©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

On the Incremental Reconfiguration of Time-sensitive Networks at Runtime

Christoph Gärtner*, Amr Rizk¶, Boris Koldehofe‡, René Guillaume§, Ralf Kundel* and Ralf Steinmetz*
*Technical University of Darmstadt, Germany, {firstname.lastname}@kom.tu-darmstadt.de
¶University of Duisburg-Essen, Germany, amr.rizk@uni-due.de
‡University of Groningen, Netherlands, boris.koldehofe@rug.nl
§Robert Bosch GmbH, Corporate Research, rene.guillaume@de.bosch.com

***Abstract*—Static configurations in Time-sensitive Networking (TSN) using the Time-aware Shaper allow precise calculations of deterministic, tight bandwidth and latency guarantees for real-time industrial application streams. It is, however, this static configuration which makes introducing flexible changes to a running TSN system at runtime very hard. Scenarios of adaptive TSN networks envision that the network configuration evolves with time in accordance to anticipated changes such as the dynamicity of machine formations and machine reconfigurations.**

In this paper, we propose a notion of flexibility of scheduler configurations along a network path that facilitates introducing changes to TSN network configurations at runtime. Based on this notion, we develop and analyze algorithms to incrementally reconfigure TSN using the Time-Aware Shaper. These reconfigurations include determining the admissibility of new or changed streams that may possess individual deadlines.

I. Introduction

Time-sensitive Networking (TSN) allows the deployment of real-time industrial applications with bandwidth and latency guarantees on top of a converged and centrally controlled Ethernet-based infrastructure [1]. To support strict real-time requirements, TSN is endowed with the IEEE 802.1Qbv Time-Aware Shaper (TAS), allowing to deploy scheduled traffic, i.e., streams with strict timing requirements to allow synchronized execution of tasks being distributed across the network. To meet the strict real-time requirements, TAS enforces time-division multiplexed egress ports at switches through so-called Gate Control Lists (GCL), that control precomputed cyclic schedules on a number of queues per output port. Essentially, these schedules determine a mapping of data streams (more precisely queues at the output port) to transmission time points, i.e., the *gate opening times* when the first packet in a queue is allowed to be transmitted.

The main drawback of this model is the static mode of this configuration and deployment. TAS configuration, i.e., egress port schedules, can be computed using a number of methods based on constraint-programming, such as using SMT-solvers or ILP formulations [2]–[4]. Given these configurations, bandwidth and latency guarantees can be calculated using analytical tools for real-time network performance evaluation such as network calculus [5]. However, we postulate that this static model is not suitable for future industrial scenarios, e.g., manufacturing pipelines. These are expected to benefit from flexibility, in particular, the dynamic connecting and disbanding of physical components, e.g., machines of the manufacturing process [6]. This enables context-specific tasks on top of TSN applications. Currently, the required dynamicity is pre-planned before deployment and results in under-utilization of statically reserved network resources.

To summarize, the need for dynamicity comes from adding, removing, or changing machine tasks flexibly at runtime. While the standard approach would require to recalculate and redeploy a global configuration, that potentially requires interrupting the running system, we postulate in this work that this is not necessary. We show in this work how to obtain incrementally extended schedules that still adhere to bandwidth and latency guarantees by leveraging a formulation for the flexibility of deployed schedules.

An important concept in the context of this paper is the *flexcurve* introduced in [7]. A flexcurve is a notion of schedule flexibility for a given path in a network. It indicates the number of feasible arrangements within the bottleneck schedule along the given path. This way, schedulers can create schedules or select suitable paths, which maximizes the ability to incorporate future changes. This proposed flexibility notion can give this information for streams of arbitrary sizes, is however, limited in fixed cycle periods and undefined deadlines. We adopt this notion and enhance it by providing optional deadline awareness and optimizations for the creation and incremental update support, that arise out of real-world deployment requirements.

Equipped with this formulation for path flexibility, we summarize the contributions of this work as follows:

- We show that the computation time of incremental schedules using this flexibility-based approach is far less than the standard recalculation methods.
- We extend the concept of the flexcurve to include streams that possess different latency deadlines.
- We present and analyze a search algorithm to find stream embeddings in switch schedules along a path that adhere to the stream specific deadlines and minimize the end-to-end stream latency.

ISBN 978-3-903176-48-5© 2022 IFIP

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

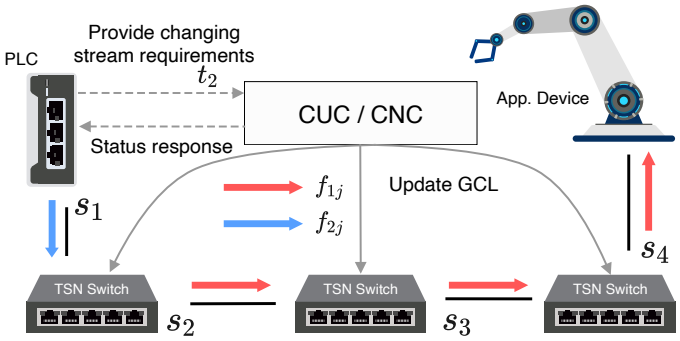

Fig. 1. A centralized controller (CUC/CNC) is responsible for managing the stream deployment and updating gate control list entries in the network. Example with two tasks.

II. System Model and Problem Statement

A. TSN System Model

We consider a time-sensitive network which schedules a set of dynamic *tasks* consisting of packet *streams* of equal priority with zero jitter requirements. Tasks t_v with $v \in [l]$ are temporally composed in a directed acyclic graph representing a network application. We use the symbol $[l]$ to denote the set $\{1, ..., l\}$. Each task t_v is equipped with a duration $T_v \in \mathbb{R}_+$. The task model mimics process models known for example from [8]. An example of such a network application is sketched in Fig 1. A static network application is given through a set of tasks $\{t_v\}_{v\in[l]}$ for a fixed number of tasks l. We define a dynamic network application to contain a set of tasks $\{t_v\}_{v\in[l]}$ with variable $l(t)$ that is non-decreasing over time, i.e., the network application may include additional tasks at runtime in case new requirements arise. Each task t_v may contain periodic real-time traffic streams. We assume that a task t_v requires a set of n streams $\{f_{vj}\}_{j\in[n]}$ to be simultaneously deployed in the network. A single stream consists of network traffic with application specific real-time requirements between a pair of TSN end-devices.

We use the above definition of network streams to define network paths that are comprised of a set of $m \in \mathbb{N}$ egress ports that are used by a stream on a network path. Each port is equipped with an active Time-Aware Shaper (TAS) mechanism and up to eight separate queues. We also assume time synchronization of all participating network devices as required by TAS. Each port has an egress schedule denoted s_i with $i \in [m]$. A schedule is an ordered sequence of time slots s_i at which a stream's packets are sent from a specific queue. Every port schedule s_i is calculated in a way that all stream requirements are met. Among such requirements is the cycle-period, data-size, deadline and the stream's path. Recall, that a path for stream f_{vj}, i.e., the jth stream of task v, consists of a sequence of m TSN switch ports $P = (p_i)_{i\in[m]}$ that are utilized by this stream.

Since the size of the stream packets is known and fixed, the schedule is translated to open and close instructions for the gate control list of the Time-Aware Shaper. Note that, time slots in the schedule are relative to the network cycle period of K, often given by the least common multiple period of all present streams. In other words, each schedule repeats every K slots and the schedules of all TSN switches in the network are time synchronized.

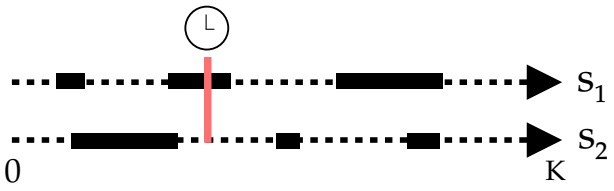

Fig. 2. Schedule execution is time synchronized between each participating network device.

Since time granularity is finite in any practical TSN system, we assume time slotted scheduling in the following. We consider a cycle period of K slots $j \in [K]$ with

$$s_i(j) := \left\{ \begin{array}{lcl} 1 & : & \text{slot is free} \\ 0 & : & \text{slot is taken} \end{array} \right\}$$

which indicates the occupancy of slot j in schedule s_i. The kth successor slot $s_i(j+k \bmod K)$ is denoted by $\text{succ}(k, s_i(j))$ with $\text{succ}(0, s_i(j)) = s_i(j)$.

We call a stream f of size c slots admissible at the jth slot of schedule $s_i(j)$ if $\forall k < c$ it holds

$$\sum_{k=0}^{c-1} \text{succ}(k, s_i(j)) = c, \tag{1}$$

i.e., there are at least c contiguous free slots from slot j to accommodate the stream with period K. The stream f is admissible in schedule s_i if $\exists j \in [K]$ for which (1) holds.

A stream f of size c slots is said to be admissible on a path $P = (p_i)_{i\in[m]}$ consisting of m ports if there exists at least one sequence of time points $A = (a_i)_{i\in[m]}$ where f is admissible at the corresponding schedule slots $(s_i(a_i))_{i\in[m]}$.

We define the delay between the time points $s_i(a_i)$ and $s_{i+1}(a_{i+1})$ at two consecutive ports $(i, i+1)$ as

$$d(a_i, a_{i+1}) = \left\{ \begin{array}{lcl} a_{i+1} - a_i & : & a_{i+1} \geq a_i + c \\ K + a_i - a_{i+1} & : & \text{otherwise} \end{array} \right\} \tag{2}$$

Note that (2) is based on our counting convention (cf. Fig. 2) that the schedule start (in absolute time) of port schedules along a path is synchronized. Now, we can obtain the end-to-end delay for a stream along a path with assignment A as

$$D(A) = c + \sum_{i=1}^{m-1} d(a_i, a_{i+1}) \tag{3}$$

B. Flexcurve-based Scheduling and Problem Description

Dynamic TSN configurations for dynamic TSN applications are hard. Each change requires the introduction of new streams and the removal of obsolete ones in the network. Note, each update consumes significant computational overhead at the controller (cf. Fig 1) in charge of updating the real-time schedule. In particular, this overhead is caused by state-of-the-art methods, generating schedules for given streams

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

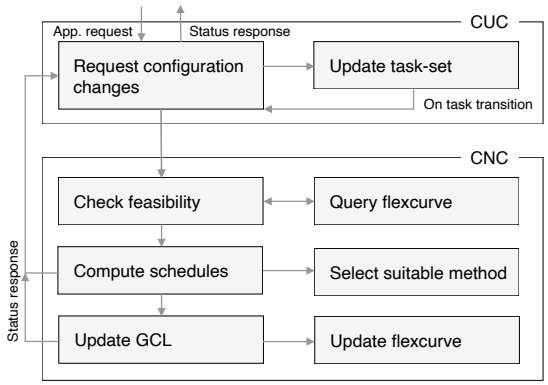

Fig. 3. Control flow of CUC and CNC for admitting new streams leveraging flexcurves.

without considering future changes. These standard scheduling methods take a significant amount of time to re-compute if a new stream is added. The computational cost for the controller can be reduced by improved update mechanisms, that can either adapt the schedule incrementally with regards to future changes and avoid entire schedule re-calculations.

Now, consider a centralized configuration model for a TSN network, including Centralized User Configuration (CUC) units and one Centralized Network Controller (CNC) as depicted in Fig. 3 which shows the control flow for flexible TSN reconfiguration at runtime. Incoming streams designated for embedding are either requested online or are queued in advance at the CUC (App. request). The CUC acts as an arbiter between requesting real-time applications and the network configuration, which is handled by the CNC. The CNC returns to the CUC whether an embedding is possible and may also recommend the order of stream embeddings for the current network situation. A tool to generate these recommendations is the *flexcurve*, which provides the current path flexibility state in regards to embedding additional streams.

The quantification of the flexibility of switch configurations, i.e., of port schedules, was first attempted in [7]. This flexibility measure, denoted as *flexcurve*, describes the number of stream embeddings along a network path as a function of the stream size c. For a given path P, described as a concatenation of port schedules, the flexcurve is given as

$$h(c) = \min_{k \in P} \sum_{\tau=0}^{K-c} 1_{\{C_k(c+\tau) - C_k(\tau) = c\}}. \tag{4}$$

where $C_k(n)$ is a non-decreasing function that accumulates the free capacity along the schedule of port k up to the n_{th} time slot. Hence, for a path consisting of one port with an empty schedule $h(c)$ in Equation (4) simply resembles a decreasing staircase function starting from the number of time slots in the schedule. In general, given a stream of size c then the value of the flexcurve $h(c)$ resembles the number of possible embeddings along the path P. Note that the considered streams are periodic and require *contiguous* embedding into the port schedule. An application of this flexibility measure is to select a path in the TSN which maximizes $h(c)$ and this way yields highest flexibility for future changes.

In light of the work at hand, we observe that a further benefit of the formulation above is that it can be adapted under online schedule reconfigurations. However, to allow seamless schedule reconfigurations we require fast initialization of (4) and efficient updates. While the approach in [7] is promising, it is limited to a definition of real-time streams, that does not extend to stream variable properties such as delay deadlines, queuing delays and cycle-period bounds.

In the following sections, we enable fast dynamic TSN reconfigurations by constructing schedule flexcurves that lend themselves to rapid initialization using a subset lookup and allow for fast incremental updates after stream admission. Further, we develop a formulation of the flexibility of TAS schedules for network paths that include streams with differentiated deadlines and, finally, analyze the admission problem in (1) to find TAS schedules to adhere to stream delay deadlines while minimizing the flexibility impairment.

III. Query and Build Optimization for TSN Flexcurves

Given the formulation of the flexcurve in (4), we observe that the possibility of entire precomputation before querying information about specific paths of interest is a major benefit. The formulation can be computed asynchronously as it does not depend on the number of streams currently present along each path, given each port's egress-schedule. However, generating the flexcurve itself depends on the number of slots and the free slots within the hyperperiod, which are directly dependent on the duration and the time granularity the scheduler uses (e.g. with nanosecond granularity a 1ms period results in 10^6 slots). Calculating (4) in this way, considering the hyperperiod K as variable parameter, results in a computational complexity of $\mathcal{O}(mK)$ for a path P of length m, as we need to check each schedule along the path and calculate the number of free-slots for each slot in the schedule. To build the complete flexcurve, we apply (4) K times, thus resulting in a time complexity of $\mathcal{O}(mK^2)$. Now, we note that this computationally heavy generation of the flexcurve for every stream size may not be required as only a few stream sizes c are interesting for future admissions. An optimized approach, where single values of (4) are more efficiently computed as required is hence desirable.

To enable a fast computation of flexcurve values and partial updates after stream admission, we leverage the schedule's data structure. A schedule s_i is in essence an ordered list of time-points when specific streams are scheduled to egress. Knowing the hyperperiod K, we can linearly transform this schedule to also encode contiguous free slots — the basis for computing the flexcurve (see $C_k(c+\tau) - C_k(\tau)$ in (4)). We denote the sequence of gaps of schedule s_i as $(g_{i,j}, \Delta_{i,j})_j$ where the jth gap has a starting time $g_{i,j}$ and duration $\Delta_{i,j}$. We note that the number of gaps in a schedule is usually much smaller than the number of slots. Next, we use the gaps to generate an optimized version of the flexcurve that leads to a strong reduction of the initial computation time compared to computing (4).

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

Given a port schedule, each gap in that schedule s_i encodes a so-called gap-local flexcurve, i.e., a staircase function of the form

$$h_{i,j}(n) = (\Delta_{i,j} - n + 1)1_{\{n \leq \Delta_{i,j}\}} \tag{5}$$

with the indicator function $1_{\{\cdot\}}$. The accumulation of all gap-local flexcurves gives the so-called port-local flexcurve of the corresponding port schedule s_i as $\hat{h}_i(n) = \sum_j h_{i,j}(n)$. An illustration is given in Fig. 4 where the schedule of s_1 contains four gap-local flexcurves $h_{1,j}(n)$ (in blue). To obtain the flexcurve of a network path, the port-local flexcurve for each port schedule along the path must be computed. Note that the generation of port-local flexcurves, with gap-local intermediate representations, has a worst-case time complexity of $\mathcal{O}(mK)$ for a path P of length m. As required gaps $(g_{i,j}, \Delta_{i,j})_j$ can be found by iterating once over each element of s_i. Note that the worst case of the number of entries $\mathcal{O}(K)$ per schedule differs, however, significantly from the expected case as the number of entries in a schedule s_i can be estimated by $\frac{1}{E[c_f]} \sum_f c_f$ where c_f is the size of stream f and its expectation is $E[c_f]$. Note the difference in runtime and variables between this approach and (4). The initialization of this approach works on the higher level schedule entries, and is *free* if a schedule's data structure supports gap information inherently, whereas (4) always works directly at the fine grained slot level.

In addition to the reduction of the time complexity of the computation, this initialization procedure saves a significant amount of memory. As the number of slots is usually much larger than the number of gaps in a schedule. This procedure does not precompute flexcurve values for each stream size, but rather stores the gap sequences $(g_{i,j}, \Delta_{i,j})_j$ per schedule.

A. Computing initial flexcurve values

We can now compute the flexcurve of a path P that utilizes the gap-local flexcurves $h_{i,j}$ of the schedules s_i from (5) as

$$h_P(n) = \min_{i \in P} \sum_j h_{i,j}(n) \tag{6}$$

where P is the path of traversed ports as defined in Sect. II-A. For a queried stream size n we obtain the value of $h_P(n)$ by calculating the value of the port-local flexcurve as the sum the gap-local flexcurves at n and returning the minimum value over the port-local flexcurve. Note that we deliberately express (6) in terms of the gap-local flexcurves $h_{i,j}$ instead of the port-local flexcurve $\hat{h}_i$ as we will allow a simple incremental update procedure. To this end, we also keep the realization of this procedure based on $h_{i,j}$, i.e., we keep at each port a list of gap-local flexcurves. The time complexity of querying this optimized flexcurve data structure is $\mathcal{O}(j)$ for a path P with j number of gaps along the path.

B. Incremental schedule updates

One benefit when using gap-local flexcurves as the underlying data structure to calculate flexcurve values is the support of direct incremental updates. In contrast to the formulation (4) this can now be achieved without the need to update values outside of impacted gaps in the schedule. To update an affected port-local flexcurve, we need to distinguish two cases:

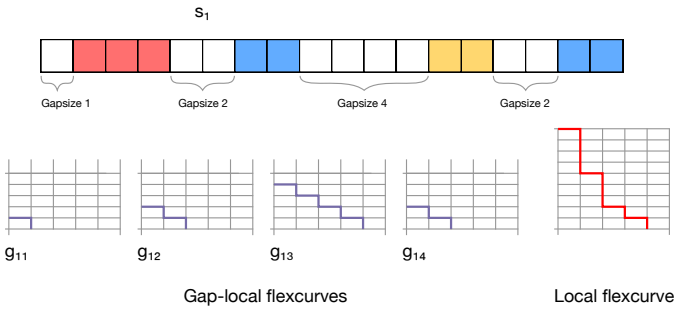

Fig. 4. Port-local flexcurve (red) built from gap-local flexcurves (blue). This enables fast generation of the required flexcurve data structure to enable a quick query of specific values $h(n)$ of (4).

A stream admission results in occupying free schedule slots equal to the stream size c. Recall that we consider streams that are embedded within one gap in the schedule. This is the key to incremental schedule and flexcurve updates: To update the port-local flexcurve, the gap-local flexcurve of the affected gap is diminished or removed. There exists three possibilities for contiguous stream embedding here: (i) $\Delta_{i,j} = c$, i.e., the gap is filled by the new stream, (ii) $\Delta_{i,j} > c$ and the stream is embedded at the beginning or end of the gap, i.e., the gap size is diminished as $\Delta_{i,j} \leftarrow \Delta_{i,j} - c$ or finally (iii) if the stream is embedded in the middle of the gap, i.e., we obtain two gaps $(g_{i,j}, \Delta^1_{i,j})$ and $(g_{i,j+1}, \Delta^2_{i,j})$ with $\Delta^1_{i,j} + \Delta^2_{i,j} = \Delta_{i,j} - c$ instead of the original gap $(g_{i,j}, \Delta_{i,j})$. The resulting $h_{i,j}$ for the new gap(s) are simply added to the list of gap-local flexcurves at the corresponding port. Affected ports will therefore have an updated local flexcurve. The flexcurve from (4) can then either be restored using (6), or by utilizing the updated port-local flexcurves $\hat{h}_i$ after the port was affected using

$$h'_P(n) = \min\left\{h_p(n), \hat{h}_i(n)\right\} \tag{7}$$

resulting in h'_P as an exact up-to-date flexcurve of the path P.

Updating the port-local flexcurves when removing streams is more involved than the addition of streams as described above. In fact, stream removal may either (i) create a new gap if the stream to be removed is not adjacent to any existing gap or (ii) extend a gap, $\Delta_{i,j} \leftarrow \Delta_{i,j} + c$ i.e., for a stream of size c that is adjacent to exactly one gap. In case that the removed stream of size c was adjacent to two gaps $(g_{i,j}, \Delta_{i,j}), (g_{i,j+1}, \Delta_{i,j+1})$ those gaps are removed and we obtain one new gap $(g_{i,j}, \Delta_{i,j} + \Delta_{i,j+1} + c)$ The flexcurve of the path $h_P(n)$ can then be incrementally updated as in (7).

IV. Deadline Awareness

One of the major limitations of the flexcurve as defined in (4) is the lack of stream-specific deadline differentiation in its representation. In this section, we resolve this limitation, but still only consider streams with cycle periods equal to the hyperperiod K. As discussed in Sect. II, the control flow of CUC and CNC for stream admission control takes a stream description together with a query of the flexibility measure of the running schedules (flexcurves) to compute the stream

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

embedding and to update the GCL and the flexcurves. Now, we consider stream requirements that include a stream-specific delay deadline of utmost $(m-1)K$, with m denoting the length of the selected path. This is the worst-case delay when scheduling the stream at least once at every hop along the path P. We define the scheduled stream delay as the time-span from first scheduled time, to the last scheduled time of a stream, i.e., the time a stream is traversing on the network path. The scheduled delay may not exceed the stream's deadline and is therefore dependent on the actual stream-embedding.

For a stream of size n with a delay deadline d, a deadline aware flexcurve $h_p^d(n,d)$ considers *only* stream-embeddings where the stream-deadline is not violated. Hence this limits the flexibility of a network path towards admitting this stream. The rationale behind this is that a CNC would calculate many possible embeddings for the admission of a stream if it does not possess a delay deadline while having a delay deadline may restrict the number of possible embeddings for stream admission making network paths seem less flexible for certain streams. In this sense, we argue that the flexibility of a network path (that is captured by the flexcurve, i.e., counts of possible stream embeddings in the port schedules along that path) is a property that is stream specific. Hence, we write for the flexcurve $h_P^d(n,d)$ of a network path P in combination with a delay deadline d along that path that

$$\begin{aligned} &h_P^d(n,d) \leq h_p(n) \\ &h_P^d(n,d) = h_P(n) \text{ if } d \geq (m-1)K \end{aligned} \tag{8}$$

i.e. any value of the deadline aware flexcurve is smaller or equal to that of the flexcurve (4).

A. Minimum Latency Stream Admission

The delay of a newly admitted stream depends on the port schedules s_i along the designated network path. We assume that the path is previously determined. With the usage of classical TAS scheduling approaches, e.g. with Satisfiability Modulo Theories (SMT) solvers, it's possible to compute an embedding that satisfies a stream's deadline requirements. However, using SMT [2] or similar approaches requires significant computing time. In the following, we present a search algorithm that is able to embed a single stream f of size c slots and cycle period K with the smallest possible network delay in linear runtime at the worst-case. We will use intermediary results of the algorithm to generate a deadline-aware flexcurve.

Applying a first-fit heuristic for stream admission allows to quickly find a stream embedding, i.e., for single streams, it returns an embedding or it fails. Similarly, the admissibility of a stream of size c can be determined by looking up the value of the flexcurve for the corresponding stream size [1]. The resulting embedding of the first-fit heuristic gives the minimal achievable latency of the first viable starting position in the schedule s_1 of the first port along the path. However, the resulting delay may still violate the stream's deadline requirements.

[1]Note that the flexcurve formulation (4) enables simultaneously calculating admissibility for multiple streams.

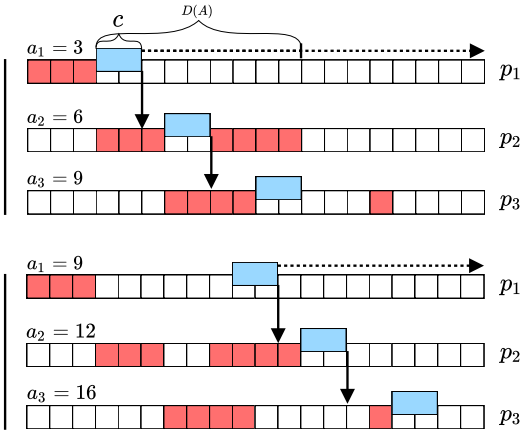

Fig. 5. Example for two states of Alg. 1 with a shift of the time point a_1 that is the start of the initial assignment to the schedule s_1 at port p_1. Red slots are occupied by other streams.

Other starting positions in s_1 might provide lower delays to keep the deadline. Repeating the first-fit heuristic for each viable starting position answers this question of finding the stream embedding with minimal latency, however it results in quadratic runtime, given that the first-fit heuristic can be implemented linearly.

Instead of repeatedly applying the first-fit heuristic, we propose a search algorithm (Alg. 1) that is able find an embedding sequence of time points $A = (a_1, \ldots, a_m)$ for stream f with the lowest possible delay for any starting time in the schedule s_1 of the first port along the designated path P. The algorithm can terminate earlier if an embedding was found that satisfies the deadline requirements.

In Fig. 5, we depict two states of the algorithm. Given a path P as a sequence of m distinct ports, the algorithm works by shifting the embedded starting position of the first schedule a_1 by one slot in each step assuming the resulting embedding is valid if $\sum_{k=0}^{c-1} \text{succ}(a_1+1+k, s_1(a_1)) = c$. Here the algorithm finds a concrete embedding $\forall j$ in (1) of the first schedule s_1. This ensures latency improvements are not missed in the search due to the starting time of f within the first schedule s_1.

With each shift of a_1 to the right, the algorithm checks for a_i, $i = 2, \ldots, m$ the current stream embedding falls behind their predecessor's a_{i-1} position (indicated by the arrow position pointing to the next hop). Should this be the case, the new scheduled position of each a_i is set to the next viable position. Setting the next viable position efficiently is done by leveraging contiguous gaps $(g_{i,j}, \Delta_{i,j})$ in each port schedule as introduced in Section III. When shifting any new stream embedding, the algorithm also keeps track of which gap within the sequence is occupied. When a certain placement is not possible, the new stream position a_i is set to the starting position $g_{i,j}$ of the next gap in the sequence.

Applying this algorithm allows finding a viable stream embedding for any required deadline up to the stream delay deadline. To simplify the description and implementation of Alg. 1, we copy and append each schedule along the path mul-

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

tiple times, i.e., we virtually extend the number of slots to Km while periodically repeating the initial stream assignments. This allows disregarding assignment wrapping to the relative start of a schedule, thus allowing for simpler comparisons, and also a simplified end-to-end latency calculation as $a_m - a_1 + c$.

Since the algorithm Alg. 1 progresses to later slots, and never revisits a slot already checked, the algorithm has a worst-case runtime of $\mathcal{O}(Km)$. Here, the maximum number of slots is given by Km, with a period of K and the number of ports (schedules) on the path m. Preprocessing, i.e., creating the sequence of gaps has a linear worst-case runtime, as it is only required to iterate over a given sorted list of schedule entries. Note that the path length m is in most practical settings upper bounded by a small number, usually smaller than 10.

B. Time-Aware Shaper Configuration

While Alg. 1 can find a valid embedding within the given port schedules, the resulting schedule may be infeasible for an immediate deployment with a TAS-capable device. The reason for this lies in the fact that TAS works by following the gate control list of each output port to open and close priority queues for the egress. If packets are scheduled with a no-wait-constraint one queue is sufficient for scheduling as arriving packets are immediately processed and forwarded to the next hop. In our case, Alg. 1 finds suitable embeddings which may require the packet to queue for a finite and predetermined amount of time before the scheduled time arrives, while still respecting any deadline requirements of the stream. This waiting capability requires streams to be isolated from each other, either by arrival time or by sorting into different queues.

C. Deadline Aware Flexcurve

We recall that by definition the flexcurve describes the count of arrangements for the given path of the queried stream of size c at the corresponding bottleneck schedule. Since each state of the time point sequence A is a valid stream embedding, we can use it to generate a flexcurve as well. Saving snapshots of A, if the end-to-end delay is below the deadline, after each shift of a_1 in Alg. 1 results in a set of one or multiple possible embeddings. Counting the number of distinct slot assignments and returning the bottleneck count, i.e., the minimum count of assignments along the path $p_1, \ldots, p_m$, constructs the *deadline aware* flexcurve. One may be tempted to assume that this corresponds to the flexcurve value for a stream of size c that possess the deadline d, however, this is only a necessary condition. We note that saving snapshots of the embedding search algorithm is not sufficient to satisfy the constraint of the deadline aware flexcurve (8). This may result in underestimating the flexcurve values as some stream embeddings of intermediate schedules are not counted. These embeddings do not improve the latency so they are not guaranteed to appear in the time point sequence A.

To extend the search algorithm Alg. 1 to include intermediate embeddings not necessarily returned by the search algorithm, we need to shift all embeddings from $a_2, \ldots, a_m$ after each shift of a_1 to the maximum delay allowed by the deadline. This results finding all valid stream embeddings up to the deadline. A snapshot of each embedding *precisely* gives the number of valid assignments for each schedule along the path. The minimum number of valid assignments corresponds to that of the deadline aware flexcurve value $h_P^d(n, d)$.

Algorithm 1:

Data: g, Δ, $P = (p_1, ..., p_m)$, c, deadline;
Result: A/False: Embedding/Stream is inadmissible
// Initial fitting of stream to free slots
// with extended address space
$A = (a_1, ..., a_m)$, gap$(a_1, ..., a_m) \leftarrow$ InitialFit(Open_Slots);
if $a_m - a_1 + c \leq$ *deadline* **then**
  return A;
while $a_1 < K$ **do**
  // Shift stream of starting schedule
  $a_1 \leftarrow a_1 + 1$;
  $j \leftarrow$ gap(a_1);
  // Check if current gap is large enough
  if $g_{p_1,j} + \Delta_{p_1,j} - a_1 < 0$ **then**
    // Otherwise, skip to next viable gap
    gap$(a_1) \leftarrow j + 1$;
    if *gap*(a_1) *is invalid* **then**
      return False;
    $a_1 \leftarrow g_{s_{p_1},j+1}$;
  for i *in* $2...m$ **do**
    if $a_i < a_{i-1} + c$ **then**
      $a_i \leftarrow a_{i-1} + c$;
      $j \leftarrow$ gap(a_i);
      if $g_{p_i,j} + \Delta_{p_i,j} - a_i < 0$ **then**
        gap$(a_i) \leftarrow j + 1$;
        if *gap*(a_i) *is invalid* **then**
          return False;
        $a_i \leftarrow g_{p_i,j+1}$;
  delay $\leftarrow a_m - a_1 + c$;
  // We can save snapshots of the current
  assignment A here if the calculated delay
  adheres to deadline or return directly:
  if *delay* $\leq$ *deadline* **then**
    return A;
return False;

V. Evaluation

In the following, we split our evaluation into several distinct parts: First, we look at the runtime performance of incrementally embedding streams, then we evaluate the lookup time of the flexcurve presented in Sect. III. Further, we show how different scheduling algorithms affect the path flexibility before, finally, highlighting the impact of multiple variables on the computation runtime for deadline-aware flexcurves.

For all evaluations, we use two different topologies. First, a basic line topology consisting of 4 hops, *i.e.*, 4 port schedules. Streams are embedded along the entire line. Secondly, we consider a more complex machine topology (cf. Fig. 6), consisting of three switch-hierarchies. The chosen topology consists of three sections, three subsections with six leaf nodes each. Each node is a time-aware network device or switch. All ports are uniquely numbered. Streams are set between one single controller situated at a leaf node and all other leaf nodes and back resulting in a total of 106 streams.

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

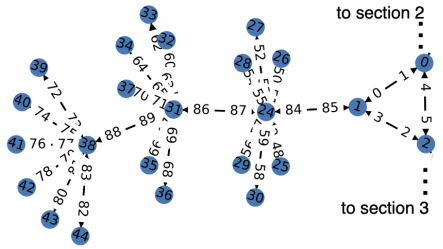

Fig. 6. One section of the evaluated topology of a complex machine network. The topology consists of three identical sections (only one discplayed here). A leaf node acting as a controller is exchanging traffic with all other leaf nodes.

Table (I) lists the used stream properties. The line rate is set globally to 100 Mbps and time-slots are allocated with μs granularity. We assign the stream sizes for the machine topology uniformly at random between 60 to 300 bytes.

TABLE I
STREAMS USED FOR EVALUATION.

Topology	Period	Deadline	Size	Avg. Pkt Size
Line	1.0 ms	1.0 ms	100 byte	100 byte
Machine	1.0 ms	0.5 ms	60 byte–300 byte	180 byte

A. Incremental stream admission and embedding

In this section, we evaluate the search algorithm given in Alg. 1 that finds an embedding for an incoming single stream request. We compare the runtime of Alg. 1 given the line and machine topology to a standard constraint-based SMT counterpart, and to an incremental SMT solution. The used SMT solver is Z3. Such an incremental scheduling approach is known for example from [9]. Alg. 1 terminates and returns when a valid embedding is found. Fig. 7 shows the results of this performance evaluation.

We consider the runtime of each scheduling approach in terms of the time required for adding a single stream, dependent on the current number of streams currently in the system. The standard SMT approach tries to schedule all streams together, whereas the incremental SMT approach fixes the already scheduled streams as constraints, therefore reducing the number of variables significantly. Note that this makes a comparison of Fig. 7a and Fig. 7b difficult.

The line topology is a worst case scenario in the sense that every new stream conflicts on the same ports. In Fig.7a, we show the time required to solve the problem of finding an embedding for a number of streams. Note that as the standard SMT approach is not incremental, adding one new stream requires here to restart the schedule computation for a total number of streams N as defined on the x-axis. The figure shows that for a small number of streams in the network this standard SMT scheduling is comparably fast for dynamic TSN network reconfigurations. However, for increasing number of streams incremental approaches outperform the standard SMT scheduling. For example, adding a new stream to the considered network that already contains 30 streams requires a new calculation for 31 streams that takes roughly 100 seconds. In Fig. 7b, we observe that the runtime of the incremental SMT-based approach increases rapidly with every stream added. In contrast Alg. 1 performs as expected, having a linear increase in runtime with every newly added stream.

B. Computing TSN network path flexibility

In Sect. III, we introduced an approach to calculating a path flexcurve that allows incremental updates as it is based on local, i.e., port specific, data structures. Next, we consider the build and query times for the computation of the flexcurves according to the related work approach, i.e., (4), and according to our incremental approach in (6). We look at the schedules of the line topology provided by Alg. 1 with ns granularity, i.e. $K = 10^6$ slots. Table II confirms the analysis of Sect. III as the computation of (6) requires significantly less time for initialization. Next we consider the time required to query (access) a flexcurve value, i.e., given a stream of size c that is to be admitted the CNC queries the flexcurve of the designated path P to obtain the value (4) or (6). Accessing one or multiple values of the standard approach (4) requires a constant time 157ms. In contrast, accessing a single value (i.e. when embedding a single stream) of (6) is significantly faster. In fact accessing up to 23% of the flexcurve values at once is faster than (4). In addition, for frequent schedule updates the flexcurve (4) requires re-initialization, whereas our version supports incremental updates inherently.

TABLE II
COMPUTATION TIMES FOR INITIALIZATION, QUERY (ACCESS) AND BUILDING A FLEXCURVE VALUES ARE ROUNDED TO [MS].

Type	Incremental Flexcurve (6)	Flexcurve (4)
Initialization	12 ms	157 ms
Single Access	0.5 µs	—
23% Build	156 ms	—
50% Build	302 ms	—
100% Build	563 ms	—

C. Comparing the flexibility of different TAS schedules

Next, we use the flexcurve concept to compare the flexibility of the schedules resulting from different TAS scheduling algorithms. Here, we schedule 10 streams in the considered line topology. We compare Alg. 1, the standard SMT and incremental SMT approach and a publicly available open-source scheduler called *TSNSched* [4], that also based on SMT to compute a schedule. To keep the comparison fair, no scheduling approach has additional constraints or optimizations to optimize the flexcurve. The resulting flexcurves in Fig. 8 for the line topology path reveal different results: *(i)* The incremental SMT approach and Alg. 1 essentially produce a coherent no-gap schedule (first-fit heuristic) which maximizes the schedule flexibility. The standard SMT approach produces gaps which lead to a change in slope, and less overall flexibility. TSNSched reserves more bandwidth for scheduled traffic leading to overall less flexibility.

D. Computation time for deadline-aware flexcurves

In (3), we show that the end-to-end stream delay depends on the starting embedding position a_1 in the first schedule

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

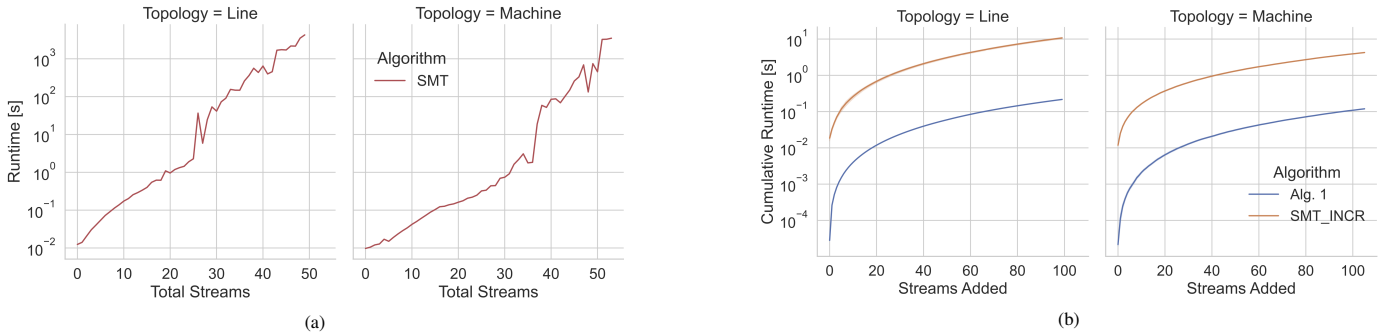

Fig. 7. Standard vs. incremental stream embedding: The runtime of different scheduling approaches given the line topology, and the machine topology (Fig. 6). (a) Since the standard constraint-based SMT approach is not incremental by nature the figure shows the required runtime for embedding a total number of streams. (b) Streams are sequentially added to an empty network (x-axis). The figure shows the cumulative runtime for incremental SMT and Alg. 1.

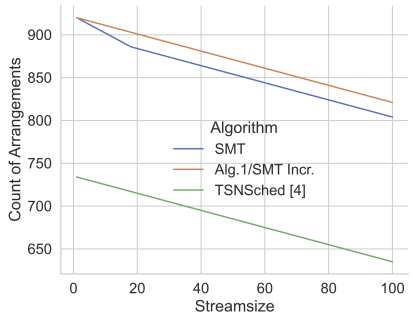

Fig. 8. Flexcurves of schedules created by different scheduling approaches are comparable. A flexcurve is independent of the scheduling-approach.

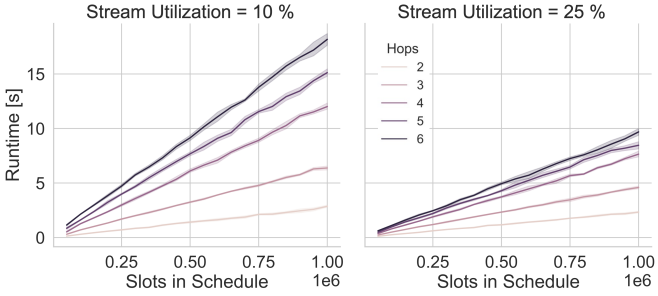

Fig. 9. Runtime evaluation for computing the value of a deadline-aware flexcurve. All schedules are randomly allocated each with 50 continuous occupied slots using different overall schedule occupancy. The deadline was set to 100 slots with a search for a stream of 10 slots length.

s_1. Hence, computing the deadline-aware flexcurve essentially requires to check the end-to-end delay of every possible starting time point a_1. This can be computed using Alg. 1 and its corresponding extension (cf. Sect. IV-C). In Fig. 9, we show the empirically obtained required time for computing one value for the deadline aware flexcurve in terms of the granularity of the schedule, given different schedule utilization. Note that a schedule with a high utilization restricts the search space leading to a faster computation result as can be seen in the figure. Already admitted streams occupy the schedule in a uniformly random manner. We observe that smaller deadline lookups also require less absolute runtime which is due to a fewer absolute number of search operations.

VI. Related Work

In the following, we briefly highlight relevant related works in the context of enabling dynamicity or flexibility for Time-Sensitive Networking. To enable reconfigurability of real-time scheduling in the context of time-sensitive software-defined networks the authors of [10] use a solely departure scheduled network and outline two ILPs to incrementally add single streams to their global schedule. In contrast to the work at hand [10] does not leverage TSN with the Time-aware Shaper. With a similar goal as our work, a decentralized CUC/CNC architecture for TSN reconfigurations that are based on application requests are proposed in [11]. This is similar to Section II-B with intended support for mechanisms to maximize the number of streams. The original idea of the quantification of the flexibility of TSN TAS schedules in form of a flexcurve is given in [7]. In addition to the shortcomings in terms of the time required to build and query the original formulation in Sect. II-B, we note here that the formulation in [7] does not capture important stream properties such as delay deadlines, queuing delays and cycle-period bounds.

There exists a number of related works on incremental scheduling using SMT. In [2], [9] the approach is based on incrementally adding streams to a previously fixed schedule. With the ability to backtrack to an earlier time-point if scheduling fails. This results in a different embedding for the discarded streams, and thus a disruption if the schedule was deployed already. One approach to highlight is given in [12], where the authors use bin packing with a first-fit heuristic, while remaining resource overlaps are resolved using SMT, which significantly improves scheduling speed. In addition to constraint-based scheduling methods, e.g. [2]–[4], heuristic scheduling methods are becoming more prevalent. Heuristics, such as [13], [14], often lend themselves easily to dynamic network situations as their scheduling mechanism is incremental. These approaches limit the stream addition to single streams and are not guaranteed to find an embedding for a requested stream, even if scheduling capacity is sufficient. Note that the main difference to the reviewed approaches above is that they do not consider the impact of scheduling on future stream admissions as, e.g., captured by the flexcurve. The search algorithm presented in Section IV-A can select at least one possible embedding if the deadline can be satisfied and directly

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

measures its future impact. This algorithm is however also restricted in the number of simultaneous embeddings and feasible cycle periods. To treat the increase of complexity in industrial networks, the authors of [15] propose a hierarchical scheduling approach to improve classical scheduling performance. We regard this approach as complementary to ours as it may be introduced to guide a more flexible deployment.

Since we consider stream-based scheduling with equal priority we are restricted in approaches that increase flexibility by a form of window aggregation. This approach is explored in [16]. There the authors increase the flexibility of real-time streams by combining asynchronous scheduling approaches with the isolation that TAS can provide by building shared TAS windows, thus improving the overall asynchronous performance. A formal performance analysis of flexible Window-Based GCL Scheduling that relaxes the non-overlapping strictness of GCL can be found in [17].

One of the main methods to analyze the performance of TSN is Deterministic Network Calculus [18] as it captures the worst case behavior of some TSN mechanisms. It allows deriving a description of the service provided to one stream of interest at a single or multiple TSN switch ports along a network path [5]. Combining this with a deterministic upper bound on the amount of stream data to be transmitted, the network calculus system model [18] readily provides performance bounds as on the end-to-end latency. One such attempt to analyze time-triggered Ethernet (TTEthernet) is given in [19]. A comprehensive model of TSN AVB (802.1. Qav) using network calculus is given in [5], [20]. There the authors provide service curve descriptions for traffic classes in TSN credit-based shaping that allow deriving deterministic upper end-to-end latency bounds. Further applications of the framework include the analysis of different TSN scheduling mechanisms [21] and a comparative analysis of TSN traffic shapers in [22].

VII. Conclusion

Current industrial TSN applications do not allow for highly dynamic network environments, as the state of the art in scheduling TSN resources mandates planning before deployment. In this paper, we contributed an extension of the flexcurve concept to support incremental TSN reconfigurations at runtime. In particular, we allow a flexible stream admission faster than state-of-the-art methods and provide a stream deadline-aware flexibility notion of TSN schedules. Our concepts are an important building block to enable dynamicity in industrial networks without planning changes well in advance. Our evaluations show that this incremental approach requires much less computation time compared to classical approaches and even adapted incremental approaches to admit and embed new TSN streams at runtime. In addition, we provide path-flexibility metrics that can be used to assess present scheduling mechanisms. This work allows incremental changes to TSN configurations that are obtained on the basis of network flexibility. Dynamicity of multiple simultaneous cycle periods in the context of TAS schedules remains a challenging problem that is left for future work.

VIII. Acknowledgment

This work has been funded by the German Research Foundation (DFG) as part of projects B4, T3 within the Collaborative Research Center (CRC) 1053 – MAKI.

References

[1] R. Belliardi *et al.*, "Use Cases IEC/IEEE 60802 v1.3," Sep. 2018.
[2] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks," in *Proc. of ACM RTNS*, 2016, pp. 183–192.
[3] F. Dürr and N. G. Nayak, "No-Wait Packet Scheduling for IEEE Time-Sensitive Networks (TSN)," in *Proc. of ACM RTNS*, 2016, pp. 203–212.
[4] A. Santos, B. Schneider, and V. Nigam, "TSNSCHED: Automated schedule generation for time sensitive networking," in *Proc. of Formal Methods in Computer Aided Design (FMCAD)*, pp. 69–77.
[5] J. A. R. De Azua and M. Boyer, "Complete modelling of avb in network calculus framework," in *Proc. of ACM RTNS*, 2014, p. 55–64.
[6] B. Alt *et al.*, "Transitions: A Protocol-Independent View of the Future Internet," *Proc. IEEE*, vol. 107, no. 4, pp. 835–846, 2019.
[7] C. Gärtner, A. Rizk, B. Koldehofe, R. Hark, R. Guillaume, and R. Steinmetz, "Leveraging flexibility of time-sensitive networks for dynamic reconfigurability," in *IFIP Networking Conference (IFIP Networking)*, 2021.
[8] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Information Systems*, vol. 25, no. 2, pp. 117–134, 2000, the 11th International Conference on Advanced Information System Engineering.
[9] W. Gao, B. Zhao, and X. Mao, "Research on Incremental Scheduling Backtracking Algorithm for Time-triggered Ethernet," in *2020 2nd International Conference on Advances in Computer Technology, Information Science and Communications (CTISC)*, 2020, pp. 75–79.
[10] N. G. Nayak, F. Dürr, and K. Rothermel, "Incremental flow scheduling and routing in time-sensitive software-defined networks," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2066–2075, May 2017, publisher: IEEE.
[11] A. Nasrallah, V. Balasubramanian, A. Thyagaturu, M. Reisslein, and H. ElBakoury, "Reconfiguration Algorithms for High Precision Communications in Time Sensitive Networks: Time-Aware Shaper Configuration with IEEE 802.1Qcc," Jun. 2019.
[12] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll, "Static scheduling of a Time-Triggered Network-on-Chip based on SMT solving," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 509–514.
[13] R. Bhatia, T. Lakshman, M. F. Ozkoc, and S. Panwar, "FlowToss: Fast Wait-Free Scheduling of Deterministic Flows in Time Synchronized Networks," in *2021 IFIP Networking Conference (IFIP Networking)*. IEEE, Jun. 2021, pp. 1–6.
[14] P. Pop, M. L. Raagaard, M. Gutierrez, and W. Steiner, "Enabling Fog Computing for Industrial Automation Through Time-Sensitive Networking (TSN)," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 55–61, 2018.
[15] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer, and F. Dürr, "Scaling TSN Scheduling for Factory Automation Networks," in *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*. IEEE, 2020.
[16] A. Grigorjew, N. Gray, and T. Hoßfeld, "Dynamic Real-Time Stream Reservation with TAS and Shared Time Windows," in *2021 IFIP Networking Conference (IFIP Networking)*, 2021, pp. 1–6.
[17] L. Zhao, P. Pop, Z. Gong, and B. Fang, "Improving Latency Analysis for Flexible Window-Based GCL Scheduling in TSN Networks by Integration of Consecutive Nodes Offsets," *IEEE Internet of Things Journal*, vol. 8, no. 7, pp. 5574–5584, 2021.
[18] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.
[19] L. Zhao, P. Pop, Q. Li, J. Chen, and H. Xiong, "Timing analysis of rate-constrained traffic in TTEthernet using network calculus," *Real-Time Systems*, vol. 53, no. 2, pp. 254–287, Mar. 2017.
[20] L. Zhao, P. Pop, Z. Zheng, H. Daigmorte, and M. Boyer, "Latency Analysis of Multiple Classes of AVB Traffic in TSN With Standard Credit Behavior Using Network Calculus," *IEEE Transactions on Industrial Electronics*, vol. PP, pp. 1–1, Sep. 2020.

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

©IFIP, (2022). This is the author's version of the work. It is posted here by permission of IFIP for your personal use. Not for redistribution.
On the Incremental Reconfiguration of Time-sensitive Networks at Runtime To appear in the proceedings of IFIP Networking Conference 2022.

[21] E. Mohammadpour, E. Stai, M. Mohiuddin, and J. Le Boudec, "Latency and backlog bounds in time-sensitive networking with credit based shapers and asynchronous traffic shaping," in *Proc. of International Teletraffic Congress (ITC)*, 2018.

[22] L. Zhao, P. Pop, and S. Steinhorst, "Quantitative performance comparison of various traffic shapers in time-sensitive networking," 2021.

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.