KuberneTSN: a Deterministic Overlay Network for Time-Sensitive Containerized Environments

KuberneTSN: a Deterministic Overlay Network for Time-Sensitive Containerized Environments

Andrea Garbugli, Lorenzo Rosa, Armir Bujari, Luca Foschini
University of Bologna
Department of Computer Science and Engineering
Bologna, Italy
name.surname@unibo.it

***Abstract*—The emerging paradigm of resource disaggregation enables the deployment of cloud-like services across a pool of physical and virtualized resources, interconnected using a network fabric. This design embodies several benefits in terms of resource efficiency and cost-effectiveness, service elasticity and adaptability, etc. Application domains benefiting from such a trend include cyber-physical systems (CPS), tactile internet, 5G networks and beyond, or mixed reality applications, all generally embodying heterogeneous Quality of Service (QoS) requirements. In this context, a key enabling factor to fully support those mixed-criticality scenarios will be the network and the system-level support for time-sensitive communication. Although a lot of work has been conducted on devising efficient orchestration and CPU scheduling strategies, the networking aspects of performance-critical components remain largely unstudied. Bridging this gap, we propose KuberneTSN, an original solution built on the Kubernetes platform, providing support for time-sensitive traffic to unmodified application binaries. We define an architecture for an accelerated and deterministic overlay network, which includes kernel-bypassing networking features as well as a novel userspace packet scheduler compliant with the Time-Sensitive Networking (TSN) standard. The solution is implemented as *tsn-cni*, a Kubernetes network plugin that can coexist alongside popular alternatives. To assess the validity of the approach, we conduct an experimental analysis on a real distributed testbed, demonstrating that KuberneTSN enables applications to easily meet deterministic deadlines, provides the same guarantees of bare-metal deployments, and outperforms overlay networks built using the *Flannel* plugin.**

***Index Terms*—time-sensitive networking, container, kubernetes, cloud continuum, network virtualization, bounded latency**

I. Introduction

The promise of edge computing is that of increasingly low latency, high bandwidth communication, and improved data security and privacy. Therefore, a stronger push for edge applications and service deployment is to be expected [1]. However, in contrast to traditional cloud deployment environments, the edge has limited resources and may not be able to satisfy the overlapping and heterogeneous resource demands of all such applications. This fact has motivated researchers to extend the well-established cloud computing paradigm into the idea of *edge-cloud computing* where an increasingly rich and heterogeneous set of resources between datacenters and the network edge, often called *cloud continuum*, can be virtualized to host cloud-like services [2]. The power of this paradigm relies on the combination of the well-known advantages of the cloud model, in particular flexibility, cost-effectiveness, and reconfigurability, with the performance advantage of running services as close to their final user as possible.

The success of this model is clear from its rapid and wide adoption in several heterogeneous domains, including application domains that embody time-sensitive requirements. As an example, the reference architecture of 5G and beyond standards relies on virtualized applications deployed in edge datacenters, or even co-located with the widely distributed base stations [3]. Control applications in the domains of Cyber-Physical Systems (CPS), Industrial Internet of Things, Tactile Internet, and in many other fields are increasingly pursuing the disaggregation trend, with virtualized application components deployed across the whole continuum of available resources, embodying heterogeneous Quality of Service (QoS) requirements, even among their internal components [4], [5]. Although many of those requirements can be easily met just by placing services physically closer to their final users, reducing key metrics such as latency or response time, core parts of these systems still struggle to balance strict performance demand with the overhead introduced by virtualization.

To mitigate this overhead, lightweight virtualization techniques like containerization have become the standard technology for platform-independent prototyping, development, and deployment of edge components. Compared to hypervisor-based virtual machines, containers are generally characterized by reduced overhead and higher scalability, representing a potential for innovation in service patterns, in virtue of setting up a unified service provisioning platform capable of adhering to applications' QoS specifications [6]. Furthermore, containers are seamlessly integrated into resource management and orchestration platforms, with Kubernetes in its full or reduced versions (e.g., k3s) as the *de-facto* standard technology [7]. Resource management and orchestration are paramount in the edge cloud, as it automatically deploys, monitors, and migrates containerized application components across the shared infrastructure, enforcing applications' QoS specifications.

However, containerization alone is not a panacea. Given the highly distributed nature of edge cloud applications, specific attention to network and system-level aspects is paramount to effectively support the most performance-demanding components. Yet, previous work mostly focused on efficient orchestration and CPU scheduling of containers [8]–[10], leaving those aspects largely unstudied.

In this paper, we design a cost-efficient solution to enable *accelerated and deterministic communication* among containerized applications. To this end, we define a novel architecture for a container overlay network that combines two techniques for high-performance communication. First, we adopt a form of kernel-bypassing networking to remove the overhead of the kernel networking stack [11]. Second, we propose a novel userspace packet scheduler, compliant with the Time-Sensitive Networking (TSN) standard, to allow the time-bounded data distribution and communication among networked components [12]. We implement our proposal as *tsn-cni*, a novel Kubernetes network plugin that can be seamlessly integrated alongside existing options (e.g., Flannel, Calico). This way, application designers are free to choose the most appropriate support for traffic flows with different degrees of criticality. Finally, we evaluate *tsn-cni* on a real testbed, showing that containerized TSN applications can achieve determinism and performance comparable to bare metal applications, and better than using the network fabric set up by the popular *Flannel* plugin.

II. Background

This section provides a brief introduction to container overlay networks, their rationale, and support in the Kubernetes platform. Next, we provide a concise background on the TSN standard and kernel-bypassing techniques.

A. Container Overlay Networks

Containers generally have four networking modes available: bridge, host, macvlan, and overlay. The overlay mode is the most popular, especially in combination with Kubernetes, as it provides better isolation, ease of use, and security; hence we limit our description to this scheme. In this mode, as depicted in Fig. 1, containers are connected on an overlay network, potentially spanning multiple physical nodes even across different networks. On each container, a virtual network interface is created, to which applications can assign an arbitrary IP address. This interface is connected to the outside through a virtual switch, located in the host operating system kernel, which has two main roles: it works as a network bridge to allow communication among co-located containers, and it tunnels network traffic toward the remote container(s) across the physical network. This way, containers on the same overlay network have an isolated address namespace and configuration settings, disjoint from the host network or from other overlays.

When using Kubernetes, by default each container has a single network interface for all the network traffic, including management and control plane interactions (e.g., with the Kubernetes master). To distinguish among different traffic classes, the Multus plugin [13] allows attaching additional interfaces to containers. Multus is a meta-plugin, as it defines a *container network interface* (CNI) that other plugins can implement to configure a Layer 3 network fabric and optionally provide additional advanced features. Several such plugins are available, such as Flannel, Calico, or Weave. Unfortunately, none of those supports the definition of an accelerated and deterministic communication channel among containers. Compared to these alternatives, in this work we design a novel plugin architecture to offer such guarantees. We still rely on a virtual switch, but we move the sender-side datapath to userspace and provide a novel packet scheduler compliant with the TSN standard. This choice allows users to obtain enhanced network performance with no modifications to application binaries and atop off-the-shelf hardware and operating systems, without requiring any patches or specific configurations from the final user.

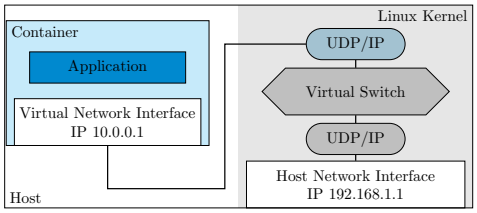


Fig. 1: Container networking in overlay mode.

B. Time-Sensitive Networking

Designed to support soft real-time industrial traffic, the set of standards grouped under the name of Time-Sensitive Networking aims to introduce determinism to IEEE 802.1 networks via a set of features, including but not limited to time synchronization, programmability, etc. [12]. First, TSN requires that all the communication participants share a unique time reference, and the IEEE 802.1AS standalone protocol provides an adequate mechanism to ensure this synchronization [14]. A second key concept in TSN is packet scheduling. The IEEE 802.1Qbv standard defines a traffic shaper, called Time-Aware Shaper (TAS), that can prioritize the frames belonging to classes of traffic with different time criticality. This prioritization is based on time-aware communication windows, called *time-aware traffic windows*, that repeat cyclically. Each window is divided into *time slots* that can be associated to different traffic classes: frames belonging to the same class are buffered until the next opening of the associated time slot. This way, TSN guarantees bounded latency and jitter to time-critical traffic, as well as no interference from best-effort traffic.

From a practical viewpoint, to enable this kind of communication, developers must configure the kernel-based Traffic Controller (TC), which implements a TAS shaper, to set up the desired number of traffic classes, their priority, and time slots duration. Then, applications open a datagram socket with the `SO_TXTIME` flag, so that they can associate a desired transmission time to each outgoing message. Unfortunately, there are two obstacles to the adoption of this standard from containerized environments. First, we noted that some OS images do not support the `SO_TXTIME`. Second, the transmission time is never forwarded outside the container network namespace to the virtual switch. To overcome these limitations, KuberneTSN intercepts the container TSN traffic and forwards it to a novel userspace scheduler, responsible to enforce the TAS shaping. This component, which replaces the Linux-specific kernel-based scheduler, is the key architectural

element that we leverage to provide time-sensitive networking features to containerized applications, and it is fully integrated into the *tsn-cni* Kubernetes plugin.

C. Kernel-bypassing Networking

In a container overlay network, each outgoing packet must cross the networking stack twice, one in its isolated network namespace and one in the host namespace, and must also cross through a virtual switch (Fig. 1). The combination of all these steps adds significant per-packet communication overhead [15], unacceptable for time-sensitive edge applications.

In recent years, several *kernel-bypassing* networking approaches, also known as network acceleration techniques, have emerged to support performance-critical applications. Among them, the Data Plane Development Kit (DPDK) [16] is an increasingly popular library that adopts this approach without requiring special hardware or OS support. DPDK lets applications access a userspace version of the network device drivers (*Poll Mode Drivers*) to directly send or receive Ethernet packets on the network. Applications and drivers exchange data through a shared memory area registered with the network card for Direct Memory Access (DMA), thus communication is *zero-copy* and avoids kernel/user context changes. This way, communication is much more efficient, and, in principle, applications in the edge cloud would immensely benefit from the related performance improvements. However, DPDK exposes a low-level C interface, very difficult to use and scarcely integrated within virtualization engines [17].

In KuberneTSN, we accelerate the outgoing container data path using DPDK transparently to user applications. Specifically, we design KuberneTSN to bypass the kernel networking stack in the container namespace, sending data directly to a userspace virtual switch. Then, we adopt a userspace version of a widely used and open-source virtual switch, Open vSwitch (OVS) [18], which in turn uses DPDK to bypass the kernel networking stack in the host namespace.

Overall, KuberneTSN combines three well-known networking approaches, namely overlay networks, TSN scheduling, and kernel-bypassing networking, and leverages them to offer the option of a deterministic and accelerated inter-container communication, well integrated into the state-of-the-art Kubernetes orchestrator and complementary to existing networking approaches for best-effort traffic.

III. Related Work

Previous research on the containerization of critical application components mainly focused on orchestration strategies and CPU scheduling [8], [10]. These works investigate the best strategies to place components on suitable resources and ensure that those resources can schedule the execution of containerized applications according to their requirements. Yet, they never take network and system-related aspects into account. We consider these works complementary to our proposal, as we envision that network and computing resources for edge applications should be orchestrated together.

Despite the importance of networking for edge applications, researchers paid less attention to the networking requirements of critical applications. Abeni et al. [19] evaluate different kernel-bypass approaches for inter-container communications, outlining the great potential of DPDK as network accelerator compared to the kernel-based approach. However, their contribution is limited to a framework for performance evaluation.

Slim [15] proposes a solution to reduce the processing overhead on container overlay networks. At its crux, the proposal avoids processing packets multiple times on the same host (see Sec. II); instead, it defines a component that intercepts calls to the socket API and directly translates network addresses from the overlay into the host namespace (and vice versa). This way, packets traverse the kernel networking stack only once. SocksDirect [20] uses the same interception technique to re-route packets on an accelerated kernel-bypassing datapath, but this is possible only with the *host* container networking mode. Both these works introduce the idea of accelerating container inter-networking, and both show significant performance advantages for a wide range of applications built on top of them. However, these solutions are not integrated with standard production-ready technologies such as Kubernetes. Furthermore, as they target datacenter environments, their focus is on accelerated support for reliable connection-oriented transport protocols (TCP), and they do not provide any support for time-sensitive applications such as TSN, a key requirement for edge applications. In this work, we adopt similar techniques (socket interception, kernel-bypassing) to accelerate network operations, but we also provide guarantees on connection determinism (through TSN) and implement our solution as a plugin for highly standard development and deployment technologies.

Finally, the use of TSN in virtual environments is a relatively new trend, as the standard was originally intended for bare-metal industrial applications. Leonardi et al. [21] first hypothesized this possibility, identifying three distinct architectural approaches to enhance hypervisor-based virtualization with time-triggered communication. In a previous work [22], we showed for the first time on a real testbed that TSN applications can execute in remote virtual machines, embodying even better performance than bare-metal thanks to the adoption of kernel-bypassing techniques. In this paper, we target containerized applications and take a step further by implementing our solution as a Kubernetes network plugin, thus allowing an application to select the most appropriate overlay network meeting their requirements.

IV. KuberneTSN: an Accelerated and Deterministic Overlay Network

KuberneTSN defines the architecture for a novel *accelerated* and *deterministic* container overlay network, addressing the time-sensitive requirements of containerized business or control logic. To achieve this goal, we intervene and modify the packet processing pipeline for the *outgoing* container traffic through the use of two novel architectural components: a user library named *LibKTSN*, and a daemon named *KTSNd*.

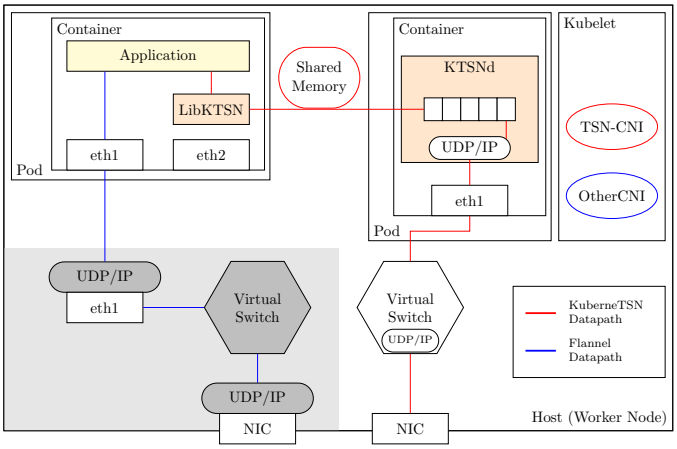


Fig. 2: The architecture for an accelerated and deterministic overlay network, implemented as a Kubernetes CNI plugin.

Fig. 2 shows those components and the role they play in the definition of a new data path for time-sensitive traffic.

LibKTSN exposes the standard POSIX socket interface to the application binaries. This way, any time the application issues a send operation on a datagram socket, the library intercepts it and forwards the packets to a memory area shared with the KTSNd daemon. We are interested in servicing time-sensitive traffic, so we only capture outgoing transmissions that have an explicit transmission time, i.e., TSN traffic, with the `SO_TXTIME` socket option. Otherwise, packets are forwarded onto the regular data path. This approach enables TSN networking regardless of the container images, unlike the currently available alternatives (see Sec. II). LibKTSN is the only component of our solution that should be present in the application container. We provide it as a shared library and use the flag `LD_PRELOAD` to transparently intercept traffic: hence, no changes are required to the application code.

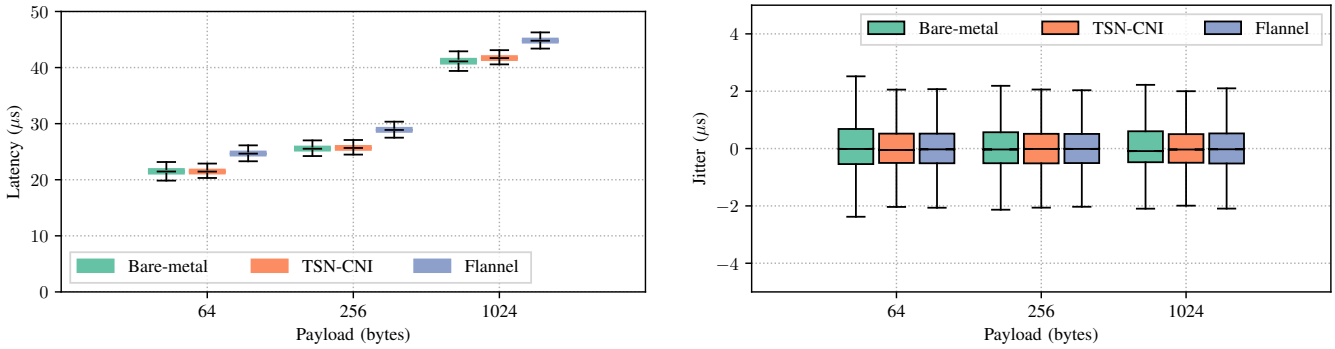
The *KTSNd* daemon represents the key component of our proposal, as it works both as a packet scheduler and a network accelerator. Once it detects a new packet from an application, KTSNd schedules its actual transmission based on the application-provided transmission time. Although we design the daemon to be agnostic to the specific scheduling strategy, by default it works as a Time-Aware Shaper (TAS) compliant with the IEEE 802.1Qbv standard (see Sec. II). Currently, this packet scheduling option is not available for containerized applications, as popular virtual switches (e.g., Linux bridge, Open vSwitch, etc.) do not support it. Therefore, our solution is the first to provide deterministic packet scheduling for unmodified application binaries running in containers.

When time comes to transmit a scheduled packet, the scheduler must send it on the network on behalf of the original application, preserving the source MAC, IP addresses, and UDP ports, and minimizing the packet processing delays to meet the user-required transmission time as precisely as possible. To satisfy these requirements, we adopt a kernel-bypassing approach and move the entire transmission pipeline in userspace. This way, we avoid the expensive double-crossing of the kernel networking stack and the unnecessary user/kernel thread context switches (see Sec. II) and instead provide our own simple and efficient implementation of the UDP/IP stack directly within KTSNd, using the DPDK library to forward packets on the virtual L2 link. This choice allows us to preserve the original packet metadata, as we can manipulate protocol headers directly, and significantly reduce the processing overhead. As shown in Fig. 2, packets are then handled by a userspace virtual switch that, in turn, should provide its own UDP/IP userspace stack to forward them on the physical network. In our implementation, we adopt a widely-used, state-of-the-art userspace virtual switch, Open vSwitch [18], which also uses DPDK for kernel-bypassing.

The simple yet powerful design makes KuberneTSN easy to integrate into standard platforms such as the Kubernetes orchestrator in its various distributions, making it ready to use for critical networked applications embodying stringent requirements. To this aim, we build a Kubernetes network plugin, *tsn-cni*, that implements our architecture. Specifically, *tsn-cni* implements the Multus CNI interface [13] and thus a Layer 3 network fabric that includes our accelerated and deterministic data path. The plugin requires applications to include LibKTSN in their execution environment, and it encapsulates the KTSNd daemon in a separate container. This approach is strategic to support time-sensitive edge applications: because multiple network plugins can be used at the same time, developers can choose standard ones (e.g., Flannel, Calico) for best-effort traffic, and *tsn-cni* for time-sensitive networking, as represented in Fig. 2. Therefore, KuberneTSN and its *tsn-cni* implementation enhance the capabilities of the edge-cloud not only by supporting deterministic networking but also by integrating this option in a familiar ecosystem for application designers. By tagging application components as time-sensitive, they can instruct Kubernetes to automatically deploy KTSNd alongside the application containers, thus transparently obtaining support for performance-sensitive workloads.

V. Experimental Evaluation

In this section, we evaluate the performance of the *tsn-cni* plugin, which implements the KuberneTSN architecture. The purpose of the experimental assessment is twofold: on the one hand, we want to show that the *accelerated* datapath we propose is indeed faster than the current state-of-the-art networking options; on the other hand, we demonstrate that our solution can in fact provide *deterministic* guarantees. In particular, we compare *tsn-cni* against two alternatives. The first is a bare metal setting that reproduces the way typical TSN applications are deployed, in order to assess the overhead introduced by the virtualization layer. The second is *Flannel*, a popular CNI plugin for Kubernetes. In its recommended configuration, Flannel uses a Linux bridge in combination with VXLAN encapsulation to implement the virtual switch, thus building an overlay network that corresponds to the *regular datapath* of Fig. 2. By comparing *tsn-cni* and Flannel, we



(a) Latency.

(b) Jitter.

Fig. 3: Performance comparison among three deployment options for the latency test application: bare metal, containerized with *tsn-cni*, containerized with *Flannel*. The experiment is repeated for increasing payload sizes: 64 B, 256 B, 1024 B.

assess whether KuberneTSN meets its design goal of providing additional performance benefits and deterministic properties to inter-container networking.

For the purpose of this evaluation, we build a simple TSN application consisting of two processes, a talker and a listener, each running inside a container on two remote hosts. We then set up a latency test in which the talker sends UDP packets with a cycle of 1 ms. The test measures two representative indicators of time-sensitive communications: end-to-end latency and jitter. The end-to-end latency of a message is defined as the time interval between the time of transmission predicted by the talker, sometimes also called transmission time, and the time of actual reception by the listener. The jitter measures how much the actual arrival time of each message differs from the 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 work, $T = 1ms$). It is noteworthy to point out that the bare-metal and the *tsn-cni* test suites are implemented as actual TSN applications, which associate a desired transmission time to each packet. However, for the test adopting Flannel as a communication choice, this option is not available, as the TSN scheduling would not be enforced (see Sec. II). Instead, the only alternative is to send one message and then sleep, repeating this behavior every T.

A. Experimental Settings

The evaluation analysis is conducted on a real testbed which reproduces an edge deployment scenario. The testbed comprises two Dell Workstations, each equipped with an Intel I225 NIC, an Intel i9-10980XE 18/36 CPU, and 64 GB RAM. The two hosts are interconnected through a physical TSN-compliant switch. Each host runs Ubuntu 22.04 with Linux kernel 5.16. When using Open vSwitch [18], we adopt its two variants, the kernel-bypassing on the sender side, and the kernel-based on the receiver side. As required by TSN, the clocks of the two hosts are synchronized using two PTP daemons. Finally, we pin the processes to dedicated cores so to avoid any bias in the measurements induced by the CPU scheduling policy.

B. End-to-end Latency

Figure 3a reports the end-to-end latency and jitter measured for three typical data sizes (64 B, 256 B, 1024 B) for each of the considered deployment scenarios: bare-metal and containerized applications with *tsn-cni* or Flannel as network plugin. A first consideration is that the performance of *tsn-cni* is always very good, with median latency values ranging from 21.5 µs in the case of small packets (64 B) to 41.7 µs for 1024 B. These values are almost identical to those registered for the bare metal deployment, with a small variation in the ns scale starting to appear for the 1KB packet size. Latency variability is negligible in both cases. If we consider Flannel, we note a slight, but evident latency increase (12 % on average). This is the result of the expensive in-kernel packet processing, which we avoid thanks to the kernel bypassing technique embodied in our solution. The same trend observed for latency is confirmed by the analysis of the jitter metric reported in Fig. 3b: the median value is zero in almost all cases and the variability is negligible. Therefore, we can conclude that KuberneTSN and its *tsn-cni* implementation succeed in minimizing the packet processing overhead for containerized applications, achieving the goal of an *accelerated* data path.

Overall, our experiments show that both *tsn-cni* and Flannel show good latency numbers, although our kernel-bypassing solution shows lower median values. In principle, one could expect even better performance from *tsn-cni*, as raw DPDK is particularly fast [19]. However, we noted that the OVS-DPDK implementation introduces a non negligible overhead on our userspace datapath, consisting of at least 23% of the total reported latency. Nevertheless, we decided to keep it in our system as it is a widely-used tool, supported by an active community. Even more importantly, while still demonstrating better performance, it supports a rich set of additional features for virtual networking, e.g. OpenFlow programmability, compared to the basic Linux bridges used by Flannel.

C. Determinism

To assess whether KuberneTSN can effectively provide deterministic guarantees to time-sensitive flows, we consider again the latency test results discussed before, but in Fig. 4 we

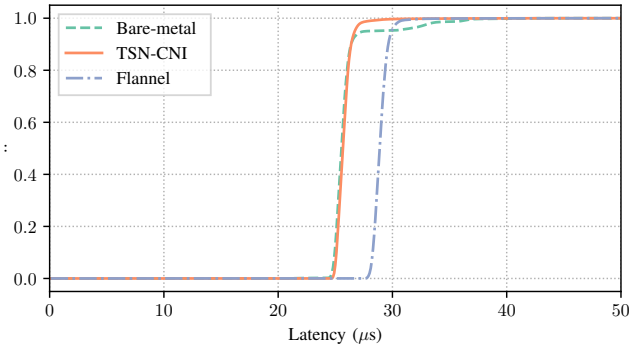


Fig. 4: CDF with packets of 256 bytes.

plot the respective Cumulative Distribution Function (CDF). Ideally, the curve should be as vertical as possible, implying a highly predictable packet reception time. In this context, the bare metal application and the containerized application using *tsn-cni* show overlapping performance, very close to the ideal behavior. In particular, for *tsn-cni* the 90 % and the 99 % probability correspond to 26.4 µs and 28.1 µs respectively. Instead, for Flannel these thresholds are 29.6 µs and 30.7 µs respectively, implying a less precise arrival time interval.

This difference demonstrates the advantage of using KuberneTSN for time-sensitive traffic. The main reason for this behavior is the way the test application sends messages: when using Flannel, we cannot explicitly set a transmission time, as this feature is not supported in current containerized environments. Hence, we are constrained to fall back to a classic send-and-sleep loop, mimicking a periodic send operation. The effect of this difference is minimal in our experiment, as we do not have other competing flows; however, previous work [23] demonstrates that time-sensitive flows require dedicated support. *tsn-cni* serves this purpose by providing essential support to containerized applications so as to meet heterogeneous flow requirements in mixed-criticality scenarios.

VI. Conclusion and Future Work

We presented KuberneTSN, an architecture for an accelerated and deterministic container overlay network. KuberneTSN defines a novel userspace TSN packet scheduler and adopts a kernel-bypassing approach to minimize packet processing delays. We implemented KuberneTSN as a network plugin for the Kubernetes orchestrator, called *tsn-cni*, so that it can be used alongside existing network fabrics to better support time-sensitive edge applications. The solution was evaluated on a real testbed, showing that containerized applications using *tsn-cni* have the same level of performance and determinism as bare metal applications, outperforming the widely used Flannel network plugin.

Future work will include a detailed performance characterization of KuberneTSN under different traffic conditions, and a demonstration of the use of *tsn-cni* in combination with other network plugins. In the longer term, as performance-demanding AI/ML components are increasingly moved to the network edge, we are interested in a systematic performance study of the inter-container datapath to highlight further optimization opportunities.

Acknowledgements

This work was partially supported by the H2020 TERMINET project (Grant agreement #: 957406).

References

[1] F.Bonomi, R. Milito, J. Zhu and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proc. of the MCC Workshop on Mobile Cloud Computing*. New York, NY, USA: ACM, 2012, p. 13–16.

[2] 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.

[3] 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.

[4] 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.

[5] R. Ali, Y. B. Zikria, A. K. Bashir, S. Garg and H. S. Kim, "URLLC for 5G and Beyond: Requirements, Enabling Incumbent Technologies and Network Intelligence," *IEEE Access*, vol. 9, pp. 67 064–67 095, 2021.

[6] A. Randal, "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers," *ACM Comput. Surv.*, vol. 53, no. 1, 2020.

[7] Cloud Native Computing Foundation, "Kubernetes." [Online]. Available: https://kubernetes.io

[8] L. Toka, "Ultra-reliable and low-latency computing in the edge with kubernetes," *Journal of Grid Computing*, vol. 19, no. 3, pp. 1–23, 2021.

[9] R. Eidenbenz, Y. -A. Pignolet and A. Ryser, "Latency-Aware Industrial Fog Application Orchestration with Kubernetes," in *Proc. of FMEC*, 2020, pp. 164–171.

[10] J. Harmatos and M. Maliosz, "Architecture Integration of 5G Networks and Time-Sensitive Networking with Edge Computing for Smart Manufacturing," *Electronics*, vol. 10, no. 24, 2021.

[11] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang and R. Agarwal, "Understanding Host Network Stack Overheads," in *Proc. of ACM SIGCOMM 2021*, 2021, p. 65–77.

[12] J. Farkas, L. L. Bello and C. Gunther, "Time-Sensitive Networking Standards," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 20–21, 2018.

[13] Kubernetes Network Plumbing Working Group, "Multus CNI." [Online]. Available: https://github.com/k8snetworkplumbingwg/multus-cni

[14] 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.

[15] D. Zhuo *et al.*, "Slim: OS Kernel Support for a Low-Overhead Container Overlay Network," in *Proc. of USENIX NSDI*, Boston, MA, 2019, pp. 331–344.

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

[17] L. Rosa and A. Garbugli, "Poster: Insane – a uniform middleware api for differentiated quality using heterogeneous acceleration techniques at the network edge," in *Proc. of IEEE ICDCS*, 2022, pp. 1282–1283.

[18] B. Pfaff *et al.*, "The Design and Implementation of Open vSwitch," in *Proc. of USENIX NSDI*, Oakland, CA, May 2015, pp. 117–130.

[19] G. Ara *et al.*, "Comparative Evaluation of Kernel Bypass Mechanisms for High-performance Inter-container Communications," in *Proc. of the CLOSER*, 2020, pp. 44–55.

[20] B. Li, T. Cui, Z. Wang, W. Bai and L. Zhang, "SocksDirect: Datacenter Sockets can be Fast and Compatible," in *Proc. of ACM SIGCOMM Conference*, 2019.

[21] L. Leonardi, L. L. Bello and G. Patti, "Towards Time-Sensitive Networking in Heterogeneous Platforms with Virtualization," in *Proc. of IEEE ETFA*, vol. 1, 2020, pp. 1155–1158.

[22] A. Garbugli, L. Rosa, L. Foschini, A. Corradi and P. Bellavista,, "A Framework for TSN-enabled Virtual Environments for Ultra-Low Latency 5G Scenarios," in *Proc. of IEEE ICC*, 2022, pp. 5023–5028.

[23] A. Garbugli, A. Sabbioni, A. Corradi and P. Bellavista, "TEMPOS: QoS Management Middleware for Edge Cloud Computing FaaS in the Internet of Things," *IEEE Access*, vol. 10, pp. 49 114–49 127, 2022.