Distributed Dynamic Packet Scheduling for Handling Disturbances in Real-Time Wireless Networks

Tianyu Zhang[†§*], Tao Gong[‡], Chuancai Gu[†], Huayi Ji[‡], Song Han[‡], Qingxu Deng[§], Xiaobo Sharon Hu[†]
[†]Dept. of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 46556
[†]Email: {tzhang4,cgu,shu}@nd.edu
[‡]Dept. of Computer Science and Engineering, University of Connecticut, Storrs, CT, 06269
[‡]Email: {tao.gong,huayi.ji,song.han}@uconn.edu
[§]School of Computer Science and Engineering, Northeastern University, Shenyang, China
[§]Email: dengqx@mail.neu.edu.cn

***Abstract*—Real-time wireless networks (RTWNs) are fundamental to many Internet-of-Things (IoT) applications. Packet scheduling in an RTWN plays a critical role for achieving desired performance but is a challenging problem especially when the RTWN is large and subject to unexpected disturbances from the environment. Few solutions exist to tackle this challenge but they suffer serious limitations. This paper introduces a novel distributed dynamic packet scheduling framework, D^2-PaS. D^2-PaS aims to minimize the number of dropped packets while ensuring that all critical events due to disturbances are handled by their deadlines. D^2-PaS builds on a number of observations that help reduce the scheduling overhead, and thus is efficient and scalable. Besides extensive simulation, D^2-PaS has been implemented on an RTWN testbed to validate its applicability on real hardware. Both testbed measurements and simulation results confirm the effectiveness of D^2-PaS. Compared to the best known work, D^2-PaS reduces packet drop rates by 65% and 90% on average and in the best case, respectively, and also achieves 100% success for all the randomly generated task sets.**

I. Introduction

Real-time wireless networks (RTWNs) are fundamental to many Internet-of-Things (IoT) applications in a broad range of fields such as military, civil infrastructure and industrial automation [1]–[3]. An RTWN typically consists of spatially distributed sensors, actuators, relay nodes and a gateway, where the components communicate over wireless network media to collectively accomplish the execution of certain real-time tasks running on the RTWN. The Quality of Service (QoS) offered by an RTWN is often measured by how well it satisfies the end-to-end (from sensors via gateway to actuators) deadlines of these real-time tasks. Packet scheduling at the data link layer of RTWNs plays a critical role in achieving the desired QoS, and it becomes more challenging when the network size grows. The fact that most RTWNs must deal with external disturbances in the environment being monitored and controlled (e.g., detection of an intruder, sudden pressure change) further aggravate the problem.

Static packet scheduling for RTWNs has been well studied in the literature (e.g. [4]–[6]). Static approaches support deterministic real-time communication, but are unsuitable for handling dynamic changes. Dynamic changes can be classified into network changes (due to node and link failure, etc.) and workload changes caused by external disturbances. A number of centralized dynamic scheduling approaches for handling network changes have been proposed (e.g., [7]–[10]). Studies on addressing external disturbances (or just disturbances), the focus of this paper, are relatively few. The approaches in [11] do not support systems containing multiple tasks ending at different actuators, while [12] dose not consider cases when not all tasks can meet their deadlines.

Extensive research efforts have been devoted on modeling and responding to disturbances in real-time systems. [13] proposes sampling rate adjustments to improve the performance of event-triggered control systems. Rate-adaptive and rhythmic task models which allow tasks to change periods and relative deadlines are introduced in [14] and [15], respectively. Schedulability analysis for Rate Monotonic (RM) and Earliest Deadline First (EDF) policies are also presented in [15] and [14]. Though these existing task models can be adopted by RTWNs, their associated scheduling approaches and schedulability analyses cannot be readily applied since they do not consider end-to-end packet delivery required in RTWNs.

To the best of our knowledge, OLS in [16] is the state-of-the-art dynamic approach to handle disturbances in RTWNs. OLS is a centralized framework built on the rhythmic task model and has been shown to be quite effective for relatively small RTWNs. However, for larger RTWNs (with, say over 30 nodes), OLS can incur significant computational overhead and may even not be able to find an feasible schedule. Moreover, OLS may drop more periodic packets than necessary due to the limited packet lengths in RTWNs. All these drawbacks lead to degraded system performance when disturbances occur.

This paper introduces a novel distributed dynamic packet scheduling framework, referred to as D^2-PaS, to handle disturbances in RTWNs. As a distributed approach, D^2-PaS lets each node construct its own schedule so as to significantly reduce the amount of schedule related information to be broadcast by the gateway when disturbances are detected. As a dynamic approach, D^2-PaS determines on-line at the gateway which packets can be dropped in response to disturbances and broadcast minimal amount of information to the rest of the nodes in the RTWN. To ensure that such a dynamic distributed approach is effective and efficient, we design a lightweight (in terms of memory usage and computing capability) scheduler to be used at each node. Furthermore, we develop a low-complexity algorithm to be executed by the gateway to determine a short interval in which some packets need to be dropped, and minimize the number of such dropped packets.

We have implemented D^2-PaS on a real-time wireless network testbed to validate the applicability of the proposed

*The first two authors have equal contribution to this work.

978-1-5090-5269-1/17 $31.00 © 2017 IEEE

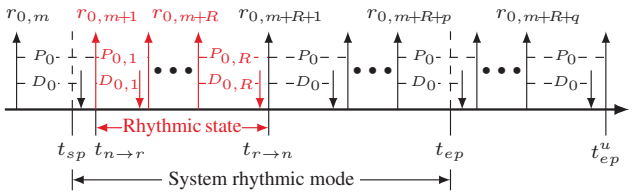

Fig. 1. Timing parameters for τ_0 and for the system in the rhythmic mode.

framework. We also evaluated the performance of D^2-PaS through both extensive simulation studies and testbed based experiments. Compared to the state-of-the-art OLS approach in [16], D^2-PaS on average reduces the number of dropped packets by 65% and in the best case by 90%. D^2-PaS achieves 100% success in meeting the deadlines of the rhythmic packets while OLS only achieves 90% success on average and 62% success in the worst case. In terms of computation overhead, D^2-PaS can be 2000X to 10,000X faster than OLS.

II. System Model

We adopt the system architecture of a typical RTWN, in which multiple sensor nodes (S-nodes) and actuator nodes (A-nodes) are wirelessly connected to a single gateway (G-node) directly or through relay nodes (R-nodes). We assume that both S-nodes and A-nodes have routing capability and they each are equipped with a single omni-directional antenna to operate on a single channel in half-duplex mode. The network is described by a directed graph $G = (V, E)$, where the node set $V = \{\{V_0, V_1, \dots\}, V_g\}$. V_g represents the G-node and the rest are referred to the *device nodes* (all other types of nodes). A direct link $(V_i, V_j) \in E$ represents a reliable link from node V_i to node V_j. (We assume that all links are reliable in this paper. Handling unreliable links can leverage existing methods and is left for future work.) V_g connects all device nodes to control logic. It also contains a network manager which is responsible for network configuration and resource allocation.

For simplicity, we assume that the system runs a fixed set of tasks $\mathcal{T} = \{\tau_0, \tau_1, \dots, \tau_n, \tau_{n+1}\}$. τ_i $(0 \le i \le n)$ is a unicast task following a pre-determined single routing path with H_i hops. It periodically takes a sample at S-node V_i, generates a packet to forward the sampled data to the G-node, and delivers the generated control message to the respective A-node. τ_{n+1} is a broadcast task. It runs on the G-node and is responsible for disseminating the updated schedule information in the network by following a pre-determined broadcast graph [4].

To capture abrupt increases in network resource demands when disturbances are detected, we adopt the rhythmic task model [15] since it is shown to be effective for handling disturbances in resilient event-triggered control systems [16].[1] Specifically, each unicast task τ_i has two states: *nominal state* and *rhythmic state*. In the nominal state, τ_i follows the nominal period P_i and nominal relative deadline $D_i \le P_i$, which are all constants. When a disturbance occurs, τ_i enters the rhythmic state in which its period and relative deadline are first reduced, in order to properly respond to the disturbance, and then gradually return to their nominal values by following some monotonically non-decreasing function. We use vectors $\overrightarrow{P_i} = [P_{i,x}, x = 1, \dots, R]^T$ and $\overrightarrow{D_i} = [D_{i,x}, x = 1, \dots, R]^T$ to represent the periods and relative deadlines of τ_i when it is in the rhythmic state. As soon as τ_i enters the rhythmic state, its period and relative deadline adopt sequentially the values specified by $\overrightarrow{P_i}$ and $\overrightarrow{D_i}$, respectively. τ_i returns to the nominal state when it starts using the period and deadline again.

[1]Note that our D^2-PaS framework is not limited to the rhythmic task model and is applicable to any task model that provides the workload changing patterns for handling disturbances.

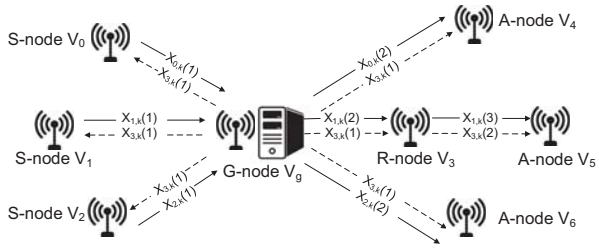

Fig. 2. An example RTWN with 3 unicast tasks and 1 broadcast task running on 8 nodes (1 gateway node, 3 sensors, 3 actuators and 1 relay node).

We further assume that at any time during system operation, there is at most one unicast task in the rhythmic state. To simplify the notation, we refer to any task currently in the rhythmic state as *rhythmic task* and denote it as τ_0 while task τ_i $(1 \le i \le n)$ is a *periodic task* which is currently not in the rhythmic state. When a task enters the rhythmic state, we also say that the system switches to the *rhythmic mode*. The system returns to the *nominal mode* when the disturbance has been completely handled, typically some time after the task has returned to the nominal state. Since disturbances may cause catastrophe to the system, the rhythmic task has a hard deadline when the system is in the rhythmic mode while periodic tasks can tolerate occasional deadline misses.

Each task τ_i consists of an infinite sequence of instances. The k-th instance of τ_i, referred to as packet $\chi_{i,k}$, is associated with release time $r_{i,k}$, deadline $d_{i,k}$ and finish time $f_{i,k}$. Without loss of generality, we assume that τ_0 enters the rhythmic state at $r_{0,m+1}$ (denoted as $t_{n\to r}$) and returns to the nominal state at $r_{0,m+R+1}$ (denoted as $t_{r\to n}$). Thus, τ_0 stays in its rhythmic state during $[t_{n\to r}, t_{r\to n})$, where $t_{n\to r}$ and $t_{r\to n}$ satisfy $t_{r\to n} = t_{n\to r} + \sum_{x=1}^{R} P_{0,x}$. Any packet of τ_0 released in the system rhythmic mode is referred to as *rhythmic packet* while the packets of task τ_i $(1 \le i \le n)$ are *periodic packets*. The delivery of packet $\chi_{i,k}$ at the h-th hop is referred to as a transmission denoted as $\chi_{i,k}(h)$ $(1 \le h \le H_i)$. Following industrial practice, we assume a Time Division Multiple Access (TDMA) data link layer in our model. Every node follows a given schedule to transmit or receive packets, and each transmission $\chi_{i,k}(h)$ must be completed in a single time slot. Fig. 1 depicts the rhythmic task model, and Table I summarizes the notation frequently used in this paper.

III. Overall Framework

In this section, we first give a motivational example to show the drawbacks of static and centralized scheduling approaches in handling rhythmic tasks. We then describe our proposed distributed dynamic packet scheduling framework, D^2-PaS.

A. Motivation

Consider an example RTWN shown in Fig. 2. It consists of 4 tasks (τ_0, τ_1, τ_2 and τ_3) running on 8 nodes ($V_0, \dots, V_6$

TABLE I. SUMMARY OF IMPORTANT NOTATION

Parameter	Definition	Parameter	Definition
V_i, $i = 0, 1, \ldots$	Device nodes including sensors, actuators and relay nodes	$\chi_{i,k}$	The k-th released packet of task τ_i
V_g	Gateway	$\chi_{i,k}(h)$	The h-th transmission of packet $\chi_{i,k}$
τ_0 and τ_{n+1}	Rhythmic task and broadcast task	$r_{i,k}$, $d_{i,k}$, $f_{i,k}$	Release time, deadline and finish time of $\chi_{i,k}$
τ_i, $1 \le i \le n$	Periodic task	SS_j	Schedule segment of node V_j
H_i	Number of hops of τ_i	R	The number of elements in $\overrightarrow{P_0}$ ($\overrightarrow{D_0}$)
P_i, D_i $0 \le i \le n$	Period and relative deadline of rhythmic or periodic task τ_i	$\Gamma(t_{ep})$	Set of end point candidates
		$t_{n\to r}$, $t_{r\to n}$	Slot when τ_0 leaves its nominal state and its rhythmic state, respectively
$\overrightarrow{P_0}$, $\overrightarrow{D_0}$	Period and relative deadline vectors of τ_0		
t_{sp}, t_{ep}, t^c_{ep}, t^u_{ep}	Start point, end point, end point candidate and end point upper bound	$\Psi(t)$	Set of active packets within $[t_{sp}, t)$
		t_{np}	No-carry-over-packet (NCoP) point
Δ^d	The maximum allowed number of dropped packets	$\rho[t_{sp}, t)$	Set of dropped periodic packets within $[t_{st}, t)$

TABLE II. TASK PARAMETERS FOR THE MOTIVATIONAL EXAMPLE

Task	Routing Path	P_i	D_i	$\overrightarrow{P_i}$	$\overrightarrow{D_i}$
τ_0	$V_0 \to V_g \to V_4$	10	9	$[4, 6]^T$	$[3, 5]^T$
τ_1	$V_2 \to V_g \to V_6$	10	8	N/A	N/A
τ_2	$V_1 \to V_g \to V_3 \to V_5$	10	7	N/A	N/A
τ_3	$V_g \to *$ [2]	10	10	N/A	N/A

and V_g) where V_0, V_1 and V_2 are S-nodes, V_4, V_5 and V_6 are A-nodes, V_3 is an R-node and V_g is the G-node. Task τ_0 is the rhythmic task, task τ_1 and τ_2 are periodic tasks and task τ_3 is the broadcast task. Their routing paths, periods and relative deadlines, as well as $\overrightarrow{P_0}$ and $\overrightarrow{D_0}$ for τ_0 are given in Table II.

The periodic tasks and rhythmic task are synchronous and all of their first packets $\chi_{0,1}$, $\chi_{1,1}$ and $\chi_{2,1}$ are released at time slot 0. When the system starts, it uses the predetermined static schedule which is of length 10 repeatedly as shown in Fig. 3(a). Here, (i, h) within a slot indicates that this slot is allocated to the h-th hop of task τ_i. Suppose that at time slot 10, τ_0 enters the rhythmic state (i.e., $t_{n\to r} = 10$). Based on $\overrightarrow{P_0}$ and $\overrightarrow{D_0}$ in Table II, τ_0 returns to the nominal state at time slot $t_{r\to n} = t_{n\to r} + P_{0,1} + P_{0,2} = 20$. If we continue to use the static schedule after $t_{n\to r}$, rhythmic packet $\chi_{0,2}$ released at time slot 10 would miss its deadline at time slot 13. Therefore, the network cannot properly respond to the period and deadline changes of τ_0 using the static schedule.

In a centralized dynamic scheduling approach, the gateway constructs a temporary schedule for the network in the rhythmic mode, and broadcasts the differences between the dynamic and static schedule to each device node. The system resumes the static schedule when it returns to the nominal mode. Fig. 3(b) shows one possible dynamic schedule, which can accommodate all rhythmic and periodic packets, but introduces 6 updated slots (11, 12 and 15-18) with respect to the static schedule in Fig. 3(a). These updated slots must be piggybacked to a broadcast packet and propagated to all nodes in the RTWN. Since the payload size of a broadcast packet is always bounded, the maximum number of allowed updated slots (NUT) is limited. Suppose, in this example, the limit equals to 4. Then the dynamic schedule in Fig. 3(b) cannot be piggybacked to one broadcast packet. The on-line scheduling framework (OLS) proposed in [16] considers the constraint on NUT, and tries to drop some periodic packets to satisfy such constraint. Fig. 3(c) shows the dynamic schedule constructed

[2]Task τ_3 is a broadcast task, which has 2 hops. The first hop is from V_g to $V_0, V_1, V_2, V_3, V_4, V_6$, and the second hop is from V_3 to V_5.

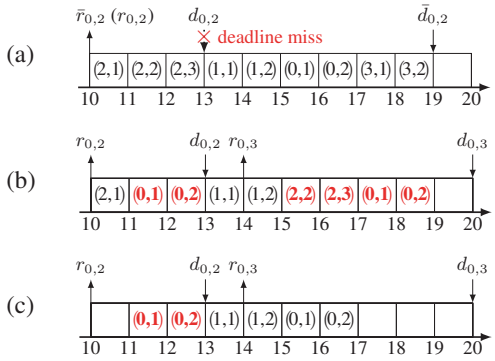

Fig. 3. (a) An example static schedule; (b) A dynamic schedule with 6 updated slots; (c) The dynamic schedule created by OLS. $\bar{r}_{0,2}$ and $\bar{d}_{0,2}$ denote the release time and deadline of $\chi_{0,2}$ when τ_0 is in the nominal state.

by OLS, which updates only 2 slots, but has to drop one periodic packet ($\chi_{2,2}$) to satisfy both the timing constraint of rhythmic task τ_0 and the constraint on NUT.

Though centralized dynamic scheduling approaches such as OLS can outperform static scheduling ones, they have two major drawbacks. First of all, they suffer from the limit imposed on NUT. If the NUT exceeds the payload size of a broadcast packet, more periodic packets have to be dropped. Secondly, centralized approaches tend to incur high latency for generating dynamic schedules, and the latency grows quickly as the size of the RTWN increases. If the updated schedule cannot be generated before the next broadcast slot, the current rhythmic event cannot be handled properly. We propose a new approach to address these drawbacks.

B. Overview of D^2-PaS

We observe that in a centralized approach the restriction on NUT is mainly due to the choice that the gateway undertakes all the work to handle disturbances while other device nodes only need to update its own schedule according to the slot update information received from the gateway. Such an approach implicitly assumes that device nodes have no local computing capability, which however is not true for RTWNs nowadays. For example, the CC2538 SoC running the OpenWSN network that we tested, has potential to do more computation than just run the OpenWSN wireless stack. Although these device nodes cannot perform as complex computations as the gateway does, they can afford some basic computations. Therefore, we propose to leverage such local computing capability and

design a distributed scheduling framework which can achieve better performance in terms of fewer dropped packets and more feasible task sets than centralized approaches.

Algorithm 1 Main function of D^2-PaS

```
1: while true do
2:    Every node generates and follows its local schedule.
3:    if a disturbance is detected and reported to the gateway then
4:       V_g checks the schedulability of the system after t_{n→r} and
         calculates the time duration of the system rhythmic mode.
5:       if the system is overloaded then
6:          V_g determines the periodic packets to be dropped.
7:       end if
8:       V_g propagates the rhythmic task information and dropped
         packet set to all nodes.
9:    end if
10: end while
```

Alg. 1 gives an overview of our distributed dynamic packet scheduling framework, D^2-PaS. Key steps in D^2-PaS will be elaborated in the next two sections. D^2-PaS works as follows. After system initialization, when a broadcast packet is received from the gateway, each node generates a schedule for a specific time duration according to some scheduling policy and then follows the schedule to operate. We adopt EDF [17] as the scheduling policy on each node. To generate such a schedule, each node uses the task and routing information to determine for each time slot whether it should send/receive a particular packet or stay idle. This can be done by simply following the EDF simulation process. Since every node maintains the same task and routing information, the schedules generated at different nodes are consistent.

When disturbance is detected, the S-node sends a rhythmic event request via τ_0 (following the current schedule) to the gateway. Upon receiving the request at time t' (see Fig. 4)[3], V_g first checks the schedulability of the system assuming τ_0 will be in the rhythmic state. If the system is overloaded, the gateway determines the dropped periodic packets to guarantee the deadlines of all rhythmic packets. V_g then piggybacks the information about dropped packets and the rhythmic task (task ID with corresponding $\overrightarrow{P_i}$ and $\overrightarrow{D_i}$) to a broadcast packet and disseminates it to all nodes in the network at time t''. Otherwise, only the rhythmic task information needs to be broadcast. Thus, instead of broadcasting the entire updated schedule, D^2-PaS only piggybacks the indices of the packets to be dropped to the broadcast packet when the system is overloaded. Upon receiving such broadcast information at or before t_{sp}, each node generates its local schedule accordingly using the updated information and the system enters the rhythmic mode at start point t_{sp}.

In order to ensure that D^2-PaS works properly, we need to tackle a few challenges. First, ideally, each node could construct and store the entire schedule for the hyperperiod in the nominal mode. This is however not practical due to the limited computing capability and memory at device nodes. Second, since constructing the schedule takes time, such computation should not occur when the node is supposed to send or receive packets. Third, to allow fast response to disturbances, an efficient method is needed at the gateway to determine which packets to drop. In the next two sections, we discuss in detail how D^2-PaS solves these problems.

[3]A system does not go into the rhythmic mode immediately after disturbance is detected. It only enters the rhythmic mode (and τ_0 enters the rhythmic state) after each device receives the broadcast packet at start point t_{sp}.

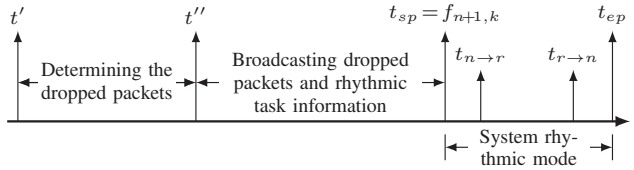

Fig. 4. Network operations after disturbance is reported to the gateway V_g. t' denotes the time slot at which V_g receives the rhythmic event request. t'' denotes the time slot at which V_g sends the first hop of the broadcast packet.

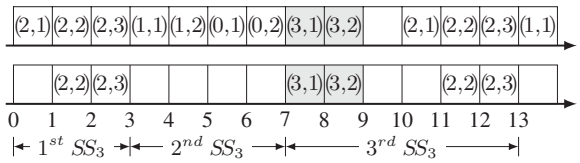

Fig. 5. Schedule segments of node V_3. The top (bottom) is the global (local) schedule of the system (V_3). Blank slots are global and local idle slots, respectively. Gray slots are assigned for transmissions of broadcast packets.

IV. Local Schedule Generation

This section focuses on addressing the challenges of local schedule generation at device nodes. The basic idea is to incrementally construct and store the schedule. Specifically, we follow two schedule design principles when constructing the schedule at each node: (i) construct one segment of the entire EDF schedule at a time to avoid generating the whole schedule, and (ii) perform local schedule generation in the idle slots of each node. The questions that need to be answered include (1) what these segments should be, (2) what is the upper bound on the lengths of these segments, and (3) what need to be maintained from one segment to the next to support the incremental computation. Below we first introduce a few definitions and then present our answers to these questions.

Definition 1 (Local Busy Slot) *A time slot is a* local busy slot *for node V_j if V_j either sends or receives a transmission belonging to any task in the time slot.*

Definition 2 (Local Idle Slot) *A time slot is a* local idle slot *for node V_j if it is not a local busy slot for V_j.*

Definition 3 (Schedule Segment) *A schedule segment, denoted as SS_j, is a segment of the local schedule constructed by node V_j at a time. SS_j starts either at each time slot when V_j receives a broadcast packet, or at the first local idle slot after V_j completes a sequence of consecutive local busy slots.*

We design D^2-PaS such that each node V_j generates its local schedule incrementally according to the definition of SS_j. A local schedule is constructed by simply following the EDF policy to determine that each time slot in SS_j should send/receive which packet or be idle. Generating a schedule segment must be completed within a local idle slot or a broadcast slot to guarantee that D^2-PaS works properly. Since no acknowledgement process is needed in a broadcast slot in IEEE 802.15.4e, the data link layer activities will not exceed 8 ms in a broadcast slot. Thus, the rest of the time slot is sufficient for local schedule construction as measured

in Section VII-B1. Using the broadcast slot ensures that schedule generation is always based on the most up-to-date system status when disturbances are detected. This approach automatically satisfies the schedule design principle (ii) as it uses local idle slots to derive the schedule. (Note that a broadcast slot at V_j is also a local idle slot for V_j based on the above definition.) Considering the motivation example in Section III-A, Fig. 5 shows the first three schedule segments of node V_3 in the example RTWN in Section III-A.

The time needed to generate SS_j depends on the length of SS_j. As shown in Fig. 5, each SS_j consists of one or more consecutive local idle slots followed by a sequence of local busy slots. Because only the busy slots determines the computation time for constructing SS_j, we only need to find the maximum length of consecutive local busy slots. Since the gateway can handle complex computations, we only consider the length of SS_j on device nodes. Lemma 1 and Theorem 1 below specify the upper bound on any SS_j as a function of the number of tasks passing through V_j.

Lemma 1 *If the system is schedulable, i.e. each packet completes all its transmissions before the deadline, for any device node V_j and task τ_i passing through V_j, there exists at least one local idle slot at V_j among any three consecutive transmissions*[4] *of τ_i passing V_j.*

Proof: Any three consecutive transmissions from task τ_i passing V_j can only be either (i) from three different packets of τ_i or (ii) from two different packets of τ_i. In case (i), suppose V_j finishes transmitting $\chi_{i,k}$. If there were no local idle slot at V_j before V_j works on a transmission from subsequent $\chi_{i,k+1}$, τ_i would only have one hop from source (V_j) to the destination or from the predecessor to actuator (V_j). (Otherwise, V_j would have at least one local idle slot for another transmission of $\chi_{i,k}$ not involving V_j.) This means that either τ_i does not pass through G-node or τ_i has no A-node or S-node. This contradicts our system model. For case (ii), assume that two of the three transmissions are an incoming and outgoing transmission of $\chi_{i,k}$, and the third one is the incoming (or outgoing) transmission of $\chi_{i,k+1}$ (or $\chi_{i,k-1}$). If no local idle slot exists at V_j among these three consecutive transmissions, it indicates that task τ_i only has two hops on its path from the source (S-node) to the destination (A-node). (Otherwise, there must exist a local idle slot at V_j when packet $\chi_{i,k}$ is transmitted on a third hop that passes V_j.) In this case, V_j must be the gateway since according to our system model, each task must pass through G-node. This, however, contradicts the assumption that V_j is a device node. □

Theorem 1 *If the system is schedulable, the maximum length of a schedule segment SS_j at any device node V_j is $2 \times n_j$, where n_j is the number of tasks passing through V_j.*

Proof: We first consider the case in which the length of SS_j equals to $2 \times n_j$. We use $\tau_1, \ldots, \tau_{n_j}$ to denote all tasks passing through V_j. Suppose V_j is the second node on the paths of all these tasks and the deadlines decrease from τ_1 to τ_{n_j}, i.e., $d_1 > \ldots d_{n_j}$. Assume packets are released following the pattern shown in Fig. 6. According to EDF, each packet requires two time slots on V_j (for receiving and sending) and is then preempted by a higher priority packet. In this case, the length of SS_j equals to $2 \times n_j$.

[4]In between consecutive transmissions from a single task, there may be transmissions from other tasks.

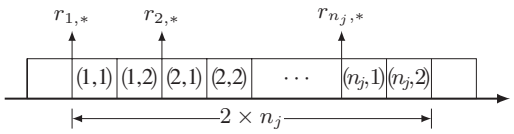

Fig. 6. An example showing the worst-case length of SS_j. $r_{i,*}$ denotes the release time of an arbitrary packet of task τ_i.

TABLE III. THE SCHEDULE TABLE STORED IN NODE V_3 AT $t = 0$.

Task	H_i	P_i	D_i	$\langle I_i, s_i, r_i \rangle$	$\langle c_i, p_i \rangle$
τ_0	2	10	9	N/A	$\langle 2, 1 \rangle$
τ_1	2	10	8	N/A	$\langle 2, 1 \rangle$
τ_2	3	10	7	$\langle 2, V_g, V_5 \rangle$	$\langle 3, 1 \rangle$
τ_3	2	10	10	N/A	$\langle 2, 1 \rangle$

Next we prove by contradiction that the length of SS_j cannot be larger than $2 \times n_j$. Assume that V_j is busy for more than $2 \times n_j$ slots consecutively. The number of tasks passing through V_j is n_j, indicating that there is at least one task requiring more than two time slots on V_j within SS_j. By Lemma 1, there exists at least one local idle slot on V_j among the transmissions. This contradicts our assumption. □

Now we examine what data need to be stored at each node to compute the local schedule. First the generated schedule must be stored. To save memory, only the information associated with the slots (e.g., slot 1, 2, 11 and 12 in Fig. 5) is stored. Second, to generate the local schedule, V_j must have the full knowledge of task time parameters. Furthermore, it may seem that the entire route information for each task is needed by V_j. However, this is actually not necessary since V_j only needs to generate the schedule involving V_j itself.

To facilitate the incremental computation of the local schedule, we maintain a schedule table at each node to assist the local schedule computation. Specifically, the schedule table at node V_j is initialized at the initialization phase of the network. A broadcast packet containing the complete task set information is propagated to all nodes in the RTWN. When V_j receives the broadcast packet, it extracts and stores necessary information in its schedule table. Table III shows the schedule table maintained at node V_3 at time slot 0 for the example RTWN in Section III-A. Here H_i, P_i and D_i are as defined in Section II. Tuple $\langle I_i, s_i, r_i \rangle$ captures the routing information of task τ_i's transmissions related to V_j. I_i indicates that V_j is the receiver of the I_i-th transmission of τ_i (and thus V_j is the sender of the (I_{i+1})-th transmission of τ_i). Note that, $I_i = 0$ indicates that V_j is the source (S-node) of τ_i and $I_i = H_i$ indicates that V_j is the destination (A-node) of τ_i. s_i and r_i denote the sender and receiver of transmissions χ_{i,I_i} and χ_{i,I_i+1}, respectively. The last column stores vector $\langle c_i, p_i \rangle$ representing the execution status of τ_i at the time slot when the currently generated schedule ends. As expected, the task information needs to be updated dynamically when V_j completes each computation of its schedule. Specifically, c_i denotes the remaining number of hops to its destination (A-node) and p_i indicates that the last packet activated at the end of the current SS_j is the p_i-th packet of τ_i. With such a properly maintained schedule table, local schedules can be efficiently computed and stored in our D^2-PaS framework.

V. Scheduling in the Rhythmic Mode

The local-schedule generation approach introduced in Section IV can work properly when the system is in the nominal mode. However, when the system enters the rhythmic mode, the rhythmic task adopts new periods and deadlines. Some periodic packets may need to be dropped to ensure all rhythmic packets are delivered by their deadlines. The information in the schedule table would not be sufficient for each node to generate a consistent local schedule. Though information such as all possible rhythmic task parameters can be stored locally, the device nodes need to know which task is entering the rhythmic state and what packets should be dropped if any. In this section we present the details on how D^2-PaS handles rhythmic events when disturbances are detected.

When a disturbance is reported to the gateway, D^2-PaS needs to find an updated schedule for the system to use in the rhythmic mode such that (i) all rhythmic packets meet their deadlines, and (ii) the minimum number of periodic packets are dropped. This updated schedule should be used in the time interval defined by the start point and end point of the rhythmic mode, i.e., $[t_{sp}, t_{ep})$, (see Fig. 4). Suppose the broadcast packet $\chi_{n+1,k}$ reaches all nodes at time slot $f_{n+1,k}(H_{n+1})$. We can simply use $f_{n+1,k}(H_{n+1})$ as t_{sp}. The selection of t_{ep} must guarantee that all packets released after it meet their nominal deadlines if no task enters the rhythmic state again. Thus the main problem to solve by the gateway is to determine (i) the end point of the rhythmic mode, t_{ep}, and (ii) which periodic packets to drop in the updated schedule. Let $\rho[t_{sp}, t_{ep})$ be the set of dropped periodic packets in $[t_{sp}, t_{ep})$. Our goal is to

$$\min |\rho[t_{sp}, t_{ep})| \tag{1}$$

subject to the following constraints.

Constraint 1 *Each rhythmic packet $\chi_{0,k}$ must meet its deadline $d_{0,k}$, i.e., $f_{0,k} \leq d_{0,k}$.*

Constraint 2 $f_{0,m+R} \leq t_{ep} \leq t^u_{ep}$

Here, t^u_{ep} is a user specified parameter which bounds the maximum allowed latency for handling the current rhythmic event. $f_{0,m+R} \leq t_{ep}$ ensures that the current rhythmic event can be completely handled before the system goes back to the nominal mode. Since the size of the broadcast packet is limited, we set a maximum allowed number of dropped packets in $[t_{sp}, t_{ep})$ and denote it by Δ^d. Thus we have

Constraint 3 $|\rho[t_{sp}, t_{ep})| \leq \Delta^d$.

In summary we aim to solve the following problem:

P1: Given task set $\mathcal{T}$, $t_{n\to r}$, t_{sp}, t^u_{ep} and Δ^d, determine t_{ep} and the set of dropped packets such that objective function (1) is achieved and all the three constraints above are satisfied. Below, we first discuss how to decide the end point and then describe the packet dropping algorithm used in D^2-PaS.

A. Selecting End Point of System Rhythmic Mode

The choice of end point t_{ep} impacts not only the length of the system rhythmic mode but also the number of dropped periodic packets. The concept of $[t_{sp}, t_{ep})$ is also used in [16]. In [16], t_{ep} (referred to as switch point) must be aligned with a nominal release time since the centralized approach needs to reuse the static schedule when the system returns to the nominal mode. In D^2-PaS, since each node generates its own local schedule, this requirement is no longer needed, which opens up possibilities for dropping fewer periodic packets.

A feasible end point must ensure that (i) the current rhythmic event is completely handled before it and (ii) all packets released after it can meet their nominal deadlines. We first identify a sufficient condition for a feasible end point.

Theorem 2 *t_{ep} is a feasible end point if the following conditions are satisfied.*

Condition 1 *$t_{ep} \geq f_{0,m+R}$, where $f_{0,m+R}$ is the finish time of the last packet released in τ_0's rhythmic state.*

Condition 2 *All rhythmic packets $\chi_{0,k}$ released earlier than t_{ep} must be finished by their rhythmic and nominal deadlines in τ_0's rhythmic state and nominal state, respectively.*

Condition 3 *All periodic packets $\chi_{i,k}$ released earlier than t_{ep} must be either finished by $\min(d_{i,k}, t_{ep})$ or dropped.*

We omit the proof of Theorem 2 since it is straightforward. The three conditions in Theorem 2 collectively guarantee that 1) the current rhythmic event can be completely handled, 2) all rhythmic packets in the system rhythmic mode can meet their deadlines, and 3) there is no workload carried over into the system after t_{ep}. Note that according to EDF [17], all packets released after t_{ep} can meet their nominal deadlines.

Our goal is then to select a t_{ep} such that the conditions in Theorem 2 are satisfied. As all rhythmic packets must be delivered by their deadlines, to satisfy Condition 1, the selected t_{ep} must satisfy $t_{ep} \geq r_{0,m+R} + H_0$, the earliest possible finish time of $\chi_{0,m+R}$. Similarly, it is easy to select a t_{ep} satisfying Condition 2 as long as it guarantees that $t_{ep} \geq r_{0,m+R+p} + H_0$ where $\chi_{0,m+R+p}$ is the last packet of τ_0 released before t_{ep}. Let $\chi_{0,m+R+q}$ denote the last packet of τ_0 released before t^u_{ep}. For Condition 3, we note that any time slot in $(r_{0,k}, r_{0,k} + H_0)$ $(m + R + 1 \leq k \leq m + R + q)$ cannot be t_{ep} since the packet released at $r_{0,k}$ needs at least H_0 time to finish. Therefore, time slot t, satisfying $t \in [r_{0,m+R} + H_0, t^u_{ep}] \setminus \cup_{m+R+1\leq k\leq m+R+q}(r_{0,k}, r_{0,k} + H_0)$, can be an end point satisfying Condition 3. Selecting the time slot in this set that leads to the minimum number of dropped packets can be very time consuming. To tackle this challenge, we first define the concept of *carry-over packet* and introduce a key lemma below.

Definition 4 (Carry-Over Packet) *A carry-over packet at time t is a packet released before t with a deadline after t.*

Lemma 2 *Suppose t^* is an end point satisfying the conditions in Theorem 2. Let $r_{i,k}$ $(0 \leq i \leq n)$ be the earliest release time satisfying $r_{i,k} \geq t^*$. It holds that $|\rho[t_{sp}, t^*)| \geq |\rho[t_{sp}, r_{i,k})|$.*

Proof: First consider that all carry-over packets at t^* are finished before t^*. Then slots in $[t^*, r_{i,k})$ are all idle slots. This indicates that $|\rho[t_{sp}, t^*)| = |\rho[t_{sp}, r_{i,k})|$. Next assume that some periodic carry-over packets are dropped when t^* is set as the end point. In this case, consider shifting the end point from t^* to $r_{i,k}$. Since $r_{i,k}$ is the earliest release time satisfying $t^* \leq r_{i,k}$, there are no packets released in $[t^*, r_{i,k})$. Because EDF is applied as the scheduling policy and $t^* \leq r_{i,k}$, packets

finished before shifting the end point can still be finished after the shift of the end point. Furthermore, since dropped periodic carry-over packets at t^* may be finished after shifting the end point as they have more time to finish, it leads to equal or fewer dropped carry-over packets for using $r_{i,k}$ as the end point. i.e., $|\rho[t_{sp}, r_{i,k})| \leq |\rho[t_{sp}, t^*)|$. □

Lemma 2 indicates that using the time slots between two successive release times as t_{ep} leads to dropping more packets than just using the later release time as t_{ep}. Thus, we only need to select end point from the release times satisfying both Condition 1 & 2. We refer to the set of all these feasible release times as the end point candidate set $\Gamma(t_{ep})$, where

$$\Gamma(t_{ep}) = \left\{ \forall r_{i,j} \in [r_{0,m+R} + H_0, t^u_{ep}] \setminus \bigcup (r_{0,k}, r_{0,k} + H_0) \right\}$$
$$where\ 0 \leq i \leq n, m+R+1 \leq k \leq m+R+q, \quad (2)$$

Performing schedulability test and running the packet dropping algorithm on every end point candidate can still be time consuming and often unnecessary. Instead, we can leverage the concept of *no-carry-over-packet point* to identify the end point directly.

Definition 5 (No-Carry-Over-Packet (NCoP) Point) *A time slot t_{np} is an NCoP point if and only if all carry-over packets at t_{np} are finished before t_{np} under EDF*[5].

Based on the definition of NCoP point t_{np}, there is no workload carried over into the system after t_{np}. This is consistent with Condition 3 of being a feasible end point. Furthermore, for an NCoP point to satisfy Condition 1 and 2, t_{np} should be larger than or equal to $\min(f_{0,m+R}, d_{0,m+R})$. With these constraints, all packets of τ_0 released within $[t_{sp}, t_{np})$ have sufficient time to be finished before their deadlines. Thus, any NCoP point t_{np} ($t_{np} \geq \min(f_{0,m+R}, d_{0,m+R})$) can be an end point candidate. Theorem 3 states that the earliest NCoP point satisfying the aforementioned condition is an end point that yields the fewest number of dropped packets.

Theorem 3 *The first NCoP point after $\min(f_{0,m+R}, d_{0,m+R})$ yields the minimum number of dropped periodic packets among all end point candidates in $\Gamma(t_{ep})$.*

Proof: Let t_{np} be the first NCoP point after $\min(f_{0,m+R}, d_{0,m+R})$, t_1 and t_2 be any end point candidate earlier and later than t_{np}, respectively. Their dropped periodic packet sets generated by an optimal packet dropping algorithm are denoted as $\rho^*[t_{sp}, t_1)$, $\rho^*[t_{sp}, t_2)$ and $\rho^*[t_{sp}, t_{np})$, respectively. We first prove $|\rho^*[t_{sp}, t_{np})| \leq |\rho^*[t_{sp}, t_1)|$. According to the conditions of a feasible end point, t_1 guarantees that all un-dropped packets released later than t_1 meet their nominal deadlines and all rhythmic packets finished by their rhythmic deadlines. If we drop exactly the same set of periodic packets as in $\rho^*[t_{sp}, t_1)$ when t_{np} is set as the end point, all conditions in Theorem 2 are satisfied. This indicates that $\rho^*[t_{sp}, t_1)$ is also a feasible solution when t_{np} is the end point. Since $\rho^*[t_{sp}, t_{np})$ is generated by an optimal algorithm, we have $|\rho^*[t_{sp}, t_{np})| \leq |\rho^*[t_{sp}, t_1)|$.

Next we prove $|\rho^*[t_{sp}, t_{np})| \leq |\rho^*[t_{sp}, t_2)|$ by contradiction. Assume that $|\rho^*[t_{sp}, t_{np})| > |\rho^*[t_{sp}, t_2)|$. $\rho^*[t_{sp}, t_2)$ consists of dropped packets released before t_{np} and at or after t_{np}, denoted as S_1 and S_2, respectively. That is, $|\rho^*[t_{sp}, t_2)| = |S_1| + |S_2|$. According to the assumption, we have $|S_1| < |\rho^*[t_{sp}, t_{np})|$. Since $\rho^*[t_{sp}, t_2)$ is a feasible solution, S_1 should guarantee that all rhythmic packets and un-dropped periodic packets with both release times and deadlines earlier than t_{np} can meet their deadlines. Furthermore, by the definition of NCoP point, all carry-over packets at t_{np} are finished before t_{np}. Hence S_1 could also be a feasible solution when using t_{np} as the end point, and $|S_1| < |\rho^*[t_{sp}, t_{np})|$ as assumed. This clearly contradicts the fact that $\rho^*[t_{sp}, t_{np})$ is generated by an optimal algorithm. Thus the theorem holds true. □

[5]If a packet misses its deadline, its unfinished part is dropped but the finished part is kept.

Algorithm 2 Determining End Point of Rhythmic Mode

```
Construct EDF schedule for all the packets released before t^u_ep
S ← {t_np | min(f_{0,m+R}, d_{0,m+R}) ≤ t_np ≤ t^u_ep}
if S ≠ ∅ then
    t_np ← min(S)
    if no deadline misses before t_np then
        return {NULL}
    end if
    return Γ(t_ep) = {t_np};
end if
Construct Γ(t_ep) according to Equation (2)
return Γ(t_ep)
```

According to Theorem 3, if there exists a NCoP point after $\min(f_{0,m+R}, d_{0,m+R})$ and before t^u_{ep}, it can be directly set to be t_{ep}. Combining Lemma 2, Equation (2) for $\Gamma(t_{ep})$, and Theorem 3, we have Alg. 2 (executed by the gateway) for determining the end point of the system rhythmic mode.

B. Determining Dropped Packets

In the real-time scheduling literature, the problem of minimizing the number of late jobs is well studied [18]–[21]. In this part, we show how our problem of determining the minimum number of dropped packets can be mapped to the problem of minimizing the number of late jobs, which can then be solved in polynomial time using Lawler's algorithm in [18].

The late-job minimization scheduling problem is categorized into different classes according to the processor environment, the constraints associated with the jobs and the scheduling criteria. These three parameters are described using notation $C_1|C_2|C_3$ [22]. Following this notation, our packet dropping problem can be reduced to the problem **P2** ($1 \mid pmtn, r_j \mid \sum w_j \times L_j$) studied by Lawler [18]: Given a set of jobs τ_j, with each job having release time r_j, execution time e_j, deadline d_j and weight w_j, find a preemptive schedule on a single processor such that the sum of the late job's weight is minimized. A job is on time ($L_j = 0$) if it completes before its deadline, otherwise it is a late one ($L_j = 1$).

To construct the mapping, we first introduce the concept of *active packet* for each end point candidate $t^c_{ep} \in \Gamma(t_{ep})$.

Definition 6 (Active Packet) *$\chi_{i,k}$ is an active packet for time slot t if and only if at least one of the two conditions: (1) $t_{sp} \leq r_{i,k} < t$, and (2) $t_{sp} < d_{i,k} \leq t$, holds.*

Now let $\Psi(t^c_{ep})$ be the active packet set containing all active packets to be scheduled within $[t_{sp}, t^c_{ep})$. Naturally, every packet with release time and deadline both in $[t_{sp}, t^c_{ep})$ must be an active packet and included in $\Psi(t^c_{ep})$. For carry-over packets at the start point t_{sp}, we only consider packets that

have not finished before t_{sp}. Here the gateway can run an EDF scheduling simulator from the time slot that V_g receives the rhythmic event request till t_{sp} to check which packets are not finished. The release times of these packets are set to be t_{sp}. Their execution times are set to be their remaining number of hops by t_{sp}. For carry-over packets at end point, t^c_{ep}, according to Condition 2 and 3 in Theorem 2, the deadlines of all carry-over packets are set to be t^c_{ep}. The execution times for these packets $\chi_{i,k}$ are set to be H_i.

Mapping the packet dropping problem to **P2** is done as follows. Each packet $\chi_{i,k}$ in $\Psi(t^c_{ep})$ with release time $r_{i,k}$, deadline $d_{i,k}$ and number of hops H_i can be mapped to a job with the same release time, deadline and execution time. (Note that a job executing for one time unit on a CPU is equivalent to a single packet transmission in an RTWN.) Let n_p denote the number of periodic packets in $\Psi(t^c_{ep})$. We then set the weight of each rhythmic packet to $n_p + 1$, and the weight of each periodic packet to 1. That is, the weight of any rhythmic packet is larger than the sum of the weights of all periodic packets. It ensures that any rhythmic packet is not dropped before the system drops all periodic packets. Since the execution time of any rhythmic packet is less than or equal to its deadline, no rhythmic packet would be dropped after all the periodic packets are dropped, which satisfies the definition of **P2**. It follows that a schedule for the job set that minimizes the weighted sum of the late jobs is also a feasible schedule that minimizes the number of dropped periodic packets for $\Psi(t^c_{ep})$. Using Lawler's method in [18], such a schedule can be found in $O(n^5_\Psi)$ time where n_Ψ is the number of packets in $\Psi(t^c_{ep})$.

The $O(n^5_\Psi)$ algorithm can be rather costly since the size of the active packet set for each t^c_{ep} can be relative large and the number of t^c_{ep} may also be not small. It is desirable to find a more efficient packet dropping algorithm. Our packet dropping problem bears similarity to the imprecise-computation scheduling problem [23]. In the imprecise-computation model with 0/1 constraint, each job is decomposed into two sub-jobs, a mandatory sub-job that must be finished by its deadline and an optional sub-job that can be either finished or dropped. If the optional sub-job is dropped, this job is imprecise. Our packet dropping problem can be reduced to a special case of the imprecise-computation scheduling problem where each rhythmic packet only has a mandatory part and each periodic packet only has an optional part. Thus, a schedule for an imprecise-computation job set that minimizes the number of imprecise jobs is also a feasible schedule that minimizes the number of dropped periodic packets in $\Psi(t^c_{ep})$. Scheduling imprecise computation jobs with 0/1-constraint can be done in $O(n^2_\Psi)$ time using a heuristic, proposed by Ho et al. in [23]. It is proved in [23] that the number of imprecise jobs produced by the heuristic is at most twice that of an optimal schedule. Due to page limit, we omit the detail on applying the heuristic for solving our problem.

Alg. 3 summarizes how the gateway determines which packets to drop in order to solve **P1**. For each end point candidate found by Alg. 2, the corresponding active packet set is constructed and a packet dropping algorithm (Lawler's algorithm or Ho-Heuristic) is applied to generate a dropped periodic packet set $\rho[t_{sp}, t^c_{ep})$ (Lines 2-3). If $|\rho[t_{sp}, t_{ep}{}^c)| \leq \Delta^d$, this dropped packet set is accepted and stored in $\mathcal{S}$. Finally, the solution with the minimum number of dropped packets in $\mathcal{S}$ is returned (Line 6). If the solution exceed the maximum allowed number of dropped packets Δ^d, D^2-PaS uses the earliest end point candidate in $\Gamma(t_{ep})$ as the end point and all periodic packets in the corresponding active packet set are dropped.

Algorithm 3 Determining Dropped Packets

Input: $\Gamma(t_{ep})$
Output: $\rho[t_{sp}, t_{ep})$
1: **for** $(\forall t^c_{ep} \in \Gamma(t_{ep}))$ **do**
2: Construct $\Psi(t^c_{ep})$; // active packet set
3: Generate a dropped periodic packet set $\rho[t_{sp}, t^c_{ep})$ based on $\Psi(t^c_{ep})$ using Lawler's algorithm or Ho-Heuristic;
4: $\mathcal{S} \leftarrow \mathcal{S} \bigcup \{\rho[t_{sp}, t^c_{ep})\}$;
5: **end for**
6: **return** $\arg\min_{\rho[t_{sp},t_{ep})\in\mathcal{S}} |\rho[t_{sp}, t_{ep})|$

VI. System Implementation

We have implemented D^2-PaS on a real-time wireless network testbed to validate its applicability in real-world RTWNs.

A. Overview of the Testbed

Our testbed is based on OpenWSN [24] with required enhancements to support D^2-PaS. OpenWSN is an open source implementation of the 6TiSCH protocol suite [25] which aims to enable IPv6 over the TSCH (Time Synchronized Channel Hopping) mode of IEEE 802.15.4e. As shown in Fig. 7(a), an OpenWSN network consists of multiple OpenWSN devices, an OpenWSN Root, and an OpenLBR (Open Low-Power Border Router). The Root and OpenLBR communicate through a wired connection (e.g., UART), using OpenBridge protocol. They together form the gateway.

The OpenWSN network adopts a TDMA based data-link layer. Adaptive synchronization mechanisms [26] are incorporated to ensure network-wide time synchronization among all device nodes. A time slot can be one of the following 5 types: OFF, TX, RX, SerialRX, and SerialTX. When an IPv6 packet is generated by a device node, it is compressed to a 6LoWPAN packet, and then transmitted in a dedicated TX slot to its neighbor on the path to the Root. This process repeats on the neighbor node until the packet reaches the Root. The Root forwards the packet to OpenLBR in an SerialTX slot, where the 6LoWPAN packet is decompressed and sent to Linux kernel for forwarding. If the destination of the packet is within the same network, the packet is forwarded back to OpenLBR. OpenLBR compresses it again to an 6LoWPAN packet and adds the 6LoWPAN source routing header, by examining the network topology stored in the RPL routing module. The 6LoWPAN packet is then sent to the Root in the next SerialRX slot. The Root transmits the packet over the air in the next available TX slot to its neighbor as specified in the source routing header. This process repeats until the packet reaches the final destination.

As shown in Fig. 7(b), our testbed consists of 8 wireless devices (TI CC2538 SoC + SmartRF06 evaluation board). Among these 8 devices, one is configured as the Root and connected to OpenLBR. The others are configured as device nodes and form a multi-hop RTWN. Four Segger J-Link JTAG Probes are used to debug the wireless stack. They are also used to retrieve logs from the devices to verify the correctness of the generated local schedules.

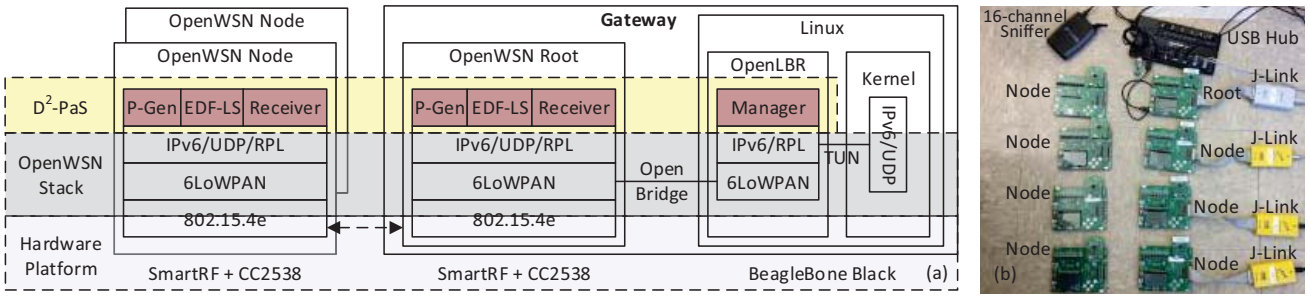

Fig. 7. An overview of the real-time wireless network testbed: (a) the system software architecture with the D^2-PaS implementation being highlighted; (b) a picture of the hardware components of the testbed.

B. Software Stack Enhancement to Support D^2-PaS

To implement D^2-PaS on the testbed, as highlighted in Fig. 7(a), we added the following four software modules. One of them (i.e., Manager) is implemented on OpenLBR. The other three modules are implemented in the application layer of the OpenWSN stack.

EDF Local Scheduler (EDF-LS) implements the distributed local schedule generation algorithm on the nodes (see Section IV). In the first idle slot of each schedule segment, it cleans up the existing link schedule, constructs the local schedule in the new schedule segment and installs them to the slotframe.

Packet Generator (P-Gen) constructs periodic packets according to its task information. It is invoked before the end of a time slot when MAC layer finishes all activities. If a packet needs to be transmitted in the next slot, P-Gen samples the sensor and prepare the packet. A broadcast packet, however, is initiated by the gateway in a broadcast slot instead of by P-Gen, and forwarded by the nodes according to a calculated broadcast graph.

Receiver binds to a UDP port to receive packets from the Manager. It handles three types of messages: 1) the link info message to install broadcast slots, 2) the task info message for the P-Gen and EDF-LS modules to construct periodic packets and schedule segments, respectively, in the nominal mode, and 3) the rhythmic event response message for the P-Gen and EDF-LS modules to construct rhythmic packets and schedule segments, respectively, in the rhythmic mode.

Manager is responsible for installing the broadcast graph in the network and initializing the P-Gen and EDF-LS modules in the device nodes. It also runs the end-point selection and packet dropping algorithms to handle rhythmic event requests. The results along with the rhythmic task information are broadcast to the device nodes for constructing local schedules.

Below we briefly summarize how D^2-PaS works in the testbed. After the network forms, it runs with the 6TiSCH minimal configuration, which has only one shared slot for exchanging management packets. The Manager generates the broadcast graph and installs broadcast slot(s) for each device node. The Manager then encapsulates all task information in a broadcast packet and disseminates it to the network. This task information will be used by the EDF-LS and P-Gen modules in individual device nodes to generate periodic packets and local schedule segments, respectively. In the meantime, the Receiver activates the EDF-LS and P-Gen modules at the specified time slot as indicated by the absolute slot number (ASN) in the broadcast packet. In our experiments, we set the

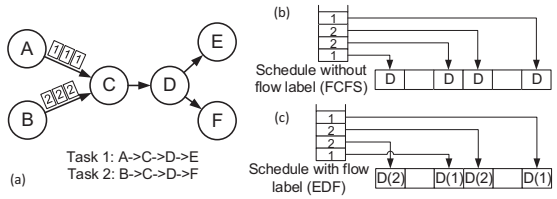

Fig. 8. (a) An example topology where two tasks have joint nodes. (b) Original schedule without flow label. (c) Modified schedule with flow label.

slotframe length to be larger than the longest schedule segment according to Theorem 1. This ensures that EDF-LS executes at least once in each slotframe. When the Manager receives a rhythmic event request, it executes the end-point selection and packet dropping algorithms to decide which packets to drop, and then broadcast their sequence indices to the network. Those packets will not be considered when the device nodes generate their local schedule segments.

To support the aforementioned four software modules, we made the following modifications on the OpenWSN stack:

1) Encapsulating task ID in the flow label: In the current IEEE 802.15.4e data link layer design, a time slot is allocated for a link between two neighbors. This, however, creates problems when multiple packets from different tasks share the same next hop on their routes. As shown in Fig. 8(a), *Task 1* and *Task 2* both pass through node C and D. Node C forwards packets from both tasks to node D in an FCFS manner, shown in Fig. 8(b). To ensure D^2-PaS functions correctly, the EDF-LS module on the device node requires the slots to be allocated for specific tasks, which should be as shown in Fig. 8(c). Without the task ID being encapsulated within the packets, the data link layer may not transmit the correct packet indicated by the EDF policy. To address this problem, we utilize the 32-bit *flow label* in IPv6 header to encapsulate the task ID information. On the device node, we extended the data structure of the time slot to include its associated task ID. A matching function was implemented to identify the earliest packet in the transmit queue which has the same task ID as the next time slot to be served. On the Root, a similar matching mechanism was also added to the serial driver to ensure that the serial forwarding procedure also follows the schedule generated by the EDF-LS.

2) Broadcasting IPv6 packets: D^2-PaS relies on IPv6 broadcast packets to disseminate the periodic task information to all device nodes in the nominal mode, and the rhythmic task information and dropped packet information in the system rhythmic mode. Since in an OpenWSN network, an IPv6

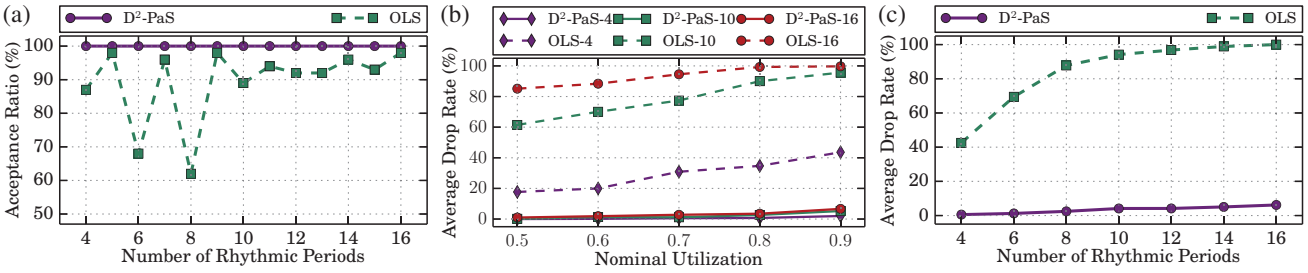

Fig. 9. Simulation results. (a) Comparison of the acceptance ratio with a nominal utilization $U^* = 0.5$; (b) Comparison of the average drop rate under different settings of R (denoted as D^2-PaS-R or OLS-R in the legend); (c) Comparison of the average drop rate with a nominal utilization $U^* = 0.9$.

packet needs to be compressed to a 6LoWPAN packet before transmitted to neighbor(s), we modified the stack and implemented the multicast bit in the 6LoWPAN layer to compress an IPv6 broadcast packet to a 6LoWPAN multicast packet. It is then transmitted in a broadcast slot and received by all neighbors who are listening on that slot. The 6LoWPAN multicast however is only 1-hop multicast. To achieve network-wide broadcast, the broadcast relay was implemented in the P-Gen module to re-broadcast the received broadcast packets.

3) Synchronizing P-Gen and EDF-LS with MAC: The application layer of the original OpenWSN stack runs with its own timer. In D^2-PaS, we prefer the task to releasing a packet precisely before its TX slot, and require EDF-LS to execute in the first idle slot of each schedule segment. Thus, the P-Gen and EDF-LS modules, although residing at the application layer, are required to synchronize with MAC layer time slot. To achieve this, we installed callbacks for both modules at the MAC layer, to trigger them when radio activities complete. The callbacks check whether the next slot is to generate a packet (by P-Gen) or to construct the local schedule segment (by EDF-LS), and invoke the corresponding modules to execute.

4) Enabling full stack on the Root: The original OpenWSN Root only implements the PHY and MAC layers of the OpenWSN stack. It exchanges packets between device nodes and OpenLBR over the OpenBridge directly, and no packet is directly handled by the Root itself. Since D^2-PaS requires the Root to run the EDF-LS and Receiver modules, the other upper layers of the Root were enabled in our testbed to support sending and receiving packets to and from the Root. The forwarding functions between MAC layer, the upper layers, and the OpenBridge were also modified accordingly.

VII. Performance Evaluation

In this section, we summarize key results from our simulation studies and testbed experiments on the applicability and performance of the D^2-PaS framework in RTWNs.

A. Simulation Studies

In our simulation studies we compare D^2-PaS to the state-of-the-art approach, OLS [16], which has been compared with a baseline algorithm and showed clear advantages.

1) Simulation Model and Parameters: To better control workload, we vary the nominal utilization of the task set deployed on the simulated RTWNs. Specifically, we use random periodic task sets generated according to a target nominal utilization U^*. Each task set is generated by incrementally adding random periodic tasks to an initially empty set $\mathcal{T}$.

Each periodic task τ_i is generated based on the following parameter settings: (i) the number of hops H_i follows a uniform distribution in $[2, 10]$, and (ii) period P_i is equal to deadline D_i and follows a uniform distribution in $[15, 50]$. (Note that the unit of R_i and D_i values is intentionally left not specified since only their relative values are important to schedulality study. The ranges of the parameters are chosen to reflect realistic RTWN applications.) After a task set is generated, we randomly select one task to be the rhythmic task τ_0. The period vector, $\overrightarrow{P}_0$ ($\overrightarrow{D}_0 = \overrightarrow{P}_0$), is generated with the following parameters: (i) the initial rhythmic period ratio, $\gamma = P_{0,1}/P_0$, is fixed to 0.2, and (ii) the number of elements in $\overrightarrow{P_0}$, R, can be any integer in the set of $\{4, 6, \ldots, 16\}$. We fixed γ and used R to tune the workload of the rhythmic task during the rhythmic mode, because R provides better control on changing the workload of the rhythmic task. We assume that upon entering the rhythmic mode, rhythmic task period $P_{0,k}$ first drops from P_0 to $P_{0,1}$ according to γ, and then gradually increases back to P_0 following the pattern of $P_{0,k}\,(1 \le k \le R) = \lfloor P_0 \times (\gamma + (k-1) \times \frac{1-\gamma}{R}) \rfloor$.

Additional parameters needed by both D^2-PaS and OLS are set as follows. (i) Switch point scaling factor $\alpha = 2$ (same value as used in [16] for comparing with other approaches.) α determines the switch point upper bound in OLS where $t^u_{sw} = t_{r\to n} + (\alpha - 1) \times P_0$ [16]. To ensure fair comparison, we also set the upper bound on the end point in D^2-PaS as $t^u_{et} = t^u_{sw}$; (ii) The payload of a broadcast packet is set to 90 bytes. To represent one updated slot, OLS needs 3 bytes where 13, 7 and 4 bits are used for slot ID, task ID and hop index, respectively. This indicates that the maximum allowed number of updated slots Δ^u in OLS is 30. Instead, in D^2-PaS, we only need 2 bytes to represent one dropped packet, where 7 and 9 bits are used for task ID and packet ID, respectively. Thus the maximum allowed number of dropped packets in D^2-PaS is $\Delta^d = 45$. (iii) Another parameter in OLS is the maximum allowed number of maintained child schedules in dynamic programming, denoted as β. For fair comparison, we set $\beta = 40$ which is one of the best values of β for the performance of OLS [16].

Note that, since the performance of the packet dropping algorithms in both D^2-PaS and OLS are dependent on the task set but not network topology, we only control the generation of the task set and assume the same network topology when comparing their performance.

2) Performance Comparison: We compare the performance of D^2-PaS and OLS using three metrics: (i) acceptance ratio (AR), defined as the fraction of feasible task sets over all the task set, (ii) average drop rate (DR), defined as the ratio between the number of dropped periodic packets and the total number of active periodic packets within $[t_{sp}, t_{ep})$, and (iii) computation overhead (CO), defined as the time taken for handling one rhythmic event request. A task set is feasible if the deadlines of all rhythmic packets are satisfied.

Fig. 9(a) shows AR as a function of number of rhythmic elements, R, for the task sets generated with a nominal utilization U^* of 0.5. Other utilization levels have the similar behavior and are omitted. Each data point is the average results of $1,000$ trials. We can observe from Fig. 9(a) that OLS cannot always find a feasible switch point to guarantee the timing requirements of all rhythmic packets. Its AR is around 90% but can be as low as 60%[6]. On the other hand, D^2-PaS can achieve 100% AR for all data points since a feasible end point always exists in D^2-PaS.

Fig. 9(b) shows DR as a function of the nominal utilization U^* under different settings of R. It can be clearly observed that D^2-PaS significantly outperforms OLS under all settings. As shown in Fig. 9(b), D^2-PaS exhibits a very low drop rate (1% on average) when the nominal utilization is less than 90%, while the performance of OLS drops significantly with the increase of R. To provide a finer granularity comparison, we set the nominal utilization $U^* = 0.9$ and increase R from 4 to 16 with a step size of 2. Fig. 9(c) shows that D^2-PaS performs much better than OLS, and the improvement on DR increases from 40% ($R = 4$) to 95% ($R = 16$).

We also compared CO of D^2-PaS and OLS for handling one rhythmic event. The execution times are from an Intel i3-2120 CPU (3.30GHz) used as the gateway. The average execution time of OLS varies from 2.2s to 10.4s when the nominal utilization U^* increases from 50% to 90%, while D^2-PaS needs no more than 1ms to handle the same rhythmic event. This is mainly because OLS relies on a dynamic programming based approach, which can be rather expensive when the number of active packets and the number of switch point candidates are large. A serious consequence of high computation overhead is that it may prevent OLS from successfully constructing a dynamic schedule before the next broadcast slot and thus the rhythmic event request may not be handled in time.

B. Testbed Experiments

Our testbed experiments focus on validating the functional correctness of D^2-PaS and measuring its efficiency. We tested 4 different task sets with varying numbers of tasks. Here we show a representative task set which contains 5 tasks: $\tau_0 = (5, 25)$, $\tau_1 = (6, 33)$, $\tau_2 = (5, 34)$, $\tau_3 = (5, 41)$ and $\tau_4 = (5, 60)$, where $\tau_i = (H_i, P_i = D_i)$. τ_0 is the rhythmic task with $\overrightarrow{P_0} = \overrightarrow{D_0} = [5, 7, 9, 11, 13, 15, 18, 20]$. τ_1, τ_2 and τ_3 are periodic tasks and τ_4 is the broadcast task.

[6]The AR curve of OLS is not monotonic, which agrees with the observation in [16]. This is due to the way that the switch point is selected in OLS. In OLS, the switch point must be aligned to a release time of the rhythmic task in its nominal state so as to reuse the static schedule. This requirement sometimes makes the rhythmic task miss its deadline even if all the periodic packets are dropped. However, no such requirement is needed in D^2-PaS.

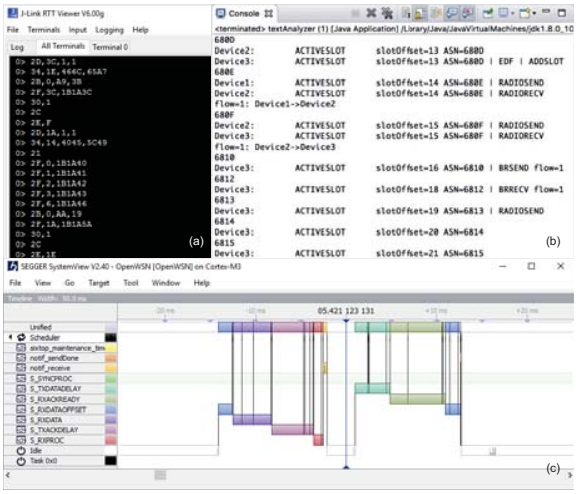

Fig. 10. (a) RTT Viewer to record events; (b) Log Analyzer to compile logs from device nodes; (c) SystemView timeline shows the activities and durations

1) Functional validation: The functional correctness of D^2-PaS is validated through logging and analyzing the debug information generated from the device nodes during the experiments. We used two tools (Segger RTT Viewer and SystemView) to retrieve two types of debug information, procedure check point and time measurement.

Procedure check point: To verify that the network runs correctly following the specified task information, we programmed each node to generate a log when it encounters an event. These events include: 1) active slot event with the associated absolute slot number (ASN) and slot type, 2) TX/RX events from both MAC and OpenBridge, 3) EDF-LS execution event with the computed active slots, and 4) packet release event from P-Gen. We used RTT Viewer to collect these events from individual nodes and record them into a log file (see Fig. 10(a)). We developed a system-wide log analyzer to process these logged events. This tool utilizes the ASN information associated with the events to align them into a system-wide log, and display the events generated at the same ASN from different nodes as the output, as shown in Fig. 10(b). By analyzing the system-wide log, we confirmed the following observations: 1) the P-Gen module releases periodic packets following the specified task information; 2) the EDF-LS module generates correct schedule segments according to our local schedule generation algorithm; and 3) the MAC layer operates correctly following the generated schedule segments.

Time measurement: Since the radio operations, schedule segment calculation, and packet generation all happen within a time slot, the timing of each operation needs to be carefully measured to ensure that all these operations can be completed within one time slot. We kept the task set running in the testbed for 30 minutes and used Segger SystemView to generate a system view of the time spent on each operation in each time slot. Fig. 10(c) shows the time measurements of the activities in an active RX slot followed by an active TX slot. From the measurements, we observed that in an active slot, after all radio activities, the P-Gen module can generate a packet and insert it to the MAC layer queue within 0.4ms. For the EDF-LS module, we measured the time it took to calculate and install each schedule segment by CPU-cycle stamping. The results

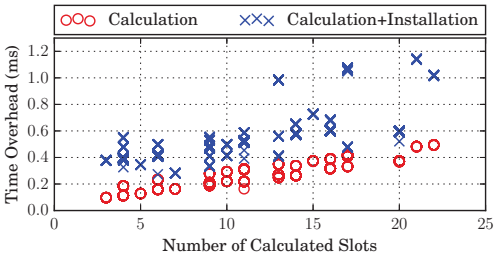

Fig. 11. Time measurements of schedule segment calculation and installation

TABLE IV. COMPARISON OF STACK FOOTPRINTS

Profile	Standard		with D^2-PaS	
	ROM	RAM	ROM(+%)	RAM(+%)
O0	54.4k	11.9k	71.7k (31.8%)	13.3k (11.8%)
O1	50.5k	11.9k	66.8k (32.3%)	13.3k (11.8%)
O2	41.4k	10.7k	48.7k (17.6%)	11.8k (10.3%)
O3	39.8k	10.7k	46.1k (15.8%)	11.8k (10.3%)

in Fig. 11 show that the time spent on the schedule segment calculation is proportional to the size of active slots in that schedule segment. The installation time is not constant but is always less than 1ms. This gives a total time of less than 1.2ms for each execution of the EDF-LS module.

2) Footprint comparison: By adding the new modules to the OpenWSN stack and making changes to the existing modules, both ROM and RAM usages increase. Table IV summarizes the footprint comparison between the standard OpenWSN stack and our modified version running D^2-PaS. The measurements are collected with four different levels of code optimization profiles (O0-O3). Considering the 512k ROM and 32k RAM available in CC2538 SoC, the increased ROM usage ($<$18k) and RAM usage ($<$1.4k) is moderate.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we introduce D^2-PaS, a distributed dynamic packet scheduling framework, to handle external disturbances in RTWNs. Different from a centralized packet scheduling approach where dynamic schedules are generated in the gateway and disseminated to all nodes, D^2-PaS leverages the computing capability on device nodes to generate schedules locally. The gateway only executes a packet dropping algorithm and disseminates the information of dropped packets if the system is overloaded. Our extensive simulation studies and testbed experiments demonstrate the applicability and effectiveness of D^2-PaS in real-world RTWNs. As future work, we will extend D^2-PaS to support handling simultaneous external disturbances in multi-channel RTWNs with unreliable links.

IX. ACKNOWLEDGEMENT

This work is supported in part by the U.S. NSF under Grant No. CNS-1319904 and ECCS-1446157, the NSF of China under Grant No. 61472072 and 61528202 and the China Scholarship Council. The authors would like to thank the reviewers for their valuable comments.

REFERENCES

[1] X. Hei, X. Du, S. Lin, and I. Lee, "PIPAC: Patient infusion pattern based access control scheme for wireless insulin pump system," in *INFOCOM*, 2013.

[2] K. Gatsis, A. Ribeiro, and G. Pappas, "Optimal power management in wireless control systems," in *ACC*, 2013.

[3] V. M. Karbhari and F. Ansari, "Structural health monitoring of civil infrastructure systems," CRC Press, 2009.

[4] S. Han, X. Zhu, D. Chen, A. K. Mok, and M. Nixon, "Reliable and real-time communication in industrial wireless mesh networks," in *RTAS*, 2011.

[5] Q. Leng, Y.-H. Wei, S. Han, A. K. Mok, W. Zhang, and M. Tomizuka, "Improving control performance by minimizing jitter in RT-WiFi networks," in *RTSS*, 2014.

[6] A. Saifulah, C. Lu, Y. Xu, and Y. Chen, "Real-time scheduling for WirelessHART networks," in *RTSS*, 2010.

[7] T. L. Crenshaw, S. Hoke, A. Tirumala, and M. Caccamo, "Robust implicit edf: A wireless mac protocol for collaborative real-time systems," *ACM Trans. Embed. Comput. Syst.*, 2007.

[8] W. Shen, T. Zhang, M. Gidlund, and F. Dobslaw, "SAS-TDMA: a source aware scheduling algorithm for real-time communication in industrial wireless sensor networks," *Wireless Networks*, 2013.

[9] M. Sha, R. Dor, G. Hackmann, C. Lu, T.-S. Kim, and T. Park, "Self-adapting mac layer for wireless sensor networks," in *RTSS*, 2013.

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

[11] O. Chipara, C. Lu, and G.-C. Roman, "Real-time query scheduling for wireless sensor networks," in *RTSS*, 2007.

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

[13] L. Li, B. Hu, and M. Lemmon, "Resilient event triggered systems with limited communication," in *CDC*, 2012.

[14] G. Buttazzo, E. Bini, and D. Buttle, "Rate-adaptive tasks: Model, analysis, and design issues," in *DATE*, 2014.

[15] J. Kim, K. Lakshmanan, and R. Rajkumar, "Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems," in *ICCPS*, 2012.

[16] S. Hong, X. S. Hu, T. Gong, and S. Han, "On-line data link layer scheduling in wireless networked control systems," in *ECRTS*, 2015.

[17] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, 1973.

[18] E. Lawler, "New and improved algorithms for scheduling a single machine to minimize the weighted number of late jobs," *Preprint, Computer Science Division, University of California*, 1989.

[19] E. L. Lawler and J. M. Moore, "A functional equation and its application to resource allocation and sequencing problems," *Management Science*, vol. 16, no. 1, pp. 77–84, 1969.

[20] J. M. Moore, "An n job, one machine sequencing algorithm for minimizing the number of late jobs," *Management science*, vol. 15, no. 1, pp. 102–109, 1968.

[21] P. Baptiste, "An $\mathcal{O}(n^4)$ algorithm for preemptive scheduling of a single machine to minimize the number of late jobs," *Operations Research Letters*, vol. 24, no. 4, pp. 175–180, 1999.

[22] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of discrete mathematics*, vol. 5, pp. 287–326, 1979.

[23] K. I. Ho, J. Y. Leung, and W. Wei, "Scheduling imprecise computation tasks with 0/1-constraint," *Discrete applied mathematics*, vol. 78, no. 1, pp. 117–132, 1997.

[24] T. Watteyne, X. Vilajosana, B. Kerkez, F. Chraim, K. Weekly, Q. Wang, S. Glaser, and K. Pister, "OpenWSN: a standards-based low-power wireless development environment," *Transactions on Emerging Telecommunications Technologies*, vol. 23, no. 5, pp. 480–493, 2012.

[25] T. Watteyne, M. Palattella, and L. Grieco, "Using IEEE 802.15.4e time-slotted channel hopping (TSCH) in the internet of things (IoT): Problem statement," RFC 7554, May 2015.

[26] D. Stanislowski, X. Vilajosana, Q. Wang, T. Watteyne, and K. S. Pister, "Adaptive synchronization in ieee802. 15.4e networks," *IEEE Trans. on Industrial Informatics*, vol. 10, no. 1, pp. 795–802, 2014.