ReSF: Recurrent Low-Latency Scheduling in IEEE 802.15.4e TSCH Networks

Article *in* Ad Hoc Networks · November 2017

DOI: 10.1016/j.adhoc.2017.11.002

CITATIONS
12

READS
341

4 authors:

Glenn Daneels
University of Antwerp
11 PUBLICATIONS **69** CITATIONS
SEE PROFILE

Bart Spinnewyn
University of Antwerp
14 PUBLICATIONS **99** CITATIONS
SEE PROFILE

Steven Latré
University of Antwerp
176 PUBLICATIONS **1,888** CITATIONS
SEE PROFILE

Jeroen Famaey
University of Antwerp
128 PUBLICATIONS **1,444** CITATIONS
SEE PROFILE

ReSF: Recurrent Low-Latency Scheduling in IEEE 802.15.4e TSCH Networks

Glenn Daneels, Bart Spinnewyn, Steven Latré, Jeroen Famaey

IDLab, University of Antwerp - imec, Belgium

Middelheimlaan 1, 2020 Antwerp, Belgium

Abstract

The recent increase of connected devices has triggered countless Internet-of-Things applications to emerge. By using the Time-Slotted Channel Hopping (TSCH) mode of the IEEE 802.15.4e MAC layer, wireless multi-hop networks enable highly reliable and low-power communication, supporting mission-critical and industrial applications. TSCH uses channel hopping to avoid both external interference and multi-path fading, and a synchronization-based schedule which allows precise bandwidth allocation. Efficient schedule management is crucial when minimizing the delay of a packet to reach its destination. In networks with recurrent sensor data transmissions that repeat after a certain period, current scheduling functions are prone to high latencies by ignoring this recurrent behavior. In this article, we propose a TSCH scheduling function that tackles this minimal-latency recurrent traffic problem. Concretely, this work presents two novel contributions. First, the recurrent traffic problem is defined formally as an Integer Linear Program. Second, we propose the Recurrent Low-Latency Scheduling Function (ReSF) that reserves minimal-latency paths from source to sink and only activates these paths when recurrent traffic is expected. Extensive experimental results show that using ReSF leads to a latency improvement up to 80 % compared to state-of-the-art low-latency scheduling functions, with a negligible impact on power consumption of at most 6 %.

Keywords: IEEE 802.15.4e, Time-Slotted Channel Hopping, MAC scheduling, 6TiSCH

1. Introduction

In recent years, the increasing number of devices connected to the Internet has converged into a new paradigm called the Internet-of-Things (IoT). Countless IoT applications have emerged that enrich our daily lives by connecting more intelligent devices to the Internet. An important concern of IoT networks is the power-constrained nature of the connected devices. This is due to the low expected cost per device, as well as the mobility of the nodes and their deployment in difficult-to-reach locations. Despite this power restriction, many applications have strict demands in terms of reliability and delay. To bridge that gap, a new IETF Working Group (WG) called *IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH)* was created in 2013, as a response to the inevitable convergence between deterministic industrial networks and traditional IP networks [1, 2]. The goal of 6TiSCH is to create a link-layer standard that combines industrial performance in terms of reliability and power consumption with an IPv6-enabled upper stack, focusing on multi-hop Low-power Lossy Networks (LLNs).

In order to achieve this goal, the 6TiSCH architecture uses the Time-Slotted Channel Hopping (TSCH) mode of the IEEE 802.15.4e MAC layer. TSCH uses channel hopping to increase the link reliability and minimize the effects of external interference and/or multi-path fading, by calculating a pseudo-random frequency channel for every transmission. A time-synchronized schedule tells the nodes exactly when to send/receive and thereby avoids energy-harmful contention periods and - when the schedule is managed efficiently - idle listening. Intelligent scheduling is crucial to also perform well in terms of other network metrics such as delay and throughput.

Both the 6TiSCH WG and other researchers have defined several scheduling functions to manage the schedule (i.e., centralized versus distributed) and thereby optimize different metrics (e.g., latency, power consumption) for different types of applications. In this work, we focus on power-constrained sensor networks where each node periodically sends measurement updates to a sink (i.e., recurrent traffic) and demands low-latency dissemination of these data. Current state-of-the-art scheduling functions, such as the Low Latency Scheduling Function (LLSF), also focus on low-latency forwarding of packets but do not anticipate the recurrent behavior. This results in higher latencies, because starting the time-expensive resource allocation process when the recurrent traffic already arrived is too late to maintain a low packet delay. We propose the distributed Recurrent Low-Latency Scheduling Function (ReSF). It explicitly supports recurrent traffic and as far as we know, is the first TSCH scheduling function that takes this recurrent traffic behavior into account when scheduling resources. ReSF reserves a minimal-latency path from source to sink and only activates this reserved path when traffic is expected. This al-

Email addresses: glenn.daneels@uantwerpen.be (Glenn Daneels), bart.spinnewyn@uantwerpen.be (Bart Spinnewyn), steven.latre@uantwerpen.be (Steven Latré), jeroen.famaey@uantwerpen.be (Jeroen Famaey)

lows resources to be reused more intelligently, thus improving throughput and latency. As such, it has only minimal impact on battery while significantly decreasing the packet delay.

This article presents two specific contributions. First, we define the problem of minimal-latency scheduling of recurrent data transmissions in sensor networks formally, using an Integer Linear Program (ILP). Second, we present ReSF as well as its collision prevention algorithm and integration with 6TiSCH. ReSF is a distributed scheduling function that solves the presented recurrent data transmission scheduling problem in real-time. Additionally, we provide extensive simulation results based on the official 6TiSCH simulator, which we extended with a fully functional 6top Protocol (6P) implementation. A comparison is provided with an extended version of LLSF [3], the state-of-the-art minimal-latency scheduling function for 6TiSCH. Our proposed TSCH scheduling solution is the first to explicitly exploit the recurrent nature of many transmissions in Wireless Sensor Networks (WSNs).

The remainder of this article is structured as follows. First, we introduce 6TiSCH and related work on scheduling functions in Section 2. Subsequently, Section 3 introduces the recurrent minimal-latency scheduling problem and Section 4 proposes our ReSF algorithm to solve the problem in a distributed manner in real-time. The proposed algorithm is evaluated and compared to the theoretical optimum as well as state-of-the-art scheduling functions in Section 5. Finally, Section 6 presents the conclusions of our work.

2. Background and Related Work

In this section we briefly introduce 6TiSCH, a link-layer IETF draft that bridges the gap between industrial and traditional IP networks. We provide the necessary background on TSCH, an important underlying mechanism of 6TiSCH. Further on, we give more information on the 6P protocol that manages the resources in 6TiSCH and give an overview of related work regarding TSCH scheduling functions.

2.1. 6TiSCH Architecture

The 6TiSCH WG focuses on enabling IPv6 over the TSCH mode of the IEEE 802.15.4e standard for reliable multi-hop LLNs. While TSCH only focuses on the MAC layer, the 6top sublayer, the 6LoWPAN adaptation and compression layer, the Routing Protocol for Low-Power and Lossy Network (RPL) routing protocol, and the COAP application protocol are combined on top of TSCH in the 6TiSCH protocol stack.

In contrast to traditional CSMA/CA MAC protocols, in TSCH networks each node follows a schedule that tells the node exactly when it has to transmit data to, or receive data from, neighboring nodes. This tight time synchronization-based schedule allows for low-power operations as the radio is only turned on when indicated by the schedule. Moreover, it increases reliability as contention can be avoided by assigning each slot to non-interfering stations only. The schedule consists of *cells* (or *slots*) that last long enough to allow a node to send/receive 1 packet and receive/send the associated acknowledgement, typically either 10 ms or 15 ms long. There are four cell types: `TX`, `RX`, `SHARED` and `OFF` cells. `TX` and `RX` cells are dedicated cells to send and receive data, `SHARED` cells allow multiple nodes to contend for the medium using a back-off algorithm and during `OFF` cells the radio is turned off. `SHARED` cells can usually not be removed from the schedule (when set as so-called *hard* cells) and are often used to bootstrap the network. A *slotframe* defines the width of the schedule and is actually a group of cells that repeats over time. The schedule height is defined by the number of available channels (i.e., frequencies) in which the nodes can transmit/receive data. The rows in the schedule represent channel offsets to perform *channel hopping*.

Management of this schedule is done using a scheduling function. How the scheduling should be done is not defined by 6TiSCH and mostly depends on the type of application and the metrics that need to be optimized. The tasks of a scheduling function include (i) adding cells between two nodes when current resources are not sufficient to handle the traffic load, and (ii) removing cells when the schedule is over-provisioned and valuable resources are at risk of being wasted. Both centralized and distributed approaches are possible, or in the simplest case a static schedule can be uses as defined in the minimal 6TiSCH configuration [4].

While the schedule allows for low-power and reliable operations in 6TiSCH networks, channel hopping further increases the link-reliability by leveraging the problem of packet loss due to multi-path fading and external interference [5]. Channel hopping uses frequency diversity, calculating the frequency channel at which both sender and receiver transmit and receive data using a pseudo-random calculation.

In Figure 1, an example is given of a TSCH schedule for a tree topology consisting of 4 nodes with root node R. Both leaf nodes Y and Z generate one packet each slotframe. Each slotframe starts with a `SHARED` slot that allow the nodes to make dedicated cell reservations. While node Y and Z only have 1 transmit cell towards node X, node X has 2 cells towards the root because it needs one cell for each packet coming from node Y and Z. As seen in the second slotframe, the channel offset of the cell for the link between Y and X is the same as in the first slotframe, but because of the channel hopping the frequency channel will differ from the previous one.

2.2. 6top Protocol (6P)

6P allows neighboring nodes in a 6TiSCH network to add and delete cells and is part of the 6TiSCH Operation Sublayer (6top) IEEE 802.15.4e sublayer which provides the mechanisms to do distributed scheduling in 6TiSCH [6]. It is the scheduling function that decides when to add or delete cells, and it uses 6P to effectively execute the resource allocation.

When new cells need to be added or deleted the 6P protocol starts a so-called 6P transaction that is actually the complete cell negotiation between two nodes. There is both a 2-step and 3-step negotiation process: we implemented the 2-step negotiation. A 2-step transaction between node A and node B means that node A sends a 6P `ADD` request to node B, specifying the number of cells it wants and the list out of which node B can pick, with a cell being ($timeOffset$, $channelOffset$). When sending the request, a timer is set to abort the transaction when

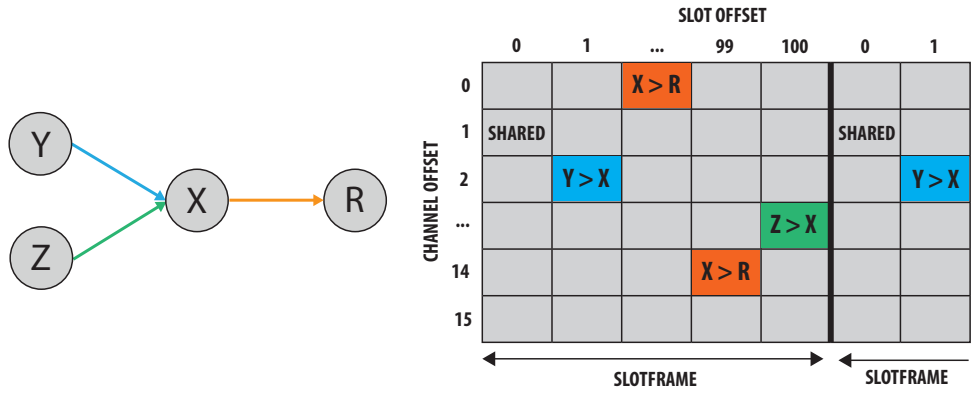


Figure 1: Example of a TSCH schedule, where nodes Y and Z each generate one packet per slotframe and node X forwards these packets to root R. There are 101 cells per slotframe and 16 frequencies.

the receiver did not respond when the timer expires. Node B answers with a 6P `RESPONSE` containing the cells that suit node B best. The 6P transaction for deleting cells is similar, but the 6P `ADD` request is replaced by a 6P `DELETE` request containing the cell list that node A wants to be removed from its schedule.

We extended the 6TiSCH simulator with the 6P protocol and downwards management traffic to simulate more realistic cell reservations and removals to both parents and children.

2.3. Existing Scheduling Approaches

The increasing popularity of 6TiSCH networks in the recent years has driven the development of numerous new algorithms and the further improvement of already existing scheduling functions. On top of that, the 6TiSCH community has also standardized several scheduling functions such as Scheduling Function Zero (SF0) and Scheduling Function One (SF1) [7, 8]. In this section, we discuss recent work on TSCH scheduling and relate it to our proposed solution ReSF.

2.3.1. Centralized Scheduling

A centralized scheduling approach requires a central entity that builds the schedule for all nodes, based on reported or monitored network information.

Palattella et al. [9] propose the Traffic Aware Scheduling Algorithm (TASA) which uses *matching* and *coloring* procedures to build a TSCH schedule and which assumes full topology and traffic awareness. However, having such knowledge requires a lot of signaling which greatly affects the network performance in terms of latency and power consumption. Recently, Farías et al. [10] also introduced a centralized queue-based 6TiSCH scheduler for industrial applications. Huynh et al. [11] developed a centralized scheduler that focuses on reliability and energy efficiency by applying opportunistic forwarding at the MAC and routing layer.

A central approach typically has the advantage of being able to build near-optimal, collision-free schedules that can approach the theoretical optimum. However, scalability limitations and extra signaling overhead make centralized schedulers not feasible for dynamic environments.

2.3.2. Distributed Scheduling

In networks with a distributed scheduling approach, nodes maintain their own schedule. These solutions typically scale better because there is less signaling overhead. However, less signaling also means that the nodes are less informed about the network which makes it harder to build efficient schedules.

A commonly used distributed scheduling function is SF0 which was originally introduced as On-the-Fly Scheduling Function (OTF) by the 6TiSCH community and is currently being further developed as a work in progress in an Internet Engineering Task Force (IETF) Internet-Draft [7]. SF0 is a distributed scheduling algorithm: each node dynamically adapts the amount of resources between itself and its neighbors, based on its current resource allocation and its resource requirements. Two major drawbacks of SF0 are (i) that is does not take into account the recurrent behavior of traffic, meaning that each reserved cell repeats itself every slotframe and thus wastes resources if packet generation is not equally frequent and (ii) cells are randomly allocated in a slotframe, risking that packets can not be forwarded immediately and have to wait an additional slotframe. These major issues are addressed by ReSF. SF0 is used as a baseline in the experimental evaluation. Based on SF0, Chang et al. developed an improved version called LLSF that daisy-chains cells over the different links up to the root, rather than picking them randomly [3]. While LLSF does not introduce extra overhead, it also does not anticipate recurrent behavior and thus still leaves room for improvement. An enhanced version of LLSF, which is discussed in Section 4.6, is also used for comparison in the experimental evaluation.

Another scheduling function Internet-Draft being investigated by the 6TiSCH community is SF1 [8]. SF1 is and end-to-end distributed resource scheduler with hop-by-hop reservation, using a distributed Resource Reservation Protocol (RSVP). It allocates a dedicated path from source to destination which is called a *track*. In contrast to ReSF, so-called TrackIDs are used to filter data packets for certain tracks. In ReSF recurrent cells can be used by any data packet in the queue. The draft is still in an early stage and specific features taking into account recurrent behavior or algorithms to efficiently allocate cells are not present as of yet. As such, we consider our work com-

plementary to SF1, and it could serve to inspire standardization. Morell et al. [12] worked out a solution that combines Resource Reservation Protocol - Traffic Engineering (RSVP-TE) and Generalised Multiprotocol Label Switching (GMPLS) to manage the schedule and connect the network nodes using labelled switched paths. However, the authors do not focus on the recurrent traffic and their numerical evaluation does not take into account the 6P signaling process. Theoleyre et al. [13] also focus on traffic isolation by introducing different tracks for different applications and consider contiguous reserved cells as well as random reserved cells. Their distributed algorithm calculates the required number of cells based on the amount of forwarded traffic, but it also does not consider the recurrent behavior of sensor data generation, where the reserved cell may only be needed every so many slotframes. It thus risks wasting resources because of a repeated reservation process.

Other distributed approaches have been proposed such as Decentralized Traffic Aware Scheduling (DeTAS) [14]. DeTAS aims at minimizing buffer overflow and queue utilization. It applies an hierarchical approach that builds *micro-schedules* (with alternating transmission and reception cells) for the different network sinks which are aggregated into a global *macro-schedule*. To build the micro-schedules, DeTAS uses a considerable amount of 1-hop signaling. A similar approach of neighborhood signaling is used by Municio et al. [15]. The DeBras scheduler targets dense network deployments. It broadcasts scheduling information to avoid internal interference and thus reduces the number of collisions. However, DeBras is not focused on latency optimization and therefore it does not consider contiguous cells nor does it take traffic characteristics into account. Another distributed approach is *Wave* [16] that constructs the schedule out of different so-called waves which are basically cell matrices that accommodate packet transmissions. However, Wave is also limited by its neighbor signaling that informs nodes about conflicting transmissions.

A radically different way of resource allocation in TSCH is Orchestra [17], which does not use a distributed nor a centralized scheduler. It manages its schedule locally and does not negotiate with its neighbors. It assigns different types of slotframes to multiple traffic planes (i.e., TSCH beacons, RPL signaling or application traffic) and thereby targets high reliability.

In contrast to the related works, ReSF specifically focuses on IoT applications with recurrent traffic patterns. The state-of-the-art approaches either reserve isolated tracks towards the root and/or reserve resources that are continuously activated. Meanwhile, ReSF tries to guarantee fairness, low-latency and energy-efficiency by not isolating the low-latency paths and only activates resources when traffic is expected, maximizing cell reuse and performance.

3. Recurrent Low-Latency Scheduling

In this section, we introduce the recurrent low-latency scheduling problem, which minimizes end-to-end data dissemination latency for devices with recurrent data transmissions (e.g., transmission of sensor measurement values after fixed periods). First, we discuss our motivation to tackle the recurrent low-latency problem. Second, we formally formulate the problem and the associated optimal solution as an Integer Linear Program (ILP).

3.1. Motivation

Periodical traffic patterns are typical for Wireless Sensor Networks (WSNs) as sensors usually perform measurements at fixed intervals and/or measured data is accumulated over fixed periods and only then sent to the root. ReSF was developed to exploit this recurrent behavior by allocating resources only when traffic is expected. Existing scheduling functions such as SF0 and LLSF allocate new resources based on historical data: during the so-called periodic housekeeping periods the scheduling function monitors how many packets arrive and are generated by the node itself. Afterwards, it calculates the number of cells it needs to reserve during the next housekeeping period. When reserving these cells, the exact arrival times of the incoming/generated packets are not taken into account, which leads to inefficient resource allocation in terms of latency and power consumption. Moreover, if the data generation period of the sensor is higher than the housekeeping period, an estimate based on the previous housekeeping intervals may not be representative for the current one.

A simplified example of such inefficient resource allocation is given in Figure 2. The given schedule (for the simplicity of this example with only one channel) is the one of the forwarding node B. Node A sends traffic every 8 slotframes to node B, which should forward it as quickly as possible to the root. During housekeeping at node B, the node decides to reserve a cell to its parent so the incoming packet can be forwarded. To do so, first a 6P `ADD` transaction between node B and its parent needs to take place. Meanwhile the latency of the packet is increasing as it is only in the third slotframe (after 9 timeslots) that a `TX` cell is available to forward the packet to the root. Node A only generates traffic periodically every 8 slotframes, so the next 2 `TX` cells to the root are not used (and thus radio resources at the receiver are wasted) and even after the housekeeping a third cell is still reserved because the deletion of the cell is not yet complete. One should also take into account the power consumption for sending/receiving these 6P `DELETE` and 6P `RESPONSE` messages. When the next periodical packet of A arrives, the process at node B repeats itself. Taking into account the recurrent behavior of the traffic of node A, the increased latency and power consumption introduced by the reservation process can be avoided. This results in a more efficient scheduling function process, which is exactly the goal of ReSF.

3.2. Problem formulation

In this section we formally approach low-latency end-to-end packet scheduling in a Wireless Sensor Network (WSN) with recurrent transmissions, as a resource allocation problem. We formulate the underlying optimization problem as an ILP, consisting of input variables, decision variables, constraints and an objective function. Algorithms such as branch and bound can be used to find the optimal solution, by determining the values

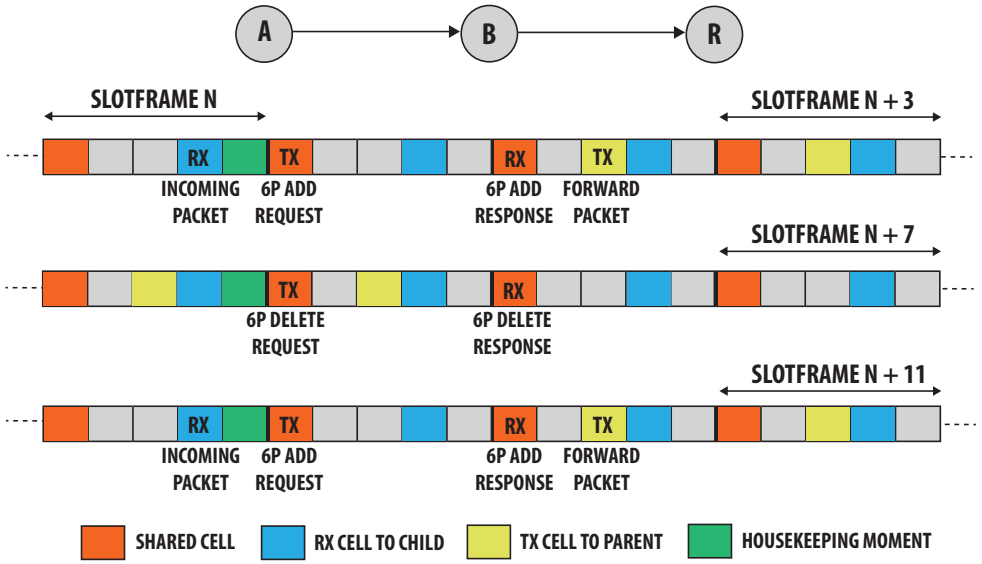


Figure 2: Illustration of inefficient resource allocation when not taking into account the recurrent behavior of sensor traffic. The schedule shown belongs to node B.

of the decision variables that maximize (or minimize) the objective function, while satisfying the constraints, given the inputs. As such, the formulation should be considered a resource allocation optimization problem. While the ILP formulation assumes any generated packet to be directed towards the root node, extension to support communication between any two interconnected nodes is trivial. The ILP solution is only used for theoretical comparison purposes in Section 5. Solving an ILP on real hardware nodes would be infeasible due to its computationally complexity and the need to be solved centrally. The remainder of this section describes the different aspects of the ILP formulation.

3.2.1. Input variables

The WSN comprises a set of nodes $\mathbf{N}$. Packets originate from a set of sources $\mathbf{V} \subset \mathbf{N}$, with the root node $n_R \in \mathbf{N}$ as destination. These sources are recurrent, i.e. they generate a packet at fixed time intervals $S_v + k \times T_v, k \in \mathbb{N}_0$. For each source $v \in \mathbf{V}$: T_v and S_v are its period and offset respectively. Note that the period T_v may change over time.

The system is analyzed when it has reached steady state conditions, i.e., the number of packets generated per period is constant. The nodes each have only 1 radio, which can either send or receive packets. The combined period of the sources, referred to as the system period, is then given by

$$T_{sys} = \text{LCM}(T_0, T_1, \ldots, T_{|\mathbf{V}|-1}), \tag{1}$$

where LCM is a function that calculates the Least Common Multiple (LCM) of its integer arguments. When the scheduling is also periodic with period T_{sys}, than it suffices to analyze the system during one system period, beginning when all sources have generated at least one packet. Hence we analyze the system only for time slots

$$i \in \mathbf{I}, \mathbf{I} = \{S_{max} + 1, \ldots, S_{max} + T_{sys} - 1\}, \tag{2}$$

Table 1: Input variables.

Symbol	Description
$\mathbf{N}$	set of nodes
n_R	root node
$\mathbf{V}$	set of source nodes
S_v	offset of source node v
T_v	period of source v
$\mathbf{P}_v$	edges on the path from v to n_R
$P_{v,p}$	p^{th} edge on the path from v to n_R

Table 2: Auxiliary symbols.

Symbol	Description
T_{sys}	system period, i.e. $LCM(T_0, T_1, \ldots, T_{\mathbf{V}-1})$
$\mathbf{I}$	time slots in one system period, i.e., $S_{max} + 1, \ldots, S_{max} + T_{sys} - 1$
$\mathbf{J}_v$	set of packets generated by source v during one system period

where the largest system period $S_{max} = \max(S_0, \ldots, S_{|\mathbf{V}-1|})$. The set of packets generated by source v in the T_{sys} time slots in $\mathbf{I}$ is given by

$$\mathbf{J}_v = \{0, 1, \ldots, \frac{T_{sys}}{T_v} - 1\}. \tag{3}$$

A complete list and description of the input and auxiliary variables introduced above can be found in Tables 1 and 2 respectively. Moreover, Figure 3 illustrates the notations graphically. In this example, both nodes A and B generate traffic: with $T_A = 6$ and $T_B = 2$, therefore they generate a packet every 6 slots and 2 slots respectively. This results in $T_{sys} = 6$ as this is the least common multiple of 6 and 2.

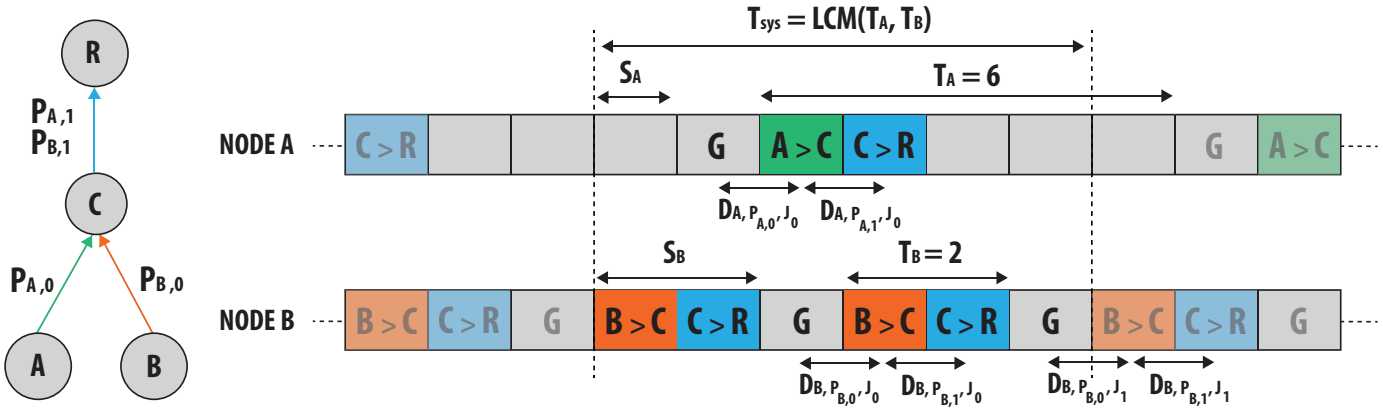


Figure 3: Illustration of the notations used in the ILP-formulation. Both nodes A and B generate traffic at time slots denoted by G.

3.2.2. *Decision variables*

The decision variables represent the solution of the ILP model. $x_{v,p,j}(i)$ is a binary variable, and equals 1 if the packet (v, j) is transmitted over edge $P_{v,p}$ during time slot 1, else it is 0. $T_n(i)$ and $R_n(i)$ are 1 when node n is respectively transmitting or receiving during time slot i. $D_{v,p,j}$ is an integer variable holding the delay contribution of each packet (v, j). $O_{v,p,j}$ is a binary variable and a value of 1 indicates that the delay contribution of the p^{th} path element for packet (v, j) crosses the edge of the analyzed interval $\mathbf{I}$ (Equation 2). In this case the packet arrives only in the next system period, hence T_{sys} is added to the arrival time.

3.2.3. *Constraints*

This section outlines the constraints, which determine the allowed values of the decision variables, as a function of the inputs.

Multi Commodity Flow (MCF). The net packet flow leaving node n_1, corresponding to each packet for source v, depends on whether this node is the source, destination (root node) or neither to this specific packet. For each node n_1, and for each packet (v, j), the difference between the total number of packets flowing out and into n_1 corresponding to (v, j), during one system period, equals either 1, −1, or 0, based on the relation of n_1 to this packet (source, root, or neither).

$$\forall n_1 \in \mathbf{N}, v \in \mathbf{V}, j \in \mathbf{J}_v : \sum_{i \in \mathbf{I}} \left[\sum_{(n_1,n_2) \in \mathbf{P}_v} x_{v,(n_1,n_2),j}(i) - \sum_{(n_3,n_1) \in \mathbf{P}_v} x_{v,(n_3,n_1),j}(i) \right] = \begin{cases} 1, & \text{if } n_1 = v \\ -1, & \text{else if } n_1 = n_R \\ 0, & \text{else} \end{cases} \quad (4)$$

Transmission and reception. A packet is transmitted/received during slot i if a packet corresponding to any of the sources is transmitted or received during this time slot:

$$\forall n_1 \in \mathbf{N}, i \in \mathbf{I} : T_{n_1}(i) = \sum_{v \in \mathbf{V}} \sum_{j \in \mathbf{J}_v} \sum_{(n_1,n_2) \in \mathbf{P}_v} x_{v,(n_1,n_2),j}(i), \quad (5)$$

and

$$\forall n_2 \in \mathbf{N}, i \in \mathbf{I} : R_{n_2}(i) = \sum_{v \in \mathbf{V}} \sum_{j \in \mathbf{J}_v} \sum_{n_1 \in N} x_{v,(n_1,n_2),j}(i) \quad (6)$$

Single radio. A node has only one radio and therefore cannot receive and transmit at the same time.

$$\forall n \in \mathbf{N}, i \in \mathbf{I} : T_n(i) + R_n(i) \leq 1 \quad (7)$$

Delay. The delay for each packet is the sum of the individual delay contributions of the edges along the path from the source of the packet towards the root. We distinguish two types of delay contribution. The delay contribution of the first edge on the path is the difference between the time slot at which the packet is first transmitted, and the time slot at which it was generated:

$$\forall v \in \mathbf{V}, j \in \mathbf{J}_v, p = P_{v,0} : D_{v,p,j} = (\sum_{i \in \mathbf{I}} i \times x_{v,p,j}(i) + T_{max} \times O_{v,p,j}) - (S_v + j \times T_v) \quad (8)$$

The delay contribution for the other path elements is the difference between the time slot at which the packet reaches the target of the element, minus the time slot at which it had reached its source:

$$\forall v \in \mathbf{V}, j \in \mathbf{J}_v, p \in \mathbf{P}_v \setminus \{P_{v,0}\} : D_{v,p,j} = (\sum_{i \in \mathbf{I}} i \times x_{v,p,j}(i) + T_{max} \times O_{v,p,j}) - \sum_{i \in \mathbf{I}} i \times x_{v,p-1,j}(i), \quad (9)$$

where the final term accounts for packets that do not arrive at the source and destination of $P_{v,p}$ during the same system period.

A packet cannot be transmitted by a node during the time slot of its generation, and it cannot traverse multiple edges during one single time slot. Hence, the delay contribution of each path element is limited to

$$1 \leq D_{v,p,j} \leq T_{sys} - 1. \quad (10)$$

3.2.4. Objective function

The objective is to minimize the average delay of the $T_{max} \times |\mathbf{V}| / \sum_{v \in \mathbf{V}} T_v$ packets generated in a system period:

$$\min \sum_{v \in \mathbf{V}} \sum_{j \in \mathbf{J}_v} \sum_{p \in \mathbf{P}_v} \frac{T_v}{T_{max} \times |\mathbf{V}|} D_{v,p,j} \tag{11}$$

4. Recurrent Low-Latency Scheduling Function (ReSF)

In the previous section, the underlying resource problem of recurrent scheduling was stated formally and an optimal mathematical solution was provided. However, such an optimal solution is computationally complex and needs to be solved centrally as it assumes perfect network knowledge. Therefore, in this section we propose the new ReSF scheduling function that solves the recurrent low-latency scheduling problem in real-time and a distributed manner. First, the goal and features of ReSF are discussed in a general overview of the scheduling function, which is clarified with an example. Afterwards, detailed information on the scheduling algorithm, packet loss, the schedule collision prevention and queue housekeeping using Enhanced Low Latency Scheduling Function (eLLSF) are presented. Finally, the integration of ReSF with the 6P protocol is discussed in detail.

4.1. General Overview

ReSF was designed to minimize the latency of periodic data transmissions while keeping the reservation overhead, i.e., the number of control messages and the number of reserved slots, to a minimum. It targets IoT systems where traffic is periodically sent, following fixed or slowly changing patterns. The 4 main steps are outlined below:

Scheduling an ReSF reservation. ReSF assumes a source node knows its periodic traffic pattern. Using this information, ReSF constructs a so-called *recurrent* path that consists of recurrent cells which are cells that are only activated in slotframes when traffic is expected and are deactivated afterwards. An ReSF reservation is based on the following tuple: $(start, stop, period)$. $Start$ is the Absolute Sequence Number (ASN) at which the first data packet is generated on the source node, $stop$ is the ASN at which point no data will be generated anymore and *period* represents the periodicity of the data transmission. The tuple is sent from the source node to the next hop and forwarded to the ReSF destination: at each hop the *start* ASN is incremented to an ASN as closely as possible following the received *start* ASN, resulting in a daisy-chained ReSF path from source to destination. A path is however not explicitly reserved for one particular packet stream: if a node makes a ReSF reservation that is forwarded all the way up to the destination node, the allocated recurrent cells at an intermediate node along that path can be used by any packet (originating from any node) that is first in the transmission queue of that intermediate node, guaranteeing fairness and lowest average latency.

Anticipating packet loss. ReSF anticipates packet loss (cf. Section 4.4) by reserving back-up tuples. The number of extra reservations depends on the measured link quality and allows a node to retransmit multiple times consecutively.

Preventing schedule collisions. A *schedule collision* is caused when multiple ReSF reservations, located in the same node, want to occupy exactly the same cell at a particular ASN. While theoretically it is possible to search for reservation tuples between sender and receiver that do not share overlapping cells, such a reservation process will take too long and is therefore practically infeasible. Instead, a node that wants to schedule an ReSF reservation message searches, in a pool of reservation tuples, for candidate tuples with the lowest schedule collision rate. The receiver picks the required number of reservations out of the sent candidate tuples by again relying on the lowest collision rate. To efficiently calculate this schedule collision rate, we propose an algorithm explained in Section 4.5.

Queue housekeeping using eLLSF. ReSF prevents failing packets from congesting the queue by doing additional periodical housekeeping. This is done using eLLSF, an enhanced version of the LLSF scheduling function. During this periodical eLLSF housekeeping, a node reserves (or deletes) extra cells that repeat until the next housekeeping moment in order to keep the queue as empty as possible.

4.2. Example

As an example, consider Figure 4. There are 4 nodes and 1 root R. It is assumed that nodes A and C are far enough away from nodes B and D, so there is no internal interference between these two links. Both node A and node B have a sensor application that periodically generates packets. The traffic destination is root node R. The shown schedule is an aggregated schedule of all individual nodes. The housekeeping moments and decisions of node B are also shown.

The traffic generation pattern of node B looks like ($start = 31, stop = 600, period = 12$), meaning that the first packet starts at ASN 31 and gets repeated every 12 cells. The ReSF reservations from node B to node D and from node D to node R are already installed. Because the Expected Transmission Count (ETX) of the link between node B and node D is 2, there are two recurrent cells from node B to node D.

The traffic pattern tuple of node A is described by ($start = 80, stop = 790, period = 36$). To reserve the ReSF recurrent cells, node A sends its first ReSF 6P `ADD` message in the `SHARED` cell at ASN 60. Node C responds at its first opportunity in the next `SHARED` cell at ASN 66. To construct a minimal-latency path to the root, node C forwards the reservation to node R, to which the root responds at ASN 78. The ReSF 6P `ADD` reservation from node A to node C contains 6 possible reservation tuples, which is the maximum number of tuples a 6P `ADD` can contain, and that are picked based on the lowest calculated collision rate. Out of these 6 tuples, node C picks the one with the lowest collision rate and the closest to the original start ASN, i.e., ($start = 81, stop = 790, period = 36$). From node C to node R the ideal reservation tuple would be

($start = 82, stop = 790, period = 36$) as we assume that the processing of a packet takes less than one cell duration. However, because of the schedule collision that would happen at ASN 82 (with the transmission from node D to R), reservation tuple ($start = 83, stop = 790, period = 36$) has a better collision rate and is agreed upon.

Now assume that due to external interference, both transmissions from node B to D at ASN 80 and 81 fail. Because of those failures, the generated data packet of node B ends up in the queue two slotframes in a row. The next housekeeping of node B notices that two out of three times there was an extra packet in the queue at the end of the slotframe. As such, it sends out a reservation at ASN 90 for a housekeeping cell which gets confirmed at ASN 96. Because of this extra cell the queued packet gets forwarded to node D and after that, during the third housekeeping round, the node decides that the housekeeping cell can be deleted.

4.3. Scheduling Function Description

The ReSF scheduling function allows nodes to add new recurrent reservations for new data-generating sensor applications, remove deprecated reservations when applications stop generating data and update reservations when an application for example changes the period after which it generates data. To do so, ReSF assumes a source node knows its periodic traffic pattern, e.g., reported by a sensing application, and gets cross-layer updates when this pattern changes. In this section, the different functionalities are described in detail.

4.3.1. Scheduling Reservations

We assume that each node periodically generates data for transmission, defined by a (*start*, *stop*, *period*) tuple. ReSF uses the procedures in Algorithm 1 to, respectively, send a new reservation and/or respond to a received reservation request.

The `ScheduleRequest` procedure schedules a new reservation based on the traffic tuple received from the application layer or the tuple derived from an incoming ReSF reservation. It looks for `max_tuples` tuples that equals 6, which is the maximum number of reservations that fit in a 6P `ADD` message, as explained in Section 4.7. To determine the number of tuples that should be reserved to anticipate packet loss, `GetReqTuples` takes into account the link quality as explained in Section 4.4. Using the `GetTuplePool` procedure, all tuples in the interval [($start + 1$, $stop$, $period$), ($start + 1 + reservation\_buffer$, $stop$, $period$) are returned. By starting from $start + 1$, ReSF makes sure that a new reservation is reserved after the packet is generated or received from the previous hop: this way cells are *daisy-chained* over multiple hops towards the destination. The `PopBestTuple` procedure returns the reservation tuple with the lowest calculated *schedule collision rate* (cf. Section 4.5), after checking for collisions with all already activated reservations on that node. The collision rate is the total number of (unique) collisions divided by the total number of planned transmissions for that reservation. Thus, the *reservation_buffer* value represents the trade-off between looking for a minimal collision rate and the risk of introducing more delay by picking a reservation tuple with a large gap between the time slot at which the packet is generated and the actual time slot at which the recurrent packet will be sent. In Section V-B1, we experimentally determine the best *reservation_buffer* value. Finally, the ReSF request with the chosen candidate tuples is sent to the next hop.

The `ReceiveRequest` procedure is straightforward: when it receives an ReSF reservation request, it chooses `num_tuples` tuples from the received `tuple_list`, by using the `PopBestTuple` procedure. The receiver acknowledges the sender with this list of `num_tuples` tuples. If the node is not the final destination of the ReSF reservation, it prepares to forward a new reservation, using `ScheduleRequest`.

4.3.2. Unscheduling and updating reservations

When a sensor application stops generating data, all the reserved ReSF slots towards the reservation destination node should also be removed to avoid energy waste. The procedure for removing cells is fairly simple: a node transmits a delete message to the next hop (towards the destination node), complemented with the ReSF reservation `ID`. When the next hop receives this reservation removal message it will forward a delete message with the same `ID` to its next hop which is repeated until the delete message reaches the destination node and all ReSF reservations with that `ID` are removed.

Every reservation has a *stop* value at which the node stops generating data. When this *stop* ASN is reached, the reservation gets removed and should be renewed when a node wants to send additional data. Reservations also get removed when they are unused for a fixed amount of time.

It may happen that an ReSF reservation needs to be updated, for example when the sensor data generation periodicity changes. In that case, the node sends a new reservation message to its next hop with the same ID (of the old reservation that needs to be updated) but it will propose a new reservation tuple. This avoids the delay and energy waste of first removing the slots via delete messages and only afterwards reserving new cells. When sending the updated reservation message the same reservation process applies as described earlier in this section. However, there is one additional step: when a node and its next hop agree on a new set of reservation tuples for that particular `ID`, the old reservation is deleted automatically.

4.4. Anticipating Packet Loss

To deal with possible packet loss (due to interference) in advance, instead of reserving one recurrent cell per packet, ReSF by default allocates a number of recurrent cells equal to the ceiled ETX value of the link to its next hop, $num\_tuples = \lceil ETX(link_{nextHop}) \rceil$, which is calculated in the `GetReqTuples` in Algorithm 1. ETX is the expected number of transmissions a packet needs to reach its destination. When forwarding the reservation message, its next hop will apply the same formula for allocating a number of cells to its next hop and so on. Doing this over-provisioning, ReSF copes with possible packet loss due to bad link quality.

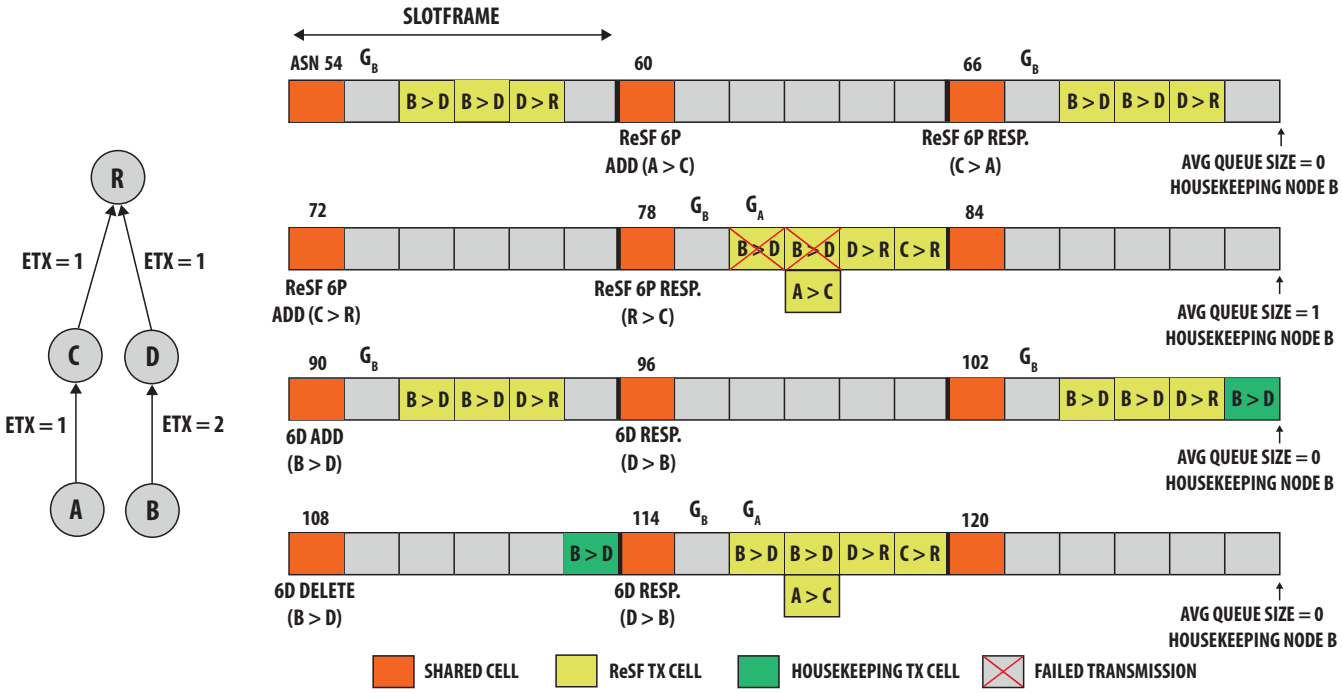


Figure 4: ReSF scheduling example with two nodes A and B generating traffic. The schedule represents an aggregation of all individual schedules. The queue size and how this affects the housekeeping of node B is also shown.

4.5. Preventing Schedule Collisions

When multiple transmitting and/or receiving ReSF reservation tuples, located in the same node, each expect the exact same cell to be activated at a specific ASN, this is called a ReSF *schedule collision*. Allowing multiple tuples to use the same cell will result in packet transmission/reception failures whereby those packets are queued to be retransmitted and the delay inevitably increases. In order to minimize this delay, such schedule collisions should be kept to a minimum. However, detecting collisions between recurrent reservations that only reoccur every so many slotframes and at different time slots in those slotframes, is much harder than detecting collisions between traditional reservations. The latter ones persist every slotframe at the same time slot, which makes identifying collisions trivial. In this section, we present an efficient algorithm to search for reservation tuples with the least amount of schedule collisions.

To efficiently calculate the number of collisions and the collision ASNs of two reservations $(start_1,\ stop_1,\ period_1)$ and $(start_2,\ stop_2,\ period_2)$, we search for the solution to:

$$\forall x, y \in \mathbb{N} : start_1 + period_1 \cdot x = start_2 + period_2 \cdot y \quad (12)$$

in the interval $[start_{max},\ stop_{min}]$ with $start_{max} = max(start_1, start_2)$ and $stop_{min} = min(stop_1, stop_2)$. Equation 12 can be rewritten into the standard form of a linear Diophantine equation $ax + by = c$:

$$period_1 \cdot x - period_2 \cdot y = start_2 - start_1 \quad (13)$$

A linear Diophantine equation has solutions if and only if the $\gcd(a, b)|c$. If not, this means there are no collisions between the two reservations and the reservation is accepted as valid. If $\gcd(a, b)|c$, the equation has infinitely many solutions. The solution (x_0, y_0) can be calculated using the extended Euclidean algorithm. Using (x_0, y_0) and the Diophantine solution standard form, one can calculate all solutions:

$$\forall n \in \mathbb{N} : x = x_0 + n \cdot \frac{period_2}{gcd(period_1, period_2)} \quad (14)$$

$$\forall n \in \mathbb{N} : y = y_0 + n \cdot \frac{period_1}{gcd(period_1, period_2)} \quad (15)$$

We are interested in the values of n where the two reservation sequences can collide:

$$start_{max} \leq start_1 + period_1 \cdot (x_0 + n \cdot \frac{period_2}{gcd(period_1, period_2)}) \leq stop_{min} \quad (16)$$

Based on Equation 16, we calculate the lower- and upper-bound of the n values that we should consider:

$$n_{start} = (\frac{start_{max} - start_1}{period_1} - x_0) \cdot \frac{\gcd(period_1, period_2)}{period_2} \quad (17)$$

$$n_{stop} = (\frac{stop_{min} - start_1}{period_1} - x_0) \cdot \frac{\gcd(period_1, period_2)}{period_2} \quad (18)$$

The number of schedule collisions *numCollisions* between the two reservations $(start_1, stop_1, period_1)$ and $(start_2, stop_2, period_2)$ equals the numbers of integers in the

Algorithm 1 Scheduling and receiving ReSF reservation requests.

```
1:  procedure ScheduleRequest(start, stop, period, id, dest)
2:      max_tuples ← 6                                   ▷ Based on 6P ADD length
3:      num_tuples ← getReqTuples(getNextHop(dest))      ▷ Required nr. of tuples, based on link quality
4:      tuple_pool ← getTuplePool(start + 1, stop, period, resv_buffer)   ▷ Tuples to select from
5:      tuple_list ← list
6:      while size(tuple_list) ≠ max_tuples do           ▷ Find max_tuples tuples to send
7:          append(popBestTuple(tuple_pool), tuple_list)   ▷ Select tuple with lowest collision rate
8:      end while
9:      SendRequest(tuple_list, num_tuples, id, dest)    ▷ Forward the suggested tuple list to the next hop
10: end procedure
11: procedure ReceiveRequest(tuple_list, num_tuples, id, dest)
12:     ack_tuple_list ← list
13:     while size(ack_tuple_list) ≠ num_tuples do       ▷ Find num_tuples
14:         append(popBestTuple(tuple_list), ack_tuple_list)   ▷ Select tuple with lowest collision rate
15:     end while
16:     Respond(ack_tuple_list, id)                      ▷ Respond with the acknowledged tuple list
17:     if ShouldForward(dest) then
18:         new_start ← getLatestTuple(ack_tuple_list)   ▷ Get start value of latest tuple
19:         ScheduleRequest(new_start, stop, period, id, dest)   ▷ Forward the reservation
20:     end if
21: end procedure
```

interval $[n_{start}, n_{stop}]$ which is $n_{stop} - n_{start} + 1$. A node can immediately and exactly calculate all collision ASNs, by filling in the integers in the interval $[n_{start}, n_{stop}]$ using:

$$start_1 + period_1 \cdot (x_0 + n \cdot \frac{period_2}{gcd(period_1, period_2)}) \qquad (19)$$

4.6. Queue Housekeeping using eLLSF

While extra recurrent cells are reserved to anticipate inferior link quality and action is taken to limit the number of schedule collisions, packets can still end up in the queue at the end of a slotframe when the number of reserved recurrent slots, including the over-provisioned slots, does not suffice due to more unanticipated packet loss or non-recurrent traffic.

To empty the queue and preventing the queued packets from taking up cells that were meant for other ReSF reservations, we use the distributed scheduling function eLLSF. It uses a periodical *housekeeping* moment at which it reserves a required number of cells for the slotframes to come, i.e., cells that repeat *every* slotframe. To keep the delay of those queued packets as low as possible, ReSF calculates the required number of cells for the next eLLSF houskeeping period by averaging the number of queued packets of every slotframe since the last housekeeping, as to have extra resources to clear the queue and further minimize latency. The housekeeping period is a configurable parameter.

eLLSF is actually an extended version of LLSF [3]. The idea behind LLSF is to daisy-chain receiving and transmitting cells used in a multi-hop path in order to decrease the latency. The authors however only described how LLSF behaves in a multi-hop path where each node has one child and one parent. Based on their description, we extended LLSF and implemented eLLSF so it can deal with multiple children and use it for both up- and downstream reservations.

The process of adding and removing cells in eLLSF is similar to LLSF. However, in contrast to LLSF, eLLSF makes a difference between up- and downstream packets when performing slot reservations and removals. This differentiation is crucial as eLLSF considers the (possible) multiple children of a node.

Scheduling eLLSF cells. If a node wants to reserve n transmit cells to its parent, it uses the following four-step process:

1. For each reception cell from a child, count the number of cells between that reception cell and the previous reception cell of that child.
2. For each child, pick the cell with the *largest gap* to its left: this is the largest amount of cells between two reception cells of that child.
3. Distribute the cells that are to be reserved evenly and randomly among all children. For example, if a node wants to reserve five cells to its parent and has three children, first assign one transmit cell to each of the children of the node. After that, assign the remaining two cells to two random children.
4. For each child that is assigned one or more transmission cells, place the transmit cell(s) as closely as possible to the right of the reception cell with the largest gap of that child.

When making a transmit cell reservation in the other direction i.e., to one of its children, the 3-step LLSF reservation process is used: only the reception cells of the child – to which the reservation is made – are considered when looking for the

largest gap. This way, minimum-delay communication between that child and its parent is encouraged.

Figure 5 shows an example of a eLLSF housekeeping moment. The schedule of node D before and after the housekeeping moment is shown (for simplicity reasons only one channel is used). Node D wants to reserve 4 cells to its parent. Therefore, it first determines the largest gap RX cells of each child. Afterwards, it places a transmission cell as closely as possible to the largest gap reception cell of each child, with one transmission cell to spare (because there are only 3 children). This last transmission cell is randomly assigned to a node: in this case it is assigned to node B and placed in the beginning of the slotframe because there are no empty cells left at the end of the slotframe.

Unscheduling eLLSF cells. When a node has too many cells to a neighbor (i.e., parent or child), it will remove 1 or more cells to that neighbor. Again, eLLSF makes a distinction between removing a cell to a parent or to a child. When removing a transmission slot to a parent, the algorithm looks for the largest gap between a transmission cell to that parent and the reception cells of *all* children. Then it removes the transmission cell with the largest gap to its left. In the case a transmission cell to a child is being removed, we use the unscheduling process of LLSF. It first looks for the largest gap between a transmission cell to that child and the previous reception cell of that same child. Then it removes the transmission cell to that child with the largest gap to its left.

Preventing collisions between eLLSF and ReSF. In order to prevent the eLLSF housekeeping from reserving recurrent cells that were meant for ReSF reservations, a 2-step process is used: (1) during an interval $[current\_time, current\_time + buffer]$, calculate all the cells occupied by active ReSF reservations (with $buffer$ being a preset parameter currently fixed at the length of 10 slotframes), (2) schedule the housekeeping cells using eLLSF while not considering the slots calculated in step 1 as available slots.

4.7. 6P Integration

This section clarifies how to integrate ReSF in the 6P protocol. When looking at the format of a normal 6P `ADD` transaction, a ReSF 6P transaction is very similar: `ADD` and `DELETE` requests contain an additional ReSF `ID` that identifies the reservation, the `Destination` of the reservation and a list of ReSF reservations that have to be added/deleted. The receiver answers with the number of requested ReSF reservation tuples that fit the node best. If the receiver does not agree with any of the proposed tuples, it answers with an empty 6P `RESPONSE` which indicates that the sender should propose other ReSF reservation tuples.

In Figure 6, both a 6P `ADD` message and ReSF reservation format are shown. The maximum length of a 6TiSCH packet is 127 bytes. The IEEE 802.15.4 header has a length of 23 bytes (including the `FCS` field) while the 6top header only has a length of 8 bytes when leaving out the list of cells. This leaves 96 bytes to specify the ReSF reservation. The ReSF `ID` can be represented by a 2-byte integer, the `Destination` by 8 bytes. A ReSF reservation has a length of 14 bytes containing two 5-byte ASN values for the *start* and *stop* value, a 2-byte channel offset and a 2-byte *period* value. This mean that a 6P `ADD` reservation can include 6 reservations. There are 2 bytes left for future use.

Table 3: The default experiment parameters.

Parameter	Value
Grid size	4.5 km x 4.5 km
Inter-node default distance	0.230 km
Nr. of runs per experiment	20
Simulated time	30 min
Frequency	2.4 GHz
Nr. of channels	16
Stable RSSI	−93.6 dBm (PDR ~ 0.5)
Nr. of stable neighbors	1
Avg. nr. of hops (25/100 nodes)	3.4 (σ = 0.4) / 5.8 (σ = 0.4)
Avg. nr. of children (25/100 nodes)	2.2 (σ = 0.4) / 1.7 (σ = 0.1)
Slotframe size	101
Nr. of `SHARED` cells	3
6top housekeeping	False
SF0 threshold	0
RPL parent set size	1
RPL DIO Period	5 s
ReSF reservation_buffer	64
Housekeeping period	10 s
Timeslot duration	10 ms

5. Evaluation

This section evaluates the performance of ReSF and compares it to the state-of-the-art TSCH scheduling functions SF0 and eLLSF. A variety of experiments was conducted while observing several metrics including packet latency and power consumption. First, we present the different experiment parameters. Afterwards, we determine the optimal value of the $reservation\_buffer$. Finally, we show the performance of ReSF with both static and dynamic traffic patterns.

5.1. Simulation Setup

In order to evaluate the performance of ReSF, we used the 6TiSCH simulator developed by the 6TiSCH WG members [18]. This is an open-source, event-driven Python simulator, allowing easy parameter configuration. By default, the simulator supports IEEE 802.15.4e TSCH mode, RPL, 6top and SF0. We extended the simulator with a real message-passing 6P protocol as the implemented 6top sublayer did not support realistic 6P transactions. The presented ILP formulation is solved using Gurobi, which uses a hybrid solution procedure that combines three different approaches to find an exact solution: (1) cutting planes, (2) branch and bound, (3) relaxation and decomposition [19].

During all experiments, the nodes are placed on a grid, with the root node positioned in the center. The (x, y) grid position of each node (except for the root node) is determined at random – with a different seed for each experiment iteration – following the normal distributions $\mathcal{N}(x_0, (d/8)^2)$ and $\mathcal{N}(y_0, (d/8)^2)$ respectively, with (x_0, y_0) being the initial grid position of each

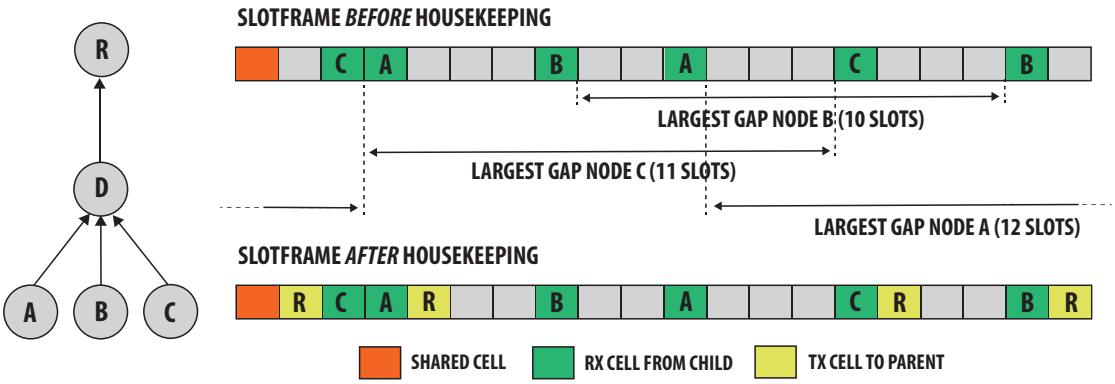


Figure 5: eLLSF housekeeping on node D that wants to reserve 4 transmission cells to its parent.

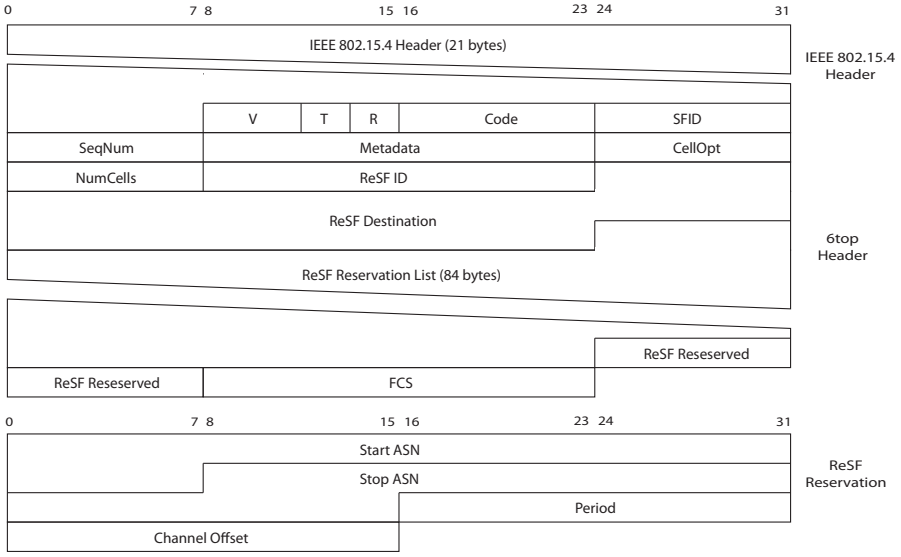


Figure 6: Example of a 6P `ADD` request format for ReSF reservations (maximum length of 127 bytes) and a ReSF reservation that includes the channel offset and the reservation tuple (14 bytes).

node. The distance is d, set to 230 meters, to ensure the topologies did not end up being star topologies, while still maintaining end-to-end multi-hop connectivity between the root and every other node. Generating heterogeneous traffic is done sampling a normal distribution $\mathcal{N}(t, (t/4)^2)$, with t the mean traffic rate, for which 3 different values are used: 1 packet/min, 6 packets/min and 12 packets/min per node. All network traffic is sent to the same sink node: the network root. The default parameter values are summarized in Table 3. The optimal housekeeping period parameter is experimentally determined and is set to 10 s for all three scheduling functions (i.e., SF0, eLLSF and ReSF) as they all employ periodical housekeeping. 6top housekeeping (i.e., relocation of cells) is not enabled as this was not added to the message passing 6P implementation. The 3 `SHARED` cells, that are used to bootstrap the network, are located at the first 3 cells of every slotframe.

All results in this section exclude the initial warm-up period in which the network topology converges, meaning that each node has already selected a preferred parent and has negotiated one dedicated cell to that preferred parent. In case of ReSF, it also means that each ReSF reservation of a node has already reached the root. It is assumed that a link-layer `ACK` message does not fail and that the intermediary processing of a data packet takes less than one cell length (and thus can be forwarded in a cell immediately following the cell in which it was received).

The *latency* metric encompasses the total time it takes a data packet to reach the root, from the moment it is generated. It is important to mention that a packet is only dropped after it has been retransmitted at least 5 times and the queue of the node is full. The *battery power consumption* metric is the aggregated radio power consumption of all the nodes, taking into account idle listening, transmission and reception of each data and acknowledgement packet. The consumption values used in the simulator that correspond to these different actions are based on the work by Vilajosana et al. who provided a realistic TSCH energy consumption model [20]. All experiment results show the mean of 20 (random) iterations and the associated standard error.

5.2. Reservation Buffer

The *reservation_buffer* parameter represents the number of tuples a node will compare to all already activated reservations, when looking for the tuple with the smallest collision rate, as explained in Section 4.3.1. As can be observed in Figure 7, a higher *reservation_buffer* has a significant impact on the network latency when there is a network load of 12 packets/min: taking the optimal value of 64 improves 56 % in delay over not considering any extra tuples (i.e., a value of 0). When looking at lower traffic loads, increasing the buffer leads to higher delays. At lower traffic loads the schedule is less congested and low collision rate tuples are commonly available. Looking at tuples with later *start* ASNs to find an even better collision rate results in an increased latency because of the extra difference in time between the packet arrival and the moment it can be sent to the next hop. The *reservation_buffer* parameter is set to a value of 64 for all experiments. For the simplicity of parameter configuration, we show the results for one *reservation_buffer* value for the different traffic loads. This, of course, has an impact on the lower traffic loads: for 1 packet/min the latency result is 16 % worse than the optimum (i.e., a *reservation_buffer* value of 0) and for 6 packets/min are 6 % worse than the optimal results at a value of 4.

To show the computational impact of choosing a *reservation_buffer* value, we measured the computation time when identifying collisions between two reservation tuples during a certain interval length, i.e., the period in which the two tuples can collide. Figure 8 shows that the computation time linearly increases for increasing interval lengths. To identify an acceptable length, Figure 9 shows the normalized packet collision probability when calculating the collisions between two reservation tuples during different interval lengths. The probability is normalized relative to the collision probability when considering an interval length of 7200 s. Calculating the collisions between two reservation tuples over an interval length of 3000 s is sufficient for all traffic rate means, i.e., for both 6 packets/min and 12 packets/min, the difference in accuracy is negligible (i.e., less than 1 %) when using a length of 3000 s compared to using a length of 7200 s. For a traffic rate of 1 packet/min, the difference is 13.5 % which is considered acceptable. As shown in Figure 8, the computation time for a period of 3000 s is 1 µs. This means that for a single ReSF 6P `ADD` request that can contain 6 tuples and a *reservation_buffer* value of 64, the collision computation time is 0.384 ms for every existing reservation on the node the new request is compared to. However, as can be seen in Figure 9, for higher traffic rate means, comparing over a length of 1000 s already suffices, which would divide the computation time by two.

5.3. Static Traffic

Figure 10 shows the average latency and power consumption for both 25 and 100 nodes in topologies with static traffic patterns, meaning that a node will not change the period with which it generates traffic throughout the experiment.

The results show that taking into account recurrent traffic behavior combined with daisy-chaining the cells up to the root,

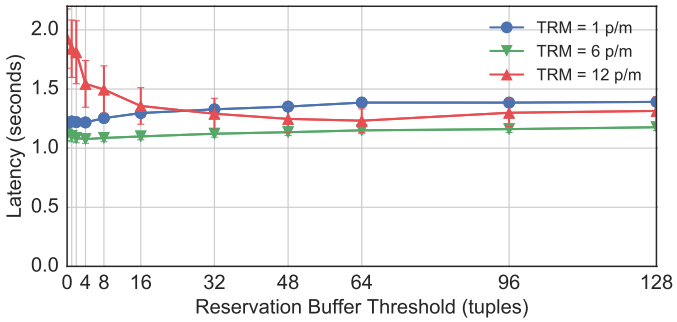


Figure 7: Reservation buffer parameter experiment with 100 nodes.

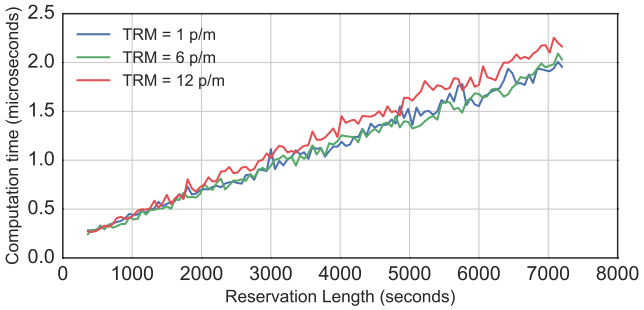


Figure 8: The computation time (in microseconds) of finding the collisions between two reservation tuples for different interval lengths.

results in superior performance in terms of latency. Looking at results of 100 nodes with the traffic load means of 1 packet/min, 6 packets/min and 12 packets/min per node, the relative latency improvements of ReSF over eLLSF are respectively 76 % (from 5.74 s to 1.40 s), 80 % (from 5.77 s to 1.16 s) and 78 % (from 5.53 s to 1.23 s). Looking at the ReSF results for 100 nodes when traffic increases, the latency is almost constant. This means that because of the traffic behavior-aware reservations ReSF is well equipped to deal with different traffic rates. Looking at the results in Figure 10a, the latency of ReSF is even slightly decreasing at higher mean traffic rates. This is because ReSF defines the number of cells needed for a packet transmission as the *ceiled* ETX value per link which is dynamically determined based on the number of retransmissions. For ex-

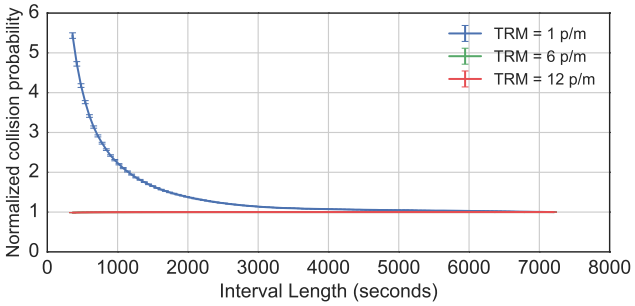
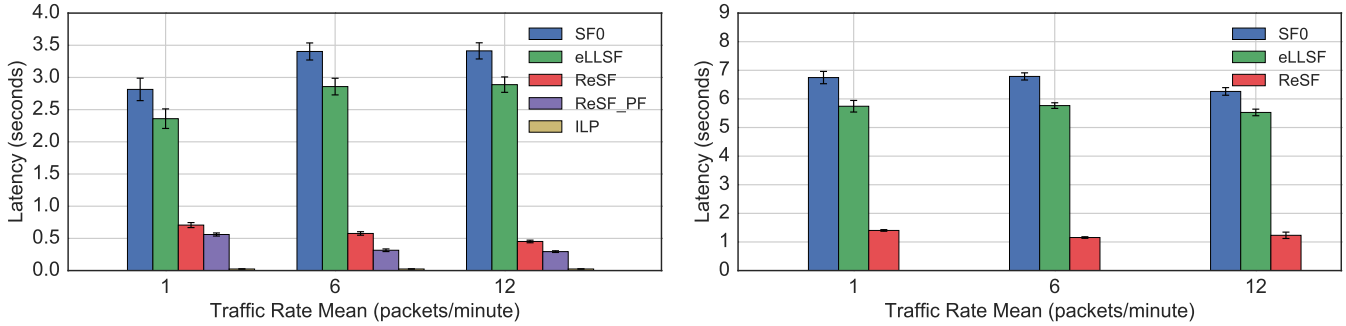
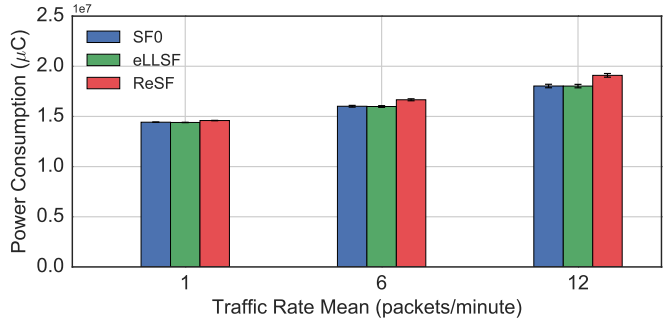


Figure 9: The normalized probability, i.e., relative to the probability based on a interval length of 7200 seconds, for different interval lengths.

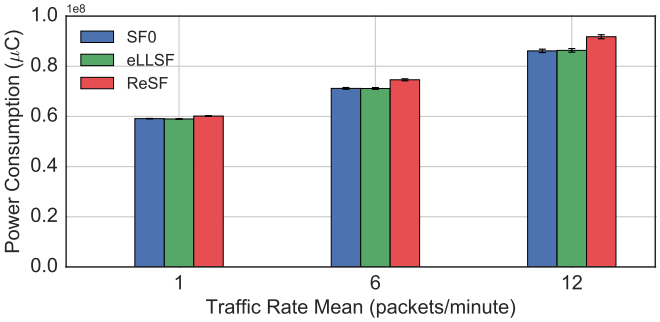


(a) Latency for 25 nodes.

(b) Latency for 100 nodes.



(c) Power consumption for 25 nodes.



(d) Power consumption for 100 nodes.

Figure 10: Results for static traffic with 25 and 100 nodes, comparing ReSF to SF0 and eLLSF, as well as the optimal solution obtained from solving the ILP formulation (the latter for 25 nodes only), as a function of the average traffic generation period.

Table 4: Comparison of eLLSF and ReSF latency and packet loss values for a traffic rate mean of 12 packets/ min for different network sizes.

Size	eLLSF		ReSF	
	Latency (s)	Loss (%)	Latency (s)	Loss (%)
25	2.89	0.14	0.45	0.0
100	5.53	1.7	1.23	0.8
150	6.93	4.08	1.94	4.24
200	8.44	9.02	2.83	9.78

ample, if a link has an ETX value of 1.1, ReSF will reserve 2 cells per link per packet sent, i.e., an over-provisioning of 1 cell. However, with an ETX value of 1.1 the probability that the packet will actually need two cells is rather small. This means that the over-provisioned cell(s) can be used by other packets. This effect is magnified when dealing with higher traffic rates as there will be more over-provisioned cells. Combined with additional packet losses at higher traffic rates, this explains the slightly improved latency results when increasing the traffic rate mean.

As expected, due to the daisy-chained paths scheduled by eLLSF, it also improves over the random reserved cells of SF0 with respectively 15 %, 15 %, and 12 %. Packet loss is minimal for all 3 scheduling functions: when considering the high traffic load scenario of 12 packets/ min with 100 nodes, ReSF has the least amount of packet loss with 0.8 % of the packets lost, followed by eLLSF with 1.7 % and SF0 with 1.8 % packet loss. Considering the other traffic loads, the packet loss of ReSF is negligible at a maximum of only 0.04 %. For eLLSF and SF0,

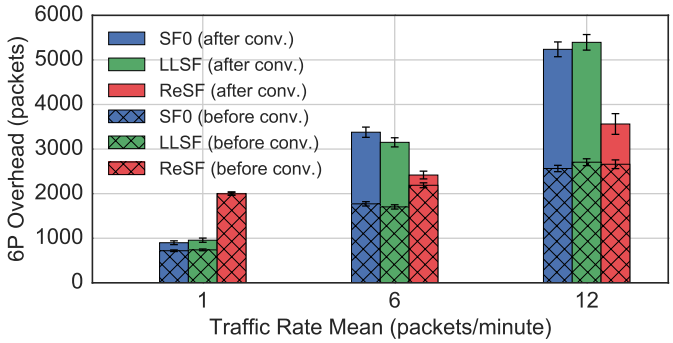
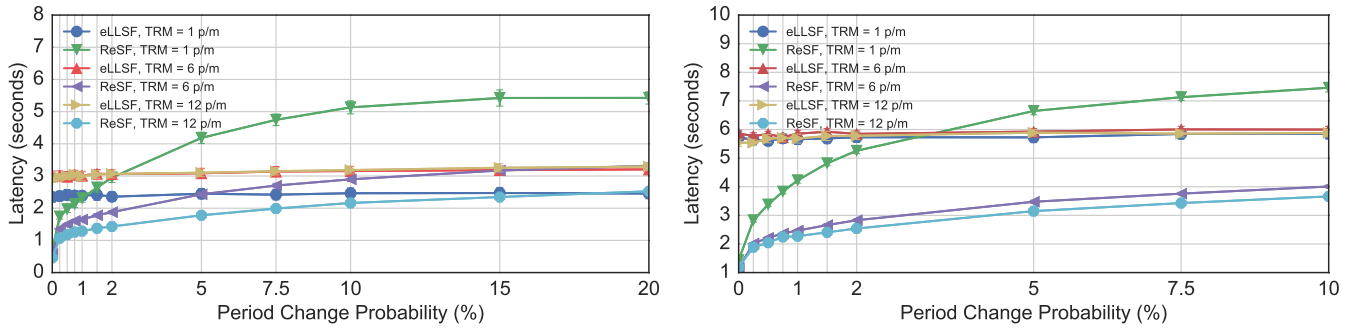


Figure 11: Number of sent 6P messages, before (i.e., hatched bars) and after the network convergence.

the maximal packet loss values are 0.6 % and 0.7 %.

Table 4 shows that the same trend holds for larger network sizes up to 200 nodes and a traffic rate mean of 12 packets/ min. For 150 and 200 nodes, ReSF has an average latency of 1.94 s and 2.83 s respectively, while eLLSF has an average latency of 6.93 s and 8.44 s respectively. However, the packet losses introduced by ReSF increase to 4.24 % for 150 nodes and to 9.87 % for 200 nodes, while for eLLSF the packet loss increases to 4.08 % and 9.02 % respectively. These results show that while ReSF has slightly more packet losses compared to eLLSF when the network scale increases, ReSF scales significantly better than eLLSF in terms of latency.

Figure 10a also shows the optimal solution. It is calculated by solving the ILP formulation of the recurrent traffic prob-



(a) Latency for 25 nodes.

(b) Latency for 100 nodes.

Figure 12: Results for dynamic traffic with 25 and 100 nodes, comparing ReSF to eLLSF, as a function of the probability that the traffic generation period of each node changes every second.

lem, as defined in Section 3.2. Optimal results for 100 nodes are omitted, due to the exponential execution time increase in terms of network size. As the ILP formulation assumes perfect link conditions without interference (i.e., $ETX = 1$), the graph also shows ReSF results under perfect network conditions (i.e., RESF_PF) for a more fair comparison. The ILP results average around 25 ms which is significantly better than the normal ReSF results or the ReSF experiment with perfect links of which the latency for all traffic rates averages around 250 ms. The reason that the ILP solution has a latency about 10 times lower than ReSF_PF is because its result is the theoretical optimal solution that avoids schedule collisions using perfect global information, and it does not take into account interference or signaling overhead.

The power consumption results show that there is only a minimal impact of ReSF on battery life: a maximum increase of 6.3 % increase over eLLSF (and 6.54 % over SF0) for a traffic load of 12 packets/min per node and a minimum of 1.96 % (and 1.74 % over SF0) for a traffic load of 1 packet/min. The fact that there is a slight increase in battery consumption is due to the extra cells that ReSF reserves in the slot frames when traffic is expected. For each scheduling function, all nodes in the network have – by default – a dedicated transmission cell to their parents to allow fast management communication (i.e., 6P transactions) and also data transmissions. Next to those default reserved dedicated cells, ReSF also reserves recurrent cells that are only activated when traffic is expected and housekeeping cells to empty the queue of any remaining traffic. SF0 and eLLSF also have to reserve extra cells when dealing with unanticipated high traffic loads and because they do not anticipate the recurrent traffic with recurrent cells they have to send significantly more 6P overhead as shown in Figure 11. The figure shows the number of sent 6P packets before and after network convergence. For the lowest traffic load, ReSF has higher total overhead, but this is only because of the sent ReSF reservations during network convergence. When the network has converged, the overhead of SF0 and eLLSF is significantly higher.

5.4. Dynamic Traffic

In these experiments, ReSF and eLLSF are tested on how they deal with dynamic traffic. Every second every node has a probability that its traffic generation period changes. When the traffic generation period changes, the new period is drawn from the sample normal distribution out of which the original traffic period was drawn.

The results in Figure 12 show that eLLSF is not affected by the traffic period changes. Because the changes in traffic period are not that significant, the housekeeping of eLLSF does not continuously need to send 6P `DELETE` and `ADD` to adjust the number of resources. The number of 6P messages thereby equals the overhead as if there were no traffic period changes, leading to similar latency results. This is in contrast to ReSF, where for each traffic period change at a node, a new recurrent reservation needs to be forwarded to the root. This additional ReSF 6P overhead will hold up more data packets in the queue, which results in additional 6P ReSF housekeeping overhead. All this extra overhead decreases the performance of ReSF. However, the graphs show that for 25 nodes, 12 packets/min and a probability up to 20 % (meaning that, on average, every second 5 nodes change their traffic period), ReSF can deal with these dynamic traffic periods and it improves on latency while maintaining a throughput equal to eLLSF. The graph shows that ReSF actually deals better with the changing traffic periods when the traffic rate mean is higher. This is because when the traffic rate is higher, recurrent cells are more frequently available to forward the updated 6P ReSF reservations and thus less data packets are obstructed from being forwarded. When observing 100 nodes at 12 packets/min, ReSF performs better than eLLSF up to somewhere between 1.5 % and 2 %. While the latency graph shows a significant improvement by ReSF, the throughput results showed that at 1.5 % the throughput of ReSF is better than eLLSF while at 2 % it was slightly less (i.e., ReSF had 0.28 % less throughput).

It is important to note that such frequent traffic rate changes are unlikely for most real-world sensing applications, such as temperate or heart rate measurements. For example, in a topology with 100 nodes a 2 % change probability means that it is expected that 2 out of 100 nodes change their traffic period every second and every node would change it every 50 s on average. Considering a generation mean of 1 packet/min such a change probability becomes even more unrealistic as a node

would change its generation period faster than it generates an actual packet. So, a probability of 2 % is very high which means that ReSF is well-equipped to handle dynamic traffic rates.

6. Conclusion

In this work, we explicitly focused on minimizing the latency of recurrent traffic in WSNs. We presented two novel contributions. First, we stated the problem of minimal-latency scheduling of recurrent transmissions formally, using an ILP. Second, we presented ReSF, a distributed TSCH scheduling function, specifically designed for IoT applications with recurrent traffic, such as sensor measurement applications. ReSF builds a minimal-latency path from source to root and activates the recurrent cells on this path only when traffic is expected, and deactivates them immediately afterwards. We have conducted numerous experiments using the 6TiSCH simulator, comparing ReSF both to SF0 and eLLSF. The results show significant performance improvements. When considering 100 nodes and each node having a static traffic pattern, ReSF improves up to 80% in terms of latency compared to eLLSF while only having a minimal power consumption impact of at most 6.3%. We have also experimentally shown that ReSF can handle a per-second traffic rate change probability up to 20 % when considering 25 nodes and between 1.5 % and 2 % in a topology of 100 nodes. Traffic rate changes in most real-world sensor applications are typically much less dynamic. We conclude that ReSF is well-equipped to maintain a minimal delay in both static and dynamic recurrent traffic rate scenarios.

For future work, ReSF will be integrated into a TSCH firmware implementation in order to deploy and test it in a real-world TSCH network.

Acknowledgment

Part of this research was funded by the Flemish FWO SBO S004017N IDEAL-IoT (Intelligent DEnse And Long range IoT networks) project, and by the ICON project CONAMO. CONAMO is a project realized in collaboration with imec, with project support from VLAIO (Flanders Innovation and Entrepreneurship). Project partners are imec, Rombit, Energy Lab and VRT.

References

[1] D. Dujovne, T. Watteyne, X. Vilajosana, P. Thubert, 6TiSCH: Deterministic IP-enabled Industrial Internet (of Things), IEEE Communications Magazine 52 (2014) 36–41.

[2] M. R. Palattella, P. Thubert, X. Vilajosana, T. Watteyne, Q. Wang, T. Engel, 6TiSCH Wireless Industrial Networks: Determinism Meets IPv6, Springer International Publishing, Cham, pp. 111–141.

[3] T. Chang, T. Watteyne, Q. Wang, X. Vilajosana, LLSF: Low Latency Scheduling Function for 6TiSCH Networks, in: 2016 International Conference on Distributed Computing in Sensor Systems (DCOSS), pp. 93–95.

[4] X. Vilajosana, K. Pister, T. Watteyne, Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration, RFC 8180, 2017.

[5] T. Watteyne, S. Lanzisera, A. Mehta, K. S. J. Pister, Mitigating Multipath Fading through Channel Hopping in Wireless Sensor Networks, in: 2010 IEEE International Conference on Communications, pp. 1–5.

[6] Q. Wang, X. Vilajosana, T. Watteyne, 6top Protocol (6P), Internet-Draft draft-ietf-6tisch-6top-protocol-08, Internet Engineering Task Force, 2017. Work in Progress.

[7] D. Dujovne, L. A. Grieco, M. R. Palattella, N. Accettura, 6TiSCH 6top Scheduling Function Zero / Experimental (SFX), Internet-Draft draft-ietf-6tisch-6top-sfx-00, Internet Engineering Task Force, 2017. Work in Progress.

[8] M. Zhang, C. E. Perkins, S. Anand, S. Anamalamudi, A. R. Sangi, Scheduling Function One (SF1) for hop-by-hop Scheduling in 6tisch Networks, Internet-Draft draft-satish-6tisch-6top-sf1-03, Internet Engineering Task Force, 2017. Work in Progress.

[9] M. R. Palattella, N. Accettura, M. Dohler, L. A. Grieco, G. Boggia, Traffic Aware Scheduling Algorithm for reliable Low-Power multi-hop IEEE 802.15.4e Networks, in: 2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC), pp. 327–332.

[10] A. A. Faras, D. Dujovne, A Queue-based Scheduling Algorithm for PCE-enabled Industrial Internet of Things Networks, in: 2015 Sixth Argentine Conference on Embedded Systems (CASE), pp. 31–36.

[11] T. Huynh, F. Theoleyre, W.-J. Hwang, On the Interest of Opportunistic Anycast Scheduling for Wireless Low Power Lossy Networks, Comput. Commun. 104 (2017) 55–66.

[12] A. Morell, X. Vilajosana, J. L. Vicario, T. Watteyne, Label switching over IEEE802.15.4e networks, Transactions on Emerging Telecommunications Technologies 24 (2013) 458–475.

[13] F. Theoleyre, G. Z. Papadopoulos, Experimental Validation of a Distributed Self-Configured 6TiSCH with Traffic Isolation in Low Power Lossy Networks, in: Proceedings of the 19th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM '16, ACM, New York, NY, USA, 2016, pp. 102–110.

[14] N. Accettura, E. Vogli, M. R. Palattella, L. A. Grieco, G. Boggia, M. Dohler, Decentralized Traffic Aware Scheduling in 6TiSCH Networks: Design and Experimental Evaluation, IEEE Internet of Things Journal 2 (2015) 455–470.

[15] E. Municio, S. Latré, Decentralized Broadcast-based Scheduling for Dense Multi-hop TSCH Networks, in: Proceedings of the Workshop on Mobility in the Evolving Internet Architecture, MobiArch '16, ACM, New York, NY, USA, 2016, pp. 19–24.

[16] R. Soua, P. Minet, E. Livolant, Wave: a Distributed Scheduling Algorithm for Convergecast in IEEE 802.15.4e TSCH Networks, Transactions on Emerging Telecommunications Technologies 27 (2016) 557–575. Ett.2991.

[17] S. Duquennoy, B. Al Nahas, O. Landsiedel, T. Watteyne, Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH, in: Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys '15, ACM, New York, NY, USA, 2015, pp. 337–350.

[18] T. Watteyne, K. Muraoka, N. Accettura, X. Vilajosana, 6TiSCH Simulator, `https://bitbucket.org/6tisch/simulator`, 2017.

[19] I. Gurobi Optimization, Gurobi Optimizer Reference Manual, 2016.

[20] X. Vilajosana, Q. Wang, F. Chraim, T. Watteyne, T. Chang, K. S. J. Pister, A realistic energy consumption model for tsch networks, IEEE Sensors Journal 14 (2014) 482–489.