

Adopting User-Space Networking for DDS Message-Oriented Middleware

Vincent Bode*, Carsten Trinitis† and Martin Schulz¶

Department of Informatics
Technical University of Munich
Garching, Germany

*bodev@in.tum.de, †trinitic@in.tum.de, ¶schulzm@in.tum.de

David Buettner‡ and Tobias Preklik§

Siemens AG
T CED SES-DE

Munich and Erlangen, Germany

‡david.buettner@siemens.com, §tobias.preklik@siemens.com

Abstract—Due to the flexibility it offers, publish-subscribe messaging middleware is a popular choice in Industrial IoT (IIoT) applications. The Data Distribution Service (DDS) is a widely used industry standard for these systems with a focus on versatility and extensibility, implemented by multiple vendors and present in myriad deployments across industries like aerospace, healthcare and industrial automation. However, many IoT scenarios require real-time capabilities for deployments with rigid timing, reliability and resource constraints, while publish-subscribe mechanisms currently rely on components that are not strictly real-time capable, such as the Linux networking stack, making it hard to provide robust performance guarantees without large safety margins.

In order to make publish-subscribe approaches viable and efficient also in such real-time scenarios, we introduce user-space DDS networking transport extensions, allowing us to fast-track the communication hot path by bypassing the Linux kernel. For this purpose, we extend the best-performing vendor implementation from a previous study, CycloneDDS, to include modules for two widespread user-space networking technologies, the Data Plane Development Kit (DPDK) and the eXpress Data Path (XDP), and we evaluate their performance benefits against four existing DDS implementations (OpenDDS, RTI Connext, FastDDS and CycloneDDS). The CycloneDDS-DPDK and CycloneDDS-XDP extensions offer a performance benefit of 31% and 18% reduced mean latency, respectively, as well as an increase in bandwidth and sample rate throughput of up to 59%, while reducing the latency bound by at least 94%, demonstrating the performance and dependability advantages of circumventing the kernel for real-time communications.

I. INTRODUCTION

Developing flexible, fast and future-proof communication layers for systems is a challenge that many software projects face, with a multitude of project requirements and constraints often complicating the implementation of a seemingly simple software component. This is the reason why industrial systems have been leveraging communications middleware to out-source this burden to off-the-shelf implementations ever since the inception of the concept in 1968 [1]. For the Industrial Internet of Things (IIoT), the publish-subscribe architecture is a popular choice for message-oriented middleware because of its ease-of-use and extensibility. There are a multitude of publish-subscribe technologies, but one standard gaining traction is the Data Distribution Service (DDS), an API and protocol specification for distributed data-centric message exchange maintained by the Object Management Group [2] and

implemented in open-source and commercial systems backed by a large variety of vendors.

DDS as a messaging middleware features multiple benefits over the traditional socket communication model, including reduced coupling and increased dependability/extensibility by means of abstraction and encapsulation, as well as transparently handling communication life-cycle management, message tracking and retransmission. All of these make DDS an attractive candidate for adoption in IIoT scenarios, with deployments including robotics systems (via the integration into the Robot Operating System ROS2) [3], military communications [4], mobile medical diagnosis equipment [5], and factory automation [6, 7].

However, in many use cases we face reliability and timing restrictions, which requires every component in the system to offer real-time and reliability guarantees for the system as a whole to be considered dependable. This runs contrary to the pressure of adding ever more features into control systems, leading them to be more versatile at the cost of complexity. The latest push of integrating machine learning into time-constrained systems only adds to the difficulty of guaranteeing boundedness and reliability. These timing constraints are also a problem for most communication frameworks, including DDS implementations, as off-the-shelf middleware generally relies on technologies that do not offer real-time functionality or timing guarantees natively, like the Linux networking stack [8]. Despite this, DDS has found its way into many timing-critical applications, where careful configuration of DDS is necessary in addition to using large performance safety margins, as only this ensures safe system operations in both normal and adverse conditions. However, further adoption is hindered by a lack of studies showing reliable system operation under unfavorable conditions, making it harder for application developers to trust in the middleware’s resilience. We aim to advance this area of soft-real-time guarantees by reducing the number of real-time incapable components on the transmission path, allowing us to shrink the safety margins and with that improve overall performance, reliability and thus trustworthiness.

In this paper, we evaluate the existing soft real-time capabilities of four vendors and extend a vendor implementation to include support for two tried-and-tested user-space networking technologies, drastically improving latency bounds and reduc-

ing performance variability while further enhancing other key performance metrics. We offer the following contributions:

- Analysis of performance considerations and software engineering requirements when combining DDS with user-space networking technologies,
- Extension of the best-performing vendor implementation to include support for two major user-space networking technologies, improving its soft real-time characteristics,
- Evaluation of the reliability and performance benefits of both the DDS user-space networking extensions against four DDS implementations using DDS-Perf cross-vendor benchmarks.

To survey the state-of-the-art performance, we selected four DDS implementations that are widely adopted in the DDS community and that emphasize applications in IIoT scenarios. Apart from their significant total market share, vendors developing these systems specifically advertise system performance as a selling point. The first system, OpenDDS, is maintained by Object Computing, and is open-source. RTI Connex by Real Time Innovations is a performance-oriented but closed-sourced commercial implementation. Further open-source implementations examined include FastDDS (eProsima) and the relatively new CycloneDDS (Eclipse Foundation).

We chose two user-space networking technologies, DPDK and XDP, to demonstrate the usefulness of user-space networking for DDS due to their suitability for (Industrial-) IoT applications: (1) Data-Plane Development Kit (DPDK), a user-space networking technology maintained by the Linux Foundation and often referred to as the de facto standard for acceleration of packet processing [9], which relies on specialized drivers to bypass the kernel and deliver packets directly to DPDK-enabled applications; and (2) eXpress Data Path (XDP), a Linux kernel mechanism that uses Berkeley Packet Filters (BPFs) to do in-kernel packet processing outside the regular network stack [10]. The former is ubiquitous in performance packet processing and the latter was recently mainstreamed into the Linux kernel, thus both options are production-ready and widely available. We will be examining performance aspects of using the chosen network technologies with DDS throughout this paper. To the best of our knowledge, we offer the first implementations of user-space networking extensions for any DDS system, which is available to the public as open-source¹.

The remainder of this paper is structured as follows: We first introduce DDS and examine its current real-time capabilities in Section II. The case for user-space networking and the two chosen technologies is made in Section III. In Section IV, we cover implementation aspects of the CycloneDDS extensions and subsequently evaluate the performance obtained in Section V. Finally, in Section VI we discuss the results and possibilities for future work.

II. DDS AND CURRENT REAL-TIME CAPABILITIES

The advantages of the publish-subscribe paradigm have helped it spread throughout communications technologies, with both middleware systems like MQTT and protocols like IGMP adopting the pattern. It supports a strong decoupling of sender and receiver, which makes it appealing for dynamic networks of distributed systems. DDS is a pub-sub middleware standard built for a very wide range of use-cases, and it brings the necessary configurability and adaptability to suit many applications. Due to implementations and backing by over a dozen vendors and two decades of standard updates, the range of deployments has grown and now pervades many sectors. Unlike many of its competitors, DDS also does not rely on a centralized broker or message queue but functions fully distributed. Because industrial use-cases often prioritize reliability, performance and resource-consumption requirements, the ability to pick from a diverse set of implementations makes DDS attractive for projects that need solutions tailored to their needs. This diversity comes at the cost of complexity, though, as assessing the capabilities of DDS as a whole requires studying a range of implementations. To truly evaluate the performance benefits of user-space networking for DDS, we will consider four major DDS implementations, covering both commercial and open source implementations.

A. Previous Work

From the beginnings of DDS with the standard's finalization in 2004 [2], OMG has continued to norm the DDS ecosystem, publishing three further core standards and 15 auxiliary standards. These all regulate facets of the DDS ecosystem today, specifying anything from programming language specific APIs to wire formats for interoperability. It is not uncommon, however, to find vendor-specific extensions in place of standardized mechanisms, as these often predate standardization. Because of the lack of an official compliance certification process for DDS, benchmarking DDS implementations is fairly challenging. Thus, cross-vendor benchmarking tools have been developed, like DDS-Perf [11] to hide the complexity of dealing with multiple DDS vendors and to enable the evaluation of performance in a platform-independent way.

In the past, other publish-subscribe messaging standards, like OPC UA or ZeroMQ, have been assessed against one or more DDS vendors [12, 13]. However, these studies lack any analysis of specialized tuning for real-time aspects, such as latency bounds. Previous work has also examined higher-level protocols built on top of DDS, such as ROS2 for robot communication [3]. An investigation was conducted to evaluate DDS performance in the real-time context [14], which found a reduction in latency jitter was possible by changing network driver settings, however it focuses on just a single and unnamed DDS vendor and a specific network card. Still others have benchmarked the performance of different DDS extensions, like DDS-XRCE, for resource constrained environments [15], or the virtualization overhead of containerizing DDS [16]. Many comparison studies are vendor driven, but in

¹Git Repository: <https://github.com/caps-tum/cyclonedds-dpdk-xdp>

2013 an independent survey compared two DDS libraries to each other (OpenSplice and RTI) [17].

Surprisingly, to the best of our knowledge there have been no efforts to examine combining DDS with user-space networking technologies, despite their well-known advantages of minimizing kernel interference in communication for latency and bandwidth [10], and the widespread adoption in other disciplines, such as high-performance computing. Some adjacent efforts are currently ongoing at the time of writing to harmonize DDS with Time-Sensitive Networks (TSNs) [18], however TSNs are a much more involved solution that requires extensive hardware and software support to provide deterministic Ethernet behavior. In contrast, user-space networking is implemented mostly in software, leading to deployability advantages. Additionally, while high hopes are placed on DDS-TSN and some vendors already claim TSN support [19], these developments are in very early stages and currently lack production readiness. In fact, we have not yet been able to reproduce any DDS vendor’s TSN demonstration. Meanwhile, DPDK and XDP are very well established user-space networking technologies, and can be seen as a compromise between highly specialized TSN and widely-deployed traditional Linux networking. It is here that we can start to explore accelerating DDS using DPDK and XDP.

B. The Problem: Latency Bounds

Previously, Bode et al. conducted performance studies on DDS using DDS-Perf across the four vendors. There is a general performance study available [11], as well as a real-time focused study [20]. Throughout the real-time focused study, several hardware- and software-based tuning techniques were applied to optimize the out-of-the-box performance of each vendor, targeting the reference performance metrics provided by DPDK’s Layer 2 benchmarking tool `l2reflect`. While they were able to bring down average latencies to around $100\mu\text{s}$ for the better performing systems (CycloneDDS, followed by FastDDS), the maximum observed latencies among 3 million packets were still an order of magnitude higher, with rare but significant latency spikes of up to $4000\mu\text{s}$ for the worst performing systems (OpenDDS) even after tuning. While this far outperforms the out-of-the-box latency bounds (OpenDDS: $17400\mu\text{s}$), a latency outlier will still observe up to $40\times$ the mean observed latency [20].

A 0.0001% (or one-in-a-million) chance of a deadline missed by 4ms might seem insignificant, but this is rather unacceptable for many types of industrial control systems. Consider a piston engine controller that actuates ignition of the combustion chamber. The controller needs to actuate the ignition according to the input parameters with very reliable timing, potentially thousands of times per second. At a signal rate of 1kHz (1000 occurrences per second), an outlier occurs once every 15 minutes, which can cause either service outage (engine stall) or potential damage to the machinery through e.g. misfires if the delayed signal arrival is not handled correctly. Consequences are not just limited to property damage, a similar mean time between failures

(MTBF) to meet the deadline in a life-support system such as an oxygen flow monitor would mean opening a window for undetected dangerous conditions, failing to trigger necessary alarms and thus posing obvious health and safety risks. While redundancy is often employed to reduce risks of failure, it can only try to mitigate the effects but cannot eliminate the root cause of the failure. Clearly, both the rate of occurrence of outliers and their magnitude need to be further reduced before we can consider introducing DDS as a soft-real-time technology suitable for these applications.

III. THE CASE FOR USER-SPACE NETWORKING

Traditional real-time applications are implemented close to the hardware, often with embedded technology stacks that are lightweight and purpose-built. DDS however, not being a technology aimed solely at real-time applications and their constraints, sits on top of a much more heavy-weight technology stack, with a typical setup relying on a fully fledged operating system (in our case Jupiter, a real-time flavor of Linux) and the applications and drivers that come with it. The ability to use commodity hardware and software offers a multitude of advantages for development and operations/maintenance, but each additional layer of abstraction and isolation makes it more difficult to provide reliability and timing guarantees across the entire stack. Furthermore, technology stacks are growing in size and complexity, with the advent of connected, smart and AI-enabled devices only adding to the existing difficulties. Whereas an application running on a programmable micro-controller typically has direct and immediate hardware access to its communication facilities, a message transmission on a typical server system traverses many layers of software and hardware before arriving at its destination.

When looking at the technology stack that a sample (DDS term for a message/packet) needs to traverse during delivery (Figure 1), it may be surprising that the DDS implementations can even reach the average performance of real-time oriented technologies as demonstrated by Bode et al. [20]. Especially the path through the kernel is often expensive, as the `write` and `read` system calls cause processor mode switches and management overhead that make the call alone around 100-500 times more expensive than a regular function call on x86-64 [21], and that overhead is incurred before the kernel even starts doing anything useful. Afterwards, the packet needs to traverse the kernel I/O routines, the network stack and the NIC driver before being sent off over the wire. Repeating this process thousands of times per second is therefore not only expensive, but also prone to interference from resource contention, which can cause critical delays in delivery and unexpected packet delay variation. Many DDS systems take steps to minimize the effects of this problem, e.g., CycloneDDS uses vectored I/O using `readv` and `writv` to reduce the total number of system call invocations. However, the problem remains that all packets need to eventually traverse the kernel and its real-time incapable network stack twice on the critical transmission path, or four times when

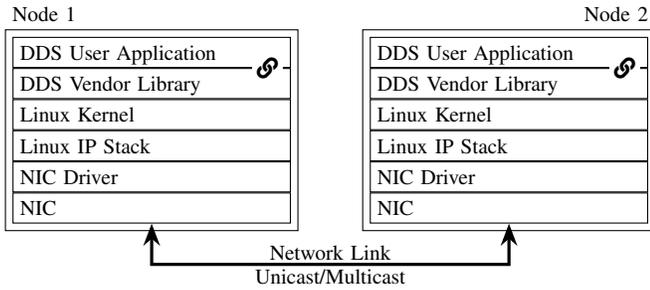


Fig. 1. The technology stack that a DDS sample (DDS term for a message/packet) needs to traverse downwards on the publisher and upwards on the receiver. The many software layers make it difficult to provide timing or reliability guarantees as each layer can introduce unexpected delays.

a round-trip is necessary to handle, e.g., acknowledgements during reliable message delivery.

DDS is not the first communication system that encounters this problem. Disciplines, such as high performance computing or high-throughput packet processing, have already struggled against similar issues with performance-critical networking. Eventually, the idea was born to circumvent the kernel entirely and thus cut out several layers of software from the critical path. This technology is now known as user-space networking, allowing a user-space application or library (such as DDS) to communicate directly with the hardware in user code without involving the kernel and its related security mechanisms (except for initial communication setup, which is still secured by the kernel). While this has obvious performance improvement potential, it comes at the cost of losing many of the services the kernel offers in the first place, such as resource sharing, per-sample security or safety measures, and memory management. Despite these drawbacks, user-space networking is still viable for specialized deployments, especially in scenarios where dedicated resources can be made available, such as in high performance computing or embedded use-cases. To enable users of DDS to leverage the advantages offered by user-space networking technologies, we adapted a high-performance DDS implementation (CycloneDDS) to work with the DPDK and XDP user-space networking technologies to examine potential performance and reliability advantages. We implement this functionality as two extension modules, CycloneDDS DPDK-L2 and CycloneDDS XDP-L2, that operate directly on Layer 2 Ethernet to facilitate hardware-accelerated message exchange.

A. DPDK

The Data-Plane Development Kit (DPDK) is a user-space networking technology originally developed by Intel and now maintained by the Linux foundation [22]. It relies on specialized NIC drivers that allow it to circumvent the regular Linux networking stack and still take advantage of hardware capabilities like offloading. As such, DPDK applications usually have exclusive access to the network interface (although mechanisms for sharing exist) and the NIC is practically invisible to Linux and any other networked application. A DPDK driver is referred to as a Poll-Mode Driver (PMD), as it largely avoids expensive interrupts in favor of polling

for incoming messages in a run-to-completion model. DPDK is frequently deployed in high-performance packet processing applications with processing requirements beyond 10 million packets per second [23].

B. XDP

The eXpress Data Path (XDP) [10] is an in-kernel packet processing mechanism that can also be used to implement user-space networking. It allows programs to inject a Berkeley Packet Filter (BPF) into the kernel that executes packet processing logic like routing, rewriting or replying. BPFs can also selectively redirect packets into user-space programs before they enter the Linux network stack, which we use to deliver DDS packets directly to our CycloneDDS-XDP extension. With sufficient hardware and software support, BPFs can run directly on the NIC using offloading, as BPFs are compiled to specialized byte code using LLVM designed to be Just-In-Time (JIT) translated for the NIC at runtime. Due to its integration with the Linux kernel, XDP-enabled applications can be deployed on virtually any modern Linux system.

C. Comparison

While both technologies shift kernel responsibilities to the application developers in order to improve performance, their underlying design philosophies make them distinct. Of these, the following trade-offs are relevant for users developing and deploying applications:

a) *Portability*: While DPDK requires special drivers that need to be managed by the user, XDP has more widespread support due to its integration in the Linux kernel. In fact, XDP programs can run either on the NIC directly, in the NIC driver, or in the Linux kernel as a fallback depending on the hardware and software support available. Thus, XDP can live entirely without hardware and driver support if necessary, albeit at the cost of performance. An XDP enabled program can be shipped like any other Linux application, and it can load the BPF program into the kernel by itself given sufficient privileges. Meanwhile, DPDK is a much more disruptive solution: while the application itself can be delivered as usual, a working DPDK setup requires the user or administrator to work through an 11-chapter “Getting Started” guide [22] to set up libraries, drivers, and kernel modules and to manually configure the NICs by hand, which is a much more involved process.

b) *Interoperability*: The exclusive NIC access required by DPDK is another potential issue for users looking to adopt a user-space networking technology. While special bifurcation drivers exist [22], allowing DPDK applications to share the NIC with regular applications that require Linux network functionality given that hardware support is available, the setup is complex. Meanwhile, user and kernel space networking coexistence is supported out of the box with XDP’s BPFs, with a BPF only acting as intermediate logic between the hardware and the kernel. We use this functionality to redirect only packets that match the CycloneDDS XDP-L2 extension’s `ethertype`, letting all other traffic take its regular path through the kernel.

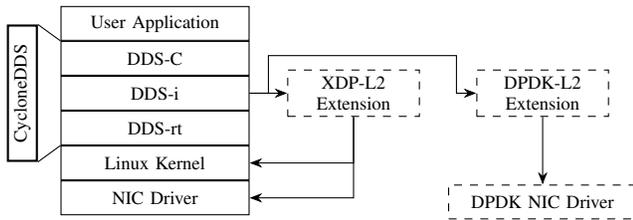


Fig. 2. The CycloneDDS system stack. We intercept messages at the implementation layer (DDS-i) for both the XDP-L2 extension (center) and the DPDK L2 extension (right). In the case of XDP, the extension forwards it through an XSK socket to the kernel or NIC driver (depending on the hardware and software support). On the other hand, the DPDK extension communicates directly with the specialized DPDK driver.

c) *Performance*: While the above aspects generally weigh in favor of XDP, DPDK’s strong point is performance. Performance aspects were considered to be so important for implementing DPDK, that the manual even includes information about how many cache lines are loaded per packet and which lines need to be cached for efficient access to header fields. In fact, developers of XDP are using DPDK as a performance benchmark [24] to optimize their implementations. Because of this, we expect DPDK to significantly outperform XDP as a transport layer for DDS, at the cost of the aforementioned drawbacks.

Due to the slightly different use-cases for these technologies, we decided to adopt both technologies for use in DDS. This will allow us to compare their suitability against each other and the regular Linux IP stack, while also allowing users to choose the right technology based on their requirements.

IV. IMPLEMENTATION

For our study of user-space networking in DDS, we chose CycloneDDS as our base implementation. Compared to some traditional DDS systems, it is a comparatively modern implementation of the DDS standard. Among the open source systems, it offers several advantages when implementing new transport technologies, the most important being that CycloneDDS is structured into layered components (Figure 2). The uppermost layer (DDS-C) consists of user-facing APIs specified by the DDS standard, as well as CycloneDDS specific extensions. The DDS-C layer then interfaces with the implementation layer (DDS-i), which includes most of the business logic required to implement a DDS system. The final layer is the runtime layer (DDS-rt), which handles platform and environment specifics, such as the implementation differences between Linux and other operating systems as well as different levels of hardware capability.

The layered system allows us to effectively intercept the internal API calls made by CycloneDDS without too many changes to the core CycloneDDS source code itself. Besides its good performance in the previous studies [20], this was a key reason why CycloneDDS is our first choice for implementing user-space networking extensions. We choose to inject the DPDK-L2 and XDP-L2 extension code near the lower end of

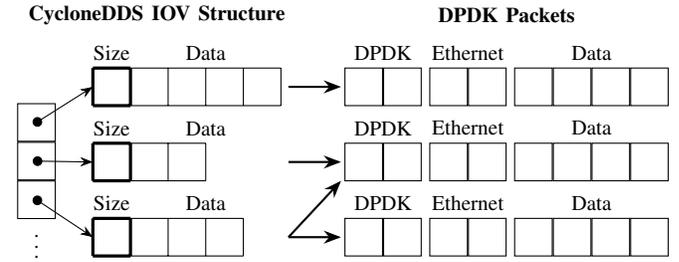


Fig. 3. The difference between the CycloneDDS/Linux IOV buffer format (left) and the DPDK packet buffer format (right). CycloneDDS buffers cannot be cleanly wrapped in DPDK packets, which is why the serialized data needs to be copied when traversing the extension on the outbound path.

the DDS-i layer. Inserting the module at this position allows us to take advantage of the following properties:

A. Automatic Entity Lifecycle Management

The CycloneDDS DDS-i layer implements the discovery of remote participants (using the standardized DDS Simple Participant Discovery Protocol, SPDP) and the discovery of remote endpoints (SEDP, respectively) in the upper half of the DDS-i layer. Because both our L2 extensions are implemented in the lower half of the layer, we can take full advantage of CycloneDDS’s integrated discovery, liveness detection and security mechanisms for the small cost of needing to support the integrated addressing schemes (locators) and multicast communication. Since the underlying Layer 2 technology (Ethernet) supports both multicast and broadcast delivery, implementing this boils down to effectively managing the DPDK Poll-Mode Driver (PMD) or the XDP kernel data structures and efficiently converting CycloneDDS messages to Ethernet frames and back. The ability to use built-in discovery and lifecycle management facilities thus allows us to support all related Quality of Service (QoS) settings out of the box and enables us to maximize standards compliance while minimizing additional user configuration and code maintenance.

B. Serialization and Buffer Management

Similarly, the DDS-i layer contains the necessary logic to serialize and deserialize the samples that the user wants to transfer in the upper half of the DDS-i layer. This allows the L2 extensions to benefit from the built-in standard serialization mechanism, which future proofs the extension by allowing it to leverage vendor encoding interoperability, future CycloneDDS support updated wire formats, and DDS extensible data types. It also relieves the extensions of taking care of data alignment, endianness, and other typical networking burdens. However, intercepting CycloneDDS communications below the message serialization sublayer also comes with a downside: it is not possible to control the buffers that CycloneDDS serializes its data to. This is problematic because CycloneDDS serializes its messages into several smaller buffers to pass them to the kernel via the IOV Linux kernel data structures (used by the syscalls `readv` and `writew`) to make passing the data to the kernel efficient (Figure 3, left). The IOV data structure consists

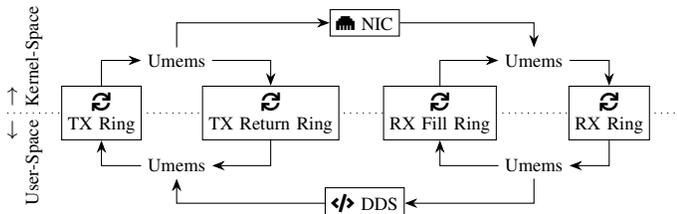


Fig. 4. The passing of data frames between user- and kernel-space in XDP. A umem is allocated by the user-space application and loaned to the kernel by either filling it with data and submitting it to the TX ring for sending, or by submitting it to the RX fill ring to let the kernel fill it with incoming packets. The kernel processes the requests and returns umems via the sibling ring.

of an array of pointers to buffers that each have a size and a variable-length sequence of data octets. Usually, the outgoing packet will therefore not lie contiguously in memory.

Meanwhile, the packet buffer structure that DPDK expects is also rigid (Figure 3, right), with each buffer containing DPDK managed metadata, followed by an Ethernet header managed by the DPDK-L2 extension and a fixed number of data octets for payload. To avoid needing to allocate the DPDK buffers when packets are sent and received, they are stored in a memory pool by our extension. However, since both the CycloneDDS and the DPDK buffer formats are ultimately outside the extension’s control, creating a copy of the data is unavoidable. When constructing DPDK frames to back the serialized CycloneDDS data stream, two possibilities can occur for each data fragment in the IOV structure:

- 1) If the data in the IOV fragment fits into the current DPDK frame, it simply needs to be copied from the IOV buffer into the DPDK frame’s data buffer and then the frame is immediately transmitted through the DPDK driver.
- 2) If the data in the IOV buffer does not fit into the DPDK frame, the rest of the available space in the DPDK frame is filled with data from the IOV, the frame is queued for transmission, and a new frame is retrieved from the memory pool. This step is repeated until all data from the IOV fragment is packed into DPDK frames.

After one IOV fragment is successfully packed into the DPDK frame, we reiterate the algorithm above until all buffers in the IOV data structure have been processed. This replicates the fragmentation of the data stream into packets normally implemented in the kernel and NIC driver, which needs to be reversed for reconstruction of the original data stream on the receiving side by defragmentation and reordering. DPDK/XDP shift both these responsibilities from the kernel to the user.

A similar problem exists for XDP, where the extension also needs to manually assemble Ethernet frames in user-space. The frames are passed between the user-space application and the kernel or NIC driver via special blocks in user-space memory called `umems`. This works by using four single-producer single-consumer rings, two each for the TX path and the RX path (Figure 4), which the `umems` traverse in cyclic fashion. For each path, one side is responsible for filling the buffers while the other side empties them. If hardware

support is available, then the NIC can send packets directly to or from the `umem` buffer without needing to copy it first. However, we cannot get CycloneDDS to read from/write directly to the `umem` buffer, which is why at least one data copy is still necessary. If CycloneDDS offers an API to control serialization buffers in the future, zero-copy all the way from the serialization to the NIC hardware would become viable in both extensions, potentially enhancing performance further.

C. Independence from Kernel Data Structures

In contrast to the DDS-rt layer, which interacts very closely with the kernel, the DDS-i layer is still largely independent of the underlying kernel and physical environment and thus more portable. The relative distance in the DDS-i layer from the kernel makes circumventing the kernel for user-space networking easier, since we do not need to replicate much of the kernel’s functionality, which we want to avoid for performance and code complexity reasons. It also allows us to bypass both the DDS-rt layer and the kernel simultaneously to interact directly with the NIC driver, further shortening the critical path for increased latency performance, while eliminating another potential source for unexpected latency.

However, the abstraction is not perfect and for efficiency reasons there are exceptions where CycloneDDS hands kernel data structures all the way through to the DDS-i layer. One of these is the file descriptor (socket) used for network communication. CycloneDDS typically uses many sockets simultaneously for communication, and the DDS-i layer manages a collection of file descriptors to make this possible. Apart from configuring settings on the sockets, the CycloneDDS DDS-i layer repeatedly queries the DDS-rt layer for the next file descriptor that has data available for reading, which is implemented in the DDS-rt layer using the `select` system call. While the need to provide sockets is not an issue for the XDP-L2 extension, since the XSK API (XDP sockets) provides us with a pseudo socket that can be used with the aforementioned system calls, this does pose a problem to the DPDK-L2 extension, as there are no valid socket handles it could provide the DDS-i layer for managing.

There are two possible solutions to this problem: Either the DDS extension creates proxy file descriptors in the kernel or the need for having socket handles is removed altogether in favor of a different method of querying for available data. Since going through the kernel using proxy file descriptors defeats the purpose of using user-space networking technologies in the first place, we utilize the second option and modify the DDS-i layer to eliminate the need for file descriptors. Unfortunately, this is not possible without modifying the CycloneDDS core source code. However, it is possible to trick the DDS-i layer into automatically skipping most socket operations if it is told which transport to next read from ahead of time, before it attempts to determine it by itself. We implement this with a simple rule: when the DPDK/XDP extension is active, we assume that the extension is always the transport that should be read from. This simplification allows us to implement the DPDK-L2 extension with only moderate modifications to

TABLE I

IMPORTANT *Application & QoS* SETTINGS USED IN DDS-PERF [20]

Topics	Sample Rate (1/s)	Size	Type	Randomization
2	1000 or unlimited	32 Bytes	Unkeyed	Partial
Reliability	History	Durability	Deadline	Latency Budget
Reliable	KEEP_ALL	VOLATILE	INFINITE	0

the CycloneDDS core, and without compromising the regular functioning of CycloneDDS when the module is not loaded.

D. Implementation Results

The aforementioned advantages of this design allow us to implement both of these extensions quite compactly. Because some functionalities in the extensions like packet handling are similar regardless of the actual user-space networking technology used, we can modularize the extensions, saving several hundred lines of code. This results in a total code base of under 3000 lines of extension code, with the XDP extension encompassing more code than the DPDK extension due to the need to manage the kernel BPF module. We deem it a particular success that the number of code changes inside the CycloneDDS core are minimal, with under 50 modified lines of code. The DPDK and XDP extensions can be compiled into CycloneDDS and activated at runtime using the same Quality of Service (QoS) settings file that also controls all other aspects of a DDS system’s behavior, ensuring easy configurability using established DDS mechanisms. Finally, with the completed extensions available in our deployed version of CycloneDDS, we now quantify the performance benefits that can be achieved by benchmarking the implementation.

V. BENCHMARKS

To assess the intended advantages of user-space networking extensions for DDS against the traditional Linux networking stack, we require a benchmarking approach that is platform and DDS implementation agnostic. To achieve this, we leverage DDS-Perf [11] and adopt the same environment employed by the authors in a prior real-time study [20]. This maintains consistency in hardware and software configurations, ensuring a fair and comparable evaluation with the original findings.

A. Software Setup

The DDS-Perf application is configured for two participants: one generating samples via Topic 0 and the second relaying them back to their origin in Topic 1. A 32-Byte sample is generated every millisecond, which corresponds to a fixed sample rate of 1000 samples/s each way and 64 KiB/s total bandwidth usage due to bidirectionality. This remains comfortably below peak performance for all vendors, even on lower-end hardware [11], providing a safety margin akin to critical systems. All samples are unkeyed (a DDS feature we do not require for benchmarking), and partially randomized to make compression more difficult, emulating real-life conditions.

For comparability, we need to ensure that DDS’ Quality of Service (QoS) settings are identical across all implementations (Table I). To mirror the requirements of critical systems,

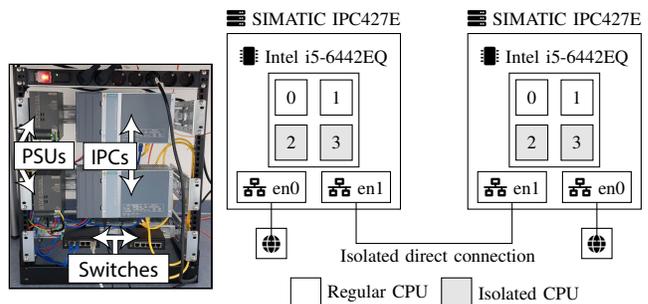


Fig. 5. The Siemens SIMATIC IPC427E systems for industrial IoT use (left side), each with the IPC box running a real-time tuned variant of Debian, a PSU, and the network. Schematic of the hardware setup (right side): Each system has two four-core CPUs with two isolated cores each connected by 2× Gigabit Ethernet switches.

we ask DDS to ensure strict reliability (the combination of RELIABLE reliability and KEEP_ALL history), as we want to ensure every sample is delivered. A viable alternative that is less demanding on the DDS implementation would be BEST_EFFORT reliability or a lower history setting for systems that can tolerate interim packet losses. There are many more available QoS settings in DDS, with the OpenDDS manual [25] listing over 100 of them. Generally, we choose all unmentioned settings to be least constraining, allowing the DDS vendor to choose the optimal setting freely. Auxiliary data collection (such as the collection of system metrics) occurs in a separate process to avoid interfering with DDS-Perf or the underlying DDS library. Orchestration occurs outside the test system by a dedicated manager node.

B. Hardware Setup

We evaluate performance on two dedicated Siemens IPC units (Figure 5) running Debian 11.5. These systems use a PREEMPT_RT kernel (Linux jupiter 5.18.0-0.deb11.4-rt-amd64) for low latency scheduling. Each unit has an Intel i5-6442EQ @ 1.90GHz CPU, with two general purpose cores and two isolated cores (via the `isolcpus` kernel option) useful for interference-free computation. This prevents any tasks from being scheduled on them unless the corresponding task affinity is set first, but load balancing does work between the cores if the affinity allows it. The systems do not use Non-Maskable Interrupts (NMIs). Each system has 8GB of DDR4 RAM, with swap turned off to prevent latency caused by page faults, and is interconnected by one general purpose and one dedicated network. The dedicated network is wired directly point-to-point, with no intermittent switches to avoid additional packet delay variation. All components are rack-mounted, using passive cooling. With the system environment in place, we must now reproduce the original study’s DDS configuration.

C. Setup Reproduction

In the original study [20], the authors iteratively refined their system configuration to improve performance of the vendors with DDS-Perf. To make our results comparable, we replicate

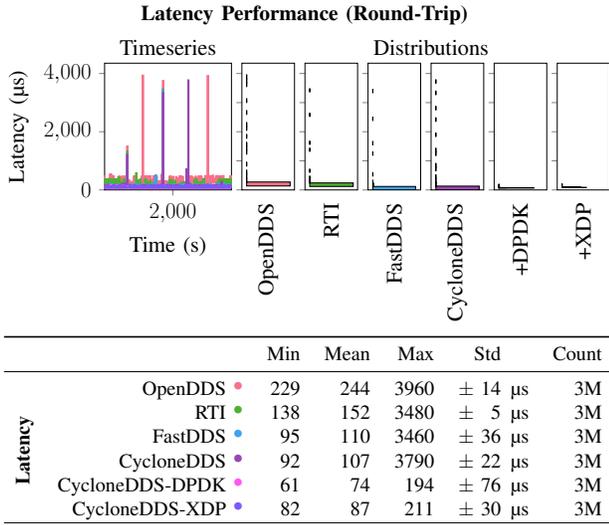


Fig. 6. Latency analysis of the original four vendors and the two user-space networking extensions, with a time-series (left) and vertical histograms (right). The user-space networking extensions outperform the other systems in all latency metrics, with the DPDK surpassing the XDP extension. Apart from the latency reduction, the performance variability is also reduced.

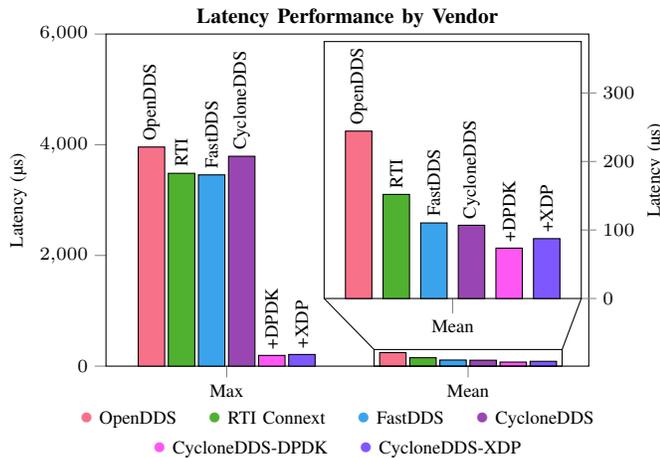


Fig. 7. Latency performance summary for maximum observed latency/latency bound (left) and average latency performance (right). The CycloneDDS extensions offer 95% improved latency bounds and at least 18%–31% improved average latency over the traditional DDS systems.

the final configuration from Section VI.E, requiring us to perform the same hardware and software tuning. Both the core isolation and the network isolation were reproduced on the hardware side and the relevant QoS and kernel adjustments were made on the software side. We verified the equivalency of our setup by comparing with the original performance data. Each profile shown contains at least 1 million samples, and while we show only one trial each experiment was validated at least three times.

D. Results

Latencies and latency bounds are the primary concern in critical real-time systems. These are optimal when the

workload’s characteristic overall system utilization is low, which means sending small samples and limiting at a rate of 1000 samples/s. From the results in Figures 6 and 7, we observe a significant round-trip time improvement for both user-space networking technologies. For CycloneDDS-DPDK, we achieve a mean latency of $74 \mu\text{s}$, equivalent to a reduction of 31% on our test system when compared to the regular implementation of CycloneDDS. As expected due to the larger involvement of the kernel, the performance of XDP is slightly worse than DPDK at $87 \mu\text{s}$, but this still corresponds to a latency reduction of 18% over the status quo. A typical round trip for the worst performing system, OpenDDS, will now take around 3x the time it takes to complete a CycloneDDS-DPDK round trip even after tuning OpenDDS. The better mean latency performance of our extensions allows us to effectively deliver sub-0.1-millisecond round-trips, or sub-0.05-millisecond derived one-way latency, exceeding what was possible using conventional DDS systems.

Previously, we also highlighted the importance of minimizing latency outliers. Our results indicate progress here as well, with a maximum observed latency of $194 \mu\text{s}$ and $211 \mu\text{s}$, respectively. Where previously we observed rare but high magnitude outliers of up to 35x the mean latency for CycloneDDS, with CycloneDDS-DPDK this margin is reduced to only 2.6x. While this still exceeds the latency bound of DPDK’s native tool `l2reflect` [20], it is nevertheless a sizable 95% improvement on the status quo. The corresponding worst latency in 1 million packets observed (with 0.0001% chance of occurring) is $184 \mu\text{s}$ for CycloneDDS-DPDK and $203 \mu\text{s}$ for CycloneDDS-XDP, demonstrating considerable performance-reliability improvements. This is also seen in a substantially reduced packet delay variation (PDV), which is more than halved to 50% and 57% lower PDV than regular CycloneDDS, respectively. For the first time on this hardware/software combination, we have thus achieved a round-trip time latency bound well below 1 millisecond for around 3 million packets.

While we focus on latency for real-time behavior analysis, other performance benefits of the CycloneDDS extensions are also of interest. We determine performance under two additional workload characteristics that exhaust the system capacity, one optimal for sample rate and one for bandwidth. If we remove the target throughput of 1000 samples/s, we can determine the peak sample rate for our testbed (Figure 8), achieved through small samples at high frequency. CycloneDDS-DPDK reaches a mean sample rate of more than 250 000 samples/s, constituting a 59% increase over regular CycloneDDS with 167K samples/s, albeit with a somewhat higher performance variability. Curiously, CycloneDDS-XDP doesn’t reach the same level of performance as CycloneDDS-DPDK or regular CycloneDDS, even though both extensions rely on much of the same code. We infer that the process of reading packets one-by-one hurts XDP performance to the point where regular kernel packet-batching outperforms the XDP peak packet processing rate, while reading individual packets does not have such a big performance penalty for DPDK.

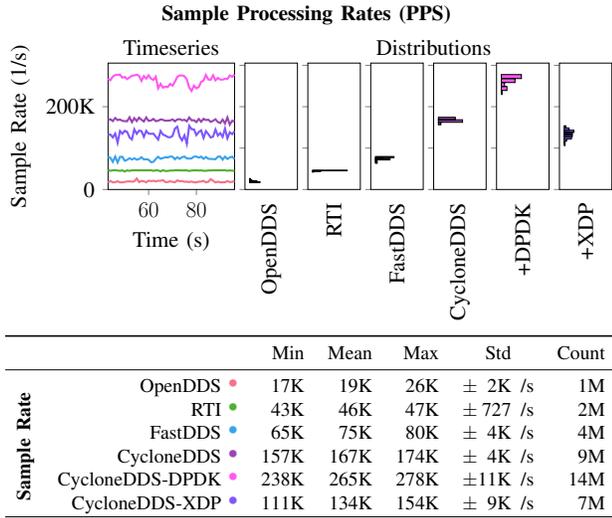


Fig. 8. Maximum sample rates achieved by the DDS implementations at 32 Bytes/sample. While CycloneDDS-DPDK is able to reach extremely high sample rates, CycloneDDS-XDP cannot maintain the throughput that regular CycloneDDS achieves. All three variants of CycloneDDS outperform the other systems, with OpenDDS achieving the lowest peak sample rate overall.

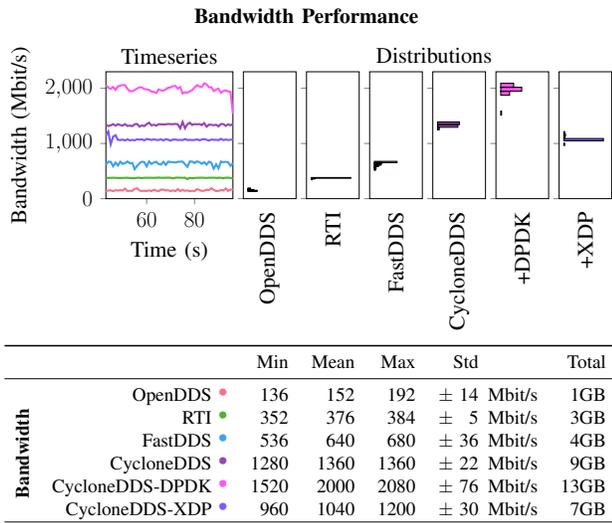


Fig. 9. Maximum achievable DDS bandwidth for the extensions at 1KB/sample. Similarly to peak sample rate performance, CycloneDDS-DPDK outperforms in terms of throughput, but regular CycloneDDS beats CycloneDDS-XDP. All three CycloneDDS-based implementations show a performance lead on the competing implementations.

A similar effect is also visible for the maximum achievable bandwidth (Figure 9), achieved by transferring large samples at high frequency. CycloneDDS already performs quite well by default, with a 1360 Mbit/s bandwidth saturating most of the available NIC bandwidth of 2000 Mbit/s (Gigabit Ethernet is 1000 Mbit/s full duplex, so double the data rate is possible for bidirectional communication). With the DPDK accelerated extension, we can saturate the entire interface, averaging at 2000 Mbit/s. Again, CycloneDDS-XDP falls slightly short, but the mean bandwidth of 1040 Mbit/s still exceeds the 1 Gbit

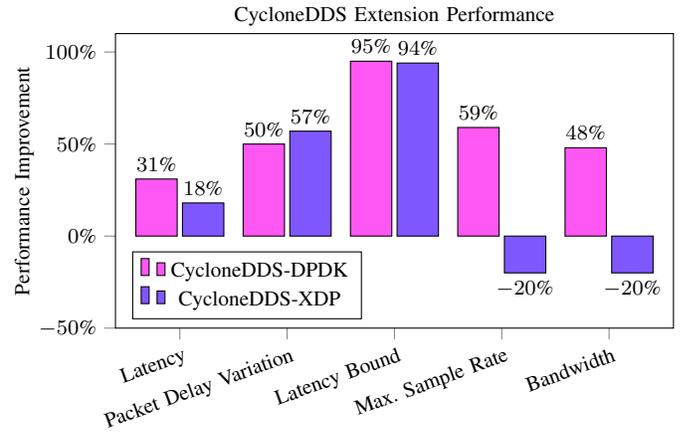


Fig. 10. The relative performance gains for the CycloneDDS user-space networking extensions (higher is better). CycloneDDS-DPDK outperforms regular CycloneDDS in every performance metric significantly. CycloneDDS-XDP also has significant improvements for both target metrics latency and packet delay variation, however these benefits come at the cost of slightly reduced maximum sample rate and bandwidth.

mark and outperforms the other DDS implementations.

In summary, CycloneDDS user-space networking extensions can significantly enhance various key performance metrics (Figure 10) compared to the regular CycloneDDS implementation. The largest improvement is evident in the drastically reduced latency bound, which is one of the most important performance metrics for industrial applications. Since CycloneDDS already offers very competitive performance out of the box and after tuning, this also means that the extensions outperform competing DDS implementations, resulting in the best observed DDS performance on this system to date. User-space networking technologies still have their usual downsides due to their invasive nature, but the performance benefits still makes their adoption attractive for specialized deployments. XDP is promising as it integrates DPDK-like latency performance advantages seamlessly without infrastructure disruption. For critical performance needs, we recommend the DPDK extension, which further enhances sample rate and bandwidth, compensating for deployability drawbacks.

VI. DISCUSSION & FUTURE WORK

While embedded systems still have the edge when it comes to real-time communications, the gap between hard and soft real-time systems is closing. Our CycloneDDS extensions contribute to DDS as a performance-oriented communication middleware catching up, but there are other developments in the DDS world working in the same direction.

IEEE 802.1 Time Sensitive Networking (TSN) is a family of standards that brings real-time capabilities and determinism to Ethernet networks. With compatible hardware (network interfaces and switching equipment), it is possible to schedule traffic and allocate network streams that can be used interference free, which will further reduce unpredictable latency by providing network delay bounds at the hardware level. While TSN capabilities are currently implemented within

the Linux kernel, there is also the possibility of combining TSN capabilities with user-space networking technologies to get both deterministic packet handling and the performance benefits from circumventing the kernel. At the time of writing, discussions are ongoing about XDP's suitability for TSN delivery, but no proof of concept is yet available [26].

Another possibility for increasing efficiency is removing the need for data copying on the transmission and reception paths, which is especially cumbersome and thus latency inducing for large message sizes. This is theoretically possible because DDS controls the data structures that the application uses to communicate, they are defined at compile time in Interface Definition Language (IDL) structs. Thus, data could be laid out with the necessary headroom for packet headers in DPDK or `umem` buffers so that the packets may be filled by the application, which just sees its regular IDL-defined data structure. The headers would then be filled in-place and the packet could be directly written to wire by the NIC. While this promises performance advantages for simple data structures, a zero-copy approach with no serialization quickly turns complicated when less trivial structures with features like variable length messages are involved. Our research suggests that the need for dynamic memory allocation because of variable length messages hurts the ability to offer reliable latency bounds for some vendors. Additionally, laying out the data for complete zero-copy support would require extensive modifications on all levels of CycloneDDS code, an expensive endeavor that will likely outweigh the benefits of fewer data movement.

Depending on the deployment, security considerations can also be an important concern. With DDS user-space networking, there are two aspects to consider: Maintaining the security of the data being transmitted and ensuring that the additional privileges of the user-space networking technologies cannot compromise the operation of adjacent systems. Since DDS has built in support for encryption that is also supported by our CycloneDDS extensions, data security is supported out of the box through the DDS-security module. This makes the extensions just as secure as CycloneDDS itself in this regard. As an additional measure, industrial systems will often employ dedicated networks where physical access can be controlled. The matter of privileges is more difficult to secure though, as the user-space networking technology essentially has promiscuous access to the network card and is therefore able to read all traffic, even traffic that is unrelated to the application. Ensuring that the packet processing in the extensions cannot be hijacked, e.g. through a bug in the user application, would therefore be desirable in the future. For XDP, this would be possible by verifying the integrity of the packet filter when loading it into the kernel. These security lock-down mechanisms are subject to future research.

Finally, we observe that CycloneDDS-XDP has some performance penalties that occur due to the reading of single packets from the NIC, a problem likely also present in CycloneDDS-DPDK, but which appears to be negligible there. Reading packets one-by-one offers the best latency performance, which is the main metric we are interested in. However,

the more congestion starts occurring when the throughput is increased, the more the overhead of one-by-one processing starts becoming visible. The DDS-i layer is only capable of fetching a single packet from the extensions at a time, but theoretically the extension could fetch multiple packets from the NIC and buffer extraneous packets for later processing in CycloneDDS, thus reducing the communication with the NIC. Such a buffering mechanism would likely introduce some additional latency at low load due to larger transfer requests from the NIC as well as additional memory management, but especially for XDP a performance benefit for sample rate and bandwidth could likely be realized.

VII. CONCLUSION

In this paper, we introduced user-space networking technologies for use with the DDS message-oriented middleware standard. Analyzing previous studies, it was found that DPDK's user-space networking technology outperforms four DDS implementations using the traditional Linux networking stack in latency bounds, showing a potential for DDS performance benefits by leveraging a user-space networking stack. We implemented two extensions for the high performance DDS implementation CycloneDDS based on the popular user-space networking technologies DPDK and XDP. These extensions handle all Layer 2 Ethernet packet management within the CycloneDDS library and communicate directly with the NIC, circumventing the kernel on the communication hot-path. Using the CycloneDDS-DPDK and CycloneDDS-XDP extensions as accelerators, we demonstrated that both a significant reduction in mean latency is achievable (DPDK: 31%, XDP: 18%), that the packet delay variation can be more than halved and the maximum latency bounds can be reduced drastically for both extensions to under 250 μ s (\sim 95% improvement). Finally, in the case of DPDK the peak sample rate and bandwidth can be further increased by 59% and 48%, respectively. Using these CycloneDDS extensions, it is now possible to provide soft latency bounds of well under one millisecond on the DDS platform, which was not previously achievable on an industrial edge system. With the continuing effort to bring DDS to more timing-critical applications, we think that DDS with user-space networking technologies strikes a good balance between the offered performance and the necessary specialization on the spectrum of out-of-the-box messaging middleware over Linux IP to the use of dedicated microcontroller communication. We hope that the community benefits from our contributions to CycloneDDS and invite users to try out the CycloneDDS-DPDK and CycloneDDS-XDP extensions on their systems.

REFERENCES

- [1] P. Naur and B. Randell, "Software engineering: Report of a Conference Sponsored by the Nato Science Committee, Garmisch, Germany, 7th-11th October 1968," 1969.
- [2] Object Management Group. "The OMG Specifications Catalog." (Feb. 25, 2023), [Online]. Available: www.omg.org/spec/#categories/DDS/.

- [3] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the Performance of ROS2," in *Proceedings of the 13th International Conference on Embedded Software*, New York, NY, USA, 2016, ISBN: 9781450344852. [Online]. Available: <https://doi.org/10.1145/2968478.2968502>.
- [4] RTI Inc. "Operational Avionics Layer (OPAL) - IAI Aviation Group." (May 9, 2020), [Online]. Available: https://www.rti.com/hubfs/_Collateral/Custom_Snapshots/rti-customer-snapshot-iai-aviation.pdf (visited on 04/25/2023).
- [5] Object Computing, Inc. "Achieving Global Health with Leading-Edge Diagnostics." (Apr. 12, 2021), [Online]. Available: <https://objectcomputing.com/case-studies/a-healthier-future> (visited on 04/25/2023).
- [6] I. Calvo, F. Prez, I. Etxeberria-Agiriano, and O. G. de Albniz, "Designing High Performance Factory Automation Applications on Top of DDS," *International Journal of Advanced Robotic Systems*, vol. 10, no. 4, p. 205, 2013. eprint: <https://doi.org/10.5772/56341>. [Online]. Available: <https://doi.org/10.5772/56341>.
- [7] I. Calvo, F. Prez, O. G. de Albeniz, and I. Etxeberria-Agiriano, "Towards a OMG DDS communication backbone for factory automation applications," in *ETFA2011*, 2011, pp. 1–4.
- [8] C. S. V. Gutierrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications," *arXiv preprint arXiv:1808.10821*, 2018.
- [9] A. Belkhir, M. Pepin, M. Bly, and M. Dagenais, "Performance analysis of DPDK-based applications through tracing," *Journal of Parallel and Distributed Computing*, vol. 173, pp. 1–19, 2023, ISSN: 0743-7315. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0743731522002271>.
- [10] T. Hiland-Jrgensen *et al.*, "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66, ISBN: 9781450360807. [Online]. Available: <https://doi.org/10.1145/3281411.3281443>.
- [11] V. Bode, D. Buettner, T. Preklik, C. Trinitis, and M. Schulz, "Systematic Analysis of DDS Implementations," in *Proceedings of the 24th International Middleware Conference*, ser. Middleware '23, Bologna, Italy: Association for Computing Machinery, 2023, pp. 234–246, ISBN: 9798400701771. [Online]. Available: <https://doi.org/10.1145/3590140.3629118>.
- [12] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: Performance Evaluation of Industry 4.0 Protocols," in *2019 IEEE International Conference on Industrial Technology (ICIT)*, 2019, pp. 955–962.
- [13] Z. Kang and A. Dubey, "Evaluating DDS, MQTT, and ZeroMQ Under Different IoT Traffic Conditions," 2020.
- [14] X. Chen, X. Kong, Y. Ling, and X. Cao, "DDS Performance Evaluation for PREEMPT_RT Linux," in *2021 International Conference on Computer, Blockchain and Financial Development (CBFD)*, 2021, pp. 84–89.
- [15] S. Solpan and K. Kucuk, "DDS-XRCE Standard Performance Evaluation of Different Communication Scenarios in IoT Technologies," *EAI Endorsed Transactions on Internet of Things*, Nov. 2022. [Online]. Available: <https://www.doi.org/10.4108/eetiot.v8i4.2691>.
- [16] R. Serrano-Torres, M. Garca-Valls, and P. Basanta-Val, "Virtualizing DDS middleware: Performance challenges and measurements," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013.
- [17] P. Bellavista, A. Corradi, L. Foschini, and A. Pernaflini, "Data Distribution Service (DDS): A performance comparison of OpenSplice and RTI implementations," in *2013 IEEE Symposium on Computers and Communications (ISCC)*, 2013, pp. 377–383.
- [18] Object Management Group. "DDS Extensions for Time Sensitive Networking (DDS-TSN)." (Mar. 1, 2023), [Online]. Available: <https://www.omg.org/spec/DDS-TSN/1.0/Beta1/PDF> (visited on 04/25/2023).
- [19] E. Guijarro Cameros and L. Chan, "How dds and tsn are driving interoperability and performance in automotive systems," *ATZelectronics worldwide*, vol. 17, no. 10, pp. 52–55, 2022.
- [20] V. Bode, C. Trinitis, M. Schulz, D. Buettner, and T. Preklik, "DDS Implementations as Real-Time Middleware A Systematic Evaluation," in *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2023, pp. 186–195.
- [21] Petr Skocik. "System calls overhead." (Jan. 21, 2017), [Online]. Available: <https://stackoverflow.com/questions/23599074/system-calls-overhead> (visited on 04/25/2023).
- [22] DPDK. "DPDK Users's Guide." (Mar. 1, 2023), [Online]. Available: https://doc.dpdk.org/guides/linux_gsg/.
- [23] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi, "FloWatcher-DPDK: Lightweight Line-Rate Flow-Level Monitoring in Software," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 1143–1156, 2019.
- [24] M. Karlsson and B. Tpel, "The path to DPDK speeds for AF XDP," in *Linux Plumbers Conference*, 2018.
- [25] Object Computing, Inc., *OpenDDS Developer's Guide*, 2021. [Online]. Available: <https://download.objectcomputing.com/OpenDDS/previous-releases/OpenDDS-3.17.pdf> (visited on 06/21/2022).
- [26] XDP-Project. "Investigate if XDP can be used for TSN." (Jan. 15, 2019), [Online]. Available: <https://github.com/xdp-project/xdp-project/issues/2> (visited on 04/25/2023).