REACT: an Agile Control Plane for Industrial Wireless Sensor-Actuator Networks

Dolvara Gunatilaka and Chenyang Lu*
Faculty of Information and Communication Technology, Mahidol University
*Cyber-Physical Systems Laboratory, Washington University in St. Louis

***Abstract*—Industrial automation is embracing wireless sensor-actuator networks (WSANs) as the communication technology for industrial Internet of Things. Due to the strict real-time and reliability constraints imposed by industrial applications, industrial WSAN standards such as WirelessHART employ centralized management to facilitate deterministic communication. However, a centralized management architecture faces significant challenges to adapt to changing wireless conditions. While earlier research on industrial WSANs has primarily focused on improving the performance of the data plane, there has been limited attention on the control plane, which plays a crucial role for sustaining the data plane performance in dynamic environments. This paper presents REACT, a reliable, efficient, and adaptive control plane for industrial WSANs. Specifically optimized for network adaptation, REACT significantly reduces the latency and energy cost of network reconfiguration, thereby improving the agility of WSANs in dynamic environments. REACT comprises (1) a *Reconfiguration Planner* employing flexible scheduling and routing algorithms and reactive update policies to reduce rescheduling cost, and (2) an *Update Engine* providing efficient and reliable mechanisms to report link failures and disseminate updated schedules. REACT has been implemented for a WirelessHART protocol stack. Evaluation results based on two testbeds demonstrate that REACT can reduce network reconfiguration latency by 60% at 50% of energy cost when compared to standard approaches.**

I. Introduction

Wireless sensor-actuator networks (WSANs) provide a flexible and cost-effective communication technology to incorporate Internet of Things (IoT) in industrial process control and automation. Industrial applications are inherently subject to real-time and reliability requirements, i.e., sensor data or control commands must be successfully delivered to its destination by their deadlines. Therefore, industrial WSAN standards such as WirelessHART [1] and ISA100 [2] adopt specific features to address such challenges. For instance, the WirelessHART network utilizes a Time Slotted Channel Hopping MAC (TSCH) [3], a TDMA-based protocol offering deterministic communication. The data plane relies on a *centralized* network manager to generate routes and transmission schedule for all the flows in the network. The centralized management approach offers several key advantages in industrial settings dominated by concerns about predictability, reliability, and observability of network operations. The centralized manager produces conflict-free transmission schedules that enable all packets to be delivered within their deadlines, thereby enhancing the predictability and reliability of communication. Furthermore, the centralized manager allows plant operators to inspect and log communication schedules and control commands, providing observability that is desirable in industrial operations. However, a key challenge faced by the centralized manager is to adapt quickly in response to network dynamics. This is because the centralized manager must gather network connectivity information from network devices to generate routes and a global schedule. The newly computed schedule is then disseminated into the network through a multi-hop wireless mesh network. This schedule update process can incur significant communication overhead, especially in a large, many-hop network. We note that in many industrial settings, nodes are stationary in a plant and environmental conditions are usually stable. Consequently, a global schedule may stay valid for a relatively long time. However, when changes occur, the network needs to be able to promptly adapt to the changes given the critical nature of many industrial applications.

Despite extensive research on industrial WSANs, most prior works have focused on the data plane, e.g., routing and scheduling algorithms. Comprehensive reviews of these works can be found in [4], [5]. However, there has been limited research on the control plane, which is responsible for monitoring and maintaining the performance of the data plane. Due to the unpredictable nature of wireless environment, a network may suffer link quality degradation or disconnection, which may lead to failure in industrial plants. While different techniques are used to enhance network reliability, e.g., multi-path forwarding [6] and per-link retransmissions [7], [8], when link failure occurs, a network must have the capability to promptly reconfigure communication routes and schedules given mission-critical nature of many industrial applications. Moreover, although industrial standards (e.g., WirelessHART) provide some high-level guidelines and specifications for its control plane, it leaves open the details regarding the actual design and implementation. To address the open challenge, we have developed ***REACT***, a reliable, efficient, and adaptive control plane optimized for adaptation in industrial WSANs. Specifically, REACT integrates two key components.

- ***Reconfiguration Planner*** employs strategies to compute and update routes and transmission schedules. Designed to minimize the changes to transmission schedules, the planner effectively reduces the latency and energy cost in disseminating a new schedule.
- ***Update Engine*** provides the mechanisms for network adaptation, including health reporting, failure notifica-

978-1-7281-6602-5/20/$31.00 ©2020 IEEE
DOI 10.1109/IoTDI49375.2020.00013

tion, and an efficient and reliable schedule dissemination mechanism. Our design also allows critical flows impacted by link failures to recover faster than other ones.

We have implemented REACT as the control plane of a WirelessHART protocol stack. Evaluation results based on two wireless testbeds demonstrate that REACT can significantly reduce the latency and energy cost of network reconfiguration, thereby supporting agile adaptation in industrial WSANs.

The rest of the paper is organized as follows. Section II reviews related works. Section III introduces the background. Section IV provides REACT architecture. Section V presents the policies used in the Reconfiguration Planner. Section VI details the design of the Update Engine. Section VII presents evaluation results, and Section VIII concludes the paper.

II. Related Works

Earlier works on real-time scheduling for a TDMA-based network with a centralized manager addressed different objectives, such as optimizing real-time performance [9], [10] or enhancing reliable communication [6], [7]. However, these works schedule transmissions without taking into account the need for a schedule to be updated due to network dynamics. Consequently, a network may incur considerable adaptation cost. Our system includes a scheduling policy that helps mitigate schedule reconfiguration cost by reducing schedule-related information that must be disseminated by a network manager.

There are other scheduling approaches to facilitate network adaptation. Dezfouli et al. introduced Rewimo [11], a wireless solution for real-time scheduling in mobile networks. Rewimo incorporates a scheduling technique for a TSCH network that enables schedule update in response to workload changes. Nevertheless, the scheduling policy does not consider the need for schedule updates due to link failure. To tackle the unreliable nature of wireless links, Yang et al. [12] and Shen et al. [13] developed scheduling algorithms that support efficient schedule reconfiguration. Their scheduling polices have limitations because they only consider a single channel protocol, and only single class of flow, i.e., all flows have the same period and deadline. Moreover, there are previous efforts [14], [15] that developed custom industrial WSAN protocols, which include network maintenance and adaptation mechanisms, based on a centralized management architecture. Again, these protocols have function limitations, e.g., they are designed for a single channel TDMA network, whereas REACT is geared towards a TSCH network operating on multiple channels.

Additionally, Livolant et al. [16] compared the cost of installing and updating schedules among existing protocols such as CoAP [17] and CoMI [18] that run on top of a 6TiSCH [19] network with a centralized manager, and a custom WSAN protocol OCARI [20]. This work investigated only the impact of these protocols on the number of messages required to install a global schedule and focused on optimizing the protocol headers, while REACT offers a more complete solution for schedule reconfiguration in an industrial WSAN.

In contrast to centralized management, a decentralized management architecture allows nodes to construct their own schedules, which enables them to adapt locally when network connectivity changes. For instance, the recent 6TiSCH standard combines the TSCH MAC and RPL routing. Duquennoy et. al [21] developed Orchestra, where nodes autonomously build their own schedules without requiring additional signaling among neighbors. uRes [22] introduced a schedule negotiation mechanism between neighbor nodes to compute local schedules. Accettura et al. [23] presented DeTAS, a decentralized scheduling protocol that ensures the smallest end-to-end latency, and reduces neighbor-to-neighbor signaling to generate local schedules. Although these protocols enhance network adaptability, they provide only best-effort service, and cannot guarantee conflict-free transmissions and real-time performance.

FD-PaS [24] and DistributedHART [25] are the recent distributed scheduling algorithms that support real-time communication. In contrast to our work, FD-PaS assumes a single channel communication, and is designed to handle external disturbances by adjusting data flow's period. While DistributedHART is designed for node-level multi-channel scheduling with spatial reuse and for graph routing, REACT offers a solution that follows industrial WSAN standards with a centralized manager that ensures conflict-free communication and provides observability of network operations.

Glossy [26] provides fast and efficient network flooding by exploiting constructive interference and the capture effect. Blink [27] combines a Glossy-based Low-Power Wireless Bus protocol [28] and real-time scheduling, and offers a promising alternative to TSCH-based networks. In contrast to the above-mentioned efforts, Blink is topology independent. Any change in network connectivity does not impact its global schedule. In comparison, REACT aims to enhance existing industrial standards for process automation through an agile control plane for centralized management.

In spirit, REACT is similar to Software Defined Networking (SDN) by separating the control planes from data planes and utilizing centralized software to control the behavior of a network [29]. However, traditional control planes of SDN can incur significant overhead for a WSAN. Baddeley et al. presented initial efforts to adopt SDN architecture for 6TiSCH networks [30] by mitigating the effects of SDN's control overhead on data traffic. In contrast, REACT is a control plane specifically designed to reduce the control overhead and latency. Furthermore, REACT is integrated with the WirelessHART architecture widely adopted in process industries.

III. Background

In this section, we provide background on the network model, and describe TSCH scheduling.

A. Network Model

Industrial WSAN standards such as WirelessHART select a set of specific network features to enable real-time and highly reliable communication. They adopt the IEEE 802.15.4

physical layer, offering a low data rate and low power communication, and operating on the 2.4 GHz band with 16 channels. On top of the physical layer is TSCH MAC protocol. TSCH is a TDMA-based protocol in which time is divided into slots. Each slot is 10 ms, long enough to accommodate a transmission of a packet and its acknowledgement. To prevent channel contention, only one transmission is allowed per channel in each time slot. TSCH also supports channel hopping, i.e., a node can hop to different channels in every time slot. Channel hopping provides frequency diversity, which helps mitigate the effect of interference on network reliability. At the network layer, we consider *source routing* that provides a single path from a source to a destination. A WSAN is supervised by a centralized manager that manages and optimizes operations of a network throughout the network lifetime to ensure industrial application requirements are satisfied.

A WSAN consists of a set of real-time flows $F = \{F_1, F_2, ...F_k\}$. For each flow F_i, a source node generates a packet at a periodic interval P_i. The packet must be delivered through a route ϕ_i to the destination within the deadline D_i, where $D_i \leq P_i$. A route ϕ_i is composed of a sequence of links, and a transmission over a link j of a flow i is denoted as t_{ij}. A flow F_i consists of a sequence of transmissions $\Gamma_i = \{t_{i1}, t_{i2}, ..., t_{in}\}$. Let T be the hyper-period (i.e., the least common multiple of the periods of flows) of a set of flows F. A *superframe* is a set of repeated slots of length T. Within a superframe, a flow F_i releases T/P_i packets. A set of flows F is *schedulable* if all the flows in F can be scheduled to meet their deadlines.

B. TSCH Scheduling

A centralized scheduler residing on the network manager constructs a global schedule. Building a schedule involves allocating a *time slot* and a *channel offset* to each transmission. To generate a feasible schedule, a scheduler must follow these ***scheduling constraints***:

1) Slot constraints:
   a) Due to a half-duplex nature of the IEEE 802.15.4 radio, a node can only send or receive at a given time, so no two transmissions in a time slot can share a common node.
   b) To prevent intra-network interference, only one transmission can be scheduled on each channel in a time slot.
2) Precedence constraint: a sender must receive a packet before forwarding it, i.e., transmission t_{ij} must be scheduled in a time slot before transmission t_{ij+1}.
3) Real-Time constraint: all flows must be scheduled to meet their deadlines.

Our work adopts ***fixed priority scheduling***, which offers an efficient and computationally inexpensive heuristic. Hence, it is commonly used for real-time systems. Each flow F_i is associated with a priority. A flow F_i has higher priority than a flow F_l if $i < l$. Priorities are commonly assigned based on a flow's period or deadline, and a scheduler schedules flows in decreasing order of priority. For each flow F_i, with respect to the scheduling constraints, a scheduler allocates a time slot and a channel offset to each transmission t_{ij}.

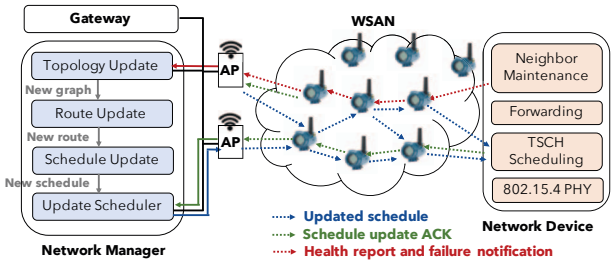


Fig. 1: REACT architecture and adaptation process.

IV. REACT Architecture

Figure 1 depicts REACT architecture, consisting of (1) a gateway enabling communication between network devices (sensors or actuators) and the centralized controller (2) a host running network manager software (3) a WSAN comprising multiple network devices forming a wireless mesh network, and (4) access points (AP) wired to the network manager and the gateway. Providing more than one access points offers redundant paths between the WSAN and the backplane. With REACT, the network reconfiguration process is triggered when link failure is reported to the network manager. To support this reconfiguration process, REACT includes two major components: a *Reconfiguration Planner* and an *Update Engine*.

A. Reconfiguration Planner

The reconfiguration planner resides on the network manager, and consists of the following components:

- Topology Update: A manager collects link statistics from all network devices, and generates a network topology represented as a graph. When a link fails to meet the reliability requirement, the link is removed from the topology. Conversely, if a node detects a new neighbor with reliable connectivity, a link between the node and its neighbor is added into the topology.
- Route Update: The module is responsible for creating flow routes and constructing a *broadcast graph* for disseminating schedule information from the manager downward to all network devices. With an updated topology, it computes new routes for flows associated with a failed link and adjusts the broadcast graph if necessary. Our work incorporates a *Partial Reroute policy*, which helps reduce the amount of flow's schedule that must be updated.
- Schedule Update: Based on the new routes, the scheduler recalculates a new global schedule. It utilizes *Gap-Induced scheduling* scheme, which complements our *rescheduling policy* in granting only schedules of flows affected by link failure to be updated, thereby lowering the schedule dissemination cost.

B. Update Engine

The update engine is composed of two different mechanisms to enable schedule reconfiguration.

- Health report and failure notification: Each network device maintains statistics pertaining to connectivity between a node and its neighbors. It learns these statistics through regular transmissions of data packets and periodic neighbor-discovery packets. These statistics are then reported to the network manager through *upstream flows*, which provide routes from nodes to access points. When a node identifies a failed link, it notifies the network manager through the same upstream flow.
- Schedule dissemination: An *Update Scheduler* installed on the network manager constructs packets. It schedules updates based on flow priority. Therefore, more critical flows receive updates and recover faster than less critical flows. Schedule-related information is propagated through a broadcast graph to destinations. Nodes receiving update commands modify their TSCH schedules, and send acknowledgement (ACK) back to the network manager, using the *upstream flows*. The ACK allows the network manager to handle packet losses and ensures all affected nodes receive their new schedule.

V. Reconfiguration Planner

In this section, we present policies that are incorporated into our reconfiguration planner to reduce schedule reconfiguration cost. These policies include the *gap-induced scheduling policy*, which the scheduler follows while creating or updating a schedule, the *partial reroute scheme* for a route update, and the *rescheduling policy* for a schedule update.

A. Gap-Induced Scheduling Policy

We propose a scheduling policy based on fixed priority scheduling to support efficient transmission rescheduling in response to link failure. Our policy is designed to (1) decrease the likelihood that an update to the schedule of a higher priority flow will impact the schedules of its lower priority flows. (2) enable a flow's schedule to be reused partially when possible. Therefore, only limited portions of a schedule are modified, which lowers the schedule dissemination overhead.

1) Key Ideas: With fixed priority scheduling, flows are scheduled in descending order of priority. For each flow, (1) transmissions are assigned to the *earliest* feasible slot in a sequential order [31] or (2) a scheduler schedules transmissions in reverse order by selecting the *latest* possible slot [11] meeting the scheduling constraint. In contrast to these two traditional approaches, our scheduling algorithm adopts the following polices to reduce the rescheduling cost.

- It ensures that for each flow F_i, the same schedule is repeated for every released packet $1, 2, ...T/P_i$ within a superframe. With a regular scheduling pattern, the network manager can disseminate less schedule-related information to wireless devices. In practice, flow periods are usually harmonic, so this policy is then easy to satisfy when we schedule or reschedule transmissions.

Schedule 1: Earliest Possible

ch/slot	1	2	3	4	5	6	7	8	9	10
0	a→b	b→c	c→d			a→b	b→c	c→d		
1			b→f	f→g	g→h					

Schedule 2: Introducing Gap

ch/slot	1	2	3	4	5	6	7	8	9	10
0	a→b		b→c		c→d	a→b		b→c		c→d
1		b→f				f→g				g→h

Schedule 3: Fewer Transmissions per Slot

ch/slot	1	2	3	4	5	6	7	8	9	10
0	a→b	b→c	c→d			a→b	b→c	c→d		
1				b→f	f→g				g→h	

Fig. 2: Examples of schedules constructed based on different heuristics. Each schedule consists of two flows: flow 1 ($P_1 = D_1 = 5$, $\phi_1 = a \to b \to c \to d$) is labeled in black, and flow 2 ($P_1 = D_2 = 10$, $\phi_2 = b \to f \to g \to h$) is labeled in red.

- It adds gaps between transmissions belonging to the same flow. This policy provides two benefits: (1) it allows a flow to be rescheduled partially when the route of a flow does not entirely change, and (2) it reduces the chance of lower priority flow schedules hindering the modification of a higher priority flow's schedule, so a change made to higher priority flows has less chance of impacting lower priority flows. For instance, in Figure 2, there are two flows $F1$ and $F2$, which are scheduled based on earliest possible slot (*Schedule 1*) and gap-induced (*Schedule 2*) policies. Initially, a flow sends a packet through route $a \to b \to c \to d$. Suppose $b \to c$ fails and a new route of F_1 is $a \to b \to f \to c \to d$. With *Schedule 1*, in order to schedule F_1 to meet its deadline at slot 5, the scheduler has to reschedule F_2 as well, since $b \to f$ and $f \to g$ block time slots 3 and 4. Otherwise, these two slots can be allocated to $f \to c$ of F_1. On the other hand, with *Schedule 2*, a scheduler can reschedule F_1 without affecting F_2 by removing $b \to c$, then adding $b \to f$ and $f \to c$ to slots 3 and 4, respectively. Here, the slots allocated to $a \to b$ and $c \to d$ also do not change.
- It reduces the number of concurrent transmissions in each time slot, which spreads out transmissions belonging to different flows. This heuristic also helps prevent lower priority flow from blocking updates on higher priority flow schedules. As shown in Figure 2, for *Schedule 3*, the scheduler avoids scheduling multiple transmissions in the same time slot. Again, supposing link $b \to c$ fails, then the scheduler needs only to modify the schedule of F_1 by adding $b \to f$ to slot 2, $f \to c$ to slot 3, and moving $c \to d$ to slot 4, and does not need to update F_2's schedule.

2) Gap-Induced Scheduling Algorithm: A detailed description of our gap-induced scheduling policy is presented in Algorithm 1. Here, a time slot means a slot offset in the superframe. With a superframe length of T, the slot offset

is within the range $[1, T]$. Both the periods and deadlines of flows are measured in slots.

The input of the algorithm is a set of flows F and c available channels, and the output is a global schedule S. The algorithm schedules flows in descending order of flow priority. Each flow F_i consists of a sequence of transmissions $\Gamma_i = \{t_{i1}, t_{i2}, ..., t_{in}\}$, and $|\Gamma_i|$ is the size of Γ_i. During a superframe, a flow generates T/P_i packets. Each packet q is released at time r_{iq} and has a deadline d_{iq}. The first step is to obtain a set of feasible slots χ_{ij} for each transmission t_{ij} (function $ObtainFeasibleSlot()$) starting from the last transmission t_{in} to the first transmission t_{i1}. A time slot x is added to χ_{ij} if $\forall q$, a slot $(q-1) * P_i + x$ satisfies the slot constraint for t_{ij} and is within $[r_{iq}, d_{iq}]$. Because the scheduler will assign only a slot in χ_{ij} to each t_{ij}, it is guaranteed that a flow can repeat the same schedule for every packet released in the superframe. The scheduler also determines the latest feasible slot $\lambda_{ij} \in \chi_{ij}$ for each transmission t_{ij}, where $\lambda_{ij} < \lambda_{ij+1}$ for transmission 2 to n-1. Note that a function $S.assign()$ assigns a time slot x_{ij} and an unused channel offset in x_{ij} to t_{ij}.

To spread out transmissions and introduce gaps between transmissions of a flow, the algorithm schedules the first transmission t_{i1} of flow F_i at the earliest possible slot $x_{i1} \in \chi_{i1}$, and assigns the latest feasible slot $x_{in} \in \chi_{in}$ to the last transmission t_{in}. For each remaining transmission t_{ij}, from $j = 2$ to $n-1$, the scheduler computes an ideal slot y (line 16). Suppose x_{ij-1} is a slot assigned to the previous transmission t_{ij-1}, and $|\Gamma_i| - j$ is the number of remaining unscheduled transmissions. Here, if the scheduler divides slots in the range $[x_{ij-1}+1, x_{in}-1]$ into $|\Gamma_i|-j+1$ equal segments, and assigns a time slot at the end of segment 1 to segment $|\Gamma_i|-j$ to each remaining transmission, then t_{ij} and remaining transmissions after t_{ij} can be spaced evenly between $[x_{ij-1}+1, x_{in}-1]$. Hence, in other words, y is the last slot in the first segment.

A function $SelectSlot()$ determines the best slot for t_{ij}. It computes the cost of choosing slot $x \in \chi_{ij}$ (line 41), considering the distance between x and y and the number of transmissions already scheduled in x ($\#trans(x)$). The scheduler selects the lowest-cost slot x_{ij} for t_{ij} (i.e., the slot closest to y and already allocated to the fewest transmissions), where $x_{ij} \in [x_{i,j-1}+1, \lambda_{ij+1}-1]$. The algorithm enforces t_{ij} to be scheduled after a slot $x_{i,j-1}$ (to preserve the precedence constraint) and before the latest feasible slot λ_{ij+1} of the next transmission $t_{i,j+1}$ (to ensure the flow's schedulability). The algorithm terminates when (1) transmissions of all flows are scheduled within their deadline, or (2) a scheduler cannot find a slot for t_{ij}, i.e., no slot in χ_{ij} is within the range $(x_{i,j-1}, \lambda_{ij+1})$.

B. Partial Reroute

Selecting a new route for a flow associated with link failure influences the flow's new schedule. To allow affected flows to reuse parts of their old schedules, the routing algorithm should choose a new route that includes links in the old route. A network topology is represented as a graph $G(V, E)$, where

ALGORITHM 1: Gap-Induced Scheduling

Input : A flow set F ordered by priority, c = the number of available channels
Output: A global schedule S

1 **foreach** *flow F_i, where i=1 to k* **do**
2 **foreach** *transmission t_{ij}, where $j = n$ to 1* **do**
3 $\chi_{ij} = ObtainFeasibleSlots(t_{ij})$;
4 **if** *j==1* **then**
5 x_{i1} = earliest slot in χ_{ij};
6 $S.assign(t_{i1}, x_{i1}, unusedChannel(x_{i1}))$;
7 **else if** j==n **then**
8 x_{in} = latest slot in χ_{ij};
9 $\lambda_{in} = x_{in}$;
10 $S.assign(t_{in}, x_{in}, unusedChannel(x_{in}))$;
11 **else**
12 λ_{ij} = latest slot in χ_{ij}, where $\lambda_{ij} < \lambda_{ij+1}$;
13 **end**
14 **end**
15 **foreach** *transmission t_{ij}, where $j = 2$ to $n-1$* **do**
16 $y = x_{ij-1} + (x_{in} - x_{ij-1}+1)/(|\Gamma_i| - j + 1)$;
17 $x_{ij} = SelectSlot(t_{ij}, \chi_{ij}, x_{ij-1}, \lambda_{ij+1}, y)$;
18 **if** x_{ij}==*-1* **then**
19 **return** $\emptyset$;
20 **end**
21 $S.assign(t_{ij}, x_{ij}, unusedChannel(x_{ij}))$;
22 **end**
23 **end**
24 **return** S;
25 **Function** $ObtainFeasibleSlots\,(t_{ij})$ **:**
26 **for** *$q = 1$ to T/P_i* **do**
27 Find X_q = set of slots in $[r_{iq}, d_{iq}]$ that meets the slot constraint
28 **foreach** *slot x in X_q* **do**
29 $x = (x-1) \mod P_i + 1$;
30 **end**
31 $\chi_{ij} = \bigcap_{q=1}^{T/P_i} X_q$;
32 **end**
33 **return** χ_{ij};
34 **Function** $SelectSlot\,(t_{ij},\ \chi_{ij},\ x_{ij-1},\ \lambda_{ij+1},\ y)$ **:**
35 **if** $\forall x \in \chi_{ij}, x \notin (x_{ij-1}, \lambda_{ij+1})$ **then**
36 **return** -1;
37 **end**
38 **foreach** *slot $x \in \chi_{ij}$ and $x \in (x_{ij-1}, \lambda_{ij+1})$* **do**
39 $dist(x, y) = |x - y| + 1$;
40 $w(x) = (\#trans(x) + 1)/c$;
41 $cost(x) = dist(x, y) * w(x)$;
42 **end**
43 **return** $\arg\min_x cost(x)$;

V is a set of network devices and E is a set of links. A link is added between nodes u and v if u and v can reliably communicate with each other, i.e., the packet reception ratios (PRR), the number of packets successfully received over the total number of packets sent, of both links $u \rightarrow v$ and $v \rightarrow u$ must be higher than the reliability requirement in all c channels used. Due to channel hopping, nodes must be able to reliably communicate in all channels used.

After the network manager detects a failed link, it updates a graph by removing an edge corresponding to the failed link from $G(V, E)$. It then calculates a new route for each affected flow in decreasing order of flow priority. Let ϕ_i be an old route of a flow i. For each affected flow, we update a weight of each link in $G(V, E)$. If a link is not included in the old route ϕ_i, the weight of the link is set to w. Otherwise, the weight of the link is assigned to $w/2$. Then, the algorithm uses the Dijkstra's algorithm to calculate a new route. By doing so, the algorithm will output the least-cost route, i.e., a shorter route containing links used in ϕ_i.

Due to the centralized control architecture, a packet needs to be propagated upstream from a source to an access point and a gateway, and downstream from a gateway and an access point to a destination. If the network contains more than one access points, there can be multiple combinations of an upstream and a downstream route for a new route ϕ_i'. Hence, the routing algorithm must select the combination with the lowest cost. Here, the cost is computed based on the number of operations required to update a route from ϕ_i to ϕ_i', which reflects the minimum cost of modifying a flow schedule. The cost is defined as $(nDEL * a) + (nADD * b)$, where $nDEL$ and $nADD$ are the numbers of deleting and adding operations for a flow's schedule, respectively. Here, a and b are the numbers of bytes required to DELETE or ADD a schedule of a transmission (See Section VI.B).

C. Schedule Reconfiguration Policy

Based on fixed priority scheduling, if a schedule of a higher priority flow is updated due to a failed link, all of its lower priority flows must be rescheduled as well, although these lower priority flows are not pertinent to the failed link. Such an approach introduces more modification to a global schedule, which implies a corresponding higher schedule dissemination cost. The goal of this policy is to reduce the schedule update cost by (1) rescheduling only flows affected by the failed link and (2) reusing a flow's schedule as much as possible. Consequently, fewer packets are required to update a global schedule. A schedule constructed based on the gap-induced scheduling policy has a higher chance of being successfully reconfigured to meet these two goals. For the same reason, the reconfiguration policy also adopts gap-induced scheduling scheme when rescheduling transmissions.

The input of the algorithm is a set of affected flows F', ordered from highest to lowest priority, and a current global schedule S. The output is a new global schedule S'. Let Γ_i be the old set of transmissions, and let $\Gamma_i' = \{t_{i1}, t_{i2}, ..., t_{im}\}$ be the new set of transmissions of flow F_i. The algorithm aims to assign new time slots and channel offsets to transmissions in $\Gamma_i' - \Gamma_i$ without modifying the schedules of the remaining transmissions in Γ_i'. For each affected flow F_i, the algorithm reschedule flow transmissions as follow:

Step 1: Let $\Gamma_s \subset \Gamma_i'$ be a set of sequential transmissions $\{t_{ip}, t_{ip+1}, ..., t_{iq}\}$, where every transmission in Γ_s is not a member of Γ_i. If Γ_s contains only one transmission, then $p = q$. For instance, suppose Γ_i is $\{a \rightarrow b, b \rightarrow c, c \rightarrow d, d \rightarrow e\}$ and Γ_i' comprises $\{a \rightarrow b, b \rightarrow c, c \rightarrow g, g \rightarrow e\}$. Here, $\Gamma_s = \{c \rightarrow g, g \rightarrow e\}$ is the only set of transmissions to be rescheduled. Transmissions $a \rightarrow b$ and $b \rightarrow c$ should be allocated to the same slots and channel offsets as in schedule S.

Step 2: The algorithm reschedules transmissions in Γ_s following the gap-induced scheduling policy, and time slots allocated for these transmissions must be within the range $[upper\_bound, lower\_bound]$ to avoid modifying the schedule of those transmissions that are not in Γ_s.

- If t_{ip} is the first transmission of F_i, then the $lower\_bound$ is the first slot of the superframe.
- If t_{iq} is the last transmission of F_i, the $upper\_bound$ is the flow deadline D_i.
- Otherwise, the $upper\_bound$ and $lower\_bound$ slots are $x_{ip-1}+1$ and $x_{iq+1}-1$, respectively, and x_{ip-1} and x_{iq+1} are time slots already assigned to the transmissions t_{ip-1} and t_{iq+1} (in the old schedule S).

Step 3: If the scheduler fails to schedule transmissions in Γ_s, it will obtain Γ_s' by performing one of the followings:

- If t_{iq} is already the last transmission of F_i, the scheduler will add one or more transmissions prior to t_{ip} to Γ_s and adjust a $lower\_bound$.
- If t_{ip} is the first transmission of F_i, the scheduler will include at least one transmission after t_{iq} to Γ_s and computes a new $upper\_bound$.
- Otherwise, the algorithm will compute which of the two options is better. The better option is the one providing the higher ratio of the number of slots in $[lower\_bound, upper\_bound]$ of Γ_s' to the number of transmissions in Γ_s'.

In both cases, the algorithm keeps adding transmissions to Γ_s until it reaches the next transmission that is already assigned a time slot, which sets a new $upper\_bound$ or a new $lower\_bound$. The scheduler determines the $upper\_bound$ and $lower\_bound$ and reschedules Γ_s' as in Step 2. By doing so, the algorithm gradually updates the schedule of those transmissions existing in the old route only when necessary, which enables a flow to preserve its old schedule as much as possible.

Step 3 is repeated until (1) Γ_s' can be rescheduled successfully, or (2) $\Gamma_s' = \Gamma_i'$ and the $upper\_bound$ and $lower\_bound$ reach their limits (i.e., $upper\_bound = D_i$ and $lower\_bound = 1$). For case (2), the algorithm will terminate since it cannot modify a schedule of F_i without rescheduling other flows. Otherwise, the algorithm ends when all the affected flows are scheduled to meet their deadlines.

VI. Update Engine

In this section, we present the designs of our update engine to handle network adaptation. We first discuss health reporting and failure notification mechanisms, and then a schedule dissemination mechanism.

A. Health Report and Failure Notification Mechanisms

Before a WSAN network is made operational, to determine a reliable set of links and channels for calculating routes and schedules, the network manager must collect complete topology information from every node and on all 16 channels. Because environmental conditions in a plant change over time, the manager must update the network topology throughout the network's lifetime, and must ensure the current flow routes and schedule stay valid. Therefore, while the network is operating, nodes must have the capability to report link condition to the network manager.

To maintain neighbor statistics, each node deploys a neighbor table of link statistics for each of its neighbors, e.g., PRR. A node obtains the PRR from periodic data packets and neighbor-discovery packets. Each node is required to broadcast a neighbor discovery packet periodically on all channels used, so other nodes hearing the packet can maintain their connectivity with the sender node. In particular, for each link, a node records if a packet is successfully received, and uses a sliding window to compute the current PRR.

Instead of periodically reporting to the network manager the link statistics of every neighbor, our system reduces the amount of data uploading to the network manager by letting nodes send a health report only when they discover new neighbors or when the link quality between a node and any of its neighbor falls below the reliability requirement, thereby saving the network resources. A node activates a link failure notification whenever an active link, i.e., a link used by any data flow, fails to meet the reliability requirement. To ensure that a notification will eventually arrive at the gateway, both the sender and receiver associated with the failed link will repeatedly send out notifications until they receive a new schedule. Moreover, to conserve network resources, these control information can be piggybacked onto periodic data packets. We allocate a separate upstream control flow for a node only when it has no upstream data flow passing through.

B. Schedule Dissemination Mechanism

Our schedule dissemination mechanism supports reliable and efficient schedule updates using the following features:

1) Update Scheduler: It is important that critical flows (flows with higher priority) promptly recover from link failure. Therefore, we design our system to ensure that schedules of more critical flows are updated first. To facilitate such a design, the update scheduler disseminates packets containing schedules of higher priority flows before packets with lower priority flow schedules. The update scheduler must first form packets containing schedule update commands and then determine the sequence in which these packets will be disseminated.

To reduce schedule dissemination overhead, the network manager identifies the differences between the old and the newly computed schedules, and distributes only the modified portions of the global schedule to the affected network devices. Our system supports two schedule modification commands ADD and DELETE. A DELETE command removes the transmission of a flow from a time slot, while an ADD command schedules a new transmission in a time slot. An ADD command contains a schedule entry associated with a new transmission. A schedule entry is represented by the following attributes: slot offset (2 bytes), channel offset (4 bits for the maximum of 16 channels), sender (1 byte), receiver (1 byte), and flow ID (1 byte). The flow ID indicates which flow a transmission belongs to. For a DELETE command, it is sufficient to specify only the sender (1 byte), the receiver (1 byte), and the time slot (2 bytes) of the transmission to be removed. The WirelessHART standard suggests that the number of nodes in a network should not exceed 80, therefore, 1 byte is enough to address all nodes in the network. Hence, it requires 6 bytes to ADD a new schedule, and 4 bytes to remove a schedule entry. Note that the maximum MAC payload is 98 bytes [3].

To constructs packets, the update scheduler first sorts the schedule update commands in decreasing order of flow priority, so that higher priority flow schedules are included in earlier packets. For each flow, DELETE commands precede ADD commands, since a schedule entry needs to be removed before a new entry is added. It adds schedule update commands into a packet until the packet is full, then creates a new packet and assigns each packet a sequence number.

The update scheduler distributes packets containing update commands of higher priority flows first, and ensures that all intended recipients have received the packets before disseminating the packets for lower priority flows. Once a node receives each packet, it immediately updates its TSCH schedule, and executes a new schedule in the next superframe. Therefore, higher priority flows can recuperate and use a new schedule, while the update scheduler is still disseminating and updating schedules of the lower priority flows.

2) Broadcast Graph: We use a broadcast graph to disseminate update commands. Compared to installing multiple downstream control flows from access points to every network device, a broadcast graph requires fewer time slots allocated for delivering a packet to every network device. In addition, with fewer allocated time slots, nodes can save the energy cost when no packet is being distributed, because every node scheduled to receive in a time slot must always listen on a channel for at least 2.2 ms [1] to detect if there is an incoming packet or not. If there is not, the node will turn off its radio. Otherwise, it continues to receive the packet.

A broadcast graph consists of two root nodes (access points), multiple intermediate nodes that receive and forward packets, and multiple leaf nodes, which only receive packets. To provide route diversity and ensure reliable dissemination of packets containing update commands, we require that an

intermediate or a leaf node must have two parents, and an intermediate node must receive a packet from both parents before broadcasting a packet to its children.

Since each packet contains schedules intended for different sets of nodes, it is not efficient to disseminate every packet to every node. Therefore, intermediate nodes in a broadcast graph will forward a packet only when they are on a path to packet's destinations. To realize such a forwarding mechanism, we install on intermediate nodes information about each node's children and its descendants in a broadcast graph. Hence, when a node checks the destinations of a packet, it can decide whether to forward a packet.

When a broadcast graph is also affected by link failure, its schedule will be updated as well. A broadcast graph's new schedule will be disseminated after the schedules of all affected data flows have been updated. To ensure that all nodes use the same broadcast graph during the current round of schedule update, a new broadcast graph schedule is executed at a later time, when the network again requires schedule reconfiguration.

3) Handling Packet Losses: We use an acknowledgement scheme to confirm that nodes have received all their schedule update packets. For every new packet a node receives, the node sends an ACK back to the network manager through an upstream flow. Similar to health reports and failure notifications, we allow an ACK to be piggybacked onto a data packet. Since there can be more than one packet intended for a node, an ACK must contain a packet number identifying the specific packet a node received. The update scheduler ensures that it receives ACKs from all recipients of the packet. Otherwise, it needs to resend that packet until the packet reaches all destinations.

4) Handling multiple link failures: REACT can handle multiple incoming link failure notifications. If a new failure notification arrives, while the network manager is computing a new schedule or when a new schedule has not been disseminated yet, then the network manager can halt those operations and recalculate a new schedule. Contrarily, if the dissemination process is on going as a new notification arrives, the manager will have to stop disseminating packets and start recomputing a new schedule again. In the meantime, nodes that already have received a schedule update will execute those partial new schedules. Since REACT updates schedules based on flow priority, flows that already received an update will run a new schedule, while the other flows execute on the old schedule, as they wait for the manager to compute and distribute the most recent schedule.

VII. Evaluation

To evaluate the performance of REACT, we develop the reconfiguration planner and the update engine to support schedule reconfiguration. We implement network manager software running on our server, and a protocol stack running on TinyOS 2.1.2 [32] and TelosB motes. The network manager can update the network topology, generate and update schedules and routes, and schedule update commands. In addition, we incorporate features necessary for schedule reconfiguration including health reporting, failure notification, and TSCH schedule updating into the network protocol stack, which supports TSCH MAC and source routing. We designate two nearby nodes in our testbed as access points. The server communicates with the access points through serial interfaces.

Following common practices for process monitoring and control applications, flows release packets periodically, and the periods of flows are harmonic. The periods are uniformly selected from the range $P = \{2^x, 2^{x+1}, ..., 2^y\}$. The manager constructs a collision-free TSCH schedule using fixed-priority scheduling, where only one transmission is allowed per channel in a time slot. We consider two fixed-priority scheduling policies commonly adopted for real-time systems: deadline monotonic and rate monotonic policies. Following a deadline monotonic policy, flows with shorter deadlines have higher priority, while a rate monotonic policy assigns flows with higher rates with higher priority. With the deadline monotonic policy, if a flow F_i has a period $P_i = 2^x$, then its deadline D_i is randomly selected from the range $\{2^{x-1} + |\Gamma_i| * 2, 2^x\}$, where $|\Gamma_i|$ is the number of transmissions of F_i. For the rate monotonic policy, D_i is configured to be equal to P_i. We adopt WirelessHART source routing, which provides a single route from a source to a destination. For each transmission belonging to a data flow, the manager reserves an additional time slot for a sender to retransmit a packet if the sender does not receive an ACK from the receiver.

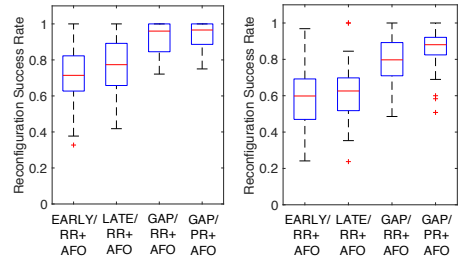
We quantify the performance of the reconfiguration planner based on two metrics: (1) the success rate in rescheduling a flow without modifying schedules of other flows, and (2) the number of packets required to update flow schedules. For the update engine, we run experiments on the local testbed, present the resulting schedule reconfiguration timelines, and measure the schedule dissemination latency and the energy cost. Table I summarizes different scheduling, route update, and schedule update policies that we compare against our work. By choosing combinations of the three policy types, we construct different approaches, each of the form *scheduling policy/route update policy + schedule update policy*.

A. Reconfiguration Planner Evaluation

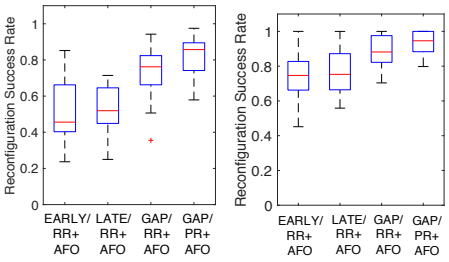
To evaluate the reconfiguration planner, we conduct simulation studies based on our local testbed topology containing 60 nodes spanning across three floors of the Jolley Hall at Washington University, and the Indriya testbed [33] topology consisting of 80 nodes. The topology information includes the PRRs of all links in the network in all 16 channels. We use the topology to construct a communication graph in which links added to the graph have PRRs of no less than 90% in all channels used and in both directions. Here, we use four channels. We randomly generate 50 flow sets under different traffic loads (i.e., when the numbers of flows are 24, 28, and 32) by varying the locations of sources and destinations of flows and access points. For each flow set, we obtain a set of links, where each link in the set is randomly picked. In each experiment, one link in this set is selected as a failed link.

TABLE I: SCHEDULING, ROUTE, AND SCHEDULE UPDATE POLICIES

Fixed-Priority Scheduling Policy
GAP: schedule transmissions based on a gap-induced scheduling policy
EARLY: schedule transmissions of a flow in sequential order and select the earliest feasible slot for a transmission
LATE: schedule transmissions of a flow in a reverse order and select the latest feasible slot for a transmission
Route Update Policy
PR: apply a partial reroute policy, and compute a new route using Dijkstra's shortest path algorithm
RR: reroute using Dijkstra's shortest path algorithm
Schedule Update Policy
AFO: reschedule affected flows only
ALL: reschedule affected flows and all of their lower priority flows



(a) 24 flows (local) (b) 28 flows (local)



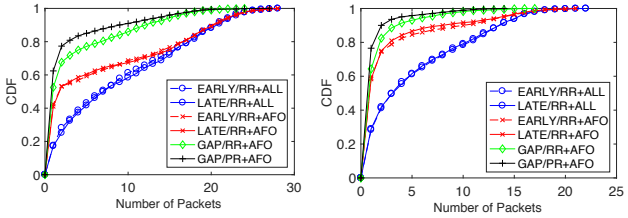
(c) 32 flows (local) (d) 32 flows (Indriya)

Fig. 3: Box plots of schedule reconfiguration success rates of AFO under the rate monotonic policy.

We set $P = \{2^{-1}, 2^0, 2^1\}$. Periods are uniformly assigned to flows in a flow set.

1) Schedule Reconfiguration Success Rate: To evaluate the effectiveness of our gap-induced scheduling policy in enabling the AFO reconfiguration policy to update only the schedules of those flows affected by link failure, we quantify the schedule reconfiguration success rate of AFO. The success rate is defined as the fraction of cases of a flow set in which AFO successfully reschedules only flows using the failed link. We compare our two approaches GAP/RR+AFO and GAP/PR+AFO, against EARLY/RR+AFO and LATE/RR+AFO.

Figure 3 shows box plots of the reconfiguration success rates of AFO under the rate monotonic policy. GAP/PR+AFO demonstrates better improvement as the traffic load increases (Figures 3a, 3b, and 3c). This is because it is more difficult for AFO to avoid rescheduling flows unaffected by link failure as more flows occupy the schedule, especially under the EARLY and LATE scheduling policies. In Figure 3c, GAP/PR+AFO increases the median success rate by 87.9% and 65%, compared to EARLY/RR+AFO and LATE/RR+AFO, respectively. Similar result can be observed under the Indriya testbed topology (Figure 3d).



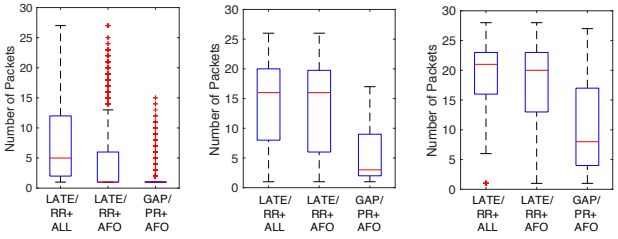
(a) Overall (local) (b) Overall (Indriya)

Fig. 4: CDFs of the total number of packets to be disseminated when the network has 32 flows, and under the rate monotonic policy.

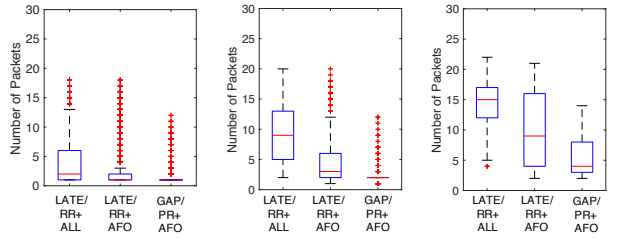
The results manifest the benefit of GAP in improving the reconfiguration success rate of AFO and in preventing the modification of higher priority flow schedules from impacting those of lower priority flows. In addition, it also shows that PR policy can further enhance the reconfiguration success rate of AFO by at most 12.5% (Figure 3c) because PR allows flows' schedules to be partially reused.

2) Number of Packets to Disseminate: We examine the ability of GAP/PR+AFO to reduce the cost of adapting to a failed link by computing the number of packets required to update the global schedule, which is a direct quantification of the schedule reconfiguration overhead. We compare our approach with two additional baselines, EARLY/RR+ALL and LATE/RR+ALL. Moreover, if AFO cannot successfully reconfigure a flow's schedule, the manager will reschedule the flow, along with all of its lower priority flows, starting from the highest priority flow that AFO fails to reconfigure.

Figures 4 presents the Cumulative Distribution Function (CDF) of the total number of packet to be disseminated under our local testbed and the Indriya testbed. GAP/PR+AFO proves to be most effective in reducing the total number of



(a) 1-5 flows (local) (b) 6-10 flows (local) (c) 11+ flows (local)



(d) 1-5 flows (Indriya) (e) 6-10 flows (Indriya) (f) 11+ flows (Indriya)

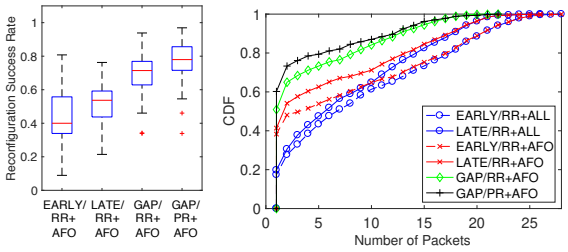
Fig. 5: Box plots of the number of packets required to update a schedule as the number of flows associated with a failed link increases, and under the rate monotonic policy.

packets required to update a schedule than other approaches. Furthermore, we investigate how the number of flows associated with link failure impacts the number of packets needed to update a schedule (Figure 5). We show only the results of LATE/RR+AFO and LATE/RR+ALL, since the LATE policy can perform better than or similar to the EARLY scheme. GAP/PR+AFO significantly outperforms the baselines when there are more flows associated with link failure. For instance, GAP/PR+AFO reduces the median number of packets by 60% and 55% compared to LATE/RR+AFO under the local and the Indriya testbed, respectively, when the number of affected flows is more than 10. GAP/PR+AFO offers a notable reduction over the baselines in the number of packets required to reconfigure a schedule, which translates into shorter reconfiguration latency and lower energy consumption.

We repeat the evaluation with the deadline monotonic scheduling policy, and observe similar results for both reconfiguration success rate and number of packet to disseminate. For brevity, we only show the results when the network contains 32 flows as presented in Figure 6.

B. Update Engine Evaluation

We assess if REACT helps reduce schedule reconfiguration latency and energy cost by conducting experiments on the local testbed consisting of 50 TelosB motes. Figure 7 shows the local testbed topology where two nodes are designated as access points. We explore three different configurations, i.e., when the network has 16, 24, and 32 flows. Transmission schedules are generated based on the deadline monotonic policy. We opt to compare our GAP/PR+AFO approach against only LATE/RR+ALL and LATE/RR+AFO, since they outperform



(a) Schedule reconfiguration success rate (b) Number of packets to be disseminated

Fig. 6: Schedule reconfiguration with 32 flows, and under the deadline monotonic policy (local).

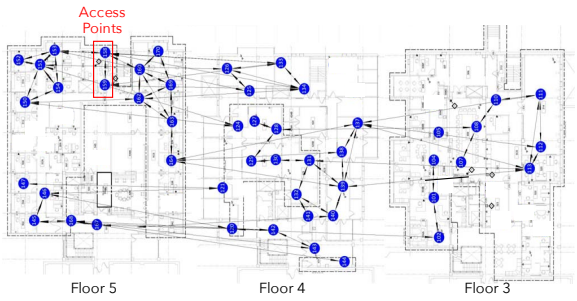
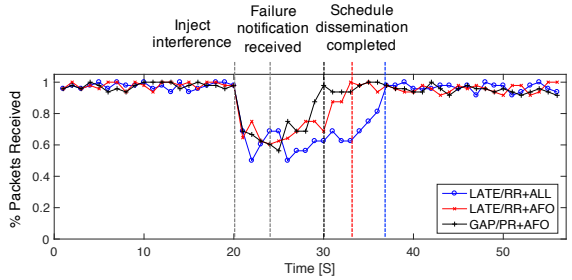
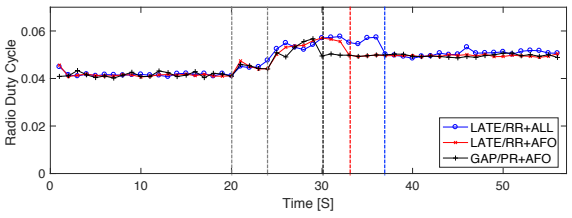


Fig. 7: Local testbed topology.



(a) Percentage of packets received



(b) Radio duty cycle

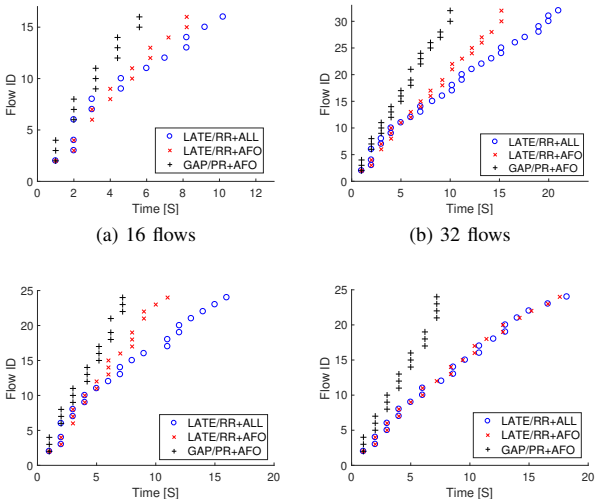
Fig. 8: Schedule reconfiguration timeline.

the baselines employing the EARLY policy under the deadline monotonic policy. We schedule a broadcast graph every 1 second. All flows generate packets with similar periods of 1 second. The deadline of each flow is chosen randomly from the range $[|\Gamma| * 2, 2^0]$, where $|\Gamma|$ is the number of transmissions of a flow. We allow control data (e.g., failure notification and ACK) to be piggybacked onto data packets, so no additional control flow is installed. All configurations follow the same settings, except where stated otherwise.

To demonstrate the benefit of REACT in reducing rescheduling overhead, we pick one critical link used by 85%-90% of flows as a failed link. To simulate network dynamics, WiFi interference is introduced close to the selected link. We use two Raspberry PIs to generate 5MB traffic on WiFi channel 1 overlapping with IEEE 802.15.4 channels 11 to 14 while the nodes communicate on channels 13 to 15. A node reports a link failure to the network manager once a link's PRR falls below 90%, and employs a sliding window of size 100 to calculate the link's PRR. To obtain average measurements, we repeat the experiment five times for each configuration.

Note that with centralized scheduling, schedule reconfiguration process involves failure detection, schedule recomputation, and schedule dissemination. Failure detection time depends on several factors (e.g., how often a node communicate with its neighbor and the manager, the size of the sliding window for calculating PRR, the reliability requirement, etc.), which introduce different tradeoffs. For instance, scheduling upstream flows to the manager less frequently preserves network resources, but incurs more reconfiguration latency. Selecting the optimized values for these parameters is not within the scope of this work. Schedule recomputation time depends on the complexity of schedule reconfiguration algorithm, the size of the network, and the number of flows associated with link failure, while schedule dissemination latency is mainly subject to the amount of information to be distributed and the period of a broadcast graph. Our work focuses on lowering the schedule dissemination latency by reducing the change to the route and schedule so fewer packets are required for the schedule update, and on offering an efficient and reliable mechanisms to update the schedule. In addition, our schedule recomputation algorithm incurs relatively low overhead compared to the failure detection and schedule dissemination processes. For example, based on our simulation studies, with 32 flows, we observe a maximum execution time of 13.5 ms for PR+AFO. The execution time is measured on a Macbook Pro laptop with a 2.7 GHz Intel Core i7.

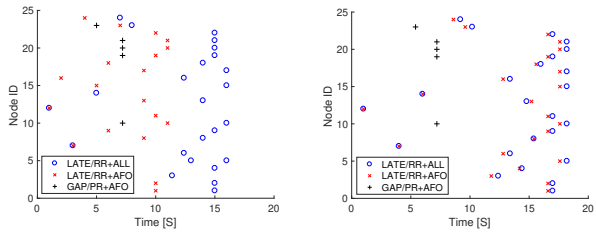
1) Schedule Reconfiguration Timeline: Figure 8 presents the schedule reconfiguration timeline to validate the correctness of our implementation. We quantify two metrics: (1) the radio duty cycle, the fraction of time a node has its radio on, and (2) the percentage of packets successfully received at their destinations. The network consists of 16 flows. When the network is stable, all approaches achieve a high percentage of packets received and incur low duty cycles. At time=20, interference is injected to degrade the link quality, and the percentage of packets received starts to decrease. Because the



(a) 16 flows (b) 32 flows

(c) 24 flows (d) 24 flows (AFO fails to reconfigure LATE)

Fig. 9: Flow schedule update latency under different traffic loads.

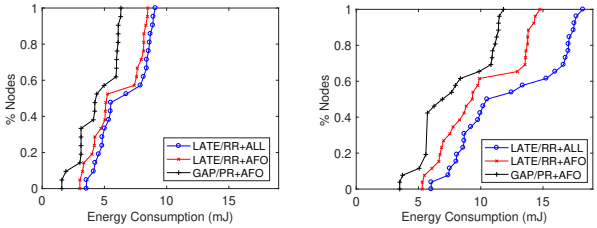


(a) 24 flows (b) 24 flows (AFO fails to reconfigure LATE)

Fig. 10: Latency in which each affected node has received a completed schedule.

failed link is used by multiple flows, nodes can detect link quality degradation and notify the network manager quickly. In this setting, the network manager takes less than 10 ms to recompute a new schedule. So after receiving the notification at time=24, the manager can promptly begin disseminating a new schedule at time=25.

During the schedule reconfiguration phase, we notice the percentage of packets received drops, and the radio duty cycle increases because nodes need to retransmit a packet more often and they also participate in schedule distribution. Schedule reconfiguration finishes at times 30, 33, and 37 for GAP/PR+AFO, LATE/RR+AFO, and LATE/RR+ALL, respectively. GAP/PR+AFO provides 25% and 43.8% improvements in schedule reconfiguration latency over LATE/RR+AFO and LATE/RR+ALL, since GAP/PR+AFO requires fewer packets to update the schedule. After schedule reconfiguration, the network performance returns to normal, and the slightly higher radio duty cycle is



(a) 16 flows (b) 32 flows

(c) 24 flows (d) 24 flows (AFO fails to reconfigure LATE)

Fig. 11: CDFs of node's energy consumption in mJ.

due to longer flow routes, which requires more transmissions.

Note that process monitoring applications may be able to tolerate some packet losses during the reconfiguration process. However, for more time-sensitive control applications, it is crucial to ensure that flows can still operate to meet the real-time requirement when the network suffers from link failure. Therefore, graph routing (a multi-path routing strategy supported by the WirelessHART protocol) should be adopted for this class of application to ensure reliable communication.

2) Schedule Dissemination Latency: We first validate that our policies indeed meet their goal of reducing schedule dissemination time. Latency is measured from when the first packet is disseminated until the schedules of all impacted flows are modified. Figure 9 presents the latency (in seconds) when a schedule of each flow is updated under different workloads, and Figure 10 plots the time required for each related node to receive a complete schedule.

We first consider the case where AFO successfully reschedules only flows affected by the failed link for LATE/RR+AFO. In Figures 9a, 9b, and 9c, GAP/PR+AFO reduces the schedule dissemination latency by approximately 45.1%-55% and 31.7%-34.5% over LATE/RR+ALL and LATE/RR+AFO, respectively. This is because GAP and PR enable AFO to reuse more of flows' old schedules. In addition, Figure 9d presents the result with 24 flows when AFO fails to update only flows associated with the failed link for LATE/RR+AFO. Here, the manager reschedules the remaining flows that could not be reconfigured by AFO, and also reschedules all of their lower priority flows. Compared to LATE/RR+AFO, GAP/PR+AFO further lowers the schedule update latency by 60%.

These results indicate the schedule dissemination latency achieved with GAP/PR+AFO is considerably lower than that of other approaches. This reduction shows the complementary benefit of our scheduling and reconfiguration policies in reducing the amount of schedule-related information to be broadcast once link failure is detected. Furthermore, the results in Figure 9 also verify that our update scheduler indeed modifies schedules of higher priority flows (i.e., flows with smaller ID) first. Therefore, these more critical flows suffer less from packet losses.

3) Energy Consumption: We next examine the performance of REACT in terms of energy efficiency. We measure the radio on time on each node and compute the energy consumption in mJ. According to the CC2420 radio specification [34], the power requirements for a transmission and a reception are 52.2 mW and 59.2 mW, respectively. The results presented in this section are obtained from the experiments in the previous section.

Figure 11 shows the CDFs of a node's average energy consumption. GAP/PR+AFO significantly improves the energy cost over the two baselines, especially the case when AFO fails to reschedule only the affected flows (Figure 11d). For example, with 32 flows (Figure 11b), under GAP/PR+AFO, 42% of the nodes consume less than 6 mJ, while for LATE/RR+AFO, only 8% of the nodes have energy costs lower than 6 mJ. In contrast, with LATE/RR+ALL, all nodes require more than 6 mJ. This is due to the fact that REACT disseminates fewer packets, and it reduces the number of nodes affected by the schedule update. As shown in Figure 10, many fewer nodes are affected by the schedule modification under GAP/PR+AFO than under either baselines. Reducing the number of nodes impacted by the schedule update results in fewer nodes participating in schedule dissemination, since we allow nodes in the broadcast graph to forward a packet only when they are on a path to the packet's destinations, thereby improving the node's energy efficiency.

VIII. Conclusion

To meet the stringent demands of industrial applications for real-time and reliable performance, industrial WSAN standards adopt centralized management to provide deterministic communication. The centralized management demands a highly efficient control plane to reconfigure the network in response to link failures. In this work, we design and implement REACT, a novel control plane to handle network adaptation. REACT includes a reconfiguration planner and an update engine to support efficient and reliable schedule reconfiguration. We implement and evaluate REACT with a WirelessHART protocol stack on a WSAN testbed. The results show that our system reduces the schedule dissemination latency by over 60%, and improves the node energy efficiency.

Acknowledgment

This work was sponsored by NSF through grants 1646579 (CPS) and by the Fullgraf Foundation.

REFERENCES

[1] "Wirelesshart, 2007," http://www.hartcomm2.org.

[2] "ISA100.11a," https://www.isa.org/.

[3] "IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH)," https://tools.ietf.org/html/rfc7554.

[4] C. Lu, A. Saifullah, B. Li, M. Sha, H. Gonzalez, D. Gunatilaka, C. Wu, L. Nie, and Y. Chen, "Real-Time Wireless Sensor-Actuator Networks for Industrial Cyber-Physical Systems," *Proceedings of the IEEE*, vol. 104, no. 5, pp. 1013–1024, May 2016.

[5] M. Nobre, I. Silva, and L. A. Guedes, "Routing and Scheduling Algorithms for WirelessHART Networks: a Survey," *Sensors*, vol. 15, no. 5, pp. 9703–9740, 2015.

[6] S. Han, X. Zhu, A. K. Mok, D. Chen, and M. Nixon, "Reliable and Real-Time Communication in Industrial Wireless Mesh Networks," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 3–12.

[7] S. Munir, S. Lin, E. Hoque, S. Nirjon, J. A. Stankovic, and K. Whitehouse, "Addressing Burstiness for Reliable Communication and Latency Bound Generation in Wireless Sensor Networks," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 303–314.

[8] R. Brummet, D. Gunatilaka, D. Vyas, O. Chipara, and C. Lu, "A Flexible Retransmission Policy for Industrial Wireless Sensor Actuator Networks," in *2018 IEEE International Conference on Industrial Internet (ICII)*, 10 2018, pp. 79–88.

[9] A. Saifullah, Y. Xu, C. Lu, and Y. Chen, "Real-Time Scheduling for WirelessHART Networks," in *2010 31st IEEE Real-Time Systems Symposium*, Nov 2010, pp. 150–159.

[10] O. Chipara, C. Wu, C. Lu, and W. Griswold, "Interference-Aware Real-Time Flow Scheduling for Wireless Sensor Networks," in *2011 23rd Euromicro Conference on Real-Time Systems*, July 2011, pp. 67–77.

[11] B. Dezfouli, M. Radi, and O. Chipara, "REWIMO: A Real-Time and Reliable Low-Power Wireless Mobile Network," *ACM Trans. Sen. Netw.*, vol. 13, no. 3, pp. 17:1–17:42, Aug. 2017.

[12] D. Yang, Y. Xu, H. Wang, T. Zheng, H. Zhang, H. Zhang, and M. Gidlund, "Assignment of Segmented Slots Enabling Reliable Real-Time Transmission in Industrial Wireless Sensor Networks," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3966–3977, June 2015.

[13] 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," *Wirel. Netw.*, vol. 19, no. 6, pp. 1155–1170, Aug. 2013.

[14] W.-B. Pöttner, H. Seidel, J. Brown, U. Roedig, and L. Wolf, "Constructing Schedules for Time-Critical Data delivery in Wireless Sensor Networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 10, no. 3, pp. 44:1–44:31, 2014.

[15] T. O'donovan, J. Brown, F. Büsching, A. Cardoso, J. Cecílio, P. Furtado, P. Gil, A. Jugel, W.-B. Pöttner, U. Roedig *et al.*, "The GINSENG System for Wireless Monitoring and Control: Design and Deployment Experiences," *ACM Transactions on Sensor Networks (TOSN)*, vol. 10, no. 1, pp. 4:1–4:40, 2013.

[16] E. Livolant, P. Minet, and T. Watteyne, "The Cost of Installing a 6TiSCH Schedule," in *AdHoc-Now 2016 - International Conference on Ad Hoc Networks and Wireless* , Lille, France, Jul. 2016. [Online]. Available: https://hal.inria.fr/hal-01302966

[17] "CoAP: The Constrained Application Protocol," https://tools.ietf.org/html/rfc7252.

[18] "CoMI: CoAP Management Interface," https://tools.ietf.org/html/draft-ietf-core-comi-01.

[19] "IPv6 over the TSCH mode of IEEE 802.15.4e (6tisch)," https://datatracker.ietf.org/wg/6tisch/about/.

[20] K. Al Agha, G. Chalhoub, A. Guitton, E. Livolant, S. Mahfoudh, P. Minet, M. Misson, J. Rahme, T. Val, and A. Van Den Bossche, "Cross-Layering in an Industrial Wireless Sensor Network: Case Study of OCARI," *journal of networks*, vol. 4, no. 6, pp. 411–420, Aug. 2009. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00390488

[21] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne, "Orchestra: Robust Mesh Networks through Autonomously Scheduled TSCH," in *Proceedings of the ACM Conference on Embedded Network Sensor Systems (Sensys)*, 2015, pp. 337–350.

[22] "uRES," https://openwsn.atlassian.net/.

[23] N. Accettura, E. Vogli, M. R. Palattella, L. A. Grieco, G. Boggia, and M. Dohler, "Decentralized Traffic Aware Scheduling in 6TiSCH Networks: Design and Experimental Evaluation," *IEEE Internet of Things Journal*, vol. 2, no. 6, pp. 455–470, 2015.

[24] T. Zhang, T. Gong, Z. Yun, S. Han, Q. Deng, and X. S. Hu, "FD-PaS: A Fully Distributed Packet Scheduling Framework for Handling Disturbances in Real-Time Wireless Networks," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2018, pp. 1–12.

[25] V. Modekurthy, A. Saifullah, and S. Madria, "DistributedHART: A Distributed Real-Time Scheduling System for WirelessHART Networks," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2019.

[26] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient Network Flooding and Time Synchronization with Glossy," in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE, 2011, pp. 73–84.

[27] M. Zimmerling, L. Mottola, P. Kumar, F. Ferrari, and L. Thiele, "Adaptive Real-time Communication for Wireless Cyber-Physical Systems," *ACM Transactions on Cyber-Physical Systems*, vol. 1, no. 2, p. 8, 2017.

[28] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-power Wireless Bus," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems (Sensys)*. New York, NY, USA: ACM, 2012, pp. 1–14.

[29] M. Ndiaye, G. Hancke, and A. Abu-Mahfouz, "Software Defined Networking for Improved Wireless Sensor Network Management: A Survey," *Sensors*, vol. 17, p. 1031, May 2017.

[30] M. Baddeley, R. Nejabati, G. Oikonomou, S. Gormus, M. Sooriyabandara, and D. Simeonidou, "Isolating SDN Control Traffic with Layer-2 Slicing in 6TiSCH Industrial IoT Networks," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2017, pp. 247–251.

[31] A. Saifullah, Y. Xu, C. Lu, and Y. Chen, "End-to-End Communication Delay Analysis in Industrial Wireless Networks," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1361–1374, May 2015.

[32] "TinyOS," http://tinyos.stanford.edu/tinyos-wiki/index.php/Main_Page.

[33] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda, "Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed," in *TRIDENTCOM*, 2011, pp. 302–316.

[34] "CC2420 Documentation," http://www.ti.com/lit/ds/symlink/cc2420.pdf.