Usage of TSN Per-Stream Filtering and Policing

Marc Boyer

Usage of TSN Per-Stream Filtering and Policing

Marc Boyer
ONERA/DTIS, Université de Toulouse
F-31055 Toulouse – France
https://www.onera.fr/staff/marc-boyer
Marc.Boyer@onera.fr

Abstract

The Per Stream Filtering and Policing mechanism (PSFP) defined in amendment 28 of the 802.1Q standard presents a set of mechanisms designed to protect the network from faults other than losses (especially to protect from a switch sending more data than expected, leading to latency increase and buffer overflow).

This report presents a comprehensive survey of the mechanisms, gives advice on its configuration, but also highlight some limits of the current standard.

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

Keywords and phrases TSN, Time Sensitive Networking, PSFP, Per-stream filtering and policing, 802.1Qci

Contents

History

Version	Date	Comment	Sections
1	06/27/2023	Initial version	-
2	07/10/2023	Precision on the number of meters Add of Thanks section	1, 2.3, 3.6.6, 3.6.7 4
3	6/9/2023	Typos, Reference to `portMediaDependentOverhead` Better TAS error scenario Precisions on steam_handle and input port	3.6.4 3.6.3 2.1 (new), 2.2, 3.6.1
4	8/11/2023	Update of Figure 10, Missing references.	

1 Introduction

The 802.1 TSN working group is in charge of defining as set of extensions transforming Ethernet into a reliable real-time network.

Among other, the Per Stream Filtering and Policing mechanism has been introduced in [10], and now part of [6], to increase the reliability of a TSN network.

This report presents PSFP and give some hints on how to configure it. Note that a few changes have been done between [10] and [6], and the version presented here is the last one.

This report also shows that PSFP does not fulfil the requirements expected by critical systems, and propose some enhancements to the standard.

Section 2 presents a PSFP overview. Recommendations on PSFP usage are given in Section 3. In particular, Section 3.6 focuses on the possibility offered by PSFP to contain local faults. All along the paper, when limitations are listed, some trade-off are presented and even some extensions to the standard mechanisms are suggested.

2 PSFP presentation

The Per Stream Filtering and Policing mechanism has been introduced in [10]. This mechanism is set as the last step of the filtering pipe between reception port and the queuing of frames, as shown in [10, Fig. 8-11].

The “Flow metering” stage is going to check each frame, to let it pass or drop it, or even modify its local priority, and to update some related counters. It is called Per-stream filtering and policing (PSFP).

PSFP is itself a pipeline of tests and updates, based on three tables:

1. the “Stream filter” instance table,
2. the “Stream gate” instance table, and
3. the “Flow meter” instance table.

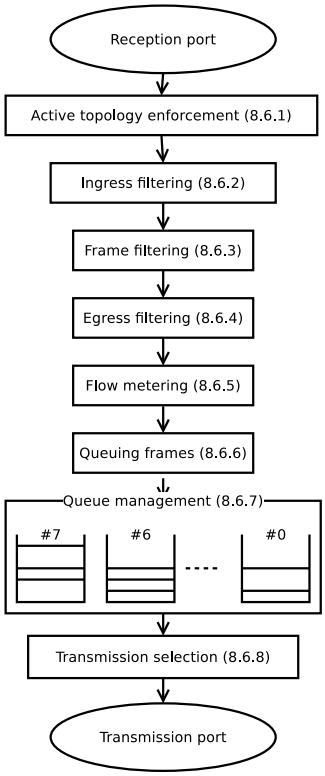

Figure 1 Forwarding process functions, from [10, Fig. 8-11].

These three tables are not independent.

Also note that a flow in PSPF is identified by its stream_handle, has defined in [3].

2.1 On stream, streamID and stream_handle

A *stream* is a *"unidirectional flow of data (e.g., audio and/or video) from a Talker to one or more Listeners"* [6, § 3.259], which is identified by a unique 64-bit long *StreamID* [6, § 3.261].

But a switch/bridge will not use this 64-bits field, but a *stream_handle*, associated to each packet, used to identify *"the Stream to which the packet belongs"* [3, § 6.1]. This stream_handle is a 32-bits unsigned integer [6, § 48.6.11]. This handle is a local value, i.e. there is no requirement that the same packet should receive the same stream_handle on each switch along its path.

The stream_handle is generated by a stream identification function, placed just above the MAC layer [3, Figure 6-1]. There are four kinds of stream identification function, and each function generated the stream_handle of a packet "based on" some packet parameters, listed in Table 1.

It is still unclear for us if the stream identification function can take into account the reception port of the packet. In other words, it the stream identification function able to give different stream_handle to two identical packets received on different ports.

Note 1 of [6, § 8.6.5.3] claims that "the use of stream_handle and priority, along with the wild-carding rules previously stated, allow configuration possibilities that go beyond the selection of individual streams, for example [...] per-priority per-reception Port filtering and policing can be configured using these rules.". But this is, up to our knowledge, the only

Stream identification function	Examines
Null Stream identification	destination_address, vlan_identifier
Source MAC and VLAN Stream identification	source_address, vlan_identifier
Active Destination MAC and VLAN Stream identification	destination_address, vlan_identifier
IP Stream identification	destination_address, vlan_identifier, IP source address, IP destination address, DSCP, IP next protocol, source port, destination port

Table 1 Stream identification function list (from [3, Table 6-1])

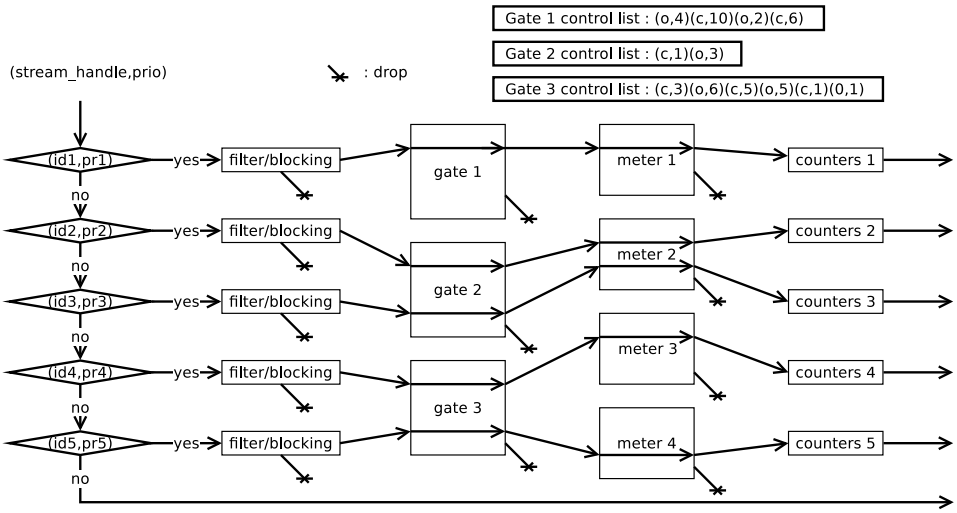

Figure 2 PSFP architecture, from [10, Fig. 8-12]

reference to the reception port related to stream_handle.

As will be presented in Section 2.3, PSPF has no way to know the input port of a packet if it is not encoded in the stream_handle, and this may be of interest when considering fault tolerance (cf. Section 3.6.6).

2.2 Per-stream filtering and policing global architecture

The global architecture of PSFP is presented in [10, Fig. 8-12], but we chose a different representation, given in Figure 2.

Each frame will cross an (ordered) list of tests (based on its stream_handle and priority), each test is part of a "stream filter". At the first successful match, the frame pass some stream filter rules and, if succeed, is forwarded to some gate, that may also either drop or update and forward the frame to a meter, that may also either drop or update and forward the frame. During the whole process, some metric counters are updated. Some internal counters of the gate and meter may also been updated at frame crossing. This means that when several flows share the same gate or meter, they may also have to share some budget.

Note that the relative order the stream filter is of importance (since the first match apply), but not the one between stream gates and flow meter. It has been drawn in Figure 2 in a way that avoid arrow crossing, but any other numbering will have the same effects (if of

course the identifiers in the stream filters are updates accordingly).

That is to say, one can

- transform gate stream 1 into gate stream 4,
- update, in stream filter 1, the gate identifier from 1 to 4,
- transform meter 3 into flow meter 5,
- update, in stream filter 4, the meter identifier from 3 to 5,

and get exactly the same behaviour. Whereas inverting stream filters 1 and 2 may change the behaviour if some flow may match both.

2.3 The stream filter instance table

The stream filter instance table is a table of stream filter. Each stream filter is identified by an integer identifier (which is of course unique). The order between the identifiers has an importance since a frame will select the first matching stream filter, in ascending order (i.e. from smaller values to larger values).

A stream filter is made of several components, listed in the following.

- a pair made of *stream_handle specification, priority specification*: the stream_handle specification is either a stream_handle value or a wildcard, and the priority value is either a priority (an integer value in $[0, 7]$) or a wildcard.

  A frame matches the stream_handle specification either if the specification is the wildcard or if the stream_handle of the frame equals the stream_handle of the rule. The same applies for the priority.

  Examples: (2305,3) matches the specification pairs (2305,3), (2305,*), (*,3) and (*,*).

  If a frame matches no rule, it is forwarded to the queuing process as if PSFP was not implemented in the bridge.
- a *MaxSDUSize*, that drops frames exceeding this size (can be disabled by setting value 0)
- a *stream gate instance identifier* that identifies the stream gate to forward the frame if not dropped. A gate can be shared by several filters.
- an optional *FlowMeterInstance* that identifies the flow meter to forward the frame after the stream gate, if not dropped by the gate. A flow meter can be shared by several filters. Note that this part have been clarified when inserting [10] into [6][1].
  Having several Maximum SDU size is nonsense since it is sufficient to keep the smallest one.
- a pair of Boolean *StreamBlockedDueToOversizeFrameEnable, StreamBlockedDueToOversizeFrame*: if the *StreamBlockedDueToOversizeFrameEnable* is true, then, if a frame is dropped due to a size larger than the Maximum SDU size allowed by a filter, then the *StreamBlockedDueToOversizeFrame* is set to true. When *StreamBlockedDueToOversizeFrame* is true, all frame passing the stream filter are dropped (as if the Maximum SDU size was set to 0).
  The standard does not specify any automatic way to set *StreamBlockedDueToOversizeFrame* to false, i.e. to re-open the stream filter. Such operation is a maintenance operation

[1] In [10], the condition was referring to "zero or more *filter specification*" allowing to have several flow meter instance for a single filter. The ability to have a sequence of several flow meters associated with one stream filter would have been be beneficial to prevent fault propagation, as will be presented in Sections 3.6.6 and 3.6.7.

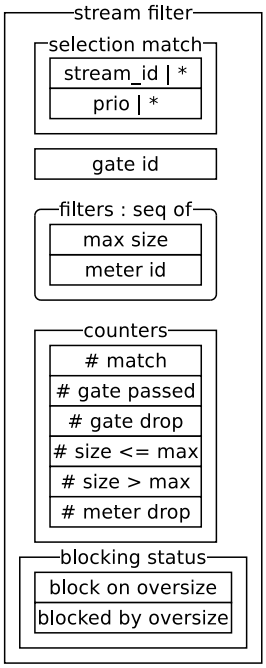

Figure 3 A single stream filter instance

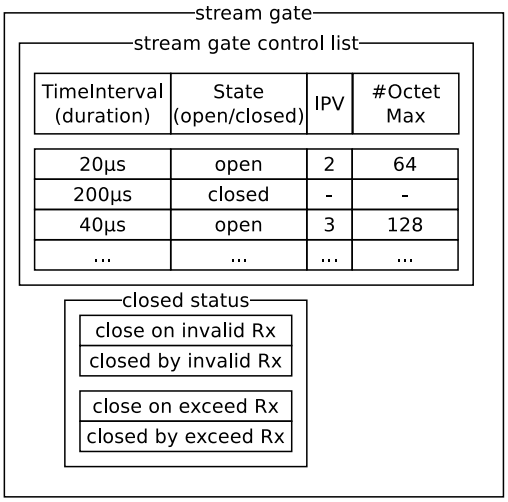

Figure 4 A single stream gate instance

(allowed, since this parameter has Read/Write access, [10, Table 13-31] and the MIB object *ieee8021PSFPStreamBlockedDueToOversizeFrame* in [10, Table 17-30, § 17.7.24]).

- a set of counters:
  - *MatchingFramesCount*: number of frames that have matched the pair *stream_handle specification, priority specification*
  - *PassingFramesCount*: number of frames that have passed the stream gate
  - *NotPassingFramesCount*: number of frames that did not passed the stream gate
  - *PassingSDUCount*: number of frames that have passed the Maximum SDU size filter
  - *NotPassingSDUCount*: number of frames that did not passed the Maximum SDU size filter
  - *REDFramesCount*: number of frames dropped by the flow meter

Also note that the ATS scheduler is also part of the filter instance, even if this part is not addressed by this report.

On group of flows

Note that there is now way to specify that a stream filter must be applied to a set of flows. Then, set-based handling must be done using the priority fields of frames or by having a group of stream filters.

Consider for example a CQF class. As will be shown in Section 3.3, the behaviour of a CFQ class relies on PSFP mechanism. Then, if a CFQ class handle 100 flows, the PSFP filter must be able to identify these 100 flows. It could be either by allocating a priority to the CQF class: all flows with this priority belongs to this class. The other solution consists in being able to identify each stream, i.e. having one stream filter instance per flow.

This tends to require large-capacity for stream filter instance tables.

On relative order between tests

The order in which the different tests are applied seems not to be fixed in the text of the standard. Our interpretation (stating that the test on maximum size is applied before the stream gate that is applied before the stream meter) is based on [10, Fig. 8-12], represented in Figure 2. This is confirmed by [6, Fig. 8-13], reported in this report in Figure 17.

Some tests on the counters value may confirm or inform this order. For example, if the Maximum SDU size filter is just before the stream gate, all frames going out the size filter will be forwarded to the gate, where they will either forwarded or dropped. In this case, the following relation should hold

$$PassingSDUCount = PassingFramesCount + NotPassingFramesCount. \quad (1)$$

The same way, if the Maximum SDU filter is the first test, after the section match, and if *StreamBlockedDueToOversizeFrameEnable* is not set, the following relation should hold

$$MatchingFramesCount = PassingSDUCount + NotPassingSDUCount. \quad (2)$$

2.4 The stream gate instance table

The stream gate instance table is a table of stream gate. Each stream gate is identified by an integer identifier (which is of course unique). Conversely to the stream filter table, the order between the identifiers has no importance. A given stream gate may be used (or shared, depending on the point of view) by several stream filters.

A typical stream gate instance is made of

- a *stream control list*, which is a table. At any moment, a single line/entry is active. Each entry of the table is made of
  - a *StreamGateState* which is either open or closed,
  - a *TimeInterval* that states how long this line must be active (at end of active time, the next line is activated, in a cyclic way)
    These two entries are the same than in the Gate Control List (GCL) defined in [8].
    The semantics is slightly different: when the gate is in a closed state, the frames are dropped by PSFP, while a closed gate in an output port just blocks the head of queue up to next opening.
  - an *IPV* (Internal Priority Value) that (if set) locally overrides the frame priority value when selecting in which traffic class (i.e. queue) the frame must be stored,
  - an *IntervalOctetMax* that fixes an upper bound on the number of MSDU octets that can pass the gate while this entry is active (the difference between frames, SDU and MSDU will be given in Section 3.2).
    Note that this value test is implemented with a counter that is shared by all the stream filters using this stream gate.
- a pair of Booleans *GateClosedDueToInvalidRxEnable*, *GateClosedDueToInvalidRx*: if *GateClosedDueToInvalidRxEnable* is true, then, then, if a frame is received by the gate when the gate is closed, then *GateClosedDueToInvalidRx* is set to true and all frames forwarded to this gate will be dropped, as if the gate was in the closed state.
  The standard does not specify any automatic way to set *GateClosedDueToInvalidRx* to false, i.e. to re-open the gate. Such operation is a maintenance operation (allowed, since this parameter has Read/Write access, [10, Table 12-32] and the MIB object *ieee8021PSFPStreamBlockedDueToOversizeFrame* in [10, Table 17-30, § 17.7.24]).

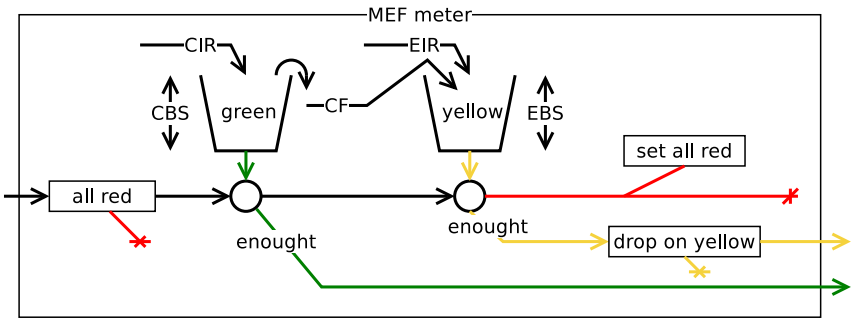

Figure 5 A single stream meter instance

- a pair of Booleans *GateClosedDueToOctetsExceededEnable*, *GateClosedDueToOctetsExceeded*: like for the previous pair, if *GateClosedDueToOctetsExceededEnable* is true, and if too much octets are received in on interval, then the *GateClosedDueToOctetsExceeded* is set to true and all frames forwarded to this gate will be dropped, as if the gate was in the closed state.

  The does not specify any automatic way to set *GateClosedDueToOctetsExceeded* to false, but the same as for *GateClosedDueToInvalidRx* apply.

A stream gate instance can also have no *stream control list*, but then it must have a state (open or closed) and an (optionnal) IPV. We will call such a gate a static gate (this name is from the author, and does not appear in the standard).

Also note that each stream gate instance has its own time base, the *OperBaseTime* that is the origin of time of the stream control list, the reference offset (like for GCL).

In [5], this parameter is renamed as *StreamGateOperBaseTime*.

2.5 The stream meter instance table

The stream meter instance table is a table of stream meter. Each stream meter is identified by an integer identifier (which is of course unique). Like for stream gate table, the order between the identifiers has no importance. A given stream meter may be used (or shared, depending on the point of view) by several stream filters.

A stream meter instance is a sequence of two token-buckets (used as classifier), and the content of the token-buckets is shared by all flows crossing the token-buckets, i.e. the stream meter instance is shared by all stream filters using this stream meter.

A stream meter instance has several parameters, given in [10], but the semantics of some of these parameters is defined in another one, [22]. Note that [22] take into account the Ethernet frame size (called *length*) to test the token-buckets. Then, it does not account for the physical layer overhead (for example, the preamble of an Ethernet Packet and the Inter Packet Gap, cf. Section 3.2).

Each frame crossing a meter crosses the sequence presented in Figure 5. During this process, framed will be virtually marked as green, yellow or red. This notion of color is used only in this meter sub-system and is not visible outside (and in particular not written in the frame).

The parameters of a stream meter are:

- a pair of parameters, *Committed information rate (CIR)* and *Committed burst size (CBS)* are respectively the replenishment rate and the maximal size of the first token-bucket.

If there are enough tokens when a frame tests this first token-bucket, the frame is simplify forwarded. As in any token-bucket, the number of tokens is decremented by the size of the frame.
Otherwise, the frame is forwarded to the next token-bucket.
Note that the abbreviation CBS also denotes the Credit-Based Shaper shaping algorithms defined in AVB/TSN [1].

- a second pair of parameters, *Excess Information Rate (EIR)*, and *Excess burst size (EBS)* are respectively the replenishment rate and the maximal size of the second token-bucket. If there are enough tokens when a frame tests this first token-bucket, the frame is marked as yellow. Otherwise, the frame is marked as red.
  Red frames are dropped.
  The handling of yellow frames depends on the next parameter, *DropOnYellow*.
- a Boolean parameter *DropOnYellow* specifies what to do with yellow frames. If *DropOnYellow* is true, yellow frames will be dropped. If *DropOnYellow* is false, the *drop_eligible* bit of the frame is set to true.
  The semantics of this *drop_eligible* bit is given in [4, § 8.6.7]. It just says that any "queue management algorithm that attempts to improve the QoS provided by deterministically or probabilistically managing the queue depth" can drop frames, and that the *drop_eligible* bit must not decrease the probability of dropping a frame.
- a pair or Boolean parameter *MarkAllFramesRedEnable*, *MarkAllFramesRed* has the same kind of semantics of the other Boolean pairs: if *MarkAllFramesRedEnable* is true, once a frame is marked red, *MarkAllFramesRed* is set to true, and all frames are dropped.
- a last parameter *color mode (CM)* has only two possible values, *color-aware* or *color-blind*. In [22, Fig. 27], if the color mode is *color-aware*, only green frames can test the green (committed) token-bucket, the yellow frames are directly forwarded to the yellow (exceed) token-bucket. But in [10], frames have no color at meter entry (in fact, the term "yellow" does not appear in the whole document [4], except as internal value of PSFP).
  This have been signalled in 802.1 maintenance database as item 361, and solved by stating that if the incoming frame has drop_eligibile field true, then the frame is yellow, and green otherwise.

Note that it would have been of interest to allow the PSFP to have an IPV associated to the colour of the frame. In some context, it could have been better to set a yellow or red frame in a low priority queue than to drop it. This possibility is not offered by TSN.

3 PSFP usage

This aim of this section is to provide a guideline on the usages of PSFP.

3.1 A global remark on the stream control lists

As presented in Section 2.4, a typical stream gate has a stream control list (SCL), which is a table of states, executed in a cyclic way, with a static duration for each state. Each state specifies a gate state (open or close), an optional IPV and a maximal amount of octets.

This system is quite close to the gate control list (GCL) of [8] but is more efficient on two aspects: the interleaving and the hyper-period.

Also note that the notions presented in the next sections are for illustrative purpose, to help the reader to see the differences. They are not a qualitative comparison, since SCL and GCL are for different purposes.

SCL 1

Duration	Status
6	Closed
4	Open

SCL 2

Duration	Status
4	Closed
6	Open

GCL

Duration	Status (1,2)
4	(Closed,Closed)
2	(Closed,Open)
4	(Open, Open)

Table 2 One list-based example of interleaving of SCL in GCL.

Figure 6 Two time-based examples of interleaving of SCL in GCL.

For example, one may consider "exclusive gating" for GCL, i.e. the fact that when the gate of a traffic class is open, all others are closed (this is a common assumption to implement a Time Aware Shaper – TAS). Whereas this does not hold for SCL, designed to check arrival patterns.

3.1.1 Interleaving

Let start with the creation of lines due to interleaving. To illustrate it, consider simply two gates, each with its own cyclic behaviour. Consider that both gates have the same period, e.g. 10, and only two states, encoded each with two lines. Consider that the first gate is closed between 0 and 6, and open between 6 and 10, whereas the second is closed between 0 and 4 and open between 4 and 10. Then, the resulting GCL has three lines, from 0 to 4 where both gates are closes, from 4 to 6, where the first is closed and the second is open, and a third one, from 6 to 10, where both are open, as encoded in Table 2. A time-based interpretation is represented on the left part of Figure 6. A more complex situation is given on right part of this Figure (but the table-based representation is not given).

In fact, merging two SCL of same period and respective length n and n' lead to a GCL whose size is between $\max(n, n')$ and $n + n' - 2$.

3.1.2 Hyper-period

A second difference between the usage of SCL and GCL appears when considering systems with different period. A GCL have a unique period, that must represent all behaviours, whereas each SCL can have its own period, independently of the others.

To illustrate it, consider simply two flows or classes, each with its own cyclic behaviour, or respective period T and T'.

In a GCL, since both behaviours must be captured in the same table, the GCL length must be the least common multiple (aka hyper-period) of T_1 and T_2. Let k_1, k_2 be the minimal integers such that $k_1 T_1 = k_2 T_2$, then if the behaviour of T requires n lines, the GCL must have approximately kn lines[2], whereas a SCL requires only n lines. Then, the encoding

[2] It is not exactly, since if the first and the last line represent the same state, when unrolling, the last

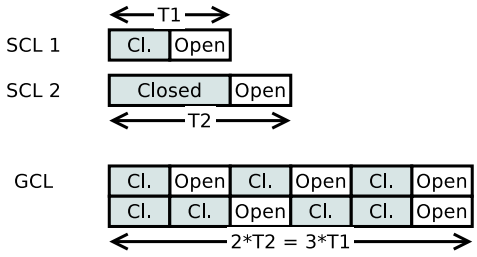

Figure 7 Two time-based examples of merging of two SCL with different periods in a GCL.

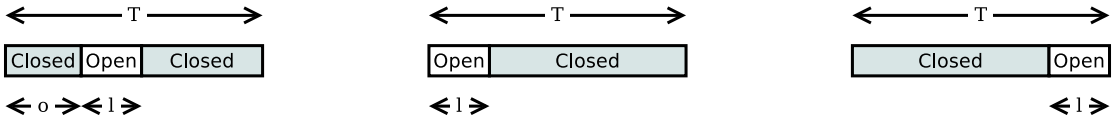

Figure 8 Three SCLs representing the same behaviour with three different time base.

as two distinct SCL requires $n_1 + n_2$ lines, whereas the encoding in a single GCL require about $k_1 n_1$ or $k_2 n2$ lines.

This difference is illustrated in Figure 7.

3.1.3 The choice of the time base

We have shown in the previous paragraphs that an SCL can be build mostly independently of other SCLs, contrary to the case of a GCL that must find an hyper-period and handle some interleaving between the behaviours of each gate.

In the previous paragraphs, no origin of time was mentioned. But checking that a periodic message is received in the expected time window is not only a question of period but also of origin of time.

Consider a periodic message sent by the previous node with period T and some offset o. If we neglect propagation time and clock accuracy (this will be addressed further), this behaviour can be captured by a stream gate instance with a null time base (i.e. the same time reference as the local clock) and three entries in the SCL, as illustrated in Figure 8, left part. But one can also set *OperBaseTime* equals to o, and use only two entries, like in the middle of Figure 8, or symmetrically, set set *OperBaseTime* equals to $T - o$, use only two entries, like in right part of Figure 8. Of course, each value of the *OperBaseTime* (modulo the period T) will lead to a different SCL. Only three representative ones have been illustrated.

3.2 On sizes

The size of a frame is of importance mainly in two aspects: the time required to send a frame, and the memory required to store that frame.

But the exact meaning of the word “frame” is not exactly the same depending on the authors. Then, this part will recall the naming used in IEEE standards, and also introduce some specific terms.

In this report, we consider only basic frames, not envelope frames neither jumbo frames.

state and the first state of two consecutive cycles can be merged into a single one.

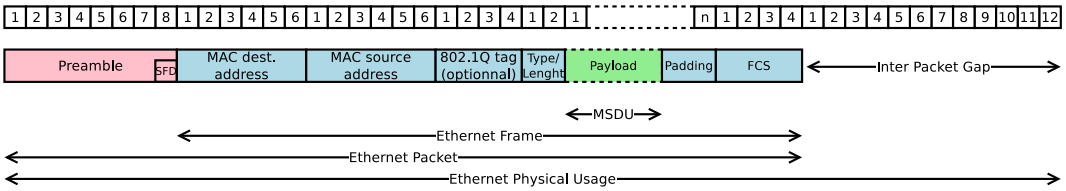

Figure 9 Ethernet frame and Ethernet packet (the value of n depends on the kind of frame).

3.2.1 On Ethernet frame sizes

3.2.1.1 Vocabulary: Ethernet frame, Ethernet Packet, MSDU, SDU, Physical usage

Formally speaking, an *Ethernet frame* (see [20, § 1.4.315]) is a Layer 2 PDU, made of

- a destination MAC address (6 bytes),
- a source MAC address (6 bytes),
- zero or more 802.1Q option or header (4 bytes each), made by 2 bytes of option type and 2 bytes of "content". The most known one is the 802.1Q header, using the 16 bits of content to encode a priority (PCP – Priority Code Point) on 3 bits, the drop eligible indicator (DEI) on 1 bit, and the 12 resulting bits encoding the VLAN Identifier (VID)
- a 2 bytes field representing either the EtherType or the size of the frame
- the payload plus some optional padding (the padding enforce the Ethernet frame to reach 64 bytes, if necessary – the payload must be at most 1500 bytes, cf. [20, Table 4–2]).
- an error detecting code (Frame Check Sequence, FCS) on 4 bytes.

This report will consider only Q-tagged frame [20, § 1.4.416,§ 3.2.7], i.e. frames with one and only one 802.1Q tag.

An *Ethernet packet* is a Layer 1 PDU, made of

- a preamble on 7 bytes,
- a Start Frame Delimiter (SFD) on 1 byte,
- and an Ethernet Frame.

After the emission of an Ethernet packet is introduced a 12 bytes time called either Inter Frame Gap (IFG) or Inter Packet Gap (IPG).

In standard related to [4], one also use the term MSDU (Media Access Control Service Data Unit, [19]) and SDU (Service Data Unit).

Following [19, § 7.2], the MSDU is what is presented as the *payload* of a frame[3]. Indeed, it appears clearly in [19, § 7.2] that the source and destination MAC addresses are not part of the MSDU, and [4, § 34.4] explicitly states that the 802.1Q options, the EtherType/Size, the padding and the FCS are neither not part of the MSDU.

On the SDU, our current interpretation is that the term "SDU" in [4, 10] is the same thing than the expression "Ethernet frame".

When considering real-time behaviour, one have to consider non only MSDU or SDU size, but the time required to send the frame, that is the time required do send the Ethernet packet plus the IPG (it is partially addressed [4, § 34.4]). We call it "physical usage", and

[3] "The primitives of the MAC Service comprise a data request and a corresponding data indication, each with MAC destination address, MAC source address, a MAC Service Data Unit (MSDU) comprising one or more octets of data, and priority parameters. Taken together these parameters are conveniently referred to as a *frame*".

MSDU	Ethernet Frame	Ethernet Packet	Physical usage
$n \in [0, 1500]$	$\min(n, 42) + 22$	$\min(n, 42) + 30$	$\min(n, 42) + 42$
$\begin{cases} n - 22 \text{ if } n \geq 64 \\ 42 - \textit{padding} \text{ otherwise} \end{cases}$	$n \in [64, 1522]$	$n + 8$	$n + 20$
-	$n - 8$	$n \in [72, 1530]$	$n + 12$
-	$n - 20$	$n - 12$	$n \in [84, 1542]$

Ethernet Frame Overhead: dest./src MAC address (6B*2) + 802.1Q tag (4B) + Type/Length (2B) + Padding + FCS (4B)

Ethernet Packet Overhead: Preamble (8B)

Table 3 Relation between sizes, assuming one 802.1Q tag, in bytes.

for convenience, it will be expressed in bytes, not in a time unit, to get rid of the link speed parameter. And we may use the term "byte time" as the time required to send a byte.

In this report, we do no consider preemption, and especially the associated overheads [7].

3.2.1.2 Relations between sizes

Once defined the different terms, we can summary the relations between these sizes, also leading to minimal and maximal sizes.

Remind that we assume one and only one 802.1Q tag.

First, the payload should be in $[1, 1500]$. Note that we found no explicit lower bound value in the standard document, so we consider 1, like in the Wireshark documentation [25] but it is unclear for us if a value 0 could be a valid frame.

When considering only Q-tagged frames, a MSDU of size n bytes will be encapsulated in a SDU/Ethernet frame of size $\min(n, 42) + 22$ bytes (leading to a minimal size of 64 bytes, the classical 64 minimal Ethernet frame), encapsulated in a Ethernet packet of $\min(n, 42) + 30$ bytes, using the link during the emission of $\min(n, 42) + 42$ bytes (leading to a minimal size of 84 bytes). This is stated in the first line of Table 3.

In this context, an Ethernet Frame of size n is encapsulated into a Ethernet Packet of size $n + 8$, using the link during the emission of $n + 20$ bytes, as stated in the second line of Table 3.

The same correspondences are given in the third and fourth lines of this table.

3.2.2 Physical usage associated to contracts

Once we have recalled the relations between MDSU, SDU and physical usage, the question is: what is the total physical usage required to send a given amount of MDSU or SDU.

First, consider each frame of a stream or a set of streams.

If we known that each frame will have a MSDU size in range $[s^{\min}, s^{\max}]$, each frame will use the output link during $[\min(s^{\min}, 46) + 42, \min(s^{\max}, 46) + 42]$ byte time.

If we known that each frame will have a size in range $[l^{\min}, l^{\max}]$, each frame will use the output link during $[l^{\min} + 20, l^{\max} + 20]$ byte time.

Now, consider a global amount of MSDU or of SDU sent by a stream of a set of stream, and have a look on the possible link physical usage (as will be shown later, some contracts will be given either in terms of amounts of MSDU bytes or of SDU bytes).

Consider an amount of L SDU bytes, and consider that each SDU must have a size in range $[l^{\min}, l^{\max}]$ (these values always exists, at least with $l^{\min} = 64$ and $l^{\max} = 1522$), then

they can be decomposed into at most $\left\lceil \frac{L}{l^{\min}} \right\rceil$ frames and at most $\left\lceil \frac{L}{l^{\max}} \right\rceil$ frames. Since each frame will generate an overhead of 20 bytes, the total usage time of the link correspond to a number of bytes time in the range

$$\left[L + 20 \left\lceil \frac{L}{l^{\max}} \right\rceil ; L + 20 \left\lceil \frac{L}{l^{\min}} \right\rceil \right] \tag{3}$$

Consider an amount of S SDU bytes, and consider that each SDU must have a size in range $[s^{\min}, s^{\max}]$ (these values always exists, at least with $s^{\min} = 1$ and $s^{\max} = 1500$). A frame of size s will generate an padding of $\max(0, 46 - s)$ bytes, and another overhead of 12+4+2+4=22 bytes. Since an amount of S SDU bytes can generate at most $\left\lceil \frac{S}{s^{\min}} \right\rceil$ frames and at most $\left\lceil \frac{S}{s^{\max}} \right\rceil$, the total usage time of the link correspond to a number of bytes in the range

$$\left[S + (22 + \max(0, 46 - s^{\max})) \left\lceil \frac{S}{s^{\max}} \right\rceil ; S + (22 + \max(0, 46 - s^{\min})) \left\lceil \frac{S}{s^{\min}} \right\rceil \right] \tag{4}$$

To shorten expression, in the following, we will use the notations.

$$U^{SDU}(L, l) = L + 20 \left\lceil \frac{L}{l} \right\rceil \qquad U^{MSDU}(S, s) = S + (22 + \max(0, 46 - s)) \left\lceil \frac{S}{s} \right\rceil \tag{5}$$

3.3 Using PSFP to implement CQF

The CQF mechanism has been presented in [12, § 2]. The principle consists in slicing the time into even and odd intervals, all of the same length, and to forward the frames received in the even interval in one queue, and the one of the odd interval in the other queue.

This is a cyclic behaviour, with two states, and the most natural way to capture it is a SCL made of two entries. Assume a CQF class with period T, guard band S, using queues q_0 and q_1, on an output port of capacity R.

So, a typical SCL is made of two lines. All lines have Open state and duration T. The first line forwards frames to queue q_0, the second line forwards frames to queue q_1, as illustrated in Table 4.

The correct behaviour of CQF relies on the fact that all frames received in an interval will be emitted in the next interval. This requires that the cycle boundaries of adjacent nodes are enough aligned [17] and that the amount of data is not too large w.r.t. the cycle time.

One may want to use the *IntervalOctetMax* parameter to ensure than not too much date has been received.

In a cycle, the possible emission time is $T - 2S$, minus a blocking time at end of cycle due to the fact that a frame can not start its emission if it can not ends before the next gate closing.

But the *IntervalOctetMax* parameter only allows to check the accumulated size of the MSDU. Let $s^{\min}$, $s^{\max}$ be the minimal and maximal MSDU sizes of the streams in the CQF class. Then, the blocking time at end of cycle can be bounded by $U^{MSDU}(s^{\max}, s^{\max}) - 1$.

Then, to *guarantee* that all received frames can be forwarded in the next cycle, one have to select a value I such that

$$U^{MSDU}(I, s^{\min}) \leq R(T - 2S) - U^{MSDU}(s^{\max}, s^{\max}) + 1 \tag{6}$$

Conversely, any amount of MSDU bytes that can I be forwarded respects the relation

$$U^{MSDU}(I, s^{\max}) \leq R(T - 2S) \tag{7}$$

State	TimeInterval	IPV	IntervalOctetMax
Open	T	q_0	$I \in [I^{\min}_{T-2S}, I^{\max}_{T-2S}]$
Open	T	q_1	$I \in [I^{\min}_{T-2S}, I^{\max}_{T-2S}]$.

Table 4 Typical configuration of a stream gate list for CQF implementation.

Let $I^{\min}_{T-2S}, I^{\max}_{T-2S}$ the respective maximal solutions of eq. (6) and eq. (7), Then, depending on the kind of protection expected, on have to select a value of *IntervalOctetMax* in the range $[I^{\min}_{T-2S}, I^{\max}_{T-2S}]$.

This is what is the fourth row of Table 4.

3.4 Maximal frame size to deal with blocking factor

We have shown in Section 3.2 that the difference between the minimal and maximal frame size of flow have an impact on the over-provisioning of bandwidth for this flow.

But it may be also important to test the maximal size of a stream to protect higher priority flows.

Indeed, when a frame tries to access to the output port, it may be blocked by a lower priority frame. This blocking time is equal to the transmission time of the frame, ie. the size of the frame divided the output port transmission speed. This blocking time is taken into account by analysis methods, but they have to make some assumption on the maximal size of any lower priority frame.

One may consider that the maximal valid SDU size of a TSN frame is 1522 bytes (cf. Section 3.2), but in other contexts (encapsulation, tunnelling, jumbo frames), it may exists larger Ethernet frames.

Then, our proposal is to add to each stream filter (and especially the default rule, that will be proposed in Section 3.5) a filter on the maximal SDU size.

One may get rid of this rule for a flow if preemption is implemented, if this flow is forwarded to a *preemptable* queue, and if all higher priority flows are marked as *express*.

One may also ignore this rule if all higher priority flows have an exclusive access to the output port implemented with an adequate configuration of gates (this is commonly known, as “exclusive gating”).

3.5 Always set a default (best effort) rule

It has been shown in Section 2.3 that a frame that did not match any stream filter is forwarded like if no PSFP was present on the system.

This rule can make sense for flexibility reason: one may want a TSN switch to carry a set of well identified and registered real-time flows, but also some other flows, without having to manage the list of these “other” flows.

But these unregistered flows will have an influence on registered ones, and these interferences can lead to deadline misses or buffer overflows.

Several solutions can exists to avoid interferences between flows.

One may consider that the strongest solution is the temporal and spatial partitioning: dedicating a specific queue to unregistered flows, with a time partition between this queue and the others (exclusive gating). But during time windows dedicated to unregistered flows, the real-time flow will have no access to the output, and this blocking time should be at least large enough to allow an unregistered frame to be forwarded.

A simpler solution consists in using the static priority arbiter to always put a higher priority to registered frames. The cost related to preemption will be at most the size of one maximal unregistered frames size, which is smaller that the one implied by the temporal partitioning.

The ability to set an IPV can help to satisfy both requirements. We propose three rules.

1. The first proposition consists in dedicating the lower priority queue (#7 in a switch with 8 queues) to best effort traffic. Then, the last stream filter of the stream filter table has $(*, *)$ as matching pair *stream_handle specification, priority specification*, catching all frames (that have not been catch before). The state of this rule is Open, and the IPV is set to the number of the best effort queue. This is one case were a static gate is sufficient.
2. Having a single best-effort queue can be not smart enough. One may want to offer for example to unregistered frames with high priority a better service than the unregistered frames with a low priority.

   Then, the second proposition reserve $n \geq 2$ queues to unregistered frames. Then one have to define a mapping $m : [0, 7] \to [1, n]$ and set at the end of the stream filter table height stream filter rules. For each $i \in [0, 7]$, set one filter rule with matching pair $(*, i)$, and one static gate with open state and IPV equals to $m(i)$. Note that one may use some bandwidth sharing mechanisms between these n queues (based on ETS or CBS).
3. Of course, one may want to drop all unexpected frames. Then, the third proposition uses only a last rule with matching pair $(*, *)$ and a closed state.

3.6 Using PSFP to prevent faults

In this section, we will show how to use PSFP for checking different kinds of errors, depending on the kind of flow.

In critical systems, it is common to require that some service is provided even in case of some faults. It is often provided by using redundancy mechanisms, faults detection, recovery, etc. For a given architecture, safety analysis is in charge of giving a formal definition of "some faults", ie. giving explicit definition of the kind of fault (fail silent system, malicious system, hardware error rate, etc.) and of quantifying "some".

Such property in general relay on some containment property, that forbid a local fault to be propagated in the whole system.

For networks, we are using the following "fault containment" definition.

▶ **Definition 1** (Network fault containment). *A network is said to "contain faults" if a data flow crossing only switches in nominal state experiences a quality of service conforms to the contract established at configuration time (for statically configured systems), or at admission time.*

Of course, some faults are more easier to contain that others. For example, a frame loss only impacts the receivers of the data (except if this frame embodies network configuration data), whereas a babbling idiot may use all bandwidth and create buffer overflows in downstream switches.

The rest of this section will show that PSFP does not provide efficient fault containment for many kinds of faults.

In particular,

- Section 3.6.4 shows that protecting from an upstream switch sending frames smaller than the minimal SDU frame for a given stream can create a reservation overhead of 95% without padding and even more if padding is considered,

- Section 3.6.4 shows that protecting from an upstream switch sending frames smaller than the minimal SDU frame can create a reservation overhead of 95% (without padding) and even more

3.6.1 Routing error

In a context where the routing is statically set by design, one may want to prevent routing errors, and drop frames trying to use an unexpected path.

It exists a set of mechanisms, designed before TSN, that are designed to filter such frames: [4, § 6.16, § 8.6.1-8.6.4]. Such rules are base on the input port, the source and destination MAC addresses and the VLAN Identifier (VID). Since they have been designed before TSN, they have no notion of stream, can not distinguish between two streams having the same source and destination MAC addresses and VID. Note that such situation is quite common in a redundant system: one may have two redundant streams, with same source and destination and VLAN (if the VLAN is not used to encode the stream identifier), but supposed to use different paths.

The stream filter rules can accept/drop frames on a per stream basic. So, they can drop any frame which is not supposed to cross this node. But they can no react to the fact that a frame was received on an unexpected input port, if the stream_handle dos not encode the input port (cf. 2.1).

This is a difference with AFDX, where the policing rules are set per input port.

As will be shown in Section 3.6.6, this may have an influence on the performances of the system.

3.6.2 Configuring policing elements

A policing element is in charge of checking that each frame conforms to its contract, and mark or drop out of contract frames.

This section will first discuss the generic problem, and present how a stream can describe its contract in TSN. Then, the different ways to configure policing elements in function of the kind of traffic will be discussed in others sections.

3.6.2.1 Tests and the false positive/negative problems

Then, an ideal policing element must accept any conform frame (i.e. avoid “false positive”) and mark/drop any out of contract frame (i.e. avoid “false negative”).

The problem is that contracts and policing are expressed in TSN in slightly different ways, and we can not always avoid such false negative/positive, and we will have to do some assumptions on the kind of faults that we have to check.

In particular, as already mentioned in Section 3.2, the real-time behaviour of a TSN system relies on the time used to transmit each frame. But this time is dependant of the Ethernet payload but also of any additional overhead and silence time (the IFG).

Then, the real-time contracts must take into account any possible overhead, whereas some TSN policing elements can only check amounts of payload, not the actual number of frames.

That is to say, if a policing element can forward twice more data than the nominal behaviour, a bandwidth allocation that is tolerant to fault of upstream nodes must be twice the one of a bandwidth allocation based only on nominal behaviours.

And the same for the computation of latency, jitters and buffer usage.

Also note that clock drift may also require to increase the traffic contract [24], but this effect is quite small with regards to others aspects of TSN, and is not considered in this report.

3.6.2.2 Traffic contract in TSN

The traffic specification (TSpec) defined by SRP consists in a maximal number of frames *MaxFramesPerInterval*, with a maximal MSDU size *MaxFrameSize*, on some per-class measurement interval (CMI, which was either $125\mu s$ or 250 μs in [2, § 35.2.2.8.4], but that have been renamed in *classMeasurementInterval* and can now be freely set for each class since [9, § 35.2.2.10.6, § 46.2.3.5]).

This contract is either a sliding window or a tumbling window (depending on the presence of a *TSpecTimeAware* parameter).

Sliding window If no *TSpecTimeAware* parameter is provided, the traffic specification specifies a sliding window.

In general, respecting a sliding window of length L with at most b bytes per sliding window is equivalent to a token-bucket with replenishment rate $\frac{b}{L}$ and bucket size b (also known as "burst"). If such a stream crosses a network element with a jitter J (the jitter being defined as the difference between the minimal ans maximal crossing time), at output, it will respect a token-bucket shape of replenishment rate $\frac{b}{L}$ and bucket size $b\frac{L+J}{L}$). When crossing a sequence of elements of respective jitters $J_1, \ldots, J_n$, the bucket size become $b\frac{L+J_1+\cdots+J_n}{L}$).

Then, a stream having experienced a cumulative jitter J will respect a MSDU shape of token-bucket parameter (r^{MSDU}, b_J^{MSDU}) with

$$b_0^{MSDU} = \mathit{MaxFramesPerInterval} \times \mathit{MaxFrameSize} \tag{8}$$

$$b_J^{MSDU} = b_0^{MSDU} \times \frac{\mathit{CMI} + J}{\mathit{CMI}} \qquad r^{MSDU} = \frac{b_0^{MSDU}}{\mathit{CMI}} \tag{9}$$

But when considering the amount of bytes (or pseudo byte) at the physical level, like in [4, § 34.4], and considering Section 3.2, the parameter are

$$b_0^{SDU} = \mathit{MaxFramesPerInterval} \times (\mathit{MaxFrameSize} + \mathit{TSN\text{-}Overhead}) \tag{10}$$

$$\mathit{TSN\text{-}Overhead} = \min(46 - \mathit{MaxFrameSize}) + 42 \tag{11}$$

$$b_J^{MSDU} = b_0^{MSDU} \times \frac{\mathit{CMI} + J}{\mathit{CMI}} \tag{12}$$

$$r^{MSDU} = \frac{b_0^{MSDU}}{\mathit{CMI}} \tag{13}$$

Tumbling window When a *TSpecTimeAware* parameter is provided, the contract should also provide a pair *EarliestTransmitOffset*, *LatestTransmitOffset*. The semantics is the following. The time in divided in intervals of length *CMI*, that start at "the time epoch that is synchronized on the network" [10, § 46.2.3.5.1]. Let t_k denote the start of the kh interval. Then, in each interval $[t_k + \mathit{EarliestTransmitOffset}, t_k + \mathit{LatestTransmitOffset}]$, the stream talker can send up to *MaxFramesPerInterval* frames, each one having a size not greater than *MaxFrameSize*.

Such a contract can be used to specify Time-Triggered behaviour. Nevertheless, it should be noticed that the all streams in the same class must share the same *CMI*.

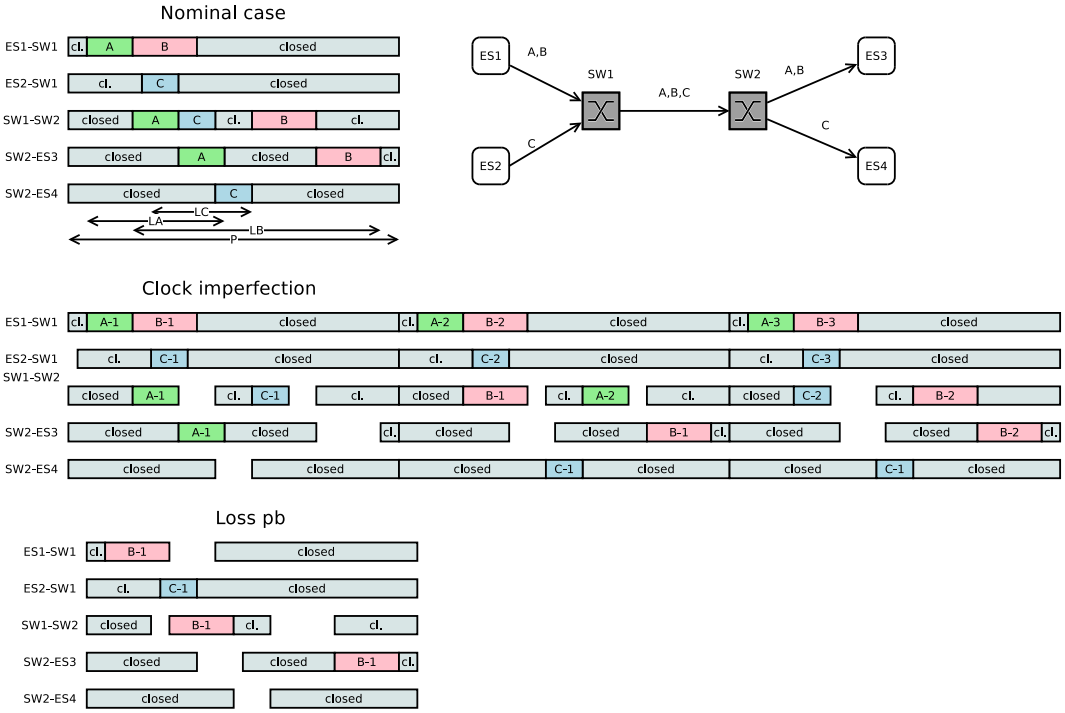

Figure 10 Time-Triggered communication in TSN: principle and limits

3.6.3 Time conformance of TAS frames

As presented in [12, § 1], the GCL mechanisms can be used to implement a Time-Triggered communication: such approach is called "Time Aware Shaper".

One well-known problem is that the loss of frame or some error in clock synchronisations between adjacent switches may break the expected scheduling [14], as illustrated in Figure 10.

The "Nominal case" represents a system with three flow, A, B, C, all with the same period P, sharing the same class, and the expected behaviour related to gate closing and opening during one period. The expected latencies LA, LB, LC related to A, B and C are also drawn.

In the "clock imperfection" scenario, a frame C-1 is received too late w.r.t. its expected slot, and then can not be forwarded on this expected slot. Assume that it is not discarded, then it put in the output queue of SW1. Then, it will be forwarded to SW2 in the slot devoted to B-1. But it will nevertheless be too late to be forwarded to ES4 during this period, and it will be sent during the slot devoted to C-2. And so on, each frame of stream C will suffer a latency LC+P (instead of LC) even if the synchronisation of between elements is reestablished, until some loss occur between ES2 and SW1, or until the application skip some emission opportunity.

Moreover, since C-1 uses the slot devoted to B-1, B-1 also stays in the queue, until the next opening: the slot dedicated to A-2... Then, the fault on the synchronisation of ES2 has a negative impact on flows send by ES1.

Of course, one may add constraints while building the scheduling to avoid such situation (e.g. the "flow isolation" [14]), but an adequate configuration of PSFP may also drop the frame received at unexpected instants to avoid such effects.

This can be done either on a per-stream basic, but it will require to have one stream gate

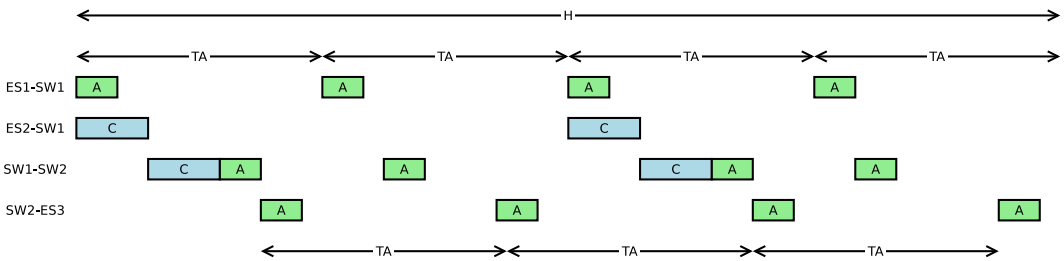

Figure 11 Time-Triggered communication in TSN: non strictly periodic schedule for a flow

with one SCL per scheduled stream. And the size of the SCL must be large enough. If the schedule is strictly periodic, a SCL of length 2 is sufficient.

However, a TAS schedule is generally build on the hyper-period of the set of flows. And the solver that generates the global schedule may produces a schedule that is not, in the network, strictly periodic for each flow.

Consider for example the of Figure 11, with only two flows crossing the same network than in Figure 10. The flow A has a period TA, and the flow C a period 2 times larger. Assuming a zero offset (imposed by the application for example), one out of two A frame will be in conflict with a C frame. One possible global schedule is the one represented in Figure 11. In this schedule, the flow A is no more strictly periodic on the link between the switches. Then, an SCL at SW2 input must have a size of 4 to capture this behaviour.

Of course, one can in this case build a schedule where A is strictly periodic on this link. Our claim is just to warn that a small SCL may impose some constraint on the schedule generation.

Also, note that the length of an SCL may not be the only constraint. One may be limited by the number of stream gates. In this case, one may use one SCL to check several streams. If the frames are not too close one to each other, the risk that a frame of one stream use the time window of another is limited. But it will require a larger SCL, especially if the flow do not share the same period.

3.6.4 Time conformance of asynchronous flows: the frame size problem

In this section, we will show that a faulty switch can create buffer overflows in the downstream switches, and that PSFP can not protect from it in the general case, and even in common case, can lead to a waste of half of the guaranteed bandwidth.

Guaranteed real-time communications can be achieved only with some envelope on each data flow and static priority, as demonstrated in AFDX.

But to guarantee a bounded waiting time in each queue, one have to check that each flow conforms to its contract (and drop it otherwise). More precisely, one have to know how much time will be used by each Ethernet packet and its associated IFG (cf. Section 3.2 and Section 3.6.2). That is so say, what is the physical usage associated to a stream contract?

As presented in Section 3.6.2.2, an asynchronous stream generates a data flow that conforms to a token-bucket with parameters given in eq. (8)–(9).

Such contract can be checked by a stream meter with corresponding (CIR,CBS) parameters, and (EIR=0,EBS=0). Such meter will never discard frames of a stream that conforms to its traffic specification.

The other question is: what is the amount of non-conform traffic that this meter can forward? What happen if the previous switch sends an infinite sequence of frames with only

one byte of payload?

Then, the stream meter will accept all frames has long has there are tokens...

Consider a stream whose contact is to send one Ethernet frame of 480 bytes each millisecond (and a MSDU of 458). The physical usage is of 500 bytes time, corresponding to a bandwidth of 500Kb/s. Then, to forward any conform frame, the meter must be configured with CIR $= 458Kb/s$ and $CBS = 458$. Then, a faulty or malicious upstream node can send 458 Ethernet frames per millisecond, each one with a single byte of payload. This flow will be forwarded by the meter. But it will generating a throughput of 458*84 b/ms = 38.472 Kb/s = 38.472 Mb/s. Of course, one may configure the meter with CIR $= 38.472Mb/s$, but this lead to a major waste of capacity at reservation (even if at run time, the difference can be used by best-effort traffic).

One may consider that sending frames of one byte is an extreme example, but even if the previous node sends Ethernet frames of 200 bytes (corresponding to a MSDU of 178 bytes), it can generate $\frac{458}{178}$ frames per millisecond, leading to $\frac{458}{178} * 200 = 514.60$ Kb/s.

More generally, and considering only the throughput, not the burstiness, if a stream uses at physical level a bandwidth r (in b/s), with frames of MSDU size $s \geq 42$, it correspond to a MSDU bandwidth of $\frac{s}{s+42}r$.

Then, if a fault stream sending frames of MSDU size s' sends the same amount of MSDU bytes than a valid same sending frames of size s, and having a reserved physical bandwidth b, then the faulty stream will use the bandwidth

$$\frac{s'+42}{s'}\frac{s}{s+42}r. \tag{14}$$

In the extreme case where $s = 1500$ and $s' = 42$, the faulty stream can use 95% more physical bandwidth than the nominal one without being detected by the meter.

If $s = 100$ and $s' = 50$, the overhead is 30%, and if $s = 1000$ and $s' = 500$, the overhead is only 4%.

In conclusion, consider a switch receiving a stream whose nominal behaviour consists in sending frames with MDSU size in $[s^{\min}, s^{\max}]$ with (physical) throughput R, then, its PSFP must have a CIR being at least $\frac{s^{\min}}{s^{\min}+42}$. But to protect himself from an upstream switch sending frames using this stream id but of size 64 without any padding, every reservation computation mus be done considering that this flow may use $2\frac{s^{\min}}{s^{\min}+42}R$.

To avoid, or at least limit this behaviour, one may consider what is done in AFDX. It provides two solutions.

Per-frame token-bucket one solution is to have a token-bucket where each token corresponds to a full frame. Combining this semantics with a "Maximum SDU size" filter allows to upper bound the throughput of each stream forwarded to the output link, whatever the size of the frames are.

Minimal frame size another solution consists in adding a filter that forbid padding and requires a minimal size, and otherwise drop the frame. Combining a first test with minimal SDU size $l^{\min} \geq 64$ and a second with maximum SDU size $l^{\max}$ limit the ratio between non-conform and conform traffic to a ratio $\frac{l^{\min}}{l^{\min}-22}\frac{l^{\max}-22}{l^{\max}}$.

Note that these solutions are not in the standard, but may be implemented as extension with a limited effort.

Adding a new *filter specification* that checks a *Minimum SDU size* seems simple, it is just the symmetric of the *Maximum SDU size* existing filter.

Counting frames and not MSDU bytes is a major change in the point of view, but looks quite simple in the implementation itself.

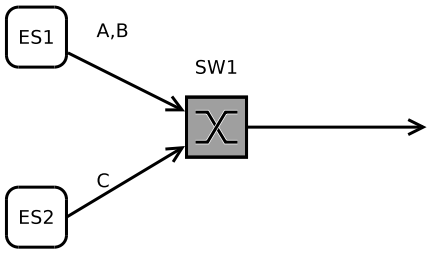

Figure 12 Flow convergence example.

Last, another solution can be to monitor not the number of bytes in the MSDU (or even the SDU) but at the physical usage, i.e. decrease the token not only by the MSDU size of the frame, but also by the media overhead (the Ethernet Packet size plus the IFG for IEEE 802.3 point-to-point media). This information exists in the standard, this is the `portMediaDependentOverhead` [6, § 12.4.2.2]. This is the solution adopted by ATS, that uses a function `length(frame)` computing "*the length of the frame, including all media-dependent overhead*" [6, § 8.6.11.3.11].

Nevertheless, it seems to exclude the case when a single stream is forwarded to different physical layers (for multicast feature, or use of Frame Replication and Elimination for Reliability [3]).

One trade-off could be to allow this metering overhead to be set by configuration, reporting the choice to the system designer.

3.6.5 Limits on the number of meters

The name of the amendment, "Per-Stream Filtering and Policing" suggest to look at the characteristics of each stream. But the number of meters may be less than the number of streams. Then, it appears possible to share a meter between several streams. From a pure token-bucket point of view, having two streams f_1, f_2 of conforming to token-buckets of respective parameter (b_1, r_1), (b_2, r_2) look equivalent of having a flow f conforming $(b_1 + b_2, r_1 + r_2)$.

But since the meter works on MSDU size, but the buffer occupancy is influenced by the physical usage, the over-provisioning implied by difference in frame size is increased if the minimal sizes of both streams are different.

Moreover, if a meter is shared by two streams f_1, f_2 one stream may use all the tokens in the bucket leading to reject nominal frames of the other stream.

To sum up, if it recommended to share meters only for stream with similar frame sizes, and to enforce network fault containment, it is required to avoid that two stream from different input ports share the same meter.

3.6.6 Time conformance of in absence of shaper: the routing problem

Routing errors may also lead to over provisioning or deadline violation or even buffer overflow if the input port is not present in the stream_handle.

Consider the simple example of Figure 12 and consider that flows A and B have the a low priority, C a higher priority, they share a common constant frame size, a very large period w.r.t. other constants of the systems. Consider that the switch has a very short commutation time and that all links have the same speed. Let T be the transmission time of one frame of A, B or C.

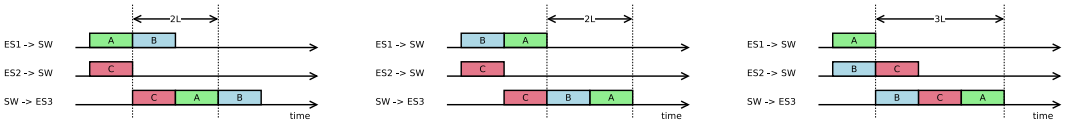

Figure 13 Flow convergence and waiting times

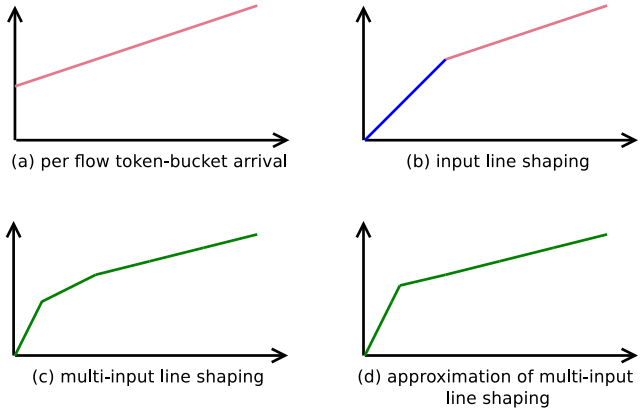

Figure 14 Illustration of line shaping effect

In this case, the flow A will compete on the output port with B or C, but never both, and its worst latency in the switch is bounded by $2L$ (L for queue waiting time plus L for it own transmission time).

Two cases are illustrated in Figure 13: when A and B arrive in the input port back-to-back, either A if before B, or the opposite, but in both cases, the latency in the switch is upper-bounded by $2L$.

Now, consider that B is received on the same output port than C (due to some routing error upstream, or even because of a malicious switch that creates malicious frames). Then, if B is received just before A, and C just after B, then A has to wait the transmission of B and C before starting its own transmission leading to a traversal of $3L$.

By generalising this example to more flows A-1,A-2,..., A-n, B-1,B-2,..., B-n and C, the expected routing leads to a delay bound $(n+1)L$ whereas the other one leads to a delay bound $(2n+1)L$.

This is because, when several flows share the same link, they can not arrive at the same time. This effect is well known in the domain of network latency bounding, and is called either shaping [13], grouping [15, 26] or serialisation [11].

Consider Figure 14. The sub-diagram 14.a represents the output of a token-bucket, what can be send by one stream or a group of streams. This is what can for, for example, forwarded by a stream meter to an output queue. However, this output queue cannot send at a rate higher than the link speed. Then the next node receives the token-bucket output, which is limited by the line throughput, leading to a curve with two slopes, illustrated in sub-diagram 14.b. This lead to a smaller burst and it gives smaller delay bounds. It can be illustrated with a server with a single input port and a single output port: if both input line and output line have the same speed, and if all frames have the same size, event if frames are sent back to back, where a frame enters the server, the previous one just finishes its emission, and the waiting time is only the transmission time, there is not waiting time in queues. When several input ports converge to the same output ports, the benefit is not so big (since several frames

can arrive at the same time), but this effect is still to be considered. In a realistic AFDX configuration, considering the line shaping decrease the bounds by 40% [16]. The effect is illustrated sub-diagram 14.c.

The point is that the shape of the curve depends on the input link speed and the routing. Consider one switch and a stream that, in nominal case, uses a link shared by a lot of streams (so limiting the burst of this group). If that stream, due to some routing error, enters the switch by another input port, which, in nominal case, carries a small number of stream, then, it will increase the global burstiness at switch input, and increase the delay.

Note that even if a different routing may only slightly change the en-to-end delay of a flow (the competition between A and B should be done somewhere in the network) it nevertheless changes the local buffer occupancy.

Solutions can be proposed to contain the problem.

The ideal solution rely on the ability for PSFP to drop frames coming from unexpected input ports. It requires either that the stream identification function can take into account the reception port when generating stream_handle, or to give this input port to the stream filter rules.

Another solution to deal with routing errors is then to consider all possible allocations of stream to input ports. Another consists in doing an upper-approximation, keeping (and extending up to intersection) only the first and last segments, as illustrated in sub-diagram 14.d.

But all these solutions leads to over-provisioning. And this over-provisioning can be of a factor of 100% (going from $(n+1)L$ to $(2n+1)L$ is the previous example).

One may rely on existing Ethernet solutions to drop frame B. As presented in Section 3.6.1, some filtering can be done based on input port, source address, destination address and VLAN Identifier (VID). But if, unfortunately, it exists another valid stream with the same source, address and VID which is supposed to goes to the second port (the one receiving stream C), this will not be sufficient.

Then, one solution could be to impose, at design, that never two flows with the same source address, the same destination address and the same VID can enter a given switch through two different ports. But this can be the case with FRER [3, Figure C-15].

3.6.7 Time conformance in case of CBS shaper

The case when a stream comes out from a CBS class is very similar to the case when there is no shaper, due to the fact that CBS tries to mimic a virtual link with a throughput set by configuration. It has been shown in [27] that CBS introduced a shaping, similar to the line-shaping presented in Section 3.6.6, that can be represented as a token-bucket (where the replenishment rate is the *idleSlope* parameter, and the burst depends on the maximal credit value).

Then, consider once more the example of Figure 12. Assume that streams A and B share the same CBS class in the upstream switch. Let (r, b) be the token-bucket parameters of this CBS class, and (r_A, b_A), (r_B, b_B) the respective parameters of the streams. The interesting case is when $b \leq b_A + b_B$ and $r > r_A + r_B$[4] In this example, an efficient algorithm will consider a workload bound represented by the minimum of both, as illustrated in Figure 15-(a).

[4] If $b \geq b_A + b_B$, it means that CBS is useless. The case $r < r_A + r_B$ is a configuration error. The default configuration is $r = r_A + r_B$, but for performances reason, one may set $r > r_A + r_B$. An example is provided in Section A.

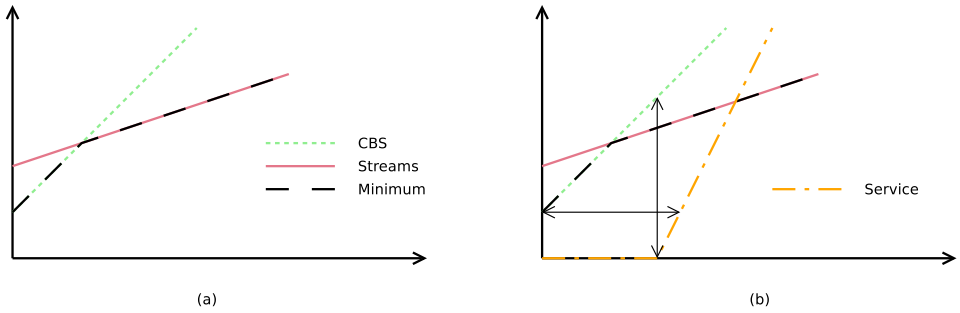

Figure 15 Arrival curve of a set of flows coming from one CBS class.

Since a flow can cross only one meter, there are three possibilities.

One CBS meter One may configure a meter for the CBS class. The two streams share the meter. Then we have no guarantee that each flow will respect its contract. Then, one have to consider that a faulty upstream switch may use all the bandwidth, and that the CBS meter will allow it, leading to a bandwidth consumption larger than the sum of the bandwidth of the accepted flows.

Our experience in network calculus let us guess that considering only the CBS contract will often lead to the same delay bound than the full contract, but it may be not the case for the buffer usage.

This is illustrated in Figure 15-(b): the slope of the CBS contract (its idleSlope) is in general lower that the residual rate of the output port (the residual rate being the bandwidth of the port minus the bandwidth of higher priority flows), and the horizontal deviation (the operator used to measure the delay in network calculus) is the same in bot cases. But the buffer usage, measured with the vertical deviation, depends on the relative position of the service latency and the intersection point of the stream token-bucket and the CBS token-bucket.

This intuition requires of course more studies to be confirmed.

Also note that, as shown in Section 3.6.5, this mean that one flow may use all the tokens of the shared meter, and lead to frame drop of valid frames of the other stream.

But since both two flows comes from the same upstream switch, this does not violate the "network fault containment".

One meter per stream One may either have one meter per stream. In this case, there is no way to check that the CSB shaping is respected, and a faulty upstream switch may forward frames without applying any shaping. Then, any analysis trying to cope with faults has to do as if no CBS was present in the previous switch.

That is to say, CBS is active but it cannot be used for (safe) performance analysis since there is no way to drop frames that does not respects its shape.

One shared stream meter The worst solution consist in using only one meter for all stream, whose configuration is based on stream characteristics. As in the case "one meter per stream", the benefit of CBS can not be considered, but like in the case "one CBS meter", one faulty flow can lead to drop valid frames.

Note that, for illustrative purpose, we have considered a queue receiving only two flows from the same upstream switch coming from the same CBS class, but the same kind of behaviour occurs with more flows and several CBS class. The principles are the same, but the global arrival curve has more segments. For the same simplification purpose, the shaping due to links is not represented (cf. Section 3.6.4).

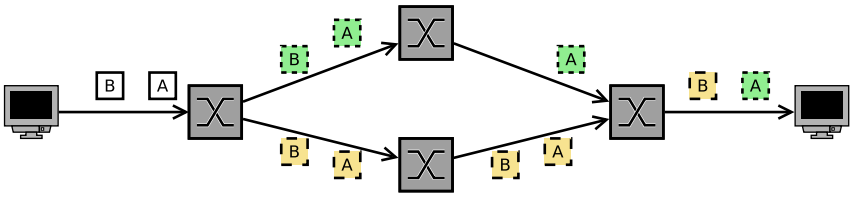

Figure 16 Illustration of FRER configuration: frames A, B are duplicated in the first switch, the B frame is lost on the upper path, and the recovery function removes one A duplicate.

Like for the routing problem, it an be solved by an extension of the standard. If a stream can cross several meters during the filtering phase, one may configure one meter per input stream, and one meter per input CBS class. Each frame will cross both, ensuring that the global load sent to the queue will respect its contract, even in case of fault of an upstream faulty switch.

3.6.8 Time conformance of ATS streams

The status of ATS stream checking with PSFP is very specific, and deserves some discussion.

Two uses cases for ATS are considered in this report:

- the *per-flow* shaping, with one ATS shaper per flow, that correspond to the original proposal [23] and that offers the *shaping for free* property [21],
- the *per input port* shaping, proposed in [18] that acts has a greedy shaper.

ATS is, by itself, a shaper that can be seen has a way to contain faults: since each flow will be shaped by ATS (seen as an interleaved regulator), it ensures that any flow competing to the output port will respects a local contract, independent of any fault in upstream nodes. But delaying the head-of-queue create a delay on the other frames in the queue. It has been shown in [23, 21] that, if each flows respects the contract before the previous arbiter, and if this arbiter uses a FIFO policy, and if the shaping queues are grouped per input port, then ATS reshaping does not increase the worst case. But we found no way to check, using PSFP tools, or even with some PSFP extension, to check that these properties are satisfied. It means that a misbehaviour in the upstream node on one flow can have a negative impact on the other flows sharing this queue in the local node. But, due to ATS grouping rules, it will only corrupt flows coming from this upstream node.

That is to say, a non-conform behaviour in the a node can create a misbehaviour on a downstream node, but this misbehaviour will impact only the flows that have crossed this node.

In case of *per input port* shaping, the situation is the same as in absence of shaper, presented in Section 3.6.6. The waiting time in the shaping queue is computed only on the per-flow token-bucket curve plus the line shaping.

3.7 Relative position between FRER and PSFP

The Frame Replication and Elimination for Reliability addenda (FRER, [3]) has been designed to improve the reliability of the network. FRER allows to duplicate frames in the network and to remove duplicates at joining points (as illustrated in Figure 16).

What is important to notice is that, as illustrated in Figure 17, the removing of duplicates is done after the PSFP filtering and policing. Then, the configuration of PSFP must be

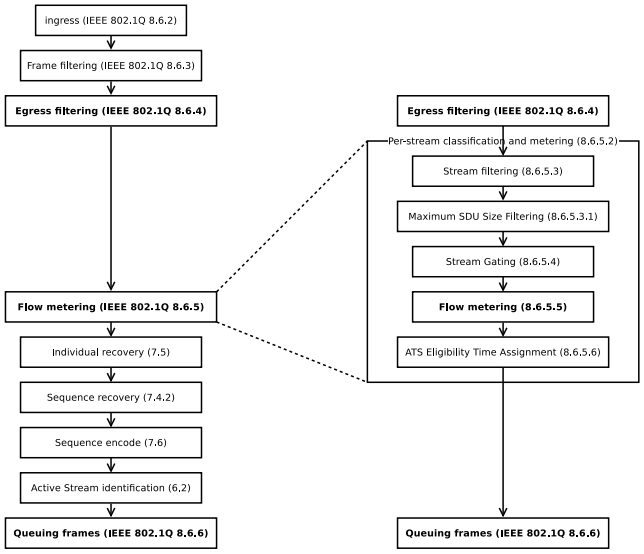

Figure 17 Inclusion in the global forwarding process of FRER (left) and ATS (right), inspired from [3, Fig. 8-2] and [5, Fig. 8-13]

done considering all entering frames, without knowing each or how many will be dropped or forwarded.

3.8 Equipment tests

Here is a list of a few capacities of a PSFP implementation that have an impact on the ability of a PSFP system to protect a system from faults in upstream nodes.

1. Testing the ability to have several stream meters associated to a single stream filter (cf. Section 2.3)
2. Testing the order between the tests in a stream filter (cf. Section 2.3)
3. Testing the absence of confusion between SDU and MSDU in the stream checking rules (testing that the *Maximum SDU size* checks the SDU/frame size, that the meter checks the SDU/frame size, and that the IntervalOctetMax checks the MSDU)
4. Testing if the capacities (maximal size) of the stream filter instance table, the stream gate instance table and the stream meter instance table are large enough to implement all the tests proposed in this report.

4 Conclusion

In this paper, we have presented the mechanisms of PSFP, and shown how they can be used to guarantee real-time behaviour even in presence of faults. We have proposed, for each kind of TSN traffic, a PSFP configuration pattern. We have shown that the main problems come from the fact that the per-frame overhead on the physical layer is hard to take into account in an accurate way, leading to very large over-provisioning.

We proposed several configuration rules:

- if the stream identification function is not able to take into account the input port, to cope with routing error, a valid configuration may never admit that two flows with the same source address, destination address and VID enter the same switch by different ports (but this is a common use can in FRER [3, Figure C-15]),

- to cope with switches were a stream can cross only one single meter, use CBS only the benefit in the deadline compensate the increase of the memory bound.

We have highlighted the impact of the limit of the number of meters.
We proposed several simple extensions to the standard:

- clarify the possibility for the stream identification function to take into account the input port of a frame, or pass this information to the stream filter,
- add of an IPV to stream meter, to change the class of non-conforming frames,
- replace the per-bit semantics of token-buckets in stream meter by a per-frame semantics,
- taking into account the media dependant overhead `portMediaDependentOverhead` in the token-bucket stream meter (like done by ATS), to limit the difference between MSDU and bandwidth usage
- add of a minimal size filter.

Thanks

The author would like to thank

- Matheus Ladeira, from RealTime-at-Work (RTaW), for pointing out the evolution of the interpretation on the number of meters between [10] and [6].
- Ludovic Thomas, from LORIA, for useful feedback and especially the existence of the `portMediaDependentOverhead` parameter and discussions on stream_handle (and pointing out [6, Note 1, § 8.6.5.3]).

References

1 Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. Technical Report IEEE 802.1Qav, IEEE, 2010. doi:https://doi.org/10.1109/IEEESTD.2009.5375704.

2 Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP). Technical Report IEEE 802.1Qat, IEEE, 2010.

3 IEEE standard for local and metropolitan area networks – frame replication and elimination for reliability. Technical Report 802.1CB, IEEE, September 2017. doi:https://doi.org/10.1109/IEEESTD.2017.8091139.

4 IEEE standard for local and metropolitan area networks – bridges and bridged networks. IEEE Standard 802.1Q, IEEE, 2018. doi:https://doi.org/10.1109/IEEESTD.2018.8403927.

5 IEEE standard for local and metropolitan area networks – asynchronous traffic shaping. Technical Report 802.1Qcr, IEEE, September 2020. doi:https://doi.org/10.1109/IEEESTD.2020.9253013.

6 IEEE standard for local and metropolitan area networks – bridges and bridged networks. IEEE Standard 802.1Q, IEEE, 2022.

7 IEEE standard for local and metropolitan area networks – bridges and bridged networks – amendment 26: Frame preemption. IEEE Standard 802.1Qbu, 2016. doi:10.1109/IEEESTD.2016.7553415.

8 IEEE standard for local and metropolitan area networks–bridges and bridged networks–amendment 25: Enhancements for scheduled traffic. IEEE Standard 802.1Qbv, IEEE, 2015. doi:10.1109/IEEESTD.2016.8613095.

9 IEEE standard for local and metropolitan area networks– bridges and bridged networks – amendment 31: Stream reservation protocol (srp) enhancements and performance improvements. Standard 802.1Qcc, IEEE, 2018.

10 IEEE standard for local and metropolitan area networks–bridges and bridged networks–amendment 28: Per-stream filtering and policing. Standard 802.1Qci, IEEE, 2017. doi:10.1109/IEEESTD.2017.8064221.

11 Henri Bauer, Jean-Luc Scharbarg, and Christian Fraboul. Improving the worst-case delay analysis of an AFDX network using an optimized trajectory approach. *IEEE Trans. Industrial Informatics*, 6(4):521–533, 2010.

12 Marc Boyer. Déterminisme tas, tas+cbs et du mécanisme cqf en calcul réseau (v2) – on-2.2.4. Technical Report RT 6/31136 DTIS, ONERA, March 2022.

13 Marc Boyer and Christian Fraboul. Tightening end to end delay upper bound for AFDX network with rate latency FCFS servers using network calculus. In *Proc. of the 7th IEEE Int. Workshop on Factory Communication Systems Communication in Automation (WFCS 2008)*, pages 11–20. IEEE, May 21-23 2008. doi:10.1109/WFCS.2008.4638728.

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

15 Fabrice Frances, Christian Fraboul, and Jérôme Grieu. Using network calculus to optimize AFDX network. In *Proceeding of the 3thd European congress on Embedded Real Time Software (ERTS06)*, Toulouse, January 2006.

16 Jérôme Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, Institut National Polytechnique de Toulouse (INPT), Toulouse, Juin 2004.

17 Damien Guidolin-Pina, Marc Boyer, and Jean-Yves Le Boudec. Configuration of guard band and offsets in cyclic queuing and forwarding. Technical Report hal-03772877, HAL, 2022. URL: https://hal.archives-ouvertes.fr/hal-03772877.

18 Hao Hu, Qiao Li, Huagang Xiong, and Bingwu Fang. The delay bound analysis based on network calculus for asynchronous traffic shaping under parameter inconsistency. In *2020 IEEE 20th International Conference on Communication Technology (ICCT)*, pages 908–915, 2020. doi:10.1109/ICCT50939.2020.9295939.

19 IEEE. Media access control (mac) service definition. Technical Report 802.1AC, IEEE, 2016.

20 IEEE. IEEE standard for ethernet. Technical Report 802.3, IEEE, 2018.

21 Jean-Yves Le Boudec. A theory of traffic regulators for deterministic networks with application to interleaved regulators. *IEEE-ACM Transactions On Networking*, 26(6):2721–2733, 2018. doi:10.1109/TNET.2018.2875191.

22 MEF. Subscriber ethernet service attributes. Technical Report MEF 10.3, MEF Forum, 2013. URL: https://www.mef.net/wp-content/uploads/2013/10/MEF-10-3.pdf.

23 Johannes Specht and Soheil Samii. Urgency-based scheduler for time-sensitive switched ethernet networks. In *Proc. of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016)*, 2016. doi:10.1109/ECRTS.2016.27.

24 Ludovic Thomas and Jean-Yves Le Boudec. On time synchronization issues in time-sentive networks with regulators and nonideal clocks. *Proceedings of the ACM on Measurement and Analysis of Computing SystemsJune 2020 Article No.: 27*, 4(2), June 2020. URL: https://dl.acm.org/doi/10.1145/3392145, doi:10.1145/3392145.

25 Wireshark. Ethernet (IEEE 802.3) – allowed packet lengths. https://wiki.wireshark.org/Ethernet#allowed-packet-lengths.

26 Luxi Zhao, Qiao Li, Ying Xiong, Zhong Zheng, and Huagang Xiong. Using multi-link grouping technique to achieve tight latency in network calculus. In *Proc. of the 32nd IEEE/AIAA Digital Avionics Systems Conference (DASC 2013)*, pages 2E3–1–2E3–10, East Syracuse, NY, USA, Oct 2013. doi:10.1109/DASC.2013.6712551.

27 Luxi Zhao, Paul Pop, Zhong Zheng, Hugo Daigmorte, and Marc Boyer. Latency analysis of multiple classes of AVB traffic in TSN with standard credit behavior using network calculus. *IEEE Transactions on Industrial Electronics*, 2020. doi:10.1109/TIE.2020.3021638.

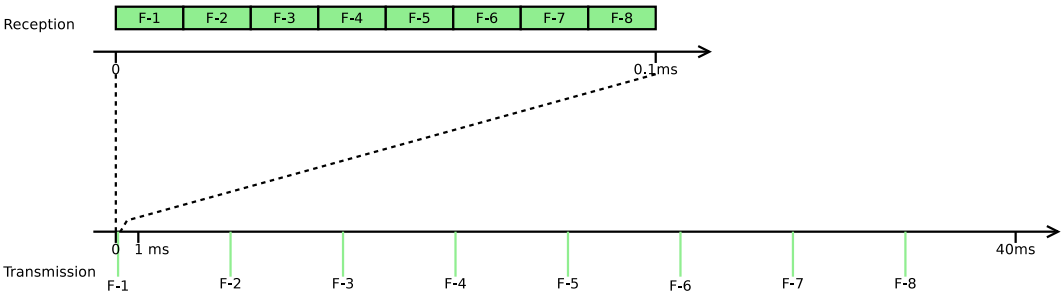

Figure 18 A CBS port receiving a burst of 8 frames of 1250B each and transmiting with idleSlope of 2Mb/s on 100Mb/s links.

A On CBS slopes

The part of the bandwidth allocated to a CBS class (the idleSlope of the class) is left to the network administrator (up to some maximal value, cf [4, § 34.3]), but the all document presents this parameter as a bandwidth parameter. Then it may appear sufficient to have a reservation just equal to the bandwidth of the stream using the class. But this may not be the case, especially in case of bursty stream, since the idleSlope also have an impact on the latency.

Consider a video stream sending 25 pictures per second (i.e. one picture per 40ms), each picture having the same size of 10KB, leading to the transmission of 8 frames of 1250 bytes each (ignoring header and any overhead for sake of simplicity). This leads to a 2Mb/s throughput.

Assume a 50ms deadline requirement, and consider than this flow is the only one in the CBS class. What is the adequate bandwidth reservation for this class?

One may consider allocating to the class exactly its bandwidth usage, 2Mb/s. Assume now a network with 100Mb throughput on all links, and look what happen on the first switch. Consider no interference from any other stream.

The end-systems sends, at time t=0, the burst of 10 frames. It is transmitted in 0.1ms. Consider that the frames goes from input port to output port in negligible time, the transmission of the first frame by the switch CBS output port starts at time $t = 0.01ms$. But since the idleSlope is only 2Mb/s, the next frame is sent only 5ms after the start of the transmission of this one, i.e. at $t = 5.01ms$, and the last one at $t = 35.01ms$.

More generally, if a CBS class may receive a burst of n frames on a periodic interval T, and if the CBS class have been allocated exactly the corresponding throughput, the delay between the start of transmission between the first frame and the last frame is $\frac{n-1}{n}T$. For systems with implicit deadline (where the latency constraint is the period), it means that rest of the network must be crossed in less than $\frac{T}{n}$.

Of course, the next CBS server will not re-impose such large delay, since the frames will be received already with an inter arrival time conform to the CSB shaping requirement.

This example is of course a toy example, but equivalent situations, with a larger n, have been known by the author (but not reported for confidentiality consideration). In such system, it may then be necessary to allocate to the CBS class an idleSlope greater than the total throughput of the streams using this class.