End-to-end Real-time Guarantees in Wireless Cyber-physical Systems

Romain Jacob* Marco Zimmerling† Pengcheng Huang* Jan Beutel* Lothar Thiele*
*Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
†Networked Embedded Systems Group, TU Dresden, Germany
firstname.lastname@tik.ee.ethz.ch marco.zimmerling@tu-dresden.de

***Abstract*—In cyber-physical systems (CPS), the communication among the sensing, actuating, and computing elements is often subject to hard real-time constraints. Real-time communication among wireless network interfaces and real-time scheduling for complex, dynamic applications have been intensively studied. Despite these major efforts, there is still a significant gap to fill. In particular, the integration of several real-time components to provide *end-to-end* real-time guarantees between interfaces of distributed applications in wireless CPS is an unsolved problem. We thus present a distributed protocol that considers the complete transmission chain including peripheral busses, memory accesses, networking interfaces, and the wireless real-time protocol. Our protocol provably guarantees that message buffers along this chain do not overflow and that all messages received at the destination application interface meet their end-to-end deadlines. To achieve this while being adaptive to unpredictable changes in the system and the real-time traffic requirements, our protocol establishes at run-time a set of *contracts* among all major elements of the transmission chain based on a worst-case delay and buffer analysis of the overall system. Using simulations, we validate that our analytic bounds are both safe and tight.**

I. Introduction

Cyber-physical systems (CPS) tightly integrate components for sensing, actuating, and computing into distributed feedback loops to directly control physical processes [1]. The successful deployment of CPS technology is widely recognized as a grand challenge to solving a number of societal problems in domains ranging from healthcare to industrial automation. To reach into new areas and realize systems with unprecedented capabilities, there is a trend toward increasingly smaller and autonomously powered CPS devices that exchange data through a low-power wireless communication substrate. As many CPS applications are mission-critical and physical processes evolve as a function of time, the communication among the sensing, actuating, and computing elements is often subject to real-time requirements, for example, to guarantee stability of the feedback loops [2].

Challenges. These real-time requirements are often specified from an *end-to-end application* perspective. For example, a control engineer may require that sensor readings taken at time t_s are available for computing the control law at $t_s + D$. Here, the *relative deadline* D is derived from the activation times of *application tasks*, namely the sensing and control tasks, which are typically executed on physically distributed devices. Meeting such end-to-end deadlines is non-trivial, because data transfer between application tasks involves multiple other tasks (*e.g.*, operating system, networking protocols) and shared resources (*e.g.*, memories, system busses, wireless medium).

The problem of real-time communication *between network interfaces* of sources and destinations in a low-power wireless network has been studied for more than a decade [3]–[5]. Today, standards such as WirelessHART [6] and ISA100.11a [7] for control applications in the process industries exist [8], and considerable progress in real-time transmission scheduling and end-to-end delay analysis for WirelessHART networks has been made [9], [10]. Despite these efforts, the problem of integrating a wireless real-time protocol, such as WirelessHART [6] or Blink [11], with the rest of the system to provide end-to-end real-time guarantees between *distributed application interfaces* remains unsolved. To fill this gap, we argue that the entire transmission chain involving peripheral busses, memory accesses, networking interfaces, and the wireless networking protocol must be taken into account.

To support a broad spectrum of CPS applications, a solution to the problem should provide the following properties

- **P1** Messages received by the destination application interface do so before their *hard end-to-end deadlines*.
- **P2** Messages received at the wireless network interface are *successfully delivered* to their destination application interface (*i.e.*, local buffer overflows are prevented).
- **P3** At runtime, the solution *adapts* to dynamic changes in the system state and the real-time traffic requirements.
- **P4** Existing hardware and software components can be freely *composed* to satisfy specific application's needs, without altering the properties of the integrated parts.
- **P5** The solution *scales* to large system sizes and *operates efficiently* with regard to limited resources such as energy, wireless bandwidth, computing capacity, and memory.

A major challenge in providing these properties is to funnel messages in real-time through tasks that run concurrently and access shared resources. Interference on such resources can delay tasks and communication arbitrarily, therefore hampering timing predictability (**P1**–**P3**) and composability (**P4**).

To avoid interference on the wireless medium, real-time protocols like WirelessHART [6] typically use a time division multiple access (TDMA) scheme, whereby a centralized scheduler allocates exclusive time slots to nodes for message transmissions. One way of integrating the wireless protocol with the rest of the system while avoiding interference would be to jointly schedule transmissions in the network and *all*

other tasks in the system. Although such a completely time-triggered approach may be suitable for some wired embedded systems [12], it is hardly practical in a dynamic wireless setting, where tasks are often triggered by external events and the system must adapt to changes in environmental conditions [13], traffic and computing demands, available energy [14], and node failures and recoveries [15] (**P3**, **P5**).

Contribution. This paper proposes an approach to integrate a wireless real-time communication protocol into CPS. By considering the whole transmission chain, we define constraints on application schedules such that end-to-end real-time guarantees between application interfaces can be enforced. Our solution supports properties **P1–P5** by acting on two levels.

On the device level, we propose to dedicate a communication processor (CP) exclusively to the real-time network protocol and to execute all other tasks on an application processor (AP). We leverage the Bolt interconnect [16], which decouples two processors in the time, power, and clock domains, while allowing them to asynchronously exchange messages within predictable time bounds. Thus, on each device, we *decouple* communication from application tasks, which can be *independently* invoked in an event- or time-triggered fashion. As a result, we guarantee the faithfulness of the network interface (**P2**), we support composability (**P4**), and we leverage the recent trend toward ultra low-power multi-processor architectures that can be chosen to match the needs of the application and the networking protocol efficiently (**P5**).

On the system level, we design a *distributed real-time protocol* (DRP) that provably guarantees that all messages received at the application interfaces meet their end-to-end deadlines (**P1**) and all message buffers along the transmission chain do not overflow (**P2**). To accomplish this while being adaptive to unpredictable changes (**P3**), DRP dynamically establishes *at run-time* a set of *contracts* depending on the current real-time traffic demands in the system. A contract determines the mutual obligations between a source or destination device and the networking protocol, both in terms of minimum service provided and maximum demand generated. In this way, we guarantee end-to-end deadlines without impairing the decoupling of communication from application tasks.

After discussing related work in Sec. II and stating the problem in Sec. III, we describe our design throughout Secs. IV and V. In Sec. VI, we describe how to practically implement the design concepts of DRP. It requires an analysis of worst-case end-to-end communication delay and message buffer sizes, which we perform using classical analysis techniques for distributed real-time systems [17], [18]. Sec. VII explores the impact of the protocol's design parameters on the system performance and the corresponding theoretical limits. Finally, based on real performance numbers of Bolt [16] and parameters of the Blink wireless real-time communication protocol reported in the literature [11], [19], [20], we simulate DRP. The results, discussed in Sec. VIII, show that our analytical bounds are safe and have a low degree of pessimism: in some cases the simulation results are within 4 % of the analytical worst-case bounds. The Appendix provides details on the worst-case delay analysis.

II. Related Work

Providing end-to-end guarantees in distributed networked systems has a long history in the context of the Internet. Notable developments are the resource reservation protocol (RSVP) that combines flow specification, resource reservation, admission control, and packet scheduling to achieve end-to-end quality of service (QoS) [21]. Network calculus [17] provides some of the necessary theoretical concepts to determine bounds on buffer sizes and delay in communication networks. Extension toward hard real-time computing and communication systems is known as real-time calculus [18]. The analysis of distributed hard real-time systems also has a long history [22], and so do compositional analysis frameworks, such as MAST [23], SymTA/S [24] and MPA [25].

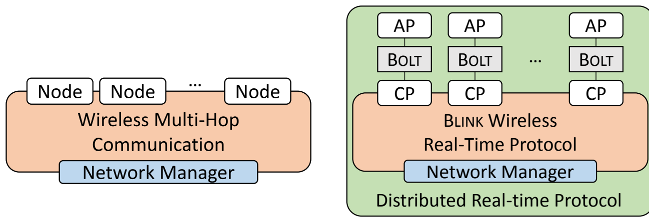
Early works on real-time communication in sensor networks consider classical non-deterministic routing protocols [3]–[5], thus providing only soft guarantees. Stankovic et al. [4] even argue that specific message delivery orderings, such as those useful to apply established dependability techniques [15], are impossible to guarantee in a multi-hop low-power wireless network. More recently, standards like WirelessHART [6] have been analyzed to provide communication guarantees [9], [10]. But [9] is based on NP-hard multiprocessor scheduling and requires a global network view, which limits its adaptability to dynamic changes in the system [26]. Other wireless real-time protocols have been described recently [8], [27]. However, the integration of these protocols into a methodology to provide end-to-end real-time guarantees between *application interfaces* is unsolved. We address this problem in this paper.

Recently, a game-changing approach to wireless multi-hop communication using synchronous transmissions has been described [11], [19], [20]. It avoids the computation of multi-hop routing paths and per-node communication schedules based on, for example, neighbor lists and link qualities, because the protocol logic is independent of such volatile network state. Experiments on several large-scale testbeds show that the approach is highly adaptive and achieves an end-to-end packet reliability higher than 99.9 % [19], [20]. Furthermore, the few packet losses can be considered statistically independent [28], which eases the design of CPS controllers that can deal with intermittent observations [29]. Although our approach can be adapted to other types of communication protocols, the paper is based on this concept of synchronous transmissions.

III. System Model and Problem Statement

The problem we aim to solve in this paper is a function of the application requirements and the system architecture.

Application requirements. CPS use feedback loops to control physical processes [2]. Because physical processes evolve over time, their timing must be intimately connected to the timing in the cyber domain of computing and communication. For this



(a) A set of nodes $\mathcal{N}$, each with a single processor, execute the application and exchange messages via wireless multi-hop communication.

(b) Application (AP) and communication (CP) processors exchange messages via Bolt; the CPs run the Blink real-time wireless protocol.

Figure 1. Traditional (a) and our proposed (b) system architecture. *A logically global network manager arbitrates access to the shared wireless medium.*

reason, the exchange of sensor data and control signals among distributed CPS devices is subject to *real-time constraints*.

Let $\mathcal{F}$ be the set of real-time message *flows* in the system. Each flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$ is defined by a *source application* running on *source node* n_i^s that *releases* messages with a *minimum message interval* T_i and *jitter* J_i ($J_i < T_i$), such that the time span of n successive messages is never smaller than $(n-1)T_i - J_i$ for any n. Thus, message release is *sporadic with jitter*. Every message released at n_i^s should be delivered to the application running on *destination node* n_i^d within the same *relative end-to-end deadline* $\mathbf{D}_i$.

System architecture. Fig. 1(a) shows the overall system architecture. It consists of a set of *nodes* $\mathcal{N}$ that exchange messages via *wireless multi-hop communication*; that is, messages sent from a source node to a destination node are possibly relayed by multiple other nodes. A logically global *network manager* arbitrates access to the shared networking resource. Physically, the network manager may run on one of the nodes. The source and destination applications of a flow F_i run on physically distributed nodes n_i^s and n_i^d. A node can send and receive messages to and from several other nodes in the system.

Problem statement. The problem is to design a protocol that supports properties **P1**–**P5** such that every message of every flow $F_i \in \mathcal{F}$ released at the source node n_i^s, given its successful transmission by the wireless network, is delivered to the destination application on node n_i^d within $\mathbf{D}_i$ time units.

IV. Design Overview

We present a solution to the above problem. Before delving into the details of our design, we provide in this section a high-level overview of the principles underlying our solution. Conceptually, our design is based on three building blocks:

1) a decoupling of (wireless multi-hop) communication from application using a Bolt-based *dual-processor architecture* [16] to avoid interference at the device level;
2) a *wireless real-time protocol* that delivers a high fraction of messages from source to destination *network interfaces* within a given *network deadline* D;
3) a *distributed real-time protocol* that manages resources across the network, decouples responsibilities between components, and ensures that end-to-end deadlines $\mathbf{D}$ *between application interfaces* are met.

We now motivate the use of each of these building blocks and explain how they support achieving properties **P1**–**P5**.

Dual-processor architecture. When using the traditional system architecture shown in Fig. 1(a), application and communication tasks execute concurrently on a node. When both tasks attempt to simultaneously access shared resources (*e.g.*, memory, processor, system bus), one of them will be delayed for an arbitrary amount of time. Such resource interference hampers end-to-end timing predictability, and may alter the functional properties of a task, thus defeating system composability.

To tackle this issue, we use the system architecture shown in Fig. 1(b), where each node is replaced with a dual-processor platform. One processor (AP) runs the application, while the other processor (CP) only runs the wireless multi-hop communication protocol. Using the Bolt processor interconnect [16], AP and CP are decoupled in time, power, and clock domains, and can asynchronously exchange messages with bounded delay. As a result, this building block helps toward end-to-end timing predictability (**P1**) and allows for composing processors and software components to satisfy the application demands (**P4**), which also aids in achieving low-power operation (**P5**).

Wireless real-time protocol. As discussed in Sec. II, providing real-time guarantees across multi-hop low-power wireless networks is challenging. Out of the many solutions that have been proposed, Blink [11] is one of the few wireless real-time protocol satisfying our needs. Specifically, Blink delivers messages within real-time deadlines (**P1**), reliably (**P2**), and at low energy costs (**P5**) between the CPs (*i.e.*, the network interfaces, see Fig. 1(b)), while being highly adaptive to dynamic changes in the wireless network and the traffic demands (**P3**). We use Blink to illustrate how our approach materializes in a concrete solution. Nevertheless, the underlying principles we present could be adapted to other wireless real-time protocols.

Distributed real-time protocol. Using Bolt, we decouple the Blink wireless real-time protocol running on the CPs from the application running on the APs. This decoupling has many benefits, including the flexibility in how each of the two operates (*i.e.*, time- vs. event-triggered), but it also poses a major challenge: while AP and CP should execute independently, it is their *joint* operation that determines whether or not messages exchanged between two APs (*i.e.*, application interfaces) meet their end-to-end deadlines.

To address this challenge, we introduce a distributed real-time protocol (DRP) as the third building block of our solution. DRP strikes a balance between the decoupling of APs, CPs, and Blink on the one hand and (predictable) end-to-end latency of messages between application interfaces (*i.e.*, the APs) on the other hand. To realize this trade-off, DRP entails the notion of *contract*. A contract settles the minimum required agreement between APs, CPs, and Blink so that they can operate as much as possible independently from each other, thus preserving **P4**, while ensuring that end-to-end deadlines $\mathbf{D}$ are met (**P1**, **P2**). DRP establishes contracts at runtime as new flows are requested and existing ones are removed (**P3**), and scales well to large sets of real-time message flows (**P5**).

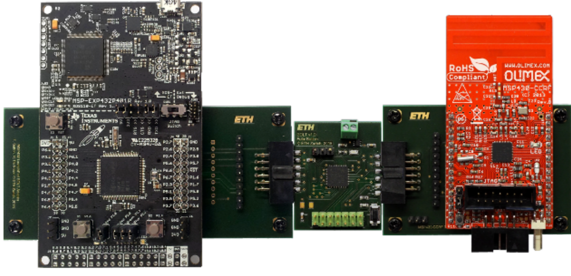


Figure 2. Example of a custom-built heterogeneous dual-processor platform. *Bolt interconnects a powerful application processor (TI MSP432) on the left with a state-of-the-art communication processor (TI CC430) on the right.*

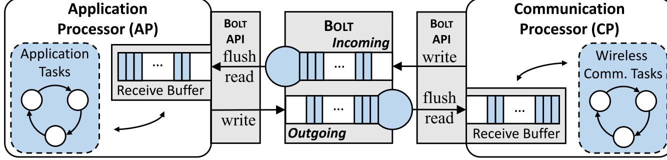


Figure 3. Conceptual view of the Bolt processor interconnect. *Using functions* `write`*,* `read`*, and* `flush`*, the application (AP) and communication (CP) processors can asynchronously exchange messages with predictable latency.*

V. Detailed Design

We now detail the three building blocks of our solution. We first describe how APs and CPs exchange messages through Bolt, then the operation of the Blink wireless real-time protocol, and finally the detailed design of DRP.

A. Bolt Processor Interconnect

Bolt provides predictable asynchronous message passing between two arbitrary processors, and hence decouples the processors with respect to time, power, and clock domains [16]. Fig. 2 shows an example dual-processor platform with Bolt in the middle. The processor on the left is a TI MSP432, which features a powerful ARM Cortex-M4 microcontroller suitable for application processing (AP); the processor on the right is a TI CC430, which has a low-power wireless radio that is driven by a wireless multi-hop communication protocol (CP).

As illustrated in Fig. 3, two message queues with first-in-first-out (FIFO) semantics, one for each direction, form the core of Bolt. Bolt allows for concurrent read and write operations by AP and CP on both queues.

Application programming interface (API). The API of Bolt includes three functions, as listed in Table I. Function `write` appends a message to the end of the outgoing queue, whereas `read` reads and removes the first message from the incoming queue. Calling `flush` results in a sequence of `read` operations with the goal of emptying the incoming message queue.

The implementation of `flush` is peculiar. As Bolt allows for concurrent `read` and `write` operations, in theory, a `flush` may result in an infinite sequence of `read` operations. To prevent this, the number of `read` during a `flush` is upper-bounded by f_{max}. We set f_{max} to the number of messages that fit into one Bolt queue, denoted by S_{Bolt},

$$f_{max} = S_{Bolt} \tag{1}$$

Thus, a `flush` terminates either when the incoming queue is found empty or when f_{max} messages have been read out.

Due to its design, Bolt features predictable execution times for all three functions, regardless of the interconnected processors [16]. We denote by C_w, C_r, and C_f the worst-case execution times (WCETs) of `write`, `read`, and `flush`.

Table I
Bolt application programming interface (API)

Function	Description	WCET
`write`	Append a message to outgoing queue	C_w
`read`	Read and remove the first message from incoming queue	C_r
`flush`	Perform up to f_{max} `read` operations, or until incoming queue is empty	$C_f = f_{max} * C_r$

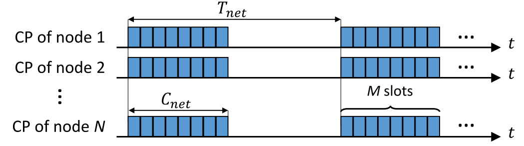


Figure 4. Operation in Blink is globally time-triggered. *Communication occurs in rounds of equal duration C_{net}. Each round consists of a sequence of up to M exclusive time slots, each of which serves to send one message. The interval between the start of consecutive rounds T_{net} may vary. During a round, all CPs in the system participate in the communication.*

B. Blink Wireless Real-time Protocol

In Blink [11], wireless multi-hop communication is globally time-triggered and occurs in *rounds* of equal *duration* C_{net}. Fig. 4 shows that each round serves to send up to M messages within exclusive time slots. In each time slot a message is sent from a given CP to all other CPs with a reliability above 99.9 % [19]. The interval between the start of consecutive rounds, denoted by T_{net}, is determined by the network manager at runtime and is based on the current real-time traffic demands. T_{net}^{min} and T_{net}^{max} are implementation-specific bounds on T_{net}. During a round, all CPs in the system are busy executing Blink, so other tasks (*e.g.*, exchanging messages through Bolt) can only be executed between rounds. In addition to computing the communication schedule for each round, the network manager also checks whether a request for a new flow can be admitted using a schedulability test.

In principle, Blink expects *periodic message arrivals* with known *initial phase* of the first packet. We refer to this as the *expected arrival pattern*. Blink guarantees that for all messages matching the expected arrival pattern, if one is *successfully received* at the destination CP, it is available before its *relative network deadline* D. Because the CPs are busy executing Blink during rounds, the network deadline must be bigger than the round interval (*i.e.*, $D \geq T_{net} \geq T_{net}^{min}$).

In our case, however, APs and CPs operate independently and the actual message release from the APs is sporadic with

jitter, as described in Sec. III. To resolve this mismatch, we let the network manager *assume* that messages are released periodically at Blink's interface, choose arbitrarily the initial phase of flows, and compute Blink's communication schedule based on that expected arrival. The analysis of the system (see Sec. VI) bounds the maximal mismatch between the actual and expected arrival patterns. Our design of DRP uses that bound to guarantee that end-to-end deadlines $\mathbf{D}_i$ are met nonetheless.

C. DRP: Distributed Real-time Protocol

Contracts. DRP aims at providing end-to-end real-time guarantees between distributed application interfaces. Blink already provides real-time guarantees between the network interfaces (*i.e.*, CPs). In addition, DRP dynamically establishes contracts at runtime that satisfy properties **P1** and **P2** by

1) avoiding overflows of message buffers (*e.g.*, the Bolt queues) at the source and destination nodes, thus preventing message losses (**P2**);
2) ensuring that messages are handled "fast enough" between the network (*i.e.*, CPs) and the application (*i.e.*, APs) interfaces, at the source and destination nodes, such that they all meet their end-to-end deadlines (**P1**).

To avoid overflows 1), DRP defines *maximum time intervals* between two `flush` of Bolt by the CPs and APs, denoted by T_f^s and T_f^d respectively. We statically set T_f^s for all CPs so that it does not constrain the achievable end-to-end deadline. On the other hand, DRP dynamically adjusts T_f^d for each AP in the system upon registration of a new flow.

Meeting end-to-end guarantees 2) entails that DRP decides on the *distribution of responsibilities* among the source node, Blink, and the destination node of a flow F_i with regard to meeting the end-to-end deadline $\mathbf{D}_i$. To this end, DRP uses the *deadline ratio* $r \in (0, 1)$, a configuration parameter chosen at design time. The joint responsibility of the source and Blink is a function of the source flushing interval T_f^s and the flow's network deadline D_i, which is computed by DRP. They are responsible for meeting a fraction r of the end-to-end deadline

$$f(T_f^s, D_i) \ \leq \ r * \mathbf{D}_i \tag{2}$$

The remaining part of the end-to-end deadline defines the responsibility of the destination, which is a function of its flushing interval T_f^d

$$g(T_f^d) \ \leq \ (1 - r) * \mathbf{D}_i \tag{3}$$

In Sec. VI, we derive concrete expressions for functions f and g, and we specify how DRP computes D_i and T_f^d. In Sec. VII we detail how the choice of the deadline ratio r influences key performance metrics of wireless CPS application.

Overall, DRP dynamically establishes two contracts for each newly admitted flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i) \in \mathcal{F}$

- **Source $\leftrightarrow$ Blink**: F_i's source application, which runs on AP_s at node n_i^s, agrees to write no more messages than specified by the minimum message interval T_i and the jitter J_i. The attached CP_s prevents overflows of Bolt and its local message buffer. In turn, Blink agrees to serve flow F_i such that any message matching the expected arrival of F_i meets the network deadline D_i (if received).
- **Blink $\leftrightarrow$ Destination**: Blink agrees to deliver no more messages than specified by T_i. In turn, AP_d and CP_d agree to read out all delivered messages such that overflows of Bolt and CP_d's local buffer are prevented and all messages meet F_i's end-to-end deadline $\mathbf{D}_i$.

For any flow, if both contracts are fulfilled, all messages that are successfully delivered by Blink will meet their end-to-end deadline. In practice, the contracts fulfillment is guaranteed by a set of *admission tests*, which are performed in sequence upon registration of a new flow, as described next.

Flow registration. Fig. 5 shows the full procedure for registering a new flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$ in DRP. The flow's source application running on AP_s first computes the network deadline D_i (see how in the next Sec.) before it writes the request to the attached CP_s through Bolt. CP_s uses its admission test to check whether it could still prevent overflows of Bolt and its local memory if F_i were present. If so, CP_s forwards the request to the network manager, which checks the schedulability using Blink's admission test [11]. In case Blink admits the flow, the destination node's CP_d and AP_d check whether they can prevent overflows of CP_d's local memory and Bolt, respectively. Moreover, AP_d re-computes its required flushing interval T_f^d and checks using mainstream schedulability analysis [30] whether it can support this new load (in addition to the load incurred by other tasks running on AP_d). DRP registers a flow only if all admission tests succeed, which triggers changes in the runtime operation (*i.e.*, schedule) of AP_s, Blink, and AP_d.

Flow requests and acknowledgments are sent through specific flows, registered at system bootstrapping.

VI. Concrete Realization of DRP

To implement DRP, one needs to define the fixed flushing interval T_f^s of the CPs (Sec. VI-A), and how to dynamically compute the network deadline D_i of a flow F_i and the flushing interval T_f^d of each AP (Sec. VI-B). Then, a worst-case buffer analysis (Sec. VI-C) will allow to formulate admission tests (Sec. VI-D), one for APs and one for CPs. The success of all admission tests guarantees that both contracts **Source $\leftrightarrow$ Blink** and **Blink $\leftrightarrow$ Destination** can be satisfied by DRP.

Fig. 6 summarizes the various inputs and outputs of DRP. Hardware parameters (related to Bolt) and design parameters (*i.e.*, the length of a communication round C_{net}, the deadline ratio r, and the number of slots per round M) are constants known to all components. The application's real-time communication requirements may change at runtime as new flows are requested and existing flows are removed. DRP determines T_f^s statically, while all other outputs are dynamically computed whenever the set of flows changes, according to the procedure illustrated in Fig. 5.

A. Setting CPs' Flushing Interval

To guarantee that all CPs fulfill their share of the contracts (*i.e.*, prevent buffer overflows), we conceive a time-triggered

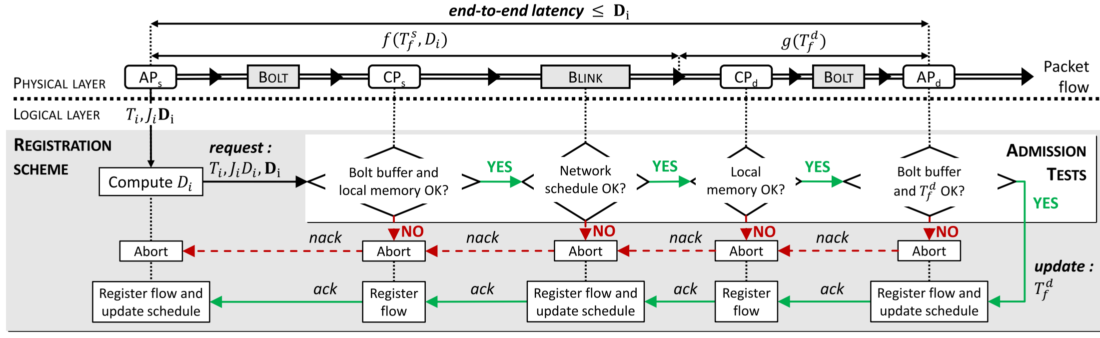


Figure 5. Steps and components involved when registering a new flow in DRP. *Given a request for a new flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$, the source application running on AP_s at node n_i^s computes the flow's network deadline D_i. Then, all components check one after the other using specific admission tests whether they can admit the new flow. DRP registers a new flow only if all admission tests succeed, which eventually triggers changes in the runtime operation (i.e., schedule) of Blink as well as of the source and destination application processors AP_s and AP_d.*

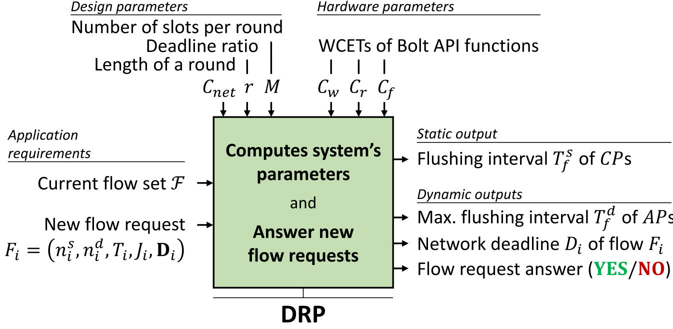


Figure 6. Inputs and outputs of DRP. *Hardware and design parameters are fixed at design time, while the application requirements may change at runtime. DRP statically computes the flushing interval of CPs; all other outputs are dynamically computed whenever the flow set $\mathcal{F}$ changes.*

approach to schedule all tasks of CPs. It consists of (*i*) setting the flushing interval T_f^s of all CPs to the same constant value, and (*ii*) letting the round interval T_{net} be a multiple of T_f^s. As motivated in Sec. V, we would like T_f^s not to constrain the achievable deadline, so we intend to set it as short as possible. To do so, recall the three tasks every CP needs to perform

- flushing Bolt before each communication round,
- participating in the communication during the rounds,
- writing all received messages into Bolt after the rounds.

Performing those tasks altogether takes $C_{CP} + C_{net}$ time units, where $C_{CP} = C_f + M * C_w$, and M denotes the number of time slots in one round. Hence, $C_{CP} + C_{net}$ is the smallest admissible round interval (otherwise CPs' task set is not schedulable). Thus we set for all CPs in the system,

$$T_f^s = C_{CP} + C_{net} \tag{4}$$

and we let the round interval be a multiple of T_f^s. In other words, for $k \in \mathbb{N}$, $k > 0$,

$$T_{net} = k * T_f^s \tag{5}$$

For a given C_{net}, a larger k entails less available bandwidth but also lower energy consumption. Blink dynamically adjusts k to match the bandwidth requirements and save energy.

B. Computing Network Deadlines & APs' Flushing Interval

Having fixed CPs' flushing interval, we now turn to the problem of dynamically computing the network deadline D_i of flow F_i and the flushing interval T_f^d of F_i's destination AP_d such that the end-to-end deadline $\mathbf{D}_i$ is met. To this end, we need to define expressions for the functions f and g (introduced in Sec. V), and deduce values for D_i and T_f^d such that equations (2) and (3) are satisfied.

Theorem 1. *For any flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i)$, and given the duration of communication rounds C_{net}, functions f and g are upper-bounded as follows*

$$f(T_f^s, D_i) \;\le\; T_i + D_i + \overline{J_i} + \delta_f^{const} \tag{6}$$
$$g(T_f^d) \;\le\; T_f^d(n_i^d) + \delta_g^{const} \tag{7}$$

where δ_f^{const} and δ_f^{const} are constant delays that depend on the WCETs of the Bolt API functions, on the maximum number of messages M that can be served by Blink in one round, and on the fixed flushing interval T_f^s of CPs,

$$\delta_f^{const} = C_w + C_f + T_f^s \tag{8}$$
$$\delta_g^{const} = M * C_w - (M-1) * C_r + C_f \tag{9}$$
$$\overline{J_i} = \left\lfloor (J_i + C_f - C_r)/T_f^s \right\rfloor \cdot T_f^s \tag{10}$$

Proof. Function f is the time between when a message is written into Bolt by the source AP_s and when the communication round in which the message is sent by Blink ends (*i.e.*, when the message is available at the destination CP_d). This is the sum of two delays: δ_{source}, the time until the message is available for communication at the source CP_s; and $\delta_{network}$, the time until the message is shipped over the network to CP_d.

Similarly, function g is the time between when a packet is available at the destination CP_d and the end of the `flush` operation that reads the message out of Bolt at the destination AP_d (*i.e.*, when the message can be processed by the destination application). We refer to this delay as δ_{dest}.

Hence, the expressions for functions f and g in (6) and (7) directly follow from the delays expression given in Lemmas 4, 5, and 6, which are presented and proven in the Appendix. ■

We use Theorem 1 to express conditions on D_i and T_f^d such that (2) and (3) are satisfied. In particular, it is sufficient that for any flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i) \in \mathcal{F}$

$$T_i + D_i + \overline{J_i} \ \leq \ r * \mathbf{D}_i - \delta_f^{const} \quad (11)$$

$$T_f^d(n_i^d) \ \leq \ (1-r) * \mathbf{D}_i - \delta_g^{const} \quad (12)$$

Furthermore, due to the limited bandwidth of low-power wireless networks, it makes sense to choose the network deadline D_i as large as possible, thus helping the schedulability of flows in the network. However, Blink does not support network deadlines larger than T_i [11] nor smaller than T_{net}^{min} (see Sec. V-B). Hence, for any flow F_i, it must hold that

$$T_{net}^{min} \ \leq \ D_i \ \leq \ T_i \quad (13)$$

Finally, to satisfy all contracts in the system, (11), (12) and (13) must hold for all flows $F_i \in \mathcal{F}$. Hence, the values for D_i and T_f^d computed dynamically at runtime must satisfy for any flow $F_i = (n_i^s, n_i^d, T_i, J_i, \mathbf{D}_i) \in \mathcal{F}$ and any $n \in \mathcal{N}$

$$D_i \ = \ \min\left(T_i \ , \ r * \mathbf{D}_i - \delta_f^{const} - T_i - \overline{J_i} \right) \quad (14)$$

$$T_f^d(n) \ \leq \ \min_{F_j \in \mathcal{F}, n = n_j^d} \left((1-r) * \mathbf{D}_j - \delta_g^{const} \right) \quad (15)$$

If using (14) leads to a violation of the constraint in (13) or if T_f^d results in a load that AP at node n cannot handle, DRP rejects the flow since the two contracts cannot be guaranteed.

C. Worst-case Buffer Analysis

Satisfying all contracts also entails preventing overflows of message buffers in the system. Specifically, as shown in Fig. 5,

- APs are responsible for ensuring that the incoming Bolt queues do not overflow, and
- CPs are responsible for ensuring that their local message buffers and the outgoing Bolt queues do not overflow.

To formulate the admission tests for APs and CPs, we first need the worst-case buffer sizes (*i.e.*, maximum number of messages in a buffer) induced by a given flow set $\mathcal{F}$. For ease of exposition, we make the following hypothesis.

Hypothesis 1. *For a given flow set $\mathcal{F}$, an AP (or CP) never writes more messages into Bolt than can be flushed by CP (or AP) in one* `flush` *operation in the time span between two* `flush`.

This hypothesis implies that the Bolt queues are always empty at the end of a `flush` operation. We prove at the end of this section that our admission tests effectively guarantee that Hypothesis 1 always holds.

Lemma 1. *Given a flow set $\mathcal{F}$, the buffer size of the outgoing Bolt queue of node $n \in \mathcal{N}$, $B_{Bolt,out}(n)$, is upper-bounded,*

$$B_{Bolt,out}(n) \ \leq \sum_{F_i \in \mathcal{F}, \, n = n_i^s} \left\lceil \frac{T_f^s + C_w + C_r + J_i}{T_i} \right\rceil \quad (16)$$

Proof. According to the **Source $\leftrightarrow$ Blink** contract, AP_s at node n does not write more than one message every T_i with jitter J_i into the outgoing Bolt queue. Based on Hypothesis 1, the buffer size is bounded by the number of messages that can be written by AP_s during the maximum time a message can stay inside the queue, which is $\Delta = T_f^s + C_w + C_r$ (see Fig. 9). The maximum number of messages that can be written by AP_s within any time interval Δ is $\lceil (\Delta + J_i)/T_i \rceil$ for each flow F_i sourced by n, which concludes the proof. ∎

The worst-case buffer size of a CP depends on (i) the maximum time a message can stay in CP's local memory awaiting to be served by Blink, and (ii) the number of messages that can be sent within one round to a node.

Lemma 2. *Given a flow set $\mathcal{F}$, the buffer size of CP's internal memory of node $n \in \mathcal{N}$, $B_{CP}(n)$, is upper-bounded,*

$$B_{CP}(n) \leq \sum_{\substack{F_i \in \mathcal{F}, \\ n = n_i^s}} 1 + \left\lceil \frac{D_i + \overline{J_i} + C_f}{T_i} \right\rceil + \sum_{\substack{F_i \in \mathcal{F}, \\ n = n_i^d}} 1 \quad (17)$$

Proof. On the source side, we make the conservative assumption that all messages read out during a `flush` occupy memory in CP_s from the beginning of the `flush`. Hence, the maximum waiting time in CP_s for a message until it is served by Blink is $\delta_{network} + C_f$ (see Lemma 5 in the Appendix). The number of messages in CP_s due to the source is upper-bounded by the maximum number of messages AP_s can write during this time interval, given by $\lceil \delta_{network} + C_f)/T_i \rceil$. Using Lemma 5, this leads to at most $1 + \lceil (D_i + \overline{J_i} + C_f)/T_i \rceil$ per outgoing flow.

One the destination side, during a round, CP_d may receive several messages, which it immediately writes into Bolt after the round. However, Blink expects one packet every T_i from each flow, which it serves within D_i. As $D_i \leq T_i$, Blink never schedules more than one packet per round for each flow. Thus, the maximum number of messages in CP_d due to the destination is 1 packet per incoming flow. ∎

Lemma 3. *Given a flow set $\mathcal{F}$, the buffer size of the incoming Bolt queue of node $n \in \mathcal{N}$, $B_{Bolt,in}(n)$, is upper-bounded,*

$$B_{Bolt,in}(n) \ \leq \sum_{\substack{F_i \in \mathcal{F}, \\ n = n_i^d}} \left\lceil \frac{T_f^d(n) + C_w + C_r + D_i}{T_i} \right\rceil \quad (18)$$

Proof. As specified in the **Source $\leftrightarrow$ Blink** contract, Blink delivers packets from any flow F_i before the network deadline D_i (see Sec. V). Therefore, Blink delivers at most one packet every T_i time units, with a jitter equal to D_i, which are written into Bolt immediately after the round.

Based on Hypothesis 1, the buffer constraint of the incoming Bolt queue is bounded by the number of packets that can be written by CP_d during the maximum elapsed time before a packet is read out by AP_d. As in the proof of Lemma 1, there are at most $\left\lceil (T_f^d(n) + C_w + C_r + D_i)/T_i \right\rceil$ such messages from each flow F_i that has node n as destination. ∎

D. Admission Tests

One can now combine the above results and formulate the admission tests for CPs and APs, which are the corner stone of DRP's registration mechanism described in Sec. V-C. We

further show that the computation complexity of the admission tests is not only small but *constant*, and hence supports the desired properties of adaptability (**P3**) and scalability (**P5**).

Let F_j be the flow for which a request has been issued, and $\mathcal{F}_{new} = \mathcal{F} \cup \{F_j\}$. The CP of node n is responsible for preventing overflows of its local memory (of size S_{CP}) and of the outgoing Bolt queue of node n (of size S_{Bolt}).

Theorem 2 (Admission Test of CP)**.** *If*

$$S_{Bolt} \geq \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^s}} \left\lceil \frac{T_f^s + C_w + C_r + J_i}{T_i} \right\rceil \quad \text{and}$$

$$S_{CP} \geq \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^s}} 1 + \left\lceil \frac{D_i + \overline{J_i} + C_f}{T_i} \right\rceil + \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^d}} 1$$

then the requested flow F_j can be safely admitted by CP.

Proof. Immediate from Lemmas 1 and 2. ■

The AP of node n is responsible for preventing overflows of the incoming Bolt queue of node n (of size S_{Bolt}) and for guaranteeing its share of the end-to-end deadline.

Theorem 3 (Admission Test of AP)**.** *If there exists $T_f^d(n)$ such that*

$$T_f^d(n) \leq \min_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^d}} \left((1 - r) * \mathbf{D}_i - \delta_g^{const} \right) \quad \text{and}$$

$$S_{Bolt} \geq \sum_{\substack{F_i \in \mathcal{F}_{new}, \\ n = n_i^d}} \left\lceil \frac{T_f^d(n) + C_w + C_r + D_i}{T_i} \right\rceil$$

then the requested flow F_j can be safely admitted by AP.

Proof. Immediate from Lemma 3 and equation (15). ■

Finally, we verify that Hypothesis 1 holds, showing the validity of our buffer analysis. From (1) we have $f_{max} = S_{Bolt}$. Thus, by performing the admission tests at runtime, it follows from Theorems 2 and 3 and Lemmas 1 and 3 that f_{max} is always bigger than the filling level of any Bolt queue, which entails Hypothesis 1 is true.

VII. Effect of Design Parameters on Performance

The previous section presented admission tests for AP and CP that ensure all contracts are satisfied after the admission of a new flow for given design parameters: the duration of a round in Blink C_{net} and the deadline ratio r. In this section, we analyze the influence of these parameters on the achievable performance of DRP (*i.e.*, responsiveness and bandwidth).

A. Responsiveness: Minimal Admissible End-to-end Deadline

Let us assume that the duration of communication rounds C_{net} is given. DRP handles messages between application interfaces (*i.e.*, the APs) and constrains the destination AP_d to flush Bolt (at least) every T_f^d. Naturally, there exists a lower bound on the admissible T_f^d; let us refer to this bound as $T_{f,\,min}^d$. Given these parameters, we are interested in the minimal admissible end-to-end deadline $\mathbf{D}_{min}$, or in other words, the maximal responsiveness of the protocol.

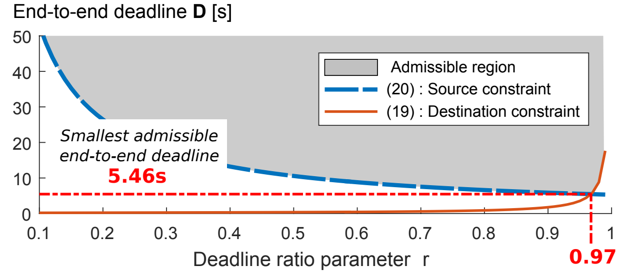


Figure 7. Finding the smallest admissible end-to-end deadline $\mathbf{D}_{min}$ for $C_{net} = 1\,\text{sec}$ and $T_{f,\,min}^d = 0.1\,\text{sec}$. (19) *and* (20) *each define a feasible region for* $(r, \mathbf{D})$ *tuples. The intersection defines the admissible region.*

From the previous remark on T_f^d and eq. (15) it follows

$$T_{f,\,min}^d \leq T_f^d \leq \left((1 - r) * \mathbf{D} - \delta_g^{const} \right)$$
$$\Rightarrow \quad \mathbf{D} \geq \frac{T_{f,\,min}^d + \delta_g^{const}}{(1 - r)} \tag{19}$$

From (11) we also have

$$T + D + \overline{J} \leq r * \mathbf{D} - \delta_f^{const}$$
$$\Rightarrow \quad \mathbf{D} \geq \frac{T + D + \overline{J} + \delta_f^{const}}{r}$$

We look for the minimal expression of the right-hand side term. (13) : $T_{net}^{min} \leq D_i \leq T_i$ yields $T_{min} = D_{min} = T_{net}^{min}$. Moreover, combining (4) and (5) entails $T_{net}^{min} = T_f^s = C_{net} + C_{CP}$. Hence T_{min} is fixed given C_{net}. Finally, in the best case, there is no (or small) jitter (*i.e.*, $\overline{J} = 0$), and we obtain

$$\Rightarrow \quad \mathbf{D} \geq \frac{2T_{min} + \delta_f^{const}}{r} \tag{20}$$

(19) and (20) define two lower bounds on the minimal admissible end-to-end deadline $\mathbf{D}_{min}$ induced by the contracts. Combining them, it follows that

$$\mathbf{D}_{min} = \min_r \left(\frac{T_{f,\,min}^d + \delta_g^{const}}{(1 - r)}, \frac{2T_{min} + \delta_f^{const}}{r} \right)$$

The minimal value $\mathbf{D}_{min}$ is reached for $r_{opt} = (2T_{min} + \delta_f^{const})/(T_{f,\,min}^d + \delta_g^{const} + 2T_{min} + \delta_f^{const})$ and it yields

$$\mathbf{D}_{min} = T_{f,\,min}^d + 2T_{min} + \delta_f^{const} + \delta_g^{const} \tag{21}$$

Using the parameters in Table II from real-world prototypes, if $C_{net} = 1\,\text{sec}$ and $T_{f,\,min}^d = 0.1\,\text{sec}$, the minimal end-to-end deadline that can be supported is $\mathbf{D}_{min} = 5.46\,\text{sec}$, with $r = r_{opt} = 0.97$, and the minimum message interval $T = T_{min} = 1.074\,\text{sec}$. This case is illustrated in Fig. 7.

B. Bandwidth: Maximal Duration of Communication Rounds

Conversely, let us now assume that the minimal end-to-end deadline to be supported is given by $\mathbf{D}$, and consider the same assumption on T_f^d. The maximal bandwidth achievable by Blink is M/T_{net}^{min} packet/ sec. The round length C_{net} is a linear function of the number of packets per round M (*i.e.*, a constant time per packet plus some overhead), and

(5) : $T_{net}^{min} = C_{net} + C_{CP}$. Hence, the maximal bandwidth actually grows with C_{net}. Thus, we now investigate the maximal admissible duration of communication rounds C_{net} that yields the maximum available network bandwidth.

From (11) we have $T + D + \overline{J} \leq r * \mathbf{D} - \delta_f^{const}$, and, as previously, $T_{net}^{min} = C_{net} + C_{CP} \leq D \leq T$. We get

$$C_{net} \leq \frac{1}{2}(r * \mathbf{D} - \delta_f^{const}) - C_{CP} \tag{22}$$

From (19), given $\mathbf{D}$ and $T_{f,min}^d$, the maximal admissible value for r is $r_{max} = 1 - (T_{f,min}^d + \delta_g^{const})/\mathbf{D}$, and finally

$$\Rightarrow C_{net} \leq \frac{1}{2}\left(\mathbf{D} - \delta_f^{const} - \delta_g^{const} - T_{f,min}^d\right) - C_{CP} \tag{23}$$

Using the parameters from Table II, if we need to satisfy end-to-end deadlines of $\mathbf{D} = 10\,\text{sec}$ and $T_{f,min}^d = 3\,\text{sec}$, the maximal round length that can be supported is $C_{net} = 2.82\,\text{sec}$, with $r = r_{max} = 0.69$, and the minimum message interval $T = C_{net} + C_{CP} = 2.89\,\text{sec}$. That upper-bound also yields the maximal achievable network bandwidth.

C. Effect of Deadline Ratio on System Performance

We presented in Sec. VII-A that given C_{net} and $T_{f,min}^d$, there is an optimal value for r that minimizes the admissible end-to-end deadline $\mathbf{D}$. If one tolerates "larger" deadlines, r can be increased to allow for a bigger round length C_{net} (see (22)), which increases the maximal network bandwidth.

However, (15) yields $T_f^d \leq (1 - r) * \mathbf{D} - \delta_g^{const}$. Hence, the bigger r is the smaller T_f^d must be, which may results in more flows rejected by the destination application. On the contrary, if r is set to its minimal value $r_{min} = (2 * T_{min} + \delta_g^{const}) \,/\, \mathbf{D}$ (obtained from eq. (20)), it yields $T = D = T_{min} = 1.074\,\text{sec}$ and $\overline{J} = 0\,\text{sec}$. In other words, the maximal admissible jitter (obtained from (10)) is $J < T_f^s + C_r - C_f \approx 0.390\,\text{sec}$.

How to set the parameters for DRP depends on the application. For instance, if one consider an acoustic sensing scenario, responsiveness is usually quite critical, and the sensors (*i.e.*, the APs) should spend most of their time on sensing, not being busy with flushing Bolt. Thus, we want to support a rather small $\mathbf{D}_{min}$ while having a strong constraint on $T_{f,min}^d$. This will come at the cost of a "small" network bandwidth.

VIII. Simulation of DRP

DRP builds upon Bolt and Blink. As pictured in Fig. 2, the underlying hardware and software of the dual-processor platform containing AP, CP, and Bolt has been designed, produced, and extensively used in sensor network prototypes. Moreover, Bolt's real-time and power properties have been formally verified [16]. Similar statements hold for Blink. The real-time guarantees it provides have been formally verified, the protocol has been implemented in physical networks and intensively tested [11]. Moreover, Blink is a real-time layer built on top of the basic communication primitive Glossy [19] and the Low-power Wireless Bus (LWB) [20]. Glossy is based on the disruptive concept of network-wide synchronous transmissions. All the above implementations have been deployed on many testbeds, with up to more than one hundred nodes, widely varying node densities and network diameters, and in all cases achieved more than 99% data yield [11], [19], [20]; As demonstrated in [19], it is even possible to achieve end-to-end packet reliabilities as high as 99.9999 %.

Table II
SIMULATION PARAMETERS

	Parameter	Symbol	Value
Bolt	WCET of `write`	C_w	116 µs
	WCET of `read`	C_r	112 µs
	WCET of `flush`	C_f	684 ms
Blink	Round length	C_{net}	1 sec
	Packet size	.	32 Bytes
	Max number of slots in one round	M	46
DRP	Number of nodes	.	20
	Deadline ratio	r	0.5
	Flushing interval of CP	T_f^s	1.074 sec

We evaluate the run-time behavior of DRP based on values and parameters from these physical implementations. The simulation framework we use for the evaluation tracks the latency of each individual packet through the cyber-physical system including all APs, CPs, Bolt and the wireless communication network. This setting enables a practical evaluation of DRP. Physically, the simulation runs on Matlab scripts.

Objective. In the last section, we have derived optimal performances that DRP can achieve, according to our protocol analysis. However, in order to provide hard guarantees on end-to-end deadlines and buffer sizes, the analysis is based on worst-case scenarios, which can be pessimistic. To support the relevance of this "optimal" performance, we now investigate the tightness of the analysis.

Use case. Let us assume that our network models an acoustic wireless sensor network monitoring, for example, high alpine regions. When a rock cracks, sensor data must be collected and forwarded to a sink node for processing. As the local memory is limited, the data must be sent as fast as possible to the sink, but it should not be at the cost of data reliability, otherwise hindering the data processing. This realistically motivates the use of DRP in such a scenario.

Procedure. We use parameters from the physical Bolt platform presented in Fig. 2 and from a running implementation of Blink. The complete list of parameters is provided in Table II.

One node acts as the sink (say node 1) and communicates with all other nodes in the network. As described in Sec. V, DRP is initialized with a basic set of flows $\mathcal{F}_{init}$, which is necessary in order to register subsequent flows

$$\mathcal{F}_{init} = \begin{Bmatrix} (1\,,\ \text{n}\,,\ T = 10\,\text{sec}\ ,\ J = 0\,\text{sec}\ ,\ \mathbf{D} = 30\,\text{sec}) \\ (\text{n}\,,\ 1\,,\ T = 10\,\text{sec}\ ,\ J = 0\,\text{sec}\ ,\ \mathbf{D} = 30\,\text{sec}) \end{Bmatrix}$$

for $n \in (2..20)$. In practice, such flows can also be used to send low-priority data (*e.g.*, status data) regularly to the sink.

Blink computes schedules assuming the first packet of each flow is available for communication at $t = 0\,\text{sec}$. The actual epoch at which the APs write the first packet of each flow is randomized between 0 sec and the flow's minimal message interval T; subsequent packets are sent with period T.

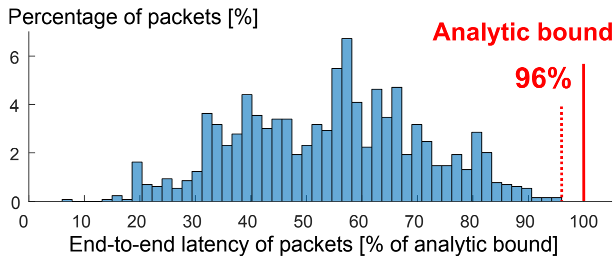


Figure 8. Distribution of end-to-end packet latency in percentage of the analytical worst-case bound. *A few packets experience a latency close to the worst-case bound, showing that our analysis is both safe and tight.*

Upon occurrence of an event, four nodes (say nodes 2 to 5) concurrently detect it and emit a request for a new flow to the sink node. To transfer the event data as fast as possible, the message interval is chosen as small as possible (*i.e.*, equal to T_f^s, the flushing interval of CP – Refer to (4), (5) and (13)),

$$\mathcal{F}_{new} = \left\{ (\text{n}\,,\ 1\,,\ T = 1.074\,\text{sec}\ \ ,\ J = 0\,\text{sec}\ \ ,\ \mathbf{D} = 10\,\text{sec}) \right\}$$

for $n \in (2..5)$. We record the actual end-to-end latency of all packets during one minute, in which about 220 packets are exchanged in the network.

Results. Fig. 8 shows the distribution of end-to-end packet latency in percentage of the worst-case bound predicted by our analysis (see Th. 1). We see that a few packets indeed experience an end-to-end latency up to 96 % of the analytic worst-case bound. Our simulations also indicate that, in many cases, the worst-case buffer sizes of CP and Bolt are reached. Overall, these results support our analysis of DRP, showing that our worst-case bounds are safe and tight, and the optimal performance discussed in the previous section is sound.

IX. Conclusions

In this paper, we tackle the problem of providing end-to-end real-time guarantees between interfaces of distributed applications in a wireless cyber-physical system. Unlike prior work, we look at the complete chain of concurrent tasks and shared resources involved in the message transfer between applications. Based on the decoupling of wireless real-time communication from application tasks on individual devices, we design a distributed protocol that preserves the decoupling to the largest extent possible, while guaranteeing that all messages received by a destination application meet their end-to-end deadlines. We analyze our design and derive worst-case bounds on buffer sizes and end-to-end delay along the chain. Simulations validate that our bounds are both safe and tight.

Acknowledgments. This work was supported by Nano-Tera.ch with Swiss Confederation financing, and by the DFG within the Cluster of Excellence "Center for Advancing Electronics Dresden" (CFAED) and Priority Program 1914.

References

[1] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proc. of ACM/IEEE DAC*, 2010.
[2] J. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *Computer*, vol. 38, no. 11, 2005.
[3] 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 *Proc. of IEEE RTAS*, 2002.
[4] J. A. Stankovic, T. F. Abdelzaher, C. Lu, L. Sha, and J. C. Hou, "Real-time communication and coordination in embedded sensor networks," *Proc. IEEE*, vol. 91, no. 7, 2003.
[5] T. He, J. Stankovic, C. Lu, and T. Abdelzaher, "SPEED: A stateless protocol for real-time communication in sensor networks," in *Proc. of IEEE ICDCS*, 2003.
[6] WirelessHART, "Wirelesshart," 2007. [Online]. Available: http://en.hartcomm.org/main_article/wirelesshart.html
[7] ISA100, "Wireless compliance institute," 2009. [Online]. Available: http://www.isa100wci.org/
[8] T. Watteyne, V. Handziski, X. Vilajosana, S. Duquennoy, O. Hahm, E. Baccelli, and A. Wolisz, "Industrial wireless IP-based cyber-physical systems," *Proc. IEEE*, vol. 104, no. 5, May 2016.
[9] A. Saifullah, Y. Xu, C. Lu, and Y. Chen, "Real-time scheduling for wirelesshart networks," in *Proc. of IEEE RTSS*, 2010.
[10] ——, "End-to-end communication delay analysis in industrial wireless networks," *IEEE Trans. Computers*, vol. 64, no. 5, 2015.
[11] M. Zimmerling, L. Mottola, P. Kumar, F. Ferrari, and L. Thiele, "Adaptive real-time communication for wireless cyber-physical systems," ETH Zurich, Tech. Rep., 2016.
[12] H. Kopetz, "The time-triggered model of computation," in *Proc. of IEEE RTSS*, 1998.
[13] N. Baccour *et al.*, "Radio link quality estimation in wireless sensor networks: A survey," *ACM Trans. Sen. Netw.*, vol. 8, no. 4, 2012.
[14] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srinivastava, "Power management in energy harvesting sensor networks," *ACM Trans. Embed. Comput. Sys.*, vol. 6, no. 4, 2007.
[15] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Virtual synchrony guarantees for cyber-physical systems," in *Proc. of IEEE SRDS*, 2013.
[16] F. Sutton, M. Zimmerling, R. Da Forno, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel, and L. Thiele, "Bolt: A stateful processor interconnect," in *Proc. of ACM SenSys*, 2015.
[17] R. L. Cruz, "A calculus for network delay. i. network elements in isolation," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, 1991.
[18] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. of IEEE ISCAS*, 2000.
[19] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with Glossy," in *Proc. of ACM/IEEE IPSN*, 2011.
[20] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-power wireless bus," in *Proc. of ACM SenSys*, 2012.
[21] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource reservation protocol," *IEEE Network*, vol. 7, no. 5, 1993.
[22] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and Microprogramming*, vol. 40, 1994.
[23] M. González Harbour, J. G. García, J. P. Gutiérrez, and J. D. Moyano, "Mast: Modeling and analysis suite for real time applications," in *Proc. of ECRTS*, 2001.
[24] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis-the symta/s approach," in *IEE Proc. Computers and Digital Techniques*, vol. 152, no. 2, 2005, pp. 148–166.
[25] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System architecture evaluation using modular performance analysis: a case study," *Int. Journal on Software Tools for Technology Transfer*, vol. 8, no. 6, 2006.
[26] J. Åkerberg, F. Reichenbach, M. Gidlund, and M. Björkman, "Measurements on an industrial wireless HART network supporting PROFIsafe: A case study," in *Proc. of IEEE ETFA*, 2011.
[27] T. O'donovan *et al.*, "The ginseng system for wireless monitoring and control: Design and deployment experiences," *ACM Trans. on Sensor Networks*, vol. 10, no. 1, 2013.
[28] M. Zimmerling, F. Ferrari, L. Mottola, and L. Thiele, "On modeling low-power wireless protocols based on synchronous packet transmissions," in *Proc. of IEEE MASCOTS*, 2013.
[29] B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M. I. Jordan, and S. S. Sastry, "Kalman filtering with intermittent observations," *IEEE Trans. Autom. Control*, vol. 49, no. 9, 2004.
[30] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

APPENDIX

Worst-case analysis of the source delay.

Definition 1 (Source delay – δ_{source})**.** *The source delay is the elapsed time from a packet being written in Bolt by the source AP_s until the end of the* `flush` *operation where it is read out of Bolt by the source CP_s. For a flow F_i, it is denoted by $\delta_{source,\,i}$.*

Lemma 4. *For any flow F_i, the source delay is upper-bounded by*

$$\delta_{source,\,i} \;\leq\; C_w + T_f^s + C_f \tag{24}$$

Proof. Let us recall that a `flush` is a sequence of `read` operations. When the Bolt queue is found empty, the `flush` is terminated and no other `read` is performed until the next `flush` (refer to V-A for details). Therefore, if the Bolt queue is empty and a `write` operation terminates just after a `flush` is triggered, that `flush` immediately terminates and the packet is delayed until to the end of the next `flush`. Possible jitter on the `write` operation pattern does not have any influence on the worst-case for $\delta_{source,\,i}$. This worst-case scenario for the source delay is illustrated on Fig. 9. ∎

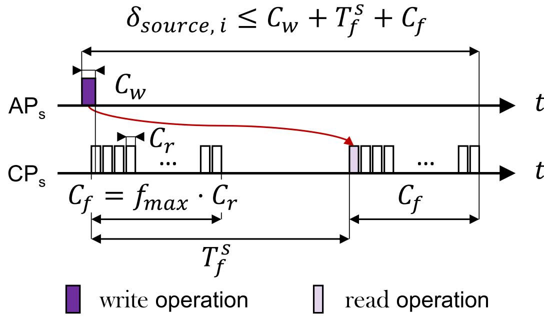


Figure 9. Worst-case analysis of the source delay. *A packet is written as early as possible such that it misses a* `flush` *and must wait until the next one.*

Worst-case analysis of the network delay.

Definition 2 (Network delay – $\delta_{network}$)**.** *The network delay is the elapsed time from a packet being available for communication at the source CP_s until the end of the communication round where it is served by the wireless protocol (i.e., when it is available at the destination CP_d). For a flow F_i, it is denoted by $\delta_{network,\,i}$.*

Lemma 5. *For any flow F_i, the network delay is upper-bounded by*

$$\delta_{network,\,i} \;\leq\; T_i + D_i + \left\lceil \frac{J_i + C_f - C_r}{T_f^s} \right\rceil \cdot T_f^s \tag{25}$$

Proof. As presented in V-B, Blink guarantees that every packet matching the *expected arrival* is served in a round that terminates before the network deadline D_i. Hence, the *delay of an expected packet* is no more than D_i.

However, the actual arrival of packets at the source CP_s does not match the expected arrival in general, but results from `flush` operations, which occur every T_f^s time unit. Hence, a packet may arrive *earlier* than the next expected packet. That mismatch between the two arrival times (actual and expected) adds up with the delay of the expected packet (i.e., D_i).

Let us consider first that the flow F_i has no jitter (*i.e.*, $J_i = 0$) and let m be the mismatch between actual and expected arrival time at CP_s. m cannot be larger than the flow's minimum message interval T_i

$$m \;\leq\; T_i$$

The intuition is given with Fig. 10. See the caption for details.

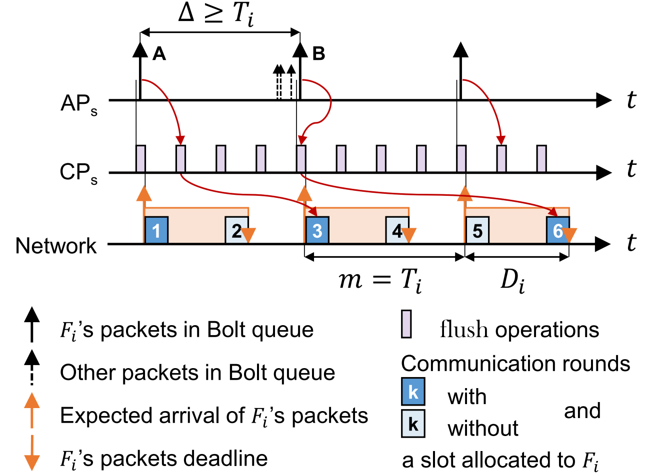


Figure 10. Worst-case analysis of the network delay without jitter. *Because of the Bolt queue being empty, packet **A** misses the first flush operation (similarly as in Fig. 9), hence the slot allocated to F_i in round **1** is wasted. Due to packets released from other flows in the meantime, packet **B** is flushed directly in the operation preceding round **3**, in which flow F_i is allocated a new slot. However, as packet **A** is still in queue, packet **B** is not served right away but is delayed until the next allocated slot* (i.e., *in round **6**). This creates a mismatch of T_i for packet **B**. Furthermore, the mismatch cannot get bigger; assume **B** were to be available at CP_s earlier* (i.e., *one flush operation before, at least), because the time interval between **A** and **B** must be at least T_i, **A** would arrive earlier as well. Hence, **A** would not miss the slot in round **1**, **B** would be served in round **3**, and thus it would yield a smaller mismatch for packet **B**.*

Now, if flow F_i has also jitter J_i, this may entail a bigger mismatch. Actual "arrival" of packets (*i.e.*, the epoch when a packet is available for communication at the source CP_s, according to the definition of the network delay) can occur only every T_f^s (*i.e.*, at the end of one `flush` operation). Therefore, one can see that jitter may induce an extra delay, or mismatch, of roughly $\left\lceil J_i / T_f^s \right\rceil \cdot T_f^s$. A more precise analysis of the flushing dynamics (see Fig. 11 for details) entails that, overall, the worst-case mismatch m is bounded by

$$m \;\leq\; T_i + \left\lceil \frac{J_i + C_f - C_r}{T_f^s} \right\rceil \cdot T_f^s \tag{26}$$

and finally,

$$\delta_{network,\,i} \;\leq\; D_i + m$$

$$\delta_{network,\,i} \;\leq\; T_i + D_i + \left\lceil \frac{J_i + C_f - C_r}{T_f^s} \right\rceil \cdot T_f^s \qquad ∎$$

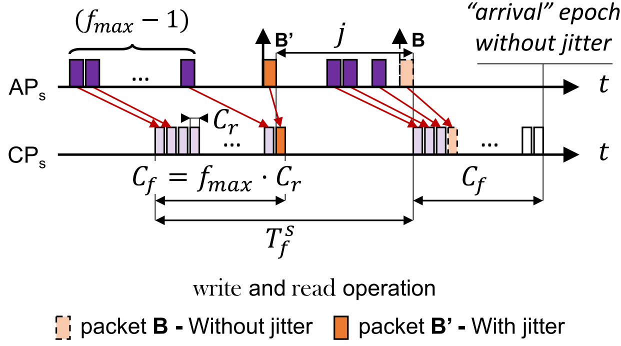


Figure 11. Influence of jitter on the network delay. *Let us have a closer look at packet **B** from the previous figure, positioned as early as possible* (i.e., *if it were earlier, so would be **A**, which would then not miss its slot in round **1**). Due to jitter, **B** is released earlier, say by a amount j. This can yield packet **B'** (**B** with jitter) to be read out in a previous* `flush` *operation. In the worst-case, packet **B'** is read out one operation earlier as soon as j is bigger than $T_f^s - C_f + C_r$, which increases the mismatch m by T_f^s. Similarly, m increases by $k \cdot T_f^s$ when j reached $k \cdot T_f^s - C_f + C_r$, which yields $k = \left\lfloor \frac{j + C_f - C_r}{T_f^s} \right\rfloor$ and concludes to equation* (26).

Worst-case analysis of the destination delay.

Definition 3 (Destination delay – δ_{dest})**.** *The destination delay is the elapsed time from a packet being available at the destination CP_d until the end of the* `flush` *operation where it is read out of Bolt by the destination AP_d (i.e., when it is available for the application). For a flow F_i, it is denoted by $\delta_{dest,\,i}$.*

Lemma 6. *For any flow F_i, the destination delay is upper-bounded by*

$$\delta_{dest,\,i} \;\leq\; M * C_w - (M-1) * C_r + T_f^d + C_f \qquad (27)$$

Proof. The situation is similar as for the source delay, except that CP_d writes every T_{net} time unit (*i.e.*, after each round) all the packets it received during the last round, which can be as many as M packets. The maximal delay for a packet occurs when it is written too late to be read out during an ongoing `flush` and must wait for the next one.

A careful analysis of the Bolt dynamics shows that the `read` operation is slightly shorter than `write` [16] (i.e., $C_r < C_w$, see Table II). Hence, the more packets are written at once by CP_d, the later a `flush` can start and still miss the last written packet. The worst-case is illustrated on Fig. 12. ■

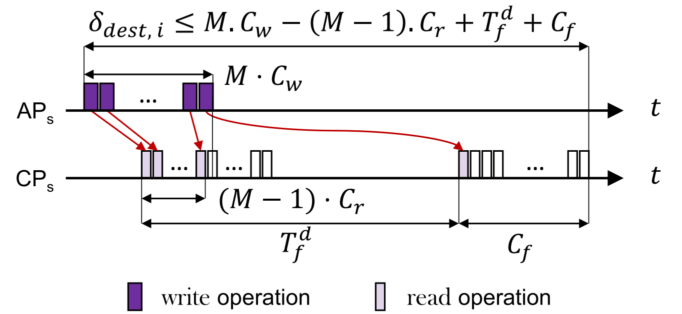


Figure 12. Worst-case analysis of the destination delay. *A packet is written as early as possible such that it misses a* `flush` *and must wait until the next one.*