High-Accuracy Packet Pacing on Commodity Servers for Constant-Rate Flows

Mohammed Hawari, Juan-Antonio Cordero-Fuertes, Thomas Heide Clausen

High-Accuracy Packet Pacing on Commodity Servers for Constant-Rate Flows

Mohammed Hawari, Juan-Antonio Cordero-Fuertes, Thomas Clausen

***Abstract*—This paper addresses the problem of high-quality packet pacing for constant-rate packet consumption systems, with strict buffering limitations. A mostly-software pacing architecture is developed, which has minimal hardware requirements, satisfied by commodity servers – rendering the proposed solution easily deployable in existing (data-centre) infrastructures. Two algorithms (*free-running* and *frequency-controlled* pacing, for explicitly and implicitly indicated target rates, respectively) are specified, and formally analysed. The proposed solution, including both algorithms, is implemented, and is tested on real hardware and under real conditions. The performance of these implementations is experimentally evaluated and compared to existing mechanisms, available in general-purpose hardware. Results of both exhaustive experiments, and of an analytical modeling, indicate that the proposed approach is able to perform low-jitter packet pacing on commodity hardware, being thus suitable for constant rate transmission and consumption in media production scenarios.**

***Index Terms*—Packet pacing, jitter, constant rate, SMPTE, implementation, buffering, Pacing Assistant, systems architecture**

I. Introduction

In packet-switched networks, the packet transmission rate needs to be controlled so that the receive buffer never overflows. Known as *flow control*, that task is usually achieved asynchronously, *e.g.,* with the stop-and-wait Automatic Repeat reQuest (ARQ) protocol (introduced in [1]), or with a sliding-window-based algorithm, *e.g.,* as in the Transmission Control Protocol (TCP) [2]. For either of those, the receiver sends an explicit signal, authorising the sender to transmit more data. TCP achieves flow control by permitting a maximum number — the current *receiving window* — of transmitted and unacknowledged data. A TCP Acknowledgment (ACK) decreases the amount of unacknowledged bytes and thus acts as authorisation for the sender to transmit subsequent data.

Some applications will consume packets from their receive buffer at a given rate, independently from the state of the buffer. An example hereof is a video player, which expects to always have enough packets available to build the next video frame. If incoming packets are late, these applications might try to consume a packet from an empty buffer, a condition known as *buffer starvation* – which, in the video player example, results in a visually unsatisfactory user experience.

The design of the system, the sender, and the receiver, should prevent this from happening, *i.e.,* each packet in such a stream must be received *before* a certain time. This is different from flow control, which seeks to delay transmission by the *sender* until *after* it has been confirmed that there is buffer space available by the receiver.

A. Towards Software Pacers

If a sender transmits, and receiver consumes, packets at the exact same rate, the receive buffer will be subject to neither overflow nor to starvation – assuming that the delay introduced by the network between sender and receiver is *close* to constant. That approach is adopted by the Society of Motion Picture and Television Engineers (SMPTE) to standardise the transport of production-grade multimedia streams over IP/Ethernet networks. For example, SMPTE 2022-6 [3], SMPTE 2059 [4] and SMPTE 2110-21 [5] define timing constraints on production-grade multimedia streams, and on network equipment, so as to avoid buffer overflow and starvation. Those standards were designed under the assumption that packets belonging to such streams are to be transmitted at Constant Rate (CR).

Therefore, as the SMPTE standards are intended as enablers for the broadcasting industry to migrate media processing from dedicated to commodity hardware, the latter must be able to generate and carry SMPTE-compliant, *i.e.,* sufficiently regular, CR streams.

Furthermore, migration from *"all media-dedicated hardware"* to *"all software running on commodity hardware"* includes an intermediate state with *gateway devices*, *i.e.,* with dedicated hardware, consuming SMPTE-compliant CR streams received over IP/Ethernet, and converting them into the Serial Digital Interface (SDI) format [6], used by media-dedicated hardware. As a transition technology, gateway devices may have (for reasons of cost-containment) limited memory, *i.e.,* limited receive buffers. Consequently, upon reception of an *insufficiently regular* stream, such buffers are potentially exposed to overflow and starvation.

This motivates the design of a *packet pacer* – a system, which buffers an incoming, and potentially *insufficiently regular*, packet stream and releases the packets *sufficiently regularly* for a CR receiver to never be subject to buffer overflow, nor to starvation. In support of transitioning from dedicated, to commodity, hardware, such a *packet pacer* should be a *generic* software solution, requiring also only *generic* hardware.

B. Related Work

Previous work related to packet pacing can be categorised into techniques to avoid congestion control and reduce queueing delays, into specifically scheduling of packet transmission for video-streaming, and into implementability considerations for real-time packet transmission.

M. Hawari is with Cisco Systems Paris Innovation and Research Laboratory (PIRL), 92782 Issy-les-Moulineaux, France; email {mhawari}@cisco.com.

M. Hawari, J.-A. Cordero-Fuertes and T. Clausen are with École Polytechnique, 91128 Palaiseau, France; emails {mohammed.hawari,juan-antonio.cordero-fuertes,thomas.clausen}@polytechnique.edu.

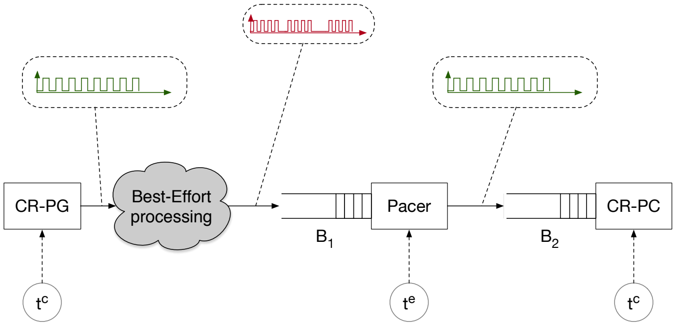


Figure 1: System overview: A stream of packets, generated by a Constant-Rate Packet-Generator (CR-PG), undergoes some Best-Effort processing, and hence a loss of regularity. Those packets are buffered into B_1 by the Pacer, which transmits them, with sufficient regularity, so that the receive buffer B_2 of a Constant-Rate Packet-Consumer (CR-PC) never undergoes overflow or starvation. Packet transmissions (by the CR-PG) and consumptions (by the CR-PC) occur at times, sorted into a sequence t^c.

1) Techniques to avoid congestion and reduce queuing delays: In [7], the effects of pacing on TCP flows are shown to bring higher fairness at the cost of lower overall throughput, especially as the number of concurrent flows increases. Further, [8] identifies a *point of inflexion* in the number of flows, above which pacing reduces throughput. This is explained by a synchronisation phenomenon when TCP flows experience simultaneous congestion window reductions, leading to inefficient network usage, first described in [9]. Pacing is also a building block of Bottleneck Bandwidth and Round-trip time (BBR) [10], a congestion control algorithm for TCP that reacts to congestion by reducing the pacing rate, instead of by solely reducing the TCP congestion window. In [11] and [12], dynamic pacing schemes are proposed to improve transport-layer performance in small-buffer networks and reduce congestion. Queue-Length Based Pacing (QLBP), described in [11], uses the number of buffered packets to determine the pacing rate. In [12], a poly-logarithmic-complexity online pacing algorithm is shown to reduce short-timescale burstiness, while still transmitting each packet before a given deadline.

2) Video streaming: In [13], the problem of scheduling packet transmissions so that a CR receiver experience neither buffer overflow nor starvation is formulated in the framework of *network calculus*, and the set of feasible packet departure curves is determined, given the characteristics of the receiver and the network. In [14], this problem is addressed for stored video, and an optimal packet transmission schedule is constructed, given a non-necessarily-constant packet consumption profile at the receiver. This is applicable, *e.g.,* to Variable Bit-Rate (VBR) video streaming.

3) Implementability: Given the real-time nature of the pacing task, implementing it as a piece of software proves challenging. In [15], the scalability of traditional implementations of pacing (e.g. the Linux Hierarchical Token Bucket (HTB) or the Fair Queue (FQ) queueing discipline (qdisc)) is shown to be limited for large number of flows to be paced, due to the intrinsic per-flow cost.

Software pacing without relying on timers is proposed in [16], by introducing a concept of *gap-frames*: interleaving carefully-sized IEEE 802.3 flow control frames (as defined in [17]) and packets from a flow of interest. IEEE 802.3 flow control frames are dropped by a packet switch on the network path, resulting in pacing of the flow of interest. In [18], a timer-based implementation is introduced, compared to the gap-frame approach and is experimentally shown to outperform the latter in terms of accuracy. That conclusion is challenged in [19], which uses the gap-frame approach to implement rate-control in a packet generator and evaluates the resulting pacing accuracy to be higher than in [16] and [18]. A possible explanation is that the implementation in [19] relies on the Data Plane Development Kit (DPDK) [20], which is designed to provide higher and more predictable performance than is the Linux network stack, which was used in [16] and [18].

C. Statement of Purpose

The objective of this paper is to propose a software-based packet pacer, able to run on commodity hardware, able to accept an *insufficiently regular* packet stream, and able to return a *sufficiently regular* packet stream, suitable for any CR receiver – including receivers expecting an SMPTE compliant packet stream. This is illustrated in figure 1.

Further, given that the aforementioned production *gateway devices* are opaque packets consumers, this paper – unlike *e.g.,* [13] – assumes that the pacer sees the Constant-Rate Packet Consumer (CR-PC) as an opaque component, *i.e.,* with an unknown packet-consumption curve.

The required real-time properties make running a pacer as software on commodity hardware challenging. This paper analyses and formalises those challenges. This paper also generalises the previously discussed use of gap-frames [16], [19] for accurate packet scheduling, into the notion of a Pacing-Assistant – which enables abstract expression and analysis of packet pacing algorithms.

Finally, within this framework, software pacing algorithms are analysed, are implemented, and are subjected to exhaustive experimental tests, confirming the viability of the postulated approach.

D. Paper Outline

The remainder of this paper is organised as follows. Section II formalises the pacing problem and section III discusses the limitations of a pure-software approach. Section IV introduces Pacing-Assistants (PA), as a form of hardware-assistance, and proposes algorithms using it. Those are supported by a theoretical analysis provided in section V, and are shown to be effectively implementable on general-purpose hardware in section VI. Section VII presents the associated experimental evaluation. Section VIII discusses the obtained results and their practical impact for media production. Section IX concludes this paper.

II. System Model

While the *theoretical* objective of packet pacing is to generate a *perfectly* constant-rate packet stream, the formalism of *time* *sequences* is introduced in this section, to provide a *practical*, quantitative definition of a *sufficiently-regular* constant-rate packet stream.

A. Time sequences

The regularity of packet stream is only described by the sequence of transmission times. Defined hereafter, the formalism of *time sequences* enables describing any sequence of recurring events.

Definition. *A* ***time sequence*** *is a nondecreasing sequence of times $t = (t_i)_{i\in\mathbb{N}}$, such that $\lim\limits_{n\to+\infty} t_n = +\infty$. The abstract event happening at time t_i is called the i-th* ***cycle*** *of t, or the i-th t-cycle.*

Definition. *The following metrics are used to quantify the potential periodicity of a time sequence.*

- *The* ***period*** *of t is, when it exists, $T := \lim\limits_{n\to+\infty} \frac{t_n}{n}$. This definition is equivalent to the average duration between two consecutive events,* i.e., $T := \lim\limits_{n\to+\infty} \frac{1}{n}\sum\limits_{i=1}^{n} t_i - t_{i-1}$.
- *The* ***frequency*** *of t is the inverse of the period, when it exists, $f := \frac{1}{T}$.*
- *The* ***Peak Period jitter*** *of t is:*

$$J_p\left(t\right) = \limsup_{i\geq 0} \left|T - \left(t_{i+1} - t_i\right)\right|$$

- *The* ***Asymptotic Long Term (ALT)*** *jitter of t is:*

$$J_{ALT}(t) = \inf_{\substack{i_0\in\mathbb{N}\\ u\geq 0}} \sup_{\substack{i\geq i_0\\ t_i\geq u}} \left|T(i - i_0) - (t_i - u)\right|$$

$J_P(t)$ asymptotically measures how far the duration between two consecutive cycles, $t_{i+1} - t_i$, is from the period T. The definition of $J_{ALT}(t)$ is interpreted by comparing t with a perfectly-regular time sequence t' whose i-th cycles occur at $t'_i = T \times i$. t and t' have the same period T. As it is perfectly-regular, t' can be used to measure the current time with a granularity of T, provided that a well-known number i of t'-cycles have elapsed. $J_{ALT}(t)$ asymptotically quantifies the maximal error on that measurement if t is used in place of t'. Therefore, $J_{ALT}(t)$ provides a certain measure of the irregularity of t when compared with the perfect time sequence t' of same period T.

In the rest of this paper, t^p is the time sequence corresponding to the packet transmission times by the pacer of figure 1. t^c is the time sequence corresponding to the packet production times by the Constant-Rate Packet Generator (CR-PG), and consumption times by the CR-PC[1]. The periods and frequencies are respectively denoted by T^p, T^c, f^p and f^c.

B. (b, f)-paced streams

In the following, a definition of a *sufficiently regular* constant-rate packet stream is given as a property, relating the time sequence t^p associated with packet transmission, with a buffer size b and a packet consumption rate f, both associated with the receiver. It is assumed that there is no delay between the sender and a receiver. Upon reception of the i-th packet, an infinite-sized buffer at the receiver would contain $i - ft^p_i$ packets. That motivates the following definition:

Definition. *A system sends a (b, f)-paced stream if the packet transmission time sequence $(t^p_i)_{i\in\mathbb{N}}$ are such that there exists a time, t_0, and an integer, i_0, satisfying:*

$$\forall i \geq i_0 : t^p_i \geq t_0 \implies 0 \leq (i - i_0) - f \times (t^p_i - t_0) \leq b \quad (1)$$

i.e., a virtual receiver, consuming packets at a rate f and with a b-sized input buffer, may choose an integer number i_0 and a time t_0, so that, dropping the i_0 first packets and only starting consumption after time t_0 prevents any overflow or starvation.

This definition reflects only the asymptotic behaviour of packet transmissions, *i.e.,* is sensitive neither to the initial variability of the stream, nor to the initial behaviour of the consumer. Moreover, the provided definition is still valid if the delay between the transmitter and the receiver is constant — and not null as initially assumed — because any constant delay can be captured in t_0, *i.e.,* the time packets start being consumed.

Using equation (1) for i and $i + 1$, taking the difference, and dividing by f, yields:

Property 1. *If a stream of packet transmission time sequence t^p is (b, f)-paced, then there exists an integer i_0 such that for all integers $i \geq i_0$:*

$$\left|\frac{1}{f} - \left(t^p_{i+1} - t^p_i\right)\right| \leq \frac{b}{f}$$

i.e., the buffer of a virtual receiver, as in the definition of a (b, f)-paced stream, never experiences variations larger than b between two arrivals.

Property 1 gives necessary conditions — *real-time* constraints — that must be verified if a stream is to be qualified as *sufficiently-regular*. Specifically, the $(i + 1)$-st packet can be output neither before time $t^p_i + \frac{1-b}{f}$ nor after time $t^p_i + \frac{1+b}{f}$. Consequently, if a (b, f)-paced stream is to be generated by software, the latter must be executed on a system providing (i) some notion of time consistent with the frequency f, so that software can wait at least $\frac{1-b}{f}$ between two transmissions, and (ii) execution-time guarantees so that a transmission can never occur more than $\frac{1+b}{f}$ after the previous transmission. Those constraints shall be used in section III to demonstrate the difficulty of pure-software pacing.

For the time sequence t^p associated with the transmission of a (b, f)-paced stream, property 1 only provides necessary conditions, not sufficient ones. As a counter-example, a packet stream transmitted at times $t'_i = Ti + \ln(i)$ verifies property 1 without being (b, f)-paced.

The following property uses the peak period jitter and ALT jitter to provide metrics, quantifying to what extent a stream

[1] The assumption here is, that the CR-PC would be able to – was it connected directly to CR-PG, such as would be the case for a video camera and a viewscreen – consume packets at the precise rate at which they were generated. Pacing is intended for compensating for the *best effort* processing variation that occurs in case when CR-PC and CR-PG are *not* directly connected, as in figure 1.

is constant-rate. In particular, it provides a necessary and a sufficient condition to qualify a (b, f)-paced stream.

Property 2. *t is a time sequence of frequency f, associated with the packet transmission times of a stream.*

1) *If the stream is (b, f)-paced, $J_{ALT}(t) \leq \frac{b}{2f}$.*
2) *Conversely, if $J_{ALT}(t) < \frac{b}{2f}$, then the stream is (b, f)-paced.*
3) *If t satisfies*

$$\exists i_0 : \forall i \geq i_0 : \left| \frac{1}{f} - (t_{i+1} - t_i) \right| \leq \frac{b}{f} \tag{2}$$

then $J_p(t) \leq \frac{b}{f}$.

4) *Conversely, if $J_p(t) < \frac{b}{f}$, then t satisfies equation (2).*

Proof: (See Appendix in the provided supplementary material) ■

While, as per property 1, a too high value of J_p is incompatible with sufficient regularity, the latter is shown to be equivalent to a low value of J_{ALT} by property 2. J_{ALT} shall therefore be used to verify the correctness of the algorithms developed in this paper.

III. Limitations of a pure software approach

In this section, a software execution model is given, and shown to be rich enough to describe commonly-used commodity hardware. As part of that model, *timers* are defined as components, enabling software to have access to time – which is necessary to transmit a sufficiently-regular CR stream. By inspecting the available timers on a commodity server and the achievable latency, software-pacing is finally proven to be challenging.

A. Software Execution Model

In this paper, software execution on a commodity server is modelled *asynchronously*. A server is modelled as a set of programmable components, each considered as a *reactive system*, maintaining an *internal state*, and receiving *notifications* from other components. Software run by a component consists of the specification, for each received notification, of a sequence of actions to be executed by the component. This sequence, called the notification *handler*, consists of actions which are either updates to the internal state, or transmissions of notifications to other components[2].

Not only can such an execution model describe *interrupt-driven* components — as a notification models an interrupt, and a handler models an Interrupt Service Routine (ISR) — but it can also describe components *busy-waiting* for some condition involving an *external state*, and executing some action as soon as that condition is verified. In that latter case, the notification is the external state update, and the handler is the action executed whenever the condition is satisfied. The *latency* of a component is the — generally unknown — execution duration of the handler associated with a received notification.

That execution model describes any commodity server based on an interruptible Central Processing Unit (CPU), and running a preemptive multitask operating system. Considering, *e.g.,* an x86_64 server running a Linux kernel, three types of components can be described, each exhibiting an asynchronous behaviour.

1) ***A user-space application:*** is composed of a sequence of instructions, either updating an internal state (such as registers or private memory), reading from or writing to an external state (such a read/write operation in shared memory, or a nonblocking system call), or performing a blocking system call. Therefore, the application is modelled as a component, interacting with external states and the operating system's kernel by way of notifications. For example, a system call is modelled as a notification sent to the operating system's kernel. The completion of a system call is modelled as the reception of a notification, whose handler models the subsequent instructions.

2) ***The operating-system kernel:*** registers ISRs within the interrupt controller, and, when a device (conceptually another component) raises an interrupt (conceptually sends a notification), the CPU jumps to the address of the corresponding ISR. In the absence of runnable threads, the kernel is idle and only waits to be interrupted.

3) ***In hardware:*** the CPU performs Input/Output (I/O) operation (*e.g.,* `inw` or `outw` instructions) or accesses nonlocal memory (via Direct Memory Access (DMA), or memory that does not reside in a register) by sending requests to the relevant components (*e.g.,* the memory controller or the DMA controller). The CPU effectively stalls until a response is notified back, which can be modelled as the subsequent instructions being the handler to that response.

B. Timers

While the aforementioned model does not allow to express a time at which a given action shall be performed, software-based generation of a paced packet stream still requires the ability to perform an action after a certain *release-time*. The solution is provided by a component, *a timer*, sending notifications at well-specified times. Any handler executed upon such a notification is therefore known to necessarily complete after a determined time. In particular, when a program is generating a (b, f)-paced stream, each packet transmission must occur upon notification from a timer.

Generalising the software architecture observed in the Linux kernel, timers available to the different components of a commodity server are classified in two categories. Timers from the first category — *clock event devices* in Linux terminology — are components, raising interrupts (periodically or at programmable times), so that software may have a notion of time. Timers from the second category are derived from clocks — *clock sources* in Linux terminology — which must be explicitly requested for the current time, expressed as a numerical timestamp. Timers from the second category are implemented by probing a clock for the current time, and sending a notification whenever the probed value is posterior to a given time.

[2] A more formal treatment of that topic is provided by *input/output automata* defined in [21].

Considering timers available to the three types of components described in section III-A:

*1) **In hardware**:* the CPU and chipset have direct access to the clocks originating from one or multiple oscillators on the motherboard. These clocks are used to maintain clock sources such as the Time Stamp Counter (TSC), and clock event devices such as the Local Advanced Programmable Interrupt Controller (LAPIC) timer. The TSC and the LAPIC timer will be used as typical examples of clock sources and clock event devices. This is also without loss of generality, as the following is also applicable to other clock sources and clock event devices (*e.g.,* the High Precision Event Timer (HPET) on x86_64 or the Generic Timer on ARM). The TSC is a register, containing the number of elapsed CPU-cycles since the last reset, and which exhibits an access latency in the order of a few dozen nanoseconds [22]. The LAPIC timer is a device operating asynchronously, and which is to be programmed to raise an interrupt at a programmed time. $t^{TSC,h}$ denotes the time sequence corresponding to TSC-cycles, and $t^{L,h}$ denotes the one corresponding to interrupts from the LAPIC timer.

*2) **The operating system kernel**:* can register an ISR to be executed upon notifications from the LAPIC timer. For the i-th interrupt raised by the LAPIC timer, the time at which the matching ISR starts is denoted $t_i^{L,k}$. That defines a time sequence $t^{L,k}$, different from $t^{L,h}$. Therefore, in kernel-space, software can be specified to be executed *after* times defined by that time sequence. The latency $t_i^{L,k} - t_i^{L,h}$ between the LAPIC i-th interrupt and the start of execution of the matching ISR may vary, intuitively making the time sequence $t^{L,k}$ less regular than $t^{L,h}$.

In kernel-space, software is also able to read the TSC register, compare its value to a given threshold, and execute an action, whenever the read value is larger than the threshold. This effectively enables the kernel to execute a sequence of actions, each starting at times corresponding to a time sequence $t^{TSC,k}$, different from $t^{TSC,h}$, as the former takes into account the latency between the update of the TSC, and the execution of the corresponding action in kernel-space.

*3) **In user-space**:* any system call, specified to suspend a program for a fixed amount of time — such as `nanosleep` — or to schedule a notification at a given time — such as `alarm` — relies on a clock event device (*e.g.,* the LAPIC timer), itself relying on an interrupt. It is serviced in kernel space, the corresponding notification is dispatched to the suspended program, which is finally awoken by the scheduler. Consequently, a user space program can specify actions to be executed at times corresponding to a time sequence $t^{L,u}$ with $t_i^{L,u} - t_i^{L,k}$ being the latency between the beginning of the i-th LAPIC ISR execution, and the time when the user-space program resumes execution.

A user space program can also compare the current time — obtained from a clock source by way of a system call such as `clock_gettime` — to a predefined threshold, so as to start executing an action after a specified time. That allows a user space program to specify actions to be executed according to a time sequence $t^{TSC,u}$ derived from the times at which the TSC is updated[3]. $t^{TSC,u}$ is different from $t^{TSC,k}$ due to the latency between the update of the TSC, and the execution of the specified action in user space.

C. Timer limitations: drift

In the setup in figure 1, if the packet pacing frequency f^p is slightly different from the packet consumption frequency f^c, the buffer B_2 is necessarily subject to overflow or starvation. The cause for this drift between f^c and f^p can be traced to insufficient *timer accuracy*.

The nominal frequencies of hardware clocks are usually given with a tolerance expressed in parts per million (ppm). For example, a $1\,\mathrm{MHz}$ clock with a tolerance of $100\,\mathrm{ppm}$ has an actual frequency between $999.9\,\mathrm{kHz}$ and $1.001\,\mathrm{MHz}$. Because of that inaccuracy, the notion of time given by any timer is never exact. Moreover, due to thermal fluctuations, the frequency of a periodic timer is not stable. Consequently, if a pacer and the CR-PC do not have access to a common source of time, drift will necessarily occur.

D. Latency

In section III-B, hardware and software layers of a typical commodity server were shown to have access to different timers. Conceptually, and following the terminology of section III-A, timer notifications from hardware to the different software layers are propagated with a certain latency. While most of the literature on real-time operating systems emphasises obtaining latency upper bounds, the definition of pacing given in section II is insensitive to the addition of any constant delay to the packet transmission time sequence. Consequently, the achievability of packet pacing only depends on obtaining a reduced amplitude between maximal and minimal latency. The sources of those *latency variations* are categorized as:

1) State-induced: From different initial states, software execution time will differ, yielding differences in latency. At the hardware layer, the micro-architectural state of the CPU — *e.g.,* the state of the caches, or the out-of-order execution pipeline — has an impact on the execution time of an instruction sequence. This variability can be analysed and exploited, as in [23]. Depending on the power-save state of the CPU, the start of execution of an ISR may be subject to a variable latency. At the software layer, the state of the kernel varies from one execution of an ISR to another, which may incur branching in some parts of the code, also yielding variable latency, detectable in user space.

2) Contention-induced: Latency variations may also be caused by *contention* for a given resource; when two tasks are contending for some resource the latency experienced by one of them varies, depending on whether it has access to the resource before the other one.

Preemption, *i.e.,* suspension of the current execution to process another notification, is a case of contention for a CPU core, and is responsible for latency variations. User-space software may be preempted by higher-priority tasks, scheduled

[3] A user space program can also directly probe the TSC without any system call.

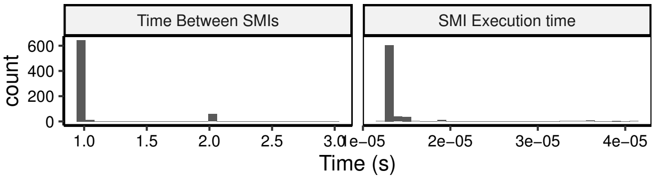


Figure 2: Distribution of the time between SMIs and their duration.

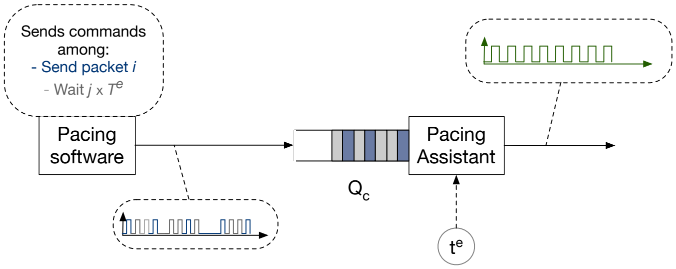


Figure 3: Architecture of a PA-based pacer

on the same CPU core. A user-space or kernel-space program may be preempted by a non-masked interrupt, because the matching ISR will then need to be executed.

Conversely, the existence of *non-preemptible* sections in the operating system, *i.e.,* sections which cannot be preempted by a high-priority task, is a cause for contention for a CPU core; because a high-priority task is unable to preempt the operating system, it is sporadically delayed, thus, experiences latency variations.

Contention-induced sources are already controllable with Real-Time Operating Systems (RT-OS) such as the PREEMPT-RT patch for the Linux kernel [24] or Xenomai [25]. As much as possible, RT-OSs reduce non-preemptible sections in their kernel, to allow a task, specified as high-priority by the user, to always preempt a low-priority task. For example, the PREEMPT-RT patch replaces the majority of spinlocks from the kernel source code with mutexes, allowing preemption to occur at those points. Consequently, an RT-OS has a lower maximal latency, especially on a highly-loaded system as shown in [24].

However, System Management Interrupts (SMI) are not maskable, may happen sporadically, and do execute transparently to the kernel, *i.e.,* instead of being serviced by an ISR specified by the kernel, the current code execution is suspended to execute firmware code, opaquely to the operating system. SMIs may indistinctly preempt the execution of kernel code or user-space code and are impossible to disable, making them a major source of latency variations, even on an RT-OS. Reducing non-preemptible sections in the operating system's kernel (as performed by the PREEMPT-RT patch for the Linux Kernel) has no impact on SMIs, as those are executed in a context which is transparent to the kernel. A more detailed analysis of the impact of SMIs on system performance is given in [26].

E. Quantitative analysis of the impact of SMIs

In the following, the pacer of figure 1 is a *pure-software* pacer, *i.e.,* is only implemented as a a reactive system, driven by timers, described in section III-B. Due to the latency variations, described in section III-D, the instants at which packets are enqueued into the Network Interface Card (NIC) for transmission, are subject to peak period jitter, itself yielding peak period jitter on the packet transmissions on the wire.

Measurements performed in [27] suggest that SMIs are responsible for a peak period jitter in the order of 20 µs. That is experimentally confirmed by running the `hwlat_detector` tracer, which is part of the Linux tracing subsystem. `hwlat_detector` consists of kernel-space code, specifically designed to measure the impact of SMIs, by polling the TSC in an uninterruptible loop. SMIs are detected when the TSC evolution undergoes irregular jumps, as those mean that the current CPU core was executing non-kernel code, *i.e.,* it was servicing an SMI. Figure 2 shows the obtained results, and confirms that SMIs are happening frequently (every second), and incur a latency ranging from 10 µs to 40 µs.

For a SMPTE-compliant stream received by a typical gateway device, the target frequency f^p of packet transmissions times t^p is in the order of $f^p = 135\,000\,\text{Hz}$, while the receive buffer size is in the order of $B_2 = 4$. As per property 1, the maximal admissible peak period jitter is $J_p(t^p) = \frac{B_2}{f^p} \approx 30\,\mu\text{s}$, which approaches the measured and sporadic additional latency from SMIs. Therefore, in the context of media production, software generation of a (B_2, f^p)-paced stream is potentially unreliable. That result is quantitatively and qualitatively confirmed by the experimental results presented in section VII.

IV. Pacing with a Pacing-Assistant

In sections III-C and III-E, the peak period jitter obtained by pure software pacing methods was shown to be at least in the order of 40 µs, which is too high for the media production use cases described in the introduction.

As the root cause to that jitter is the unpredictable and unavoidable latency spikes incurred by CPU preemption to service SMIs, a natural solution consists of delegating the time-sensitive tasks of *sending* packets, and of *waiting* between transmissions, to an uninterruptible external system, communicating with the main server through a *command queue*. That approach enables packet-pacing with the required accuracy (contrary to a purely software-based one), while still being implementable with commodity servers and networking hardware (see section VI), *i.e.,* in a commodity data centre.

A. Assisted Pacing

That uninterruptible external system is modelled by an abstract component, called a Pacing-Assistant (PA), capable of sending packets spaced by a *precise* amount of time, and not subject to the previously discussed jitter sources, as it is uninterruptible. In other words, that component can send packets, synchronously with a time sequence of negligible ALT-jitter. Defined as an abstraction in this section, a PA is shown to be constructible with general-purpose networking hardware in section VI.

Figure 3 depicts the architecture of a pacer relying on a PA. A PA is periodically notified by a timer, at times defining a time sequence t^e, with a period T^e, and frequency f^e. The

Algorithm 1: Packet pacing

```
Primitive      : wait, send, dur(p) as defined in section IV-A
Primitive      : dequeuePacket: returns an input packet from $B_1$
Parameter      : $n_{min}$: minimum value for wait
Parameter      : $n_{max}$: maximum value for wait
Parameter      : $\tau$: target pacing period (in $t^e$ cycles)
Precondition   : $n_{max} > 2n_{min}$
Precondition   : $\forall p : n_{min} \leq \tau -$ dur(p)
Precondition   : dequeuePacket always returns a packet.
$s \leftarrow 0$;
while True do
    if $s < 1$ then
        p $\leftarrow$ dequeuePacket();
        send(p);
        $s \leftarrow s + \tau -$ dur(p);
    end
    else if $s \geq n_{max} + n_{min}$ then
        wait($n_{max}$);
        $s \leftarrow s - n_{max}$;
    end
    else if $s \leq n_{max}$ then
        wait($\lfloor s \rfloor$);
        $s \leftarrow s - \lfloor s \rfloor$;
    end
    else
        wait($n_{min}$);
        $s \leftarrow s - n_{min}$;
    end
end
```

software part of the pacer communicates with the PA by way of *commands*, inserted into a command queue Q_c.

At each t^e-cycle, the PA dequeues a command from Q_c, if one is available. Commands are one of the following categories:

- wait(n) : when this command is dequeued by the PA, the next n t^e-cycles are skipped, which is equivalent to waiting $n \times T^e$. Afterwards, the next command in Q_c is dequeued, if available. It is assumed that there is a minimum value n_{min} and a maximum value n_{max} to the admissible values for n. In order to prevent holes in the range of values that PA can wait:
$$n_{max} > 2 \times n_{min} \tag{3}$$
- send(p) : when this command is dequeued by the PA, the transmission of packet p starts, and lasts for a number of cycles, dependent on the size of p. After the transmission completes, the next command in Q_c is dequeued, if available.

The *cost* of a command is the number of t^e-cycles spent processing it, *i.e.,* the cost of wait(n) is n, and the cost of send(p) is the number of t^e-cycles necessary to transmit p, denoted by dur(p).

B. PA-based free-running pacing

As a PA is able to send packets aligned with a time sequence t^e of negligible jitter, a software pacer only needs to enqueue a sequence of wait and send operations, whose execution (upon dequeueing by the PA) will generate a (B_2, f^c)-paced stream. τ is the target *number of t^e-cycles* between two consecutive packets. For a target period T^c, τ is defined as the ratio $\frac{T^c}{T^e}$. Algorithm 1 specifies the sequence of operations to enqueue, so that the PA generates a stream with period T^c.

As used in algorithm 1, the wait and send primitives **enqueue** the corresponding operation in the command queue Q_c if it is not full, and are blocking otherwise.

Assumption 1. *The command queue Q_c never starves,* i.e., *at each t^e-cycle, the PA is always either processing a wait or a send command. Also, the commands are reliably enqueued in Q_c.*

Algorithm 1 alternates between issuing a sequence of wait commands, and issuing a single send command. s is the (possibly fractional) number of t^e cycles between the execution (by the PA) of the last enqueued wait command and the execution of the next send command to be enqueued.

Thus, a send command is enqueued as soon as there is less than a full t^e-cycle in s. If $s > \lfloor s \rfloor$ (*i.e.,* s is fractional), the remaining fraction of t^e-cycle is accumulated for the next sequence of wait operations. Also, as the parameter passed to wait must be within $[n_{min}, n_{max}]$, the algorithm slices s appropriately, so that only compliant wait calls are performed.

Discussion: Algorithm 1 uses the explicit value τ. If t^e and t^c are neither derived from the same oscillator, nor otherwise synchronized, T^c is not necessarily a rational multiple of T^e. Thus, the quotient $\tau = \frac{T^c}{T^e}$ cannot be meaningfully digitally represented. Also, as stated in section III-C, the nominal values of T^e, and T^c are given with a non-zero tolerance (due, *e.g.,* to the underlying hardware being subject to thermal noise), and thus, cannot be used to compute τ.

Therefore, algorithm 1 is only usable when the timers originating the time sequence t^e and t^c are synchronized, so that τ is known exactly.

C. PA-based frequency-controlled pacing

When τ is unknown, *frequency-controlled* pacing replaces τ with a *frequency-controller*. A frequency-controller is an external signal F, such that, after the elapse of u t^e-cycles, $F(u)$ estimates the total number of packets transmitted by a CR packet generator of period T^c.

Formally, a frequency-controller is an integer-valued function F, satisfying:
$$\lim_{u \to \infty} \frac{u}{F(u)} = \frac{T^c}{T^e} \tag{4}$$

Being an integer-valued function, F is an alternative to an explicit value τ, as it avoids the representation problem described in the discussion of section IV-B.

Algorithm 2 uses F to achieve pacing at the target frequency f^c. That algorithm is derived from algorithm 1, but replaces the input parameter τ with a variable τ_{cur}. The core idea driving frequency-controlled pacing, consists of using F to periodically update τ_{cur}. The period of those updates is determined by an input parameter W.

Algorithm 2 maintains a variable y_{now}, containing the total cost of all enqueued operations at a given point of the execution. Under assumption 1, this means that, if an operation is enqueued when $y_{now} = y$, then it will be executed at y-th t^e-cycle by the PA. The value y_{last} of y_{now} at the last update of τ_{cur} is also maintained. Every W t^e-cycles, τ_{cur} is updated to a new value $\frac{y_{now} - y_{last}}{F(y_{now}+W) - F(y_{last}+W)}$. As estimated by F,

Algorithm 2: Controlled packet pacing

```
Primitive     : Same as in algorithm 1
Parameter     : $n_{min}, n_{max}$ as in algorithm 1
Parameter     : $F$: frequency-controller
Parameter     : $W$: update window
Precondition  : $n_{max} > 2n_{min}$
Precondition  : $\forall p, \forall a \in \mathbb{N}$ :
                $n_{min} \leq \frac{W}{F(a+W)-F(a)} - \texttt{dur(p)}$
$s \leftarrow 0$;
$\tau_{cur} \leftarrow 0$;
$y_{last} \leftarrow -W$;
$y_{now} \leftarrow 0$;
while True do
    if $s < 1$ then
        if $y_{now} - y_{last} \geq W$ then
            $\tau_{cur} \leftarrow \frac{y_{now}-y_{last}}{F(y_{now}+W)-F(y_{last}+W)}$;
            $s \leftarrow 0$;
            $y_{last} \leftarrow y_{now}$;
        end
        p ← dequeuePacket();
        send(p);
        $s \leftarrow s + \tau_{cur} -$ dur(p);
        $y_{now} \leftarrow y_{now} +$ dur(p);
    end
    else if $s \geq n_{max} + n_{min}$ then
        wait($n_{max}$);
        $s \leftarrow s - n_{max}$;
        $y_{now} \leftarrow y_{now} + n_{max}$;
    end
    else if $s \leq n_{max}$ then
        wait($\lfloor s \rfloor$);
        $s \leftarrow s - \lfloor s \rfloor$;
        $y_{now} \leftarrow y_{now} + \lfloor s \rfloor$;
    end
    else
        wait($n_{min}$);
        $s \leftarrow s - n_{min}$;
        $y_{now} \leftarrow y_{now} + n_{min}$;
    end
end
```

that value is the average (over a duration of $y_{last} - y_{now}$ t^e-cycles) number of t^e-cycles between two packet arrivals of a CR-stream of period T^c. In other words, algorithm 2 performs the same pacing as does algorithm 1, but with a parameter, τ, adjusted periodically to reflect the spacing between packet transmissions estimated by F.

Also, at a given point of the execution time, frequency-controlled pacing only needs to evaluate F in $F(y_{now}+W)$ and $F(y_{last}+W)$, *i.e.,* two digitally-representable, finite-precision values, making frequency-controlled pacing implementable.

Discussion: Frequency-controlled pacing is designed to avoid drift due to the numerical value $\tau = \frac{T^c}{T^e}$ not being accurately accessible. Algorithm 2 thus implicitly extracts the target pacing rate from an external signal (the frequency-controller), which is therefore required.

V. Analysis

This section analytically quantifies the period and ALT-jitter of the packet transmissions obtained when applying the algorithms detailed in section IV. As per property 2, those metrics are sufficient to assess whether the transmissions are (B_2, f)-paced. Also, the correctness of those algorithms is verified.

A. Safety

A program is safe when it can be guaranteed that no program execution will cause an undesirable state to be reached. It is achieved for algorithms 1 and 2 if and only if the arguments passed to the `wait` primitive are within the interval $[n_{min}, n_{max}]$.

The two algorithms assume that, at any time, for any dequeued packet p, and for any value of τ (either given, for algorithm 1, or dynamically computed for algorithm 2) $n_{min} \leq \tau - \texttt{dur(p)}$. The opposite would require the PA to somehow wait less than n_{min} cycles, *i.e.,* that the pacer is requested to pace at a higher-than-its-maximum-frequency. Considering both algorithms at the beginning of the j-th iteration of the `while` loop, the values of the state variables are denoted with index j so that, *e.g.,* the value of s is s_j. $\tau_j = \tau$ for algorithm 1, and $\tau_j = \tau_{cur}$ at the j-th iteration for algorithm 2.

Property 3. *s_j satisfies the following.*

1) *For all $j > 0$, $s_j \in [0, 1) \cup [n_{min}, +\infty)$.*
2) *$s_j \geq 1 \implies n_{min} \leq s_j - s_{j+1} \leq n_{max}$.*

Proof: (See Appendix in the provided supplementary material) ■

Algorithms 1 and 2 only call the `wait` primitives when $s_j \geq 1$, with parameter $s_j - s_{j+1}$. The second part of property 3 proves that parameter is always in $[n_{min}, n_{max}]$, hence the safety of the algorithms.

B. Free-running pacer period

Analysing the state of algorithm 1 at the beginning of the j-th iteration of the main loop, s_j is defined as the value of state variable s, N_j as the total count of enqueued `send` operations, and y_j as the total cost of all enqueued operations. Initially, $s_1 = 0$, $N_1 = 0$ and $y_1 = 0$. At each iteration, s is decremented by the cost of the enqueued PA-operation and incremented by τ if a packet is transmitted, yielding:

$$s_j = \tau N_j - y_j \tag{5}$$

For all integers $i > 0$, j_i is defined, so that the j_i-th iteration of the loop enqueues the `send` operation sending the i-th packet. Then, equation (5) implies:

$$s_{j_i} = (i-1)\tau - y_{j_i} \tag{6}$$

As $s_{j_i} \in [0; 1[$ by construction, and y_{j_i} is an integer number of t^e-cycles, then $y_{j_i} = \lfloor (i-1)\tau \rfloor$. Per assumption 1, y_{j_i} is also the number of elapsed t^e-cycles, when the i-th packet will be transmitted. The time of that transmission is therefore given by:

$$t_i^p = t^e_{\lfloor (i-1)\tau \rfloor} = \tau T^e i + o(i) \tag{7}$$

This proves that the period of the paced stream is τT^e.

C. Frequency-controlled pacer period

A similar analysis is performed for algorithm 2. At the j-th iteration of the main loop, y_j is the total cost of all enqueued operations. j_i is defined such that, the i-th `send` operation is enqueued at the j_i-th iteration of the main loop.

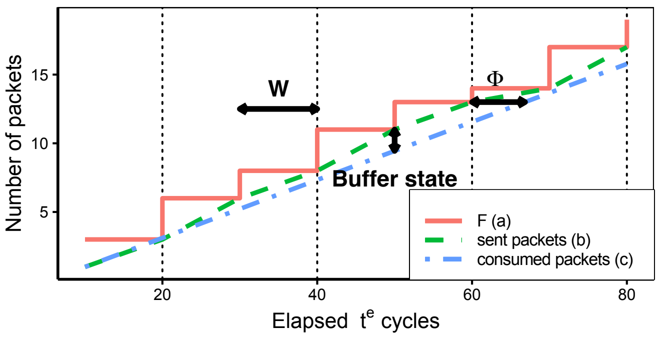


Figure 4: Frequency-controlled pacer: F, sent packets , consumed packets by a perfect CR-consumer. Note: the CR consumer on the figure starts at an arbitrary time, *i.e.,* an X-axis offset, and after an arbitrary number of dropped packets, *i.e.,* a Y-axis offset. The ALT jitter is the asymptotic maximum horizontal distance Φ between curve (b) and an optimally-shifted curve (c)

Property 4. *With the previous notations, y_{j_i} satisfies:*

$$y_{j_i} = \left\lfloor W \frac{i - 1 - F(W(k(i) - 1))}{F(Wk(i)) - F(W(k(i) - 1))} \right\rfloor + W(k(i) - 1) \tag{8}$$

with $k(i)$ defined as

$$k(i) = \max\{k \geq 1 \mid F(W(k-1)) + 1 \leq i\}$$

Furthermore, τ_{cur} is updated exactly at the iterations of the main loop where the state satisfies $y_{now} - y_{last} = W$.

Proof: (See Appendix in the provided supplementary material) ■

By definition of $k(i)$, and by maximality:

$$F(W(k(i) - 1)) + 1 \leq i < F(W(k(i))) + 1$$

The second precondition of algorithm 2 yields:

$$F(W(k(i) - 1)) + \frac{W}{n_{min}} \geq F(W(k(i)))$$

Combining those two last equations:

$$i - 1 \leq F(W(k(i))) \leq i + \frac{W}{n_{min}} - 1$$

yielding: $F(W(k(i))) = i + o(i)$. Applying in the equation given by property 4 yields:

$$y_{j_i} = O(1) + \frac{W(k(i))}{F(W(k(i)))}(i + o(i)) = \frac{T^c}{T^e} i + o(i)$$
$$t^p_i = T^c i + o(i)$$

Consequently, algorithm 2 results in a packet transmission time sequence of frequency f^c.

D. ALT-Jitter

The ALT-jitter of the stream, generated by algorithms 1 and 2, can be evaluated for each of the algorithms, as follows

1) ALT jitter for the free-running pacer: For all i_0 and t_0, and i so that $i \geq i_0$ and $t^p_i \geq t_0$ and from equation (7):

$$\begin{aligned} &|T^p(i - i_0) - (t^p_i - t_0)| = |\tau T^e(i - i_0) - (t^e_{\lfloor i\tau \rfloor} - t_0)| \\ \leq &|T^e(\lfloor i\tau \rfloor - \lfloor i_0\tau \rfloor) - (t^e_{\lfloor i\tau \rfloor} - t_0)| + |T^e(\{i\tau\} - \{i_0\tau\})| \\ \leq &\sup_{\substack{j \geq \lfloor i_0\tau \rfloor \\ t^e_j \geq t_0}} |T^e(j - \lfloor i_0\tau \rfloor) - (t^e_j - t_0)| + T^e \end{aligned}$$

Considering the supremum over all possible i, then the infimum over all i_0 and t_0 yields:

$$J_{ALT}(t^p) \leq J_{ALT}(t^e) + T^e \tag{9}$$

The ALT-jitter of the stream, $J_{ALT}(t^p)$, is thus bounded by a component originating from the internal jitter of the PA, and a component bounded by one period T^e. The latter component disappears if T^p is a multiple of T^e, *i.e.,* if $\tau \in \mathbb{N}$.

2) ALT-jitter for the frequency-controlled pacer: For algorithm 2, t^e is assumed to be perfectly periodic, *i.e.,* that $\forall i : t^e_i = T^e i$. This is justified by the fact that the ALT-jitter of t^e is negligible with respect to the one introduced by the variations in τ_{cur}. Figure 4 illustrates equation (8) by representing (a) the function F, (b) the total number of packets transmitted by the pacer, and (c) the cumulative number of consumed packets by a perfect CR-consumer. On this figure, bounding the ALT jitter consists of asymptotically bounding the horizontal distance Φ between (b) and (c).

$A_W(F)$ and $C_W(F)$ are defined as:

$$\begin{cases} A_W(F) & := T^e \limsup_k |\tau(F(W(k+1)) - F(Wk)) - W| \\ C_W(F) & := T^e \inf_{\substack{k_1 \in \mathbb{N} \\ k_0 \in \mathbb{N}}} \sup_{k > k_0} |\tau F(kW) - kW + k_1 W| \end{cases}$$

$A_W(F)$ is the maximum time interval error when using F to measure a time interval of length W t^e-cycles. $C_W(F)$ measures how accurately τF tracks the number of elapsed t^e cycles on the long term. An upper bound on the ALT-jitter is given by Property 5.

Property 5. *Under the assumption that t^e is perfectly periodic, the ALT-jitter of the stream t^p generated by algorithm 2 is bounded by:*

$$J_{ALT}(t^p) \leq A_W(F) + C_W(F) + T^e \tag{10}$$

Proof: (See Appendix in the provided supplementary material) ■

This bounds the ALT jitter of t^p into three components:

- $A_W(F)$, which bounds the error $|\tau_{cur} - \tau|$;
- $C_W(F)$, which quantifies how accurately F tracks the number of packets consumed by a perfect consumer, when evaluated at multiples of W; and
- T^e, which bounds the rounding error, since τ_{cur} is not necessarily integer.

VI. Constructing a Pacing-Assistant and a frequency-controller

Algorithms 1 and 2 rely on a Pacing-Assistant, providing hardware assistance for pacing. Algorithm 2 also requires a frequency-controller. Procedures for building those elements with commodity hardware are described in this section.

A. Constructing a Pacing-Assistant

This software-based method for implementation of a Pacing-Assistant relies on a commonly-available NIC, and the network infrastructure connected to it. The NIC and network infrastructure must satisfy the following:

- The NIC has a well-defined line rate r, and is able to saturate its output interface at rate r, by consuming packets from a Transmit (TX) queue. The latter consists of an in-memory queue, into which software inserts packets to be sent, and out of which the NIC dequeues packets and actually transmits them on the wire.
- There is a type of packets — *gap* packets — consumed by the NIC at rate r and dropped by some equipment along the network path, before reaching CR-PC, as introduced in [16], [19]. Two possible options for gap packets are (i) packets with a bad Cyclic Redundancy Check (CRC), dropped by any receiver but still effectively consumed by the NIC at rate r, and (ii) IEEE 802.3 Flow Control frames, which should be dropped by any network equipment (*e.g.,* a layer 2 switch) with disabled IEEE 802.3 Flow Control. The maximum IEEE 802.3 frame size (usually at least 1538 bytes) must also be larger than twice the minimum frame size (usually around 64 bytes), so that condition (3) from section IV-A holds.

Given that, a PA can be implemented as follows. The PA command queue Q_c is the TX queue of the NIC. The timer of the PA (which events occur at time sequence t^e) is implemented as the *byte-clock* of the NIC, *i.e.,* the timer whose cycles correspond to the transmission of a byte from the TX queue to the wire. The `wait(`u`)` and `send(`p`)` PA commands are implemented as u-sized gap packets and p packets, respectively.

As discussed in section IV-B, for an Ethernet NIC whose byte-clock is derived from a local oscillator, and is not synchronized with any external source, drift will occur, effectively limiting the applicability of algorithm 1.

However, the Synchronous Ethernet (SyncE) standard [28], [29] specifies a network architecture, where the Ethernet physical clock of multiple devices — Network Elements (NE) — is derived from a common master by way of a Phase-Locked Loop (PLL), replacing the local free-running oscillator. Consequently, if the NIC used to build the PA relies on SyncE, and the CR-PC relies on the same external time source, then the nominal value T^c/T^e is exact and algorithm 1 can be used.

B. Constructing a frequency controller: basic version F_b

The frequency-controlled pacer of algorithm 2 relies on a function, F, satisfying equation (4), section IV-C. If the pacing software is implemented on a system receiving notifications from two timers, at respective time sequences t^α and t^β, and of respective periods T^c and T^e, then at the u-th notification from the second timer, it can count $F_b(u)$ as the total number of received notifications from the first timer, *i.e.,* $F_b(u)$ is interpreted as the cumulative number of consumed packets by the CR-PC (as t^α is of period T^c) after u t^e-cycles have elapsed (as t^β is of period T^e). Following, and by definition:

$$F_b(u) := \max\left\{i \in \mathbb{N} \;\middle|\; t_i^\alpha \le t_u^\beta\right\}$$

Also, by definition $t^\alpha_{F_b(u)} \le t^\beta_u$ and by maximality $t^\alpha_{F_b(u)+1} > t^\beta_u$.

$$\frac{u}{F_b(u)} \ge \frac{u}{F_b(u)} \frac{t^\alpha_{F_b(u)}}{t^\beta_u} \xrightarrow{u \to +\infty} \frac{T^c}{T^e}$$

And

$$\frac{u}{F_b(u)} < \frac{u}{F_b(u)} \frac{t^\alpha_{F_b(u)+1}}{t^\beta_u} \xrightarrow{u \to +\infty} \frac{T^c}{T^e}$$

Thus, F_b satisfies equation 4 and is thus a frequency-controller. Consequently, if the system on which algorithm 2 is implemented receives notifications from two timers of period T^c and T^e, a basic version of a suitable frequency controller can be constructed. Methods to implement access to such timers are detailed in the following.

1) Receiving notifications at t^α: In the abstract setup, depicted in figure 1, the CR-PG is transmitting packets with a period T^c. If t^α is defined as the arrivals at the pacer of an *auxiliary* CR packet stream, output directly by the CR-PG, and bypassing best-effort processing, then t^α is a time sequence of period T^c. In the context of media processing, often relying on IP multicast streams, such an auxiliary stream can be implemented by simply replicating (in the network path) the original media stream transmitted by the CR-PG.

2) Receiving notifications at t^β: At the u-th t^e-cycle, $Q_c(u)$ is the total cost of all the operations in Q_c. With assumption 1 from section IV-B, the command queue Q_c is never subject to starvation. In algorithm 2, y_{now} is the cost of all the commands enqueued by the software. With $y_{now}(u)$ as the value of y_{now} at the u-th t^e-cycle, and per assumption 1:

$$Q_c(u) = y_{now}(u) - \max\left\{i \in \mathbb{N} \;\middle|\; t_i^e \le u\right\}$$

As $Q_c(u)$ is bounded, $\lim\limits_{u \to +\infty} \frac{y_{now}(u)}{u} = f^e$. The time sequence t^β is defined as the sequences of times at which y_{now} increases by 1, *i.e.,* $\forall k : y_{now}(t_k^\beta) = k$. Then:

$$\lim_{k \to +\infty} \frac{t_k^\beta}{k} = \lim_{k \to +\infty} \frac{t_k^\beta}{y_{now}(t_k^\beta)} = T^e$$

The period of t^β is thus T^e, making that time sequence suitable for constructing a frequency-controller F_b. Following that construction, at any time, the value $F_b(y_{now})$ used in algorithm 2 is the number of elapsed t^α cycles and $F_b(y_{last})$, is the value of $F_b(y_{now})$ at the previous iteration.

C. Constructing F: N_W-regularized version, F_r

Algorithm 2 only evaluates F at multiples of W (see section V), and thus, estimates τ_{cur} at the k-th update as $\frac{W}{F(kW)-F((k-1)W)}$. Consequently, the variations of τ_{cur} will increase with the variability of $F(kW) - F((k-1)W)$. Increasing W has the disadvantage of reducing how often τ_{cur} is updated, making the algorithm more likely to deviate from the targeted period T^c.

Given parameters W and N_W, the N_W-regularized construction of F, F_r is derived from F_b as obtained in section VI-B, and is defined as:

$$F_r(t) = \frac{1}{N_W} \sum_{l=0}^{N_W - 1} F_b(t - lW)$$

Following that construction, instead of periodically updating τ_{cur} by using the increments $F_b(W + y_{now}) - F_b(W + y_{last})$, algorithm 2 uses a moving average of these increments over the past $N_W \cdot W$ t^e-cycles. That approach allows to smooth the variations of the basic version F_b, but still keeps updating τ_{cur} every W t^e-cycles.

D. Implementation considerations of algorithms 1 and 2

The validity of the analysis provided in section V, and hence the correctness of the obtained pacing system is conditioned by the assumption 1 in section IV-B, *i.e.,* the non-starvation of the command queue Q_c. This is equivalent to the non-starvation of the TX queue of the NIC, which motivates the following implementation choices.

First, algorithms 1, and 2 were implemented as a user-space application, using DPDK for direct access to the NIC. This allows to busy-wait on the state of the TX queue, and to enqueue a packet as soon as possible, hence maintaining the TX queue full as often as possible. Using DPDK instead of the network stack of the OS kernel also prevents any kernel-originated cross-traffic, which would be injected into the TX queue along with the gap packets and the stream to be paced, and would behave as spurious additional `wait` commands.

Then, to minimise the number of times the OS kernel suspends that user-space application, it is assigned to a specific CPU core, isolated from any other tasks by the `isolcpus` kernel boot option. All hardware interrupts are also rerouted to a CPU core different from that. To avoid any spurious page fault, a call to `mlockall` is performed to guarantee that all the memory used for pacing is locked into physical memory.

As the pacing software thread is alone to be runnable on its assigned CPU core, the Linux kernel is prevented from issuing periodic ticks on that CPU core, as the `nohz_full` kernel boot option is enabled, and the kernel, compiled with the `CONFIG_NO_HZ_FULL` option. The used version of the Linux kernel is the 4.19.3, without the PREEMPT-RT patch, as, in the absence of any concurrent task on the used CPU core, and with all hardware interrupts rerouted to different CPU cores, there is no need for preemption, and hence, no reason to make the kernel more preemptible.

Finally, SMIs are the only remaining cause for the preemption of the pacing software. As stated in section III-E, SMIs may last up to $40\,\mu s$, while the time taken by a typical $10\,\text{Gbit/s}$ NIC to drain all the 1500-bytes-sized packets from its 512-packets TX queue is in the order of $0.5\,\text{ms}$. As a consequence, SMIs are unlikely to be responsible for the starvation of the TX queue, and, therefore, do not impact assumption 1.

VII. Experimental Evaluation

The pacing algorithms 1 and 2 are evaluated in experimental scenarios sourced from a media production setup. The experimental setup and methodology are described in section VII-A. Qualitative and quantitative results are provided in section VII-B, with metrics derived from the peak period jitter and ALT jitter of the paced stream t^p. The frequency-controlled approach is experimentally analysed in section VII-C, and an experimental estimation of the two constants $A_W(F)$ and $C_W(F)$, evoked in section V, is also provided.

Setup	Video output	Video failure type
No Intermediary	Yes	n/a
Linux + iptables	No	Permanent
DPDK forwarding	Yes with failures	Sporadic
pacer (freerun)	Yes with failures	Periodic
pacer (controlled)	Yes	n/a

Table I: Video status of the CR-PC

A. Setup and methodology

The setup used to evaluate algorithms 1 and 2 is an implementation of the abstraction from figure 1, with (i) a CR-PG and a CR-PC, both implemented as commodity off the shelf broadcasting pieces of equipment sending and receiving a SMPTE 2022-6 1080i59.94 CR video stream, at a nominative packet rate of $f^c = \frac{4497 \times 30}{1.001} \approx 134\,775.22\,\text{pkts/s}$, (ii) a pacer implemented as software running on an x86_64 Linux server, and implemented as described in section VI (iii) the time sequence t^c (corresponding to packet consumptions by the CR-PC) defined by the signal output by a *tri-level sync generator*, *i.e.,* a piece of broadcasting equipment distributing an out-of-band common clocking signal to CR-PG and CR-PC.

The pacer runs in either frequency-controlled, or free-running, mode, depending on the algorithm under test. The PA is as described in section VI-A, and uses IEEE 802.3 flow control frames. The pacing software is implemented as described in section VI-D. A $10\,\text{Gbit/s}$ network switch ensures the interconnection between the CR-PG, the CR-PC, and the pacer. As described in section VI-B, F is constructed by way of an auxiliary stream, implemented as a statically-configured multicast replication of the stream transmitted by the CR-PG. The stream transmitted by the pacer is also replicated to a device — detailed below — for quantitative evaluation.

Unless stated otherwise, the frequency-controlled mode uses a value $W = 1\,\text{s} = 1\,250\,000\,000\,\text{B}$ and an N_W-regularised F with $N_W = 2$. The best-effort processing stage depicted on figure 1 is not part of the experimental setup, as the pacer accumulates the packets of the stream of interest into B_1, making it lose all its timing properties. Given this setup, the methodology to obtain quantitative results is described from the perspective of *instrumentation*, and *baseline experiments*.

1) Instrumentation: An accurate evaluation of the jitter of a periodic stream is difficult to obtain using software methods on commodity platforms, for the exact same reasons as those motivating the use of hardware-assisted pacing, *i.e.,* variable latency due to unavoidable causes, *e.g.,* SMIs. Consequently, the measurements are performed using dedicated hardware, implemented on a NetFPGA-SUME programmable card, [30]. The *Open Source Network Tester (OSNT)*[4][31] for this card appears suitable capturing packets with accurate timestamps. However, a suboptimal Direct Memory Access (DMA) design of OSNT prevents packets and timestamps acquisition, at a rate as high as that of the flow of interest. Consequently, for the purpose of the experiments in this section, the original DMA design in OSNT was replaced by the Xilinx DMA/Bridge Subsystem for PCI Express (XDMA) intellectual property core

[4] OSNT offers a Verilog/VLSI design for the NetFPGA SUME card.

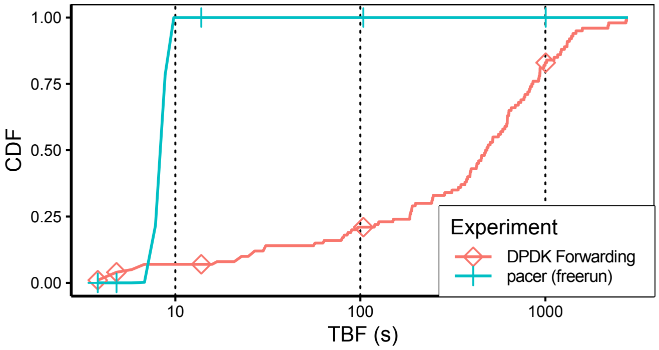


Figure 5: Empirical distribution of the Time Between Failures

– and, a minimal DPDK driver for acquisition of packets and of timestamps, was implemented.

Timestamp acquisition on this platform is dependent on the accuracy (how valid the nominal frequency is) and jitter of its internal clock. It is assumed (from the datasheet [32] of the Silicon Lab Si5324 oscillator used on the NetFPGA SUME board), that this jitter, being in the order of the nanosecond, is negligible for the results presented in this paper. However, because of the accuracy limitations of any hardware oscillator — as described in section III-C — and because no clock drift compensation was implemented, the absolute values of the measured packet transmission timestamps t^p are not independently exploitable, and must be compared to baseline values, obtained as described in the following.

2) Baseline experiments: The free-running and frequency-controlled PA-based pacers are compared to three baseline experiments:

- No intermediary element exists between the CR-PG and CR-PC. This is expected to show the most regular behaviour.
- The PA-based pacer is replaced with a Linux `iptables`-based setup, redirecting the received stream from the CR-PG to the CR-PC, and using the NetFPGA board for measurement. This is designed to quantify the impact of the Linux networking stack on the periodicity of the stream, thus motivating the need for PA-based pacing, even when the stream undergoes minimal processing.
- The pacer is replaced with a basic DPDK-based forwarder, sending the packets to the CR-PC and the NetFPGA board as fast as possible. This allows to assess whether bypassing the Linux kernel is sufficient to maintain a reliably low-jitter compatible with the CR-PC.

B. Results

As the used CR-PC gives no indication about the occupation of its 8-packets buffer, no direct quantitative experimental data can be extracted. However, the CR-PC consumes SMPTE 2022-6 packets to produce video, which gives a qualitative feedback: the presence, or absence, of video. Qualitative and quantitative results are presented hereafter.

1) Qualitative results: status of CR-PC: Table I summarizes the state of the video output of CR-PC. In the case where there is no intermediary node between CR-PG and CR-PC, no buffer overflow or starvation is observed, *i.e.,* the video output of CR-PC never stops. Similarly, the frequency-controlled pacer also generates a sufficiently regular stream, so that the video output does not stop.

When going through the Linux network stack, the stream is not sufficiently regular for consumption by the CR-PC, and buffer overflow or starvation are so frequent, that the video never locks, *i.e.,* the receiver never consumes a sufficient number of consecutive SMPTE 2022-6 packets to be able to generate valid video output.

In case (i) of the DPDK-based forwarder, and (ii) of the free-running pacer, video is output by the CR-PC, but interruptions occur, due to overflows or starvations. Qualitatively, the failures seem to happen sporadically in (i), and periodically in (ii). In order to better understand the nature of those failures, the Time Between Failures (TBF) is measured in both cases over twenty-four hours, and its Cumulative Distribution Function (CDF) is shown on figure 5.

In case (i), failures are effectively periodic as the CDF is close to a step function, *i.e.,* there is only one value of TBF. Periodic and repeated failures are explained by a mismatch between the period T^p of the output of the free-running pacer, and the period T^c of CR-PC. Because of that mismatch, the buffer occupation increases at a fixed frequency $\Delta f = \frac{1}{T^p} - \frac{1}{T^c}$. As a consequence, after a fixed amount of time proportional to the receive buffer capacity B_2 and the drift Δf, overflow will occur (or starvation if Δf is nonpositive). After that event, a video failure occurs, the CR-PC is reset, and the process repeats with the same drift Δf, hence, failure after the same duration. Consequently, drift-induced failures are periodic.

In case (ii), the CDF shows that the TBF values are spread across a wide range of possible values, *i.e.,* failures are sporadic. This is due to the latency spikes experienced by the DPDK-based forwarding process, triggering violations of property 1, thus starvation or overflow.

2) Quantitative results: timed captures with the NetFPGA board: For each setup, and corresponding output packet time sequence t^p, and for the i-th transmitted packet, the Packet Inter-arrival Times (PIT), defined as $t^p_{i+1} - t^p_i$, is measured with the NetFPGA board. The statistical deviation of the PIT from the period value T^p is an indicator of how often property 1is violated. Figure 6 shows that both versions of the PA-based algorithm produce PIT values which present a step-function-like CDF, *i.e.,* a very reduced peak-period jitter.

Surprisingly, according to the experimental data, the CR-PG does not actually generate a perfectly constant-rate stream, as the PIT distribution is observably different from a Dirac.

The DPDK-based and Linux-based forwarders are *work-conserving* setups, *i.e.,* they do not artificially delay the incoming packets from the CR-PG. As such, the difference between the measured PIT distribution at the output of both of those setups, and the PIT distribution for the setup with no intermediary, quantifies the distortion introduced by the operating system or the hardware itself. The high-spread observed in both cases in figure 6 confirms that, without hardware assistance, accurate pacing is not feasible.

The PIT distributions for algorithms 1 and 2 are indistinguishable on figure 6, as the main difference between them is

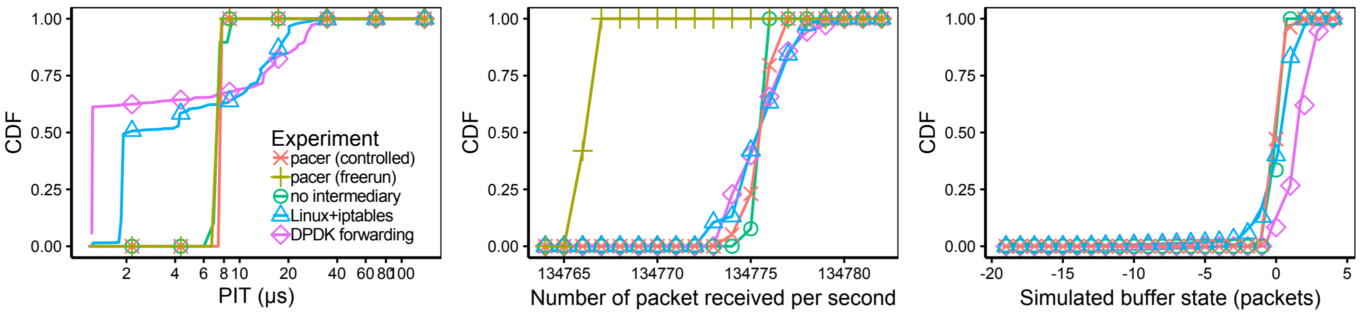


Figure 6: Empirical distribution of Packet Inter-arrival Times (left), of packets received during one second (middle), and of a simulated representation of the CR-PC ingress buffer (up to an additive constant).

their mean value, *i.e.,* T^p. The CDF of the number of packets received in one-second samples is therefore plotted as well on figure 6. This shows that the work-conserving setups and frequency-controlled pacer maintain the target T^p, whereas the free-running version introduces a frequency-drift, which leads to the observed periodic video failures.

The PIT distribution gives fine-grained information quantifying how often the peak period jitter J^p is too high. In order to construct fine-grained information quantifying how often the ALT-jitter is too high, for each i, and for each setup, the value $i - f^c t_i^p$ is computed. This value is the simulated state of a virtual receiver upon reception of the i-th packet of the studied stream, if the receiver starts consuming packets immediately after receiving the first one (*i.e.,* $i_0 = 0$ and $t_0 = 0$, see section II). Figure 6 also shows the empirical CDF of this value.

The analysis of figure 6 allows to conclude as to why no video was observed in the Linux forwarding case: the graph shows a significant fraction of values far from each other hinting at frequent, large variations of the buffer state, necessarily leading to overflows or starvation. The frequency-controlled pacer leads to a contained buffer occupancy, between -1 and 2, hence $J_{ALT}(t^p) \leq 2$, validating that the built pacer generates a $(4, f^c)$-paced stream as per property 2.

C. *Experimental qualification of* F

The bound on the ALT jitter from section V-D2 depends on the behaviour of function F, summarised as two constants $A_W(F)$ and $C_W(F)$. As they only depend on the values of F at multiples of W, *i.e.,* values $F(Wk)$ for all integers k, a sample of values $f_{W,k} = F(Wk)$ is experimentally acquired by using the frequency-controlled pacer with parameters $N_W = N_W^0 = 1$ and $W = W^0 \approx 100\,\mu\text{s}$. From this base sample, the constants $A_W(F)$ and $C_W(F)$ are estimated for all values of W which are multiple of W^0 and all N_W. Figure 7 illustrate these estimates by showing, the sensitivity of these constants both to varying W for a fixed N_W, and to N_W for a fixed W.

This figure gives a conclusive argument for increasing N_W instead of W. A_W — which quantifies the deviation from a perfect CR stream due to the instantaneous frequency-error of the pacer — can be observed to be insensitive to increasing only W. That result confirms the intuition motivating N_W-regularisation: solely increasing W improves the τ_{cur} estimate, but at the cost of less frequent τ_{cur} updates (as they happen every W). Globally, A_W is therefore not improved by an increased W.

Figure 7 also shows that, fixing W and increasing N_W considerably reduces A_W. That result also confirms the intuition: increasing N_W yields a more accurate estimate τ_{cur} but does not change how often τ_{cur} is update.

Figure 7 also shows that C_W is similarly sensitive to increasing W or N_W. This is interpreted as an unexpected burst of t^α cycles (see sections VI-B) being smoothed by averaging over a longer duration (which is the consequence of both increasing W and N_W). That smoothening improves how well $\tau \times F(Wk)$ tracks the number of t^e-cycles, *i.e.,* Wk.

Finally, $A_W + C_W$, *i.e.,* the bound established in section V, decreases faster when increasing N_W, than it does when increasing W. That gives a conclusive argument for increasing N_W instead of W.

The measured values of $A_W + C_W$ remain higher than the observed ALT jitter for $W = 1\,\text{s}$ and $N_W = 1$. Especially if the ALT jitter were equal to the value of $A_W + C_W$ predicted on figure 7, the used CR-consumer would necessarily have starved with an 8-packets buffer. That shows $A_W + C_W + T^e$ to be a fairly conservative bound on the ALT jitter.

D. *Operational perspective*

The experimental results show that (B_2, f^c)-pacing is feasible with the frequency-controlled pacing approach. Also, property 5, and the method used to experimentally evaluate F, yields a practical policy for choosing N_W and W: if a stream is to be paced with a target buffer size B_2, repeating the experiments performed in section VII-C allows to determine which values of N_W and W need to be chosen so that $A_W + C_W + 1 \leq \frac{B_2}{f^c}$.

As per property 5, the stream will then be guaranteed to be (B_2, f^c)-paced.

VIII. Discussion

In a professional broadcasting environment, the loss of more than a single video-frame (due to a single packet loss) per day is unacceptable [33]. In the absence of high-capacity buffers, jitter and drift must be contained. Considering a system with a receive buffer of reduced capacity (in the order of 10 packets), and receiving a packet stream with a frequency of 134775.22 packets per second, the impact of the approach proposed in this paper is discussed hereafter.

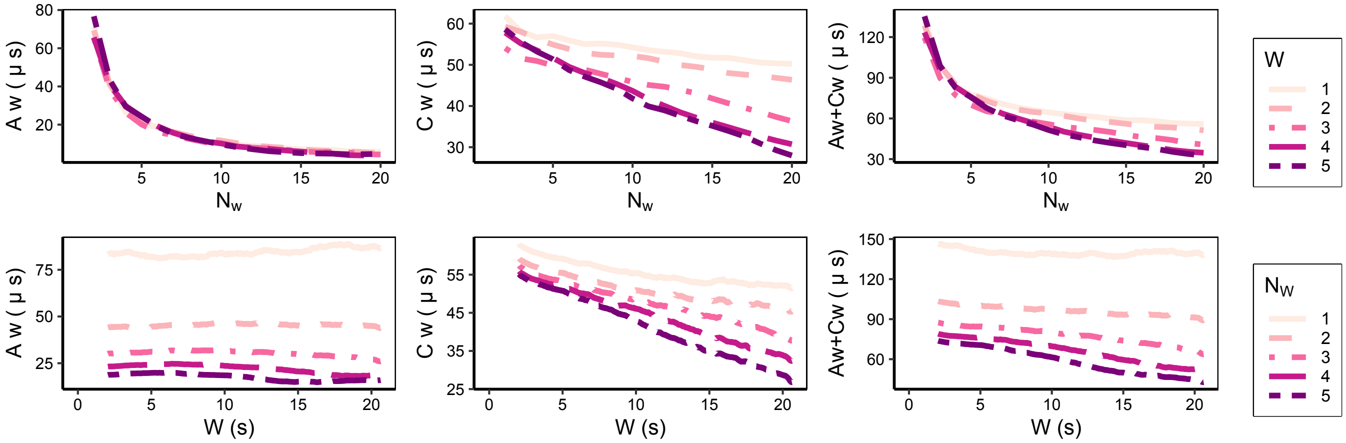


Figure 7: Experimental estimation of $A_W(F)$ (maximal error when F is used to measure a time interval of length W), $C_W(F)$ (maximal error when τF is used to track the current time modulo W) and $A_W(F) + C_W(F)$, by varying N_W (regularisation parameter used in the N_W-regularised version of F) and W (sampling period of F).

A. Practical impact of jitter reduction

The model from section II shows that a consistently small jitter (in the order of a few dozen microseconds) is necessary to enable lossless reception and timely consumption at a CR-PC with small buffers. From an operational perspective, if the experienced jitter is too high, the receiving CR-PC needs to provision a larger buffer. This is not necessarily practically possible, for example, IP-to-SDI gateway devices are usually implemented on FPGAs, with limited and non-evolutive buffering capacity.

Because the proposed pacing system uses commodity servers and general-purpose networking hardware, it is flexible enough to absorb any jitter, and to adapt to any CR-PC.

Finally, the proposed pacing algorithms assume that, whenever a `send` operation is enqueued, the corresponding packet must be available to the pacer. That condition requires that the pacer stores a sufficient number of packets (in the B_1 buffer of figure 1), before starting the execution of the chosen pacing algorithm. That initial buffering necessarily introduces some unavoidable delay, depending on the jitter of the input stream. However, this buffering is not specific to pacing; even in the absence thereof, before starting packet consumption, the CR-PC would need buffer the same amount of packets — and, therefore, add the same delay — as would the pacer.

B. Quantitative impact of drift compensation

The SMPTE stream used in the experimental evaluation has a nominal PIT of $\left(\frac{4497*30\,\mathrm{Hz}}{1.001}\right)^{-1} \approx 7419.761\,32\,\mathrm{ns}$. Introducing, for example, a $0.1\,\mathrm{ns}$ error results in a PIT of $7419.76132 - 0.1 = 7419.661\,32\,\mathrm{ns}$, hence an effective packet rate of $\frac{1}{7419.661\,32\,\mathrm{ns}} = 134777.04$ packets per second. That is, a drift of $134777.04 - 134775.22 \approx 1.82$ packets per second.

In the context of professional broadcast, SMPTE streams are unidirectional (allowing, *e.g.,* multicast transmission), and, therefore, no explicit flow control is performed. In that context, if the CR-PC receives 1.82 packets more than consumed every second, a 10 packets receive buffer will overflow after at most $10/1.82 \approx 5.5\,\mathrm{s}$, which is far from the expected reliability of a single frame loss per day. Consequently, achieving pacing for professional broadcast requires the frequency-controlled approach.

IX. Conclusion

Systems relying on constant-rate packet consumption, and using receivers with small buffers, require transmitted packets to be regularly paced. This paper shows that such high-accuracy packet pacing can be implemented in software, through designs imposing minimal hardware requirements, captured in the notion of Pacing-Assistant.

Data processing workflows requiring CR packet streams (such as media processing for broadcasting), and which are traditionally implemented using dedicated hardware, can, using the approach proposed in this paper, be replaced by *software running on commodity hardware*, and still benefit from a guaranteed *sufficiently regular stream*. Two pacing algorithms were presented and analysed. While the free-running algorithm is only applicable when the Pacing-Assistant and the packet consumer internal clocks are synchronised, the frequency-controlled algorithm has a broader-scope, at the expense of increased operational complexity, arising from the construction and parametrisation of a frequency-controller (as described in section VII-D).

An *implementation* of the approach proposed in this paper has been tested in *real conditions and hardware*, and the viability of software-based packet pacers has been experimentally demonstrated for media-production streams.

From among the conclusions of this paper, the experiments and analysis presented demonstrate, that the proposed approach is able to bring additional functionalities (pacing regularity and minimisation of buffer occupation), which are not available through standard mechanisms as provided in general-purpose hardware.

References

[1] R. Benice and A. Frey, "An Analysis of Retransmission Systems," vol. 12, no. 4, pp. 135–145. [Online]. Available: http://ieeexplore.ieee.org/document/1088975/

[2] J. Postel, “Transmission Control Protocol.” [Online]. Available: https://www.rfc-editor.org/info/rfc0793

[3] “ST 2022-6:2012 - SMPTE Standard - Transport of High Bit Rate Media Signals over IP Networks (HBRMT),” pp. 1–16.

[4] *ST 2059-2:2015 : SMPTE Profile for Use of IEEE-1588 Precision Time Protocol in Professional Broadcast Applications*. The Society of Motion Picture and Television Engineers.

[5] “ST 2110-21:2017 - SMPTE Standard - Professional Media Over Managed IP Networks: Traffic Shaping and Delivery Timing for Video,” pp. 1–17.

[6] *ST 292-1:2011 : 1.5 Gb/s Signal/Data Serial Interface*. The Society of Motion Picture and Television Engineers. [Online]. Available: http://standards.smpte.org/lookup/doi/10.5594/S9781614824299

[7] A. Aggarwal, S. Savage, and T. Anderson, “Understanding the performance of TCP pacing,” in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, vol. 3. IEEE, pp. 1157–1165. [Online]. Available: http://ieeexplore.ieee.org/document/832483/

[8] M. Ghobadi and Y. Ganjali, “TCP Pacing in Data Center Networks,” in *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*. IEEE, pp. 25–32. [Online]. Available: http://ieeexplore.ieee.org/document/6627732/

[9] L. Zhang, S. Shenker, and D. D. Clark, “Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic,” in *Proceedings of ACM Sigcomm*, vol. 91, pp. 133–147.

[10] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and Van Jacobson, “BBR: Congestion-based congestion control,” vol. 60, no. 2, pp. 58–66. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3042068.3009824

[11] Y. Cai, T. Wolf, and W. Gong, “Delaying Transmissions in Data Communication Networks to Improve Transport-Layer Performance,” vol. 29, no. 5, pp. 916–927. [Online]. Available: http://ieeexplore.ieee.org/document/5753556/

[12] V. Sivaraman, H. Elgindy, D. Moreland, and D. Ostry, “Packet Pacing in Small Buffer Optical Packet Switched Networks,” vol. 17, no. 4, pp. 1066–1079. [Online]. Available: http://ieeexplore.ieee.org/document/4895292/

[13] P. Thiran, J.-Y. Le Boudec, and F. Worm, “Network calculus applied to optimal multimedia smoothing,” in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3. IEEE, pp. 1474–1483. [Online]. Available: http://ieeexplore.ieee.org/document/916643/

[14] J. D. Salehi, Z.-L. Zhang, J. F. Kurose, and D. Towsley, “Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing,” vol. 24, no. 1, pp. 222–231. [Online]. Available: http://portal.acm.org/citation.cfm?doid=233008.233047

[15] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, “Carousel: Scalable Traffic Shaping at End Hosts,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM ’17*. ACM Press, pp. 404–417. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3098822.3098852

[16] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa, “Design and evaluation of precise software pacing mechanisms for fast long-distance networks.”

[17] “IEEE Standard for Ethernet,” pp. 1–4017.

[18] R. Takano, T. Kudoh, Y. Kodama, and F. Okazaki, “High-resolution timer-based packet pacing mechanism on the linux operating system,” vol. 94, no. 8, pp. 2199–2207.

[19] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, “MoonGen: A Scriptable High-Speed Packet Generator,” in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference - IMC ’15*. ACM Press, pp. 275–287. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2815675.2815692

[20] D. Intel, “Data plane development kit.”

[21] N. A. Lynch and M. R. Tuttle, “An introduction to input/output automata,” vol. 2, no. 3, pp. 219–246.

[22] A. Fog, “Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs,” vol. 93, p. 110.

[23] S. Müller, “CPU Time Jitter Based Non-Physical True Random Number Generator,” in *Linux Symposium*. Citeseer, p. 23.

[24] F. Cerqueira and B. Brandenburg, “A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS RT,” in *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. SYSGO AG, pp. 19–29.

[25] J. H. Brown and B. Martin, “How fast is fast enough? Choosing between Xenomai and Linux for real-time applications,” in *Proc. of the 12th Real-Time Linux Workshop (RTLWS’12)*, pp. 1–17.

[26] B. Delgado and K. L. Karavanic, “Performance implications of System Management Mode,” in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, pp. 163–173. [Online]. Available: http://ieeexplore.ieee.org/document/6704682/

[27] A. Toussaint, M. Hawari, and T. Clausen, “Chasing Linux Jitter Sources for Uncompressed Video,” in *2018 14th International Conference on Network and Service Management (CNSM)*, pp. 395–401.

[28] J.-L. Ferrant, M. Gilson, S. Jobert, M. Mayer, M. Ouellette, L. Montini, S. Rodrigues, and S. Ruffini, “Synchronous ethernet: A method to transport synchronization,” vol. 46, no. 9, pp. 126–134.

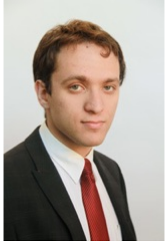
[29] G. ITU, “8261-Timing and synchronization aspects in packet networks.”

[30] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, “NetFPGA SUME: Toward 100 Gbps as Research Commodity,” vol. 34, no. 5, pp. 32–41. [Online]. Available: http://ieeexplore.ieee.org/document/6866035/

[31] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. Mckeown, N. Feamster, B. Felderman, M. Blott, A. Moore, and P. Owezarski, “OSNT: Open source network tester,” vol. 28, no. 5, pp. 6–12. [Online]. Available: http://ieeexplore.ieee.org/document/6915433/

[32] “ANY-FREQUENCY PRECISION CLOCK MULTIPLIER/ JITTER ATTENUATOR.” [Online]. Available: https://www.silabs.com/documents/public/data-sheets/Si5324.pdf

[33] B. J. Olsson, “IP QoS Objectives For Broadcast Services,” in *NAB 2014 Conference Contribution (May Be Retrieved at https://www. Researchgate. Net/Publication/261476525_IP_QoS_Object Ives_For_Broadcast_Services)*.



Mohammed Hawari received the Diplôme d’Ingénieur from École Polytechnique (Palaiseau, France) in 2015, and the Masters degree in Advanced Communication Networks (École polytechnique and École nationale supérieure des télécommunications) in 2016. He is currently undertaking an industrial PhD under joint supervision of André Surcouf (Cisco Systems) and Thomas Clausen (École polytechnique). His research interests include high-performance and deterministic networking, real-time systems, cloud computing, and high-precision network instrumentation.

Juan Antonio Cordero Fuertes is a research assistant professor at École polytechnique. He graduated in Mathematics (“Licenciatura”, M.Sc.) and Telecommunication Engineering (B.Sc.+M.Sc., “Ingeniería Superior”) at the Universidad Politécnica de Cataluña (Technical University of Catalonia, UPC, Spain) in 2006 and 2007, respectively. He got his Ph.D. at École polytechnique in 2011, with a dissertation on the optimization of link-state routing protocols for operation in MANETs and compound (wired/wireless) Autonomous Systems. As part of his PhD, he participated in the development of OSPF for MANETs. Before joining faculty at École Polytechnique, Juan Antonio held postdoctoral research positions at the Université catholique de Louvain (UCL, Belgium) and at the Hong Kong Polytechnic University (Hong Kong SAR, People’s Republic of China). Juan Antonio’s scientific interests include wireless mesh and mobile ad hoc networking, routing protocols and information dissemination algorithms, integration of wired and wireless networks, Internet measurements and future Internet architectures.

Thomas Clausen is a graduate of Aalborg University, Denmark (M.Sc., PhD – civilingeniør, cand.polyt), and a Senior Member of the IEEE. Thomas has, since 2004 been on faculty at École Polytechnique, France's leading technical and scientific university, where he holds the Cisco-endowed "Internet of Everything" academic chaire.
At École Polytechnique, Thomas leads the computer networking research group. He has developed, and coordinates, the computer networking curriculum, and co-coordinates the Masters program in "Advanced Communication Networks" (ACN). He has published more than 80 peer-reviewed academic publications (which have attracted more than 12000 citations) and has authored and edited 24 IETF Standards, has consulted for the development of IEEE 802.11s, and has contributed the routing portions of the recently ratified ITU-T G.9903 standard for G3-PLC networks – upon which, e.g., the current SmartGrid & ConnectedEnergy initiatives are built. He serves on the scientific council of ThinkSmartGrids (formerly: SmartGridsFrance).