Efficient Recurrent Low-Latency Scheduling in IEEE 802.15.4e TSCH Networks

Glenn Daneels, Steven Latré, Jeroen Famaey
IDLab, University of Antwerp - imec
Antwerp, Belgium
firstname.lastname@uantwerpen.be

***Abstract*—Recently, the interest in the Time-Slotted Channel Hopping (TSCH) mode of the IEEE 802.15.4e MAC layer has grown significantly. It is used especially in both Internet-of-Things (IoT) and industrial networks. TSCH uses channel hopping to increase link reliability and manages a time slotted schedule that instructs every node which action to take to allow for low-power operation. In our previous work we introduced the distributed Recurrent Low-Latency Scheduling Function (ReSF) that manages the TSCH schedule in such a way that it is optimized for power-constrained sensor networks where each node periodically sends recurrent measurement data to a sink and low-latency dissemination of the data is required. This work presents a more efficient version of ReSF and introduces three new contributions. First, we provide heuristics for the collision solving algorithm of ReSF. Second, we provide an additional feature to better avoid schedule collisions. Finally, we add increased support for sporadic traffic in ReSF networks. Extensive experimental results show a latency improvement up to 45.3 % with our previous work and up to 65.1 % with a state-of-the-art low-latency scheduling function.**

***Index Terms*—IEEE 802.15.4e, Time-Slotted Channel Hopping,MAC scheduling, 6TiSCH**

I. Introduction

In recent years, the TSCH mode of the IEEE 802.15.4e MAC layer has gained a lot of interest in both IoT and industrial networks. TSCH is used especially in IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) networks that bridge the gap between deterministic industrial networks and a traditional IPv6-enabled upper stack. It uses channel hopping to increase link reliability and minimize the effects of external interference and/or multi-path fading and manages a time slotted schedule that at any moment in time instructs each node exactly what to do (e.g., transmit, receive, sleep) to allow for low-power operation.

Both the 6TiSCH Working Group (WG) and other researchers have defined scheduling solutions to manage that schedule and thereby optimize different metric (e.g., latency, power consumption) for different types of applications. In our previous work we introduced the Recurrent Low-Latency Scheduling Function (ReSF). It focuses on power-constrained sensor networks where each node periodically sends measurement data to a sink (i.e., recurrent traffic) and demands low-latency dissemination of these data [1]. In contrast to other scheduling approaches, ReSF takes the recurrent traffic behavior into account when allocating resources and reserves a daisy-chained, minimal-latency path from source to sink that is only activated when traffic is expected. This results in low packet delays while maintaining low-power operation.

This article presents a more efficient version of the previously proposed ReSF and proposes three specific improvements. First, we provide heuristics as an alternative for the computationally-intensive schedule collision solver algorithm that is used in the original ReSF. Second, we propose an additional feature to avoid schedule collisions as to improve the packet loss caused by these scheduled collisions. Finally, we add better support for sporadic traffic (i.e., non-recurrent) in the ReSF networks. Additionally, we provide extensive simulation results based on the official 6TiSCH simulator [2].

The remainder of this article is structured as follows. First, we introduce the related work on 6TiSCH scheduling functions, specifically focusing on distributed solutions in Section II. Second, in Section III, we overview the most important concepts of the original ReSF. Furthermore in that section, we explain in details the contributions of this work. Consecutively, these contributions are evaluated in Section IV. Finally, Section V presents the conclusion of this work.

II. Related Work

A well-known 6TiSCH distributed scheduling function is the Minimal Scheduling Function (MSF) introduced by the Internet Engineering Task Force (IETF) 6TiSCH Minimal Scheduling Function draft [3]. MSF is a distributed scheduling algorithm that dynamically adapts its resources based on the use of its currently allocated resources. It randomly allocates resources in a slotframe in contrast to ReSF that daisy-chains the cells up to the destination. Chang et al. present the Low Latency Scheduling Function (LLSF) scheduling function that daisy-chains cells over the different links up to the root [4]. Morell et al. [5] have proposed a combination of 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. Theoleyre et al. [6] focus on traffic isolation by introducing tracks for different applications and consider contiguous reserved cells as well as random reserved cells. Escalator [7] also aims to daisy-chains timeslots for convergecast purposes and does this in an autonomous manner i.e., without additional signaling. In contrast to ReSF, all these works do not consider the recurrent behavior or sensor data.

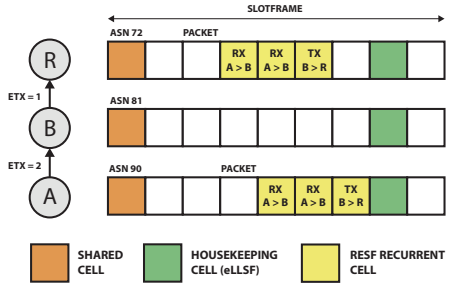

Fig. 1. ReSF scheduling example with node A generating traffic. The schedule shown is the schedule of node B.

III. Recurrent Low-Latency Scheduling

In this section we detail the exact implementation of the ReSF scheduling function. First, we review the original ReSF scheduling function as proposed in our previous work [1]. Afterwards, the collision solver heuristics, a new approach for collision avoidance and support for sporadic traffic are discussed in detail.

A. General Overview

The original ReSF was designed to minimize the latency of periodic data transmissions. The four most important features are outlined below. A detailed description can be found in [1].

a) Scheduling an ReSF reservation: ReSF assumes a source node knows its periodic traffic pattern, which it uses to construct a *recurrent* path. This path consists of recurrent cells which are cells that are only activated in slotframes when traffic is expected and are deactivated immediately after. An ReSF reservation is based on the following tuple: $(start, period)$. $Start$ is the Absolute Sequence Number (ASN) at which the first data packet is generated on the source node and $period$ represents the periodicity in ASNs 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 not explicitly reserved for one particular traffic stream: if a node makes an 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. Figure 1 shows the schedule of node B in a topology of three nodes in which node A transmits sensor data to node R every 19 time slots starting at ASN 74. Therefore it sent an ReSF reservation with tuple $(start = 74, period = 19)$ to node B that forwarded it to node R. This results in an ReSF daisy-chained path from A to node R for the reservation tuple. Node B daisy-chained the TX tuple to node R just after the RX slots in which he receives from node A.

b) Preventing schedule collisions: A *schedule collision* is caused when multiple ReSF reservations, located in the same node, want to occupy the same cell at a particular ASN. The original collision solver that aims to prevent these collisions, and the new proposed heuristics are detailed in Section III-B.

c) Anticipating packet loss: ReSF anticipates packet loss by reserving back-up tuples. The number of extra reservations depends on the measured link quality (i.e., Expected Transmission Count (ETX)) to allow a node to retransmit multiple times consecutively. As seen in Figure 1, node A reserved two cells to node B because the ETX of the link equals two.

d) Queue housekeeping using Enhanced Low Latency Scheduling Function (eLLSF): Additionally to anticipating packets loss, the original ReSF allowed allocating extra cells employing the principle of eLLSF [1], to prevent failing packets from congesting the queue. Figure 1 shows that node B has one additional eLLSF cell allocated to node R, to empty the queue from possibly failed packets. In Section III-D, we discuss which algorithm triggers the (de-)allocation of these *back-up* cells and we detail the new queue housekeeping approach.

B. Fast Collision Solving

When two or more ReSF reservations on the same node want to have the exact same cell activated, this is called a ReSF *schedule collision*. More specifically, this means that multiple children and/or the parent of the node are expecting to send to the node or receive a packet from the node at the same ASN. When these multiple transmissions happen at the same time, the probability for packet collisions increases, leading to TX/RX failures and subsequently packet loss and delay. Detecting collisions between recurrent reservations that only reoccur every so many slotframes and at different slots in those slotframes, is much harder than detecting collisions between traditional cell reservations that persist every slotframe at the same time slot, which makes identifying collisions trivial.

To avoid the negative effects of schedule collisions, an ReSF node aims to keep these schedule collisions to a minimum by applying a collision solver algorithm to a list of possible candidate tuples. Whenever a node proposes a new reservation tuple to another node or when a node receives an ReSF request, the node selects the reservation tuple candidate with the lowest number of schedule collisions as calculated by the algorithm.

Below we explain how we calculate the number of unique collisions between one and multiple other ReSF tuples. Afterwards, we introduce less computationally-intensive heuristics.

1) Exact Collision Solving with Multiple Tuples: In this section we explain how to calculate the exact number of collisions between one tuple and multiple other tuples. We make use the algorithm we explained in our previous work on how to calculate the exact number of collisions between two ReSF reservation tuples [1].

If we calculate the exact number of unique collisions between a new candidate reservation tuple $(start_k, period_k)$ and all other already installed $k-1$ tuples on that node, we have to pair-wise calculate the collisions of the tuple $(start_k, period_k)$ with every other reservation tuple $(start_i, period_i)$, as shown in the `calcNrUniqueCollisions` procedure in Algorithm 1. When determining the collisions between

Algorithm 1 Exact Collision Solving with Multiple Tuples

```
1: procedure CALCNRUNIQUECOLLISIONS(start, period, tupleSet, startMax, LCM)
2:     ASNList ← LIST
3:     for each t ∈ tupleSet do
4:         nStart, nStop ← ...                                   ▷ See definition in [1]
5:         for each n ∈ [nStop, nStart] do
6:             ASN ← start + period · (x0 + n · −t.period / GCD(period, −t.period))   ▷ x0 calculated by the ext. Euclidean algorithm
7:             if ASN ∉ ASNList then
8:                 APPEND(ASN, ASNList)
9:     return LENGTH(ASNList)
```

Algorithm 2 Collision Solving Heuristic Using Summation

```
1: procedure CALCSUMCOLLISIONS(start, period, tupleSet, startMax, LCM)
2:     sum ← 0
3:     for each t ∈ tupleSet do
4:         nStart, nStop ← ...                                   ▷ See definition in [1]
5:         sum ← sum + ⌊nStart⌋ − ⌈nStop⌉ + 1
6:     return sum
```

two tuples, we calculate the collisions in the interval $[start_{max}, start_{max} + LCM(period_1, period_2, \ldots, period_k)]$ with $start_{max} = max(start_1, start_2, \ldots, start_k)$ and with $LCM(period_1, period_2, \ldots, period_k)$ the *least common multiple* of the periods. More specifically, we start at $start_{max}$ so every reservation is sending and after a length of $LCM(period_1, period_2, \ldots, period_k)$, the same sequence repeats itself. The candidate reservation tuple $(start_k, period_k)$, the $k-1$ tuples, $start_{max}$ and $LCM(period_1, period_2, \ldots, period_k)]$ are given as parameters to `calcNrUniqueCollisions`. The procedure iterates over every tuple $(start_i, period_i)$ and calculates the n_{start} and n_{stop} for each tuple. These were already defined in our previous work. Then at lines 5 and 6, for every $(start_i, period_i)$, we iterate over all n values in the interval $[n_{stop}, n_{start}]$ and use them to calculate every collision ASN relating to a n value. Subsequently, as show at lines 7 and 8, if the result does not exist yet in the list of unique collisions ASNs, it is added. The returned length of the `ASNList`, at line 9, is the total number of unique collisions of the candidate tuple $(start_k, period_k)$. The node will use this number to compare this candidate with the other candidate tuples and select the best candidate, i.e., the candidate with the lowest number of schedule collisions.

Algorithm 1 shows that the computational performance of the exact collision solving approach is dependent on the size of `tupleSet`, but also of the length of the interval $[n_{stop}, n_{start}]$, i.e., the number of schedule collisions, for each tuple in `tupleSet`.

2) Collision Solving Heuristic Using Summation: We introduce a heuristic to approximate the exact algorithm in terms of collision solving performance, while avoiding the computationally-intensive procedure of calculating all the unique collision ASNs between one and multiple ReSF tuples.

The heuristic is demonstrated in Algorithm 2. In the `calcSumCollisions` procedure, we iterate over all the existing tuples in `tupleSet` and calculate the number of collision ASNs, which is obtained by filling in $\lfloor n_{start} \rfloor - \lceil n_{stop} \rceil + 1$, that the candidate tuple $(start_k, period_k)$ has with each of the $k-1$ tuples. All these totals are aggregated in the `sumCollisions` variable which results in an estimation of the unique number of collisions. Likely, this total is an overestimation as it can include collisions ASNs that were counted multiple times with different tuples.

In contrast to the exact approach, the heuristic is only dependent on the size of `tupleSet`, and not on the number of schedule collisions for each tuple in `tupleSet`, making it computationally more attractive than the exact solution.

3) Collision Solving Heuristic Using Minimal Delay: We introduce another heuristic that always picks the earliest tuple available in the list of candidate tuples (i.e., with the smallest $start$ value), which is the tuple with the time slot directly after when the packet is generated or received. Theoretically, this results in minimal delay and a perfectly daisy-chained path from source to root. However, a possibly high number of schedule collisions with this tuple leads to packet loss and increased latency. Computationally, always choosing the earliest and thus first tuple makes this heuristic the most attractive.

C. Improved Collision Avoidance

Next to the collision solving algorithm, we introduce an additional feature that aims to avoid schedule collisions. In the original ReSF the housekeeping scheduling function, i.e., eLLSF, could not reserve cells that were occupied by ReSF for a future period in time. However, determining the length of this period is not trivial. In the updated ReSF version we propose an alternative and limit the choice of the eLLSF slots to a pre-determined set. When allocating slots for housekeeping, eLLSF is only allowed to choose slots out of this set and ReSF can never activate a cell at any of these time slots for a recurrent reservation. Additionally, when a node applies the collision solver algorithm to select the best candidate tuple, every slot in that set is included as a separate reservation to compare to. Those slots are characterized by the tuple $(start = ASNSlot, period = slotframeLength)$, where $ASNSlot$ is the ASN of the time slot in the first slotframe. By explicitly incorporating these slots in the collision solver process, the selected candidate tuple will cause much less schedule collisions between ReSF and eLLSF.

TABLE I
THE 6TISCH SIMULATOR PARAMETERS.

Parameter	Value
Nr. of runs per experiment	25
Simulated time	1 h
Frequency	2.4 GHz (16 channels)
Stable RSSI	-91 dBm (PDR $\sim$ 0.75)
Slotframe size	101 (with 15 ms slots)
Nr. of SHARED bootstrap cells	7
6top housekeeping	False
TSCH [min, max] back-off exp.	[1, 1]
Bayesian broadcast probability	0.33
RPL parent set size	1
RPL DAO period	90 s
MSF Join process	False
MSF NumCellsUsed [low, high]	[4, 12]
MSF NumCellsPassed	16
ReSF reservation buffer	20
ReSF reserved back-up slots	20

D. Supporting Sporadic Traffic

In the previous version of ReSF, allocating extra cells to empty the queue from failed packets was based on a periodic housekeeping moment. During that moment the average number of packets in the queue since the last housekeeping was used as an estimation for the (de-)allocation of extra cells until the next housekeeping. In this updated version of ReSF, the extra cell adaptation algorithm is changed to the traffic adaptation algorithm of MSF [3]. Using this algorithm, ReSF does not look at queue contents, but at the used extra cells during the last *NumCellsPassed* elapsed extra cells. *NumCellsPassed* is a configurable parameter. Additionally, ReSF limits the (de-)allocation of extra cells to only one at a time. In the scenarios with less frequent traffic and more sporadic traffic, by applying these changes the traffic adaptation algorithm should fluctuate less and management signaling in the network decreases.

IV. EVALUATION

In this section, we evaluate the proposed ReSF improvements. First, we present the simulation configuration. Afterwards, we evaluate the collision solving heuristics and the collision avoidance feature. Finally, we compare the improved ReSF to the original ReSF and the scheduling function eLLSF.

A. Simulation Setup

To properly evaluate the performance of ReSF, we use the official 6TiSCH simulator [2]. It implements the 6TiSCH stack up to the routing layer. We extended the simulator with eLLSF which is mainly based on LLSF and daisy-chains its cells for low-latency performance [1], [4]. The resource adapting algorithm of eLLSF is changed to that of MSF [3]. A summary of the simulator configuration can be found in Table I.

During the different iterations, the nodes are placed on a grid with the root node positioned in the center and each node its final grid position is adjusted slightly following a normal distribution around its initial grid position. The initial inter-distance between the node is 100 m and the average hop count is 5.5 $\pm$ 0.5. The scheduling functions are compared for

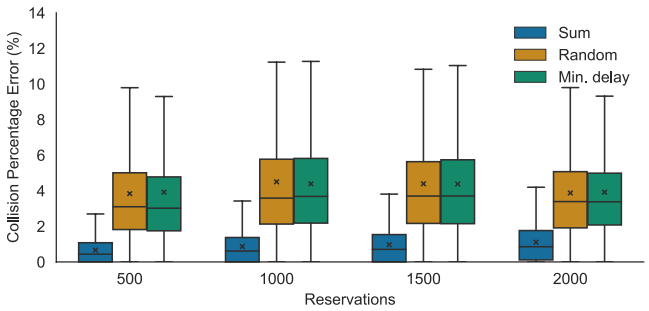

Fig. 2. Collision percentage error of the different heuristics, compared to the optimal collision approach.

three different traffic scenarios, called *fast* (i.e., 3 s, 6 s or 9 s), *frequent* (i.e., 30 s, 45 s or 60 s) and *non-frequent* (i.e., 300 s, 450 s or 600 s). At the start of the experiment, a node pick one of the three transmission intervals of the specific scenario uniformly at random. All network traffic is sent up to the root. The results discussed discard the bootstrap of the network during which the network converges, meaning that each node already has a reserved dedicated SHARED cell to its preferred parent. In the case of ReSF, this also means that each node's reservation reached the root.

The *latency* metric discussed in the evaluation is defined as the time it takes for a data packet to reach the root, measuring from the moment it was generated on the source node.

B. Collision Solving Approaches

Below the performance of the proposed collision solving heuristics is evaluated. We compare the exact solution to the sum and the minimal delay heuristic and a random approach. The random approach randomly selects a tuple out of the proposed candidates. First, we compare the different approaches by observing the error in estimated schedule collision percentage. Afterwards, we evaluate the computational performance of the heuristics on real hardware. Finally, the different solutions are evaluated in a simulated 6TiSCH network in terms of packet delivery latency.

1) Collision Percentage Error: To evaluate the performance of different approaches, we introduce the *collision percentage error* metric, defined as the absolute difference in collision percentage (i.e., the number of unique collisions over the total number of transmissions of the candidate tuple), between the tuple chosen by the exact approach and the one chosen by the heuristic.

We performed a standalone experiment (i.e., without the simulator) in which we calculated the best candidate for the ReSF tuple $(start_{new}, period_{new})$ for different numbers of reservations already present on the node. The experiment was repeated 2000 iterations. For each iteration the start time and the period of all tuples were randomly chosen between 101 time slots (i.e., the length of a slotframe) and 6000 time slots (i.e., 90 s) and the number of candidates tested was 64, as determined in our previous work. We limited the least common multiple of the periods to 12 hours (in case it was larger).

TABLE II
DURATION COMPARISON ON AN OPENMOTE B BOARD BETWEEN THE EXACT COLLISION SOLVING ALGORITHM AND THE HEURISTIC.

Nr. collisions	Duration (μs) Exact Algorithm	Sum	Min. delay
1	94	94	< 1
100	188	94	< 1
500	594	94	< 1
1000	1094	94	< 1

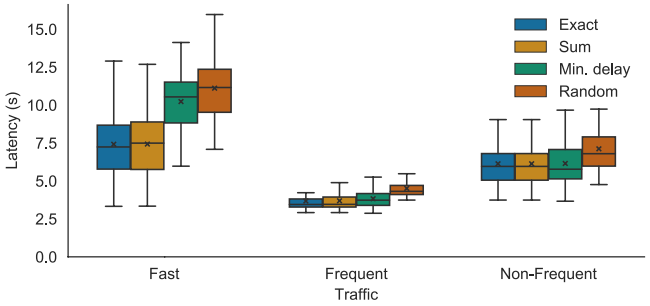

Fig. 3. Latency comparison between the exact and heuristic ReSF collision solving approaches.

Figure 2 shows the results. As expected, the sum heuristic outperforms the two other heuristics as its maximal error only goes up to 1.1 %, at 2000 tuples, while the minimal delay and random approach have an error up to 4.4 % and 4.5 %, both at 1000 tuples with outliers that exceed a 10 % error. For the sum heuristic, this means that on average the tuple that the sum heuristic chooses only has a 1 % difference in terms of number of collisions with the tuple that the optimal approach would calculate to have the best collision percentage.

2) Hardware Performance on an OpenMote B: To show the difference in computational performance of the different collision solving approaches, we tested them on an OpenMote B[1]. The OpenMote B is an open-source hardware board developed to accelerate the development of industrial IoT. It contains an ARM Cortex-M3 micro-controller with 32KB on-chip RAM. We compare the algorithms for different numbers of collisions. The exact algorithm calculates every separate collision ASN (i.e., here we neglect the additional overhead of adding them to a list and checking if a collision is unique or not) while the sum heuristic only calculates the sum and adds it to a total. The minimal delay heuristic will just take the proposed tuple, so its duration is less than 1 μs. Table II shows the results. It is clear that the exact algorithm's execution time increases with the number of collisions, while that of the sum heuristic is constant. As an example, when a node wants to send an ReSF reservation and it considers 20 candidate tuples (i.e., the ReSF reservation buffer is 20) and it has to compare to 50 other reservations with on average 1000 collisions per tuple, this will take a total of 1.1 s (i.e., 1094 us x 20 x 50) and 0.09 s (i.e., 94 us x 20 x 50) for the exact approach and the sum heuristic respectively.

[1]http://www.openmote.com/product/openmote-b-bronze-kit/

TABLE III
PACKET LOSS AND LATENCY FOR RESF WITHOUT, WITH THE OLD CA AND WITH THE NEW CA FEATURE IN A 200 NODE NETWORK.

Traffic	No CA Loss (%)	Old CA Loss (%)	New CA Loss (%)
Fast	40.8	47.1	27.8
Frequent	0.1	0.4	0.2
Non-frequent	0.3	0.3	0.4
	Latency (s)	**Latency (s)**	**Latency (s)**
Fast	15.4	22	8.9
Frequent	2.7	3.1	3.6
Non-frequent	4.7	4.7	5.8

3) Heuristic Performance in 6TiSCH network: To evaluate how the heuristics perform in terms of actual network performance, each heuristic was tested in a 200 node 6TiSCH simulation for all traffic scenarios, with a ReSF reservation buffer of 64. Figure 3 shows that the performance of the sum heuristic is nearly identical to the exact one, while the latency values of the minimal and random approach are higher. Especially in the fast traffic scenario, the latency decrease of the sum heuristic is 27.4 % and 33.1 % better than those of the minimal delay and random approach respectively. Because of this significant performance increase, in the remainder of this evaluation we use the sum heuristic. However, when there are a lot of tuples to compare to and the computational performance of the sum heuristic would be a problem, one can always switch to the minimal delay heuristic, which, especially in the frequent and non-frequent scenarios, performs well.

C. Collision Avoidance

To evaluate the collision avoidance (CA) feature, we compare packet loss and latency in a 200 node network. For the new CA feature, we experimentally set the number of slots reserved for the housekeeping cells to 20. Table III shows the results for ReSF without any CA, with the old CA and the new proposed CA. We observe that for the highly saturated traffic case, i.e., the fast scenario, there are significant improvements of 13 % and 19.3 % in packet loss and the latency decreases with 6.5 s and 13.1 s, compared to ReSF without CA and with the old CA respectively. However, for the other scenarios with frequent and non-frequent traffic, we observe that preventing ReSF of using the reserved housekeeping slots actually has a negative effect on the latency. This is due to the reduced amount of contention, and therefore collisions, making the reserved housekeeping slots less useful as they prevent ReSF from daisy-chaining all resources from source to root.

D. Recurrent Traffic

In this section we evaluate the improved ReSF, i.e., with the sum heuristic, the CA feature and the new housekeeping adaptation algorithm, to eLLSF and the original ReSF, i.e., with the periodic housekeeping moment and the old CA, in the different recurrent traffic scenarios. Figure 4 shows the results with only recurrent traffic.

In all scenarios, the decrease in latency between eLLSF and the improved ReSF is significant, with respectively 40 %,

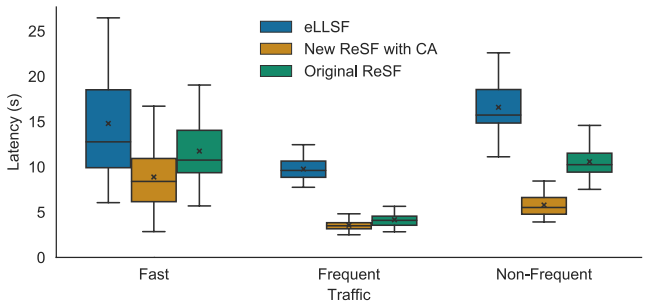

Fig. 4. Comparison of eLLSF, the original ReSF and new ReSF in a 200 node network with recurrent traffic.

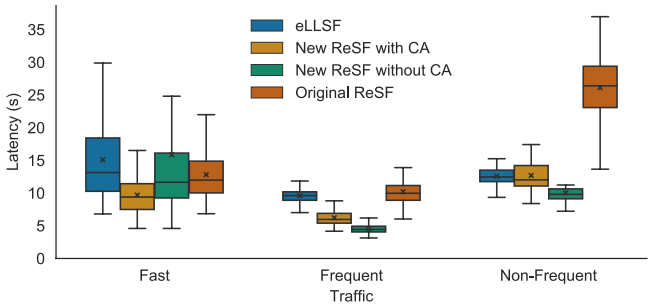

Fig. 5. Comparison of eLLSF, the original ReSF, the new ReSF with and without CA in a 200 node network with recurrent and sporadic traffic.

63.5 % and 65.1 %. Moreover, the packet delivery ratio (PDR) of the improved ReSF is 72 % in the highly saturated fast traffic scenario, while that of eLLSF is only 66.7 % (not depicted in graphs). In the frequent and non-frequent scenarios the PDR of ReSF is higher than 99.5 %, while the PDR of eLLSF is 98.6 % and 96.6 % respectively. The improved ReSF also outperforms the original ReSF, especially in the non-frequent scenario with a latency decrease of 45.3 %. The PDR improves from 97 % to 99.5 %. These results show the effectiveness of the improved ReSF.

E. Recurrent and Sporadic Traffic

In this section we compare the new ReSF to eLLSF and the original ReSF, and also show the performance of new ReSF without the CA feature. In addition to the recurrent traffic, every node also sends sporadic traffic for which the node randomly waits approximately 30 s, 60 s or 90 s before sending a new sporadic packet. Figure 5 shows the results.

In all scenarios, an improved version of ReSF (with or without CA) outperforms the original ReSF in terms of latency up to 61.5 % in the non-frequent scenario. We also observed that in terms of PDR the original ReSF is outperformed by the newer versions, with an increase from 57.1 to 70.8 % in the fast scenario. The traffic adaptation algorithm of the original ReSF seems too responsive to the sporadic traffic and floods the network with resource management signaling (especially in the frequent and non-frequent scenarios), which hinders the data propagation. When comparing the improved ReSF to eLLSF, we observed that the ReSF version with CA performs better in the saturated fast traffic scenario, while ReSF without CA performs better in the other scenarios with a maximum decrease in latency of 53.2 %. While in the saturated scenario the reserved cells for housekeeping help ReSF to avoid schedule collisions. In other scenarios these reserved slots limit ReSF from sending the data with minimal delay.

V. Conclusion

This work focused on improving the originally proposed ReSF towards a more efficient and low-latency solution. We proposed three improvements. First, we provided heuristics as replacements for the original computationally-intensive collision solving algorithm. Second, we provided an additional schedule collision avoidance feature. Finally, we added increased support for sporadic traffic in ReSF networks. Additionally, we have conducted numerous simulation experiments by comparing the updated ReSF to the eLLSF and the original ReSF. The results showed that the performance of the summation heuristic is nearly identical to the performance of the exact algorithm in terms of latency and that the collision avoidance feature significantly improves packet loss in a saturated traffic scenario. Additionally, it was shown that the updated ReSF improved on eLLSF and the original ReSF up to 65.1 % and 45.3 %. We conclude that the newly proposed ReSF is better equipped than its predecessor to maintain minimal delays in wireless sensor networks with a mix of sporadic and recurrent sensor traffic.

Acknowledgment

Part of this research was funded by the Flemish FWO SBO S004017N IDEAL-IoT (Intelligent DEnse And Longe range IoT networks) project, and by the imec ICON project IoS.

References

[1] G. Daneels, B. Spinnewyn, S. Latr, and J. Famaey, "Resf: Recurrent low-latency scheduling in ieee 802.15.4e tsch networks," *Ad Hoc Networks*, vol. 69, pp. 100 – 114, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870517302019

[2] E. Municio, G. Daneels, M. Vučinić, S. Latré, J. Famaey, Y. Tanaka, K. Brun, K. Muraoka, X. Vilajosana, and T. Watteyne, "Simulating 6tisch networks," *Transactions on Emerging Telecommunications Technologies*, p. e3494, 2018.

[3] T. Chang, M. Vuini, X. Vilajosana, S. Duquennoy, and D. Dujovne, "6TiSCH Minimal Scheduling Function (MSF)," Internet Engineering Task Force, Internet-Draft draft-chang-6tisch-msf-02, Jul. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-chang-6tisch-msf-02

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

[5] A. Morell, X. Vilajosana, J. L. Vicario, and T. Watteyne, "Label switching over IEEE802.15.4e networks," *Transactions on Emerging Telecommunications Technologies*, vol. 24, no. 5, pp. 458–475, 2013. [Online]. Available: http://dx.doi.org/10.1002/ett.2650

[6] F. Theoleyre and 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*, ser. MSWiM '16. New York, NY, USA: ACM, 2016, pp. 102–110. [Online]. Available: http://doi.acm.org/10.1145/2988287.2989133

[7] S. Oh, D. Hwang, K.-H. Kim, and K. Kim, "Escalator: An autonomous scheduling scheme for convergecast in tsch," *Sensors*, vol. 18, no. 4, p. 1209, 2018.