Energy-efficient Real-time Communication in Multi-hop Low-power Wireless Networks

TIK Report No. 356 – September 25, 2014

Marco Zimmerling, Pratyush Kumar, Federico Ferrari, Luca Mottola*, Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich
**Politecnico di Milano and SICS Swedish ICT*
{zimmerling,kumarpr,ferrari,thiele}@tik.ee.ethz.ch *luca.mottola@polimi.it

Abstract

Low-power wireless holds the promise of improving reliability and reducing costs in control applications. The key challenge in achieving these goals is to deliver packets within *real-time deadlines* across devices with *limited energy*. Existing approaches either can not provide end-to-end guarantees due to a fully localized operation, or hardly scale as they are sensitive to dynamic changes in the network state. Our key insight is that a fully global approach can overcome these limitations by being agnostic to the current network state. To substantiate this claim, we build Blink, a real-time low-power wireless protocol that provides hard guarantees on end-to-end packet deadlines, scales to large multi-hop networks, and seamlessly handles dynamic changes in network state and real-time requirements. We achieve this by leveraging an existing best-effort protocol that uses only flooding for communication, and by designing novel scheduling algorithms based on the earliest deadline first (EDF) policy. Using a dedicated priority queue data structure, we demonstrate a viable implementation of our algorithms on resource-constrained devices. Results from a 94-node testbed and an instruction-level emulator show that Blink: (*i*) meets almost 100% of packet deadlines, missing only a few due to packet losses over the wireless channel; (*ii*) keeps the network-wide energy consumption to a minimum; and (*iii*) schedules 200 real-time packet streams in less than 80 milliseconds on a 16-bit 8 MHz microcontroller.

1 Introduction

The many advantages of low-power wireless in automation and control are, by now, widely acknowledged. Key industry players argue: "... the possibilities are endless: wireless technology will unlock value in one's process chain far beyond merely avoiding the wiring costs" [3]. Such benefits include improved control safety and process reliability, lower installation and maintenance costs, and higher flexibility when selecting sensing and actuation points [9, 44]. Example applications are level control of dangerous liquids to protect against environmental threats [3], rapid prototyping of automation solutions when retrofitting buildings [8], and minimally invasive monitoring of safety-critical assets [2]. Because of lower costs and ease of installation, battery-powered embedded devices with low-power wireless network interfaces are often preferred in concrete deployments [3, 44].

A characteristic common to these applications is that packets must be delivered within *hard end-to-end deadlines* [14], for example, to ensure the stability of the controlled processes [9]. The hardness of deadlines entails that packets not meeting their deadlines have *no* value and count as lost. Support for this type of real-time traffic is mainstream in wired fieldbuses [6, 7], but hard to attain in a low-power wireless network. This is due to, for example, the dynamics of low-power wireless links [10], the need for multi-hop communication to cover large areas, and the resource scarcity of the employed devices.

1.1 Prior Work

Existing efforts to tackle these challenges can be broadly classified depending on whether they use local or global knowledge to compute packet schedules.

Local knowledge. For example, in SPEED [25], each device continuously monitors the nodes within direct radio range, for example, to detect transient congestion. Using only node-local information, each device computes and follows its own transmission schedule. On a conceptual level, the localized approach of SPEED is also adopted by various MAC-layer solutions [24, 27, 31, 46] and in systems supporting specific traffic patterns [26].

Albeit these solutions scale well because of their local-

ized nature, they *cannot* support hard real-time applications. As described in Sec. 2, the hard real-time requirements are indeed specified from an end-to-end perspective. However, a device that reasons based on local information and that can only influence its surroundings is oblivious of the global picture and unable to affect it.

Global knowledge. By contrast, the WirelessHART [1] standard and solutions alike [4, 5, 32, 40] compute schedules based on global information about the network state. This essentially takes the form of a connectivity graph in which the weight of edge $A \rightarrow B$ represents the link quality (*e.g.*, the packet reception rate) seen by node B when receiving packets from node A. Using this global information, a central entity computes and distributes transmission schedules tailored to each node, thereby forming end-to-end routing paths from the sources to the destination(s). Then, each node follows its own schedule locally.

Two fundamental problems impact these approaches:

1. Network state information is extremely time-varying because of fluctuating low-power wireless links [42], environmental influences [10], and device outages or node mobility [47]. Any such change in network state must be communicated to the central entity for updating the connectivity graph and possibly re-computing and re-distributing packet schedules. As this happens, new changes might happen, requiring to re-iterate the same processing over and over again. Meanwhile, packets are lost due to inconsistent routing paths or miss their deadlines because of obsolete schedules.
2. The need to compute per-node transmission schedules also causes severe scalability issues in large networks. Existing works map the problem to that of *multiprocessor* task scheduling [38]. Because of this, in WirelessHART networks, for example, any optimal solution scales at least exponentially with the network diameter [38]. Despite some attempts to address this issue [15, 38, 48], WirelessHART schedulers are hardly practical in networks of more than 3 hops [15].

Therefore, these solutions *cannot* support hard real-time applications either, as also acknowledged by industry players who contributed to the WirelessHART standard: "... none of the technologies provide any hard guarantees on deadlines, which is needed if you should dare to use the technology in critical applications" [35].

Thus, the problem of providing hard real-time guarantees over low-power wireless remains unsolved, hampering a widespread application of the technology at hand.

1.2 Our Contribution

We present the design and implementation of Blink, the first real-time low-power wireless protocol that provides hard guarantees on end-to-end packet deadlines in large multi-hop networks, while simultaneously achieving low energy consumption. Blink seamlessly handles dynamic changes in the network state and in the application's real-time requirements, and readily supports scenarios with multiple controllers and actuators.

The approach we adopt in Blink differs fundamentally from prior art: Blink uses *global* knowledge to compute a single *global* schedule that applies to the entire network. As described in Sec. 3, the key idea is to detach the protocol operation from the network state, whose rapid variations would cause severe scalability issues [15, 38, 48].

As illustrated in Sec. 4, the design of Blink rests upon three main components:

- *Communication support:* we leverage the Low-Power Wireless Bus (LWB) [21], an existing best-effort protocol that exclusively employs network-wide flooding for communication [22]. This allows us to abstract away the network state when scheduling packets. In addition, LWB's time-triggered operation allows us to treat the entire network as a single device that runs on a single clock. As a result, we can map the scheduling problem to *uniprocessor* task scheduling, making it way easier to solve than prior art [38].
- *Real-time scheduling policies:* we conceive scheduling policies based on the earliest deadline first (EDF) principle [30]. Using these policies, Blink computes online a schedule that provably meets all deadlines of a set of admitted real-time packet streams while minimizing energy consumption, tolerating changes both in the network state and the set of streams.
- *Efficient data structures and algorithms:* we design and implement a highly efficient priority queue to enable EDF-based scheduling on resource-constrained devices. As a result, we demonstrate the first implementation of EDF on this class of devices. This is notable per se: due to its run-time overhead, EDF has seen little adoption even on commodity hardware, despite its realtime-optimality [13, 39].

Sec. 5 reports on the evaluation of our Blink prototype in two real-world testbeds of up to 94 nodes [11, 29] and using a time-accurate instruction-level emulator [20]. Our results show that Blink meets almost 100 % of the deadlines; the few missed deadlines are exclusively due to packet losses, which can not be fully avoided in a wireless setting. Moreover, Blink can schedule 200 real-time packet streams in less than 80 ms on an MSP430 microcontroller running at 8 MHz, in a scenario with the highest processing demand we could possibly trigger.

We discuss trade-offs and limitations of our Blink prototype in Sec. 6 and conclude in Sec. 7.

2 Background

The scheduling problem is a function of the application requirements, the characteristics of the deployment, and the devices employed. We discuss these aspects next.

Applications. The applications expose hard real-time requirements in either or both the monitoring and control phases. Example domains are industrial and building automation, process control, or smart grids [9, 45]. In these scenarios, a set of sensing devices periodically streams sensed data under given temporal constraints. Every device may source multiple such streams. The data is then used for monitoring or to feed control loops. In the latter, the loop may execute right on the actuators that affect the environment, or on a few dedicated controller devices that periodically distribute commands to the actuators, again subject to tight temporal constraints.

Let Λ denote the set of all n streams in the network. Each stream $s_i \in \Lambda$ releases one packet at a regular periodic interval P_i, called the *period* of stream s_i. The *start time* S_i is the time when stream s_i releases the first packet. Every packet released by stream s_i must be delivered to the destination within the same *relative deadline* D_i. The next packet is only released after the *absolute deadline* of the previous packet, so deadlines are less than or equal to periods (*i.e.*, $D_i \leq P_i$). 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 same profile, we also write $k\langle \cdot, \cdot, \cdot \rangle$.[1]

The actual real-time requirements are application-specific. The physical dimension recorded through sensors often dictates a stream's period P_i. For example, temperature control in liquid volumes [34] demands periods in the order of minutes, whereas compressor speed control requires periods down to microseconds. Low-power wireless is applicable with the greatest advantages in the former type of applications [45]. The monitoring or control process governs a stream's deadline D_i and starting time S_i. For example, closed-loop control typically requires shorter deadlines than open-loop control [33].

Deployments and platforms. Resource-constrained embedded devices amplify the benefits in ease of installation and maintenance [9, 34]. Typical devices feature 8- or 16-bit MCUs with a few KB of data memory, and rely on non-renewable energy sources [9, 45]. This motivates the use of low-power wireless, which reduces the energy costs but limits the bandwidth and makes the system susceptible to interference and environmental factors [10].

Deployments consist of tens to some hundred devices. The devices are normally installed at fixed locations, determined by application requirements. Because of energy constraints that limit the individual radio ranges, designers rely on multi-hop networking to ensure overall connectivity. The nodes may be added or moved to optimize the measurement locations or to prototype improvements over existing installations [47]. The network may therefore exhibit some degree of mobility, which adds to the temporal dynamics of low-power wireless links [10, 42].

Problem. As a result, a solution to support real-time low-power wireless must meet the application-defined packet deadlines, while also achieving:

- *Scalability* in terms of the number of streams and the size of the deployment;
- *Adaptiveness* to accommodate changes in the set of active streams and the network topology;
- *Energy-efficiency* to achieve long system lifetimes in the face of limited energy resources;
- *Viability* with respect to limited bandwidth and computational resources.

Subject to these, we can formulate the problem at hand as finding one (or multiple) packet schedule(s) such that, given n streams, $n = |\Lambda|$, for every stream $s_i \in \Lambda$, every packet released by s_i is delivered within D_i time units.

3 Breaking New Ground "in a Blink"

We describe the key insights enabling our sharply different approach and give an overview of Blink's design.

Foundation. To detach Blink's operation from the time-varying network state, we use the Low-Power Wireless Bus (LWB) [21] as underlying communication support. LWB is a non-real-time protocol that turns a multi-hop wireless network into a shared bus, where all nodes are potential receivers of all packets.

To this end, LWB maps *all* communications onto efficient network floods [22], which blindly propagate every message to all nodes. The nodes are time-synchronized and communicate in a *time-triggered* fashion according to a global communication schedule. The schedule determines how subsequent *communication rounds* unfold over time, as shown in Fig. 1 (A), whereby nodes get the chance to send and receive packets. Nodes keep their radios off between rounds to save energy. Because packet transmissions occur through network-wide flooding, the schedule can arbitrate access to the (wireless) bus *independent* of network state. Although a flooding-based approach may seem wasteful, LWB outperforms prior solutions in packet reliability and energy efficiency [21].

Besides freeing Blink's real-time scheduler from considering the current network state, LWB's time-triggered

[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.

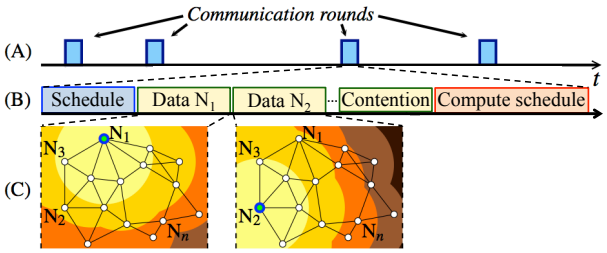


Figure 1: Time-triggered operation in LWB.

operation allow us to treat the network as a single device that runs on a single clock, independent of network state. We can thus map the problem to *uniprocessor* task scheduling, making it simpler to solve than the multiprocessor problem encountered in prior solutions [38].

Blink. Two key issues prevent using LWB "as is" for real-time traffic. First, the existing LWB scheduler [21] is oblivious of packet deadlines, and only meant to reduce energy consumption. We must thus conceive a suitable policy to schedule packets such that all deadlines are met while minimizing the network-wide energy consumption. Second, the real-time scheduler must execute within strict time limits on severely resource-constrained devices. We must therefore provision an efficient implementation fully cognizant of the platform restrictions.

To address these issues, the design of Blink is driven by two goals that we must achieve simultaneously:

1. *Realtime-optimal [39] scheduling*, which entails (*a*) admitting a new stream if an only if there exists a scheduling policy able to deliver all packets by their deadlines, and (*b*) deliver all packets of admitted streams by their deadlines.
2. *Minimum network-wide energy consumption*, which requires the ability to minimize the number of rounds over any given time interval, because every round incurs a fixed energy overhead independent of the number of packets that are actually sent in the round [21].

Each of these goals arguably presents a difficult problem on its own, and a significant challenge if considered together. Three functionalities are thus needed:

1. Because the set of streams may change over time, we must check at runtime whether accepting a new stream prevents meeting the deadlines of the existing streams. We refer to this as *admission control*.
2. Given the set of admitted streams, we must decide when the next round starts so as to meet all deadlines while minimizing energy consumption. We refer to this problem as *start of round computation*.
3. Once the start time of a round is computed, we must decide which and how many packets are actually sent in the round. We refer to this as *slot allocation*.

To realize these functionalities, Blink builds on three core components: (*i*) the underlying communication support, (*ii*) scheduling policies that are provably realtime-optimal and minimize energy consumption, and (*iii*) efficient data structures and algorithms to let the scheduling policies run "in a blink" on resource-constrained devices. We discuss each component in the next section.

4 Design and Implementation

In the following, we briefly describe in Sec. 4.1 the basic LWB operation as a prerequisite to the following material. Thereafter, Secs. 4.2 and 4.3 discuss the slot allocation and start of round computation, respectively, assuming the set of streams is schedulable. Sec. 4.4 describes how to ensure this condition through admission control.

4.1 Communication Support

As shown in Fig. 1 (A), the operation of LWB is confined within *communication rounds*, executed simultaneously by all nodes. Each round consists of at most B non-overlapping *communication slots*, as shown in Fig. 1 (B). All nodes engage in the communication in every slot: one node places a message on the bus (*i.e.*, initiates a flood) and all other nodes read the message from the bus (*i.e.*, receive and relay the message), as shown in Fig. 1 (C). The probability that all nodes receive the message ranges above 99.9 % in real-world experiments [22]. At the end of a round, only the intended receivers (properly encoded in the packet) deliver the message to the application.

Every round starts with a slot used by a specific node, called *host*, to distribute the communication schedule, as shown in Fig. 1 (B). The schedule specifies when the next round will start and which nodes can send messages in the immediately following slots. If a node receives the schedule, it time-synchronizes with the host and participates in the round; otherwise, it does not take any action until the next round. To inform the host about their traffic demands, nodes may compete in a final *contention slot*. Because of a phenomenon called capture effect [28], with high probability one node succeeds anyways and reaches the host. Based on all traffic demands it received thus far, the host computes the schedule for the next round.

Besides the conceptual leap it enables, as discussed in Sec. 3, LWB additionally contributes useful functionality towards a practical system implementation. It supports different traffic patterns, including one-to-many, many-to-one, and many-to-many, with no changes to the protocol operation, easing the deployment in scenarios with multiple actuators or separate controllers. Moreover, unlike existing solutions (*e.g.*, [1, 5]), LWB features mecha-

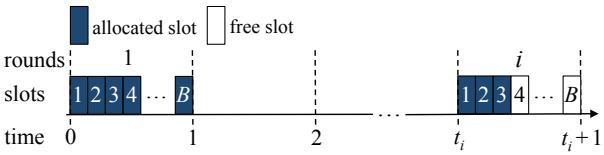


Figure 2: Discrete time model of LWB. *Each round is of unit length and consists of B slots, each of which is either allocated to a packet or free. The i-th round starts at time t_i and ends at $t_i + 1$, where t_i is a non-negative integer*

nisms to resume the system operation after a host failure, thus overcoming single-point of failure problems [21].

In the following, we consider a discrete time model, as shown in Fig. 2. In this model, each round is *atomic* and of unit length. This is so because during a LWB round the MCU is continuously busy serving radio interrupts [22], so any other event, including packet delivery, can only be served before or after a round. As a result, the timeline of possible start times of a round is also discrete.[2]

4.2 Slot Allocation

Determining a schedule for a given round requires to address two questions: (*i*) With B slots available per round, how many packets should we actually allocate? (*ii*) How should we prioritize packets of different streams?

Algorithms. To answer question (*i*), 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 Blink. Indeed, in the next round, the set of packets to schedule will be the same or larger, which can only worsen the overall schedulability. Furthermore, the energy *overhead* in Blink depends on the number of rounds, not on the number of allocated slots therein [21]. Therefore, we allocate as many pending packets as possible to the available slots in any given round.

As described in Sec. 3, the approach we adopt allows us to treat the network as a single device. We can thus resort to uniprocessor scheduling policies to answer question (*ii*). Among these, the *earliest deadline first (EDF)* policy is provably realtime-optimal [16]. This entails that if a set of streams can be scheduled such that all packets meet their deadlines, then EDF also meets all deadlines. This holds also for stream sets demanding the full bandwidth, whereas other policies, such as rate-monotonic (RM), may fail to meet all deadlines at much lower bandwidth demands [30]. Finally, as the packet priorities are computed while the system executes, EDF can cope with run-time changes in the set of streams [39], which is important in dynamic low-power wireless networks.

[2]The general validity of both algorithms and system designs is discussed in App. A.

Table 1: Required operations on the set of streams Λ.

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

Applying EDF in Blink entails allocating the next slot in a round to the packet whose deadline is closest to the start time of the round. This seemingly simple logic bears a significant run-time overhead [13]. To implement EDF efficiently, one needs to maintain the streams in increasing order of absolute deadline while the latter is updated from one packet to the next. This is one reasons why EDF is rarely used in real systems, such as operating system kernels [13]. We describe next how we tackle this issue.

Design and implementation in Blink. Key to enabling EDF in Blink is the provision of a data structure to efficiently maintain the current set of streams in order of increasing absolute deadline. A suitable data structure must support all operations required to manipulate the streams during EDF scheduling, as summarized in Table 1.

First, EDF requires a FINDMIN() operation to retrieve the stream with the earliest absolute deadline, which is to be served first. A *priority queue* is thus the most natural data structure to employ, where streams with smaller absolute deadline are given higher priority, and thus served next. Second, after serving a stream, we need to update its absolute deadline to the following packet. We thus require a DECREASEKEY(s, δ) operation that propagates an increment of δ in the absolute deadline of stream s in the priority queue. As a result, the priority of stream s is *decreased* relative to the other streams in the queue.

These two operations are supported by almost all priority queue implementations [12]. Nevertheless, due to the answer to question (*i*) above—allocate as many pending packets as possible—we also need operations to perform an efficient *EDF traversal* of the stream set while streams with pending packets are updated. It should thus be possible to position a *traverser* t at the highest-priority stream using FIRST(t). Then, we visit streams in EDF-order using repeated NEXT(t) calls. During the traversal, we update the priority of any stream t that has a pending packet with DECREASEKEY(t, δ).

One way to support all required operations would be to use a binary search tree, such as a right-threaded red-

Figure 3: Bucket queue implemented as a circular array of $2\overline{P}$ doubly-linked lists. *Here, the upper bound on the period of any stream is $\overline{P} = 8$. The queue contains seven streams ordered by increasing key: $s_5, s_2, s_7, s_1, s_4, s_3, s_6$.* FINDMIN() *sets index L to 14, because this is the bucket currently containing the stream with the smallest key.*

black tree [36]. The following properties of our specific problem, however, allow us to use a much simpler and more efficient priority queue data structure:

1. A stream's absolute deadline, henceforth referred to as the *key* of a stream, is a non-negative integer.
2. The key of a stream increases monotonically as it is being updated from one packet to the next.
3. The range of keys in the priority queue at any one time is bounded, as stated in the follow theorem (the proof is in App. B):

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 priority queue at any one time.*

Given these properties, we can consider using a monotone integer priority queue. In particular, we use a simple *one-level bucket queue* [17] implemented as a circular array $\mathbf{B}$ of $2\overline{P}$ doubly-linked lists, as illustrated in Fig. 3. $\overline{P}$ is an upper bound on the period of any stream. Stream s_i with key d_i is stored in $\mathbf{B}[d_i \bmod 2\overline{P}]$. Because a stream's deadline D_i is no longer than its period P_i, the 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. As a result, all streams in a bucket have the *same* key. As an example, the keys in the bucket queue of Fig. 3 are in the range $[30, 45]$, and the two streams in $\mathbf{B}[2]$ have key 34.

INSERT(s), DELETE(s), and DECREASEKEY(s, δ) all take constant time, because the buckets are implemented as doubly-linked lists. INSERT(s) inserts a stream s with key d into $\mathbf{B}[d \bmod 2\overline{P}]$.[3] DELETE($s$) removes s from the list containing it. DECREASEKEY(s, δ) first performs a DELETE(s) and then inserts s into $\mathbf{B}[(d+\delta) \bmod 2\overline{P}]$.

To support FINDMIN(), we maintain an index L, initialized to 0. We update L on every invocation of FINDMIN(). If $\mathbf{B}[L]$ is empty, we increment L until we find a non-empty bucket; otherwise, FINDMIN() returns the first stream on the list in $\mathbf{B}[L]$. FIRST(t) works similarly, using another index I. NEXT(t) moves to the next stream on the list in $\mathbf{B}[I]$. When the end of the list is reached, we increment I until we find the next non-empty bucket. The three operations run in $O(2\overline{P})$ worst-case time.

Our bucket queue implementation underpins all Algorithms 1, 2, 3, 4 required for energy-efficient EDF-based real-time scheduling in Blink. Its efficiency stems mainly from two observations. First, DECREASEKEY(s, δ) is a frequently used operation and at the same time highly efficient. Second, the cost of searching for a non-empty bucket amortizes. For example, NEXT(t) needs to increment index I in the worst case $2\overline{P}-1$ times, but the following n calls to NEXT(t) require no searching at all because all n streams are necessarily in $\mathbf{B}[I]$. With these implementation choices, Blink can indeed schedule hundreds of streams using EDF on resource-constrained devices even when the system is in a continuous state of change, as we show through experiments in Sec. 5.

4.3 Start of Round Computation

We now turn to the problem of computing a round's starting time. We use an illustrative example with $B = 5$ slots per round and the following streams: $3\langle 0, 5, 4\rangle$, $4\langle 2, 7, 5\rangle$, and $5\langle 1, 15, 12\rangle$. Fig. 4 shows the release times and deadlines of packets in the first 14 time units. Using our EDF-based slot allocation policy, when should a round start to meet all deadlines while minimizing energy consumption (*i.e.*, minimizing the number of rounds)?

Algorithms. One possibility, called *contiguous scheduling* (**CS**), is to start the next round immediately after the previous one. **CS** offers the highest possible bandwidth and therefore necessarily meets all deadlines, provided the streams are schedulable. However, **CS** wastes energy, because it triggers more rounds than necessary. Looking at Fig. 4, we see that 8 out of the first 14 rounds are empty when using **CS**, causing unnecessary energy overhead.

Greedy scheduling (**GS**) improves on **CS** by triggering a round only when there are pending packets. **GS** is realtime-optimal just like **CS**, because it schedules packets as soon as possible. In addition, it can reduce the energy consumption compared to **CS** in certain situations. Fig. 4 shows that using **GS** results in only 6 rounds in the first 14 time units. However, 8 slots are still left unused,

[3] When $2\overline{P}$ is a power of two, the modulo operation is equivalent to a bit-wise AND of d with $2\overline{P}-1$ as the mask, which is highly efficient.

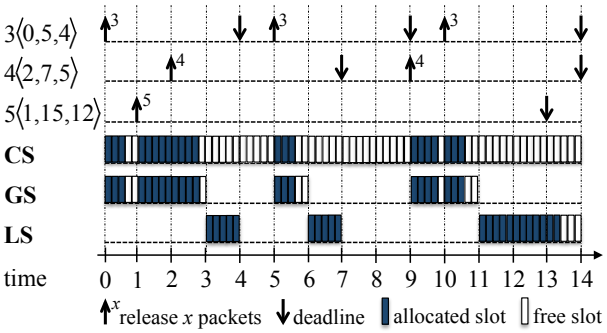


Figure 4: **CS** and **GS** waste energy by scheduling more rounds than necessary. **LS** meets all deadlines while minimizing energy (*i.e.*, minimizing the number of rounds).

raising the question whether we can do even better.

The crucial observation is that **GS** triggers a round no matter how "urgent" it really is. If there was some time until the earliest deadline among all pending packets, we could defer the next round further into the future. Meanwhile, we can await more packet arrivals and thus allocate more slots in the next round. However, this strategy may do more harm than good. Without knowing how much bandwidth demand arises in the future, we could end up deferring the next round to a time where the number of packets to be sent exceeds the available bandwidth. This would inevitably cause deadline misses.

Lazy scheduling (**LS**) addresses this issue. At the core of **LS** is the notion of *future demand* $h_i(t)$, quantifying 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 have not been served until the end of round i. This materializes in the following expression

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

where P_j denotes the period and d_j the absolute deadline of stream s_j. **LS** uses $h_i(t)$ to compute the latest possible start time of the next round while meeting all deadlines. This leads us to the following result (proven in App. D).

Theorem 2. ***LS*** *is real-time optimal and minimizes the network-wide energy consumption.*

Let us look at how **LS** works. As a concrete example, assume we want to compute the start time of the third round, t_3, in Fig. 4 with **LS**. We proceed in three steps:

1. *Compute* $h_2(t)$. As illustrated in Fig. 5, $h_2(13) = 5$ as $t = 13$ is the absolute deadline of the $5\langle 1,15,12\rangle$ streams whose packets are still 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, this takes $\lceil h_2(13)/B \rceil = 1$ round. Thus, we get a first latest possible start time $t_3^1 = 13 - 1 = 12$. We indicate this in Fig. 5 by casting a shadow back on the time axis. Further, we see that $h_2(14) = 12$ 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 $t_3^2 = 14 - 3 = 11$, etc.
3. *Take the minimum of all computed start times as* t_3. Based on the reasoning in step 2., pushing the start of the third round beyond the beginning of the shady area at $\min\{t_3^i\} = 11$ in Fig. 5 would cause deadline misses. At the same time, an earlier start time could, in the long run, lead to more rounds than needed. As a result, the third round should start at $t_3 = 11$.

Figure 5: Illustration of how **LS** computes the latest possible start time of the third round in Fig. 4.

The three steps above illustrate the main intuition behind **LS**. Nevertheless, two important questions remain: (*i*) For which times t do we need to perform steps 1. and 2.? (*ii*) How far do we need to look into the future?

To answer (*i*), we observe that $h_i(t)$ is a step function: its value increases only at times of deadlines (see Fig. 5). Thus, we can skip all intervals where $h_i(t)$ is constant.

The answer to (*ii*) is based on two observations. First, we can defer the start of the next round by at most T_{max} after the start of the previous round. This is to ensure that nodes can update their synchronization state sufficiently often in the face of clock drift [21]. To find out whether we can indeed defer the next round by T_{max}, we also need to evaluate $h_i(t)$ for at least T_{max} time units after the end of the previous round at $t_i + 1$, as illustrated in Fig. 6. In addition, we have to consider any future demand arising *after* $t = t_i + 1 + T_{max}$ that could prevent us from deferring the next round by T_{max}. Thus, we have to look for another

Figure 6: Illustration of how far **LS** needs to look into the future when computing the start time of the next round.

T_b time units into the future (see Fig. 6). T_b is known as the *synchronous busy period* [41]. Informally, this is the time needed to contiguously serve the maximum demand that a given stream set can possibly create.[4] By looking up to $t = t_i + 1 + T_{max} + T_b$ into the future, we ultimately ensure that all deadlines will be met.

The following theorem formalizes what we discussed thus far (the proof is in App. E).

Theorem 3. *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 deferred, and B the number of slots available in a round. Using **LS**, the start time of each round t_i for all $i = 0, 1, \ldots$ is computed as*

$$t_{i+1} = \min(t_i + T_{max}, T_i), \tag{2}$$

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

$$T_i = \min_{t \in \mathscr{D}_i} \left(t - \left\lceil \frac{h_i(t)}{B} \right\rceil \right). \tag{3}$$

$\mathscr{D}_i$ is the set of deadlines in the interval $[t_i + 1, t_i + T_{max} + T_b + 1]$ of packets that are unsent until the end of round i.

Design and implementation in Blink. The main challenge in implementing Theorem 3 is to efficiently compute the future demand $h_i(t)$. Using concepts from the real-time literature [43], we derived the expression in (1).

We implemented this method on a TelosB [37], which is representative of the low-power wireless devices used in the scenarios of Sec. 2, and observed prohibitive running times due to excessive divisions. This is expected also on other resource-constrained platforms, as they typically lack hardware support for accelerating divisions.

To tackle this problem, we opt for a different approach, and compute $h_i(t)$ without performing divisions by *simulating* the execution of **CS**. Algorithm 1 shows the corresponding pseudocode to compute the start time of the next round t_{i+1}. 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 allocates slots in EDF order, using variable *futureDemand* to keep track of the number of allocated slots. In addition, it maintains a variable *minSlack*, which ultimately determines how far we can defer the start of the next round.

Algorithm 1 Compute Start Time of Next Round

Input A bucket queue based copy of the current set of streams (smaller *absDeadline* implies 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 ∞
set s to highest-priority stream using s = FINDMIN()
t = s.absDeadline
while t ≤ t_i + T_max + T_b + 1 do
    futureDemand = futureDemand + 1
    s.absDeadline = s.absDeadline + s.period
    update priority of s using DECREASEKEY(s, s.period)
    set s to highest-priority stream using s = 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(⌊minSlack/B⌋, T_max)
```

By avoiding divisions and using our bucket queue implementation, our implementation of Algorithm 1 for the TelosB achieves several-fold speed-ups over the analytic method. As detailed in App. F, we use the same techniques to efficiently compute the length of the synchronous busy period T_b, which is crucial to **LS** and admission control (see Sec. 4.4), achieving similar speed-ups over an existing iterative method [43]. These improvements in processing time are instrumental to the viability of Blink on resource-constrained platforms.

4.4 Admission Control

So far we assumed the stream set is schedulable, yet this is not always the case. As Fig. 7 exemplifies, for $B = 5$, the set consisting of $9\langle 8, 4, 3\rangle$ and $7\langle 0, 25, 2\rangle$ streams is not schedulable. The streams require a total of 16 slots in the interval $[100, 103]$; however, there are only 15 slots available in the same 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 hinted by the example above, the system misses a deadline if, over some time interval, the demand exceeds the available bandwidth. As already mentioned, the available bandwidth is maximum for **CS**. Hence, ad-

[4] The maximum demand arises when all streams release their packets at the same time. **CS** essentially serves this demand "as fast as possible." 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.

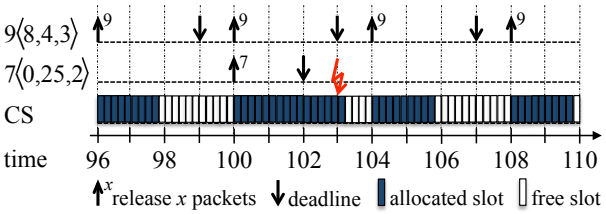


Figure 7: Example of a stream set that is not schedulable.

mission control under all scheduling policies in Sec. 4.3 amounts to checking if **CS** can meet all deadlines.

For efficiency reasons, we seek to perform this check only over an interval in which the demand is highest. If the bandwidth is sufficient even in this extreme situation, we can safely admit the new stream. Precisely identifying this situation is, however, non-trivial, in that the streams' different start times and periods may defer this situation until some arbitrary time. For example, in Fig. 7 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 forcing all streams to release a packet at $t = 0$. Using the concept of synchronous busy period T_b, 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 existing results [41]:

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

$$\forall t \in \mathscr{D},\ h_0(t) \leq t \times B, \tag{4}$$

where $\mathscr{D}$ 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. An efficient implementation of Theorem 4 faces the same challenges as previously discussed in Sec. 4.3. Although a closed-form expression of the demand $h_0(t)$ exists [14], using it represents a performance hog on resource-constrained platforms due to many costly divisions.

We thus perform admission control again by simulating the execution with **CS**. Algorithm 4 in App. G takes as input a deep copy of the current set of streams including the new stream to be admitted, and the synchronous busy period T_b. 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 t (*i.e.*, the demand). If this quantity exceeds the bandwidth available in $[0, t]$ for any t in $[0, T_b]$, the new stream cannot be admitted.

Figure 8: Main steps in Blink's real-time scheduler. *The algorithm to compute the start time of the next round depends on whether **CS**, **GS**, or **LS** is used. When **LS** is used, the synchronous busy period T_b is carried over to subsequent rounds unless the set of streams changes.*

4.5 Blink In Practice

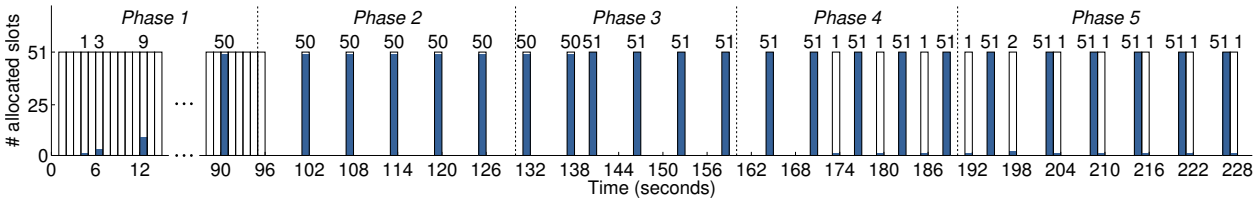
At the end of each round, the algorithms described above unfold as shown in Fig. 8. With a pending request for a new stream s, the scheduler computes the (new) synchronous busy period for the stream set $\Lambda \cup \{s\}$ and checks if s can be admitted or not. In any case, the scheduler computes the start time of the next round and finally allocates slots to pending packets. In the worst case, a single scheduler execution needs to proceed through all four steps in Fig. 8. Experiments in Sec. 5 demonstrate that our implementation can schedule hundreds of streams generating a wide range of realistic traffic loads within the time bounds allowed by our prototype.

We implement the processing in Fig. 8 in a Blink prototype on top of the Contiki [18] operating system. Our prototype targets the TelosB platform, which comes with an 8 MHz MSP430 MCU, an IEEE 802.15.4-compliant 250 kbps low-power wireless radio, 10 kB of RAM, and 48 kB of code memory [37]. We use the default settings in LWB [21], including the 1 second duration of a round that corresponds to the time unit used throughout Sec. 4. As the only exception, we set the number of slots in a round to $B = 51$, leaving 100 ms to compute the schedule at the end of around. We also modify the time synchronization mechanism of LWB to support frequently varying intervals between consecutive rounds in Blink.

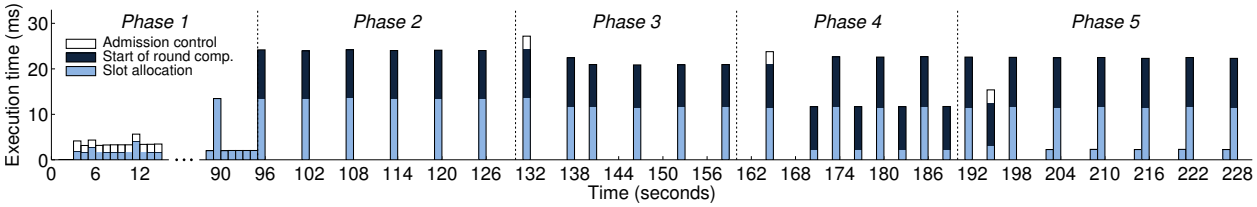
5 Evaluation

This section evaluates Blink along four lines: (*i*) the ability to cope with dynamic changes in the set of streams, (*ii*) the level of real-time service provided with regards to meeting packet deadlines, (*iii*) the energy efficiency of the scheduling policies, and (*iv*) the scalability properties of **LS** scheduling. In the scenarios we test, we find that:

- Blink promptly adapts to dynamic changes in the set of streams without unnecessarily increasing energy.



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



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

Figure 9: A real trace of Blink dynamically scheduling different streams with varying real-time requirements. *After bootstrapping the network, Blink uses **LS** to save energy, while meeting all deadlines of the changing set of streams.*

- Blink meets 99.97 % of the packet deadlines; misses are entirely due to (unavoidable) packet losses.
- **LS** improves energy consumption by up to a factor of 2.5× over **CS** and **GS**, depending on the streams.
- **LS** scheduling takes less than 80 ms for the highest processing demand we could possibly trigger.

Note that dynamic changes in network state, including those due to link quality changes, node mobility, and node failures, are already effectively dealt with by the underlying LWB communication support, as shown in [21].

5.1 Adaptiveness to Stream Set Changes

In our first experiment, we show how Blink dynamically adapts to changes in the set of streams. We use 29 TelosB nodes on the FlockLab testbed [29], which has a diameter of 5 hops. One node acts as the host to run the scheduler, and three randomly chosen nodes are destinations. The remaining 25 nodes serve as sources generating $2\langle 0,6,6\rangle$ streams each. Eventually, we also let two sources request and update a third stream with different parameters.

Execution. Fig. 9a shows the number of slots allocated in each round over time, while Fig. 9b shows a breakdown of the scheduler execution time in each round.

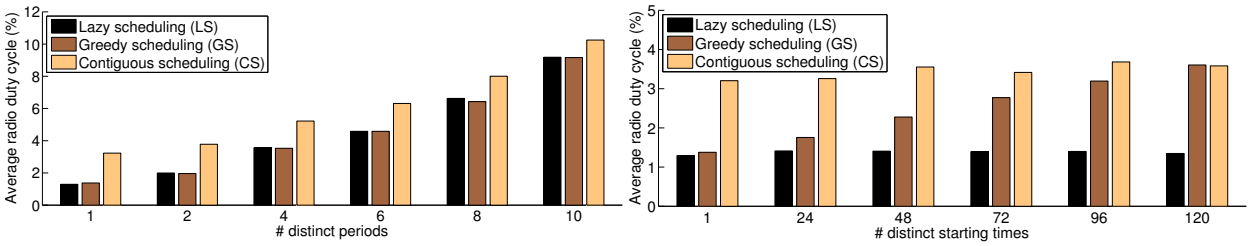
In *Phase 1*, the system is bootstrapping after powering on the devices, so Blink schedules rounds contiguously to allow all nodes to quickly time-synchronize and submit their stream requests. This happens for the first time after 3 seconds, as visible in Fig. 9b from the increase in processing time to perform admission test and slot allocation. During the following rounds, the host gradually receives all initial stream requests and, consequently, admission test and slot allocation take longer.

In *Phase 2*, because no new stream request has recently arrived, Blink adapts its functioning to the normal operation and starts to dynamically compute the start of rounds using **LS**. Fig. 9b shows that it takes about 10 ms to do so. In these conditions, **LS** schedules a round every 6 seconds, postponing rounds until right before the packets' deadlines, which minimizes energy consumption.

At the beginning of *Phase 3*, a request for a new stream $\langle 0,6,3\rangle$ arrives. Admission control executes as visible in Fig. 9b at $t = 131$ seconds. The new stream is admitted and accounted for starting from $t = 140$ seconds. Rounds are scheduled again every 6 seconds and with all $B = 51$ available slots allocated. Unlike most existing systems, Blink has accommodated a new stream without jeopardizing the existing ones, and still maintains the minimum energy consumption. In a WirelessHART network, for example, changes to the stream set tend to be way more disruptive, likely affecting existing streams [48].

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

In *Phase 5*, the node that requested the stream in *Phase 3* extends its deadline from 3 to 6 seconds. Thus, the current 52 streams all have the same deadline and period. Again, because 52 packets do not fit into a single round, Blink schedules a complete round with 51 allocated slots 2 seconds before the packets' deadlines, followed by an-



(a) Varying number of distinct periods.

(b) Varying number of distinct starting times.

Figure 10: Average radio duty cycle of Blink with **LS**, **GS**, and **CS** across 94 nodes in the w-iLab.t testbed. *Depending on the stream set, **LS** achieves up to a 2.5× reduction in energy consumption compared to **GS** and **CS**.*

other round for the remaining packet. This shows that a seemingly minor change in the real-time requirements of one stream can have a significant impact on how rounds unfold over time, which Blink can effectively handle.

5.2 Packet Deadlines and Energy Consumption

We next assess Blink's ability to meet deadlines and the energy costs of **LS** compared to **CS** and **GS**.

Metrics and settings. We use two key performance metrics in real-time low-power wireless [38]. The *deadline success ratio* measures the fraction of packets that meet their deadlines, indicating the level of real-time service provided to applications. We compute this figure based on sequence numbers embedded into packets and timestamps taken at both communication ends. The *radio duty cycle* is defined as the fraction of time a node has the radio on, which is widely used as a proxy for energy consumption [23]. This indicates the energy cost the system incurs in providing a given level of real-time service. We compute this figure using Contiki's power profiler [19].

We run experiments with 94 TelosB devices on the w-iLab.t testbed [11] with a diameter of 6 hops. We let 90 nodes act as sources, one as the host, and 3 as sinks, mimicking a scenario with multiple controllers [34]. Each source triggers two streams, hence we have in total 180 streams generating packets with a 10-byte payload.

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, resembling configurations used when combining primary and secondary control [33]. In this way, we generate varying demands between 2.9% and 19.4% of the available bandwidth. Then, we set the period P_i of all streams to 2 minutes and vary the number of distinct starting times from 1 to 120. This emulates situations where, for example, sources are added to a running system over time with no explicit alignment in the packet release times [34]. Deadlines are equal to periods in all runs. Each run lasts for 50 minutes. To be fair across runs, we give nodes enough time to submit their stream requests and start to measure after 20 minutes.

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 [9]. We thus argue that Blink can effectively operate in several of these scenarios. Second, we verify that such deadline misses are entirely due to packets lost on the wireless channel, that is, the issue is orthogonal with respect to packet scheduling. These results experimentally confirm the reasoning and theoretical results of Sec. 4.

As for radio duty cycle, Fig. 10a shows the average across all 94 nodes for **LS**, **CS**, and **GS**. We see that 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 here: 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 unnecessary rounds plays a minor role at higher bandwidth demands.

Fig. 10b shows the radio duty cycle as the number of distinct starting times increases. The bandwidth demand remains constant across all settings, 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** is forced to schedule more rounds. Instead, **LS** 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**.

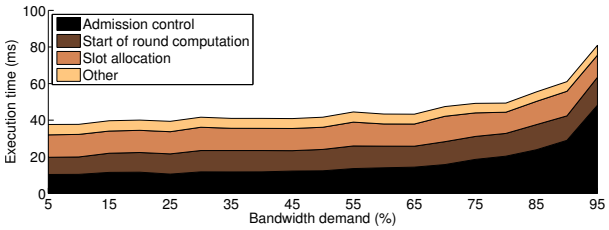


Figure 11: Breakdown of **LS** scheduler execution time for varying bandwidth demands. *Even at a bandwidth demand of 95 %, computing the schedule takes only 80 ms.*

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

5.3 Scheduler Execution Time

In our final experiment, we look at the scalability of our TelosB implementation of the **LS** scheduler in Blink.

Approach. As hinted at in Sec. 4, the processing complexity of the different steps in Fig. 8 grows with the number of streams, the bandwidth demand, and the synchronous busy period T_b. Precisely quantifying how the combination of the three factors determines the running time of the **LS** scheduler is non-trivial. We therefore opt for an empirical approach that confidently approximates the worst-case execution time for **LS**. Specifically, we use 200 streams—the maximum that fits into the memory of a TelosB node using our Blink prototype—and determine their profiles such that the synchronous busy period T_b is maximum for a given bandwidth demand.[5]

To conveniently measure the scheduler's execution time, we run the same code of the real nodes with the MSPsim time-accurate instruction-level emulator [20]. We emulate a 2.5 h execution in which 200 streams are initially admitted one after the other, and measure the execution time of the different steps in Fig. 8 in each round.

Results. Fig. 11 plots the breakdown of the **LS** scheduler execution time as the bandwidth demand increases from 5 % to 95 %. The total execution time increases slowly in the beginning and ramps up as the bandwidth demand exceeds 70 %. As visible in the plot, this is mostly due to an increase in the time needed for admission control, especially for computing the new synchronous busy period prior to the actual admission test. The time for computing the start time of the next round also increases slightly, whereas the time for slot allocation remains almost constant. Overall, we see that the **LS** scheduler needs less than 80 ms even for very high bandwidth demands.

While approximating the worst-case execution time of the **LS** scheduler, the stream sets used in Fig. 11 are arguably quite unrealistic. Our review of existing scenarios in Sec. 2 rather indicates that the typical bandwidth demands would rarely exceed 20 %. In this regime, we measure execution times of 40 ms with 200 streams, which is well below the upper bound of 100 ms we use in our current Blink prototype. Thus, there is room for employing even more constrained platforms, or for scaling up the number of streams with more available memory.

[5] We describe the reasoning and method that led us to determine the stream profiles in App. H.

6 Discussion and Limitations

Our current Blink prototype supports deadlines between 1 and 255 seconds. Thus, it already satisfies the needs of many time-critical monitoring and control applications, especially those featuring installations with many nodes across large areas [9, 33, 34, 45]. In specific closed-loop control settings, however, smaller networks with tens of nodes and tighter deadlines of 10–500 ms are needed [9].

Blink can also support these scenarios by reducing the length of a round. This essentially means to reduce the number and size of slots in a round and the time allotted to the scheduler (see Fig. 1 (B)). In App. I, we detail how the former two are influenced by factors such as packet size and network diameter; for example, in a 3-hop network, 10-byte packets (enough for sensor readings and control signals), $B = 20$ slots per round, and 40 ms for computing the schedule, we can tune our Blink prototype to support deadlines as short as 200 ms.

7 Conclusions

We presented Blink, the first low-power wireless protocol supporting hard real-time traffic in large multi-hop networks at low energy costs. The approach underlying Blink effectively overcomes the fundamental limitations of prior art with respect to adaptiveness to changes in the application's real-time requirements and scalability with respect to time-varying network state. As a result, we demonstrated that, in a real-world setting, our Blink prototype can predictably meet nearly 100 % of packet deadlines, regardless of changes in the stream set or the network state, while being able to schedule 200 real-time packet streams in less than 80 ms on a 16-bit microcontroller running at 8 MHz. We thus maintain that our work provides a major stepping stone to a widespread deployment of time-critical low-power wireless applications.

References

[1] HART communication foundation. http://goo.gl/9WWhD7.

[2] Health and safety legislation—laws in the workplace. http://goo.gl/qiQZVC.

[3] Honeywell Inc: Honeywell Process Solutions—White Paper. http://goo.gl/tlEHe7.

[4] IEEE 802.15.4e Wireless Standard - Amandment1: MAC sublayer. http://goo.gl/W4HIiF.

[5] ISA-100 Wireless Compliance Institute. http://goo.gl/5nySj.

[6] ISO 11898—Controller Area Network (CAN). http://goo.gl/FYyhOV.

[7] ISO 17458—FlexRay Communications System. http://goo.gl/kZhgYY.

[8] Y. Agarwal, B. Balaji, S. Dutta, R. K. Gupta, and T. Weng. Duty-cycling buildings aggressively: The next frontier in HVAC control. In *IPSN*, 2011.

[9] J. Åkerberg, M. Gidlund, and M. Björkman. Future research challenges in wireless sensor and actuator networks targeting industrial automation. In *INDIN*, 2011.

[10] N. Baccour, A. Koubâa, L. Mottola, M. A. Zúñiga, H. Youssef, C. A. Boano, and M. Alves. Radio link quality estimation in wireless sensor networks: A survey. *ACM Trans. Sens. Netw.*, 8(4), 2012.

[11] S. Bouckaert, W. Vandenberghe, B. Jooris, I. Moerman, and P. Demeester. The w-iLab.t testbed. In *TridentCom*, 2010.

[12] G. S. Brodal. A survey on priority queues. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*. Springer, 2013.

[13] G. Buttazzo. Rate monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29:5–26, 2005.

[14] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[15] O. Chipara, C. Wu, C. Lu, and W. Griswold. Interference-aware real-time flow scheduling for wireless sensor networks. In *ECRTS*, 2011.

[16] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, 1974.

[17] R. B. Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969.

[18] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – A lightweight and flexible operating system for tiny networked sensors. In *EmNets*, 2004.

[19] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *EmNets*, 2007.

[20] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón. COOJA/MSPSim: Interoperability testing for wireless sensor networks. In *SIMUTools*, 2009.

[21] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Low-power wireless bus. In *SenSys*, 2012.

[22] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with Glossy. In *IPSN*, 2011.

[23] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *SenSys*, 2009.

[24] Y. Gu, T. He, M. Lin, and J. Xu. Spatiotemporal delay control for low-duty-cycle sensor networks. In *RTSS*, 2009.

[25] T. He, J. A. Stankovic, C. Lu, and T. F. Abdelzaher. A spatiotemporal communication protocol for wireless sensor networks. *IEEE Trans. Parallel Distrib. Syst.*, 16(10), 2005.

[26] Q. Huang, C. Lu, and G.-C. Roman. Mobicast: Just-in-time multicast for sensor networks under spatiotemporal constraints. In *IPSN*, 2003.

[27] V. Kanodia, C. Li, A. Sabharwal, B. Sadeghi, and E. Knightly. Distributed multi-hop scheduling and medium access with delay and throughput constraints. In *MobiCom*, 2001.

[28] K. Leentvaar and J. Flint. The capture effect in FM receivers. *IEEE Trans. Commun.*, 24(5), 1976.

[29] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *IPSN*, 2013.

[30] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[31] C. Lu, B. M. Blum, T. F. Abdelzaher, J. A. Stankovic, and T. He. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *RTAS*, 2002.

[32] T. O'Donovan, J. Brown, F. Büsching, A. Cardoso, J. Cecilio, J. Do Ó, P. Furtado, P. Gil, A. Jugel, W.-B. Pöttner, U. Roedig, J. Sá Silva, R. Silva, C. J. Sreenan, V. Vassiliou, T. Voigt, L. Wolf, and Z. Zinonos. The GINSENG system for wireless monitoring and control: Design and deployment experiences. *ACM Trans. on Sensor Networks*, 10(1), 2013.

[33] K. Ogata. *Modern Control Engineering*. Prentice Hall PTR, 4th edition, 2001.

[34] M. Paavola and K. Leiviska. *Wireless Sensor Networks in Industrial Automation*. Springer, 2010.

[35] Personal communication with Tomas Lennval of ABB Corporate Research.

[36] B. Pfaff. An introduction to binary search trees and balanced trees. Online at http://goo.gl/1tr8IP.

[37] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *IPSN*, 2005.

[38] A. Saifullah, Y. Xu, C. Lu, and Y. Chen. Real-time scheduling for WirelessHART networks. In *RTSS*, 2010.

[39] L. Sha, T. Abdelzaher, K.-E. Årzén, T. Cervin, Anton adn Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004.

[40] S. Shahriar Nirjon, J. Stankovic, and K. Whitehouse. IAA: Interference-aware anticipatory algorithm for scheduling and routing periodic real-time streams in wireless sensor networks. In *INSS*, 2010.

[41] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, INRIA, 1996.

[42] K. Srinivasan, P. Dutta, A. Tavakoli, and P. Levis. An empirical study of low-power wireless. *ACM Trans. on Sens. Netw.*, 6(2), 2010.

[43] J. A. Stankovic, K. Ramamritham, and M. Spuri. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[44] S. Svensson. ABB Corporate Research: Challenges of wireless communication in industrial systems. Keynote talk at ETFA 2011, online at http://goo.gl/0yHyi0.

[45] T. Whittaker. What do we expect from wireless in the factory? In *ETSI Wireless Factory*, 2008.

[46] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. In *MobiCom*, 2001.

[47] F. Xia, Y.-C. Tian, Y. Li, and Y. Sun. Wireless sensor/actuator network design for mobile control applications. *IEEE Sensors*, 7(10), 2008.

[48] H. Zhang, P. Soldati, and M. Johansson. Optimal link scheduling and channel assignment for convergecast in linear WirelessHART networks. In *WiOPT*, 2009.

A Validity of Algorithms

Throughout Sec. 4, 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. 2.

The reason for (*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 [21]. 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 LWB 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 (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.

Although the presentation in Sec. 4 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 device sourced by a single clock) can only occur at discrete times.

B Proof of Theorem 1

Let d_i be the absolute deadline of stream s_i at some point in time; that is, d_i is the deadline of s_i's current packet. Since s_i's relative deadline D_i can be shorter than its period P_i, the current packet may not yet have arrived at this point in time. To determine the maximum number of distinct keys (*i.e.*, absolute deadlines) in the priority queue 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 $s_i, s_j \in \Lambda$.

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. 12. 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

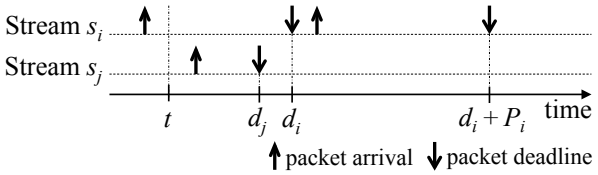


Figure 12: Illustration of the proof of Theorem 1.

$$\Delta_{ij} = d_i + P_i - d_j < t + P_i + P_i - t \leq 2\overline{P}, \qquad (5)$$

where $\overline{P}$ is an upper bound on the period of any stream. Because all absolute deadlines are integers, the strict inequality in (5) implies that Δ_{ij} is at most $2\overline{P} - 1$.

C 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 because there may be less than B pending packets by the time the next round starts. Algorithm 2 determines the 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. 13, 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

$$d_i - \overline{P} > d_{min} + \overline{P} - 1 - \overline{P} \geq t_{i+1}. \qquad (6)$$

Algorithm 2 Slot Allocation

Input Bucket queue of the current set of streams (smaller *absDeadline* implies higher priority), the start time of the next round t_{i+1}, and the upper bound on any stream's period $\overline{P}$.
Output Allocation of streams in EDF order to at most B slots.

```
initialize slot counter c to 0
position traverser t at highest-priority stream using FIRST(t)
horizon = t.absDeadline + P̄ − 1
while (c < B) and (t.absDeadline ≤ horizon) do
    if t.releaseTime ≤ 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 with DECREASEKEY(t, t.period)
    end if
    advance t to next stream in EDF-order using NEXT(t)
end while
```

The strict inequality in (6) 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.

D Proof of Theorem 2

A *schedule* $\mathbf{S}$ 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 can not be any schedule $\mathbf{S'} \neq \mathbf{S^{LS}}$ such that $\mathbf{S'}$ is realtime-optimal and some round starts *later* in $\mathbf{S'}$ 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 network-wide energy consumption.

Let $t_i^{\mathbf{LS}}$ and t'_i denote the start times of the i-th round in $\mathbf{S^{LS}}$ and $\mathbf{S'}$, 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 $\mathbf{S'}$, respectively.

Assume some round in $\mathbf{S'}$ starts later than in $\mathbf{S^{LS}}$. Let the m-th round be the first such round, that is,

$$m = \min\{i \mid t'_i > t_i^{\mathbf{LS}}\}. \tag{7}$$

In $\mathbf{S^{LS}}$, the m-th round starts at $t_m^{\mathbf{LS}}$ since, according to Theorem 3, 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 [21],
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, from the definition of m in (7), we have $t'_m - t'_{m-1} > t_m^{\mathbf{LS}} - t_{m-1}^{\mathbf{LS}} = T_{max}$. This violates the requirement that the time between the start of any two consecutive rounds in $\mathbf{S'}$ must not exceed T_{max}.

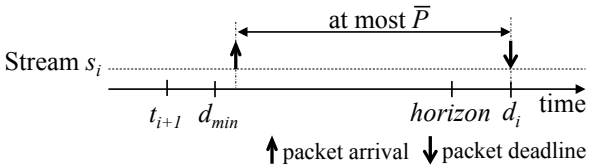


Figure 13: Illustration of the second termination criterion in Algorithm 2. *Any stream s_i with an absolute deadline d_i greater than horizon $= d_{min} + \overline{P} - 1$ releases its current packet after the start of the next round at time t_{i+1} and therefore need not be considered for slot allocation.*

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

$$\eta^{\mathbf{LS}}(t_m^{\mathbf{LS}}, t^*) > B(t^* - t_m^{\mathbf{LS}} - 1). \tag{8}$$

On the other hand, since $t'_m > t_m^{\mathbf{LS}}$ due to the definition of m in (7), the number of packets transmitted in $\mathbf{S'}$ in the interval $[t'_m, t^*]$, denoted $\eta'(t'_m, t^*)$, is upper-bounded as

$$\eta'(t'_m, t^*) \leq B(t^* - t'_m) \leq B(t^* - t_m^{\mathbf{LS}} - 1). \tag{9}$$

From (8) and (9) follows a strict inequality

$$\eta^{\mathbf{LS}}(t_m^{\mathbf{LS}}, t^*) > \eta'(t'_m, t^*). \tag{10}$$

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 $\mathbf{S'}$, and in $\mathbf{S^{LS}}$ as many pending packets as possible are sent in each round. Combining this with (10), we have

$$\eta^{\mathbf{LS}}(0, t^*) > \eta'(0, t^*). \tag{11}$$

Thus, $\mathbf{S^{LS}}$ tightly meets all deadlines at time t^*, while sending more packets than $\mathbf{S'}$. As $\mathbf{S^{LS}}$ prioritizes packets using EDF, which is realtime-optimal [16, 30], $\mathbf{S'}$ necessarily misses a deadline at or before time t^*. This contradicts the assumption that $\mathbf{S'}$ is realtime-optimal.

For either condition that impacts the choice of $t_m^{\mathbf{LS}}$, we have shown that the assumptions on $\mathbf{S'}$ are contradicted.

E Proof of Theorem 3

Due to time synchronization constraints imposed by the LWB communication support, 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 [21]. This implies the first component of the min-operation in (2).

We now show that the second component of the min-operation in (2) 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,

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

Dividing both sides by the positive quantity B,

$$t - t_{i+1} \geq h_i(t)/B. \tag{13}$$

Since $m \geq x$ if and only if $m \geq \lceil x \rceil$ for any integer m and real number x,

$$t - t_{i+1} \geq \lceil h_i(t)/B \rceil. \tag{14}$$

Rearranging terms,

$$t_{i+1} \leq t - \lceil h_i(t)/B \rceil. \tag{15}$$

In particular,

$$t_{i+1} \leq \min_{t \geq t_i+1 \wedge h_i(t)>0} (t - \lceil h_i(t)/B \rceil). \tag{16}$$

The min-operation in (16) 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 $\hat{t} > t_i + T_{max} + T_b + 1$,

$$\hat{t} = \operatorname*{arg\,min}_{t \geq t_i+1 \wedge h_i(t)>0} (t - \lceil h_i(t)/B \rceil). \tag{17}$$

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 $t_i + T_{max}$,

$$t_{i+1} = \hat{t} - \lceil h_i(\hat{t})/B \rceil < t_i + T_{max}. \tag{18}$$

Rearranging terms,

$$\lceil h_i(\hat{t})/B \rceil = \hat{t} - t_{i+1}. \tag{19}$$

Since $\lceil x \rceil = m$ if and only if $m - 1 < x \leq m$ for any integer m and real number x,

$$\hat{t} - t_{i+1} - 1 < h_i(\hat{t})/B. \tag{20}$$

Multiplying both sides by the positive quantity B,

$$(\hat{t} - t_{i+1} - 1)B < h_i(\hat{t}). \tag{21}$$

We interpret (21) 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 (18). However, T_b is the length of the synchronous busy period, which is the longest possible busy period [41]. This contradicts the supposition on the existence of $\hat{t}$.

Second, we need to perform the min-operation in (16) only at times when $h_i(t)$ has discontinuities. In fact, $h_i(t)$ is a right-continuous function with discontinuities only at times that coincide with the deadline of a packet. Thus, we can restrict the domain of the min-operation to the set of deadlines $\mathscr{D}_i$ 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 (16) yields the largest possible t_{i+1} in the case of equality, we obtain the second component of the min-operation in (2)

$$T_i = \min_{t \in \mathscr{D}_i} (t - \lceil h_i(t)/B \rceil). \tag{22}$$

Finally, because EDF is realtime-optimal [16, 30], the necessary condition in (22) is also a sufficient one.

F 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 [43]

$$\omega^0 = \frac{n}{B} \quad \text{and} \quad \omega^{m+1} = \frac{1}{B} \sum_{i=1}^{n} \left\lceil \frac{\omega^m}{P_i} \right\rceil \tag{23}$$

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

As discussed in Sec. 4.3, 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 essentially entails going through the same processing that underlies (23) 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

Algorithm 3 Compute Synchronous Busy Period

Input A bucket queue based copy of the current set of streams, where $s.releaseTime$ is initialized 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 = FINDMIN()
while (s.releaseTime = 0) or (s.releaseTime < T_b and c = 0)
or (s.releaseTime ≤ T_b and c > 0) do
    if current round has only one free slot (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 DECREASEKEY(s, s.period)
    set s to highest-priority stream using s = FINDMIN()
end while
if current round has at least one allocated slot (c > 0) then
    set T_b = T_b + 1 to round up to the next discrete time
end if
```

time.[6] 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.

G Performing Admission Control

As noted in Sec. 4.4, the challenge in implementing Theorem 4 is to efficiently compute the demand $h_0(t)$. Using the closed-form expression found in the literature [14]

$$h_0(t) = \sum_{i=1}^{n} \left\lfloor \frac{t + P_i - D_i}{P_i} \right\rfloor \tag{24}$$

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

Thus, as already described in Sec. 4.4, 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 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.

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

Algorithm 4 Admission Control

Input A bucket queue based copy of the current set of streams including the new stream, with $s.absDeadline$ initialized to D_i for all streams s_i (smaller *absDeadline* implies higher priority), the utilization as well as the deadline-based utilization of that stream set, and the new synchronous busy period T_b.
Output Whether the new stream is to be admitted or 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 ≤ availableBandwidth do
    set s to highest-priority stream using s = FINDMIN()
    if s.absDeadline > t then
        if s.absDeadline > T_b then
            return "admit"
        end if
        availableBandwidth = t × B
        t = s.absDeadline
    end if
    demand = demand + 1
    s.absDeadline = s.absDeadline + s.period
    update priority of s using DECREASEKEY(s, s.period)
end while
return "reject"
```

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 [30]. 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 [43]. We incrementally update both types of utilizations as streams are added and removed at runtime.

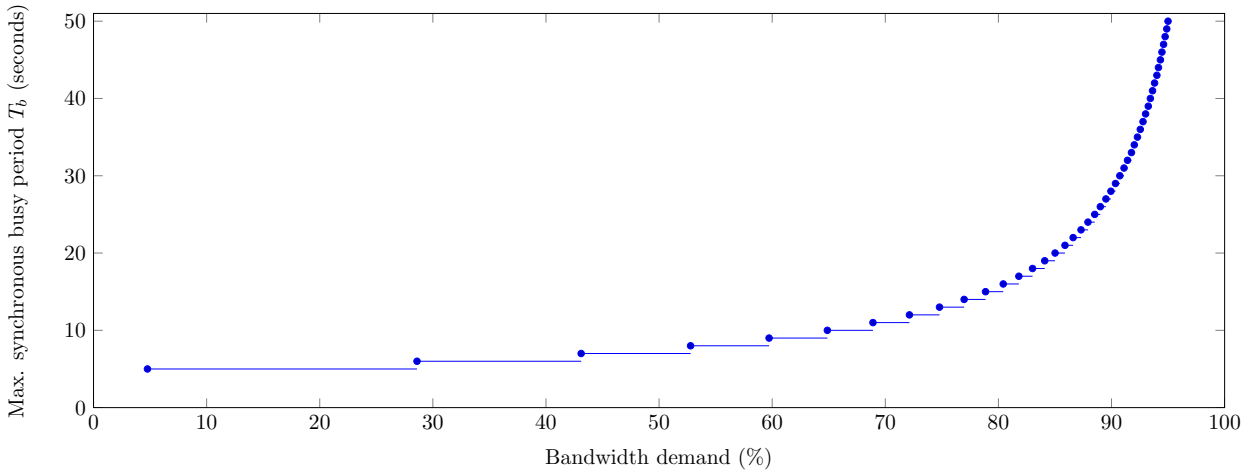


Figure 14: The maximum synchronous busy period T_b as a function of the bandwidth demand, for $N = 200$ streams, a maximum stream period of $\overline{P} = 255$ seconds, and $B = 51$ available slots per round.

H Approximating the Worst-case Execution Time of the LS Scheduler

In the worst case, one single execution of the **LS** scheduler needs to proceed through all four steps in Fig. 8. A careful analysis of the corresponding algorithms (see 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. 5.3.

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 $x_1, x_2, \ldots, x_{\overline{P}}$, 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:

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

The objective function is the bandwidth demand. The inequality constraints ensure that at the end of each interval $[0, t]$, $t \in \{1, 2, \ldots, T_b - 1\}$, 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

Table 2: Worst-case stream profiles used in the execution time experiment of Sec. 5.3. 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 shown in the table.

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 × 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

a function $f(u)$ that gives the maximum T_b for a given bandwidth demand u, as shown in Fig. 14. 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:

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

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

Settings. We proceed in two steps to approach the worst-case execution time of the **LS** scheduler in Sec. 5.3.

First, 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 95 %, with $B = 51$ available slots per round. Table 2 lists the different sets of worst-case stream profiles.

Second, we emulate for each set of 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 we also use on the real nodes [20]. During an execution, requests for each of the 200 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. 8). We keep measuring even after all streams are admitted. Then, we take *for each step individually* the maximum execution time we measured throughout the 2.5 h execution. Fig. 11 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. 5.3 are *higher* than the maximum total execution time we actually measured in one round.

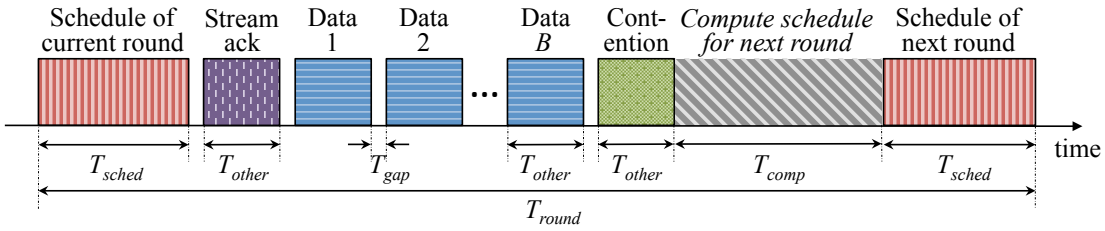


Figure 15: Communication slots and processing in a complete LWB round in our Blink prototype.

I Supporting Sub-second Deadlines

The fixed length of a round, T_{round}, determines the shortest possible period P_i and relative deadline $D_i \leq P_i$ of a stream s_i in Blink. To make our Blink prototype support shorter deadlines than $T_{round} = 1$ second, we must reduce the length of a round while carefully considering a number of influencing factors, as discussed in the following.

Length of a round. To reason about (a lower bound on) T_{round}, we must consider a *complete* round composed of all possible slots and processing activities. Fig. 15 provides a zoomed-in view of Fig. 1 (B), showing the slots and activities in a complete round in our prototype.

Every round starts with a slot in which the host distributes the schedule for the current round. This schedule allows each node to update its synchronization state and specifies the nodes that send in the following slots. Next, there is a slot in which the host sends a possible stream acknowledgment, informing a node whether its requested stream has been admitted or not. Nodes send their data packets in the following B data slots. In the contention slot, nodes compete to transmit their stream requests.[7] Based on received stream requests and all streams admitted thus far, the host computes the schedule for the next round. Finally, the host distributes the new schedule, so the nodes know when the next round starts and can thus turn off their radios until then to save energy.[8]

As visible in Fig. 15, there is a small gap between consecutive slots. This gap gives LWB just enough time to put a received packet into the incoming packet queue (at the intended receives) and to fetch the packet that is to be sent in the next slot from the outgoing packet queue (at the respective senders). To enable this operation, the application must ensure that all released packets are in LWB's outgoing queue before a round starts. Conversely, LWB ensures that when a round ends, all received packets are in the incoming queue. In fact, as mentioned earlier, it is not until this time that the application gains control of the MCU to process received packets.

Using the notation in Fig. 15, we add up the durations of the different slots, gaps, and schedule computation to obtain an expression for the (minimum) length of round

$$T_{round} = 2T_{sched} + (B+2)(T_{other} + T_{gap}) + T_{comp}. \quad (25)$$

From (25) follows that we can make a round shorter by reducing (*i*) the number of data slots in a round B, (*ii*) the time for computing the schedule T_{comp}, or (*iii*) the size of schedule T_{sched} and other slots T_{other}. (*i*) is an application-specific trade-off between a higher energy overhead per round (relative to the B useful data slots) and the possibility to support shorter periods and deadlines. (*ii*) depends on the scheduling policy and the processing demand induced by the application-dependent streams. Finally, (*iii*) is a function of several application, deployment, and platform characteristics, as discussed next.

Length of a slot. Each slot within a round consists of a Glossy flood. To obtain (a lower bound on) the length of a slot, we need to briefly recap the operation of Glossy.

As shown in Fig. 16, the *initiator* (*i.e.*, the node that is to send in a slot according to the schedule) starts the flood by transmitting its packet, while all other nodes, the *receivers*, have their radios turned on. Nodes within the initiator's radio range, the *1-hop receivers*, receive the packet at the same time and, by ensuring a constant processing time across all nodes, they also relay the packet at the same time. As this operation continues, nodes that are two, three, or more hops away from the initiator also receive and relay the packet. To achieve packet reliabilities above 99.9 %, each node transmits the packet up to N times during a Glossy flood [22]. For example, in Fig. 16, each node transmits $N = 2$ times.

The length of a slot, T_{slot}, should be sufficient to allow also the nodes that are farthest away from the initiator to *receive* N times. Let H be the network diameter (*i.e.*, the maximum hop distance between any two nodes) and T_{hop} the time between consecutive transmissions during a flood (see Fig. 16). T_{hop} is constant during a flood, since nodes do not alter the packet size. Therefore, the length

[7] A node uses the contention slot only to submit its *first* stream request. Once a node has an admitted stream, it submits further request to add, remove, or update streams by piggybacking on data packets [21].

[8] Like the original LWB scheduler [21], Blink's real-time scheduler allocates slots for a stream acknowledgment and data packets as needed, and schedules the contention slot less often if no stream request recently arrived. The length of a round remains constant nevertheless.

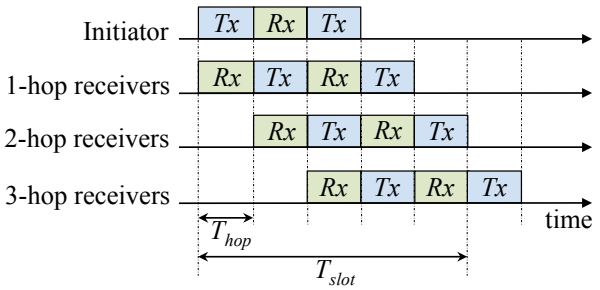


Figure 16: Illustration of a Glossy flood in a 3-hop network, where each node transmits $N = 2$ times. *The length of a slot T_{slot} should be sufficient to give all nodes in the network the chance to receive the packet N times.*

of a slot such that each node in a H-hop network gets the chance to receive the packet N times during a flood is

$$T_{slot} = (H + 2N - 2)T_{hop}. \tag{26}$$

We obtain an expression for T_{hop} by adding up the time required for the actual packet transmission (or reception) T_{tx}, the processing delay of the radio at the beginning of a packet reception T_d, and the software delay introduced by Glossy when triggering a packet transmission T_{sw}

$$T_{hop} = T_{tx} + T_d + T_{sw}. \tag{27}$$

T_d is a radio-dependent constant and T_{sw} is a constant specific to the Glossy implementation for a given platform; Table 3 lists their values for the TelosB and the CC2420 radio. The time needed by the CC2420 to transmit an IEEE 802.15.4-compliant packet is the sum of the time needed to calibrate the radio's internal voltage controlled oscillator T_{cal}, the time for transmitting the 5-byte synchronization header and the 1-byte PHY header T_{header}, and the time for transmitting the MAC protocol data unit $T_{payload}$, which contains the actual payload

$$T_{tx} = T_{cal} + T_{header} + T_{payload}. \tag{28}$$

The values of T_{cal} and T_{header} are listed in Table 3. The time required to transmit a payload of size $L_{payload}$ (between 1 and 127 bytes accoring to IEEE 802.15.4) using

Table 3: Constants specific to the CC2420 radio and the Glossy implementation for the TelosB platform we use.

Name	Value
T_d	$3\,\mu$s
T_{sw}	$23.5\,\mu$s
T_{cal}	$192\,\mu$s
T_{header}	$192\,\mu$s
R_{bit}	250 kbps

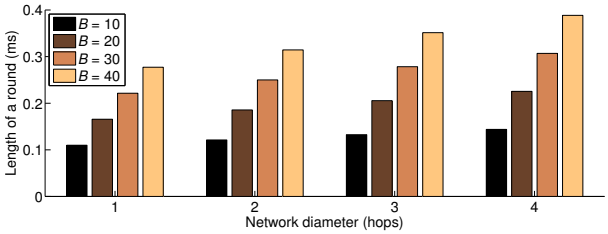


Figure 17: Length of a round in Blink depending on network diameter H and number of data slots in a round B.

a radio that has a transmit bit rate of R_{bit} (see Table 3) is

$$T_{payload} = 8L_{payload}/R_{bit}. \tag{29}$$

Discussion. We use the above expressions to estimate the length of a round T_{round} in our Blink prototype, targeting the TelosB platform and the CC2420 radio. T_{round} determines the shortest possible period P_i and deadline D_i of a stream in Blink. Since short periods are particularly important in specific closed-loop control scenarios [9], we consider typical characteristics of those.

The networks are typically small, containing up to 50 nodes [9]. Assuming that the number of streams is on the same order, our results from Sec. 5.3 with 200 streams suggest that the **LS** scheduler completes for sure within $T_{comp} = 40$ ms for a wide range of bandwidth demands. A payload of $L^{other}_{payload} = 10$ bytes is sufficient for actuation signals, sensor readings, stream requests, and stream acknowledgments. A schedule packet consists of a 7-byte header and a sequence of node/stream IDs, so the payload of schedule packets is $L^{sched}_{payload} = 7 + 2(B+2)$ bytes. We set $T_{gap} = 3$ ms and want every node in the network to receive the packet at least $N = 2$ times—at this setting, Glossy provides a packet reliability above 99.9 % in real-world experiments with more than 100 nodes [22].

Given these settings, Fig. 17 plots the length of a round T_{round} depending on network diameter H and number of data slots in a round B. For example, in a 3-hop network and $B = 20$ slots per round, we can tune our Blink prototype to support periods and deadlines as short as 200 ms. Thus, under given assumptions, Blink satisfies the needs of specific closed-loop control scenarios in terms of high refresh rates and hard end-to-end packet deadlines [9].