2024 Design, Automation & Test in Europe Conference (DATE 2024)
Special Initiative "Autonomous Systems Design"

Orchestration-aware optimization of ROS2 communication protocols

Mirco De Marchi†, and Nicola Bombieri*
*Dept. of Engineering for Innovation Medicine, University of Verona, Italy. mirco.demarchi@univr.it
†Dept. of Computer Science, University of Verona, Italy. nicola.bombieri@univr.it

***Abstract*—The robot operating system (ROS) standard has been extended with different communication mechanisms to address real-time and scalability requirements. On the other hand, containerization and orchestration platforms like Docker and Kubernetes are increasingly being adopted to strengthen platform-independent development and automatic software deployment. In this paper, we quantitatively analyze the impact of topology, containerization, and edge-cloud distribution of ROS nodes on the efficiency of the ROS2 communication protocols. We then present a framework that automatically binds the most efficient ROS protocol for each node-to-node communication by considering the architectural characteristics of both software and edge-cloud computing platforms. The framework is available at https://github.com/PARCO-LAB/ros4k.**

***Index Terms*—ROS2, Docker, Kubernetes.**

I. Introduction

The Robot Operating System (ROS) has been upgraded to ROS2 to meet real-time and scalability requirements. In ROS2, developers can leverage the *data distribution service* (DDS) and different mechanisms to implement efficient communication between ROS nodes [1]. In this context, there is a growing interest in combining ROS-compliant software development with *containerization* and *orchestration* [2]. Containerization allows for better resource utilization, platform-independent development, and secure deployment of SW [3]. Orchestration automates the deployment, networking, scaling, and availability of containerized workloads and services [4].

ROS is well-suited for utilization within edge-cloud systems, where ROS nodes are distributed across multiple computing devices. Utilizing the *standard-copy* (*SC*) communication protocol, each ROS node communicates by *publishing* or *subscribing* to one or more strongly typed, unidirectional messaging channels known as *topics* (see Fig. 1(a)). SC communication relies on standard networking sockets, allowing for the deployment and redeployment of containerized ROS nodes across edge-cloud devices without requiring manual code modifications.

On the other hand, the latency associated with this socket-based inter-process communication (IPC) method is notably high and escalates linearly as the message size increases. Consequently, it frequently results in significant performance deterioration when dealing with large data sizes, such as RGB-D camera and Lidar images [1].

Different DDS solutions have been proposed in the literature to improve communication efficiency based on SC [5], [6]. Although they allow multiple copies of data to be avoided on the network stack or even at the DDS level, they do not avoid data copies at the ROS node level.

This study was carried out within the PNRR research activities of the consortium iNEST (Interconnected North-Est Innovation Ecosystem) funded by the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022, ECS_00000043). This manuscript reflects only the Authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

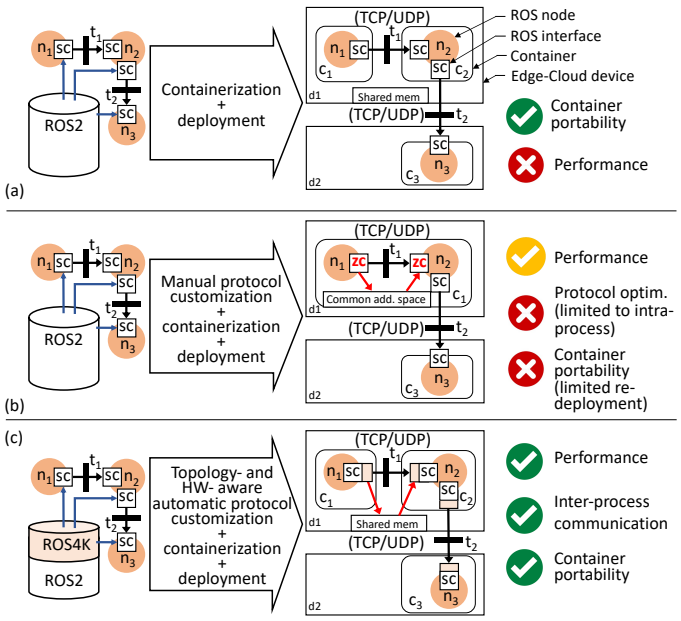

Fig. 1. Optimization vs. portability of ROS2 communication protocols on edge-cloud architectures through containerization and orchestration: (a) homogeneous standard-copy (SC) interfaces, (b) manually customized with zero-copy (ZC) interfaces, and (c) the proposed approach based on ROS4K.

The ROS *nodelet* package [7] was introduced to enhance the efficiency of node-to-node communication. It allows multiple ROS nodes to operate within a single operating system (OS) process and leverage a shared address space for achieving *zero-copy* (*ZC*) data transfer, even at the task level (Fig. 1(b)). Nevertheless, it cannot be employed to establish communication between nodes in different processes (i.e., *inter-process communication* - *IPC*). Consequently, this package cannot communicate between ROS nodes mapped into distinct containers.

For IPC, more recent solutions (e.g., TZC [8]) have been developed, which implement efficient communication by taking advantage of *shared memory*. These solutions can be implemented when ROS nodes are mapped onto the same device and the device supports shared memory at the application level.

In this scenario, selecting and manually implementing the most efficient node-to-node communication protocol for each ROS node, based on a particular deployment scheme within the edge-cloud cluster, becomes time-consuming and error-prone. Furthermore, any interface customization can restrict SW portability and lead to noticeable performance degradation when applied to other orchestration schemes. Given that orchestration platforms such as Kubernetes implement dynamic *re-distribution* of containerized SW, determining the optimal balance between SW portability and performance is an open

challenge.

In this paper, we address this challenge with the following contributions. We first present a performance analysis of ROS-compliant topic-based communication to show how factors like communication topology, containerization, and edge-cloud distribution of ROS nodes can impact the performance of the different communication protocols (i.e., SC, ZC, or communication based on shared memory - SHC). We then propose a framework that, based on such impacting factors, binds the most efficient protocol for each node-to-node communication (Fig. 1(c)). We finally present a library called *ROS4K* that extends the standard ROS2 primitives for the automatic binding of protocols to the ROS nodes at compile time (https://github.com/PARCO-LAB/ros4k). The library aims at customizing the communication protocols without modifications to the application code to preserve its portability. Since the libraries at the state of the art for inter-process communication (i.e., TZC [8] and LOTROS [9]) are available only for ROS1, we re-implemented such a communication protocol for ROS2.

We present the results obtained by applying the framework to optimize the SW implementing the mission of an industrial Robotnik RB-Kairos autonomous mobile robot.

The paper is organized as follows. Section II presents the related work. Section III presents the SHC zero-copy inter-process communication protocol, the analysis of the ROS2 protocols, and the framework based on ROS4K. Section IV presents the experimental results, while Section V is devoted to the conclusion.

II. Related work

Several research works have been done to evaluate the efficiency of the ROS2 communication protocols. In [1], the authors conducted a proof of concept for adopting the DDS in ROS communication. They evaluated the potential and limitations of the DDS-ROS2 combination. *RAPLET* [10] allows the latency of the publish/subscribe mechanisms to be measured in detail. In [11], efficient communication techniques exploit the shared-memory architecture of edge devices to improve performance in ROS-compliant CPU-GPU nodes.

In [12], the authors investigated the end-to-end latency of ROS2 for distributed systems and different DDS middlewares. They profiled the ROS2 stack and quantified the latency of each stack component in the adopted DDS.

In [5], the authors proposed a mechanism to allow multiple DDS implementations in a system and to bind one into each node-to-node communication dynamically. The mechanism can estimate the communication area, maximum data size, and quality of service control requirements for each communication channel to select the most suited DDS implementation.

Iceoryx [6] implements zero-copy at the DDS level and strongly improves SC-based communication. Nevertheless, since it relies on a master node (i.e., *RouDi*) to manage the message allocation and sharing, it cannot be adopted for *inter-container* communication.

All the previous works address the efficiency of DDS in ROS, disregarding the ROS protocol implementation (in particular, zero-copy) *at node-level*. It is important to note that since DDS is a middleware beneath each of the different ROS protocol implementations, all the other literature solutions for efficient DDS can be exploited in our framework.

The first node-level zero-copy for inter-process communication has been implemented in *TZC* [8]. The primitives split the messages into two parts in case the message is heavy or multidimensional. The first part is the data pointer and is exchanged through SC; the second is the payload and is exchanged through shared memory.

In [9], the authors proposed an evolution of such an inter-process communication called *LOTROS*. They integrate *smart pointers* and synchronization primitives into ROS1. These obey the same semantics and exhibit the same performance as their C++ standard library counterparts, making them preferable to other local IPC mechanisms. Nevertheless, such an interface cannot be adopted for orchestration-aware automatic customization due to two main limitations. It does not rely on the ROS stack (i.e., it implements a custom communication mechanism), and consequently, nodes communicating through smart pointers are hidden from the ROS topology. Also, it is built on top of ROS1 and is not compatible with ROS2 as it requires a centralized master node to manage communication.

In general, different from the work in the literature, this paper addresses the problem of portability versus efficiency of ROS-compliant software in distributed systems based on containerization and orchestration. It shows that different factors, including containerization and orchestration schemes, impact protocol efficiency in various ways. It presents a framework based on an extension of the ROS2 library to automate the protocol customization for each scheme.

III. Methods, library, and framework

We assume all ROS nodes composing the robotic software implement communication through the SC protocol. The assumption relies on the fact that SC is, by default, the key interface to connect processes over single or multiple devices via peer-to-peer connections. It is the de-facto standard communication protocol that guarantees code portability [13], [14]. In contrast, ZC communication allows for more efficient communication between nodes at the cost of portability. Since ZC is limited to intra-process communication, we adopt zero-copy communication at a task level, leveraging shared memory. We present an efficient and race-condition-free implementation of this inter-process communication (*SHC*) in Section III-A.

We show that the SW configuration (i.e., node communication topology and message size) strongly and differently impacts the efficiency of the three mechanisms: SC, ZC, and SHC (Section III-B).

Customizing ROS nodes from SC-based communication to ZC or SHC requires modifications to the node structure and initialization, publish and subscribe primitives, and message structure. The proposed approach aims to implement such a protocol customization automatically. To select the most efficient protocol for each node-to-node communication, we propose a framework (ROS- and Kubernetes-compliant) that considers the SW characteristics and the constraints given by the node orchestration across the edge-cloud platform (Section III-C). We finally provide an overview of *ROS4K*, which extends the ROS2 library to bind automatically, at compile time, each ROS node with the corresponding protocol (Section III-D).

A. ROS-compliant inter-process communication based on shared memory (SHC)

SHC takes advantage of the shared memory when the nodes are containerized and deployed on a shared-memory architecture. The goal is to provide safe access to the data by different

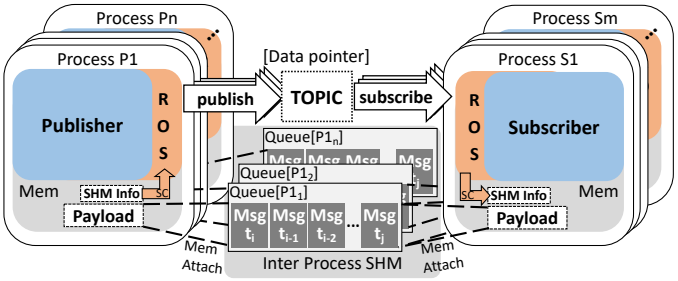

Fig. 2. Overview of SHC-based communication between n publishers and m subscribers.

(containerized) processes, such that the following conditions are satisfied: (a) each process thread has a consistent view of the data, (b) no data to be exchanged is corrupted, (c) no race conditions arise, (d) no deadlocks arise, and (e) process threads can perform the same or distinct operations independently on the same or different instances of the topic channels.

Fig. 2 shows an overview of SHC for communication among n publishers and m subscribers. With 1 producer and m consumers, the producer creates a new shared memory section and sends the data pointer to the consumer(s) through the whole stack of the ROS2 middleware by using the SC publish-subscribe paradigm (ROS reliable quality of service). This guarantees condition a. It shares the payload through the shared memory. The subscriber(s) receives the SC message with the shared memory information from the ROS stack queues and accesses the shared memory in read-only mode through a constant reference to the message payload to guarantee condition c. The publisher manages the release of the shared memory sections periodically for performance reasons. The period can be optimized by considering the publish frequency and the data size. The shared memory release is regulated through two counters per section, protected by a mutex lock. The first counts the start of a subscriber reading, and the second counts the end. The memory section is released only when the two counters are equal to guarantee condition b.

The publish and subscribe primitives are asynchronous and non-blocking to guarantee condition e. Sending the pointer and the publisher creating a new shared memory section for each new message guarantee condition d.

The same considerations hold in the case of n producers, as each producer creates and manages its private shared memory sections modularly and independently.

Since SHC relies on SC for transmitting shared-memory information and is built on top of the ROS2 communication stack, it enables customization of inter-process communication between containerized nodes using various quality of service (quality of service) levels, such as TCP or UDP, as well as different DDS implementations. To support inter-container communication, SHC has been implemented using Posix IPC.

B. SW configuration and protocol applicability/efficiency

ROS nodes are first containerized to be then distributed and orchestrated. We assume the nodes-to-containers and containers-to-devices schemes are given as starting points. ZC and SHC applicability strongly depends on them.

ZC can be adopted only when the ROS nodes are in the same process and, consequently, in the same container. They communicate through the address space shared between threads. In

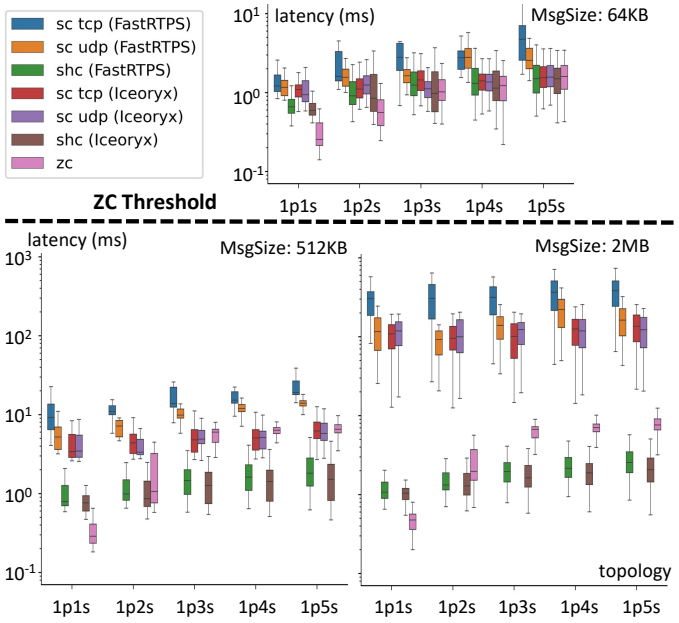

Fig. 3. Intra-process 1_publisher-n_subscriber(s) communication latency with FastRPS (default DDS in ROS2) and Iceoryx [6].

these configurations, ZC is the most efficient protocol for 1 publisher - 1 subscriber configuration due to the thread-level communication. With many subscribers, ZC implements additional copies in virtual memory space to avoid race conditions. This makes ZC less efficient than SHC, with two subscribers or more. As an example, Fig. 3 shows the benchmark results on an NVIDIA Jetson NX device. Our experimental results confirm that such behavior is independent of the computing architecture and capability. It is important to note that ZC skips the whole DDS and network communication stacks.

ZC is also the most efficient in any topology configuration (one or more subscribers) when the size of the data exchanged through the topic is small (message size $\leq ZC_{Threshold}$). This is because the additional thread-level copies are lighter than implementing a new standard copy of the data pointer and memory info across the ROS stack through SHC.

SHC extends the zero-copy paradigm for inter-process, intra-container, and inter-container communication on a device with shared memory available to the SW application. It increases the communication performance w.r.t. SC linearly with the increase of subscribers and the size of messages. On the other hand, for small messages ($\leq SC_{Threshold}$), the framework keeps SC, as SC and SHC performance is comparable with any n producer - m consumer configuration (see Fig. 4 as a representative profiling example of our benchmarks on an NVIDIA Jetson NX device). This is because both protocols rely on at least one IPC through the ROS stack. SHC starts achieving performance gain when the payload is no longer negligible.

Iceoryx achieves better performance w.r.t. FastRTP (i.e., the default DDS in ROS2) with SC and SHC as it implements zero copy at the DDS level. Nevertheless, it cannot be applied for inter-container communication as it relies on a centralized master (RouDi), which cannot be supported in Kubernetes-based distributed systems.

We also implemented a ROS2 version of the inter-process communication mechanism proposed in [8]. Since we measured

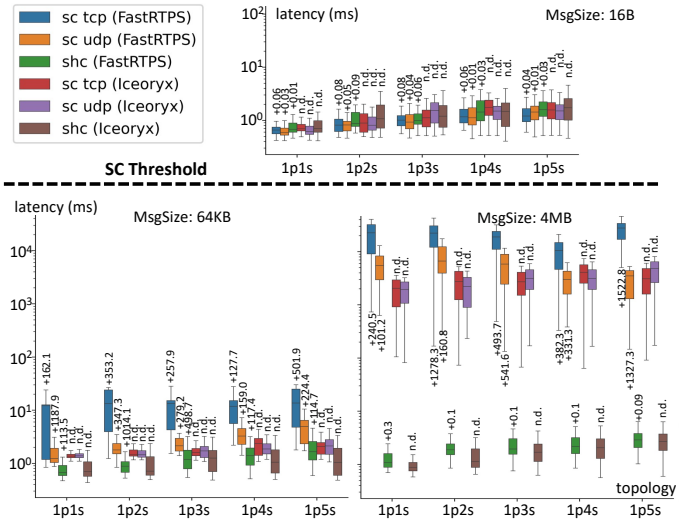

Fig. 4. Inter-process 1_publisher-n_subscriber(s) communication latency with inter-container overhead (n.d. if inter-container communication is not supported).

performance comparable to the SHC mechanism proposed in Section III-A, we included this last in the framework as we can guarantee safe conditions for data access.

The $ZC_{Threshold}$ and $SC_{Threshold}$ values depend on the device architecture and computing capability. We implemented a synthetic benchmark (in the ROS4K library) that profiles the latency of communication between nodes in any device with the three protocols over the message size to extrapolate such values, which will be used for the protocol identification in the binding phase.

As confirmed by our experimental results, containerizing ROS nodes introduces some latency overhead. However, this increase in latency is linear and remains negligible. The same constraints regarding protocol applicability and efficiency considerations apply when dealing with multiple publishers.

C. ROS node-protocol binding

We define N as the set of ROS nodes composing the SW application, which are containerized on a set C of containers, which in turn are mapped by an orchestrator (Kubernetes in our implementation) on a set D of devices (i.e., the edge-cloud system). T is the set of ROS topics on which the ROS nodes publish or subscribe. All nodes publishing or subscribing (*publishers* and *subscribers* in the following) to a topic implement the same communication protocol.

The *ROS Topology* is a quadruple $< N, T, Pub, Sub >$ where $Pub = \{(n,t) \mid n \in N, t \in T\}$ and $Sub = \{(t,n) \mid t \in T \wedge n \in N\}$. The couple (n,t) represents a publisher n to the topic t, while (t,n) represents a subscriber n to the topic t. To identify all publishers and subscribers on a topic, we define $tp : T \rightarrow N$, where $tp(t) = \{n \in N \mid (n,t) \in Pub \wedge (t,n) \in Sub\}$. In particular, $pubs(t) = \{n \in N \mid (n,t) \in Pub\}$ and $subs(t) = \{n \in N \mid (t,n) \in Sub\}$.

We define $ncmap : N \rightarrow C$ as the function that returns, for a node $n \in N$, the container $c \in C$ in which n has been organized by the developer. $NCMap = \{(n,c) \mid n \in N \wedge c = ncmap(n) \in C\}$. Similarly, $cdmap : C \rightarrow D$ returns the device $d \in D$ in which c has been mapped by the orchestrator. $CDMap = \{(c,d) \mid c \in C \wedge d = cdmap(c) \in D\}$.

Through the combination of $NCMap$ and $CDMap$, $ndmap : N \rightarrow D$ returns, for a node $n \in N$, the device $d \in D$ in which n has been mapped by the orchestrator. $NDMap = \{(n,d) \mid n \in N \wedge d = ndmap(n) \in D\}$.

For each topic $t \in T$, the $NPBinder$ procedure selects a communication protocol to be implemented in all nodes publishing or subscribing to t, as shown in equation 1.

$$NPBinder(t) = \begin{cases} \text{SC} & \text{if } \exists n_1, n_2 \in tp(t) \mid (ndmap(n_1) = d_1 \\ & \text{and } ndmap(n_2) = d_2 \; where \; d_1 \neq d_2) \\ & \text{or } (ncmap(n_1) \neq ncmap(n_2) \\ & \quad \text{and } size(t) \leq \text{SC}_{Threshold}) \\ \text{ZC} & \text{if } \forall n \in tp(t) \; \exists c \in C \mid ncmap(n) = c \\ & \text{and } (|subs(t)| = 1 \\ & \quad \text{or } size(t) \leq \text{ZC}_{Threshold}) \\ \text{SHC} & \text{otherwise} \end{cases} \tag{1}$$

If at least two ROS nodes, publishers, or subscribers, under the same topic t, are mapped to distributed devices (i.e., no shared memory available for node communication), *SC* is the only communication possible for all nodes of t. In the other cases, SC provides the same performance w.r.t. the possible alternative SHC when the nodes are grouped in different containers, and the message size is small.

If all nodes, either publishers or subscribers to t, are mapped to the same device and the same container, *ZC* is adopted either if the messages are small or if the message sizes are larger than the corresponding threshold, the topic has only one subscriber. In all other cases, the binder selects *SHC* for all nodes of t.

Our framework relies on the commands provided by the ROS middleware (e.g., `ros2 topic list|info`) to extrapolate the ROS topology information. It implements a Kubernetes plugin to monitor the $CDMap$ information after the orchestration phases (sorting/filtering/scoring/binding of cluster devices to SW tasks [15]) to substitute the original SC-based container with the customized one. The user defines $NCMap$. The result is a node-to-protocol map ($NPMap$ in Fig. 5), which is used for the protocol customization by ROS4K at compile time.

D. The ROS4K library

Fig. 5 shows the overview of *ROS4K*. It reimplements the ROS2 classes involved in the SC communication: node initialization, node-task binding, message initialization, and the publish procedure (for the publishers) and the subscribe procedure (for the subscriber). It extends the standard `rclcpp::Node` initialization class so that the node is flagged for intra-process communication in the case of ZC. The task is bound to such a node in the `rclcpp::spin` modality (i.e., infinite loop) in case of SC and SHC, while as `rclcpp::executors` (i.e., thread-based process) if ZC. For these two steps, ROS4K inherits from the original classes implemented in ROS2 for SC and ZC, and the protocol selector drives the selection.

The message initialization mainly consists of dynamic memory allocation. The allocation for the three protocols differs from the size of the allocated memory and the access modality. Consequently, ROS4K reimplements the original allocation adopted for SC (i.e., `make_shared`) as follows. It inherits the original class in case SC has been selected. It inherits the `make_unique` class in the case of ZC to control the race conditions of multithread accesses. It reimplements from

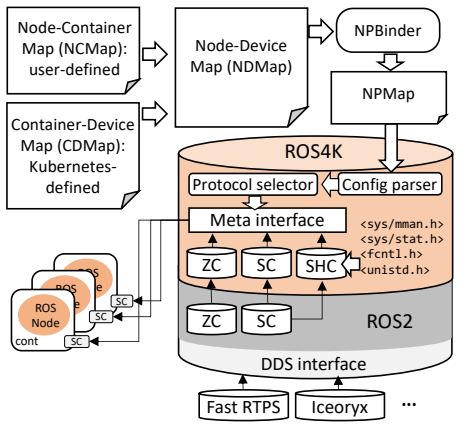

Fig. 5. Overview of ROS4K, the data flow to retrieve the node-device map from ROS and Kubernetes, and the node-to-protocol binding.

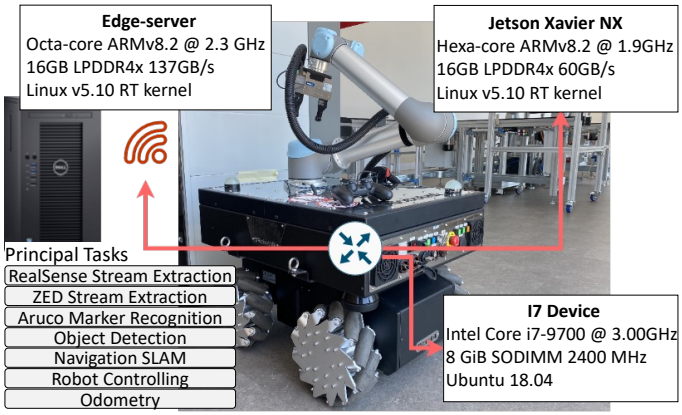

Fig. 6. Overview of the case study.

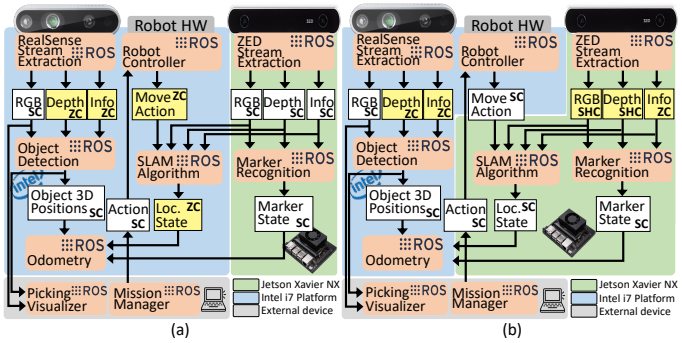

Fig. 7. Kubernetes orchestration schemes.

scratch the memory allocation through POSIX primitives in the case of SHC (Section III-A).

The `create_publisher` method inherits its behavior from the original SC procedures (e.g., `create_publisher(t)->publish(msg))` or ZC procedures (e.g., `create_publisher(t)->publish(move(msg))` in the case of SC or ZC, respectively. A similar approach is followed for the `create_subscription` method. In the case of SHC, this method is reimplemented to include sending the message pointer along with the shared memory information and conducting periodic control for shared memory release. This design enables every ROS node, originally implemented with the standard SC protocol, to be customized through re-compilation without requiring any modifications to the ROS node source code.

IV. Experimental results

We evaluated the proposed framework based on *ROS4K* to customize the software that implements a Robotnik RB-Kairos mission, a skid-steering mobile platform equipped with a Universal Robots UR5 and a Schunk WSG50 gripper. The gripper has an Intel RealSense camera that generates an RGB-D 640x480 stream for object recognition. The front of the robot base is equipped with a Stereolabs ZED camera that produces an RGB-D 720p stream for navigation tasks and Aruco marker recognition.

The software consists of 80 ROS nodes, which are distributed across three computing units of an edge-cloud cluster (see Fig. 6): two on-board programmable devices, i.e., an NVIDIA Jetson Xavier with Linux kernel $v.5.10$ and a desktop with an i7 9700 locked at 3.0 GHz with 8GB RAM, Linux OS. The on-board devices communicate through a Gigabit Ethernet switch (802.3ab). The third device consists of an off-board desktop with an octa-core CPU and 16GB of RAM, with Linux kernel $v.5.10$. It communicates with the other (on-board) devices through WiFi (802.11ac). The ROS nodes have been containerized through Docker (i.e., one container per ROS node) and are orchestrated through K3S Kubernetes.

For the sake of space, we report the results of the most critical ROS nodes (lower left corner of Fig. 6) and on two different orchestration schemes to evaluate Kubernetes' dynamic SW deployment capability. The communication volume of the other nodes is negligible, and the communication protocol selection does not significantly impact the overall system performance. First, we applied ROS4K to customize the SW code for the orchestration scheme depicted in Fig. 7(a). The figure underlines which topic and the corresponding publishers/subscribers have been identified for the protocol customization. The second main column of Table IV reports the original SC-based ROS nodes' performance (latency and frequency) and the corresponding node-to-node communication. The table reports the performance of each node in the device without interferences (*Performance* column) and the node's performance by considering the critical data path. Column *Real perf. (deployed)* reports the performance of the actual software deployed on the cluster. It includes the overhead due to resource contention (i.e., CPUs and memory access).

The third main column reports the performance of the software after customization with $ROS4K$. The performance of the reported key nodes running on the Jetson device does not change since no protocol can be customized. This is because all topics include at least one external (inter-device) ROS node in communication; consequently, SC is the only protocol to adopt. In the i7 desktop, the performance of the object detection and odometry nodes increases after deploying the optimized code (23 to 27 and 3 to 4 Hz, respectively). The improvement is due to the system-level optimization given by ZC in the different protocols, which leads to a lower workload in the shared resources (i.e., memory accesses). The performance increase of $Odometry$ is also due to the improvement of the $LocalizationState$ topic from $SLAM$ (3 to 4 Hz) and the consequent improvement in the corresponding critical path.

On the other hand, the system requires a minimum frequency of the robot controller node (i.e., 120Hz) to control the arm

TABLE I
EXPERIMENTAL RESULTS.

ROS node [Topic|Size|Direction]				Original SC-based communication Performance ms (Hz)	Original SC-based communication Performance (crit. path) ms (Hz)	Original SC-based communication Real perf. (deployed) ms (Hz)	ROS4K protocol optimization Performance (crit. path) ms (Hz)	ROS4K protocol optimization Real perf. (deployed) ms (Hz)		Original SC-based communication Performance ms (Hz)	Original SC-based communication Performance (crit. path) ms (Hz)	Original SC-based communication Real perf. (deployed) ms (Hz)	ROS4K protocol optimization Performance (crit. path) ms (Hz)	ROS4K protocol optimization Real perf. (deployed) ms (Hz)
ZED S. E.			Jetson NX	55.3 (18)	55.3 (18)	59.9 (17)	55.3 (18)	59.1 (17)	Jetson NX	55.3 (18)	55.3 (18)	**65.8 (15)**	55.3 (18)	**60.9 (16)**
RGB	2.6GB	to S.		269.1 (4)	269.1 (4)	278.4 (4)	269.1 (4)	272.6 (4)		89.7 (11)	89.7 (11)	96.8 (10)	55.3 (18)	60.9 (16)
		to M.R.		87.1 (11)	87.1 (11)	91.0 (11)	87.1 (11)	88.4 (11)		87.1 (11)	87.1 (11)	93.7 (11)	55.3 (18)	60.9 (16)
Depth	3.5GB	to S.		179.4 (6)	179.4 (6)	185.6 (5)	179.4 (6)	181.6 (6)		59.8 (16)	59.8 (16)	64.4 (16)	55.3 (18)	60.9 (16)
		to M.R.		61.0 (16)	61.0 (16)	63.9 (16)	61.0 (16)	61.7 (16)		61.0 (16)	61.0 (16)	64.8 (15)	55.3 (18)	60.9 (16)
Cam info	0.3KB	to S.		3.6 (278)	55.3 (18)	59.9 (17)	55.3 (18)	59.1 (17)		1.2 (826)	55.3 (18)	65.8 (15)	55.3 (18)	60.9 (16)
		to M.R.		1.2 (826)	55.3 (18)	59.9 (17)	55.3 (18)	59.1 (17)		1.2 (826)	55.3 (18)	65.8 (15)	55.3 (18)	60.9 (16)
Marker recognition (M.R.)				38.8 (26)	87.1 (11)	98.7 (10)	87.1 (11)	94.2 (11)		38.8 (26)	87.1 (11)	**106.5 (9)**	55.3 (18)	**64.7 (15)**
Marker state	0.1 KB	to O.		1.5 (666)	87.1 (11)	98.7 (10)	87.1 (11)	94.2 (11)		1.5 (666)	87.1 (11)	106.5 (9)	55.3 (18)	64.7 (15)
SLAM (S.)			i7 device	153.2 (7)	269.1 (4)	**296.4 (3)**	269.1 (4)	**260.5 (4)**		55.9 (18)	89.7 (11)	**107.8 (9)**	55.3 (18)	**64.5 (16)**
Localization state	0.3KB	to O.		0.7 (1,383)	269.1 (4)	**296.4 (3)**	269.1 (4)	**260.5 (4)**		4.2 (238)	89.7 (11)	107.8 (9)	55.3 (18)	64.5 (16)
Robot controller (R.C.)				7.4 (135)	7.4 (135)	**8.3 (121)**	7.4 (135)	**8.1 (123)**		7.4 (135)	7.4 (135)	7.8 (128)	7.4 (135)	7.7 (130)
Move action	0.1KB	to S.		0.5 (2,232)	7.4 (135)	8.3 (121)	6.9 (145)	8.1 (123)		1.2 (833)	7.4 (135)	7.8 (128)	7.4 (135)	**7.7 (130)**
RealSense S.E.				4.5 (222)	4.5 (222)	5.2 (192)	4.5 (222)	5.1 (196)		4.5 (222)	4.5 (222)	5.0 (198)	4.5 (222)	5.0 (199)
RGB	0.9GB	to O.D.		8.0 (124)	8.0 (124)	8.7 (115)	8.0 (124)	8.9 (112)		8.0 (124)	8.0 (124)	8.7 (116)	8.0 (124)	8.4 (119)
		to P.V.		24.1 (42)	24.1 (42)	26.3 (38)	24.1 (42)	26.9 (37)		24.1 (42)	24.1 (42)	26.2 (38)	24.1 (42)	25.2 (40)
Depth	0.6GB	to O.D.		3.9 (255)	4.5 (222)	5.2 (192)	4.5 (222)	5.1 (196)		3.9 (255)	4.5 (222)	5.0 (198)	4.5 (222)	5.0 (199)
Cam info	0.3KB	to O.D.		0.6 (1,544)	4.5 (222)	5.2 (192)	4.5 (222)	5.1 (196)		0.6 (1,544)	4.5 (222)	5.0 (198)	4.5 (222)	5.0 (199)
Object detection (O.D.)				29.8 (33)	29.8 (33)	**43.4 (23)**	29.8 (33)	**37.4 (27)**		29.8 (33)	29.8 (33)	36.5 (27)	**29.8 (33)**	**37.1 (27)**
Object position	0.1KB	to O.		0.8 (1,330)	29.8 (33)	43.4 (23)	29.8 (33)	37.4 (27)		0.8 (1,330)	29.8 (33)	36.5 (27)	29.8 (33)	37.1 (27)
		to P.V.		2.2 (454)	29.8 (33)	43.4 (23)	29.8 (33)	37.4 (27)		2.2 (454)	29.8 (33)	36.5 (27)	29.8 (33)	37.1 (27)
Odometry (O.)				66.4 (15)	269.1 (4)	**306.2 (3)**	269.1 (4)	**264.3 (4)**		66.4 (15)	89.7 (11)	112.3 (9)	**66.4 (15)**	**79.1 (13)**
Picking visualizer (P.V.)				13.0 (77)	29.8 (33)	37.8 (26)	29.8 (33)	37.9 (26)		13.0 (77)	29.8 (33)	32.2 (31)	29.8 (33)	31.2 (32)
Mission manager (M.M.)			Edge server	0.2 (5,181)	0.2 (5,181)	0.3 (3,949)	0.2 (5,181)	0.3 (3,940)	Edge server	0.2 (5,181)	0.2 (5,181)	0.2 (4,218)	0.2 (5,181)	0.2 (4,018)
Action	0.1 KB	to R.C.		0.8 (1,250)	0.8 (1,250)	1.0 (1,043)	0.8 (1,250)	1.0 (1,015)		0.8 (1,250)	0.8 (1,250)	1.0 (964)	0.8 (1,250)	1.0 (1,007)

hardware safely. A lower frequency communication is tolerated for short periods. Exceeding those periods leads the arm into safe mode, and the system halts operations. Even though the optimization through ROS4K leads to a slight increase in the controller performance (121 to 123 Hz), this orchestration scheme still leads the robotic system to the border of safe working conditions. To this end, a new distribution scheme has been explored (Fig. 7(b)), in which the workload has been lowered to decrease resource contention, which is the leading cause of the bottlenecks.

The original SC-based software with the new configuration results in a performance slowdown of some nodes in the Jetson (i.e., ZED stream extraction and marker recognition) due to the workload increase. In contrast, the SLAM takes advantage of the Jetson GPU and increases its performance. While the stream extraction slowdown is tolerated, the marker recognition, whose performance was at the boundary of the real-time frequency, gets even worse (from 11 to 9 Hz) than the original SC-based protocols. The performance sensibly increases with ROS4K thanks to the customization of the protocols related to the RGB, Depth, and Cam info topics, which satisfy the scheme's constraints to be customized into SHC/ZC. On the other hand, the performance of nodes on the i7 desktop sensibly increases after the SLAM migration. The odometry performance improves from 4 to 13 Hz in the new configuration. This is due to the SLAM migration and the optimization of the topics in its critical path.

V. CONCLUSION

This paper tackled the challenge of balancing portability and efficiency within ROS-compliant software when this software is containerized and orchestrated on distributed systems. We demonstrated that various factors, such as containerization and orchestration schemes, impact the efficiency of the ROS communication protocols in different ways. We presented a Kubernetes-compliant framework based on an extension of the ROS2 library to automate the protocol customization for each scheme. We evaluated the applicability and efficiency of the framework by optimizing the software to implement the mission of an industrial Robotnik RB-Kairos autonomous mobile robot.

REFERENCES

[1] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proc. of the 13th International Conference on Embedded Software, EMSOFT 2016*, 2016.

[2] J. Zhang, F. Keramat, X. Yu, D. M. Hern, J. P. Queralta, and T. Westerlund, "Distributed robotic systems in the edge-cloud continuum with ros 2: a review on novel architectures and technology readiness," 2022. [Online]. Available: https://arxiv.org/abs/2211.00985

[3] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 3566–3576, 2021.

[4] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *In Proc. of ACM/IEEE Symposium on Edge Computing*, 2018, pp. 373–377.

[5] R. Morita and K. Matsubara, "Dynamic binding a proper dds implementation for optimizing inter-node communication in ros2," in *In Proc. of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2019, pp. 246–247.

[6] K. Knese and M. Pohnl, "Iceroyx." [Online]. Available: http://https://iceoryx.io,https://github.com/eclipse-iceoryx/iceoryx

[7] T. Foote and B. R. R., "Nodelet package." [Online]. Available: http://wiki.ros.org/nodelet

[8] Y.-P. Wang, W. Tan, X.-Q. Hu, D. Manocha, and S.-M. Hu, "Tzc: Efficient inter-process communication for robotics middleware with partial serialization," in *Proc. of IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 7805–7812.

[9] C. Iordache, S. Fendyke, M. Jones, and R. Buckley, "Smart pointers and shared memory synchronisation for efficient inter-process communication in ros on an autonomous vehicle," in *Proc. of IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2021, pp. 6441–6448.

[10] K. Nishimura, T. Ishikawa, H. Sasaki, and S. Kato, "Raplet: Demystifying publish/subscribe latency for ros applications," in *Proc. of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2021, pp. 41–50.

[11] M. De Marchi, F. Lumpp, E. Martini, M. Boldo, S. Aldegheri, and N. Bombieri, "Efficient ros-compliant cpu-igpu communication on embedded platforms," *Journal of Low Power Electronics and Applications*, vol. 11, no. 2, 2021.

[12] T. Kronauer, J. Pohlmann, M. Matthe, T. Smejkal, and G. Fettweis, "Latency analysis of ros2 multi-node systems," in *In Proc. of IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, 2021.

[13] Z. Gao, T. Wanyama, I. Singh, A. Gadhrri, and R. Schmidt, "From industry 4.0 to robotics 4.0 - a conceptual framework for collaborative and intelligent robotic systems," in *Procedia Manufacturing*, vol. 46, 2020, pp. 591–599.

[14] V. DiLuoffo, W. Michalson, and B. Sunar, "Robot operating system 2: The need for a holistic security approach to robotic architectures," *Int. Journal of Advanced Robotic Systems*, vol. 15, no. 3, 2018.

[15] Deployments. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/