# YSF: a 6TiSCH Scheduling Function Minimizing Latency of Data Gathering in IIoT

Yasuyuki Tanaka, Pascale Minet, Mališa Vučinić, Xavier Vilajosana, Thomas Watteyne

# YSF: a 6TiSCH Scheduling Function Minimizing Latency of Data Gathering in IIoT

Yasuyuki Tanaka, Pascale Minet, Mališa Vučinić, *Member, IEEE* Xavier Vilajosana, *Senior Member, IEEE*, Thomas Watteyne, *Senior Member, IEEE*

***Abstract*—Data gathering systems in the Industrial IoT require an end-to-end latency as low as one second with coverage of a few hundred meters. The 6TiSCH standard is well suited for these types of applications. A 6TiSCH network is a multi-hop wireless IPv6 network which uses Time Slotted Channel Hopping (TSCH). TSCH is a medium access mode of IEEE802.15.4 which provides deterministic properties, and increases robustness against external interference and multipath fading. A key component of TSCH is its scheduling function that builds the communication schedule, which greatly impacts network performance. Although there are several proposed TSCH scheduling solutions in the literature, most of them are not directly applicable to 6TiSCH for real-world deployments because they fail to take into consideration the dynamics of a network. Some of them assume a *fixed* routing topology, which does not match 6TiSCH where the routing topology dynamically changes with the radio environment. In this article, we propose a full-featured 6TiSCH scheduling function called YSF, that autonomously takes into account all aspects of network dynamics, including network formation phase and parent switching. YSF aims at minimizing latency and maximizing reliability for data gathering applications. We evaluate YSF by simulation, and compare it to MSF, the state-of-art scheduling function being standardized by the IETF 6TiSCH working group.**

***Index Terms*—Industrial IoT, 6TiSCH, Scheduling. TSCH**

## I. Context and Motivation

IEEE802.15.4 [1] is arguably the most used technology in low-power wireless mesh networks. Time Slotted Channel Hopping (TSCH) is a medium access mode of IEEE802.15.4. TSCH has been standardized to meet end-to-end latency, end-to-end reliability and network lifetime requirements of industrial, city, home, building and environmental applications.

The core idea of TSCH is to combine Time Division Multiple Access and Frequency Division Multiple Access. The atomic resource for communication is a "cell" in the communication schedule, defined as a slot offset and a channel offset. The slot offset is the relative time within the slotframe of that cell. The channel offset is used to compute the communication frequency to use. The schedule tells each node what to do at each timeslot: transmit, listen, or sleep. The scheduling function is the algorithm that builds and maintains the communication schedule of the network. How the schedule is built determines several key performance indicators (KPIs) of the network, including end-to-end latency, end-to-end reliability, and the battery lifetime of the nodes. The more collisions the schedule has, the more internal interference the network would suffer, which impairs the KPIs.

The IETF 6TiSCH working group standardizes how to build a secure multi-hop wireless IPv6 network that uses TSCH [2]. 6TiSCH has standardized the 6top Protocol (6P) [3], a signaling protocol allowing neighbor nodes to locally negotiate changes to their schedule. 6TiSCH is now finalizing a secure join mechanism [4] and a distributed scheduling function called MSF [5].

MSF is the first full-featured standardized 6TiSCH scheduling function that covers the following aspects, all of which are vital for a real-world deployment:

- Network dynamics: the routing topology adapts to changes in the radio environment.
- Security: the TSCH schedule is protected against unauthorized nodes including joining nodes and attackers.
- Control traffic handling: control traffic is in charge of keeping the network operational and secure.

In addition, MSF adapts to traffic changes dynamically modifying the number of scheduled cells. One drawback of MSF is that it cannot always yield low end-to-end latency because MSF schedules cells at randomly selected slot offsets. This means the MSF schedule is *not* collision-free, and not optimized for low latency.

In this article, we propose (yet) another full-featured 6TiSCH scheduling function, YSF, optimized for multipoint-to-point traffic, which is the communication pattern commonly seen in data gathering applications. These applications require an end-to-end latency as low as one second [6]. We show how YSF achieves both an end-to-end latency and an end-to-end reliability that outperform MSF. YSF further adapts to changes in the amount of traffic and in the routing structure. YSF is implemented on the top of the 6TiSCH protocol stack in the 6TiSCH Simulator [7], which demonstrates YSF is ready for actual implementation with an operating system such as OpenWSN and Contiki-NG.

The remainder of this article is organized as follows. Section II introduces related work. Section III describes how YSF interacts with the 6TiSCH stack, and how it schedules cells. Section IV shows through an extensive simulation campaign how YSF outperforms MSF in terms of end-to-end latency and end-to-end reliability. Section V discusses possible extensions of YSF. Finally, Section VI concludes this paper.

## II. Related Work

The performance of a 6TiSCH network depends heavily on its communication schedule. The role of the scheduling

Y. Tanaka with Toshiba and Keio University, Japan.
P. Minet, M. Vučinić and T. Watteyne, are with Inria, France.
X. Vilajosana is with the Open University of Catalunya (UOC-IN3) and Worldsensing S.L, Spain.

function is to build a schedule in an automated way, meeting system requirements such as latency and reliability. In this section, we review related scheduling functions.

The state-of-the-art scheduling function is **MSF** [5]. MSF is full-featured and takes into account all aspects of real-world deployments, as discussed in Section I.

MSF implements a simpler version of **OTF** [8], which schedules TX (i.e., Transmission) cells to the parent[1] at randomly selected slot offsets. MSF dynamically adds or removes communication cells to adapt to traffic changes. In addition, MSF relocates cells which are undergoing scheduling collisions. MSF also introduces the autonomous cell, whose position in the schedule is determined by the MAC address of a corresponding node. The concept of autonomous cell was first proposed in **Orchestra** [10]. Since the autonomous cell does not require any negotiation between nodes, it is used as a "rendez-vous" cell for the secure join process and for the initial cell allocation of a pair of nodes. Thanks to the autonomous cell, MSF can use the minimal shared cell [11] only for broadcast frames such as Enhanced Beacons.

**6TiSCH-MC** [12] and **DT-SF** [13] take theoretical approaches to design 6TiSCH scheduling functions. 6TiSCH-MC models the 6TiSCH join process using the Markov Chain model. DT-SF models TSCH cell allocations to child nodes a Mixed-Integer Convex Programming problem. The resulting scheduling functions are distributed ones as MSF.

One weakness of distributed scheduling is that local decisions allocate cells, which are not optimized for end-to-end latency. To achieve low end-to-end latency, a node needs to have a TX cell right after an RX cell. There are two scheduling approaches found in the literature that minimize the gap between TX and RX cells: bottom-up and top-down.

In the bottom-up approach, cell allocation is done from the application source node toward the application destination node, which is usually the root of the routing tree. Examples are **LLSF** [14] and **ReSF** [15], which can be seen as variants of OTF. They schedule a TX cell to the next hop node right after the RX cell from the previous hop node. This causes a packet to be forwarded immediately, reducing end-to-end latency. The signaling protocol of bottom-up scheduling functions tends to be simple and easy to implement. The major drawback of this approach is that, the closer the node is to the sink, the busier its schedule is, so the more likely it is that TX cells cannot be scheduled at their optimal slot offsets.

In this sense, the top-down approach, where cell allocation starts from the root, seems more appropriate when minimizing end-to-end latency. Examples are **DeTAS** [16] and **Morell** *et al.* [17]. Each node in the network sends a bandwidth request to its parent aggregating traffic received from its children and traffic generated by itself. Once the root receives requests from all the nodes in the network, the root starts scheduling cells with 1-hop nodes. Cell allocation then continues with 2-hop nodes, 3-hop nodes, etc., until reaching the leaf nodes.

The problem when using these top-down scheduling functions with 6TiSCH is that they do not adapt to network dynamics. DeTAS was evaluated using TelosB motes with the OpenWSN stack in [16]. The authors used *fixed* topologies for the evaluation: the double-chain topology and the binary-tree topology. They assumed there is an explicit timing when the root triggers a series of allocations. However, this is not the case in 6TiSCH, which uses RPL (IPv6 Routing Protocol for Low-Power and Lossy Networks) [9]. Each node in a 6TiSCH network keeps evaluating links to its neighbors, and switches its parent to another neighbor when it finds the link to the new neighbor to be better than the current one. This means the routing topology keeps changing with the radio environment even when the number of nodes in the network do not change. There are in particular many topology changes when a network builds.

As discussed above, top-down scheduling is preferable to minimize end-to-end latency. However, to the best of our knowledge, no top-down scheduling functions in the literature can handle the network dynamics observed in a 6TiSCH network. Their use is more suited to fixed routing topology, or when using a centralized path computing element. The novelty of YSF is that it is a full-featured top-down scheduling function which adapts to network dynamics while remaining fully 6TiSCH compliant.

[1]In 6TiSCH, the term of "parent" means the default router of a corresponding node. In Fig. 2, node *1* is the parent of node *2*. A parent is selected from neighbors by an RPL Objective Function [9] based on link metrics.

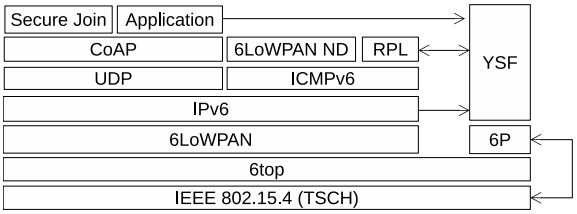


Fig. 1: The 6TiSCH Protocol Stack, with YSF.

## III. YSF

YSF is a scheduling function which minimizes end-to-end latency while maintaining high end-to-end reliability. The main target application of YSF is data gathering, traffic pattern of which is multipoint-to-point [9]. Optimizations for point-to-multipoint traffic and point-to-point traffic are outside the scope of YSF.

We assume that the sink of a data gathering system is also the root of the routing structure of the network. In this article, we use the terms "sink" and "root" interchangeably. Each non-root node is an ordinary 6TiSCH node [2], which knows nothing about the network at the beginning. A node gets aware of the network by receiving beacons, performs the join process, and then becomes part of the routing structure.

### A. YSF in a Nutshell

YSF is designed to run as part of the 6TiSCH protocol stack [18]. As show in Fig. 1, YSF sits on top of 6P [3]. YSF operates along the 6TiSCH minimal configuration [11]. YSF interacts with 6P and the TSCH MAC layer, as well as IPv6 and RPL. Because YSF exclusively uses the protocols already standardized by 6TiSCH, it is fully standards-compliant, which we believe is an important quality to have for adoption.

In order to prevent a single routing change from causing a global rebuild of the entire schedule, YSF avoids a schedule which is tightly coupled with the whole routing topology. Instead, YSF schedules cells on a per traffic flow basis. Removing or adding cells for a given traffic flow does not require any changes in cells for other flows.

In YSF, a traffic flow is defined as a sequence of packets from a source node to the sink, with the same Quality of Service (QoS). YSF proceeds per traffic flow by assigning a set of cells to each node visited by the flow, in a top-down approach. Without loss of generality, we explain only the case where each node has a single traffic flow, even though a node can have multiple traffic flows.

Fig. 2 shows a desired cell allocation for a traffic flow sourced by node *k*. The left side of the figure shows the routing path from node *k* to node *0*. The box on the right side of each node is the slotframe of that node. YSF schedules two TX cells at each hop except for the sink, and an additional pair of cells between node $k$ and node $k-1$. The number of TX cells per hop is determined by parameter $n$. By definition, the minimum value of $n$ is 1. $n$ can be set to larger than 1 to cope with frame drops. In this example, $n$ is set to 2.

The set of cells assigned on any node for a given traffic flow are scheduled one after the other. These cells are called *permanent cells*, which are optimized for application traffic to the sink.

TX cells can be used for both application packets and unicast control packets. The RX cell at the source node is used mainly for 6P communication. Having a dedicated RX cell from the parent immediately before a dedicated TX cell enables the node and its parent to complete a 2-step 6P transaction within two slots in the best case, which results in faster 6P transactions.

Since 6P is a signaling protocol between one-hop neighbors, the duration between the cell request and the cell assignment can be long. To minimize network performance degradation during this period, YSF introduces two additional types of cells: the autonomous cell and the transient cell. *Autonomous* cells are scheduled without involvement of 6P, and are aimed at completing a 6P transaction quickly with a new neighbor. *Transient* cells are mainly used for handling incoming traffic until permanent cells are scheduled. Without these types of cells, either a scheduling process cannot complete in time, or the end-to-end reliability is severely degraded. The allocation of these cells is detailed in Section III-D.

### B. Scheduling Process

A node starts a scheduling process by sending a cell allocation request to its parent. The scheduling process consists of two phases: the request phase and the allocation phase. 2-step 6P SIGNAL transactions are used in the request phase. 2-step 6P ADD transactions are used in the allocation phase.

Fig. 3 depicts a typical message sequence of the scheduling process. At the beginning of the request phase, node $k$ sends a cell allocation request for its traffic flow. The cell allocation request contains the corresponding traffic flow identifier, and the hop count to the traffic flow source. The recipient, node *k-1*, responds to node *k* and then sends a request to its parent.

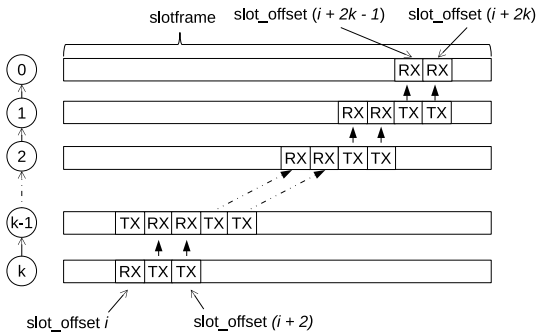



Fig. 2: The desired cell allocation for the flow sourced by node *k*. Here, $n = 2$ and node *0* is the sink.

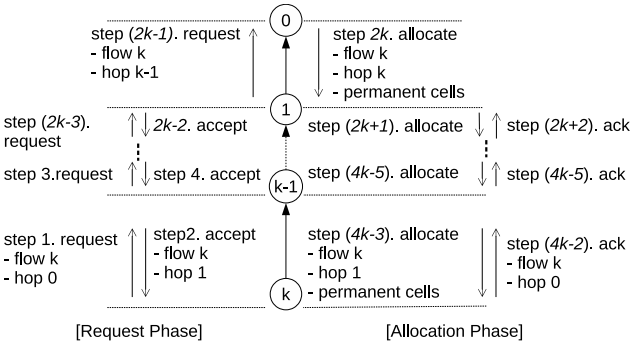



Fig. 3: A typical message sequence of a cell allocation for the flow sourced at node $k$.

The `hop_count` value is incremented at each hop. The request phase continues until the sink receives a request for that traffic flow. At the end of the request phase, the sink knows how many hops away the traffic flow source is.

In the allocation phase, permanent cells are scheduled hop-by-hop, starting from the sink. That is, YSF schedules permanent cells "top down". At the end of the allocation phase, the traffic source (node *k*) gets permanent cells scheduled with its parent. This hop-by-hop cell reservation is similar in nature to the RSVP protocol [17].

### C. Permanent Cell Allocation Algorithms

This section explains the cell allocation algorithms used during the allocation phase. This results in a collision-free schedule of permanent cells once the network has converged.

*1) Slot Offset Selection:* At the beginning of the allocation phase, the sink selects slot offsets for a new set of permanent cells for a given traffic flow, using Algorithm 1.

Three inputs are given to the algorithm. `n` is the number of TX permanent cells assigned per flow at each hop. `hop_count` is the hop count from the sink to the traffic flow source, computed by incrementing the hop count field in the request at each hop. `downward_neighbor` is the MAC address of the child from which the sink received a cell allocation request for the traffic flow. In the case of Fig. 2, `n`=2, `downward_neighbor` is the MAC address of node 1, and `hop_count`=$k$.

The main part of the algorithm is to identify available slot offsets. First, slot offset zero is excluded because this slot

**Algorithm 1 Permanent cell slot selection**

**Input:** n, hop_count, downward_neighbor
**Output:** slots

```
1:  available_slots = all slots in the slotframe
2:  available_slots.remove(0)
3:  for all existing cell_allocations do
4:      slots = get_slots(cell_allocation.permanent_cells)
5:      busy_slots = slots
6:      if cell_allocation.match(downward_neighbor) then
7:          if cell_allocation.hop_count == 2 then
8:              busy_slots += left_slots(slots, n + 1)
9:          else if cell_allocation > 2 then
10:             busy_slots += left_slots(slots, n)
11:         end if
12:         busy_slots += right_slots(slots, n + 1)
13:     end if
14:     available_slots -= busy_slots
15: end for
16: if hop_count == 1 then
17:     num_slots = n + 1
18: else
19:     num_slots = n
20: end if
21: return consecutive_slots(available_slots, num_slots)
```

offset is used for the minimal shared cell [11]. Then, slot offsets already used for existing permanent cells are excluded. If the set of existing permanent cells for a traffic flow is scheduled with the same neighbor as `downward_neighbor`, some slot offsets immediately before the permanent cells are excluded as well, because these slot offsets are expected to be used by this neighbor with its child. In addition, some slot offsets immediately after the existing permanent cells are also excluded, in order to avoid a scheduling conflict with newly scheduling permanent cells.

After that, the algorithms selects and returns consecutive slot offsets out of the remaining available slot offsets, for a new set of permanent cells scheduled with the downward neighbor. The number of returned slot offsets depends on the value of `hop_count`. If `hop_count` is equal to 1, the algorithm returns `n + 1` slots since the downward neighbor is the traffic flow source, which needs an additional RX permanent cell.

Fig. 4 depicts an example of values returned by the functions used in the algorithm. `get_slots()` takes a list of cells and returns their slot offsets. `left_slots()` takes a list of slots and a number of slots as its arguments, and returns as many slot offsets immediately before the given slots. `right_slots()` is similar to `left_slots()`, but it returns slot offsets immediately after the given cells.

At an intermediate node, once the intermediate node scheduled permanent cells with its parent, it schedules a new set of permanent cells scheduled with its child, using consecutive slot offsets immediately before the permanent cells to its parent.

*2) Channel Offset Selection:* YSF assigns a single channel offset to all permanent cells for a given traffic flow. In the example of Fig. 5, the permanent cells for the traffic flow of node 3 have channel offset $X$, while the permanent cells for

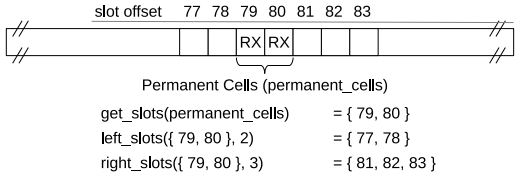



Fig. 4: Example of `slots` functions.

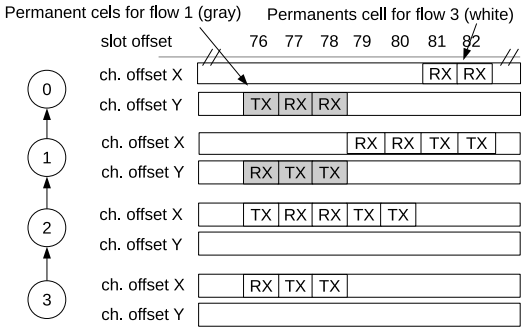



Fig. 5: An example of channel offset assignment.

the traffic flow for node 1 have channel offset $Y$.

The sink selects a channel offset for permanent cells of a given traffic flow out of channel offsets reserved for the permanent cell usage, following Algorithm 2. The channel offsets from 1 to 13 are reserved for the permanent cells in our implementation. The channel selection algorithm takes as parameters `slots_for_permanent_cells` as well as `n` and `hop_count`, which are the same definitions as for Algorithm 1. `slots_for_permanent_cells` is the slot offsets for permanent cells which the sink is going to schedule. With the three inputs, the sink can deduce all the slot offsets which will be scheduled along the routing path for permanent cells of a given traffic flow. Similarly, the sink can deduce all the slot offsets of the existing permanent cells along the routing path for a traffic flow which the sink is aware of. If there are common slot offsets between them, the channel offset assigned for the existing permanent cell is considered busy.

**Algorithm 2 Permanent cell channel offset selection**

**Input:** slots_for_permanent_cells, n, hop_count
**Output:** channel_offset

```
1:  available_channel_offsets = permanent cell channel offsets
2:  num_slots = n × hop_count + 1
3:  right_edge = right_slots(slots_for_permanent_cells, 1)
4:  all_slots_on_path = left_slots(right_edge, num_slots)
5:  for all cell_allocations do
6:      if cell_allocation.has_common_slots(all_slots_on_path)
        then
7:          permanent_cells = cell_allocation.permanent_cell
8:          busy_one = channel_offset(permanent_cell)
9:          available_channel_offsets.remove(busy_one)
10:     end if
11: end for
12: return pick_one(available_channel_offsets)
```

After iterating this procedure for all the existing permanent cells, the list of available channel offsets is obtained. The channel selection algorithm picks one channel offset among them.

*3) Complexity:* Complexity of Algorithm 1 is $O(N)$, where N is the number of nodes in a network, since the dominant part is the for-all loop starting at line 3. The loop goes through all the existing cell allocations, the number of which is $N$ at most. Similarly, complexity of Algorithm 2 is $O(N)$ because of the for-all loop starting at line 5. Any intermediate node, after having scheduled permanent cells with its parent, schedules a new set of permanent cells with its child, using consecutive slot offsets immediately before the permanent cells to its parent. This algorithm has a complexity in $O(1)$. As a consequence, the processing complexity is kept in the sink, which, unlike the other nodes, is not limited in CPU and memory capacity.

### D. Handling Parent Switching

Whenever the RPL routing protocol selects a new parent, the node triggers a new scheduling process.

If the node previously had a different parent, YSF removes all the cells scheduled with the old parent silently. The remaining cells on the old parent side are then removed automatically by an expiration timer. That is, each permanent cell has a lifetime, which is reset each time an IPv6 packet of the corresponding traffic flow is processed. With this mechanism, permanent cells for a traffic flow which has not been used for a certain period are removed when the timer expires. The lifetime should be longer than the packet interval of the corresponding traffic. At the same time, the lifetime should be reasonably short to handle cases when a traffic flow stops, or the source of the flow leaves the network. In our implementation, the lifetime is set to 300 s as default value, which was confirmed by simulation to be long enough to cover all the simulation settings in Section IV.

If the node had cells for traffic flows originating at its descendants, the existing cells scheduled with downward neighbors are kept. The node informs the new parent about the other traffic flows as well as its own traffic flow in a cell allocation request after the parent switch. The initial cell allocation request have multiple pairs of traffic flow identifier and hop count values. Cell allocations for all the traffic flows are executed concurrently, and the message sequence remains the same (see Fig. 3).

The further the node sending the initial cell allocation request is from the sink, the longer the scheduling process of YSF takes. This is critical especially for a node having many descendants which generate application packets. To prevent packet drops during the scheduling process, we introduce the autonomous cell and the transient cell. The *autonomous cell* (explained below) allows the scheduling process to complete fast. The *transient cell* (explained below) is temporary bandwidth assigned to a node to handle packets from its children during the scheduling process. Note that they are not collision-free unlike the permanent cell.

*1) Autonomous Cell:* YSF schedules autonomous cells over which a new pair of parent and child nodes perform 6P communication and keep-alive exchanges.

The autonomous cell is scheduled without the neighbor nodes needing to communicate. Each node maintains one autonomous cell to receive frames from its neighbors. Unlike Orchestra, YSF selects a slot offset randomly among its unused slot offsets for the autonomous cell. The autonomous cell of a node is moved to another randomly selected slot offset when a transient cell or a permanent cell is scheduled at the same slot offset. The channel offset for the autonomous cell is pre-defined and reserved. In our implementation, the channel offset of 15 is reserved for the autonomous cell. The slot offset of the autonomous cell is advertised in DIOs (Destination-Oriented Directed Acyclic Graph Information Objects). Since DIOs are link-layer encrypted, only joined nodes can learn the location of the autonomous cells of their neighbors. Therefore, when the RPL routing protocol selects a new parent, the node is aware of both the new parent's MAC address and where its autonomous cell is in the schedule. The child node sends a 6P request over the autonomous cell of the parent, which contains information about its own autonomous cell. This allows the parent to send back a 6P response to the child, over its autonomous cell. When transient cells are scheduled, the child removes the parent's autonomous cell from its schedule, and vice versa.

*2) Transient Cell:* Just after switching to a new parent, the node only has a single TX cell to its parent, which is the autonomous cell. If the node has many RX cells from its children, it would drop forwarding packets due to the shortage of TX cells.

In order to handle the incoming traffic until permanent cells are in place, transient cells are scheduled during the request phase of the scheduling process. A child schedules $n$ transient TX cells to its parent per flow. If the child is the source of a flow, it additionally schedules one transient RX cell, which is used mainly for following 6P communication. The channel offset of the transient cell is pre-defined and reserved. In our implementation, the channel offset of 14 is reserved for the transient cell. When a parent receives a cell allocation request, it randomly selects slot offsets for transient cells among unused slot offsets. The parent responds to the child with the transient cell information. At the end of this request-response exchange, parents and child nodes schedule the transient cells. The transient cells are removed when permanent cells are scheduled.

### E. Consideration of Joining Nodes

Every 6TiSCH node except the sink is a joining node just after booting up. A joining node needs to perform a network access authentication process, such as the secure join protocol defined in [4], to obtain the link-layer security keys.

In order to protect the TSCH schedule and the network itself from external attackers, YSF allows only nodes having the right link-layer key to participate in YSF's scheduling process. All 6P communications must be encrypted at the link layer. In addition, information about the autonomous cell must be

TABLE I: Simulation parameters.

| Parameter | Settings |
|---|---|
| Application packet interval | 5 s, 15 s, 30 s, or 60 s |
| Application packet interval randomization | ±5% |
| RPL DAO interval | 60 s |
| TSCH slotframe length | 101 slots |
| TSCH slot duration | 10 ms |
| TSCH TX queue length | 10 frames |
| TSCH keep-alive interval | 10 s |
| TSCH EB transmission probability | 0.33 |
| TSCH max. number of retransmissions | 5 |
| Number of radio channels | 16 |

conveyed in encrypted frames. As a result, a joining node uses only the minimal cell [11] until it completes its network access authentication process to get the link-layer key in use.

## IV. Performance Evaluation

### A. Simulation Settings

We compare YSF and MSF by simulation. We implement them in v1.2.0 of the 6TiSCH Simulator [7] that supports all the key protocols and simulates internal interference properly.

We used a random placement of 50 nodes for our evaluation, where the x,y coordinates of the nodes are determined randomly in a 2 km $\times$ 2 km space, for every simulation run. We use the Pister-hack connectivity model [7]. Each node has at least three links to neighbors whose link PDR (Packet Delivery Ratio) values are above 50%. The deepest routing tree observed in the simulations is 7-hop deep, the shallowest is 4-hop deep, with a median of 5-hop.

One particular node, node *0*, acts as the DODAG root; it starts forming a 6TiSCH network at the beginning of each simulation run. Node *0* is also the sink of the application. Each other node sends an application packet with a configurable inter-packet period. When MSF is used, a node sends application packets only when it has at least one dedicated TX cell to its parent. When YSF is used, a node sends application packets only when it has permanent cells to its parent.

In addition to the scheduling function, the RPL objective function is another key factor affecting performance of a 6TiSCH network. To highlight performance differences only due to the scheduling function in use, we used an objective function called BestLinkPDR. BestLinkPDR keeps selecting the best possible parent among the sources of received DIOs, using the simulator's internal data instead of evaluating them by observed ETX (Expected Transmission Count) values. Even when using BestLinkPDR, node $A$ does not select node $B$ as its parent as long as it has not received a DIO message from $B$. A node may change its parent multiple times during a simulation run. While having instantaneous knowledge about the PDR of a link is not realistic, we believe this method is a good trade-off between realism and the ability to fairly compare MSF to YSF.

We enable the join process [4]. All non-root nodes complete a join process after getting synchronized with the network, before joining the RPL routing structure.

For each combination of scheduling functions and application packet intervals, we run each simulation 100 times with different topologies. We take into account all the application packets for latency and reliability calculation, from the beginning of the simulation to the end, including the network formation phase where many parent switches happen. The duration of one simulation run is 6 h (360 min). Parameter $n$ is set to 1. Table I shows the simulation parameters.

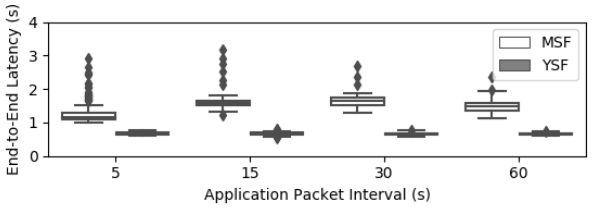



Fig. 6: Average E2E Latency, indicated on Y-axis, in seconds plotted in boxplots with outliers.

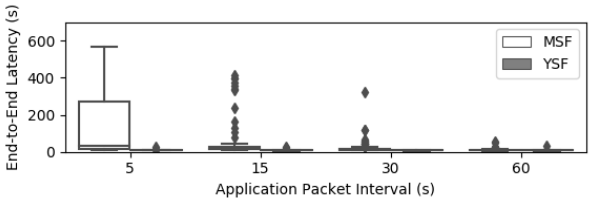



Fig. 7: Maximum E2E Latency, indicated on Y-axis, in seconds plotted in boxplots with outliers.

### B. End-to-End Latency

We call end-to-end latency the duration between the time a source node sends an application packet, and the time the sink receives that packet.

Fig. 6 shows the average end-to-end latency, each sample is an average latency calculated over all messages received by the root in a corresponding simulation run. YSF yields low latency with a small variance regardless of the application packet interval settings. In the case of a 5 s application packet interval, the median is 0.66 s, and its interquartile range is 0.05. Comparing their median values, YSF reduces the end-to-end latency of MSF by 30-60 %.

Fig. 7 shows the maximum end-to-end latency results, each sample being the maximum of a simulation run. The performance trends of YSF are almost the same as seen in the average end-to-end latency shown in Fig. 6.

These results highlight differences between YSF and MSF. YSF schedules cells for a given flow in a cascading manner, which make application packets forwarded without delay. This is not the case with MSF that schedules cells at randomly selected positions. In addition, YSF has less chance for retransmissions to happen than MSF since a resulting schedule of YSF is collision-free.

### C. End-to-End Reliability

We compute the end-to-end reliability as the ratio between number of application packets received by the sink, and the number of application packets generated during the simulation.

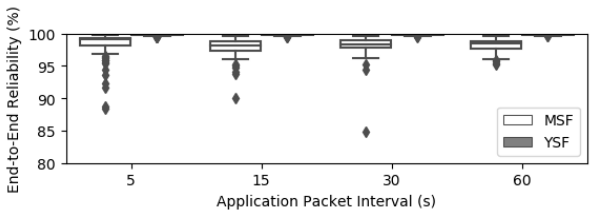
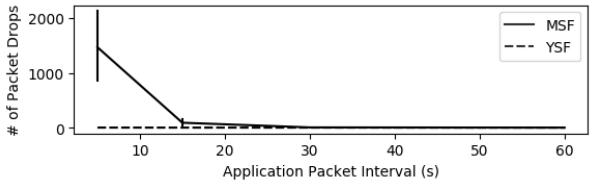
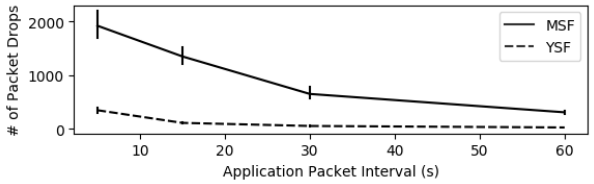



Fig. 8: End-to-End Reliability, plotted in boxplots with outliers. Y-axis indicates reliability (%).




(a) TX Queue Overflow




(b) TX Failure

Fig. 9: Application Packet Drops, averaged over 100 simulation runs and plotted with a 95 % confidence interval.

As shown in Fig. 8, YSF keeps high reliability with a small variance regardless of the application packet interval. Even with a 5 s application packet interval, YSF yields 99.97 % of the highest reliability, and 99.36 % of the lowest reliability. While MSF has 99.66 % in the best case, it has larger variance and more outliers, especially with an application packet interval of 5 s.

In Fig. 9, we plot the number of application packet drops for each drop cause. If the TX queue is full when a new packet arrives, the packet is dropped and counted as TX queue overflow. If a node does not receive a MAC acknowledgement after the maximum number of retransmissions (see Table I), the corresponding packet is dropped and counted as TX failure. According to Fig. 9a, MSF has many packet drops due to TX queue overflows when the application packet interval is 5 s. This is because MSF potentially schedules collision cells, and because cells are not scheduled in order, resulting in fuller queues. YSF addresses both these shortcomings.

Looking at Fig. 9b, MSF has more TX failures than YSF. As the objective function of BestLinkPDR selects always the best possible parent for a node, the number of TX failures due to poor link quality is minimized regardless of the scheduling function in use. Therefore, the difference seen in Fig. 9b indicates that MSF has more collisions than YSF.

### D. Resulting Schedule

Fig 11 shows example TSCH schedules built by MSF and YSF at the end of the first simulation, using an application packet interval of 5 sec.

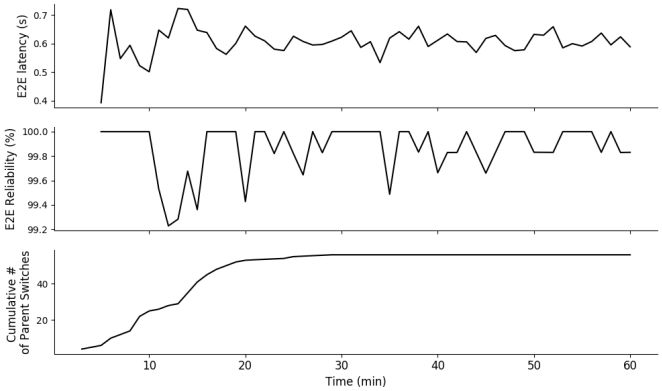



Fig. 10: Network Performance of YSF over Time. Results are taken from the first 60 min of a simulation run of the application packet interval of 5 s.

In MSF, two cell allocation patterns are observed: autonomous cells allocated diagonally in the schedule, and dedicated cells allocated randomly. MSF has several TX cells where potentially more than one transmission occurs at the same time and cause collisions. Although MSF has a mechanism to resolve collision cells by relocation, it is reactive and takes time to detect them.

On the other hand, YSF yields a collision-free schedule of sets of cells allocated contiguously. There are no cells found at channel offset 14 and 15, which are reserved for the transient cell and the autonomous cell, respectively. These cells are meant to be used only during the scheduling process, and are removed at the end of the process.

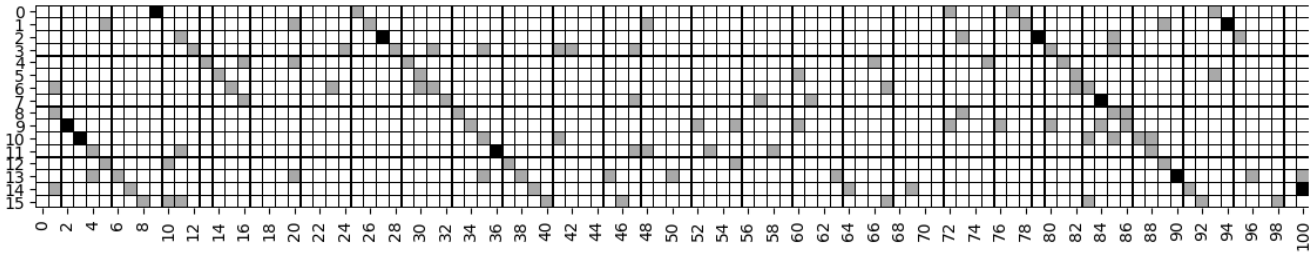
### E. Performance Stability of YSF

As discussed in Section III, YSF is designed to deal with network dynamics of 6TiSCH. Fig 10 shows average end-to-end latency and average end-to-end reliability evolving over time, during the first 60 min of a simulation run. Cumulative numbers of parent switches are shown as well in Fig 10.

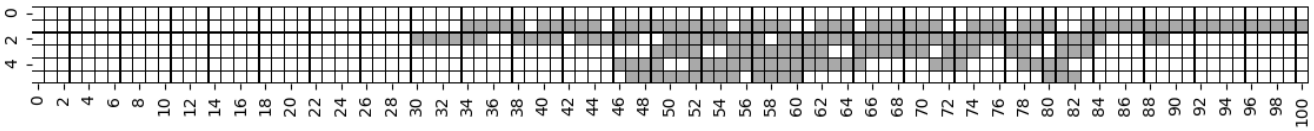
Despite topological changes during that period, the network formation phase, YSF achieves over 99% reliability and sub-second latency, constantly. Note that more control packets, which could impede application performance, are generated during the network formation phase than after the routing topology converges.

## V. Discussion

One important property of YSF shown in Section IV is that the performance of YSF is stable regardless of the topology. The variance of each plot for YSF in Fig. 6 to Fig. 8 is small. This is a critical property for industrial applications, as YSF's performance is highly predictable. The key for this is that YSF builds a collision-free schedule in a cascading manner. With a collision-free and cascading schedule, the TX queue on a node can be kept short, which leads to stable low latency and stable high reliability. In contrast, MSF allocates cells at



(a) MSF



(b) YSF

Fig. 11: Global view of resulting schedules of MSF and YSF, from the first simulation runs (run_id 0). X-axis is the slot offset; Y-axis is the channel offset. A gray cell represents a collision-free TX cell, where there is only one transmitter. A black cell represents a TX cell, where there is more than one transmitter. In YSF, no cells are scheduled at channel offsets from 6 to 15.

randomly selected positions, which tends to cause packet drops by collision and queue overflow as highlighted in Fig. 9.

YSF builds a collision-free schedule of permanent cells without any assumption on routing topology. The existing top-down scheduling functions introduced in Section II assume that any node at depth $h$ has no radio interference with any other node at depth $h + W$ on the routing path. For instance, DeTAS has a variable $W$ to control how frequently a channel offset is reused on a routing path. DeTAS assigns the same channel offset every $W$ hops, however, there is no guarantee that $W$ is a safe distance to avoid radio interference in any real network deployment.

Currently, YSF allocates the same number of permanent TX cells at each hop regardless of QoS level of a traffic flow. However, some traffic flow may need more permanent TX cells than others, for reliability reasons or for higher transmission rates. Additionally, it would be better to adapt parameter $n$ to the link quality, and take ETX into account. Introducing more flexibility on $n$ is one of the possible extensions to YSF.

As future work, we plan to implement YSF on an actual 6TiSCH stack such as OpenWSN and Contiki-NG.

## VI. Conclusion

In this article, we propose a distributed 6TiSCH scheduling function optimized for data gathering applications. Simulation results show that YSF yields low end-to-end latency and high end-to-end reliability, regardless of network topology. Cells assigned by YSF minimize TX queue length at each node. Unlike other top-down scheduling functions, YSF does not rely on any assumption regarding network topology or traffic load, and is therefore more robust in real network deployments.

## References


[1] *802.15.4-2015 - IEEE Standard for Low-Rate Wireless Networks*, IEEE Std., April 2016.

[2] X. Vilajosana, T. Watteyne, T. Chang, M. Vučinić, S. Duquennoy, and P. Thubert, "IETF 6TiSCH: A Tutorial," *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 595–615, Firstquarter 2020.

[3] Q. Wang, X. Vilajosana, and T. Watteyne, "6TiSCH Operation Sublayer (6top) Protocol (6P)," RFC8480, IETF, 2019.

[4] M. Vučinić, J. Simon, K. Pister, and M. Richardson, "Minimal Security Framework for 6TiSCH," draft-ietf-6tisch-minimal-security-15, IETF, 2019.

[5] T. Chang, M. Vučinić, X. Vilajosana, S. Duquennoy, and D. Dujovne, "6TiSCH Minimal Scheduling Function (MSF)," draft-ietf-6tisch-msf-18, IETF, 2019.

[6] A. Seferagić, J. Famaey, E. De Poorter, and J. Hoebeke, "Survey on Wireless Technology Trade-Offs for the Industrial Internet of Things," *Sensors*, vol. 20, no. 2, p. 488, 2020.

[7] E. Municio *et al.*, "Simulating 6TiSCH networks," *Transactions on Emerging Telecommunications Technologies, Wiley*, vol. 30, no. 3, Mar. 2019.

[8] M. R. Palattella *et al.*, "On-the-fly bandwidth reservation for 6TiSCH wireless industrial networks," *IEEE Sensors Journal*, vol. 16, no. 2, pp. 550–560, 2016.

[9] R. Alexander *et al.*, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," RFC6550, IETF, March 2012.

[10] S. Duquennoy, B. A. Nahas, O. Landsiedel, and T. Watteyne, "Orchestra: Robust mesh networks through autonomously scheduled tsch," in *ACM SenSys 2015*, Seoul, South Korea, November 2015, p. 337–350.

[11] X. Vilajosana, K. Pister, and T. Watteyne, "Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration," RFC8180, IETF, May 2017.

[12] A. Kalita and M. Khatua, "Autonomous Allocation and Scheduling of Minimal Cell in 6TiSCH Network," *IEEE Internet of Things Journal*, 2021.

[13] O. Tavallaie, J. Taheri, and A. Y. Zomaya, "Design and Optimization of Traffic-Aware TSCH Scheduling for Mobile 6TiSCH Networks," in *ACM IoTDI '21*, May 2021, pp. 234–246.

[14] T. Chang, T. Watteyne, Q. Wang, and X. Vilajosana, "LLSF: Low Latency Scheduling Function for 6TiSCH Networks," in *IEEE DCOSS 2016*, May 2016, pp. 93–95.

[15] G. Daneels, B. Spinnewyn, S. Latré, and J. Famaey, "ReSF: Recurrent low-latency scheduling in IEEE 802.15. 4e TSCH networks," *Ad Hoc Networks, Elsevier*, vol. 69, pp. 100–114, Feb. 2018.

[16] 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 *IEEE WoWMoM 2013*, June 2013, p. 1–6.

[17] A. Morell, X. Vilajosana, J. L. Vicario, and T. Watteyne, "Label switching over IEEE802. 15.4 e networks," *Transactions on Emerging Telecommunications Technologies, Wiley Online Library*, vol. 24, no. 5, pp. 458–475, 2013.

[18] P. Thubert, "An Architecture for IPv6 over the TSCH mode of IEEE 802.15.4," draft-ietf-6tisch-architecture-30, IETF, 2019.