Alma Mater Studiorum - Università di Bologna

Dottorato di Ricerca in
Computer Science and Engineering

XXXVI Ciclo

An Architecture for Network Acceleration as a Service in the Cloud Continuum

Candidato
Lorenzo Rosa

Coordinatore Dottorato
Prof.ssa Ilaria Bartolini

Supervisore
Prof. Antonio Corradi

Settore concorsuale
09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore scientifico disciplinare
ING-INF/05 SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Esame Finale Anno 2024

Abstract

The pervasive availability of connected devices in any industrial and societal sector is pushing for an evolution of the well-established cloud computing model. The emerging paradigm of the *cloud continuum* embraces this decentralization trend and envisions virtualized computing resources physically located between traditional datacenters and data sources. By totally or partially executing closer to the network edge, applications can have quicker reactions to events, thus enabling advanced forms of automation and intelligence. However, these applications also induce new data-intensive workloads with low-latency constraints that require the adoption of *specialized resources*, such as high-performance communication options (e.g., RDMA, DPDK, XDP, etc.). Unfortunately, cloud providers still struggle to integrate these options into their infrastructures. That risks undermining the principle of generality that underlies the cloud computing scale economy by forcing developers to tailor their code to low-level APIs, non-standard programming models, and static execution environments. This thesis proposes a novel system architecture to empower cloud platforms across the whole cloud continuum with *Network Acceleration as a Service* (NAaaS). To provide commodity yet efficient access to acceleration, this architecture defines a layer of agnostic high-performance I/O APIs, exposed to applications and clearly separated from the heterogeneous protocols, interfaces, and hardware devices that implement it. A novel system component embodies this decoupling by offering a set of agnostic OS features to applications: memory management for zero-copy transfers, asynchronous I/O processing, and efficient packet scheduling. This thesis also explores the design space of the possible implementations of this architecture by proposing two reference middleware systems and by adopting them to support *interactive* use cases in the cloud continuum: a serverless platform and an Industry 4.0 scenario. A detailed discussion and a thorough performance evaluation demonstrate that the proposed architecture is suitable to enable the easy-to-use, flexible integration of modern network acceleration into next-generation cloud platforms.

Acknowledgments

The path to obtain a Ph.D. is deeply influenced by the role of people. Good ideas rarely come up in isolation but rather from the continuous interaction with brilliant and curious minds. I was lucky to find several of those people on my path and some had a prominent role not only in the work presented in this thesis but in all these years of academic life.

The first mention is to my supervisor Antonio Corradi, a constant source of motivation and concrete support from the initial definition of my research topic to the completion of this thesis, from both academic and personal perspectives. Together with Paolo Bellavista, Luca Foschini, and Armir Bujari, Antonio created a young and lively research group, where many ideas presented in this work are constantly discussed and investigated. Within the group, I want to specifically thank my colleague Andrea Garbugli, who has been an endless source of inspiration and companionship. We spent many hours envisioning possible new research directions and much work in this thesis owes something to him, whether it be the material implementation of parts of the INSANE system or assistance with the intricacies of the newest technologies.

At the same time, a long-term collaboration with Ken Birman from Cornell University represented an essential part of my Ph.D. It is hard to overstate the influence that Ken had on my personal and professional growth during these years. He hosted me twice in his research group and has been an incredibly impactful mentor, collaborator, and tireless driving force in my academic life, always providing constructive suggestions even in the most difficult moments. Ken's group also played a key role at the beginning and at the end of my research experience: Sagar Jha shaped my expertise with systems programming, and Weijia Song, alongside his mentoring activity, has always made me feel welcome both from academic and personal perspectives.

I am very grateful to Gianni Antichi and Ana Klimovic for taking the time to review an early draft of this document and providing constructive feedback on how to improve it. I also thank Ana for the availability to host me at ETH Zürich for the last few months of my Ph.D.

A crucial acknowledgment is extended to Beatrice, who has not only patiently stood by my side in every circumstance but has also occasionally contributed to the design of figures, namings, and ideas in this work. Likewise, my family has consistently offered a discreet yet unwavering presence, providing me with the privilege of a secure and stable foundation to pursue my research with complete serenity.

Contents

1 Introduction

In the last two decades, cloud computing has become a cornerstone of modern digital economy, pushing organizations across the globe to radically shift their approach to IT resources, now perceived as *utilities* accessed anytime from anywhere on a *pay-per-use* basis. The success of this model is rooted in the possibility to achieve significant economies of scale. By efficiently spreading the high costs of operations, such as hardware purchase, maintenance, power supply, etc., over a large customer base, cloud providers can offer IT resources *as a service* at competitive prices. Hence, the cloud model represents a cost-effective option for any organization, as it allows developers to focus exclusively on their core business logic and to outsource traditionally burdensome duties such as infrastructure maintenance, resource scaling, and support to continuous availability. Under a technical perspective, two key concepts are among the key pillars of cloud computing: the use of *homogeneous general-purpose* technologies at scale, which allows providers both to maximize their customer base and to reduce costs by purchasing off-the-shelf hardware components, and the *centralization* of those resources in large-scale datacenters, which minimizes maintenance costs and maximizes resource usage efficiency [18, 39].

In parallel, an unprecedented process of digitalization has been reshaping virtually any application domain. The widespread adoption of the Internet of Things (IoT) concept [11] and the exponential growth in the number of connected devices is fueling the *digital transformation* of areas such as automotive and transportation, industrial automation (Industry 4.0), healthcare, telecommunications, tourism, education, entertainment, and many others. At the core of this transformation is the possibility to collect huge volumes of raw data that a new generation of *smart* applications can transform into insightful information, possibly by leveraging the recent, significant advancements in next-generation telecommunication networks (5G and beyond) and in Artificial Intelligence (AI) techniques [14, 20, 76, 92, 119, 123, 96]. Powered by cloud platforms, such information enables innovative processes, services, and products in any industrial and societal sector, such as the concepts of *digital twins* and *smart cities* [69, 128].

In this context, the possibility for devices deployed virtually everywhere to connect to the cloud is enabling a new class of *interactive* cloud applications that hold a high potential for the digital transformation and innovation of companies in any sector, by combining the advantages of the cloud model with the opportunity for customized and real-time decision-making, innovative adaptive services, and advanced forms of automation [21, 113, 119]. However, these applications are also pushing the well-established cloud computing model to show its inherent limitations. These applications are indeed substantially different from those traditionally handled by cloud infrastructures, because they require that events initiated from outside the datacenter trigger cloud-hosted tasks and produce timely answers back to the event source. The production of timely answers to remote events entails demanding system requirements: to move and query a large amount of data (e.g., camera inputs), to perform fairly heavy computations on them (e.g.,

Machine Learning inference models), and to minimize response latencies. Instead, centralized cloud infrastructures are designed and optimized to collect and process huge quantities of data through *offline* batched processing, such as data analytics, and tend to privilege throughput at the expense of tail latency [8, 30, 71]. As a result, the design of many cloud platforms is currently ill-suited to support these new requirements in terms of *online* data-intensive and low-latency responses. For example, many industrial processes require sub-millisecond latencies, whereas a round-trip to the cloud may take tens of milliseconds.

In recent years, new trends emerged to provide the necessary infrastructural support to interactive applications. Among them, two are particularly relevant for this work: on the one hand, the *physical proximity* of resources improves the response times to events [23, 113]; on the other hand, the adoption of modern software and hardware *acceleration technologies* may potentially reduce the computation and communication overhead of cloud services [85, 24, 59, 48]. As we briefly summarize in the following, these trends promote the principles of *resource decentralization* and *hardware specialization*, thus raising questions about the future evolution and economic sustainability of large-scale infrastructures that are based on opposite foundations [121]. To answer these questions, this thesis proposes an architecture for the integration of heterogeneous acceleration technologies *as a Service* into the cloud model and for the portability of interactive applications across the cloud continuum.

A new computing paradigm: the cloud continuum

To support the emerging class of interactive applications, modern cloud infrastructures are expanding beyond their traditional boundaries, by including a hierarchy of virtualized computing resources physically located between traditional cloud datacenters and data sources, according to the idea that physical proximity reduces the communication latency. The resulting computing model is a fluid dissemination of virtualized resources named as *cloud continuum* [113, 115]. In the continuum, providers offer cloud-like features, for example by assigning slices of the resources to different applications, by guaranteeing isolation and by distributing the workload at all levels of the infrastructure [23]. The adoption of a cloud-based model ensures the *portability* and the fluid migration of user-defined applications across the whole continuum, characterized by a higher resource heterogeneity than in centralized datacenters, thus conveying the emerging trend of *resource decentralization* within the standard cloud paradigm.

Modern I/O acceleration technologies

With the end of Moore's law, processor performance increased of just about 3.5% per year in the last eight years. Over the same period, other hardware components significantly improved: for example, the standardized Ethernet link speed increased from 1 Gbps to 1 Tbps [40, 58]. This different performance evolution trend reverses the traditional assumption in computer systems that I/O operations are slower than host processors in moving data: as the standard software stacks available in common operating systems involve the processor in I/O operations, they are becoming the bottleneck of datacenter I/O [17, 29, 67]. As a consequence, datacenter providers are either increasingly offloading I/O operations to *hardware accelerators*, devices that implement in hardware basic functions such as computing (e.g., Graphics Processing Units, GPUs), storage (e.g., Non-Volatile Memory Express, NVMe), and networking (e.g., *smart* Network Interface

Cards, NICs), or to alternative software stacks (e.g., the Data Plane Development Kit, DPDK). By removing the processor from data plane operations (*CPU-bypassing* approach), or at least by reducing its intervention (*kernel-bypassing*), accelerators let applications leverage the full speed of modern hardware, and providers to dedicate a bigger portion of CPUs to user applications.

However, the practical large-scale adoption of these accelerators in cloud platforms still comes with several technical challenges. On the one hand, a lack of support to virtualization and sharing among multiple users makes them *difficult to integrate* within existing cloud infrastructures [24, 28, 59]. On the other hand, they introduce a fundamental problem of *code portability*: these technologies expose deeply different and very low-level programming abstractions and associated interfaces, so that developers must tailor their code to one specific technology, thus harming portability and maintainability, and binding it to specific execution environments [25, 131].

The need for an integrated approach

To support the development, deployment and execution of interactive applications, this thesis proposes to integrate heterogeneous *acceleration technologies* as core components of the *cloud continuum* model. Currently, these aspects have largely been considered separately: research on cloud continuum focused on reducing response latency by moving computation, at least partially, closer to the data sources, whereas a significant body of work investigated techniques to maximize the performance of accelerators for backend services in datacenters. As a result, today user applications in the cloud, from *core* to *edge* platforms, cannot leverage acceleration options even if these are available, because of the *cloud integration* issues of these options previously discussed.

The lack of an integrated approach represents a serious obstacle for the success of the digital transformation in any societal and industrial environment. If applications either in *core* or *edge* datacenters cannot directly access I/O acceleration options *as a service*, they would either rely on legacy alternatives, unable to meet the desired performance requirements, or at least partially reject the cloud model by designing custom hardware and software solutions. That is already happening in *core* clouds: major providers currently offer forms of dedicated physical resources (*bare-metal instances*) with direct access to specific accelerators [6, 13, 53]. This solution is far from ideal and definitely not cost-effective, as it forces providers to manage separate infrastructures and users to pay higher prices to rent these services, thus substantially negating the benefits of the cloud model. Nevertheless, there is a need for heterogeneous, specialized hardware and low latency for applications within the cloud as many applications involve multiple services in the cloud communicating with each other and they need to be power-efficient. This need is even more pressing at the network edge: although the physical proximity to datasources helps reducing response latencies, the overhead introduced by standard general-purpose software and hardware is still unacceptable in many *critical* areas, such as industrial automation or autonomous transportation [50, 105].

This thesis considers three main challenges that currently prevent the integration of acceleration technologies within the cloud model. First, *application portability*: in the cloud continuum, application code should run unmodified across heterogeneous resources, including possibly different acceleration technologies whose interfaces and programming models are very heterogeneous. To tackle this issue, this work investigates the possibility of defining a technology-agnostic interface to provide cloud users with a uniform access point to heterogeneous I/O acceleration options. That possibility would bring significant advantages in terms of code reusability and main-

tainability, as well as ease of development and deployment in the heterogeneous context of the cloud continuum. At the same time, the definition of a uniform interface for high-performance I/O carries several relevant research questions that this thesis discusses in depth. On the one hand, the choice of a certain programming model has a strong impact on both the achievable I/O performance and on the compatibility with legacy applications and the effort required by users to become familiar with it. On the other hand, the definition of a certain abstraction level for that interface entails a trade-off between the simplicity of the exposed API and the degree of visibility and control users have on the underlying I/O technologies. This thesis will propose new options within this space and discuss their advantages and drawbacks.

Another integration challenge is represented by the *special-purpose, low-level abstractions* that acceleration devices expose to user applications. The majority of the I/O acceleration technologies bypass the standard I/O stack for the sake of performance, by offloading it to specialized hardware devices or by letting users provide a more efficient software implementation. However, bypassing the standard general-purpose datapath of current OSes also has several drawbacks: in particular, it forces developers to re-implement from scratch basic OS features such as memory management, thread scheduling, packet scheduling, etc. These requirements make it difficult for inexperienced system developers to efficiently leverage the modern option of I/O accelerations and raise the development and maintenance costs for accelerated solutions tailored to specific application scenarios. Building on a significant body of previous work, this thesis considers the possibility of providing a general-purpose datapath that implements those features on behalf of the users, similar to what OSes offer for standard networking, but designed for high-performance I/O. In particular, this work identifies a set of design principles that all modern acceleration technologies adopt to guarantee high I/O performance. By following these principles, this thesis proposes a new technology-agnostic architecture for general-purpose, high-performance I/O that can be mapped to several heterogeneous acceleration technologies with negligible performance overhead.

The third integration obstacle considered in this thesis is the significant *lack of flexibility* in the acceleration solutions. Cloud computing traditionally relies on a layer of virtualization to decouple the physical resources from the user view. By letting applications directly interact with hardware devices, acceleration technologies tend to bypass even that layer, making it difficult for providers to support typical cloud features such as multi-tenancy, live migration, enforcement of Quality of Service (QoS) and security policies. However, the majority of hardware manufacturers do not currently offer built-in support for these fundamental cloud features (with few notable exceptions [44, 70]). This thesis is among the first attempts to consider the need for virtualization and flexibility when designing a software-based datapath for general-purpose accelerated networking, to minimize the requirements on cloud platforms to support accelerated I/O.

An architecture for Acceleration-as-a-Service in the cloud continuum

This thesis proposes a solution for the integration of heterogeneous acceleration technologies as first-class citizens of cloud infrastructures across the continuum, ranging from standard *core* clouds to *edge* cloud platforms. This solution is based on a novel architecture for *Network Acceleration as a Service* (*NAaaS*) that provides general-purpose, system-level support for the definition of *portable* and *accelerated* cloud applications, specifically targeting the emerging class of interac-

tive applications previously introduced. The architecture is organized in three layers, addressing the three integration challenges defined above.

- The *interface* layer defines a high-level set of primitives that make I/O-accelerated applications *easy to program* while retaining performance efficiency. These primitives must be agnostic to technology-specific details, thus ensuring transparent *code portability*.
- The *system layer* provides a set of technology-agnostic system features for high-performance I/O that are typically bypassed by acceleration technologies, offering them *as a service* to applications. These features include memory management for *zero-copy* data transfers, thread scheduling strategies to efficiently handle asynchronous I/O operations, and packet scheduling policies for traffic prioritization.
- The *plugin layer* specializes for each specific acceleration technology the features defined by the system layer, adopting the necessary optimizations to *maximize performance*. These implementations must be self-contained *plugins*, thus enabling the dynamic attachment of user code to different technologies.

This three-layered architecture effectively decouples the application code from the underlying I/O acceleration technologies and guarantees that the acceleration performance is preserved, thus fulfilling the requirements for *NAaaS*. To demonstrate that, this thesis explores the design space of the possible implementations of this architecture, highlighting the trade-offs that emerge in terms of application portability, ease of programming, performance efficiency, and integration with virtualized environments. To support the discussion, two complete and original *reference systems*, implementing the proposed architecture, will be introduced: a data distribution middleware and a userspace OS module. These systems are designed to offer one or more network acceleration techniques (RDMA, DPDK) as a service to applications in cloud environments. The choice of these systems is motivated by several use cases of *interactive* applications in heterogeneous domains. In particular, this thesis will show how these systems can be effective in supporting *NAaaS* for two reference scenarios: a platform for *serverless computing* and a framework to support virtual Programmable Logic Controllers (vPLCs), key components in the Industry 4.0 revolution.

In the last part of the thesis, an extensive quantitative evaluation of the two reference systems, first in isolation, then in comparison with existing alternatives, and finally within the two proposed use cases, will prove that the proposed architecture results in superior performance, better portability, and full integration with existing cloud platforms and tools.

Thesis structure and organization

The remainder of the thesis is organized according to the following structure. Chapter 2 provides a more detailed definition of the concept of *cloud continuum* and motivates why and in which scenarios it is reasonable to consider that network acceleration options are available even outside the traditional large-scale cloud datacenters. To this end, two application scenarios are introduced: the emerging concept of *serverless computing* and the rapidly developing idea of Industry 4.0. Chapter 3 gives the necessary background about network acceleration technologies, with a specific focus on both the difficulty of using them in general-purpose programs, and the

complexity of their integration into existing software stacks. To demonstrate that, the Chapter reports the experience of optimizing a complex application, Derecho [63] (a library for State Machine Replication), to be used with RDMA, proving that only a careful design allows to properly leverage the potential benefits of that acceleration option.

Chapter 4 defines the concept of *Network Acceleration as a Service* and discusses the significant body of previous work that paved the way for the contributions of this work. Chapter 5 introduces the architecture proposed by this thesis, discussing the role and responsibility of each layer, and the trade-offs that might emerge while implementing them. Then, Chapter 6 proposes two complete reference implementations of the architecture, guided by the requirements of different application domains. These systems are first described in isolation, and then used to support a more complex and domain-specific scenarios where *interactive* applications are needed. Chapter 7 reports the results of a thorough quantitative evaluation of those systems, discussing the advantages and the open challenges related to the adoption of the proposed architecture in the selected application scenarios. Finally, Chapter 8 concludes this thesis by providing insights into future research directions.

2 The Cloud Continuum

This Chapter briefly reviews the evolution of the cloud computing paradigm toward a more distributed model called cloud continuum, which embraces the recent trend of *resource decentralization*. In particular, the discussion will focus on the challenges that arise from the introduction of virtualization in application scenarios characterized by a high heterogeneity of the available resources and of the application requirements. As these challenges emerge specifically in connection with the networking infrastructure, the main techniques currently used for *I/O and network virtualization* are briefly introduced, showing how these might add significant latency overhead.

The last part of the Chapter introduces two scenarios in which interactive applications are enabled by the cloud continuum. On the one hand, the emerging *serverless computing* service model allows users to easily deploy fine-grained, event-triggered function pipelines across the continuum, thus allowing developers in several domains to easily define even complex applications. On the other hand, the possibility to deploy cloud services in relatively powerful edge cloud platforms is driving the digital transition of many industrial areas, a process sometimes referred as *Industry 4.0*: the physical proximity of cloud resources to industrial equipment allows companies to replace specialized technologies with general-purpose tools while still fulfilling their performance requirements. Overall, these will be the reference scenarios for the implementation of the architecture proposed in this thesis.

2.1 Introduction to Cloud Computing

Since its definition, the cloud computing paradigm gained wide popularity in virtually any economical and social sector. The key reason of that success resides in the availability of computation, networking, and storage resources *as a service*, accessible anytime and anywhere, so that companies in any economical sector no longer need to buy and maintain their own on-premise IT infrastructure. Cloud providers transparently manage the physical infrastructure, billing users for their real resource usage, e.g., charging per time unit, number of requests to a service, or amount of transferred data (*pay-per-use* model). Thus, companies can elastically scale resources based on the actual demand, saving upfront costs and avoiding to pay for idle machines under low traffic, but still being able to respond to peaks of demand. Depending on the agreement between users and providers (Service Level Agreement, SLA), different kinds of cloud resource offerings are possible: a widely popular classification distinguishes between Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) depending on the user visibility of the underlying computing resources [80]. Although this categorization no longer captures the whole spectrum of available offerings, it is still considered an important reference [36].

This thesis focuses on the IaaS model, where users obtain exclusive access to VMs or containers. Users get access to all the machine resources, their operating systems and applications, whereas

the provider keeps control on the underlying physical cloud infrastructure. Indeed, *resource virtualization* is a key pillar of the cloud computing model, as it decouples the users perspective of working on dedicated resources from the provider physical view and plays a crucial role in the dynamicity that characterizes any cloud offering. For example, it enables the elastic scaling of resources in response to the current load of users, saving them significant costs compared to on-premise approaches. Virtualization also enables *multi-tenancy*: the same physical resources can be allocated to different users, even belonging to different organizations. Because providers can optimize the use of their equipment, they can also offer a cheaper service: multi-tenant clouds generally represent the most economically appealing solution for most enterprises.

2.2 The Cloud Continuum

The success of the Internet of Things (IoT) concept and its widespread adoption across various application domains are driving an evolution in the cloud computing paradigm. With the pervasive availability of connected devices, there is an increasing demand for applications to consume, analyze, and generate diverse data from a variety of sources. Centralized cloud infrastructures can only partially meet these demands and new computing infrastructures, able to host applications at edge devices, have started to appear in recent years, improving aspects such as response time and reducing bandwidth use. For example, the concepts of *Mobile Cloud Computing* (MCC) [34] and *Fog Computing* [23, 26] have promoted the offloading of compute-intensive tasks to cloud services running on the telco infrastructure, which is increasingly able to host general-purpose virtualized application components [1]. Concurrently, the success of the *Edge Computing* [115] has demonstrated that deploying resources co-located with data sources can be crucial to allow cloud services to meet key performance targets.

Combining the ability of running performance-sensitive, localized applications both at the edge and within the telco infrastructure with the high-capacity from the cloud, the *Cloud Continuum* has emerged as a paradigm that can support heterogeneous requirements of small and large applications through multiple layers of a computational infrastructure that combines resources from the edge of the network as well as from the cloud [113]. The resulting infrastructural model is a composition of edge, fog, and cloud to support IoT-based applications, thus constituting a three-tiered hierarchical infrastructure that is illustrated in Fig. 2.1 and discussed in the following.

- *Cloud*. Centralized *core* datacenters host large-scale computing capacity in buildings specially designed to host them, deployed in few locations because of special infrastructure requirements, such as power, space, cooling, as well as the need for specialized workforce and the associated cost management. Cloud providers offer users virtually infinite resources through a pay-per-use model, as described in Section 2.1.

- *Fog*. The fog infrastructure, in turn, is organized in a hierarchy of resources, sometimes called *fog nodes*, spanning among the edge of the network and the cloud datacenters. Although the density of fog nodes or the number of layers may vary depending on the location, the lower fog layer is typically considered to be one hop away from the edge layer (e.g., WiFi or cell phone antennas). Additional layers may enhance the computing capacity of the fog tier and ease data processing or movement: generally, the lower devices are in this hierarchy, the lower is the re-

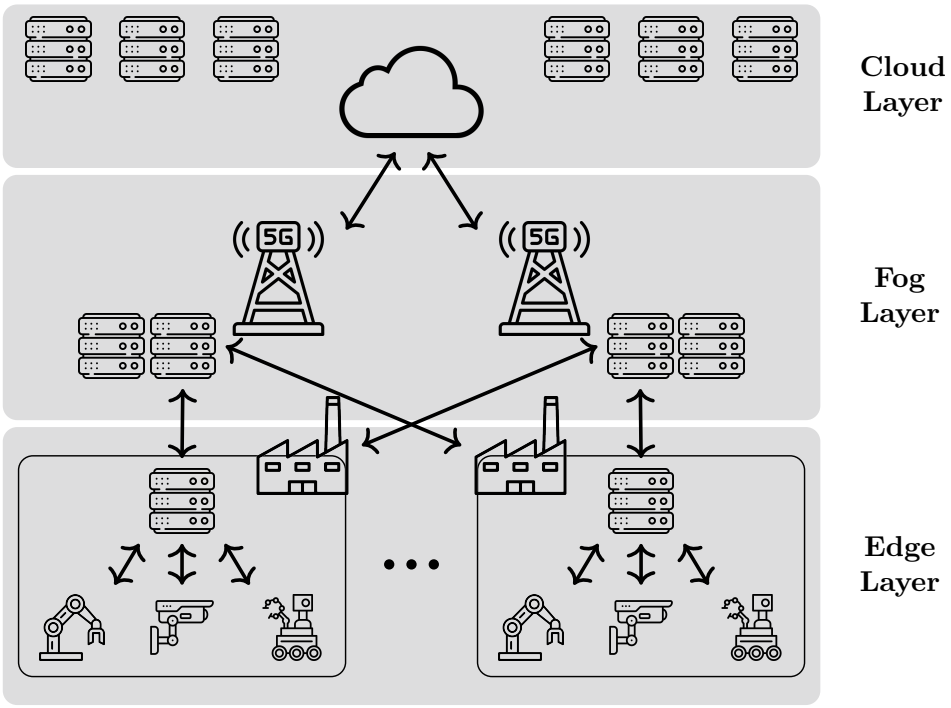


Figure 2.1: The three-tiered hierarchical infrastructure of the cloud continuum.

sponse latency they can provide, but also the smaller their computing capacity. The telco Mobile Edge Computing (MEC) model is a clear example of this infrastructure: the availability of powerful general-purpose servers close to antennas and base stations not only support the most modern cellular standards (5G and beyond), but can potentially host any kind of service that is offloaded from user end-devices [1].

- *Edge*. The edge layer corresponds to the geographically distributed locations where end-devices are deployed and usually accommodates the most latency-sensitive applications. Depending on the considered scenario, this layer may correspond directly to the set of end-devices that seek cloud connectivity (e.g., mobile phones, smart cameras, etc.) or to a more structured environment able to host even powerful resources in local small-scale datacenters. In the first case, devices equipped with a small amount of computing resources can directly pre-compute or pre-process raw data for applications to avoid soaking the cloud. In the second case, instead, the concept of *edge* datacenter emerges to place cloud service instances directly one network hop away from large device deployments. Section 2.2.1 will discuss more about the latter scenario.

Overall, these layers are characterized by a high degree of heterogeneity. On the computing side, several devices with very different capabilities are involved at each layer, ranging from powerful servers to small devices with only pre-processing capabilities. Under a networking perspective, data must generally cross one or more tiers of the Edge-Fog-Cloud infrastructure, potentially involving different connection technologies: for instance, mobile devices will typically adopt wireless connections to communicate with a fog node, whereas monitoring sensors within a factory

will likely employ wired connections. As a consequence, different communication protocols are involved, and different communication links will provide different properties in terms of mobility and performance.

The hierarchical model of the cloud continuum is very effective to support applications that have very different QoS requirements, as their components can be distributed among devices at different levels of the Edge-Fog-Cloud infrastructure depending on the application needs (e.g., latency, computing capacity, data locality). However, such intrinsic heterogeneity poses a significant problem of *code portability* and maintainability: different hardware, software stacks, and communication protocols require to tailor applications for a specific deployment environment, making it complex to dynamically leverage the flexibility of the Edge-Fog-Cloud continuum.

For that reason, a key idea in the cloud continuum paradigm is to rely on a *layer of virtualization* that, like in cloud datacenters, decouples physical resources from the application code. The resulting computing model is a continuum of virtualized resources offered as a service that enable the dynamic migration of application components across the different layers, thus overcoming the issues arising from the high degree of heterogeneity of the continuum. Across the continuum, providers can thus operate according to a cloud-like model, for example by assigning slices of the resources to different tenants, by guaranteeing isolation, and by distributing the workload at all levels of the infrastructure. As a result, it is possible to flexibly and dynamically support heterogeneous QoS requirements and to significantly improving response times and service interactivity for performance-critical components [23, 113].

Performance Challenges in the Cloud Continuum

Although resource virtualization is the key of the success of the cloud model, even across the Edge-Fog-Cloud continuum, it can also become an obstacle when users try to deploy the most performance-critical applications. As discussed in Chapter 1, a founding principle of the cloud model is to provide a general-purpose platform to support any application in any domain, thus enlarging the number of potential customers and making the model economically sustainable for both customers and providers. The need to provide generality through virtualization necessarily leads to the introduction of various software layers between user applications and the underlying infrastructure. However, in the most performance-critical scenarios, the performance overhead introduced by these layers can significantly impact the possibility of applications to meet their performance requirements, even when cloud services are in execution at the Edge layer. We will discuss these issues extensively in Chapter 3, observing how the networking infrastructure is generally the most affected by these overhead even in those scenarios where low network latencies are crucial to ensure the correct operations of applications. Preliminary to that is the discussion in Section 2.3 about the main techniques to virtualize networking in cloud infrastructures.

This thesis will mainly target the traditional *Cloud* and the *Edge* layers of the the Edge-Fog-Cloud infrastructure. In particular, this thesis will consider scenarios where the edge infrastructure is composed by large deployments of devices, such as wired sensors and actuators, directly connected to a local *edge* datacenter, which we will better define in the next Section. The interplay between these two layers is indeed crucial for the application domains considered in this work, and the availability of fairly powerful resources at both these locations make it possible to design cloud-enabled support systems that can provide both flexibility and high-performance.

2.2.1 The cloud edge datacenter

Whereas the concept of *core cloud* datacenter is well known in literature, the more recent idea of *edge cloud* is still not widely recognized. In this thesis, with the term *edge cloud* we refer to small-scale computing environments deployed in the same location as edge devices (e.g., IoT devices) but managed as full-fledged cloud platforms. The kind of resources available in these scenarios are comparable to those in core clouds, although at a smaller-scale, powerful enough to run fairly heavy workloads and serve as a first hop to interact with the smaller devices, thus ensuring minimal response latencies from local instances of critical services.

Besides the different scale, a key feature of edge cloud platforms is a much higher degree of resource heterogeneity than core cloud datacenters. The latter are indeed few centralized infrastructures, usually managed by a single providers and thus mostly equipped with homogeneous hardware and software. The scattered geographical distribution of edge clouds, their higher number, and their management by several different actors make them quite different from each other, also depending on the specific domain they must serve. Nevertheless, these details are generally hidden to the end user by a virtualization layer which ensures the portability of the code and its management through usual cloud tools, such as orchestrators.

In recent years, edge cloud platforms have started to appear in several diverse application domains. The most well-known example in literature, although properly belonging to the *Fog* layer, is the telco Multi-access Edge Computing (MEC) [1]. The MEC concept consists in the positioning of general-purpose powerful devices co-located with antennas and base stations in order to deploy cloud services that can gain contextual information and real-time awareness of their local environment. More recently, the process of industrial digital transformation known as Industry 4.0 has promoted the deployment of proper small-scale datacenters, such as factory-local server racks, to improve cloud service responsiveness [50, 105]. In the transportation industry, driven by new applications of autonomous use cases, the need for local high processing power is transforming vehicles (including cars, trains, planes, etc.) into high-tech units where typical datacenter technologies are required [86].

2.3 Networking in the Cloud Continuum

From a networking perspective, virtualization is critical to support the communication patterns of modern cloud applications, independently of their deployment position in the cloud continuum hierarchy. Cloud users tend to instantiate multiple VMs to host their application components, which reciprocally communicate. To support this pattern, cloud providers must enforce two forms of virtualization: not only an efficient network access (*I/O virtualization*), but also virtual private overlay networks among machines (*network virtualization*). There are multiple technical challenges associated with those two kinds of virtualization: in particular, which *interface* customers should use to access the network and which *virtualization techniques* allow VMs of the same tenant to efficiently communicate while preserving their isolation.

Although these aspects are certainly not the only concerns for cloud providers, according to the scientific literature they are today the most clear example of the limits of the cloud model, as the overhead introduced by these forms virtualization is emerging as the main networking bottleneck. To help the reader understand the motivation for the alternative solutions introduced in

the next chapters, this Section provides an overview of the current standard approaches to these fundamental cloud networking aspects (network access interfaces, virtualization techniques, serviceability). Then, Chapter 3 will provide a brief background on the emerging alternative options to these approaches, and Chapter 5 will introduce an architecture to integrate them within the cloud model, to support *Network Acceleration as a Service* (*NAaaS*) for faster cloud networking.

2.3.1 Network Access Interface

The IaaS model provides users with the ability to obtain the same environment they would have on a bare-metal cluster, by accessing instances of VMs or containers. Under a networking perspective, this model brings several advantages, particularly the capability to run unmodified application binaries within the virtualized environment, which enables users to take advantage of virtualization without any change to their existing applications. At the same time, cloud developers can create new applications without the burden of learning and understanding new frameworks, programming languages, or interfaces, as they can continue to use the same tools they are familiar with. Such flexibility stems from the availability of *virtual* network interfaces within VMs and containers, which cloud users may access by using standard system and programming tools: in particular, applications can use the standard and ubiquitous *POSIX Socket API* to communicate over the network. Unfortunately, the ease of programming guaranteed by this standard interface comes at the price of multiple copies of the payload data. Modern options for high-performance networking base their better performance on *zero-copy* data transfers, but trading portability and ease of programming for performance efficiency (see Chapter 3).

2.3.2 Virtualization techniques

Resource virtualization is a fundamental principle across the whole cloud continuum infrastructure. From a networking perspective, we already distinguished between two kinds of virtualization, *I/O virtualization* and *network virtualization*, according to previous literature on this topic [57, 94]. Although those two aspects are closely intertwined, the former case refers to the mechanisms to enable Virtual Machines (VMs) or containers running on a shared physical host to access an external network. Instead, the latter case considers the techniques to create virtual private overlay networks among a set of VMs or containers belonging to the same users or tenants. Both these aspects are crucial for cloud networking as they influence the performance, flexibility, and isolation properties of communication. The following paragraphs briefly discuss the standard approaches typically used by cloud providers to enforce those two forms of virtualization [49, 50, 57, 68, 74, 109, 124].

I/O virtualization

A critical challenge for applications running in virtualized environments is to efficiently access I/O devices, especially when network performance is a critical concern. The techniques to obtain an efficient end-host network performance differ for VMs and containers. For VMs, the most prominent approaches are direct device assignment and paravirtualization, represented in Fig. 2.2 left and center.

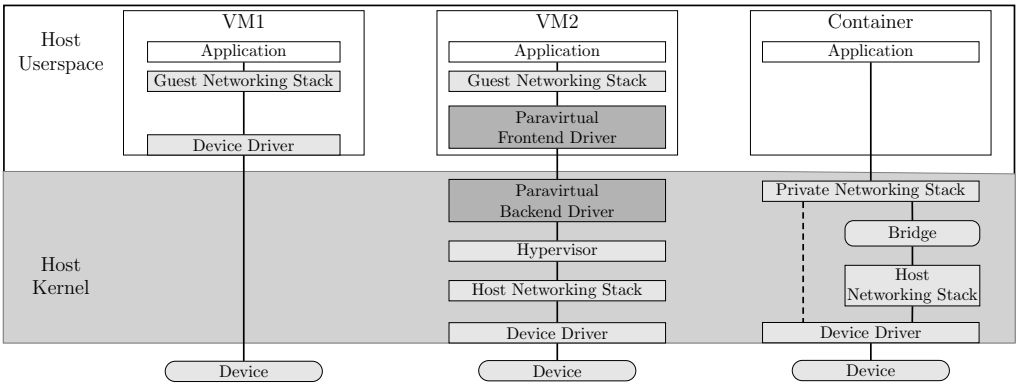


Figure 2.2: The two prominent techniques for I/O virtualization for VM: direct device assignment (left) and paravirtualization (center). On the right, two modes for container networking: host (dashed line) and overlay (solid line).

Direct device assignment reserves a device instance exclusively to a VM or container (*passthrough*), so each virtual environment (VM or container) requires a distinct physical network adapter. To mitigate this heavy scalability limit, recent devices support a form of *hardware-assisted virtualization* called Single Root IO Virtualization (SR-IOV [91]) that makes them appear as multiple separate devices called Virtual Functions (VFs). Each VF can be assigned to different VMs as if it were a distinct device. Either way, direct device assignment allows to exclude the hypervisor from the network critical path: virtualized applications can access the network as if they were physical hosts, thus achieving the best network performance. However, this technique tightly couples network devices and virtual environments, strongly limiting the inherent flexibility of virtualization: for example, live migration becomes impossible to support, because the hypervisor cannot create a snapshot of the network state [95].

The paravirtualization technique, instead, splits the device driver into a *frontend driver*, located in the guest OS of a VM, and a *backend driver* on the host (Fig. 2.2), where those two drivers exchange commands through a dedicated communication channel. This separation lets the hypervisor in full control over the network control and data planes, thus providing a high degree of flexibility: because traffic is mediated by software, it can be easily controlled. However, this also introduces overhead on data path operations, especially when crossing the guest/host boundaries. The virtio [109] framework is the *de facto* standard tool for paravirtualization, and it allows the hypervisor to expose paravirtualized devices to the guests. To mitigate the performance overhead introduced by paravirtualization, virtio clearly separates the data plane, which handles the actual network traffic between the host and the guest, and the control plane, which allows to exchange control messages about the data plane. The data plane is implemented as a set of shared memory areas, called *virtqueues*, between the frontend driver on the guest and the backend driver on the host. Those memory regions are managed as couples of ring buffers holding the network data to be received and transmitted, similarly to the actual queues of physical network devices. Each virtual device can have zero or more queues associated, with the limitation that each queue must have associated a distinct vCPU. Conversely, the control plane consists of a notification mechanism used between the frontend and the backend driver to discover and signal new data in the

queues. For network devices, that notification mechanism is implemented as a direct inter-process communication channel between the two drivers.

The paravirtualization technique is popular for VMs, but it introduces an overhead that is often unacceptable in the case of containers. Indeed, containers are considered a lightweight form of virtualization compared to virtual machines, as they let applications execute directly on the host operating systems [98]. Various mechanisms are used to isolate containerized applications from the host; from the networking perspective, each container is generally assigned a *network namespace*, which includes a separate instance of the kernel networking stack (Fig. 2.2 right). Hence, it is possible to create interfaces directly in that namespace, without resorting to virtio, to assign arbitrary IP addresses, and to use the same device drivers installed in the host. In *host* mode, data are forwarded directly to the device driver, so the network configuration is the same as bare-metal applications (dashed line in Fig. 2.2). Otherwise, data can be directed by using a software-based datapath to a software bridge (*overlay* mode) and then forwarded to the host networking stack for the actual network access (solid line).

Network virtualization

In case of I/O direct device assignment for VMs (or *host* networking mode for containers), the traffic of a virtualized application appears on the network as the traffic of a bare-metal application on a physical host. This option might make sense in a dedicated infrastructure, such as a private cloud or on-premise infrastructure, but is not suitable for multi-tenant scenarios, where traffic isolation among tenants is paramount. Instead, generally providers adopt a paravirtualization approach (for VMs) or the *overlay* networking mode for containers, with the goal of segregating user traffic within controlled boundaries (*network virtualization*). The key technology to enable network virtualization is the *virtual switch*, a software module, traditionally located in the host kernel, that acts as packet dispatcher among network interfaces. Once the traffic of a virtualized application reaches the backend driver (in case of VMs, see Fig. 2.2 center) or exits the network namespace (in case of containers, see Fig. 2.2 right), the switch forwards it on the physical network. In the cloud context, virtual switches represent the first network hop for user applications. Hence, they are the main tool for providers to build logically isolated, virtual private overlay networks to connect user application components, or to implement customer-supplied network spaces. Importantly, the software flexibility allows cloud providers to offer a richer network semantics, by configuring traffic shaping policies and dynamically adapting such configuration to external events, e.g., mutated network conditions, new policies to enforce, or user requests. With software switches on each host, providers can easily scale such network control actions to a high number of servers, while at the same time keeping the actual physical network simple, scalable, and thus very fast. Important traffic shaping policies include network isolation of VMs and containers via different forms of tunneling, such as VXLAN [77], security, migration, QoS enforcement, and generally all the *observability* aspects that providers adopt to monitor their infrastructure. For instance, to leverage this flexibility as much as possible, major cloud providers design their own software switches [43], although open-source versions are widely available (e.g., Open Virtual Switch [89]).

2.4 Application scenarios

This last Section introduces two scenarios, *serverless computing* and *Industry 4.0*, in which the availability of the cloud continuum infrastructure potentially enables companies to bring the flexibility of cloud applications even in domains characterized by demanding performance requirements. These scenarios are particularly interesting also because they offer examples of the challenges that still need to be addressed in the cloud continuum: although the literature in this field tends to trade performance for the flexibility of resource virtualization, sometimes that is not practically viable. Hence, new forms of infrastructural support should be adopted to further reduce the performance footprint of resource virtualization, especially in terms of network latency.

Depending on the performance requirements, applications in both areas might be deployed across the cloud continuum: in the core cloud, if more processing power is needed, or in the edge cloud, when real-time constraints are paramount. When considering the edge cloud, this thesis will focus on deployments in which devices are connected with cloud platforms through a wired link. Although wireless connectivity is critical in many use cases, such as those with mobility involved, the scenarios characterized by wired connections are typically the most demanding on the supporting cloud infrastructure, as even µs-scale networking overhead become critical for the correct application behavior, just like in large-scale datacenters [17].

2.4.1 Serverless computing

Serverless computing is an emerging proposition in the cloud offering landscape that promotes a higher level programming abstraction, further decoupling application code from the underlying system and hardware infrastructure. Following a wider trend of resource disaggregation, the serverless model relies on the execution of short-lived stateless *functions* in response to events, which can be internal or external to the execution platforms. In cloud platforms, serverless computing was mainly introduced as *Function as a Service* (FaaS), a service model which tasks customers only with the creation of the business logic of functions and transparently offloads any other aspect to the platform, including scaling, deployment, monitoring, security [15, 99]. Major cloud providers have embraced this paradigm and already provide managed FaaS offerings, and many open-source projects are under active development as well [116].

The unit of execution in FaaS consists of a stateless function that is instantiated and executed in response to an incoming event. This capability allows for fine-grained resource control and elastic auto-scaling, which are appealing advantages for a broad range of applications including IoT, interactive data analytics, and even ML tasks. For instance, the possibility to spawn a high number of fine-grained short-lived functions might significantly reduce both the costs and the response latency of highly parallel jobs, such as stream processing [45]. Hence, this model is potentially well-suited for the cloud continuum: because all the infrastructural aspects are offloaded to the providers, user functions might in principle be executed at any layer of the IoT-Fog-Cloud infrastructure, depending on the performance requirements.

One novel concept rapidly developing across these platforms is the capability of composing functions to create tailored processing pipelines. By decoupling complex functionalities into simpler ones, function composition encourages modularity and promote function reusability, thus further reducing the development effort and also the time-to-market of even complex applica-

tions. Many interactive IoT applications in the cloud continuum can be easily expressed in terms of function pipelines: for instance, a traffic monitoring applications would have a first ingestion layer, which receives real-time video streams from monitoring cameras, a second layer that tracks the objects (vehicles) visible in the images, a third layer that predicts the trajectory of each object and detects potential collisions, and a final stage that outputs the results whether in local storage or as a warning back to the camera location. All these layers can be easily obtained as off-the-shelf *functions* that, combined together, create a non-trivial intelligent and reactive application [119]. Furthermore, by deploying the pipeline as close as possible to the event source, providers could minimize its response latency.

Despite its great advantages under many aspects, FaaS support in cloud platforms across the continuum still faces several challenges. Among the most relevant for this thesis, the ephemeral nature of the functions poses several issues when fine-grained state sharing needs arise, demanding efficient, scalable and cost-effective forms of I/O management. That includes both storage and networking primitives, such as broadcast, aggregation and shuffling [90]. This aspect is particularly relevant when considering function pipelines that must respect some form of time constraint. Indeed, current function composition solutions exhibit some performance issues: response latencies can materialize not only from a bad user-defined chaining logic but also from inefficient infrastructural support to function composition [16]. At the same time, most FaaS solutions are not optimized to handle bursts of short-lived functions, an inherent property of this increasingly popular approach, that can amplify the overhead in the function invocation path [126]. In addition, no current FaaS platform adopts resource-aware optimizations to exploit the specificity of the underlying hardware and software resources, a paramount feature in the context of cloud continuum [112, 117].

This thesis will investigate the challenges related to the efficient I/O operations of pipelines of functions in FaaS platforms. In particular, Section 6.2.3 will introduce DIFFUSE, a middleware solution based on shared memory that acts as a conveyor of data to improve function-to-function communication performance. The implementation of this scheme is totally transparent to the end users, but it can opportunistically leverage modern network software and hardware available at any deployment site across the cloud continuum, thus enabling, when possible, high-performance zero-copy communication between functions (see Chapter 6).

2.4.2 Industry 4.0 and beyond

The term Industry 4.0 refers to the digital transformation of industrial processes, especially in manufacturing and automation industries, through the integration or the replacement of traditional specialized Operation Technologies (OT) with general-purpose Information Technologies (IT). The goal of this transition, also known as the *fourth* industrial revolution, is the creation of *smart factories*, through which companies expand their operations beyond local and limited environments to a broader, global, and interconnected industrial sector. Machines continuously generate and export data that are filtered, processed, and analyzed in near real-time to extract business insights and facilitate accurate and cost-effective decision-making. Thanks to the introduction of the Industrial Internet of Things (IIoT), sensors and software are embedded in smaller and smarter connected devices that allow *cloud-native* communications and *immediate* actions on the surrounding environment [27, 119].

In this context, the increasing amount of raw data produced by machinery and the necessity of analyzing them is rapidly pushing companies to replace or adapt machine field technologies from proprietary ad hoc industrial protocols to open and more flexible standards [46, 88]. Industries have a real opportunity to enhance the automation level and the cohesion between OT and IT in a cost-effective and affordable manner by utilizing Commercial-off-the-Shelf (COTS) hardware and software. This has several benefits: increased community support, reduced maintenance effort, continuous updates, and improved cybersecurity.

However, despite the several advantages of such integration, the practical replacement of specialized technologies, such as embedded logic controllers (PLCs), with IT-enabled components is still difficult to achieve: OT were specifically designed to support demanding requirements in terms of latency, jitter, and Quality of Service (QoS), whereas IT for best-effort behavior and a general-purpose use. As a result, even when cloud-native solutions, such as virtualized controllers (vPLCs) are deployed in edge clouds, they cannot always offer the deterministic behavior and low network latency required by traditional specialized solutions, or do so by sacrificing the generality of IT. This thesis will consider various use cases in the context of Industry 4.0 and show that, by integrating the option for *Network Acceleration as a Service*, it is possible for cloud services to meet even the most demanding constraints of industrial applications.

2.5 Conclusion

The increasing number of connected (Industrial) Internet of Things (IIoT) devices has fueled a trend of *decentralization* of computing resources from traditional centralized datacenters to locations closer to data sources, with many advantages such as a better service responsiveness and enhanced automation at the network edge. In recent years, the cloud computing paradigm has evolved to embrace that trend. The concept of *cloud continuum* promotes the management of devices, distributed even outside datacenters, as a continuum of virtualized resources, where user code can be dynamically deployed depending on application requirements. However, the introduction of virtualization is not always a panacea: under a networking perspective, traditional virtualization techniques can introduce additional performance penalties. Significant examples of these challenges arise in the reference scenarios introduced in this Chapter: *serverless computing* and Industry 4.0.

The next Chapter introduces a set of *I/O acceleration technologies* that have proved successful as high-performance alternatives to the standard I/O software stacks. Although these technologies hold the potential to mitigate the performance overhead introduced by virtualization, especially at the network edge, they are also quite hard to integrate within cloud platforms. The architecture introduced in Chapter 5 aims at making such integration possible, thus offering the long-sought option for *Network Acceleration as a Service* to cloud developers.

3 End-Host Network Acceleration

System developers are increasingly adopting modern hardware and software technologies that accelerate the I/O operations of their applications. The need for I/O acceleration is driven by a significant shift in the workloads that general-purpose infrastructures need to support: from offline computations on large batches of data to online and possibly intelligent responses to events. These workloads are extremely demanding in terms of resources and executing them on general-purpose CPUs would not be as efficient as adopting *specialized technologies* to support them, such as Graphics Processing Units (GPUs), Non-Volatile Memory Express (NVMe), or Data Processing Units (DPUs).

This Chapter provides the necessary technical background to understand the potential benefits, the challenges, and the trade-offs associated with the use of I/O accelerators, with specific focus on end-host networking, as motivated in the Introduction. Rather than delving into complex technical details, this Chapter aims at presenting the *common design principles* behind different network acceleration technologies (RDMA, DPDK, XDP, etc.), which are all based on the fundamental concept of clear separation between the control and the data planes of communication [93, 19].

Although all these options share the same design principles and goals, each technology implements them in very different ways. This heterogeneity, combined with a programming model completely different from the standard POSIX interfaces, makes it difficult for inexperienced users to leverage these options and requires developers to carefully tailor their code for a specific technology. As an example of this complexity, the Section 3.5 reports the experience of the optimization of Derecho [63], a complex system library for State Machine Replication. By carefully re-designing the internal protocols, it was possible to obtain a 10x improvement in the throughput of the system for small messages, but at the price of a careful, low-level, and time-consuming process of code tailoring to properly leverage the high-performance capabilities of RDMA [62].

3.1 Design principles

With the end of Moore's law, processor performance increased of just about 3.5% per year in the last eight years, in sharp contrast with the growth rate of communication links in datacenters and the rapid standardization of higher Ethernet network link speeds [40, 58]. Fig. 3.1 shows this evolution of over the past 25 years. This different performance evolution trend reverses the traditional assumption in computer systems that network operations are slower than host processors in processing packets: because the CPU is involved in the data processing pipeline, the standard networking stack available in common operating systems is becoming the bottleneck of datacenter networking [17, 29, 67]. At the same time, the increasing amount of CPU cycles spent for high-performance networking is subtracted to user applications, i.e., to the core business of cloud

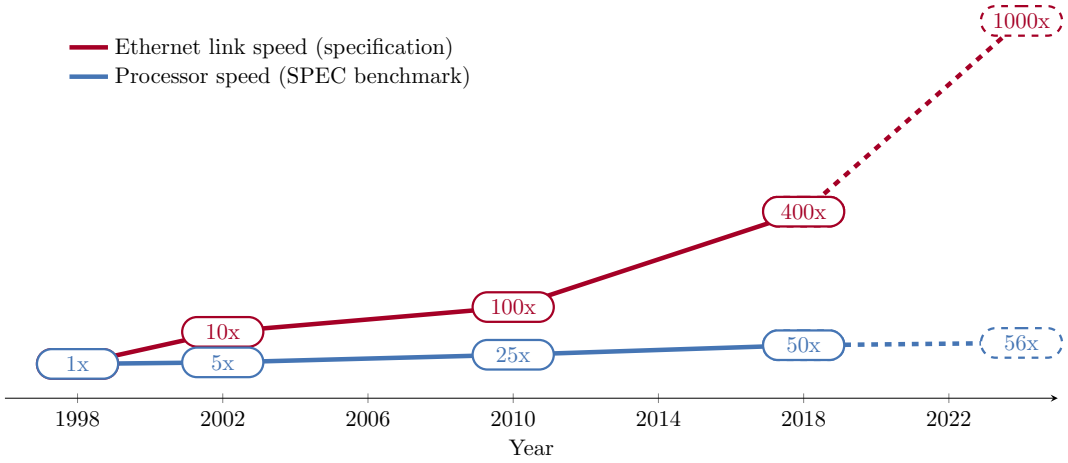


Figure 3.1: A comparison between the performance improvement of Ethernet link speeds and processor speeds in the last 25 years.

and datacenter providers. For example, in a typical long-lived TCP flow between two hosts, more than 50% of the total CPU cycles are spent for data copies at the end hosts [29]. For an average request, a server running a simple key-value store spends only 6% of the total CPU cycles within the application logic, whereas 85% is spent within the kernel networking stack [67].

Based on these considerations, network acceleration technologies tend to improve end-host networking by minimizing the processor intervention on the data plane. This idea belongs to an overarching trend that reflects a shifting balance that many recent researchers have highlighted: modern networking links are so fast that to utilize its full potential, developers must clearly separate the *control plane*, which expresses the application logic and thus entails CPU intervention, from the *data plane*, where data must be free to move at the link speed [19, 93]. This way, applications can leverage the full speed of modern communication links, and datacenter providers dedicate a bigger portion of their CPUs to user applications.

The clear separation of control and data planes to minimize the CPU intervention in data transfers has important consequences on the design of end-host high-performance I/O stacks. It is possible to identify three key *design principles* that modern network acceleration technologies follows to guarantee such separation, as discussed in the following.

- **Zero-copy Data Transfers**. Memory copy operations are by far the most significant bottleneck for end-host networking [29, 67]. Ideally, a complete separation between control and data planes would require a model in which any host involved in the communication has a single copy of any given data item at a certain memory location: the network equipment places incoming data at in a designed area (or reads them from it), from which data is never moved. The control plane refers to the items by sharing references to that location. Although maintaining just a single copy of each data item might not be always practically viable for many reasons (isolation, security, etc.), high-performance network protocols should strive to minimize their number.

- **Minimal Context Switching**. Context switches between processes, especially at the boundary between userspace and the kernel, use precious CPU cycles. Hence, it should be avoided to perform protocol processing and other data plane actions in the kernel, as that involves passing controls multiple times between user and kernel processes. Ideally, data plane operations should be performed by a single process or thread in userspace, thus also improving data and instruction cache locality, or, even better, completely offloaded to hardware.

- **Asynchronous Processing**. Because control and data plane execute at very different speeds, network interfaces and protocols should be ideally designed to be completely asynchronous and lock-free, thus avoiding that control plane stalls data plane, or vice versa. That is practically not always possible, for instance when parallel network processing is performed on multi-core processors, but the more this separation is enforced, the better network performance will be.

As shown in Fig. 3.2, these principles are quite a disruptive departure from the design of standard end-host networking, as they are based on opposite assumptions: standard interfaces (e.g., POSIX sockets) and protocol stacks (e.g., kernel TCP/IP) impose multiple copies of payload data, are kernel-based, and synchronous by default. Instead, modern high-performance networking techniques force a redesign of the systems and applications that want to leverage their advantages, as Section 3.5 will show with an example. That is not only an issue for end-users, which should refactor their applications, but also for cloud providers. Cloud platforms are indeed currently not ready to accommodate this kind of acceleration: in cloud computing, physical resources are hidden behind a layer of virtualization, whereas acceleration technologies remove even the operating system layer between applications and the hardware. Chapter 5 will discuss that in more detail, by introducing a new architecture that designed to make this integration much easier.

To make the picture even more complex, not only network acceleration technologies follow opposite design principles compared to the standard ones, but they implement these principles in ways that substantially differ across technologies, in terms of API, resource usage, hardware requirements, and performance efficiency. Such diversity reflects the original specific purposes they were built for, but as they are increasingly adopted as general-purpose options, it practically be-

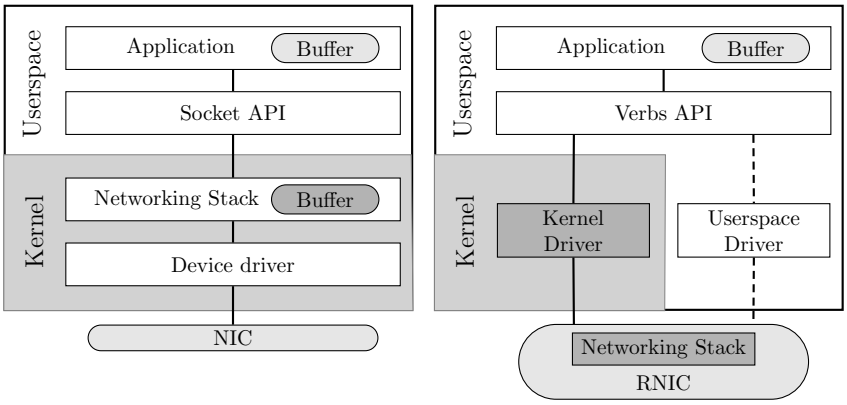


Figure 3.2: Standard kernel-based (left) vs accelerated network stack (right) of a hardware-based technology, RDMA. RDMA clearly separates the control path (solid line) from the data path (dashed line).

Technology	Kernel integration	API	Zero-copy	CPU intervention
Kernel TCP/IP	In-kernel	`AF_INET` Socket	No	Per-packet
XDP	In-kernel	`AF_XDP` Socket	Yes	Per-packet
DPDK	Kernel-bypassing	RTE	Yes	Busy polling
RDMA	Kernel-bypassing	Verbs	Yes	Hardware offloading

Table 3.1: A comparison between the main options for end-host networking in the edge cloud.

comes an obstacle for developers, because code tailored for one of those will not be easy to adapt for another. This problem is especially relevant for edge computing scenarios, which are characterized by a high degree of resource heterogeneity. Furthermore, as these interfaces are typically designed with performance in mind, they are also quite low-level, detailed, and require the knowledge of several different concepts and abstractions.

The following Section will provide a brief introduction to a number of these technologies, by distinguishing between *software-based acceleration*, which includes the Linux Accelerated Data Path (XDP) and the Data Plane Development Kit (DPDK), and *hardware-based acceleration*, which includes Remote Direct Memory Access (RDMA). The goal of this discussion is not to delve into the details of each option, but rather to present their main features and understand how the common design principles previously introduced are practically implemented. In particular, the discussion will focus on the network interface exposed to end users, on zero-copy networking, on the context switches they require, and on the asynchronicity of the programming model they impose. Table 3.1 reports a summary of these considerations.

3.2 Software Acceleration

Software-based acceleration techniques improve end-host network performance without requiring any special hardware to be installed. On the one hand, that is an economical advantage, as the existing general-purpose network equipment can be used. On the other hand, the absence of dedicated hardware means that compute-intensive operations such as protocol processing must be performed by the CPU.

This Section considers two options, Linux XDP and DPDK. Both of them focus on removing the other sources of overhead, in particular data copies. XDP is the most conservative option and executes within the kernel, whereas DPDK adopts a *kernel-bypassing* approach that removes also the user/kernel context switches. The trade-off among these option is between performance and observability: DPDK is faster because bypasses the kernel, but XDP retains the benefits of running in-kernel, such as the standard tools for monitoring. In the following, more details are discussed for each of these technology.

3.2.1 The Linux Express Data Path (XDP)

The Linux kernel introduced the eXpress Data Path (XDP) as the lowest layer of its network stack, located within the driver of network devices [122]. At this stage, XDP allows the execution of user-provided code (*eBPF programs*) for each packet, including forwarding the packet itself to and from

a userspace socket: this way, XDP allows to send and receive packets without involving the other network stack components, thus avoiding expensive operations such as memory allocation for incoming packets. The price to pay is that some amount of CPU is spent to forward each packet between the driver and the socket.

To use XDP, developers have first to open a socket of type AF_XDP and a shared memory area to allow the zero-copy packet writes/reads (directly or through higher-level libraries such as *libxdp* [129]). Then, users send packets by placing data into the memory area and writing a packet descriptor to the socket. Once received the descriptor, the *eBPF program* will send the packet on the network without copies. Packet reception works in the same way, but roles are reversed. If the network card supports it, it is possible to offload the *eBPF program* execution to the hardware. Therefore, this approach bypasses the kernel TCP/IP network stack, achieving efficient zero-copy and low-overhead data transfers. In turn, however, the user has to provide its own implementation of the network and of the transport layer protocols (e.g., mTCP [61]).

3.2.2 The Data Plane Development Kit (DPDK)

The Data Plane Development Kit (DPDK) is a set of C libraries designed to accelerate packet processing workloads running on a wide variety of CPU architectures [48]. Originally designed by Intel in the context of the softwarization of network functions, the flexibility and performance of these library has rapidly made it an interesting option also for end-host networking. Compared to XDP, DPDK takes a step further and completely bypass the OS kernel, hence representing a *kernel-bypassing* technology. This approach results in a reduced scheduling overhead, because there is no context change between userspace and kernel processes on the critical datapath.

The key insight of DPDK is that its libraries let users directly interact with a userspace version of the network device drivers, called *Poll Mode Drivers* (PMDs). The user application and the userspace driver exchange packet data on a shared memory area called *memory pool*. To send a packet, the user must provide to the driver a pointer to the appropriate memory area. On the receive side, incoming packets are placed by the driver into a pre-established memory area, and the corresponding pointers are returned to the user. Just like XDP, DPDK allows the user to directly manipulate L2 (Ethernet) frames. If applications need higher-level packet processing, they must provide their own protocol stack.

Beside the bypassing of the kernel, a second key feature of DPDK is that it allows asynchronous processing: DPDK libraries do not notify applications of incoming data, but it is responsibility of the user to periodically check for that. To minimize latency, usually applications employ one or more threads (*lcores*), each pinned to a separate core, that continuously check for messages from the applications to be sent on the network and from the network to be sent to the application. This *busy polling* approach is very effective in terms of performance, but it also represents one of the major drawbacks of DPDK because it induces a high CPU consumption.

3.3 Hardware Acceleration

Hardware-based acceleration techniques offload one or more computationally intensive tasks to a *network accelerator*, a special-purpose hardware device. In the networking domain, these devices can effectively execute performance-sensitive operations, such as packet processing, much faster

than general-purpose processors, thus allowing application developers to leverage the full performance potential of modern network links. On the reverse side, the requirement of special hardware might become a significant entry barrier to developers, although the prices of these devices are increasingly more accessible.

Network accelerators can be significantly heterogeneous in terms of hardware implementation, supported tasks and protocols, interface exposed to developers, and programmability. From a hardware perspective, most devices are built using three main technologies, namely Field Gate Programmable Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), or Systems-on-a-Chip (SoCs), each with different characteristics and properties. Depending on their goals in terms of product cost and complexity, manufacturers and researchers can decide to embed specific network functions in a network card (e.g., [84, 75]), or to provide them through an additional device (*bump-in-the-wire* approach [31, 44]). From a software perspective, developers interact with these devices through interfaces that can have different degrees of standardization, expressiveness, ease of programmability, and associated performance overhead, depending on the underlying hardware.

Toward a unification, Remote Direct Memory Access (RDMA) has emerged in the last decade as a convenient, cost-effective approach to network acceleration, supported by several different kinds of hardware. Compared to XDP and DPDK, not only it bypasses the network, but it relies on a hardware-based implementation of network protocols that further reduces the CPU intervention on network operations.

3.3.1 Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) is a standard communication model that allows a process to directly access the memory address space of another process on a remote machine [85]. The RDMA specification defines an asynchronous, general-purpose semantics that does not require a specific implementation: any hardware device, including in principle general-purpose CPUs, may implement its model [9]. It is in that way that, as shown in Figure 3.2, RDMA introduces a uniform access layer to hardware acceleration. Originally designed to work on special-purpose Infiniband network fabric, RDMA recently became available also for Ethernet networks (RDMA over Converged Ethernet, RoCE [10]), which made it available also for general-purpose networking, and became a popular networking technique both in industry and academia, resulting in a wide deployment of this technology even in cloud datacenters.

RDMA communication follows two main steps. First, the communicating peers should register a local memory area with the network card (*memory region*) and establish a remote connection by opening a Queue Pair (QP), which comprises a couple of *work queues* for send and receive operations. Then, the actual communication may start. RDMA operations are asynchronous by nature: a node can issue a series of *service requests* to be executed by the hardware, pushing them to the proper queue. Those requests include the transfer of portions of local memory to remote memory regions, or vice versa. The network card enforces these requests in a transparent way, by implementing in hardware the necessary protocols. There are two possible kinds of transfers: *two-sided*, which requires the receiver to actively listen to incoming data, and *one-sided*, which allows a process on one machine to asynchronously access a region of application memory on a remote

node. A great advantage of the latter is that the remote CPU is not involved at all, thus making the latter kind of operations generally faster.

Despite its great performance advantages, however, RDMA has proved quite difficult to use in general-purpose settings. The *de facto* RDMA interface, called *Verbs*, is very low-level and requires the manipulation of several objects. Moreover, the high performance achievable with RDMA links (today, up to 400 Gbps) is difficult to leverage and require a careful application design, as Section 3.5 reports. Furthermore, because RDMA was originally conceived for High Performance Computing, it presents several issues in terms of isolation, security, and scalability that arise when considering its integration in cloud platforms. Therefore, although RDMA is one of the most promising technologies for network accelerators, it is also considered one of the most difficult to use and to integrate in the cloud model.

3.4 Deterministic Networking

Network acceleration allows applications to minimize the end-host overhead associated with data copies, protocol processing, and process scheduling. That overhead does not only impact network metrics such as latency and throughput, but also the *predictability* of network operations. This property is increasingly important in the context of the cloud continuum, in which general-purpose network hardware and protocols is progressively replacing domain-specific solutions.

Although it is not properly an *acceleration* technique, this Section introduces Time-Sensitive Networking (TSN), a vital protocol in those scenarios, such as Industry 4.0 (see Section 2.4.2), in which network predictability is paramount to support real-time industrial traffic.

3.4.1 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 [42].

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 [83].

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 [83].

The TSN protocol is therefore *complementary* to the network acceleration techniques presented in the previous part of the Chapter, and it mainly acts on packet scheduling strategies. Because a complete solution for high-performance applications in the cloud continuum must also support these kinds of requirements, one of the reference systems presented in Chapter 6 will also include a scheduler compliant with TSN.

3.5 Challenges of network acceleration

This Section presents an example of the challenges associated with the efficient use of high-speed network acceleration technologies, in particular of RDMA. The goal of the following discussion is to further motivate the need for an abstraction layer that hides from cloud developers, whose expertise is generally not system-level programming, the complexity of dealing directly with acceleration technologies. Modern cloud applications are indeed complex systems, consisting of multiple interacting components and software layers: the efficient use of acceleration technologies in those contexts requires a high degree of knowledge of both the application and the specific, low-level details of the target acceleration technology. Otherwise, the risk is to leave unused a significant portion of the performance provided by acceleration technologies: the example we propose in this Section is highly representative of how complex systems can expose subtle causes of inefficiencies, which only a specialized set of optimizations can effectively address. We claim that unifying these optimizations in a general-purpose abstraction for high-performance I/O would greatly help developers, who would benefit from them without having to delve into the low-level details of heterogeneous acceleration technologies.

In particular, the following discussion introduces three optimizations, generally applicable to any high-performance communication systems, and describes how they were identified starting from a careful performance analysis of the inefficiencies of an RDMA-based library, Derecho [63], considered as an example of a modern complex system that relies on high communication performance to provide its services. Derecho is a mature, RDMA-capable, open-source library for State Machine Replication. It offers point-to-point and multicast communication options, supporting failure atomicity, total ordering, and optional message logging with durability. We noticed that the system was unexpectedly slow when the objects to replicate were small (<10 KB), and identified three main causes for this poor performance and three optimization techniques to address them. Those solutions turned out to have general applicability in other coordination-based distributed libraries running on high-speed networks, so we grouped them under a single methodology called Spindle [62]. For the sake of brevity, this Section will present a summary of the work, highlighting the aspects relevant for this thesis.

This work is best understood as the next step in a progression of insights concerned with leveraging modern high-speed communication devices. Prior work explored optimizing the match between RDMA and data movement [35, 65]. Derecho introduced a novel monotonic representation of control data that facilitates opportunistic batching. Spindle shifts the focus to the interaction between the application and the RDMA library.

3.5.1 Spindle: Techniques for optimizing atomic multicast on RDMA

My contribution to this work, developed with Sagar Jha and Ken Birman at Cornell University, consisted in the implementation of the ideas, equal participation in formulating some of the receiver-side optimizations, and the evaluation of the system over the multiple iterations of the work.

The Derecho library

The Derecho library [63] implements fault-tolerant State Machine Replication for groups of processes, and its protocols have been proved correct using standard techniques and the implementation checked using the Ivy protocol verifier [22]. Derecho shines when running over RDMA: it sends small messages with one-sided RDMA writes, and large ones in a binomial tree pattern using two-sided RDMA transfers.

When handling the replication of small data-objects, Derecho adopts a protocol called Small Message Protocol (SMC). It manages a set of ring-buffers within a table, shared among the group members: one row per sender. To send a message, an application must wait until a slot of the table is free, fill it with relevant data, and then increment a "messages ready" counter. SMC will then issue the corresponding RDMA write to first push the data to the remote group members, and then the counter. Ideally, the size of the buffer should be large enough to avoid senders running out of free slots, thus enabling continuous sending.

Periodically, a thread polls a series of predicates that check for new events in the table. Three are of special interest when exchanging small messages. A *send predicate* detects whether the application has prepared new messages that are ready to be sent. A *receive predicate* monitors the SMC slots to discover new messages, and the corresponding trigger acknowledges the reception of these messages to the other members. Finally, a *delivery predicate* checks the SST for messages whose reception has been acknowledged by all the members, and thus are ready to be delivered to the application. The performance of these predicates is crucial for the performance of the applications, hence we targeted them in our optimization work.

Small Message Optimization

During the evaluation of the Derecho performance for small messages, i.e. under about 10KB, we noticed that the system was unexpectedly slow, identified three main causes for this poor performance, and defined three optimization techniques to address them. Those solutions, which we refer collectively as Spindle, turned out to have general applicability in other coordination-based distributed settings.

Opportunistic batching. A key source of inefficiency in SMC was that the latency of sending control data (e.g., acks for receiving a message) is comparable to the latency of sending the application messages themselves. Figure 3.3 shows that the RDMA write latency for small messages increases only marginally with the data size, rising from 1.73 µs for 1-byte data to 2.46 µs for 4KB data. This behavior particularly affects the Derecho *receive* and *delivery* predicates previously described, which generate an ack for every new receive and delivery event. Worse, posting each RDMA request to the NIC takes about 1 µs in our setting, a significant delay in light of the critical role of the predicate thread for Derecho. We address this issue by batching events at different stages of the communication pipeline: send, receive and delivery. Instead of letting the system

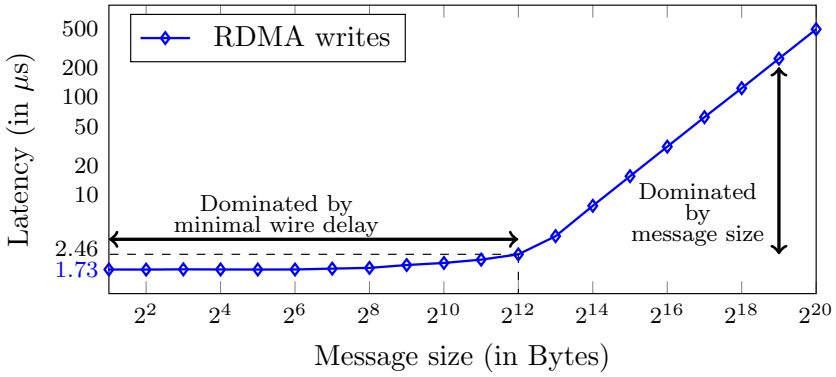


Figure 3.3: RDMA latency vs data size. Latency remains almost constant for up to 4KB message size.

wait to accumulate a fixed-sized batch of messages, which would disrupt performance, we adopt a form of self-balancing *opportunistic batching*: a batch can be smaller or larger depending on the number of events a predicate discovers as it loops. For example, the send predicate checks to see if multiple messages are ready. If so, it aggregates them on the fly, and sends a batch. Interestingly, the opportunistic batching of messages or acknowledgments leads to an improvement of both throughput and latency. In contrast, traditional batching mechanisms wait to collect each batch, thus hurting latency.

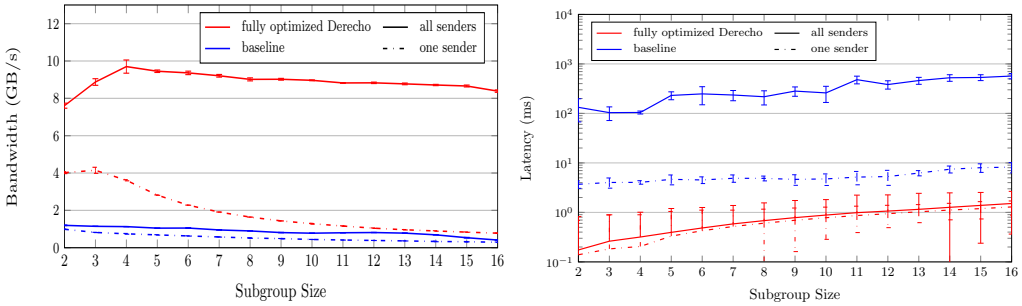
Null-send scheme. Derecho employs a fixed, ordered membership for each epoch, delivering messages in round-robin order by sender. However, this implies that if a sender is not ready to send its next message, the delivery of messages from other senders could be delayed. The problem is that application sending rates might not be steady, and even if they are, delays can be introduced by the OS (e.g., scheduling or interrupt-servicing). To address this issue, we introduce a *null-send scheme*: when a sender node detects that it is due to send a message but has none ready, it sends a dummy zero-sized message (called *null*), permitting continued delivery of messages from other senders. This scheme introduces a negligible bandwidth overhead while making the system adapt very well to real-time delays.

Efficient thread synchronization. Applications access shared data structures when accessing messages or preparing new ones to send. Here, locking protects against concurrency conflicts but can delay the predicate thread. Additionally, many predicates were interleaving accesses to that state and RDMA write operations. As the latter are costly (they can consume 20-50% of the total predicate time), we refactored the predicate code and placed RDMA operations only at the end. Since the logic of a predicate does not depend on the state at a remote node, but only on what is present in the local SST, it is safe to release locks before proceeding with the time-consuming communication operations. Moreover, this optimization increases batch sizes.

Performance evaluation

We assess the performance impact of those three optimizations by comparing the optimized system against a *baseline* version of the Derecho protocols.

Our performance tests employ an application replicated on an increasing number of processes (group size from 2 to 16), each running on a separate physical host. In one test every member



(a) Avg bandwidth utilization for different group sizes.

(b) Avg latency for different group sizes

Figure 3.4: Performance of the atomic multicast protocol with one subgroup and 10KB message size, before and after our proposed optimizations. The term *subgroup* refers to a subset of processes that replicate the same data-objects.

is a sender, whereas there is a single sender in the other test case. Each sender node sends a total of 1 million messages using the new, strongly consistent, QoS option. All members receive every message, and deliver them in the same order. In this graph we fix the message size at 10KB, but we also evaluated smaller sizes and obtained very similar results. The tests were executed on our local cluster equipped with 16 physical machines connected with a 12.5GB/s (100Gbps) RDMA Infiniband switch. Each machine has 16 physical cores and 100GB of RAM.

Figure 3.4 plots the results. Overall, we see that Derecho's bandwidth utilization increased from 1GB/s to 9.7GB/s in the "all senders" case, and network utilization improves from 10% to 77.6%. Even with just one sender, where performance declines with the subgroup size due to increased coordination overheads, our optimizations significantly improved both bandwidth and latency.

3.6 Conclusion

Motivated by the increasingly pressing need to meet stringent performance deadlines, in recent years cloud developers have started to adopt different forms of *I/O acceleration*. All these options follow the overarching trend of separation between control and data plane, which, in the networking domain, translates into the design principles of zero-copy data transfers, userspace network protocols, and asynchronous processing. This Chapter introduced three modern *network* acceleration technologies that implement those principles to varying degrees, ranging from in-kernel acceleration (XDP), to kernel-bypassing (DPDK), to specialized hardware offload (RDMA). These technologies provide significant performance advantages to applications, but tend to sacrifice the ease of use as they expose low-level interfaces and complex programming models. For example, this Chapter reported on the experience of optimizing a State Machine Replication library for RDMA, showing that only careful code tailoring allows to leverage the maximum network speed.

As a consequence, these technologies are not currently ready for an integration within standard cloud platforms in the cloud continuum. The next Chapter will introduce a novel architecture that, by decoupling user code from the specific implementations of these technologies, paves the way toward the option for *Network Acceleration as a Service* in the cloud.

4 NAaaS Requirements and Related Work

In the last decade, cloud computing has embraced the trend of *resource decentralization* through the concept of cloud continuum (Chapter 2). Today, the pressing need to fulfill increasingly demanding performance requirements is pushing toward a process of *resource specialization* that clashes with the core cloud principle of virtualization (Chapter 3). The first part of this Chapter defines the concept of *Network Acceleration as a Service* (*NAaaS*) as a way to make the emerging forms of network acceleration accessible through the standard cloud service model.

Then, the second part of the Chapter considers the significant amount of previous research that set the ground for this thesis. In particular, previous work is organized in three parts, corresponding to the three key challenges that currently prevent the integration of specialized acceleration technologies into the cloud continuum paradigm: the definition of *agnostic interfaces* to accelerate I/O operations (Section 4.2.1); the *system architecture* to support them (Section 4.2.2); and the virtualization of acceleration solution in cloud platforms (Section 4.2.3).

4.1 Network Acceleration as a Service (NAaaS)

The cloud model is founded on the key principle of resource virtualization (see Chapter 2). Virtualization enables users to obtain any kind of resource as a service, ensuring immediate access, unrestricted usage, and the added advantage of elastic scaling. At the same time, it enables providers to flexibly and automatically manage their infrastructure and monitor its real-time status.

Virtualization also introduces an overhead that used to be acceptable until recent years. As software and hardware I/O acceleration becomes available (see Chapter 3), such interposition becomes a performance bottleneck. To contrast this trend, the short-term strategy of major providers has so far consisted in the almost complete removal of such virtualization layer: customers interested in acceleration can rent *bare-metal instances* equipped with specialized devices [5, 6, 12, 13, 52, 53]. Although this solution retains the performance properties granted by those devices, that model has several disadvantages compared to traditional cloud platforms: tenant isolation is enforced through dedicated resources, a solution that is not cost-effective neither for providers nor for the end customers. Furthermore, the almost complete absence of a virtualization layer minimizes the flexibility of those instances, to the point of preventing typical cloud features, such as live migration or the definition of virtual private overlay networks, to be available.

Concept definition

By embracing this trend of specialization, this thesis claims the need for the full integration of acceleration devices as first-class citizens into cloud platforms across the whole cloud continuum

toward the ultimate goal of enabling *Network Acceleration as a Service*. Expanding upon the existing forms of I/O and network virtualization, *NAaaS* envisions that user applications in a virtualized cloud environment have the option to configure *accelerated overlay networks* that offer the native link performance through a standard, high-level, and easy to use interface. This proposition is particularly important in the context of performance-sensitive applications, as their success is contingent upon the availability of these advanced networking options, as thoroughly exposed in the remainder of this work.

Main challenges

The practical implementation of *NAaaS* encounters two primary challenges. A first critical question concerns the selection of the most suitable technology cloud platforms should provide access to. Given the dynamic nature of the cloud continuum and the imperative to ensure code portability, there is not an obvious choice. Chapter 3 reviewed some of the available options, noting how their heterogeneity may harm portability. Hence, the interface exposed to users must ideally be technology-agnostic, but even so, an additional concern is whether such an interface should be similar to the native ones of the supported accelerators, which require a certain expertise to be used, or should rather provide a higher-level, easier-to-use set of primitives.

A second concern is the compatibility between the cloud virtualization layer, which decouples applications from resources, and network acceleration options, which provide high performance by bypassing any system interposition later. That is a significant challenge for providers, as they rely on such decoupling to enforce infrastructure control, including observability tools and automatic management actions.

4.2 Related work

A significant body of previous work paved the way to the ideas and concepts introduced in this thesis. This Section presents a summary of the most significant contributions for this work, organized in three parts. First, Section 4.2.1 surveys the works that propose *agnostic interfaces* to accelerate I/O operations, considering in particular the programming model and the level of abstraction of the proposed APIs. Second, to support agnostic APIs, a suitable system architecture is required: Section 4.2.2 categorizes different approaches proposed in literature. Finally, Section 4.2.3 presents a summary of recent contributions toward more efficient forms of I/O virtualization in cloud platforms. The works in the latter category have the goal to let applications designed for a specific acceleration technology (in most cases, RDMA) to run unmodified within VMs or containers, while also preserving the native performance of the underlying network. Table 4.1 summarizes this classification and highlights the most significant works in each category.

4.2.1 Agnostic interfaces for I/O acceleration

Network acceleration technologies are based on the key principle of separation between control and data planes, as discussed in Section 3.1. The most efficient way to materialize this principle is to require applications to adopt an *asynchronous* programming model: as a result, the native interfaces of these technologies are all fundamentally asynchronous. However, that is also a significant

Section	Category	Most relevant works
Section 4.2.1 Agnostic interfaces for I/O acceleration	Synchronous interfaces	VSocket [124], SocksDirect [74] RSocket [107], VMA [81] Remote Regions [3]
	Asynchronous interfaces	Libfabric [54], Demikernel [131]
Section 4.2.2 System support for agnostic interfaces	LibraryOS	IX [19], Demikernel [131]
	OS module	Snap [78], TAS [67]
Section 4.2.3 Virtualization of I/O acceleration	Hardware offload	AccelNet [44]
	Efficient paravirtualization	FreeFlow [68], VSocket [124]
	Hybrid virtualization	HyV [94], MasQ [57]

Table 4.1: A schematic classification of the most relevant literature for this thesis.

departure from standard network APIs, which are based on an opposite *synchronous* programming model. In this context, previous research on agnostic network acceleration interfaces has either tried to privilege portability over performance or vice versa.

Synchronous interfaces

A flourishing line of research focused on adapting standard APIs, such as POSIX Sockets or the POSIX file system access interface, to be transparently accelerated through RDMA: among the most relevant examples, RSocket [107], VMA [81], and Remote Regions [3]. This is a challenging task because, as discussed in Chapter 3, acceleration technologies and standard interfaces follow opposite design principles. Indeed, all these contributions sacrifice performance in favor of application portability and ease of programming, as they allow applications to execute without modifying their source code or requiring developers to learn new programming abstractions. VSocket [124] and SocksDirect [74] are recent attempts to minimize this performance overhead, in particular by reducing the number of copies on the datapath to those strictly required by the standard interfaces. By providing a cloud-native support for a zero-copy datapath behind the Socket layer, both these solutions achieve much better performance than previous alternatives. Although the overhead induced by data copies remains non-negligible, especially for big payloads, for many users that is acceptable in light of the portability advantages of this approach.

Finally, all these solutions have a relevant issue: the system support that backs them is tailored for RDMA, making the use of other acceleration options impossible. A solution that satisfies the *NAaaS* requirements should instead support a set of acceleration technologies, possibly *pluggable* as this thesis proposes.

Asynchronous interfaces

Recently, Libfabric [54] and Demikernel [131] emerged as the most complete proposals for new agnostic interfaces based on an *asynchronous programming model* and also not tailored to a specific

acceleration technologies. They differ for the different abstraction layers of their API: libfabric offers a driver-level interface, whereas Demikernel targets system-level developers.

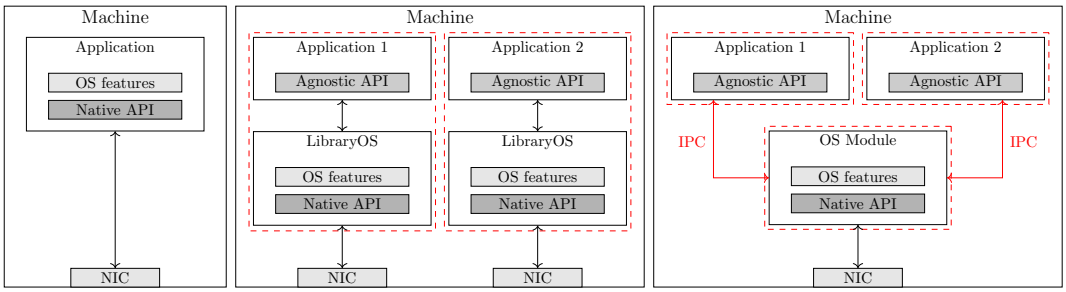
The libfabric library is an industry-grade, mature API that enables applications to run on a wide set of high-performance communication technologies. Developers code against a transparent set of primitives, which the library then translates to the native operations of the specific technology chosen by the user when launching the application. Libfabric supports RDMA, for which was mainly designed, but also other technologies, including DPDK and kernel-based TCP/IP. A key design choice in libfabric is that the interface exposed to the users is very low-level, and it closely follows the structure and programming model of RDMA Verbs. That makes it a very thin layer between applications and the underlying technology, limiting its duties to little more than an adapter and thus minimizing the overhead introduced on each operation. Behind this design is the goal to serve experienced developers that need full control on system resources (e.g., memory management) and benefit from the most advanced features of the native technology (e.g., HPC, RDMA databases, RPC libraries, etc.): all these aspects remain in control of the user application.

On the opposite, Demikernel represents a recent effort that targets general users looking to improve the performance of their applications without specific knowledge on network acceleration. To this end, Demikernel exposes a higher-level interface designed to be familiar to its target users: an extension of the standard POSIX Socket interface that lets applications submit asynchronous I/O operations. Differently from libfabric, within these libraries Demikernel implements typical OS features (memory management, I/O scheduling, network stacks) on behalf of the users, when the corresponding services are bypassed by the selected acceleration technology, following an approach better described in the next Section.

Overall, these two interface solutions aim at maximizing the networking performance and thus ask programmers to adopt a non-standard programming model. Therefore, the target of these works are mainly *new* applications developed with performance awareness. Furthermore, these interfaces do not assume the availability of a specific acceleration technology, thus representing a suitable solution for application portability across heterogeneous environments.

4.2.2 System support to agnostic interfaces

Any kind of agnostic interface requires some form of system-level support to implement the primitives exposed to applications for a specific acceleration technology. Different architectural choices are possible to build such support, and two of them have been recently proposed in literature as the most efficient in combination with network acceleration: on the one hand, Demikernel proposes an approach based on the *libraryOS* model, whereas Snap [78] and TAS [67] introduce the idea of a *microkernel-inspired* userspace OS module. The discussion in this Section defines these two approaches and their differences, which are also graphically represented in Figure 4.1. In both cases, the fundamental idea is that applications should not be in charge of re-implementing typical system features bypassed by each technology, such as memory management or protocol stacks, which are instead provided by the system support. Although that is not the only possible design choice, because, for example, Libfabric leaves these features to be implemented directly by applications, that is the most suitable approach to meet the needs and the experience level of standard cloud users.



(a) No system support (b) LibraryOS architecture (c) OS Module architecture

Figure 4.1: Different system architectures to support agnostic interfaces for high-performance end-host networking. Dashed red lines represent address spaces.

LibraryOS architecture

To support its agnostic interface, Demikernel adopts an approach known in literature as *library OS* [19, 93, 97, 64]. According to this model, the same interface is implemented by a set of userspace libraries, each specialized for a different network acceleration technology (DPDK, RDMA, and kernel TCP/IP are supported). Within the library, all the kernel-based systems features that are bypassed by the associated technologies are re-implemented in userspace, relieving the end user from this burden. In this case, all the code (application and system support) executes in the *same address space* of the application that is using them, resulting in a high efficiency and minimal overhead. However, each application also needs the visibility of at least one suitable network interface. For example, if RDMA is used, an RDMA NIC should be visible to the application, because a new instance of an RDMA-compliant libraryOS will be started. If DPDK is chosen, the application needs to "own" the network interface, removing it from the visibility of other applications in the same system. When applications are deployed in the cloud, for instance as containers running in VMs, these requirements might be challenging to support, as Section 4.2.3 discusses.

OS Module architecture

On the other hand, another line of research proposed a more dynamic model. Although these works do not directly target network acceleration technologies, they investigate strategies to improve the system-level support to application I/O, by defining new techniques for high-performance userspace network processing. These works introduce a *microkernel-inspired* model to support faster network operations. Both Snap [78] and TAS [67] design a *userspace OS module* that centralizes network processing, offering it *as a service* to all the applications running on the same host. Under this model, applications submit I/O operations to the OS module through shared-memory channels, and the module executes the corresponding network processing. TAS adopts this model to provide a *fast path* for a userspace implementation of the TCP protocol in the specific context of RPC workload. Snap is a more general work that allows the definition of custom packet processing modules.

This microkernel-inspired models is particularly suitable for high-performance networking because it retains the advantages of a centralized network stack, as it currently happens for kernel-based networking, even in presence of kernel-bypassing technologies. Among these advantages is

the efficient management of the system resources, including memory allocation, cache-efficient thread scheduling, and the support to transparent software upgrades. This model also promotes higher flexibility: applications can move among different environments and, as long as there is a running instances of the userspace OS module, dynamically attach to the network service on the local host, without the need to instantiate additional resources. In contrast, the use of uncoordinated *OS libraries* would instantiate a new datapath *per application*, requiring dedicated resources (e.g., CPU cores for polling).

The centralized management of the system resources promoted by this model has also a practical consequence for applications that execute in cloud platforms: whereas the use of *OS libraries* requires that each application has the visibility of a suitable network interface, with this centralized approach that should be enforced only for the userspace OS module, significantly simplifying the requirements to provide infrastructural support for network acceleration in cloud platforms. However, these advantages come at the price of an increased overhead for Inter-Process Communication (IPC), because the application and the system code run in *different address spaces*.

4.2.3 Virtualization of I/O acceleration

When considering the integration of I/O acceleration as a commodity into cloud platforms, providers must support the efficient access of virtualized applications to the physical devices. A good I/O virtualization solution consists in the definition of a *virtual* I/O device that preserves the performance of the corresponding physical device as much as possible. Chapter 2 has briefly introduced the standard techniques currently adopted in cloud platforms for this purpose, highlighting how these are not well-suited to support a set of technologies based on opposite design principles. Starting from this consideration, recent research works have proposed alternative strategies to achieve efficient forms of I/O virtualization for network hardware accelerators.

This Section classifies these contributions in three main categories, which we visually represent in Fig. 4.2: new forms of *hardware offload* (AccelNet [44]), a more efficient *paravirtualization* technique (FreeFlow [68], VSocket [124]), and *hybrid solutions* that combine the advantages of both (HyV [94], MasQ [57], SocksDirect [74]).

Hardware offload

The hardware-based virtualization provided by techniques such as SR-IOV [91] (see Section 2.3) follows an *all-or-nothing* approach that lacks the flexibility of software-based control and data paths, making features like monitoring and live migration almost impossible to achieve. To overcome this issue, *AccelNet* [44] has explored the possibility to combine the efficiency of the hardware-based approach with the flexibility and programmability of paravirtualization: instead of exposing a SR-IOV Virtual Function (VF) directly to applications, AccelNet creates a standard *virtual interface* through which applications connect with negligible overhead. During regular network operations, this interface is attached to a SR-IOV VF, benefiting from its close-to-line-rate network performance. Only when dynamic actions are required (e.g., live migration), the virtual interface is temporarily and transparently attached to a traditional paravirtualized datapath to ensure the sufficient degree of flexibility.

Fig. 4.2 shows this technique on the left, where dotted lines represent the control and data paths during transition periods. Assuming that most applications do not often need to be migrated

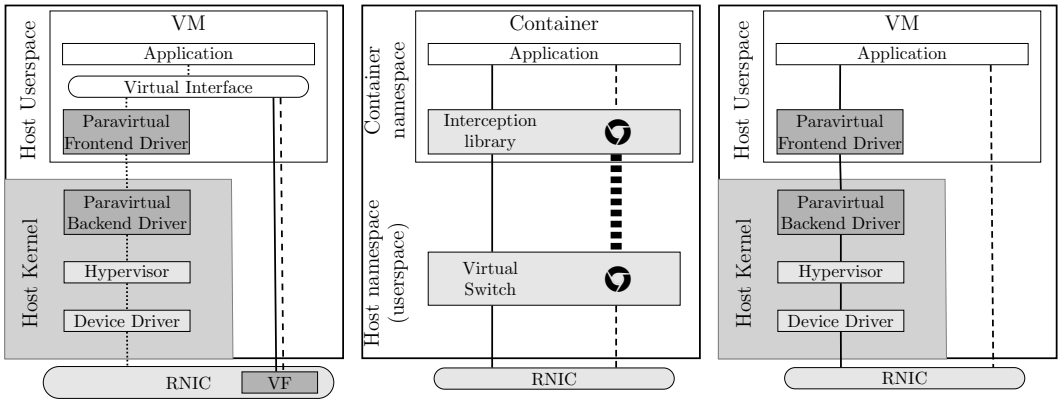


Figure 4.2: I/O virtualization approaches for network accelerators: *hardware offload* (left), *efficient paravirtualization* (center), and *hybrid virtualization* (right). The solid lines represent the control path, the dashed lines the data path. The dotted line on the left figure represent the *temporary* data and control paths during dynamic network operations. The thick dots on the center figure represent shared memory.

or to change their network configuration, this technique enables to leverage the full hardware speed and introduces a high performance overhead only during the temporary transition periods. However, when the datapath is offloaded to the hardware, cloud providers do not have visibility on the network operations, making certain actions such as monitoring still impossible to achieve in software: indeed, AccelNet proposes to integrate them directly into the hardware NIC.

Efficient paravirtualization

FreeFlow [68] and VSocket [124] propose a more efficient version of paravirtualization (see Section 2.3) designed for RDMA support. These proposals clearly separate the control and data planes, with the goal to remove the processor involvement from the data path and to enable forms of asynchronous, zero-copy communication to move packet payloads between the frontend and the backend drivers (see Fig. 4.2 center). For the data plane, these works propose the use of shared memory areas between the two drivers, generally implemented as *ring buffers*, where message payloads will be placed. That way, the two drivers only have to exchange notifications about new messages in the designated area. In turn, the backend driver might register the shared memory area with the RNIC, thus enabling true zero-copy operations: every time a VM and container posts a write operation, the payload of the request is accessible directly from the NIC, and vice versa for reception, thus removing time-consuming payload copies from the data path.

However, device drivers must still involve the processor to exchange request descriptors and completion notifications. Although these descriptors are generally exchanged through asynchronous communication channels, such as UNIX sockets or *virtio* queues, these mechanisms might still introduce too much overhead and ultimately become the performance bottleneck for RDMA. FreeFlow mitigates also this problem through the definition of a *fast path* for latency-critical applications, such that even the notification exchange happens over a shared memory area. Even though this mechanism is effective, especially to improve latency, it also requires a spinning thread

that continuously polls for updates the memory area, a solution not ideal for cloud environments where processor cores are precious assets [44].

Hybrid virtualization

A number of works have introduced a form of *hybrid virtualization* to balance the flexibility of the paravirtualization mechanism with the performance efficiency of I/O acceleration option [94, 57, 74]. Compared to the form of efficient paravirtualization described in the previous paragraph, hybrid solutions take a step further toward the separation between data and control planes, removing from the data plane even the overhead of the software-based notification channels. Indeed, as shown in Fig. 4.2 on the right, *hybrid virtualization* adopts a standard paravirtualization mechanism for control-plane actions (e.g., connection establishment), but lets the hardware accelerator directly access the memory area where guest applications place payload data through a complex mechanism of memory remapping.

This approach represents a interesting trade-off between flexibility and performance, because the hypervisor maintains control on the data plane to enforce properties such as isolation and portability, but without performance penalties. However, hybrid virtualization also comes with some relevant drawbacks. First, it requires a modified version of the vendor-specific driver for the NIC to be loaded into the guest kernel, thus limiting the portability of the VMs and introducing a strong maintenance constraint. Secondly, just like in hardware-based solutions, the hypervisor loses the possibility to enforce the different data plane policies that are typically used by cloud providers to manage traffic on established data connections.

A final consideration about hybrid approaches is that all the existing proposals only consider VMs, except for SocksDirect [74], which offers a socket interface for containerized applications: a monitor process runs in each host, acting as a backend driver with the role to set up the direct data path among local or remote containers. Communication occurs through a shared memory channel for containers located on the same host, whereas remote containers interact directly without the mediation of the monitor, according to the hybrid approach. In this case, since a container is just a process for the operating system, there is no need for a custom frontend driver and containerized applications can directly use the host driver.

4.3 Conclusion

This Chapter opened with the definition of the concept of *NAaaS* as a way to integrate the growing trend of resource specialization, and specifically of network acceleration, within cloud computing platforms, including the traditional *core* datacenters and the more recent extension of the *cloud continuum*. Although this concept is still in its infancy, it derives from a significant body of literature that explored the most important challenges related to its implementation. The second part of this Chapter surveyed the most relevant of these works, highlighting the trade-offs between *portability* and *performance* that emerge at the interface, at the system, and at the infrastructure layers. Combined together, these contributions hold the potential to enable *NAaaS* in cloud platforms.

However, finding the best combination that suits the heterogeneous scenarios emerging in the cloud continuum is not an easy task and previous work did not investigate that in an integrated

way: the contributions surveyed above focus each on a specific aspect, but none proposes a solution that spans all of them. Considering the network interfaces, standard APIs favor portability over performance, whereas other agnostic proposals adopt an asynchronous programming model to maximize performance. At the system level, previous work mostly focused on the definition of supports for fast packet processing alternative to the standard kernel-based stack, but without specific attention to the integration of heterogeneous acceleration technologies. At the infrastructural layer, instead, the goal of the presented works is to enable the efficient virtualization of specific acceleration options (usually, RDMA), but the possibility to have multiple and heterogeneous technologies is not considered. In light of these observations, the next Chapter introduces a novel architecture to support *NAaaS* in cloud platforms that takes all these considerations into account.

5 An Architecture for NAaaS in the Cloud Continuum

The previous Chapter defined the concept of *NAaaS* and reviewed the significant body of research that sets the ground for this work. Although the combination of these previous contributions hold the potential to enable *NAaaS* in cloud platforms, previous work did not investigated this problem in an integrated way that could be suitable for the heterogeneous scenarios emerging in the cloud continuum.

This Chapter first introduces a novel architecture to provide system support to the transparent I/O acceleration of applications in the cloud continuum. The goal of this architecture is twofold: on the one hand, to allow the *portability* of performance-sensitive applications across the whole continuum. On the other hand, to maximize the *performance* provided by acceleration options at the local application deployment site. The proposed architecture is organized into three layers: the *interface*, the *system*, and the *plugin* layers. The second part of this Chapter discusses the role of each layer, the possible implementations, and the trade-offs between portability and performance that emerge from each of them.

5.1 A novel system architecture for NAaaS

This thesis proposes an architecture for *Network Acceleration as a Service* in cloud platforms, to enable applications spanning the whole cloud continuum to transparently accelerate their I/O operations. Based on the discussion in Section 4.1, the requirements for *NAaaS* are twofold: user applications must be portable across a continuum of virtualized resources and must have the option to efficiently leverage any acceleration technology available at the deployment site, even if these options may be very heterogeneous and difficult to integrate into virtualized environments.

Novelty of the contribution

The related work discussed in Section 4.2 significantly contributed to the conceptualization of the architecture proposed in this thesis. The main novelty of this proposal is the adoption of an integrated perspective on several topics previously considered only in isolation: agnostic interfaces for heterogeneous I/O acceleration options, their system support, and the virtualization of high-performance I/O. In this work, all those aspects are combined to provide a complete, unified, and general-purpose solution to provide *NAaaS* in the context of the cloud continuum.

First, several previous contributions investigated different possible design choices for a high-performance, general-purpose I/O datapath, alternative to the standard kernel-based option (Sections 4.2.1 and 4.2.2). However, these options did not consider the need to support heterogeneous forms of high-performance networking, including those that offload some features to dif-

ferent kinds of hardware devices. On the contrary, this thesis adopts the same design principles of those contributions to define a unified abstraction for the coexistence of heterogeneous high-performance I/O options within a single framework, to reduce the degree of complexity exposed to users and, at the same time, to expand the range of I/O acceleration options they can leverage, thus accounting for the typical heterogeneity of the cloud continuum scenarios.

Two more recent contributions already considered the possibility of integrating different I/O acceleration technologies within the same system [131, 25]. However, they propose two different systems that are specialized for a specific deployment scenario, respectively a datacenter and a platform for the execution of virtual network functions. In contrast, this thesis takes a step further and defines a more general architecture, by isolating and characterizing three main components that must be provided by any solution for the integration of heterogeneous I/O acceleration. The definition of these components derives from the abstraction and definition of the three common design principles of network acceleration technologies, which this work is the first to explicitly state (Section 3.1). Furthermore, this work explores two novel implementations of the proposed architecture: a data distribution service, which allows the transparent interchange of heterogeneous I/O technologies to support communication within distributed applications (Section 6.2); a new middleware that decouples the user requirements on communication, expressed via QoS policies, from the actual technology that carries user data (Section 6.3).

Overall, there is still no complete solution to the challenge of offering a high-performance I/O datapath, agnostic to the specific acceleration technology that supports it, to virtualized applications in cloud platforms. The related literature on the virtualization of acceleration technologies (Section 4.2.3) only considers solutions specialized for a specific hardware or software option. In contrast, the architecture proposed in this thesis is designed for heterogeneity. On the one hand, the architecture requires that user applications are portable among different acceleration options, thus accounting for situations in which only some of the latter are supported by the provider. On the other hand, the implementations that we propose are designed to minimize the requirements for virtualization on the provider (e.g., by allowing multiple applications to share the same NIC) and easily adapt to the possible future evolution of I/O accelerators toward better support of virtualization features in next-generation cloud platforms (Chapter 8).

Definition of the architecture

The key insight of the solution proposed in this work is a clear separation between the layer of agnostic API exposed to applications and its implementation by specific acceleration technologies. This separation is embodied by a novel system layer that defines a technology-agnostic set of OS features, corresponding to those typically bypassed for the sake of performance, designed to follow the principles of zero-copy transfers, minimal context switches, and asynchronous processing introduced in Section 3.1. This decoupling approach benefits both cloud users and providers dealing with acceleration. Application developers are freed from the burden to re-implement these system features from scratch and can avoid tailoring their code for a specific accelerator. At the same time, it is easy for providers to integrate these agnostic features with the specific I/O mechanisms provided by the actual acceleration technologies.

Based on these considerations, this thesis proposes a modular architecture for *NAaaS* based on three layers that define how applications access high-performance I/O capabilities, which are the

agnostic systems features that must be provided to support these capabilities, and how these can be combined with the low-level mechanisms defined by different acceleration technologies:

- An *interface* layer exposes a technology-agnostic set of primitives to user applications, thus promoting code *portability* across heterogeneous deployment sites. A key responsibility of this layer is to define how users designate which communication channels require acceleration. This choice might be either *explicit*, hence leaving to the user a certain degree of visibility on which specific technology should be actually used, or *implicit*, hence delegated to the platform based on user-provided hints. Section 5.1.1 discusses the role of this layer more in detail, by also addressing the trade-offs that can emerge in terms of portability and performance.
- A *system layer* embodies the actual decoupling abstraction between the application code and the underlying technologies. This layer defines a set of OS system features *as a service* to applications that would otherwise need to implement them from scratch. Among these features, there is memory management for *zero-copy* data transfers, thread scheduling to handle asynchronous operations, and packet scheduling to allow traffic prioritization. Section 5.1.2 further details these features and how they are designed after the design principles of modern acceleration techniques to be easily combined with their low-level mechanisms.
- A *plugin layer* implements the applications' network operations by using the native interface of the actual acceleration technologies and by interacting with the agnostic features defined by the system layer. This layer must be organized in a modular way to segregate the technology-specific details into *pluggable components*, thus potentially allowing applications to dynamically attach to different acceleration options. Within each modular component, providers can adopt the necessary optimizations to *maximize performance*, such as those introduced in Section 3.5, and provide additional system features if required (e.g., protocol stack). Section 5.1.3 provides more insights into the role of this layer, including how this modular architecture favors the integration of I/O acceleration options in cloud infrastructures.

The decoupling role of the system layer is the distinguishing feature of this architecture for *NAaaS*: by programming against an agnostic interface, users can design portable applications

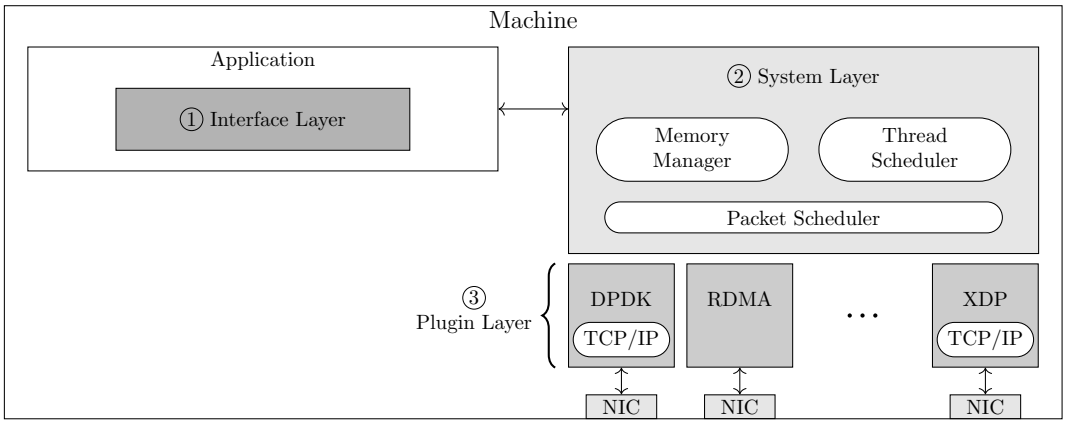


Figure 5.1: A schematic representation of the proposed architecture.

that, thanks to the modular plugin layer, can attach to the possibly heterogeneous acceleration options available in different deployment sites. The role of each layer is discussed in more detail in the following Sections, as different design choices and trade-offs emerge in terms of application portability, performance efficiency, and integration with virtualized environments.

5.1.1 The interface layer

The interface layer defines the set of primitives through which applications transparently access accelerated I/O operations. The ultimate goal of this layer is to enable a *write once, run everywhere* service model, so it is paramount that these primitives remain unchanged across the whole cloud continuum. Beyond that, the definition of a suitable interface for this layer also requires to consider two other crucial aspects. On the one hand, the interface must define a *programming model* that is compatible with high-performance I/O. On the other hand, because applications in the cloud continuum have deeply heterogeneous requirements, it must define a mechanism for developers to designate which communication channels require acceleration, and more broadly, which are their QoS requirements.

Programming model

The previous discussion in Section 4.2.1 on agnostic interfaces highlighted that two different programming models, synchronous and asynchronous, have been adopted in literature to provide access to high-performance I/O, with a different balance between application portability and performance. On one side, APIs such as the standard Socket guarantee ease of programming and backward compatibility because of their ubiquitous adoption, but they are also bound to a synchronous programming model that requires a significant processor intervention on the data path. On the other side, alternative agnostic interfaces propose an asynchronous model to maximize performance, based on zero-copy data transfers and minimal processor involvement in I/O operations, but require developers to redesign applications to take full advantage of it.

Both models are a possible choice for this layer. Indeed, although a high-performance interface for *NAaaS* would be based on the asynchronous model, the vast amount of legacy cloud applications still need to be supported through standard APIs, even if that reduces the effectiveness of the acceleration support.

Choice of Acceleration

Applications in the cloud continuum are characterized by heterogeneous QoS requirements, as widely discussed in Chapter 2. The most important QoS parameter considered in this thesis is the need for I/O acceleration, but several others are relevant, such as resource consumption. Hence, this architecture requires that any implementation of the interface layer let users specify the QoS requirements of their communication channels: for instance, which ones require acceleration.

The kind of mechanisms exposed to users to operate this choice is critical to determine the degree of visibility and control they have on the whole infrastructure. Different interfaces can make users *explicitly* choose which acceleration technology they want to bind their code to, or instead leave this choice *implicit*, delegating the mapping to the system support. The explicit approach is already adopted in literature by driver-level (e.g., libfabric) and system-level (e.g., Demikernel)

agnostic interfaces: the desired mapping in the local deployment environment is a configuration option to set before the application instantiation. This option is *static*, as the mapping remains the same once the application is started, and it is suitable for scenarios where the kind of locally available resources is known a priori. Instead, the implicit approach moves to the underlying system layer the choice of which acceleration technology must be used to support the user code, leaving to users the possibility to provide *hints* about the mapping through high-level *QoS* options. This is a more *dynamic* option that is particularly useful in contexts characterized by heterogeneous resources and high mobility of application components, such as those arising at the network edge, and potentially enables to attach at runtime the same code to different technologies.

Existing examples of the *implicit* approach are middleware interfaces, such as the OMG Data Distribution Service (DDS) [88] and the OPC-UA [46], which are indeed used mostly at the network edge, where such dynamic use cases typically arise. However, although these interfaces offer a wide range of high-level QoS parameters to be associated to I/O channels, none o them currently considers the option for acceleration. The next Chapter introduces two possible implementations of middleware-level interfaces that follow this approach.

5.1.2 The system layer

The system layer is the core of the proposed architecture as it offers system support for high-performance I/O *as a service* to applications running on the same machine. This layer represents an alternative to the standard I/O stack of common operating systems by providing an equivalent set of features, but designed according to the modern principles of zero-copy transfers, minimal context switches, and asynchronous processing. The technology-agnostic design of these features effectively supports the uniform interface exposed to applications, and decouples user code from the internal mechanisms used by each acceleration technology to provide them.

The features provided by this layer correspond to those typically bypassed by applications that use acceleration techniques and whose efficient re-implementation is typically burdensome for the average cloud user. Because data copies are the most relevant source of overhead in traditional stacks, *memory management* for zero-copy data transfers is one of the main responsibilities of this layer, which must manage memory allocation on behalf of applications to minimize copies. Strictly related to memory management is the scheduling of the asynchronous tasks that act on such memory, such as commanding output actions, detecting incoming data, or progressing packet processing: following the experience reported in Section 3.5, another crucial feature needed by applications is to interact with the OS to *efficiently schedule* one or more threads to execute these tasks, balancing performance and resource consumption. Finally, many applications especially at the network edge have specific QoS requirements that require the prioritization of traffic belonging to certain I/O channels, such as in industrial environments (see Section 3.4): to meet these needs with minimal overhead, the system layer must also provide the possibility to *schedule I/O operations* according to these priorities. The next paragraphs will describe these features in more details, explaining how they can be designed in a technology-agnostic way.

The design and implementation of these features in an agnostic way is key to the goal of enabling *NAaaS* in the cloud continuum. However, this choice also forces the system layer to only provide access to the minimum set of common functions among the supported technologies. For example, a distinguishing feature of RDMA is one-sided data transfers (see Section 3.3.1). How-

ever, either an agnostic version of this feature is provided at this layer (similarly to Snap [78]), or it cannot be exposed to applications. This limitation derives from the intended target of the proposed architecture: average cloud users that need application portability across the cloud continuum. Hence, this architecture is not designed to support applications with more advanced needs, such technology-specific features of a higher degree of control on the system resources. Other architectures, such as the libfabric library, are more suitable for those applications.

Zero-copy memory management

Modern I/O acceleration techniques adopt a common approach to achieve the goal of zero-copy data transfers. During a preliminary registration phase, one or more memory areas are registered with the NIC for Direct Memory Access (DMA). Then, applications place data to send and retrieve received data from these memory areas.

To generalize this mechanism and hide it from application, the system layer directly manages memory allocation on behalf of the served applications. When an application attaches to the system, the system registers memory areas with the NIC. Then, whenever needed, the application asks the system for a memory buffer, writes on it, and submits an output request. On the receive side, the application gets a buffer containing the incoming data from the system, consumes it, and then releases it back. This general mechanism has the advantage of being independent from the specific details of each acceleration technology, but also to reproduce the common behavior of all these options. Hence, it is easy for the system to interact with the plugin layer as both support the same kind of operations.

Thread scheduler for asynchronous operations

According to the principle of data and control plane separation, applications designed for high-performance I/O follow an asynchronous model, such that I/O operations and protocols are driven by the application needing the data instead of data receive or transmit events driving the application. In this context, the architecture proposed by this thesis leaves to user applications only the responsibility to submit send requests to the lower layers and to detect incoming data from them. These requests are then translated by the plugin layer, which will be better described in Section 5.1.3, into technology-specific operations.

To this end, the role of the system layer is to ensure that the asynchronous I/O and protocol processing actions of the plugins are executed frequently enough to avoid stalling the send and receive pipelines, which would hurt performance, but also not too frequently, as that would induce a higher CPU consumption than necessary. To balance these needs, the system layer must define a thread scheduling strategy, possibly by interacting with the operating system, that considers two key aspects: which and how many threads will execute the plugin logic, and how these threads are scheduled in the OS, to ensure they have priority over other application processes.

The choice of the number of threads to dedicate to the plugin asynchronous operations is not trivial. On the one hand, modern powerful multi-core processors allow multiple threads to operate in parallel on I/O operations, thus compensating for the relative slow speed of a single core (see Chapter 3). However, multi-core I/O processing also carries a set of issues. First, it may require the introduction of locks on shared resources, thus potentially stalling the overall processing. Second, handling multiple cores also increases the need for multiple context changes, which is a major

source of overhead as previously discussed. Finally, single-thread processing allows to maximize the use of the data and instruction cache, which can be very efficient (see also Section 3.5). Overall, because different threading models can have a significant impact on performance and resource consumption, the system layer should ideally define a set of possible strategies and leave the choice to users or providers on which is the most appropriate for their specific requirements.

Packet scheduler

Performance-sensitive applications, especially at the network edge, might require the prioritization of some I/O channels that carry critical traffic. Section 3.4 introduced an example of the techniques typically used for that purpose, including the definition of standard protocols such as TSN to implement this prioritization across all the network components involved in the transmission. Considering end-host I/O, the key component to ensure traffic prioritization is a packet scheduler, which must detect packets belonging to the critical flows and prioritize their forwarding to/from the network.

According to the design principle of minimal context changes, the system layer should not rely on externally-provided scheduler, such as those provided by the OS (e.g., Traffic Control in Linux), but must define its own scheduler and integrate its actions within the threading strategy previously described. Ideally, this scheduler should provide a default behavior, e.g., FIFO scheduling, but also support custom strategies from users. That would allow, for example, the system layer to natively support the TSN scheduling required by industrial edge applications.

5.1.3 The plugin layer

The responsibility of the plugin layer is to translate the agnostic primitives exposed to user applications into the corresponding operations of the selected acceleration technology. Whereas memory management, threading model and packet scheduling are managed directly by the system layer, the plugins interact with these features to define the actual operations for memory registration and data send/receive using technology-specific mechanisms. As previously anticipated, the fundamental design choice proposed in this thesis is to organize the support for each technology as a modular and self-contained component that can be attached as a plugin to the system layer, thus effectively decoupling the general system support for high-performance I/O from the specific implementation details. This choice has two main consequences that are discussed in the following: it gives the opportunity for technology-specific optimizations that are transparent to the final user, and it greatly simplifies the use of network accelerators from cloud environments.

Internally, each plugin might find it necessary to provide additional system features required by the associated acceleration technology. In the case of networking, a typical missing feature is a network protocol stack (usually, TCP/IP), as all the network acceleration options bypass the standard version provided by the OS. Whereas some options implement it in hardware, such as RDMA, others require that a software version is provided. By freeing the end-user from this responsibility, each plugin may also introduce more efficient protocol implementations and performance optimizations: whereas the in-kernel implementation is designed for maximum generality at the expenses of performance, custom implementations can explore different trade-offs and possibly reduce generality in favor of a more efficient processing. Overall, the plugin-based architecture

segregates all the necessary optimizations for a specific technology (e.g., the techniques identified in Section 3.5), without requiring the end-user to be involved.

A second important consequence of this modular organization is the easier integration of network acceleration options in cloud platforms. Currently, cloud applications in need for acceleration should directly interact with a suitable network interface, which is hard to provide while maintaining both the native performance efficiency and the typical cloud flexibility, as widely discussed in Chapter 3. The architecture proposed in this thesis removes this constraint: user applications access I/O capabilities through an agnostic interface, and it is the plugin layer that interacts with network interfaces. Hence, applications have maximum flexibility, as they can be moved wherever and whenever necessary and attach to a different local plugin. In turn, providers can effectively adopt the most efficient form of I/O virtualization (e.g., hardware offloading) without compromising application portability.

5.2 Conclusion

This Chapter introduced a novel architecture designed to enable *NAaaS* in cloud platforms across the whole cloud continuum. The architecture was presented through the definition of the three layers that constitute it: the interface, the system, and the plugin layer. This design neatly separates the agnostic system features exposed to users from the specific mechanisms adopted by the acceleration technologies: this decoupling enables the development and deployment of *portable* cloud applications and their transparent yet efficient access to I/O acceleration options. Furthermore, this decoupling also eases the integration of these acceleration options in cloud platforms, thus effectively paving the way for a practical implementation of *NAaaS*.

The next Chapter will focus on how the three layers of the proposed architecture can be practically implemented, by proposing two *reference implementations* systems and discussing how they represent two different design choices in the space of the possible implementations.

6 Reference Implementations

The general architecture proposed in this thesis defines the specification for the system support to *NAaaS* in cloud platforms, but it does not mandate a specific implementation, leaving many practical design choices to system developers. This Chapter explores the different implementation choices possible at the different layers of the architecture and shows that these may lead to very different systems, thus flexibly adapting to heterogeneous application requirements. At the same time, these implementations still retain the fundamental properties of application portability and efficient access to acceleration technologies from cloud environments across the whole cloud continuum as required by the *NAaaS* definition.

After an initial general discussion about the possible implementation choices, this Chapter introduces two reference implementations of the proposed architecture: a data distribution middleware called *DerechoDDS*, and a userspace OS module called *INSANE*. The key difference between them is at the system layer, where the two implementations follow two distinct models to answer their application requirements. To demonstrate their effectiveness in supporting *interactive* applications across the cloud continuum (see Chapter 1), a practical use case is presented for each of them: DerechoDDS is used to support the creation of faster function pipelines in a serverless computing platform, whereas INSANE is used to support a demanding application in the Industry 4.0 domain, potentially in combination with a 5G network infrastructure.

6.1 Implementation design choices

This Section explores the implementation choices that are possible for each of the architecture layers defined in Chapter 5, and motivates why some of them are particularly interesting for this work. Although each layer has a distinct and well-defined purpose, the implementation choice at one layer may have relevant consequences on the others: for instance, the choice of an interface may influence the design of the system layer, which in turns provides the mechanisms that plugins must specialize. Therefore, the next Section 6.2 and Section 6.1 will show how these choices can be combined together to create two systems with different properties.

6.1.1 Interface layer

The interface layer specification leaves great implementation freedom to system developers, recognizing that standard interfaces, usually synchronous, are still widely used although not particularly suitable for high-performance I/O. The only constraint on the system interface is the availability of a mechanism for users to identify which I/O channels need acceleration.

The POSIX Socket interface has indeed the advantage of being standard, but it is also limiting in terms of performance, as widely discussed in the previous chapters. If the application requirements are compatible, as often happens in the cloud continuum, a possible standard alternative

to sockets might be a higher-level messaging middleware interface, for instance that of the OMG Data Distribution Service (DDS) [88]. DDS exposes a standardized and technology-agnostic interface that lets applications exchange data according to a publish-subscribe model: messages are exchanged as updates to topics and delivered to all the topic subscribers. DDS uses high-level parameters, called QoS policies, to let developers associate communication properties to topics, such as whether I/O operations should be asynchronous. Therefore, the DDS interface is well-suited as an implementation of the interface layer, because it combines a standardized set of operations with an *implicit* choice of the underlying technology through QoS parameters, as well as the the option for asynchronous I/O. However, DDS also carries some intrinsic problems: the high number of possible QoS parameters makes it quite complex to use for the average developer; on top of that, it is not a general-purpose interface, thus not suitable for any kind of application.

Alternative to standard interfaces are the custom proposals surveyed in Section 4.2, in particular Demikernel [131], which introduces system-level primitives similar to the standard sockets but oriented to an asynchronous programming model. The Demikernel interface could indeed be considered a possible implementation of the interface layer, although the choice of which I/O technology to use is *explicitly* required to the user, making the mapping too static for the most dynamic edge use cases. By combining the desirable features of DDS with concepts from the Demikernel experience, Section 6.3 will introduce a novel middleware interface that is general-purpose, based on QoS parameters for an *implicit* choice of which technology map to I/O operations, and designed to be easy to use for the average developer. These characteristics make it a suitable choice for applications across the whole continuum, and an ideal candidate as an implementation of the interface layer.

6.1.2 System layer

Section 4.2.2 introduced two system models that are well-known in literature to support high-performance userspace I/O operations.

system features from userspace components. A first approach is the *libraryOS*: the system-level code runs in the same address space of the application that is using it, removing inter-process overhead but also statically binding that application instance to a specific I/O technology and, usually, to a specific network interface within the deployment environment. The second possibility is to have a *OS module* that runs as a separate process and centralizes the system features for all the applications running on the same host. In this latter case, the inter-process communication overhead between applications and the module is compensated by a more efficient use of the local resources, such as processor caches and network interfaces, within the module itself.

These two system models answer to different application requirements. The libraryOS approach is more static in terms of mapping applications to a specific I/O technology, because it substantially collapses the system features (system layer) and their technology-specific implementation (plugin layer) into a set of uncoordinated libraries, one per technology. That might be efficient and easier to implement, especially if there is only one or a very small number of applications on a single machine. Conversely, the OS module approach keeps the system and the plugin layer well separated, allowing a higher degree of dynamicity for the supported applications, as typically required in edge environments, and resulting in a better resource usage in case of multiple applications using its services.

Because of the relevance of both these models, this thesis presents an implementation example for each of them: the DerechoDDS system introduced in Section 6.2 follows the libraryOS model, whereas INSANE in Section 6.2 is designed as an OS module.

6.1.3 Plugin layer

At the plugin layer, the main challenge is to preserve the efficiency of the raw acceleration technologies in combination with the agnostic features provided by the system layer. The reference implementations proposed by this Chapter will focus on two network acceleration technologies: RDMA, as an example of hardware-based acceleration technology, and DPDK, as an instance of a software-based option. In the latter case, this layer should provide additional systems features, in particular a network stack (usually, TCP/IP) as DPDK does not provide it and bypasses the kernel-level implementation. Overall, this layer could also offer more sophisticated features, such as in the case of the DerechoDDS system that is presented below.

6.2 DerechoDDS: A Data Distribution Middleware

The discussion presented in this Section is a brief summary of a series of papers developed in collaboration between our research group and Ken Birman at Cornell University. I was the leading author of this work and contributed to the idea formulation and to the system implementation of the system, with help from Weijia Song about the integration with the Derecho library [102, 103]

The first reference implementation introduced in this Chapter is DerechoDDS, a novel system compliant with the OMG Data Distribution Service standard [88]. Many applications delegate the management of the communication among their components to a middleware layer deployed between their code and the underlying operating systems, protocol stacks and hardware. This approach reduces development time and effort, facilitating integration, reusability, extensibility, and better overall scalability. Among the available options, the DDS standard is already designed to transparently support a set of different network technologies that can be used interchangeably without application code changes. To achieve this goal, the DDS standard clearly separates its interface, called Data-Centric Publish-Subscribe (DCPS), from the underlying technologies. However, no commercial implementation currently supports network acceleration.

The new implementation presented here provides the option to transparently accelerate DDS applications through RDMA by providing a new pluggable library designed according to the *libraryOS* system model. Indeed, especially at the network edge, many applications that adopt DDS have limited dynamicity requirements, but strong constraints in terms of minimal communication overhead, such as in safety-critical settings like automotive or avionics. Therefore, a libraryOS approach is more suitable for a new implementation of this middleware, and the pressing need for high performance motivates the adoption of a form of network acceleration like RDMA.

Interestingly, the acceleration provided by RDMA enables the DDS middleware to provide additional guarantees: not only improved communication performance, but also strong properties such as *data consistency* among distributed components without heavy performance penalties. By mapping the DCPS interface to Derecho [63], a library for high-performance state machine replication (see Section 3.5), DerechoDDS provides an optimal RDMA hardware mapping, breaks

past performance records, and is also able to offer additional consistency guarantees. That opens the perspective of novel use cases for this middleware, such as for applications that need a significant degree of fault-tolerance, or to share some plan of action among components that will be each responsible for a distinct aspect, as Section 6.2.3 will demonstrate.

The following discussion provides first a brief background on the DDS interface, explaining how it is particularly suitable to be accelerated through RDMA while still remaining transparent for the user. Then, the system-level implementation of DerechoDDS is described, with a focus on how the libraryOS approach basically collapses together the system and the plugin layer of the reference architecture.

6.2.1 Standard interface and programming model

The standard Data-Centric Publish-Subscribe (DCPS) interface is based on the abstraction of a *Global Data Space* (GDS), where topics are represented by distributed objects of a given type (Figure 6.1). Within the application, publishers write their messages as updates to their local copy of the topic-object, whereas subscribers monitor the object for updates. Under the covers, the DDS system layer intercepts these updates and propagates the update to the local copies of the interested subscribers. The implementation of this abstraction through standard communication protocols requires the plugin layer to operate an explicit translation between object updates and messages to be sent on the network. Intuitively, this translation is not necessary with RDMA: by registering the corresponding memory area with the NIC, the middleware could simply command the transfer of the local object copy to the relevant remote subscribers, leaving the message translation to the RNIC.

A rich set of parameters, called *QoS policies*, allows the developer to control several properties about how the update propagation should happen. For example, the *Durability* QoS controls the lifetime of data written to the GDS. It supports four values: (1) *Volatile*, if data should be discarded immediately after delivery; (2) *Transient Local*, if data should be stored in the local cache of publishers and subscribers, thus allowing late joiners to catch up; (3) *Transient*, which ensures

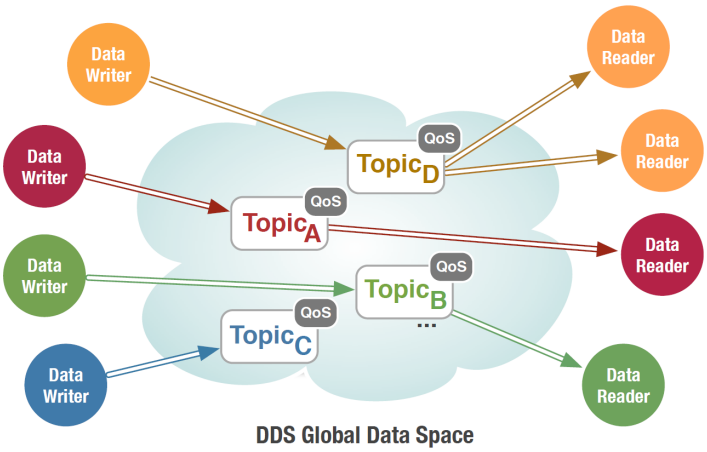


Figure 6.1: The DDS abstraction of a Global Data Space [33].

that data are kept even beyond the lifetime of single publishers or subscribers; (4) *Persistent*, which stores those data in persistent memory to make them survive system failures.

As anticipated in Section 6.1, a key insight of this work is that it is possible to extend this set of QoS parameters to let user express the need for higher performance. Hence, in this work we suggest the introduction of two new QoS parameters. One should be the *Acceleration QoS*, which can be used by developers to signal the need for a certain topic to be accelerated. Furthermore, as previously discussed, the availability of a fast communication option also allows to introduce the option for a stronger degree of consistency among distributed copies of the GDS: these properties can be expressed through new *Consistency QoS* parameter.

6.2.2 System library implementation

The implementation of the interface layer in DerechoDDS is collapsed together with its own RDMA implementation, according to the libraryOS model. That results in a pluggable library that can be used interchangeably with other libraries, in turn providing support for other communication technologies such as the kernel-based TCP/IP. In addition to the motivations previously introduced for this choice, this model is also suitable to offer the option for a higher degree of data consistency in DDS by supporting a state machine replication model (SMR) [114].

Because the primary concern of DerechoDDS is communication performance, it implements the DCPS interface by mapping it to the Derecho library, already introduced in Section 3.5, which is highly optimized for RDMA and is particularly suitable to implement the GDS. Indeed, the publisher simply operates directly on the shared object, with the effect that the published message is created "in place," i.e., directly in the memory region that Derecho will copy to remote peers via RDMA.

DerechoDDS stores

The first step to build DerechoDDS was to model the concept of DDS Global Data Space (GDS). We addressed this by defining a set of distributed key-value stores, one for each *Durability* option: *NoStore* for volatile, *TransientStore* for transient local, and *PersistentStore* for persistent (Fig. 6.2). As a consequence, we can represent DDS Topics as objects of a user-defined type that live in one of these three K/V stores: The object key serves as a topic name (for *keyless* topics [88]) or the name plus a set of fields of the corresponding type (for *keyed* topics). Once we defined this mapping, we implemented these three stores as Derecho replicated objects. First, we defined a basic set of

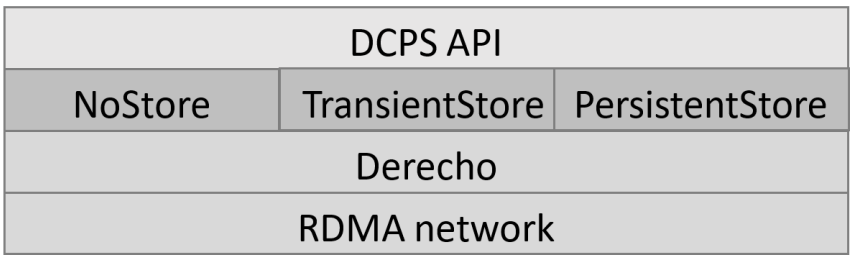


Figure 6.2: Architecture of DerechoDDS.

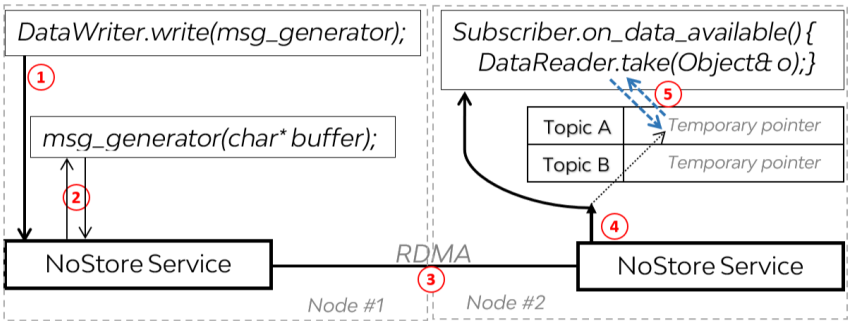


Figure 6.3: Zero-Copy data path of DerechoDDS for Volatile durability.

operations to access the store: the most important one is `put`, which inserts a new object in the store, if the corresponding key does not exist, or updates its value if it already exists. Then we modeled the state on which such operations act, which depends on the desired durability level: the *NoStore* store does not have an actual state, as updates fade after application delivery. The *Transient* store keeps the last n received updates in main memory, as well as the *Persistent* store which additionally backs them up in persistent storage. For reasons of brevity, in the following we will focus on the Volatile case.

Zero-copy data path and Acceleration QoS

Once the stores were defined, we mapped the standard DDS interface onto them as seen in Figure 6.3. As a first step the publisher registers the generating function by calling `DataWriter.write`. Internally, DerechoDDS calls the `put` operation to update the topic value on the corresponding store, which in turn will ask Derecho a free memory buffer. When the buffer is available, Derecho requests that the user-supplied message generation logic build the message (②), after which it can push the message remotely via Derecho multicast (③). On reception, the Derecho core notifies DerechoDDS of the update. The middleware places in a topic-specific variable a pointer to the received message, and then invokes the subscriber's listener (④). Within the listener, the user-provided logic retrieves the value (⑤).

Thanks to this zero-copy data path, DerechoDDS can be configured to achieve much better performance than existing commercial implementations, as Chapter 3 will demonstrate. This work proposes to introduce a new *Acceleration QoS* parameter to let users control this aspect in a transparent way, thus potentially opening up the possibility that other forms of acceleration are supported. The most basic version of this option could have two values, such as to have *accelerated* or *non-accelerated* topic. More sophisticated alternatives would be possible as well, such as those that will be introduced by INSANE in Section 3.

Consistency QoS

This work proposes the introduction of a new QoS policy, *Consistency*, to allow users to enhance the maximum DDS consistency level from *eventual* to *atomic*. Consistency offers two possible values. The *eventual* setting selects for the standard OMG behavior, while the *atomic* option selects for SMR guarantees. Derecho itself has two forms of multicast: a weakly reliable one, and

Durability QoS	Consistency QoS	Derecho Service
Volatile	Eventual	NoStore, Unordered subgroup
	Atomic	NoStore, Ordered subgroup
TransientLocal	Eventual	TransientStore, Unordered subgroup
	Atomic	TransientStore, Ordered subgroup
Persistent	Eventual	PersistentStore, Unordered subgroup
	Atomic	PersistentStore, Ordered subgroup

Table 6.1: Mapping of DDS *Durability* QoS on Derecho.

an atomic option. Accordingly, it suffices for DerechoDDS to select the appropriate primitive, as seen in Table 6.1. If *eventual* consistency is selected, DerechoDDS will deliver any update to the relevant subscribers as soon as it is available. In contrast, for *atomic* consistency, DerechoDDS selects the Derecho atomic multicast, which will delay delivery until the SMR obligations of totally ordered, fault-tolerant delivery can be assured. As such, the latency costs of the atomic option are of particular interest, and we evaluate them carefully in Section 7.2.1.

6.2.3 Use case: a serverless platform

This Section summarizes two works developed as an integration between the research project of my colleague Andrea Sabbioni, which works mainly on serverless computing, and my work on DerechoDDS and on network acceleration technologies [110, 111]. We contributed equally to the formulation of the ideas and to the experimental evaluation of the system, whereas I led the work on the actual implementation of a prototype of the proposed serverless platform.

The availability of a data distribution middleware that couples the advantages of network acceleration with the abstraction of a potentially strongly consistent distributed object store opens up new usage scenarios: although most of the commercial DDS implementations are oriented to support mission-critical applications at the network edge, the proposed DerechoDDS can be useful for a broader range of applications across the whole cloud continuum.

To demonstrate that, this Section presents DIFFUSE: a DIstributed and decentralized platForm enabling Function composition in Serverless Environments. DIFFUSE embodies an innovative infrastructural support in FaaS environments (see Section 2.4.1), thus enabling the efficient and transparent composition of functions through the distributed shared-memory abstraction implemented by DerechoDDS. In this work, the middleware is used as a pluggable support, serving as a conveyor of messages among the platform components.

Additional background

Section 2.4.1 already introduced the concepts of serverless computing, FaaS, and the current open challenges related specifically to the problem of composing functions into efficient pipelines. In addition to these considerations, Figure 6.4 depicts the typical components of a FaaS platform.

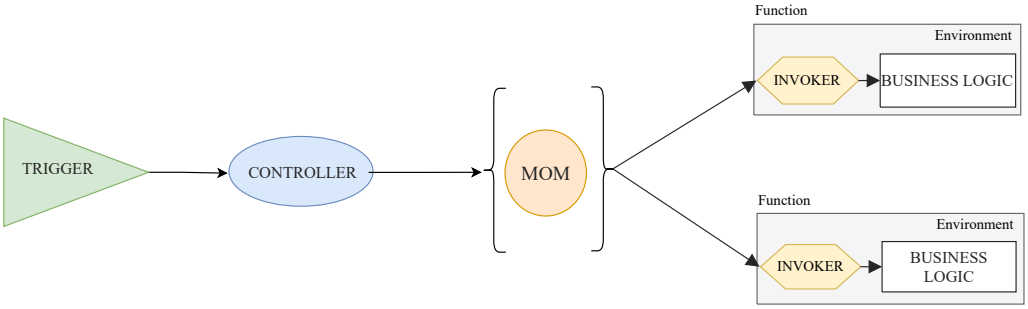


Figure 6.4: High level FaaS architectural model based on a Message-oriented Middleware (MoM), where the middleware decouples the controller and the function. In alternative approaches (*direct invocation* scheme) the function invocation is enacted by the controller.

The first element, directly interfacing with the incoming event, e.g., event generated at the network edge, is the *trigger*. This component receives external events from heterogeneous sources, via potentially different protocols, and converts them to local events for the FaaS platform. The events generated by the trigger are then managed by a *controller* which, based on configuration parameters provided by the user, forwards the event(s) to the proper function. Overall, the controller is tasked with the function lifecycle management.

The *function* constitutes the unit of execution in FaaS, encapsulating the business logic used to process specific events. A function is composed by an *environment*, e.g., Java, an *invoker*, and the *business code*. The invoker receives events, injects the business code deployed by the end-user, and successively launches the function to execution. The code is always executed inside a proper *environment* comprising all the required dependencies, e.g., system libraries.

Mainstream FaaS platforms implement this function invocation scheme either through a client/server pattern or through a publish/subscribe approach by exploiting a Message-oriented Middleware (MoM) as an additional component of the architecture. The reliance on a publish/subscribe scheme allows the controller to be relieved from part of its responsibilities such as load balancing, event delivery, etc., delegating them to the MoM [41].

Outline of DIFFUSE

In this Section, we present a detailed view of DIFFUSE, outlining its main components, interaction flow(s), and the mechanism at the basis of the function composition feature. This proposal relies on DerechoDDS as a strongly-consistent shared-memory middleware capable of exploiting modern hardware capabilities, embodying a similar semantic to the in-host shared-memory communication. To emphasize the role of DerechoDDS in the support of this work, the last part of this Section briefly discussion on the characteristics of other middleware frameworks that could potentially be adopted in place of DerechoDDS, thus also evidencing the broad spectrum of deployment options currently supported.

Function composition architecture

The composition mechanism comprises two layers: (i) the configuration and coordination layer instrumenting the FaaS platform components, and (ii) the function-to-function communication layer serving as a conveyor of messages between components.

Configuration and coordination. Our proposal offers to users the capability to define custom processing pipelines expressed via association rules, residing outside the functions' business logic. Currently, the association rules are shipped to the controller in a JSON-based format, and allow the definition of generic, graph-shaped processing pipelines whereby the pipeline continuation is determined by the output of the executed function. This also supports run-time modifications and updates to the processing pipeline, adding to the flexibility of the approach.

Indeed, at instantiation time and periodically, the FaaS components - trigger and invoker - can request the necessary configuration from the controller and participate in advancing the execution of the processing pipeline. These asynchronous updates allow moving the FaaS *controller* outside the chain invocation mechanism, thus reducing the function response times compared to the approach where the controller is involved in each function invocation.

Fig. 6.5 shows an example of a processing pipeline that is composed of three functions, namely A, B and C. In a hypothetical scenario, the execution of the pipeline is triggered because of a user issued request, calling function A into execution. Upon function A termination, depending on its returned output inspected by the invoker component, the continuation of the pipeline will be either the execution of function B or C. It is important to note that in contrast to the *reflective invocation* mechanism where the (control) burden is offloaded to the controller entity, in our approach the control logic is decentralized and distributed to invoker entities.

Function-to-function communication mechanism. Once a pipeline configuration file is pushed to the platform, the components establish a series of communication queues (*topics*) used to exchange application and control data among them. In particular, the function-to-function com-

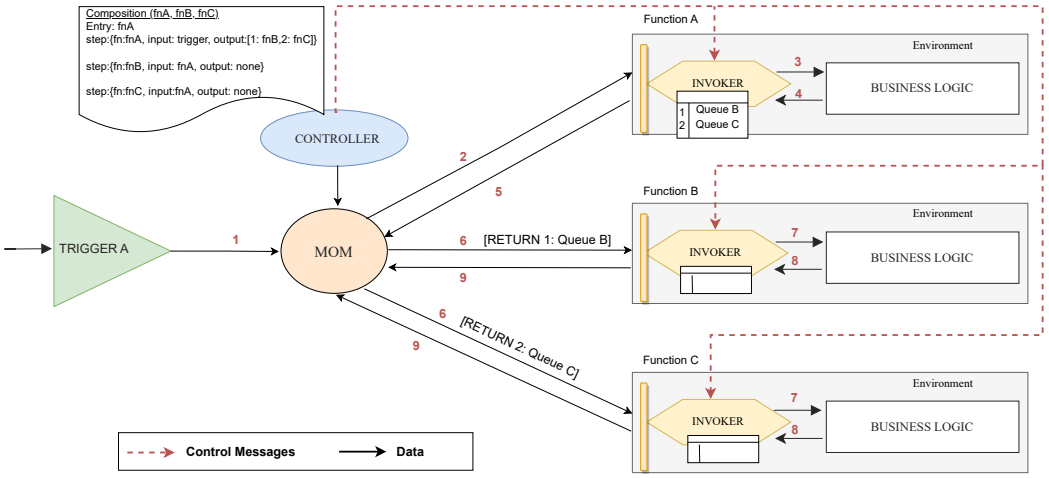


Figure 6.5: DIFFUSE relies on a MoM-based approach for function-to-function communication; invokers retrieve function invocation requests directly from the MoM triggering function execution.

munication mechanism relies on publish/subscribe middleware to transparently invoke the next function in the pipeline. Back to our example, once function A terminates the execution, the invoker consults the output and depending on the value, forwards the output either to Queue A or Queue B, consequently triggering the execution of function B or C, respectively. That allows to dynamically scale the number of *invoker* instances and to increase the level of parallelism.

In these settings, the MoM acts as a conveyor for all messages and events, hence it is of paramount importance that the solution be efficient and gracefully scale with the number of requests. At the same time, it is desirable the platform be agnostic and decoupled by the specificities of the underlying MoM solution, promoting portability and openness to future extensions. To this end, we have introduced an abstraction layer (orange box near the Invoker and Controller in Fig.6.5) decoupling the components from the specific MoM APIs by implementing a set of high-level primitives such as the creation of a communication channel, send and receive of messages, etc.

In the following, we present how we adapted DerechoDDS to implement the DIFFUSE plugin interface, thus allowing us to exploit modern hardware, guaranteeing low-latency and high-bandwidth communication. Next, we discuss the other MoM alternatives and overall characteristics, adding to the deployment spectrum of our platform.

DSMQueue: a Distributed Shared-Memory Queue

The Distributed Shared-Memory Queue (DSMQueue) is a thin interface layer, alternative to the full-fledged OMG DDS interface, that we built to adapt the DIFFUSE MoM plugin interface to DerechoDDS. Beyond a greater simplification of the syntactical details, there is no semantic difference to the interface presented for DerechoDDS, and its implementation relies on the same library that is described in Section 6.2.2.

DSMQueue represents a zero-copy, *delete-after-read* data transfer mechanism embodying a similar semantic to the the Linux kernel `mqueue` primitive, but it is able to move data across remote hosts. More in detail, DSMQueue is a distributed queue that exposes a *push* and a *pop* operation and may be configured to offer different semantics, such as FIFO (default) or LIFO. In the following, we present the other MoM solutions already integrated in our framework, discussing their characteristics and tradeoffs that emerge.

MoM Deployment Considerations

Adding to the deployment spectrum of our proposal, we identified two other state-of-the-art MoM solutions, namely Apache Kafka and Redis Stream [7, 51]. In specific, Kafka is a highly scalable, open-source event streaming platform, while Redis Stream is a streaming abstraction built on top of the widespread persistent Redis database.

Table 6.2 provides a summary of some characteristics the different MoM solutions embody. All the options offer advanced state replication and consistency mechanisms for improved load distribution and fault tolerance. In particular, Kafka exploits a multi-broker mechanisms with a configurable level of topic (channel) replication, while Redis employs a classical Driver-Worker active replication scheme. Similarly, our DSMQueue proposal replicates queue state (data) exploiting RDMA to guarantee the highest possible performance. DSMQueue, which is based on the Derecho library, adopts the same active replication pattern of Redis, but the logic is completely decentralized, thus eliminating the need for a driver node on the critical data path. In this setting,

MoM	Delivery semantic	Delivery Order	Load Balancing
Kafka	Exactly-Once, At-Least-Once, At-Most-Once	Within single partition total ordering	Producer-side: Static, Round-Robin
Redis Stream	At-Least-Once At-Most-Once	Total Ordering	Consumer-Side: First come First served
DSMQueue	Exactly-Once	Total Ordering	Consumer-Side: First come First served

Table 6.2: Properties of different MoMs supported by DIFFUSE.

all the nodes are equal peers that agree on the same shared state, thus achieving the maximum possible degree of parallelism.

Concerning the delivery semantics, DSMQueue offers an exactly-once semantic, while Redis offers an at-least-once embodying less synchronization overhead when compared to DSMQueue. This behavior may lead to a lower use of the network resources, but does not guarantee the consistency of the shared state in case of failure of one or more nodes, which DSMQueue is always able to guarantee. Kafka is the only one of the three solutions that, thanks to its deep integration with Apache Zookeeper, allows choosing among all the three delivery semantics at-most-once, at-least-once, and exactly-once at a topic granularity.

Section 7.3 will present an experimental evaluation of DIFFUSE, assessing our proposal while varying the underlying MoM support. In particular, we evaluate and compare the capabilities of DSMQueue with the other traditional MoMs, identifying possible deployment trade-offs.

6.2.4 Concluding remarks

This Section presented DerechoDDS as the first reference example of the architecture proposed in this thesis. Its implementation as a *libraryOS* is motivated by the kind of applications typically supported by the DDS middleware, which are mostly static but highly demanding in terms of minimal performance overhead. By introducing the option for the transparent RDMA acceleration of DDS-based communication, this Section showed that not only to guarantee the application portability, but also to expand the capabilities of the middleware by providing additional semantic guarantees (data consistency).

The second part of the Chapter showed how the possibility to obtain transparently accelerated communication opens up new use cases for the DDS middleware itself: DIFFUSE is an an innovative platform that, based on DerechoDDS, enables the efficient and transparent composition of functions in the context of the emerging FaaS computing paradigm.

6.3 INSANE: A userspace OS module

The system described in this Section is the result of a joint research effort with my colleague Andrea Garbugli [101, 104]. I lead the INSANE project and proposed the initial idea formulation, whereas Andrea is the reference author for the Industry 4.0 use case and the TSN-related features [49, 50, 105]. We both contributed equally to the prototype implementation and experimental evaluation.

The second reference implementation of the proposed architecture builds on the experience of DerechoDDS with the goal to offer general-purpose and easier-to-use access to accelerated I/O, and also a more dynamic support for applications in the cloud continuum. According to these design goals, INSANE (Integrated aNd Selective Acceleration for the Network Edge) is a novel middleware optimized for the emerging class of edge cloud applications that combine intelligent logic, stringent performance requirements, and heterogeneous deployment scenarios. Overall, INSANE is a system natively designed to offer Network Acceleration as a Service and follows the OS module model. It consists of two main components: a runtime, which must be in execution on each participating machine, and a client library that exposes the API to the applications, allowing them to interact with the runtime. A set of *datapath plugins* then specializes the runtime for a wide set of acceleration technologies, including RDMA, DPDK, and Linux XDP.

On the one hand, similarly to the DDS interface but in a much more simplified fashion, the INSANE interface lets developers declare their communication requirements through high-level QoS policies and uses them as hints to dynamically bind each I/O channel to the most appropriate network acceleration technology available at the deployment site. On the other hand, the INSANE system support is designed as a userspace *OS module* that effectively decouples application code from the specific technology dynamically found at the participating nodes, thus maintaining high network efficiency while also easing code development and portability. By clearly separating the system layer and the plugin layer, INSANE also supports much more dynamic use cases in which containerized applications can be seamlessly migrate across different locations without requiring time-consuming configuration actions.

In the following, we describe more in detail the INSANE API and how it can ease the portability of latency-sensitive and network-intensive edge applications. Then, we provide an overview of the runtime architecture to understand how the INSANE primitives are mapped to heterogeneous network technologies. Finally, Section 6.3.5 proposes a use case to highlight the benefits of using INSANE in a typical Industry 4.0 scenario: we implement in software a Programmable Logic Controller (PLC) and use INSANE to make it run in a container while also fulfilling its demanding performance requirements.

6.3.1 The INSANE interface

The INSANE client library exposes a minimal interface that meets three key requirements. First, developers must find it easy to use, in contrast with the currently available interfaces of network acceleration techniques that require them to know a myriad of complex and low-level details. At the same time, the interface must be expressive enough to enable the efficient implementation of heterogeneous domain-specific abstractions on top of INSANE. Furthermore, the interface must

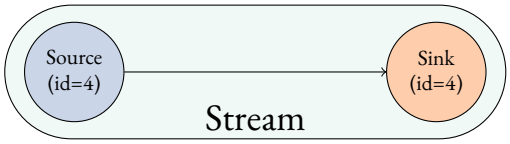


Figure 6.6: An INSANE *channel* is created between sources and sinks with the same *channel id* within the same *stream*.

be agnostic to the underlying transport protocols and only expose high-level policies to inform the middleware about the quality requirements of different data flows.

To keep the interface as simple as possible, the INSANE API defines few basic concepts. A *communication channel* represents a unidirectional data flow among endpoints, which can interact locally or through the network. A channel may only exist within a *stream*, an abstract concept that associates a set of quality requirements to one or more channels. In the context of a stream, a communication channel is established among endpoints called *sources*, which produce data, and *sinks*, which consume data. Each channel is uniquely identified by an application-provided *channel id*, that users must pick according to their higher-level business logic. For example, an INSANE-based Message-oriented Middleware (MoM) would typically assign channel ids according to topic names. Figure 6.6 shows an example of an INSANE channel: sources and sinks opened within the same stream and with the same *channel id* will communicate on the same channel.

The concept of the *stream* is fundamental in this interface. Only sources and sinks belonging to the same stream can exchange data, because the stream defines the set of quality requirements for the communication. Depending on those requirements, INSANE will transparently map the channel to a technology-specific concept, e.g., a kernel-based socket. When sinks and sources are co-located, we enable direct data forwarding using shared memory.

Figure 6.7 shows the complete INSANE APIs. Any application must first open a communication session with the local runtime. Then, it can open one or more *streams* by specifying a set of quality options, which the next paragraph will cover extensively. Once a stream is open, it is possible to create sinks and sources to define the desired communication channels using the *channel id* mechanism previously described.

All the available operations on sinks and sources are asynchronous in order to ease zero-copy communication. To send a new message from a source, users have to first require a memory area (*buffer*) from the runtime. Then, the application can write the message into that buffer and *emit* it, thus signaling to the middleware that data is ready to be sent. This operation returns a token that can later be used to retrieve the outcome of the operation. Similarly to Demikernel [131], we do not offer *after-write protection*: developers must not modify the buffer content once it has been emitted. On the sink side, we offer three different ways to receive data. Users can register a callback to be called every time a new message is received for that sink. Alternatively, users can directly call the `consume` operation, which can be configured to either return immediately, regardless of the presence of new data, or to block until new data is available. In any case, to preserve the zero-copy semantic, new data is returned as a pointer to a memory area borrowed from the runtime. Hence, as soon as the user finishes processing the data, it should return the memory to the middleware by explicitly *releasing* that buffer.

```
1  /* Open and close a session */
2  int init_session();
3  int close_session();
4
5  /* Stream */
6  stream_t create_stream(options_t opts);
7  void close_stream(stream_t stream);
8
9  /* Source APIs */
10 source_t create_source(stream_t stream, int channel);
11 void close_source(source_t source);
12 buffer_t get_buffer(source_t src, size_t size, int flags);
13 int emit_data(source_t src, buffer_t buffer);
14 int check_emit_outcome(source_t source, int id);
15
16 /* Sink APIs */
17 sink_t create_sink(stream_t stream, int channel, data_cb cb);
18 void close_sink(sink_t sink);
19 int data_available(sink_t sink, int flags);
20 buffer_t consume_data(sink_t sink, int flags);
21 void release_buffer(sink_t sink, buffer_t buffer);
```

Figure 6.7: The INSANE interface.

We believe that this set of primitives answers our design goals of simplicity, flexibility, and transparency toward multiple network acceleration options. At the same time, this API is expressive enough to allow the definition of very different higher-level interfaces. To demonstrate this claim, in Section 6.3.4 we report our experience in implementing and deploying two very different applications, a decentralized messaging queue and an image streaming framework. Both the applications were easy to develop and demonstrate a significant performance advantage from the selective acceleration capabilities guaranteed by INSANE.

6.3.2 The INSANE QoS policies

A key contribution of this work is the possibility to associate a set of *quality requirements* to each communication channel through the concept of *stream*. These requirements are defined in terms of high-level Quality of Service (QoS) policies, thus effectively making INSANE transparent toward the low-level network details. In line with our goal of maximum simplicity, we reduce the number of available options to the essential. INSANE currently defines three possible *quality options* that can be associated to a stream: the degree of *datapath acceleration*, the level of tolerable *resource consumption*, and the *time-sensitive* constraints of a data stream.

The *datapath acceleration* policy signals to the middleware whether a specific data flow requires any network acceleration or the regular kernel-based networking would suffice. In case the acceler-

ation is needed, edge developers must have control over the associated cost. For this purpose, users can set the *resource consumption* policy to specify whether resource usage is a concern to take into account when mapping data flows to specific technologies. For example, DPDK requires a high CPU consumption that may be unacceptable in some contexts. Finally, a third policy allows users to characterize data flows depending on their *time sensitiveness*. This policy specifies the packet scheduling strategy for the packets of that flow. By default, a FIFO scheduler handles all the packets and sends them to the network as soon as the user code emits them. Instead, if the stream is labeled as time sensitive, we offer a scheduling strategy compliant with the Time-Sensitive Networking (TSN) standard [42] to provide a deterministic network behavior (see Section 3.4).

As soon as a new stream is created, INSANE maps the stream quality requirements to the most appropriate network technologies available in the dynamically determined deployment environment, according to a user-configured mapping strategy. If no custom strategy is provided, INSANE acts as follows. If no acceleration is required, the kernel-based UDP protocol is always used. Otherwise, RDMA is the best alternative, because it offers the best network performance for a low resource usage (network operations are offloaded to the NIC). However, RDMA is typically used in bare-metal deployments and is not yet available in most cloud settings. Hence, INSANE alternatively maps user code to DPDK if resource usage is not a concern, otherwise to XDP. In fact, XDP is generally slower but does not require a set of CPU cores to continuously spin to detect the arrival of new packets [66]. Because this mapping is performed at runtime by INSANE, triggered by the creation of a stream, the user code always remains unchanged, independently of the actual deployment execution. In any case, INSANE considers these policies as hints about the application performance requirements and adopts a best-effort attempt to build the mapping between quality and actual technologies. Thus, if acceleration is required but no acceleration technology is available, INSANE will fall back to the standard kernel-based network stack and warn the user about this decision.

Following a precise design choice, INSANE does not offer additional communication control policies. Thus, for example, there is no built-in way to define a specific fault tolerance semantic. The adopted approach is that developers are responsible to design mechanisms as part of their own custom logic. In this way, we leave them free to easily re-implement existing solutions on top of INSANE with little effort. This is in line with many middleware systems, such as the OMG DDS [88], that already assume a *best-effort* network and provide their own solutions to build additional guarantees [87].

6.3.3 The INSANE runtime

This Section discusses the architecture of the INSANE runtime, which is designed to be a userspace OS module offering Network Acceleration as a Service. In particular, more than what discussed for DerechoDDS, the focus of this Section is how the different system features of the architectural system layer (Section 5.1.2) are implemented to uniform the network operations of heterogeneous technologies, which we use as a support for the primitives discussed in the previous Section.

According to the *OS module* design, the client library and the runtime framework of INSANE reside in separate processes. The advantages of this model in terms of flexibility, dynamicity, and address space isolation come at the price of a necessary inter-process communication (IPC) between the two components, which is absent in systems that run their own logic in the same polling

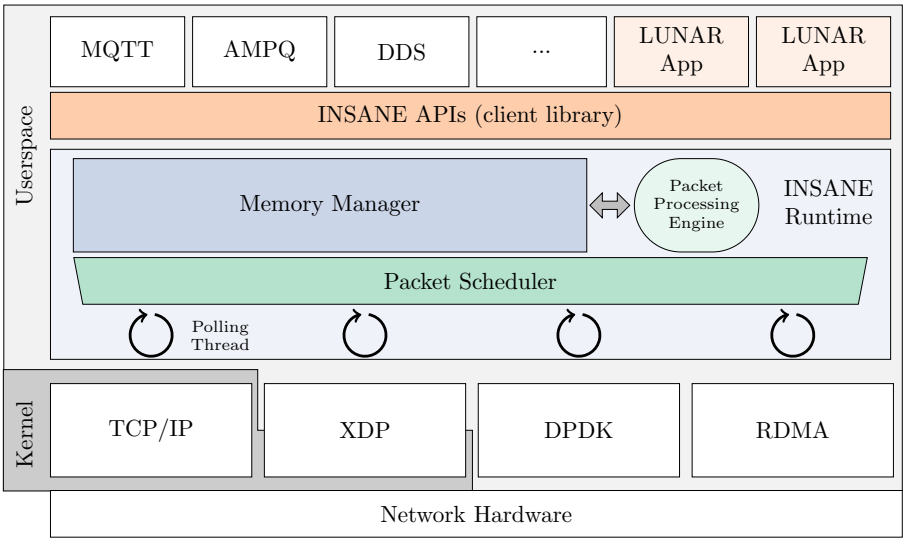


Figure 6.8: The INSANE Architecture.

thread. However, not only the associated overhead is small in our case of zero-copy networking [78], but many factors contribute to minimize it while also retaining the advantages of this model: in particular, state-of-the-art lock-free queues [47, 125], combined with modern multi-core processors and IPC optimization techniques [120, 56, 82].

The INSANE runtime has four main components, represented in Figure 6.8: a *memory manager*, a *packet scheduler*, a pool of *polling threads*, which answer the need for agnostic system-layer features (Section 5.1.2). A set of *datapath plugins*, clearly separated from the other features, implement the plugin layer (Section 5.1.3). The memory manager, in particular, effectively implements the abstraction that decouples the homogeneous interface offered to the applications from the highly heterogeneous details of each transport technology. At the system startup, the memory manager reserves a memory area (*memory pools*) to contain application data. That area is divided into *memory slots*, uniquely identified within the pool by a *slot id*. When a new application connects to the runtime, it maps part of that area in its own address space. From then on, the application and the memory manager communicate by exchanging *slot id*s that refer the position of relevant data in that area.

Figure 6.9 illustrates the communication flow between a sink and a source. As a preliminary operation, each application must connect to the runtime (`init_session`). Then, to send a new packet, the application requires to the manager a memory slot (①). If a free slot exists, the manager sends the corresponding *slot id* to the client library, which provides the application with a pointer to the associated memory area. Thus, the user can directly write the packet content in the shared memory. Once finished writing, the application *emits* the packet (②) and the INSANE client library communicates the corresponding *slot id* to the runtime. Once received the token, the *packet scheduler* schedules the packets for send according to the *time sensitiveness* policy. By default, our scheduler adopts a FIFO strategy, but others are possible as we discuss in the next paragraph. On the reception side, the mechanism works symmetrically. The NIC places the newly arrived packets in a designated memory area. When the manager detects them, it sends the

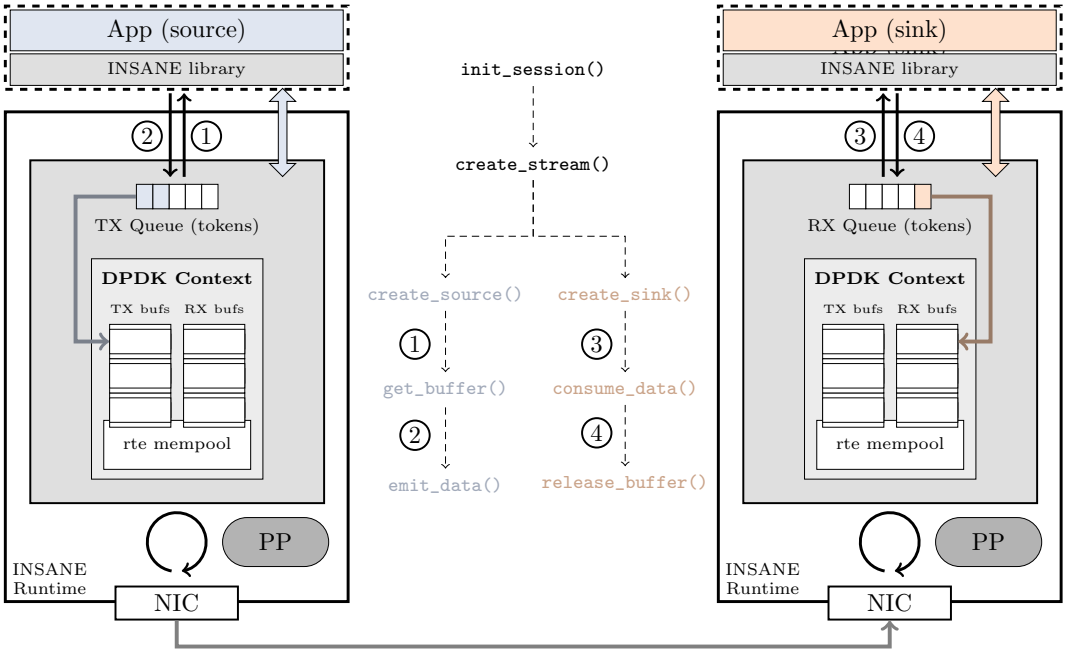


Figure 6.9: The INSANE communication flow when using the DPDK plugin.

relevant *slot id*s to the client library, which offers applications a pointer to the same memory are where data has previously been placed (③). Once done, the application must return the token to the runtime to make it available for subsequent operations (④).

The implementation of this general mechanism for the different network technologies is responsibility of the *datapath plugins*. Each plugin, one per available network acceleration technique, must define a send and a receive operation. The send operation sends the scheduled packets to the currently bound network, using the low-level API of each specific technology. Before that, in the case of DPDK and XDP, the *packet processing engine* processes the outgoing packets through the userspace network protocol stack; this step is unnecessary for kernel-based networking, which uses the kernel stack, and for RDMA, which offloads the task to the hardware. On the reception side, the datapath plugins use the technology-specific API to check for newly arrived packets. Such new packets are first processed by the packet processing engine, if necessary, and are then dispatched to the relevant applications according to the previously described mechanisms.

The execution of the datapath logic is responsibility of a pool of *polling threads*. The number of these threads and their mapping to the datapath plugins is flexible and configurable depending on the user needs in terms of performance, scalability, and resource consumption. Depending on performance goals, one or more threads can be dedicated to a specific datapath, thus leveraging cache locality and packet processing parallelism. On the opposite, when resource consumption is paramount, INSANE can be configured to run more than one plugin on a thread, at the cost of a lower performance. In any case, to avoid scheduling overhead, each polling thread is pinned to a different processor core; at the same time, threads are automatically paused when idle.

Packet scheduler

The packet scheduler provided by INSANE detects new packets from applications and schedules their actual transmission based on the application-provided *time-sensitiveness* QoS. The scheduler is designed to be general and to possibly allow users to customize the scheduling strategy according to their need. By default, in addition to providing a FIFO strategy, INSANE also supports the Time-Sensitive Networking (TSN) standard (see Section 3.4). In this configuration, the scheduler works as a Time-Aware Shaper (TAS) compliant with the IEEE 802.1Qbv standard, specifically designed for soft real-time applications.

This solution is particularly innovative considering the currently available options, as today this packet scheduling option is not available as a userspace component. Bare-metal applications can use directly the kernel-based one, which however forces them to use the standard kernel-based TCP/IP networking and to inherit all the overhead we highlighted for the kernel-based datapath. Containerized applications cannot even access this option, 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, and a faster userspace version for applications running bare metal or in VMs.

Cloud integration

As anticipated at the beginning of this Chapter, a significant advantage of the OS module design is the possibility for an easier integration of I/O acceleration technologies into cloud platforms. Because applications access I/O as a service from the module, they do not need to directly interact with a network interface. That makes both the I/O and network virtualization tasks (see Section 2.3) much easier for cloud providers, as the dynamicity requirements of applications are already fulfilled by the OS module. That also allows to support multiple applications with the same network interface, instead of requiring the provider to offer a distinct NIC (as a SR-IOV VF

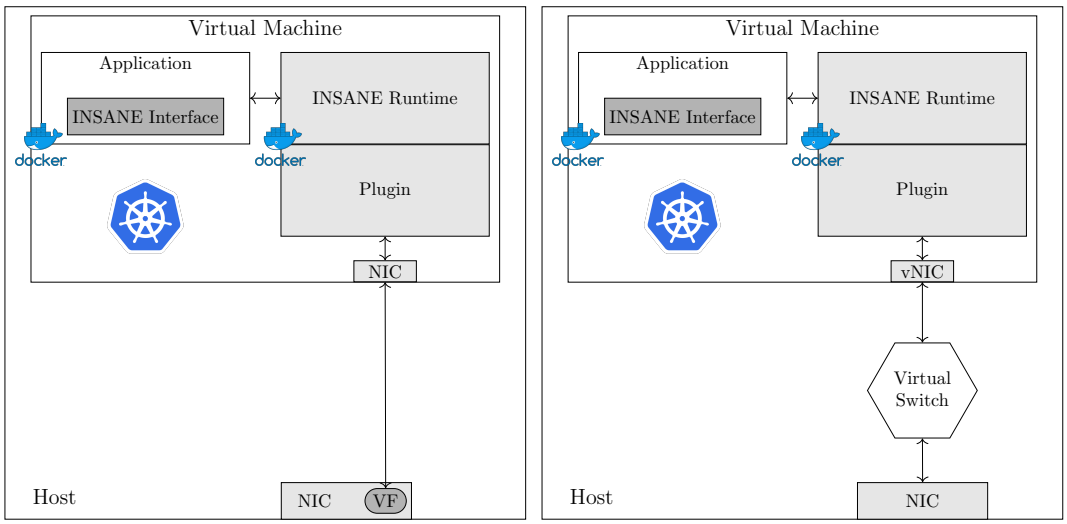


Figure 6.10: Two possible INSANE deployments in a cloud platform.

or as a vNIC) to each of them. Yet that does not hurt performance, as the actual I/O operations are implemented using the native technology interface.

Figure 6.10 shows two possible deployment options of INSANE in a cloud platform. On the left side, a hardware approach is shown. That is the most efficient option, as the interface accessed by the INSANE runtime is a slice of the physical NIC directly assigned to the VM. However, that approach requires the provider to include in the NIC a switching component to operate the necessary *network virtualization* (see Section 2.3). That feature could in turn require a custom design such as that in AccelNet [44] or the exploitation of specific SmartNIC features [84]. Alternatively, the provider should rely on a software switch, that is less efficient even in its most optimized forms as commented in Section 4.2, but easier to set up and deploy. The choice of the most suitable support is up to cloud providers, but INSANE is designed to be compliant with both options.

Furthermore, independently of the solution adopted for network virtualization, INSANE is also designed to allow a flexible management of both applications and of the INSANE runtime through standard cloud tools. Both applications and the runtime can indeed be deployed as containers (e.g., Docker containers) and thus be integrated with standard cloud orchestration tools (e.g., Kubernetes [32]). To further promote the integration of INSANE with the cloud ecosystem, we implemented INSANE as a Kubernetes network plugin that can be seamlessly integrated alongside existing options (e.g., Flannel, Calico) [72]. That makes it possible to automate the deployment of INSANE-based applications by requiring the availability of an INSANE runtime directly from the configuration and deployment automation tools cloud developers are already used to adopt for this purpose.

6.3.4 Application example: LUNAR applications

A key design goal for INSANE is to ease the development of a broad set of general-purpose applications with heterogeneous requirements in edge cloud nodes. To demonstrate that our interface effectively answers this purpose, we use the INSANE API to build two typical edge applications, a message-oriented middleware (*Lunar MoM*) for data distribution and a data streaming framework (*Lunar Streaming*). We demonstrate that INSANE enables the portability of these applications across various network technologies while delivering very close (ns-scale overhead) performance to the native technology interfaces. The performance evaluation of these two applications is presented in Section 7.4.

LUNAR MoM

The previous discussion about DerechoDDS and the DIFFUSE serverless platform demonstrated the wide range of possible usage scenarios for Message-oriented Middleware systems. We built a simple decentralized MoM, called *LunarMoM*, using the INSANE API. Mapping the MoM abstractions to the INSANE primitives is straightforward: the resulting application, consisting of just 135 lines of C code, defines two main primitives to publish or subscribe on a topic, `lunar_publish` and `lunar_subscribe`. The publish function takes the topic name, which is then hashed to obtain the topic id, and a callback function as arguments, and opens a INSANE source if this is the first publication for that topic. Then, it gets a buffer from INSANE, executes the user callback to fill it, and sends it. Under the hood, INSANE will forward the messages to the

reachable remote INSANE runtimes and deliver them to the subscribed sinks. The subscriber function is symmetric.

Therefore, we conclude that INSANE dramatically simplifies the development of a lightweight messaging system that, as the next Chapter will show, also outperforms currently available alternatives. Additionally, LunarMoM is portable across all supported networking technologies, making it a promising solution for data dissemination at the network edge. LunarMoM is still a prototype, but we believe it shows how existing messaging systems could leverage INSANE to significantly improve their performance and portability.

LUNAR Streaming Framework

In edge cloud scenarios, users often have to deal with applications involving real-time streaming and analysis of huge amounts of data, such as intelligent applications based on ML or image processing. Especially in an industrial environment, we can easily be faced with a type of application where, during the manufacturing process, a series of cameras take images of the product during different stages of production. These images are usually transmitted in real-time to a central computing node. If defects are detected in the semi-finished product, the control systems might interact with the production line to reactively handle the failure.

Such real-time streaming applications can be designed in a client-server manner, where one or more clients ask to receive a stream of data, and the server sends them adapting the bit-streams according to network and QoS requirements [127]. To support their QoS requirements, streaming applications frequently exploit data fragmentation and/or compression techniques. For our prototype, called *Lunar Streaming*, we use only fragmentation, leaving compression as future development, as it is outside the scope of our framework.

Lunar Streaming exposes a simple set of APIs, starting with `lnr_s_open_server` to open the server-side application and with `lnr_s_connect` that allows clients to connect to it. Thus, the server application must implement a simple interface by exposing two methods: `get_frame` and `wait_next`. The first allows to get a new frame, while the second pauses the server waiting for the next frame. To start streaming, the server application must invoke `lnr_s_loop` which performs the following steps: (i) requesting a new frame (ii) fragmenting and sending the frame and (iii) waiting for the next frame to restart the loop until the end of streaming.

Implementing a full stack of streaming protocols is beyond the scope of this work, but this Section demonstrated that the INSANE interface allows the creation of even complex applications,

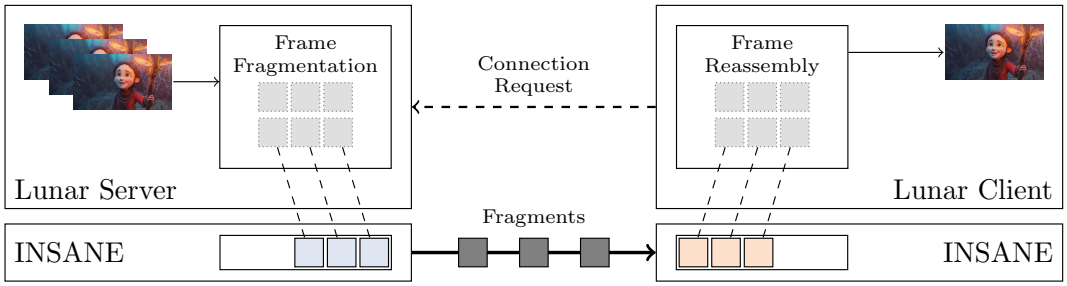


Figure 6.11: Lunar Streaming framework application.

such as a streaming framework, with few lines of code. The evaluation proposed in Section 7.4 will then show how that simplicity is not at the expenses of performance, which remains close to that of the raw network acceleration technologies.

6.3.5 Use case: a framework for Industry 4.0 applications

The lack of a cloud support capable of providing high-performance I/O operations has been a significant obstacle to the pervasive adoption of the cloud paradigm in several domains. That is particularly clear in the case of Industry 4.0 (see Section 2.4.2), in which the increasingly pervasive digitalization still falls short of expectations of a full cloud integration. This Section discusses how the INSANE middleware is a promising solution to bridge this gap, and can be even used in combination with the 5G cellular infrastructure for more complex usage scenarios.

Industry 4.0 and the role of vPLCs

The increasing amount of scattered data produced by machinery and the necessity of analyzing them is rapidly pushing companies to replace or adapt machine field technologies from proprietary *ad hoc* industrial protocols to open and more flexible standards, enhancing the automation level and the cohesion between Operation Technologies (OT) and Information Technologies (IT) in a cost-effective and affordable manner by utilizing Commercial-off-the-Shelf (COTS) hardware and software. This has several benefits: increased community support, reduced maintenance effort, continuous updates, and improved cybersecurity. A noticeable example of such integration is the idea of Virtual Programmable Logic Controllers (vPLCs), which enhance the functionalities of a Programmable Logic Controller (PLCs) with the flexibility only virtualized software can guarantee. Historically, the introduction of PLCs was an essential building block of the *automation revolution* in industrial control systems. Nowadays, vPLCs stand as the ideal choice to embody the integration of OT and IT: coupled with containerization tehcnologies and general-purpose hardware, vPLCs integrate the flexibility of the microservice architecture, becoming even more portable and allowing migration of cloud services closer to the machine field.

However, the actual implementation of vPLCs and other IT-enabled components is still difficult to achieve: OT has demanding requirements in terms of latency, jitter, and Quality of Service (QoS), whereas IT is designed for best-effort behavior. As a consequence, current cloud-native virtualized controllers cannot offer the deterministic behavior and low network latency required by traditional specialized solutions or do so by sacrificing the generality of IT.

Overview of the solution

This Section proposes an open framework that combines a set of vendor-agnostic technologies to fully support the adoption of containerized PLCs in industrial control infrastructures. At the same time, the framework guarantees compliance with typical OT requirements such as deterministic network behavior and low-latency communication with the controlled devices. Within our framework, containerized vPLCs are managed by the Kubernetes orchestrator [32] and use the OPC-UA middleware interface [46] to distinguish traffic towards cloud-based nodes (*IT traffic*) and toward the controlled devices (*OT traffic*). The key novelty of our solution is a clear separation between the infrastructural support for those communications. Whereas IT traffic follows

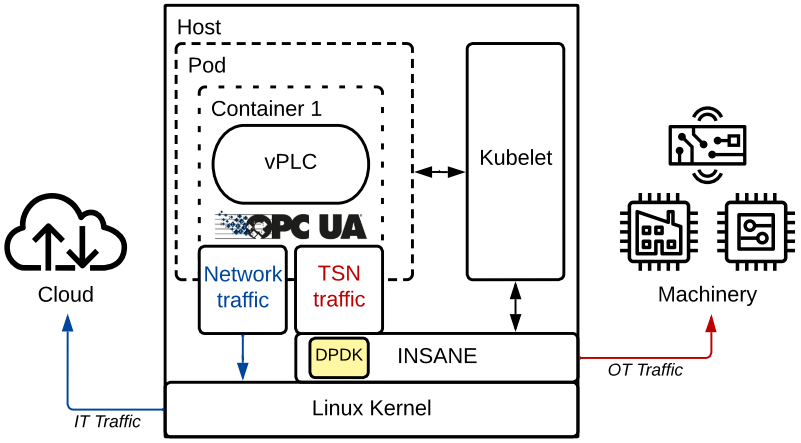


Figure 6.12: Overall Architecture of the vPLC Framework

the standard *best-effort* datapath of general-purpose operating systems, OT traffic relies on the INSANE runtime to remove typical I/O virtualization overhead and to provide determinism.

We begin the description of our solution from its core component, the vPLC, which we place within a container. That choice minimizes the overhead of virtualization while retaining its several benefits in terms of enhanced portability and scalability, isolated and reproducible environments, simplified dependency management, and improved resource efficiency. Containerization allows industrial control systems to benefit from orchestration tools: in our framework, we use Kubernetes as the orchestrator and insert the vPLC container in a Kubernetes *pod*, thereby ensuring its automated scaling, high availability, reliability, and efficient resource allocation in the control infrastructure.

As previously discussed, the hardest challenge for a framework supporting vPLCs is to fulfill their mixed-criticality communication requirements. On the one hand, the vPLC communicates with the IT infrastructure on a best-effort network, exchanging data with the cloud (or edge cloud). On the other hand, it must also interact with the controlled devices on the time-critical network fabric with no or minimal difference from traditional dedicated connections. Given the substantial differences between those two classes of traffic, we decided to provide them a corresponding substantially different infrastructural support, at the same maintaining programming transparency for software PLC programmers as well as compatibility with existing PLC software.

Middleware interface

We consider that vPLC adopts the OPC-UA middleware for both IT and OT communications [27, 73]. OPC-UA guarantees developers a single point of access to the network, transparent scalability for the interaction with the cloud, and also the rich and standard OPC information model to interact with the OT devices. Given this requirement, we decided not to use the INSANE

API - which is more suitable for newly-designed applications - but to layer the standard OPC-UA interface on top of the INSANE runtime, routing through it only the relevant traffic.

We leverage a specific configuration of OPC-UA, the TSN profile, to let developers signal time-critical traffic directed to the OT fabric (*OT traffic*) and thus requiring determinism and bounded latency, whereas we assume that best-effort guarantees suffice for any other communication (*IT traffic*). We route IT traffic (blue line in Figure 6.12) through the standard datapath of containerized applications on general-purpose operating systems, which makes packets cross various software layers before reaching the physical network. In parallel, we leverage INSANE to offer a high-performance, TSN-enabled datapath for OT traffic.

System support and cloud integration

INSANE is used in combination with a userspace virtual switch in the host (Figure 6.10 right) to create a virtual overlay network between the vPLC and its controlled devices: any packet sent on that overlay is handled by INSANE, which is configured to use the TSN-compliant scheduling policy that let applications associate an expected *transmission time* to each output packet. When this time comes, the scheduler will send the packet on an *accelerated* communication channel. This approach bypasses the performance overhead and the intrinsic variability of the standard in-kernel datapath, thus ensuring deterministic and low-latency communication.

Overall, this solution combines a set of open-source tools, protocols, and technologies to support the effective deployment of vPLCs as containers in cloud platforms. That would significantly reduce the development and operationalization cost of traditional PLCs, allowing much more flexibility, and guaranteeing the respect of the demanding performance requirements of OT. Furthermore, the use of open-source technologies protects our solution from new and hidden forms of vendor lock-in (e.g., the use of proprietary hypervisors). The next Chapter demonstrates these properties by running vPLCs within our framework over a real industrial testbed.

Ultra-Low Latency 5G Scenarios

The INSANE middleware presented in this Section, as well as all the implementations proposed in this thesis, mainly focus on *core* or *edge* datacenters, characterized by wired connectivity and thus more sensitive to any I/O overhead. However, the increasingly wider deployment and adoption of the 5G cellular standard provides the option for a ultra-reliable, low-latency communication option even on Wide-Area Networks. In the following, we briefly comment whether the INSANE implementation would allow containerized applications deployed in different locations (e.g., two factories) to communicate while also respecting Ultra-Low Latency (ULL) requirements.

A commonly-agreed definition is that ULL refers to the possibility of remote processes to communicate with sub-millisecond latencies [130]. If these processes are industrial controllers deployed in different facilities, this goal becomes challenging to guarantee and requires that each involved actor optimize its operations. In particular, application developers could generally assume the availability of no more than 40% of the total latency budget, leaving the remaining to the external provider operations in WAN. Figure 6.13 provides a graphical representation of this subdivision: if a subscriber should receive any update within at most 1 ms from its publication in a remote location, the application developer should consider to have no more than 400 µs available for its own network operations at both sides. Consequently, very often the current trend

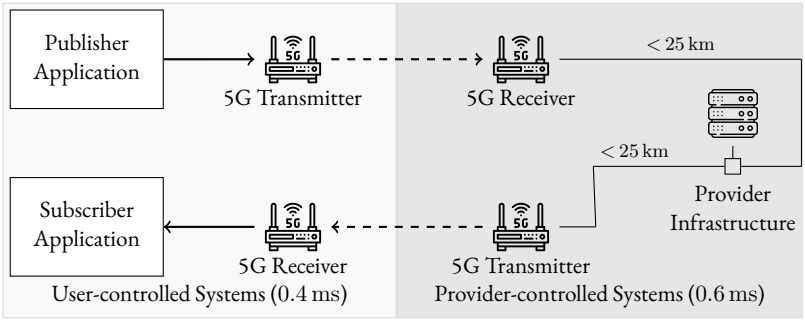


Figure 6.13: Typical latency budget distribution in ULL applications.

of applications in mission-critical domains is to try to minimize their share of network overhead, including that induced by resource virtualization, thus trading flexibility for performance.

In contrast, the new option for *NAaaS* provided by INSANE would allow developers to meet these stringent deadline without forgoing flexibility, by reducing the unpredictability of traditional I/O virtualization and bypassing the typical sources of overhead in the standard datapath. In the evaluation of the INSANE-powered vPLC, the next Chapter will briefly show that a virtualized industrial controller can communicate with remote devices and keep latency values well below the threshold set by ULL requirements, thus further proving how the INSANE support is a suitable tool for applications across the whole continuum.

6.3.6 Concluding remarks

This Section presented the INSANE middleware as the second reference system for this thesis. The choice of implementing the system layer as a userspace OS module is the distinguishing feature of that system and brings many advantages compared to DerechoDDS: not only a higher flexibility, but also a better integration with cloud platforms (I/O virtualization) and with standard cloud tools for operation automation, such as Kubernetes.

Coupled with a general-purpose, easy-to-use interface, the design of INSANE is indeed very effective in providing the option for *NAaaS* in cloud platforms across the whole continuum. To prove that, this Section first introduced two applications (a MoM and a streaming framework) that showcase the ease of programming deriving from the novel INSANE API, as well as its generality. Then, a more complete use case was presented: the implementation of a PLC, a crucial component of industrial automation, as a containerized software application that still manages to meet very demanding performance requirements. Furthermore, these advantages can be coupled with the emerging cellular standards (5G and beyond) to support even larger-scale use cases.

6.4 Conclusion

This Chapter presented two reference implementations of the general architecture for *NAaaS* proposed in this thesis: DerechoDDS and INSANE. Although the discussion involved all the three architectural layers, the key difference between them is at the system layer. DerechoDDS, which follows an approach based on the *libraryOS* model, is effective in supporting accelerated

applications, even beyond its traditional employment: the DIFFUSE serverless platform demonstrates that. By strictly coupling the offered system features with their implementation from an acceleration technology, the resulting system is indeed very efficient especialy for applications with strong performance requirements and little need for dynamicity. Conversely, the INSANE middleware adopts a *OS module* approach. This choice introduces additional IPC overhead for the interactions between applications and the INSANE runtime, but the possibility to centralize the I/O processing in the runtime also makes a much more efficient use of the system resources, is much easier to support in cloud platforms, and enables multiple applications to dynamically attach and detach from the runtime.

Whereas the discussion in this Chapter focused mostly on the design choices of the presented implementations, on the application portability they enable, and on their cloud integration, Chapter 7 will propose their experimental evaluation and that of the associated use cases, thus proving that they also preserve the performance properties of the native acceleration technologies and their advantages over the currently existing solutions.

7 Experimental Assessment and Results

This Chapter reports the experimental evaluation of the two reference implementations described in Chapter 6 and of the systems and frameworks proposed for the associated use cases. Whereas the previous discussion mostly focused on the architectural aspects of these systems and on their design with respect to different system models, here these systems are considered under a performance perspective to validate the claim that they can provide *NAaaS* by adding only a minimal datapath overhead.

The discussion is structured as follows. After a description of the experimental settings, both DerechoDDS and INSANE are first evaluated in isolation, to investigate their performance properties, and then in comparison with similar systems on basic performance metrics such as latency and throughput. In the second part, the solutions proposed for the two use cases of Serverless Computing and Industry 4.0 are considered.

7.1 Experimental testbed

The performance evaluation in this Chapter was mainly conducted on three different testbeds, whose specifications are reported in Table 7.1. The first two match typical *edge cloud* environments. In the first setting, two nodes are directly interconnected in order to minimize the overhead of network operations and magnify any system-induced delay on the measured metrics. In the second one, eight nodes are used, but equipped with a less powerful processor and NIC. Instead, the third scenario represents a typical *core cloud* infrastructure, where we reserved two nodes interconnected by a switch in order to have complete control on all the performance-related aspects.

Testbed	CPU	RAM	NIC	Switch
Edge Cloud 1	18-core Intel i9-10980XE @ 3.00GHz	64GB	Mellanox DX-6 100Gbps	—
Edge Cloud 2	10-core Intel E5-2640v4 @ 2.4GHz	64GB	Mellanox DX-4 25Gbps	Dell Z9264F-ON
Core Cloud	32-core AMD 7452 @ 2.35GHz	128GB	Mellanox DX-5 100Gbps	Dell Z9264F-ON

Table 7.1: Setup of the and public testbeds for the evaluations.

These relatively resource-rich testbeds are realistic for our target scenarios and, at the same time, answer the practical need of pushing the middleware systems to their limit. For instance, given the high performance of the considered acceleration technologies, a less powerful NIC would capped the bandwidth and artificially flattened the differences among different acceleration options.

7.2 Microbenchmarks

The goal of this first Section is to investigate the basic performance properties of the two systems. In both cases, the main metrics are the network *latency* and the *throughput* achievable in different scenarios. To put these numbers in perspective, these systems are also compared with existing alternatives, highlighting the differences that the emerge and discussing their causes.

7.2.1 DerechoDDS evaluation

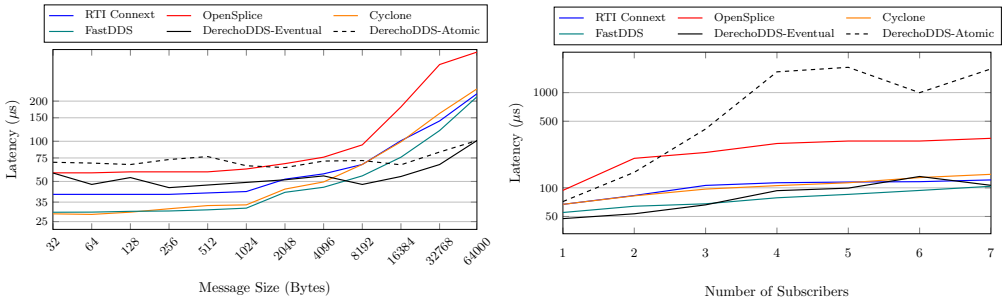
The goal of the DerechoDDS evaluation is to demonstrate that not only the added consistency guarantees do not harm performance when compared with weak consistency, but also that DerechoDDS can match the performance limits of the hardware even in the strongly consistent mode, and can ride out periods when the network briefly becomes fully saturated. We believe that this opens the door to use of SMR even in today's most demanding mission-critical scenarios, as well as in all those environments that need these guarantees coupled with high network performance. All the tests in this Sections were performed on the *Cloud Edge 2* testbed, by selecting for all the topics the DerechoDDS *accelerated* QoS that maps communication to RDMA.

Latency and Throughput

The first evaluation of DerechoDDS is a performance comparison against other four commercial DDS implementations that we selected for their widespread adoption in the community. Two are mature products: RTI Connext 6.0 [108] and Adlink OpenSplice Community 6.9 [2]. The other two are emerging implementations, Eclipse Cyclone DDS 0.7 [37] and eProsima FastDDS 2.3 [38]. Our comparison includes a latency and a throughput test, each repeated at least five times for different payload sizes and for different numbers of subscribers. The single publisher and the subscribers are all on different nodes to stress the network performance. We used *volatile* durability, *reliable* reliability, and UDP multicast as the transport protocol to obtain a consistency level equivalent to the *eventual* consistency of DerechoDDS on RDMA.

The latency test is a simple ping-pong application designed to highlight any overhead in the DDS send and receive pipeline. This test measures the round-trip time (RTT) of every sample published on a "ping" topic and received by a remote subscriber which sends it back to the publisher. In case of multiple subscribers, the first "pong" message is considered. We run this test for 60 seconds, disabling sample batching and using a *keep-last-1* history.

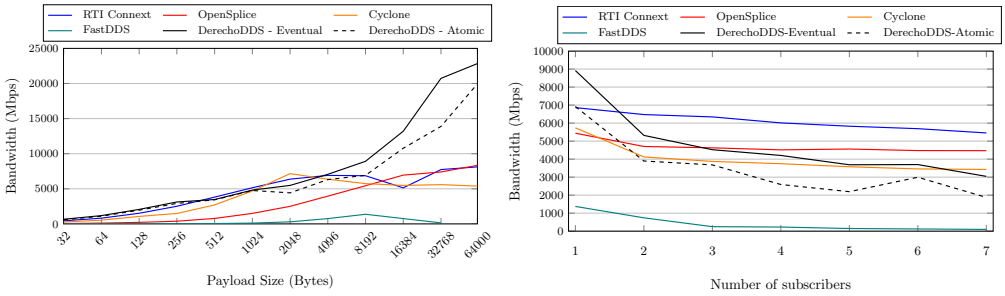
With small messages in a single subscriber scenario (Fig. 7.1a), DerechoDDS in *eventual* consistency mode exhibits approximately 50% higher latency than the best UDP-based alternatives, even though it still performs better than some of the products. On average, the strong consistency guarantee adds another 40% performance penalty for those sizes. These numbers could be substantially improved with a careful optimization of our prototypical implementation, as RDMA



(a) Increasing payload sizes with 1 publisher and 1 subscriber

(b) 8KB payload size and an increasing number of subscribers

Figure 7.1: Median Round-Trip Time for different payload sizes and numbers of subscribers.



(a) Increasing payload sizes with 1 publisher and 1 subscriber

(b) 8KB payload size and an increasing number of subscribers

Figure 7.2: Average throughput for different payload sizes and number of subscribers.

is incredibly sensitive even to tiny delays [62]. But this pattern only holds up to 4KB. With message sizes higher than 8KB, the situation is reversed: even the *atomic* mode has a 2x lower median latency than the alternatives: this is where the true potential of RDMA emerges, as protocol-induced delays becomes negligible. If we scale the 8KB case to more subscribers (Fig. 7.1b), we see that the performance advantage is substantially preserved for the eventually consistent case, but that DerechoDDS with its atomic guarantee incurs a delay while waiting to ensure that the SMR properties have been achieved. This delay rises to as much as 1 ms, but then stabilizes and remains constant as the number of subscribers is increased from 4 to 7.

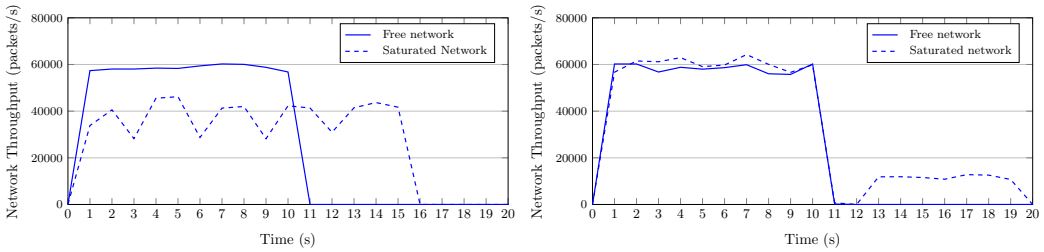
The throughput test asks whether each DDS can fully saturate available bandwidth when a publisher continuously updates a topic with one remote subscriber: a crucial capability on high-speed networks. Fig. 7.2 plots the results. We observe that FastDDS is much slower than the others. This product lacks data batching, putting it at a substantial disadvantage. DerechoDDS on RDMA almost saturates the available bandwidth for bigger message sizes, a result impossible to obtain when using UDP and the *reliable* QoS: the traditional networking stack cannot handle such a high throughput, so many packets are lost and the retransmission cost increases. We also observe that the *atomic* mode in DerechoDDS does not suffer an excessive overhead, and for 64KB payload size it is still 2.3x faster than the best existing DDS implementation.

Reliability

This experiment compares DerechoDDS with RTI Connext (fastest among the four DDS products). One publisher writes small (128 byte) critical updates with *reliable* QoS. DerechoDDS always runs in reliable mode, but is additionally configured with *atomic* consistency. We picked a constant data publishing rate of 60K samples per second, which guarantees that no packets are lost by RTI Connext, and publish for 10s. Both systems deliver all messages within 11s (Fig. 7.3).

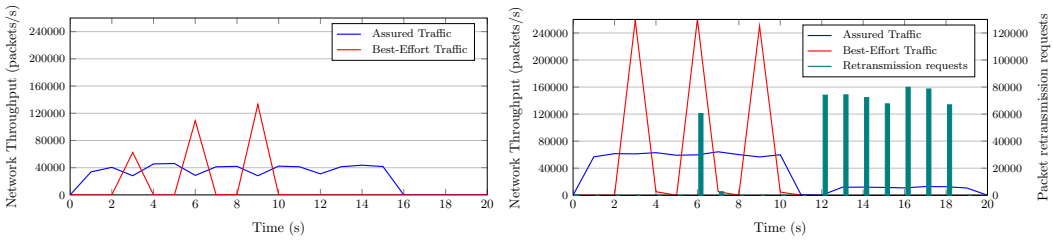
Next, we introduce a second publisher and a second remote subscriber on the same network. These produce and consume low-importance data, expressed using the *best-effort* reliability for RTI Connext and *eventual* consistency for DerechoDDS. In our first experiment, we configure the second producer to generate a steady rate of background traffic designed to fully saturate the network link when both publishers are running at once (dashed lines). We see that DerechoDDS obtains a reduced share of the network, requiring 16 seconds to complete the transmissions, but then is finished. In contrast, RTI exceeds the peak network capacity, causing some packets to be dropped because of congestion. In reliable QoS mode, these must later be retransmitted, so we see a series of retransmission requests (blue bars) and a second wave of deliveries, ending after 20s.

As a final experiment we reconfigure our second publisher to be bursty: it pauses for 2s, then sends rapidly for 1s. Fig. 7.4 plots the results. RTI Connext sends at full speed regardless of network load, causing a high rate of lost packets, so the subscriber issues many retransmission requests and the total test time once again jumps from 11s to 20s. DerechoDDS runs at a slightly lower bandwidth but with no loss: the RDMA hardware has a built-in mechanism that only transmits data when the receiver is ready for the incoming bytes. Fig. 7.3a shows that although the data



(a) DerechoDDS over RDMA (b) RTI Connext over UDP multicast

Figure 7.3: Throughput of the critical traffic flow under different network conditions



(a) DerechoDDS over RDMA (b) RTI Connext over *reliable* UDP multicast

Figure 7.4: Impact of bursts of low-importance traffic on a critical traffic flow.

rate of the critical flow drops from 60K to 40K packets per second, the experiment completes in 16 seconds, 20% faster than for RTI. We should note that RTI connect offers a proprietary API with which the application can explicitly throttle its rate of publications. A knowledgeable user could configure the two DDS applications (critical and background) to prevent loss in this experiment. We did not evaluate this option because it is not automated: the application designer must anticipate the congestion conditions and specify the peak rate of transmission for each topic.

Overall, the evaluation of DerechoDDS shows that even in its strongest QoS configuration, communications on the *accelerated* topics of DerechoDDS is highly efficient, equaling or exceeding the bandwidth of existing DDS products while also reducing latency.

7.2.2 INSANE evaluation

For this evaluation, we build a C prototype of the INSANE runtime that supports two network technologies, namely kernel-based UDP and DPDK. The integration of RDMA and XDP is ongoing work, but we prioritized the two former options because these are the most commonly adopted in the edge cloud ecosystem: unlike RDMA, they do not require special hardware, are easy to use from cloud environments, and yet are representative of the differences between kernel-based and kernel-bypassing networking.

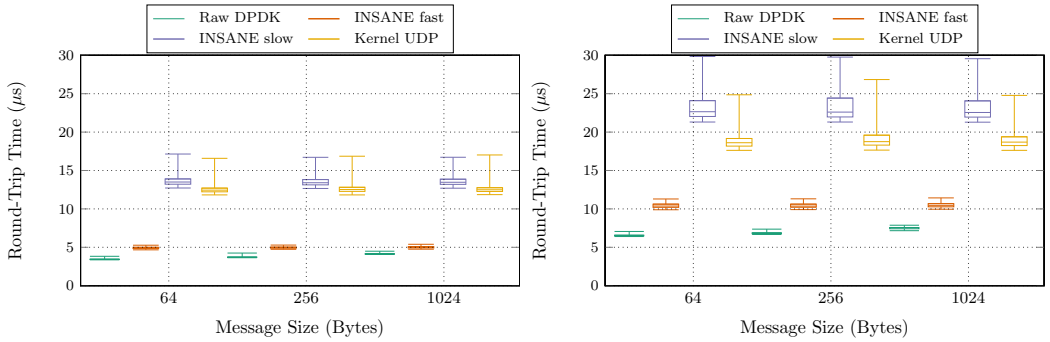
Our evaluation of INSANE focuses proving that the agnostic features provided by the system layer actually introduce minimal overhead compared to the native communication technologies. To put the results in perspective, we compare INSANE to Demikernel [131], the most complete and state-of-the-art alternative option to transparently access kernel-bypassing technologies, and show that the additional dynamicity provided by INSANE comes with comparable or even better performance. Furthermore, we also consider the packet scheduling strategies and show that our time-sensitive policy allows to achieve a more predictable network behavior.

Latency and Throughput

To demonstrate that INSANE introduces a minimal overhead compared to using each native technology directly, we build a benchmarking application for latency and throughput. For latency we used a simple *ping-pong* application designed to highlight any overhead in the send and receive pipeline. It measures the round-trip time (RTT) of a single message sent from one host and immediately echoed back by a remote receiver. We repeat this test for 1 million messages. The throughput benchmark is a *stress test* application that evaluates how much of the available network bandwidth is practically achievable when a sender continuously sends 1 million messages at full speed to a remote receiver. We measure throughput as the amount of payload data (*goodput*) received in the time unit. We run every throughput experiment 10 times. We implement

Interface	**Lines of Code (LoC)**	**Increase**
INSANE	189	—
UDP socket	227	+20%
DPDK	384	+103%

Table 7.2: LoC to implement the benchmarking application.



(a) Edge Cloud Testbed 1

(b) Core Cloud Testbed

Figure 7.5: Round-Trip Time (RTT) for increasing payload sizes.

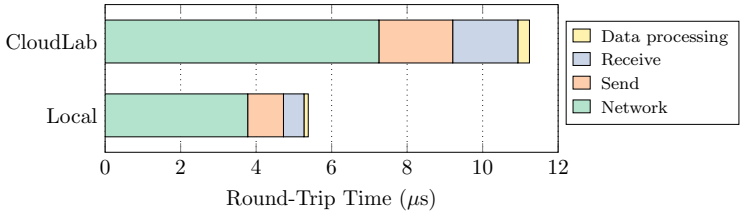


Figure 7.6: *INSANE fast* latency breakdown (64B)

the benchmarking application in three versions: one that uses UDP sockets, one that uses native DPDK, and one that uses the INSANE API. First, even for such a simple benchmarking application, INSANE minimizes the amount of code necessary for networking, as Table 7.2 summarizes, without requiring developers to understand the details of each technology.

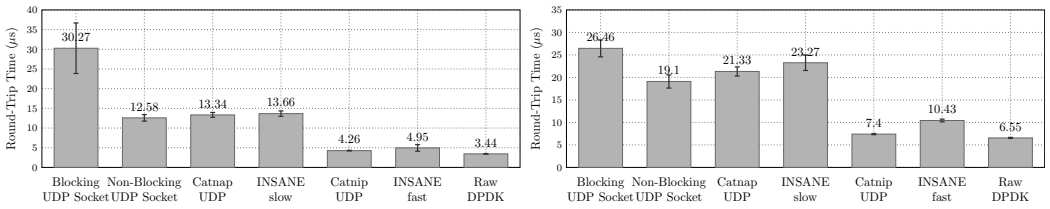
Figure 7.5a and Figure 7.5b report the latency of INSANE for increasing payload sizes when using two different datapath acceleration QoS: *slow*, which maps network operations to UDP sockets, and *fast*, which maps to DPDK. Overall, we note that there is no significant difference among different payload sizes. In the *Edge Cloud* testbed, we observe that *INSANE fast* keeps very close to raw DPDK, with an increase of the median RTT values of at most 1 µs. The same gap separates *INSANE slow* from the pure kernel-based UDP benchmark. Hence, we can conclude that INSANE introduces on average a 500 ns overhead on each UDP packet both in fast and slow mode. In the *Core Cloud* setup, we note a general increase in RTT values, as we expect, because of the introduction of a switch between the two hosts. According to our measurements, the switch adds on average 1.7 µs and packets must traverse it twice. However, INSANE's latency increases more than expected, adding around 1.7 µs to the raw DPDK median values. We investigate this increase by breaking the latency value into its main components in Figure 7.6. In addition to the expected increase of the network latency, we also observe a significantly higher time spent by INSANE in the send and receive operations.

The culprit of this behavior is that the processor on the cloud servers is significantly slower than in our edge testbed[1]. Although INSANE tries to minimize the processor intervention on

[1] `https://www.cpubenchmark.net/high_end_cpus.html`

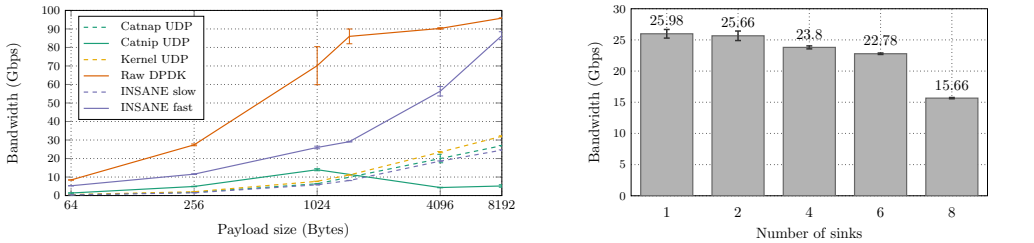
the critical path, the requirement to support multiple applications running as separate processes makes it hard to further reduce the amount CPU cycles required for internal operations. This overhead could be reduced by parallelizing the datapath plugins over multiple polling threads in order to better leverage the multi-core capabilities of modern processors.

To put our INSANE performance in perspective, in Figure 7.7 we expand our latency experiments to include a wider range of systems, reporting the average RTT for 64B payload size, so to consider a challenging case where any protocol overhead is magnified. In particular, we include two versions of the pure UDP socket benchmark, one with blocking receive, and one that continuously polls a non-blocking socket. Without surprise, we note that the former is much slower than the latter, as process wake-ups are costly in terms of latency. Furthermore, we implement the same test using Demikernel [131], binding it to two of the libraries it offers: *Catnap*, which maps network operations to kernel-based sockets, and *Catnip*, which maps to DPDK. Those libraries correspond to INSANE with *slow* and *fast* datapath QoS respectively. We observe that Catnap is slightly slower than the native socket application in both testbeds. *INSANE slow* has almost the same performance as Catnap in the edge cloud setup, and 1.9 µs slower on average in the core cloud setting. If we consider DPDK, we observe the same trend discussed in the previous paragraph. On the edge testbed, *INSANE fast* adds 690 ns to Catnip's latency, which in turn adds 820 ns to the raw DPDK performance. When we consider the performance in the cloud, all the latencies increase. However, unlike *INSANE fast*, Catnip preserves almost the same gap to raw DPDK. Indeed, Demikernel has a much simpler logic to deliver the payload to applications, as it is a library compiled with the application. *INSANE fast* suffers more from the slower processor, but its runtime still shows a competitive latency performance despite the additional dynamicity it can offer to multiple concurrent applications.



(a) Edge Cloud 1 (b) Core cloud

Figure 7.7: Average RTT of raw network technologies, INSANE, and Demikernel for 64B payload size.



(a) Throughput for increasing payload size. (b) Throughput for increasing number of sinks (1KB)

Figure 7.8: Throughput benchmark for INSANE and the other reference systems.

Although latency is a crucial metric in edge cloud, applications also expect to fully leverage the available network bandwidth when they need to quickly transfer big data payloads, e.g., camera images for remote analysis. In this case, we found no significant performance difference between the two testbeds; hence, we only report data for the edge cloud testbed. Figure 7.8a evaluates the throughput of *INSANE fast* and *INSANE slow*, comparing it with the corresponding Demikernel libraries, with kernel-based UDP sockets, and with raw DPDK for increasing payload size. To avoid the fragmentation overhead, we enable jumbo frames for payloads bigger than 1.5KB. We observe that raw DPDK can quickly saturate our NIC, as it does not perform any data processing. Despite the need for inter-process communication, *INSANE fast* shows the second best performance, reaching peaks of 90 Gbps for the biggest payload; whereas Catnip shows a significantly lower throughput. This difference reflects a different use of the underlying DPDK library: Catnip is optimized for latency [131] and sends one packet per time on the network. Conversely, INSANE adopts a form of *opportunistic batching* [65, 62] at sender side, similar to the technique presented in Section 3.5: messages ready for send are sent as a batch, but never waiting for a fixed-size batch to fill up. This way, we reach the highest throughput under intense traffic without harming latency significantly, as shown in the previous paragraphs. Indeed, when we do not adopt this technique, like in *INSANE slow*, we observe that Demikernel and INSANE perform in the same way.

Finally, one of the distinguishing points of INSANE, modeled after the OS module approach, is that it can support multiple applications on the same host at the same time. In Figure 7.8b we repeat the throughput test by increasing the number of sinks connected to the runtime on the receiver host, listening on the same *channel id*, but from separate applications. The plot reports the average throughput received by all the sinks for 1KB of payload size. We note that for up to 6 concurrent sinks, the average received throughput drops only by 8 % compared to the single-sink solution. A significant degradation starts to emerge with 8 sinks (−39 %), a number of co-located applications that we consider unusually high for a typical edge context.

Overall, our experiments demonstrate that INSANE can achieve µs-scale latencies and tens of Gbps bandwidth utilization, showing competitive or even better performance than other kernel-bypassing systems, on different environments, despite the added dynamicity, portability and flexibility it offers to developers. Even better, we showed that INSANE can serve multiple concurrent applications with no or minimal performance degradation.

Deterministic network behavior

A core component of INSANE is the packet scheduler, which in the previous experiments was operating using the default FIFO strategy. Here, we want to assess whether this INSANE component can effectively provide deterministic guarantees to time-sensitive flows in a cloud environment. Hence, we consider again the latency test, but instead of the RTT between two remote applications here we focus on the one-way latency, assuming that the two nodes of the *cloud edge 1* testbed are synchronized (see Section 3.4). Figure 7.9 represents the deployment adopted for this test: two remote applications, a publisher and a subscriber, deployed as containers on the two physical hosts. We use INSANE with its DPDK plugin, and a userspace version of Open Virtual Switch (OVS) which in turn uses DPDK to bypass the kernel network stack.

The goal of this test is to measure the network *determinism*, i.e., whether a packet expected by the subscriber at a certain time succeeds in meeting its deadline: the publisher sends a UDP

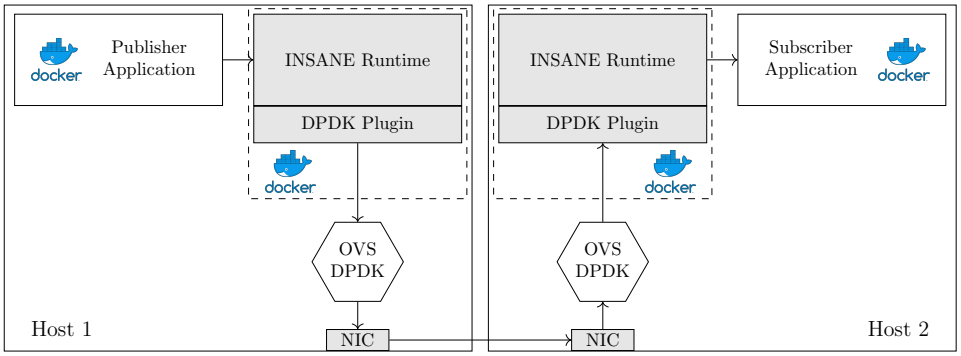


Figure 7.9: Container-based deployment of INSANE on the *Cloud Edge 1* testbed.

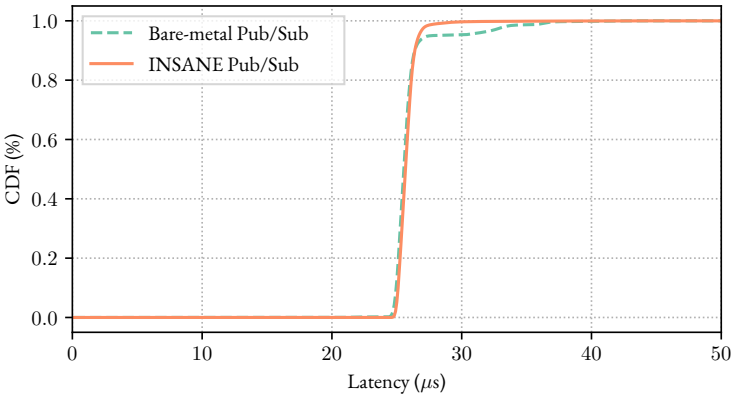


Figure 7.10: CDF with packets of 256 bytes.

packet every 1 ms, hence the subscriber expects to receive it with the same periodicity. To compare the results, we also consider a *bare-metal* application, where the publisher and the subscriber are regular applications that send/receive using the kernel-level datapath and TSN scheduler.

Figure 7.10 plots the Cumulative Distribution Function (CDF) of the experiment results for the two cases, INSANE-based and kernel-based applications. 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 INSANE show overlapping performance, very close to the ideal behavior. In particular, for INSANE the 90 % and the 99 % probability correspond to 26.4 µs and 28.1 µs respectively. These absolute latency values are higher than those presented in the previous Section: the reason is the adoption of OVS, a software-based solution for I/O and network virtualization.

In the top part of the graph, we also observe that the INSANE-based deployment is more precise than the bare-metal one: that is the effect of a dedicated packet scheduler, whereas the kernel-based datapath and scheduler are source of additional unpredictability. These results show that INSANE is able to provide the same guarantees than the kernel-based support for determinism, while also retaining all the advantages we previously discussed.

7.3 Use case: Serverless Computing

This Section presents the performance evaluation for the DIFFUSE serverless platform, which was described in Section 6.2.3 as a use case for the DerechoDDS middleware. The overall goal of this evaluation is to show that the strongly-consistent shared memory abstraction provided by DerechoDDS not only simplifies the creation of function pipelines, but also brings significant performance advantages compared to other MoM solutions.

In particular, the three considered MoM solutions (Kafka, Redis, and the DerechoDDS-based DSMQueue) are evaluated under three representative workloads: (i) a constant-rate stream of requests, (ii) a stream of incoming requests issued at an increasing rate, and (iii) a large batch of requests submitted to the system in a small amount of time. The first workload aims to assess the properties of our serverless platform in a steady regime of incoming requests, whereas the second and the third scenarios reproduce a typical traffic pattern that arises when a high number of concurrent events need to be processed in batch (e.g., process all the tweets with a specific hashtag).

For this evaluation, we employ a lightweight, short-lived business logic with an execution time of about 60 µs. Upon termination, the last function of each pipeline appends a timestamp to its output, later on used to compute the different metrics. Response times are measured as the time-lapse between the moment the request is issued and the termination timestamp of that function (*end-to-end* latency). We define as *throughput* the number of satisfied requests per unit of time. We examine how these metrics, as well as the total execution time, vary under an increasing composition length. In this assessment, we vary the number of composable functions from 2 to 5, and each function is packaged as a distinct container, although embodying the same business logic. Also, the application graphic is a linear path, hence no branching logic is considered.

The experiments are conducted on the *Cloud Edge 1* testbed. On each machine, we run a single instance of the function invoker, which has access to the code of the function to be executed in the experiment. On one of the nodes, we also run a traffic generator process, which we use as a trigger to simulate different ingress traffic patterns: the trigger forwards the invocation requests to the invokers using the MoM.

We configure Kafka with at-least-once semantic to avoid the overhead introduced by transactions in an exactly-once mode, and for the same reason, the number of partitions in the topic is set equal to the number of nodes with a replication factor of 1. Redis was deployed as a single instance in one of the two nodes and set-up in order to create a Redis Stream with one single active group, i.e., function invokers cooperate to consume a different portion of the same stream of messages. Finally, we configure DSMQueue to replicate the shared-memory queue across a group of three processes, the trigger and the two invokers. We configure the underlying DerechoDDS support to enforce strong consistency across the replicas and to use RDMA-accelerated topics, and to keep the shared state in volatile memory, with no persistence support. In the following, we discuss the experimental results and the trade-offs that emerge.

7.3.1 Constant-rate stream of incoming requests

In this first experiment, we would like to investigate the system behavior under a steady regime. Hence, the trigger issues a fixed number of requests at a constant rate of 1000 requests/second, and the experiment is run by varying the length of the function composition from 2 to 5.

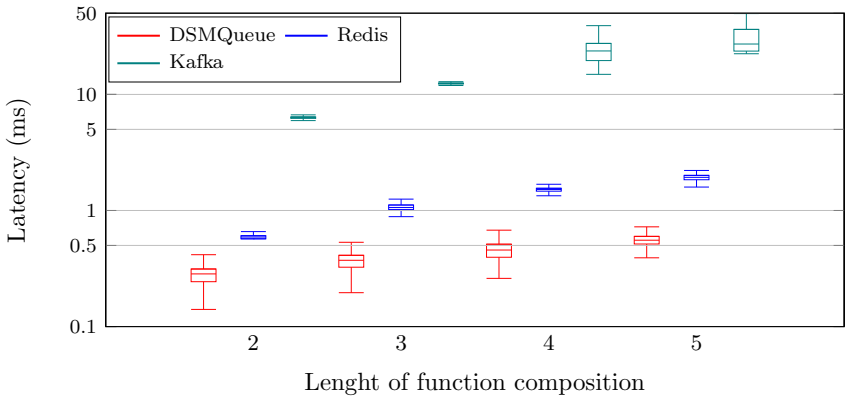


Figure 7.11: End-to-end latency at a steady regime. Note that the logarithmic y-axis magnifies the whiskers bars in the case of DSMQeue.

Figure 7.11 shows the end-to-end latency of each execution as a function of the composition length. Note the logarithmic scale in the y-axis, which magnifies the length of the whiskers for the smaller values. We can observe two important trends. First, as expected, the latency increases with the composition length. This increment is generally attributed to the time taken to execute more functions, and the time spent in the (de)queuing operations. At this request rate, the MoMs can sustain the traffic with little to no queueing effects, hence the delay contribution is mainly to be attributed to networking and synchronization of concurrent requests. In all configurations, the latency increment is linear, but there are important differences. For DSMQeueue, the median latency shows a 2x increment when switching from 2 to 5 functions: much of it is the function execution time, whereas only 33% is caused by additional middleware operations. This increment is more evident in Redis, which demonstrates a higher (3x) latency increment between 2 and 5 functions. As the function execution time is constant, the additional latency time is caused by the middleware operations, which in this case account for the 86% of the total increment. Kafka exhibits similar behavior, but with an even higher increment factor (4x).

The second consideration is about the relative performance of the different middleware solutions. For the simplest case of two functions in the composition, DSMQueue shows the best median latency (284 µs). While Redis is able to keep up (2x slower), Kafka demonstrates an order of magnitude higher latency (22x slower). The amount of these latency gaps increases as the composition length increases: for a composition of length 5, Redis is 3.2x worse than DSMQueue (554 µs), and Kafka is again out of scale (49x higher latency).

In conclusion, the DerechoDDS-based DMSQueue with RDMA support outperforms the other alternatives, which rely on traditional TCP/IP networking and higher-layer constructs to implement advanced capabilities. Kafka adopts a similar semantic to the other MoMs (see Section 6.2.3), yet it shows the worse performance by far, and this is to be attributed to its default message/topic persistency support. DSMQueue and Redis have a more similar architecture, but Redis is between two and three times slower in this context.

7.3.2 Incremental rate stream of incoming requests

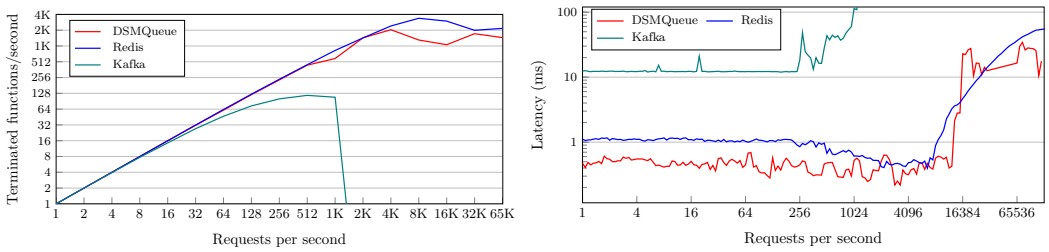
Herein, we would like to assess system scalability by subjecting the platform to an increased rate of incoming requests, varying from 1 to 65K requests/seconds. In this scenario, the composition length is kept constant to 3, representing a common option in real-world scenarios e.g., simple map-reduce operations etc. Figure 7.12 shows the end-to-end throughput and latency (y-axis in log scale) of the proposal under a varying rate of incoming requests.

Figure 7.12a shows that the different configurations gracefully scale up the resources to keep up with demand, and throughput increases linearly up to a certain inflection point before starting to decline. This critical point corresponds to the maximum input rate that a middleware can sustain without queuing any request: after a threshold, new requests begin to queue up, competing with the existing invocation requests, using up the available resources. The critical rate is similar for DSMQueue and Redis, which start to queue requests between 8K and 12K requests/second, whereas this behavior emerges much earlier in Kafka, at about 240 requests/second.

As one may expect, the competition for resources between incoming and enqueued requests has a direct effect on latency (Figure 7.12b). Up to the critical input rate, DMSQueue shows a better end-to-end latency than Redis, averaging about 500 µs versus 1 ms. Shortly after the critical rate, DSMQueue shows an increasingly oscillatory effect, whereas, surprisingly, Redis exhibits a decline in latency. Finally, between 8K (Redis) and 16K (DSMQueue) requests/second, performance degrades rapidly and reaches a similar regime of much higher latency (tens of ms), although DSMQueue still demonstrates a much better behavior. Finally, we observe that Kafka, even in low request regimes, demonstrates an order of magnitude higher latency than both Redis (10x slower) and DSMQueue (20x slower).

Overall, DSMQueue performs well in terms of latency (2x) and has comparable throughput to Redis. This behavior remains constant up to a critical ingress rate, as well as for the highest input rates, while the behavior of both systems becomes unstable during the transition between those two phases. In addition to the motivations provided for the constant-rate stream experiment, it is noteworthy to point out that the DSMQueue *zero-copy datapath* fully manifests its benefits as the message size grows. This leads to extra spare time, not spent on copying data.

On the other hand, the poor performance of Kafka is to be attributed to the MoMs consistency mechanism used to maintain a distributed, structured and durable commit log of events: any request - ingress data to functional components of the chain - must be acknowledged prior to serving successive ones. Considering the high ingress load and the additional load generated by



(a) End-to-End throughput

(b) End-to-End latency

Figure 7.12: End-to-end latency and throughput with a varying rate of ingress traffic.

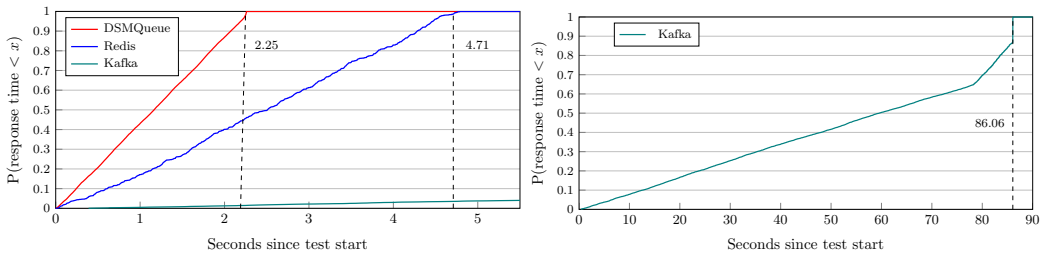
intermediate results of function executions, the topics acquire an increasing backlog of requests (events) subject to the dynamics of the commit log. As a consequence, the invoker entities tasked with the execution of functions and output serialization to Kafka are subjected to ever-increasing waiting times, expecting an acknowledgment from the broker. This in turn results in a lower end-to-end chain throughput with a respective spike in terms of latency. The specific interval where the phenomenon manifests itself is tied to the current testbed characteristics (CPU, RAM etc.): to mitigate it, could rely on the topic partitioning feature of Kafka, distributing the load among cluster nodes according to design-time criteria. It is noteworthy to point out that in our current setting, Kafka is configured with an "at-least-one" semantic, more optimistic in terms of performance with respect to an "at-most-one" semantic.

7.3.3 Burst of incoming requests

In this experiment, we assess the behavior of the platform when subjected to a sudden burst of concurrent requests. To this end, our trigger produces a burst of 10K invocation requests at the highest possible sending rate. This way, we intentionally exacerbate the queuing effect described for the previous experiments: the message queue will fill up with invocation requests, as the invokers will not be able to consume them at the same rate. We keep the composition length constant to 3 functions: this further stresses the queue, as per our architecture each pipeline execution requires the invokers to produce and consume new requests to and from the queue.

In this setting, we are interested in the total time the system takes to consume the entire batch of concurrent requests. Figure 7.13a breaks down the total execution time by plotting the Cumulative Distribution Function (CDF) of the pipeline execution time.

In this case, the different behavior of the considered systems depends on the different waiting times between the execution of two consecutive functions. Such waiting time is determined by the different communication overhead introduced by each solution, which directly affects the speed at which they process the backlog of requests. In particular, the trend that we observe is the same we described for the previous experiment. The shared memory approach of DSMQueue is the fastest in processing the request batch (2.25 seconds), Redis takes about twice that time (4.71 seconds), and Kafka is an order of magnitude slower (86.06 seconds) as shown in Figure 7.13b.



(a) MoM comparison for a composition of length 3

(b) Kafka performance in isolation for a composition of length 3

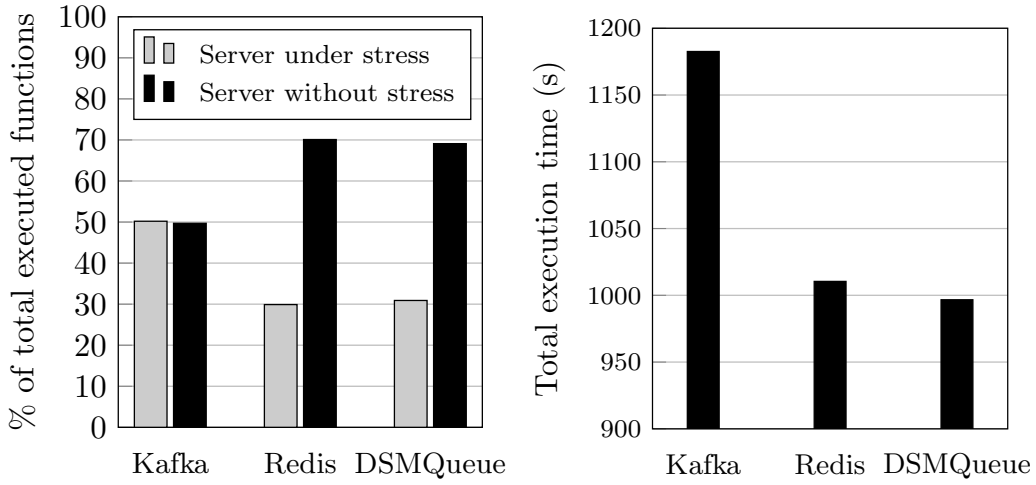
Figure 7.13: Response time CDF with varying MoM support.

7.3.4 MoM enabled load balancing

In this last experiment, we investigate how the different properties of the three MoMs (Kafka, Redis, and the DerechoDDS-based DSMQueue) impact the distribution of the pipeline workload across the available nodes. Indeed, one driving motivation for this work is to enable DIFFUSE to scale across a varying number of hosts and to efficiently use all the available resources. To effectively measure such efficiency, we are interested in how the workload is distributed when the available machines are subject to different load conditions.

To this end, we run the same experiment discussed for the burst experiment, but this time the adopted function is more computationally expensive than the one in the prior experiments, totaling an average execution time of 60 ms. Also, to better highlight the differences between the MoMs, one of the two available hosts executes a background application, saturating its computing, memory, and disk resources. This way, we expect that the same function will take a different execution time depending on the host it is executing on: in one case, the function will compete with the background application to acquire the necessary resources, whereas those will be immediately available on the other host. We want to understand if and how each MoM takes the server load condition into account when deciding, transparently to the user, how to distribute the incoming workload.

Figure 7.14a shows the results. As expected, the same function on the two hosts takes a significantly different amount of time to complete: on average, 35 ms on the idle one, and more than twice, about 85 ms, on the other. We observe that Redis and DMSQueue execute about 30% of the workload on the saturated server, leaving almost 70% of it to the idle machine. On the contrary, Kafka assigns the same number of functions to both hosts. This different behavior is directly linked to the way each MoM implements the queue abstraction. In DSMQueue and Redis Stream, the queue is a (logically) single FIFO buffer that processes compete to access, either when producing or consuming new data. In our setting, these processes correspond to the two



(a) Distribution of functions across the hosts

(b) Total test execution time

Figure 7.14: Load balancing behavior of the different MoMs

invokers. Since the function execution on the idle host takes approximately half of the time taken on the saturated one, the invoker on the idle host ends up consuming more than twice the number of functions than the invoker on the saturated host: an indirect form of load balancing induced by the load on each node. Kafka, instead, blindly follows a round-robin scheme, assigning an equal number of functions to each host. While this approach eliminates the need for coordination among the invokers - which no longer need to compete to access the queue - it also does not take the actual server load into account. As a consequence, many more functions are scheduled on the saturated host, leaving unused resources on the idle host: this is clearly inefficient, and it results in a significant increase in the time needed by Kafka to process the function batch (Figure 7.14b).

Even though Redis and DSMQueue already provide an implicit form of load balancing, an explicit mechanism could lead to faster function execution times, and, as a consequence, to improvements on the overall system throughput. The development of a new load balancing mechanism requires the introduction of an observability layer, providing real time information on the resource usage.

Overall, these results show that networking techniques like RDMA may bring significant performance advantages and enhanced QoS guarantees. At the same time, systems based on standard networking interfaces represent a valid alternative in environments with more conventional settings or specific constraints on development, deployment, or scale of the infrastructure. Because DerechoDDS gives both options, we conclude that it is a suitable support for applications with heterogeneous requirements, even for uses cases not generally served by major commercial implementations.

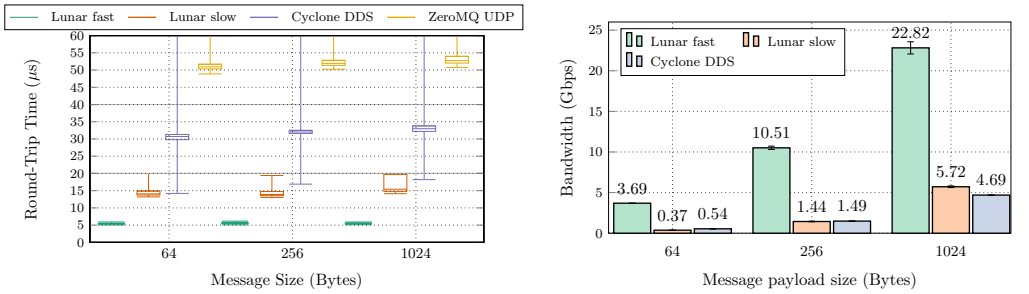
7.4 INSANE-based LUNAR applications

To prove the ease of programming and of I/O acceleration provided by the INSANE interface, Section 6.3.4 introduced two applications: LUNAR MoM and LUNAR Streaming. The evaluation proposed in this Section aims at supporting the claim that the provided simplicity is not traded for performance, and that applications that are developed using the INSANE high-level interface can effectively achieve the native performance of I/O acceleration technologies.

Lunar MoM

To evaluate the performance of LunarMoM, we compared LunarMoM against two widely used decentralized messaging systems in that environment, Cyclone DDS [37] and ZeroMQ [60]. We configured these systems to use a UDP transport and conducted two performance benchmarks: a ping-pong test, to measure the round-trip time between a publisher and a remote subscriber, and a throughput test, to evaluate effective bandwidth utilization. The tests were conducted on the *Edge Cloud 1* testbed.

The results, as shown in Figure 7.15a, indicate that LunarMoM has the lowest latency in both fast (using DPDK) and slow (using UDP) modes. Compared to the raw INSANE performance (Figure 7.5a), we observed that LunarMoM adds ns-scale overhead to INSANE, resulting in stable low latency. The performance of Cyclone ($+45\,\%$) is comparable to that of systems that use blocking sockets in their receiver thread, although with higher variability. ZeroMQ's UDP support, on the other hand, adds additional 20 µs latency compared to Cyclone. Similar consider-



(a) Latency of MoMs for increasing payload sizes.

(b) Throughput of MoMs for increasing payload sizes

Figure 7.15: Performance benchmark for Lunar MoM and other reference systems.

ations apply to the throughput evaluation (Figure 7.15b), where DPDK allows LunarMoM to significantly increase bandwidth utilization, while Cyclone and LunarMoM slow have similar behaviour. ZeroMQ showed unstable performance and was excluded from the graph.

In conclusion, our experimentation demonstrates that INSANE dramatically simplifies the development of a lightweight messaging system that outperforms currently available alternatives, with ns-scale latency overhead compared to the INSANE interface. Additionally, LunarMoM is portable across all supported networking technologies, making it a promising solution for data dissemination at the network edge. LunarMoM is still a prototype, but we believe it shows how existing messaging systems could leverage INSANE to significantly improve their performance and portability.

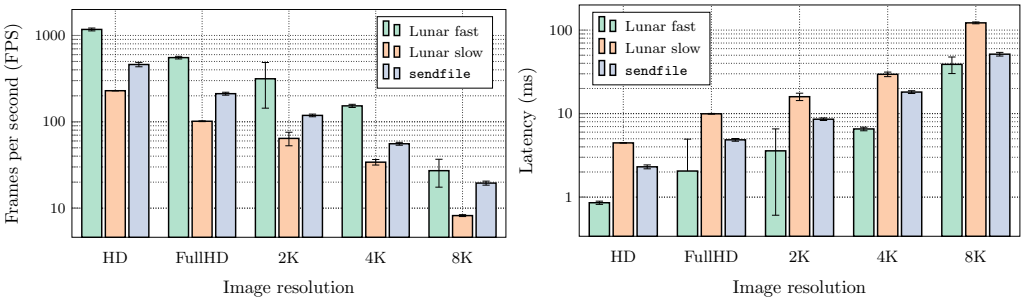
LUNAR Streaming framework

To test Lunar Streaming we implemented a simple application that streams raw images, i.e., for each image frame we send RGB values for every single pixel (Figure 6.11). We use sample images of different common sizes (Table 7.3) and compare our INSANE-based implementation with one that uses the `sendfile` primitive. Since `sendfile` sends data directly from a file descriptor loaded into the kernel without involving user space, it actually implements a sender-side zero-copy technique. For this reason, we believe it can be a good reference for our framework.

To demonstrate the performance of our streaming prototype, we evaluate: (i) the number of frames per second (FPS) the client application can handle (Figure 7.16a), and (ii) the average end-to-end latency for frame transmission (Figure 7.16b), i.e., the time between the server application sending a frame (including fragmentation) and the client application receiving the reconstructed frame. As we can see Lunar streaming allows very good results in both latency and FPS, especially in the fast case. For the latter, the system consistently performs better than the `sendifle` version.

Resolution	HD	Full HD	2K	4K	8K
Size (MB)	2.76	6.22	11.6	24.88	99.53

Table 7.3: Size of the images sent in the streaming benchmark.



(a) FPS for increasing image resolution.

(b) Latency per frame for increasing image resolution

Figure 7.16: Benchmark for Lunar Stream and `sendfile`.

In particular, for images up to 4K, we can support frame rates above 100 FPS, and even above 1000 FPS in the case of low-quality images. Latency never exceeds 10 ms for images up to a maximum resolution of 4K, making Lunar streaming an excellent candidate in applications such as tactile internet [130] or real-time simulations (e.g., cloud gaming [79]). Hence, even just by sending raw images, we obtained excellent results: we emphasize that INSANE can be easy to use and effective in accelerating existing streaming frameworks [4].

7.5 Use case: Industry 4.0

This Section is dedicated to the evaluation of the Industry 4.0 use case presented in Section 6.3.5, in which the INSANE middleware - in particular, its system layer - is used to support the execution of a virtualized industrial controller within an edge cloud platform. To make this solution acceptable, however, it is necessary to demonstrate that the proposed deployment can meet the demanding performance requirements of this industrial controller.

To this end, we evaluate the performance of a vPLC application running within our framework with a twofold purpose. First, we want to assess the *virtualization overhead* introduced on the

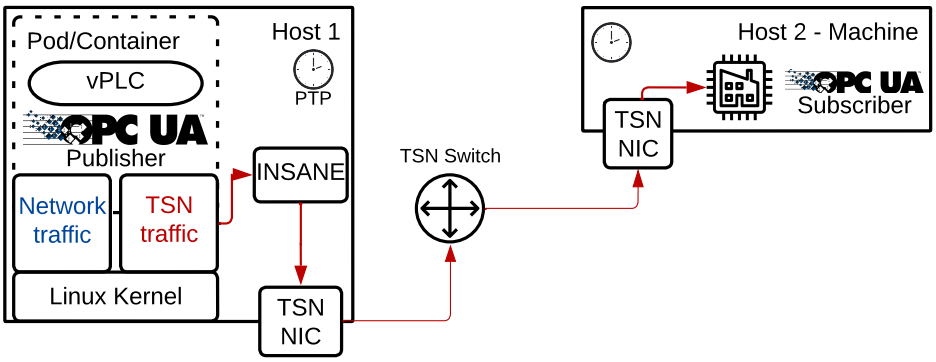


Figure 7.17: Schematic representation of the vPLC testbed.

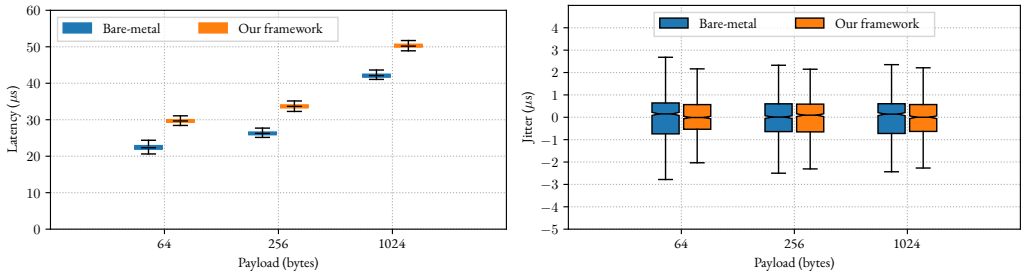
network by the use of containers. Hence, we compare the behavior of the same vPLC application running in two configurations (supported by INSANE in a cloud edge, and bare-metal) in a real industrial testbed, represented in Figure 7.17. Second, we evaluate the *compliance* of the results with the requirements of the strictest industrial communication scenarios.

For the purpose of this evaluation, we implement a simple software PLC that we consider a black-box, as we only investigate its networking performance, and that runs in a Docker container. Internally, the vPLC implements an OPC-UA publisher using the open-source OPC-UA implementation open62541 [100]. The evaluation analysis is conducted on the *Cloud Edge 1* testbed. As shown in Figure 7.17, on one host we deploy the INSANE runtime through Kubernetes. As in previous evaluations, we use the INSANE DPDK plugin. On the other host, on the same local network, we run an OPC-UA listener to reproduce the behavior of an industrial device. The two nodes and the switch are synchronized using the PTP protocol, as required by TSN. In particular, the two nodes run the linuxptp implementation and are configured as PTP slave clocks, where the switch works as the PTP master clock of the network.

7.5.1 Virtualization Network Overhead

In this first part, we evaluate the *virtualization overhead* associated with containerization by comparing the performance of the vPLC (1) containerized within our framework and (2) running bare-metal on the same hardware. In both cases, the vPLC is configured to publish OPC-UA messages with a cycle of 25 µs, a typical value in the most demanding industrial scenarios (see also Section 7.5.2). 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 transmission time set by the publisher and the actual reception time by the OPC-UA subscriber. 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, $T = 25\mu s$).

Figure 7.18 reports the end-to-end latency and jitter measured for the two considered cases and for three typical payload sizes (64 B, 256 B, 1024 B). A first consideration is that the performance of the containerized version of the vPLC is always very good (orange boxes in Figure 7.18a), with median latency values ranging from 29.7 µs in the case of small packets (64 B) to 50.1 µs for



(a) Latency. (b) Jitter.

Figure 7.18: Performance of the test vPLC running bare-metal (green) and within our framework (orange). The experiment is repeated for increasing payload sizes.

1024 B. These values are very close to those registered for the bare-metal deployment, showcasing a constant difference of about 7.6 µs, whereas latency variability is negligible in both cases. The constant performance difference originates in the additional network steps required for packets to reach the network in the containerization case: instead of being directly sent and received on the wire, in our framework they have to cross the INSANE scheduler and a virtual switch, as well as a VXLAN encapsulation step and vice-versa on the receiver side. Nevertheless, the overhead of these steps is minimal.

The performance strength of our approach is even clearer by considering jitter, reported in Figure 7.18b. The median value is around 0 in all cases, as expected on a deterministic network, but the variability, although minimal, is lower in the containerization case. This is the effect of the INSANE scheduler, already commented in Section 7.2.2: a userspace TSN scheduler introduces less variability than the standard kernel-based version, even in this small-scale experiment with no background traffic to introduce noise.

From these results, we conclude that containerization in our framework introduces minimal overhead in terms of network latency, and even improves determinism by supporting the OT traffic with a more efficient packet scheduler. In the next paragraph, we comment on how these results are suitable for the most demanding industrial control applications.

7.5.2 Industrial Communication Compliance

We now briefly comment on whether our framework effectively meets its design goals of flexibility and high-performance support for virtualized control applications. On the flexibility side, we execute vPLCs in Docker containers managed by Kubernetes, on a general-purpose operating system and COTS hardware, adopting standard communication protocol stacks. These are all open-source resources easy to integrate with IT platforms: hence, we consider meeting the goal of an open and vendor-independent framework for vPLCs. On the performance side, previous work [55] considers that the most demanding industrial applications, such as closed-loop motion control, require cycles under 1 ms with a jitter of at most 1 µs. Our evaluation proves that vPLCs within our framework can support even significantly shorter cycles (25 µs), with a jitter below the 1 µs for more than 90% of the times (Figure 7.18b), despite not being co-located with the controlled machines as in traditional PLC deployments. Therefore, our framework successfully enables vPLCs to also meet the strictest performance requirements of the OT traffic, thus paving the way for full integration of OT and IT in the next-generation industrial control infrastructures.

7.5.3 Ultra-low Latency 5G scenarios

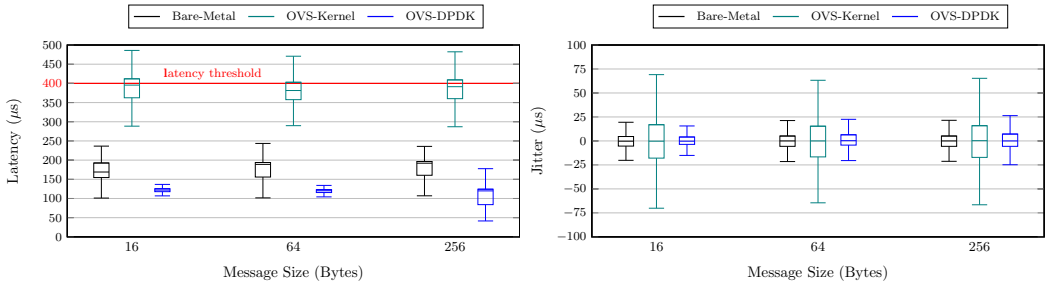
Section 6.3.5 briefly commented on the new possibilities emerging with the increasingly wider deployment of 5G (and beyond) standard cellular networks. These technologies promise to support very low latencies across remote locations: even in the demanding scenario of Industry 4.0 it is possible to envision use cases with component distributed across the continuum and still able to meet Ultra-Low Latency requirements. Practically, we previously considered that in such a scenario the application developers would have a very limited latency budget for their overall network operations, which is generally considered to be no more than 400 µs considering all the involved locations.

To prove that our INSANE middleware is capable of meeting even these demanding requirements, we set up an experiment similar to that presented in Section 7.5. However, instead of using containers, we considered that VMs would generally be used in this more distributed case. Hence, our testbed *Cloud Edge 1* was configured of a VM on one host, hosting a publisher application supported by INSANE, and a VM on the other host, where the subscriber is hosted, in turn supported by INSANE. The goal of the test is to demonstrate that the one-way latency between these two application components remains below the ULL threshold of 400 µs. As in the previous configuration, INSANE is configured to use the DPDK plugin and to adopt a TSN-compliant packet scheduling strategy, and we use the OVS DPDK implementation to set up a software datapath on the host.

As a comparison, we consider two other possible configurations. First, a kernel-based virtual switch instead than OVS-DPDK to create an overlay network between the two machines. Second, we also run the tests by deploying the publisher and the subscriber on the bare-metal hosts.

Performance results

Figure 7.19 and Figure 7.20 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. 7.20 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 the median values reported in Fig. 7.19a 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. 7.19b). 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. 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.



(a) End-to-end latency

(b) End-to-end jitter

Figure 7.19: End-to-end latency and jitter for different payload sizes and network virtualization techniques.

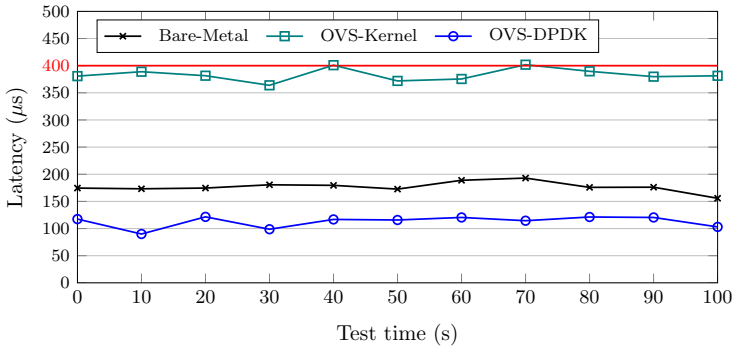


Figure 7.20: End-to-end latency averaged every 10 seconds, for 64 bytes payload size. The red line is the latency threshold.

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 Figure 2.2 in Section 2.3). 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, 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 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 INSANE-based 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.

In conclusion, these performance results demonstrate that applications that rely on INSANE can fulfill ULL constraints even when executing in virtual machines. We also noted that 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.

7.6 Conclusion

The key requirement to enable *NAaaS* in cloud platforms is that user applications access heterogeneous high-performance I/O with minimal overhead and through a uniform interface. Through a thorough experimental assessment, this Chapter demonstrated how the two reference implementations of the architecture proposed in this thesis, *DerechoDDS* and *INSANE*, can indeed achieve those goals. The evaluation also showed that they can support a wide spectrum of applications across the whole continuum, combining the advantages of the cloud model with the performance of I/O acceleration. In the following, we briefly summarize the key insights that derive from our experimental evaluation.

- **Portable high-performance**. The microbenchmarks for *DerechoDDS* and *INSANE* show that these systems introduce only ns-scale overhead compared to the raw network technologies supported, in line with the requirements of modern µs-scale applications in the cloud continuum. At the same time, *DerechoDDS* and *INSANE* offer such performance through a uniform API, making applications portable across heterogeneous deployment scenarios. In particular, the design chosen for the *INSANE* implementation (userspace OS module) adds further dynamicity, enabling multiple applications to dynamically attach and detach to the system, as required, for instance, during the live migration of a cloud service.

- **Ease of use**. The uniform and high-level interface exposed by *DerechoDDS* and *INSANE*, according to the proposed architecture, makes it possible to easily develop new applications that automatically inherit the performance benefits of the supported I/O acceleration technologies. The description and evaluation of the *INSANE*-based LUNAR applications demonstrate both the ease of use of the proposed interface and the associated performance advantages.

- **Reduced overhead for strong properties**. The availability of an easy-to-use accelerated datapath makes it possible for applications to amortize the overhead typically associated with offering strong properties. We showed that by considering two examples. First, the *strong distributed consistency* of *DerechoDDS*. Then, the *determinism* of the virtual PLC, designed for a performance-critical *Industry 4.0* scenario.

- **Support to heterogeneous QoS requirements**. The plugin-based design of the proposed architecture allows applications to adapt their behavior to heterogeneous constraints in terms of development, deployment, or scale of the available infrastructure. As discussed in particular for the *serverless* use case based on *DerechoDDS*, different I/O technologies can be suitable for different requirements: applications that use an implementation of our proposed architecture can dynamically and transparently adapt to heterogeneous and changing requirements.

The next Chapter will consider future research directions to improve two key aspects covered by this thesis: the perspectives for a better integration of the proposed reference implementations with cloud platforms, and the open challenges that currently obstacle this possibility.

8 Conclusion and Future Work

This Chapter concludes this thesis by summarizing the lessons learned from the design and the implementation of an architecture for Network Acceleration as a Service for the cloud continuum. The implementation of two reference systems and their deployment in support of heterogeneous use cases raised several research questions of broad interest as the system community shifts to consider the network edge as an integral part of the cloud computing ecosystem. In that setting, the emerging network acceleration technologies promise to enable new reactive applications even far from centralized datacenters, as those introduced in this work, but they also bring additional heterogeneity in an ecosystem that already struggles to define standard system practices.

The following discussion summarizes the most important open challenges toward the goal of *NAaaS* by distinguishing two planes: architectural and infrastructural. From the architectural perspective, each layer poses interesting problems to system developers aiming to ease access to I/O acceleration for the average cloud users. From the infrastructural perspective, various obstacles currently prevent a full integration of I/O acceleration technologies in cloud platforms. Finally, possible future extensions of this thesis work are discussed.

8.1 Architectural challenges

The discussion in this Section considers the three layers of the system architecture proposed in Chapter 5 and, based on the implementation experiences reported in this thesis, reviews the open challenges for each of them.

8.1.1 Interface layer

The heterogeneity and complexity of the native interfaces of I/O acceleration technologies have pushed researchers to define agnostic interfaces, like those proposed in this thesis, that provide efficient yet transparent access to these options. The discussions reported in the previous Chapters highlighted the need for different applications to have agnostic interfaces at different abstraction layers (see Section 4.2). However, many competing options are available even at the same abstraction layer and most of them are relatively recent in literature. This lack of standardization, although natural at such an early stage of the research on this topic, is potentially an obstacle to application portability.

The OMG DDS standard API, which is supported by one of the reference systems described in this work, is an exception to this consideration because it was standardized more than a decade ago [88]. Nonetheless, it suffers from a problem of high complexity and developers often refrain from its use. On top of that, it was also never specifically designed for the asynchronous programming model of modern I/O acceleration technologies, as demonstrated by the absence of an appropriate QoS policy (see Section 6.2).

Therefore, as the use of acceleration technologies becomes more and more common to support the emerging class of cloud applications, the definition of asynchronous, easy-to-use, and standard access interfaces will be a paramount concern for developers. By combining the most successful features of previous proposals, the interface introduced in this thesis for the INSANE middleware goes in this direction and supports these properties by design.

8.1.2 System layer

The system layer decouples the agnostic interface layer from the technology-specific implementation of the plugin layer. To enforce this separation, this thesis introduced a set of agnostic system features, which are designed after the common principles of the I/O acceleration options but independent of the actual mechanisms that provide them: a zero-copy memory manager, a thread scheduler, and a packet processing engine. However, the actual implementation of these features in the two reference systems described in Chapter 6 raised several issues in terms of isolation and security, observability, and multi-core datapath processing.

Isolation and security

The availability of a zero-copy datapath is paramount for the performance of accelerated I/O options. However, in multi-tenant environments, such as cloud platforms, it is also crucial to enforce strict isolation between the customers' code, which is considered untrusted, and the system resources, including memory. Currently, system developers willing to leverage acceleration technologies must re-implement typical system features within their code and thus require the direct visibility of system resources, which is unacceptable for providers. The availability of memory management as a service, as proposed in this thesis, is a first but insufficient step for providers to regain control over resources: a simple programming error may easily break isolation and provide access to other applications' memory. The definition of more advanced and efficient memory isolation mechanisms is still an open research problem, as well as, more broadly, the safety and security of acceleration technologies in shared environments [106, 118].

Observability

By bypassing the Operating System kernel for the sake of performance, I/O acceleration technologies also bypass the standard observability tools typically implemented in the kernel. In the cloud, that also means losing the management and monitoring tools typically implemented in virtual switches (see Section 2.3). Although this thesis did not specifically investigate this aspect, a future research direction will be the introduction, at the system layer, of an agnostic feature for that purpose, which the plugin layer would then need to specialize for each technology. That would offer cloud users not only the option for a transparent I/O acceleration but also a transparent interface to observe the behavior of I/O acceleration.

Multi-core processing

The implementation of both DerechoDDS and INSANE maps a single *plugin* (i.e., the code that manages the I/O operations in a specific technology) to a single thread. This choice has many

advantages, especially for the small-scale applications that were supported for the performance evaluation: reduced resource consumption, optimal use of data and instruction cache, and absence of synchronization issues. However, in particular when the system implementation follows an OS module approach, the possibility to leverage the modern, powerful multi-core processors for asynchronous end-host I/O operations is appealing. However, multi-core threading strategies are also very hard to leverage, as the detailed study in [78] demonstrates. This thesis leaves the investigation of multi-threaded strategies for high-performance end-host networking to future work.

8.1.3 Plugin layer

The role of the plugin layer is the implementation of the agnostic I/O operations exposed to end-users with the actual mechanisms provided by the native interfaces of the acceleration technologies. A practical yet fundamental problem revealed by the actual implementation of the two reference systems is that when large amounts of data must be sent on the network, a form of fragmentation, at some level of the network stack, is unavoidable. However, although some of the considered network technologies support zero-copy packet *fragmentation*, only RDMA is currently capable also of zero-copy packet *reconstruction*. In all the other cases, the plugin code at the receiver must copy the payloads of the incoming fragments to their final memory destination, which is provided by the memory manager in the system layer. The introduction of plugin-level support for fragmentation risks choking the receive pipeline with time-consuming data copies for reconstruction. That contrasts with the goal of *true* zero-copy transfers. To avoid this issue, the INSANE prototype currently does *not* support UDP/IP packet fragmentation: we resorted to the use of jumbo frames for tests with the biggest payload sizes, following the same approach as Demikernel [131], or to application-level data fragmentation and reconstruction. The definition of a technique for zero-copy data reconstruction remains an open research challenge that will probably require the availability of programmable hardware offloading mechanisms to be solved.

8.2 Infrastructural challenges

The integration of I/O acceleration options in cloud platforms at any level of the cloud continuum is still a hot research topic. Section 2.3 surveyed several proposals on the virtualization of RDMA-based applications, all striving to balance the need to preserve the raw hardware performance with the typical dynamicity of software programmability. From our experience with the INSANE cloud deployment (Section 6.3), none of the proposed virtualization solutions emerged as a clear winner. Approaches based on a software virtual switch introduce CPU overhead into the data path, which slows down performance and wastes valuable provider resources. Hybrid approaches reduce this overhead but sacrifice a considerable degree of control on the data plane, which would likely be unacceptable for major cloud providers. Yet, both the above solutions are cost-effective and retain the flexibility advantages of a software-based control path. On the contrary, solutions based on specially designed hardware, such as AccelNet [31, 44], offload the data plane operations to a custom device while also providing a high degree of control and programmability. However, these approaches require a high upfront investment for research, development, and deployment that only major datacenter providers can afford, and propose highly customized

solutions difficult to generalize. Recent products released by major hardware manufacturers (e.g., NVIDIA [84]) have started to include commodity network virtualization functionalities, but their use is currently still very limited.

The architecture proposed by this thesis introduces a novel perspective toward the availability of *NAaaS*: because applications access I/O as a service from the system layer, the actual virtualization mechanisms are hidden from users. That simplifies the provisioning of efficient acceleration support, without performance compromises thanks to the plugin-based implementation of the zero-copy datapath: the evaluation in Chapter 7 proves that this model allows even software-based I/O virtualization strategies to meet stringent performance requirements. However, a hardware-based approach would be even more effective in minimizing the overhead of the control plane intervention on the data plane, but that would be sustainable only if proper support be available from hardware manufacturers, without requiring expensive customization, especially when adopting acceleration options at small-scale in edge cloud environments.

To this end, it appears unavoidable that only close cooperation among all involved actors can solve the open challenges that still clash with the goal of *NAaaS*: cloud providers and hardware manufacturers are called to re-design their proposals and products to address the needs of next-generation cloud platforms. On the one hand, cloud providers need to properly re-design their infrastructures according to the core principles of kernel-bypassing and hardware-accelerated techniques. On the other hand, hardware manufacturers should explicitly support the *cloudification* of their products, by enhancing the existing device features with built-in support for crucial aspects such as I/O and network virtualization, observability, and security.

8.3 Future Work

The architecture for *NAaaS* proposed in this thesis embraces the trend of *resource specialization* that is emerging globally in response to the pervasive digitalization of society. This trend has pushed cloud infrastructures, built on large-scale general-purpose hardware, to show their limitations. More broadly, the growing success of expensive, specialized, and fragmented technologies is leading to an overall *decline of general-purpose technology* [121]. In this context, the integration of such specialized technologies *as a service* into next-generation cloud platforms might obtain the crucial effect of mitigating and even reversing the negative effects of this trend, mainly related to an excessive fragmentation of the computing landscape. The availability of specialized, up-to-date technology through a standard, easily accessible, and widely used computing paradigm would enable global organizations to meet their increasingly demanding requirements through a homogeneous point of access without committing to massive upfront investments. That would also give cloud providers the economical power to drive the evolution of hardware accelerators according to their needs, such as cloud-friendly built-in support for resource virtualization.

This thesis contributed to this integration effort by focusing on *end-host networking* as the most evident example of this specialization trend, fueled by the rapid recent advancements in networking hardware. However, the architecture presented in Chapter 5 is actually generally applicable to any form of I/O acceleration, including storage and computing technologies. Therefore, future work will focus on adapting the reference implementations presented in Chapter 6 to include any form of I/O acceleration. This research direction is consistent with a new vision for

next-generation cloud platforms based on a more radical separation between a data plane, almost completely offloaded to specialized accelerators, and a control plane, running on CPU, that offers them as a service to the applications. By providing the broader option for *Acceleration as a Service*, next-generation cloud platforms will be able to embrace the current specialization trend and make it scalable and sustainable, technically and economically, for the wide and general public of cloud users across the whole continuum.

Bibliography

1. Nasir Abbas et al. "Mobile Edge Computing: A Survey". *IEEE Internet of Things Journal* 5:1, 2018, pp. 450–465. DOI: 10.1109/JIOT.2017.2750180.
2. Adlink. *Vortex Opensplice*. URL: https://www.adlinktech.com/en/vortex-opensplice-data-distribution-service.
3. Marcos K. Aguilera et al. "Remote regions: a simple abstraction for remote memory". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 2018, pp. 775–787. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/aguilera.
4. Aaro Altonen et al. "Open-Source RTP Library for High-Speed 4K HEVC Video Streaming". In: *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*. 2020, pp. 1–6.
5. Amazon. *Amazon EC2 P4 Instances*. URL: https://aws.amazon.com/ec2/instance-types/p4.
6. Amazon. *Amazon AWS High Performance Computing*. URL: https://aws.amazon.com/hpc.
7. *An Introduction to Redis Streams*. URL: https://redis.io/topics/streams-intro.
8. Michael Armbrust et al. "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Association for Computing Machinery, Houston, TX, USA, 2018, pp. 601–613. ISBN: 9781450347037. DOI: 10.1145/3183713.3190664. URL: https://doi.org/10.1145/3183713.3190664.
9. InfiniBand Trade Association. *InfiniBand Architecture Specification*. Release 1.0. 2000.
10. Infiniband Trade Association. *Supplement to InfiniBand Architecture Specification*. Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE). 2010.
11. Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey". *Computer Networks* 54:15, 2010, pp. 2787–2805. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2010.05.010. URL: https://www.sciencedirect.com/science/article/pii/S1389128610001568.
12. Microsoft. *Microsoft Azure N-series Virtual Machines*. URL: https://azure.microsoft.com/pricing/details/virtual-machines/series.
13. Microsoft. *Microsoft Azure High Performance Computing*. URL: https://azure.microsoft.com/solutions/high-performance-computing.

14. Emna Baccour et al. "Pervasive AI for IoT Applications: A Survey on Resource-Efficient Distributed Artificial Intelligence". *IEEE Communications Surveys & Tutorials* 24:4, 2022, pp. 2366–2418. DOI: 10.1109/COMST.2022.3200740.

15. Ioana Baldini et al. "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Ed. by Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya. Springer Singapore, Singapore, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8_1.

16. Ioana Baldini et al. "The Serverless Trilemma: Function Composition for Serverless Computing". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Association for Computing Machinery, Vancouver, BC, Canada, 2017, pp. 89–103. ISBN: 9781450355308. DOI: 10.1145/3133850.3133855. URL: https://doi.org/10.1145/3133850.3133855.

17. Luiz Barroso et al. "Attack of the Killer Microseconds". *Commun. ACM* 60:4, 2017, pp. 48–54. ISSN: 0001-0782. DOI: 10.1145/3015146. URL: https://doi.org/10.1145/3015146.

18. Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature, 2019.

19. Adam Belay et al. "The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane". *ACM Trans. Comput. Syst.* 34:4, 2016. ISSN: 0734-2071. DOI: 10.1145/2997641. URL: https://doi.org/10.1145/2997641.

20. Paolo Bellavista, Luca Foschini, and Alessio Mora. "Decentralised Learning in Federated Deployment Environments: A System-Level Survey". *ACM Comput. Surv.* 54:1, 2021. ISSN: 0360-0300. DOI: 10.1145/3429252. URL: https://doi.org/10.1145/3429252.

21. Ken Birman, Bharath Hariharan, and Christopher De Sa. "Cloud-Hosted Intelligence for Real-Time IoT Applications". *SIGOPS Oper. Syst. Rev.* 53:1, 2019, pp. 7–13. ISSN: 0163-5980. DOI: 10.1145/3352020.3352023. URL: https://doi.org/10.1145/3352020.3352023.

22. Ken Birman et al. "Invited Paper: Monotonicity and Opportunistically-Batched Actions in Derecho". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Shlomi Dolev and Baruch Schieber. Springer Nature Switzerland, Cham, 2023, pp. 172–190. ISBN: 978-3-031-44274-2.

23. Luiz Bittencourt et al. "The Internet of Things, Fog and Cloud continuum: Integration and challenges". *Internet of Things* 3-4, 2018, pp. 134–155. ISSN: 2542-6605. URL: https://www.sciencedirect.com/science/article/pii/S2542660518300635.

24. Christophe Bobda et al. "The Future of FPGA Acceleration in Datacenters and the Cloud". *ACM Trans. Reconfigurable Technol. Syst.* 15:3, 2022. ISSN: 1936-7406. DOI: 10.1145/3506713. URL: https://doi.org/10.1145/3506713.

25. Nicola Bonelli et al. "Programming Socket-Independent Network Functions with Nethuns". *SIGCOMM Comput. Commun. Rev.* 52:2, 2022, pp. 35–48. ISSN: 0146-4833. DOI: 10.1145/3544912.3544917. URL: https://doi.org/10.1145/3544912.3544917.

26. Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Association for Computing Machinery, Helsinki, Finland, 2012, pp. 13–16. ISBN: 9781450315197. DOI: 10.1145/2342509.2342513. URL: https://doi.org/10.1145/2342509.2342513.

27. Hugh Boyes et al. "The industrial internet of things (IIoT): An analysis framework". *Computers in Industry* 101, 2018, pp. 1–12. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2018.04.015. URL: https://www.sciencedirect.com/science/article/pii/S0166361517307285.

28. Rajkumar Buyya et al. "A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade". *ACM Comput. Surv.* 51:5, 2018. ISSN: 0360-0300. DOI: 10.1145/3241737. URL: https://doi.org/10.1145/3241737.

29. Qizhe Cai et al. "Understanding Host Network Stack Overheads". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. Association for Computing Machinery, Virtual Event, USA, 2021, pp. 65–77. ISBN: 9781450383837. DOI: 10.1145/3452296.3472888. URL: https://doi.org/10.1145/3452296.3472888.

30. M. Castro et al. "Scribe: a large-scale and decentralized application-level multicast infrastructure". *IEEE Journal on Selected Areas in Communications* 20:8, 2002, pp. 1489–1499. DOI: 10.1109/JSAC.2002.803069.

31. Adrian M. Caulfield et al. "A cloud-scale acceleration architecture". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783710.

32. Cloud Native Computing Foundation. *Kubernetes*. URL: https://kubernetes.io.

33. Angelo Corsaro. *The Data Distribution Service Tutorial*. 2014.

34. Hoang T. Dinh et al. "A survey of mobile cloud computing: architecture, applications, and approaches". *Wireless Communications and Mobile Computing* 13:18, 2013, pp. 1587–1611. DOI: https://doi.org/10.1002/wcm.1203. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcm.1203.

35. Aleksandar Dragojević et al. "FaRM: Fast Remote Memory". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 2014, pp. 401–414. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%5C'c.

36. Yucong Duan et al. "EVERYTHING AS A SERVICE (XAAS) ON THE CLOUD: ORIGINS, CURRENT AND FUTURE TRENDS". In: *Proceedings of the IEEE 8th International Conference on Cloud Computing*. 2015, pp. 621–628.

37. Eclipse Foundation. *Eclipse Cyclone DDS*. URL: https://projects.eclipse.org/projects/iot.cyclonedds.

38. eProsima. *FastDDS*. URL: https://www.eprosima.com/products-all/eprosima-fast-dds.

39. Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. 1st. Prentice Hall Press, USA, 2013. ISBN: 0133387526.

40. Ethernet Alliance. *Ethernet Roadmap*. 2022. URL: https://ethernetalliance.org/technology/ethernet-roadmap.
41. Erwin van Eyk et al. "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". *IEEE Internet Computing* 23:6, 2019, pp. 7–18. DOI: 10.1109/MIC.2019.2952061.
42. Janos Farkas, Lucia Lo Bello, and Craig Gunther. "Time-Sensitive Networking Standards". *IEEE Communications Standards Magazine* 2:2, 2018, pp. 20–21. DOI: 10.1109/MCOMSTD.2018.8412457.
43. Daniel Firestone. "VFP: A Virtual Switch Platform for Host Sdn in the Public Cloud". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. USENIX Association, Boston, MA, USA, 2017, pp. 315–328. ISBN: 9781931971379.
44. Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*. NSDI'18. USENIX Association, Renton, WA, USA, 2018, pp. 51–64.
45. Sadjad Fouladi et al. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. USENIX Association, Boston, MA, USA, 2017, pp. 363–376. ISBN: 9781931971379.
46. OPC Foundation. *Unified Architecture*. 2023. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/.
47. The Linux Foundation. *DPDK Ring Library*. 2023. URL: https://doc.dpdk.org/guides/prog_guide/ring_lib.html.
48. The Linux Foundation. *The Data Plane Development Kit*. URL: www.dpdk.org.
49. Andrea Garbugli et al. "A Framework for TSN-enabled Virtual Environments for Ultra-Low Latency 5G Scenarios". In: *ICC 2022 - IEEE International Conference on Communications*. 2022, pp. 5023–5028. DOI: 10.1109/ICC45855.2022.9839193.
50. Andrea Garbugli et al. "KubernetTSN: a Deterministic Overlay Network for Time-Sensitive Containerized Environments". In: *ICC 2023 - IEEE International Conference on Communications*. 2023, pp. 1494–1499. DOI: 10.1109/ICC45041.2023.10279214.
51. Nishant Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.
52. Google. *Google Cloud GPU*. URL: https://cloud.google.com/gpu.
53. Google. *Google Cloud High Performance Computing*. URL: https://cloud.google.com/solutions/hpc.
54. OpenFabrics Interfaces Working Group. *libfabric*. 2023. URL: https://github.com/ofiwg/libfabric.
55. Michael Gundall, Calvin Glas, and Hans D. Schotten. "Feasibility Study on Virtual Process Controllers as Basis for Future Industrial Automation Systems". In: *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*. Vol. 1. 2021, pp. 1080–1087. DOI: 10.1109/ICIT46573.2021.9453651.

56. Hermann Härtig et al. "The Performance of μ-Kernel-Based Systems". In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP '97. Association for Computing Machinery, Saint Malo, France, 1997, pp. 66–77. ISBN: 0897919165.

57. Zhiqiang He et al. "MasQ: RDMA for Virtual Private Cloud". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Association for Computing Machinery, Virtual Event, USA, 2020, pp. 1–14. DOI: 10.1145/3387514.3405849.

58. John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017.

59. Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. "GPU Virtualization and Scheduling Methods: A Comprehensive Survey". *ACM Comput. Surv.* 50:3, 2017. ISSN: 0360-0300. DOI: 10.1145/3068281. URL: https://doi.org/10.1145/3068281.

60. iMatix. *ZeroMQ*. 2023. URL: https://zeromq.org.

61. EunYoung Jeong et al. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 2014, pp. 489–502. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

62. Sagar Jha, Lorenzo Rosa, and Ken Birman. "Spindle: Techniques for Optimizing Atomic Multicast on RDMA". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 2022, pp. 1085–1097. DOI: 10.1109/ICDCS54860.2022.00108.

63. Sagar Jha et al. "Derecho: Fast State Machine Replication for Cloud Services". *ACM Trans. Comput. Syst.* 36:2, 2019. ISSN: 0734-2071. DOI: 10.1145/3302258. URL: https://doi.org/10.1145/3302258.

64. Kostis Kaffes et al. "Shinjuku: Preemptive Scheduling for µSecond-Scale Tail Latency". In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. NSDI'19. USENIX Association, Boston, MA, USA, 2019, pp. 345–359. ISBN: 9781931971492.

65. Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 2016, pp. 437–450. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

66. Magnus Karlsson and Björn Töpel. "The path to DPDK speeds for AF XDP". In: *Linux Plumbers Conference*. 2018.

67. Antoine Kaufmann et al. "TAS: TCP Acceleration as an OS Service". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Association for Computing Machinery, Dresden, Germany, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303985. URL: https://doi.org/10.1145/3302424.3303985.

68. Daehyeok Kim et al. "FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 2019, pp. 113–126.

69. Ayca Kirimtat et al. "Future Trends and Current State of Smart City Concepts: A Survey". *IEEE Access* 8, 2020, pp. 86448–86467. DOI: 10.1109/ACCESS.2020.2992441.

70. Dario Korolija, Timothy Roscoe, and Gustavo Alonso. "Do OS abstractions make sense on FPGAs?" In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 991–1010. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/roscoe.

71. Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. 2011. Athens, Greece. 2011, pp. 1–7.

72. Kubernetes Network Plumbing Working Group. *Multus CNI*. URL: https://github.com/k8snetworkplumbingwg/multus-cni.

73. Seung-Yong Lee and Minyoung Sung. "OPC-UA Agent for Legacy Programmable Logic Controllers". *Applied Sciences* 12:17, 2022, pp. 1–20. ISSN: 2076-3417. DOI: 10.3390/app12178859. URL: https://www.mdpi.com/2076-3417/12/17/8859.

74. Bojie Li et al. "Socksdirect: Datacenter Sockets Can Be Fast and Compatible". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. Association for Computing Machinery, Beijing, China, 2019, pp. 90–103.

75. Jianshen Liu et al. "Performance Characteristics of the BlueField-2 SmartNIC". *CoRR* abs/2105.06619, 2021. arXiv: 2105.06619. URL: https://arxiv.org/abs/2105.06619.

76. Yifang Ma et al. "Artificial intelligence applications in the development of autonomous vehicles: a survey". *IEEE/CAA Journal of Automatica Sinica* 7:2, 2020, pp. 315–329. DOI: 10.1109/JAS.2020.1003021.

77. Mallik Mahalingam et al. *Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks*. RFC 7348. 2014. DOI: 10.17487/RFC7348. URL: https://www.rfc-editor.org/info/rfc7348.

78. Michael Marty et al. "Snap: A Microkernel Approach to Host Networking". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Association for Computing Machinery, Huntsville, Ontario, Canada, 2019, pp. 399–413.

79. Laura Mazzuca et al. "Towards a Resource-Aware Middleware Support for Distributed Game Engine Design". In: *Proceedings of the 2022 ACM Conference on Information Technology for Social Good*. GoodIT '22. Association for Computing Machinery, Limassol, Cyprus, 2022, pp. 409–413. ISBN: 9781450392846. DOI: 10.1145/3524458.3547126. URL: https://doi.org/10.1145/3524458.3547126.

80. Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. en. 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145.

81. *Mellanox Messaging Accelerator*. URL: http://www.mellanox.com/page/software_vma.

82. Zeyu Mi et al. "SkyBridge: Fast and Secure Inter-Process Communication for Microkernels". In: EuroSys '19. Association for Computing Machinery, Dresden, Germany, 2019. ISBN: 9781450362818.

83. Ahmed Nasrallah et al. "Ultra-Low Latency (ULL) Networks: The IEEE TSN and IETF DetNet Standards and Related 5G ULL Research". *IEEE Communications Surveys & Tutorials* 21:1, 2019, pp. 88–145. DOI: 10.1109/COMST.2018.2869350.

84. NVIDIA. *ConnectX-6 Dx*. 2019. URL: https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx.

85. NVIDIA. *RDMA Aware Networks Programming User Manual*. URL: https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

86. NVIDIA. *Self-driving cars Technologies & Solutions*. 2023. URL: https://www.nvidia.com/en-us/self-driving-cars.

87. Object Management Group (OMG). *The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification*. 2018. URL: https://www.omg.org/spec/DDSI-RTPS/2.3/Beta1/PDF.

88. *OMG Data Distribution Standard*. URL: https://www.dds-foundation.org/omg-dds-standard.

89. *Open vswitch*. 2018. URL: http://openvswitch.org/.

90. Panos Patros et al. "Toward Sustainable Serverless Computing". *IEEE Internet Computing* 25:6, 2021, pp. 42–50. DOI: 10.1109/MIC.2021.3093105.

91. *PCI SIG. Single Root I/O Virtualization*. URL: https://pcisig.com/specifications/iov/single_root/.

92. Ricardo Silva Peres et al. "Industrial Artificial Intelligence in Industry 4.0 - Systematic Review, Challenges and Outlook". *IEEE Access* 8, 2020, pp. 220121–220139. DOI: 10.1109/ACCESS.2020.3042874.

93. Simon Peter et al. "Arrakis: The Operating System Is the Control Plane". *ACM Trans. Comput. Syst.* 33:4, 2015. ISSN: 0734-2071. DOI: 10.1145/2812806. URL: https://doi.org/10.1145/2812806.

94. Jonas Pfefferle et al. "A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces". *SIGPLAN Not.* 50:7, 2015, pp. 17–30.

95. Maksym Planeta et al. "MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 2021, pp. 47–63. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/planeta.

96. Michele Polese et al. "Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges". *IEEE Communications Surveys & Tutorials* 25:2, 2023, pp. 1376–1411. DOI: 10.1109/COMST.2023.3239220.

97. George Prekas, Marios Kogias, and Edouard Bugnion. "ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Association for Computing Machinery, Shanghai, China, 2017, pp. 325–341. ISBN: 9781450350853.

98. Allison Randal. "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers". *ACM Comput. Surv.* 53:1, 2020.

99. Thomas Rausch et al. "Towards a Serverless Platform for Edge AI". In: *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*. USENIX Association, Renton, WA, 2019. URL: https://www.usenix.org/conference/hotedge19/presentation/rausch.

100. GitHub Repository. *open62541*. 2023. URL: https://github.com/open62541/open62541.

101. Lorenzo Rosa and Andrea Garbugli. "Poster: INSANE – A Uniform Middleware API for Differentiated Quality using Heterogeneous Acceleration Techniques at the Network Edge". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 2022, pp. 1282–1283. DOI: 10.1109/ICDCS54860.2022.00134.

102. Lorenzo Rosa, Sagar Jha, and Ken Birman. "DerechoDDS: Efficiently leveraging RDMA for fast and consistent data distribution". In: *CARS 2021 6th International Workshop on Critical Automotive Applications: Robustness & Safety*. Münich, Germany, 2021. URL: https://hal.science/hal-03365892.

103. Lorenzo Rosa et al. "DerechoDDS: Strongly Consistent Data Distribution for Mission-Critical Applications". In: *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 2021, pp. 684–689. DOI: 10.1109/MILCOM52596.2021.9653032.

104. Lorenzo Rosa et al. "INSANE: A Unified Middleware for QoS-aware Network Acceleration in Edge Cloud Computing". In: *Proceedings of the 24th ACM/IFIP International Middleware Conference*. Middleware '23. Association for Computing Machinery, Bologna, Italy, 2023, p. 14. DOI: 10.1145/3590140.3629105. URL: https://doi.org/10.1145/3590140.3629105.

105. Lorenzo Rosa et al. "Supporting VPLC Networking over TSN with Kubernetes in Industry 4.0". In: *Proceedings of the 1st Workshop on Enhanced Network Techniques and Technologies for the Industrial IoT to Cloud Continuum*. IIoT-NETs '23. Association for Computing Machinery, New York, NY, USA, 2023, pp. 15–21. ISBN: 9798400703027. DOI: 10.1145/3609389.3610566. URL: https://doi.org/10.1145/3609389.3610566.

106. Benjamin Rothenberger et al. "ReDMArk: Bypassing RDMA Security Mechanisms". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 4277–4292. ISBN: 978-1-939133-24-3.

107. *rsocket(7)*. 2019. URL: https://linux.die.net/man/7/rsocket.

108. RTI. *RTI Connext DDS. Core Libraries User's manual*. URL: https://community.rti.com/documentation.

109. Rusty Russell. "virtio: Towards a de-Facto Standard for Virtual I/O Devices". *SIGOPS Oper. Syst. Rev.* 42:5, 2008, pp. 95–103.

110. Andrea Sabbioni et al. "A Shared Memory Approach for Function Chaining in Serverless Platforms". In: *2021 IEEE Symposium on Computers and Communications (ISCC)*. 2021, pp. 1–6. DOI: 10.1109/ISCC53001.2021.9631385.

111. Andrea Sabbioni et al. "DIFFUSE: A DIstributed and decentralized platForm enabling Function composition in Serverless Environments". *Computer Networks* 210, 2022, p. 108993. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2022.108993. URL: https://www.sciencedirect.com/science/article/pii/S138912862200161X.

112. Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. "An Overview of Service Placement Problem in Fog and Edge Computing". *ACM Comput. Surv.* 53:3, 2020. ISSN: 0360-0300. DOI: 10.1145/3391196. URL: https://doi.org/10.1145/3391196.

113. José Santos et al. "Towards Low-Latency Service Delivery in a Continuum of Virtual Resources: State-of-the-Art and Research Directions". *IEEE Communications Surveys Tutorials* 23:4, 2021, pp. 2557–2589.

114. Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". *ACM Comput. Surv.* 22:4, 1990, pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167.

115. Weisong Shi et al. "Edge Computing: Vision and Challenges". *IEEE Internet of Things Journal* 3:5, 2016, pp. 637–646. DOI: 10.1109/JIOT.2016.2579198. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84987842183&doi=10.1109%2fJIOT.2016.2579198&partnerID=40&md5=a975bcadfcbc0402da6444a53205ecde.

116. Simon Shillaker and Peter Pietzuch. "FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing". In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USENIX Association, USA, 2020. ISBN: 978-1-939133-14-4.

117. Swapnil Sadashiv Shinde, Dania Marabissi, and Daniele Tarchi. "A Network Operator-biased approach for Multi-service Network Function Placement in a 5G Network Slicing Architecture". *Computer Networks* 201, 2021, p. 108598. ISSN: 1389-1286.

118. Anna Kornfeld Simpson et al. "Securing RDMA for High-Performance Datacenter Storage Systems". In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

119. Weijia Song et al. *Cascade: A Platform for Delay-Sensitive Edge Intelligence*. 2023. arXiv: 2311.17329 [cs.OS].

120. Chandramohan A. Thekkath et al. "Implementing Network Protocols at User Level". *SIGCOMM Comput. Commun. Rev.* 23:4, 1993, pp. 64–73. ISSN: 0146-4833.

121. Neil C. Thompson and Svenja Spanuth. "The Decline of Computers as a General Purpose Technology". *Commun. ACM* 64:3, 2021, pp. 64–72. ISSN: 0001-0782. DOI: 10.1145/3430936. URL: https://doi.org/10.1145/3430936.

122. Marcos A. M. Vieira et al. "Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications". *ACM Comput. Surv.* 53:1, 2020. ISSN: 0360-0300.

123. Cheng-Xiang Wang et al. "Artificial Intelligence Enabled Wireless Networking for 5G and Beyond: Recent Advances and Future Challenges". *IEEE Wireless Communications* 27:1, 2020, pp. 16–23. DOI: 10.1109/MWC.001.1900292.

124. Dongyang Wang et al. "VSocket: Virtual Socket Interface for RDMA in Public Clouds". In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2019. Association for Computing Machinery, Providence, RI, USA, 2019, pp. 179–192.

125. Jiawei Wang et al. "BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 2022, pp. 249–262.

126. Liang Wang et al. "Peeking Behind the Curtains of Serverless Platforms". In: *Proc. of USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, Boston, MA, 2018, pp. 133–146. ISBN: ISBN 978-1-939133-01-4.

127. Dapeng Wu et al. "Streaming video over the Internet: approaches and directions". *IEEE Transactions on Circuits and Systems for Video Technology* 11:3, 2001, pp. 282–300.

128. Yiwen Wu, Ke Zhang, and Yan Zhang. "Digital Twin Networks: A Survey". *IEEE Internet of Things Journal* 8:18, 2021, pp. 13789–13804. DOI: 10.1109/JIOT.2021.3079510.

129. XDP-project. *libxdp*. [Online]. URL: https://github.com/xdp-project/xdp-tools.

130. Z. Xiang, et al. "Reducing Latency in Virtual Machines: Enabling Tactile Internet for Human-Machine Co-Working". *IEEE Journal on Selected Areas in Communications* 37:5, 2019, pp. 1098–1116.

131. Irene Zhang et al. "The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Association for Computing Machinery, Virtual Event, Germany, 2021, pp. 195–211. ISBN: 9781450387095.