The Cost of Installing a 6TiSCH Schedule

Erwan Livolant, Pascale Minet, Thomas Watteyne

The Cost of Installing a 6TiSCH Schedule

Erwan Livolant, Pascale Minet, and Thomas Watteyne

Inria-Paris, EVA team, France
firstname.lastname@inria.fr

Abstract. Scheduling in an IEEE802.15.4e TSCH (6TiSCH) low-power wireless mesh network can be done in a centralized or distributed way. When using centralized scheduling, a scheduler computes a communication schedule, which then needs to be installed into the network. This can be done using standards such CoAP and CoMI, or using a custom protocol such as OCARI. In this paper, we compute the number of messages installing and updating the schedule takes, using both approaches, on a realistic example scenario. The cost of using today's standards is high. In some cases, a standards-based solution requires approximately 4 times more messages to be transmitted in the network, than when using a custom protocol. This paper makes three simple recommended changes to the standards which, when integrated, reduce the cost of a standards-based solution by 18% to 74%. Since they are still being developed, these recommendations can easily be integrated into the standards.

Keywords: Low-Power Wireless Mesh Networks, IEEE802.15.4e TSCH, 6TiSCH, CoAP, CoMI, OCARI.

1 Introduction

Industrial low-power wireless mesh network applications have strong requirements in terms of latency, energy efficiency and reliability. To cope with these requirements, the IEEE802.15.4e amendment [7] introduces the Time Slotted Channel Hopping (TSCH) mode. In a TSCH network, nodes are synchronized, and time is cut into timeslots, each typically 10 ms long. All communication is orchestrated by a communication schedule, which indicates to each node what to do in each slot: transmit, listen or sleep. This schedule can be built to enable collision-free communication, yielding predictable behavior, ultra-high reliability and years of battery lifetime.

A schedule consists of a number of timeslots which continuously repeat over time. An example schedule is depicted in Fig. 2 with 9 timeslots and 3 logical channels. The index of a timeslot (the x-axis of the matrix in Fig. 2) is called `slotOffset`. The `channelOffset` represents the communication channel (the y-axis of the matrix in Fig. 2).

Building the schedule consists in assigning a source node, a destination node and a channel to cells in the schedule. Installing the schedule into the network means indicating to each node the list of cells it is involved in, either

as transmitter or as receiver. Each cell is represented by a tuple [`slotOffset`, `channelOffset`, `nodeAddress`, `linkType`] (`linkType` indicates whether it is a transmit – TX – and/or receive – RX – cell).

We want to compare the number of packets it takes a central scheduler to install and update a schedule, using the different approaches listed in Section 2. We are particularly interested in comparing standards-based and custom-built protocols. The goal is *not* to explore edge cases, but rather to take an example representative enough that we can learn lessons and made recommendations.

We consider the topology depicted in Fig. 1. The network first consists of 12 nodes where Node 1 is the root of the network. Arrows represent the links that are used for communication and that therefore need to appear in the schedule. The goal is to *install* the schedule from Fig. 2 into the network. When node 13 is added, the scheduler computes the schedule depicted in Fig. 3, in which 12 cells differ. The goal is then to *update* the schedule in the network.

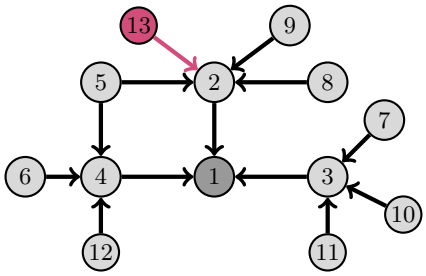

Fig. 1. Logical network topology.

chan. \ slot	0	1	2	3	4	5	6	7	8
0	4→1	2→1	4→1	2→1	4→1	2→1	4→1	2→1	3→1
1	3→1	10→3	3→1	11→3	3→1	10→3	3→1	7→3	
2	5→2	6→4	2→1	12→4	9→2	5→4	8→2		

Fig. 2. Schedule computed by MODESA for 12 nodes.

chan. \ slot	0	1	2	3	4	5	6	7	8
0	4→1	2→1	4→1	2→1	4→1	2→1	4→1	2→1	4→1
1	3→1	10→3	3→1	11→3	3→1	12→4	3→1	5→4	3→1
2	2→1	5→4	9→2	6→4	8→2	10→3	13→2	7→3	

Fig. 3. Updated schedule after node 13 is added. The black cells are the ones that differ from the schedule for 12 nodes.

Several centralized scheduling algorithms exist. This paper does *not* recommend one or the other, nor does it attempt to survey them. Rather, we use a particular scheduling algorithm, MODESA [10], and measure the number of packets to install the schedule *once it is computed*. The choice of MODESA is, as far as this paper is concerned, arbitrary.

We focus on two approaches to install the schedule: using standards (detailed in Section 2.2), and using a custom protocol (detailed in Section 2.3). By executing both approaches on a representative example scenario, we show that using a standards-based approach can cost 4 times more frames than using a custom-built protocol. In Section 3, we make three simple recommended changes to the standards which reduces the overhead of the standards-based approach by 18% to 74% percent.

2 Approaches to Install the Schedule

We consider different approaches to install a schedule: either using the CoAP and CoMI standards (Section 2.2), or using a custom protocol called OCARI (Section 2.3). Both use IEEE802.15.4 [6] as the underlying physical layer. Section 2.1 first details the notation used.

2.1 Notation

We define the following notations:

- $Depth(u)$, the depth of node u in the topology.
- $Child(u)$, the set of children of node u.
- $Trans(u)$, the number of user data frames transmitted by node u, including both the ones generated by node u itself and the ones received from its children and forwarded.
- P, the number of nodes which are not leaf nodes.
- $B(u)$, number of CoAP blocks sent to node u.
- B_{sched}, number of CoAP blocks needed to broadcast the complete schedule.
- Let N_{field} number of fields updated per cell.
- $ScheduleNumber$, version of the schedule, incremented each time the schedule is computed.

2.2 CoAP and CoMI: a Standards-Based Approach

6TiSCH is an active IETF working group which standardizes how to build and maintain a TSCH communication schedule [4]. While 6TiSCH supports both centralized and distributed scheduling, this paper focuses on the former. The protocol stack considered by 6TiSCH consists of IEEE802.15.4e [7] (*physical and link layers*), 6LoWPAN [5], RPL [14] (*network layer*), CoAP [3, 9, 12] and CoMI [11] (*application layer*). The resulting protocol stack is depicted in Fig. 4.

Table 1 depicts the format of an IEEE802.15.4 frame which encapsulates those protocols. The Maximum Transmission Unit (MTU) at the IEEE802.15.4 [6] PHY layer is 127 bytes. MAC header and footer require a total of 29 bytes in the context of the IEEE802.15.4e TSCH [7] mode[1]. The Auxiliary Security header is encoded with 2 bytes and the Message Integrity Code (MIC) is coded with

[1] [13] indicates that data messages must provide in their MAC header 64-bit addresses for the destination and the source.

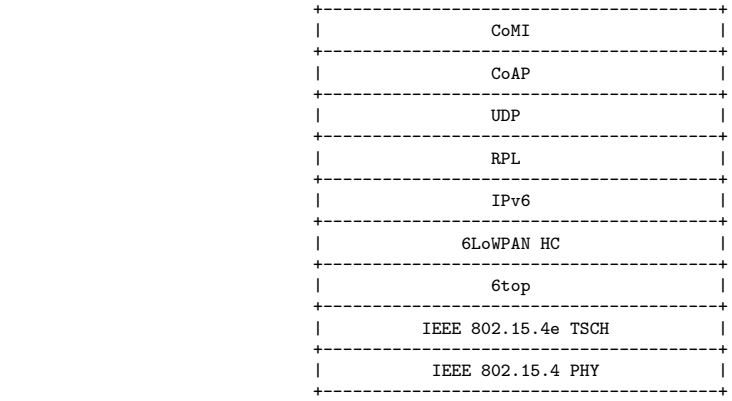

Fig. 4. The 6TiSCH Protocol Stack.

Table 1. Packet format when using the standards-based approach. Numbers indicate the number of bytes in the fields.

	Frame Control	Seq Num	Dest PAN	Dest Address	Src Address	Aux Sec Header	MIC	**Payload**	FCS
IEEE802.15.4e TSCH	2	1	2	8	8	2	4	**98**	2

	LOWPAN_IPHC	Hop limit	Src Address	Dest Address	**Payload**
6loWPAN - IP	2	1	8	8	**79**

	LOWPAN_NHC	Port src + dest	**Payload**
6loWPAN - UDP	1	1	**77**

	Header	URI	Payload marker	**Payload without block**
COAP without frag	4	11	1	**61**

	Header	URI	Option delta + Length	Option Delta extended	Option value	Payload marker	**Payload with block**
COAP with $\leq$ 16 frag	4	11	1	1	1	1	**32**
COAP with $\leq$ 4096 frag	4	11	1	1	2	1	**32**

4 bytes. The MAC payload can contain up to 98 bytes. At the Network layer, the protocol requires 19 bytes for the IP compressed header (2 bytes), the Hop limit (1 byte) and the Source and Destination addresses coded on 8 bytes each. At the Transport layer, the payload is reduced by 2 bytes corresponding to the UDP compressed header and the destination and source port coded together in 1 byte. At the Application layer, the size of the applicative payload depends on the CoAP options used. 4 bytes of header and 1 byte of payload marker are required. The URI targeting the resource is defined as follows: `/mg/6t/hash` where `/mg/6t/` is the main path to the 6top management resources and `hash` is a 30-bit hash (encoded on 4 bytes) defining the rest of the path towards the target resource. We look at three options for using CoAP: without fragmentation, less than 16 fragments and with than 4096 fragments. In any case, we get a useful

payload whose size is less than 61 bytes. For the *Block* option, it is specified in CoAP that the block size should be a power of two. As a consequence, the only possibility with this payload size is a block size of 32 bytes.

As a conclusion, in the 127-byte IEEE802.15.4 frame, only 32 bytes are available for the actual encoding of the schedule, as the remainder of the frame is occupied by the headers of the different standards.

The standards do not indicate the example approach to install the schedule. We consider three approaches, which we call "Single", "`PATCH`" and "Broadcast". Each is detailed below.

Updating a Single Field at a Time ("Single"). The central scheduler issues a separate confirmable CoAP `POST` message to write each field of each cell it installs into a node. This solution assumes that for any network node a route to reach it from the scheduler node is known by all nodes within the route. Fig. 5 shows an example CoAP `POST` to set nodeAddress = 3 for the cell at `slotOffset` = 3 and `channelOffset` = 1 in the schedule.

```
REQ:
     POST
     url:  coap://<ip>:5683/mg/6t/cellList/nodeAddress/?slotOffset=3&channelOffset=1
     body: 3
```

Fig. 5. A CoAP `POST` addressing the nodeAddress value of a cell.

The central scheduler needs to install a cell on both communicating neighbors. For each field, the scheduler issues a CoAP `CON` and receives CoAP `ACK`. Since each cell contains N_{field} fields, the scheduler sends N_{field} CoAP `CON` messages and receives N_{field} CoAP `ACK` acknowledgments. When sent to node u, each of these messages travels over $Depth(u)$ hops. Moreover, node u is involved in $Trans(u)$ transmissions and $\sum_{v \in Child(u)} Trans(v)$ receptions. This results in the total number of messages in (1).

$$numFrames_{Single} = 2 \cdot Nfield \cdot \sum_{u \neq sink} Depth(u) \left(Trans(u) + \sum_{v \in Child(u)} Trans(v) \right) \quad (1)$$

Table 2 presents the number of messages required for first installing the schedule (see Fig. 2) computed for a network of 12 nodes, then updating this schedule taking into account the new node 13 (see Fig. 3). When the schedule is updated, node 2 has two cells where two fields are modified and two other cells with only one field changed. Hence in Table 2, the element at line 2 and column "Cells * $Nfield$" contains 2*2 + 2*1.

Table 2. Total number of messages required for installing and updating the schedule with the Single method.

	Install			Update	
Node	Cells	$Nfield$	Messages	Cells * $Nfield$	Messages
2	8	4	64	2*2 + 2*1	8 + 4
3	9	4	72	3*1	6
4	7	4	56	3*1 + 2*4	6 + 16
5	2	4	32	2*2	16
6	1	4	16	1*1	4
7	1	4	16	1*1	4
8	1	4	16	1*1	4
9	1	4	16	1*1	4
10	2	4	32	1*1	4
11	1	4	16	0	0
12	1	4	16	1*2	8
13	-	-	-	1*4	16
Total	-	-	**352**	-	**100**

Sending a `PATCH` to Each Node ("`PATCH`"). The scheduler contacts each node once and transfers a list of cells that must be updated, encoded as a CoAP `PATCH`. This allows it to send only the differences between the current schedule and the new one. An example CoAP `PATCH` which modifies the `nodeAddress` and `linkType` fields of the cell with `slotOffset` = 1 and `channelOffset` = 2 is presented in Fig. 6.

```
REQ:
     PATCH
     url:  coap://<ip>:5683/mg/6t/cellList
     body: [
             {
               "op":    "replace",
               "path":  "/nodeAddress?slotOffset=1&channelOffset=2",
               "value": 4
             },
             {
               "op":    "replace",
               "path":  "/linkType?slotOffset=1&channelOffset=2",
               "value": 1
             }
           ]
```

Fig. 6. A CoAP `PATCH` modifying the `nodeAddress` and `linkType` fields of the cell.

When the payload of the `PATCH` is too long for a single frame, CoAP Block is used for application-layer fragmenting. $B(u)$ blocks are transmitted to node u, the scheduler receives an acknowledgment per block, as recommended in [11]. The resulting total number of messages is given in (2).

$$numFrames_{PATCH} = 2 \cdot \sum_{u \neq sink} B(u) \cdot Depth(u) \tag{2}$$

Table 3 presents the number of messages required for installing and updating the schedule computed for our illustrative network (see Fig. 1). Detailed CoAP `PATCH` messages for installing the schedule can be found in [8].

Table 3. Total number of messages required for installing and updating the schedules with the `PATCH` method.

	Install			Update		
Node	CBOR bytes	Blocks	Messages	CBOR bytes	Blocks	Messages
2	1049	33	66	397	13	26
3	918	29	58	208	7	14
4	918	29	58	533	17	34
5	263	9	36	269	9	36
6	132	5	20	67	3	12
7	132	5	20	70	3	12
8	132	5	20	67	3	12
9	132	5	20	67	3	12
10	263	9	36	70	3	12
11	132	5	20	0	0	0
12	132	5	20	136	5	20
13	-	-	-	132	5	20
Total	-	-	**374**	-	-	**210**

Broadcasting the Complete Schedule ("Broadcast"). The scheduler broadcasts the complete schedule over CoAP. The schedule is represented as a CBOR-encoded [2] JSON document. Each node u filters the cells it is involved in as transmitter or receiver. An example of broadcast with the complete schedule is proposed in Fig. 7 where each tuple is [`slotOffset`, `channelOffset`, source node, destination node].

To ensure that each node correctly receives the schedule, the scheduler uses CoAP Observe to monitor the value of the schedule number on each node. Each time a new schedule is pushed to a node, the scheduler expects to receive a CoAP Observe notification, confirming the successful reception of the schedule update by the node.

At network initialization, the scheduler issues an CoAP Observe request on each of the nodes. Broadcasting the schedule requires B_{sched} blocks of 32 bytes to be broadcast into the network, resulting in $P \cdot B_{sched}$ frames, assuming ideal flooding. Each time a schedule is installed, the waves of CoAP Observe notifications account for $\sum_{u \neq sink} Depth(u)$ messages. The total number of frames to install a schedule is given in (3); δ_P is equal to 1 when at least one new node has joined the network since the last transmission CoAP Observe notification, 0 otherwise.

$$numFrames_{Broadcast} = P \cdot Bsched + \sum_{u \neq sink} Depth(u) + \delta_P \cdot P \tag{3}$$

The length of the CBOR transcription of the JSON document describing our schedule (see Fig. 7) is 149 bytes. This CBOR transcription is divided into 5 fragments of 32 bytes, hence $Bsched = 5$. In our example topology, the number of parents broadcasting the schedule is $P = 4$ (node 1, 2, 3, 4) and the sum of depths of nodes is $\sum_{u \neq sink} Depth(u) = 19$. For this first schedule, as all nodes just joined the network $\delta_P = 1$. Finally, $msg(Broadcast) = 4 \times 5 + 19 + 1 \times 4 = 43$.

```
REQ:
     POST
     url:  coap://<ip>:5683/mg/6t/schedule
     body: {
             "ScheduleNumber": "1",
             "Schedule":
             [
               [0, 0,  4, 1],
               [0, 1,  3, 1],
               [0, 2,  5, 2],
               [1, 0,  2, 1],
               [1, 1, 10, 3],
               [1, 2,  6, 4],
               [2, 0,  4, 1],
               [2, 1,  3, 1],
               [2, 2,  2, 1],
               [3, 0,  2, 1],
               [3, 1, 11, 3],
               [3, 2, 12, 4],
               [4, 0,  4, 1],
               [4, 1,  3, 1],
               [4, 2,  9, 2],
               [5, 0,  2, 1],
               [5, 1, 10, 3],
               [5, 2,  5, 4],
               [6, 0,  4, 1],
               [6, 1,  3, 1],
               [6, 2,  8, 2],
               [7, 0,  2, 1],
               [7, 1,  7, 3],
               [8, 0,  3, 1]
             ]
           }
```

Fig. 7. A CoaP `POST` broadcasting the complete schedule.

For the second schedule depicted in Fig. 3, the length of the CBOR transcription of the JSON document describing our schedule is 159 bytes. The detailed CoAP `POST` for the broadcast of this second schedule can be found in [8]. As in the previous case, this CBOR transcription is divided into 5 fragments of 32 bytes, hence $Bsched = 5$. The number of parents broadcasting the schedule is the same $P = 4$ but now the sum of depths of nodes is $\sum_{u \neq sink} Depth(u) = 21$. Node 13 just joined the network, hence $\delta_P = 1$. Finally, $msg(Broadcast) = 4 \times 5 + 21 + 1 \times 4 = 45$.

2.3 OCARI, a Custom Protocol

An alternative is to use OCARI [1], a custom-built non-standards-based protocol. We compare a standards-based approach to it to provide a lower bound on the number of packets needed to install a schedule.

OCARI is compliant with IEEE802.15.4, but not with upper-layer protocols such as 6LoWPAN or CoAP. Each OCARI node is assigned a 2-byte identifier, unique in the network and given during its association.

OCARI schedules activities in the network in a cycle organized in four periods, as depicted in Fig. 8. First, OCARI synchronizes the network in a collision-free multi-hop way during the [T0,T1] period. All nodes periodically send beacons to maintain this synchronization. The period [T1,T2] is dedicated to control

traffic used to collect network characteristics in order to compute a schedule for user data. Period [T2,T3] allows user data gathering. Finally, period [T3,T0'] is a sleep period, all nodes sleep to save energy.

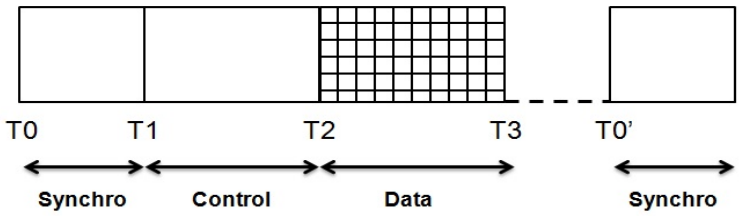

Fig. 8. Cycle provided by OCARI

A new schedule computation is kicked off when the topology or the application-needs change. When a new schedule is ready, it is broadcast into the network by piggy-backing the entire schedule into the beacons sent by all parent nodes. The schedule is a sequence of 7-byte cell tuples [`slotOffset`, `channelOffset`, SourceNode, DestinationNode], each describing a single cell in the schedule.

Octets: 2	1	2	2	4	variable	2
Frame Control	Seq. Num. Frame	Source PAN Id.	Source Address	Reserved *for IEEE802.15.4 compliance*	**Beacon Payload**	FCS
MHR					Payload	MFR

Octets: 1	1	1	4	2	variable	2	2	variable
Packet Type	Num. of Beaconing Nodes	Seq. Num. Beacon	Beaconing Interval	Contention Slot Duration	Addresses Beaconing Nodes	Num. of Cells	First Cell Index	Cell Tuples
Beacon Payload								

Fig. 9. Packet format when using the OCARI custom protocol.

Fig. 9 summarizes the format of an OCARI beacon. For the topology depicted in Fig. 1, the first schedule (see Fig. 2) to install is 24 tuples long, the second one (see Fig. 3) is 26 tuples long. The scheduler hence needs to transmit $24 \times 7 = 168$ bytes for the first schedule, $26 \times 7 = 182$ bytes for the second. Since in this typical case the maximum payload available for cell tuples in a beacon is 80 bytes, up to 11 cell tuples can be transported in a single beacon. If the schedule contains more than 11 cells, the full schedule is fragmented across different beacons. The entire schedule hence need 3 beacons to be transferred, or a total of $3 \times 4 = 12$ messages, since in our example $P = 4$ parent nodes.

Details about OCARI can be found in [1].

3 Recommended Optimizations

Table 4 summarizes the total number of messages to install and update the schedule. The detailed computation of the number of messages can be found in [8]. It shows that a standard-based approach is less efficient than a custom-built protocol by a factor of 4 or more.

Simple changes to the standards allow them to be much more efficient. This section lists three simple optimizations which, when applied, yield a reduction in number of messages between 18% and 74% as depicted in Table 4.

Table 4. Total number of messages traveling in the network for installing and updating the schedule, with none, some or all optimizations.

Approach	Optimizations	Install Schedule	Relative Gain	Absolute Gain	Update Schedule	Relative Gain	Absolute Gain
Single	*None*	352	N.A.	N.A.	100	N.A.	N.A.
PATCH	*None*	374	N.A.	N.A.	206	N.A.	N.A.
	+ Short addresses (3.1)	206	45%	45%	104	50%	50%
	+ `CellId` (3.2)	136	34%	64%	72	31%	65%
	+ `PATCH` syntax (3.3)	98	28%	74%	58	19%	72%
Broadcast	*None*	43	N.A.	N.A.	45	N.A.	N.A.
	+ Short addresses (3.1)	35	19%	19%	37	18%	18%
	+ `CellId` (3.2)	35	0%	19%	37	0%	18%
Custom	*None*	12	N.A.	N.A.	12	N.A.	N.A.

3.1 Use Short MAC Addresses

In today's 6TiSCH standards, only 64-bit long MAC addresses are used. Implementing an association mechanism would enable a coordinator (typically the root of the network) to assign a 16-bit address unique in the network to each device. This association mechanism would save 12 bytes in each frame. The payload at the MAC layer can then be extended to 110 bytes and the maximum applicative payload could be equal to 73 bytes, allowing a 64-byte CoAP Block size.

Table 5 presents the number of messages required for installing and updating the schedule computed for our illustrative network (Fig. 1). We see a dramatic saving for this `PATCH` method compared to the previous results in Section 2.2.

With the method of broadcasting a schedule to all nodes in the network, the savings in term of the number of messages is also significant. In our example topology, we obtain $msg(Broadcast) = 4 \times Bsched + 19 + 1 \times 4 = 35$ messages with $Bsched = 3$ for the installing the schedule. For updating the schedule, we obtain $msg(Broadcast) = 4 \times Bsched + 21 + 1 \times 4 = 37$ messages with $Bsched = 3$.

Per Table 4, using short MAC addresses reduces the number of frames by 45% (respectively 50%) to Install (respectively Update) the schedule when using the `PATCH` approach, and 19% (respectively 18%) when using the Broadcast approach.

Table 5. Total number of messages required for installing and updating the schedules with the PATCH method.

	without node 13			with node 13		
Node	CBOR bytes	Blocks	Messages	CBOR bytes	Blocks	Messages
2	1049	17	34	397	7	14
3	918	15	30	208	4	8
4	918	15	30	533	9	18
5	263	5	20	269	5	20
6	132	3	12	67	1	4
7	132	3	12	70	1	4
8	132	3	12	67	1	4
9	132	3	12	67	1	4
10	263	5	20	70	1	4
11	132	3	12	0	0	0
12	132	3	12	136	3	12
13	-	-	-	132	3	12
Total	-	-	**206**	-	-	**104**

3.2 More efficient `CellId` Representation

According to the IEEE802.15.4e standard [7], `slotOffset` and `channelOffset` are each encoded on 2 bytes. Since there are only 16 channels available for a IEEE802.15.4 radio operating at 2.4GHz, only 4 bits are needed to encode the `channelOffset`. Moreover, since the slotframe length in an IEEE802.15.4e TSCH network is rarely longer than 1000 timeslots (10 s when using 10 ms timeslots), `channelOffset` can be encoded using 12 bits. a single 16-bit number called "`CellId`" can encode `slotOffset` and `channelOffset`, in which the 4 most significant bits represent the `channelOffset` and the remaining 12 bits the `slotOffset`.

Table 6 presents the number of messages required for installing and updating the schedule computed for our illustrative network (see Fig. 1). We notice that this simple coding mechanism brings a greater gain compared to the previous results in Section 2.2.

Table 6. Total number of messages required for installing and updating the schedules with the `PATCH` method using a more efficient `CellId` Representation.

	without node 13			with node 13		
Node	CBOR bytes	Blocks	Messages	CBOR bytes	Blocks	Messages
2	729	12	24	275	5	10
3	636	10	20	132	3	6
4	636	10	20	371	6	12
5	181	3	12	132	3	12
6	92	2	8	44	1	4
7	92	2	8	45	1	4
8	92	2	8	44	1	4
9	92	2	8	44	1	4
10	183	3	12	44	1	4
11	92	2	8	0	0	0
12	92	2	8	44	1	4
13	-	-	-	92	2	8
Total	-	-	**136**	-	-	**72**

Fig. 10 depicts the schedule description using the `CellId` coding in the context of the broadcasting method.

```
REQ:
     POST
     url:  coap://<ip>:5683/mg/6t/schedule
     body: {
             "ScheduleNumber": "1",
             "Schedule":
             [
               [0, 4, 1],
               [1, 3, 1],
               [2, 5, 2],
               [16, 2, 1],
               [17, 10, 3],
               [18, 6, 4],
               [32, 4, 1],
               [33, 3, 1],
               [34, 2, 1],
               [48, 2, 1],
               [49, 11, 3],
               [50, 12, 4],
               [64, 4, 1],
               [65, 3, 1],
               [66, 9, 2],
               [80, 2, 1],
               [81, 10, 3],
               [82, 5, 4],
               [96, 4, 1],
               [97, 3, 1],
               [98, 8, 2],
               [112, 2, 1],
               [113, 7, 3],
               [128, 3, 1]
             ]
           }
```

Fig. 10. A CoAP `POST` broadcasting the first schedule using `CellId`.

Per Table 4, using a `CellId` rather than a [`slotOffset`, `channelOffset`] tuple saves an additional 34% (respectively 31%) when installing (respectively updating) the schedule using the `PATCH` approach. Its impact is negligible when using the Broadcast approach.

3.3 Shorter `PATCH` Operators

CoAP `PATCH` [12], even if if were designed for constrained devices, still uses the multi-character operators "op", "path", "value", "replace". Reducing those to the shorter operators "o", "p", "v", "rpl" would be syntactically equivalent, but more efficient. An example resulting CoAP `PATCH` is described in Fig. 11.

Table 7 presents the number of messages required for installing and updating the schedule computed for our illustrative network (Fig. 1) with the recommended shorter operators.

The CBOR representation of the JSON document above is 26% shorter than the CBOR representation of the same JSON document with longer operator string. According to Table 4, using shorter `PATCH` operators saves an additional

```
REQ:
     PATCH
     url:  coap://<ip>:5683/mg/6t/cellList
     body: [
             {
               "o": "rpl",
               "p": "/nodeAddress?CellId=18",
               "v": 4
             },
             {
               "o": "rpl",
               "p": "/linkType?CellId=18",
               "v": 1
             }
           ]
```

Fig. 11. A CoAP `PATCH` with shorter operators.

Table 7. Total number of messages required for installing and updating the schedules with the `PATCH` method with shorter operators.

	without node 13			with node 13		
Node	CBOR bytes	Blocks	Messages	CBOR bytes	Blocks	Messages
2	537	9	18	203	4	8
3	468	8	16	96	2	4
4	468	8	16	275	5	10
5	133	3	12	96	2	8
6	68	1	4	32	1	4
7	70	1	4	33	1	4
8	68	1	4	32	1	4
9	68	1	4	32	1	4
10	135	3	12	32	1	4
11	68	1	4	0	0	0
12	68	1	4	32	1	4
13	-	-	-	68	1	4
Total	-	-	**98**	-	-	**58**

28% (respectively 19%) when installing (respectively updating) the schedule using the `PATCH` approach. Its impact is negligible when using the Broadcast approach.

4 Conclusion

This paper discusses the efficiency of installing and updating a communication schedule in a 6TiSCH network with a central scheduler. Not surprisingly, it shows that a set of (generic) standards-based protocols is less efficient than a custom-built protocol. It shows how simple optimizations to the standards can reduce the number of frames by up to 74%.

It is, to the best of our knowledge, the first work to provide a system-wide analysis of the different protocols involved in a 6TiSCH network, and highlight the inefficiencies when "putting them all together". We believe this paper makes a strong contribution to the standardization activities at the IETF, in particular in the 6TiSCH, RPL and 6lo working group. It makes a strong (and quantified) case for adding an association step to the 6TiSCH protocol suite (which requires an additional protocol but reduces the overhead by up to 50%), and reducing

the length to the CoAP `PATCH` operators (a trivial change which saves up to 28% overhead).

Our future work will focus on distributed scheduling in 6TiSCH networks. This paper is an example of the collaboration between the IETF and the academic world which the newly created Thing-to-Thing Research Group is fostering.

References

[1] Al Agha, K., Chalhoub, G., Guitton, A., Livolant, E., Mahfoudh, S., Minet, P., Misson, M., Rahmé, J., Val, T., van den Bossche, A.: Cross-layering in an Industrial Wireless Sensor Network: Case Study of OCARI. Journal of Networks 4(6), 411–420 (2009)

[2] Bormann, C., Hoffman, P.: Concise Binary Object Representation (CBOR). IETF RFC7049 (October 2013)

[3] Bormann, C., Shelby, Z.: Block-wise Transfers in CoAP. draft-ietf-core-block-18 [work-in-progress] (September 2015)

[4] Dujovne, D., Watteyne, T., Vilajosana, X., Thubert, P.: 6TiSCH: Deterministic IP-enabled Industrial Internet (of Things). IEEE Communications Magazine 52(12) (December 2014)

[5] Hui, J., Thubert, P.: Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. IETF RFC6282 (September 2011)

[6] IEEE: IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006) (September 2011)

[7] IEEE: IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC Sublayer. IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011) (April 2012)

[8] Livolant, E., Minet, P., Watteyne, T.: The Cost of Installing a New Communication Schedule in a 6TiSCH Low-Power Wireless Network using CoAP. Tech. Rep. RR-8817, Inria (10 December 2015)

[9] Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). IETF RFC7252 (June 2014)

[10] Soua, R., Minet, P., Livolant, E.: MODESA: An Optimized Multichannel Slot Assignment for Raw Data Convergecast in Wireless Sensor Networks. In: International Performance Computing and Communications Conference (IPCCC). pp. 91–100. IEEE, Austin, TX, USA (2012)

[11] Van der Stok, P., Andy, B., Schönwälder, J., Sehgal, A.: CoAP Management Interface. draft-vanderstok-core-comi-07 [work-in-progress] (July 2015)

[12] Van der Stok, P., Sehgal, A.: Patch Method for Constrained Application Protocol (CoAP). draft-vanderstok-core-patch-02 [work-in-progress] (October 2015)

[13] Vilajosana, X., Pister, K.: Minimal 6tisch configuration. draft-ietf-6tisch-minimal-12 (September 2015)

[14] Winter, T., Thubert, P., Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Vasseur, J., Alexader, R.: RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. IETF RFC6550 (March 2012)