WARP: On-the-fly Program Synthesis for Agile, Real-time, and Reliable Wireless Networks

Ryan Brummet, Md Kowsar Hossain, Octav Chipara, Ted Herman, and Steve Goddard
(ryan-brummet,mdkowsar-hossain,octav-chipara,ted-herman,steve-goddard)@uiowa.edu
Department of Computer Science, University of Iowa
Iowa City, USA

ABSTRACT

Emerging Industrial Internet-of-Things systems require wireless solutions to connect sensors, actuators, and controllers as part of high data rate feedback-control loops over real-time flows. A key challenge is to provide predictable performance and agility in response to fluctuations in link quality, variable workloads, and topology changes. We propose WARP to address this challenge. WARP uses programs to specify a network's behavior and includes a synthesis procedure to automatically generate such programs from a high-level specification of the system's workload and topology. WARP has three unique features: (1) WARP uses a domain-specific language to specify stateful programs that include conditional statements to control when a flow's packets are transmitted. The execution paths of programs depend on the pattern of packet losses observed at runtime, thereby enabling WARP to readily adapt to packet losses due to short-term variations in link quality. (2) Our synthesis technique uses heuristics to improve network performance by considering multiple packet loss patterns and associated execution paths when determining the transmissions performed by nodes. Furthermore, the generated programs ensure that the likelihood of a flow delivering its packets by its deadline exceeds a user-specified threshold. (3) WARP can adapt to workload and topology changes without explicitly reconstructing a network's program based on the observation that nodes can independently synthesize the same program when they share the same workload and topology information. Simulations show that WARP improves network throughput for data collection, dissemination, and mixed workloads on two realistic topologies. Testbed experiments show that WARP reduces the time to add new flows by 5 times over a state-of-the-art centralized control plane and guarantees the real-time and reliability of all flows.

CCS CONCEPTS

• **Networks** → **Network dynamics**; **Network reliability**; **Cyber-physical networks**.

KEYWORDS

Wireless networks, real-time wireless networks, reliability, software synthesis

ACM Reference Format:
Ryan Brummet, Md Kowsar Hossain, Octav Chipara, Ted Herman, and Steve Goddard. 2021. WARP: On-the-fly Program Synthesis for Agile, Real-time, and Reliable Wireless Networks. In *The 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021) (IPSN '21), May 18–21, 2021, Nashville, TN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3412382.3458270

IPSN '21, May 18–21, 2021, Nashville, TN, USA
2021. ACM ISBN 978-1-4503-8098-0/21/05.
https://doi.org/10.1145/3412382.3458270

1 INTRODUCTION

Over the last decade, we have seen the development of transmission scheduling techniques (see [18, 21] for overviews) and their widespread adoption in process control industries (e.g., [18, 22]). These efforts have primarily focused on supporting real-time and reliable communication over low-power networks composed of resource-constrained devices that often run on batteries. In contrast, the future Industrial Internet-of-Things (IIoT) applications will employ sophisticated sensors such as cameras, microphones, or LIDAR, which generate large volumes of data. The generated data will be processed by embedded devices that are (grid) powered and have sufficient processing resources to run machine learning and computer vision algorithms. Such IIoT applications require that, in addition to real-time and reliable communication, wireless protocols also support higher data rates, handle variable workloads, and cope with short-term fluctuations in link quality and topology changes.

As a concrete example, consider a John Deere's 32-row planter with 15 sensors per row [7], each of the 480 sensors generating 5 samples per second. A modern John Deere herbicide sprayer requires even higher throughput to handle the traffic generated by the approximately 250 cameras necessary for "sub-inch accuracy of the application of herbicide" [13]. Each sensor is powered with a serial line running along the tractor's boom or chassis and wirelessly transmits images over multi-hop *real-time flows* to the tractor processing platform where machine learning techniques are used to classify the types of weeds in the video stream. In response, commands are sent to herbicide spray nozzles to target the identified weeds. Mesh networking techniques can be applied for fault-tolerance. An effective wireless solution must also handle the dynamic wireless conditions observed in the field (e.g., variable humidity and dust, which affects signal propagation) and adapt communication schedules as the sensing rates change with the driving speed and field conditions.

The contribution of this paper is WARP – a novel approach that uses software synthesis to meet the demands of the future IIoT applications. Software synthesis usually relies on SMT solvers and requires resource-rich machines. These techniques need to be adapted to embedded devices and extended to build wireless networks that provide real-time and reliable performance in the presence of network dynamics. Three components support this goal: (1) a data plane whose operation is specified as programs that are more expressive than traditional schedules; (2) a scalable software synthesis procedure that generates programs that satisfy the flows' deadline

and reliability constraints; and (3) an efficient control plane to handle workload and topology changes.

A network operator programs a WARP network by specifying its workload. The workload is composed of real-time flows that carry data between sensors, controllers, and actuators. Flows are periodic, may span multiple hops, and deliver packets subject to deadline and reliability constraints. We have developed techniques to synthesize *programs* that specify the data plane's behavior based on the supplied workload and topology. The programs are written in a domain-specific language (DSL) that can express more complex behaviors than traditional scheduling approaches. A feature that distinguishes programs from schedules is that programs are stateful and may include conditional statements. The execution of an instruction to forward a flow's packet changes the flow's state to indicate whether the transmission succeeded or not. Subsequent instructions may query this state and make conditional decisions on what packets to forward. The combination of state and conditional actions leads to programs having *multiple execution paths* that depend on the pattern of packet losses observed during their execution. This opens the opportunity to synthesize programs whose actions are optimized to handle likely packet loss patterns and readily adapt to the rapidly evolving packet loss patterns observed during their execution. Our experiments demonstrate that programs provide higher performance than scheduling approaches while remaining simple to analyze.

The synthesis procedure ensures that the execution of programs will not result in nodes performing conflicting operations. Additionally, the synthesis procedure also guarantees that the likelihood of a flow's packets reaching their destination by their deadline exceeds a user-specified threshold. This property holds under our Threshold Reliability Model (TLR) that allows link quality to vary arbitrarily from slot-to-slot but always remains above a minimum level of reliability. WARP cannot guarantee the network's performance when the quality of links falls below the minimum packet delivery rate (PDR). In this case, the minimum PDR may be lowered or the flow's routes may need to be adjusted to include links that meet the minimum PDR. TLR is inspired by Emerson's best practices for deploying WirelessHART networks [19], which encourage network operators to ensure that links have a minimum PDR of 60% – 70% at deployment time. We have developed a computationally efficient approach to reason about the probability of delivering packets under TLR by considering all packet loss scenarios and their associated execution paths and likelihoods.

WARP includes a decentralized control plane that can adapt to workload and topology changes without synthesizing new programs. Our insight is that when nodes share the same topology and workload information, they can run the same synthesis procedure locally to generate identical programs. This approach significantly improves the network's agility since workload or topology changes only require disseminating the added or modified flows and routes. Such updates can be encoded efficiently, most often, in a single packet. However, the challenge is to implement WARP's program synthesis on embedded devices on which it is infeasible to run SMT solvers that are commonly used in software synthesis. Within 10 ms slots, WARP can synthesize and execute programs on-the-fly on TelosB and Decawave DWM1001 devices.

Protocol	Data Rate	Workload	Energy
WirelessHART	low	fixed	constrained
REACT [15] FD-PaS [24] DistributedHART [20]	low	variable	constrained
WARP	high	variable	unconstrained

Table 1: Requirements space of IIoT protocols

We have evaluated WARP through simulations and testbed experiments using workloads typical in IIoT systems. Our simulations indicate that WARP can significantly improve throughput across typically IIoT workloads in two realistic topologies. We have implemented our techniques on Decawave DWM1001, which use 802.15.4a radios, and on TelosB, which use 802.15.4 radios. The testbed experiments demonstrate the feasibility of using synthesis techniques to construct programs on-the-fly. Furthermore, our experiments validate that WARP can satisfy the real-time and reliability constraints of flows and adapt quickly to workload changes.

2 RELATED WORK

Data plane: At the heart of wireless networks that support real-time communication are transmission scheduling techniques (e.g., [16, 20, 23, 24] and surveys [18, 21]). For example, WirelessHART specifies the data plane's behavior using a (repeating) super-frame, a two-dimension matrix whose cells grant access to a specific slot and channel. Real-time traffic is supported by using dedicated cells in which a single transmission is scheduled, while best-effort traffic is supported using shared slots. Under these and similar assumptions about the data plane, researchers have developed numerous algorithms to construct efficient schedules under different workload scenarios and analyze their real-time and reliability performance. A contribution of WARP is an intermediary language to specify a "smart" data plane that allows a richer set of behaviors, including the ability of multiple real-time flows to share the same entry for improved performance.

The closest related work is Recorp [4], which proposes a policy-based approach that, similar to WARP, can also share entries across multiple flows. WARP improves upon Recorp in several important ways: First, WARP's DSL can specify more expressive programs than Recorp policies. As a consequence, WARP can provide better performance under more varied workloads than Recorp. Second, Recorp is only suitable for IIoT applications whose workloads are fixed and known apriori since it may take up to a minute to construct a Recorp policy. In contrast, WARP can synthesize and execute programs on-the-fly within 10 ms slots on embedded platforms. Finally, WARP includes a new decentralized control plan that can adapt effectively to changes in topology and workload.

Control plane: A common critique of approaches like WirelessHART is that their centralized design leads to networks that cannot adapt effectively to dynamics. Indeed, this is problematic since any workload or topology changes requires creating a new super-frame and disseminating it to all nodes. This process can be particularly expensive when the super-frame is large and does not fit within a single packet. Four broad approaches — incremental scheduling, autonomous scheduling, distributed scheduling, and

flooding-based approaches — have been proposed to address this issue. REACT [15] reduces the control overhead by constructing schedules incrementally and only disseminating the differences between the current and the updated schedule. However, our experiments show that REACT still needs to disseminate many packets when the workload or topology changes significantly. Autonomous scheduling approaches (see [11] for a survey) build super-frames using the (partial) routing information already present at a node. However, the increased agility comes at the cost of real-time performance, which may not be acceptable for all IIoT applications. An elegant alternative is to avoid complex control planes altogether by building systems that do not depend on the topology (e.g., Blink [25] or Low-power Wireless Bus [12]). In contrast to these approaches, WARP maintains the observability of network operations while decentralizing the control plane to support both topology and workload changes efficiently.

Two recent decentralized approaches that support real-time performance have been recently proposed: FD-PaS [24] and DistributedHART [20]. FD-PaS is designed to efficiently change the flows' rates in response to external disturbances by piggybacking rate updates in the packet acknowledgments of a flow. DistributedHART uses a node-level scheduling where each node schedules its transmissions locally and online through a time window allocation. However, FD-PaS does not consider topology changes, and such changes can trigger expensive network-wide changes in DistributedHART. We have developed a control plane based on the observation that nodes can independently construct the same WARP program when they share the same workload information. Table 1 compares and contrasts the features of WARP and related protocols.

Program Synthesis: Program synthesis has been used to synthesize network updates, routing tables, and network policies from high-level specifications (e.g., [10]). To the best of our knowledge, we are the first to use such techniques to specify and synthesize programs that control the data plane of wireless networks. Program synthesis commonly relies on powerful SMT solvers, machines with plentiful resources, and is applied in scenarios where it is acceptable to spend minutes to hours to synthesize programs. We target embedded devices that must synthesize programs on-the-fly within 10 ms slots. More importantly, the vast majority of ongoing work focuses on verifying safety and deterministic properties. We developed new techniques to analyze the reliability of flows subject to probabilistic packet losses due to variations in link quality. Specifically, we identified a subset of WARP DSL that is expressive enough to provide significant performance gains while remaining sufficiently simple to enable efficient synthesis and analysis.

3 SYSTEM MODELS

Network Architecture: A network is comprised of a *network manager*, a *base station*, and up to a hundred *nodes*. WARP is best-suited for applications that require high data rates and have a backbone of grid-powered nodes to carry this traffic (e.g., [2, 6]). A *network manager* is a resource-rich machine that is connected over a wired connection to the base station. The *nodes* form a wireless mesh network that we model as a graph $G(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ and $\mathcal{E}$ represent the devices (including the base station) and wireless links. The network manager constructs a minimum spanning tree that has the base station as its root. The route of a flow from A to B (both distinct from the base station) is obtained by concatenating the route from A to the base station and from the base station to B. We adopt this model for consistency with prior work on WirelessHART, but WARP also supports peer-to-peer routing without modification. We categorize flows as *upstream* or *downstream* depending on whether they forward data to or from the base station, respectively.

Workload Model: The traffic of a WARP network is specified as a set of real-time flows $\mathcal{F} = \{F_0, F_1, \ldots, F_N\}$. A *real-time flow* F_i is a fixed route Γ_i that packets traverse through the network and has several associated attributes described next. In the wider network community, flows are streams of packets; for real-time contexts, flows are sequences of *independent* packets generated periodically at the source and must be delivered to the destination by the deadline. A flow's packets usually carry sensor data or actuation commands. WARP allows flows to be added, removed, or modified during the network's operation. This capability is useful for applications that dynamically reconfigure the sensors they use or change their workload in response to external events.

For each flow F_i, a packet is generated on the source node with a period of P_i and phase ϕ_i. The k^{th} instance of flow $F_i - J_{i,k} -$ is released at time $r_{i,k} = \phi_i + k * P_i$ and has an absolute deadline $d_{i,k} = r_{i,k} + D_i$. We assume that $D_i \leq P_i$, which implies that only one instance of a flow is released at any time. For clarity, we will discuss the algorithms in terms of flows rather than instances.

Physical Layer: At the physical layer, WARP may use either IEEE 802.15.4, which provides 250 kbps, or the more recent IEEE 802.15.4a, which supports up to 27.24 Mbps. Both physical layers operate in the 2.4 GHz band. Channel hopping is used to improve reliability. 802.15.4 supports frequency hopping over 16 channels. 802.15.4a provides further protection against interference by using UWB techniques. We configured the DWM1001 radios to use three orthogonal channels, as described in [8].

Entries, Pushes, and Pulls: WARP programs use both time division and multiple channels. We refer to a slot and channel pair as *an entry*. Programs may include a single push or a single pull within an entry of 10 ms. A push(F_i, #ch) involves the sender transmitting F_i's data and the receiver replying with an acknowledgment, both transmissions using channel #ch. A push fails if the acknowledgment is not received by the end of the slot. A pull(F_i, #ch) involves the receiver requesting F_i's data and the sender replying with the flow's data on channel #ch. A pull fails if the data is not received by the end of the slot. A program may handle failed pushes or pulls by repeating the action several times to achieve the desired reliability. We will refer to the node initiating the communication of a push or a pull as the *coordinator* and the responding node as the *follower*.

Constraints: A WARP program must meet the following constraints:

- *Transmission constraints:* A node transmits or receives at most once per slot.
- *Channel constraints:* At most one transmission is performed in an entry to avoid intra-network interference. Additionally, a node must perform consecutive transmissions on different channels.
- *Forwarding constraints:* Each flow has at most one link that is active in any slot. The first link on the flow's route is activated at release time; subsequent links are activated as the previous ones complete relaying the packets.

- *Real-time and reliability constraint:* The likelihood that the packets of a flow reach their destination before their deadline exceeds an end-to-end probability target.

4 WARP

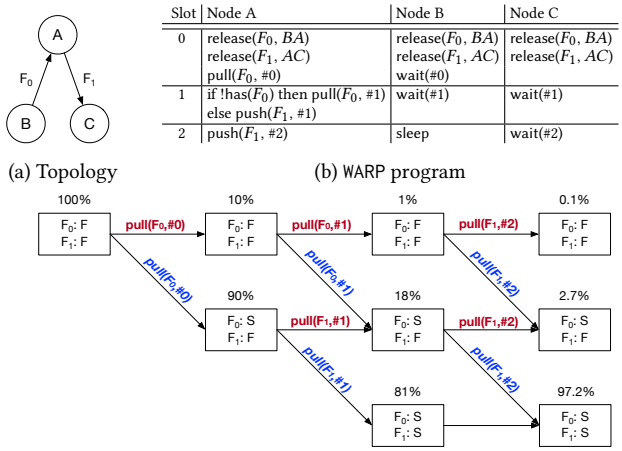
Layer Responsibilities: WARP is a practical and effective solution for IIoT applications that require high-data rates, real-time, and reliable communication in dynamic wireless environments. The data plane handles packet losses due to probabilistic and variable links, whereas the control plane handles topology and workload changes. A central aspect of WARP is that each node executes a stateful program that controls when it transmits the packets of real-time flows. Programs specified using the DSL described in Section 4.1 can express more intricate actions than possible using traditional schedules. A node's program tracks the success or failure of a flows' transmissions and, based on this state, it executes different transmissions depending on the pattern of observed packet losses. This approach improves network performance significantly. In Section 4.2, we propose techniques to analyze a program's execution in the presence of packet losses and introduce the TLR model to specify how the quality of links vary.

Data Plane: WARP incorporates a scalable program synthesis procedure that turns a high-level specification of the workload and topology into a program that is executed by the network. The synthesis procedure incorporates a generator and an estimator described in Section 4.3.1 and 4.3.2, respectively. The estimator reasons about the possible execution paths of a program under all failure scenarios and determines the likelihood of each reachable state (under the assumptions of the TLR model). Based on this information, the generator heuristically increases the number of flows that execute in each slot across multiple possible failure scenarios. The synthesized programs' execution guarantees that the likelihood of each flow's packets reaching their destination by its deadline exceeds a user-specified threshold. This guarantee holds only when the network's behavior is consistent with the TLR model: (1) consecutive transmissions are independent and (2) the chance of successful transmission can vary, but it always exceeds a minimum threshold m. When the network deviates from the TLR model, either new routes that use links that conform to the TLR model should be selected or the value of m should be lowered. The synthesis procedure may fail to generate a program when the workload is close to the network's capacity. In this case, the workload has to be adjusted manually by the network operator or automatically using rate control mechanisms. As discussed next, our goal is to run the synthesis on embedded devices. Therefore, we focus on synthesis techniques that use light-weight heuristics.

Control Plane: Similar to WirelessHART, WARP manages the topology and workload information centrally. However, we decentralize the synthesis and adaption to workload changes, as detailed in Section 4.4. In contrast to scheduling approaches, WARP does not construct an explicit schedule but instead runs the synthesis procedure on each node on-the-fly to determine its actions in a slot. All the nodes will construct the same program as long as they share the same workload and topology information. The decentralized control plane handles workload and topology changes by disseminating only the updated workload or topology information. This results in a more agile control plane since transmitting workload and topology updates requires fewer bytes than complete programs.

```
fragment      := release-blk; action_blk; drop_blk
release_blk   := release(flow, link); | release_blk
drop_blk      := drop(flow); | drop_blk
action_blk    := action | if-statement
if-statement  := if bool-expr then action else-part
else-part     := else action_blk | ϵ
bool-expr     := has(flow) | !has(flow)
action        := pull(flow, channel) | push(flow, channel) |
                 wait(channel) | sleep
```

Figure 1: EBNF rules of a WARP fragment



(c) Node A's execution DAG

Figure 2: Flows F_0 and F_1 forward packets from B to A and from A to C according to WARP program in Figure 2b. Due to packet losses, a program's execution may follow different paths and those paths are summarized in its execution DAG, shown in Figure 2c.

4.1 The WARP Domain-Specific Language

4.1.1 An Example. Before formalizing the DSL, we will introduce WARP programs informally through an example that implements a new mechanism to arbitrate access to a shared entry among multiple flows. Consider the queue of an intermediary node in the routing tree that includes the packets of different flows. Some flows go towards the base station, others away from it. These queues grow as the workload increases until one node fails to forward a packet by the flow's deadline. To support high data-rate flows, WARP generates `node programs` that coordinate the interactions of a node with its neighbors to forward its queued packets efficiently.

To illustrate the specifics of this mechanism consider Figure 2b. The included program forwards F_0's data from B to A and F_1's data from A to C. Both flows release a packet in slot 0. Each node executes a `node program` organized as an array of fragments, each fragment specifying the actions that the node performs in a slot. The table in Figure 2b shows each `node program` as a column and its fragments as the column's entries. Flows F_0 and F_1 each release a packet in slot 0 as reflected by the releases included in the slot

0 fragment of all nodes. The releases create a globally consistent view of what flows are released in the network. In slot 0, A also executes a pull(F_0, #0) to request F_0's data from B over channel #0. Node A's pull is paired with a wait(#0) executed by B, which instructs B to listen for an incoming request on channel #0. When B receives A's request, it will reply with F_0's data.

The most visible departure from schedules is the inclusion of conditional statements in programs, as shown in the fragment executed by A in slot 1. The execution state of a node program is updated to record the outcome of push and pull actions. The has instruction may be used to query a program's state. When has is combined with if statements, the node program may take different actions depending on its execution state. After completing slot 0, F_0's packet may be at either A or B, depending on whether pull(F_0, #0) has succeeded (S) or failed (F). Node A queries its state to distinguish between these two cases. If A does not have F_0's data, then A requests F_0 again; otherwise, A proceeds to transmit F_1's data using a push. In this slot, both B and C wait to respond to A's pull or push on channel #1. This example illustrates how A can coordinate the transmissions of F_0 and F_1 across different execution paths, allowing them to share slot 1 and channel #1, without introducing transmission conflicts or resorting to carrier sense. WARP programs use this mechanism pervasively, and our experiments show that it can improve the throughput by 2.6 times over scheduling approaches.

4.1.2 The DSL. Next, we formalize an intermediary DSL to specify the behavior of WARP's data plane. The DSL formally specifies the actions that WARP programs can perform in a slot and their associated run-time behavior. We opted to use a DSL to specify programs as programs should be a familiar abstraction and the analysis techniques introduced later have similarities to symbolic execution techniques used to analyze computer programs [3]. While the DSL is not intended to be used directly by network operators or developers, developers can be used it for manually analyzing network programs. The DSL is primarily intended for automated synthesis and analysis tools.

A network operator specifies the workload of a WARP network as a collection of real-time flows. The conceptual model used to program the network is a distributed collection of nodes that operate synchronously to forward the specified flows. Consistent with this node-centric perspective, a program is structured as a collection of node programs, each executed by a node. A node program is an array of fragments indexed by the slot in which the fragment is executed. The program is executed cyclically such that when a node reaches the last instruction of a node program, the node restarts executing it from the beginning. A network manager maintains the workload and constructs a routing table to forward the data of all flows. This information is replicated on each node, and node programs can access it.

The building blocks of the DSL are *global events* and *local actions*. The release(F_i, l) and drop(F_i) are global events that control when link l on F_i's route is activate. A global event is scheduled in slot t by including a release(F_i, l) or a drop(F_i) in the fragment associated with slot t in the node programs of all nodes. When a node program's execution reaches slot t, the scheduled global event is executed synchronously by all nodes without requiring communication. Note that a globally consistent view is necessary to determine whether the actions in a slot may conflict. Flow F_i can transmit over link l only in the slots ranging from the slot in which F_i is released to the slot in which F_i is dropped. Since releases and drops are global events, all nodes know the interval when the packets of F_i *may* be forwarded over link l, but not all of them know the specific slots in this interval that will be used to transmit F_i's packets[1]. The supported actions are push, pull, wait, and sleep. A fragment may include multiple actions using if-statements, but only one action can be executed at run-time. The execution state of a node program may change in response to executing a push or a pull. The effect of an action is local, affecting only the node executing the action. A node program may include a has instruction to query a flow's status and conditional statements to allow actions to be executed conditionally on its execution state.

Let us define how a node program's execution state is updated when it is executed by a node A. A maintains a dictionary M that maps a flow to its status. A flow's status is updated according to the state machine shown in Figure 3. When A executes release(F_i, AB), it adds a mapping $M[F_i]$ = RELEASED to indicate that F_i is released and AB is its active link. It is possible that F_i's packet may not be in A's receive queue if it was dropped on a previous hop. In this case, a packet containing an error indicating that F_i's packet was dropped is generated and forwarded by A as if it was the original packet of the flow. F_i's entry is removed from M when A executes drop(F_i) and F_i is considered to be COMPLETED. A flow's status is updated depending on the outcome of push and pull.

When A is the source of an outgoing flow F_i, it has to forward its packets to the next hop using pushes. Let B be this next hop, determined by A inspecting its local copy of the routing table. The execution of push(F_i, #ch) entails A sending F_i's data on channel #ch and waiting for an acknowledgment. The fragment that B executes in the same slot must include a wait(#ch) to instruct B to listen for an incoming request on channel #ch. If node B receives the pushed packet, it sends an acknowledgment to A. Node A knows whether the push(F_i, #ch) was successful if it receives the acknowledgment by the end of the slot. In this case, the $M[F_i]$ is set to SUCCESSFUL; otherwise, $M[F_i]$ remains RELEASED.

When A is the destination of an incoming flow F_i, it has to obtain its data from the previous hop using pull actions. When release(F_i, BA) is executed by A, F_i's packet is at the previous hop B. In a slot, the execution of pull(F_i, #ch) by A to request F_i's data over channel #ch is paired with the execution of wait(#ch) by B. Node A can determine whether the pull(F_i, #ch) was successful depending on whether it receives the request data from B by the end of the slot. If A receives the data, A updates $M[F_i]$ to be SUCCESSFUL; otherwise, the flow's status remains RELEASED.

The grammar of a fragment is included in Figure 1. A fragment has three blocks: a *release block*, an *action block*, and a *drop block*. The release block may include one or more releases to start forwarding packets over a flow's active link. Similarly, the drop block may include one or more drops to indicate that a flow's active link will no longer be executed for the remainder of the program. The action block may include one of push, pull, wait, or sleep in the

[1]The specific slots (and fragment) in which releases and drops are included is determined during synthesis based on the network's topology and workload.

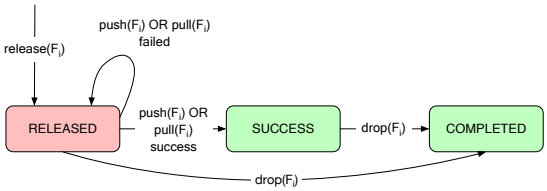
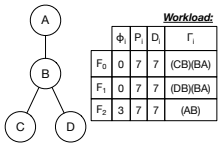


Figure 3: State machine describing how F_i's status is updated. The has(F_i) is false when F_i's status is RELEASED and true otherwise.



Workload:

	ϕ_i	P_i	D_i	Γ_i
F_0	0	7	7	(CB)(BA)
F_1	0	7	7	(DB)(BA)
F_2	3	7	7	(AB)

(a) Topology

Slot →	0	1	2	3	4	5	6
F_0	(CB)	(CB)	(BA)	(BA)	(BA)	-	-
F_1	(DB)	(DB)	(DB)	(BA)	(BA)	(BA)	-
F_2				(AB)	(AB)	(AB)	(AB)

(b) Slots when each link of a flow is active

Slot	Node A	Node B	Node C	Node D
0	release(F_0, CB) release(F_1, DB) sleep	release(F_0, CB) release(F_1, DB) pull(F_0, #0)	release(F_0, CB) release(F_1, DB) wait(#0)	release(F_0, CB) release(F_1, DB) sleep
1	sleep drop(F_0)	if !has(F_0) then pull(F_0, #1) else pull(F_1, #1) drop(F_0)	wait(#1) drop(F_0)	 wait(#1) drop(F_0)
2	release(F_0, BA) sleep drop(F_1)	release(F_0, BA) pull(F_1, #2) drop(F_1)	release(F_0, BA) sleep drop(F_1)	release(F_0, BA) wait(#2) drop(F_1)
3	release(F_1, BA) release(F_2, AB) pull(F_0, #0)	release(F_1, BA) release(F_2, AB) wait(#0)	release(F_1, BA) release(F_2, AB) sleep	release(F_1, BA) release(F_2, AB) sleep
4	if !has(F_0) then pull(F_0, #1) else pull(F_1, #1) drop(F_0)	wait(#1) drop(F_0)	sleep drop(F_0)	sleep drop(F_0)
5	if !has(F_1) then pull(F_1, #2) else push(F_2, #1) drop(F_1)	wait(#1) drop(F_1)	sleep drop(F_1)	sleep drop(F_1)
6	push(F_2, #2) drop(F_2)	wait(#2) drop(F_2)	sleep drop(F_2)	sleep drop(F_2)

(c) Program forwarding the packets of F_0, F_1, and F_2

Figure 4: Example program forwarding the data of three flows

base case. We allow for the inclusion of conditional if-then-else statements. As part of the condition, a program may use has(F_i) or its negation !has(F_i) to determine whether node A has F_i. The instruction has(F_i) is true when $M[F_i]$ is SUCCESSFUL or COMPLETED. In any slot, we constrain a node to execute at most one action per slot to guarantee its completion by the end of the slot.

In the following, we consider a more involved example that forwards the packets of three multi-hop flows. There are many possible programs to forward the flows, each having different performance trade-offs. Figure 4 shows the program generated by our synthesis procedure. The table in Figure 4b shows the slots in which each link of a flow is activate. All the nodes share this view in the network. For example, F_0 forwards packets over Γ_0 ={(CB), (BA)}. Link (CB) and (BA) are active in slots 0 – 1 and 2 – 4. Since a flow has at most one active link at a time, the two intervals do not overlap. The fragments use different combinations of pull and push actions in conjunction with conditionals to determine when and what packets to transmit. In contrast to shared slots in scheduling approaches, WARP avoids carrier sense by using conditional logic to determine which node may transmit.

4.2 Analyzing WARP Programs

A key challenge is to reason about the execution of WARP programs in the presence of stochastic packet losses. We want to estimate the likelihood of property β_i that a packet of a flow F_i reaches the destination by F_i's deadline. As a starting point, let us model the quality of all links as Bernoulli variables with known and fixed chances of success. We assume that a push or a pull performed over a link has a 90% chance of success for concreteness. To compute the likelihood of β_i, it suffices to consider all possible packet loss scenarios and sum the likelihood of all scenarios in which the property β_i holds. We use the notation $\Pi_t(s)$ to indicate the likelihood of reaching state s after executing t slots. A brute-force approach to accomplish this goal is to construct an *execution DAG* that captures all possible execution paths through the program under different failure scenarios. The execution DAGs introduced here will be later refined to support the analysis of synthesized programs efficiently.

Node A's execution DAG could be built as follows (see Figure 2c). The symbols S and F indicate whether a pull or a push executed by a flow F_i was successful or failed (i.e., when has(F_i) is true or false). Flows F_0 and F_1 are released in slot 0. Accordingly, the initial state FF indicates that A does not have the packets of either F_0 or F_1. The likelihood of reaching FF and FS after executing the pull(F_0, #0) in slot 0 is $\Pi_1(\text{FF}) = 10\% \times \Pi_0(\text{FF}) = 10\%$ and $\Pi_1(\text{SF}) = 90\% \times \Pi_0(\text{FF}) = 90\%$. It is important to note the difference in the information available during synthesis/analysis and run-time execution. During analysis and synthesis, we do not know whether A's execution state is FF or SF; we only know their likelihoods. However, at run-time, A knows its state precisely as it can observe the outcome of its actions.

In slot 1, the actions that A performs depend on its current state. In the state FF, when A does not have F_0's data, it will execute pull(F_0, #1) to request the data from F_0 again. Depending on the pull's outcome, the system transitions to either FF or SF. The state FF is reached when both pulls executed in slot 0 and 1 have failed. The state SF is reached over two paths: when pull(F_0, #0) failed in slot 0 but succeeded in slot 1 and when pull(F_0, #1) in slot 0 has succeeded and push(F_1, #1) in slot 1 failed. Accordingly, the likelihood of reaching SF by the end of slot 2 is $\Pi_2(\text{SF}) = 90\% * \Pi_1(\text{FF}) + 10\% * \Pi_1(\text{SF}) = 18\%$. The execution of the remaining instructions results in the shown execution DAG. The likelihood that A receives F_0 and F_1 by the end of the program are $E_3^0 = \Pi_3(\text{SF}) + \Pi_3(\text{SF}) = 97.2\% + 2.7\% = 99\%$ and $E_3^1 = \Pi_3(\text{SF}) = 97.2\%$.

A significant limitation of the above approach is that the obtained results are applicable only when *all* links have a quality of *exactly* 90%. What happens if the link quality increases to 95%? How about when links have different link qualities? It is hard to extrapolate a network's behavior from point estimates when each link's quality can vary from slot to slot. To broaden the applicability of this approach, we need a more realistic reliability model.

We propose the Threshold Link Reliability (TLR) model, which is both simple and realistic. TLR models the likelihood that an action (i.e., a push or pull) of flow F_i (including both exchanges of data between sender and receiver) is successful as a Bernoulli variable $LQ_i(t)$. We assume that consecutive pushes or pulls performed over the same or different links are independent. Empirical studies suggest that this property holds when channel hopping is used [14,

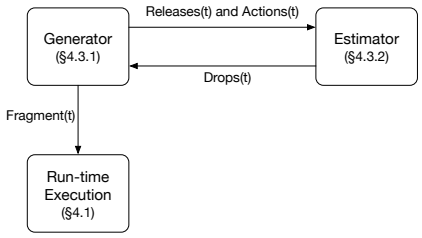


Figure 5: Data plane components

17]. TLR has only one parameter – the minimum PDR m, which lower bounds the values of $LQ_i(t)$ such that $m \leq LQ_i(t)\ \forall i \in \mathcal{F}, t \in \mathbb{N}$. A strength of TLR is that aside from the lower bound m on link quality, we make *no assumptions regarding how the quality of a link varies from slot to slot.* This characteristic makes TLR widely applicable and may be integrated with the current practices for deploying IIoT wireless networks. For example, Emerson engineers suggest that WirelessHART networks should be deployed to provide a minimum PDR between 60–70% [19]. Thus, we set m to 70%.

The problem of ensuring that the likelihood of a flow's F_i packets are delivered to the destination over multiple hops before F_i's deadline exceeds the probability target T_i can be mapped onto single-hop requirements. Let L_i be a lower bound on the likelihood that F_i is delivered at each hop. Then, the likelihood of delivering the packet end-to-end must be at least $\Pi_{h=0}^{\text{num-hops}(F_i)} L_i = L_i^{\text{num-hops}(F_i)}$, where num-hops($F_i$) is the number of hops on F_i's route. Therefore, if at each hop a flow's packet is delivered with a likelihood of at least $L_i = T_i^{\frac{1}{\text{num-hops}(F_i)}}$, then the end-to-end requirement is also met. Accordingly, we will refer to L_i as the local reliability target. We will show that when the structure of programs is appropriately constrained (as described in Section 4.3), a flow's local reliability estimated using execution DAGs with $LQ_l = m$ is a safe lower bound on the flow's reliability when $LQ_l \geq m$. WARP relies on execution DAGs to determine the states that programs may reach due to packet losses, optimize the synthesized programs across multiple execution paths, and ensure that the real-time and reliability constraints are met.

4.3 Synthesizing WARP Programs

WARP improves a network's throughput by enabling the transmissions of multiple flows to share the same entry. The synthesis procedure is modeled as a sequential decision problem that involves an estimator and a generator (see Figure 5). In a slot t, the generator emits a partial fragment per node that specifies the release and action blocks that the node will execute. The generator ensures that regardless of the possible execution paths that a program may take due to packet losses, its actions satisfy the transmission, channel, and forwarding constraints (see Section 3). Next, the estimator updates the execution DAG of each node to account for the generated partial fragment. The estimator ensures that the real-time and reliability constraint is satisfied by evaluating when a `node program` includes sufficient pushes or pulls to deliver a flow's packets over its active link with a likelihood higher than its local probability target. When this occurs, the generator emits a drop block that includes a drop for each flow that meets its local probability target. The complete fragment including the release, action, and drop blocks are then executed at run-time consistent with the semantics described in Section 4.1.

Before considering the generator's and estimator's details, let us introduce a heuristic that will be used during synthesis. We categorize flows as *upstream* or *downstream* depending on whether they forward data to or from the base station, respectively. During synthesis, we will use pulls for the transmissions of upstream flows and pushes for the transmissions of downstream flows. To build an intuition behind this heuristic, consider multiple upstream flows forwarding data towards the base station. An intermediary node in the routing tree may have several children with data to be forwarded in their queues about the same time. The intermediary node may coordinate data collection from those children using pulls since it can observe whether it has received a child's data successfully. Multiple slots are allocated and shared by the children to forward their data. Notably, over different execution paths, the children may use different numbers of retransmissions, but sufficient slots are allocated to guarantee that the likelihood that children forward their packets exceeds the local target. The opposite scenario occurs for downstream flows where an intermediary has multiple packets for its children and coordinates their transmission using pushes. We remind the reader that a push's coordinator is the sender, whereas a pull's coordinator is the receiver.

4.3.1 Generator. The generator enforces the invariant that once a flow starts its execution, it cannot be preempted (by another flow). The generator distinguishes between two types of flows. A flow is *ready* if it is released, but it has not begun execution yet. Once the flow begins execution, it is considered to be *executing*. The generator maintains a global ready queue that includes the ready flows and an exec queue per node that include the node's executing flows. The ready queue is ordered by the priority of flows. In a slot t, the queues are updated as follows:

- A flow F_i is released when $mod(t - \phi_i, P_i) = 0$, and it is added to the ready queue. The first link of the flow becomes active and considered for potential execution.
- The generator considers each flow F_i in the ready priority queue and adds F_i to the exec queue of its coordinator if it does not conflict with any of the other flows that are already executing.
- The estimator is queried to obtain a lower-bound on the likelihood that F_i has been forwarded to its next hop. If the lower-bound exceeds the local probability target, then F_i is completed and removed from exec. If F_i has not reached its destination, then the next link is activated and F_i is added to the ready queue. If F_i's deadline is reached before its packets reach the destination, then the synthesis procedure fails and returns an error. This case can be avoided by running admission control (see Section 4.4).

To check for transmission conflicts, each node maintains a conflict-list that includes a list of nodes whose activities it will coordinate[2]. Consider a flow F_i that has an active link AB where A is the coordinator and B the follower. The flow F_i may be added without conflict to A's exec queue when A is not in the conflict-list of any other node in $\mathcal{N} \setminus \{A\}$ and B is not in the conflict-list of any other node

[2]Note that the conflict-list may include a node multiple times.

in $\mathcal{N} \setminus \{A, B\}$. In this is the case, F_i is added to A's exec queue and B is added to A's conflict-list. When a flow F_i completes its execution, B is removed from its conflict-list. By choosing appropriate data structures, we can construct a generator that runs in $O(|\mathcal{F}|)$.

The generator produces the next fragment of a `node program` based on the node's exec queue. A fragment includes a release block, an action block, and a drop block. The release block includes all the flows that have been released in the current slot. The drop block includes all flows that the estimator determined to have met their local probability target. Note that since release and drop are global events, they are included in all nodes' fragments, so they have a consistent view of when flows are released and dropped. If the exec queue is empty, an action block that includes a single sleep is generated. When there is a single flow in exec, either a push or a pull is generated depending on whether the flow is downstream or upstream, respectively. The action will use a channel that is assigned by the coordinator. Each coordinator maintains a counter that is initialized with a common random number and is incremented each time it executes a push or a pull. The assigned channel is the modulus of the counter and the maximum number of channels. When exec has two or more flows, the action block is a sequence of if-statements that enforce the invariant: a flow with index k in the exec queue will be executed only if all flows with index $k' < k$ have been successful. For example, the action fragment associated with exec = $[F_0, F_1, F_2]$ is shown on top of Figure 6. In this case, push(F_0, #0) is executed only if has(F_0) is false. The pull(F_1, #0) is executed only if has(F_0) is true and has(F_1) is false consistent with servicing F_1 only if F_0 is successful. Finally, pull(F_2, #0) is executed only if F_0 and F_1 have been successful.

4.3.2 Estimator. The estimator computes lower bounds on the likelihood that a flow delivered its packet over the currently active link. An arbitrary node A runs an estimator that includes N execution DAGs, each execution DAG tracking a node's state. In each slot, the generator provides the estimator N fragments (one fragment per node), specifying the actions each node will execute in that slot. Node A updates a node's execution DAG based on that node's fragment. Each execution DAG is updated independently and identically and, in the following, we will focus on how an arbitrary execution DAG is updated. The execution DAG can be thought of as a Markov Chain (MC) that unfolds over time as instructions are executed. However, unlike traditional MCs, the number of states grows and shrinks as flows start and complete their execution.

The estimator interprets a fragment in three stages consistent with the structure of a fragment that has a release block, an action block, and a drop block. Accordingly, we group the states of the execution DAG into stages corresponding to the states obtained after interpreting the releases, actions, and drops. We use the ordered tuples (t, R), (t, A), (t, D) to refer to the stage after interpreting each block of the fragment in slot t. Each stage will be interpreted as the matrix product between the current state $\Pi_{t,PC}$ (where $PC \in \{R, A, D\}$) and a transition matrix that encodes the block's semantics. The size of the vector $\Pi_{t,PC}$ is the number of instances in the exec queue in slot t. The element $\Pi_{t,PC}[F_i]$ is F_i's status and the result of has(F_i) (see Figure 3).

Consider the case when exec = $\{F_0, F_1, F_2\}$. Since the generator enforces the invariant that a flow F_i cannot be executed until all

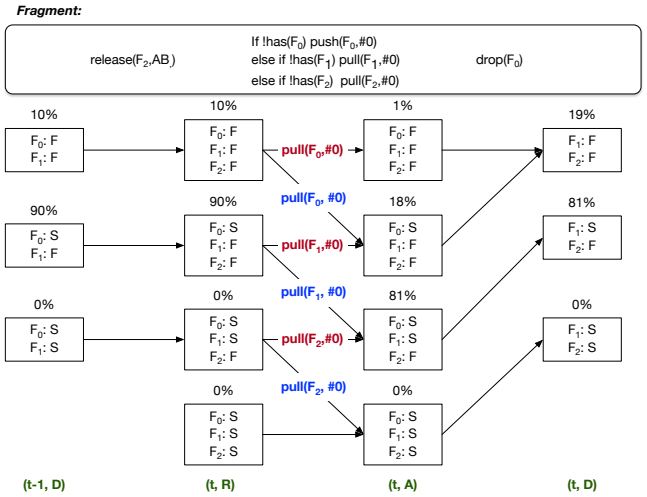


Figure 6: The fragment executed includes a release, an action-block, and a drop shown at the top of the figure. The bottom of the figure includes the execution DAG where boxes represent the states and the lines the transitions. States in the same column are in the same stage with the index shown below in bold green font.

flows that have started their execution before F_i are successful, the possible states are FFF, SFF, SSF, SSS. In general, the states of a stage can be represented as strings given by the regular expression $\mathsf{S}^k\mathsf{F}^{e-k}$, where S^k is a sequence of k successes, and F^{e-k} is a sequence of $e-k$ failures, where e is the maximum number of states in the stage. The maximum number of states in a stage e can be related to exec queue's size: e = |exec| + 1. In the above example, there are three flows and four associated states.

To illustrate how the execution DAG is managed, let us consider the execution of the fragment shown in Figure 6. The example starts in an arbitrary slot $t-1$ when exec ={F_0, F_1 }. The possible states of the stage are FF, SF, and SS. We assume that the initial likelihood is $\Pi_{t,D} = [0.1, 0.9, 0]$. We will describe the actions as transitions between a current and a next stage in the execution DAG.

The first instruction in the fragment is release(F_2, AB). F_2 is added to the end of the exec queue, and its current status is failure (F) since it was just released. A release can be interpreted as string concatenation, where each state in the current stage is considered and a next state is created by appending a F to the end the string to account for F_2's initial state. An additional state is added to the next stage to account for the case when all flows are successful. In our example, the states after executing release are FFF, SFF, SSF, and SSS. Since the status of F_2 is failure the likelihood of the next states is: $\Pi_{t,R}(\mathsf{FFF}) = \Pi_{t-1,D}(\mathsf{FF}) = 0.1$, $\Pi_{t,R}(\mathsf{SFF}) = \Pi_{t-1,D}(\mathsf{SF}) = 0.9$, and $\Pi_{t,R}(\mathsf{SSF}) = \Pi_{t-1,D}(\mathsf{SS}) = 0$. These equations can be expressed as the product of $\Pi_{t,D}$ and a release matrix R:

$$\Pi_{t,A} = \Pi_{t,D} \times \begin{array}{c} \begin{matrix} \mathsf{FFF} & \mathsf{SFF} & \mathsf{SSF} & \mathsf{SSS} \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} \mathsf{FF} \\ \mathsf{SF} \\ \mathsf{SS} \end{matrix} = [0.1, 0.9, 0, 0]$$

In general, a release is interpreted as the product of the current state and a release matrix R of size $e \times (e+1)$. All rows r of a release matrix R have a single non-zero entry $R[r,r] = 1$ for $0 \leq r < e$.

Next, we consider the action block of the fragment and generate a transition matrix A that accounts for the entire block's execution. Let LQ_t be the chance that the packets associated with the selected action are transmitted successfully. LQ_t is 90% in our example. Flow F_0 is executed only in state FFF since it is the only state in which !has(F_0) is true. If the action fails, the system remains in the same with the likelihood $1-LQ_t$; otherwise, it transitions to the state SFF with likelihood LQ_t. Flow F_1 is executed only in state SFF when has(F_0) is false (i.e., F_0's status is success) and !has(F_1) is false. Upon its execution, the system transitions to state SFF with likelihood $1-LQ_t$ and transitions to state SSF with likelihood LQ_t. Similarly, F_2 is executed in state SSF, causing the system to remain in state SSF with likelihood $1-LQ_t$ or transition to SSS with likelihood LQ_t. Finally, no flow is executed in state SSS and the system remains in this state with a likelihood of 1. The execution of the actions can be accounted for according to the following matrix multiplication:

$$\Pi_{t,D} = \Pi_{t,R} \times \begin{array}{c} \begin{matrix} \text{FFF} & \text{SFF} & \text{SSF} & \text{SSS} \end{matrix} \\ \begin{pmatrix} 1-LQ_t & LQ_t & 0 & 0 \\ 0 & 1-LQ_t & LQ_t & 0 \\ 0 & 0 & 1-LQ_t & LQ_t \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} \text{FFF} \\ \text{SFF} \\ \text{SSF} \\ \text{SSS} \end{matrix}$$

$$= [0.01, 0.18, 0.81, 0]$$

In general, the actions block can be represented as a matrix A of size $e \times e$. All rows r such that $r < e-1$ have two non-zero entries: $A[r,r] = 1-m$ and $A[r,r+1] = m$. The last row has all elements zero except the last column, which is 1.

A drop(F_0) indicates that F_0 has finished executing and is located at the head of the exec queue. Similar to the release instruction, the drop can also be treated as a string operation that considers each state and generates a new state by removing the left-most character. This results in the first two states having the same suffix and their likelihoods are added when generating the single combined new state. In our example, removing the first character of maps states FFF and FSS on the same state FF. Thus, $\Pi_{t,D}(\text{FF}) = \Pi_{t,A}(\text{FFF}) + \Pi_{t,A}(\text{SFF})$. For the rest of the states, $\Pi_{t,D}(\text{SF}) = \Pi_{t,A}(\text{SSF})$ and $\Pi_{t,D}(\text{SS}) = \Pi_{t,A}(\text{SSS})$. These equations can be specified as the following matrix product:

$$\Pi_{t,R} = \Pi_{t,D} \times \begin{array}{c} \begin{matrix} \text{FF} & \text{SF} & \text{SS} \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} \text{FFF} \\ \text{SFF} \\ \text{SSF} \\ \text{SSS} \end{matrix} = [0.19, 0.81, 0]$$

In general, interpreting a drop involves multiplying the current state with a matrix D. Each row of D has a single non-zero entry that is one. The first row the non-zero entry is $D[1,1] = 1$ and in the remaining rows r ($1 \leq r < e$) the non-zero entry is $D[r, r-1] = 1$.

Let E_t^i be the likelihood that flow F_i forwarded its packets by slot t. The local reliability of a flow F_i may then computed as the sum of all the states where the flow was successful. In our example, $E_t^0 = \Pi_{t,A}[\text{SFF}] + \Pi_{t,A}[\text{SSF}] + \Pi_{t,A}[\text{SSS}] = 0.99$.

The TLR model allows the quality of links to vary from slot to slot arbitrarily as long as it exceeds a minimum link quality m. The reliability of a flow F_i is a function $E_t^i(LQ_0, LQ_1, \ldots LQ_t)$ that depends on the LQ_t, which is likelihood that the the pull or push in slot t succeeds. We claim that a lower bound $\hat{E}_t^i(m)$ may be computed by setting LQ_t equal to the m in all matrices A that encode action blocks:

$$\hat{E}_t^i(m) = E_i^t(LQ_0 = m, LQ_1 = m, \ldots LQ_t = m) \leq E_i^t(LQ_0, LQ_1, \ldots LQ_t)$$

Theorem 4.1. *Consider a node A and a set of flows $\mathcal{F}_A = \{F_0, F_1, ..F_M\}$ such that A is as either a source or a destination of the F_i's active link ($F_i \in \mathcal{F}_A$). Let LQ_t be the likelihood that the push or pull performed in slot t succeeds such that $m \leq LQ_t \leq 1$ for all slots t ($t \in \mathbb{N}$) The local reliability $E_t^i(LQ_0, LQ_1, \ldots LQ_t)$ that F_i's packets are delivered over its active link is lower-bounded by $\hat{E}_t^i(m)$.*

Proof. Due to space limits, the proof is included in [5]. □

Due to the structure of the synthesized code, the execution DAGs can be managed very efficiently. The number of states in the execution DAG is |exec| + 1, each state requiring one floating-point per state. To control the storage and computational requirements, we have modified the generator to move an instance from the ready to the exec queue only if the exec queue's size is below a threshold. Experiments show that setting this threshold to four entries provides significant performance gains while introducing a small memory and computational overhead. Specifically, we require that a device maintain $5 \times N$ floating-points. A potential challenge with implementing the execution DAG using matrix multiplication is that floating-point operations tend to be slow or unsupported on embedded devices. Our implementation avoids matrix multiplication altogether and further reduces memory usage by converting multiplications into table look-ups. We divide the interval $[0, 1]$ of potential values into a small number of intervals. Since the transition matrix only depends on the parameter m of TLR, we cache the results of multiplying each interval with m. Our experiments indicate that caching a small number of values provides a good approximation of a flow's reliability.

4.4 Control Plane

WARP uses a control plane similar to REACT's [15] and, in the following, we will primarily highlight those differences. The unique aspect of WARP is that it does not disseminate programs. Instead, WARP disseminates the workload and topology changes, and nodes independently reconstruct a shared schedule. A workload change is initiated either by the network operator or by a node within the network, sending a request to the base station to add, remove, or modify a flow. A topology change is detected by the network manager based on the periodic health reports provided by nodes. In either case, WARP runs admission control to ensure that a feasible program may be generated. If a program is not synthesized successfully, then rate control techniques are applied to reduce the workload. Due to the efficiency of our synthesis procedure, the overhead of doing so is minimal. If a program is synthesized, then WARP disseminates the topology and workload changes to all

nodes which then synthesize the program. We use a broadcast tree mechanism proposed in REACT to disseminate the topology and/or workload changes.

Changes in workload and topology may be encoded and disseminated efficiently. The addition of a new flow requires disseminating its associated parameters: a flow identifier, phase, period, deadline, priority, and route. There are a small number of classes in most applications, each class having its temporal parameters and priority level. Accordingly, we can avoid disseminating the temporal and priority of a flow and simply disseminating the class's identifier for the flow. Similarly, rather than disseminating a route, it suffices to use a route identifier that refers to a route in the table maintained by the node. Accordingly, to add a flow, it suffices to transmit a total of 4 bytes – 2 for the identifier, 1 for the flow class, and 1 for the route identifier. A command to remove the flow only requires passing the identifier of the flow to be removed. Changes to the routing information can also be encoded efficiently. The operations that are allowed on the routing table is to add, duplicate, modify, or delete routes. A common case is for a link to change, requiring several routes to be modified. WARP handles this use case using a modify command that treats each route as a string and performs a search and replace operation. The key behind this approach's efficiency is that the operation is applied to all routes in the table.

5 EXPERIMENTS

Our experiments answer the following questions:

- Does WARP improve the throughput in typical IIoT workloads?
- Can WARP run on resource-constrained embedded devices?
- How agile is WARP's control plane to workload changes?
- Can WARP satisfy the real-time and reliability constraints and adapt to fluctuations in link quality?

We compare WARP against three baselines: CLLF, REACT, and Recorp. CLLF is a heuristic approach to building near-optimal schedules [23]. REACT improves the agility of control planes by constraining the construction of schedules such that they are easier to update in response to workload and topology changes. In response to a workload change, CLLF and REACT disseminate only the differences between the current and updated schedules. Recorp uses an SMT solver to constructs policies that can share slots across multiple flows. However, Recorp can take as long as a minute to construct policies and uses a centralized control plane. We considered two versions of WARP— WARP and WARP*. WARP's estimator uses matrix operations while WARP* converts matrix operations to table lookups to avoid floating-point operations. WARP*'s lookup table is configured to use 527 bytes to balance memory usage and performance. As the lookup table's size increases, the difference in the performance of WARP and WARP* decreases.

We set m = 70%, as suggested by Emerson's guide to deploying WirelessHART networks. In simulations, we set the probability of a successful transmission to equal m. The number of transmissions for all protocols is set to achieve 99% end-to-end reliability for all flows. Each experiment considers a different workload. However, a flow's period and deadline are equal, and its phase is zero in all workloads. Except for CLLF, which uses a heuristic to assign priorities, flow priorities are assigned such that flows with shorter deadlines have higher priority. Additionally, flows with longer routes have a higher priority and remaining ties are broken arbitrarily.

We quantified the performance of protocols using total throughput, synthesis time, control overhead, and consensus time. The total throughput is the maximum number of packets per second that flows can deliver without missing the real-time or reliability constraints. Synthesis time is the amount of time needed to synthesize programs or construct schedules. The control plane's agility is quantified by the number of bytes used for control overhead and the consensus time until all nodes update the program or schedule in response to a workload change.

5.1 Simulations

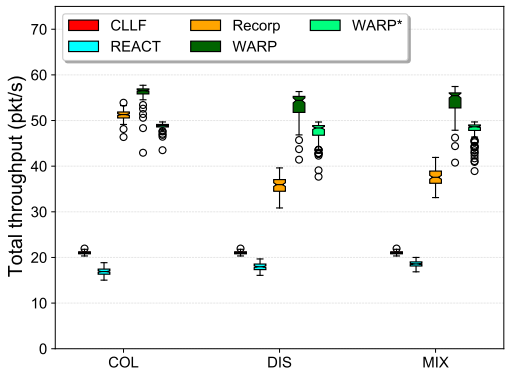
We have performed simulations on two real topologies: the Indriya topology which includes 85 nodes and has a 6-hop diameter (Indriya topology) [9], and the Washington Universtiy in St. Louis topology which includes includes 41 nodes and has a 6-hop diameter (WashU topology) [1]. The simulations use 802.15.4 with 16 channels.

To provide a comprehensive comparison between protocols, we considered three typical workloads under a multihop topology: data collection, data dissemination, and a mix of data collection and dissemination. The results presented are obtained from 100 simulation runs per workload type. In all runs, the base station is selected as the closest node to the topology center. In each run, 50 flows are created with the following constraints:

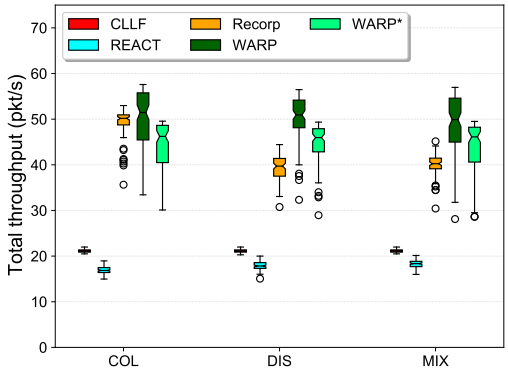
- *Data Collection (COL)*: Sources are selected randomly with the base station as the destination.
- *Data Dissemination (DIS)*: The base station is the source, and destinations are randomly selected.
- *Data Collection and Dissemination (MIX)*: Each flow is randomly selected to use either COL or DIS.

Each flow is randomly assigned to one of three classes whose periods and deadlines maintain a 1:2:5 ratio. Accordingly, if Class 1 is assigned a period of 100 *ms*, then Class 2 is assigned 200 *ms*, and Class 3 500 *ms*. We call the period of Class 1 the base period. In each run, the base period is decreased until a deadline is missed. The following results are obtained for the smallest base period for which all flows met their deadlines.

Flow throughput: Figures 7a and 7b plot the total throughput of each protocol under the considered workload type for the two topologies. We observed the same trend in both topologies: CLLF and REACT have significantly lower throughput than Recorp, WARP, and WARP*. REACT has a slightly lower median throughput than CLLF because its gap-induced algorithm forces all instances of a flow to have the same repeating pattern of transmissions. The clear winners are Recorp, WARP, and WARP*, because they can share entries across multiple flows, significantly improving the supported throughput. However, Recorp's performance varies significantly across different workload types. WARP improves upon Recorp's median throughput across all scenarios, with notable improvements for the DIS and MIX scenarios. In the Indriya topology for MIX workloads, WARP supports a medium total throughput of 55.4 packets per second compared to Recorp's 37.5 packets per second, representing a 47.68% improvement in throughput over Recorp. A slightly larger improvement of 52% is observed for the DIS workload in



(a) Indriya topology: Total throughput



(b) WashU topology: Total throughput

Figure 7: Simulation results

Protocol	Median synthesis time	Range synthesis time
CLLF	9.688 seconds	7.08 – 14.47 seconds
REACT	0.18 seconds	0.09 – 0.27 seconds
Recorp	33.67 seconds	23.88 – 63.59 seconds
WARP	0.687 seconds	0.35 – 1.25 seconds
WARP*	0.05 seconds	0.02 – 0.08 seconds

Table 2: Synthesis time

the same topology. WARP also provides approximately 2.6-times improvement over scheduling approaches. One factor contributing to these improvements is that WARP uses programs that include both pushes and pulls, whereas Recorp can only use pulls. Additionally, our heuristic to use pulls for downstream flows and pulls for upstream flows is effective in both collection, dissemination, and mixed workloads. WARP* has worse performance than WARP due to its table lookup mechanism. However, it remains competitive relative to WARP across all workloads. While the specific numbers differ between the two topologies, the conclusion remains the same: WARP consistently improves the throughput by as much as 2.6 times over scheduling approaches and as much as 50% over Recorp.

Synthesis Time: Table 2 shows the synthesis and scheduling times to build a complete schedule/program for each approach obtained for the MIX workload on Indriya topology. The simulations

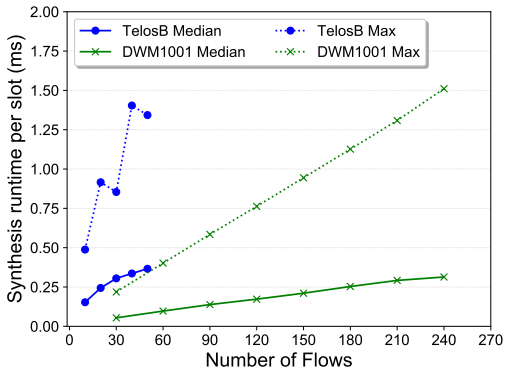


Figure 8: On-the-fly synthesis time

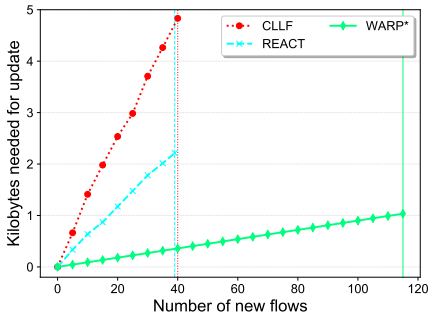
and synthesis is carried out on a 40-core Intel Xeon(R) CPU E5-2660 running at 2.60GHz and equipped with 128 GB RAM. Recorp uses a heavy-weight SMT-solver to build policies having a median and maximum synthesis time of 33.67 *s* and 63.59 *s*, respectively. CLLF has a median synthesis time of 9.688 seconds, while REACT is the most efficient scheduler with a median synthesis time of 0.18 seconds. WARP's synthesis is highly efficient: WARP can synthesize complete programs in less time than CLLF can construct schedules. WARP* reduces the synthesis time by more than a factor of 10 over WARP, making it the fastest across the considered protocols. We will later show how long it takes WARP* to synthesize a single fragment in a slot on TelosB and DWM1001 devices. Therefore, WARP's synthesis procedure not only provides significant performance improvements, but it is also efficient.

5.2 Testbed Results

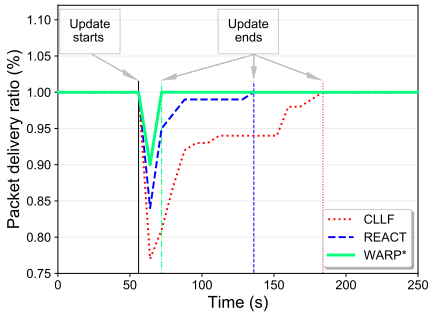
We evaluated WARP*, REACT, and CLLF on the three-hop, 16 node topology deployed at the University of Iowa using TelosB and the Decawave DWM1001. We did not evaluate Recorp on the testbed due to its long synthesis time and centralized control plane, which hinder its applicability to practical IIoT. DWM1001 radios use the 802.15.4a physical layer and support a maximum data rate of 6.8 Mbps, while TelosB uses 802.15.4 and supports the maximum data rate of 250 kbps. In all experiments, each device is configured to use its maximum data rate. TelosB and DWM1001 use packets of 38 bytes and 127 bytes, respectively. We used channels 15, 20, 25, and 26 for TelosB devices and channels 1, 3, and 5 with a pulse rate frequency of 16MHz on the DWM1001 devices. The WARP* implementations on TelosB runs on Contiki and the one on DWM1001 on FreeRTOS. The feasibility experiment is performed on both platforms, while the rest use only the DWM1001 devices.

	TelosB	DWM1001
Lookup tables	527 bytes	527 bytes
Topology	240 bytes	325 bytes
generator + estimator	500 bytes for 50 flows (10 bytes per flow)	4080 bytes for 240 flows (17 bytes per flow)

Table 3: Memory usage on TelosB and DWM1001



(a) Bytes need for update



(b) Representative trace of update time (s)

Figure 9: Adaption to workload change

Feasibility: We first consider the question of whether WARP* can synthesize and execute fragments on resource-constrained devices within 10 ms. Towards this end, we constructed MIX workloads by adding an increasing number of flows and running each workload for 10 minutes on the device. The period of all flows is 5 *s*. We measured the time to synthesize and execute a fragment as the number of added flows increased (see Figure 8). The experiment is stopped when we ran out of memory due to queuing many packets. The memory usage of our implementation on the two devices at their maximum number of supported flows is shown in Table 3. To our surprise, the decade-old TelosB handled 50 flows while the more powerful DWM1001 handled 240 flows. The synthesis and execution time is approximately 1.51 *ms* in the worst case, suggesting it is possible to have slots smaller than 10 *ms*.

Protocol	Min	Max	Mean	Median	Std. dev
CLLF	56	696	211.84	152	177.945
REACT	24	224	82.24	64	57.295
WARP*	16	16	16.0	16	0.0

Table 4: Update time (s)

Workload adaption: Next, we evaluate the agility of the control plane in response to workload changes. We set up an experiment with three flow classes having periods of 100, 200, or 400 with a ratio of 1:2:4. We opted to use harmonic periods since REACT performs the best in this case. The initial workload includes 30 flows, and we added a variable number of flows. To disseminate updates, a three-slot broadcast graph that has each intermediary node in the routing tree transmit once is scheduled every 2 *s*.

Protocol	Min	Max	Mean	Median	Std. dev
CLLF	77%	77%	77%	77%	0%
REACT	80%	84%	83%	84%	1%
WARP*	87%	99%	95%	98%	5%

Table 5: Minimum PDR during update time

The vertical bars in Figure 9a indicate the maximum number of flows that are supported by each protocol until missing a deadline. Consistent with the previous results, WARP* provided significantly higher throughput than CLLF and REACT. WARP* supports 115 flows, whereas CLLF and REACT only 40 and 39 flows, respectively.

We measure the control overhead measured as the number of bytes that must be disseminated as the number of added flows increases (see Figure 9a). The plot shows a linear relationship between control overhead and the number of added flows, but its slope differs significantly among protocols. CLLF must disseminate all additional entries and, as a consequence, must disseminate 4832 bytes to add 40 new flows. REACT's gap-induced scheduler forces all instances of a flow to have the same repeating pattern of transmissions. As a result, REACT has to disseminate the changes of a single instance and apply them across the schedule. To schedule 39 new flows, REACT requires 2214 bytes to update the schedule. WARP* must disseminate even less information than REACT as it only needs to disseminate a flow's parameters. When adding 39 new flows, WARP must disseminate only 351 bytes, which reduces the control overhead 6.3-times over REACT.

Next, we consider the run-time behavior of the protocols as they adapt in response to workload changes. Figure 9b plots the protocols' reliability in increments of 8 seconds for the duration of the experiment. Fifty-six seconds into the experiment, 10 new flows are added to the workload of initially 30 flows. All protocols experience a drop in the reliability of flows immediately after the update (new flow packets are not being delivered), but recovered after different time periods. CLLF requires 128 seconds to construct a new schedule and disseminate it to all nodes. REACT significantly improves upon CLLF, requiring only 80 seconds to perform the update. WARP* required only 16 seconds to update its schedule, a reduction of 5 times over REACT.

To understand whether these are statistically significant, we ran the experiment 25 times and recorded the update times and minimum PDR during the update time. These results are reported in Tables 4 and 5, respectively. The minimum and maximum update time of CLLF are 56 seconds and 696 seconds. Using REACT, the minimum update time is 24 seconds and the maximum is found 224 seconds. Both minimum and maximum update time of WARP* is 16 seconds, indicating substantial reduction in update times. Next, we consider the PDR during the update time. Using CLLF, the minimum and maximum values are 0.77. The minimum value using REACT is 0.80 and the maximum is 0.84. WARP* has the minimum value of 0.87 and maximum value of 0.99. This indicates that WARP* does not only provide shorter updates time but also is more reliable as the workload is updated.

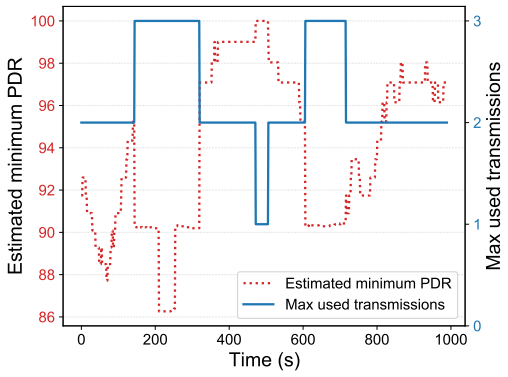


Figure 10: Estimated $\overline{m}$ and max. used transmissions of a flow's link

Reliability and Adaptability: WARP guarantees that the likelihood of a flow's packet reaching its destination exceeds a user-specified target probability when links behave according to the TLR model. No guarantees are provided when the links do not follow the TLR model. We set up an experiment where WARP* forwarded the data of 40 flows with periods and deadlines of 110 slots. The transmission power of the nodes is reduced to lower the mean link quality and increase its variability. We considered a sliding window of 100 packets shifted to the right in increments of one packet over the 1000-packet trace for a total of 901 windows. We fit a Bernoulli $\overline{m}$ random variable for each window to estimate TLR's minimum PDR. The value of $\overline{m}$ is determined such that the Bernoulli distribution with $\overline{m}$ as the chance of success provides a lower bound on the observed likelihood of bursts of failures. Accordingly, WARP's real-time and reliability constraints should be met if $\overline{m} \geq m = 70\%$. For each window, one of the following cases is possible:

- *Case* $\overline{m} \geq 70\%$*, E2E Met:* For 86.83% of the windows, the TLR model's assumptions held (i.e., $\overline{m} \geq m = 70\%$). WARP guaranteed that the end-to-end reliability of all flows exceeded the 99% target in these windows.
- *Case* $\overline{m} \geq 70\%$*, E2E Miss:* There are no cases when the assumptions of the TLR model held and the flows do not meet the target 99% reliability. These first two cases support our claim that WARP can meet the real-time and reliability constraints.
- *Case* $\overline{m} < 70\%$*:* When the actual link quality falls below the minimum link quality of $m = 70\%$, we provide no guarantees on the flow's reliability. In 13.17% of the windows, the TLR model's assumptions did not hold (since we reduced the transmission power to observe such cases). However, for all these windows, the end-to-end reliability of flows met their target reliability.

Finally, we evaluate how a node's program adapts in response to the patterns of packet losses observed during its execution. Figure 10 plots the maximum number of transmissions and estimated minimum packet delivery rate $\overline{m}$ for a link of a representative flow. The variations in $\overline{m}$ show that this link's quality can vary between 86% – 100% over time. The variations in the maximum number of transmissions used within a window show that the program changed the number of transmissions dynamically to adapt to the observed pattern of packet losses. In summary, we can conclude that WARP can adapt to variations in link quality and provide real-time and reliability guarantees when links follow the TLR model.

6 CONCLUSIONS

WARP is a new approach to building real-time and reliable wireless networks using software synthesis techniques. WARP has three core components: (1) A DSL to specify the data plane's behavior using programs that are more expressive than traditional schedules. Programs are stateful and can include conditional statements to control when a flow's packets are transmitted. Simulation experiments demonstrate that programs' more expressive behavior produces significant improvements in throughput across typical IIoT scenarios. (2) A software synthesis procedure can turn a high-level specification of the workload and topology into efficient programs that enable multiple flows to share entries in the schedule. The synthesized programs use a subset of power of the DSL to ensure that their execution can be analyzed efficiently. Our experiments indicate that the synthesized programs adapt to variations in link quality and satisfy their real-time and reliability constraints. WARP runs on-the-fly synthesis on DWM1001 devices with hundreds of flows. (3) WARP runs software synthesis on each node to reconstruct the same program based on shared workload and topology information. Consequently, WARP is agile in handling workload and topology changes by disseminating only the updates to the workload and topology information. Empirical results show that WARP can adapt more rapidly to workload changes than REACT. We hope that this work will motivate the community to consider using software synthesis for wireless networking and expand upon the techniques that we have developed.

ACKNOWLEDGEMENT

This work is funded in part by NSF under CNS-1750155.

REFERENCES

[1] 2021. WUSTL Wireless Sensor Network Testbed. http://wsn.cse.wustl.edu/index.php?title=Testbed
[2] Yuvraj Agarwal, Bharathan Balaji, Seemanta Dutta, Rajesh K Gupta, and Thomas Weng. 2011. Duty-cycling buildings aggressively: The next frontier in HVAC control. In *IPSN*.
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[4] R. Brummet, O. Chipara, and T. Herman. 2020. Recorp: Receiver-Oriented Policies for Industrial Wireless Networks. In *IoTDI*.
[5] Ryan Brummet, Md Kowsar Hossain, Octav Chipara, Ted Herman, and Steve Goddard. [n.d.]. WARP: On-the-fly Program Synthesis for Agile, Real-time, and Reliable Wireless Networks. https://drive.google.com/file/d/1K1ve9fC3Hp0VOaAUvH0bxhgD2sIuEFH3/view?usp=sharing.
[6] Alan Burns, James Harbin, Leandro Indrusiak, Iain Bate, Rob Davis, and David Griffin. 2018. Airtight: A resilient wireless communication protocol for mixed-criticality systems. In *RTCSA*.
[7] Len Calderone. 2020. Readers Choice 202:5G Is Coming to Agriculture. *Agritech Tomorrow* (Dec 2020). https://www.agritechtomorrow.com/article/2020/07/readers-choice-2020-5g-is-coming-to-agriculture/12275
[8] Pablo Corbalán and Gian Pietro Picco. 2018. Concurrent Ranging in Ultra-Wideband Radios: Experimental Evidence, Challenges, and Opportunities. In *EWSN*.
[9] Manjunath Doddavenkatappa, Mun Chan, and A.L. Ananda. 2012. Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering* 90, 302–316. https://doi.org/10.1007/978-3-642-29273-6_23
[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *NSDI*.

[11] Atis Elsts, Seohyang Kim, Hyung-Sin Kim, and Chongkwon Kim. 2020. An empirical survey of autonomous scheduling methods for TSCH. *IEEE Access* (2020).
[12] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. 2012. Low-Power Wireless Bus. In *SenSys*.
[13] Karen Field. 2020. Embracing IoT tech, John Deere plows a path to autonomous machines. *Fierce Electronics* (Nov 2020). https://www.fierceelectronics.com/sensors/embracing-iot-tech-john-deere-plows-a-path-to-autonomous-machines
[14] A. Gonga, O. Landsiedel, P. Soldati, and M. Johansson. 2012. Revisiting Multi-channel Communication to Mitigate Interference and Link Dynamics in Wireless Sensor Networks. In *ICDCS*.
[15] Dolvara Gunatilaka and Chenyang Lu. 2020. REACT: an Agile Control Plane for Industrial Wireless Sensor-Actuator Networks. In *IoTDI*.
[16] James Harbin, Alan Burns, Robert I Davis, Leandro Soares Indrusiak, Iain Bate, and David Griffin. 2019. The AirTight Protocol for Mixed Criticality Wireless CPS. *TCPS* (2019).
[17] Ozlem Durmaz Incel. 2011. A survey on multi-channel communication in wireless sensor networks. *Computer Networks* (2011).
[18] Chenyang Lu, Abusayeed Saifullah, Bo Li, Mo Sha, Humberto Gonzalez, Dolvara Gunatilaka, Chengjie Wu, Lanshun Nie, and Yixin Chen. 2015. Real-time wireless sensor-actuator networks for industrial cyber-physical systems. *Proc. IEEE* (2015).
[19] Emerson Process management. 2016. System Engineering Guidelines IEC 62591 WirelessHART.
[20] Venkata Prashant Modekurthy, Abusayeed Saifullah, and Sanjay Madria. 2019. DistributedHART: A distributed real-time scheduling system for wirelesshart networks. In *RTAS*.
[21] Marcelo Nobre, Ivanovitch Silva, and Luiz Affonso Guedes. 2015. Routing and scheduling algorithms for WirelessHART Networks: a survey. *Sensors* (2015).
[22] Pangun Park, Sinem Coleri Ergen, Carlo Fischione, Chenyang Lu, and Karl Henrik Johansson. 2017. Wireless network design for control systems: A survey. *IEEE Communications Surveys & Tutorials* (2017).
[23] Abusayeed Saifullah, You Xu, and Chenyang Lu. 2010. Real-Time Scheduling for WirelessHART Networks. In *RTSS*.
[24] Tianyu Zhang, Tao Gong, Zelin Yun, Song Han, Qingxu Deng, and Xiaobo Sharon Hu. 2018. FD-PaS: A fully distributed packet scheduling framework for handling disturbances in real-time wireless networks. In *RTAS*.
[25] Marco Zimmerlinonga, Luca Mottola, Pratyush Kumar, Federico Ferrari, and Lothar Thiele. 2017. Adaptive real-time communication for wireless cyber-physical systems. *TCPS* (2017).