A Framework for TSN-enabled Virtual Environments for Ultra-Low Latency 5G Scenarios

Andrea Garbugli, Lorenzo Rosa, Luca Foschini, Antonio Corradi, Paolo Bellavista
University of Bologna
Department of Computer Science and Engineering
Bologna, Italy
name.surname@unibo.it

***Abstract*—The recent trend of moving cloud computing capabilities to the edge of the network is reshaping the way applications and their middleware supports are designed, deployed, and operated. This new model envisions a continuum of virtual resources between the traditional cloud and the network edge, which is potentially more suitable to meet the heterogeneous Quality of Service (QoS) requirements of the supported application domains. Yet, mission-critical applications such as those in manufacturing, automation, or automotive, still rely on communication standards like the Time-Sensitive Networking (TSN) protocol and 5G to ensure a deterministic network behavior: in this context, virtualization might introduce unacceptable network perturbations. In this paper, we demonstrate that latency-sensitive applications can execute in virtual machines without disruptions to their network operations. We propose a novel approach to support the TSN protocol in virtual machines through a precise clock synchronization method and we implement it in integration with state-of-the-art and highly-efficient network virtualization techniques. Our experimental results show that it is possible to achieve deterministic and ultra-low latency end-to-end communication in the cloud continuum, for example providing a guaranteed sub-millisecond latency between remote virtual machines.**

***Index Terms*—time-sensitive networking, cloud continuum, network virtualization, ultra-low latency**

I. Introduction

The widespread adoption of the Internet of Things (IoT) concept is driving an unprecedented process of digitalization in many application domains, such as automotive, industry 4.0, healthcare, and smart cities. The exponential growth in the number of connected devices is fueling this transition by collecting huge volumes of raw data that a new generation of IoT applications should transform into insightful information. This, in turn, is enabling innovative processes, services, and products to be offered in any industrial and societal sector. A distinguishing characteristic of such transformation is the high heterogeneity that stems from devices with different computing power, battery life, mobility, etc. As a consequence, several IoT applications with extremely different goals and quality requirements have to coexist with each other, using heterogeneous technologies and resources, i.e., communication protocols, storage, computing capacity, energy requirements, and security [1].

This coexistence is one of the factors pushing the well-established traditional cloud computing paradigm to show its limitations. The traditional cloud-centric model, where few datacenters collect and process all data generated by physically distant IoT devices, is not suitable to handle the heterogeneous requirements of IoT applications. Hence, a recent trend is to integrate the traditional cloud infrastructures with a hierarchy of virtualized computing resources, sometimes identified as fog nodes, physically located between traditional cloud datacenters and IoT datasources. The resulting computing model is a fluid dissemination of virtualized resources named as *Cloud-to-Thing Continuum* (C2TC). In C2TC, providers can offer cloud-like features even outside datacenters, for example by assigning slices of the resources to different applications to satisfy their requirements, by guaranteeing isolation and by distributing the workload at all levels of the infrastructure.

Notwithstanding the potential advantages provided by the C2TC model, its adoption is currently quite limited in real-world production environments. In particular, applications in several areas may have stringent performance constraints that are difficult to fulfill while using resource virtualization: for example, software components in industrial automation or healthcare should communicate with ultra-low latency (ULL), which entails that the network infrastructure should guarantee sub-millisecond communications between remote processes. Designing, implementing, and deploying real systems that work under such tight deadlines is currently a still open research challenge, even exacerbated by the adoption of virtualization [2], [3].

Recently, various technologies for C2TC scenarios emerged at different levels, and are starting to show to be successful in coping with these strict constraints [4]. In Local Area Networks (LAN), these consist of standards like DetNet or Time-Sensitive Networking (TSN) [5], an extension to the Ethernet protocol designed to achieve a deterministic network behavior in safety-critical contexts. For larger distances, the 5G standard is on the way to wider adoption, providing an ultra-reliable, low-latency option [6]. However, an efficient use of those technologies requires the cooperation of several actors. For example, LANs are generally under the direct control of an organization, whereas WAN communications are delegated to specialized service providers. Thus, if the goal is to guarantee a maximum $1\,\mathrm{ms}$ latency between two remote processes (e.g., two industrial controllers in remote factories), each involved actor should optimize its operations: application developers should design their components to use

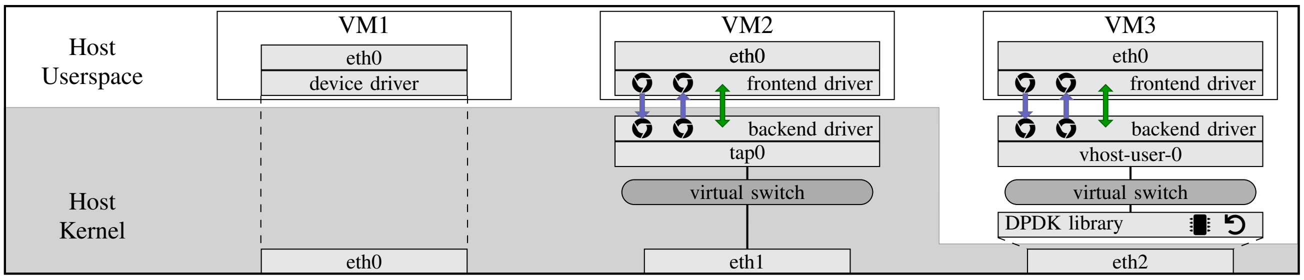


Fig. 1: Network virtualization approaches. On the left, direct device assignment. On the right, paravirtualization with kernel-level (center) or user-level (right) network virtualization.

at most $0.4\,\mathrm{ms}$, leaving the remaining latency budget to the external provider operations [7]. Consequently, very often the current trend of applications in mission-critical domains is to avoid any form of additional overhead, including resource virtualization, thus trading flexibility for performance.

In this paper, we claim that even such a small latency budget of about $0.4\,\mathrm{ms}$ suffice to operate virtualized applications under ULL constraints, thus enabling the whole potential of the C2TC model even in the most demanding scenarios of interest nowadays. To prove that, we propose a novel virtualization approach that enables TSN-based applications to run unmodified in virtual machines: our key contribution is the introduction of a precise clock synchronization method for applications running in remote VMs. We then compare how different network virtualization techniques affect the communication latency of these applications. We show that while traditional kernel-based approaches struggle to meet our demanding deadline, recent kernel-bypassing technologies consume only about $30\,\%$ of the target latency budget.

II. Background

This section provides a concise introduction to the Time-Sensitive Networking protocol for applications with stringent QoS requirements, along with a concise taxonomy of the main approaches to I/O and network virtualization.

A. Time-Sensitive Networking (TSN)

The Time-Sensitive Networking (TSN) protocol consists of a set of standards that aim to make Ethernet networks deterministic to support real-time industrial traffic [5].

The first critical requirement of real-time applications is to have a time synchronization mechanism so that all the communication participants have a unique time reference. In the context of TSN, this mechanism is provided by the IEEE 802.1AS standalone protocol that extends the Precision Time Protocol (PTP) with a specialized profile called *generic Precision Time Protocol* (gPTP). This extension defines two main entities, the *Clock Master* (CM) and the *Clock Slave (CS)*, that each network participant can associate with a network device. In this way, the device can take part in the clock synchronization process [4].

A second standard (IEEE 802.1Qbv) defines a new traffic shaper, called Time-Aware Shaper (TAS), designed to schedule network frames that belong to different types of time-critical flows. Specifically, the standard defines time-aware communication windows, called *time-aware traffic windows*, each associated with a specific queue of a network device. Each window can be used to transmit different classes of traffic, and for this reason, it is divided into *time slots* that repeat cyclically: frames belonging to the same class of traffic are buffered until the next opening of the time slot associated with their class. In this way, assured traffic is guaranteed to have low latency and jitter, and best-effort traffic cannot interfere with it. In practice, windows and slots are defined through a Gate Control List (GCL) that identifies the moments in time when one or more queues are open for frame transmission [4]. Therefore, TAS is applicable for ULL requirements but needs to have all time windows synchronized, which is why it must be used in conjunction with PTP.

B. Network virtualization techniques

A key challenge for applications running in virtual machines is to obtain efficient access to I/O devices, especially if network performance is critical. Currently, the two prominent I/O virtualization techniques are direct device assignment and paravirtualization. With direct device assignment, a dedicated device instance is assigned exclusively to a VM and becomes invisible to the host (*passthrough*). If this instance corresponds to the physical device (physical function, PF), each VM requires a dedicated physical network adapter. To mitigate this effect, recent devices support hardware-assisted virtualization (e.g., SR-IOV [8]) that makes them appear as multiple separate devices called virtual functions (VFs), which can be assigned to different VMs. Either way, this technique avoids any involvement of the hypervisor: VMs can access the network as if they were physical hosts and achieve optimal network performance. However, the direct assignment also tightly couples network devices and VMs, strongly limiting the flexibility properties of virtualization, e.g., live migration.

On the contrary, the paravirtualization technique trades performance for flexibility. For each VM, a traditional paravirtualized network stack splits the device driver into a *frontend driver* in the guest OS and a *backend driver* on the host (Fig. 1), which exchange commands using a dedicated communication channel. This separation allows the hypervisor to have full control of the network state, thus enabling a high degree of flexibility but also introducing overhead on data path operations when crossing the guest/host boundaries. In the traditional approach, the backend driver is located on the host kernel space and all the traffic should traverse the whole host network stack, which involves multiple data copies and context switches [9]. Since this overhead can be significant, it is possible to move the backend driver in the host userspace and use kernel-bypassing techniques such as the Data Plane Development Kit (DPDK) [10] to directly access the physical device with a zero-copy semantic.

The *de facto* standard framework for paravirtualization is virtio [11], which allows the hypervisor to expose paravirtualized devices to the guest. To reduce the network overhead, virtio separates the data plane, used for the actual transmission of network traffic between the host and the guest, and a control plane to exchange control messages about the data plane. The data plane consists of shared memory regions between the frontend driver on the guest and the backend driver on the host. Those memory areas, called *virtqueues*, are organized as couples of ring buffers that contain data to be received and transmitted, thus simulating the actual queues of physical devices. Each virtual device can have zero or more queues associated, but, importantly, devices with more than one queue can only be used from virtual machines with two or more virtual CPUs (vCPUs) because each queue must have its associated thread. On the other hand, the control plane consists of a notification mechanism that is used to detect and notify data in the queue between the frontend and the backend driver. For network devices, this mechanism consists of a direct inter-process communication channel between the two drivers.

Once the VM traffic reaches the backend driver, it must be forwarded to a network. In cloud environments, the common practice is to connect the VMs belonging to the same tenant to a virtualized overlay network, regardless of the host they are running on. The key component to achieve this purpose is the virtual switch, a software application that can isolate and manage traffic among VMs on the same host, but also forward data to remote switch instances through point-to-point network tunnels. For example, Open Virtual Switch [12] is a widely used solution for virtual networking. It can operate at kernel level but also in combination with DPDK to process traffic at a higher speed in the userspace. In the following, we evaluate the performance of both these options to investigate the cost of virtualization in ultra-low latency scenarios.

III. Our Framework for ULL End-to-end Communications in C2TC TSN Environments

This section describes our virtualization proposal to enable critical ULL applications based on TSN and running on top of virtualized computing and networking resources.

Let us start with a description of the typical operation context of this kind of application. Consider, for example, an industrial application where a robotic appliance (*subscriber*) periodically exchanges messages with a remote software component (*publisher*), physically located in another factory. A common requirement is that any of those messages should take no more than a specified and very small amount of time, e.g., $1\,\mathrm{ms}$, to reach the counterpart (*end-to-end latency*).

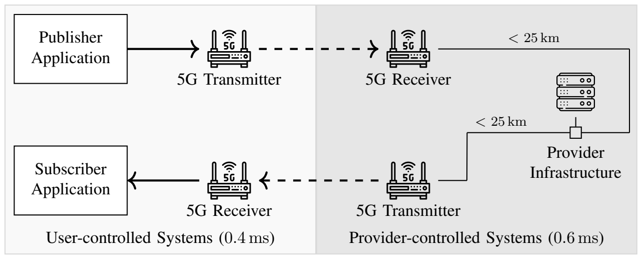


Fig. 2: Typical latency budget distribution in ULL applications.

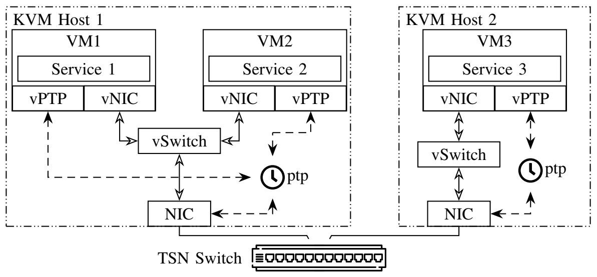


Fig. 3: Virtualization architecture for TSN-based networks.

Importantly, to meet this tight deadline, developers usually have available only a small portion of the whole latency budget, because much of the time should be allocated "externally" to the provider in charge of wide-area transmission propagation, as depicted in Fig. 2. Given that very often this kind of industrial communication is periodic, the usual solution to guarantee ULL is to reserve specific time slots for each transmission, thus effectively making the network deterministic, through the TSN protocol (see Section II).

Two reasons currently prevent the adoption of virtualization solutions in this scenario and limit the use of TSN protocol to bare-metal applications only. On the one hand, current virtualization techniques do not provide virtual network devices with a paravirtualized hardware clock, forcing the PTP-based synchronization to use a *software mode* which does not allow precise time synchronization. On the other hand, VMs require virtual network devices that support multiple queues and are efficient in their operations, because packet I/O and processing operations between the host and the guest machines can be a source of significant overhead that may jeopardize the effort towards bounded end-to-end latency. In the remainder of this section, we describe how we address those two critical aspects for mission critical applications.

1) Virtualized TSN protocol: Our first contribution is the design of a virtualization approach for TSN networks (Fig. 3). To support ULL communications, the network frame scheduler (IEEE 802.1Qbv) requires synchronization of the clocks of all participants in the TSN network (Section II). To achieve the same effect in VMs, we first use a PTP service to synchronize all physical hosts; then, we provide the guest OS with a paravirtualized clock that is transparently synchronized with the host's real-time clock by the hypervisor. This approach effectively creates a virtualized PTP clock (*vPTP*) that each VM can use as a reference to synchronize its system clock through an NTP daemon.

Once the synchronization mechanism is in place, to effectively implement the time-aware traffic windows defined by the IEEE 802.1Qbv standard, the network interfaces must support multiple broadcast queues so that each queue can be associated with a different class of traffic. Within VMs, paravirtualized devices emulate multi-queue transmission support across multiple *virtqueues* (section II). However, an important constraint is that a virtual device with multiple queues must be associated with a VM with two or more virtual CPUs, because, as discussed earlier, each queue must have its associated thread. This assumption turns out to be true even if we provide the VM with a physical TSN-enabled network card through a passthrough mechanism. In this last case we can choose to synchronize the VM using directly the physical network card and, if this is available, use offload mechanisms of traffic scheduling to improve performance. Due to space constraints, in this paper we will not focus on this case.

2) Network virtualization: A crucial aspect of virtual networking is how efficiently the host processes packets to and from VMs: inefficiencies at this level may prevent our approach to meet the stringent latency deadline, making the guest-level time-sensitive scheduling ineffective. In our framework we consider two network paravirtualization techniques, introduced in section II, that differentiate the way packets are processed (datapath operations): in the kernel of the host, or directly in the userspace (Fig. 1). Running the datapath in the kernel host is currently the most mature approach, as it guarantees broad vendor support and also enables the integration with other kernel-based tools for monitoring and control (e.g., Conntrack, BPF). However, this option may lead to poor network performance because of multiple data copies and context switches. Conversely, kernel-bypassing techniques offer the reverse: an excellent datapath performance, as they avoid those sources of overhead by directly interacting with the device driver, but a high CPU utilization (one or more host cores dedicated to poll incoming messages) and little integration with other kernel-based tools. In the context of our work, both options allow supporting the TSN protocol within a VM: developers will choose the most appropriate depending on their actual performance and resource usage constraints.

Overall, our TSN virtualization proposal enables existing TSN-based applications to seamlessly execute in virtual machines, thus taking advantage of the virtualization properties without any modification to their source code. Depending on the network virtualization approach, developers can choose to optimize the TSN traffic of the VMs for the best network performance or the lowest resource usage.

IV. Experimental evaluation

The goal of this section is to assess the performance of our virtualization framework. To this end, we built a simple TSN application consisting of one publisher and one subscriber, each running in virtual machines that execute on two remote hosts. We then set up a latency test where the publisher sends UDP packets with a publishing cycle of $1\,\mathrm{ms}$: this traffic pattern, not uncommon in real TSN applications, puts the entire network pipeline under stress as it magnifies any source of latency overhead. In particular, the test measures two representative indicators for TSN communications, *end-to-end latency* and *jitter*. The end-to-end latency of a message is defined as the time interval between its scheduled transmission time on the publisher and its actual reception time by the subscriber. The *jitter* measures how much the actual arrival time of each message differs from its expected arrival time: more precisely, if t_i is the arrival time of the i-th message, its jitter is defined as $Jitter(i) = t_i - (t_{i-1} + T)$, where T is the transmission period (in this paper, $T = 1ms$). The clocks of the two VMs are synchronized according to the mechanism presented in Section III.

To prove that this communication respects the ULL constraints, we set a threshold of $1\,\mathrm{ms}$ as the maximum end-to-end latency acceptable for each message. Recall that, according to our previous discussion, in real production environments about the $60\,\%$ of this time will be needed by the wide-area network provider to propagate data over 5G networks which are out of the control of the end-system developers (Fig. 3). To take this constraint into account, we adopt the end-system developer perspective and design our testbed as a couple of physical hosts directly interconnected by an Ethernet cable, assuming negligible propagation time between them. Consequently, our ULL deadline becomes $0.4\,\mathrm{ms}$ of end-to-end latency between the two TSN application components.

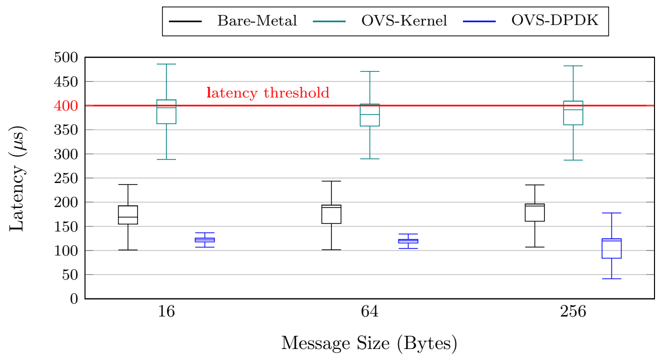
Under those assumptions, we run the latency test for typical values of message payload size in TSN applications, i.e., 16, 64, and 256 bytes. Furthermore, we repeated each test using first a virtual switch with a kernel-based datapath, and then with the kernel-bypassing approach. Each test was run for 100 seconds. Finally, to assess the impact of the whole virtualization framework on the communication properties, we also run the tests directly on the bare-metal hosts.

A. Experimental Settings

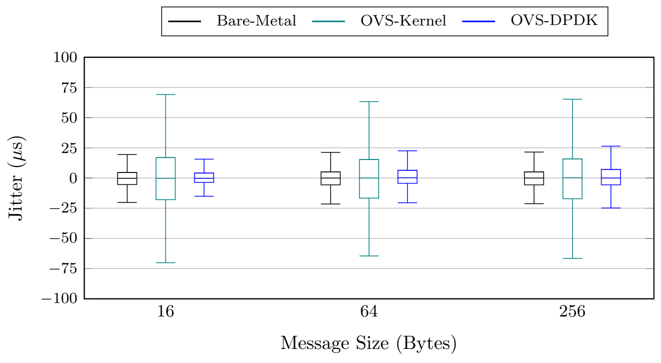
Our testbed comprises of two UP Xtreme boards, each equipped with 4 TSN NICs (Intel I210), Intel i3-8145UE 2/4 CPU, and $8\,\mathrm{GB}$ RAM. The two hosts are directly interconnected through an Ethernet cable. Each host runs Ubuntu 20.04 with Linux kernel 5.4.0. The two TSN application components execute in VMs that run the same OS of the host and are managed by QEMU/KVM (v4.2.1). Network virtualization is operated through the virtio framework [11] and OVS [12] (v2.13.3) in the two variants, with kernel-based datapath and with the kernel-bypassing DPDK library [10] (v19.11.7).

B. Performance Results

Fig. 4 and Fig. 5 show the results of the latency tests. Let us first consider the behavior of the virtualized applications. We note that the option with the kernel-based datapath struggles to meet our target deadline: in particular, Fig. 5 shows that the average message latency, computed every 10 seconds on all the messages exchanged since the previous measurement, is just below the threshold. We observe the same if we consider



(a) End-to-end latency



(b) End-to-end jitter

Fig. 4: End-to-end latency and jitter for different payload sizes and network virtualization techniques.

the median values reported in Fig. 4a for all the considered payload sides, which means that about half of the measures exceed the ULL constraints. Even worse, despite the use of the TSN protocol to reduce the latency variability, jitter remains relatively high (Fig. 4b). Instead, if we consider the kernel-bypassing approach we observe the opposite behavior: the average latency remains just above 100 µs during all the experiments and the overall median value is around 120 µs for all the message sizes. That median value is about 3.25 times lower than the kernel-based alternative and it represents just the 30% of the total available latency budget. Even better, the jitter is really small for all the considered cases, which means that this option can effectively preserve the determinism provided by TSN. At this point, it would be interesting to investigate whether this approach could preserve this behavior even on a saturated network, as this is what developers expect from a TSN application. Due to space constraints we do not discuss this aspect in this paper, but our experiments show that latency does not change significantly when the network between the two VMs is saturated. Therefore, we conclude that the kernel-bypassing network virtualization approach can effectively allow virtualized TSN applications to respect of the ULL constraints and to preserve a reduced latency variability.

The significant difference between the considered approaches depends on the way they handle the packets between the external network and the virtio backend driver (see Fig. 1). In the traditional kernel-based approach, packets forwarded by the virtual switch dataplane should still traverse the Linux kernel networking stack, which is notoriously slow (scheduling, interrupts, data copies, context switches). Then, it is not surprising that the network performance is much better, both in terms of latency and jitter, if we bypass that stack completely. Even though this speed comes at the price of dedicating 100 % of part of the CPU cores to handle packet processing (see section II), and the overall complexity in network setup and management increased, kernel-bypassing on the host can fully satisfy our ULL constraints.

Finally, we compare the performance of our virtualized TSN application against those of the same application running on bare-metal hosts. For the 64 bytes case, the average latency

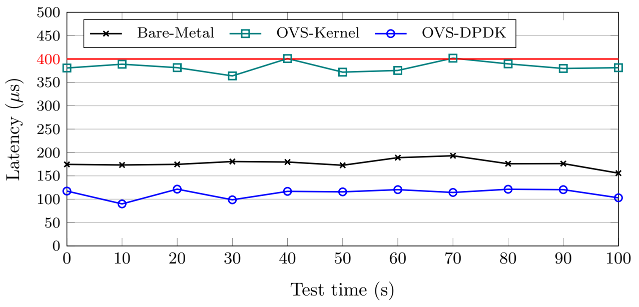


Fig. 5: End-to-end latency averaged every 10 seconds, for 64 bytes payload size. The red line is the latency threshold.

of the bare-metal application is constantly around 175 µs and the median is 190 µs, with a small jitter. These values are about two times lower than the kernel-based virtualization approach. This result is easy to explain: in the former each UDP packet should traverse only the host kernel, whereas in the latter packets are also managed by the guest kernel. On the other hand, it may appear quite surprising that the kernel-bypassing virtualization approach performs even better than the bare-metal alternative: it is true that the host kernel is bypassed, but packets still need to be handled by the guest kernel. There are two main reasons for this particular behavior. First, as we discussed, the kernel-bypassing technique is really much more efficient than the operations in the host kernel, as it avoids data copies. Second, the network operations in the host kernel require a context switch to a kernel thread, whereas the guest kernel executes in the same process that operates the VM. Thus, on our testbed, once a single UDP packet with a payload of 64 bytes is received by the host network device, it takes 20 µs to be delivered to the application on the guest. The same operation on the same packet takes 70 µs through the host kernel. Therefore, the combination of those two factors with a traffic pattern that magnifies any network overhead explains this performance effect. In fact, our virtualized TSN application appears even faster than the bare-metal equivalent (36% lower latency, considering the median value for 64B packets) and provides almost the same jitter. To obtain a more appropriate comparison we would need to create a kernel-bypassing TSN host application and confront it against our current best option, but this is impossible as currently the TSN scheduler is only available in the kernel.

In conclusion, these performance results demonstrate that latency-critical TSN application can respect the ULL constraints even when executing in virtual machines. In particular, kernel-based network virtualization solutions introduce a high latency variability and struggle to meet the target deadline, whereas kernel-bypassing techniques provide excellent results, as they consume only 30% of the available latency budget.

V. Related Work

Our TSN virtualization approach is based on several insights from previous works. In [13], Xen and KVM are proposed as suitable real-time hypervisors, but the work asses their characteristics without taking into account a deterministic network communication like one based on TSN. Another recent work proposes a container-based architecture for the flexible reconfiguration and redeployment of specific process control systems, which however does not apply to virtual machines [14]: they evaluate their proposal through a PTP-synchronized testbed and show that low-latency QoS requirements can be met, but they do not take advantage of the deterministic scheduling techniques defined in TSN nor do they focus on the impact of the network virtualization technique on latency overall. Finally, the actual feasibility of TSN virtualization is explored in [15], where three different approaches to enhance hypervisors for time-triggered communication are briefly discussed. We adopt one of them, which guarantees applications to run unmodified on an unmodified hypervisor. However, their focus is more on the architectural principles, which are only evaluated through simulations and never validated on an actual testbed.

A crucial result of our paper is about fast packet processing techniques, of which a detailed comparison is presented in [16]. This work shows the superior performance of kernel-bypassing approaches over traditional techniques, but the comparison is between containerized applications (not VMs) with a very different traffic pattern compared to ours. Finally, the use of kernel-bypassing techniques to meet the constraints of ULL applications has mainly been explored in the field of Software-Defined Networking (SDN), where network functions executing in VMs should process packets at high speed [7]. Our work is complementary to this effort as it addresses the user-controlled infrastructure instead of the provider-controlled portion.

VI. Conclusion and Future Work

Resource virtualization is an enabling factor for the heterogeneous QoS requirements of Internet of Things applications. However, applications in mission-critical domains have stringent ultra-low latency communication constraints. To meet them, developers usually rely on standards like the Time-Sensitive Networking (TSN) protocol combined with 5G networks to ensure a deterministic, time-triggered network behavior. For those applications, virtualization may introduce a source of unpredictability that overwhelms all the flexibility advantages. To address this concern, this paper introduced a novel approach to execute TSN-based applications under ultra-low latency (ULL) constraints in virtual machines: our proposal combines an effective clock synchronization approach for remote VMs with high-performance network virtualization techniques. On a real testbed, we demonstrated that this solution respects ULL constraints, thus effectively enabling unmodified critical applications to benefit from the virtualization advantages.

Future work includes surveying other kernel-bypassing techniques, either software (e.g., XDP) or hardware (OVS DPDK offload, RDMA, SmartNICs), to support high-performance packet processing using less computing resources. In the longer term, based on the considerations of this paper, we envision that a userspace implementation of the TSN scheduling may unleash the full potential of those approaches.

Acknowledgements. The authors are grateful to the anonymous reviewers for their constructive comments and suggestions.

References

[1] L. Bittencourt, et al., "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3-4, pp. 134–155, 2018.

[2] G. Sutton, et al., "Enabling technologies for ultra-reliable and low latency communications: From phy and mac layer perspectives," *IEEE Communications Surveys Tutorials*, vol. 21, no. 3, pp. 2488–2524, 2019.

[3] R. Ali, et al., "Urllc for 5g and beyond: Requirements, enabling incumbent technologies and network intelligence," *IEEE Access*, vol. 9, pp. 67 064–67 095, 2021.

[4] A. Nasrallah, et al., "Ultra-low latency (ull) networks: The ieee tsn and ietf detnet standards and related 5g ull research," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 88–145, 2019.

[5] J. Farkas, et al., "Time-sensitive networking standards," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 20–21, 2018.

[6] P. Trakadas, et al., "A cost-efficient 5g non-public network architectural approach: Key concepts and enablers, building blocks and potential use cases," *Sensors*, vol. 21, no. 16, 2021.

[7] Z. Xiang, et al., "Reducing latency in virtual machines: Enabling tactile internet for human-machine co-working," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 5, pp. 1098–1116, 2019.

[8] *PCI SIG. Single Root I/O Virtualization*. [Online]. Available: https://pcisig.com/specifications/iov/single_root/

[9] Q. Cai, et al., "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, 2021, p. 65–77.

[10] Linux Foundation, "Data Plane Development Kit (DPDK)," 2015. [Online]. Available: http://www.dpdk.org

[11] R. Russell, "virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 95–103, Jul. 2008.

[12] B. Pfaff, et al., "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015, pp. 117–130.

[13] L. Abeni, et al., "Using Xen and KVM as real-time hypervisors," *Journal of Systems Architecture*, vol. 106, no. July 2019, 2020.

[14] M. Gundall, et al., "Introduction of an Architecture for Flexible Future Process Control Systems as Enabler for Industry 4.0," in *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, vol. 2020-Septe, 2020, pp. 1047–1050.

[15] L. Leonardi, et al., "Towards time-sensitive networking in heterogeneous platforms with virtualization," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 1155–1158.

[16] G. Ara, et al., "Comparative evaluation of kernel bypass mechanisms for high-performance inter-container communications," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,*, INSTICC. SciTePress, 2020, pp. 44–55.