Enabling Containerized Computing and Orchestration of ROS-based Robotic SW Applications on Cloud-Server-Edge Architectures

***Abstract*—We present a toolchain based on Docker and KubeEdge that enables containerization and orchestration of ROS-based robotic SW applications on heterogeneous and hierarchical HW architectures. The toolchain allows for verification of functional and real-time constraints through HW-in-the-loop simulation, and for automatic mapping exploration of the SW across Cloud-Server-Edge architectures. We present the results obtained for the deployment of a real case of study composed by an ORB-SLAM application combined to local/global planners with obstacle avoidance for a mobile robot navigation.**

***Index Terms*—ROS, Docker, KubeEdge, Edge computing.**

I. Introduction and Motivations

While robots originated in large-scale mass manufacturing working very efficiently behind fences (i.e., not interacting with human operators), they are now spreading to more and more application areas. As these robots operate at human scale (e.g., physical Human-Robot-Interaction in factories, warehouses, and at homes) and perform satefy-critical tasks (like driving or surgery), the correctness requirements are stringent and include, beside functional accuracy, also non-functional constraints such as real-time, safety, data privacy, scalability, and energy efficiency [1].

To address all these requirements, robotic SW applications have to be configured to be run *across* heterogeneous Cloud-Server-Edge computing platforms [2]. Such a design and verification task is very challenging, as it requires high-expertise and time-consuming (re)configuration steps to compile and test the application code for different HW/SW configurations.

In this work, we present a toolchain that relies on a container-based packaging mechanism whereby SW components are abstracted from the environment in which they actually run and orchestrated across heterogeneous HW architectures. We extended the Docker and Kubernetes (KubeEdge) environments, which are well-established in the context of Cloud-native SW applications, to be used in the context of ROS-based robotic applications on Cloud-Edge platforms.

Recently, some open source and a number of proprietary model-based platforms have been proposed to ease the design of robotic systems (e.g., the EU funded Robmosys project, NVIDIA Isaac, Amazon AWS Robomaker). What is missing in these solutions is a toolchain able to address the complexity, the scalability, and the heterogeneity of the HW/SW domains by considering non-functional constraints. Taking into account in seamless way functional and nonfunctional constraints is a key feature to design reliable, and safe robotic applications from the specifications to the system deployment.

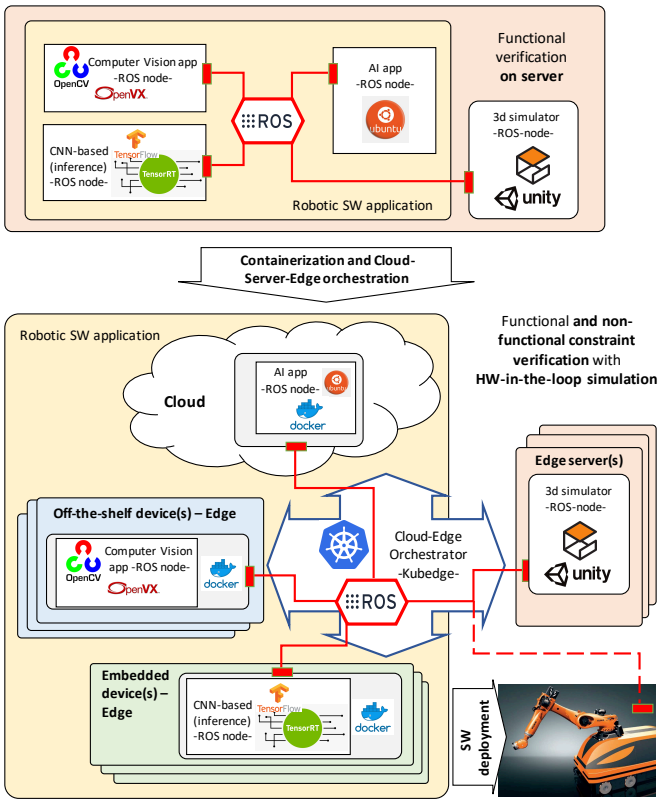

Fig. 1. Overview of the design flow

II. Design Flow

Programming today robots' missions and behaviour requires the integration of a multitude of *software components* from different domains like artificial intelligence, image recognition, sensing, machine learning, and reinforcement learning. We consider, as a starting point, each of these components as a ROS node and their communication based on the ROS standard (see the top part of Fig. 1). At the state of the art, such a ROS compatibility allows for design and functional verification of complex and heterogeneous robotic SW on server/desktop computing platforms by using 3D simulators (e.g., Gazebo, Unity).

Nevertheless, porting and testing these components as a system to the target hardware and in the target software stack is a necessary step before the deployment to verify non-functional constraints like real-time vs. accuracy, and this prompts the creation of realistic yet accessible robotic platforms. To avoid

useless iterations of such a time-consuming step during the SW-to-HW mapping exploration, the proposed toolchain first implements a containerization step of each SW component (i.e., ROS node) to abstract them from the target HW/SW environment. The containerization relies on two key concepts:

1) Containerization of applications for CPU/GPU architectures: Since image processing and inference-based components in robotic applications have to satisfy real-time constraints, we extended the containerization procedure based on Docker for edge computing in heterogeneous CPU/GPU devices. Starting from the image of standard reference containers, we implemented routines to solve the problem of the open link to drivers caused by *init hook* procedures [4]. These procedures are standard and mandatory for container initializations on GPU accelerated platforms.

2) ROS-based communication between containers: ROS nodes communicate through IP addresses and port numbers, where the IP corresponds to the device IP in the network and ports are assigned randomly. This allows to easily implements communication and synchronization of ROS nodes. Such nodes may also be distributed on different network devices. In contrast, containers of Cloud-native applications are associated to private (isolated) subnet IPs [3]. To tackle the communication issue among many containerized ROS nodes, the proposed containerization solution implements the *non-isolation of containers* to allow for communication between containerized ROS nodes distributed across different devices.

The toolchain relies on Kubernetes and KubeEdge for the mapping and orchestration of the generated containers on the actual target devices of the HW infrastructures. This allows us to verify non functional constraints of the SW application through HW-in-the-loop simulation, where the ROS-compliant 3D simulator runs on an edge server. The system supports containers running on Cloud data centers, edge servers, off-the-shelf edge devices (e.g., NVIDIA Jetson), and embedded edge devices (e.g., robot native boards).

The proposed containerization allows us to easily implement the space exploration of the mapping between SW components and HW devices, and the immediate deployment of the solution that satisfies all the constraints to the actual target robot.

III. Experimental results

For the sake of space, we report the most meaningful results we obtained with the proposed toolchain (in particular, containerization and automatic mapping exploration) to configure a SW application for a mobile robot (Robotnik Kairos) autonomous navigation. The SW implements an ORB-SLAM [5] application for the simultaneous localization and mapping through RGB cameras combined to an inference-based image recognition system [6] that controls a *move base* system based on a global and a local planner with obstacle avoidance. After a first step of functional verification, we performed the validation through HW-in-the-loop simulation by considering real-time constraints of some parts of the SW system. In particular, we mapped the global planner to an edge server as non-critical task. We considered the ORB-SLAM and local planner executing in real time at the edge and by setting a constraint of 16 FPS as minimum supported rate for the 20Hz RGB camera input stream. We first measured the performance of their containerized implementation running singularly on an NVIDIA Jetson TX2 (rows ORB(J1) and LP(J1) in Table I).

TABLE I
Real-time performance measurement with manual and KubeEdge-based automatic SW-HW mapping

Setup	Odom (ms)	Track pipe(ms)	Decod (ms)	Frame elab.(ms)	Feature ext.(ms)	Supp. (FPS)
KubeEdge-based automatic setting						
ORB(J1)	-	54.82	16.47	49.40	44.35	**18.94**
LP(J1)	1.21	-	-	-	-	-
ORB+LP(J1)	2.85	93.15	23.96	67.99	60.41	**10.74**
ORB(J1);LP(J2)	1.26	58.14	13.52	42.34	37.17	**17.20**
Manual setting						
ORB(J1)	-	50.20	16.20	42.16	34.82	**19.92**
LP(J1)	1.70	-	-	-	-	-
ORB+LP(J1)	1.43	70.54	15.02	55.60	50.24	**13.18**
ORB(J1);LP(J2)	1.60	56.20	14.00	49.32	43.95	**17.79**

Table I reports the performance in ms of the local planner (Odom) and the different stages of the ORB-SLAM (frame decoding, elaboration, feature extraction and their pipeline). The last column reports the FPS supported by the system.

We set the automatic mapping exploration through KubeEdge by testing the mapping of the two containers on the same Jetson device and on two Jetson devices (ORB+LP(J1) and ORB(J1);LP(J2) respectively in the table). In both cases, communication and synchronization relies on ROS, although physically implemented on shared memory in the first case, while through Ethernet in the second case. The results show that, even though the performance of both tasks largely satisfy the constraint when running singularly on the device, the suffer from resource (CPU) contention and communication overhead when running concurrently. In the second case (ORB(J1);LP(J2)), the results show that the system satisfies the real-time constraint although the communication overhead involved by the Ethernet communication.

We finally implemented manually all the customization and distribution of the two applications, to measure the overhead involved by containerization (last rows of the table). The results show that the manual customization of the two tasks running concurrently in the same edge device is more efficient than the corresponding containerized version but it does not satisfy the real time constraint for similar reasons. This requires a manual re-customization of the system with distributed execution of the two tasks. Finally, we found that, in general, the performance overhead due to containerization is within 5% w.r.t. the corresponding non-containerized implementation, while the memory footprint overhead is around 66MB for each container.

References

[1] H. Abbas, I. Saha, Y. Shoukry, R. Ehlers, G. Fainekos, R. Gupta, R. Majumdar, and D. Ulus. Special session: Embedded software for robotics: Challenges and future directions. In *Proc. of ACM/IEEE International Conference on Embedded Software, EMSOFT*, 2018.

[2] J. Chen and X. Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 2019.

[3] Docker. Configure networking. https://docs.docker.com/network, 2020.

[4] Kubernetes. Concepts. https://kubernetes.io/docs/concepts/workloads/pods/init-containers, 2020.

[5] R. Mur-Artal and J. D. Tardós. ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[6] NVIDIA. Deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson. https://github.com/dusty-nv/jetson-inference.