Bridging Time-Sensitive Networking and Containerization: Challenges and Strategies

Rodrigo Martins*†, Duarte Raposo*, Rui Eduardo Lopes*†, Pedro Teixeira*†, Susana Sargento*†
*Instituto de Telecomunicações, 3810-193 Aveiro, Portugal
{rodrigomartins, dmgraposo}@av.it.pt
†DETI, University of Aveiro, 3810-193 Aveiro, Portugal
{ruieduardo.fa.lopes, pedro.teix, susana}@ua.pt

***Abstract*—In recent years, the adoption of containerization solutions has grown significantly. Simultaneously, there has been a rise in applications with stricter latency and jitter requirements, such as tactile internet, autonomous vehicles, and immersive AR/VR. Time-Sensitive Networking (TSN) emerges as a promising solution for these latency-critical applications, offering tools specifically designed to manage such conditions and support mixed-criticality scenarios involving both hard and soft deadlines. This paper explores techniques and solutions for running latency-sensitive applications in containerized environments. It focuses on how high-priority traffic can coexist with best-effort traffic using TSN mechanisms, and how containerized applications can leverage hardware-based techniques to achieve precise transmission timing. The results demonstrate that, with TSN mechanisms alone on the network side, high-priority traffic can coexist with interfering traffic while maintaining accurate transmission scheduling and the target bandwidth. However, hardware-based techniques such as *Launchtime* proved crucial in achieving the most precise transmission schedule—evident in a 42-fold reduction in standard deviation. This improvement is also reflected in the P99 metric, where 99% of transmission delays remained below 0.501 ms. Finally, the paper presents mitigation strategies to address challenges introduced by container isolation, which can cause unforeseen conflicts between transmission schedules and add latency to communications.**

***Index Terms*—Time-Sensitive Networks, Containerization, Credit-Based Shaper, Launchtime, Deterministic Networking**

I. Introduction

Real-time communication is becoming increasingly critical in various scenarios, e.g., remote robotic surgery, immersive AR/VR applications, and cooperative emergency maneuvers for collision avoidance in autonomous vehicles. However, by their very nature, those applications will require an environment that not only guarantees the timing behavior of critical traffic, but that also provides temporal isolation from non-critical communication. Moreover, timing is not the only vital aspect to be considered: reliability, fault tolerance, and security are other crucial requirements for these applications [1].

Although Ethernet still plays an important role in industrial communications, it is not designed to offer hard real-time guarantees, which become increasingly important for avionics [2], automotive [3], and train industries [4] as well as in industrial automation. Recognizing the lack of real-time guarantees in Ethernet, the IEEE 802.1 Working Group established the Time Sensitive Networking (TSN) task group in 2012 to research, develop, and standardize real-time capabilities for Ethernet networks.

Regardless of the underlying communication technology, applications that aim to benefit from real-time communication should not blindly rely on the network to provide this real-time environment. Instead, they should implement mechanisms to ensure that the data they produce or consume is as reliable and timely as possible. This creates a symbiotic relationship between application or service design and the underlying network: the network guarantees bandwidth and timing requirements, while the applications ensure predictable timing behavior and obtain time information from reliable sources. In parallel with this evolution, another trend that is becoming increasingly widespread is the adoption of containerized and virtualized deployments. However, most efforts in this area have focused on resource-sharing optimizations and orchestration efficiency [5], leaving the networking aspect insufficiently studied to guarantee the determinism required for real-time communications.

This paper aims to fill this gap by focusing on the containerized approach of services and applications in time-sensitive networks. It investigates how real-time applications can operate in such environments by leveraging hardware-based features and a TSN-enabled network to enhance real-time characteristics end-to-end. Therefore, this paper begins with section II, by examining tools and techniques for enhancing real-time applications, focusing on their integration with container technologies and the challenges involved. Then, section III explores the integration of TSN standards in containerized environments, presenting an experimental scenario that highlights deployment challenges. Based on the insights gained from this practical example, an experimental setup is proposed, along with a description of the tools and configurations used, all in Section IV. The results of these experiments are presented in section V and show an increased application performance with the combination of TSN standards and other techniques, showing that these benefits can also be achieved in a containerized approach. Finally, section VI provides the conclusion and outlines future work.

978-3-903176-75-1

II. Timing Challenges in Applications

A real-time application may use hardware-based features or techniques to enhance its operation. However, in a containerized environment, these techniques/features might not be available or have degraded performance, either because of the abstraction layer added by containers or insufficient resources for the application.

An example of these features is the *LaunchTime* technology, which is present in some Intel Network Interface Card (NIC)s. This feature allows the Ethernet controller to pre-fetch Ethernet frames from the system memory to the transmission buffer inside the Ethernet Medium Access Control (MAC) controller ahead of its specified transmission time [6], therefore providing time-deterministic frame transmission, which will be essential for the experimental study described later in section IV. Consequently, this feature allows the applications to avoid the variable delay introduced by network congestion at the NIC level (variable position of frames on the outgoing queue). However, in containers, this feature is obfuscated as it depends on the capability to configure the physical interface which is not possible by default, because of the network abstraction introduced by containerization, which shadows the actual real interfaces in a host.

Accurate frame timestamping is essential for real-time communication. Software-based clocks are unreliable due to variable delays incurred while a frame traverses the Operative System (OS) network stack and from NIC congestion. This invisible and variable latency impacts both frame transmission and reception.

Hardware-level timestamping at the NIC solves this by providing a single, jitter-free point of reference. This offers a more reliable view of the communication and significantly increases reliability. For containerized applications, this functionality is contingent on the host network system being exposed and the NIC supporting hardware timestamping.

Following this line of thought, one aspect that remains constant in both the transmission and the reception of frames is the traversal of the OS network stack. This traversal introduces variable delay, which cannot be controlled as it depends on the Central Processing Unit (CPU) load. Removing such variability is advantageous, since it is not done at an application-by-application basis, but rather at the whole OS level, which in turn will also ensure that all applications can benefit from it without locking them to specific hardware considerations. [7]

III. TSN in Containerized Environments

This section highlights existing challenges with timekeeping, conflicting transmission schedules, and the Docker network itself when deploying real-time applications in a containerized TSN enabled network. It also presents a practical example involving the deployment of one or more real-time applications in containers, followed by an analysis of the resulting problems and potential solutions. The conclusion taken from the example culminates in the creation of the test scenarios and the choice of the associated tools for the following section.

A. Timekeeping

Most TSN standards require precise time synchronization between network listeners and talkers. To achieve this, the host system clock must be accurately synchronized with all other hosts in the network. In a TSN context, the Precision Time Protocol (PTP) protocol is used for synchronization-specifically, the Generic Precision Time Protocol (gPTP) profile. In a containerized environment, this synchronization process remains effective. Since containers operate without a hypervisor, they directly share the host system clock (without the ability to modify it), ensuring that all containers on the same host derive their time from a common, accurately synchronized source. Finally, if a containerized application requires access to a time source other than the system clock, it must be granted higher privileges than the default configuration allows.

B. Transmission Schedules

Assuming the correct time synchronization of the containers, the applications now must transmit their Scheduled Traffic (ST) according to a precise schedule, or run the risk of having their frames dropped, competing with other ST or Best Effort (BE) traffic. Regarding Stream Reservation (SR) traffic, this is shaped to allow fairness between other flows, which means spacing two consecutive frames of the same flow according to a minimum time interval, and if flows do not consist of small samples, they are segmented according to Stream Reservation Protocol (SRP) specifications. [8]

However, these tight schedules or shaping might not always be respected in containerized environments, as a single node might have many applications competing for the same resources, or even if many applications have incompatible transmission schedules or flows, which is further exacerbated by the isolation provided by containers. Considering that the applications independently manage the resources, the granularity of this process is increased. However, if multiple applications concurrently try to manage, for example the time-aware shaper (TAS)s Gate Control List (GCL)s, this will inevitably create conflicts where none of the applications can communicate. A possible solution to this problem is to create a separate entity to manage the different flows of applications. By offloading the intelligent management of the ST and SR flows to a central entity that knows all of the application requirements, conflicts will be more easily avoided.

C. Docker Networks

To understand how container-based, real-time applications behave, we examined a practical example of three container architectures, shown in figure 1. The first approach, figure 1a, places both real-time and BE applications on the same default Docker Bridge network. This isolates containers from the host network, preventing TSN applications from applying necessary Traffic Control (TC) rules to the Ethernet port. The result is conflicting traffic between the applications. To solve this, a second approach separates the containers into different networks, as shown in figure 1b. By not being isolated from the host network, TSN applications can access the host's

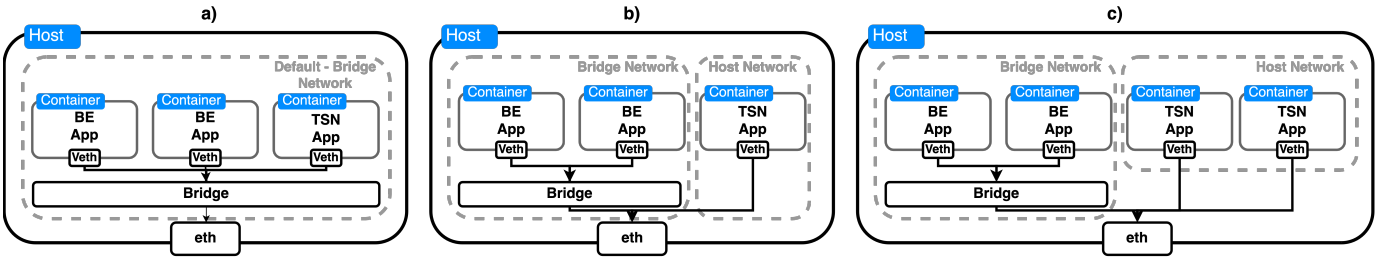

Fig. 1: a) Deployment example in Bridged network b) Deployment example in mixed networks c) Deployment example with container conflict

NIC and perform the required configurations. However, this method exposes the application to the host network, creating potential security risks and leaving open questions regarding other network types(i) and scalability(ii).

(i) Network types: Docker supports several network drivers, including bridge, host, overlay, IPVLAN, and MACVLAN, each introducing varying levels of communication delay. To measure this impact, a test was conducted where a container pinged an external device. As shown in figure 2, the results indicate that delay increases with driver complexity, which can negatively affect the timeliness of real-time communications.

(ii) Scalability: Adding more TSN containers affects the effectiveness of previously optimal configurations. This is influenced by the need to consolidate application resource requirements and to configure traffic scheduling at the host level, particularly when multiple services run natively on the host operating system. Since TSN configurations are generally static, it is typically assumed that each host runs only a single application, making multi-application scenarios uncommon. To support such setups, an intelligent scheduling management mechanism is needed. For example, it could prevent containers from being deployed on nodes lacking sufficient resources and instead redirect them to more suitable alternatives. To better explain this aspect, figure 1c represents a situation with two containers that require the configuration of the host NIC. Because of the inherent isolation offered by containers, two problems can happen: (i) the ST of each application is incompatible; (ii) there are not enough physical resources for both applications to operate normally. In either situation, the applications could not operate on the same node and the performance of these services will degrade.

IV. Experimental Setup

Building on the identified network challenges from the previous section, this section focuses on testing traffic shaping mechanisms like TAS and credit-based shaper (CBS) using the host network mode (figure 1b). We detail the experimental setup used to compare these two scenarios with a native OS deployment. The section concludes with a practical example of deploying real-time applications in containers and an analysis of the observed challenges and mitigation strategies in section V.

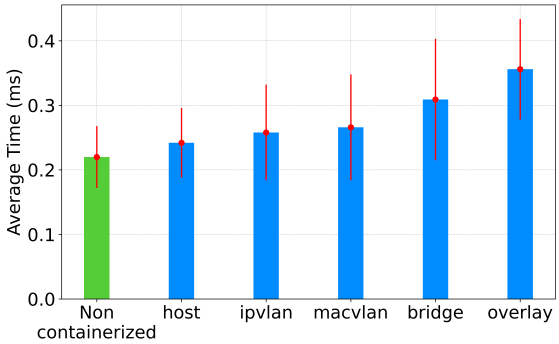

Fig. 2: Overview of Docker network types and the latency they introduce

A. Testing Scenarios

Considering the lessons learned from the previous practical example and the goal of demonstrating how TSN and hardware-based techniques affect the timeliness behavior of both regular and containerized networks, several test scenarios were designed. Among the many available TSN tools, the focus is placed on two: TAS and CBS. This choice is based on two factors: (i) both shapers are relatively straightforward to implement in Linux when compared to other standards; and (ii) their effects are more easily observed at runtime.

Therefore, to specifically test the TAS and the CBS, respectively, two scenarios were envisioned: (i) a sensor (producer) transmits constant-size data at regular intervals to a receiver (consumer). This data has tight latency requirements, as the receiver represents a hard real-time application; (ii) a video camera (producer) transmits video frames at a constant bit rate and packet size to a receiver (consumer). The receiver can tolerate some missed deadlines or packet losses, as it represents a soft real-time application. Moreover, in both scenarios, the high-priority traffic must coexist with concurrent generic BE traffic. The resulting setup is depicted in figure 3.

B. Tools

We utilized several open-source tools to implement the desired scenarios in a Linux environment. First, *ethtool*[1] was

[1]https://linux.die.net/man/8/ethtool

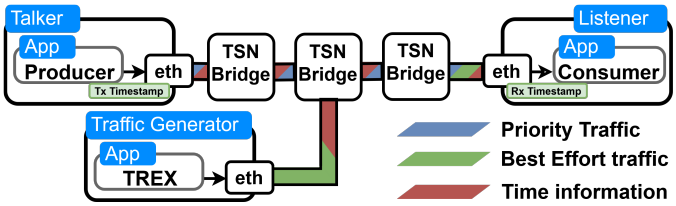

Fig. 3: Setup of talker and listener through three TSN bridges

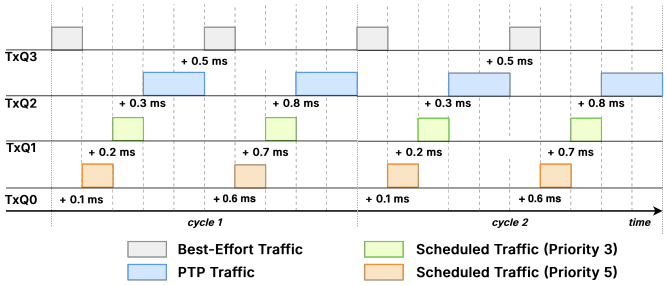

Fig. 4: Scenario 1: TAS Transmission Schedule

instrumental in querying our network interface cards to verify hardware support for PTP and other time-related features. To handle time synchronization, we deployed LinuxPTP[2], an Institute of Electrical and Electronics Engineers (IEEE) 1588-compliant PTP implementation that supports various profiles, including the gPTP profile essential for TSN, and other sub-tools to perform precise clock synchronization.

Considering hardware-specific technologies, the I210 NIC uses the `igb` driver, which offers the previously mentioned *LaunchTime* feature.

For realistic traffic generation, we used TRex[3], a fast, open-source tool leveraging Data Plane Development Kit (DPDK). This allowed us to reproduce the exact traffic profile of a real-world capture from the Aveiro smart city[4].

C. Setup

The setup, as shown in figure 3, comprises two Accelerated Processing Unit (APU)s connected via three TSN capable switches, with a third APU generating BEs traffic to the middle switch. The APUs form a PTP domain where the talker node serves as the Grandmaster (GM). All APUs run Debian 11 with the Linux kernel $5.15.39 - rt42$ and the *PREEMPT_RT* patch. *phc2sys* is used on each APU to synchronize the PTP Hardware Clock (PHC) with the system clock (GM) or vice versa (slaves).

Moreover, three configurations are tested: (i) the shapers and the *LaunchTime* are disabled, to provide the baseline performance for further comparisons; (ii) only the associated shaper is enabled, to emphasize the usefulness of the *LaunchTime*; (iii) both the shaper and the *LaunchTime* are enabled, which is expected to provide the best performance.

The transmission schedule of the TAS is configured as it appears in figure 4. In this schedule, the duration of the transmission cycle is 1 ms, and four types of traffic will be transmitted, one in a different Transmission Queue (TxQ). The BE traffic is mapped to TxQ 3, the PTP frames are mapped to TxQ 2, the priority 3 traffic is mapped to TxQ 1, and finally the priority 5 traffic is mapped to TxQ 0. Each type of traffic will have two transmission windows in 1 cycle. The BE, priority 3 and 5 traffic transmission windows will last for 0.1 ms, and the PTP frames transmission windows will last for 0.2 ms. The cycle will start at a specified base time, which will be defined by adding 2 minutes after the configurations are done.

[2]https://linuxptp.sourceforge.net/
[3]https://trex-tgn.cisco.com/
[4]https://new.aveiro-living-lab.it.pt/realtime

The CBS values are calculated taking into account the guidelines provided by the IEEE 802.1Q [9] and the characteristics of the *producer* application used. The *producer* application sends 8000 packets/s, each with size of 1250 Bytes, and therefore, maximum frame size of equal value, in a port with a maximum transmission rate of 1000 Mb/s. Therefore, the following variables can be defined, which will be used in the following equations: nPkts $= 8000$, pktSize $=$ 1250 B, portTX $=$ 1000 Mb, maxFrameSize $=$ 1250 B. The maximum frame size of 1500 B, i.e. maxIntSize $=$ 1500 B, represents the size of any burst of traffic that can delay the transmission of a frame that is available for transmission of the protected traffic class.

This *idleSlope* is given by: idleSlope $=$ nPkts $\times$ pktSize $\times$ $8 =$ 80Mb/s and represents the rate of credit increase, in bits per second (i.e., while the protected traffic class is not transmitting) and cannot exceed the *portTX*. This value of 80 Mb/s which, in a Gigabit Ethernet port, represents around 8% of reserved bandwidth for the protected traffic class. The *sendSlope* is given by is given by: sendSlope $=$ (idleSlope $-$ portTX) $=$ $-$920Mb/s and represents the rate of credit decrease, in bits per second (i.e., while the protected traffic class is transmitting). The *hicredit* is the maximum value that can be accumulated in the *credit* parameter and is determined by the worst case interfering traffic: hiCredit $=$ maxIntSize $\times$ $\left(\frac{\text{idleSlope}}{\text{portTX}}\right) = 960$. Finally, the *loCredit* is the minimum value that can be accumulated in the *credit* parameter, and is given by: loCredit $=$ maxFrameSize $\times$ $\left(\frac{\text{sendSlope}}{\text{portTX}}\right) = -9\,200$.

V. Evaluation of Test Scenarios

This section presents the results of the two scenarios tested, organized by the associated shaper used.

A. Scenario 1: TAS

The results obtained in this testing case are presented in table I and figures 5 and 6. The results of configuration 1 can be seen in figures 5a and 6a. Based on the histogram distribution, we observe that, without any form of traffic shaping, the periodic traffic is not received periodically: even though the packets are being sent anytime within their respective time windows, the packets are not leaving the NIC within that time window, or even at regular intervals, and the best-effort traffic present in the outgoing queue is delaying the priority traffic. In the test without containerization, most values are between 0.4

and 0.6 ms with the majority concentrated around the 0.5 ms mark, which is reflected in a low standard deviation value. When compared with the containerized alternative, the values are still centered around 0.5 ms; however, they are much more dispersed, which is reflected in the higher standard deviation and P99, which jumped from 0.039 to 0.287 ms (7.4x greater) and 0.589 to 1.499 ms (2.5x greater), respectively, for priority 3 for example. Regarding the packet loss, this also has a massive increase between approaches, e.g. priority 3, jumped from 0.01 to 7.77%, which represents a 777 times increase.

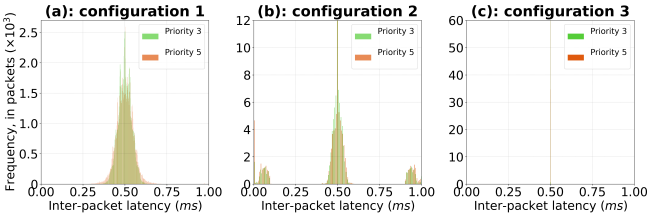

Fig. 5: Scenario 1, TAS in native deployment

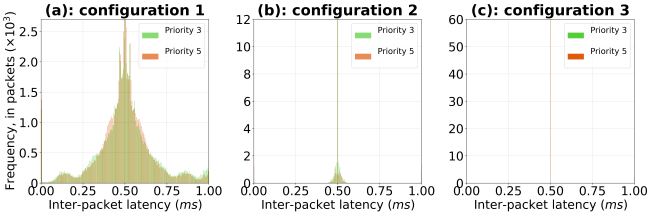

Fig. 6: Scenario 1, TAS in docker deployment

The results of configuration 2, in figure 5b and 6b, show major improvements which are apparent by the higher concentration of values at 0.5 ms with some outliers at the edges of the histogram. This reveals that the transmission interval is being respected and enforced. Most traffic is received between the 0.4 and 0.6 ms interval, which is expected because the only condition is for the packet to leave the TSN bridge anytime within the beginning and the end of the transmission window. Outliers occur when a packet misses its window and is retransmitted in the next, resulting in a 1 ms interval. In the same window, another packet will be sent, resulting in a difference inferior to 0.1 ms. The containerized approach actually performs better than the native one. Its histogram distribution is almost entirely concentrated at 0.5 ms, reflected in a standard deviation two to three times lower for priorities 3 and 5. While packet loss was slightly higher in the containerized approach, it remained under 1%.

Configuration	Container	Prio	Nº Pkts	Mean (ms)	Stdev (ms)	P99 (ms)	Max (ms)	Min (ms)	Loss (%)
1	No	3	99991	0.500	0.039	0.589	1.086	0.254	0.01
		5	99774	0.501	0.054	0.630	1.148	0.137	0.23
	Yes	3	92230	0.542	0.287	1.499	10.174	0.001	7.77
		5	94350	0.530	0.265	1.506	7.556	0.001	5.65
2	No	3	99964	0.500	0.210	1.00	5.463	0.001	0.04
		5	99921	0.500	0.269	1.034	7.500	0.001	0.08
	Yes	3	99756	0.500	0.101	1.00	2.500	0.001	0.24
		5	99867	0.500	0.081	1.000	1.494	0.001	0.13
3	No	3	99999	0.500	0.005	0.501	0.999	0.098	0.00
		5	100000	0.500	0.003	0.501	0.901	0.099	0.00
	Yes	3	99898	0.500	0.101	1.000	2.000	0.001	0.10
		5	99910	0.500	0.019	0.500	2.457	0.001	0.09

TABLE I: Scenario 1, results of TAS in native and container deployment

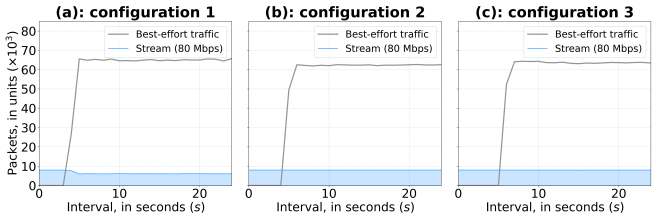

Fig. 7: Scenario 2, CBS in native deployment

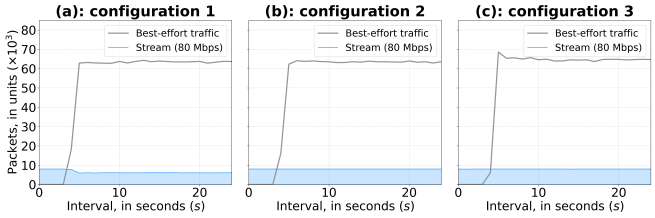

Fig. 8: Scenario 2, CBS in container deployment

The results of configuration 3, depicted in figures 5c and 6c, show near-perfect results for both approaches, which highlights the impact of *LaunchTime*. By enabling it, we obtain a single-column histogram exactly at 0.5 ms for both priorities, which is reflected in the lowest standard deviation of all tests (increase in determinism), and a P99 of 0.501 ms in the non-containerized approach. However, even though both approaches are nearly identical, the containerized approach produces slightly worse results. The standard deviation is higher for both priorities but remained under 0.2 ms; the P99 for priority 5 is 0.500 ms and for priority 3 it is 1 ms, which is double the one obtained in the other approach and differs from all other results obtained in this test. Finally, the packet loss that was absent in the non-containerized approach, is present in the containerized approach, even though it is very low, around 0.10% for both priorities.

B. Scenario 2: CBS

The results obtained in this testing case are presented in table II and figures 7 and 8. The results of configuration 1, depicted in figures 7a and 8a, show the talker application failing to reach 8000 packets/s. This configuration produces the worst results in all metrics except for the P99, which remained constant at 8000 packets/s. Both approaches produced similar results, with the containerized one achieving a slightly inferior throughput of 6418 packets/s, which represents a difference of only 0.14%. This configuration also has the highest standard

Configuration	Container	Mean (Pkts/s)	Median (Pkts/s)	Stdev (Pkts/s)	P99 (%)
1	No	6427	6061	756.090	8001
	Yes	6418	6049	766.065	8001
2	No	7998	8000	3.619	8001
	Yes	8000	8000	0.344	8001
3	No	7998	8000	9.235	8001
	Yes	7996	8000	13.914	8000

TABLE II: Scenario 2, results of CBS in native and container deployment

deviation of 756 packets/s for both approaches, which results from the variation between bursts; this emphasizes the need for some form of traffic shaping to help the coexistence of both traffic types.

The results of configuration 2, depicted in figures 7b and 8b, where CBS is active, clearly achieving the desired throughput. The shaping results in a smoother line in both figures, which is reflected in all metrics for both approaches. Clearly showing that, with a properly configured shaper like CBS, this application can have a much smoother and predictable operation under the same network conditions.

The final configuration results, depicted in figures 7c and 8c, are nearly identical to the previous test. The major difference for both approaches is in the standard deviation, which increases nearly 3 times for the non-containerized approach and 40 times for the containerized one. This configuration showcases that a tool like *LaunchTime* is not highly relevant to applications that only want to maintain a steady throughput, as compared to applications that require precise transmission of packets in time.

VI. Conclusions and Future Work

This paper investigates the challenges faced by real-time applications in modern containerized networks. It highlights key containerization aspects, outlines software development techniques to address these challenges, and discusses the role of TSN in introducing network determinism.

An experimental study was conducted to determine the impact of these techniques and TSN on application timeliness in both non-containerized and containerized environments. Although the containerized results were slightly worse, they confirmed that combining TSN with techniques like *LaunchTime* significantly improves communication determinism.

This work can be further improved in the following elements: (i) further improve the representation of real usage scenarios traffic, by having better representative samples of such scenarios which can be captured from longer periods or in key situations; (ii) this work can evolve into implementing a flow management entity like the one in [5] which implements a packet scheduler and adopts a kernel-bypassing approach to the transmission of packets which minimizes packet processing delays.

Acknowledgments

This work was supported by the European Union / Next Generation EU, through Programa de Recuperação e Resiliência (PRR) Project Nexus “Pacto de Inovação – Transição Verde e Digital para Transportes, Logística e Mobilidade” (53–C645112083-00000059), European Union / Next Generation EU, through Programa de Recuperação e Resiliência (PRR) [Project Nr. 11: New Space Portugal (02/C05-i01.01/2022.PC644936537-00000046)], European Union Programa FEDER, Operação n.º 14795 - COMPETE2030-FEDER-00929900.

References

[1] D. Cavalcanti, J. Perez-Ramirez, M. M. Rashid, J. Fang, M. Galeev, and K. B. Stanton, “Extending accurate time distribution and timeliness capabilities over the air to enable future wireless industrial automation systems,” *Proceedings of the IEEE*, vol. 107, no. 6, pp. 1132–1152, 2019.

[2] W. Steiner, P. Heise, and S. Schneele, “Recent ieee 802 developments and their relevance for the avionics industry,” in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, 2014, pp. 2A2–1–2A2–12.

[3] L. L. Bello, “Novel trends in automotive networks: A perspective on ethernet and the ieee audio video bridging,” in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–8.

[4] M. Pahlevan and R. Obermaisser, “Redundancy management for safety-critical applications with time sensitive networking,” in *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, 2018, pp. 1–7.

[5] A. Garbugli, L. Rosa, A. Bujari, and L. Foschini, “Kubernetsn: a deterministic overlay network for time-sensitive containerized environments,” in *ICC 2023 - IEEE International Conference on Communications*, 2023, pp. 1494–1499.

[6] “Intel ethernet controller i210 datasheet,” accessed= 2024-04-03. [Online]. Available: https://cdrdv2-public.intel.com/333016/333016%20-%20I210_Datasheet_v_3_7.pdf

[7] R. Rosmaninho, D. Raposo, P. Rito, and S. Sargento, “Time constraints on vehicular edge computing: A performance analysis,” in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–7.

[8] “Ieee standard for local and metropolitan area networks–virtual bridged local area networks amendment 14: Stream reservation protocol (srp),” *IEEE Std 802.1Qat-2010 (Revision of IEEE Std 802.1Q-2005)*, pp. 1–119, 2010.

[9] “IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks,” *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, pp. 1–1993, 2018.