INSANE: A Unified Middleware for QoS-aware Network Acceleration in Edge Cloud Computing

Lorenzo Rosa
lorenzo.rosa@unibo.it
University of Bologna
Bologna, Italy

Andrea Garbugli
andrea.garbugli@unibo.it
University of Bologna
Bologna, Italy

Antonio Corradi
antonio.corradi@unibo.it
University of Bologna
Bologna, Italy

Paolo Bellavista
paolo.bellavista@unibo.it
University of Bologna
Bologna, Italy

ABSTRACT

Edge cloud computing is a promising programming and deployment paradigm to empower delay-sensitive applications. By executing close to the network edge, distributed applications can have quicker reactions to event occurrence and consequently prompter dynamic adaptations. In addition, recent improvements in connectivity support allow developers to benefit from heterogeneous and alternative communication technologies (e.g., RDMA, DPDK, XDP, etc.) to meet the requirements of network-intensive edge applications. However, exploiting these technologies makes applications statically tailored to a specific network interface; this significantly limits the potential of edge cloud computing, where application components should be able to migrate seamlessly at runtime. INSANE aims at solving that issue by exposing a technology-agnostic middleware API that lets developers simply specify their QoS communication requirements; the dynamic selection of the most appropriate technology on the currently hosting edge node is delegated to INSANE. The paper also presents how it is possible to develop two different INSANE-based applications (a decentralized messaging system and an image streaming framework) with a few lines of code. Finally, an extensive performance evaluation shows that our middleware adds very limited ns-scale overhead to the raw acceleration technologies.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; **Network performance evaluation**.

KEYWORDS

Edge Cloud, Network Acceleration, QoS

ACM Reference Format:
Lorenzo Rosa, Andrea Garbugli, Antonio Corradi, and Paolo Bellavista. 2023. INSANE: A Unified Middleware for QoS-aware Network Acceleration in Edge Cloud Computing. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3590140.3629105

Middleware '23, December 11–15, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0177-1/23/12.
https://doi.org/10.1145/3590140.3629105

1 INTRODUCTION

In the last few years, edge cloud computing emerged as an extension of the cloud computing paradigm outside datacenters [6, 48]. This paradigm envisions a network of small-scale cloud environments close to data sources and promises to enable a new generation of intelligent, data-driven applications even in latency-sensitive domains, such as industrial automation, autonomous transportation, and next-generation telco services. The availability of relatively powerful local resources, combined with new cost reduction strategies for Machine Learning (ML) algorithms such as the popular Large Language Models (LLMs) [10, 12, 46], will eventually enable applications to locally provide intelligent responses to external events with µs-scale latency [5, 18, 51].

Since most of these applications are network-intensive, edge developers would greatly benefit from the adoption of state-of-the-art *network acceleration* techniques to meet their stringent performance requirements. Compared to standard networking solutions, recent technologies like the Linux eXpress Data Path (XDP), the Data Plane Development Kit (DPDK), and Remote Direct Memory Access (RDMA) [30, 39, 53] can achieve very interesting performance by minimizing overhead and by reducing data copies and context switches on critical paths [9]. More and more often, the price of these options in terms of higher resource usage, or even dedicated hardware, is affordable for edge cloud platforms.

However, the adoption of network acceleration technologies in edge cloud platforms comes nowadays with a practical yet fundamental problem of *code portability*. In fact, these technologies are relatively hard to use for non-experienced system programmers, as they expose different and very low-level programming abstractions and interfaces [58]. Hence, developers tend to tailor their code to the specific network acceleration technology used, which could be not available at any deployment node, in particular when dealing with edge cloud hosts that may be significantly heterogeneous. Even worse, the hardware and software components supporting these network acceleration techniques are in rapid evolution, thus forcing the continuous update of the application code and endangering cost-effective maintainability. Overall, at the current state-of-the-art, despite their potential performance advantages, network acceleration technologies are currently hard to combine with the intrinsic dynamicity of edge cloud computing.

To address these technical challenges, this paper presents INSANE, a middleware designed to make latency-critical edge cloud

applications portable across heterogeneous edge nodes equipped with differentiated network acceleration technologies. The key principle of our proposed approach is to move the choice of a given network acceleration technology from the developer to the middleware: with INSANE, developers do not have direct access to the native APIs of the actual technology; on the opposite, they use a general-purpose and technology-agnostic middleware API, offering easily customizable and higher-level abstractions. In fact, in INSANE developers express their communication requirements through a set of Quality of Service (QoS) parameters that control performance, resource usage, and time sensitiveness. INSANE dynamically and automatically maps these parameters into the most appropriate network technology available at runtime on the employed edge nodes, thus enabling the dynamic deployment and migration of application components across possibly heterogeneous edge locations.

By looking at its architecture, INSANE follows a *micro-kernel-inspired* approach [28, 33] and consists of two primary components: a client library that exposes the technology-agnostic API to developers, and a userspace module (*runtime*) that centralizes the host networking functionality and exposes it as a service to the local applications. The INSANE runtime abstracts the common mechanisms of network acceleration techniques, such as *zero-copy transfers*, into a novel and technology-agnostic framework for high-performance host networking. Moreover, it enables specializations via plugins (*datapaths*), one per specific technology. This design is particularly suitable for edge cloud environments because it enables developers to isolate their applications (e.g., in containers) and to transparently attach them to different network options depending on runtime conditions and situations. About performance, the paper reports an extensive quantitative evaluation of the INSANE runtime, showing how its performance-oriented design and implementation only introduce ns-scale overhead on network operations.

The remainder of the paper presents the technical characteristics and in-depth technical insights about INSANE. Section 2 introduces our definition of *edge cloud computing*. Sections 3 and 4 provide an overview of the network acceleration technologies considered in INSANE and of the related literature. Section 5 details the INSANE API and runtime. Section 6 reports our experience (and the associated quantitative performance results) with the deployment of INSANE-based application components over two different edge testbeds. Our extensive performance benchmarking has demonstrated that INSANE-based applications can achieve an average round-trip as low as 4.9 µs, and an average bandwidth utilization as high as 86.9 Gbps, when using DPDK. Section 7 proves the ease of use of our middleware API, by describing how it is possible to develop two edge cloud applications, i.e., a decentralized messaging application and a streaming one, with a few lines of INSANE code, with extremely good performance results.

2 EDGE CLOUD COMPUTING

The success of the concept of Internet of Things (IoT) and the consequent digital transformation of virtually any application domain are pushing toward an evolution of the cloud computing paradigm. The pervasive availability of connected devices increasingly requires applications to consume, analyze and generate all kinds of data from a variety of sources with heterogeneous performance constraints, which can be only partially fulfilled by the traditionally centralized cloud infrastructures, originally designed mostly to support *offline* computations on large data batches.

To support *latency-critical* applications that need to provide *online* timely answers to external events, a recent trend is to extend cloud infrastructures beyond their traditional boundaries, by including a hierarchy of virtualized computing resources physically located between traditional cloud datacenters and data sources. The resulting computing model is a *continuum* of virtualized resources, spanning from traditional centralized datacenters to the network edge [6, 48]. Across the continuum, providers can 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. In this context, companies are increasingly moving performance-critical application components physically close to data sources, thus significantly improving response times and service interactivity.

This paper will mainly target two core components of the continuum, the traditional *core cloud* datacenters and the *edge cloud* datacenters. 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 paper, 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. Edge clouds are increasingly common in many domains, such as telco (Multi-access Edge Computing, MEC [1]), industrial automation (e.g., factory-local server racks [16]), autonomous transportation, etc. 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.

Motivated by this trend, our research work aims at simplifying and supporting the development of high-performance edge services, by considering that edge resources may be far more heterogeneous than in large-scale traditional cloud datacenters and thus the *portability* of edge cloud services is still an open research challenge.

3 NETWORK ACCELERATION TECHNOLOGIES

Communication links are rapidly evolving to support higher bandwidth and lower latency, largely outpacing the evolution rate of other host resources (e.g., core speeds, cache sizes, etc.). The main consequence is that the operating system kernel-level networking stack, designed under the assumption of slower I/O operations, is no longer able to keep up with the available access link bandwidths and latencies [9]. Although this trend started in datacenter environments, the problem is especially relevant for edge computing scenarios, as datacenter-like resources are available at the network edge and latency requirements become extremely demanding [48, 51]. Major sources of network overhead in the OS kernel are data copies, inefficient cache usage, protocol processing delays, and context switches [9, 19, 42].

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

Table 1: A comparison between the main options for end-host networking in the edge cloud.

To fully exploit the communication capabilities of modern hardware, new forms of highly efficient end-host networking have emerged. Three of them, in particular, are increasingly popular: the Linux eXpress Data Path (XDP), which provides fast in-kernel packet processing [53]; the Data Plane Development Kit (DPDK) and Remote Direct Memory Access (RDMA), which *bypass* the kernel and allow the direct interaction between userspace and Network Interface Cards (NICs) [30, 39].

All these technologies follow a similar approach to improve the network performance: for example, they remove data copies by letting the hardware NIC directly access the memory of user applications (*zero-copy* transfers). However, the mechanisms to provide these advanced features substantially differ across technologies, in terms of API, resource usage, hardware requirements, and performance. Such diversity reflects the original specific purposes they were built for: for example, XDP and DPDK for fast packet processing, RDMA as a networking technique for HPC. Table 1 reports the main differences among these technologies; in the following, we discuss them with specific focus on *zero-copy* transfer capability.

The Linux kernel introduced XDP as the lowest layer of its network stack, located within the driver of network devices. At this stage, XDP is able to execute user-provided code (*eBPF programs*) for each packet, including forwarding the packet itself to and from a userspace socket. In 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* [56]). 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 userspace network and transport protocols (e.g., mTCP [22]).

DPDK and RDMA, instead, take a step further and completely bypass the OS kernel. This approach results in a reduced scheduling overhead, because there is no context change between userspace and kernel processes on the critical datapath. DPDK, in particular, consists of a set of C libraries that let users directly interact with a userspace version of the network device drivers (*Poll Mode Drivers*, PMD). Hence, also in this case the user has to provide its own network protocol stack. The user and the userspace driver exchange packet data on a shared memory area called *mempool*. To send a packet, the user will provide to the driver a pointer to the appropriate memory area. On the receive side, to minimize the communication overhead, DPDK dedicates one or more threads (*lcores*), each pinned to a separate core, to *busy poll* for new messages. Detected packets are placed by the driver into the shared memory, and the corresponding pointers are returned to the user. Although this high resource consumption makes DPDK extremely fast, it might not be suitable for all the network edge environments.

To this end, RDMA provides a more resource-efficient approach. RDMA is a network model that allows a process on one machine to directly access the memory of another process on a remote machine. Unlike XDP and DPDK, this model avoids the need for the user to provide userspace network and transport protocols. To achieve exceptional performance, including high throughput (~200 Gbps) and low latency (<1 µs), RDMA offloads the network operations directly to the network card (NIC). Thus, a compatible network card is required. After registering a memory area with the network card (*memory region*), users establish a remote connection by opening a Queue Pair (QP), which comprises a couple of *work queues* for send and receive operations. Indeed, 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 protocols such as RDMA over Converged Ethernet (RoCEv2) [3]. 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 in the network operation, thus making the latter kind of operations generally faster.

As these technologies become increasingly common to accelerate the end-host networking of general-purpose systems, INSANE abstracts their common design principles and designs a technology-agnostic userspace network stack that offers typical system features, such as memory management for zero-copy transfers, efficient packet processing, and different packet scheduling strategies. A plugin-based architecture allows the specialization of such features for each supported network acceleration technology, thus offering developers access *as a service* to the most appropriate technology for the dynamically selected context, without the hassle of dealing with heterogeneous and low-level interfaces. However, INSANE only provides access to the minimum set of common functions

among the supported technologies: for example, INSANE plans to support RDMA only through the use of `two-sided` operations. INSANE is not designed to support applications with more advanced needs, such as the use of `one-sided` RDMA or higher degree of control on the system resources. Lower-level interfaces are more suitable for those applications (§ 4).

4 RELATED WORK

INSANE enables edge cloud applications to dynamically bind to the high-performance networking capabilities available on possibly heterogeneous hosts. We achieve this goal through an innovative approach that combines a technology-agnostic API and a general-purpose runtime framework, specifically designed for the edge cloud environment. While building upon previous research, our solution stands out from prior works in this area, which consider these two aspects separately and primarily within the traditional datacenter setting.

In the space of agnostic network API, both libfabric [41] and Demikernel [58] provide a uniform interface on top of heterogeneous network acceleration technologies, although at different abstraction layers. The libfabric library enables RDMA applications to run independently of the presence of the necessary supporting hardware. Developers code against a transparent set of communication primitives, which the library translates either to native RDMA operations, if the suitable support is locally available (e.g., RDMA NIC), or to kernel-based TCP/IP, although non-RDMA transports are intended mostly for debug purposes. The libfabric interface is very low-level and is generally adopted by 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.).

On the opposite, Demikernel targets standard cloud users as it exposes a higher-level interface, specifically an extension of the standard POSIX primitives, that lets applications submit asynchronous I/O operations. Demikernel implements these primitives through a set of userspace libraries, each specialized for a different I/O technology (DPDK, RDMA, and kernel networking are supported). These libraries offer typical OS services (memory management, I/O scheduling, network stack) to users when the OS kernel is bypassed, according to the *library OS* approach also explored by some previous literature [4, 24, 43, 45].

Sharing the same motivation, the INSANE client library exposes a high-level uniform interface, but it introduces two relevantly novel aspects compared to the above works. First, it offers a higher-level interface that simplifies the development of typical edge applications (§ 5.1), thus aligning more effectively with our goals of ease of development and portability compared to reworking interfaces of commodity OSs. Second, whereas libfabric and Demikernel require the users to choose which I/O technologies to bind to the interface in a static way (at compile time), our QoS-based solution delegates this choice to the middleware, which dynamically selects the most appropriate binding among those available at deployment site.

From an architectural perspective, INSANE does not follow the *library OS* approach. On the contrary, our work is influenced by the *microkernel-inspired* model introduced by TAS [28] and Snap [33]. Although these works do not target heterogeneous network acceleration technologies, they create a userspace module that centralizes standard host networking and offers it *as a service*: applications post I/O operations through shared-memory channels, and the module executes the necessary network processing. TAS adopts this model to provide a *fast path* for the TCP protocol in the context of RPC workload; Snap is more general and allows the definition of custom packet processing modules (called *engines*). This approach retains the advantages of a centralized network stack even in presence of kernel-bypassing technologies: an efficient management of all the OS resources for all the local applications, including memory allocation, cache-efficient thread scheduling, and the support to transparent software upgrades. In the edge cloud, this model promotes reduced resource usage and higher flexibility: applications can dynamically attach to the network service on the local host, without the need to instantiate additional resources. Whereas the use of uncoordinated *OS libraries* would instantiate a technology-specific datapath *per application*, requiring dedicated resources (e.g., at least one CPU core), our centralized design instantiates each datapath at most once per physical host, within the INSANE runtime. Therefore, INSANE offers a novel solution to easily and transparently access heterogeneous networking technologies, but it is also designed to efficiently answer to the resource consumption requirements of edge cloud environments.

Some acceleration technologies require a userspace network stack (§ 3); a populated line of previous work addressed this need [8, 22, 25]. Although these solutions might be integrated into our middleware, they are usually tailored for a specific network technology, and would require profound adjustments to fit our internal design. When needed, INSANE defines a custom and minimal network stack that can introduce only ns-scale overhead on packet processing.

5 INSANE: A UNIFORM MIDDLEWARE API

In this paper, we propose INSANE, a novel middleware designed and optimized for the emerging class of edge cloud applications that combine intelligent logic, stringent performance requirements, and heterogeneous deployment scenarios. INSANE lets developers declare their communication requirements through high-level QoS policies and uses them as hints to dynamically bind each communication flow to the most appropriate network acceleration technology available locally. Thus, INSANE effectively decouples application code from the specific technology dynamically found at the participating nodes; a very relevant capability in cloud continuum scenarios; in addition, it maintains high network efficiency, while also easing code development and portability. INSANE consists of two main components: a runtime, which represents the middleware core and must be in execution on each participating host, and a client library that exposes the API to the applications, allowing them to interact with the runtime.

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 (§ 5.1 and § 5.2). Then, we provide an overview of the runtime architecture to understand how the INSANE primitives are mapped to heterogeneous network technologies (§ 5.3).

5.1 INSANE API

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 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 1 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 2 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 Section 5.2 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 [58], 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.

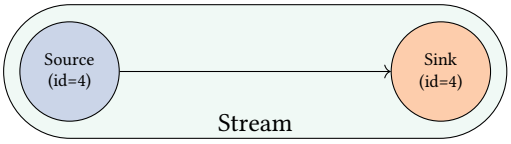


Figure 1: An INSANE *channel* is created between sources and sinks with the same *channel id* within the same *stream*.

```
/* Open and close a session */
int init_session();
int close_session();

/* Stream */
stream_t create_stream(options_t opts);
void close_stream(stream_t stream);

/* Source APIs */
source_t create_source(stream_t stream, int channel);
void close_source(source_t source);
buffer_t get_buffer(source_t src, size_t size, int flags);
int emit_data(source_t src, buffer_t buffer);
int check_emit_outcome(source_t source, int id);

/* Sink APIs */
sink_t create_sink(stream_t stream, int channel, data_cb cb);
void close_sink(sink_t sink);
int data_available(sink_t sink, int flags);
buffer_t consume_data(sink_t sink, int flags);
void release_buffer(sink_t sink, buffer_t buffer);
```

Figure 2: The INSANE API.

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 7 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.

5.2 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 acceleration 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 [14] to provide a deterministic network behavior (§ 5.3).

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 [27]. 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 [17], that already assume a *best-effort* network and provide their own solutions to build additional guarantees [40].

5.3 INSANE runtime

This section discusses the architecture of the INSANE runtime. In particular, we focus on the novel abstractions that we designed to uniform the network operations of heterogeneous technologies, which we use as a support for the primitives discussed in the previous sections.

According to the *microkernel-inspired* design (§ 4), 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 thread. However, not only the associated overhead is small in our case of zero-copy networking [33], but many factors contribute to minimize it while also retaining the advantages of this model: in particular, state-of-the-art lock-free queues [31, 54], combined with modern multi-core processors and IPC optimization techniques [20, 35, 52].

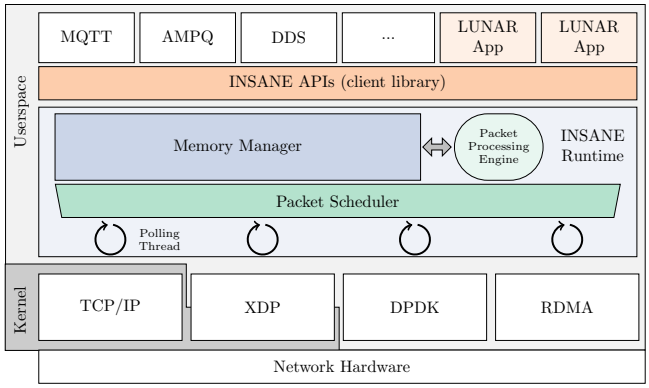


Figure 3: The INSANE Architecture.

The INSANE runtime has four main components, represented in Figure 3: a *memory manager*, a *packet scheduler*, a pool of *polling threads*, and a set of *datapath plugins*. The memory manager is the most important element, because it effectively implements the abstraction that decouples the homogeneous interface offered to the applications from the highly heterogeneous details of each transport technology. As noted in Section 3, all the considered technologies adopt a similar approach to achieve the goal of zero-copy data transfers: they place data to send or receive in a shared memory area registered with the NIC for Direct Memory Access (DMA). Starting from this insight, we designed a technology-agnostic mechanism for zero-copy communication based on shared memory. Then, we implement this abstraction differently for different transport options. 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 relevant parts of that area.

Importantly, our design based on shared memory enables applications in cloud platforms to efficiently leverage network acceleration technologies, which currently are difficult to integrate within containers and virtual machines without harming either their dynamicity (e.g., live migration) or performance [21, 29, 44]. By enabling applications to dynamically (re)attach to a local instance of the runtime, INSANE offers the innovative option of Network Acceleration *as a Service*, while also fulfilling the edge cloud requirements of application portability and seamless migration in the resource continuum (§ 2). We will discuss more about this topic in § 8.

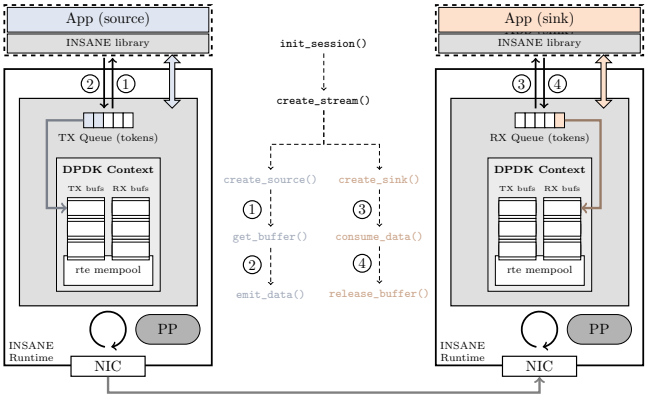


Figure 4: The INSANE communication flow.

Figure 4 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. For time sensitive data, the scheduler supports the Time-Sensitive Networking (TSN) standard, implementing the IEEE 802.1Qbv time-aware scheduler [36], specifically designed for edge soft real-time applications. 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 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.

6 INSANE EVALUATION

Our evaluation of INSANE focuses on two aspects. On the one hand, we show that our abstraction layer introduces minimal overhead compared to each native communication technology. We compare INSANE to Demikernel [58], 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. On the other hand, we prove that the ease of use and flexibility of our API significantly simplifies the design of very different edge-oriented applications. In particular, we build a decentralized Message-oriented Middleware (MoM) and a streaming application, and we compare them to similar edge applications in terms of both performance and development complexity.

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.

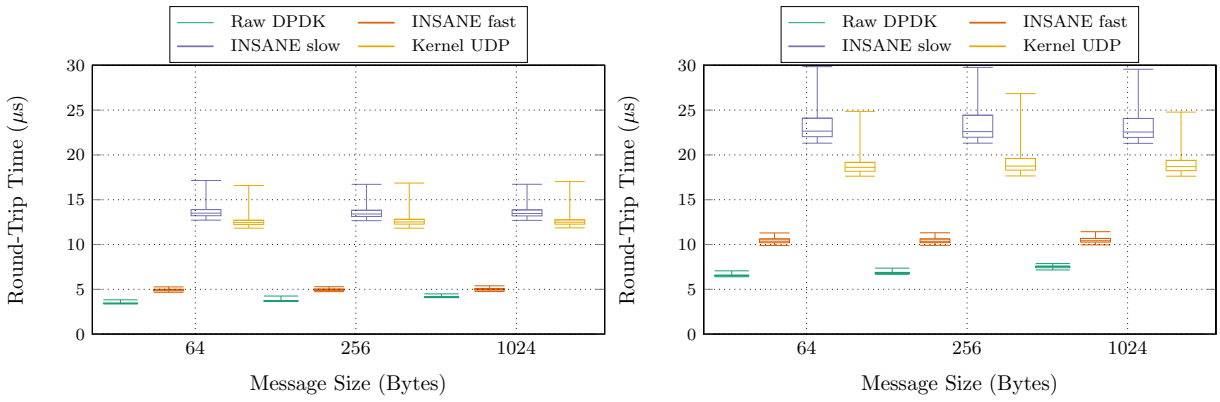
Our implementation of INSANE and INSANE-based applications is publicly available at https://github.com/MMw-Unibo/INSANE.

6.1 Experimental setup

We evaluate INSANE in two different testbeds. The first is a local setup that matches a typical edge cloud environment. In this setting, two nodes are directly interconnected in order to minimize the overhead of network operations and magnify the impact of INSANE on the measured metrics. The other is a public cloud infrastructure

Testbed	OS	CPU	RAM	NIC	Switch
Local	Ubuntu 22.04	18-core Intel i9-10980XE @ 3.00GHz	64GB	Mellanox DX-6 100Gbps	—
Public cloud	Ubuntu 22.04	32-core AMD 7452 @ 2.35GHz	128GB	Mellanox DX-5 100Gbps	Dell Z9264F-ON

Table 2: Setup of the local and public testbed for INSANE evaluation.



(a) Local testbed. **(b) Public cloud (Cloudlab)**

Figure 5: Round-Trip Time (RTT) for increasing payload sizes.

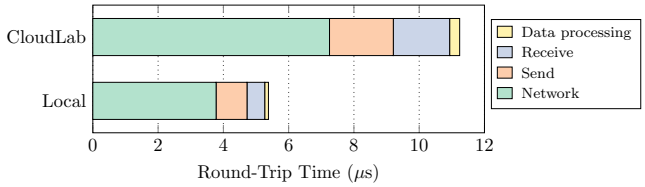


Figure 6: *INSANE fast* latency breakdown (64B)

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

Table 3: LoC to implement the benchmarking application.

(CloudLab [13]), where we reserved two nodes interconnected by a switch. The hardware and OS specifications of the nodes in both the testbeds are reported in Table 2. For DPDK, we used v22.11.

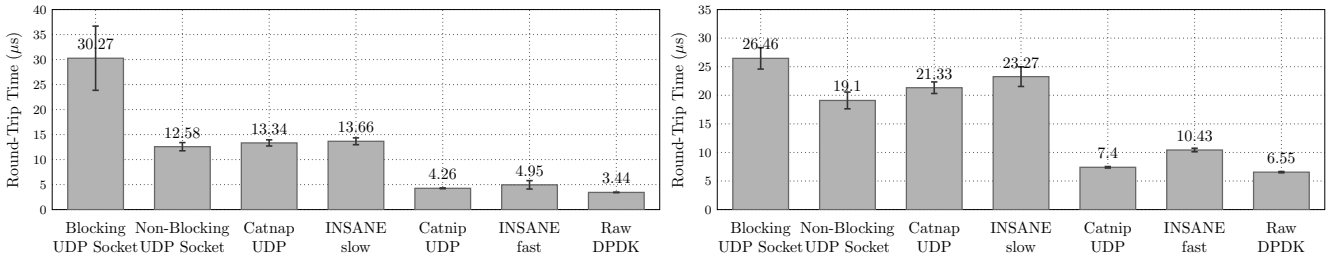
To maximize performance, we increase the Linux socket buffers to allow receivers to keep up with the highest possible send rate. To reduce OS-induced latency and scheduling variability, we pin application processes to cores, and map each datapath plugin to exactly one polling thread (§ 5.3).

6.2 Latency and throughput benchmarks

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 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 3 summarizes, without requiring developers to understand the details of each technology.

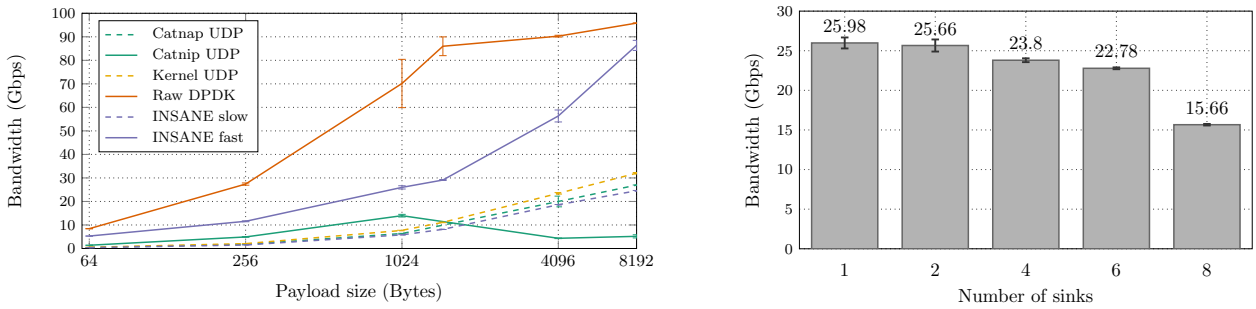
Figure 5a and Figure 5b report the latency of INSANE for increasing payload sizes when using two different datapath acceleration QoS: *slow*, which maps network operations to UDP sockets, and *fast*, which maps to DPDK. Overall, we note that there is no significant difference among different payload sizes. In the local 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 public 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 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 local testbed[1]. Although INSANE tries to minimize the processor intervention on 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

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



(a) Local testbed. **(b) Public cloud (Cloudlab)**

Figure 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)**

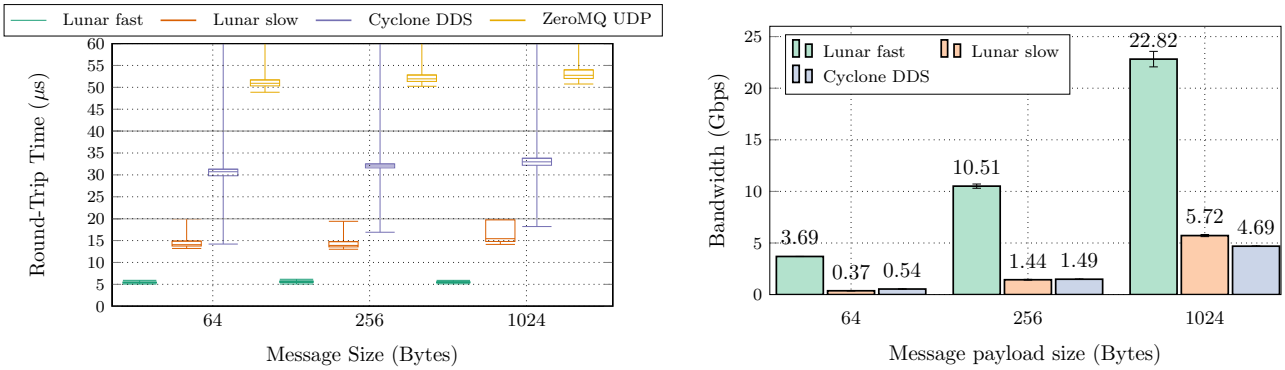
Figure 8: Throughput benchmark for INSANE and the other reference systems.

multiple polling threads in order to better leverage the multi-core capabilities of modern processors. In Section 8 we further elaborate on this point.

To put our INSANE performance in perspective, in Figure 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 [58], 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 our local setup, and 1.9 µs slower on average in the cloud setting. If we consider DPDK, we observe the same trend discussed in the previous paragraph. On the local 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.

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 local setup. Figure 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 [58] and sends one packet per time on the network. Conversely, INSANE adopts a form of *opportunistic batching* [23, 26] at sender side: 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 is that it can support multiple applications on the same host at the same time. In



(a) Latency of MoMs for increasing payload sizes.

(b) Throughput of MoMs for increasing payload sizes

Figure 9: Performance benchmark for Lunar MoM and other reference systems.

Figure 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 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.

7 EVALUATION OF INSANE-BASED APPLICATIONS

A key design goal for INSANE is to ease the development of a broad set of applications with heterogeneous requirements in edge cloud nodes (§ 5). To demonstrate that our interface effectively answers to 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 complete portability of these applications across various network technologies while delivering better performance than widely adopted similar systems.

7.1 LUNAR MoM

Heterogeneous systems at the network edge usually rely on message queuing systems or Message-Oriented Middleware (MoM) systems for asynchronous, low-overhead communication, ease of implementation, and scalability. Depending on the deployment scenario and the application needs, MoMs may require a centralized broker to disseminate messages, or they can be entirely decentralized for increased scalability. In both cases, MoMs implement a publish-subscribe communication pattern. The main concepts in this model are *topics*, which represent abstract named queues, and *publisher* and *subscribers* as producers and consumers of those queues.

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.

Our demonstration of LunarMoM, a decentralized messaging system built using the INSANE API, shows that it offers an efficient option for the edge cloud. To evaluate its performance, we compared LunarMoM against two widely used decentralized messaging systems in that environment, OMG DDS and ZeroMQ. 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 local testbed described in § 6.1.

The results, as shown in Figure 9a, indicate that LunarMoM has the lowest latency in both fast (using DPDK) and slow (using UDP) modes. Compared to the raw INSANE performance (Figure 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 considerations apply to the throughput evaluation (Figure 9b), 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.

7.2 LUNAR Streaming framework

In edge cloud scenarios, we 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 [55]. To support 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.

To test Lunar Streaming we implemented a simple application that streams raw images, i.e., for each image frame we send RGB

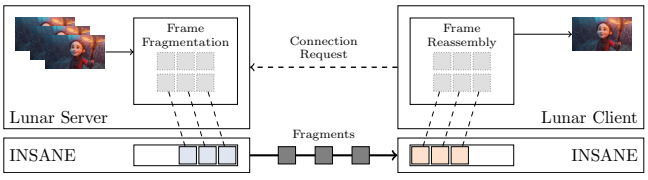
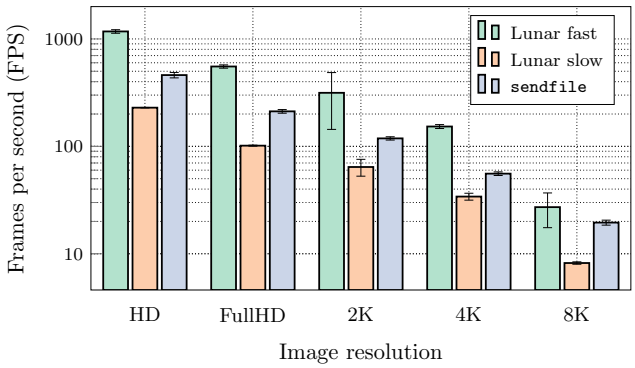


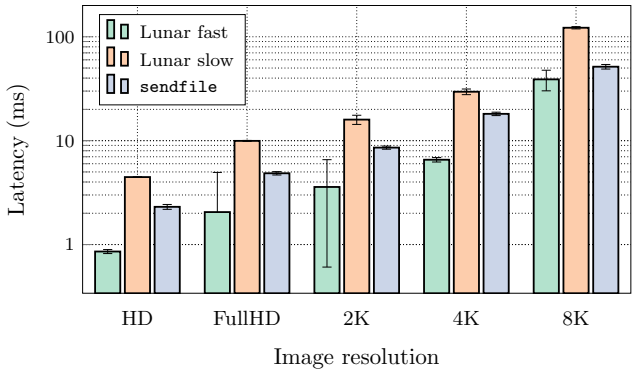
Figure 10: Lunar Streaming framework application.

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

Table 4: Size of the images sent in the streaming benchmark.



(a) FPS for increasing image resolution.



(b) Latency per frame for increasing image resolution

Figure 11: Benchmark for Lunar Stream and `sendfile`.

values for every single pixel (Figure 10). We use sample images of different common sizes (Table 4) 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 11a), and (ii) the average end-to-end latency for frame transmission (Figure 11b), 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. 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 [49] or real-time simulations (e.g., cloud gaming [34]).

Finally, as briefly anticipated, streaming applications usually send various media (i.e., video and images) in a compressed format. For example, HVEC, VP9, or the newer VVC are often used as video CODECs [32], which are transmitted using protocols such as RTP or WebRTC [7]. Implementing a full stack of streaming protocols is beyond the scope of this work, but even just by sending raw images, we obtained excellent results. Hence, we emphasize that INSANE can be effective in accelerating existing streaming frameworks [2].

8 FUTURE DIRECTIONS

The design and deployment of INSANE 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 networking technologies promise to enable data-driven intelligent applications even far from centralized datacenters, but they also bring additional heterogeneity in an ecosystem that already struggles to define standard system practices. We believe that INSANE is a first step to answer those questions, but many problems remain unsolved. In the following, we summarize the most important open challenges.

Cloud integration. Network acceleration technologies are increasingly available in both *core* and *edge* cloud infrastructures, despite scalability and security concerns from major providers [21, 29, 47]. In this context, the design of INSANE decouples the application code from the specific network acceleration technologies and, as a consequence, may already enable forms of *Network Acceleration as a Service*: by deploying the INSANE runtime in a co-located container, cloud applications can already attach to it via shared memory and obtain transparent access to the network acceleration options available at the specific deployment site. We plan to extend INSANE toward a complete integration with cloud platforms.

Thread scheduling strategies. Our evaluation of INSANE maps each *datapath plugin* to a dedicated *polling thread*. Although this choice limits the resource usage of the system, it also puts pressure on the receive pipeline, which must (i) process incoming packets through the network stack, and (ii) insert a token into the right application queue. In our evaluation, we found that a single sender easily overflows a single-core sink. Indeed, receive operations are CPU-bounded, not ideal for high-performance networking. One promising solution is to map the datapath plugins to multiple polling threads, as INSANE allows to do (§ 5.3). In this paper, space limitations prevented us to deeper investigate the performance impact of different threading strategies, but we plan to include that evaluation in future and more extended versions of our work. We believe that the detailed study in [33] on a similar system would provide a useful reference to guide our activities. An alternative approach that we plan to explore is the possibility to offload all or part of time-consuming receive-side operations to hardware devices (Smart NICs [15, 37], Data Processing Units - DPUs [38]) which will soon become commodity hardware even for edge cloud nodes.

End-to-end zero copy transfers. 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 receiver must copy the payloads of the incoming fragments to the final memory destination. For that reason, to preserve a true zero-copy semantic, the INSANE prototype currently does not support UDP/IP packet fragmentation, and we resorted to jumbo frames for tests with the biggest payload sizes, following the same approach of Demikernel [58], or to application-level fragmentation. Had we decided to support fragmentation within the network stack, we would have choked the receive pipeline with multiple data copies for reconstruction. The definition of a technique for zero-copy data reconstruction remains an open research challenge.

Packet scheduling. A careful scheduling of network operations is crucial for high-performance systems like INSANE [11, 58]. The INSANE prototype handles all packets with a FIFO strategy. To further reduce network latency for time-critical applications, we plan to introduce a form of packets prioritization by adopting a TSN-compliant scheduling strategy, and we already provide a QoS to specify this option on our streams. Such a strategy was available only in the Linux kernel until recent userspace implementation proposals [16], which may be easily integrated within INSANE.

Security. We leave a study of the security implications of INSANE to future work. To provide security guarantees with high network performance is probably our biggest challenge, especially because the security issues of acceleration technologies are still a hot research topic [47, 50]. Worse, the network edge is a far less controlled environment than a datacenter. The centralized approach of INSANE makes it easier for infrastructure providers to control the whole networking activity. Recent work on modern programmable network hardware suggests ways to further increase the security of kernel-bypassing network acceleration [57], which we are currently considering in our ongoing work to further extend INSANE, with no expectations of strong degradation of the excellent performance results of our middleware.

9 CONCLUSION

INSANE is a middleware for the edge cloud that integrates heterogeneous communication technologies such as kernel UDP/IP, XDP, DPDK, and RDMA. INSANE offer a minimal yet flexible API that eases the development of *portable* edge applications, in particular of latency-critical, network-intensive code (e.g., ML-powered applications). The user only needs to specify a set of high-level communication requirements, so that INSANE can map them at runtime to the most appropriate network technology available in the dynamically-determined deployment environment.

ACKNOWLEDGMENTS

The authors wish to express their gratitude to the anonymous reviewers for their constructive comments and suggestions, and to the many colleagues that reviewed previous drafts of the manuscript. This work was partially supported by the European Union's NextGenerationEU instrument, under the Italian National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.3, enlarged partnership "Telecommunications of the Future" (PE0000001), program "RESTART."

REFERENCES

[1] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. 2018. Mobile Edge Computing: A Survey. *IEEE Internet of Things Journal* 5, 1 (2018), 450–465. https://doi.org/10.1109/JIOT.2017.2750180

[2] Aaro Altonen, Joni Räsänen, Jaakko Laitinen, Marko Viitanen, and Jarno Vanne. 2020. Open-Source RTP Library for High-Speed 4K HEVC Video Streaming. In *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*. 1–6.

[3] Infiniband Trade Association. 2010. Supplement to InfiniBand Architecture Specification. Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE).

[4] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4, Article 11 (dec 2016), 39 pages.

[5] Ken Birman, Bharath Hariharan, and Christopher De Sa. 2019. Cloud-Hosted Intelligence for Real-Time IoT Applications. *SIGOPS Oper. Syst. Rev.* 53, 1 (jul 2019), 7–13.

[6] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. 2018. The Internet of Things, Fog and Cloud continuum: Integration and challenges. *Internet of Things* 3-4 (2018), 134–155. https://www.sciencedirect.com/science/article/pii/S2542660518300635

[7] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. 2021. WebRTC: Real-time communication for the open web platform. *Commun. ACM* 64, 8 (2021), 50–54.

[8] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. 2022. DcPIM: Near-Optimal Proactive Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 53–65.

[9] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77.

[10] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. arXiv:2305.05176 [cs.LG]

[11] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. *When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone.* Association for Computing Machinery, New York, NY, USA, 621–637.

[12] Hamza Djigal, Jia Xu, Linfeng Liu, and Yan Zhang. 2022. Machine and Deep Learning for Resource Allocation in Multi-Access Edge Computing: A Survey. *IEEE Communications Surveys & Tutorials* 24, 4 (2022), 2449–2494. https://doi.org/10.1109/COMST.2022.3199544

[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1–14.

[14] Janos Farkas, Lucia Lo Bello, and Craig Gunther. 2018. Time-Sensitive Networking Standards. *IEEE Communications Standards Magazine* 2, 2 (2018), 20–21. https://doi.org/10.1109/MCOMSTD.2018.8412457

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 51–64.

[16] Andrea Garbugli, Lorenzo Rosa, Armir Bujari, and Luca Foschini. 2023. KuberneTSN: a Deterministic Overlay Network for Time-Sensitive Containerized Environments. In *ICC 2023-IEEE International Conference on Communications*. IEEE, Rome, Italy, 1506–1511.

[17] Object Management Group. [Online]. OMG Data Distribution Standard. https://www.dds-foundation.org/omg-dds-standard

[18] Judy C. Guevara, Ricardo da S. Torres, and Nelson L.S. da Fonseca. 2020. On the classification of fog computing applications: A machine learning perspective. *Journal of Network and Computer Applications* 159 (2020), 102596.

[19] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 135–148.

[20] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. 1997. The Performance of μ-Kernel-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) *(SOSP '97)*. Association for Computing Machinery, New York, NY, USA, 66–77.

[21] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[22] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. 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, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[23] Sagar Jha, Lorenzo Rosa, and Ken Birman. 2022. Spindle: Techniques for Optimizing Atomic Multicast on RDMA. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 1085–1097.

[24] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µSecond-Scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 345–359.

[25] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[27] Magnus Karlsson and Björn Töpel. 2018. The path to DPDK speeds for AF XDP. In *Linux Plumbers Conference*.

[28] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 24, 16 pages.

[29] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. 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, 113–126. https://www.usenix.org/conference/nsdi19/presentation/kim

[30] Linux Foundation. [Online]. The Data Plane Development Kit. www.dpdk.org

[31] Linux Foundation. [Online]. DPDK Ring Library. https://doc.dpdk.org/guides/prog_guide/ring_lib.html

[32] Islem Mansri, Noureddine Doghmane, Nasreddine Kouadria, Saliha Harize, and Amara Bekhouch. 2020. Comparative Evaluation of VVC, HEVC, H.264, AV1, and VP9 Encoders for Low-Delay Video Applications. In *2020 Fourth International Conference on Multimedia Computing, Networking and Applications (MCNA)*. 38–43.

[33] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413.

[34] Laura Mazzuca, Andrea Garbugli, Andrea Sabbioni, Armir Bujari, and Antonio Corradi. 2022. Towards a Resource-Aware Middleware Support for Distributed Game Engine Design. In *Proceedings of the 2022 ACM Conference on Information Technology for Social Good* (Limassol, Cyprus) *(GoodIT '22)*. Association for Computing Machinery, New York, NY, USA, 409–413. https://doi.org/10.1145/3524458.3547126

[35] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 15 pages.

[36] Ahmed Nasrallah, Akhilesh S. Thyagaturu, Ziyad Alharbi, Cuixiang Wang, Xing Shao, Martin Reisslein, and Hesham ElBakoury. 2019. Ultra-Low Latency (ULL) Networks: The IEEE TSN and IETF DetNet Standards and Related 5G ULL Research. *IEEE Communications Surveys & Tutorials* 21, 1 (2019), 88–145.

[37] NVIDIA. 2019. ConnectX-6 Dx. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx

[38] NVIDIA. 2023. BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit

[39] NVIDIA. [Online]. RDMA Aware Networks Programming User Manual. www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

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

[41] OpenFabrics Interfaces Working Group. [Online]. libfabric. https://github.com/ofiwg/libfabric

[42] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 337–350.

[43] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4, Article 11 (nov 2015), 30 pages.

[44] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. 2021. MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 47–63. https://www.usenix.org/conference/atc21/presentation/planeta

[45] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341.

[46] Xuan Qi and Chen Liu. 2018. Enabling Deep Learning on IoT Edge: Approaches and Evaluation. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 367–372. https://doi.org/10.1109/SEC.2018.00047

[47] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)*. 4277–4292.

[48] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. 2021. 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), 2557–2589.

[49] Shree Krishna Sharma, Isaac Woungang, Alagan Anpalagan, and Symeon Chatzinotas. 2020. Toward Tactile Internet in Beyond 5G Era: Recent Advances, Current Issues, and Future Directions. *IEEE Access* 8 (2020), 56948–56991.

[50] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. 2020. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[51] Weijia Song, Yuting Yang, Thompson Liu, Andrea Merlina, Thiago Garrett, Roman Vitenberg, Lorenzo Rosa, Aahil Awatramani, Zheng Wang, and Ken Birman. 2022. Cascade: An Edge Computing Platform for Real-Time Machine Intelligence *(ApPLIED '22)*. Association for Computing Machinery, New York, NY, USA, 2–6.

[52] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. 1993. Implementing Network Protocols at User Level. *SIGCOMM Comput. Commun. Rev.* 23, 4 (oct 1993), 64–73.

[53] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (feb 2020), 36 pages.

[54] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. 2022. BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 249–262.

[55] Dapeng Wu, Y.T. Hou, Wenwu Zhu, Ya-Qin Zhang, and J.M. Peha. 2001. Streaming video over the Internet: approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology* 11, 3 (2001), 282–300.

[56] XDP-project. [Online]. libxdp. https://github.com/xdp-project/xdp-tools

[57] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. 2022. Bedrock: Programmable Network Support for Secure RDMA Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2585–2600.

[58] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211.