LaDiS: a Low-Latency Distributed Scheduler for Time-Slotted Channel Hopping Networks

Hajar Hajian*, Majid Nabi†*, Mahboubeh Fakouri*, and Farzad Veisi*

*Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan 84156-83111, Iran
†Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, the Netherlands
Email: hajar.hajian@ec.iut.ac.ir, m.nabi@tue.nl, m.fakouri@ec.iut.ac.ir, f.veysi@ec.iut.ac.ir

***Abstract*—Time-Slotted Channel Hopping (TSCH), as an operational mode of the IEEE 802.15.4 standard, is a promising medium access mechanism for industrial Wireless Sensor Networks (WSNs). However, efficient performance of such networks depends on the medium access scheduling scheme, which is not specified by the standard. This paper proposes a low-latency distributed scheduler, called LaDiS, for multi-hop tree-based TSCH networks. The main objective is to provide low end-to-end data latency in convergecast WSNs with very low communication overhead. The schedule of each node is determined by its parent based on the available local information about the routing structure and traffic requirement of that node. At the same time, LaDiS provides proper opportunity for data aggregation by relaying nodes in a multi-hop network leading to reduced traffic. The performance of the proposed scheduler as well as the existing distributed TSCH schedulers is extensively evaluated in various setups. The results show that LaDiS considerably outperforms others in terms of data latency in the networks under consideration in this work. LaDiS is implemented and integrated in the Contiki operating system.**

I. Introduction

The IEEE 802.15.4 standard [1] is one of the widely used communication standards that defines the physical and Medium Access Control (MAC) layers for low-power Wireless Sensor Networks (WSNs). Time-Slotted Channel Hopping (TSCH) is an operational mode of this standard for supporting industrial applications. In TSCH, a TDMA-based MAC approach together with a channel hopping scheme are exploited. Channel diversity makes the network more reliable by reducing the impact of cross-technology interferences and multi-path fading which are usually harsh in an industrial site. The TDMA mechanism provides a collision-free communication within the network which leads to a better predictability and reliability for industrial applications with stringent requirements. The TSCH mechanism specified in the standard does not provide any schedule for assigning TDMA timeslots to wireless nodes for their packet transmission; the scheduler design is left for the upper layers in the protocol stack.

In a distributed TSCH scheduler, wireless nodes decide about their own schedule based on their local topological information. It makes the distributed schedulers suitable for dynamic and large-scale networks. However, since the scheduling decisions are made locally by the wireless nodes without the knowledge of the entire network, the extracted schedules may be sub-optimum, and thus the required end-to-end Quality-of-Services (QoS) such as latency and Data Delivery Ratio (DDR) may be tricky to be satisfied.

This paper proposes an efficient low-latency distributed TSCH scheduling mechanism, called LaDiS, for large-scale convergecast WSNs. The network is assumed to use a multi-hop tree-based routing protocol, such as RPL [2], on top of the TSCH layer. In the proposed scheduler, nodes exploit local routing information, and exchange small scheduling messages with their children and parents in the tree structure to determine their schedule. Thus, it requires only local information exchange imposing very low communication overhead. The timeslots are assigned to the links in such a way so that the timeslot(s) of a parent is placed after the timeslots of all its children. This heavily decreases the end-to-end latency for data convergecast, and provide a perfect opprtunity for data integration and aggregation by the relaying nodes in the routing tree. In summary, the main objectives of LaDiS scheduler design are 1) low latency for delivering sensor data to a central entity (sink node), 2) low communication overhead and fast scheduling by exchanging only local information, and 3) providing appropriate possibilities for data aggregation by the relaying nodes in the RPL tree.

The performance of the proposed scheduling mechanism is evaluated in various conditions and scenarios. Simulation results show that this algorithm significantly reduces the size of slotframes. This is a requirement of many WSN applications. Moreover, the end-to-end data latency is considerably improved for the scenarios under test. Also, the variation of the achieved latency between different nodes are much less than that in the existing algorithms. The proposed scheduler is implemented and verified in the Contiki [3] operating system and its network simulator (Cooja [4]).

The rest of the paper is organized as follows. Section II reviews the related distributed TSCH scheduling algorithms. The proposed scheduler (LaDiS) is presented in Section III. In Section IV, the performance of LaDiS is evaluated and compared with the related methods. Section V concludes.

II. Related Work

The focus of this paper is on distributed TSCH scheduling mechanisms for large-scale convergecast WSNs. DeTAS [5] is a distributed traffic-aware scheduler for RPL-based networks. Starting from the RPL tree leaves, each node notifies its

parent about the traffic required by itself and its sub-tree. This procedure continues until the root of the tree receives traffic requirements of all nodes. Then the root determines the schedules for its children by sorting and categorizing them into even and odd lists based on their traffic requirements. Starting from the child with the highest traffic, every other slots are assigned for transmissions by that node leaving slots in between for transmission of the children of that child. Accordingly, each node that receives its schedule determines the schedule for its children, and this process continues until all nodes receive their schedules. The algorithm requires a procedure starting from leaves up to the root, and then down taking long time and many packet exchanges. Also, data aggregation by the parent nodes is not possible because the parents forward the received packet in the timeslot right after the reception timeslot.

Wave [6] is a distributed scheduling algorithm that uses RPL information for scheduling. In this algorithm, it is assumed that each node knows the set of its conflicting nodes. First, the root sends a start message to its children. Each node that receives this message and has the most number of packets to send, assigns a schedule to itself taking into account the links that are already assigned to the set of its conflicting nodes. Compared to DeTAS, Wave produces longer slotframes which may lead to higher data delivery latency. In addition, it has higher overhead due to more number of message exchanges for scheduling.

DIS-TSCH [7] is another distributed scheduler. During the construction of the RPL tree, it allows every node in the network to receive information about the node's location in the RPL tree from its parent. Then every node assigns timeslots to itself and selects a conflict-free channel for data communication. The initial assumption of this algorithm is that the tree formed by RPL is a full tree. In the case that this assumption does not hold, some timeslots remain unused (not assigned to any node) which is a waste of bandwidth.

Orchestra [8] scheduler consists of simple scheduling rules, and does not impose any communication overhead. Each node independently decides about its schedule based on its available RPL information. Three parallel slotframes are used for different types of traffic, namely Enhanced Beacons (EB), unicast transmissions, and broadcasts as well as RPL signaling. Orchestra provides an efficient solution with simple implementation for many WSN applications. However, it is not able to satisfy the required end-to-end latency in applications with non-uniform traffic distribution, high sampling rates, and stringent latency requirements [9].

This paper focuses on scheduler design for tree-based TSCH networks for real-time industrial monitoring applications. The schedules are determined by the parent nodes without the need for gathering the whole traffic information in the root, leading to faster scheduling with very little communication overhead. The timeslots are ordered in such a way so that the end-to-end latency is very low, and proper data agregation/integration possibility is provided.

III. LaDiS TSCH Scheduling Mechanism

This section first presents the network structure and data traffic model considered in this work. Then the LaDiS scheduling algorithm is presented.

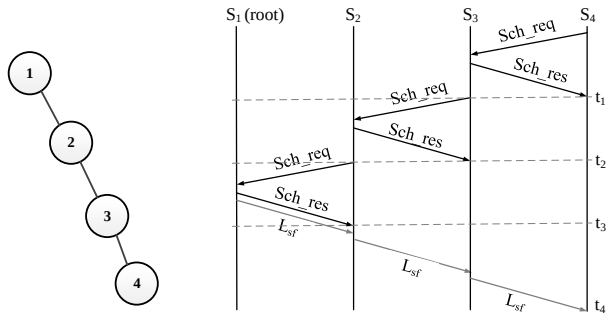
A. Network Model

Suppose that $S = \{s_1, s_2, ..., s_n\}$ is the set of n wireless sensor nodes in an industrial WSN, each running the IEEE 802.15.4 TSCH standard for their MAC layer, and a tree-based routing structure (e.g., RPL) as their multi-hop routing layer. Node s_1 is supposed to be the root of the routing tree. Large-scale multi-hop convergecast networks are considered in which all data items sampled by all the nodes in the network are supposed to be delivered to the root (s_1) with low end-to-end latency (real-time industrial monitoring). The LaDiS scheduling algorithm is responsible for determining the transmission and reception timeslots and channel offset (CH_off) for each node. The scheduler algorithm running in node s_i has access to the tree information of that node including the node's preferred parent (p_i), set of its children (C_i), and its rank (r_i) in the routing tree. The length of the TSCH slotframe (L_{sf}) may be preset at design time, or it may be determined by the scheduler.

The nodes may have different data sampling rates. In the beginning of each TSCH slotframe, every node s_i has q_i bytes of sensor data sampled during the previous slotframe. If s_i is not a leaf node ($C_i \neq \emptyset$), it is responsible to receive data from its children and forward them as well as its own data to its parent. Thus, the total data bytes that needs to be sent in each slotframe by s_i is $Q_i = q_i + \sum_{\forall s_k \in C_i} Q_k$. The LaDiS scheduler is designed in such a way so that the nodes can best exploit data aggregation or integration. It is done by scheduling the transmission slots of each node after all timeslots dedicated for transmission of its children. Assuming that the maximum number of data bytes in each packet is $Payload$, s_i puts its data traffic into at most $\lceil Q_i / Payload \rceil$ packets. This value is the number of timeslots required by s_i in each slotframe.

B. Overall Scheduling Mechanism

The LaDiS scheduling is performed after the routing tree is constructed. Before scheduling, the minimal schedule defined in the TSCH standard, consisting of only shared timeslots, is used for the required packet exchanges by the routing layer and the scheduling process. The LaDiS process starts from the leaf nodes (at whatever level in the tree they are) by sending scheduling request to their parent. A parent that receives scheduling request determines the transmission slots of the requested child, and sends the schedule back to that child. This determines the reception slots of the parent node as well. Accordingly, each (parent) node sends scheduling request to its own parent after it schedules all its children. As an illustrative example, consider the tree example shown in Fig. 1(a), where the overall packet exchanges for scheduling of such a tree is depicted in Fig. 1(b). This scheduling process proceeds until it reaches the root node (time t_3 in the example



(a) Example multi-hop routing tree

(b) Exchanged control messages

Fig. 1. Illustration of LaDiS's general mechanism

of Fig. 1(b)). The scheduling process ends when the root node (s_1) schedules all nodes in the set of its children (C_1).

At this step, the number of active slots in each slotframe is known. This value determines the duty cycle of the network if the length of slotframes (L_{sf}) is preset at design time. In this case, the network is ready for data gathering right away, and no further scheduling packet exchange is needed. As an alternative option, this number of active timeslots (plus any number of inactive slots) may be considered as the slotframe length, if the length is not preset. In such case, L_{sf} needs to be disseminated to all the nodes in the WSN. After reception of L_{sf} by all nodes (time t_4 in the example of Fig. 1(b)), network will be operational and ready for data sampling and transmission.

C. Scheduling Rules

Assume that L_i is the set of timeslots that are assigned to node s_i. The LaDiS algorithm that is run by every node in the network satisfies the conditions stated in Eqn. 1 and Eqn. 2.

$$\forall s_i \in S : \bigcap_{\forall s_j \in C_i} L_j = \emptyset \tag{1}$$

$$slot_k \in L_i \Rightarrow \ k > j, \ \forall j \in \bigcup_{\forall s_h \in C_i} L_h \tag{2}$$

Eqn. 1 ensures a collision free communication from children to their parent. Eqn. 2 enforces that the slots assigned for transmission of each node is located after all slots allocated for transmission of its children. This is the key mechanism of LaDiS to provide appropriate data aggregation/integration possibility for the nodes in the multi-hop routes towards the root node, and reduce the end-to-end latency. The rules also ensure that no node has a timeslot which is both for packet transmission to its parent, and reception from its children. The example shown in Fig. 2 clarifies the concept. When s_4 wants to transmits its packet to s_1, none of the dark-colored nodes (i.e., s_2, s_3, s_8, s_9) are allowed to transmit in the same timeslot. Note that for such nodes, the timeslots should be different even if they use different frequency channels. Since the schedules of each node is totally determined by its parents, both rules are fully satisfied by LaDiS.

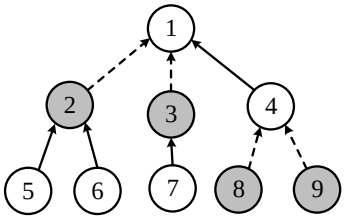


Fig. 2. Illustration of the scheduling rules

The TSCH mechanism provides the possibility to perform parallel transmissions by using different channel offsets which means different frequency channels for parallel timeslots. Except the rules stated in Eqn. 1 and Eqn. 2, such parallel transmissions can be exploited for other nodes that may be in the interference range of each other in order to avoid collisions. This reduces the size of slotframes, which in turn reduces the duty cycle of the nodes leading to lower energy consumption. LaDiS uses three different channel offsets in such a way so that the nodes in the adjacent levels of the routing tree use different channel offsets. Thus, the channel offset used by node s_i (CH_off_i) is determined based on its rank (r_i) in the tree ($CH\_off_i = r_i \ mod \ 3$). In Fig. 2, nodes s_5, s_6, and s_7 can use the same timeslot as the one used for transmission by s_4. But to avoid possible interferences, different frequency channels are used by setting different channel offsets.

D. LaDiS Scheduling Algorithm

Algorithm 1 describes the details of LaDiS's mechanism which is run by every wireless node in the network. Each node waits to receive scheduling request from its children and schedule them. The subset of children of node s_i that have been scheduled is shown by $C'_i \subseteq C_i$; this set is initially empty. This part is continued until all children of s_i are scheduled (i.e., $C'_i = C_i$). If a node is a leaf node, it bypasses this part since both C'_i and C_i are empty from the beginning.

Function WAITFORREQUEST picks a scheduling request from the buffer of the received requests from the children of s_i if there are any. Provided that s_j is a child of s_i ($s_j \in C_i$), the outputs of WAITFORREQUEST are l_j and Q_j, which are the slot number of the last slot assigned to the children of s_j, and the data traffic required by s_j, respectively. Node s_i first checks if it has already scheduled slots for s_j; it may happen if the previous scheduling response packet has not reached s_j for any reason (interference or so). In this case, s_i sends the already scheduled list of slots (L_j) to s_j (lines 7-9).

Knowing l_j, s_i can schedule all the transmission slots for s_j after the slots previously assigned to the children of s_j. It satisfies the LaDiS scheduling rule stated in Eqn. 2, which firstly reduces the end-to-end latency because all data items reach the root in a single slotframe. Secondly, it allows s_j to perform any kind of data aggregation/integration. Node s_i adds the required data traffic of s_j to its own data traffic requirements (line 10). In lines 11 to 19, s_i starts from l_jth slot in the slotframe and looks for slots that have not been yet assigned to any other children of s_i (the rule of Eqn. 1). In

ALGORITHM 1: The LaDiS scheduling algorithm by node s_i

Data:
$Payload$: Maximum data bytes per packet
p_i: preferred parent of s_i
r_i: rank of s_i
C_i: set of children of s_i
C'_i: set of children of s_i that have been scheduled ($C'_i \subseteq C_i$)
q_i: traffic requirement of s_i in bytes
L_i: set of timeslots that have been assigned to s_i
l_i: the last slot in the slotframe assigned to C_i
L'_i: set of timeslots that have been assigned to nodes in C'_i

Input:
q_i: traffic requirement of s_i in bytes

1 $C'_i = \emptyset$; // No children of s_i is yet scheduled
2 $L_i = \emptyset$;
3 $L'_i = \emptyset$;
4 $Q_i = q_i$;
5 **while** $C'_i \neq C_i$ **do** // there are unscheduled children
6 $(l_j, Q_j) \leftarrow$ WAITFORREQUEST(s_j); $\forall s_j \in C_i$
7 **if** $s_j \in C'_i$ **then** // s_j is already scheduled
8 SENDRESPONSE(s_j , L_j);
9 **else**
10 $Q_i = Q_i + Q_j$; // simple data aggregation
11 $k = l_j + 1$; // start from the slot after l_j
12 $\lambda_j = \lceil Q_j / Payload \rceil$;
13 **while** $|L_j| \neq \lambda$ **do**
14 **while** $slot_k \in L'_i$ **do**
15 $k = k + 1$;
16 **end**
17 $L_j = L_j \cup \{slot_k\}$;
18 $L'_i = L'_i \cup \{slot_k\}$;
19 **end**
20 SENDRESPONSE(s_j , L_j);
21 $C'_i = C'_i \cup \{s_j\}$;
22 **end**
23 **end**
24 **if** $i = 1$ **then** // s_i is root
25 **if** L_{sf} *is not preset* **then**
26 $L_{sf} \leftarrow$ LASTSLOT(L'_i);
27 BROADCAST(L_{sf}); // flood the slotframe length
28 **end**
29 **else**
30 $l_i \leftarrow$ LASTSLOT(L'_i);
31 SENDREQUEST (p_i, l_i,Q_i)
32 $L_i \leftarrow$ WAITFORRESPONSE(p_i);
33 CH_Off_i = r_i mod 3;
34 **if** L_{sf} *is not preset* **then**
35 $L_{sf} \leftarrow$ WAITFORLSF();
36 BROADCAST(L_{sf});
37 **end**
38 **end**

each iteration of the loop, one slot is added to L_j, which is the set of slots scheduled for packet transmissions by s_j. Also, this slot is added to L'_i, which is the set of slots assigned to the children of s_i so far. This set is needed when s_i wants to send scheduling request to its parent. This process continues until $\lambda_i = \lceil Q_i / Payload \rceil$ number of slots are assigned to s_j. After that, function SENDRESPONSE sends the assigned timeslots (L_j) to s_j, and this node is marked as scheduled (line 21).

In the second part of Algorithm 1 (from line 24 onwards), node s_i has scheduled all its children, and thus it is time for sending scheduling request to its parent p_i. If s_i is the root of the tree, this part is not needed. In this case, the scheduling process is finished. If L_{Sf} is not preset at design time, s_1 finds the last timeslot assigned to its children in the first part of the algorithm, using function LASTSLOT (line 26). This number will be used as the length of the slotframes (L_{sf}). The root node broadcast L_{sf}, and it is repeated until all nodes receive it using a simple flood-based one-to-all data dissemination.

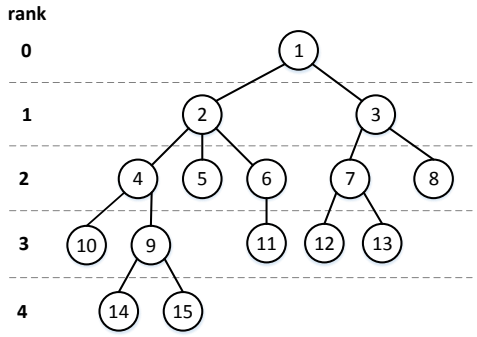
If s_i is not the root of the tree (lines 30 till 37), it sends a scheduling request to p_i using function SENDREQUEST containing the last slot scheduled to its children (l_i) and its required traffic (Q_i). Then it waits until it receives a scheduling response packet from p_i. This is done by function WAITFORRESPONSE that returns the set of slots assigned to s_i (i.e., L_i). Note that function WAITFORRESPONSE waits to receive the schedule from p_i within a certain timeout. If the timeout is over, the function retransmits the scheduling request to p_i. It is done because each of the request or response packets may get lost because of interference or collisions. After reception of the scheduling response, s_i knows its own transmission slots. In the case that L_{Sf} is not preset at design time, the node need to wait for reception of the slotframe length to start its data communication. It is done by function WAITFORLSF. All nodes participate in disseminating this small control data (L_{Sf}) by broadcasting it.

E. An Illustrative Scheduling Example

We make use of an example tree, shown in Fig. 3(a), to illustrate the LaDiS's mechanism. The network consists of $n = 15$ nodes, all running Algorithm 1. It is assumed that $q_i = 30$ bytes for all nodes, and the available data $Payload$ in each packet is 100 bytes. Thus each packet can contain up to three data items. In the first step, leaf nodes ($\{s_5,\ s_8,\ s_{10},\ s_{11},\ s_{12},\ s_{13},\ s_{14},\ s_{15}\}$) send scheduling request to their parents. As an example, s_9 receives scheduling request from its children $C_9 = \{s_{14},\ s_{15}\}$ in any order. Assume that the request of s_{14} is received first. In the request packet, $l_{14} = 0$ and $Q_{14} = 30$ bytes. Since s_9 has not allocated slots to any other child yet, it assigns the first slot in the slotframe ($slot_1$) to s_{14}, and adds $slot_1$ to L_9. Later that s_{15} sends its scheduling request, s_9 assigns the first slot after the slot previously given to s_{14}. Thus, $slot_2$ is added to L_9. The same process is done for the rest of the leaf nodes. The result of this step of the algorithm is represented in Fig. 3(b).

Now the nodes whose children have received their schedule ($\{s_6,\ s_7,\ s_9\}$) start sending schedule request to their parent. As an example, s_9 sends scheduling request to its parent (s_4) containing $l_9 = 2$ and $Q_9 = 90$ bytes. s_4 allocates $slot_3$, which is the first slot after l_9 that is not in L_4. Since Q_9 is still not more than $Payload$, only one timeslot is needed for s_9. The slot assignment process for the rest of the nodes at this step is shown in Fig. 3(c).

The remaining steps of the scheduler are represented in Fig. 3(d). For instance, when s_2 is done with determining the schedule of its children, it sends scheduling request to the root with $l_2 = 5$, and $Q_2 = 270$ bytes (the total number of the required data traffic by s_2 and its sub-tree). As a result, s_1 must assign $\lceil 270/100 \rceil = 3$ timeslots to s_2. With a high chance, s_2 sends its scheduling request after that of s_3. Thus,



(a) An illustrative routing tree example (max. degree = 3)

timeslot / CH_off	1	2
0	10 11 13	12
1	14	15
2	5 8	

(b) All leaf nodes receive their schedule.

timeslot / CH_off	1	2	3
0	10 11 13	12	9
1	14	15	
2	5 8	6	7

(c) $\{s_6,\ s_7,\ s_9\}$ receive their schedule.

timeslot / CH_off	1	2	3	4	5	6	7	8
	$l_{sf} = 8$							
0	10 11 13	12	9					
1	14	15		3	3	2	2	2
2	5 8	6	7	4	4			

(d) Scheduling of the remaining nodes.

Fig. 3. Three snapshots of the LaDiS process for an example routing tree.

s_1 assigns the timeslots after $l_2 = 5$ which are not previously assigned to s_3. Therefore, $L_2 = \{slot_6, slot_7, slot_8\}$, and the length of the slotframes is set as ($L_{sf} = 8$).

IV. Performance Evaluation

We evaluate the performance of LaDiS, and compare it with four state-of-the-art distributed TSCH schedulers, namely DeTAS [5], Wave [6], DIS-TSCH [7], and Orchestra [8]. The implementations of the 6TiSCH [10] protocol stack including the RPL routing protocol and the Orchestra scheduler are available in the Contiki [3] operating system. We implemented LaDiS and integrated it to the existing 6TiSCH stack in Contiki, and compared its performance with Orchestra using the Cooja [4] network simulator of Contiki. However, since the implementations of the other schedulers are not available, we implemented simulation models of all others as well as LadiS in MATLAB. It allows us to extensively investigate the performance of the schedulers in various network scales, topologies, and configurations.

TABLE I
Specifications of the routing trees used in simulations

Tree name	Tree type	Network size (n)	Degree	Height
TP1	full	63	2	5
TP2	full	127	2	6
TP3	full	255	2	7
TP4	full	40	3	3
TP5	full	120	3	4
TP6	full	363	3	5
TR1	random	50	1-5	1-10
TR2	random	100	1-5	1-10
TR3	random	200	1-5	1-10
TR4	random	400	1-5	1-10

A. Performance Metrics

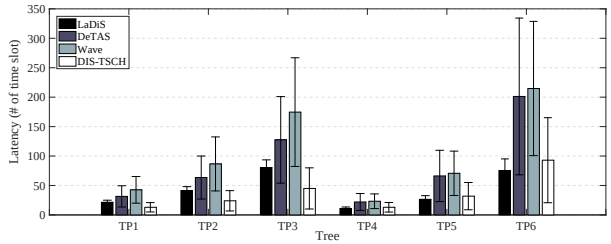
End-to-end data latency, as the time from data item generation by the source node until it reaches the root, is the main performance metric evaluated in this work. In each setup and for each scheduling algorithm, we measure the latency for data items from each node. Then the average latency over all nodes and its standard deviation are reported.

The length of the active part of the slotframe (L_{sf}) is another metric. It specifies the maximum frequency with which each node can send its data to its parent. Thus, the maximum sampling and data transmission rate of each node depends on this value. Also, in the case that the slotframe length is preset at design time, the size of its active part determines the duty cycle which is a key factor in energy consumption of the nodes. To investigate the energy efficiency of networks with different schedulers, we measure the total energy consumption in the network per data item in various setups. For each node, we count the number of transmission and receive slots, and log the length of packets sent or received in those slots. The power consumption profile of the Texas Instruments CC2650 radio chip (transmit power of $20.13mW$, and receive power of $19.47mW$) is used.

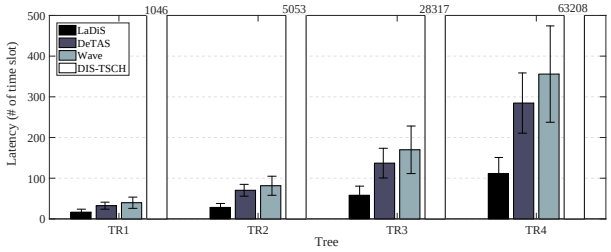
B. Simulation Results

In the MATLAB simulation, various routing tree types with different network scales are tested. Table I summarizes the specifications of the investigated trees. In one hand, we test full trees with node degree of 2 and 3 (TP1-TP6). The network size (n) is selected in different ranges; the exact values are picked to make full trees. Since routing trees are not likely to be full trees in real-world WSNs, we also tested random trees with four different sizes (i.e., $n = 50, 100, 200, 400$). For each network size, 50 different random trees are tested, and the average results over all trees are reported to have statistically more reliable results. In making each random tree, the node degree and tree's height are randomly selected below 5 or 10 for full and random trees, respectively.

It is assumed that 100 bytes of data payload can be included in each data packet. Each TSCH timeslot is used for transmission of a single packet and its optional acknowledge (timneslot size is 10 ms). For simplicity, it is assumed that all nodes generate the same amount of data in each slotframe ($q_i = 20$ bytes, $1 < i \leq n$). Each data item consists of the source node identifier, a sequence number, and sampled data

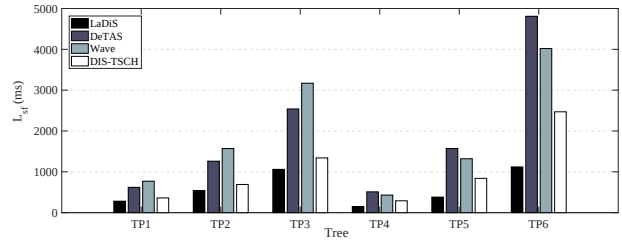


(a) Achieved latency for full trees

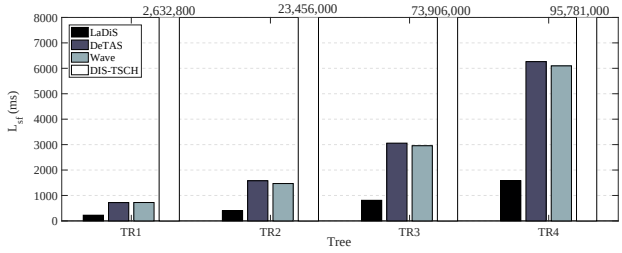


(b) Achieved latency for randomly generated trees

Fig. 4. Data latency in networks running different TSCH schedulers



(a) Achieved slotframe length for full trees



(b) Achieved slotframe length for randomly generated trees

Fig. 5. The length of slotframes (L_{sf}) in various schedulers

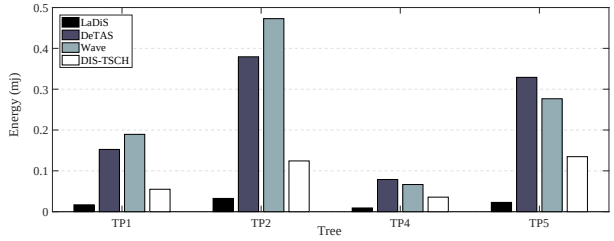


Fig. 6. Average radio energy consumption per generated data item in the networks running different TSCH schedulers

which are in total 20 bytes. Therefore, up to five data items can be put in each data packet.

Figure Fig. 4(a) shows the average of the achieved latency and its standard deviation in full trees. LaDiS provides considerably lower average latency compared to DeTAS and Wave schedulers. However, DIS-TSCH performs better for some full tress. This was expected since full trees are the best case for this schedule. In terms of the standard deviation of latency among different nodes, LaDiS performs the best. It means that latency of data items generated by different nodes is not much different, which means that a balanced distribution of network resources between all sensors is provided.

Fig. 4(b) shows the latency in random trees. LaDiS results in the lowest latency among all tested schedulers. DIS-TSCH, which performed well for full trees scenarios, suffers from very high latency and deviations in random trees. Significant latency improvement by LaDiS compared to DeTAS and Wave is due to the way of ordering timeslots and the possibility of data aggregation that LaDiS provides.

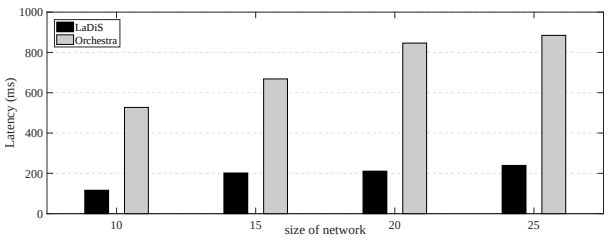
Fig. 5 shows the slotframe length determined by each scheduler. Note that for random trees, the reported L_{sf} is the average value over all 50 different random trees. LaDiS scheduler ends up with considerably shorter slotframes. It means that LaDiS is able to support higher data sampling frequency and transmission rates than the others. The difference between LaDiS and the other schedulers gets more visible when the network size scales up, which confirms our expectation about the scalability of LaDiS. This achievement is again due to the data aggregation possibility provided by LaDiS. Again DIS-TSCH performs well for full trees, but results is very long slotframes for random trees.

Fig. 6 presents the average energy consumed in one second of network operation per data item for setups that have close slotframe length. It shows that the energy consumption of the LaDiS algorithm is much lower than that in the other algorithms. The main reason for this is the shorter slotframe length provided by LaDiS, and the data aggregation which reduces the number of packet exchanges.

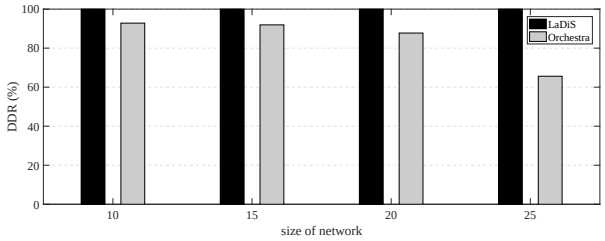
C. Contiki Implementation and Test

As a proof of concept, the LaDiS scheduler is implemented on top of the Contiki operating system, and its performance is evaluated and compared with that of the Orchestra scheduler using the Cooja simulator. Here, networks of size 10, 15, 20, and 25 nodes are tested. In each setup, the nodes are randomly deployed in an area of 100×100 meters. To have a fair comparison between LaDiS and Orchestra, the idea of data aggregation is implemented for Orchestra as well. Every node generates a data item of size 20 bytes in each slotframe.

Fig. 7(a) shows the average end-to-end latency. The results show considerably lower latency provided by LaDiS compared to that in the Orchestra-based networks. It is because, in the considered network scenarios, each node in each slotframe produces a data item. The LaDiS mechanism performs a slot allocation such that each data item produced at the beginning of a slotframe reaches the sink node in same slotframe. Therefore, the end-to-end latency of the network is less than or equal to the length of the slotframe length. In Orchestra,



(a) Average end-to-end latency



(b) Data delivery ratio

Fig. 7. The performance results from Contiki and Cooja simulations

each node has only one timeslot is each slotframe. Thus, the relaying nodes in the tree may need several slotframes to deliver their own data as well as the data items received from their lower level subtrees.

Fig. 7(b) presents the achieved DDR in various network setup. LaDiS is able to deliver almost all data items to the sink node. This achievement is due to the fact that the transmission slot of the nodes in the lower levels of the tree are scheduled earlier than their parents. Also the slot assignment is done based on the traffic requirements of each node. Thus, the memory demand for buffering the received data items in the relaying nodes, which are typically memory-limited, is very low. Therefore, no data loss occurs due to buffer overflow. On the other hand, the channel offset allocation mechanism in LaDiS helps the node to avoid collisions. In comparison, Orchestra is not a traffic-based mechanism, and many data items need to be stored in buffers leading to eventually buffer overflows for the data generation rates in the tested scenarios.

As a notion of energy consumption, average duty cycle of nodes are measured in Cooja. The average duty cycle of a random network with 20 nodes is 13.7% and 10.3% for LadiS and Orchestra, respectively. Therefore, LaDiS consumes more energy than Orchestra. This extra energy consumption is paid back by improved latency and reliability, which are of more importance than energy consumption in typical industrial applications. Orchestra is a light-weight TSCH scheduling mechanism which provides efficient communications in networks with low data generation rates. In an industrial monitoring applications, which is the focus of this work, the application may be interested to have frequent updates of small data samples. As the results showed, Orchestra is not able to support such applications, while LaDiS provides very low latency and reliable data delivery with support of higher sampling rates.

V. Conclusion

This paper proposes an efficient and low-latency TSCH scheduler, LaDiS, for large-scale convergecast wireless sensor networks. The timeslots scheduled for each node in the network are placed after the transmission slots of its all children. It leads to very low latency for data delivery to the root (boarder router/sink), since all data samples reach the root in a single slotframe. Moreover, it perfectly supports data integration/aggregation because nodes receive their children's data before their own transmission turn. Only available local tree information and limited local message exchanges between children and their parent are used for scheduling, leading to low scheduling overhead of LaDiS. The performance of the LaDiS scheduler as well as that of four other distributed TSCH schedulers are evaluated in various network setups. The results show that LaDiS is able to provide considerably lower data latency compared to the other distributed schedulers for the industrial networks scenarios considered in this work. Moreover, The length of slotframes that determines the data transmission frequency of nodes is lower in LaDiS. LaDiS is implemented, tested, and integrated in the 6TiSCH protocol stack in the Contiki operating system.

ACKNOWLEDGMENT

This work was partially supported by the SCOTT European project, that has received funding from the ECSEL Joint Undertaking under grant agreements no. 737422.

References

[1] "IEEE802.15.4-2015 - IEEE standard for low-rate wireless networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011) (April 2016)*, pp. 1–709, 2016.

[2] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander, "RPL: IPv6 routing protocol for low-power and lossy networks," *Internet Engineering Task Force (IETF), ETF RFC 6550*, 2012.

[3] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. of the 29th IEEE Conf. on Local Computer Networks (LCN)*. IEEE, 2004.

[4] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with COOJA," in *Proc. of the 31th IEEE Conf. on Local Computer Networks (LCN)*. IEEE, 2006, pp. 641–648.

[5] N. Accettura, M. R. Palattella, G. Boggia, L. A. Grieco, and M. Dohler, "Decentralized traffic aware scheduling for multi-hop low power lossy networks in the internet of things," in *Proc. of the IEEE 14th Int'l Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2013, pp. 1–6.

[6] R. Soua, P. Minet, and E. Livolant, "Wave: a distributed scheduling algorithm for convergecast in IEEE 802.15.4e TSCH networks," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 4, pp. 557–575, 2016.

[7] R.-H. Hwang, C.-C. Wang, and W.-B. Wang, "A distributed scheduling algorithm for IEEE 802.15. 4e wireless sensor networks," *Computer Standards & Interfaces*, vol. 52, pp. 63–70, 2017.

[8] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne, "Orchestra: Robust mesh networks through autonomously scheduled TSCH," in *Proc. of the 13th ACM conf. on embedded networked sensor systems*. ACM, 2015, pp. 337–350.

[9] S. Rekik, N. Baccour, M. Jmaiel, and K. Drira, "A performance analysis of orchestra scheduling for time-slotted channel hopping networks," *Internet Technology Letters*, pp. 1–6, 2017.

[10] D. Dujovne, T. Watteyne, X. Vilajosana, and P. Thubert, "6TiSCH: Deterministic IP-enabled industrial internet (of things)," *IEEE Communications Magazine*, vol. 52, pp. 36–41, 2014.