ECRTS Industrial Challenge: Designing a Resilient Time-Aware Shaper Configuration for TSN

Marc Boyer
DTIS, ONERA Toulouse, France

Rafik Henia
Thales/cortAIx Labs, Palaiseau, France

Cédric Pralet
DTIS, ONERA Toulouse, France

Abstract

In this paper, we address the ECRTS industrial challenge of reconfiguring at runtime a TSN (Time-Sensitive Networking) network connecting avionics systems, specifically in response to network failures. For this purpose, we propose WPEx-TAS, a novel schedule class that enables resilient and incremental TAS (Time-Aware Scheduling) reconfiguration by allowing critical data flows to be rerouted to already existing alternative time windows in case of failure, without disrupting the pre-computed TAS schedule. By preserving the initial certified schedule, our method reduces the cost and time required for recertification, offering an efficient solution for dynamic network reconfiguration. Furthermore, our approach optimizes bandwidth usage, ensuring that unused bandwidth in TAS windows can be allocated to non-critical traffic. Conducted experiments on the industrial challenge use-case show that a WPEx-TAS schedule can be built in an admissible time, and that the incremental insertion of new streams can be done is less than one second.

2012 ACM Subject Classification Networks → Formal specifications; Networks → Packet-switching networks; Networks → Cyber-physical networks; Networks → Traffic engineering algorithms

Keywords and phrases TSN, Time Sensitive Networking, TAS, Time-Aware Shaper, Incremental reconfiguration

Funding This work has been partially funded by the french innovation for defense agency (AID, Agence de l'innovation de défense), in the project "Resilient TSN", ANR-22-ASTR-0017-01.

1 Introduction

While current cyber-physical systems handle more and more sensors, generating more and more data transmitted to various equipments, the need for new real-time networks arises. The future deployment of Time-Sensitive Networking (TSN) in real-time systems is expected to offer significant benefits, particularly in applications that require strict timing and reliability. One of the key protocols within TSN, the Time-Aware Scheduling (TAS) protocol [3], allows for the efficient allocation of bandwidth and time slots for communication, ensuring that time-critical data is transmitted without interference, thus offering strong guarantees on latency and jitter. The principle of TAS is to specify pre-computed schedule of time windows, in which it is known in advance which data frames will be transmitted. However, the computation of a feasible TAS schedule is a difficult problem. A large set of solutions [17], [20] for this problem has been provided by the active research community.

An even more challenging problem is the reconfiguration of a TAS schedule at runtime in response to a failure in the network, as described in the ECRTS industrial challenge [4]. Indeed, this may require a lot of computation (to compute the new configuration) and a lot of data exchange (to deploy the new configuration). It may also require handling potential timing issues during the transition from the initial mode to the new mode.

In this paper, we tackle the issue of TAS reconfiguration at runtime in case of a network

failure. For this purpose, we have defined a new TAS configuration approach that anticipates the needs of a potential reconfiguration at runtime in terms of bandwidth. The configuration is done by redirecting the affected critical data flows to other time windows in the event of a network failure impacting specific equipments or wires, while preserving the initially computed TAS schedule and ensuring that the properties of critical flows, which are not directly impacted by the failure, remain unaffected. This makes our TAS configuration approach incremental, which provides a significant advantage for the recertification process. Indeed, by minimizing changes to the existing TAS schedule, we reduce both the cost and time required for recertification, making it a more efficient solution. Another major benefit of our configuration approach is that no bandwidth resources are wasted in TAS windows. In fact, instead of being exclusively reserved for the potential future transmission of redirected critical flows, these resources can be utilized by less critical flows when no critical traffic is being transmitted. We also show how our approach can considerably ease and accelerate the reconfiguration process of TSN networks at runtime, since no new scheduling information has to be computed and deployed on the network switches and nodes, thus making the networks resilient to failures and upgrades. Last, while TAS is known to be fragile with regards to losses of frames [7, 4], this issue is solved by our configuration approach, thus increasing the network resilience even more.

Based on an identified class of schedules that are resilient to frame losses, our main contribution is the definition of a new schedule class, called "window precedence exclusion TAS" (WPEx-TAS), that includes an adaptation mechanism solving the reconfiguration problem, as described above, for TSN networks with a relatively low TAS load, which is compatible with avionic networks designs. In the following paper, we distinguish between schedule classes (set of schedules having common characteristics), and scheduling algorithms (algorithms computing some schedule). Considering the classification from [20], our WPEx-TAS scheduling class is a *wait-allowed* schedule (the frames are allowed to wait in the queues, and several frames can be in a queue at a given time), an *assigned window based* schedule (the schedule is based on windows, and the set of frames in a window is known). The tolerance to frame loss is not obtained through *frames isolation* or *stream isolation* but rather through a new constraint called "window precedence exclusion". It accepts variable size of frames.

The rest of this paper is structured as follows: Section 2 introduces the architecture and behavior of TAS scheduling in TSN, detailing the taxonomy of TAS schedule classes and presenting a state of the art of incremental TAS scheduling solutions. Section 3 introduces the novel WPEx-TAS schedule class, outlining its main principles and features, along with its key benefits, as well as a formalization of the configuration constraints to the TAS windows and flows offsets. Section 4 explains the encoding of WPEx-TAS constraints using a Constraint Programming approach minimizing the total number of TAS windows used over the network links. Section 5 describes the post-processing phase to spread the obtained TAS windows in a smooth way inside their hyperperiod, thus maximizing the opportunities of adding new flows. Section 6 shows how easily streams can be safely added or removed when reconfiguring a TAS schedule designed with WPEx-TAS. Section 7 presents the experimental results on the ECRTS industrial use-case showing the efficiency of the approach, as well as its incremental nature. The paper concludes with a summary of the solution and directions for future research.

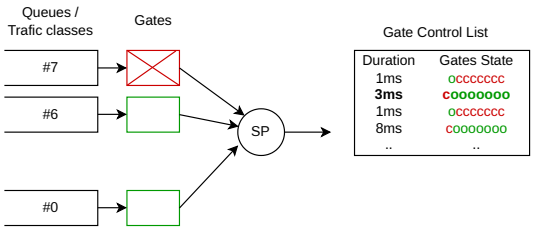

Figure 1 Architecture of a TSN output port (without Transmission Selection Algorithm): height queues (from #0 to #7), scheduled by a static priority arbiter (SP) and open/close gates, whose state is set by the time-driven Gate Control List.

2 Time-Aware Shaper Scheduling

2.1 Architecture and behavior

A TSN output port is made of (up to) 8 priority queues (a.k.a. traffic classes). Each queue is controlled by a Transmission Selection Algorithm (TSA, often called “shaper”) and a gate, which is either open or closed. The gates are controlled by a local table, the Gate Control List (GCL). Each line of the GCL has a duration, and a list determining the state (open or closed) of each gate during that duration. The system starts with the first line, and goes along the table, respecting the duration of each line, and cycles after the last row.

The TSA is in charge of allowing one frame in the queue to compete for transmission. In this paper, we will only consider the default TSA that always allows the head of queue to compete. Figure 1 presents this architecture, without the TSAs.

A frame is selected for transmission by the output port if it is selected by its TSA, if the gate is open, and if the gate will remain open enough to send the frame up to completion before the next closing event. If several frames respect these conditions, the one in the highest priority queue is selected (queue #7 having the highest priority, #0 having the lowest one).

In TAS context, each time interval between an opening and a closing of the same queue is called “window”, “time window”, “slot” or “time slice”. In this paper, we will use “window”. Both TAS queues (i.e. queues for higher priority frames) and other queues have opening and closing events, so they all have windows, but when building TAS schedule, we are mainly interested in windows of TAS queues, called "TAS windows", or simply "window" when the context is clear enough. The general principle of TAS is to schedule windows adequately w.r.t time injection of frames in the network in order to guarantee small and bounded delays and jitter to critical frames.

2.2 Taxonomy of TAS schedule classes

The basic idea of a Time-Triggered (TT) frame schedule is to pre-compute a set of windows, each window being dedicated to one single frame, belonging to a known data flow. The TTEthernet technology is a perfect illustration of this principle [16].

For example, a TT schedule is presented in Figure 4, with five flows (A, B, C, D, E) on a network made of 3 switches and 6 hosts.

Note that TAS frames are stored in FIFO queue. Then, if two frames goes to the same output port, the emission order must be the same as the reception order (conversely to TTEthernet that uses a whiteboard semantics). And if one frame is lost, the next one in the queue may use its slot. This is know as the TAS “queuing constraint”.

When a system transfers TT frames and event-triggered (ET) frames, the part of the time that is not dedicated to TT frames is used by ET frames. In TSN, one or several queues may be devoted to TT frames, and are called TAS queues. In this paper, the term "window" will refer only to windows of TAS queues.

But putting a single frame in a window is costly for other queues. Indeed, in order to ensure exclusive access to the frame in the window, the gates of the other queues are closed when the window opens. And since a frame cannot start its emission unless it can be sent up to completion before the next closing event, this may lead to bandwidth waste for other queues (this effect is sometimes called “guard band” or “dynamic guard band” [8]). One may put several frames in a window to reduce bandwidth waste.

Moreover, the size of the GCL puts an upper bound on the number of possible windows (in one hyperperiod), making it impossible to have only one frame per window [15]. Since there are many ways to build windows and assign frames to windows, there is no unique TAS, but several ones.

To distinguish all TAS flavors, we distinguish *TAS scheduling algorithms* (that will compute a schedule) and what we call *classes of TAS schedules*. A class of TAS schedules is a set of constraints that the schedule must respect. Of course, a given algorithm will compute schedules belonging to some class, but there may exist several algorithms for the same class.

This section presents the main schedule classes with their constraints. The schedule classes can be classified with regards to several criteria.

Queuing delay There are two classes of queuing delays. The *no-wait* principle requires that as soon as a frames is received in an output port, it must have a window to be forwarded without any queuing delay [9]. Other approaches are *wait-allowed*.

Scheduling entity The *frame-based* principle assumes that each window is dedicated to a single frame, whereas in *window-based*, several frames can share the same window. Several *frame-to-window* policies exist: *assigned*, *partially assigned* and *non-assigned* [2]. In [2], the term 0GCL is used to name the *frame-based*, Frame-to-Window-based GCL (FGCL) is used for *assigned frame-to-window* and two *non-assigned* strategies are presented: Window-based GCL (WND), and Flexible WND (FWND). Non-assigned strategies are presented as solutions for un-synchronized and/or unscheduled end-systems.

Queue isolation Due to the TAS “queuing constraint” (related to interaction of FIFO queues and gate opening, cf. Section 2.2), a frame loss may lead to permanent inconsistencies in the schedule [7]. Several strategies can be used: *frame isolation* (never more than on frame in a queue), *stream isolation* (never have frames from different streams in a queue), both defined in [7] or *size based isolation*, combining frame sizes, window size and frame order [6]. Note that variable frame size may lead to the same inconsistencies than frame loss, since the problem occurs when a frame is sent earlier than expected, due to the absence (or the smaller size) of a previous frame. Also note that no-wait schedules ensure frame isolation.

Number of TAS queues One may either consider *single-TAS*, only one queue has TAS behaviour (the other queues using other QoS mechanisms), or *multiple-TAS*, where several queues have TAS behaviour.

Window protection One may either require that, during a TAS window (i.e. when the gate of a TAS queue is open), all other gates are closed: this is *exclusive gating*. One may also consider *overlapping*, when no requirement is done on other gates. In this case, setting TAS queue at the highest priority is a solution to give access at the window to that frame, up to non-preemption of lower priority frames. Last, *protective gating* requires to close the other gates only at window opening. Combined with setting the highest priority to

TAS queue, it avoids the blocking related to non-preemption [5].

It is also of importance to consider possible restrictions of the input data, i.e. the set of flows to schedule. All studies assume periodic streams.

Frames sizes Some algorithms consider that there is a unique size for all frames of all streams, and other accept to schedule streams with different size.

Period restriction Some algorithms assume that the stream periods are harmonic, i.e. there exists a reference period T and each stream has a period T^k for some integer $k > 0$.

Imposed stream offset Some algorithms consider that the stream offset is an input of the problem, and others that the offset can be set by the scheduling algorithm.

Deadline The deadlines are either *implicit* (equal to the period) or *general* (no constraint).

The names "queuing delay", "scheduling entity" and "queue isolation" come from [20]. Using this classification, our scheduling class WPEx-TAS is *wait-allowed*, *window-based* with *assigned* frames in windows, its queue isolation is new, it has no constraint on the number of TAS queues, and it uses *protective gating* window protection. It has no constraint on frame sizes or stream periods, and handle general deadlines. In this paper, to ease notations, only one TAS queue is considered, but the method can be generalized to several queues[1].

2.3 State of the art of incremental TAS scheduling

Computing TAS schedules is a very active area. An overview can be found in [17], with more than 100 references, and a deep study of 17 relevant algorithms can be found in [20]. Less effort has been put on the reconfiguration of TAS or incremental TAS configuration.

An incremental algorithm is proposed in [14], but it is limited to No-Wait sheduling, with the same size for all windows (this size being the size of the longest frame multiplied by the longest path).

The notion of *flexcurve* is used in [11]. A curve h_p is computed for each port p, and $h_p(n) = m$ means that there exist m sequences of at least n free consecutive slots (it assumes a constant slot size G). This curve is useful for shortening insertion test. The incremental algorithm considers only flows with a period equal to the hyperperiod. It does not consider the TAS queuing constraint (cf. Section 2.2) and can not be applied to TAS.

A solution with redundant paths and online update can be found in [10]. The redundancy is computed offline. The online reconfiguration uses a heuristics to compute routes and offsets, considering respect of deadlines. The update of Ethernet Time-Triggered network (including stream add/removal, and routing) is considered in [18] as a bin-packing problem and provides four heuristics. It claims to address the TSN/TAS reconfiguration, but unfortunately, since it does not consider the TAS queuing constraint, it can not be applied to TSN.

None of the above studies considers the coherence problem. Once the new configuration computed, it has to be forwarded to all impacted network elements (update of switch GCL, etc). The switching from one configuration to another has to be done in a smart way. However, it is very hard to update all configuration tables exactly at the same time, and even if it would have been possible, it remains an inadequate solution since at this switching instant, frames whose transmission started during the old configuration may subsist in the switch queues. There is no guarantee that these frames will fit in the new schedule. Such pending frames from the previous configuration may use windows dedicated to the new

[1] This point will be addressed in conclusion.

frames, or even block the queues if the windows are not large enough to include them. This problem is addressed in [19], but without considering the queuing constraint.

2.4 Toward a schedule class designed for reconfiguration

TAS has very good properties in terms of timing performance. The latency can be very small (with *no-wait* configuration, it is only the sum of per switch store and forward technological delays), the jitter is also very small (for a given stream, if all windows in the last link are purely periodic and frames are of constant size, in a *frame-based* schedule, the jitter is null whereas in a *window-based* schedule, the jitter is the window size minus the frame size).

As shortly mentioned before, TAS scheduling is very fragile with regard to losses. If a frame is lost, the "next" frame can be sent in its place. But since the size and the route of this frame may be different, it can break a schedule, i.e. the loss of a frame of one flow may imply deadline miss of another flow, and this deadline miss may be *permanent* [4, Figure 5].

It is also quite complicated to update a TAS schedule (adding a few flows for example). Most scheduling algorithms are not incremental, hence an updated schedule may be very different from the previous one. It means that a new configuration must be sent to each component (hosts and switches). This configuration may update the frame production offset which in turn may lead to an update of the task scheduling on hosts. Even for incremental algorithms, the coherence problem (see Section 2.3) may invalidate the new schedule.

As stated in the introduction, this paper presents an approach that allows adding or removing data flows without reconfiguring the GCL. Therefore, we avoid the problem of redistributing the GCL and the coherence problem at switching.

3 Windows Precedence Exclusion schedule class

We now present the WPEx-TAS schedule class, first informally (Section 3.1) with a focus on the expected benefits (Section 3.2). Formal definitions are given in Sections 3.3 to 3.5.

3.1 Mains principles

The aim of Windows Precedence Exclusion TAS scheduling class is to be able to remove and add flows in existing TAS windows without modifying the GCL. This is done by combining three ideas: a window exclusion relation (giving its name to the method), a window enlargement (offering slack for adding new flows), and the protective gating that offers slack for free.

3.1.1 Precedence relation and exclusion

The principle is to define TAS windows hosting several frames (*window-based* scheduling with *assigned frame-to-window* policy). The order of the frames in the window is *unknown*. Therefore, the exact departure of each frame in the window is unknown. This implies that, for a given frame, its next window along the path must start after the end of the current window. This constraint is called the "window precedence exclusion" since two windows cannot overlap if one frame comes from one window to the other. Consider Figure 2 involving three windows, W1, W2, W3 and three frames A, B, C. The frames are received on some input port, in window W1, and forwarded to two different output ports, in windows W2 and W3. The WPEx constraint is not respected between W1 and W3, but it is between W1 and W2. One can see that a frame going from W1 to W2 has a higher latency than a

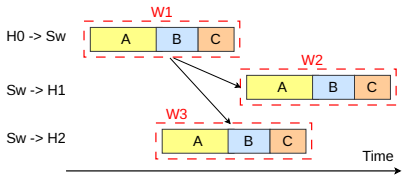

Figure 2 A window W1 forwarding frames A, B, C to windows W2, W3. Only (W1,W2) respects the WPEx constraint. W1 is on the input link of some switch, and W2, W3 are on two output links of this switch.

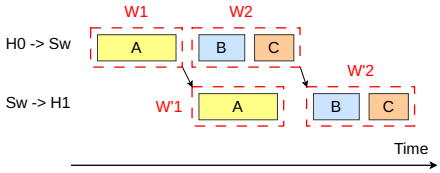

Figure 3 Overlapping scheme to be excluded.

frame going from W1 to W3 (e.g. when going from W1 to W3, frame A is forwarded as soon as it is received, whereas when going from W1 to W2, frame A has to wait the end of W1). Nevertheless the WPEx constraint has several benefits. We will enforce the "window precedence exclusion" condition by setting that if there is a frame going from one window w to another w', there is no other window on the same link that overlaps with w or w' or is between w and w'. In Figure 3, there is an overlap between W'1 and W2. In such a case, if A is absent, B may be sent in W2.

Such a schedule has two important properties:

1. The WPEx constraint makes a schedule tolerant to losses: if a frame is not in its window, it may change the emission time of the other frames in the window but the emission will still occur during the window, which is the only information used to build the schedule. A frame lost (or not being emitted by its task) will not have its place used by a frame assigned to another window (the aim is to avoid the fact that, in Figure 3, if A is absent, B will be transmitted in its window W'1). This tolerance to losses means that one flow can be removed without breaking the schedule, which was one of our goals.
2. A second property is that all frames assigned to a window are already in the queue when the windows starts. This property will be of importance in the following.

An example of a WPEx-TAS schedule is provided in Figure 4.

3.1.2 Window enlargement

The second idea, once a schedule is built, is to extend the end of each window up to the start of the nearest window (up to internal technological latency between input port and output port in the switch, to ensure that all frames assigned to a window are in the queue before window start). This enlargement gives some slack to host new frames. This enlargement is illustrated in Figures 5 and 6. The possibility to add new flows without updating the GCLs comes from this slack in enlarged windows. If the system is able to find an injection time for a new flow such that it will get a place in each window along its path, it can be added without updating the GCLs. An example of window enlargement is provided in Figure 6.

This enlargement will be small if the windows have been put back-to-back to minimize latencies. Therefore, to have maximal enlargement (i.e. to have a maximal slack), it is of importance to build a schedule with a maximal distance between windows along a frame path. This suggests that a schedule designed to be reconfigurable should not try to minimize network latencies, but only to maximize the latencies while respecting the deadlines. However, this slack can be seen as a waste of bandwidth, since it cannot be used by lower priority flows. The protective gating, presented in the next section, gives this slack for free.

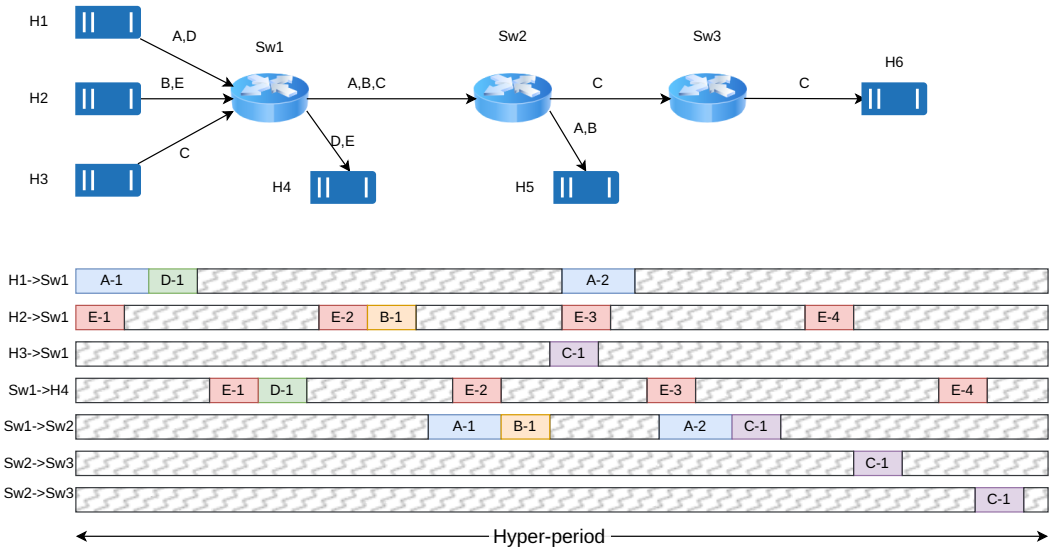

Figure 4 Example of a window non-overlapping scheduling. Only schedule of queues #7 is given. Grey dashed boxes represent closing times of gates. Box X-i represents the transmision time of the i-th frame of stream X (in a TAS window). – At start of hyperperiod, the gate in H1 is open, and frames A-1 and D-1 can be forwarded (it is assumed that they have been put in the queue). Both are forwarded to switch Sw1, where A-1 (resp. D-1) is put int the queue of the output port connected to Sw2 (resp.H4)

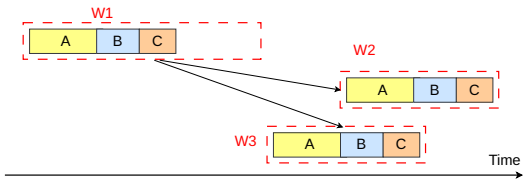

Figure 5 A window W1 forwarding frames A, B, C to windows W2 and W3. W1 is enlarged up to W3 start.

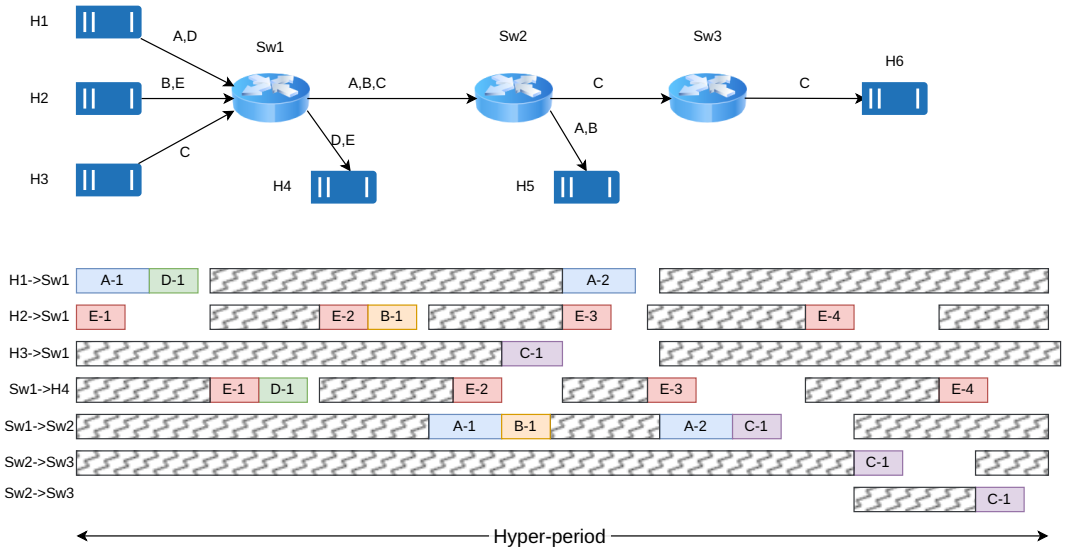

Figure 6 Enlargement of the scheduling of Figure 4. White free space is the unused part of windows, called "slack".

3.1.3 Protective gating

Then comes the third principle: protecting only the start of the window, and putting the TAS queue at the highest priority level, as described in [5] and called *protective-gating* and illustrated in Figure 7.

To protect TAS windows from interferences of other queues, one common principle in TAS is the *exclusive gating*: during a TAS window, all other gates are closed. However, thanks to priorities, this condition can be relaxed. Let the (unique) TAS queue have the highest priority (i.e. queue #7 is the only TAS queue). Now consider that the gates of the lower priority queues are open during a TAS window. Since the TAS frames have the highest priority, they will gain any arbitration with any other frame from the lower priority queues. However, due to non-preemption (or limited preemption), if a lower priority frame is being emitted while the TAS frame is ready for transmission, this lower priority frame (or a fragment of this frame) will delay the TAS frame.

Consider now a schedule such that all TAS frames to be sent during a window w are in the queue before the start of the window (i.e. the opening of the gate queue). Only the first frame may have to compete with any lower priority frame. Once the frame of highest priority has started its emission, all TAS frames are sent back-to-back (at end of emission of a TAS frame, the next TAS frame wins the arbitration with any lower priority frame).

With protective-gating, if all TAS frames assigned to a TAS window are in the queue before the window starts (which is the case with WPEx-TAS), all TAS frames are sent back-to-back from start of window, without any blocking by a lower priority frame.

Therefore, the slack obtained through window enlargement can be used by lower priority flows as soon as all TAS frames are transmitted. Note also that in case of a slack due to a variable TAS frame size (TAS frame size is smaller than the maximal expected size) protective-gating allows lower priority frame to use it.

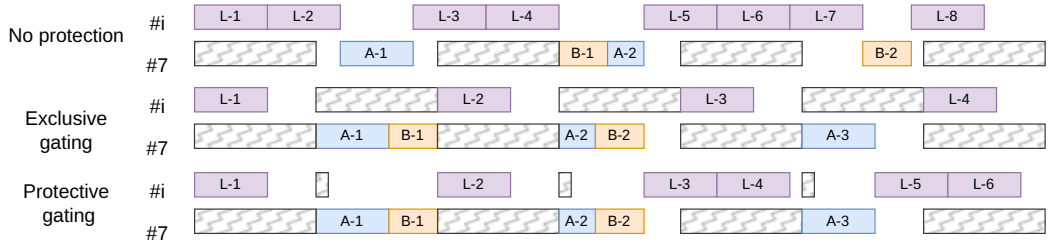

Figure 7 Illustration of exclusive gating vs protective gating. Unlabeled gray boxes represent closing time of gates in queues #7 and #i ($i \in [0, 6]$). Colored boxes represent data frames: X-i is the i-th frame of stream X. – Two TAS flows, A and B, are in competition with a low priority flow L. – With exclusive gating, the window assigned to TAS queue #7 is fully protected, and cannot be used by lower priority queue, even if it is not fully used (because of variable size frame, like A-2) or because some frame is absent (no B-3 in the third window). In protective gating, the small gate closing prevents any lower priority frame to encroach in the time window, and static priority ensure that TAS frames are sent back-to-back. The frames are fully protected, but the unused part of the window can be used by lower priority frames. If no protection was used, in a non-preemptive system, low priority frame can start their transmission before the TAS windows and encroach on the time windows.

3.2 Expected benefits of the WPEx class

The WPEx-TAS schedule class has several benefits. The two first are common with all window-based approaches, while the two others are specific to this approach.

First, like other window-based approaches, grouping several frames in a single window reduces the overhead due to gate closing. Second, as already said, the limited size of the GCL puts an upper bound on the number of possible windows (within one hyperperiod), making it impossible to have only one single frame transmitted per window when the number of streams and frames is large [15]. Window based approaches scale better.

The third benefit of the WPEx-TAS schedule class is the immunity to frame loss: the loss of a frame does not break the schedule, without imposing any restriction on the sharing of the window (in the flow-isolation, only frames from the same stream can share a window).

However, the main benefit is the fourth: the transparent addition of flows into windows (see Figure 8). By combining the non-overlapping window principle with the protective gating, it is possible to add flows in windows without modifying the gate control list. This makes the TAS reconfiguration task easier since no updated GCL information has to be sent to the switches (of course, updated routing information of redirected flows must still be sent). By simply using the window enlargement, a new TAS flow can be inserted in the TAS schedule:

1. without changing the GCL in the switches,
2. without changing any offset of other TAS flows,
3. while respecting the bounds on delays and the jitter of all other TAS flows,
4. without generating any transient load during mode change (change from one configuration another) which might have led to timing issues.

Additional benefits are:

1. a quick reconfiguration of TAS schedule (no configuration update of hosts and switches),
2. a reduced effort of re-certification (or event avoid re-certification) in case of an update of a network configuration.

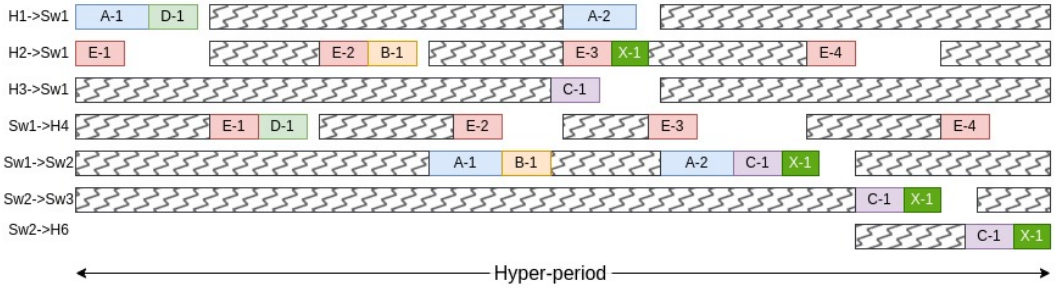

Figure 8 Add a flow X-1 along path H2->SW1->SW2->SW3->H6 to the schedule in Figure 6 without window update.

3.3 Hypotheses and notations

The system hypotheses are:

- Only the queue #7 (with highest priority) uses TAS.
- The TAS flows (i.e. the flows that will be forwarded in queues #7) are strictly periodic, and a maximal frame size is known.
- The TAS flows have no intrinsic offset constraint, the per flow offset can be provided by the algorithm computing the TAS schedule.
- To ease notations, no window will encroach the hyperperiod, and all frames must be sent and received in the same hyperperiod.[2]
- No assumption is done on other queues: any other shaper can be used.

Let $\mathbf{V}$ the set of nodes, and $\mathbf{E} \subset \mathbf{V} \times \mathbf{V}$ the set of (oriented) links. For any $v, v' \in \mathbf{V}$, $[v, v']$ denotes the link from v to v' (if it exists). A path $p = [[v_0, v_1], [v_1, v_2], \ldots, [v_{n-1}, v_n]]$ is an element of $\mathbf{E}^+$, with $\mathit{first}(p) = v_0$, $\mathit{last}(p) = v_n$, and $\mathbf{P}$ denotes the set of paths. $[v, v'] \in p$ is used to note that p matches $[\ldots [v, v'] \ldots]$. A stream s is a tuple $\langle \mathit{src}, \mathit{dst}, T, L, e2e, J, p, \phi \rangle$ where $\mathit{src}, \mathit{dst} \in \mathbf{V}$ are the source and destination of the flow, $T, L, e2e, J \in \mathbb{Q}^+$ are periods, length[3], end-to-end maximum latency, and maximum jitter, $p \in \mathbf{P}$ the stream path and ϕ the stream offset such that $\mathit{first}(p) = \mathit{src}$ and $\mathit{last}(p) = \mathit{dst}$. The set of streams is $\mathbf{S}$, and $s.\mathit{dst}, s.\mathit{src}, s.T, s.L, s.p, s.\phi$ are notations for tuple elements. All these sets are finite.

A stream s will generate an infinite number of frames that will go along the path $s.p$. The i-th frame instance on link $[v, v']$ is denoted $f_{s,i}^{[v,v']}$, and $\mathbf{F}$ is the set of frame instances, $\mathbf{F}_s$ the set of frames instances of stream s, $\mathbf{F}^{[v,v']}$ the set of frames instances on link $[v, v']$ and $\mathbf{F}_s^{[v,v']} = \mathbf{F}_s \cap \mathbf{F}^{[v,v']}$. Let $s^{-1} : \mathbf{F} \to \mathbf{S}, f_{s,\cdot} \mapsto s$.

We assume the same throughput R for all links, to ease readability of relations, but a per-link data rate $R_{[v,v']}$ may be added without changing the approach.

3.4 Constraints related to window non-overlapping

This paper considers window-based scheduling. A window w is a time interval on a link, devoted to host a fixed set of frames. A window is tuple $\langle \phi, L, [v, v'], F \rangle$ where $\phi, L \in \mathbb{Q}^+$ denote the offset and length[4] of the window, $[v, v'] \in \mathbf{E}$ the link, F the set of frames in this

[2] This is not a limit of the approach itself, but it simplifies notations.
[3] Frame size divided by link rate, including any media overhead like preamble, inter packet gap, etc.
[4] This length is expressed in bit-time.

window, and $w.\phi, w.L, w.[], w.F$ are notations for tuple elements (with $w.[] = [v, v']$). The set of windows on link $[v, v']$ is denoted $\mathbf{W}^{[v,v']}$.

In the following, the dot $\cdot$ will be used in place of a variable that is useless in a formula (i.e. existentially quantified but never used).

Let v be a node, and $w \in \mathbf{W}^{[\cdot,v]}, w' \in \mathbf{W}^{[v,\cdot]}$. Then w is a *supplier* of w' if one frame in w is forwarded in w', i.e.

$$w \textit{ is-supplier-of} w' \iff ((w.F \cap w'.F \neq \emptyset) \text{ and } (\textit{last}(w.[]) = \textit{first}(w'.[]))) \tag{1}$$

In our context, the length of a window is the sum of the length of its frames:

$$\forall w \in \mathbf{W}, w.L = \sum_{f \in w.F} s^{-1}(f).L \tag{2}$$

The non-overlapping principle is the following. Let w a supplier of w':

1. w is before w' and there is no overlapping between w and w', cf eq. (3).
2. on the same links (i.e. $w.[]$ or $w'.[]$) there is no other window overlapping with w and w', and neither between w and w' (and this double condition is expressed with eq. (4)).

$$w.\phi + w.L \leq w'.\phi \tag{3}$$

$$\forall u \in \mathbf{W}^{w.[]} \cup \mathbf{W}^{w'.[]} : (u.\phi + u.L \leq w.\phi) \text{ or } (w'.\phi + w'.L \leq u.\phi) \tag{4}$$

Eq. 3 does not consider the internal delay of the switch between the reception of a frame and its copy in the output queue, to limit the number of notations in the paper. If this delay is bounded by some constant δ, the constraint should be written $w.\phi + w.L + \delta \leq w'.\phi$.

Such a schedule has two important properties:

1. If a TAS schedule respects this non-orverlaping mechanism, then all frames of window w are in the queue before the opening of w.
2. It is tolerant to losses. The loss of a frame has no impact on the main property (the frames that have to be sent in a window are received before the window opening).

3.5 Constraints related to flows in windows

The schedule will be built only on the hyperperiod H. So for each stream i, we only have do find routes for $f_{s,i}^{[v,v']}$ for any $[v, v'] \in s.p$ and $i \in [1; \frac{H}{s.T}]$. To shorten notation, let $h_s = \frac{H}{s.T}$. Whereas we have introduced the constraints specific to non-overlapping windows with eq. (3) and (4), here are introduced the common conditions dedicated to routing.

1. Each frame belongs to a unique window along its path

$$\forall s, \forall i \in [1; h_s], \forall [v, v'] \in s.p : \exists! w, f_{s,i}^{[v,v']} \in w.F \tag{5}$$

From uniqueness, define the function $\hat{w}$

$$\hat{w} : \mathbf{F} \to \mathbf{W}, f \mapsto w \textit{ s.t. } f \in w.F \tag{6}$$

2. The first window is aligned with the stream offset, i.e. the frame is released before its window, and there is no window on the same link between both

$$\begin{aligned} &\forall s, \forall i \in [1; h_s], s.p = [[v_0, v_1] \cdots] : \\ &\left(s.\phi + (i-1)s.T \leq \hat{w}(f_{s,i}^{[v_0,v_1]}).\phi\right) \text{ and} \\ &\qquad \left(\nexists w' \in \mathbf{W}^{[v_0,v_1]} : s.\phi + (i-1)s.T \leq w'.\phi \leq \hat{w}(f_{s,i}^{[v_0,v_1]}).\phi\right) \end{aligned} \tag{7}$$

$\mathbf{V}$	(finite) set of vertices (ex: v, v')
$\mathbf{E} \subset \mathbf{V} \times \mathbf{V}$	set of oriented links (ex: $[v, v']$)
$[[v_0, v_1], \ldots, [v_{n-1}, v_n]]$ $first(p)$=v_0, $last(p)$=v_n $[v_1, v_2] \in p$	path (in $\mathbf{E}^+$) first and last nodes of a path edge $[v_1, v_2]$ is in path p
$\langle src, dst, T, L, e2e, J, p, \phi \rangle$	stream (in $\mathbf{S}$), src: source, dst: destination, T: period, L: length / frame size, $e2e$: end-to-end max. latency, J: max. jitter, p: path, Φ: offset
H	hyperperiod of streams periods
$f_{s,i}^{[v,v']}$ $\mathbf{F}_s^{[v,v']}$ $\mathbf{F}_s$ $\mathbf{F}$	i-th frame instance of stream s on link $[v, v'] \in s.p$ frames instances of stream s on link $[v, v'] \in s.p$ frames instances of stream s frames instances of all streams
$s^{-1} : \mathbf{F} \to \mathbf{S}$	stream of a frame
$w = \langle \phi, L, [v, v'], F \rangle$ $\mathbf{W}^{[v,v']}$ $\mathbf{W}$	window starting at ϕ of duration L, on link $[v, v']$, involving frames in F set of windows on link $[v, v']$ set of windows

Table 1 Main notations of the paper

3. The frame must be received before its deadline

$$\forall s, \forall i \in [1; h_s], s.p = [\cdots [v_{n-1}, v_n]] : \\ \hat{w}(f_{s,i}^{[v_{n-1},v_n]}).\phi + \hat{w}(f_{s,i}^{[v_{n-1},v_n]}).L \leq s.\phi + (i-1)s.T + s.e2e \quad (8)$$

4. The stream must respect its jitter, i.e. the difference between the latest frame arrival and the soonest frame arrival must be smaller than the jitter

$$\forall s, \forall i, j \in [1; h_s], s.p = [\cdots [v_{n-1}, v_n]] : \\ \left(\hat{w}(f_{s,i}^{[v_{n-1},v_n]}).\phi + \hat{w}(f_{s,i}^{[v_{n-1},v_n]}).L\right) - \left(\hat{w}(f_{s,i}^{[v_{n-1},v_n]}).\phi + s^{-1}(f).L\right) \leq s.J \quad (9)$$

4 Encoding using constraint programming

The problem presented before is encoded using a constraint programming approach, based on the modeling elements available in IBM ILOG CP Optimizer [13]. One of these elements is the concept of *interval variables*. In constraint programming, an interval variable itv represents a task defined by a start time $startOf(itv)$, an end time $endOf(itv)$, and a presence $presOf(itv) \in \{0, 1\}$. The presence attribute is useful to model optional tasks. An interval also has a length defined by $lengthOf(itv) = endOf(itv) - startOf(itv)$. Various constraints can be posted to express complex requirements over interval variables (e.g., non-overlapping constraints), and so-called *constraint propagation* techniques help prune the possible decisions during search. We describe below the different parts of the model built. The detailed definition of the constraint types used can be found at [12].

4.1 Window model from the point of view of the streams

The first part of the model describes the scheduling problem from the point of view of each stream s. For this, we introduce three kinds of interval variables for the ith frame of stream s, namely:

- $\forall [v,v'] \in s.p$, $WS_{s,i}^{[v,v']}$: non-optional interval variable representing the window within which the ith frame of stream s is placed on link $[v,v']$;
- $Wall_{s,i}$: non-optional interval variable representing the time period during which the ith frame of stream s "occupies" the network;
- $\forall [v,v'] \in s.p$, $WSE_{s,i}^{[v,v']}$: non-optional interval variable representing the *extended window* during which the ith frame of stream s locks link $[v,v']$, given the non-overlapping constraints of eq. 4.

All these intervals must be contained in $[(i-1)\cdot s.T, i\cdot s.T]$ and their length is contained in $[s.L, s.T]$. Several constraints are imposed over these intervals. In the following, the path of stream s is $s.p = [[v_0,v_1],\ldots,[v_{n-1},v_n]]$. Eq. 10 imposes precedence constraints between the successive windows used by the frame (encoding of eq. 3). Eq. 11 expresses that interval $Wall_{s,i}$ must cover all the windows containing the ith frame. Eq. 12 impose a maximum end-to-end duration. Eq. 13 corresponds to the jitter constraint. Eq. 14-15 express that the extended window used by the ith frame on a link $[v,v']$ covers the windows used by that frame on the previous and next links (encoding of eq. 4; see Figure 9). Eq. 16-17 correspond to a specific management of the initial and final links.

$$\forall k \in [1; n-1],\ endBeforeStart(WS_{s,i}^{[v_{k-1},v_k]}, WS_{s,i}^{[v_k,v_{k+1}]}) \tag{10}$$

$$startBeforeStart(Wall_{s,i}, WS_{s,i}^{[v_0,v_1]}) \wedge endAfterEnd(Wall_{s,i}, WS_{s,i}^{[v_{n-1},v_n]}) \tag{11}$$

$$lengthOf(Wall_{s,i}) \leq s.e2e \tag{12}$$

$$lengthOf(WS_{s,i}^{[v_{n-1},v_n]}) \leq s.L + s.J \tag{13}$$

$$\forall k \in [1; n-1],\ startBeforeStart(WSE_{s,i}^{[v_k,v_{k+1}]}, WS_{s,i}^{[v_{k-1},v_k]}) \tag{14}$$

$$\forall k \in [1; n-1],\ endAfterEnd(WSE_{s,i}^{[v_{k-1},v_k]}, WS_{s,i}^{[v_k,v_{k+1}]}) \tag{15}$$

$$startBeforeStart(WSE_{s,i}^{[v_0,v_1]}, Wall_{s,i}) \tag{16}$$

$$endAfterEnd(WSE_{s,i}^{[v_{n-1},v_n]}, Wall_{s,i}) \tag{17}$$

Last, eq. 18 defines the offset constraint for each stream s.

$$\forall i \in [1; h_s - 1],\ startOf(Wall_{s,i+1}) = startOf(Wall_{s,i}) + s.T \tag{18}$$

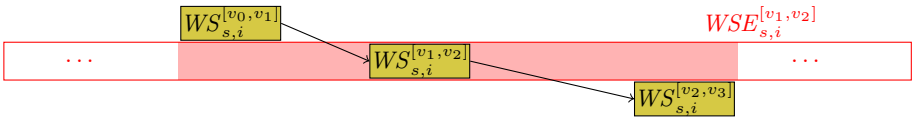

Figure 9 Windows $WS_{s,i}^{[v,v']}$ for the ith frame of a stream s such that path $s.p = [[v_0,v_1],[v_1,v_2],[v_2,v_3]]$, and extended window $WSE_{s,i}^{[v_1,v_2]}$ in the constraint programming model

4.2 Window model from the point of view of the links

For each link $[v,v'] \in \mathbf{E}$, we introduce as many candidate windows as the number of frames using that link ($|\mathbf{F}^{[v,v']}|$ windows in the worst-case). We introduce both window intervals and extended window intervals, again to model the non-overlapping constraints of eq. 4:

- $\forall m \in [1; |\mathbf{F}^{[v,v']}|]$, $WL_m^{[v,v']}$: optional interval variable representing the mth candidate window using link $[v,v']$;
- $\forall m \in [1; |\mathbf{F}^{[v,v']}|]$, $WLE_m^{[v,v']}$: optional interval variable representing the mth candidate extended window placed on link $[v,v']$.

If candidate interval $WL_m^{[v,v']}$ is created to potentially cover the ith frame of stream s, then intervals $WL_m^{[v,v']}$ and $WLE_m^{[v,v']}$ must be contained in $[(i-1)\cdot s.T, i\cdot s.T]$. Eq. 19 expresses that the extended windows placed on a link must not overlap. The formulation uses the *noOverlap* constraint, a standard constraint in constraint programming. Eq. 20 imposes a consistency between the presences of the intervals representing the windows and the extended windows. Eq. 21-22 express that the windows are contained in the extended windows.

$$noOverlap(\{WLE_m^{[v,v']} \mid m \in [1;|\mathbf{F}^{[v,v']}|]\}) \tag{19}$$
$$\forall m \in [1;|\mathbf{F}^{[v,v']}|],\ presOf(WLE_m^{[v,v']}) = presOf(WL_m^{[v,v']}) \tag{20}$$
$$\forall m \in [1;|\mathbf{F}^{[v,v']}|],\ startBeforeStart(WLE_m^{[v,v']}, WL_m^{[v,v']}) \tag{21}$$
$$\forall m \in [1;|\mathbf{F}^{[v,v']}|],\ endAfterEnd(WLE_m^{[v,v']}, WL_m^{[v,v']}) \tag{22}$$

4.3 Relationships between the stream view and the link view

To define the mapping between the windows $WS_{s,i}^{[v,v']}$ used by the frames of a stream and the candidate windows $WL_m^{[v,v']}$ available on the links, we introduce three additional sets of variables for each stream s:

- $\forall i \in [1;h_s], \forall [v,v'] \in s.p,\ x_{s,i}^{[v,v']} \in [1;|\mathbf{F}^{[v,v']}|]$: integer variable representing the index of the candidate interval on link $[v,v']$ that is used by the ith frame of stream s ($x_{s,i}^{[v,v']} = m$ means that the ith frame of stream s uses window $WL_m^{[v,v']}$);
- $\forall [v,v'] \in s.p, \forall m \in [1;|\mathbf{F}^{[v,v']}|],\ WX_{s,m}^{[v,v']}$: optional interval variable representing a window that the frames of s may use on link $[v,v']$; as shown later, these intervals correspond to a gateway between the intervals of the stream view and the intervals of the link view;
- $\forall [v,v'] \in s.p,\ \forall m \in [1;|\mathbf{F}^{[v,v']}|],\ WXE_{s,m}^{[v,v']}$: optional interval variable representing an extended window that the frames of s may use on link $[v,v']$.

From this, several constraints are imposed. Some of them are illustrated in Figure 10. Eq. 23 first exploits the *packing* constraint available in constraint programming to model multi-knapsack problems. Basically, if stream s uses link $[v,v']$, the ith frame of s can be viewed as an item of size $s.L$ that must be put in one of the candidate intervals (or containers) available on $[v,v']$. The constraint ensures that the size of interval $WL_m^{[v,v']}$ corresponds to the sum of the sizes of the frames put in that interval (i.e., such that $x_{s,i}^{[v,v']} = m$).

$$pack(\left[lengthOf(WL_m^{[v,v']}) \mid m \in [1;|\mathbf{F}^{[v,v']}|]\right],\ \begin{matrix}[x_{s,i}^{[v,v']} \mid s \in \mathbf{S}, i \in [1;h_s], [v,v'] \in s.p],\\ [\ s.L\ \mid s \in \mathbf{S}, i \in [1;h_s], [v,v'] \in s.p])\end{matrix} \tag{23}$$

Then, we impose several constraints for every stream $s \in \mathbf{S}$. Eq. 24 uses the *isomorphism* constraint available in CP Optimizer. In our case, it expresses that there must be a 1-1 matching between on one side the windows used by the frames of s on a given link and on the other side the windows that are actually present on the link (see Figure 10). This matching is defined through the $x_{s,i}^{[v,v']}$ variables. Eq. 25 synchronizes the start and end times of some intervals of the model (see Figure 10 again). Eq. 26 synchronizes their presences. In the model, the intermediate $WX_{s,i}^{[v,v']}$ intervals are useful because some $WL_m^{[v,v']}$ intervals can be present even if they are not used by stream s, as represented in Figure 10; this is why we do not define a direct isomorphism between the $WS_{s,i}^{[v,v']}$ intervals and the $WL_m^{[v,v']}$ intervals.

Eq. 27 to 29 concern the extended windows and are similar.

$$isomorphism([WX_{s,1}^{[v,v']}, \ldots, WX_{s,|\mathbf{F}^{[v,v']}|}^{[v,v']}], [WS_{s,1}^{[v,v']}, ..., WS_{s,h_s}^{[v,v']}], [x_{s,1}^{[v,v']}, ..., x_{s,h_s}^{[v,v']}]) \quad (24)$$

$$\forall m \in [1; |\mathbf{F}^{[v,v']}|],\ startAtStart(WX_{s,m}^{[v,v']}, WL_m^{[v,v']}) \wedge endAtEnd(WX_{s,m}^{[v,v']}, WL_m^{[v,v']}) \quad (25)$$

$$\forall m \in [1; |\mathbf{F}^{[v,v']}|],\ presOf(WX_{s,m}^{[v,v']}) \rightarrow presOf(WL_m^{[v,v']}) \quad (26)$$

$$isomorphism([WXE_{s,1}^{[v,v']}, ..., WXE_{s,|\mathbf{F}^{[v,v']}|}^{[v,v']}], [WSE_{s,1}^{[v,v']}, ..., WSE_{s,h_s}^{[v,v']}], [x_{s,1}^{[v,v']}, ..., x_{s,h_s}^{[v,v']}]) \quad (27)$$

$$\forall m \in [1; |\mathbf{F}^{[v,v']}|],\ startAtStart(WXE_{s,m}^{[v,v']}, WLE_m^{[v,v']}) \wedge endAtEnd(WXE_{s,m}^{[v,v']}, WLE_m^{[v,v']}) \quad (28)$$

$$\forall m \in [1; |\mathbf{F}^{[v,v']}|],\ presOf(WXE_{s,m}^{[v,v']}) \rightarrow presOf(WLE_m^{[v,v']}) \quad (29)$$

Last, eq. 30-31 allow us to avoid activating useless windows on the links.

$$\forall m \in [1; |\mathbf{F}^{[v,v']}|],\ presOf(WL_m^{[v,v']}) \rightarrow or(\{presOf(WX_{s,m}^{[v,v']}) \mid s \in \mathbf{S}, [v,v'] \in s.p\}) \quad (30)$$

$$\forall m \in [1; |\mathbf{F}^{[v,v']}|],\ presOf(WLE_m^{[v,v']}) \rightarrow or(\{presOf(WXE_{s,m}^{[v,v']}) \mid s \in \mathbf{S}, [v,v'] \in s.p\}) \quad (31)$$

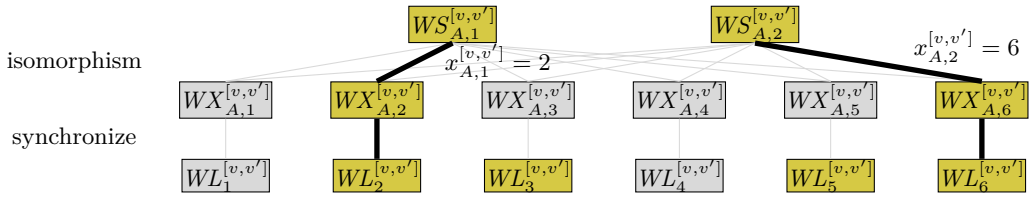

Figure 10 Relationships between the intervals associated with link $[v, v']$ and the intervals associated with stream A (present intervals in a solution represented in yellow)

4.4 Objective function

The goal is to minimize the total number of windows used over all the links, that is:

$$\text{minimize} \sum_{[v,v'] \in \mathbf{E}, m \in [1; |\mathbf{F}^{[v,v']}|]} presOf(WL_m^{[v,v']}) \quad (32)$$

4.5 Optimizations to improve the search process

Additional constraints (symmetry breaking constraints) are added to prune the search space. For each link $[v, v']$, we identify one stream s (a unique one) that has a shortest period among the streams using $[v, v']$. As s uses exactly h_s windows on link $[v, v']$ over the hyperperiod, we arbitrarily impose constraint $x_{s,i}^{[v,v']} = i$ for every $i \in [1; h_s]$.

Moreover, for each link $[v, v']$, the candidate windows defined from streams that have the same period are equivalent. For each index $m \in [1; |\mathbf{F}^{[v,v']}|]$, we look for the next index $m' > m$ such that intervals $WL_m^{[v,v']}$ and $WL_{m'}^{[v,v']}$ are equivalent in terms of minimum start time and maximum end time, and we impose constraint $presOf(WL_{m'}^{[v,v']}) \rightarrow presOf(WL_m^{[v,v']})$. This forces the solver to activate optional interval $WL_m^{[v,v']}$ before activating $WL_{m'}^{[v,v']}$.

Last, we exploit a so-called *variable ordering heuristic* to help the solver branch on the right decision variables during the exploration of the search tree. The heuristic chosen consists in branching first on the $x_{s,i}^{[v,v']}$ variables, since once these variables are fixed, the presence of all the intervals is known and CP Optimizer exploits powerful constraint propagation mechanisms to handle the remaining scheduling constraints over the present interval variables.

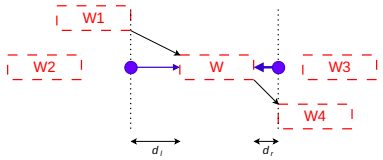

Figure 11 Forces applied to a window, and pseudo-particles setting the same force from boundaries. Violet circles are the virtual particles.

5 Post-processing

The constraint model proposed minimizes the number of windows. But to both maximize the opportunities of adding new flows and reduce the blocking imposed by TAS to lower priority flows, it is important to spread the windows in a smooth way along the hyperperiod. To this aim, we have implemented a spreading inspired by the electromagnetic repulsion force.

5.1 Inter-window repulsion force

In electromagnetic theory, two particles with the same charge apply each other a force whose strength is proportional to $\frac{1}{d^2}$ where d is the distance between both particles. Inspired by it, and considering that our system has one dimension only, we consider that if w *is-supplier-of* w', then w applies to w' a pressure $\frac{1}{w'.\phi-(w.\phi+w.L)}$ and conversely. It is also of importance to have space between windows on the same link, and two successive windows on the same link also apply on each other the same force. Then, for each window, we can compute l and r, the forces applied respectively from the left (to the right) and from the right (to the left).

But the movement of an individual window is constrained by the WPEx-TAS constraint. On its left, a window cannot be shifted more than up to overlapping its supplier windows. It can neither overlap with the previous window on the same link. Last, if it is the first window of the path of a frame, it must respect a maximal distance with the last window of this frame path. Taking the minimum of these three maximal distances gives us a maximal shift to the left, denoted by d_l. Similar constraints exist on a possible right shift, leading to d_r.

These points are illustrated in Figure 11. The window of interest, W, receives a force from left to right $\frac{1}{2}$ from its supplier W1, and $\frac{1}{4}$ from W2, leading to a total force $\frac{3}{4}$. The left movement boundary is the end of W1. It also receives two forces from right to left 1 and $\frac{1}{2}$, leading to a total of $\frac{6}{4}$.

The suggested movement for a window w with left and right forces and distances l, r, d_l, d_r is the one that would be in equilibrium between forces if they were produced by left and right particles at respective distances d_l, d_r, with respective charges c_l and c_r such that $\frac{c_l}{d_l} = l$, $\frac{c_f}{d_r} = r$. That is to say, the suggested movement x is such that $\frac{c_l}{d_l+x} = \frac{c_f}{d_r}$, leading to $x = d_r d_l \frac{l-r}{l d_l + r d_r}$. By construction, such a movement preserves the set of constraints, except for the specific case of "start-of-path" window. A *start-of-path* window is a window w on a link l that contains a frame f such that the link is the first of the frame path. Such a window has specific constraints (cf. eq. 7), related to the offset of the flow.

5.2 Looping on individual movements

The inner loop selects each window in sequence, and for each window, computes the left and right repulsion forces, and applying the suggested movement, to put the window at its equilibrium position (while respecting the WPEx constraint).

Id	Src	Dst	Size	Period	Deadline	Jitter
A	H1	H5	120	800	800	2000
B	H2	H5	80	1600	400	2000
C	H3	H6	80	1600	400	1500
D	H1	H4	80	800	400	3000
E	H2	H4	80	400	400	3000
U	H1	H4	64	1600	1600	1600
X	H2	H5	96	800	1600	800
W	H2	H4	96	800	1600	800

Table 2 Characteristics of streams for the illustrative use case.

The outer loop computes the distribution of per window slacks. The user specifies a distribution granularity n (e.g. 32 or 64). Then, the hyperperiod H is cut n times into equals intervals $[0, \frac{H}{n}), [\frac{H}{n}, 2\frac{H}{n}), \dots [(n-1)\frac{H}{n}, H]$. The outer loop counts the number of windows having a slack in each interval. The outer loop stops once this distribution stabilizes, or when a user-defined bound is reached.

This distribution evolution will be illustrated in Section 7.1 and Table 3.

6 Reconfiguration

In our context, reconfiguration means removing and adding streams from the TAS schedule.

Removing a stream is free, due to the precedence relations and the protective gating. Since the schedule is robust to losses, it is also robust to a stream removal. In addition, thanks to the protective gating, the left bandwidth can be used by lower priority flows.

Adding a stream is quite straightforward. Once a frame is put in a window (and assuming that the routing tables have been updated to route this frame), the behaviour is deterministic: the frame will go from one window in a queue to the next window on the next queue along its path. Then, adding a stream of period T consists in adding $n = \frac{H}{T}$ frame instances, i.e. finding n windows, separated by T, such that putting a frame in a window will still respect the WPEx constraints, the deadline and jitter constraints of all streams, including the new one. This can be done by simple enumeration.

7 Use cases

7.1 Illustrative use case

Consider the simple topology of Figure 4 with the stream features of Table 2. Consider that the basic configuration only involves flows A, B, C, D, E. The solver computes the schedule depicted in Figure 12, and the post-processing phase leads to the schedule in Figure 13.

The evolution of slack distribution is represented in Table 3. The hyperperiod is 1600, the granularity is 32, then each interval has length 50. The table represents only the 10 first intervals, since the 12 last are empty (except the last interval, collecting windows that are alone on a link and whose slack is the hyperperiod). Before the first iteration (line "iteration 1"), there are 9 windows with a slack in $[0, 50)$, 1 with slack in $[100, 150)$ and 1 with slack in $[150, 200)$. This corresponds to Figure 12. Then, all windows are moved in sequence to an equilibrium place. Each individual movement maintains the WPEx constraints. After the first iteration, there are 4 windows with a slack in $[0, 50)$, 1 with slack in $[50, 100)$, etc. The

interval	0	1	2	3	4	5	6	7	8	9	10
iteration 1	9	0	1	1	0	0	0	0	0	0	0
iteration 2	4	1	1	4	1	0	0	0	0	0	0
iteration 3	3	1	3	2	0	1	0	0	1	0	0
iteration 4	2	1	3	3	0	1	0	0	0	0	1

Table 3 Evolution of slack distribution for the illustrative use case (10 first intervals represented).

loop stops after the fourth iteration. With this post-precessing, for example, the window containing C-0 on link H2->SW1 has been pushed to the left, whereas the window containing C-0 on link SW3->H5 has been pushed to the right.

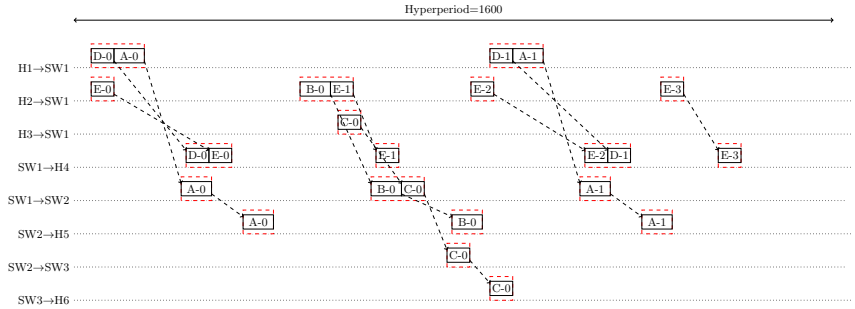

Figure 12 Schedule computed by the constraint solver for flows A,B,C,D,E from Table 2 on topology from Figure 4.

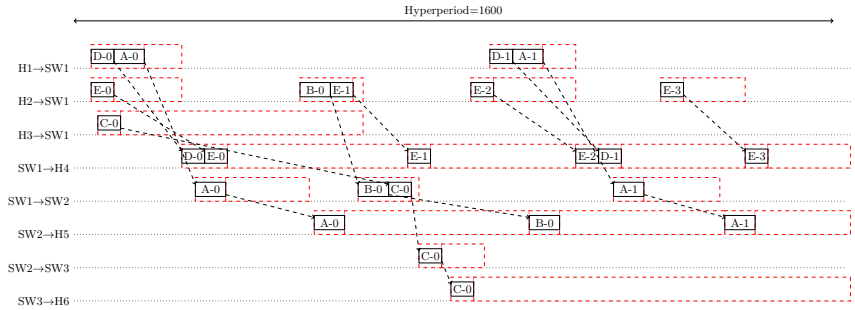

Figure 13 Schedule of Figure 12 after post-processing: windows spread by repulsion forces, and windows enlarged.

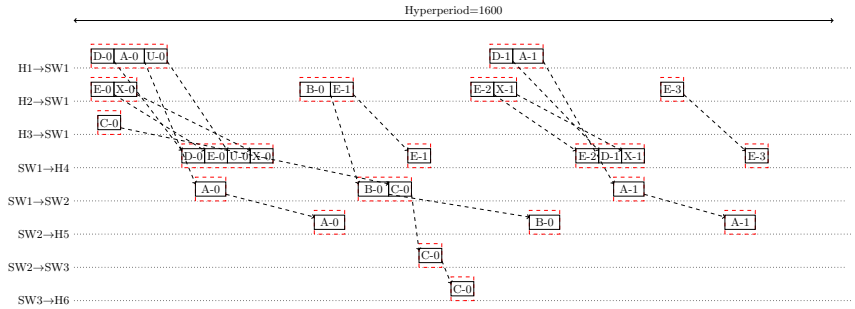

Figure 14 Addition of streams U, X, and W to the schedule of Figure 2.

Then, we try to add streams U, X and W, as presented in Section 6. The algorithm is able to place U, X and W, leading to the schedule of Figure 14.

7.2 ECRTS Industrial use case

The use case is the avionics TSN network proposed in the ECRTS industrial challenge [4]. The network is composed of 5 switches and 15 end-systems, connected by 1Gb/s links, as illustrated in Figure 15. It contains 241 streams, divided in 7 classes, whose main characteristics can be found in Table 4. The deadline and jitter constraints are given as ratio to the period. Classes #1 and #0 are best effort classes, and are not of interest in our context. The full list of streams (including paths) can be downloaded at ECRTS Industrial Challenge repository [1].[5]

The first part of the experiments consists in checking if the problem can be solved on a realistic use case. That is to say, is it possible to schedule such system while respecting the WPEx constraints? This is reported in Table 5. Column "scheduled classes" gives the list of classes that have been scheduled using the solver presented in Section 4. Column "WPEx sched. time" reports the time required to compute a solution. Column "WPEx window nb" reports the number of windows in the solutions. If a single number is provided, it means that this solution is optimal (it has a minimal number of windows). If two numbers are provided, the first number is the number of windows of the first solution found, and the second number is the number of windows found after 1h of computation. For example, it took 1.1s to find a schedule while considering only the streams of class #7, and this solution is optimal (it has a minimal number of windows). For scheduling streams of classes #6 and #7 together (both in the same queue), it took 810s to compute a solution, and this solution has 259 windows. After 1h of computation, the optimizer was able to find a solution with 218 windows. These computations have been executed on a dedicated server, with 16 Intel Xeon E5-2420 CPUs and 11Gb of memory.

Note that these computation times are comparable with state-of-the-art tools. Any precise comparison between other approaches and WPEx would be unfair, since they are not solving the same problem. Nevertheless, to get a raw intuition, we used the tsnkit tool [20]. This model implements 17 scheduling algorithms. We have selected the 7 efficient methods and run these methods on the same examples. Column "Mean tsnkit sched. time" reports the mean execution times (on a single laptop). The latter are reported only to show that the use case is neither trivial nor too complex, and that solving the WPEx problem seems to be on the same order of difficulty than the other TAS schedule classes.

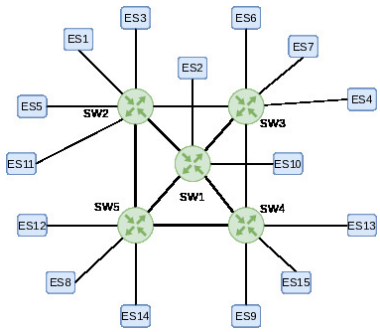

Figure 15 Topology of the use case, from [4].

[5] Note that the file is not fully compatible with the picture of the net provided in the description, cf. Figure 15. Some paths use links that do not exist on the picture. This has been signaled to authors, but the experiments have been done with commit f605151.

Class	Nb streams	Period (in µs) Min	Av	Max	Size (in bits) Min	Av	Max	Dead./Per.	Jit./Per.
#2	19	400	1,558	6,400	3,656	8,280	11,920	2	2
#3	20	400	1,280	6,400	2,152	7,564	11,760	2	2
#4	29	400	897	3,200	2,192	7,970	11,920	2	2
#5	45	400	889	3,200	3,528	7,933	11,744	1	1
#6	39	200	552	1,600	2,584	7,643	11,624	1	1
#7	32	200	406	800	2,800	6,710	11,920	.5	.2

Table 4 Number of streams, min/av/max period, min/av/max frame size, and ratio between deadline and period, and between jitter and period, for each class of the use case.

Scheduled classes	WPEx sched. time	WPEx window nb	Mean tsnkit sched. time	Inserting streams from #7	#6	#5	#4	#3	#2
#7	1.1s	75	27s	-	5	15	7	6	0
#6 and #7	810s	259 – 218	74s	-	-	17	21	1	1
#6	2.7s	150	29s	4	-	18	20	11	1
#5	5.4s	248	21s	0	9	-	25	12	1
#4	1.9s	204	7s	3	4	5	-	3	0
#3	1.5s	331	45s	1	0	2	0	-	0
#2	2.4s	363	37s	0	0	1	0	2	-

Table 5 Initial offline schedule computing times (on two hosts) and number of possible additional insertions. Insertion time are not reported, they all are between 10ms and 40ms.

We conducted additional experiments consisting in checking if our solution is incremental. Once a set of streams is scheduled, how many streams can be added without changing the GCL configuration? For evaluation, we have to pick up some new streams. The context of the ECRTS industrial challenge use case provided in [4] is reconfiguration after faults: when some links are broken, new routes are computed, and TAS windows must be allocated. But it makes the results dependent on the link choice, route choice, etc. I order to prove the incremental nature of the approach, we pick up some streams in the other classes, and see how many can be inserted. For example, the number 5 in column #6 of line #7 means that a schedule has been computed for the flows of class #7, and that 5 streams out of the 39 streams in class #6 can be added to the configuration without updating the GCL, and still respecting the jitter and deadline constraints of the streams of class #7 and the constraints of these 5 streams of class #6. The computation times are not reported for insertion, since they all are bellow 1s, on a laptop running python scripts.

Class #7, that has very strong jitter and deadline constraints (deadline and jitter smaller than period), is not able to host too many supplementary streams. Class #6 has implicit deadline and no jitter constraint. It is then able to host 18 streams of class #5 or 20 streams of class #4, or 11 streams of class #3, or 1 stream of class #2 or even 4 streams of class #7.

All the dataset can be downloaded with DOI `10.5281/zenodo.14906974`.

8 Conclusion

In this paper, we present WPEx-TAS, a solution to the ECRTS incdustrial challenge. WPEx-TAS is a novel schedule class to design resilient TAS configurations in TSN networks. In

fact, WPEx-TAS enables, in case of a reconfiguration at runtime, to safely add and remove flows in existing TAS windows without modifying the GCL in the output ports of nodes and switches or affecting other flows. This makes the TAS configuration robust to frames loss and reduces both time and costs for the recertification process, in case of reconfiguration. The WPEx-TAS schedule relies on the combination of three scheduling principles which are window non overlapping, protective gating, and maximal window enlargement. This allows creating a window-based schedule, with several frames and some slack per window. The slack can be dynamically used by lower priority flows when no critical flows are transmitted, thus ensuring that no bandwidth is wasted.

Our experiments conducted on the industrial challenge avionics use-case show that such a WPEx-TAS schedule can be built in an admissible time (computation times having the same order of magnitude as other approaches) and the insertion of new streams can be done is less than one second.

Further works may extend the computation of the initial schedule, adding the switch internal commutation time and per link bandwidth (instead of having the same bandwidth for all links): these are just new constants to add in the constraints. The generalization to several TAS queues (*multiple-TAS*) is also mainly a problem of notation. To ensure that TAS frame suffer no interference from other queues, a constraint stating that two TAS windows on the same link on different queues must not overlap must be added. And the enlargement process must also be done up to overlapping of TAS windows on the same link.

One limitation of the current approach is that a link with used by a small number of flows may have a very small number of windows. Then, it may not be able to accept new flows, whereas its load it small. One solution may be to add fake windows (i.e. windows of null size) to prepare the possible addition of stream in such links during post-processing.

References

1 Repositories related to the industrial challenge of the euromicro conference on real-time systems. URL: `https://github.com/ecrtsorg/`.

2 Mohammadreza Barzegaran, Niklas Reusch, Luxi Zhao, Silviu S Craciunas, and Paul Pop. Real-time traffic guarantees in heterogeneous time-sensitive networks. In *Proc. of the 30th International Conference on Real-Time Networks and Systems (RTNS 2022)*, Paris, France, 2022.

3 Lucia Lo Bello and Wilfried Steiner. A perspective on IEEE time-sensitive networking for industrial communication and automation systems. *Proceedings of the IEEE*, 107(6):1094–1120, 2019.

4 Marc Boyer and Rafik Henia. Industrial challenge: Embedded reconfiguration of TSN. working paper or preprint, July 2024. URL: `https://hal.science/hal-04630862`.

5 Pierre-Julien Chaine and Marc Boyer. Shortening gate closing time to limit bandwidth waste when implementing time-triggered scheduling in TAS/TSN. In *Proc. of the 15th Junior Researcher Workshop on Real-Time Computing*, Paris, France, 2022. URL: `https://rtns2022.inria.fr/files/2022/06/proceedings_jrwrtc2022_final.pdf`.

6 Pierre-Julien Chaine, Marc Boyer, Claire Pagetti, and Franck Wartel. Egress-TT configurations for TSN networks. In *Proc. of the 30th International Conference on Real-Time Networks and Systems (RTNS 2022)*, Paris, June 2022.

7 Silviu S. Craciunas, Ramon Serna Oliver, Martin Chmelík, and Wilfried Steiner. Scheduling real-time communication in IEEE 802.1Qbv time sensitive networks. In *Proc. of the 24th Int. Conf. on Real-Time Networks and Systems (RTNS'16)*, RTNS'16, pages 183–192, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2997465.2997470`.

8 Hugo Daigmorte and Marc Boyer. Impact on credit freeze before gate closing in CBS and GCL integration into tsn. In *Proc. of the 27th Int. Conf. on Real-Time Networks and Systems (RTNS 2019)*, Toulouse, France, November 2019.

9 Frank Dürr and Naresh Ganesh Nayak. No-wait packet scheduling for ieee time-sensitive networks (tsn). In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, page 203–212, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2997465.2997494.

10 Zhiwei Feng, Zonghua Gu, Haichuan Yu, Qingxu Deng, and Linwei Niu. Online rerouting and rescheduling of time-triggered flows for fault tolerance in time-sensitive networking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4253–4264, 2022.

11 Christoph Gärtner, Amr Rizk, Boris Koldehofe, René Guillaume, Ralf Kundel, and Ralf Steinmetz. On the incremental reconfiguration of time-sensitive networks at runtime. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022. doi:10.23919/IFIPNetworking55013.2022.9829815.

12 IBM Corporation. *IBM ILOG CPLEX Optimization Studio – Functions*, 2024. URL: https://www.ibm.com/docs/en/icos/22.1.2?topic=manual-functions.

13 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23(2):210–250, 2018.

14 Naresh Ganesh Nayak, Frank Dürr, and Kurt Rothermel. Incremental flow scheduling and routing in time-sensitive software-defined networks. *IEEE Transactions on Industrial Informatics*, 14(5):2066–2075, 2018. doi:10.1109/TII.2017.2782235.

15 Ramon Serna Oliver, Silviu S Craciunas, and Wilfried Steiner. IEEE 802.1 Qbv gate control list synthesis using array theory encoding. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–24. IEEE, 2018.

16 W. Steiner, G. Bauer, B. Hall, M. Paulitsch, and S. Varadarajan. TTEthernet dataflow concept. In *Proc. of Eighth IEEE International Symposium on Network Computing and Applications (NCA 2009)*, pages 319 –322, 2009. doi:10.1109/NCA.2009.28.

17 Thomas Stüber, Lukas Osswald, Steffen Lindner, and Michael Menth. A survey of scheduling algorithms for the time-aware shaper in time-sensitive networking (TSN). *IEEE Access*, 11:61192–61233, 2023. doi:10.1109/ACCESS.2023.3286370.

18 Ammad Ali Syed, Serkan Ayaz, Tim Leinmüller, and Madhu Chandra. Dynamic scheduling and routing for tsn based in-vehicle networks. In *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6, 2021. doi:10.1109/ICCWorkshops50388.2021.9473810.

19 Ye Wang, Fushen Wang, Wei Wang, Xiaowen Tan, Jiachen Wen, Ying Wang, and Peng Lin. Design and implementation of traffic scheduling algorithm for time-sensitive network. In *2022 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–6, 2022. doi:10.1109/BMSB55706.2022.9828694.

20 Chuanyu Xue, Tianyu Zhang, Yuanbin Zhou, Mark Nixon, Andrew Loveless, and Song Han. Real-time scheduling for 802.1Qbv time-sensitive networking (tsn): A systematic review and experimental study. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 108–121, 2024. doi:10.1109/RTAS61025.2024.00017.