Asynchronous Time-Aware Shaper for Time-Sensitive Networking

Miklós Máté1 · Csaba Simon1 · Markosz Maliosz1

Received: 12 March 2022 / Revised: 20 August 2022 / Accepted: 21 August 2022 © The Author(s) 2022

Abstract
The Time Sensitive Networking Task Group of the IEEE 802.1 Working Group has developed a number of enhancements to the priority queueing architecture used by Ethernet. Perhaps the most notable one is the time-aware shaper (TAS), which can perfectly isolate priority classes in time with periodically scheduled traffic gates, thereby enabling ultra low latency applications to coexist with lower priority traffic on the same network. It requires near-perfect clock synchronization across the entire network, though, and a central network controller to calculate the timings for the traffic gates, based on known propagation delays on the network links. As a cheaper alternative, we have developed the asynchronous TAS that needs no clock synchronization, network planning, or a central controller. It uses local processes at each switch port to track the high priority streams, predict the arrival times of their next frames, and control the traffic gates such that the low priority streams are not interfering. We show how a simple predictor with minimal computation requirements can be used to track the high priority streams, and we demonstrate via simulations that this gating mechanism can protect the high priority traffic from interference.

Keywords Traffic shaper · Priority queueing · Self-configuration · Prediction · Adaptive

Miklós Máté mate@tmit.bme.hu
Csaba Simon simon@tmit.bme.hu
Markosz Maliosz maliosz@tmit.bme.hu

1 Department of Telecommunication and Media Informatics, Budapest University of Technology and Economics, Magyar tudósok krt. 2., Budapest 1117, Hungary
1 Introduction

Reliability in computer networks is a fundamental design aspect: the service must ensure high probability of successful packet delivery to the destinations. Several methods are used in practice to achieve reliable packet transmissions such as bit error detection and correction codes, retransmissions, and various queuing mechanisms to avoid packet loss due to congestion [1]. Another aspect of reliability is timely arrival of mission critical packets for delay-sensitive applications. Time-domain multiplexing (TDM) schemes guarantee this by allocating an unique time slot for each source, thus creating contention-free access to the resources. The inflexibility of strict TDM systems made them gradually lose market share compared to networking technologies that are more universally applicable, e.g., the IEEE 802.3 Ethernet [2] for local area networks (LANs). A notable example of this trend is how circuit-switched telephony systems are phased out in favor of Voice over IP.

It has been a long time since Ethernet has abandoned its original design of randomized access to a shared coaxial cable in favor of using full duplex point-to-point links, connected by switches implementing the IEEE 802.1 bridging architecture [3]. This change effectively moved the contention for the shared resource from the medium access to the queues of the outgoing ports of the switches, where it is more easily manageable. This not only allowed higher link speeds, and better link utilization, but larger LAN topologies as well [4].

The 802.1 bridging architecture is used by not only Ethernet, but all IEEE 802 data link layer technologies, such as the 802.11 wireless LAN, promoted by the Wi-Fi Alliance [5].

Timely delivery of Ethernet frames has been in the focus of interest at the IEEE 802.1Q Working Group [3]. Priority queuing has been added to the bridging architecture early on, with eight priority classes ranging from best effort to low latency voice/video traffic. Initially the 802.1Q standard only included a strict priority queuing scheme, but later amendments introduced more options for queuing and shaping [6].

Emerging network use cases require ultra reliable ultra low latency (UR-ULL) communication, i.e., sub-millisecond end-to-end latency, bounded latency variation, and no packet loss due to congestion. Such use cases include industrial control loops [7], automotive applications [8], 5G wireless fronthaul [9], and tactile Internet [10].

The Time-Sensitive Networking (TSN) Task Group of the IEEE 802.1Q Working Group is the successor of the Audio–Video Bridging (AVB) Task Group, both developing more sophisticated Quality of Service (QoS) features than strict priority transmission selection among the priority queues. They have published a number of amendments to IEEE 802.1Q standard that introduce new queuing [1, 6, 11, 12], time synchronization [13], and configuration mechanisms [14, 15] to the 802.1Q architecture, with the aim of providing bounded latency and latency variation guarantees. The TSN Task Group released several amendments, in this paper we will only mention the ones relevant to our work.

An interesting new feature of TSN is frame preemption, introduced in 802.1Qbu [16] and 802.3br [17] for Ethernet, which allows the bridge to interrupt
the transmission of a low priority frame to send out high priority ones that arrived
in the queue. After all the high priority frames in the queue have been transmit-
ted, the transmission of the interrupted low priority frame resumes. This feature
requires major design changes in hardware; thus, the network equipment ven-
dors have been slow to implement it, but recently preemption-capable Ethernet
controllers have started to appear in product catalogs [18]. In this paper we will
assume that frame preemption is not available.

The most important TSN mechanism from our perspective is the time-aware
shaper (TAS) introduced in amendment 802.1Qbv [11]. It turns the priority queues
into a TDM-like system via traffic gates governed by a global clock. The need for
high-precision network-wide time synchronization and central network control limit
the applicability of TAS.

In this paper we propose the asynchronous TAS (ATAS), which is a modification
to the TAS architecture. ATAS autonomously detects the parameters of high prior-
ity streams, and schedules the traffic gates without needing time synchronization or
centralized network monitoring/management. We first presented ATAS in [19], in
this paper we give more detailed description of the proposed algorithms, extend the
analytical background, and present new simulation results.

The rest of the paper is organized as follows. Section 2 gives a detailed overview
of the TAS, and Sect. 3 presents our proposed enhancement, the ATAS. Section 4
contains simulation results that show the viability of our proposal. Section 5 sum-
marizes the literature related to this work. Finally, Sect. 6 draws the conclusions.

2 IEEE 802.1Qbv Time-Aware Shaper

Perhaps the most notable new queueing mechanism introduced by the IEEE 802.1Q
TSN Task Group is the TAS added in amendment 802.1Qbv [11]. It extends the
architecture of 802.1Q with transmission gates attached to each priority queue [20].

The main purpose of TAS is to isolate the priority classes in time, i.e., to pro-
tect the frames of one priority class from interference by frames from other priority
classes competing for the same outgoing port. This makes TAS a great enabler for
ULL applications.

2.1 Traffic Gates

The traffic gates of TAS have two states: open or closed. When a gate is closed, the
frames waiting in the corresponding priority queue are not eligible for transmission.
If multiple gates are open at the same time, the regular priority queueing mechanism
of 802.1Q is used to select the frame for transmission.

The state of the traffic gates are governed by a gate control list (GCL), as shown
in Fig. 1. The GCL is a sequence of 8 bit numbers corresponding to the 8 priority
queues, e.g., in the figure gates #7 and #5 are open currently, while the other ones
are closed. It advances with a constant tick rate, and wraps around at the end. Imple-
mentations of TAS have a finite memory capacity; thus, the GCL has a maximum
The time slice when the gate is open for a priority class is called a traffic window. The windows of the priority classes are independent, multiple ones can be open at the same time, and it is possible to close all of them. It all depends on the configuration done by the network operator. The duration of the open traffic window is also configured by the operator, e.g., a gate can be kept open all the time, it can be opened to let through only a single frame, or some duration in between.

The 802.1Qbv standard mandates that a frame is only eligible for passing through the gate, if its transmission can conclude before the traffic window closes. This implies that the minimum window size should be larger than the frames expected in the corresponding priority class. These constraints also increase the computational complexity of synthesizing a suitable GCL schedule.

### 2.2 Configuration

Configuring the TAS of the switch ports is done by a central network controller (CNC) entity in the TSN architecture [7, 21]. The CNC must compute the GCL for each switch port in the network using its knowledge about the topology and the propagation delays in the network, and the stream registration information. The protocol for distributing the configuration is standardized in the 802.1Qcc [15] standard amendment, but the control functions and algorithms are proprietary solutions of the equipment vendors.

There is a substantial amount of scientific literature about the mathematics of scheduling the time gates. Calculating the schedule is similar to the Knapsack problem, which is proven to be NP-complete [22]. The most common approach of solving the scheduling problem is to gather the constraints on the streams dictated by source timing, network delays, end-to-end latency limits, isolated access to the links etc., and feed them into an integer linear program (ILP) solver [23]. The main difference between these papers is the method they use to gather the constraints, how
they partition the problem space, and how they apply the results. In [24] an iterative algorithm for computing the schedule and minimize the number of gate transitions, while disallowing the queueing of packets, is presented. Enhancements for the scheduling are proposed in [25] to avoid reordering the frames due to queueing delays in previous nodes along the path. The authors of [26] use first-order theory of arrays to express the scheduling constraints. Scheduling the gates and routing the streams at the same time is proposed in [27] to maximize network utilization.

The reported run time of these algorithms for GCL computation ranges from a few seconds to several minutes, depending on the size of the network and the traffic mix.

In networks with QoS guarantees there is a call admission control (CAC) mechanism at the edge of the network, which allows only a limited amount of streams into the network, otherwise it would be impossible to maintain bandwidth and latency guarantees. In the TSN architecture with TAS the CAC has to register not only the traffic characteristics of the streams (e.g. maximum data rate, burst length, frame size), but their timing information as well [15].

If a frame misses its dedicated transmission window, it has to wait for the next one, which can be much longer than waiting for an interfering frame transmission to finish. Therefore, it is of utmost importance that the gate timings are configured such that the gates do not disrupt the flow of the important traffic streams.

For correct GCL configuration the CNC has to know when the frames arrive at each switch. This necessitates high precision clock synchronization across the whole network, including the traffic sources. The TSN Task Group adopted an upgraded version of the IEEE 1588 precision time protocol (PTP) in amendment 802.1AS [13]. This protocol can provide clock synchronization with precision around 1 μs, if there is a high precision clock source in the network. In this paper we are focusing on industrial automation networks [7], where a 1500 byte frame (that is the maximum size) takes around 123 μs to transmit on a 100 Mbps link (with overhead), or 12.3 μs on a gigabit link. If the PTP synchronization is properly configured, TAS can operate with sufficient precision in such networks.

TAS is also relevant in high speed networks, e.g., 5G wireless fronthaul [9]. However, if the link speeds are 10 Gbps and beyond, then the frame transmission time is comparable to the accuracy of the time synchronization, and it becomes hard to match the opening and closing of the traffic gates with the arrival time of the frames. It is not always possible to achieve perfect clock synchronization between the traffic sources and the transport network, especially if the two belong to different organizations. If the time synchronization is not sufficiently accurate, the traffic windows have to be enlarged to ensure safe passage for the frames before their traffic gates close, even if they are late. This decreases the throughput capacity for the other priority classes.

If the transmission windows are not configured to have space for frames that missed their windows, one miss causes the next window to overflow with one frame, creating an avalanche of overflows. To stop this the GCL schedule must either increase the size of the transmission windows, or include drain windows, where the overflowed frames from multiple priority classes can leave the queue. Both are
wasting link utilization, but the latter one also introduces unpredictable latency variation. Still, they are better than dropping the overflown frames.

We can thus conclude that great care needs to be taken when configuring the GCL.

In our previous work [28] we used simulations to identify configuration patterns for TAS that result in robust operation. When frame preemption is not available, we found that the safest approach for TAS is to divide the eight priority queues into two groups: use the transmission gates only on the low priority group, and let the high priority traffic pass through at any time. This protects the high priority traffic without exposing them to the perils of time synchronization errors. With more than two priorities, the middle ones would have to be gated, and thus the timings would have to be much more precise. This 2-way split roughly corresponds to the express–preemptable split of 802.1Qbu.

3 Asynchronous Time-Aware Shaper

In the previous section we have seen that the IEEE 802.1Qbv TAS is a powerful tool, capable of isolating traffic of different priority classes in time. It is quite expensive, though. It requires high precision time synchronization across the whole network, including the traffic sources, which might belong to a different entity than the transport network provider. It also requires a CNC overseeing the traffic, and computing the GCL for each port in the network. The CNC has to know detailed timing information about the traffic sources, and it must have precise measurements about propagation delays. In other words, the network has to be planned in advance for TAS.

This is too complicated, and expensive. Is it not possible to set up TAS to protect high priority traffic without large investments into network infrastructure?

Our proposed solution for this problem is the asynchronous time-aware shaper (ATAS).

The idea is to run a local process at each switch port, which tracks all high priority streams passing through, predicts the arrival time of the next frames, and closes the traffic gates of the low priority queues before the next high priority frame arrives, protecting it from the interference. This solution does not need any central control, and no clock synchronizing, because the stream tracking and prediction mechanism runs on the local clock independent of the rest of the network (and the other ports of the switch).

In ATAS we divide the eight priority queues into two groups: high and low priority. Our gate control heuristic only closes the traffic gates of the low priority queues, and always lets the high priority traffic pass unobstructed. In our previous work [28] we have analyzed TAS in various configurations, and identified this gating pattern to be robust against uncertainties in the arrival times of the high priority frames.

We have designed the ATAS gate control heuristic to ensure high probability of unobstructed pass-through for high priority traffic. In ULL networks this is the expected quality level, e.g., frame preemption was designed to eliminate latency...
variations comparable to the frame transmit times, and some of the papers on GCL computation include zero queueing among the constraint set [24].

When using ATAS, some kind of admission control is still needed to avoid overloading the network with high priority streams. This admission control can be much simpler than with normal TAS, as it does not have to register the timings of the streams, just their bandwidth requirements. Therefore, we can use the basic version of the IEEE 802.1Qat stream reservation protocol, SRP [14] without the TSN modifications [15]. SRP registers only the maximum frame rate, and the maximum frame size for the streams, the timings are automatically deduced by ATAS at each node.

The computational complexity of ATAS depends on the number of high priority streams passing through the switch port. It is independent of the size of the network, because the ports only calculate their own schedule, and the number of high priority streams routed through a single port is limited by the bandwidth of that port. Based on the literature about the traffic mix in industrial networks [29] we expect that the ULL streams are at most 10% of the total traffic, thus, ATAS has to deal with no more than a dozen ULL streams at the same time.

The design of this shaper has three main components: prediction of the next frame arrival, controlling the gates, and stream property tracking.

### 3.1 Prediction of Frame Arrivals

The gate control mechanism of ATAS is based on predictions for the arrival times of the next frames in the currently active high priority streams.

Data extrapolation, prediction, and forecasting algorithms have been extensively studied in the past several decades [30]. Widely used techniques, like Wiener filters, are designed to perform well in generic situations. In our case; however, we can exploit the traffic characteristics of high priority streams to design a much simpler predictor.

We know that the sources that generate the high priority traffic are like industrial control loops [7], and base stations of cellular networks [9], which all work in a periodic fashion. We have also seen in the previous section that the GCL of the TAS is repeated periodically, and all the papers presenting algorithms for calculating the GCL assume perfectly periodic sources [24–27]. Therefore, it is safe to assume that the input of our frame predictor is periodic: we have to estimate the period of the source, and compensate for the uncertainty due to random network delays and inaccuracies in clock synchronization.

The predictor has to converge fast on the stream parameters, because we want to minimize the number of unprotected frames at the start of the stream. The source may be augmented to start its stream with a training sequence to teach the predictors before sending the real data, but we do not require such changes in the applications.

There are performance concerns as well: we cannot afford complex computations in an Ethernet interface controller, and memory capacity is also scarce. Our proposed frame predictor is deliberately as simple as possible. In Sect. 4 we will show that even this simple predictor can efficiently protect the high priority traffic.
3.1.1 Negative Correlation-Based Predictor

Our predictor is thus designed to work on periodic traffic, and it leverages the observation that when one frame is delayed by the network, the inter-frame time before it increases, and the inter-frame time after it decreases by the same amount. We call our solution a negative correlation-based predictor, because it assumes a negative correlation between the last inter-arrival and the next one that needs to be predicted. In the followings we give a formal description of the predictor, and analyze its properties.

The traffic source in our model is quasi-periodic. It starts sending frames at $T_0$, and the $i$th frame is sent at $T_0 + iT + t_i$, where $T$ is constant, and $t_i$ are independent, identically distributed random variables with unspecified distribution, representing the processing delay from the periodic event that triggers the message. We only assume that $t_i$ have a finite expected value independent of $i$, and they have a finite variance.

Frame $i$ arrives at the switch with the ATAS queue at

$$x_i = T_0 + iT + t_i + n_i,$$

where $i = [0, 1, \ldots]$, and $n_i$ is the network delay accumulated until arriving at the outgoing queue of the TSN switch. The network delays are also independent, identically distributed random variables with unspecified distribution that has a finite expected value.

The $T_0, t_i, T$, and $n_i$ parameters are unknown to the ATAS predictor. The goal of the predictor is to give an estimate for the arrival time of the $i + 1$th frame as $\tilde{x}_{i+1}$.

The difference between two frame arrivals is

$$d_i = x_i - x_{i-1} = T + t_i - t_{i-1} + n_i - n_{i-1},$$

where $i = [1, 2, \ldots]$. The expected value of $d_i$ is $E\{d_i\} = T$, because $E\{t_i - t_{i-1}\} = 0$, and $E\{n_i - n_{i-1}\} = 0$. If we have an estimate for $d_{i+1}$ as $\tilde{d}_{i+1}$, then we can predict

$$\tilde{x}_{i+1} = x_i + \tilde{d}_{i+1}.$$  \hfill (3)

The random terms $t$ and $n$ in (1) are from a series of independent random variables, and it is impossible to predict noise; thus, the optimal choice is to estimate the expected value of the inter-frame arrival with the sample average of $d_i$. Our proposed predictor adds a twist that aims to dampen the effect of outliers and thus keep the prediction closer to the expected value. It computes

$$\tilde{d}_{i+1} = \bar{T}_i + (\bar{T}_i - d_i) = 2\bar{T}_i - T - t_i + t_{i-1} - n_i + n_{i-1},$$

and thus

$$\tilde{x}_{i+1} = x_i + \tilde{d}_{i+1} = x_i + 2\bar{T}_i - d_i.$$  \hfill (5)

Here, $\bar{T}_i$ is the average inter-arrival time, computed with the exponentially weighted moving average (EWMA) algorithm as $\bar{T}_i = ad_i + (1 - a)\bar{T}_{i-1}$, where $0 < a < 1$.

It is easy to see that $\tilde{d}_{i+1}$ is an unbiased estimator of $d_{i+1}$. From (2) and (4) the prediction error is
\[ e = \tilde{d}_{i+1} - d_{i+1} = 2\hat{T}_i - 2T + t_{i-1} - t_{i+1} + n_{i-1} - n_{i+1}. \]  
(6)

The expected value of the prediction error is thus

\[ \mathbb{E}\{e\} = 2\mathbb{E}\{\hat{T}_i\} - 2T + \mathbb{E}\{-t_i - t_{i+1} + n_{i-1} - n_{i+1}\} = 2(\mathbb{E}\{\hat{T}_i\} - T) = 0. \]  
(7)

At first glance, it would seem from (5) that \( \tilde{x}_{i+1} \) is completely independent of \( x_i \), because

\[ \tilde{x}_{i+1} = x_i + \tilde{d}_{i+1} = x_i + 2\hat{T}_i - d_i = x_{i-1} + 2\hat{T}_i. \]  
(8)

However, since we are using EWMA to compute \( \hat{T}_i \), it is only an asymptotic estimator of \( T \), and this imperfection is the key to why (5) is useful. For simplicity, we take a particular realization of this stochastic process, where

\[ \bar{T}_{i} = \tau + \epsilon + \frac{\alpha}{\tau} \]  
(9)

and thus the prediction is

\[ \tilde{d}_{i+1} = \hat{T}_i - d_i = \tau + \epsilon(2\alpha - 1). \]  
(10)

It is easy to see that \( \forall \alpha : \tilde{d}_{i+1} < \bar{T}_i \), and \( \alpha < 0.5 : \tilde{d}_{i+1} < \tau \). The prediction thus reacts to \( \epsilon \) and compensates for it.

In the next round we have \( \tilde{T}_{i-1} = \tau + \epsilon \). Assuming that the prediction was right, i.e., \( d_i = \tau + \epsilon(2\alpha - 1) \), in this round we get

\[ \bar{T}_i = \tau + \alpha^2\epsilon \]  
(11)

and the prediction becomes

\[ \tilde{d}_{i+1} = \bar{T}_i - d_i = \tau + \epsilon(2\alpha^2 - 2\alpha + 1). \]  
(12)

This time we get that \( \forall \alpha : \tilde{d}_{i+1} > \bar{T}_i \), and its minimum is at \( \alpha = 0.5 \). The prediction is thus smoothed out both before and after the outlier.

This predictor has a fast lock-in: it can start giving usable predictions after two received frames, regardless of the stream properties and the network delays. When we have \( d_1 = x_1 - x_0 \) we have an initial prediction for the third frame, and in Sect. 4 we will see that even a coarse prediction is enough to protect a frame from the low priority traffic. This predictor is also fast and its memory usage is negligible: \( x_i, d_i \) and \( \bar{T} \) for each high priority stream. If we have used, e.g., a Wiener filter, we would have needed to store several dozen samples, and compute the inverse of a huge matrix [30].

In [19] we presented a simpler model for the source with constant period \( T \). There is another way of generalizing that model to be quasi-periodic: the \( i \)th frame is sent at \( T_0 + \sum_{j=0}^{i-1} t_j \). This model describes a software interrupt that is handled with a delay, and the next interrupt is scheduled relative to the handler, so the processing delays are accumulating. The negative correlation-based predictor works equally well for this model.
3.1.2 Predicting Periodic Bursts

As mentioned in Sect. 2, there are a number of papers presenting algorithms for computing the GCL. These algorithms all assume that the traffic sources send single packets with a known period [25–27], and in some papers the period of the sources is the same as the period of the GCL [24]. Our model for the traffic sources, presented in the previous section, is more general than that, but it is still not realistic enough. One can only send 1500 bytes of data in an Ethernet frame, but it is not unthinkable that a source generates more data within a single period; thus, it needs to send a burst of frames periodically. Our frame predictor has to be able to handle such periodic bursts.

To handle periodic bursts we have developed a heuristic that uses two cycles of the negative correlation-based predictor: one that tracks the time between the starting frames of the bursts, and another that tracks the time between the frames within a single burst. The latter is necessary, because we cannot expect the frames in a burst to arrive back-to-back. This can happen for multiple reasons, e.g. when the source has sent the frames of the burst back-to-back, but we operate on an aggregate network with higher link speeds and thus shorter frame transmission times. Unrelated frames from other streams may also be mixed in between the frames of the burst, but that should not affect our per-stream predictor.

**Algorithm 1** Prediction for periodic bursts

```plaintext
function STREAM.PREDICT(stream as s, timeNow as tn)
    if s.T then
        if s.burstT then
            if tn > s.bursta + s.T*(s.burstLen-1) + s.packetSize then
                return s.bursta + 2*s.burstT - s.burstd ▷ Start of next burst (1)
            else
                if s.bursta = s.x then
                    return s.x + s.T ▷ Second frame of burst (2)
                else
                    return s.x + 2*s.T - s.d ▷ 3rd or later in burst (3)
                end if
            end if
        else
            return s.x + 2*s.T - s.d ▷ No bursts (4)
        end if
    else
        return -1 ▷ Only received 1 frame, no prediction yet (5)
    end if
end function
```

Figure 2 shows how prediction works on a bursty stream. The first frame of a burst is predicted using the negative correlation-based predictor using the first frames of the previous bursts. The prediction for the second frame of a burst is simply the arrival of the first frame, plus the average in-burst inter-arrival, because there is no previous arrival time to negatively correlate to. Unfortunately, every burst
starts with such a soft-reset on the predictor. The third and subsequent frames of the burst are predicted normally.

We also need to predict the burst length $L$, because we need to know whether to expect more frames to come in the current burst, or a long silence until the start of the next burst. If the burst length is underestimated, then the last frames of the burst will not be protected by the traffic gates. If it is overestimated, then we steal bandwidth from the low priority traffic by keeping the traffic gates closed for nothing. In our simulation study we took a safe approach by predicting the burst length to be the maximum number of frames received in the last $K = 5$ bursts.

The full heuristic is shown in Algorithm 1. The burst predictor depends on the current time to decide whether to expect more frames in the current burst, or to return the predicted start of the next burst. If we used the number of frames received in the current burst instead, then a dropped frame would mislead the predictor. The decision threshold between anticipating more frames in the current burst or the start of a new one is at the time when the frame, which the predictor thinks should be the last one in the current burst, ends.

If the admission control mechanism of the network can register the burst characteristics of the streams, that information may be useful as an initial value for the stream parameters in the predictor. In our implementation of ATAS we did not use such prior information about the streams.

### 3.2 Gate Control Heuristic

Once we have a prediction for the most likely arrival time of the next frame in a high priority stream, we need an algorithm for computing the next closing and opening time of the traffic gates of the low priority queues. The gate close times have to be calculated in advance based on the predictions, because the queues have to know if the low priority frames can be transmitted before their gates close, otherwise the frames are not eligible for transmit. This rule ensures that once the gates are closed, the low priority frames are surely not occupying the outgoing link.

In our gating scheme the high priority traffic is never gated, because this gives the strongest guarantee that the high priority traffic is not disturbed by the gating schedule [28]. This scheme cannot protect the high priority streams from each other, though.

![Fig. 2 Predictions for the first three bursts of a stream, the numbers in parenthesis refer to points in Algorithm 1](image-url)
The goal of the gating is to eliminate the interfering low priority traffic from the way of the high priority frames that belong to ULL applications. The strength of the protection mostly depends on the gate control mechanism: traffic gates of the low priority queues should be in closed state when a high priority frame arrives. Our gate control heuristic designed to be as simple as possible, and in Sect. 4 we will show that it protects the high priority traffic with high efficiency even when the prediction accuracy is low.

3.2.1 Single Stream

The time to close the gates should be the predicted arrival time of the next frame in the high priority stream. The predictor returns the most likely arrival time; thus, it makes no sense to close later than that. On the other hand, should we close earlier? There is a good chance that the frame will arrive earlier than predicted, in which case the low priority traffic can hold it back until the gates close. In practice the traffic can rarely fill its time slot when preemption is unavailable, and this is a major limiting factor in the bandwidth efficiency of TAS [31]. In ATAS we cannot depend on this behavior; thus, we need to close the low priority gates earlier than the predicted time to compensate for the prediction error. The question is: how much earlier should we close?

The earlier we close the stronger the protection is, the later we close the more bandwidth we leave for the low priority traffic. These are completely orthogonal constraints for which we cannot optimize. Perfect protection can only be reached if the low priority gates are permanently closed, but we need to let the low priority traffic through somehow.

In our implementation of ATAS we added a guard band: we track the maximum of the prediction error as $E = \forall i : \max \hat{x}_i - x_i$, and we close the gates at $\hat{x}_{i+1} - E$. Arriving earlier than predicted is limited by physical constraints, like link latencies; thus, $E$ cannot grow arbitrarily large. Its variance can, however, grow quite large due to queueing latencies in previous switches when multiple high priority frames contend for the link. We imposed a limit of half the frame’s length on the guard band. In [19] we presented our results without this guard band for simplicity.

Once the gates are closed, we need to decide how long they should remain closed. Once the anticipated high priority frame has arrived, the state of the gates become irrelevant due to the priority queue selection. Thus, the gate close duration is
essentially the timeout we are willing to wait for the high priority frame if it is late. If it arrives much later than predicted, the low priority gates are open again, and the frames that were held back are free to go and interfere with it, making it even later at the next switch. We need to minimize the risk of this scenario happening.

Assuming that the high priority frame arrives at the predicted time and its transmission starts immediately, because the gating cleared the low priority frames from its way, it will occupy the outgoing link for the duration of its transmission. If we close the gates for this transmission duration, then we get a nice long safe interval for the high priority frame to be late, and yet we do not steal bandwidth from the low priority traffic, because if the high priority frame arrives as predicted, the link becomes occupied anyway.

If the high priority frame is held up at an earlier switch by other high priority frames, it can be delayed by multiple frame lengths. Unfortunately, those frame lengths belong to streams that are potentially unknown to our predictor, and these queueing events are quite unpredictable, even for sophisticated pattern recognition algorithms that we do not have the computation budget for. This is, however, a relatively rare event, unless the network is crowded with high priority streams.

### 3.2.2 Multiple Streams

For a single high priority stream we have defined a time interval for closing the low priority traffic gates in the previous section. If multiple high priority streams are leaving the switch at the same port, the pattern for closing the gates should be the union of the individual close intervals. We need an algorithm that produces this union from the trigger events coming from the frame processing modules.

We have developed two variants for the gate control heuristic, depending on what trigger events are available in the queue implementation. The common part is that they are greedy heuristics: they look for the earliest high priority frame arrival in the future, and schedule the gate close to that time. The first variant only gets notified whenever a high priority frame gets enqueued, the second variant also gets feedback from the medium access control (MAC) module about the transmission status of high priority frames.
Algorithm 2 Gate Control Without MAC Feedback

function `onHighPrioEnqueued(frame)`
  stream ← knownStreams.find(frame)
  if stream then
    stream.update(timeNow(), frame.size())
    duration ← frame.size() / linkBandwidth()
    if openTimer.running() then \(\triangleright\) this frame is fully protected
      if openTimer.type() = Predicted \(\triangleright\) timeout based on prediction
        openTimer.reschedule(timeNow() + duration, Frame)
      else \(\triangleright\) timeout based on received frame
        openTimer.reschedule(openTimer.timeout() + duration, Frame)
    end if
  else \(\triangleright\) this frame is not protected
    if closeTimer.running() then \(\triangleright\) earlier than predicted
      closeTimer.cancel()
    end if
    openTimer.start(timeNow() + duration, Frame)
    gates.close(timeNow() + duration) \(\triangleright\) only for state consistency
  end if
  knownStreams.insert(frame, timeNow())
end if
end function

function `onCloseTimer()`
  gates.close(timeNow() + closeDuration)
end function

function `onOpenTimer()`
  (lateStart, lateStop) ← knownStreams.findLate(timeNow())
  if lateStart then \(\triangleright\) we are still waiting for a frame
    openTimer.start(lateStop, Predicted)
  else
    (nextStart, nextStop) ← knownStreams.findNext(timeNow())
    if nextStart then
      gates.open(nextStart)
    else
      closeDuration ← nextStop - nextStart \(\triangleright\) global for onCloseTimer
    end if
    closeTimer.start(nextStart)
    closeDuration ← nextStop - nextStart \(\triangleright\) global for onCloseTimer
    gates.open(\(\infty\)) \(\triangleright\) no high prio frame expected
  end if
end function
The first variant of the heuristic is shown in Algorithm 2. In addition to two timers it only uses one external trigger, the event that a high priority frame has been enqueued. In the event handler onHighPriOEnqueUed we do not have to touch the gates: if they are closed, they can stay closed, if they are open, it is too late to do anything.

If the gates are closed when a high priority frame arrives, we keep the gates closed, and adjust the open timer to the presumed end of the high priority frames. There can be multiple high priority frames queued up, when they arrive from different switch ports. If the timeout was set based on a prediction, we correct it to the real time it takes to transmit the high priority frame. If the timeout was set based on a received frame, that means the frame is still being transmitted, and the newly received frame will get queued until it finishes; thus, we need to move the timeout. The type of the open timer is set to Predicted or Frame when it is started.

If the gates are open when a high priority frame arrives, we cancel the schedule, because it became obsolete. The new schedule for the gate close timer will be based on the predictions, but this calculation needs to be deferred to the point, where all
queued high priority frames have been sent out. Without feedback from the MAC we do not know when that will happen. Therefore, here we close the gates, and schedule the opening to the presumed end of the received frame. The schedule will be computed by the callback of the gate open timer.

The frame prediction is called from inside the \texttt{findNext} and \texttt{findLate} functions, which are shown in Algorithm 3. They both go through all active high priority stream linearly, the difference is their selection criteria. In \texttt{findNext} we look for the stream with the earliest frame arrival in the future, which is the basis of the greedy algorithm for scheduling the gate close time. In \texttt{findLate} we look for a stream, where a frame should have arrived by now, but we are still waiting for it. Unfortunately, the result of these two functions cannot be cached, because the prediction is time-dependent.

In addition to the trigger event we have two timers, and their expire callbacks. We only need one of each timer, not per-stream. The two timer callbacks are also shown in Algorithm 2. The close callback simply closes the gates, and schedules the open timer, based on the duration value in the global variable. The open callback checks the schedule if we are still waiting for the arrival of a frame before opening the gates. If not, then it schedules the gate close timer based on the predictions for the future arrivals.

When changing the state of the gates, we also need to tell them the expected time of the next change. This is crucial when the gate is opened, because it has to know whether a frame can finish transmitting before the gate closes. If the gate is closed, the time when it will open is not really important; thus, we do not update it when rescheduling the open timer.

The guard band handling is omitted from the algorithm listings for simplicity. To add guard band one simply needs to decrease \textit{stime} in \texttt{findNext} by the guard band time, after \textit{etime} has been set.
The second variant of the heuristic is shown in Algorithm 4. It receives feedback from the MAC via the onHighPrioSent callback when the transmission of a high priority frame is completed. With this callback we can simplify the onHighPrioEnqueue callback: we do not have to guess the time it takes to transmit the frames, and we do not need to use the open timer to defer computing the schedule. The two timer callbacks are the same as before.

Other than simplifying the program logic, this feedback does not change the computational complexity of the gate control heuristic, because most of the calculations happen in onOpenTimer. In the simulator we did not have this feedback from the MAC.

From the timer callbacks in Algorithm 2 and the search functions in Algorithm 3 we can see that the gates are closed for the union of the close intervals of the individual streams when waiting for high priority streams. The gate close timer is scheduled according to the predicted arrival times, and the gates are opened when all the overlapping close intervals are over. In onHighPrioEnqueue we update the records about the stream, and interrupt the gate schedule, which will be resumed once all high priority frames have left the queue.

The computation complexity of the frame prediction scales linearly with the number of streams, and for a single stream it is just an if-ladder and some simple calculations, as shown in Algorithm 1. This linear scan is run twice for each high priority frame in onOpenTimer, except when the frames arrive in an overlapping burst, in which case the second scan is only run once at the end of the union of the intervals.

Although this simple gate control heuristic works for any number of high priority streams, it cannot protect them from each other, because it can only distinguish

---

**Algorithm 4 Gate Control With MAC Feedback**

```
function onHighPrioEnqueued(frame)
    stream ← knownStreams.find(frame)
    if stream then
        stream.update(timeNow(), frame.size())
        if openTimer.running() then ▷ this frame is fully protected
            openTimer.cancel() ▷ keep the gates closed
        end if
        if closeTimer.running() then ▷ earlier than predicted
            closeTimer.cancel()
        end if
        gates.updateCloseTime(∞)
    else
        knownStreams.insert(frame, timeNow())
    end if
end function

function onHighPrioSent()
    if highPriorityQueue.empty() then
        do the same thing onOpenTimer does
    end if
end function
```
between two priority levels. It does not have any explicit protection against starving the low priority traffic, either. There are two levels of starving: when some gate open intervals are too short for the queued low priority frames, and when all intervals are too short. The first level is unavoidable in our model, since the high priority streams are extremely delay-intolerant, so we cannot hold them back deliberately to send out low priority frames. The second level is not an issue, since it can only happen when at least 50% of the frames are high priority, and that ratio is unreasonably high [29].

3.3 Stream Property Tracking

Tracking the properties of the high priority streams is the most complicated part of ATAS. There are several network effects that can affect the arrival times of the frames, and there are many ways the source can alter its behavior (if it stays within the pre-arranged QoS parameter range). It is possible to create a property tracking system that works most of the time, but maybe even a highly sophisticated pattern matching scheme with neural networks cannot cover all the corner cases. In our implementation for the simulation study we only used some simple heuristics for detecting the burst parameters as a proof-of-concept, and it worked well for the scenarios we have tested. We have decided to omit the pseudocode for the stream property tracking, because the heuristic for burst handling is long and hard to follow. Instead, the components of the heuristic will be explained in the text.

The first step is to identify the high priority streams. Amendment 802.1Qci adds per-stream filtering and policing (PSFP) capability [32]; thus, the bridges can identify streams and track their properties. We reuse this capability for the ATAS for keeping track of the high priority streams. The priority of the frames is set in the VLAN tag, but PSFP is also capable of assigning internal priority value independently of that.

The data needed by the predictor are the following. When frame \( i \) arrives, we need to save \( x_i \), \( d_i = x_i - x_{i-1} \), and update the average inter-frame time as \( \bar{T}_i = \alpha d_i + (1 - \alpha)\bar{T}_{i-1} \). The EWMA algorithm is not only useful for the reasons mentioned in Sect. 3.1, but it is also fast, requires no extra memory, and it can adapt to changes in the behavior of the source, because it assigns diminishing weights to older values. The convergence speed, and the noise canceling strength of EWMA depend on the \( \alpha \) parameter.

To track the bursts in the stream due to the source generating more data in each period than what can fit into one Ethernet frame, first we need to detect if this is the case. We have implemented a simple heuristic for this: if frames start arriving periodically, then there is a large gap between two frames, and the previous inter-frame time resumes, that looks like a gap between two bursts, and thus we declare the stream as bursty, and remember the number of frames received in the burst. Our heuristic contains limits on the burst size and the length of the silence to distinguish between bursts, missed frames, and the stream stopping and restarting.

The most serious limitation of the stream property tracking comes from frame drops. In Ethernet it is generally impossible to tell if frames were dropped or re-ordered in previous switches due to the lack of sequence numbering or time stamps.
There are some higher-layer protocols that add sequence numbering, but cross-layer packet inspections are fragile and limited to a few known protocols. There are amendments to the bridging architecture, like 802.1CB frame replication and elimination for reliability (FRER), that add their own sub-headers with sequence numbering [33], but ATAS cannot depend on the presence of this.

Once we know that the stream has bursts, we need to keep track of the time between the burst starts the same way as tracking the time between the individual frames, these are $burst_x$, $burst_d$ etc. in Algorithm 1. We also need to remember $L$ ($burstLen$ in the pseudocodes), the number of frames received in the last $K$ bursts, including the current one.

We have also implemented a method for detecting if the source stopped being bursty. If the last $K$ bursts all were $L = 1$, we declare the stream to be not bursty anymore.

If the guard band is enabled for stronger protection against shifts due to prediction errors, then the stream property tracking must also measure the maximum of the prediction error, and supply it to the gate control heuristic.

Moreover, the gate control heuristic needs to know the frame size of the stream for the duration of the gate close period. For simplicity we used EWMA to track the average frame size, which was adequate for the simulations, but in reality the last frame in a burst is typically smaller than the previous ones.

This detection and prediction scheme automatically adjusts itself if the source changes its behavior regarding to its period, frame size, and burstiness. In regular TAS the sources need to inform the CNC about these changes, and wait for the new schedule to be computed and disseminated.

4 Simulation Results

We examined the behavior of the ATAS with simulations in OMNeT++ to assess its viability. The OMNeT++ framework allows fast prototyping and provides several tools for state inspection and result evaluation. We reused our implementation of the TSN queueing mechanisms for the INET model set [34] of OMNeT++ that we used in our previous work [28], where we examined the robustness of TAS under various configuration patterns. The NeSTiNg model library [35] for simulating TSN in OMNeT++ was published not long after we were done with that project. We have decided not to switch over to that model library for ATAS, because our models work almost the same way, and we were more familiar with our own code.

In the simulations we used $\alpha = 0.3$ for the EWMA algorithm, and stored the number of frames received in the last $K = 5$ bursts. These values were chosen as reasonable defaults, we will see the effect of these parameters in Figs. 8 and 9.

The simulated network topology is shown in Fig. 3. It models a slice of an industrial control ring, with the process controller on the left receiving periodic status updates from the process monitoring nodes connected to the Drop switches dangling from the corresponding Ring switches. All links are 100 Mbps. We removed the ring nodes that are not part of the shortest path between these monitoring nodes and the controller. The End Station nodes (green screens) are the sources of the high priority
streams, and the BG nodes (blue screens) send low priority background streams. The control node is split into dedicated receiver nodes for each source (with matching screen colors) to separate the streams for analysis. This topology was constructed to contain all the scenarios we need to demonstrate the capabilities of ATAS.

In a real network the source would send its frames with good accuracy, and the uncertainties would mostly come from clock synchronization issues, and random delays in the network. In our simulations these uncertainties are added at the source; thus, when there is no interference, the end-to-end latency is constant, yet the predictors at the switches see random frame arrivals. The plots in this paper were generated with variation in the frame sending intervals of the high priority stream uniformly distributed in a 40 μs range, which corresponds to the transmission time of 500 bytes on the wire. The burst size was set to three frames for all streams.

Figure 4 illustrates the prediction accuracy for a single high priority stream by comparing $\tilde{d}_i + 1$ to $d_i + 1$. The small inter-arrival times are between frames within a burst, while the large ones are the times between the bursts. The first prediction happens for the second inter-arrival time, corresponding to the third frame received, and it is quite accurate in this case. The fourth frame is incorrectly predicted, because the burstiness of the stream comes unexpected the first time it happens. The rest of the arrival times are predicted with good accuracy, even the delay pattern is matched in most cases.

Figure 5 shows the end-to-end latency for the traffic of End Station 1 and BG2. This scenario was crafted for the worst case: the two streams have identical timings, but when the streams meet at Ring SW3 the low priority frames always arrive just before the high priority frames, as Fig. 5a shows. Figure 5b shows the ATAS in action: after learning from the first few frames, the rest of the high priority stream is perfectly guarded from the interference.

Figure 6 shows the effect of the prediction errors in a special case: when a frame arrives earlier than predicted, the low priority traffic can hold it back until the predicted arrival time. To illustrate this issue, in Fig. 6a we shifted the BG2 frames forward by 100 μs to make their ends meet the start of the high priority frames. Filling the open period up until the gate close time with low priority frames is a challenge, though, we had to fine-tune the timing of BG2 to see a few high priority frames shifted by them, like the one just after the 0.11 s mark. In the next two bursts the frames of BG2 arrive too late and are delayed by the high priority frames. Figure 6b shows the same scenario with guard band enabled. Since we close the gates earlier than the predicted arrival time, all frames of BG2 are pushed behind the high priority frames.

Figure 7 shows the end-to-end latencies when all three low priority streams interfere with the traffic of End Station 1. This is also a hand-crafted worst case: when Ring SW3 locks onto the stream, and starts protecting it, the decreased delay causes it to start competing with the traffic of BG1 at Ring SW2. After that competition is eliminated, the situation repeats with BG0 at Ring SW1. The learning phase gets longer with more switches along the path, but eventually all asynchronous shapers lock onto the high priority stream, and it gets protected along the whole path.

Figure 8 shows the latencies of the frames of End Station 1 when only BG2 interferes, as a function of the uniformly distributed uncertainty of the sending interval.
at the source, for two EWMA $\alpha$ values. For low uncertainty the ATAS protects the high priority stream perfectly: the minimum and maximum latencies are the same. Beyond a threshold, however, some frames arrive too late, and cannot be protected; thus, the maximum latency increases sharply. This is a rare event, though, as seen from the relatively minor increase in the average latency. In this scenario the faster convergence of the lower EWMA $\alpha$ was beneficial for both the average and the maximum latency.

Figure 9 shows how the ATAS adapts to the source changing its burstiness: it starts sending one frame per period, then at $0.11 \text{ s}$ it switches to two frames. The shaper detects this, and recalibrates itself, which takes two periods, just like at the start of the stream. At $0.12 \text{ s}$ the source goes back to sending one frame per burst. The shaper still keeps running its protection for the second frame of the burst for $K = 5$ periods, then sees that the maximum burst length is 1, marks the source as not bursty, and stops protecting the phantom of the second frame.

Figure 10 shows the end-to-end latencies when both high priority sources are active, but with different burst periodicity (1200 $\mu$s and 2200 $\mu$s). Interference between these streams is unavoidable, but it is a relatively rare event, as the first histogram shows: the minimum and the average latency are close to each other, which means that most frames were unobstructed. In the middle histogram the background traffic is enabled. It totally disrupts both high priority streams: the three have the same latency figure even though we have strict priority queueing. In the third
In the second histogram we see that ATAS protects both high priority streams: their average is almost the same as in the first histogram. Their maximum is higher now, because ATAS cannot protect their frames when they arrive with unexpected latency due to interfering with each other.

Note that when two high priority streams have different periodicity, the period of their combined arrival pattern can be longer than the maximum period of the GCL; thus, TAS cannot always tailor the gating to match the traffic, and some high priority frames will be held back by the gates.

We repeated the experiment shown in Fig. 10 with different prediction formulas. As we have mentioned in Sect. 3.1, the goal of the negative correlation-based predictor formula is to reduce the influence of randomly occurring large delays on the predictions, and this is exactly the scenario to test that claim. The histograms for the distribution of the prediction error $\tilde{d}_{i+1} = \{d_i, \bar{T}_i, 2\bar{T}_i - d_i\}$ are shown in Fig. 11 for prediction formulas $\tilde{d}_{i+1} = \{d_i, \bar{T}_i, 2\bar{T}_i - d_i\}$. The most important difference among the three is on the right side: with $2\bar{T}_i - d_i$, the outliers are much smaller, which means the predictor does not overestimate $x_{i+1}$ as much, and we can use shorter guard band.

Histograms for prediction formulas $\tilde{d}_{i+1} = \{d_i, \bar{T}_i, 2\bar{T}_i - d_i\}$ are shown in Fig. 11 for prediction formulas $\tilde{d}_{i+1} = \{d_i, \bar{T}_i, 2\bar{T}_i - d_i\}$. The most important difference among the three is on the right side: with $2\bar{T}_i - d_i$, the outliers are much smaller, which means the predictor does not overestimate $x_{i+1}$ as much, and we can use shorter guard band.

Fig. 5 End-to-end latencies with gating a disabled and b enabled

Fig. 6 End-to-end latencies with guard band a disabled and b enabled
5 Related Work

Our work is not the first one to improve upon the 802.1Qbv TAS architecture. There are several papers on enhancing the central network control, e.g., by using auto-configuration based on preexisting knowledge about the sources [8], or developing incremental algorithms [36] for real-time computing of the new gate schedule when a stream enters or leaves the network.

There are also proposals for improving the network throughput via local optimizations to the gate schedule. In the gate-shrunk TAS model [37] the high priority traffic window has no predefined length. Instead, the talker sends a GS frame at the end of its burst, which signals to the bridges to close the high priority window, thus letting more low priority traffic through. Obviously, this only works for one high priority talker.

The size-based queuing (SBQ) method [31] attempts to reduce the link underutilization prior to closing the traffic gate when the first frame in the queue does not fit. It reorders the queued frames to find one that fits. Note that reordering the frames of a stream is bad for real-time applications, and most network equipment take extra care to avoid reordering frames.
In [38] the elastic queueing engine (EQE) was proposed, which uses dynamic queue lengths, dynamic internal priority values, and adaptive GCL to avoid dropping frames due to queue size limitations. This is accompanied by a hardware co-processor design for real-time operation in in-vehicle networks.

Replacing the TAS with another queueing architecture, which is not so sensitive to time synchronization, has also been proposed in the TSN Task Group. We will review two of them below.

Cyclic queuing and forwarding (CQF), formerly known as peristaltic shaper, has been published in amendment 802.1Qch [12]. It uses pairs of queues for each traffic class, with traffic gates opened in an alternating pattern: incoming frames are directed into the closed queue, while the open queue releases the frames it has gathered, and they switch roles periodically. This reshapes the traffic, eliminating the

![Fig. 9](image-url) End-to-end latency when the source starts and stops being bursty

![Fig. 10](image-url) End-to-end latency comparison with two high priority sources and one background

In [38] the elastic queueing engine (EQE) was proposed, which uses dynamic queue lengths, dynamic internal priority values, and adaptive GCL to avoid dropping frames due to queue size limitations. This is accompanied by a hardware co-processor design for real-time operation in in-vehicle networks.

Replacing the TAS with another queueing architecture, which is not so sensitive to time synchronization, has also been proposed in the TSN Task Group. We will review two of them below.

Cyclic queuing and forwarding (CQF), formerly known as peristaltic shaper, has been published in amendment 802.1Qch [12]. It uses pairs of queues for each traffic class, with traffic gates opened in an alternating pattern: incoming frames are directed into the closed queue, while the open queue releases the frames it has gathered, and they switch roles periodically. This reshapes the traffic, eliminating the

![Fig. 11](image-url) Prediction error comparison with different prediction formulas
previously accumulated random delays. The downsides of this are the additional delay of at least one cycle, and a newly introduced pseudorandom delay due to the cycle timer being independent of the traffic periodicity. For optimal performance it is advised to synchronize the CQF shapers to a global clock.

The asynchronous traffic shaper (ATS) project added the urgency-based scheduler (UBS) to the bridge architecture in amendment 802.1Qcr [1]. UBS does not need a global clock, because it does not do traffic gating. It redesigns the traffic at each hop with a per-stream token bucket scheduler, thereby regulating the burstiness of the streams. The advantage of UBS is that it performs per-stream token bucket shaping while only doing per-class queueing. UBS is not a replacement of TAS (the two can be used together), rather, an improvement over the credit-based scheduler (CBS) of amendment 802.1Qav by the AVB Task Group. UBS can increase link utilization by reducing contention [39], but it does not protect the different priority streams from each other like TAS. The additional delay introduced by the reshaping can hurt ULL applications, but when all of the streams conform to their registered burst characteristics, UBS is equivalent to strict priority scheduling, and no reshaping is performed.

6 Conclusions

In this paper we have proposed the ATAS, a novel mechanism for controlling the TASs of IEEE 802.1 TSN bridges without central control, clock synchronization, network planning, or prior configuration. Our system tracks the properties of the high priority streams going through the bridge, predicts the arrival of the next frame of each stream, and based on the predictions it closes the traffic gates of the low priority queues to eliminate interference on the high priority streams.

We used a simple, one step, negative correlation-based predictor that learns the properties of the stream quickly, and starts to give useable predictions after two received frames. It can handle a wider variety of traffic patterns than TAS, and it is able to quickly adapt to changes in the behavior of the source. It also has low memory requirements and computation complexity. With ATAS it is easy to create a network that can service applications that require ultra low latency, e.g., automotive networks, 5G fronthaul, and tactile Internet.

We have used simulations to show the viability of ATAS: it can efficiently protect the high priority streams even with the simple stream property tracking and gate control heuristics that we used. As a next step we plan on extending these heuristics with support for more frame arrival patterns to offer protection for a wider variety of high priority stream types. Furthermore, we expect that machine learning techniques can help us develop methods for protecting streams that have been disturbed by interference with other traffic at earlier parts of the network.

Funding Open access funding provided by Budapest University of Technology and Economics.
References

1. IEEE Standard 802.1Qcr: IEEE Standard for Bridges and Bridged Networks—Amendment: Asynchronous Traffic Shaping (2019)
2. IEEE Standard 802.3-2015: IEEE Standard for Ethernet (2015)
3. IEEE Standard 802.1Q-2018: IEEE Standard for Local and Metropolitan Area Networks: Bridges and Bridged Networks (2018)
4. Metro Ethernet Forum: MEF 12.2 Carrier Ethernet Network Architecture Framework Part 2: Ethernet Services Layer. Metro Ethernet Forum. https://www.mef.net/resources/mef-12-2-carrier-ethernet-network-architecture-framework-part-2-ethernet-services-layer/ (2014). Accessed 13 January 2022
5. IEEE Standard 802.11-2016: IEEE Standard for Local and Metropolitan Area Networks: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications (2016)
6. IEEE Standard 802.1Qav: IEEE Standard for Virtual Bridged Local Area Networks—Amendment: Forwarding and Queuing Enhancements for Time-Sensitive Streams (2009)
7. Texas Instruments: Time-Sensitive Networking for Industrial Automation. Texas Instruments http://www.ti.com/lit/wp/spry316a/spry316a.pdf (2018). Accessed 14 July 2021
8. Farzaneh, M.H., Knoll, A.: An ontology-based plug-and-play approach for in-vehicle time-sensitive networking (TSN). In: Proceedings of IEMCON’16, 2016, Vancouver, BC, Canada (2016)
9. IEEE Standard 802.1CM: IEEE Standard for Local and Metropolitan Area Networks—Time-Sensitive Networking for Fronthaul (2018)
10. Fettweis, G.P.: The tactile Internet: applications and challenges. Veh. Technol. Mag. 9(1), 64–70 (2014)
11. IEEE Standard 802.1Qbv: IEEE Standard for Bridges and Bridged Networks—Amendment: Enhancements for Scheduled Traffic (2015)
12. IEEE Standard 802.1Qch: IEEE Standard for Bridges and Bridged Networks—Amendment: Cyclic Queuing and Forwarding (2017)
13. IEEE Standard 802.1AS: IEEE Standard for Local and Metropolitan Area Networks—Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks (2011)
14. IEEE Standard 802.1Qat: IEEE Standard for Virtual Bridged Local Area Networks—Amendment: Stream Reservation Protocol (SRP) (2010)
15. IEEE Standard 802.1Qcc: IEEE Standard for Local and Metropolitan Area Networks—Amendment: Stream Reservation Protocol (SRP) Enhancements and Performance Improvements (2018)
16. IEEE Standard 802.1Qbu: IEEE Standard for Bridges and Bridged Networks—Amendment: Frame Preemption (2016)
17. IEEE Standard 802.3br: IEEE Standard for Ethernet—Amendment: Specification and Management Parameters for Interspersing Express Traffic (2016)
18. NetTimeLogic: TsnEndNode Reference Manual. https://www.nettimelogic.com/resources/Red_TsnEndNode_ReferenceManual.pdf (2021). Accessed 8 February 2022
19. Máté, M., Simon, C., Malisz, M.: Asynchronous time-aware shaper for time-sensitive networking. In: Proceedings of CNSM’21, 2021, Izmir, Turkey (2021)
20. Meyer, P., Steinbach, T., Korf, F., Schmidt, T.C.: Extending IEEE 802.1 AVB with time-triggered scheduling: a simulation study of the coexistence of synchronous and asynchronous traffic. In: Proceedings of VTC’13, 2013, Boston, MA, USA (2013)
21. Cisco: Time-Sensitive Networking: A Technical Introduction. Cisco. https://www.cisco.com/c/dam/en/us/solutions/collateral/industry-solutions/white-paper-c11-738950.pdf (2017). Accessed 14 July 2021
22. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Berlin (2004)
23. Schweissguth, E., Danielis, F., Timmermann, D., Parzyjegla, H., Mühl, G.: ILP-based joint routing and scheduling for time-triggered networks. In: Proceedings of RTNS ’17, 2017, Grenoble, France (2017)
24. Dürr, F., Nayak, N.G.: No-wait packet scheduling for IEEE time-sensitive networks (TSN). In: Proceedings of RTNS ’16, 2016, Brest, France (2016)
25. Vlk, M., Hanzálek, Z., Brejchová, K., Tang, S., Bhattacharjee, S., Fu, S.: Enhancing schedulability and throughput of time-triggered traffic in IEEE 802.1Qbv time-sensitive networks. IEEE Trans. Commun. 68(11), 7023–7038 (2020)
26. Oliver, R.S., Craciunas, S.S., Steiner, W.: IEEE 802.1Qbv gate control list synthesis using array theory encoding. In: Proceedings of RTAS’18, 2018, Porto, Portugal (2018)
27. Xu, L., Xu, Q., Zhang, Y., Zhang, J., Chen, C.: Co-design approach of scheduling and routing in time sensitive networking. In: Proceedings of ICPS’20, 2020, Tampere, Finland (2020)
28. Simon, C., Maliosz, M., Máté, M.: Design aspects of low-latency services with time-sensitive networking. IEEE Commun. Stand. Mag. 2(2), 48–54 (2018)
29. Weissengberg, M., Głąbowski, M., Hanczewski, S., Stasiak, M., Zwierzykowski, P., Bai, V.: Traffic modeling in industrial Ethernet networks. Int. J. Electron. Telecommun. 66(1), 145–153 (2020)
30. Kailath, T., Hassibi, B., Sayed, A.H.: Linear Estimation. Prentice Hall, Hoboken (2000)
31. Heilmann, F., Fohler, G.: Size-based queuing: an approach to improve bandwidth utilization in TSN networks. In: Proceedings of RTN’18, 2018, Barcelona, Spain (2018)
32. IEEE Standard 802.1Qci: IEEE Standard for Bridges and Bridged Networks—Amendment: Per-Stream Filtering and Policing (2017)
33. IEEE Standard 802.1Qcb: IEEE Standard for Local and Metropolitan Area Networks—Frame Replication and Elimination for Reliability (2017)
34. INET Framework. https://inet.omnetpp.org/ Accessed 17 May 2021
35. Falk, J., Hellmanns, D., Carabelli, B., Nayak, N., Dürr, F., Kehrer, S., Rothermel, K.: NeSTiNg: simulating IEEE time-sensitive networking (TSN) in OMNeT++. In: Proceedings of NetSys’19, Garching b, 2019, München, Germany (2019)
36. Nayak, N.G., Dürr, F., Rothermel, K.: Incremental flow scheduling and routing in time-sensitive software-defined networks. Trans. Ind. Inform. 14(5), 2066–2075 (2018)
37. Hisano, D., Nakayama, Y., Kubo, T., Shimizu, T., Nakamura, H., Terada, J., Otaka, A.: Gate-shrunk time aware shaper: low-latency converged network for 5G Fronthaul and M2M services. In: Proceedings of GLOBECOM’17, 2017, Singapore (2017)
38. Mariño, A.G., Fons, F., Gharba, A., Ming, L., Arostegui, J.M.M.: Elastic queueing engine for time sensitive networking. In: Proceedings of VTC’21, 2021, Helsinki, Finland (2021)
39. Prados-Garzon, J., Chinchilla-Romero, L., Ameigeiras, P., Muñoz, P., Lopez-Soler, J.M.: Asynchronous time-sensitive networking for industrial networks. In: Proceedings of EuCNC’21, 2021, Porto, Portugal (2021)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Miklós Máté received his MSc (2007) and PhD (2019) Degrees in Electric Engineering in the field of Infocommunication Systems at Budapest University of Technology and Economics (BME), Hungary. He is an Assistant Professor at Department of Telecommunication and Media Informatics of the same university. His research interests include intelligent transportation systems, cloud infrastructures and reliability in distributed systems.

Csaba Simon is a Software Engineer and he earned his PhD in 2012 at the Doctoral School of Informatics of the Budapest University of Technology and Economics and currently works as an Associate Professor at the Department of Telecommunication and Media Informatics of the same university. His research area includes the topics of Future Internet, 5G networks and cloud systems. He participated in numerous
national and international projects in the fields of network resource management, mobility management and smart content delivery.

Markosz Maliosz received his MSc (1998) and PhD (2006) Degrees in Computer Science in the field of Infocommunication Systems at Budapest University of Technology and Economics (BME), Hungary. He is an Associate Professor in the High-Speed Networks Laboratory at the Department of Telecommunication and Media Informatics, BME. His research interests include virtual, cloud and sensor networking along with optimization techniques. He participated in numerous national and international projects in the fields of network resource management, multimedia and smart content delivery.