plexi: Adaptive re-scheduling web service of time synchronized low-power wireless networks

G. Exarchakos[a,*], I. Oztelcan[a], D. Sarakiotis[a], A. Liotta[a]

[a]*De Zaale, 5612AZ Eindhoven, The Netherlands*

Abstract

Industrial IoT applications require highly dependable monitoring and actuation capabilities of remote low power devices. Time scheduling with channel hopping has been a well attested mechanism to address these requirements in volatile environments. Yet, scheduling algorithms to date are not adaptive enough to changes in deployed applications and their environments. *plexi* is a restful web service API for monitoring and scheduling IEEE802.15.4e network resources hiding the complexity of schedule deployment and modification. On top, *plexiflex* allows any given scheduler to adapt to network performance changes by monitoring periodic data streams coming from the nodes. It triggers resource (de)allocation aiming at stable network performance. Both *plexi* API and *plexiflex* adaptive rescheduling algorithm allow for interoperability among devices, schedulers and applications. Experiments to real TSCH network deployments have shown significant gains of *plexiflex* compared to fixed offline scheduling. Via monitoring the interarrival time of those stream data packets, *plexiflex* can identify wiring new node joining and rewiring events and reschedule when and where is needed.

Keywords: `internet of things`, resource management, time scheduling, channel hopping, CoAP, adaptive monitoring, reconfigurable networks
2015 MSC: 00-01, 99-00

1. Introduction

Dependable industrial IoT infrastructures are becoming more realistic thanks to recent advances on time scheduled channel hopping wireless sensor networks. Besides the proprietary WirelessHART [1], the upcoming TSCH IETF standard [2, 3] harnesses the advantages of time scheduling and channel hopping to deliver dependability of the wireless connections in volatile environments. The challenge of those protocols is to adapt to changing industrial application

*Corresponding author
Email addresses: `g.exarchakos@tue.nl` (G. Exarchakos), `i.oztelcan@tue.nl` (I. Oztelcan), `d.sarakiotis@tue.nl` (D. Sarakiotis), `a.liotta@tue.nl` (A. Liotta)

requirements and environment conditions [4, 5] via generating new transmission schedules for nodes. Even though various node scheduling techniques have been proposed, their adaptation at runtime to different environments and QoS requirements is limited. A transmission schedule devised for one application under certain conditions might be ineffective for another application or under different conditions.

The current work enables applications to configure and trigger at runtime new schedules of data transmissions after monitoring the network's performance. Re-scheduling TSCH networks to fulfill the expectations of one or more applications implies the network resources are continuously monitored and communication links created, moved, deleted over time. For interoperability and scheduling purposes, *plexi* exposes TSCH network resources e.g. communication channels and timeslots via a restful web interface for low power devices known as CoAP [6]. For monitoring, *plexi* captures the L2 and L3 network topology as well as performance metrics per link per node such as the number of retransmissions, packet delivery ratio, link quality indicator and received signal strength.

The second contribution of this work is *plexiflex*, a mechanism which triggers any given scheduling algorithm to modify at runtime the schedule of (a part of) a network. Based on the expectations of an application and the performance of that network, *plexiflex* may decide e.g. to postpone rescheduling if the performance degradation is predicted to be short and/or insignificant. While *plexi* provides access to IEEE802.15.4e network resources and performance metrics, *plexiflex* enables adaptivity in that network. To date, there is little work on adaptive re-scheduling of those networks [7, 8] with scheduler and adaptation control unit tightly-coupled.

The following two sections give an in-depth analysis of related work and background knowledge (Section 2), the *plexi* overall architecture and features (Section 3) and the *plexiflex* adaptive rescheduling mechanism (Section 4). We continue with the evaluation of *plexiflex* in Section 5 before the conclusion and future work at Section 6.

2. Background & Related work

The positive effects of time scheduling and channel hopping have been examined in several studies including [9, 3, 10]. This section provides an overview of background knowledge and latest developments on adaptive scheduling for IEEE802.15.4e-2012 TSCH protocol.

TSCH. Nodes in Time Slotted Channel Hopping (TSCH) networks are synchronized on a time reference set by the PAN coordinator and trickled down to all nodes by a tree directed acyclic graph (DAG) topology. A new node may join the network by hearing for Enhanced Beacons from other already joined nodes. The Enhanced Beacons carry synchronization information for the new node to use and synchronize while joining. The time progresses in timeslots, the minimum time unit usually equal to 10ms. Timeslots are packaged into repetitive slotframes. Each timeslot is sufficient for a node to transmit a packet and wait for the acknowledgement. Hence, a schedule is effective if communicating nodes are both awake at that period, one set for transmission and the other for reception. Non active nodes at a timeslot go to sleep.

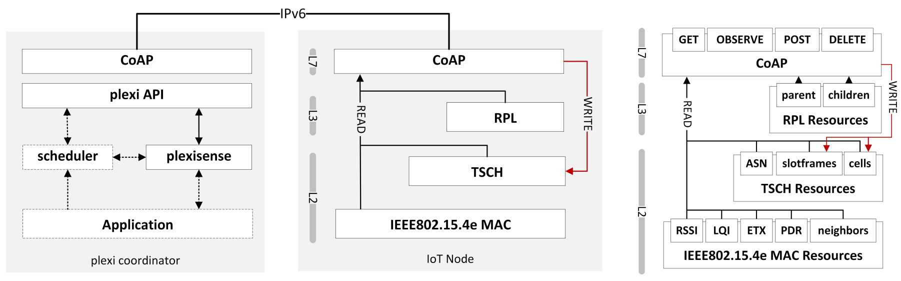
Time is the one dimension of sloframes measured in number of slot offsets; channel offsets is the second. Nodes in a TSCH network use one of 16 frequencies available at 2.4GHz at a given timeslot. Each frequency is mapped to a channel offset in the slotframe matrix. The channels shift by one position at every timeslot so that same slot offset is not likely to be allocated the same physical frequency when the slotframe is repeated. The slotframe matrix, composed of slot and channel offsets, is the scheduled template that all nodes adhere and repeat. There might be multiple slotframes running in parallel.

The second pillar of *plexi* is the Routing Protocol for Low-power Lossy networks (RPL) [11]. RPL is a routing protocol that creates an acyclic directed tree rooted (DoDAG) to one node, the border router (LBR). Each node is assigned a rank that indicates some kind of distance of the node from the root. That distance is defined and calculated by the *objective function* by combining several metrics and constraints. Based on this objective function, a node chooses the next hop in a route as well as the father in the tree. Every node broadcasts a DIO message to help other nodes in range tune their parameters to those of the RPL instance active at that moment. Nodes in range whose parent is the transmitter of DIO reply with a DAO containing routing information destined upwards.

Efforts on scheduling TSCH networks have shifted to decentralized schedulers. The authors of [12] presented the first decentralized algorithm which allocates one frequency to facilitate new nodes joining the network and one dedicated slot for targeted communication. DeTAS [13] was another decentralized scheduling technique which relies on exchanging traffic information between RPL children and their parent. That allows the parent to allocate enough resources to serve the predicted traffic. All nodes follow a common macro-schedule which consists of micro-schedules (one per subtree) calculated in a distributed way. Wave [8] is a distributed scheduling algorithm that splits the slotframe into regions (waves). All nodes with at least one packet to be sent are allocated a cell in the first wave and as long as they have more packets, they are allocated extra cells in subsequent waves. Finally, orchestra [7] is an autonomous distributed scheduler which schedules one upstream and one downstream cell per neighbor without communication between neighbors.

Though the advantages of decentralized schedulers are well documented, their limitation lies on the fact that they cannot have a complete view of the conditions in the network making optimization harder. Decentralized schedulers are mostly using local information at every node. As explained in [14, 15, 16, 17] local node metrics like link quality estimation are not as effective as time-based metrics for scheduling and QoS optimization purposes. Time-based metrics usually consider end-to-end traffic conditions that a centralized scheduler would be easier to identify. This is why *plexiflex* relies on interarrival time to trigger a re-scheduling action.

TASA [18] was designed to rely on traffic patterns in the network known



(a) Architecture overview of *plexi* components. (b) *plexi* resources.

Figure 1: *plexi*: Centralized scheduling and monitoring of IEEE802.15.4e networks over IPv6 and CoAP.

a priori. Based on the queue length of the nodes and using graph coloring and matching techniques, it generates conflict-free schedules. For completeness, TMCP [19], JFTSS [20] and MODESA [21] are other paradigms of centralized schedulers. *plexiflex* could cooperate with any of them.

3. *plexi*: TSCH scheduling interface

plexi essentially lies on a client-server software design pattern with the server residing at each IoT node and the client being any centralized scheduling or monitoring entity. Each node exposes certain L2 and L3 resources via a CoAP interface. Monitoring or scheduling entities collect data from node's exposed resources and devise a new schedule if necessary. Fig. 1a illustrates the components of *plexi* and the interaction between an IoT node and the scheduler over *plexi* via IPv6.

3.1. Architecture overview

At the *plexi* coordinator end, an application is any entity that combines monitored data and notifies a scheduler for expected performance on i.e. latency, packet delivery ratio, energy consumption etc. It is up to the scheduler to decide the suitable actions i.e. add, delete slotframes or cells to satisfy those requirements. Both the application and the scheduler may instruct *plexisense* to start monitoring metrics on a subset of nodes. Besides simple on-demand monitoring, *plexisense* can handle event- or time-based notifications generated by the node. The values collected by *plexisense* are delivered back to the requesting scheduler or application. *plexi* API translates all read/write requests coming from the scheduler or *plexisense* to CoAP messages making sure they are sent in the right order and CoAP error messages are handled. For instance, no cells can be added before the slotframe they belong to is installed to the target node.

At the other end, the IoT node receives *plexi* CoAP messages and translates them to either scheduling or monitoring actions. To satisfy those actions, *plexi* at the node exposes resources from the IEEE802.15.4e MAC, the TSCH sublayer and RPL. While resources from MAC are important for monitoring the performance of each cell, RPL is needed to efficiently navigate the whole network, rebuild snapshots of the topology and more accurately estimate the traffic flows, information useful to several schedulers. Besides reading capabilities, *plexi* modifies the TSCH sublayer by adding or removing slotframes and cells as requested by *plexi* coordinator. Though not illustrated in Fig. 1a, *plexi* coordinator may also ask for modifications on the monitored resources at a node. A resource may need to be configured i.e. if and when changes at the value of the resource should be notified back to *plexisense*.

plexi implements part of YANG model [22]. The CoAP representation of resources specified in that model is an ongoing work [23] which, however, seems currently inactive. We got inspiration but did not stick to that work for our implementation. The following subsections present the resources implemented by *plexi*, their CoAP interface and their behavior.

3.2. plexi resources

Fig. 1b illustrates the resources implemented by *plexi*. Cell performance statistics are retrieved by IEEE802.15.4 MAC and include expected transmission count (ETX), packet delivery ratio (PDR), received signal strength (RSSI), link quality indicator (LQI) as well as the neighbors from who enhanced beacons can be received. *plexi* exposes TSCH sublayer resources for both read and write access i.e. slotframes and cells. Finally, each node provides own RPL DoDAG parent and children.

As shown in Fig. 2, *plexi* uses either existing structures in the three layers to access those resources or defines new ones. All structures used are extended with certain behavior. These structures support at least one behavior out of: read as is, read aggregates and write (create, modify, delete). The first two behaviors are mapped to `GET` and OBSERVE CoAP methods (the latter being same as former with some extra options). The "write" behavior is mapped to POST and DELETE CoAP methods. To interoperate with other debugging tools, e.g. Copper [4], *plexi* uses JSON objects or arrays for message payloads. Given the memory limitations in low power devices, JSON payloads may easily become too big to fit in limited-size buffers. We assume that (de)compression algorithms are used when *plexi* messages are sent or received. The following paragraphs present the possible CoAP requests, the URI, the payload (if any) and the returned error codes per resource and subresource.

DoDAG is a structure populated with data coming from RPL parent and children and accessed on-demand via `GET` or monitored for changes via OBSERVE. Slotframe and cell lists are non-observable structures maintained in TSCH either retrieved via `GET` or `DELETE`d or created via POST. Besides attributes of the cells, the cell list maintains and allows observers on the values of cell statistics (if any). Neighbor list is added to provide on-demand (`GET`)and on-change (OBSERVE) read access to statistics per neighbor aggregated out of all the cells through which the node can communicate with that neighbor. Statistics structure is added to maintain the configuration of performance metrics per cell (if any) requested by *plexi* coordinator. These configurations may

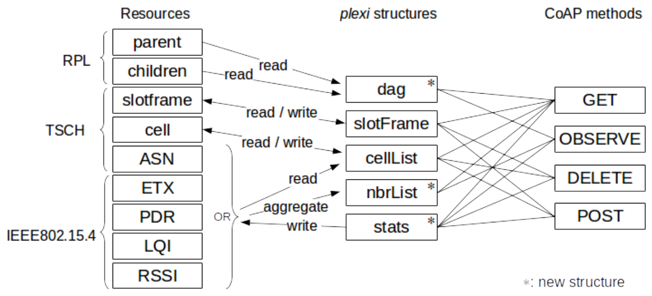


Figure 2: Mapping of node resources to *plexi* structures and CoAP methods.

Table 1: CoAP *plexi* interface template.

Request	**Response**
`Method(s)-URI`	`Code-Payload`
`GET`$^+$`,DELETE-base`	`x.xx-[{sub:val,...,sub:val},...]`
`GET`$^+$`-base/sub`	`x.xx-[val,...,val]`
`GET`$^+$`,DELETE-base?sub=val[&sub=val[...]]`	`x.xx-[{sub:val,...,sub:val},...]`
`GET`$^+$`-base/sub?sub=val[&sub=val[...]]`	`x.xx-[val,...,val]`
`POST-base {sub:val,...,sub:val}`	`x.xx`

`sub`:subresources of `base`, `val`:value of a subresource, `x.xx`: CoAP return code,
`+`:OBSERVE possible, `...`:more elements of same type

be retrieved via `GET`, `DELETE`d and created or even modified via POST. Note that this structure does not maintain the values of the requested metrics but solely their configuration. Their values are maintained in cell list and in an aggregated form in neighbor list.

Tab. 1 summarizes the interface of *plexi* resources. A `GET` operation may be observable and supports subresources and queries. Though one only level of subresources is supported, a request might have multiple query key-value pairs. A `DELETE` request cannot support subresources but the resource to be deleted may be specified using any amount of query key-value pairs. A `POST` request does not support subresources or queries and may contain one only JSON object in the payload. However, responses on `GET` and `DELETE` may contain payloads with array of elements incl. JSON objects. Tab. 2 provides a list of *plexi* resources, subresources, their URIs, which of those may be queried and which URIs may be observable.

Schedule. Though *plexi* focuses on centralized schedulers, it enables hybrid scheduling solutions, as well. CoAP `GET` method on slotframes and cells allows an external scheduler for capturing the current state of a node's schedule. `POST` and `DELETE` allows for modifying the current state. The `base`uri path accesses the complete structure of either a slotframe or a cell, subresources select an attribute and queries narrow the objects returned. The first section of Tab.

Table 2: List of *plexi* resources.

Name	`sub`	`val`	**Query**	**Observe**	**Delete**
Slotframes	template: `base[/sub][?sub=val]`				
	`base:/6top/slotFrame`				✓
	`id`	set by *plexisense*	✓		✓
	`slots`	slotframe size	✓		
Cells	template: `base[/sub][?sub=val[&sub=val[&sub=val]]]`				
	`base:/6top/cellList`				✓
	`frame`	slotframe `id`	✓		✓
	`slot`	slotoffset	✓		✓
	`channel`	channeloffset	✓		✓
	`option`	link option			
	`type`	link type			
	`tna`*	target node address			
	`stats`*	[{`id`:stats `id`,`value`:statistics value}]		periodic	
Statistics	template: `base[/sub][?sub=val[&sub=val[...]]]`				
	`base:/6top/stats`				✓
	`id`	set by *plexisense*	✓		✓
	`metric`	one of {`etx`,`pdr`,`rssi`,`lqi`}	✓		✓
	`frame`	slotframe `id`	✓		✓
	`slot`	cell `slot`	✓		✓
	`channel`	cell `channel`	✓		✓
	`enable`	one of {`0`,`1`}	✓		✓
	`window`*	# of slotframe `sizes`	✓		✓
	`tna`*	cell `tna`	✓		✓
Neighbors	template: `base[/sub][?sub=val]`				
	`base:/6top/nbrList`			periodic	
	`tna`	cell `tna`	✓		
	`asn`	last timeslot a msg was sent by `tna`			
	`etx`*	stats `metric` value			
	`pdr`*	stats `metric` value			
	`rssi`*	stats `metric` value			
	`lqi`*	stats `metric` value			
DoDAG	template: `base`				
	`base:/rpl/dag`			on change	
	`parent`	[parent node address]			
	`child`	array of children node addresses			

`sub`:subresources of `base`, `val`:value of a subresource,
`*`:not always present, `...`:more elements of same type

2 demonstrates the access to slotframe resource and its subresources. Note that *plexisense* sets the handle (`id`) of the slotframe and can query slotframes either by that `id` or by size (number of `slots`); two query key-pairs are not supported. Slotframes may be `DELETE`d either altogether using the `base`URI or one by one using the `base`and querying the `id`. As of the second section of Tab. 2, any subresource is supported in the uri path but queries are allowed on the slotframe handle, slotoffset and channeloffset only. If all the three are present in the query part, the cell is uniquely identified. Otherwise, all the cells of a slotframe or a slotoffset or a channel offset or any intersection of those is accessed. The same holds for `DELETE`; `DELETE`on `base`wipes out the whole schedule and using a query on slotframe and/or slotoffset and/or channeloffset more targeted cells may be deleted.

Statistics. Communication data collection spans to both TSCH sublayer and IEEE802.15.4 layer. Absolute Slot Number (ASN), the time unit used by TSCH, is the counter of timeslots passed since the beginning of the network. This information is maintained in TSCH. ETX, PDR, RSSI and LQI are retrieved from IEEE802.15.4. All these five metrics are measured per transmission per cell. However, ASN is a per-neighbor and the remaining four are per-cell metrics. The maintained value of ASN is the latest timeslot on which a message was received from a specific neighbor. The maintained value of each of the remaining metrics results from the exponential weighted moving average (EWMA).

Statistics structure should be seen in combination with cells and neighbors structures (see Tab. 2). Statistics are configured via `/6top/stats` and collected on-demand or by observation (periodically) via `/6top/cellList` and `/6top/nbrList`. If no statistics on a cell are configured, `/6top/cellList/stats` is not present in cells resource. Note also that the cells resource and all its subresources are not observable, except for `/6top/cellList/stats`. While the cell list is providing the detailed statistics per cell, neighbor list structure maintains an aggregated version of the same statistics per neighbor. In neighbor list, the ETX, PDR, RSSI and LQI values are the average of the corresponding values of all cells and slotframes per metric and neighbor. On the contrary, ASN (`/6top/nbrList/asn`) is always present in neighbors resource and is not configured via statistics structure.

Routing. RPL information (parent and children) are extracted via the DoDAG structure. Besides on-demand DoDAG information extraction, *plexisense* may also subscribe and receive notifications on DoDAG changes. Tab. 2 provides the CoAP interface of that resource at the end. No subresources or queries are currently supported for this resource.

3.3. Implementation constraints

A closer look at implementation decisions will help understand the behavior of *plexi*. *plexi* at node end currently supports Contiki 3.x [24] only. That restricts the CoAP engine used to Erbium [25], as well. Erbium does not comply with the complete CoAP specification for security purposes. For instance, `POST` messages with a JSON array as 0-level structure is not accepted. This limits the amount of structures transferred over CoAP to one or forces *plexi* to wrap multiple structures into some form of JSON object. This limitation appears at `POST` messages only; `GET`, `OBSERVE` and `DELETE` operations may respond with arrays of JSON structures. At this moment, *plexi* supports one only structure per `POST` aiming for multiple in the next version.

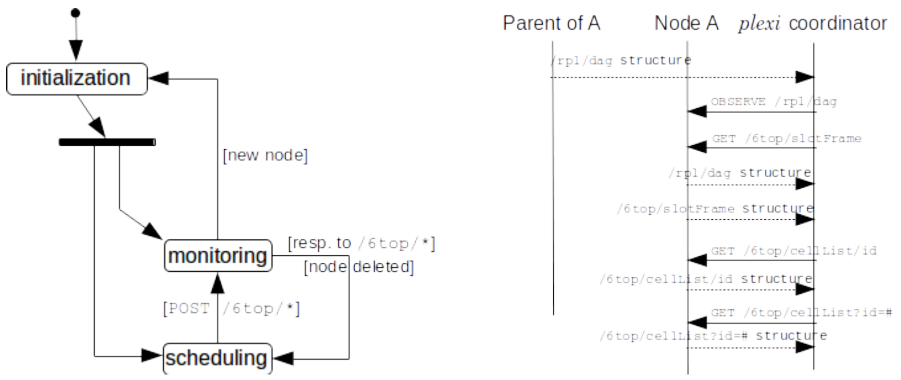
Some implementation decisions deviated from the (drafts of) IETF standards for practical reasons. As shown in Tab. 2, it is just the slotframes (`/6top/slotFrame`) and statistics configurations (`/6top/stats`) that may be accessed by their identifier (`id`). For slotframes, the `id` allows a scheduler prioritize overlapping slotframes without spending extra processing or communication time. Without an identifier, to identify a statistics configuration all attributes must be specified. For instance, the same cell may collect statistics on the same `metric` but for different `window` size; that makes it another configuration. To avoid sending `GET` commands with a complete list of attributes in the query part, `id` makes it easier.

Memory usage in *plexi* can easily become a hurdle if many cells and/or many statistics configurations are defined per cell. Due to strict memory limitations of devices Contiki 3.x is expected to run on, *plexi* at node end uses EWMA to generate the statistics values. That is, `window` attribute is not used. As shown in Tab. 1b, statistics values from cells may be retrieved either on demand or periodically. *plexi* reinterprets `window` attribute of a statistics configuration as the time interval (in number of slotframe size) between two notifications of the value back to subscriber *plexisense* .

This approach is not followed with the neighbor-specific `ASN` metric. `ASN` is, therefore, handled separately with a memory allocation process per neighbor. Subscribers to `/6top/nbrList` i.e. *plexisense* get notifications iff a neighbor was added to or removed from the list and every 10 minutes; Contiki 3.x periodically confirms the link with a neighbor every 10min.

To reduce the memory footprint of *plexi* , besides the specifications in [22], we also implemented a smaller memory footprint version of the statistics configuration. In that IETF draft, statistics configuration structure i.e. `StatisticsMetricsList` uses 16-bit numbers for the `id`, `window` and `value` and 8-bit numbers for `metric` and `enable`. That brings a 8-byte memory footprint per statistic configuration. We reduced that footprint to just 2-bytes by allocating 5 bits to `id`, 1 bit to `enable`, 4 bits to `metric` and 6 bits to `window`. This compression lies on the assumption that no more than 32 configurations will be `POST`ed to a single node, no more than 15 metrics will be monitored and the value of each statistics metric will be published at maximum every 64 slotframe sizes.

Aggregate statistics in neighbor list are not exactly the direct average of statistics kept in cell list per neighbor. This is because of the receive broadcast cells. In cell list, a metric on a receive broadcast cell refers to messages received by any source. However, neighbor list needs to differentiate those sources and aggregate these values per neighbor. *plexi* has defined separate structures to handle that so that neighbor list aggregates on all messages from either receive



(a) *plexi* states (b) *plexi* initialization state

Figure 3: *plexi* interaction engine of coordinator

broadcast or unicast cells.

Finally, some requests e.g. `GET /6top/cellList` are too generic and may return very big payloads. Though *plexi* can support those requests via CoAP block-wise transfers, the limited size memory at low power devices set also constraints to the size of buffer a response may use. For instance, if the response buffer size is set to 192 bytes and the maximum transfer unit is 64 bytes, no more than 3 blocks can be transferred. Cells are big structures and only few may be transferred within 192 bytes. Therefore, *plexi* deviated from the TSCH viewpoint and allows `GET`cells by cell-`id`. The complete schedule of a node may be fetched by, first, a `GET/6top/cellList/id` which responds with an array of cell-`id`s and, second, a sequence of `GET/6top/cellList?id=val`.

3.4. plexi-node interaction

Besides all the resource-level interaction options provided by *plexi* API, *plexisense* provides a synchronization service to any scheduler on top. That is, it supplies a scheduler with the latest L3 network topology and TSCH schedule. The former allows the scheduler know the minimum set of connections it needs to supply cells to. The latter allows *plexi* connect and start monitoring/scheduling a network at any time as well as it prevents the scheduler from unintentional overwriting of existing cells. This initialization state is succeeded by scheduling and monitoring states as shown in Fig. 3b. Monitoring state handles any reply from `/rpl/dag` and `/6top` resources. Via the former, the coordinator detects new or deleted (departing or rewired) nodes in DoDAG tree. New nodes trigger an initialization phase and deleted nodes trigger scheduling phase so that the schedule of their neighbors is adjusted to the change.

The initialization state is crucial for network state capturing and keeping the scheduler up to date. Once a new node joins the DoDAG tree, the coordinator subscribes (`OBSERVE`) to the node's `/rpl/dag` so that subsequent changes are detected. The node state capturing continues with slotframes and cells.

plexisense `GET`s the `/6top/slotFrame` structure of the new node and reinstates the same slotframes internally. Only after `GET/6top/slotFrame` has returned a response, does *plexisense* captures the complete list of cells. As mentioned in section 3.3, a cell is a big structure and a single CoAP request would be able to retrieve more than 2-3 structures. Hence, an array of cell `id`s is retrieved (`GET /6top/cellList/id`) and then iteratively *plexisense* captures cell by cell (`GET /6top/cellList?id=val` where `val` is the id value of the cell.

4. *plexiflex*: Adaptive rescheduling

In low power networks, resource over- or under-utilization has direct impact to reliability, latency and energy performance metrics. On the one hand, packets are dropped if buffers are overflown, or delayed in long queues and, on the other hand, energy is wasted if the transceivers stay on without being used. *plexiflex* allows for adaptive scheduling of TSCH resources according to traffic needs by processing the latency on a continuous stream of periodic data from every node. *plexiflex* sits in between *plexisense* and any given scheduler. The aim of *plexiflex* is to decide when and which node a scheduler should re-schedule. It tries to sustain the performance of the network by detecting drifts and acting upon them via the scheduler. It is up to the scheduler to satisfy the expectations of the application. Moreover, *plexiflex* demonstrates the functionality of *plexi* since it involves a number of both monitoring and scheduling interactions with all the nodes in the network.

In collection tree topologies with dominant upstream traffic (leaves to root), nodes closer to root receive higher traffic stress and, therefore, need more resources to handle it on time and reliably. Several approaches focus on modeling a-priori or even at runtime the traffic patterns and allocate timeslots to nodes accordingly. From a centralized entity, this is a tedious work that relies heavily on thresholds and involves extra signaling adding extra stress in the network. In a volatile situation, fine tuning or detecting the appropriate thresholds by direct monitoring of queues is a misleading approach. Relying on statistics collection to assess network performance and trigger rescheduling fails when the network experiences excessive load. Instead, *plexiflex* lies on the hypothesis that the variance of the latency of periodic data from a certain node increases with the number of sibling nodes and the depth of that node's subtree. In fact, in a tree all the data upstreams of a subtree compete for the same resources of their sub-root's uplink. *plexiflex* adjusts the allocated cells at each node based on the variance of end-to-end latency on that node's periodic upstream data.

4.1. Design of plexiflex

Without on-purpose monitoring of network performance a lot of metrics become invalid. While ETX may indicate link conditions, it cannot provide conclusive information about queue lengths and traffic load. In fact, even though ETX may be low, queues might be overflown and packets dropped (infinite latency). Moreover, a side effect of increasing ETX is latency. End-to-end latency is an indication of low link performance, long or full queues and packet drop rate. However, measuring end-to-end latency in low power networks is not a viable option.

plexiflex works with the variance of the interval between two consecutive data receptions from the same stream of periodic values. Note, the period of the upstream is not important. It is important, though, that the period stays fixed over time so that variance is noticeable. In TSCH networks, the time drifts are small enough for *plexiflex* to handle. Based on certain history, *plexiflex* needs to decide whether the performance of the network has been altered and rescheduling is needed. Very short history would trigger re-scheduling too often and very long history would react too late to changes. At the core of *plexiflex* is the Adaptive Window algorithm, `ADWIN2` [26]. Basic concept behind `ADWIN2` is that the history window expands and includes new data points as long as the mean value does not change more than a threshold. As soon as the mean value changes, `ADWIN2` calculates a part of that history window to drop.

Assuming a periodic data stream from a node to *plexisense*, *plexiflex* timestamps each data point and gets the interval from the previous data point of the same stream. This interval is stored in `ADWIN2` adaptive window. The variance of the intervals of the arriving data points is calculated based on that window. `ADWIN2` shortens or expands the window to reflect network load conditions. The window shrinks as the average interval changes. It expands as the latency stays the same. However, longer Tx queues in the path of that data stream is not necessarily reflected by higher average latency. Given the periodicity of the data, an interval that was prolonged because of a delayed data point (message) might cause the next interval to be very short if the next data point arrives on time. This may happen if multiple streams going through the same uplink have the same priority in the Tx queue and are randomly selected to be processed. For low priority data streams, overpassed by higher ones the change in the interval might be more permanent and, hence, the first `ADWIN2` layer would suffice to trigger a reaction.

The second `ADWIN2` layer is used for the variance. The variance, calculated based on the adaptive window of intervals maintained in the first layer, is fed to another `ADWIN2` algorithm. As the variance increases, it is an indication of extra load on the node. The second layer will then adapt the window and *plexiflex* will notify the scheduler to add or remove a cell to/from the uplink of the source node maintaining though at minimum one unicast cell. Algorithm 1 provides the pseudocode of *plexiflex*.

5. Performance evaluation

plexiflex is essentially an control application built over *plexi* to orchestrate any given scheduler. Hence, its performance depends on *plexi* API. In this section we present the evaluation procedure of *plexiflex* using *plexi* monitoring and scheduling capabilities.

Algorithm 1: *plexiflex* algorithm

```
Data: $src$:periodic data source node,
      $t_{last}$:timestamp of last data point,
      $adwin_{(i,src)}$:ADWIN to store intervals of stream msgs from $src$,
      $adwin_{(v,src)}$:ADWIN to store the variance of $adwin_{(i,src)}$
Result: $op \in \{ADD, REMOVE, NONE\}$
1  $now \leftarrow$ current time in miliseconds;
2  $interval = now - t_{last}$;
3  $i_{avg} \leftarrow adwin_{(i,src)}.average$ ;            // average interarrival time
4  $i_{var} \leftarrow adwin_{(i,src)}.variance$ ;       // variance of interarrival time
5  $v_{avg} \leftarrow adwin_{(v,src)}.average$ ;            // average variance of $i_{avg}$
6  $op \leftarrow NONE$;
7  $changed \leftarrow adwin_{(i,src)}.update(interval)$ ; // update adaptive window
8  if $changed$ then                  // if window resized on last update
9      if $i_{avg}$<$i_{avg}$ then
10         $op \leftarrow ADD$;
11     else if $i_{avg}$>$i_{avg}$ then
12         $op \leftarrow REMOVE$;
13 end
14 $changed \leftarrow adwin_{(v,src)}.update(i_{var})$;
15 if $op$ is $NONE$ and $changed$ then
16     if $v_{avg}$<$v_{avg}$ then
17         $op \leftarrow ADD$;
18     else if $v_{avg}$>$v_{avg}$ then
19         $op \leftarrow REMOVE$;
20 end
21 return $op$
```

Performance metrics. [t] *plexiflex* was designed to maintain a given network performance by adding/removing cells to nodes that experience changes on their load. We have identified the following metrics:

- *Detection accuracy*: percentage of network and load changes detected via indirect monitoring of periodic data streams.
- *Arrival time variance*: variance of arrival time of periodic data messages as an indication of the effectiveness of *plexiflex* at maintaining a given network performance.
- *End-to-end packet drop rate*: percentage of packets dropped at any point in their path from source to destination. This metric tests the algorithm on the hypothesis that reliability is reflected on arrival time intervals or periodic data.
- *Adaptive window size*: the number of data points kept in the adaptive

Table 3: Experiments configuration

Parameter	**1st experiment**	**2nd experiment**
background traffic	1 packet per 5 seconds	1 packets per 2 seconds
periodic data stream	1 packet every 60 seconds	1 packet every 10 seconds
number of nodes	LBR + 4 more	LBR + 4 more
minimal slotframe size	13	41
plexiflex slotframe size	11	37
EB period	10 seconds	10 seconds
Clear Channel Assessment	inactive	inactive
RPL DIO interval	4.096 seconds	4.096 seconds
plexiflex adaptivity	active	active
plexiflex scheduling	inactive	active

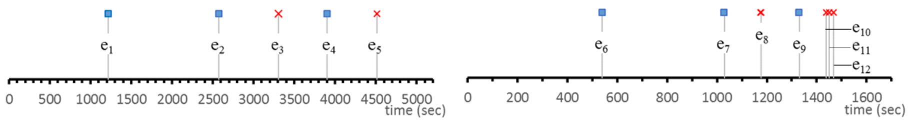
windows at any moment. The longer the history the higher is the inertia of the algorithm to react.

- *Reaction time lag*: time between an event in the network and reaction from *plexiflex* . This is expected to change together with the window size.

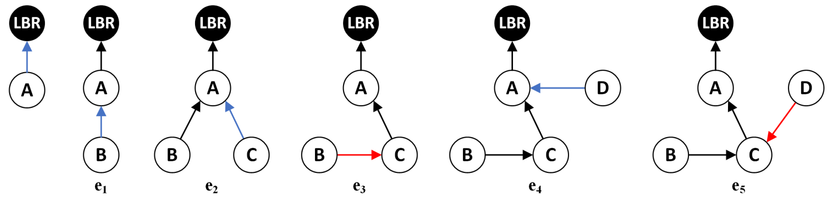
Experimental setup. All experiments were conducted in an actual home/office environment. Nodes were placed at a 3D topology between two floors at a distance of 5-7m apart from each other. The platform used for the experiments were the JN5168 NXP nodes. The size of the network was kept small (5 nodes at max 3 hops depth from gateway). Each node was configured to generate two parallel data streams with different packet rate: one to emulate background traffic with short periodicity and another with longer periodicity for *plexiflex* to subscribe to and use for interarrival time monitoring. Every node would join the network with the minimal configuration (one slotframe with one shared TxRx and Timekeeping cell). *plexiflex* was creating a new slotframe where any new cell wouldb e allocated. Tab. 3 provides the configuration of the two experiments conducted emulating low and high load conditions.

That is, each node would have to handle (a) roughly 0.57 packets per second (0.22 data + 0.1 EBs + 0.25 DIO) for the 1st experiment, and (b) roughly 0.95 packets per second (0.6 data + 0.1 EBs + 0.25 DIO) for the 2nd experiment. Given the two slotframes (minimal and *plexiflex*), a node initially has 2.44 shared slots per second to handle the whole traffic for the 1st experiment and 7.69 shared slots for the 2nd. Note that these slots are used for both transmission to and reception from any neighbor for any packet EB, DIO (DAO) or data. Nodes were joining the network one after the other as shown in Fig. 4a with blue square markers. Due to RPL, some nodes switched parents, event marked with red 'x' markers.

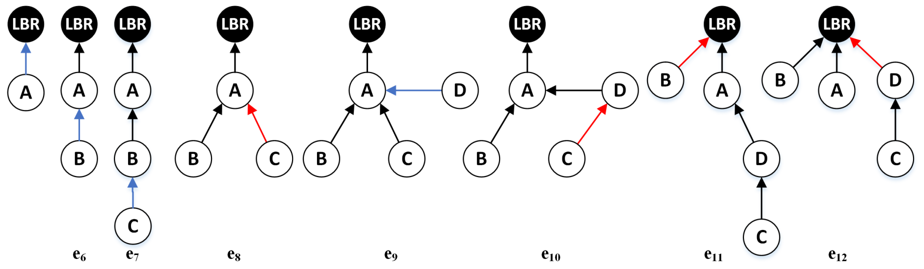
Figures 4b and 4c illustrate the evolution of the network for the two experiments upon every event. While node joining was intended by the design of the experimentation, rewiring was decided by RPL based on the zero objective function. For visualization convenience, the first connection (LBR $\leftarrow$ A) is marking the start of the data collection.



(a) Timeline and events of the two experiments on *plexi* and *plexiflex*. Blue markers for node joining and red for node parent switching events



(b) Topology evolution per event in 1st experiment



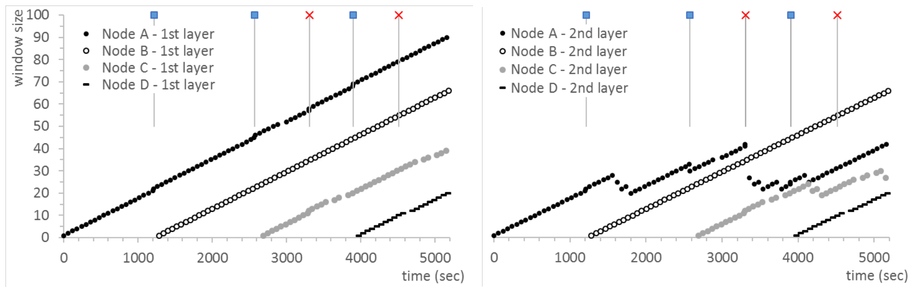
(c) Topology evolution per event in 2nd experiment

Figure 4: Topology evolution after node joining and parent switching

5.1. Results

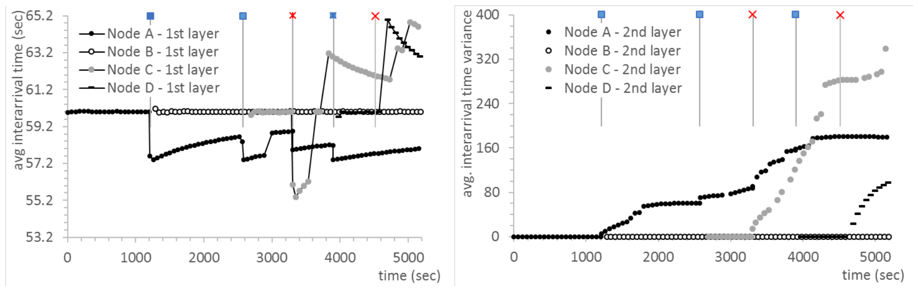
At the first experiment, *plexiflex* monitors the periodic data stream adapts the `ADWIN2` windows but does not modify the schedule. Fig. 5 illustrates the evolution of the average window sizes for every node. The 1st-layer window for every node increases with time almost linearly. Node joining or rewiring does not significantly affect the the lengt of the average interarrival time window. On the contrary, Fig. 5b confirms that the interarrival variance window decreases for nodes who have new children or siblings (either if newly joined or rewired children). *plexiflex* shrunk A's average variance window once B, C and D joined, B and D rewired to C. Similarly, the average variance window of C shrunk once D joined and rewired. However, leaf nodes B and D do not seem affected. This is because before and after rewiring, the bottleneck (node A) in their path towards LBR remains the same with the same traffic load. Since *plexiflex* measures the interarrival time and not the absolute packet delay and the packets from B and D go through the same bottleneck, there is not change in the interarrival time variance; hence, the 2nd-layer `ADWIN2` window grows. *plexiflex* is able to capture the joining and rewiring events in a network by monitoring a periodic data stream per node.

Fig. 6 illustrates the actual average interarrival time and variance and con-



(a) 1^{st}-layer `ADWIN2` window length (b) 2^{nd}-layer `ADWIN2` window length

Figure 5: Length of *plexiflex* windows per node without scheduling

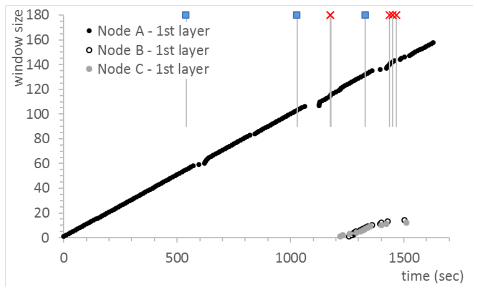


(a) Average interarrival time based on 1^{st}-layer `ADWIN2` window (b) Average interarrival time variance based on 2^{nd}-layer `ADWIN2` window

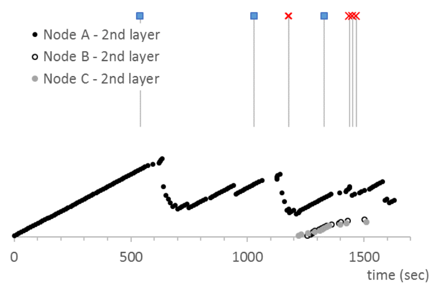
Figure 6: Average values of *plexiflex* windows per node without scheduling

firms the conclusions of Fig. 5. At every joining or rewiring event, the LBR $\leftarrow$ A connection is affected both at interarrival time between two packets and at the variance thereof. The changes at the variance are significant enough for the 2^{nd}-layer `ADWIN2` window to shrink. However, the change at the average interarrival time is not big enough to trigger *plexiflex* shrink the 1^{st}-layer `ADWIN2` window of the affected nodes. The component with the squared differences in the calculation of variance is responsible for generating sufficient triggers for the 2^{nd}-layer `ADWIN2` to adapt the window.

While at the first experiment the window size was not connected to specific scheduling actions, *plexiflex* was adding an extra upstream cell for every node's windows resizing. This was based on the evidence of the first experiment that changes in interarrival time variance reflects a new data stream sharing same resources with the affected stream. Fig. 7 illustrates the changes of the `ADWIN2` windows of the second experiment. A noticeable difference from the first experiment is the belated reception of data stream packets from nodes B,C and D. *plexiflex* missed the *plexi* notifications on `/rpl/dag` resource from those nodes. This is because either the `OBSERVE /rpl/dag` request from *plexiflex* was lost on the way to those newly joined nodes or the response of it. However, as

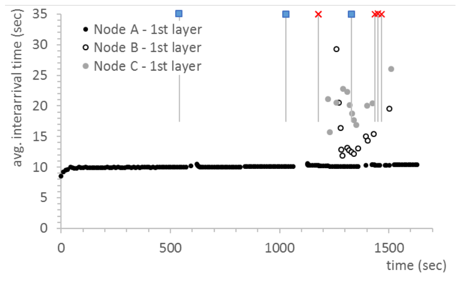


(a) 1st-layer `ADWIN2` window length

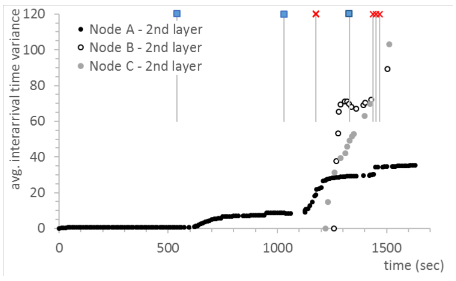


(b) 2nd-layer `ADWIN2` window length

Figure 7: Length of *plexiflex* windows per node; nodes rescheduled at every `ADWIN2` window resizing



(a) Average interarrival time based on 1st-layer `ADWIN2` window



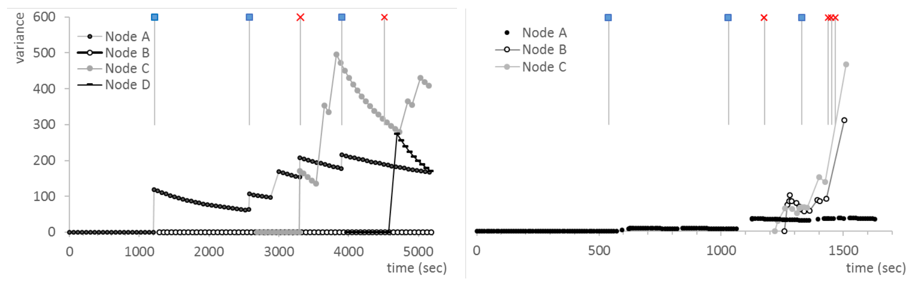
(b) Average interarrival time variance based on 2nd-layer `ADWIN2` window

Figure 8: Average values of *plexiflex* windows per node; nodes rescheduled at every `ADWIN2` window resizing

soon as node C rewired, *plexiflex* managed to restore the interaction with B and C as both of the nodes notified *plexiflex* on their `/rpl/dag` resource.

As opposed to the experiment without scheduling, *plexiflex* manages to keep the average interarrival time of packets from node A almost always constant at 10 seconds (see Fig. 8a). Even if node B and C were lost for some time, the stressed LBR $\leftarrow$ A connection is scheduled extra cells to handle the excessive traffic coming from new children or leaves in A's subtree. Moreover, Fig. 8b shows that the average interarrival time variance is also kept to less than 50% of the average variance at the first experiment. The results plotted for nodes B, C and D are not so clear as for node A in Fig. 8. The belated interaction between *plexiflex* and those nodes coincided with a number of other events i.e. rewiring and joining which involve a number of message exchange with *plexiflex* as described in section 3.3. Furthermore, these events triggered *plexiflex* to schedule extra resources massively to affected nodes. Both the simultaneous initialization phases and the wiring events generated fluctuations to the interarrival time; hence, variance.

Finally, Fig. 9 illustrates the effect of adaptive re-scheduling at the variance



(a) Nodes not scheduled (b) Nodes scheduled per window resizing

Figure 9: Variance of interarrival time based on 1^{st}-layer `ADWIN2` window

of the average interarrival time. Figures 6 and 8 show the average variance based on the 2^{nd}-layer `ADWIN2` window, whereas Fig. 9 plots the data points inserted into that window i.e. the variance of the data points in 1^{st}-layer `ADWIN2` window. *plexiflex* manages to keep the interarrival time variance very low rescheduling when and where necessary monitoring periodic data streams.

6. Conclusions and future work

This work presents *plexi* and *plexiflex*. *plexi* is an API built for monitoring and scheduling low power IEEE802.15.4 TSCH nodes. *plexi* exposes resources of the nodes in Contiki 3.x over CoAP. At the client side, an API accesses and controls those resources so that a cells and slotframes may be created/deleted in a node. This API allows different schedulers to be used and applications to request for specific QoS. *plexi* relies on standards IEEE802.15.4, TSCH, 6LowPan, IPv6, UDP and CoAP maximizing the interoperability between different hardware, implementations of those layers and schedulers.

The second contribution of the work was *plexiflex* , a re-scheduling control unit which decides when a given scheduler should issue or remove cells at run-time. *plexiflex* relies on monitoring periodic data received and measuring the interarrival time between two packets of that stream. Experiments have shown that the variance of interarrival time is a reliable metric to identify network event such as node joining or rewiring. However, that comes with the assumption that every node is generating a periodic data stream and all are monitored by *plexiflex* . The algorithm behind *plexiflex* are two adaptive windows per node, one maintaining the perceived interarrival time and the second the variance thereof. When the average variance becomes significantly different the window will shrink triggering a re-scheduling action.

Though experiments have shown clear evidence of the effectiveness of `ADWIN2` algorithm and the interarrival time, more experimentation on bigger networks and more randomized traffic is necessary. Moreover, work on improving *plexiflex* is also an important step as `ADWIN2` windows do not have a maximum size limit which may reduce adaptivity of *plexiflex* if the windows grow too big. Finally, *plexi* will be deployed online for public use and experimentation with a small set of schedulers so that engineers may use to test in their networks.

Acknowledgement

We would like to thank Simon Duquennoy (SICS), Hugh Maaskant (NXP) and Vlado Hantziski (TU Berlin) for their insightful guidance on various technical aspects. This work was supported by EIT ICT Labs RICH Activity (2014-2015).

References

[1] J. Song, S. Han, A. Mok, D. Chen, M. Lucas, M. Nixon, WirelessHART: Applying Wireless Technology in Real-Time Industrial Process Control, in: IEEE Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08, 2008, pp. 377–386. `doi:10.1109/RTAS.2008.15`.

[2] P. Thubert, An Architecture for IPv6 over the TSCH mode of IEEE 802.15.4, Tech. rep. (Nov. 2015).
URL `https://tools.ietf.org/html/draft-ietf-6tisch-architecture-08`

[3] P. Thubert, T. Watteyne, M. Palattella, X. Vilajosana, Q. Wang, IETF 6tsch: Combining IPv6 Connectivity with Industrial Performance, in: 2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013, pp. 541–546. `doi:10.1109/IMIS.2013.96`.

[4] J. Akerberg, M. Gidlund, M. Bjorkman, Future research challenges in wireless sensor and actuator networks targeting industrial automation, in: 2011 9th IEEE International Conference on Industrial Informatics (INDIN), 2011, pp. 410–415. `doi:10.1109/INDIN.2011.6034912`.

[5] V. Gungor, G. Hancke, Industrial Wireless Sensor Networks: Challenges, Design Principles, and Technical Approaches, IEEE Transactions on Industrial Electronics 56 (10) (2009) 4258–4265. `doi:10.1109/TIE.2009.2015754`.

[6] Z. Shelby, K. Hartke, C. Bormann, The Constrained Application Protocol (CoAP) (Jun. 2014).
URL `http://tools.ietf.org/html/rfc7252`

[7] S. Duquennoy, B. Al Nahas, O. Landsiedel, T. Watteyne, Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH, in: Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys '15, ACM, New York, NY, USA, 2015, pp. 337–350. `doi:10.1145/2809695.2809714`.
URL `http://doi.acm.org/10.1145/2809695.2809714`

[8] R. Soua, P. Minet, E. Livolant, Wave: a distributed scheduling algorithm for convergecast in IEEE 802.15.4e TSCH networks, Transactions on Emerging Telecommunications Technologies (2015) n/a–n/adoi:10.1002/ett.2991.
URL http://onlinelibrary.wiley.com/doi/10.1002/ett.2991/abstract

[9] T. Watteyne, A. Mehta, K. Pister, Reliability Through Frequency Diversity: Why Channel Hopping Makes Sense, in: Proceedings of the 6th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks, PE-WASUN '09, ACM, New York, NY, USA, 2009, pp. 116–123. doi:10.1145/1641876.1641898.
URL http://doi.acm.org/10.1145/1641876.1641898

[10] T. Watteyne, M. Palattella, L. Grieco, Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem Statement, Request for Comments (May 2015).
URL https://tools.ietf.org/html/rfc7554

[11] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, R. Alexander, RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, Request for Comments 6550, Internet Engineering Task Force (Mar. 2012).
URL https://tools.ietf.org/html/rfc6550

[12] A. Tinka, T. Watteyne, K. Pister, A Decentralized Scheduling Algorithm for Time Synchronized Channel Hopping, in: J. Zheng, D. Simplot-Ryl, V. C. M. Leung (Eds.), Ad Hoc Networks, no. 49 in Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer Berlin Heidelberg, 2010, pp. 201–216, dOI: 10.1007/978-3-642-17994-5_14.
URL http://link.springer.com/chapter/10.1007/978-3-642-17994-5_14

[13] N. Accettura, E. Vogli, M. Palattella, L. Grieco, G. Boggia, M. Dohler, Decentralized Traffic Aware Scheduling in 6tisch networks: design and experimental evaluation, IEEE Internet of Things Journal PP (99) (2015) 1–1. doi:10.1109/JIOT.2015.2476915.

[14] Y. Wang, M. C. Vuran, S. Goddard, Cross-layer Analysis of the End-to-end Delay Distribution in Wireless Sensor Networks, IEEE/ACM Trans. Netw. 20 (1) (2012) 305–318. doi:10.1109/TNET.2011.2159845.
URL http://dx.doi.org/10.1109/TNET.2011.2159845

[15] B. Ji, C. Joo, N. Shroff, Delay-Based Back-Pressure Scheduling in Multihop Wireless Networks, IEEE/ACM Transactions on Networking 21 (5) (2013) 1539–1552. doi:10.1109/TNET.2012.2227790.

[16] B. Li, A. Eryilmaz, R. Srikant, On the universality of age-based scheduling in wireless networks, in: 2015 IEEE Conference on Computer Communications (INFOCOM), 2015, pp. 1302–1310. doi:10.1109/INFOCOM.2015.7218506.

[17] A. Saifullah, Y. Xu, C. Lu, Y. Chen, Real-Time Scheduling for WirelessHART Networks, in: Real-Time Systems Symposium (RTSS), 2010 IEEE 31st, 2010, pp. 150–159. doi:10.1109/RTSS.2010.41.

[18] M. Palattella, N. Accettura, M. Dohler, L. Grieco, G. Boggia, Traffic Aware Scheduling Algorithm for reliable low-power multi-hop IEEE 802.15.4e networks, in: 2012 IEEE 23rd International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC), 2012, pp. 327–332. doi:10.1109/PIMRC.2012.6362805.

[19] O. Incel, A. Ghosh, B. Krishnamachari, K. Chintalapudi, Fast Data Collection in Tree-Based Wireless Sensor Networks, IEEE Transactions on Mobile Computing 11 (1) (2012) 86–99. doi:10.1109/TMC.2011.22.

[20] Y. Wu, J. Stankovic, T. He, S. Lin, Realistic and Efficient Multi-Channel Communications in Wireless Sensor Networks, in: IEEE INFOCOM 2008. The 27th Conference on Computer Communications, 2008. doi:10.1109/INFOCOM.2008.175.

[21] R. Soua, P. Minet, E. Livolant, MODESA: An optimized multichannel slot assignment for raw data convergecast in wireless sensor networks, in: Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International, 2012, pp. 91–100. doi:10.1109/PCCC.2012.6407742.

[22] Q. Wang, X. Vilajosana, 6tisch Operation Sublayer (6top) Interface (Jul. 2015).
URL https://tools.ietf.org/html/draft-ietf-6tisch-6top-interface-04

[23] P. Zand, R. Sudhaakar, 6tisch Resource Management and Interaction using CoAP (Mar. 2015).
URL https://tools.ietf.org/html/draft-ietf-6tisch-coap-03

[24] o. source, Contiki: The Open Source Operating System for the Internet of Things (2015).
URL http://www.contiki-os.org/

[25] M. Kovatsch, Erbium (Er) REST Engine - C CoAP Implementation (2012).
URL http://people.inf.ethz.ch/mkovatsc/erbium.php

[26] A. Bifet, R. Gavalda, Learning from Time-Changing Data with Adaptive Windowing., in: SIAM International Conference on Data Mining, Proceedings of, Vol. 7, SIAM, California, USA, 2007, p. 2007.
URL http://epubs.siam.org/doi/pdf/10.1137/1.9781611972771.42