structure

The communication structure and its implementation is based on work by Walter [22]. We redefined the message header definitions and grouped them for improved readability (see section 7.1 *Technical specifications*).

As described in the previous chapter, CP and AP have their own, independent task cycles. The only directly coupled interaction between the two processors happens on the *host* node during the fetching of the schedule. Using BOLT, this could still be achieved as an asynchronous message-passing scheme. Nevertheless, due to the importance of the packet, the CP cannot proceed without it and will postpone further execution by going into a low-power mode while it waits for a response.
To prevent the CP from being stuck in case the packet gets dropped due to malfunctioning of BOLT or because the AP behaves erroneously, it will automatically wake up and continue executions after a fixed time. The CP will then transmit an empty schedule with period 0, as it did not receive a valid one from the AP. For this case and as a fall-back if the scheduler should otherwise misbehave, any CP validates its received period. Such a check is critical, as a period of zero will cause the CP to set a timer whose wake-up time lies already in the past before going to sleep during the inactive phase. As this would provoke indefinite sleeping, all communication would be halted and the system could never recover from such a missed schedule. Therefore, if such an anomaly is detected, the node will automatically overwrite the period with a hard-coded duration of 2 seconds. With this identical procedure on both source and host nodes, the network still preserves its synchronization and can cope with faulty or missed packets.

4.2.1 Bootstrapping

Whereas the dependency of the CP on the AP was discussed above, we wanted to avoid any direct coupling of events in the other direction. This was deliberately avoided to prevent any preemption of user applications happening on the AP and run them asynchronously from the network and other applications. For synchronization purposes however, the AP holds its execution and waits for the *HEADER_INIT* packet during bootstrapping. This initial information is required, as its content determines the actual function of the node in the network:

Depending on the ID defined at compile time or allocated by *FlockLab*, the node either initializes itself as the *host* or becomes a normal *source* node. For this distinction, the received ID is compared to a hard-coded *host* ID.

During initialization, the memory buffers for all control streams are allocated in advance. For *source* nodes, this only concerns two streams with outgoing and incoming control information, respectively. The *host* node on the other side requires knowledge of the precise count and IDs of source nodes in the network and will allocate two streams for each of them. To achieve this, the list of source nodes is currently hard-coded just as the host ID (see section 6.2 *Future work* for suggested adaptations).
The maximum number of streams needs to be known in advance, as dynamic memory allocation can cause issues on the application processor and is not supported in the current implementation. Currently, 40 concurrent streams are supported and can be stored by the CP of any node. As all streams have the host node as one of their end-points in the current design of DRP (see section 2.4 *Distributed Real-time Protocol*), the host node experiences the maximal load and limits the scalability of the system (see section 5.1.1 *Scalability* for more details).

Before the host can allocate memory for each control stream, it needs to first check their schedulability. As these streams are treated by BLINK just as any other ordinary one, neglecting this test could cause a corrupted internal state from the beginning. However, if the control streams are not schedulable, DRP cannot function properly. Therefore, making sure that at the least scheduling all control streams is feasible is a necessary condition before any system deployment.

4.2.2 Packet types

In this section, we briefly introduce the different packets and their functionality. For a more detailed description of different header types and their corresponding definitions, see section 7.1 *Technical specifications*.[1]

All packets are defined by their *type* and have a fixed length, an exception being the *SCHEDULE* packet which varies according to the number of scheduled slots. For transmission between APs, the *type* as well as the correct *length* are stored in a prepended *HEADER* part which is only stripped away at the end-point. Notice that this header is also necessary for all BOLT communication, as BOLT might append bytes due to its SPI characteristics and return an excessive number of bytes. This given *length* parameter however is always correct and therefore offers more accurate information.

[1] All packets belonging to *BLINK* and *DRP* are implemented; however, packet types specific to *BLINK* which are not used for the implementation of *DRP* are in the following omitted to enhance the readability and are available in the original implementation paper of the first outsourced scheduler [22].

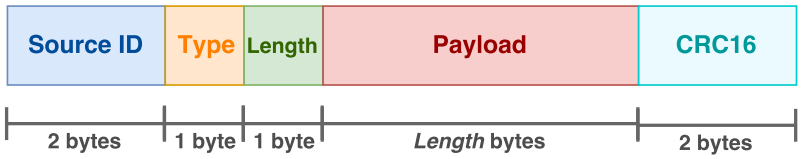

Figure 4.2: The *HEADER*, consisting of a prefix containing type & length of the packet as well as a suffix, is used for all communication between APs and is wrapped around any of the following *payload* packets.

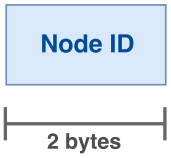

Figure 4.3: The *INIT* packet simply transfers knowledge about the current ID of the node from the CP to the AP. This information is critical, as it further defines whether a node is a host or source node at run-time.

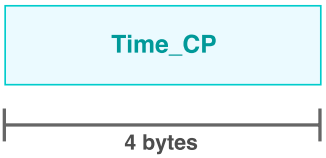

Figure 4.4: The *SCHEDULE REQUEST* packet is a feature of the outsourced scheduler and exclusively contains debug information. Its *arrival* however is important, as it triggers a reply in the form of a schedule which is sent to the CP to be distributed throughout the network.

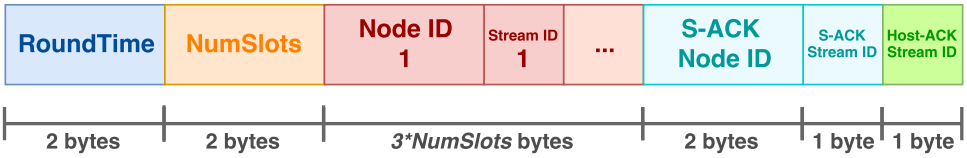

Figure 4.5: The *SCHEDULE* packet is the only one with variable length, as the number of bytes depends on the number of scheduled slots. For backward compatibility with BLINK, the *stream* and *host-stream* acknowledgements are still sent correctly.

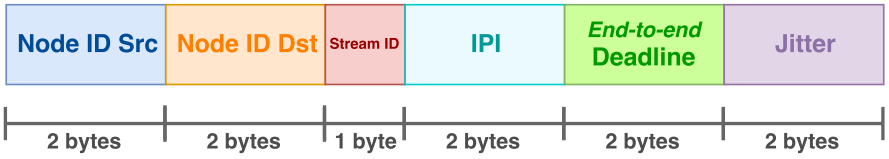

Figure 4.6: The *DRP REQUEST* packet contains all information included in the contract. Note that only the *end-to-end* deadline is sent, as the *network* deadline can be calculated deterministically from the given values.

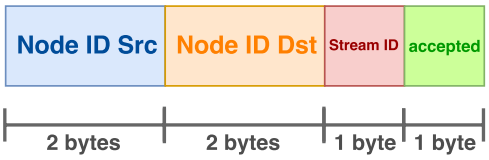

Figure 4.7: The *DRP RESPONSE* packet sends either an ACK or NACK back to the source of the request along the same path as the request. Notice that it might be that this packet arrives after the new stream was already scheduled.

4.2.3 DRP Request & Response

The original DRP paper [2] introduced the notion of requests which are sent from *source* over the *network manager* to the *destination* (see Figure 2.5). As the network manager resides physically on the *host* node which is always one of the two end-points of a stream, the request packet (see Figure 4.6) only visits two physical nodes and is always only sent directly from source to destination.

After the request has been dispatched by the destination node, the knowledge about the agreement of source, network controller & destination to accept the stream needs to spread to all involved parties. For this purpose, a response packet (see Figure 4.7) is sent back, following the reverse path of its corresponding request to reach all involved components. Those components will then fulfil their contractual obligation by setting flush times accordingly and allocating memory. In practice, it is beneficial to execute such a step already before receiving a definite answer, as this makes sure that all guarantees can already be kept for the very first packet.
In the case of a negative decision, such a definite response can already occur at an earlier stage as soon as it is rejected and does not have to propagate through all stages required for acceptance, i.e. up to the destination node.

If the *network manager* happens to reside at the destination, it is the first one to know about an accepted stream and might already start scheduling it before the corresponding control stream could transport the response packet to the source node. Therefore, a stream is already implicitly acknowledged when it is included in a schedule and the source node will react just as if it already received the response.

Important: *Network* deadlines which are included in a request **must** be a multiple of the round time in order to use BLINK. Otherwise, BLINK's assumptions [3] are violated and scheduling can result in erroneous behaviour, as the given guarantees cannot be kept anymore. Therefore, the network scheduler will always floor the calculated network deadline to a multiple of the round time.

4.3 Embedded Systems programming

As platforms in embedded systems possess only limited processing power and memory, it is important to design the system under the given constraints and keep implementations efficient. In our case, it was especially important to optimize the potentially time-consuming tasks such as the schedule computation and the admission test. As LWB is time-triggered (see section 2.1 *Low-power Wireless Bus*), those functions need to return within given bounds as the computations underlie internal hard real-time constraints and are useless if they do not finish in a given time.

4.3.1 Data structure

For the implementation of the BLINK scheduler, we employed bucket queues as proposed in the original paper [3] for efficient calculations of the schedule and the admission test. To avoid a complete deep copy of the state and decrease memory requirements, the notion of *truncated streams* was taken from an existing implementation of BLINK [23] to allow temporary alterations of state data.

Size of storage

While the CP relies on a *16bit* SoC [26], the processor on the AP is based on a *32bit* architecture [27]. Therefore, most local variables are defined as *32bit* integer values for fast computation; for any other size (even smaller ones), the processor needs to convert them first internally to 32bit values before executing the instructions, resulting in very inefficient computations. As those variables have a limited scope and are only used directly during the function execution on the stack, the involved memory overhead is negligible.

For the storage of non-volatile data such as the internal state of the scheduler and the involved streams, memory is the main constraint on the scalability of the system. Thus, we optimized the implementation towards efficient memory usage and only allocated the actually required variable sizes. To keep the timing accuracy of the design high, we opted for 16bit integers to store relative timing information in milliseconds. As such a memory block can only store up to 65535 different values, all periods, deadlines and jitter parameters have to be kept strictly shorter than this maximal value to prevent buffer overflows (i.e. in numbers less than 65'000 milliseconds). Notice that for absolute timing information such as the start time of a round, we require the higher capacity of *32bits*, as such values can easily exceed 65535 if the application duration exceeds one minute.

The packets (see section 4.2.2 *Packet types*) are designed to preserve this accuracy even when sending values over the network. While we utilize similar packets as in [22] where the storage was overdimensioned for *seconds* precision, we now use the full potential of *16bit* integers and send information with a high accuracy in the *milliseconds* scale. This preservation further facilitates the implementation and makes it less prone to programming mishaps, as the same time scale is used throughout the system. If parameters with a duration of over a minute should at some time be required or the packet should be shortened, one can switch back to *seconds* precision for the transmission over the network at the cost of accuracy.

All state-related memory allocation occurs during the initialization and cannot utilize dynamic memory allocation, as such an option is not supported on the current hardware. Therefore, the maximal number of streams, buckets and concurrent incoming requests needs to be known at compile time and is fixed.
While all incoming requests are added to a queue of fixed size and therefore allow multiple simultaneous requests, each node only possesses maximally a single pending outgoing request. This also counts for the host node and therefore does not require separate memory allocations depending on the role of the node. As a host only rarely requests new outgoing streams due to its function as a gateway for all source nodes (see section 3.1 *Problem setting*), it is not necessary to include the option of sending multiple requests simultaneously from host to source nodes at any given time.

State conservation / transparency

The BLINK scheduler (see section 2.2 *BLINK*) does not integrate the concept of schedule pre-computation in the original design. Therefore, the internal state of the scheduler is permanently affected each time such a schedule is assembled and time counters are iterated by the calculated period. This effects both global variables such as the start of the round as well as local ones such as each scheduled stream because their release times are altered.

In DRP, multiple schedules for the same round may be calculated when the stream requests are processed, as each such request may alter the demands on the current round and changes the stream set. Therefore, the schedule is updated to prevent stale data in the buffer. After each *DRP request* is processed, a new schedule is written into the alternating buffer (see Figure 3.4). During the allocation of the slots, the bucket queue and the state of the stream set are altered for scheduled slots as their release times have been modified. Because this schedule might not actually be fetched from the CP and the subsequent requests require to see the same state as before to maintain *transparency*, those changes must be reverted immediately to prohibit lasting inconsistencies.

A simple and save implementation to achieve such transparency is the creation of a deep copy of the entire bucket queue and the stream states before each request is processed. As this method is not scalable for large numbers of streams and performs poorly in terms of memory efficiency and processing power, we opted to use an alternative variant.
By leveraging that only the streams which are scheduled see a change in their permanent properties (as all other changes during admission tests and scheduling occur on the temporary, truncated data structures), we use the produced schedule itself to revert the transformation. By retracting the shifted release time and restoring its place in the bucket queue accordingly, every stream will again take its former place. Therefore, for subsequent stream requests, the entire state seems just as before the current processing took place.

Care has to be taken when a valid schedule is finally fetched by the communication processor. As any changes which occurred during the calculations were reverted as described above, the state of the actually scheduled streams must now again be updated. Therefore, the previously reverted changes have to be re-applied to take effect. While this might seem redundant, such a process is necessary to preserve uncorrupted state information as the application processor is completely oblivious to the point in time when the next fetch will occur.
To prevent concurrency issues from happening, the interrupt routine will simply store the fetched schedule locally and notify the application processor to handle the changes after it has finished with a potential request handling. Any schedule computed during the schedule fetching for the old round might still contain invalid (as stale) data and will therefore be discarded upon completion in any case. The handling of the state adjustment outside of the interrupt routine furthermore guarantees that the schedule fetch can return without any delays dependent on the current situation on the application processor.

The above mentioned techniques guarantee that state is preserved at all times and cannot be corrupted. They additionally enable the timely execution of the communication rounds, independent of any platform parameters such as the length of the bucket queue which affect the duration of the state corrections. As mentioned in section 3.3.2 *Round structure*, this keeps the active part of the round as short as possible and minimizes energy requirements.

Array handling

In the current design of DRP, it is not possible to revoke a stream and release its resources. However, the implementation presented in this thesis already incorporates the required functions fur such an extension and allows for the deletion of streams both in the DRP and the BLINK stream state. As the stream set is stored in an array of fixed length (and not e.g. a linked list), deletion will result in "holes" in the used part of the array where streams were previously stored.

An array can be managed in two fundamentally different way:

- Always preserve a cohesive *data part* without any gaps and separate, subsequent *free space* where new streams can be added. Such an implementation guarantees that every array cell in the data part contains a valid stream and simplifies iterating through the stream set. However, it requires that every time a stream should be deleted from the memory, the last used element of the array must be copied to its place in order to keep consistency. Furthermore, as the bucket queue relates to absolute addresses inside the array, it might be that pointers at various locations will point to wrong stream entries after an unexpected "delete" operation, which further increases the system complexity and requires extensive state corrections.
- Keep filling the array and adjusting a pointer to the next free cell accordingly (pointer *free_stream_id*). With this second implementation, we have to potentially iterate through the entire array in order to find a given stream as streams can be spread throughout the address space. To prevent this, the method is optimized to only search maximally until all valid entries were visited and then abort. As deletions will only occur rarely and therefore data is still rather cohesive, such a search will not perform significantly worse than the first proposition. Nevertheless, as such stream searches have to be performed once every schedule computation and an additional two times for a potential admission test, those operations can be rather expensive. Furthermore, they have to be executed every round, whereas former implementation only involves a one-time investment of copying the data.

We chose to use the latter option due to memory write issues. On certain hardware, writing into memory takes orders of magnitude longer than reading (see [23] on SRAM). As deletions are currently not used and we do not expect a large number of occurrences in future projects, such an implementation is simpler and does not require the error-prone alteration of various pointers distributed throughout the bucket queue.

4.3.2 State machine

While the CP follows a time-triggered schedule according to the LWB round (see section 3.3.2 *Round structure*), the AP is decoupled from the communication. On the AP of a *host* node, a *state machine* defines the task routine and current calculations. Its primary function is to guarantee that a pre-computed schedule is always available and that the request queue is processed if enough time is available. Due to its reliance on states, each event such as interrupts triggers clearly defined functions depending on the current situation.

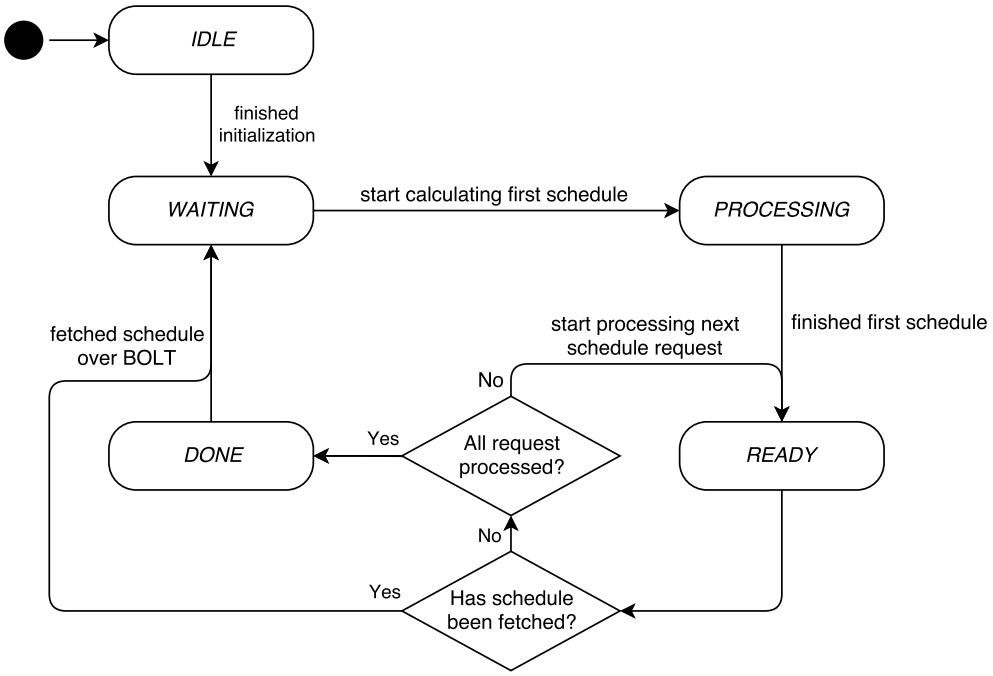

Figure 4.8: The state machine on the application processor consists of four primary states which are passed one after the other in a cyclic fashion.

As can be seen in Figure 4.8, the system is primarily in either one of four states:

- **WAITING:** Residence in this state indicates that no valid schedule has yet been computed. This might be either because the system just started or signals that a schedule was just fetched for the previous round and a new round has started. It is of outmost importance that the processor only resides in this state for a short time (e.g. to finish up ongoing request dispatching) and immediately starts the computation of the schedule for the next round.
- **PROCESSING:** This state signals that the processor is currently calculating a new schedule, but did not yet finish a valid scheduling for the current round. It is mainly used for debugging purposes, as an interrupt for schedule fetching arriving in this state indicates that schedule computation either started too late or exceeded the maximal computation time.
- **READY:** While there already exists at least one valid schedule which is available in the alternating buffer, it might be that the request queue is still filled and multiple stream requests are waiting to be processed. The AP simply dispatches them one after another as long as the pre-computed schedule is not fetched yet. At the *end* of each dispatch, i.e. after writing

the updated schedule into the buffer, it will check whether it still resides in the *READY* state; if an interrupt occurred in the meantime, the state will have switched to *WAITING*. If no such interrupt emerges, the system eventually signals a completely empty queue by continuing to the next state.[2]

- **DONE:** Similarly to *PROCESSING*, this state primarily serves for debugging and is only observable by the interrupt handler. It is the final state of any round and symbolizes that all requests were dispatched and the schedule for the next round is readily available in the buffer. As there are no more actions to be conducted in this state, the processor will go into a low-power mode and wait for the interrupt handler to wake it up again when new information arrives.

The transition between states can occur either due to processor-internal events (e.g. the start of the first schedule computation between *WAITING* and *PROCESSING*) or is triggered through external events signalled by an interrupt (e.g. that a schedule was fetched by the CP for the transition from *DONE* to *WAITING*). Nevertheless, such outside parties will never directly affect computation with immediate effect, but simply serve as input to the state machine which controls the behaviour of the AP. Therefore, the AP remains in full control of its actions and preserves its asynchrony.

[2]While the state machine will return to *WAITING* after a schedule was fetched, it does so without preempting the on-going admission test of a request to prevent state corruption. If the admission tests should take an unexpected long time due to a complex stream set, it might be advisable to implement immediate preemption directly from the interrupt handler of the schedule fetch.

Chapter 5

Evaluation

To analyse the real-world performance of the developed system, we tested the implementation on the hardware platform as described in section 4.1 *Means & methods*. In the following, we present both local evaluations directly measured at the physical node and tests over an extensive network called *FlockLab* [24]. Our investigation shows that DRP can keep all its guarantees and scales well. We demonstrate that even under high loads, the system delivers all packets according to their deadlines and provides quick responses to stream requests.

5.1 Local tests

While the main performance parameters are only observable over a network of multiple nodes, it is still important to consider the capabilities of the individual nodes and detect possible hardware limitations. For this, we investigate the maximal number of streams which can be stored on the processors and inspect the duration of a schedule computation on the AP.

5.1.1 Scalability

The number of streams which can be stored is limited by available memory. As the memory is allocated during initialization and cannot be dynamically requested, the current hardware does not enable us to redistribute memory space on the fly and adapt to the current circumstances. This would be especially favourable on the AP, as both the array for storing the state of the streams and the request queue are contending for memory.

The sizes of buffer queues and array cells used for an individual stream or stream request varies for both CP and AP, as they store different data. While the AP conserves the streams properties previously received through a contract, the CP

Memory size	**Communication Processor** (CP)	**Application Processor** (AP)
Total RAM	4000	64000
Available RAM	2400	45000
Used per stream	23 + *packetSize*	45
Used per request	0	15

Table 5.1: The amount of available memory was directly retrieved from the image which is uploaded to the processors and is given in **bytes**. The number of streams which can be stored on a node strongly depends on the size of the packets and the required request queue length.

only prepares an outgoing buffer so that packets read *in serial* from BOLT are available *in parallel* for sending over the network [22].

We observe in Figure 5.1 that the AP can be strained without major restrictions even for large networks and hundreds of streams. The number of streams on the CP however drops below ten if the maximal BOLT packet size of 256 bytes is fully utilized. By extending the available RAM using external memory of 128 kBytes, this restriction on the CP can be avoided. Therefore, the current hardware generation is already capable of multiple hundreds of streams even when sending packets with maximal length.

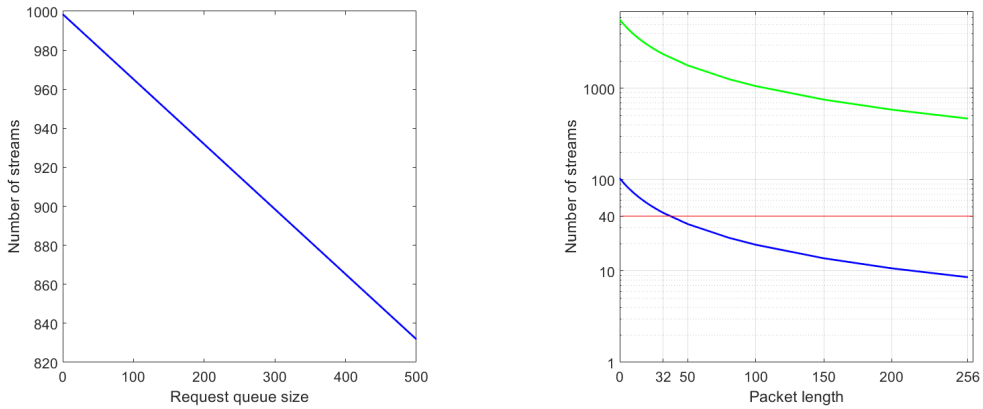

Figure 5.1: On the left, we see that the AP is only marginally influenced by the size of its request queue in the number of supported streams. On the right, it becomes clear that the current implementation (blue) can only store fewer than 40 streams for reasonably sized packets, whereas an extension (green) lifts this limitation. The red horizontal line represents the current design parameters (32B messages, resulting in a maximum of 40 streams).

5.1.2 Blink scheduling

The duration of the slot allocation is highly deterministic and does not depend on the traffic requirements of the streams, but only their number. However, the computation of the start time of the next round is complex, as it must compute the minimal *slack time* available for the system (see section 2.2 *BLINK*).

The length of the schedule computation is not directly influencing system performance, as we decouple the computation and the fetching of the schedule (see section 3.3.2 *Round structure*). The buffering of the schedule does however only function if the scheduler is already pre-computed and readily available. Therefore, keeping the computation time below the duration of a round is critical for correct functioning and for guaranteeing that deadlines are met.

The computation time of the schedule strongly depends on the given stream set. Due to the underlying concept of the *minSlack* computation, even a large number of streams with a similar period can take only a small time to compute, as the *synchronous busy period* is small; however, if one uses a small number of streams with periods consisting of differing prime factors, this can increase exponentially [3]. The companion report to the original BLINK paper [3] offers a detailed test setup and includes a reasonable stream set. Therefore, such a stress test of the scheduler needs to be performed with care in order to yield meaningful results and must include various streams of different periods.

We have striven to measure the duration of the schedule computation under various levels of utilization. For this, we simulated a fixed number of admitted streams and adjusted their periods so that they would total in the desired total utilization. The stress testing is fully implemented, readily available in the code base and can be easily adjusted for the number of streams and the step size in-between utilization levels. However, due to an issue with using more than a certain percentage of the AP's memory on the current platform, it was not possible to observe the desired measurements.

5.1.3 DRP Request processing

In contrast to the duration of the schedule computation, the local processing time of a stream request is only of minor importance as the AP can take its time and does not have to be finished within a certain deadline. In theory, such a computation could even occur over multiple rounds for extremely complicated stream sets. In practice however, we measured a maximal computation time of 130ms for a stream set of 30 streams used later-on in the networking tests (Table 5.2). With a minimum processing time of 30ms and a median time of 54ms, we are however clearly below that value for most computations.

To obtain numbers representing the actual responsiveness experienced by the source nodes, we must consider the entire process of sending the request over the network using the control streams and then receiving a reply from the host in the returning stream as done below instead of only focusing on the local performance.

5.2 FlockLab

5.2.1 Setup

Our DRP implementation was tested on *FlockLab* [24]. We employed 11 nodes distributed over an area of 1'000 square meters (see Figure 5.2). The nodes consist of multiple first-generation DPPs and can be accessed and configured over the website [25]. Both *Serial* output and GPIO traces can be observed and stored for post-processing and analysis of the experiments.

The *Serial* output includes a timestamp created by the infrastructure itself which offers highly precise timing accuracy in the range of microseconds [24]. We leverage these records to generate exact and tightly synchronized measurements of run-time latencies and round-trip times.

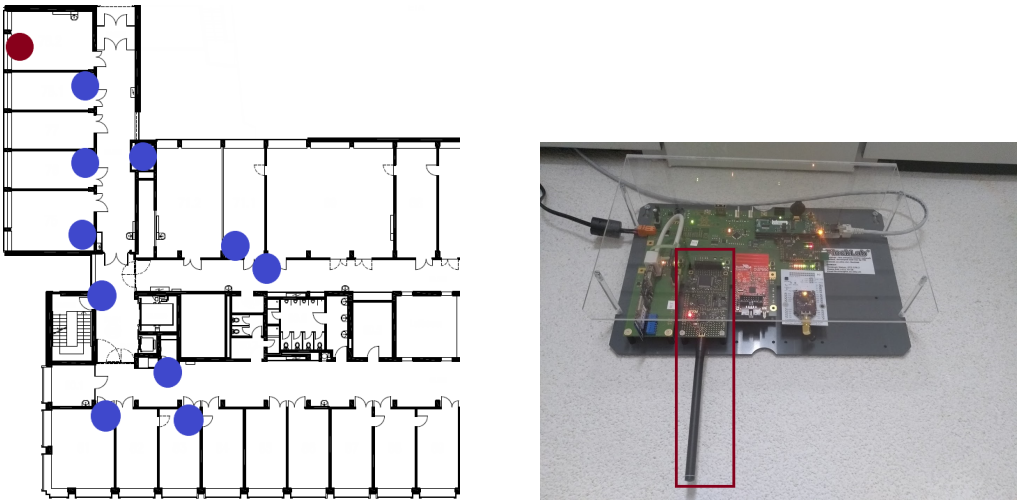

Figure 5.2: The *FlockLab* nodes are distributed throughout the *ETZ* building of ETH Zurich. On the left, we see the host (red) and the 10 source nodes (blue). The right figure shows a single node with the *DPP* framed in red.

5.2.2 DRP Protocol

As traffic demands change and source nodes need to adapt their streams, it is important to investigate the responsiveness of the host to such requests. While control streams should be scheduled as often as possible to decrease the involved latency, the control overhead and therefore resulting utilization drop for data packets results in a trade-off between efficiency and responsiveness.

Processing time

The processing time of the request itself on the host's AP is negligible in comparison to the network delay, as can be seen by comparing Table 5.2 and Table 5.3. It is interesting to see though that the processing time is rather stable when using the tested stream set and only varies by a factor of maximally four. Therefore, a source node can reliably reckon when its processed request should again be available for the returning trip through the network and estimate the resulting latency.

	Low util	**Medium util**	**High util**
Utilization	41%	60%	92%
Median time	78.1	75.8	54.5
Minimum time	29.8	32.2	30.0
Maximum time	126.4	130.7	113.2

Table 5.2: Characteristics of the processing time of DRP requests on the host AP in **milliseconds**.

Response time

The most significant figure for any control application is the latency between *action* and *reaction*. For DRP, it is crucial to provide a minimal time span between sending a request from a source node and receiving the definite reply from the host. This is of particular importance for wireless networks, as network links fluctuate and interference with other signals can vary strongly over time. Only by delivering control information on time and adapt the control streams to the environment can a feedback loop in a CPS (see section 1.2 *Motivation*) remain stable and be used in an industrial setting.

The management information in the control streams also requires resources and allocated slots which will not be available for sending data traffic over the network. Therefore, a trade-off between quick stream registration and high data rates needs to be considered. We wish to minimize the interference of the control streams with the data streams by slightly prioritizing the data streams.
One way to achieve this while still guaranteeing all deadlines is by setting the *end-to-end* deadline of a control stream particularly high while still keeping a reasonable period. This gives the scheduler the freedom to fill up schedules including unused slots with control streams by leveraging the high *network* deadline without additional energy overhead or occupying slots which could have been used for data. Furthermore, we can capitalize on the shorter period of the data streams to send control streams long before their own deadline is reached. Therefore, we can even have round-trip times shorter than one *end-to-end* deadline without paying any extra costs.

Empirically, we observe that as seen in Table 5.3, the maximal latency between sending and receiving a packet is around three times the period of the control streams for the given stream set. We also find that the return trip time from *Host* to *Source* can be very fast if the incoming and outgoing streams are correctly aligned to quickly send the response after it has been computed without having to wait for another period (see section 5.3 *Further testing*).

	Low util	**Medium util**	**High util**
Utilization	41%	60%	92%
Network S-H min	18.27	18.25	17.21
Network S-H max	34.27	33.21	33.17
Network H-S min	4.26	1.02	1.05
Network H-S max	14.87	14.87	15.96
Total min	23.58	23.60	23.58
Total max	47.09	34.29	34.27
Total max / period	3.14 x	2.29 x	2.28 x

Table 5.3: Characteristics of the latency on the network between sending and receiving requests and responses in **seconds**. "S-H" represents the *Source* to *Host* path and "H-S" the one from *Host* to *Source*.

5.2.3 Packet delivery

To measure the arrival of data packets and compare the delivery time with the corresponding *end-to-end* deadline, we sent 150 packets from each of the ten source nodes to the host. During transmission, we experienced a common drop rate of below 5%; for the formidable reliability of 99.9% arrival rate promised in the *Glossy* paper [13], we would require a denser network and higher transmission power. However, we also noticed very bursty packet drops. If packets were lost, we observed a high probability that following packets were also affected. This can happen due to temporal external influences and represents the typical behaviour of wireless packets which are usually correlated over time [28].

	Low util	**Medium util**	**High util**
Packets sent	1200	1200	1500
Packets received	1146	1142	1464
Packets dropped	54	56	36
Drop rate	4.5%	4.8%	2.4%

Table 5.4: All source nodes generated 150 packets each to send to the host. The discrepancy in the first two tests leads from a (different) pair of nodes whose request got either dropped on the path to the host or whose response was dropped on the way back.

End-to-end deadline comparison

We observe that all of the nearly 4000 packets which arrived at the destination were delivered before their deadlines and therefore met their traffic requirements. In fact, they only used a maximal percentage of 65% of the *end-to-end* deadline as seen in Table 5.5 which shows that the worst-case assumption of DRP is overly pessimistic in the general case. However, performance can only be guaranteed if the protocol is still successful even under the most severe circumstances.

The grouping of the packet arrival times at the beginning of the histograms in Figures 5.3 can be explained with the short period of the data streams compared to their *end-to-end* deadline. As the *network* deadline is maximally as long as the period and local delays between AP and CP are generally small, the packets can be delivered early most of the time. Furthermore, it might be that the control streams trigger the scheduling of a round, which can then be filled with already available data packets and therefore lowers the latency of the packets.

	Low util	**Medium util**	**High util**
Utilization	41%	60%	92%
Mean latency	13.7%	8.4%	4.4%
Median latency	14.7%	7.4%	4.1%
Minimum latency	1.1%	1.1%	1.1%
Maximum latency	58.2%	65.4%	40.7%

Table 5.5: Characteristics of the *end-to-end* latency of the data packets in percentage of the analytic bound.

5.3 Further testing

In this thesis, we address all primary objectives and demonstrate that the implementation works as expected. Nevertheless, there is still room for further investigations. For future work, we propose to analyse the following matters:

- **Alignment of control streams:** It would be interesting to test how separating the start times of control streams from *Source* to *Host* and the ones in the reverse direction impact and possibly benefit the round-trip time. By scheduling the streams from *Host* to *Source* in a manner such that the control stream with the request just arrived in the round before and could deliver a potential request, we can return with the response as fast as possible and therefore reduce latency.
- **Scalability:** Large stream sets can dramatically increase computation time for both DRP request processing and the BLINK schedule computation. This needs to be analysed so that suggestions for appropriate LWB round times can be given for a fixed maximal amount of streams (e.g. by showing that a set of Z streams can maximally result in a schedule computation time of X, wherefore a minimal round time of $X + Y$ is required).

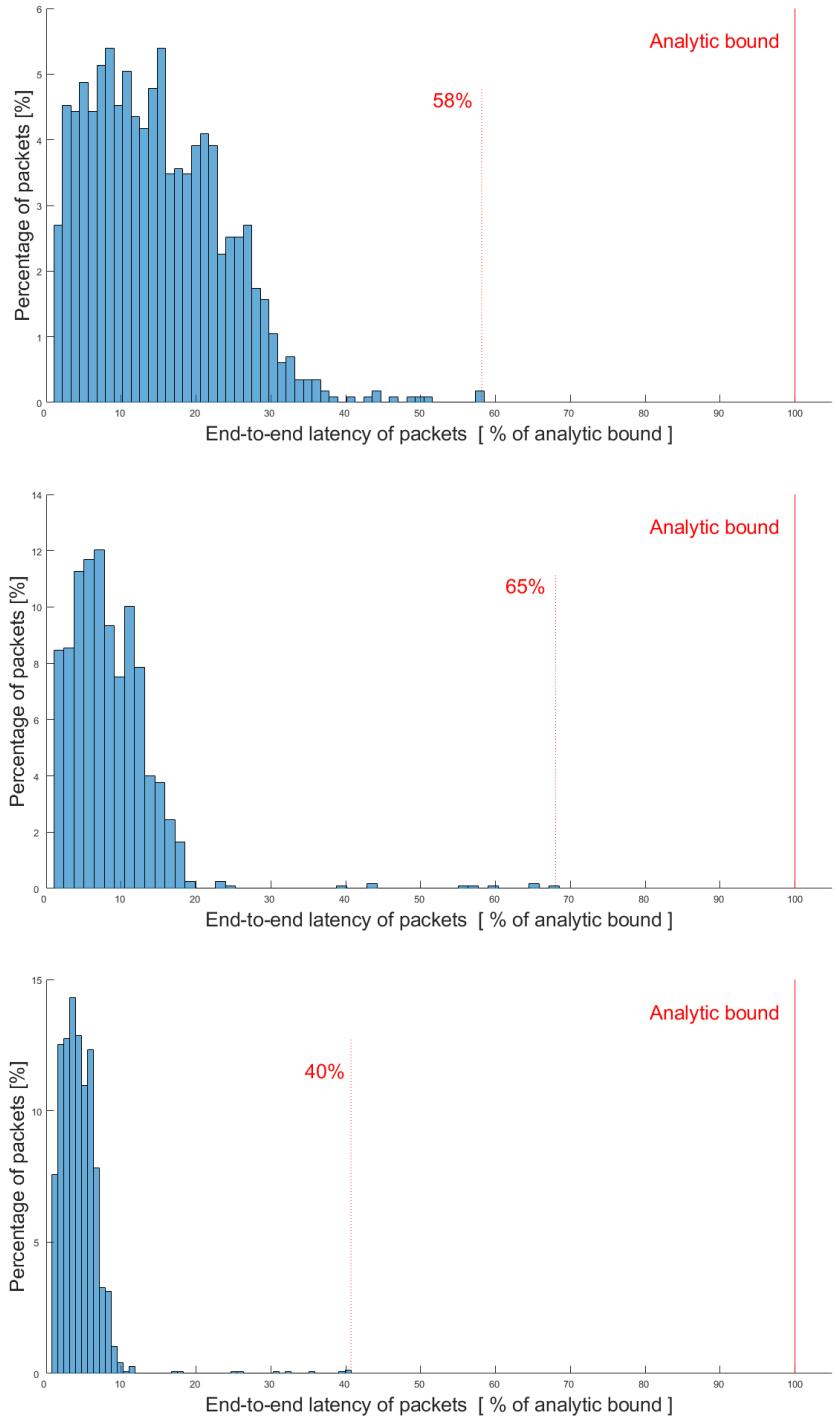

Figure 5.3: Low (top), medium (middle) and highest (bottom) utilization results in different packet distributions. For the presented figures, we exclusively regarded the data packets and did not include the control packets.

Chapter 6

Conclusion & future work

After presenting the underlying system design in *Chapter 3* and detailing the construction of the implementation in *Chapter 4*, we then showed in *Chapter 5* using networking tests on *FlockLab* that the implementation is sound and meets all *end-to-end* deadlines. In this chapter, we conclude the thesis and give some outlines on how one could extend the functionality of DRP in the future.

6.1 Findings

In this thesis, we implement the *Distributed Real-Time Protocol* (DRP) [2] on a custom platform called *Dual-Processor Platform* (DPP). We alter the round structure of LWB [4] and constrict the entire communication to deterministic TDMA access. Furthermore, we leverage BOLT [1] to decouple communication and application tasks. By using an internal request queue for DRP requests and pre-computing the round schedule in advance, we reduce the interference of the communication on the network scheduler to a minimum and therefore separate the scheduler from the network.
Our tests show that DRP guarantees that all packets which arrive at the destination meet their *end-to-end* deadlines. The network manager can respond quickly to DRP requests and can serve up-to-date responses with a total maximal latency of three periods of the control streams. Even under a very high utilization of up to 92%, the system performs well and delivers packets on time. Furthermore, the system can easily scale to larger networks. The *host* node, bearing the maximal load, is capable of sustaining multiple hundred streams in its memory and can act as a centralized controller, as it possesses an overview over the entire state of the network.

6.2 Future work

Currently, information about unused slots is received at the application processor in a special packet but is not further used for exclusion of admitted streams. Implementing this would require a *DRP Cancel* mechanism to notify the source that its stream will be cancelled due to inactivity. Furthermore, the specifications currently allow a source node to maximally send at a given rate but do not enforce a minimal frequency in order to reduce energy consumption if there is no data to be sent. Therefore, the contracts in-between devices would have to be extended to allow for one- or both-sided deregistration of a given stream, voluntarily or enforced by the host. For the deregistration, functions for deleting streams at the DRP and Blink layer are already implemented.

Another extension of DRP could include the reduction of restrictions on the role of the host. While in the current design (as detailed in section 2.4.3 *System level*), the host must be one of the directly involved parties, scalability could be improved by allowing source nodes to communicate directly with one another and skip the host as a relay node. As the network scheduler does not have to be on the path between source and destination and is logically completely decoupled from the network, this should be feasible but requires modifications in the registration mechanism.

Because the round structure decouples the application and communication processor, it is possible to adapt and extend the APs functionality. As an example, it might be preferable to send specifically urgent *DRP requests* before fetching the schedule so that those streams might already be scheduled in the next round. Such requests could simply be inserted at the beginning of the processing queue and could trigger an automatic Bolt write after admission and schedule computation, thus requiring only minor changes.

One shortcoming of DRP is its limitation to a fixed set of nodes known at compile time. This restriction could be lifted by integrating a bootstrapping phase where nodes might register themselves using a contention slot. Furthermore, it would be useful to allow such slots at regular intervals to further increase the flexibility and allow nodes to join the network even during steady-state communication.

Chapter 7

Appendix

7.1 Technical specifications

7.1.1 Hardware

We used the *Dual-processor platform* (DPP) developed at the TEC group of ETH Zurich. It contains an MSP432 [27] as an application processor, a CC430 [26] as a communication processor and BOLT [1] as an interconnection to enable asynchronous message passing as described in section 2.3.2 *BOLT*.

	Communication Processor (CP)	**Application Processor** (AP)
SoC Name	TI CC430F5147	TI MSP432P401R
Processor Type	ARM 16bit Cortex M0 (RISC)	ARM 32bit Cortex M4F
Max frequency	20 MHz	48 MHz
SRAM	4 kB	64 kB
Flash	32 kB	256 kB
Active energy consumption	160 uA/MHz	80 uA/MHz
Sleep mode consumption	1 uA	25 nA
Additional features	Low-power wireless communication	Floating-point unit (FPU)

Table 7.1: Technical specifications of the two processors used for the thesis.

	BOLT
SoC Name	TI MSP430FR5969
Processor Type	ARM 32bit Cortex M4
Max frequency	8 MHz
FRAM	64 kB
Inactive energy consumption	430 nA
Throughput	1.5 - 3.3 Mbps
Message length	16 - 128 bytes

Table 7.2: The BOLT interface sitting between AP and CP is responsible for asynchronous message passing.

7.1.2 Existing code bases

The algorithms for Blink are built as described in the original paper [3]. The code in *"blink.c"* and *"blink.h"* is inspired by a previous implementation of Blink on CC430 sensor nodes [23]. Unlike the *CC430*, the *MSP432* used in this thesis offers a much larger internal memory and does not require access of additional, external memory units.

The communication is built upon previous work by Walter [22]. He introduced the notion of multiple streams of a single node onto the platform and extended the original LWB structure by fetching the schedule from an external scheduler. However, in his works, the application processor is solely responsible for relaying the request to a *Matlab* scheduler on an external computer. *"stream_centered_output_buffer.*"*, *"defines_outsource_sched.h"* and *"sched-outsource.c"* as well as multiple changes in *"lwb.c"* have been used and adapted in the present thesis.

7.1.3 Header types

In *structure.h*, the different header types and their corresponding packet structure are described in detail. The table below lists the header types and their corresponding MACRO definitions.

Decimal	Abbreviation	Byte size
1	HEADER_INIT	1
2	HEADER_DATA	X
10	HEADER_BLINK_REQUEST_SCHED	4
11	HEADER_BLINK_REQUEST_STREAM	13
12	HEADER_BLINK_REQUEST_HOST	9
13	HEADER_BLINK_DELETE_STREAM	3
14	HEADER_BLINK_UNUSED_SLOTS	X
15	HEADER_BLINK_SCHEDULE	X
20	HEADER_DRP_REQUEST	11
21	HEADER_DRP_RESPONSE	6

Table 7.3: Header types and their length (X: size varies)

7.2 Detailed code structure

On the **application processor**, the central file is "*structure.c*" which covers all interactions between BOLT and the applications and further includes testing and debugging functions. "*drp_ap.c*" provides all DRP features and stores the entire state. "*blink.c*" is used both directly from *structure.c* for legacy reasons to access BLINK functions and as a sub-component of *drp_ap.c*.

The **communication processor** runs its main application from "*drp_cp.c*" which then calls "*lwb.c*". The entire scheduling-related tasks are handled in "*scheduler_outsourced.c*".

7.2.1 Debugging information

Functions for testing the entire chain are available both at CP and AP. At CP, "drp_cp.c" offers the option to emulate both DRP requests and responses which can then either be sent over the network or directly to its local AP. The AP features this functionality in "structure.c" and offers a function for testing every single packet functionality (locally), a test of DRP request and response over the network and code to produce data packets for checking the compliance with deadline guarantees.

It might occur that the Debugger on the AP is stuck right after launching and will never display the code. One way to solve this problem is to "Disconnect" and

"Reconnect" again (inside *Code Composer Studio*). Afterwards, the debugger will work correctly. Known causes for such behaviour is the allocation of arrays at run-time (so-called *Variable Length Arrays* (VLAs)) and the initialization of arrays directly at the definition (`uint8_t arrayname[] = { 0 };`).

If the debugger is "jumping around" and appears to not execute lines according to the specified order, it is strongly adviced to disable multi-line macros (the most prominent one being *DEBUG_PRINT*) so that the pre-processor does not change the line numbers before compiling. Furthermore, it is adviced to disable "Optimization" in the project properties or set it to a low level (e.g. "0 - Register Optimizations"). The lower optimization level is furthermore helpful if multiple variables are changed simulaneously, as they might be mapped to the same memory register.

Bibliography

[1] Sutton F, Zimmerling M, Da Forno R, Lim R, Gsell T, Giannopoulou G, Ferrari F, Beutel J, Thiele L. *Bolt: A stateful processor interconnect.* Proceedings of the 13$^{\text{th}}$ ACM Conference on Embedded Networked Sensor Systems, November 2015.

[2] Jacob R, Zimmerling M, Huang P, Beutel J, Thiele L. *End-to-end real-time guarantees in wireless cyber-physical systems.* Real-Time Systems Symposium (RTSS), IEEE, November 2016.

[3] Zimmerling M, Mottola L, Kumar P, Ferrari F, Thiele L. *Adaptive real-time communication for wireless cyber-physical systems.* ACM Transactions on Cyber-Physical Systems, February 2017.

[4] Ferrari F, Zimmerling M, Mottola L, Thiele L. *Low-power wireless bus.* Proceedings of the 10$^{\text{th}}$ ACM Conference on Embedded Network Sensor Systems, November 2012.

[5] "Implications of Internet of Things connectivity - Pew Research Center." `http://www.pewinternet.org/2017/06/06/the-internet-of-things-connectivity-binge-what-are-the-implications/`.
Online; accessed December 2, 2017.

[6] "Brief History of the Internet - Internet Society." `https://www.internetsociety.org/internet/history-internet/brief-history-internet/`. Online; accessed December 3, 2017.

[7] "Amazon has aquired 2lemetry to build out its Internet of Things strategy." `https://techcrunch.com/2015/03/12/amazon-has-quietly-acquired-2lemetry-to-build-out-its-internet-of-things-strategy/`.
Online; accessed December 3, 2017.

[8] "Verizon acquires IoT startup to make cities smarter." `https://venturebeat.com/2016/09/12/verizon-acquires-iot-startup-sensity-systems-to-make-cities-smarter-through-led-lights/`. Online; accessed December 3, 2017.

[9] "Internet of Things - Google Trends." `https://trends.google.com/trends/explore?date=2007-12-03%202017-12-03&q=%2Fm%2F02vnd10`.
Online; accessed December 3, 2017.

[10] "Cyber-Physical Systems - NIST."
`https://www.nist.gov/el/cyber-physical-systems`.
Online; accessed December 3, 2017.

[11] Jacob R, Zimmerling M, Huang P, Beutel J, Thiele L. *Towards Real-time Wireless Cyber-physical Systems.* Proceedings of Euromicro Conference on Real-Time Systems (ECRTS), July 2016.

[12] Landsiedel O, Ferrari F, Zimmerling M. *Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale.* Proceedings of the 11$^{\text{th}}$ ACM Conference on Embedded Networked Sensor Systems, November 2013.

[13] Ferrari F, Zimmerling M, Thiele L, Saukh O. *Efficient network flooding and time synchronization with Glossy.* Information Processing in Sensor Networks (IPSN), IEEE, April 2011.

[14] Leentvaar K, Flint J. *The capture effect in FM receivers.* IEEE Transactions on Communications, May 1976.

[15] He T, Stankovic JA, Abdelzaher TF, Lu C. *A spatiotemporal communication protocol for wireless sensor networks.* IEEE Transactions on Parallel and Distributed Systems, October 2005.

[16] Song J, Han S, Mok A, Chen D, Lucas M, Nixon M, Pratt W. *WirelessHART: Applying wireless technology in real-time industrial process control.* Real-Time and Embedded Technology and Applications Symposium, April 2008.

[17] Liu CL, Layland JW. *Scheduling algorithms for Multiprogramming in a Hard-Real-Time environment.* Journal of the ACM (JACM), January 1973.

[18] "What is a Wireless Sensor network - National Instruments."
`http://www.ni.com/white-paper/7142/en/`.
Online; accessed December 10, 2017.

[19] "TEC - Computer Engineering Group - ETH Zurich."
`http://www.tec.ethz.ch/`. Online; accessed December 13, 2017.

[20] "Contiki: The Open Source Operating System for the Internet of Things."
`http://www.contiki-os.org/index.html`.
Online; accessed December 13, 2017.

[21] "GitHub - ETHZ - TEC/LWB at dpp."
`https://github.com/ETHZ-TEC/LWB/tree/dpp`.
Online; accessed December 13, 2017.

[22] Walter F. *Real-time network functions for the Internet of Things.* Semester thesis, D-ITET ETH Zurich, June 2017.

[23] Acevedo J. *Real-time scheduling on resource-constrained embedded systems.* Master's thesis, TU Dresden, September 2016.

[24] Lim R, Ferrari F, Zimmerling M, Walser C, Sommer P, Beutel J. *Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems.* Proceedings of the 12th international conference on Information processing in sensor networks, April 2013.

[25] "FlockLab."
`https://www.flocklab.ethz.ch/wiki/`.
Online; accessed December 13, 2017.

[26] "CC430F5147 - TI.com"
`http://www.ti.com/product/cc430f5147`.
Online; accessed December 1, 2017.

[27] "MSP432P401R - TI.com"
`http://www.ti.com/product/msp432p401r`.
Online; accessed December 1, 2017.

[28] Zimmerling M, Ferrari F, Mottola L, Thiele L. *On modeling low-power wireless protocols based on synchronous packet transmissions.* Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), August 2013.