Adaptive Real-time Communication for Wireless Cyber-physical Systems

MARCO ZIMMERLING, TU Dresden
LUCA MOTTOLA, Politecnico di Milano and SICS Swedish ICT
PRATYUSH KUMAR, FEDERICO FERRARI, and LOTHAR THIELE, ETH Zurich

Low-power wireless technology promises greater flexibility and lower costs in cyber-physical systems. To reap these benefits, communication protocols must deliver packets *reliably* within *real-time deadlines* across *resource-constrained devices*, while *adapting* to changes in application requirements (*e.g.*, traffic demands) and network state (*e.g.*, link qualities). Existing protocols do not solve all these challenges simultaneously, because their operation is either localized or a function of network state, which changes unpredictably over time. By contrast, this paper claims a *global* approach that does *not* use network state information as input can overcome these limitations. The Blink protocol proves this claim by providing hard guarantees on end-to-end deadlines of received packets in multi-hop low-power wireless networks, while seamlessly handling changes in application requirements and network state. We build Blink on the non real-time Low-Power Wireless Bus (LWB), and design new scheduling algorithms based on the earliest deadline first policy. Using a dedicated priority queue data structure, we demonstrate a viable implementation of our algorithms on resource-constrained devices. Experiments show that Blink: (*i*) meets all deadlines of received packets; (*ii*) delivers 99.97 % of packets on a 94-node testbed; (*iii*) minimizes communication energy consumption within the limits of the underlying LWB; (*iv*) supports end-to-end deadlines of 100 ms across 4 hops and 9 sources; and (*v*) runs up to 4.1× faster than a conventional scheduler implementation on popular microcontrollers.

CCS Concepts: •**Networks** → **Network protocol design; Cyber-physical networks;** *Cross-layer protocols; Network dynamics;* Network experimentation; Logical / virtual topologies; Bus networks; •**Computer systems organization** → **Sensor networks; Sensors and actuators;** *Embedded software; Real-time system architecture;*

Additional Key Words and Phrases: Wireless multi-hop network, real-time communication, end-to-end deadline, earliest deadline first (EDF), priority queue data structure, real-world implementation, low-power platform, reliability, adaptivity, efficiency, Low-Power Wireless Bus (LWB), Glossy, synchronous transmissions

ACM Reference Format:
Marco Zimmerling, Luca Mottola, Pratyush Kumar, Federico Ferrari, and Lothar Thiele, 2017. Adaptive real-time communication for wireless cyber-physical systems. *ACM Trans. Cyber-Phys. Syst.* V, N, Article A (January 2016), 27 pages.
DOI: http://dx.doi.org/10.1145/3012005

1. INTRODUCTION

The benefits of low-power wireless communication technology in cyber-physical systems (CPS) are widely acknowledged [Honeywell 2006], ranging from better scalabil-

This work was supported by nano-tera.ch with Swiss Confederation financing, project X-Sense, by the Cluster Projects “Zero-energy Buildings in Smart Urban Districts” (EEB), “ICT Solutions to Support Logistics and Transport Processes” (ITS), and “Smart Living Technologies” (SHELL) of the Italian Ministry for University and Research, and by the German Research Foundation (DFG) through the Cluster of Excellence “Center for Advancing Electronics Dresden” (CfAED) and the Priority Program 1914, project EcoCPS.
Author’s addresses: M. Zimmerling, TU Dresden, Center for Advancing Electronics Dresden (CfAED), Networked Embedded Systems Group, Georg-Schumann-Strasse 11, BAR I-56, D-01187 Dresden, Germany; L. Mottola, Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Via Golgi 42, Building 22 (room 319, third floor), IT-20133 Milano, Italy; P. Kumar, F. Ferrari, and L. Thiele, ETH Zurich, Computer Engineering and Networks Laboratory, Gloriastrasse 35, ETZ-G87, CH-8092 Zurich, Switzerland.

DOI: http://dx.doi.org/10.1145/3012005

ity through lower installation and maintenance costs to greater flexibility in selecting sensing and actuation points [Åkerberg et al. 2011b]. Example applications include level control of dangerous liquids [Honeywell 2006], rapid prototyping of automation solutions in retrofitting buildings [Agarwal et al. 2011], and minimally invasive monitoring of safety-critical assets [Stankovic et al. 2003]. Lower costs and ease of installation motivate the use of battery-powered or energy-harvesting devices with low-power wireless transceivers and microcontrollers (MCUs) [Åkerberg et al. 2011a].

Challenges. Because CPS control physical processes that evolve over time, communication in CPS is inherently subject to *hard real-time requirements*; that is, packets that are *successfully received* at the intended destination must do so before stipulated *deadlines*, for example, to guarantee control stability [Sinopoli et al. 2004]. This entails that packets not meeting their deadline have no value to the application and count as lost. Support for this kind of *real-time traffic* is mainstream in wired fieldbuses [CAN 2004; FlexRay 2013]. In low-power wireless networks, however, the problem was recognized early on: "… because of the large scale, nondeterminism, noise, etc., it is extremely difficult to guarantee real-time properties." [Stankovic et al. 2003]. This is due to four challenges low-power wireless protocols must address simultaneously to support CPS:

— **Deadlines (D):** ensure that packets successfully received at the intended destination(s) meet *hard end-to-end deadlines* across multiple hops;
— **Reliability (R):** achieve a *high packet delivery ratio* across multiple hops despite the inherent unreliability of wireless communications;
— **Adaptivity (A):** *adapt* to unpredictable changes in application requirements (*e.g.*, traffic demands) and network state, that is, the physical-layer conditions determining the nodes' ability to communicate (*e.g.*, wireless link qualities, hop distances);
— **Efficiency (E):** operate *efficiently* with regard to limited resources, including compute power, memory, and energy, as well as large network scales.

As discussed in Sec. 2, these four challenges are yet unsolved. The approach taken by current solutions prevents them from addressing all challenges at once. Some operate in a *localized* fashion [He et al. 2005], which aids scalability but renders them unable to provide *end-to-end* guarantees. Others require time-varying network state information (*e.g.*, link qualities) as input [WirelessHART 2007], which determines their functioning. This, however, makes them susceptible to the unpredictable and non-deterministic dynamics of low-power wireless links [Baccour et al. 2012] that rapidly mutate the network state, impairing their ability to *promptly adapt* to those changes or to scale to large networks [Zhang et al. 2009; Saifullah et al. 2010; Chipara et al. 2011].

Approach. We present Blink, a real-time low-power wireless protocol that solves challenges **D**, **R**, **A**, and **E** *together*. Our approach is different than previous ones: Blink does *not* employ network state information as an input and uses *only* the application's real-time traffic requirements to *efficiently* compute a single *global* communication schedule *at runtime*, such that all received packets provably meet their deadlines. This crucially means that Blink's scheduling decisions are not determined by the current network state, and thus do not need to be adapted to its unpredictable changes. In this way, we detach Blink's operation from the network dynamics, overcoming the limitations of previous approaches and enabling adaptive real-time communication even in the face of mobile nodes and dynamically changing application requirements.

To realize our approach, we leverage the Low-Power Wireless Bus (LWB) [Ferrari et al. 2012] as Blink's underlying communication support. As described in Sec. 3, LWB is a *non real-time* protocol that maps all communication onto network-wide Glossy floods [Ferrari et al. 2011]. Real-world experiments show that Glossy achieves packet delivery ratios above 99.9 % in networks that range in size from 26 to 260 nodes, in density from a few to over 50 nodes in the same broadcast domain, in diameter from 3 to 8 hops, as well as under external interference (e.g., from Wi-Fi networks) and when a large subset of the nodes is mobile [Ferrari et al. 2011; Ferrari et al. 2012].

Contributions and road-map. Sec. 4 gives an overview of Blink, while Sec. 5 details its design and efficient implementation, which rest upon three key contributions:

— *Problem mapping.* In LWB, nodes communicate in a time-triggered fashion according to the *same global schedule*. Further, Glossy reliably delivers all packets to all nodes, while allowing us to not consider the time-varying network state as an input to the scheduling problem. We can therefore treat the whole multi-hop wireless network as a single communication resource that runs on a single clock. This allows us to map the real-time scheduling problem in Blink to *uniprocessor* task scheduling, making it easier to solve than the multi-processor case in prior works [Saifullah et al. 2010]. We note that Chipara et al. [2013] uses a similar abstraction for scheduling real-time communication in a many-to-one data collection scenario with in-network aggregation, whereas we focus on many-to-many real-time traffic in critical CPS applications.
— *Real-time scheduling policies.* Tackling the problem as uniprocessor task scheduling allows us to conceive scheduling policies based on the well-known earliest deadline first (EDF) principle [Liu and Layland 1973]. Using these policies, Blink computes a schedule so that all received packets generated by a set of admitted real-time streams provably meet their deadline, while minimizing the communication energy consumption within the constraints of the underlying LWB protocol. By computing the schedule online, Blink promptly adapts to changes in the application requirements.
— *Efficient data structure and algorithms.* Enabling EDF-based real-time scheduling on low-power, resource-constrained platforms is a significant challenge on its own. Due to its runtime overhead EDF has seen little adoption even on commodity hardware, despite its realtime-optimality [Buttazzo 2005; Sha et al. 2004]. To tackle this challenge, we exploit characteristics of our scheduling problem to design a highly efficient priority queue data structure and algorithms that take advantage of it.

In Sec. 6, we evaluate Blink on two testbeds with up to 94 nodes, four state-of-the-art MCUs, and using synthetic simulations as well as a time-accurate instruction-level emulator. For instance, our testbed results show that Blink meets all deadlines of received packets, while losing only 0.03 % of the packets over wireless. Beneficial statistical properties of Glossy [Zimmerling et al. 2013] allow to use well-known methods to deal with the very few missing packets in the design of CPS controllers [Sinopoli et al. 2004]. Further, we show that Blink supports end-to-end deadlines as small as 100 ms, thus meeting the requirements of industrial closed-loop control [Åkerberg et al. 2011a]. Such real-time performance rests a long way from the original LWB: simulations indicate that the latter meets only 35–72 % of deadlines in some settings we test. Finally, using our priority queue and scheduling algorithms, Blink achieves speed-ups of up to 4.1$\times$ compared to a conventional scheduler implementation on state-of-the-art MCUs. This proves instrumental to the viability of EDF-based real-time scheduling on widespread low-power embedded platforms, such as those based on MSP430 MCUs.

2. PROBLEM AND RELATED WORK

Guaranteeing hard real-time properties in multi-hop low-power wireless networks is a long-standing, yet unsolved problem [Stankovic et al. 2003]. The problem originates from the requirements of emerging CPS applications. We discuss these requirements next, then state the problem, and finally review previous approaches to solving it.

2.1. Application Requirements and Problem Statement

CPS embed distributed feedback loops into the physical world [Stankovic et al. 2005]. As the physical world evolves over time, timing constraints are important when embedded devices stream sensor data and control signals to drive time-critical control loops. The control logic executes right on the actuators that affect the environment, or on a few dedicated devices that periodically distribute control signals to the actuators.

Let Λ denote the set of all n *streams* in the network. Each stream $s_i \in \Lambda$ *releases* one packet at a periodic interval P_i, called the *period* of stream s_i. The *start time* S_i is the time when s_i releases the first packet. Every packet released by stream s_i should be delivered to the destination(s) within the same *relative deadline* D_i. The next packet is only released after the *absolute deadline* of the previous packet, that is, $D_i \leq P_i$. We refer to the absolute deadline of s_i as a shorthand for the absolute deadline of the last packet released by s_i. Overall, each stream $s_i \in \Lambda$ is characterized by its *profile* $\langle S_i, P_i, D_i \rangle$. If there are k streams with the exact same profile, we also write $k\langle \cdot, \cdot, \cdot \rangle$.[1]

The concrete stream profiles are application-specific. Specifically, the sensing modality and/or the nature of the feedback loop often dictate a stream's period P_i [Åkerberg et al. 2011a]. Temperature control in liquid volumes demands periods in the order of minutes [Paavola and Leiviska 2010], and coordinated multi-robot control runs with periods of at most tens of seconds [Mottola et al. 2014]. On the other hand, compressor speed control requires periods down to tens of milliseconds [Åkerberg et al. 2011a]. Greater opportunities for energy savings (*e.g.*, through duty cycling) are available in the former applications, yet we demonstrate in Sec. 6 that Blink can efficiently operate with periods in the hundreds of milliseconds range. The monitoring or control logic determines a stream's deadline D_i and starting time S_i. For example, closed-loop control typically requires shorter deadlines than open-loop control [Ogata 2001].

Deployments consist of tens to hundreds of devices. Due to the limited communication range of low-power wireless radios, *multi-hop communication* is typically needed to ensure connectivity. Some scenarios feature partially or completely mobile networks, for example, when optimizing sensing locations or coordinating mobile robots [Mottola et al. 2014]. This adds to the dynamics of low-power wireless links caused by interference, fading, and environmental changes [Srinivasan et al. 2010; Baccour et al. 2012].

Problem. Based on these requirements, one needs to solve challenges **D**, **R**, **A**, and **E** at once. Challenge **D** entails finding schedule(s) such that given n streams, $n = |\Lambda|$, for every stream $s_i \in \Lambda$, every packet released by s_i *can* be received within D_i time units.

2.2. Related Work

Previous efforts to solving the problem can be broadly classified depending on whether they need local or global knowledge as input for computing communication schedules.

An example of the former class is SPEED, where each node monitors its neighbors within radio range, for example, to acquire location information and detect transient congestion [He et al. 2005]. Using only this local information, each node computes and follows its own communication schedule. Conceptually similar approaches are adopted by Gu et al. [2009], Kanodia et al. [2001] and Lu et al. [2002]. These scale well because of their localized nature and can easily accommodate simple approaches to increase the robustness against varying application requirements and wireless dynamics, such as retransmissions to counteract packet loss [Liu et al. 2006; Gu et al. 2009; Suriyachai et al. 2010]. However, they cannot match the hard real-time requirements of CPS applications. Such requirements are specified from an *end-to-end* perspective, but

[1]For simplicity, we assume a stream releases one packet at a time. If a stream $\langle S_i, P_i, D_i \rangle$ releases k packets at a time, we implicitly transform this into $k\langle S_i, P_i, D_i \rangle$ streams each releasing one packet.

devices that reason solely based on local information and that can only influence their surroundings are unable to enforce end-to-end communication guarantees.

Instead, state-of-the-art solutions from industry [WirelessHART 2007; ISA100 2009; IEEE 802.15.4e TSCH 2012] and academia [Nirjon et al. 2010; O'Donovan et al. 2013] compute communication schedules using information about the global *network state*, that is, the physical-layer conditions that determine whether any two nodes can communicate. Global network state essentially takes the form of a connectivity graph, where the weight of edge $A \rightarrow B$ represents the quality (*e.g.*, packet reception ratio) of the link from node A to node B. Using global network state as an input, these solutions centrally compute and then distribute communication schedules tailored to each node. Nodes follow their own schedule locally, thereby forming multi-hop routing paths from sources to destination(s). Network state changes are typically handled through redundant paths and multi-channel operation [WirelessHART 2007; O'Donovan et al. 2013; Nirjon et al. 2010; IEEE 802.15.4e TSCH 2012]. These approaches can return highly optimized schedules in static, small-scale networks with little wireless dynamics [Zhang et al. 2009; Nirjon et al. 2010; Saifullah et al. 2010; O'Donovan et al. 2013]. Due to their inherent complexity, however, they suffer from two fundamental problems:

(1) Computing *per-node* schedules hardly scales to larger networks. Besides a few exceptions (*e.g.*, [Chipara et al. 2013]), existing works map the problem of scheduling real-time traffic to scheduling tasks on a *multiprocessor* [Saifullah et al. 2010], treating each node as a separate processor. As a result, in WirelessHART, computing optimal per-node communication schedules takes time at least exponential in the network diameter [Saifullah et al. 2010]. Despite attempts to address this issue [Zhang et al. 2009; Saifullah et al. 2010; Chipara et al. 2011], these schedulers are hardly practical in real networks of more than three hops [Chipara et al. 2011].
(2) Network state is volatile because of transient low-power wireless links [Srinivasan et al. 2010], environmental changes [Baccour et al. 2012], node crashes, and mobility [Xia et al. 2007]. Any change in the network state must be detected locally and communicated to the central entity to update the connectivity graph before new schedules can be computed and distributed. As this happens, new changes may occur, requiring to repeat the processing over and over again. Meanwhile, packets are lost due to inconsistent routing paths or miss their deadline because of obsolete schedules. Real-world experience shows that, for example, WirelessHART needs up to tens of minutes to adapt to network state changes [Åkerberg et al. 2011b].

Because of these problems, any solution using global network state as input cannot provide hard real-time guarantees in large-scale networks with non-negligible wireless dynamics. Thus, solving challenges **D**, **R**, **A**, and **E** at once remains an open problem.

3. BLINK FOUNDATION

This paper presents Blink, a low-power wireless protocol that solves challenges **D**, **R**, **A**, and **E** together. Before presenting the details of Blink's design and implementation, we describe LWB [Ferrari et al. 2012], which we use as communication support.

LWB is a non real-time protocol, where nodes communicate in a *time-triggered* fashion by following a *global schedule*. All communication occurs via *network-wide Glossy floods* [Ferrari et al. 2011]. Glossy distributes a packet from one node to all others over multiple hops, while time-synchronizing the whole network at no additional cost. Using Glossy as the only means of communication, LWB transforms a multi-hop wireless network into a *shared bus*, where all nodes are potential receivers of all packets.

LWB operation. As shown in Fig. 1 (A), LWB's operation unfolds in a series of *communication rounds* of *fixed* duration. Nodes keep their radios off between rounds to save

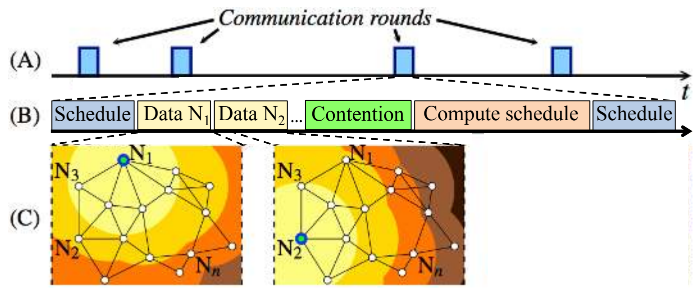


Fig. 1. Time-triggered operation in the Low-Power Wireless Bus (LWB).

energy. Each round consists of a sequence of *non-overlapping slots*, shown in Fig. 1 (B). All nodes participate in the communication in every slot: one node puts a packet on the bus (*i.e.*, initiates a flood), while all other nodes read the packet from the bus (*i.e.*, receive and relay the flood), as shown in Fig. 1 (C). At the end of a round, only the intended receivers (encoded in the packet header) deliver the packet to the application.

Each round starts with a slot allocated to a specific node, called *host*, for distributing the global schedule, as shown in Fig. 1 (B). The schedule specifies when the next round starts and which nodes are allowed to send their packet in the following *data slots*. Every round contains up to B data slots. Therefore, B and the time between subsequent rounds determine the bandwidth provided by LWB. The shorter the time between subsequent rounds, the more bandwidth is available, and vice versa. The time between subsequent rounds is upper-bounded to let nodes update their time synchronization state (via Glossy [Ferrari et al. 2011]) sufficiently often to compensate for clock drift.

Each slot corresponds to one Glossy flood, as shown in Fig. 1 (C). At the start of a flood, nodes turn on the radio and the owner of the slot transmits its packet (*e.g.*, N_1 in the first data slot in Fig. 1(B)). Glossy ensures nodes receiving the packet retransmit it *synchronously*. Due to constructive interference and capture effects [Leentvaar and Flint 1976], other nodes successfully receive the packet with high probability despite the apparent collisions. Using these *synchronous transmissions*, the flood spreads like a wave throughout the network (see Fig. 1 (C)), where the time needed to reach all nodes scales linearly with the network diameter. Note that the nodes' actions during a flood are only triggered by radio interrupts and occur *irrespective of the network state*.

To inform the host of their traffic demands, nodes compete in a dedicated *contention slot* (see Fig. 1 (B)) by initiating floods with different packets containing their stream requests. Due to capture effects [Leentvaar and Flint 1976], one packet reaches the host with high probability despite the contention. Afterwards, the host computes the schedule for the next round. The new schedule is sent in a final *schedule slot*, so nodes know right away when the next round starts and can turn off their radios until then.

Benefits. Using LWB as communication support for Blink brings a number of benefits, both in terms of performance and from a conceptual point of view.

In terms of performance, Glossy provides sub-microsecond synchronization accuracy and an end-to-end reliability above 99.9 % across a range of real-world scenarios [Ferrari et al. 2011; Ferrari et al. 2012]. In fact, by increasing the number of times a node transmits during a flood, Glossy's reliability can be pushed beyond 99.9999 % [Ferrari et al. 2011], which goes a long way towards solving challenge **R**. On top of this, Zimmerling et al. [2013] show how to provide probabilistic guarantees on LWB's end-to-end reliability. Although a purely flooding-based communication scheme may seem wasteful, LWB outperforms prior solutions also in terms of energy efficiency [Ferrari et al. 2012], thus providing a promising foundation for addressing part of challenge **E**.

From a conceptual point of view, LWB brings three key assets to the design of Blink:

— Glossy's *protocol logic* is independent of network state: nodes retransmit any packet they receive regardless of link qualities, hop distances, or other physical-layer characteristic. Thus, using Glossy as the only means to communicate, LWB readily supports scenarios with significant wireless dynamics including mobile nodes immersed in static infrastructures [Chipara et al. 2010], creating a *virtual single-hop network* where every node can directly communicate with every other node. Moreover, Glossy's network state independence allows us *not* to consider the network state as input to the scheduling problem. This solves part of challenge **A** in that there is *no need* to adapt to network state changes—a key difference compared to prior work.
— Nodes in LWB communicate in a time-triggered fashion according to a single global schedule. Together with the abstraction of a virtual single-hop network, we can treat the whole multi-hop wireless network as a *single communication resource that runs on a single clock*. This observation allows us to map the scheduling problem to that of scheduling tasks on a *uniprocessor*, making it easier to solve than the multiprocessor formulation in prior works [Saifullah et al. 2010]. Relying on this observation does not come without a cost; for example, it sacrifices some communication capacity as the network cannot simultaneously transmit messages from different sources. Sec. 6 shows quantitatively that the gains in simplicity outweigh the potential drawbacks.
— LWB readily supports different traffic patterns, such as one-to-many, many-to-one, and many-to-many, catering for CPS settings with multiple actuators or controllers. LWB also includes an effective mechanism to overcome the single point-of-failure at the host [Ferrari et al. 2012]. This contrasts what happens, for example, in case of a failure of the central network manager in WirelessHART [2007].

Limitations. Despite these benefits, LWB does not solve challenge **D** as its scheduling decisions are oblivious of packet deadlines and only meant to reduce energy consumption. Indeed, our results in Sec. 6.4 show that LWB only meets 35–72 % of deadlines across a diverse range of stream sets, even at low bandwidth demands.

On the other hand, considering deadlines in the scheduling decisions triggers frequent changes in the time between subsequent LWB rounds (see Fig. 1 (A)), even when the stream set does not change, as shown in Sec. 6.1. This behavior drastically differs from the original LWB scheduler, which keeps the time between rounds fixed unless the stream set changes, which helps deal with lost schedule packets. Without careful changes to the core LWB implementation, replacing the original scheduler with a real-time scheduler would result in low packet delivery ratio, thus reinforcing challenge **R**.

LWB also does not check if the available bandwidth can accommodate the stream requests it accepts. In case the bandwidth is insufficient, source nodes would need to locally buffer packets, eventually exhausting their memory, and ultimately losing data.

4. BLINK OVERVIEW

To overcome LWB's limitations and solve challenges **D**, **R**, **A**, and **E**, we must address the following issues. First, we need to conceive an *adaptive* policy to schedule packets in a way to meet all deadlines without unnecessarily sacrificing energy efficiency, while still allowing the set of streams to freely change at runtime. In doing so, our objectives are: (*i*) *realtime-optimal* scheduling to solve challenge **D**, and (*ii*) minimizing *communication energy consumption* as per challenge **E**. Objective (*i*) entails to admit a stream if and only if there exists a scheduling policy able to meet all deadlines, and to ensure that all received packets of admitted streams meet their deadlines. Objective (*ii*) entails to minimize the number of communication rounds over any given time interval,

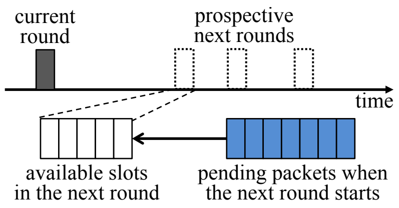

current round
prospective next rounds
time
available slots in the next round
pending packets when the next round starts

Fig. 2. Illustration of the interrelated problems we need to address in Blink.

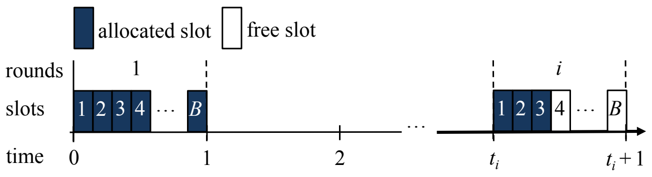


Fig. 3. Discrete-time model of Blink. *Each round is of unit length and comprises B data slots. Here, the i-th round has three allocated data slots, starts at time t_i, and ends at time $t_i + 1$.*

because each round incurs the energy overhead of distributing twice the schedule, no matter how many of the B available data slots are actually used.

Second, the adaptive scheduling policy must incur a small runtime overhead as per challenge **E**. To allow for a changeable stream set, the scheduler should execute online at the end of every round, so new stream requests are promptly accounted for. However, as shown in Fig. 1 (B), the longer it takes to compute the schedule, the fewer data slots fit within the fixed round duration, thus reducing the available bandwidth. Therefore, the scheduler needs to execute as fast as possible under severe resource constraints.

Translating these issues into a concrete protocol operation poses three interrelated problems we must address in the design of Blink:

(1) *Start of round computation.* As shown in Fig. 2 (top), at the end of a round we must decide when the next round starts. This can happen between the end of the current round and the maximum time allowed between rounds. Our decision should meet all deadlines while minimizing energy consumption. Intuitively, the earlier the next round starts, the more "chances" there are for packets to meet their deadlines, but the earlier a round starts, the more energy is consumed in the long run.
(2) *Slot allocation.* Once the start time of the next round is computed, given a number of packets waiting to be transmitted, we must decide which and how many of these packets are sent in the round, as shown in Fig. 2 (bottom). If there are more packets than the B available slots, we need to prioritize packets of different streams.
(3) *Admission control.* Changing application requirements imply that the stream set changes over time. We must therefore check whether adding a new stream prevents meeting the deadlines of the existing ones. This amounts to ensuring that the new stream set does not demand a bandwidth higher than what Blink can provide. We call this admission control, and say a stream set is *schedulable* if our solutions to (1) and (2) ensure that all received packets of all streams meet their deadlines.

The next section details our solutions to these problems. Like other time-triggered schemes [CAN 2004], Blink does not use end-to-end packet retransmissions, because Glossy keeps packet loss below 0.1 %, and CPS controllers can be designed to tolerate a small fraction of packet loss without sacrificing control stability or performance. These approaches often assume that packet losses follow a Bernoulli process [Sinopoli et al. 2004]. Since this assumption is indeed highly valid in Glossy [Zimmerling et al. 2013], we choose to trade a marginal improvement in deadline success ratio for a higher bandwidth and for supporting streams with shorter deadlines in Blink. Please note that due to the unpredictability involved when nodes contend for submitting stream requests, Blink can only give real-time guarantees after a request has passed admission control.

5. DESIGN AND IMPLEMENTATION OF BLINK

In the following, Secs. 5.1 and 5.2 discuss the slot allocation and the start of round computation in Blink, respectively, assuming the set of streams is schedulable. Sec. 5.3 describes how Blink ensures this condition through admission control.

Throughout the discussion, we consider the discrete-time model illustrated in Fig. 3. Each round is *atomic* and of unit length. This choice stems from the time-critical radio interrupts Glossy must serve during a flood: interleaving other processing may cause interference, so other events such as packet deliveries are only served before or after a round [Ferrari et al. 2012]. As a result, the timeline of possible start times of a round is also discrete. Between rounds (*e.g.*, between $t = 1$ and t_i in Fig. 3) nodes may perform any other application processing. Despite the specific discrete-time model, the concepts described below enjoy general validity, as explained in Online Appendix A available in the ACM Digital Library or in a companion technical report [Zimmerling et al. 2016].

5.1. Slot Allocation

Let us assume that the start time of the next round has been computed. We now need to determine the schedule for that next round, which raises two basic questions. With B slots available per round, how many pending packets should we allocate? How should we prioritize packets of different streams if the number of pending packets exceeds B?

Algorithms. To answer the first question, we note that delaying a packet by not sending it in an otherwise empty slot does not lead to improved schedulability or reduced energy overhead. In the following round, the number of pending packets is the same or larger, which can only worsen the overall schedulability. Further, as explained in Sec. 3, the energy *overhead* of Blink (on top of serving application data) is determined by the number of rounds over a given interval of time, not by the number of allocated data slots. Thus, we allocate as many pending packets as possible in every round.

As noted in Sec. 4, Blink allows us to treat the network as a single communication resource and hence to resort to uniprocessor scheduling policies to answer the second question. Among these, *earliest deadline first (EDF)* is provably realtime-optimal [Sha et al. 2004]: if a set of streams can be scheduled such that all deadlines are met, then using EDF also meets all deadlines. This holds also for stream sets demanding the full bandwidth, whereas other well-known policies (*e.g.*, rate-monotonic) may fail to meet all deadlines at lower bandwidth demands [Liu and Layland 1973]. In other words, EDF efficiently exploits the limited communication resources. Finally, EDF can readily cope with *runtime changes* in the stream sets [Sha et al. 2004] as the packet priorities (*i.e.*, absolute deadlines) are dynamically computed while the system executes. This is crucial to adapt to varying application requirements or crashing source nodes.

Using EDF scheduling in Blink means allocating the next free data slot in a round to the packet whose deadline is closest to the start time of the round, until the round is full or there are no more pending packets ready to be sent. This simple logic, however, bears a significant run-time overhead [Buttazzo 2005]. To implement EDF efficiently, one should maintain the set of streams in order of increasing absolute deadline, while the streams' absolute deadlines are being updated from one packet to the next as they are allocated to slots in a round. This runtime overhead is one of the reasons why EDF is rarely used in real systems, such as operating system kernels [Buttazzo 2005].

Design and implementation in Blink. Enabling EDF-based real-time scheduling on resource-constrained platforms is not a trivial problem. A data structure is needed that allows to efficiently implement the operations required when manipulating the stream set Λ during EDF-based slot allocation. Table I summarizes these operations.

Besides operations to add or remove streams, EDF requires a FINDMIN() operation to retrieve the stream with the earliest absolute deadline, which is to be served next. For this operation to be efficient, the streams should be maintained in order of increasing absolute deadline. A *priority queue*, where streams with smaller absolute deadline are given higher priority, is thus a natural candidate. Moreover, after serving stream s, its absolute deadline must be set to the deadline of its next packet. Hence, we require

Table I. Operations on stream set required for EDF scheduling. *The key of a streams is its absolute deadline.*

Operation	Description
INSERT(s)	Insert stream s into the stream set
DELETE(s)	Delete stream s from the stream set
DECREASEKEY(s, δ)	Propagate an increment of δ in the key of stream s in the stream set
FINDMIN()	Return a reference to the stream with the minimum key in the stream set
FIRST(t)	Position traverser t at the stream with the minimum key in the stream set
NEXT(t)	Advance traverser t to the stream with the next larger key in the stream set

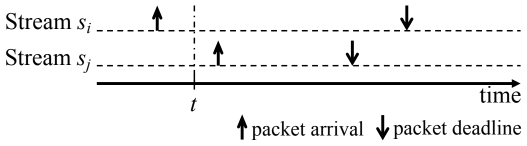


Fig. 4. Example motivating EDF traversal of stream set for efficient slot allocation.

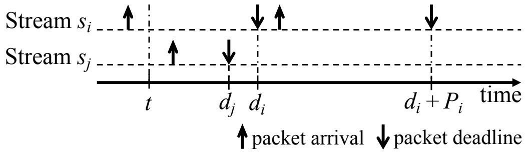


Fig. 5. Illustration of upper-bounding the difference between the absolute deadlines of any two streams.

a DECREASEKEY(s, δ) operation that propagates an increment of δ in the absolute deadline of s in the queue, so the priority of s is *decreased* relative to all other streams.

These operations are supported by nearly all priority queue data structures [Brodal 2013]. In addition, because the highest-priority stream returned by FINDMIN() may release its packet only *after* the start of the next round at time t, as illustrated in Fig. 4, we require operations to perform an efficient *EDF traversal* of the streams while *only* those with pending packets are updated. Specifically, it should be possible to position a *traverser* t at the highest-priority stream using FIRST(t), and then to visit streams in EDF-order through repeated NEXT(t) calls. During the EDF traversal, the priority of only those streams t with a pending packet is updated using DECREASEKEY(t, δ).

Finding a data structure that can support all required operations *efficiently* in time and memory is challenging. Only a few of the widely-used priority queue data structures, from the binary heap to the red-black tree used within the Linux scheduler [Molnar 2015], efficiently support an *in-order* traversal during which the data structure is possibly altered, which we need for EDF scheduling. In fact, updating a stream using DECREASEKEY(t, δ) likely changes the relative ordering of streams, which triggers structural changes inside the data structure. Thus, any kind of runtime stack or pointer used for in-order traversal becomes invalid [Pfaff 2004] and the traversal must start anew. This approach is highly inefficient for any reasonable number of streams.

We find, however, that our scheduling problem has the following characteristic properties that allow us to use a simple, yet highly efficient priority queue data structure:

(1) A stream's absolute deadline, called the *key* of a stream, is a non-negative integer.
(2) The key increases monotonically as it is being updated from one packet to the next.
(3) The range of keys in the set of streams at any one time is bounded, as stated below.

THEOREM 1. *Let $\overline{P}$ be an upper bound on the period P_i of every stream $s_i \in \Lambda$. Then, there are never more than $2\overline{P} - 1$ distinct keys in the stream set Λ at any one time.*

PROOF. Let d_i be the absolute deadline of stream s_i (*i.e.*, the deadline of s_i's current packet) at some point in time. Stream s_i's relative deadline D_i can be shorter than its period P_i, so its current packet may not have arrived yet. To determine the maximum number of distinct keys (*i.e.*, absolute deadlines) in the set of streams Λ at any one time, we must upper-bound the difference between the absolute deadlines of any two streams, that is, maximize $\Delta_{ij} = d_i - d_j$ for any two streams $s_i, s_j \in \Lambda$.

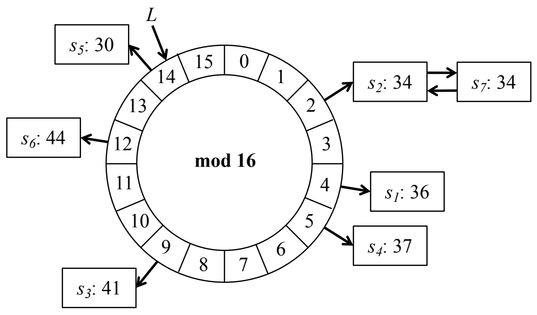


Fig. 6. Bucket queue implemented as a circular array of $2\bar{P}$ doubly-linked lists. *In this example, the largest period of any stream $\overline{P}$ is 8, so the queue consists of 16 buckets. The queue contains seven streams ordered by increasing key.* FINDMIN() *sets L to 14 because this is the bucket containing the stream with the smallest key.*

The value of Δ_{ij} is larger when packets with later deadlines are sent before packets with earlier deadlines, due to the order of packet arrival. Let us consider the example in Fig. 5. Assume at time t the current packet of stream s_i with deadline d_i is sent, while the current packet of stream s_j with an earlier deadline $d_j < d_i$ is yet to be sent. This can happen if and only if s_i's packet arrives strictly before s_j's packet; that is, at time t, s_j's packet is yet to arrive. After sending the packet of stream s_i, its absolute deadline becomes $d_i + P_i$, while the absolute deadline of stream s_j is still d_j. Thus, we have $\Delta_{ij} = d_i + P_i - d_j$. What is the upper bound on Δ_{ij}? As s_i's packet has arrived by time t, we have $d_i \leq t + P_i$. Also, as s_j's packet has not yet arrived by time t, we have $d_j > t$. With these two conditions, we can establish the following bound

$$\Delta_{ij} = d_i + P_i - d_j < t + P_i + P_i - t \leq 2\overline{P}, \tag{1}$$

where $\overline{P}$ is an upper bound on the period of any stream. Because all absolute deadlines are integers, the strict inequality in (1) implies that Δ_{ij} is at most $2\overline{P} - 1$. □

Given the properties above, we may consider a *monotone integer priority queue*. Similar reasonings apply, for example, to discrete event simulation [Brown 1988]. To the best of our knowledge, however, these have not been exploited for real-time scheduling mainly because the three properties stated above and the implications on the data structure are quite distinctive. Specifically, we use a *one-level bucket queue* [Dial 1969] implemented as a circular array $\mathbf{B}$ of $2\overline{P}$ doubly-linked lists, as shown in Fig. 6, where $\overline{P}$ is an upper bound on the period of any stream. Stream s_i with key d_i is stored in $\mathbf{B}[d_i \bmod 2\overline{P}]$. Since a stream's relative deadline D_i is no longer than its period P_i, all keys in the bucket queue are always in the range $[d_{min}, d_{min} + 2\overline{P} - 1]$, where d_{min} is the smallest key currently in the queue. Thus, all streams in a bucket have the *same* key. For example, the keys of the streams in the bucket queue of Fig. 6 are in the range $[30, 45]$, because $d_{min} = 30$ and $\overline{P} = 8$, and the two streams in $\mathbf{B}[2]$ have key 34.

INSERT(s), DELETE(s), and DECREASEKEY(s, δ) take constant time because buckets are implemented as doubly-linked lists. INSERT(s) inserts a stream s with key d into $\mathbf{B}[d \bmod 2\overline{P}]$. DELETE$(s)$ removes s from the list containing it. DECREASEKEY(s, δ) first performs a DELETE(s) and then re-inserts stream s into $\mathbf{B}[(d + \delta) \bmod 2\overline{P}]$.

We implement FINDMIN() using an index L, initially set to 0. If $\mathbf{B}[L]$ is empty, FINDMIN() increments L (modulo $2\overline{P}$) until it finds the first non-empty bucket; otherwise, it returns the first stream on the list in $\mathbf{B}[L]$. FIRST(t) works alike, using another index I. NEXT(t) moves to the next stream on the list in $\mathbf{B}[I]$. At the end of the list, NEXT(t) increments I (modulo $2\overline{P}$) until it finds the next non-empty bucket. Unlike the vast

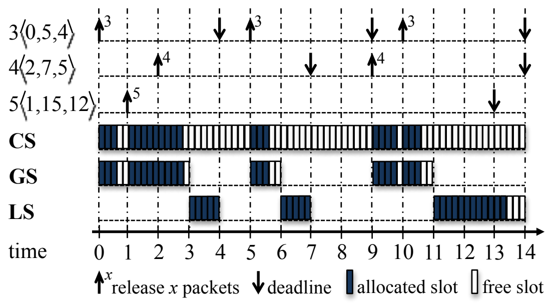


Fig. 7. Example execution of **CS**, **GS**, and **LS**. ***CS** and **GS** schedule more rounds than necessary. Instead, **LS** meets all deadlines while minimizing communication energy costs (i.e., minimizing the number of rounds).*

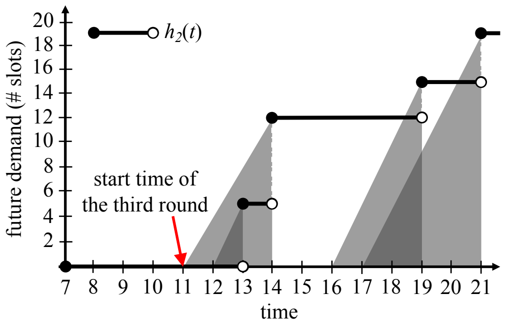


Fig. 8. Illustration of how **LS** computes the latest possible start time of the third round in Fig. 7.

majority of priority queues, this logic enables smoothly continuing an EDF traversal of streams despite updates. The three operations run in $O(2\overline{P})$ worst-case time.

Our priority queue data structure underpins not just the EDF-based slot allocation, whose pseudocode can be found in Online Appendix B, but *all* algorithms required for real-time scheduling in Blink (see, for example, Algorithm 1), which would *not* be feasible otherwise on a resource-constrained platform. Besides enabling a smooth EDF traversal, the efficiency of our priority queue implementation stems mainly from two aspects. First, DECREASEKEY(s, δ) is frequently used and at the same time extremely efficient due to its constant running time. Second, the cost of searching for a non-empty bucket amortizes: NEXT(t) needs to increment index I in the worst case $2\overline{P} - 1$ times, yet the following n calls to NEXT(t) require *no* searching as all n streams are necessarily in $\mathbf{B}[I]$. As a result, Blink can handle hundreds of streams on resource-constrained devices even when the stream set is continuously changing, as demonstrated in Sec. 6.

5.2. Start of Round Computation

We now turn to the problem of computing the start time of the next round. We use an illustrate example with $B = 5$ slots per round and twelve streams with three distinct profiles: $3\langle 0, 5, 4\rangle$, $4\langle 2, 7, 5\rangle$, and $5\langle 1, 15, 12\rangle$. Fig. 7 shows the release times and deadlines of packets generated by these streams in the first 14 time units. Using the EDF-based slot allocation described before, when should a round start to meet all deadlines while minimizing communication energy *overhead*, that is, minimizing the number of rounds? The scheduling policies we present next are all realtime-optimal; that is, if the stream set is schedulable, they guarantee that no deadlines are missed but for packet losses. They differ, however, in the energy required for providing this real-time service.

Algorithms. One way, called *contiguous scheduling* (**CS**), is to start a round immediately after the previous one. **CS** offers the highest bandwidth and thus necessarily meets all deadlines, provided the streams are schedulable. However, **CS** wastes energy by scheduling more rounds than necessary. In Fig. 7, using **CS**, 8 out of the first 14 rounds are *empty*: they contain only free slots, causing unnecessary energy overhead.

Greedy scheduling (**GS**) improves on **CS** by delaying the next round until there is at least one pending packet. **GS** is realtime-optimal just like **CS**, because it schedules packets as soon as possible. It can also reduce the energy overhead compared with **CS** in certain situations. In Fig. 7, **GS** results in only 6 rounds in the first 14 time units. However, there are still 8 free slots, raising the question whether we can do even better.

The crucial observation is that **GS** starts the next round no matter how "urgent" it really is. If there was still some time until the earliest deadline of all pending packets, we could delay the next round even further. Meanwhile, we could await more packet arrivals and thus allocate more slots in the next round, attaining a better utilization of individual rounds. This strategy, however, may do more harm than good: without knowing the future bandwidth demand, we may end up deferring the next round to a time where the number of packets that must be served over a certain interval exceeds the number of slots available in that interval. This situation would inevitably cause deadline misses. To prevent it, we need a way to forecast the bandwidth demand.

Lazy scheduling (**LS**) is precisely based on this intuition. At the core of **LS** is the notion of *future demand* $h_i(t)$ that quantifies the number of packets that must be *served* between the end of round i and some future time t. This includes all packets that have *both* their release time and deadline no later than time t, and are not served until the end of round i. This corresponds to the following expression

$$h_i(t) = \sum_{j=1}^{n} \begin{cases} \lfloor (t - d_j)/P_j \rfloor + 1, & \text{if } d_j \leq t \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where P_j is the period and d_j is the deadline of stream s_j's current packet.

LS uses $h_i(t)$ to forecast the bandwidth demand and, based on this, computes the latest possible start time of the next round that does not cause any deadline misses. As an example, let us compute the start time of the third round t_3 in Fig. 7 using **LS**.

(1) *Compute* $h_2(t)$. As illustrated in Fig. 8, $h_2(13) = 5$ because $t = 13$ is the absolute deadline of the $5\langle 1, 15, 12\rangle$ streams whose packets are pending at the end of the second round; $h_2(14) = h_2(13) + 7 = 12$, since $t = 14$ is the deadline of the 7 packets released by the other streams at $t = 9$ and $t = 10$; and so on.
(2) *Determine a set of latest possible start times* $\{t_3^i\}$. For instance, $h_2(13) = 5$ packets must be served no later than time 13. With $B = 5$ slots available in each round, serving them takes $\lceil h_2(13)/B \rceil = 1$ round. Thus, we get a first latest possible start time $t_3^1 = 13 - 1 = 12$, indicated in Fig. 8 by casting a shadow on the time axis. Next, $h_2(14) = 12$ packets must be served before time 14, which takes $\lceil h_2(14)/B \rceil = 3$ rounds. So, a second latest possible start time of the third round is $t_3^2 = 14 - 3 = 11$. The same reasoning repeats, identifying more latest possible start times.
(3) *Take the minimum latest possible start time as* t_3. Based on the reasoning in (2), delaying the start of the third round beyond the beginning of the shady area at $\min\{t_3^i\} = 11$ in Fig. 8 would cause deadline misses. Indeed, if we had served only $h_2(13) = 5$ packets between times 12 and 13, we would be left with $h_2(14) - h_2(13) = 7$ packets to serve between times 13 and 14, but these do not fit into $B = 5$ slots. Alternatively, an earlier start time could, in the long run, lead to more rounds than needed, sacrificing energy efficiency. Thus, the third round should start at $t_3 = 11$.

The example clarifies the intuition behind **LS**. Two questions remain. Which times t do we need to inspect in steps (1) and (2)? How far do we need to look into the future?

To answer the first question, we note that the future demand $h_i(t)$ is a step function: its value increases only at times of absolute deadlines (see Fig. 8). We can therefore skip all intervals where $h_i(t)$ is constant. The answer to the second question involves two observations. First, as described in Sec. 3, we can delay the next round by at most T_{max} after the previous round because Blink requires to update the nodes' synchronization state sufficiently often. To find out whether we can delay the next round by T_{max}, we need to evaluate $h_i(t)$ for at least T_{max} time units after the end of the previous round at $t_i + 1$. Second, we need to consider any demand arising *after* $t = t_i + 1 + T_{max}$, which could possibly prevent us from delaying the next round by T_{max}. Thus, we must evaluate $h_i(t)$ for another T_b time units beyond $t = t_i + 1 + T_{max}$, where T_b is the *synchronous busy period* [Spuri 1996]. Informally, T_b is the minimum time needed to serve the maximum demand a stream set may possibly create.[2] By looking up to $t = t_i + 1 + T_{max} + T_b$ into the future, we ultimately ensure that all deadlines are met.

We now formalize how **LS** computes the latest possible start time of the next round.

THEOREM 2. *Let T_b be the synchronous busy period of the stream set Λ, T_{max} the largest time by which the next round can be delayed after the previous one, and B the number of data slots available in a round. Using **LS**, the start time of each round t_i for all $i = 0, 1, \ldots$ is given by*

$$t_{i+1} = \min(t_i + T_{max}, T_i), \tag{3}$$

where $t_0 = -1$ and T_i is given by

$$T_i = \min_{t \in \mathcal{D}_i} \left(t - \left\lceil \frac{h_i(t)}{B} \right\rceil \right). \tag{4}$$

$\mathcal{D}_i$ denotes the set of deadlines in the time interval $[t_i + 1, t_i + T_{max} + T_b + 1]$ of packets that are unsent until the end of round i, and $h_i(t)$ is the future demand defined in (2).

PROOF. A *schedule* S specifies for each round i its start time t_i and the set of packets to be transmitted in the round. Let $\mathbf{S^{LS}}$ denote the schedule computed by **LS**. We prove this theorem by contradiction; that is, we show that there cannot be any schedule $S' \neq \mathbf{S^{LS}}$ such that S′ is realtime-optimal and some round starts *later* in S′ than in $\mathbf{S^{LS}}$. If we show this, it follows that amongst all realtime-optimal schedules, $\mathbf{S^{LS}}$ delays the start of each round the most and thus minimizes the communication energy consumption.

Let $t_i^{\mathbf{LS}}$ and t'_i denote the start times of the i-th round in $\mathbf{S^{LS}}$ and S′, and let $h_i^{\mathbf{LS}}$ and h'_i denote the future demands after the end of the i-th round in $\mathbf{S^{LS}}$ and S′, respectively.

Assume some round in S′ starts later than in $\mathbf{S^{LS}}$. Let the m-th round be the first such round, that is, $m = \min\{i \mid t'_i > t_i^{\mathbf{LS}}\}$. In $\mathbf{S^{LS}}$, the m-th round starts at $t_m^{\mathbf{LS}}$ since, according to Theorem 2, at least one of two conditions holds:

(1) $t_m^{\mathbf{LS}} - t_{m-1}^{\mathbf{LS}} = T_{max}$, where T_{max} is the largest interval between the start of consecutive rounds supported by the LWB communication support [Ferrari et al. 2012],
(2) $h_{m-1}^{\mathbf{LS}}(t^*) > B(t^* - t_m^{\mathbf{LS}} - 1)$ for some time $t^* > t_m^{\mathbf{LS}}$.

Assume condition (1) holds. Then, it follows from the definition of m that $t'_m - t'_{m-1} > t_m^{\mathbf{LS}} - t_{m-1}^{\mathbf{LS}} = T_{max}$. This violates the requirement that the time between the start of any two consecutive rounds in S′ must not exceed T_{max}.

Assume condition (2) holds. Then, the interval $[t_m^{\mathbf{LS}}, t^*]$ is a busy period in $\mathbf{S^{LS}}$, so the number of packets sent in this interval, denoted $\eta^{\mathbf{LS}}(t_m^{\mathbf{LS}}, t^*)$, is lower-bounded as

$$\eta^{\mathbf{LS}}(t_m^{\mathbf{LS}}, t^*) > B(t^* - t_m^{\mathbf{LS}} - 1). \tag{5}$$

On the other hand, since $t'_m > t_m^{\mathbf{LS}}$ due to the definition of m, the number of packets transmitted in S′ in the interval $[t'_m, t^*]$, denoted $\eta'(t'_m, t^*)$, is upper-bounded

$$\eta'(t'_m, t^*) \leq B(t^* - t'_m) \leq B(t^* - t_m^{\mathbf{LS}} - 1). \tag{6}$$

From (5) and (6) follows a strict inequality

$$\eta^{\mathbf{LS}}(t_m^{\mathbf{LS}}, t^*) > \eta'(t'_m, t^*). \tag{7}$$

We also know that $\eta^{\mathbf{LS}}(0, t_m^{\mathbf{LS}}) \geq \eta'(0, t'_m)$, because each round $1, 2, \ldots, m-1$ starts no earlier in $\mathbf{S^{LS}}$ than in S′, and in $\mathbf{S^{LS}}$ as many pending packets as possible are sent in

[2]The maximum demand arises when all streams release their packet at the same time. **CS** essentially serves this demand "as fast as possible." The synchronous busy period T_b is then the time between the simultaneous arrival of packets from all streams and the first idle time where no packet is pending under **CS** scheduling.

each round. Combining this with (7), we have

$$\eta^{\mathbf{LS}}(0, t^*) > \eta'(0, t^*). \tag{8}$$

Thus, $\mathrm{S}^{\mathbf{LS}}$ tightly meets all deadlines at time t^*, while transmitting more packets than S'. As $\mathrm{S}^{\mathbf{LS}}$ prioritizes packets using EDF, which is realtime-optimal [Sha et al. 2004], S' necessarily misses a deadline at or before time t^*. This contradicts the assumption that S' is realtime-optimal. Thus, for either condition that impacts the choice of $t_m^{\mathbf{LS}}$, we have shown that the assumptions on S' are contradicted. □

The following main result states that **LS** meets the two objectives from Sec. 4.

THEOREM 3. *The **LS** policy is real-time optimal and minimizes the communication energy consumption within the limits of the underlying LWB communication support.*

PROOF. Due to time synchronization constraints imposed by LWB, the start of the next round at time t_{i+1} can be deferred by at most T_{max} after the start of the previous round at time t_i. This implies the first component of the min-operation in (3).

We now show that the second component of the min-operation in (3) ensures that all packets meet their deadlines. The number of packets that must be sent between the end of round i and some time $t \geq t_i + 1$ is given by the future demand $h_i(t)$. The available bandwidth in the interval $[t_{i+1}, t]$ is $B(t - t_{i+1})$, where B is the number of slots available per round. To ensure that all packets meet their deadlines, the future demand $h_i(t)$ must not exceed the available bandwidth for any time $t \geq t_i + 1$, that is, $B(t - t_{i+1}) \geq h_i(t)$. Since $m \geq x$ if and only if $m \geq \lceil x \rceil$ for any integer m and real number x, $t - t_{i+1} \geq \lceil h_i(t)/B \rceil$. In particular,

$$t_{i+1} \leq \min_{t \geq t_i + 1 \,\wedge\, h_i(t) > 0} \left(t - \lceil h_i(t)/B \rceil\right). \tag{9}$$

The min-operation in (9) is to be performed for every time t larger than $t_i + 1$ at which the future demand $h_i(t)$ is greater than zero. We can restrict this in two ways.

First, we need to apply the min-operation only at every time t in the interval $[t_i + 1, t_i + T_{max} + T_b + 1]$, where T_b is the synchronous busy period of the stream set. We prove this by contradiction. Let for some $\hat{t} > t_i + T_{max} + T_b + 1$,

$$\hat{t} = \underset{t \geq t_i + 1 \,\wedge\, h_i(t) > 0}{\arg\min} \left(t - \lceil h_i(t)/B \rceil\right). \tag{10}$$

Let the quantity $\hat{t} - \lceil h_i(\hat{t})/B \rceil$ be equal to the start time of the next round t_{i+1} and strictly less than $t_i + T_{max}$,

$$t_{i+1} = \hat{t} - \lceil h_i(\hat{t})/B \rceil < t_i + T_{max}. \tag{11}$$

Since $\lceil x \rceil = m$ if and only if $m - 1 < x \leq m$ for any integer m and real number x, $\hat{t} - t_{i+1} - 1 < h_i(\hat{t})/B$. Multiplying both sides by the positive quantity B,

$$(\hat{t} - t_{i+1} - 1)B < h_i(\hat{t}). \tag{12}$$

We interpret (12) as follows. The future demand $h_i(\hat{t})$ exceeds the bandwidth available in $[t_{i+1} + 1, \hat{t}]$. This means that if one were to contiguously serve a demand as large as $h_i(\hat{t})$, the required time would exceed the length of the interval $[t_{i+1} + 1, \hat{t}]$. We can therefore consider $[t_{i+1} + 1, \hat{t}]$ a *busy period* of length $\hat{t} - t_{i+1} - 1 > \hat{t} - (t_i + T_{max}) - 1 > T_b$, because $t_{i+1} < t_i + T_{max}$ according to (11). However, T_b is the length of the synchronous busy period, which is by definition the longest possible busy period [Spuri 1996]. This contradicts the supposition on the existence of $\hat{t}$.

Second, we need to perform the min-operation in (9) only at times when $h_i(t)$ has discontinuities. In fact, $h_i(t)$ is a right-continuous function with discontinuities only at

Algorithm 1 Compute start time of next round according to lazy scheduling (**LS**)

```
Input  A bucket queue that is a deep copy of the set of streams $\Lambda$ (smaller $absDeadline$ means
  higher priority), the start time of the current round $t_i$, and the synchronous busy period $T_b$.
Output  The start time of the next round $t_{i+1}$ according to LS.
  initialize $futureDemand$ to 0 and $minSlack$ to $\infty$
  set $s$ to highest-priority stream using $s = \textsc{FindMin}()$
  $t = s.absDeadline$
  while $t \leq t_i + T_{max} + T_b + 1$ do
      $futureDemand = futureDemand + 1$
      $s.absDeadline = s.absDeadline + s.period$
      update priority of $s$ using $\textsc{DecreaseKey}(s,\ s.period)$
      set $s$ to highest-priority stream using $s = \textsc{FindMin}()$
      if $s.absDeadline > t$ then
          $minSlack = \min((t - t_i)B - futureDemand, minSlack)$
      end if
      $t = s.absDeadline$
  end while
  $t_{i+1} = t_i + \min(\lfloor minSlack/B \rfloor, T_{max})$
```

times that coincide with the deadline of a packet. Thus, we can restrict the domain of the min-operation to the set of deadlines $\mathcal{D}_i$ in the interval $[t_i + 1, t_i + T_{max} + T_b + 1]$ of packets that are unsent until the end of round i. Since (9) yields the largest possible t_{i+1} in the case of equality, we obtain the second component of the min-operation in (3)

$$T_i = \min_{t \in \mathcal{D}_i} \left(t - \lceil h_i(t)/B \rceil \right). \tag{13}$$

Finally, because EDF is realtime-optimal [Liu and Layland 1973], the necessary condition in (13) is also sufficient. □

Design and implementation in Blink. The challenge to using **LS** is to efficiently compute the future demand $h_i(t)$. The analytic expression for $h_i(t)$ in (2) is obtained by applying concepts from the real-time literature [Stankovic et al. 1998]. We attempt to compute this expression on a TelosB [Polastre et al. 2005] and observe prohibitive running times due to many time-consuming divisions. This is generally expected on resource-constrained platforms that lack hardware support for divisions. Nevertheless, as we show in Sec. 6.5, the approach we describe next outperforms the analytic method even on very powerful state-of-the-art platforms, including a 32-bit ARM Cortex-M4.

The key idea is to determine $h_i(t)$ by performing efficient operations on the priority queue of streams rather than computing costly divisions. Algorithm 1 shows the pseudocode to compute the start time of the next round t_{i+1} according to Theorem 2. The algorithm operates on a deep copy of the current set of streams, maintained in a bucket queue in order of increasing absolute deadline. It fictitiously serves streams in EDF order, as if it would allocate slots to pending packets. It uses variable $futureDemand$ to keep track of the number of streams served thus far and maintains variable $minSlack$ that ultimately determines how far we can delay the start of the next round.

By avoiding divisions and using our efficient priority queue data structure, our implementation of Algorithm 1 achieves several-fold speed-ups over the analytic method. We use the same techniques to efficiently compute the length of the synchronous busy period T_b (see Online Appendix C) and demonstrate similar speed-ups over an existing analytical method from the real-time literature [Stankovic et al. 1998]. Experimental results in Sec. 6.5 indicate that these improvements in processing time are instrumental to the viability of real-time scheduling in Blink on popular low-power platforms.

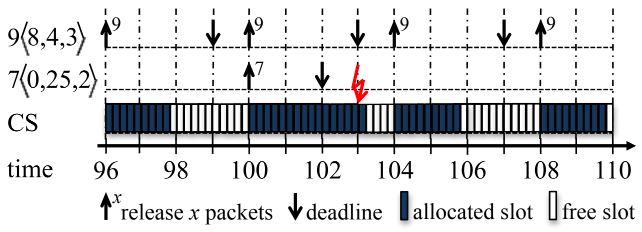


Fig. 9. Example of a stream set that is not schedulable, a situation admission control aims to prevent. *The streams demand 16 slots between time 100 and time 103, but there are only 15 slots available in this interval.*

5.3. Admission Control

So far we assumed the set of streams is schedulable, yet this is not always the case. As Fig. 9 shows, for $B = 5$, the set of $9\langle 8,4,3\rangle$ and $7\langle 0,25,2\rangle$ streams is not schedulable. The streams require altogether 16 slots in the interval between time 100 and time 103; however, there are only 15 slots available in this interval, which causes one packet to miss its deadline. We show next how to prevent such situations by checking *prior* to the addition of a new stream whether the resulting set of streams is still schedulable.

Algorithms. As illustrated by the example above, deadlines are missed if, over some interval, the demand exceeds the available bandwidth. As explained in Sec. 5.2, the **CS** policy offers the highest possible bandwidth. Hence, admission control under all scheduling policies in Sec. 5.2 amounts to checking if **CS** can meet all deadlines.

We must perform this check over an interval with highest demand. If the bandwidth is sufficient in this extreme situation, we can safely admit the new stream. Precisely identifying when this situation occurs is, however, non-trivial, because the streams' different start times and periods may defer this situation until some arbitrary time. In Fig. 9, for example, it is not until $t = 100$ that an interval of highest demand begins.

To tackle this problem, we deliberately create an interval of maximum demand by pretending that all streams release a packet at $t = 0$. Using the concept of synchronous busy period, we then check if **CS** can meet all deadlines in the interval $[0, T_b]$. From this intuition follows a theorem, whose proof descends from known results [Spuri 1996].

THEOREM 4. *For a set of streams Λ with arbitrary start times S_i, let Λ' be the same set of streams except all start times are set to zero. With B data slots available in each round, Λ is schedulable if and only if*

$$\forall t \in \mathcal{D},\ h_0(t) \le t \times B, \qquad (14)$$

where $\mathcal{D}$ is the set of deadlines in the interval $[0, T_b]$ of packets released by streams in Λ', $h_0(t)$ is the number of packets that have both release time and deadline in $[0, t]$, and $t \times B$ is the bandwidth available within $[0, t]$.

Design and implementation in Blink. Implementing Theorem 4 for admission control faces the same challenges as the start of round computation discussed in Sec. 5.2. Even though a closed-form expression of $h_0(t)$ exists (see Online Appendix D), using it would result in a performance hog on resource-constrained platforms due to many costly divisions. We thus perform admission control again by performing efficient priority queue operations rather than divisions, similar to Algorithm 1. The pseudocode of this algorithm can be found in Online Appendix D.

5.4. Scheduler Execution and Integration

At the end of a round, the algorithms for slot allocation, computation of the start of the next round, and admission control execute as shown in Fig. 10. With a pending request

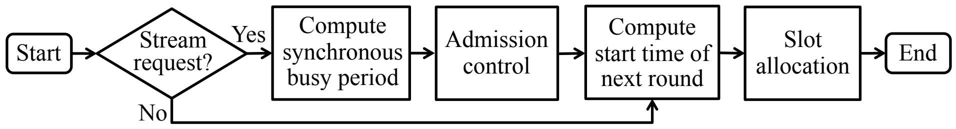

Start
Stream request?
Yes
No
Compute synchronous busy period
Admission control
Compute start time of next round
Slot allocation
End

Fig. 10. Main steps in Blink's real-time scheduler.

for a new stream s, the scheduler computes the (new) synchronous busy period for the stream set $\Lambda \cup \{s\}$ and checks if s can be admitted. Regardless of this, the scheduler computes the start time of the next round and then allocates slots to released packets.

In the worst case, one scheduler execution proceeds through all four steps in Fig. 10. To keep the scheduler's execution time under control, we limit the number of changes to the stream set that increase the required bandwidth to one per round, so admission control executes at most once. Conversely, we place no limit on the number of changes to the stream set that decrease the required bandwidth, as they do not require to run admission control. Overall, experiments in Sec. 6 show that our implementation can schedule hundreds of streams with a wide range of realistic bandwidth demands.

To deal with the reliability issues arising in the original LWB implementation when the time between rounds changes frequently, as explained in Sec. 3, after missing a schedule from the host, a node wakes up at *every* possible time instant at which a schedule packet could arrive. As rounds can only start at an integer multiple of the round length, a node selectively "scans" for schedule packets at the possible beginning and end of rounds. Compared to the strategy employed by the original LWB implementation, this approach significantly reduces both the time until a node can again participate in the communication and the energy required to search for the next schedule.

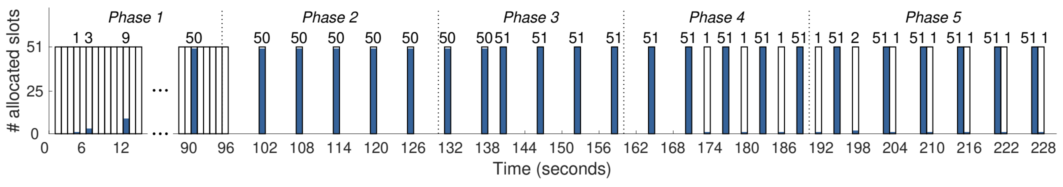
6. EVALUATION

We evaluate Blink along four lines: (*i*) its adaptability to changes in the set of streams,[3] (*ii*) its real-time service in terms of packet delivery ratio and meeting deadlines ranging from 120 sec to 100 ms, also compared to the original LWB scheduler, (*iii*) the communication energy efficiency of the different real-time scheduling policies, and (*iv*) the efficiency of our bucket queue-based implementation of the optimal **LS** policy.

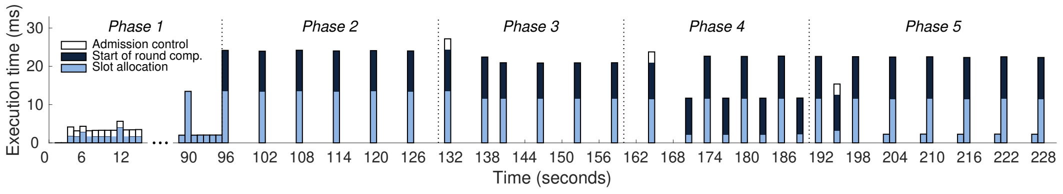
To this end, we implement Blink according to the processing shown in Fig. 10 on top of the Contiki operating system [Contiki 2011] for the TelosB and the CC430 system-on-chip (SoC) platforms. Both feature MSP430 MCUs. We perform experiments on the FlockLab [Lim et al. 2013] and w-iLab.t [Bouckaert et al. 2011] testbeds with up to 94 nodes, on two other platforms featuring state-of-the-art ARM Cortex-M0/M4 cores, in a time-accurate instruction-level emulator [Eriksson et al. 2009], and through synthetic simulations. Our results reveal the following:

— Blink promptly adapts to dynamic changes in the set of streams (*i.e.*, application requirements) without unnecessarily increasing communication energy consumption.
— On a 94-node testbed, Blink meets all deadlines of received packets, while successfully delivering 99.97 % of the packets. The few packet losses can be effectively handled by CPS controllers [Sinopoli et al. 2004; Zimmerling et al. 2013].
— **LS** reduces communication energy costs by up to 2.5× compared with **CS** and **GS**.
— In a 4-hop network with 9 sources, Blink supports 100 ms deadlines under full load.
— Simulations show that, unlike Blink, the original LWB scheduler misses on average 28–65 % of deadlines in addition to the (unavoidable) packet losses over wireless.

[3]We do not evaluate Blink's adaptability to network state changes (due to interference, node mobility, etc.), because these changes are seamlessly handled by the underlying LWB, as shown by Ferrari et al. [2012].



(a) Number of allocated slots in each round, for $B = 51$ available slots per round.



(b) Breakdown of scheduler execution time in each round.

Fig. 11. Trace of Blink scheduling streams with varying real-time requirements on FlockLab. *After bootstrapping the network, Blink uses **LS** to save energy, while meeting all deadlines of the changing streams.*

— Our bucket queue-based implementation of the **LS** scheduler executes up to 4.1× faster than a conventional **LS** implementation on popular MCUs.

Unfortunately, comparing Blink with other real-time protocols in a running network is oftentimes extremely difficult, if not impossible. Considering a protocol that is unable to guarantee *hard* real-time requirements or designed only for specific traffic patterns would be discriminatory. On the other hand, protocols designed for the same kind of real-time requirements and traffic patterns as Blink usually offer no complete open-source implementation or are incapable to run on the same hardware as Blink. WirelessHART open-source implementations, for example, only comprise the functionality that runs *inside* the network. The scheduling process, however, executes on the central network manager, which is sold as a black box (*e.g.*, by Dust Networks[4]) with no inside visibility or possibility of instrumentation to gather the needed measurements.

6.1. Adaptability to Changes in the Set of Streams

Blink promptly adapts to dynamic changes in the stream set without affecting existing streams and while maintaining efficient performance. By contrast, in WirelessHART, for example, such changes tend to be disruptive [Zhang et al. 2009], and it takes much longer to re-gain a condition of efficient performance [Åkerberg et al. 2011b].

We use 29 TelosB nodes on FlockLab [Lim et al. 2013], with a diameter of 5 hops. One node acts as the host running the scheduler, and three randomly chosen nodes are destinations. The remaining 25 nodes act as sources generating $2\langle 0, 6, 6\rangle$ streams each. We let two sources eventually request and update a third and a fourth stream with different parameters. The number of slots in a round is $B = 51$, leaving 100 ms to compute the schedule at the end of a round. The length of a round is set to 1 sec.

Execution. Fig. 11a shows the number of slots allocated in each round, while Fig. 11b shows a breakdown of the execution time of the **LS** scheduler in each round.

In *Phase 1*, the system is bootstrapping, so Blink schedules rounds contiguously to allow all nodes to quickly time-synchronize with the host and to submit their stream requests in the contention slot. This happens for the first time after 3 sec, as visible in Fig. 11b from the increase in processing time for admission control and slot allocation.

[4]http://www.linear.com/products/smartmesh_wirelesshart

During the following rounds, the host gradually receives all initial stream requests and, consequently, admission control and slot allocation take longer.

In *Phase 2*, because no new stream request recently arrived, Blink adapts its functioning to the normal operation and starts to dynamically compute the start time of rounds using **LS**. Fig. 11b shows that it takes about 10 ms to do so. In these conditions, Blink schedules a round every 6 sec, postponing rounds until right before the packets' deadlines, which minimizes the energy overhead of the underlying LWB.

At the start of *Phase 3*, a request for a new stream $\langle 0, 6, 3\rangle$ arrives. Admission control executes at $t = 131$ sec as visible in Fig. 11b. The new stream is admitted and accounted for starting from $t = 140$ sec. Rounds are scheduled again every 6 sec and with all $B = 51$ available slots allocated. Unlike existing systems, Blink accommodates a a change in application requirements represented as a new stream without jeopardizing the existing ones, and still maintains minimum energy overhead. In WirelessHART, for example, changes in the stream set tend to be way more disruptive, likely affecting existing streams [Zhang et al. 2009], and thus take much longer to accommodate. For instance, Åkerberg et al. [2011b] report that it may take WirelessHART up to 30 min to reconfigure the routing topology after a change in the network.

In *Phase 4*, another request for a stream $\langle 0, 6, 6\rangle$ arrives and passes admission control. Blink allocates the first slot to the new stream at $t = 170$ sec. However, now there are 52 pending packets, one more than the $B = 51$ available slots. Due to this, Blink schedules the following rounds every 3 sec, with the number of allocated slots alternating between 51 and 1. This shows how Blink seamlessly copes with dynamic changes in the stream set, which may result in drastic changes in its runtime operation.

In *Phase 5*, the node that requested the stream in *Phase 3* extends its deadline from 3 to 6 sec. Thus, the 52 streams now all have the same deadline and period. Because 52 packets do not fit in a single round, Blink schedules a complete round with 51 allocated slots 2 sec before the packets' deadlines, followed by another round for the remaining packet. This shows that even a minor change in the profile of one stream may have a significant impact on how rounds unfold over time, which Blink effectively handles.

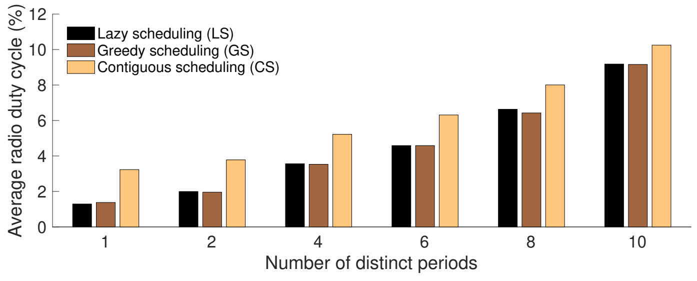
6.2. Packet Deadlines and Energy Consumption

We assess Blink's ability to meet packet deadlines and the related energy costs.

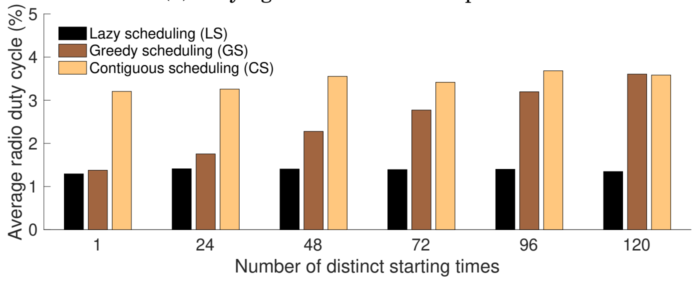
Metrics and settings. We examine the performance along two key dimensions [Saifullah et al. 2010]. The *deadline success ratio* is the fraction of packets that meet their deadlines, indicating the level of real-time service. We compute this figure based on sequence numbers embedded into packets and timestamps taken at both communication ends. The *radio duty cycle* is the fraction of time a node has the radio on, which is widely used as a proxy for energy consumption [Gnawali et al. 2009]. This metric indicates the energy cost of providing a certain level of real-time service. We measure radio duty cycles in software using Contiki's power profiler [Contiki 2011].

We run experiments with 94 TelosB nodes on the w-iLab.t testbed [Bouckaert et al. 2011]. The network has a diameter of 6 hops. We let 90 nodes act as sources, one as the host, and 3 as destinations, mimicking a scenario with multiple controllers or multiple actuators [Paavola and Leiviska 2010]. Each source generates two streams, hence a total of 180 streams generate packets with a 10-byte payload. The number of slots in a round is again $B = 51$, and so is the length of a round, which remains 1 sec.

We run two types of experiments. First, we set all starting times S_i to zero and vary the number of distinct periods in different runs, as in configurations combining primary and secondary control [Ogata 2001]. This way, we generate varying bandwidth demands between 2.9% and 19.4%. Then, we set the period P_i of all streams to 2 minutes and vary the number of distinct starting times. This models situations where, for



(a) Varying number of distinct periods.



(b) Varying number of distinct starting times.

Fig. 12. Average radio duty cycle of Blink with **LS**, **GS**, and **CS** on the w-iLab.t testbed. *Depending on the stream set, **LS** achieves up to a 2.5× reduction in energy consumption compared to the **GS** and **CS** policies.*

example, sources are progressively added to a running system [Paavola and Leiviska 2010]. Deadlines are equal to periods. Each run lasts for 50 min. We start measuring after 20 minutes to give nodes enough time to submit their stream requests.

Results. The average deadline success ratio is 99.97 %, with a minimum of 99.71 % in a single run. These figures are noteworthy in at least two respects. First, most modern control applications, including the ones we mentioned in Sec. 2, can and do tolerate such small fraction of packets not meeting their deadlines [Åkerberg et al. 2011a]. We thus demonstrate that Blink can effectively operate in several of these scenarios. Second, we verify that *deadline misses are entirely due to losses over the wireless channel*, a phenomenon that is orthogonal to real-time scheduling and cannot be fully avoided. These results thus confirm the reasoning and theoretical results from Sec. 5.

Fig. 12a shows the average radio duty cycle across all nodes. The energy costs generally increase with the number of distinct periods, since the bandwidth demand increases as well. Differences among the policies stem from scheduling fewer rounds. **LS** and **GS** perform similarly: since all streams start at the same time and because of the choice and distribution of periods, **LS** has little opportunity to spare more rounds than **GS**. Nonetheless, both **LS** and **GS** significantly improve over **CS**: they need 2.5× less energy when all streams have the same period. The gap shrinks to 1.2× with 10 distinct periods, mostly because the energy overhead of Blink has a smaller impact on the total energy costs at higher bandwidth demands.

Fig. 12b shows the average radio duty cycle as the number of distinct starting times increases. The bandwidth demand is constant, so **CS** always consumes the same energy. This time, however, **LS** and **GS** perform differently. The energy costs of **GS** increase as the number of distinct starting times increases, because packets are released at increasingly different times and thus **GS** schedules more rounds. **LS**, instead, greatly benefits from aggregating packets over subsequent release times and sending them in the same round. As a result, the energy costs of **LS** remain low and constant, whereas the energy costs of **GS** approach those of **CS**.

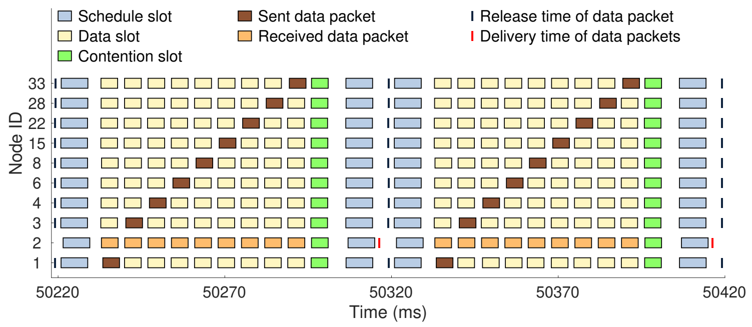


Fig. 13. Trace of Blink running on FlockLab at full load while supporting periods and deadlines of 100 ms. *A 5 ms slot is sufficient to schedule packets in this demanding scenario, and all of them meet their deadline.*

These results show that **LS** is most energy-efficient irrespective of the stream set, achieving several-fold improvements over **GS** and **CS** in some settings. With a few distinct periods and starting times, **GS** might also be an option in that it reduces the packet latency by sending packets as soon as possible.

6.3. Supporting Sub-second Periods and Deadlines under Full Load

At the other end of the settings in Sec. 6.2 are, for example, interlocking and closed-loop control. Typically, such scenarios include a small number of nodes exchanging packets over a few hops subject to deadlines in the 10–500 ms range [Åkerberg et al. 2011a]. We show that Blink equally caters for such scenarios.

Settings. To this end, we use 10 CC430 devices across 4 hops on FlockLab [Lim et al. 2013]. To emulate a typical closed-loop control setting [Åkerberg et al. 2011a], we let 9 nodes each source a stream with $D_i = P_i = 100$ ms; one node acts as the destination. We dimension Blink accordingly by setting the length of a round to 100 ms. Given this time budget, we can effectively secure the $B = 9$ data slots necessary in every round to serve all streams, and can afford up to 5 ms to compute the schedule. We use FlockLab's GPIO tracing service [Lim et al. 2013] to log the start and end of slots, and when packets are released, sent, received, and finally delivered.

Results. Fig. 13 shows a 200 ms snapshot of the trace we obtain. We see each source releases a packet every $P_i = 100$ ms, which is delivered to the destination within the stated deadline. For example, packets released at $t = 50220$ ms are sent during the following sequence of data slots; the destination (node 2) receives and delivers all packets to the application shortly before time $t = 50320$ ms. Since $D_i = P_i = 100$ ms for all streams, Blink immediately starts the next round and allocates all slots. Thus, the system is running at full load, that is, the streams demand 100 % of the bandwidth.

We can make two key observations: (*i*) using our efficient priority queue implementation, a 5 ms slot suffices to schedule packets in a demanding scenario with short deadlines under full load; (*ii*) despite this, all packets consistently meet their deadlines. This provides evidence of Blink's ability to reach into challenging application scenarios, such as interlocking and closed-loop control, yet even shorter timescales would be possible if the physical layer were to provide higher data rates.

6.4. Comparison against the Original LWB Scheduler

Our work adds real-time communication capabilities to the non real-time LWB protocol. To quantify the gap we fill, we compare the deadline success ratio of Blink's **LS** scheduler with that of the original LWB scheduler (**OS**) [Ferrari et al. 2012].

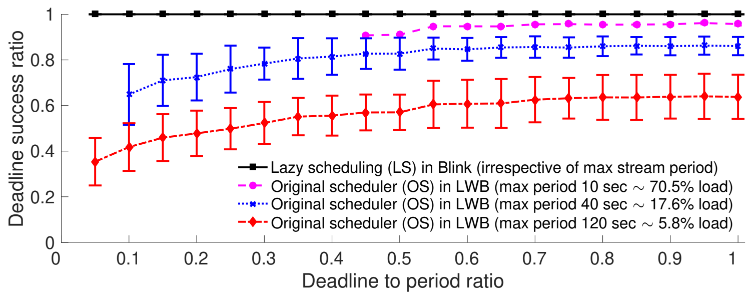


Fig. 14. Deadline success ratio against deadline to period ratio for Blink's **LS** scheduler and the original LWB scheduler (**OS**) for different bandwidth demands. *The realtime-optimal **LS** scheduler meets all deadlines across all settings, while **OS** meets on average only 35–72 % of deadlines for 5.8 % bandwidth demand.*

Setting. We consider steady-state conditions, that is, the application has already issued all stream requests and these have reached the host. Under these conditions, and barring packet losses that would equally affect **LS** and **OS**, the deadline success ratio is solely determined by the schedulers' decisions. Thus, we synthetically simulate the execution of **LS** and **OS** for a given schedulable stream set and across a given time interval, examining the fraction of packets that meet their deadline.

We consider sets of 180 streams with $S_i = 0$. For each set, we randomly generate the periods P_i uniformly between 1 sec and a given maximum (10, 40, or 120 sec), yielding average bandwidth demands of 70.5 %, 17.6 %, and 5.8 %. Deadlines are set to $D_i = \lceil \rho P_i \rceil$, where $0 < \rho \leq 1$ is the *deadline to period ratio* determining the tightness of deadlines compared to periods. For a given maximum period, we generate 100 different stream sets with random periods, and for each set we simulate **LS** and **OS** for varying ρ. We report deadline success ratios after simulating the execution for 10 min.

Results. Fig. 14 plots average and standard deviation of deadline success ratio for **LS** and **OS** against ρ for different bandwidth demands. **LS** meets 100 % of deadlines across the board, as we design it to be realtime-optimal. **OS**, however, meets on average only 35–72 % of deadlines for 5.8 % bandwidth demand. This is because **OS** is oblivious of deadlines in its scheduling decisions: it uses a fixed round period chosen as long as possible *only* based on the periods to save energy, while providing just enough bandwidth to transmit all packets [Ferrari et al. 2012]. This also explains why **OS** misses more deadlines for smaller ρ: the shorter the streams' deadlines compared to their periods, the less effective are **OS**'s scheduling decisions in terms of meeting deadlines. For higher bandwidth demands, **OS** needs to schedule more rounds to provide enough bandwidth, which has the (unintentional) side benefit that **OS** misses fewer deadlines.

6.5. Scheduler Execution Time

We look at the efficiency of our bucket queue-based **LS** implementation in Blink. The scheduler's execution time is critical as it affects the available bandwidth (see Sec. 4).

Method. In the worst case, a single scheduler execution must proceed through all four steps in Fig. 10. A careful analysis of the relevant algorithms reveals that the execution time of **LS** increases with the number of streams n, the largest possible period of any stream $\overline{P}$, the bandwidth demand of the streams, and the synchronous busy period T_b.

Precisely quantifying how the combination of these factors determines the running time of **LS** is non-trivial. We opt for an empirical approach that confidently approximates the worst case. As described in more detail in Online Appendix E, we use 200 streams—the maximum that fits into the 10 kB of RAM on a TelosB with our Blink prototype—and determine for a given bandwidth demand stream profiles with periods

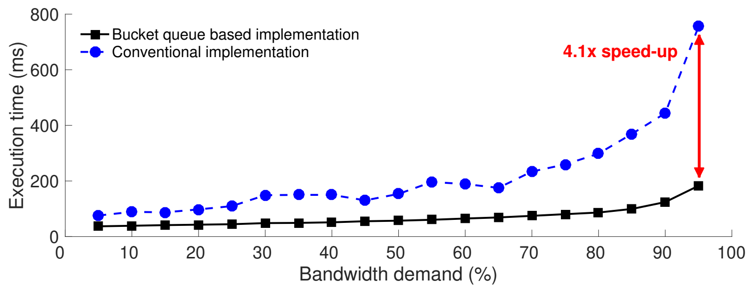


Fig. 15. Execution time of **LS** against bandwidth demand on a 16-bit MSP430 clocked at 4 MHz, for a bucket queue-based implementation and a conventional one. *The bucket queue-based implementation consistently outperforms the conventional approach based on analytic computations, achieving speed-ups of up to 4.1*$\times$.

no larger than $\overline{P} = 255$ sec such that T_b is maximum. In this way, we obtain "worst-case" stream sets for bandwidth demands between 5 % and 95 %, in 5 % steps.

To assess the effectiveness of our bucket queue-based implementation of the algorithms required in Blink, we implement as a baseline the first three steps in Fig. 10 following the conventional approach, based on analytic computations like those in (2). This includes an implementation of the fastest analytic EDF schedulability test known today [Zhang and Burns 2009] for admission control. We benchmark both implementations in a 2.5 h execution of Blink, where 200 streams are initially admitted one after the other, and measure the execution time of the different steps in Fig. 10.

We test three different MCUs: a 16-bit MSP430F1611 running at 4 MHz, which represents the class of MCUs currently used to target the lowest possible energy consumption in the CPS applications of Sec. 2; a 32-bit ARM Cortex-M0 clocked at 48 MHz; and a 32-bit ARM Cortex-M4 running at 72 MHz. The ARM cores offer higher processing power at higher energy consumption, yet they might be a viable option if some energy overhead can be traded for better computing capabilities [Ko et al. 2012].

We use msp430-gcc v4.6.3 for the MSP430 and IAR build tools for the two ARM cores; we choose the highest optimization level that makes the binaries still fit into program memory. We deploy the binaries in the MSPsim time-accurate instruction-level emulator [Eriksson et al. 2009] and on evaluation boards from STMicroelectronics for the ARM cores. Execution times are measured in software with microsecond accuracy.

Results. Fig. 15 plots the total execution time of the two scheduler implementations on the MSP430 as the bandwidth demand increases from 5 % to 95 %. The time increases slightly in the beginning, but ramps up severely for the conventional implementation as the bandwidth demand exceeds 65 %. This is due to an increase in the times needed for synchronous busy period computation, admission control, and start of round computation, whereas the time needed for slot allocation remains almost constant.

Our bucket queue-based implementation consistently outperforms the conventional one, culminating in a 4.1$\times$ speed-up at 95 % bandwidth demand. For this demand, the reduced execution time (182 ms *vs.* 756 ms) means there is space for 44 instead of only 3 data slots per round. Fictitiously simulating the system's evolution using an efficient bucket queue implementation as we do, rather than explicitly performing analytic computations, is thus instrumental to the viability of realtime-optimal scheduling using **LS** on this class of devices. At the same time, **LS** ensures minimal communication energy consumption within the limits of the underlying LWB.

Despite approximating the worst-case execution time of **LS**, the stream sets above are not often seen in *low-power* applications. Our review in Sec. 2 indicates that the typical demands would rarely exceed 20 % of the maximum available bandwidth. Under this regime, we measure execution times below 43 ms with the bucket queue-based implementation: well below the upper bound of 100 ms in our Blink prototype. Thus, due to a $2.3\times$ speed-up over the conventional implementation in this regime, there is plenty of room for employing more constrained ultra-low-power platforms, or for considerably scaling up the number of streams with more available memory.

These observations also hold for the ARM cores. As expected, the conventional implementation benefits from the richer instruction sets, in particular on the Cortex-M4, which features SIMD instructions and a hardware divide. Therefore, we consistently measure scheduler execution times below 30 ms. Nevertheless, our bucket queue-based implementation achieves speed-ups of 1.6–$2\times$ on both cores for realistic bandwidth demands of up to 20 %. This is mostly because using the bucket queues, the next time t that the loop in Algorithm 1 should examine is readily available due to the EDF-based ordering of streams. Instead, using the conventional approach, the next time t must be explicitly computed, which costs as much as computing $h_i(t)$ via (2).

We therefore demonstrate that a bucket queue-based implementation of **LS** is beneficial even on less constrained state-of-the-art platforms. In general, faster processing may either allow one to increase the bandwidth by reducing the time allocated to the scheduling step in a round, or to handle more streams with the same time allocated to scheduler execution. By providing a further reduction of the time required for scheduling, our implementation thus amplifies the benefits of faster processors.

7. CONCLUSIONS

Blink supports hard real-time communication in large wireless multi-hop networks at low energy costs. It overcomes fundamental limitations of prior art in terms of scalability and adaptivity to changes in application requirements and network state. Our experiments demonstrate that Blink meets all deadlines of received packets, successfully delivers 99.97 % of packets regardless of such changes, and consumes minimum energy within the limits of the underlying LWB communication support. This performance applies to periods and deadlines down to 100 ms, while significantly improving over the original LWB scheduler, which would miss a large fraction of deadlines. Our efficient priority queue data structure enables speed-ups of up to $4.1\times$ over a conventional scheduler implementation based on analytic computations on popular low-power microcontrollers. We thus maintain that Blink provides a key stepping stone towards the adoption of low-power wireless technology in mission-critical CPS applications.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to thank Reto Da Forno for his help with some of the experiments, and Jan Beutel, Tonio Gsell, and Roman Lim for their excellent technical support.

References

Yuvraj Agarwal, Bharathan Balaji, Seemanta Dutta, Rajesh K. Gupta, and Thomas Weng. 2011. Duty-Cycling Buildings Aggressively: The Next Frontier in HVAC Control. In *Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN)*.

Johan Åkerberg, Mikael Gidlund, and Mats Björkman. 2011a. Future Research Challenges in Wireless Sensor and Actuator Networks Targeting Industrial Automation. In *Proc. of the IEEE Int. Conf. on Industrial Informatics (INDIN)*.

Johan Åkerberg, Frank Reichenbach, Mikael Gidlund, and Mats Björkman. 2011b. Measurements on an Industrial WirelessHART Network Supporting PROFIsafe: A Case Study. In *Proc. of the IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*.

Nouha Baccour, Anis Koubâa, Luca Mottola, Marco Antonio Zúñiga, Habib Youssef, Carlo Alberto Boano, and Mário Alves. 2012. Radio Link Quality Estimation in Wireless Sensor Networks: A survey. *ACM Trans. Sen. Netw.* 8, 4 (2012), 1–34.

Stefan Bouckaert, Wim Vandenberghe, Bart Jooris, Ingrid Moerman, and Piet Demeester. 2011. The w-iLab.t Testbed. In *Proc. of the ICST Int. Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*.

Gerth Stølting Brodal. 2013. A Survey on Priority Queues. In *Space-Efficient Data Structures, Streams, and Algorithms*. Springer-Verlag, 150–163.

Randy Brown. 1988. Calendar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM* 31, 10 (1988), 1220–1227.

Giorgio Buttazzo. 2005. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems* 29, 1 (2005), 5–26.

Giorgio Buttazzo. 2011. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Vol. 24. Springer Science & Business Media.

CAN. 2004. ISO 11898-4:2004–Road vehicles–Controller area network (CAN)–Part 4: Time-triggered communication. (2004).

Octav Chipara, Chenyang Lu, Thomas C. Bailey, and Gruia-Catalin Roman. 2010. Reliable Clinical Monitoring using Wireless Sensor Networks: Experiences in a Step-down Hospital Unit. In *Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys)*.

Octav Chipara, Chenyang Lu, and Gruia-Catalin Roman. 2013. Real-time Query Scheduling for Wireless Sensor Networks. *IEEE Trans. Comput.* 62, 9 (2013).

Octav Chipara, Chengjie Wu, Chenyang Lu, and William Griswold. 2011. Interference-Aware Real-Time Flow Scheduling for Wireless Sensor Networks. In *Proc. of the Conf. on Real-Time Systems (ECRTS)*.

Contiki. 2011. Contiki: The Open Source OS for the Internet of Things. (2011). `http://www.contiki-os.org/`

Robert B. Dial. 1969. Algorithm 360: Shortest-Path Forest with Topological Ordering [H]. *Commun. ACM* 12, 11 (1969), 632–633.

Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, and Thiemo Voigt. 2009. COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks. In *Proc. of the EIA Int. Conf. on Simulation Tools and Techniques (SIMUTools)*.

Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. 2012. Low-Power Wireless Bus. In *Proc. of the ACM Conference on Embedded Network Sensor Systems (SenSys)*.

Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. 2011. Efficient Network Flooding and Time Synchronization with Glossy. In *Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN)*.

FlexRay. 2013. ISO 17458-1:2013–Road vehicles–FlexRay communications system–Part 1: General information and use case definition. (2013).

Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. 2009. Collection Tree Protocol. In *Proc. of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*.

Yu Gu, Tian He, Mingen Lin, and Jinhui Xu. 2009. Spatiotemporal Delay Control for Low-Duty-Cycle Sensor Networks. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*.

Tian He, John A. Stankovic, Chenyang Lu, and Tarek F. Abdelzaher. 2005. A Spatiotemporal Communication Protocol for Wireless Sensor Networks. *IEEE Trans. Parallel Distrib. Syst.* 16, 10 (2005), 995–1006.

Honeywell. 2006. Choosing the Right Industrial Wireless Network. (2006). `https://www.honeywellprocess.com/library/support/Public/Documents/WirelessWhitePaper_Nov2006.pdf`

IEEE 802.15.4e TSCH. 2012. 802.15.4e-2012–IEEE Standard for local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer. (2012).

ISA100. 2009. Wireless Compliance Institute. (2009).

Vikram Kanodia, Chengzhi Li, Ashutosh Sabharwal, Bahareh Sadeghi, and Edward Knightly. 2001. Distributed Multi-hop Scheduling and Medium Access with Delay and Throughput Constraints. In *Proc. of the ACM Int. Conf. on Mobile Computing and Networking (MobiCom)*.

JeongGil Ko and others. 2012. Low Power or High Performance? A Tradeoff Whose Time Has Come (and Nearly Gone). In *Proc. of the European Conf. on Wireless Sensor Networks (EWSN)*.

Krijn Leentvaar and Jan H. Flint. 1976. The Capture Effect in FM Receivers. *IEEE Trans. Commun.* 24, 5 (1976), 531–539.

Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. 2013. FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems. In *Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN)*.

Chung Laung Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (1973), 46–61.

Ke Liu, Nael Abu-Ghazaleh, and K-D Kang. 2006. JiTS: Just-in-time scheduling for real-time sensor data dissemination. In *Proc. of the Int. Conf. on Pervasive Computing and Communications (PERCOM)*.

Chenyang Lu, Brian M. Blum, Tarek F. Abdelzaher, John A. Stankovic, and Tian He. 2002. RAP: A Real-Time Communication Architecture for Large-Scale Wireless Sensor Networks. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Ingo Molnar. 2015. The Linux Completely Fair Scheduler (CFS). (2015). `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`

Luca Mottola, Mattia Moretta, Kamin Whitehouse, and Carlo Ghezzi. 2014. Team-level Programming of Drone Sensor Networks. In *Proc. of the ACM Conf. on Embedded Network Sensor Systems (SenSys)*.

S. M. Shahriar Nirjon, John A. Stankovic, and Kamin Whitehouse. 2010. IAA: Interference Aware Anticipatory Algorithm for Scheduling and Routing Periodic Real-time Streams in Wireless Sensor Networks. In *Proc. of the IEEE Int. Conf. on Networked Sensing Systems (INSS)*.

Tony O'Donovan and others. 2013. The GINSENG System for Wireless Monitoring and Control: Design and Deployment Experiences. *ACM Trans. Sen. Netw* 10, 1 (2013), 1–40.

Katsuhiko Ogata. 2001. *Modern Control Engineering* (4th ed.). Prentice Hall.

M. Paavola and K. Leiviska. 2010. *Wireless Sensor Networks in Industrial Automation*. Springer-Verlag.

Ben Pfaff. 2004. An Introduction to Binary Search Trees and Balanced Trees. (2004). `http://adtinfo.org/libavl.html/`

Joseph Polastre, Robert Szewczyk, and David Culler. 2005. Telos: Enabling Ultra-Low Power Wireless Research. In *Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN)*.

Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. 2010. Real-Time Scheduling for WirelessHART Networks. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*.

Lui Sha, Tarek F. Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. 2004. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems* 28, 2-3 (2004), 101–155.

Bruno Sinopoli, Luca Schenato, Massimo Franceschetti, Kameshwar Poolla, Michael I. Jordan, and Shankar S. Sastry. 2004. Kalman Filtering with Intermittent Observations. *IEEE Trans. Automat. Control* 49, 9 (2004), 1453–1464.

Marco Spuri. 1996. *Analysis of Deadline Scheduled Real-Time Systems*. Technical Report 2772. INRIA.

Kannan Srinivasan, Prabal Dutta, Arsalan Tavakoli, and Philip Levis. 2010. An Empirical Study of Low-power Wireless. *ACM Trans. Sen. Netw.* 6, 2 (2010), 1–49.

John A. Stankovic, Tarek F. Abdelzaher, Chenyang Lu, Lui Sha, and Jennifer C. Hou. 2003. Real-Time Communication and Coordination in Embedded Sensor Networks. *Proc. IEEE* 91, 7 (2003), 1002–1022.

John A. Stankovic, Insup Lee, Aloysius Mok, and Raj Rajkumar. 2005. Opportunities and Obligations for Physical Computing Systems. *IEEE Computer* 38, 11 (2005), 23–31.

John A. Stankovic, Krithi Ramamritham, and Marco Spuri. 1998. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers.

Petcharat Suriyachai, James Brown, and Utz Roedig. 2010. Time-Critical Data Delivery in Wireless Sensor Networks. In *Proc. of the Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*.

WirelessHART. 2007. WirelessHART. (2007). `http://en.hartcomm.org/main_article/wirelesshart.html`

Feng Xia, Yu-Chu Tian, Yanjun Li, and Youxian Sung. 2007. Wireless Sensor/Actuator Network Design for Mobile Control Applications. *Sensors* 7, 10 (2007), 2157–2173.

Fengxiang Zhang and Alan Burns. 2009. Schedulability Analysis for Real-Time Systems with EDF Scheduling. *IEEE Trans. Comput.* 58, 9 (2009), 1250–1258.

Haibo Zhang, Pablo Soldati, and Mikael Johansson. 2009. Optimal Link Scheduling and Channel Assignment for Convergecast in Linear WirelessHART Networks. In *Proc. of the Int. Symp. on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOPT)*.

Marco Zimmerling, Federico Ferrari, Luca Mottola, and Lothar Thiele. 2013. On Modeling Low-power Wireless Protocols Based On Synchronous Packet Transmissions. In *Proc. of the IEEE Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

Marco Zimmerling, Luca Mottola, Pratyush Kumar, Federico Ferrari, and Lothar Thiele. 2016. *Adaptive Real-time Communication for Wireless Cyber-physical Systems*. Technical Report. ETH Zurich. `ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-356.pdf`

Received March 2016; revised September 2016; accepted October 2016

Online Appendix to: Adaptive Real-time Communication for Wireless Cyber-physical Systems

MARCO ZIMMERLING, TU Dresden
LUCA MOTTOLA, Politecnico di Milano and SICS Swedish ICT
PRATYUSH KUMAR, FEDERICO FERRARI, and LOTHAR THIELE, ETH Zurich

A. VALIDITY OF ALGORITHMS

Throughout Sec. 5, we consider a discrete time model in which (*i*) each round is atomic and of unit length, and (*ii*) each round starts at an integer multiple of the unit length of a round, as illustrated in Fig. 3.

The reasoning behind (*i*) is that the single MCU on today's low-power wireless platforms is responsible for both application processing and interacting with the radio (*e.g.*, to transfer a packet to the radio's transmit buffer and start a transmission). These radio interactions are time-critical and occur frequently during a Glossy flood [Ferrari et al. 2012]. Because each slot in a round consists of a Glossy flood, the MCU essentially has no time for application processing during a round. As a result, the application must release packets before a round starts and can only handle received packets after a round. We therefore consider rounds atomic. (*ii*) is beneficial in a practical Blink implementation. For example, it allows a node that got out-of-sync to *selectively* turn on the radio in order to receive the next schedule transmitted by the host (and thereby synchronize again) rather than keeping the radio on all the time, which consumes more energy but is unavoidable if rounds can start at arbitrary times. Blink exploits this opportunity, as mentioned in Sec. 5.4.

Although the presentation in Sec. 5 is based on this discrete-time model, our analysis and algorithms are also valid for streams with start times S_i, periods P_i, and deadlines D_i that are not integer multiples of the unit length of a round. For example, fractional packet release times are simply postponed to the next discrete time (by taking the ceiling) and fractional packet deadlines are preponed to the previous discrete time (by taking the floor). Thus, the atomicity of rounds does not prevent any stream from meeting its deadlines. This is essential to the validity of our EDF-based scheduling policies in that "preemptions" in the execution of the underlying resource (*i.e.*, the entire network which we abstract as a single communication resource that runs on a single clock) can only occur at discrete times.

B. COMPUTATION OF SLOT ALLOCATION

Algorithm 2 shows the pseudocode to allocate as many pending packets as possible to the B available slots in the next round. The algorithm operates on the current set of streams, maintained in a bucket queue in order of increasing absolute deadline. Starting from the stream with the smallest (earliest) absolute deadline, it visits streams in EDF order through repeated NEXT(t) calls. Whenever it sees a stream t with a pending packet, it allocates a slot to stream t and updates its priority in the queue using DECREASEKEY(t, $t.period$). The algorithm stops when all B available slots are allocated, or when it sees a stream with an absolute deadline larger than the $horizon$.

This second termination criterion using the horizon is needed as there may be less than B pending packets by the time the next round starts. Algorithm 2 determines the

© 2016 Copyright held by the owner/author(s). 2378-962X/2016/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/3012005

Algorithm 2 EDF-based slot allocation

```
Input  A bucket queue of the current set of streams $\Lambda$ (smaller $absDeadline$ means higher priority), the start time of the next round $t_{i+1}$, the upper bound on any stream's period $\overline{P}$, and the number of data slots available in a round $B$.
Output  Allocation of packets that are pending by time $t_{i+1}$ in EDF order to the available slots.
  initialize slot counter $c$ to 0
  position traverser $t$ at highest-priority stream using FIRST($t$)
  $horizon = t.absDeadline + \overline{P} - 1$
  while ($c < B$) and ($t.absDeadline \leq horizon$) do
      if $t.releaseTime \leq t_{i+1}$ then
          allocate a slot to stream $t$ and set $c = c + 1$
          $t.releaseTime = t.releaseTime + t.period$
          $t.absDeadline = t.absDeadline + t.period$
          update $t$'s priority using DECREASEKEY($t$, $t.period$)
      end if
      advance traverser $t$ to next stream in EDF-order using NEXT($t$)
  end while
```

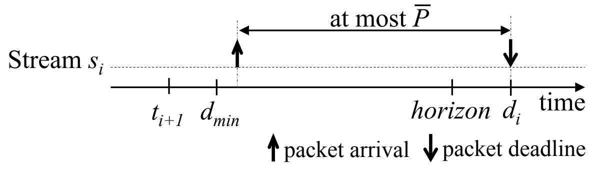


Fig. 16. Illustration of the second termination criterion in Algorithm 2. *Any stream s_i with an absolute deadline d_i greater than $horizon = d_{min} + \overline{P} - 1$ releases its current packet after the start of the next round at time t_{i+1} and therefore need not be considered for slot allocation.*

horizon initially, $horizon = d_{min} + \overline{P} - 1$, where d_{min} is the earliest absolute deadline of all streams at this time and $\overline{P}$ is an upper bound on the period of any stream. As shown in Fig. 16, the next round starts before d_{min}, that is, $t_{i+1} \leq d_{min} - 1$. Thus, stream s_i with absolute deadline $d_i > horizon$ releases its current packet no earlier than

$$d_i - \overline{P} > d_{min} + \overline{P} - 1 - \overline{P} \geq t_{i+1}. \tag{15}$$

The strict inequality in (15) implies that s_i releases its current packet only after the start of the next round. Thus, stream s_i need not be considered for slot allocation in the next round. As Algorithm 2 visits streams in EDF order, it can terminate when it sees the first such stream.

C. SYNCHRONOUS BUSY PERIOD COMPUTATION

The synchronous busy period T_b is crucial to admission control and computing the start time of the next round in **LS**. It denotes the time needed to contiguously serve the maximum demand that a given set of streams creates when all streams release a packet at the same time. The real-time literature suggests computing the synchronous busy period T_b through an iterative process [Stankovic et al. 1998]

$$\omega^0 = \frac{n}{B} \quad \text{and} \quad \omega^{m+1} = \frac{1}{B}\sum_{i=1}^{n}\left\lceil \frac{\omega^m}{P_i} \right\rceil \tag{16}$$

which terminates when $\omega^{m+1} = \omega^m$; then, $T_b = \lceil \omega^m \rceil$.

As discussed in Sec. 5.2, implementing this iterative method on a resource-constrained platform leads to prohibitive running times due to many costly divisions. To overcome this problem, we compute T_b by *simulating* the execution of **CS**, which

Algorithm 3 Compute synchronous busy period

```
Input  A bucket queue that is a deep copy of the set of streams $\Lambda$, where $s.releaseTime$ is initial-
    ized to 0 for all streams $s_i$ and streams with smaller $releaseTime$ are given higher priority.
Output  The synchronous busy period $T_b$ of the set of streams.
  initialize $T_b$ and slot counter $c$ to 0
  set $s$ to highest-priority stream using $s = \text{FINDMIN}()$
  while ($s.releaseTime = 0$)
  or ($s.releaseTime < T_b$ and $c = 0$)
  or ($s.releaseTime \leq T_b$ and $c > 0$) do
      if current round has only one free slot (i.e., $c = B - 1$) then
          set $T_b = T_b + 1$ and $c = 0$ to "start" a new round
      else
          set $c = c + 1$ to "allocate" a slot in the current round
      end if
      $s.releaseTime = s.releaseTime + s.period$
      update priority of $s$ using $\text{DECREASEKEY}(s, s.period)$
      set $s$ to highest-priority stream using $s = \text{FINDMIN}()$
  end while
  if current round has at least one allocated slot (i.e., $c > 0$) then
      set $T_b = T_b + 1$ to "round up" to the next discrete time
  end if
```

essentially entails going through the same processing that underlies (16) in a step-by-step manner. To this end, we trigger the maximum demand by letting all streams release a packet at time $t = 0$. Using **CS**, we then serve this demand "as fast as possible" until we find the first idle time where no packet is pending.

Algorithm 3 shows the pseudocode. The algorithm operates on a deep copy of the current set of streams, maintained in a bucket queue in order of increasing release time.[5] It fictitiously allocates slots to packets in the order in which they are released. T_b keeps track of the number of (full) rounds, and slot counter c keeps track of the number of allocated slots in the current round. The algorithm executes as long as there is a stream s whose initial packet is still to be sent (*i.e.*, $s.releaseTime = 0$), or there is a packet that was already pending before the new round started (*i.e.*, $s.releaseTime < T_b$ **and** $c = 0$), or there is any pending packet while the current round has at least one allocated slot (*i.e.*, $s.releaseTime \leq T_b$ **and** $c > 0$). Otherwise, the algorithm has encountered the first idle time and thus the end of the synchronous busy period.

D. PERFORMING ADMISSION CONTROL

As noted in Sec. 5.3, the challenge in implementing Theorem 4 is to efficiently compute the future demand $h_0(t)$. Using the closed-form expression from the real-time scheduling literature [Buttazzo 2011]

$$h_0(t) = \sum_{i=1}^{n} \left\lfloor \frac{t + P_i - D_i}{P_i} \right\rfloor \tag{17}$$

is not a viable option as it involves many costly divisions.

Thus, as already described in Sec. 5.3, we perform admission control by simulating the execution with **CS**. Algorithm 4 shows the pseudocode. The algorithm takes as input a deep copy of the current set of streams, including the new stream to be admitted, and the new synchronous busy period T_b. All streams start at time $t = 0$ to trigger an

[5]This results in high efficiency, because in each iteration the algorithm needs the earliest release time of all streams in order to check whether it has encountered the first idle time where no packet is pending.

Algorithm 4 Admission control

```
Input  A bucket queue that is a deep copy of the stream set $\Lambda$ including the new stream, with
  $s.absDeadline$ initialized to $D_i$ for all streams $s_i$ (smaller $absDeadline$ means higher priority),
  the utilization as well as the deadline-based utilization of that joint stream set, and the new
  synchronous busy period $T_b$ of that joint stream set.
Output  Whether the new stream is to be admitted or is to be rejected.
  if utilization exceeds 1 then
      return "reject"
  end if
  if deadline-based utilization does not exceed 1 then
      return "admit"
  end if
  initialize $demand$, $availableBandwidth$, and $t$ to 0
  while $demand \leq availableBandwidth$ do
      set $s$ to highest-priority stream using $s = \text{FindMin}()$
      if $s.absDeadline > t$ then
          if $s.absDeadline > T_b$ then
              return "admit"
          end if
          $availableBandwidth = t \times B$
          $t = s.absDeadline$
      end if
      $demand = demand + 1$
      $s.absDeadline = s.absDeadline + s.period$
      update priority of $s$ using $\text{DecreaseKey}(s,\ s.period)$
  end while
  return "reject"
```

interval of maximum demand, so the absolute deadline of each stream s_i is initialized to the relative deadline D_i. Using a bucket queue to keep streams in EDF order, the algorithm repeatedly updates the absolute deadline of the highest-priority stream as if it were executing **CS**. In doing so, the algorithm keeps track of the number of deadlines seen until time t (*i.e.*, the $demand$). If this quantity exceeds the $availableBandwidth$ in the interval $[0, t]$ for any t in $[0, T_b]$, the new stream cannot be admitted.

Algorithm 4 contains two optimizations that improve the average-case performance. First, it checks whether the end of the interval $[0, T_b]$ has been reached and updates the $availableBandwidth$ only when t has advanced. This avoids unnecessary processing when multiple deadlines coincide. Second, it performs two simple checks before the while-loop. The new stream can be rejected without further processing if the *utilization*, defined as $\frac{1}{B}\sum_{i=1}^{n}\frac{1}{P_i}$, exceeds one [Liu and Layland 1973]. Further, since $D_i \leq P_i$ for any stream s_i, the new stream can be admitted if the *deadline-based utilization*, defined as $\frac{1}{B}\sum_{i=1}^{n}\frac{1}{D_i}$, does not exceed one [Stankovic et al. 1998]. We incrementally update both types of utilizations as streams are added and removed at runtime.

E. APPROXIMATING THE WORST-CASE EXECUTION TIME OF THE LS SCHEDULER

In the worst case, an execution of the **LS** scheduler needs to proceed through all four steps in Fig. 10. A careful analysis of the corresponding algorithms (Algorithms 1, 2, 3, and 4) reveals that the execution time of the **LS** scheduler grows with: (*i*) the number of streams n, (*ii*) the largest possible period of any stream $\overline{P}$, (*iii*) the bandwidth demand of the streams, denoted u, and (*iv*) the synchronous busy period T_b of the streams.

Both n and $\overline{P}$ are application-dependent, yet the memory available on a given platform determines their maximum value. For example, in our Blink prototype, the required memory scales linearly with n and $\overline{P}$; for $\overline{P} = 255$ seconds it supports up to

$n = 200$ streams on the TelosB, which comes with 10 kB of RAM. Let us denote with N the maximum number of streams supported for a given upper bound on the period of any stream $\overline{P}$. On the other hand, quantities u and T_b vary depending on the parameters of the streams. Based on this insight, we aim to compute, for a given bandwidth demand u, N stream profiles with periods no larger than $\overline{P}$ such that the synchronous busy period T_b is maximum. We describe how we determine such worst-case stream profiles, and then discuss the concrete settings we use in the experiment of Sec. 6.5.

Determining worst-case stream profiles. We use two integer linear programs (ILPs). In both ILPs, the decision variables are the periods of the streams, which are integers in the interval $[1, \overline{P}]$. We encode the periods through variables $x_1, x_2, \dots, x_{\overline{P}}$, where x_i denotes the number of streams with period i. All start times of streams are assumed to be zero, and deadlines are equal to periods.

First, we tackle the problem of minimizing the bandwidth demand u of N streams, given their synchronous busy period T_b, by solving the following ILP:

$$
\begin{aligned}
\underset{\{x_1, x_2, \dots, x_{\overline{P}}\}}{\text{minimize}} \quad & \sum_{i=1}^{\overline{P}} x_i/i \\
\text{subject to} \quad & \sum_{i=1}^{\overline{P}} x_i = N \\
& \sum_{i=1}^{\overline{P}} x_i \lceil 1/i \rceil > B \\
& \sum_{i=1}^{\overline{P}} x_i \lceil 2/i \rceil > 2B \\
& \qquad \vdots \\
& \sum_{i=1}^{\overline{P}} x_i \lceil (T_b - 1)/i \rceil > (T_b - 1)B \\
& \sum_{i=1}^{\overline{P}} x_i \lceil T_b/i \rceil \le T_b B
\end{aligned}
$$

The objective function is the bandwidth demand. The inequality constraints ensure that at the end of each interval $[0, t]$, $t \in \{1, 2, \dots, T_b - 1\}$, there are one or more pending packet, while there is no pending packet at the end of interval $[0, T_b]$. Note that although the non-linear ceiling function arises in the above ILP, it does not operate on the variables x_i and T_b is a known input. Thus, the left-hand side of each inequality is linear in the variables, so the program is efficiently solved by an ILP solver.

Solving this ILP for different values of T_b, we obtain a function $f(u)$ that gives the maximum T_b for a given bandwidth demand u, as shown in Fig. 17. We now want to invert this function, that is, compute a stream set with a bandwidth demand as close as possible to a given target bandwidth demand u, and with the maximum possible T_b. To this end, we solve the following modified ILP:

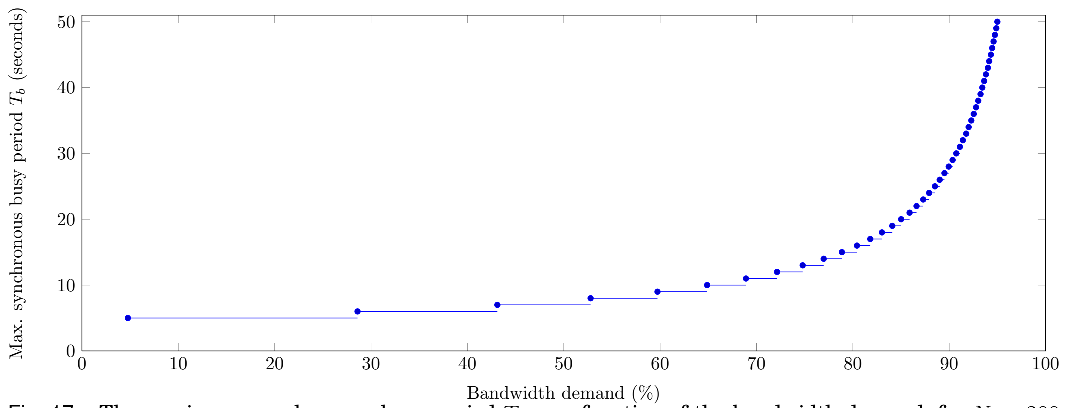


Fig. 17. The maximum synchronous busy period T_b as a function of the bandwidth demand, for $N = 200$ streams, a maximum stream period of $\overline{P} = 255$ seconds, and $B = 51$ available slots per round.

$$\begin{aligned}
\underset{\{x_1,x_2,\ldots,x_{\overline{P}}\}}{\text{minimize}} \quad & \sum_{i=1}^{\overline{P}} x_i/i \\
\text{subject to} \quad & \sum_{i=1}^{\overline{P}} x_i = N \\
& \sum_{i=1}^{\overline{P}} x_i/i \geq u \\
& \sum_{i=1}^{\overline{P}} x_i \lceil 1/i \rceil > B \\
& \sum_{i=1}^{\overline{P}} x_i \lceil 2/i \rceil > 2B \\
& \qquad \vdots \\
& \sum_{i=1}^{\overline{P}} x_i \lceil (f(u)-1)/i \rceil > (f(u)-1)B \\
& \sum_{i=1}^{\overline{P}} x_i \lceil f(u)/i \rceil \leq f(u)B
\end{aligned}$$

Again, the non-linear functions do not operate on the variables, and this time u and $f(u)$ are known inputs.

Settings and method. We proceed in two steps to approach the worst-case execution time of the **LS** scheduler in Sec. 6.5.

In a first step, we solve the ILPs above to determine the worst-case stream profiles. To this end, we consider the maximum number of streams $N = 200$ supported by our Blink prototype on the TelosB for a maximum stream period of $\overline{P} = 255$ seconds. We determine different sets of $N = 200$ streams for bandwidth demands between $5\,\%$ and

Table II. Worst-case stream profiles used in the execution time experiment of Sec. 6.5. For each stream s_i, the start time S_i is set to 0 and the deadline D_i is equal to the period P_i, which is shown in the table below.

Bandwidth demand (%)	**Synchronous busy period (sec)**	**Periods of the worst-case stream profiles (number of streams with a given period in seconds, written as "#streams $\times$ period")**
5	5	1×2, 4×3, 195×255
10	5	5×3, 11×4, 184×255
15	5	4×1, 8×3, 1×4, 187×255
20	5	4×1, 15×3, 2×4, 179×255
25	5	3×1, 1×2, 25×3, 1×4, 170×255
30	6	3×1, 1×2, 6×3, 36×4, 1×5, 153×255
35	6	3×1, 39×3, 5×4, 153×255
40	6	7×1, 36×3, 4×5, 153×255
45	7	19×1, 1×2, 4×3, 5×4, 1×5, 170×255
50	7	15×1, 24×3, 6×4, 2×5, 153×255
55	8	11×1, 44×3, 1×4, 8×5, 136×255
60	9	27×1, 6×6, 14×7, 153×255
65	10	31×1, 3×2, 166×255
70	11	31×1, 1×6, 14×7, 18×9, 136×255
75	13	35×1, 1×5, 1×8, 1×9, 10×10, 14×11, 138×255
80	15	36×1, 1×3, 44×11, 119×255
85	19	39×1, 1×3, 1×5, 1×7, 1×12, 40×14, 5×17, 112×255
90	28	42×1, 5×2, 4×13, 4×24, 9×25, 136×255
95	50	46×1, 3×3, 2×8, 3×40, 5×41, 2×42, 5×43, 5×44, 2×45, 5×46, 5×47, 117×255

95 %, with $B = 51$ available slots per round. Table II lists the different sets of worst-case stream profiles we compute.

In a second step, we emulate for each set of worst-case streams a 2.5 h execution of our Blink prototype using MSPsim. MSPsim is a time-accurate instruction-level emulator that runs the same code that we also use on the real nodes [Eriksson et al. 2009]. During each 2.5 h execution, requests for each of the 200 worst-case streams are submitted one by one in consecutive rounds, and we measure in each round the individual execution times of the four different steps in the **LS** scheduler (see Fig. 10). We keep measuring for some time even after all streams are admitted. Then, we take *for each step individually* the maximum execution time we measured during the 2.5 h execution. Fig. 15 plots for a given bandwidth demand the *sum* of these individual maximum execution times, combining the times of synchronous busy period computation and admission control to aid visibility. Thus, the total execution times we show in Sec. 6.5 are *higher* than the maximum total execution time we actually measured in one round.