Cube Bot — A Smart Factory Showcase for the Real-Time Container Architecture

1st Joseph Hirsch*
joseph.hirsch@outlook.de
2nd Marius Lichtblau*
marius@lichtblau.io
3rd Marian Lingsch Rosenfeld*
M.Rosenfeld@campus.lmu.de
University Augsburg
Augsburg, Germany

4th Kilian Telschig
Siemens Technology
Munich, Germany
kilian.telschig@siemens.com

5th Alexander Knapp
University Augsburg
Augsburg, Germany
knapp@informatik.uni-augsburg.de

*Authors contributed equally to this work

***Abstract*—Dynamic reconfiguration is one of the key challenges in adaptive systems. As many tasks of adaptive systems are carried out by software and flexible machines, adaptation can be mainly defined by reconfiguration. The Real-Time Container Architecture provides a novel framework for these updates by running distributed embedded software in containers and enabling real-time reconfiguration of these software components following a reconfiguration plan. Yet, it limits I/O access of the containerized software components to GPIO, besides UDP-based communication. We extend the platform with new I/O methods and evaluated the extension with the smart factory showcase system *Cube Bot*. The extensions enable access to a camera and a robot from within the containers in compliance to the platform concepts. Within the application containers, different languages, state-of-the-art AI technology and the serial interface of the robot can be used. This will show the usefulness of the Real-Time Container Architecture in an even broader industrial context and its capability for extension to other use cases.**

***Index Terms*—Dynamic reconfiguration, adaptive systems, RTCA, framework, real-time, I/O, container**

I. Introduction

With the ongoing digitalization in industry, a key selling point for machines is their software and the ability to perform updates. Especially in production, which is experiencing a shift to smaller lot sizes due to a larger degree of customization [1], updates are crucial and need to succeed even between two products. Since downtime of a production line has to be avoided due to large costs, these updates inherit hard real-time constraints from the applications running industrial machines. With the increasing number of updates, the development and testing of the applications cannot be done on the real system and a containerized architecture is a natural fit for those scenarios. Telschig et al. in [2] propose a Real-Time Container Architecture (RTCA) to address these requirements for industrial systems, allowing distributed applications with hard real-time constraints to run in a containerized architecture. The RTCA also supports reconfiguration without shutting down the distributed control system, e.g. updates of the containers [3]. However, as only GPIO was supported within the containers, the RTCA could not be used and evaluated with industrial equipment that uses more complex I/O technologies.

In this work we evaluate the applicability of the RTCA with a smart factory showcase called *Cube Bot*. The system consists of a two-node setup where one node is performing object classification, e.g. at a conveyor belt, and the other node controls a robot, which is sorting items based on the detection result. We extended the RTCA to support serial and visual I/O and developed application components for the efficient object classification and the robot control. Both components use different programming languages and their own libraries and therefore use the benefits the containerized architecture provides. For the nodes, we use two SIMATIC IOT2040 controllers, which are Siemens IoT devices with a single-core CPU clocking at $400\,\mathrm{MHz}$ and $1\,\mathrm{GB}$ of main memory. One of the nodes is connected to a basic USB webcam and the other node is connected to a Lynxmotion robotic arm via RS-232. Both nodes are interconnected on a wired network. We classify two different objects: A red and a green cube. The approach is applicable to other kinds of products, though, since it uses an individually trained image classifier. The conveyor belt is left out for simplicity. Figure 1 shows the experimental setup. In order to test the real-time software update functionality, we also implemented a third software component that is deployed on the detection node and uses user input from a button in order to determine the class of the object. We then update the node to use the object detection component, instead, where the update has to succeed in real-time. The remainder of this work is structured as follows: We first review related work on reconfigurable distributed embedded applications. Next, the RTCA is briefly introduced and the two I/O extensions are described. After that we cover some details about the implementations of the two software components. We conclude this paper with a discussion about our and future work.

II. Related Work

Dynamic software reconfiguration has multiple applications in industrial settings. This is especially important for systems composed of distributed IoT devices, since such architectures

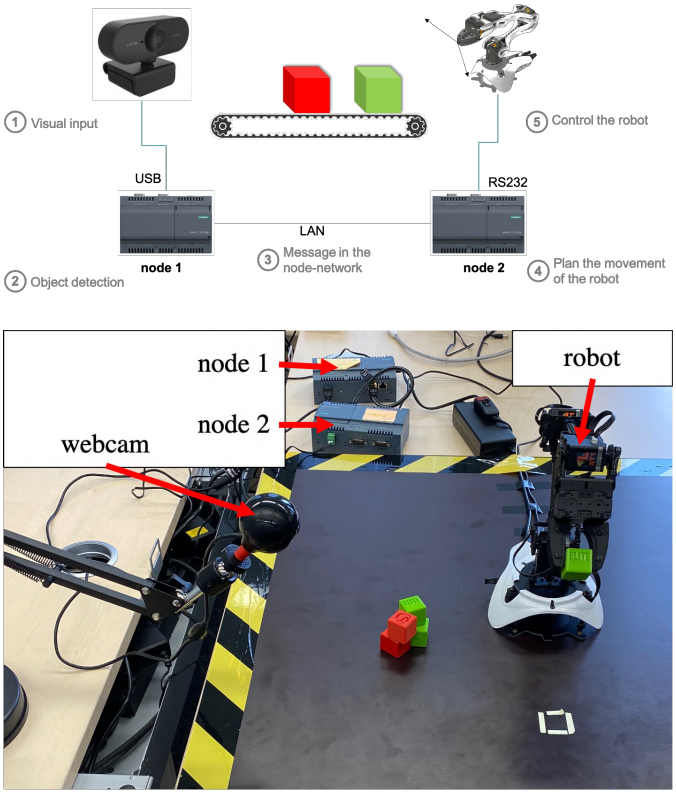

Fig. 1. Showcase setup

might be used in a smart factory setting to control the systems, while it is configured at a higher abstraction level [4].

Multiple approaches for dynamic reconfiguration of distributed systems exist, an overview is provided by Hammer [5]. One of the first was proposed by Kramer and Magee [6] in 1990, which, however, assumes that the system being reconfigured is completely inactive temporarily. Multiple improvements, based on this work have been proposed, most notably [7]. In general, proposals to achieve dynamic configuration consider different abstraction levels and scenarios, for example [8] approaches the problem from an architecture perspective, while [9] considers the programming languages used and their execution environment.

We aim at dynamic reconfiguration of distributed IoT systems with real-time constraints and state transfer between the old and new components. This case requires non-blocking reconfiguration, since the system must remain reactive at all times. Sha et al. [10] considered this problem in 1996 for the first time and proposed a reconfiguration based on atomic switchovers. More recent approaches include [4], who showed that the automation standard for distributed control systems IEC 61499 [11] enables dynamic reconfiguration, and [12], who used a similar approach to enable dynamic configuration for HTL [13]. These approaches do not consider distributed dependencies between the different nodes. This is considered in an event-based approach by Prenzel and Steinhorst [14] and by the time-triggered RTCA framework [2], which we extend

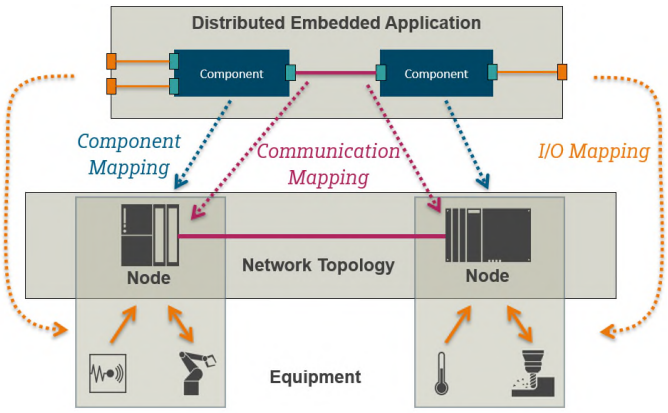

Fig. 2. Distributed control systems in the RTCA: Application parts are dynamically mapped to the computing infrastructure (figure based on [2]).

and use to study the *Cube Bot* industrial application use case.

III. RTCA Overview

The RTCA is a runtime platform concept for distributed embedded applications with hard end-to-end real-time requirements. It enables component-based software development to the extent that the set of components can be changed during full operation, which enables dynamic evolution of running distributed embedded applications. Figure 2 shows how distributed embedded applications are running on a networked set of computing nodes via three kinds of mappings. A component mapping leads to execution of a component's cyclic task within its lxc container on a specific node. Connector mappings along the communication chain orchestrate the cyclic, UDP-based inter-component communication. I/O mappings enable access by components to equipment installed at the local node. These mappings are dynamically managed by a real-time container agent (agent) on each node. Besides dynamic orchestration, the primary aim of the agent is to enforce the Logical Execution Time (LET) paradigm [15], so that all task executions, message transmissions, and I/O handling are logically done in isochronous periods across nodes and network connections in the system. The LET paradigm leads to deterministic end-to-end real-time behavior while temporally decoupling the software components. It avoids scheduling-dependent behavior and does not require configuration of priorities and time slices. This is also the basis for deterministically reconfiguring distributed embedded applications without downtime, especially in case of breaking changes which require temporally coordinated reconfigurations across nodes. Figure 3 shows an overview of the runtime architecture on one node. All software components run their tasks within a dedicated and isolated lxc container. Without active engagement by the agent, no information should be exchanged with the outside of the container. For network messages, this is ensured by a special *barrier* queueing discipline installed at the virtual network interfaces. For GPIO-based inputs and outputs, a fake *sysfs* is provided within the containers. The agent has a cyclic task which runs at highest priority while all local tasks and

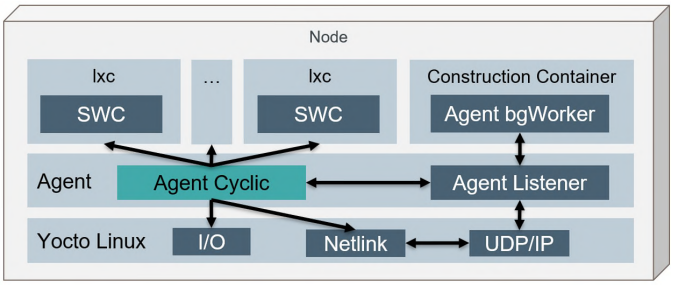

Fig. 3. RTCA runtime architecture: an agent dynamically orchestrates the containers and manages the timing of their interactions with each other and with the equipment based on the mappings. The mappings can be changed during full operation using reconfiguration plans, part of which are executed by a background worker component of the agent (figure based on [16]).

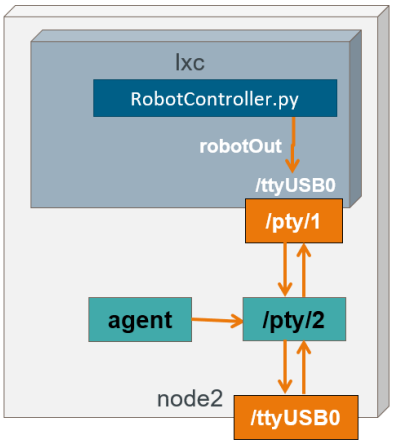

Fig. 4. Serial I/O: One of the pseudo terminals is mounted inside the container at the same location as the device on the host. The agent controls the other pseudo terminal and the real device in the cycle turnover.

messages are waiting for the next cycle. In each cycle the agent checks the health of all tasks. Then it processes the mapped outputs of the containers. In case of GPIOs the agent copies contents between the faked and the real *sysfs*. For UDP messages the agent sends command messages to the *barrier*. After output processing, the agent processes the mapped inputs accordingly. Finally, the agent triggers the tasks of mapped components and waits for the next cycle. Consequently, the components are computing outputs from the inputs during the remaining cycle without further interaction with the outside of their containers. If a reconfiguration plan is provided, the agent additionally performs applicable reconfiguration steps in three hooks either before output processing, between output and input processing, and after input processing. The reconfiguration steps are either for container lifecycle management, inter-node coordination, or actual time-critical modification of the the running distributed embedded application. The modification steps can change the mappings (add/remove a component, a communication mapping, an I/O mapping) and perform state transfer operations (based on file processing and packet capturing). Coordination steps ensure that modification steps with distributed breaking changes are executed synchronously, i.e., in pre-determined cycle offsets relative to an agreed synchronization point. This leads to temporal determinism of the reconfiguration, so that it can be done consistently without or with reduced and predictable downtime. The same mechanism is also used to consistently start a distributed embedded application. For more details on the RTCA we refer to the previous publications [2], [3], [16].

IV. RTCA Extensions

We extended the RTCA so that additional I/O types are supported. Previously, only GPIO inputs and outputs could be used by application components within containers. We added support for USB camera input and bidirectional serial communication as follows. The main issue is to enable this in compliance with the platform concepts described before, especially the LET paradigm, i.e., logically freezing I/O during the cycle.

A. Serial I/O

Communication via serial protocols is especially important in industrial settings as it is widely used to communicate with different kinds of hardware. RS-232 is an example of a standard, that is used by a variety of devices, such as the robotic arm used in our demo application. Serial I/O differs from the existing RTCA extension as it has to provide bidirectional communication via a single mapped port. The extension should provide all input and handle all output at cycle turnover. One additional challenge is, that many devices provide software libraries that expect the devices to be mounted on a certain serial port by the UNIX operating system. To be able to use these libraries the serial RTCA extension needs to mount the host port in the correct place in the container.

We use pseudo terminals (pty) [17] to fulfill these requirements. This Linux feature provides a pair of connected virtual terminals, to enable two processes to communicate. In our solution, one terminal is mounted inside the container and can be accessed by the application. Another terminal outside of container is used by the agent. At cycle turnover the agent reads the new data from his pseudo terminal and writes it to the host terminal. Afterwards it reads from the host terminal and writes to the pseudo terminal to make the new data available to the application. This ensures that the agent stays in control over when the data passes the boundaries of the container. Figure 4 shows how this works on an example node. The physical device is mounted at */ttyUSB0* of the host file system. When the agent should add the serial I/O according to its reconfiguration plan, it creates the pair of pseudo terminals, */pty/1* and */pty/2*. The container end is mounted at */ttyUSB0* within the root file system of the lxc container in which the software component is running. This ensures that the software component does not need to know that it communicates with a pty instead of the real device, so libraries continue to work. It is the responsibility of the application to ensure that there is no active transmission at the end of the cycle.

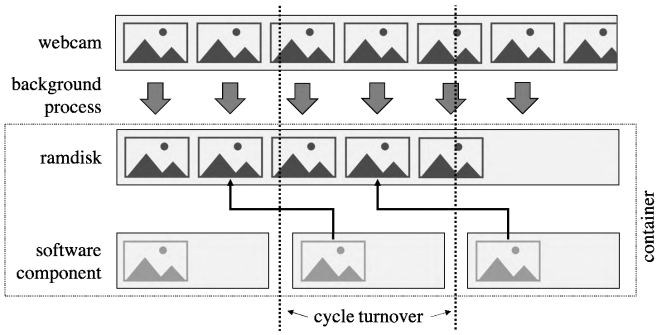

Fig. 5. Camera Input: The background process periodically stores frames on a RAM disk that is mounted in the container. On cycle turnover the agent generates a link to the current frame for the software component.

B. Visual I/O

To enable the input from a camera into a container application we extended the RTCA for this I/O method as well. A problem with the camera input is the long time that it takes to store one frame in memory. On the SIMATIC IOT2040 it takes about 10–20 ms. Most of this time is spent on waiting for the camera while the CPU is on idle. However, this time is often still not feasible for an execution between two cycles, since this period should be as short as possible. On the other hand, the design of the RTCA requires the software component to have all of its input available at the beginning of the cycle, which includes a full frame when considering video input. But different to the GPIO and the pseudo terminal I/O even a copy operation takes too long and can not be performed on cycle turnover. We decided to add a background process to the RTCA, which has a quota limit of a very small fraction of the available time during a period (e.g. 3 %). This process periodically requests a new frame from the camera, waits for it to be in memory and then stores this frame to a RAM disk which is mounted to the container as well. A file rotation mechanism is used to store the last couple of frames. The number of files has to be selected in a way that an overwriting of a frame in the same cycle is excluded, e.g. for a cycle frequency of 2 Hz and a camera frame rate of 30 fps, the number of frames to keep must be 15 at the very least. At the beginning of a cycle, the agent creates a link to the current frame for the software component, which results in very little overhead between two cycles. See Figure 5 for a visual representation of this process.

V. Object Classification Component

In the *Cube Bot* system, a containerized software component for object classification is running on the node that is connected with the camera. In each cycle it analyzes the last full frame captured by the camera to detect red or green cubes. This frame is made available as a JPEG-encoded image file that was stored on the RAM disk as described in Section IV-B. To access the file, the software component uses the link that the agent updates in each cycle turnover.

input shape	layer	output shape
$(60, 80, 3)$	conv. $(16, 5, 2)$	$(28, 38, 16)$
$(28, 38, 16)$	max pooling	$(14, 19, 16)$
$(14, 19, 16)$	conv. $(16, 5, 1)$	$(10, 15, 16)$
$(10, 15, 16)$	max pooling	$(5, 7, 16)$
$(5, 7, 16)$	flatten	(560)
(560)	fully connected	(128)
(128)	fully connected	(3)

Fig. 6. Layers of the neural network used in the object classification component. For convolutional layers (conv.(a, b, c)), a is the number of filters, b is the kernel size and c is the stride.

For the classification of items on the conveyor belt we trained a convolutional neural network for image classification. The input shape of the images is set to 80x60 pixels in RGB color layers. The layer architecture of the network is described in Figure 6. The training data for our simple use cases consisted of 900 labeled images, 300 in each of the classes red, $green$ and no_cube. We trained the model with the Tensorflow Python library using Jupyter notebooks. In order to address overfitting, we used a combination of input randomization (random flip and random rotation for the training images), batch normalization between the two convolutional layers as well as a dropout layer between the convolutional layers and the fully connected layers. We achieved a test accuracy of 100 % for 12 random images of each class.

Due to the very limited hardware resources on the device and the real-time requirements of the use case, we decided to quantize the model to 8-bit integer arithmetic after the training. This results in a significant speedup in inference time especially on devices without FPU or GPU such as the SIMATIC IOT2040. The inference was executed using the Tensorflow for microcontrollers library for C++ which is a fast and lightweight option to use Tensorflow models. All these measures applied we ended up with an inference time of below 300 ms for a single frame which is considered practicable for our use case.

VI. Robot Component

In order to move the cubes into their required position, we used a Lynxmotion 4DOF robotic arm, which can be controlled through the RS-232 protocol. To control the robotic arm, we developed a robot component based on the Lynxmotion LSS Library[1] for Python. Depending on input from the other software component, the containerized robot component sends commands to the robot to achieve the following movement pattern, starting from an established position:

- Open gripper, move arm down, close gripper in order to pick up the cube, move arm up
- Turn left or right depending on the instruction received from the camera or button component
- Move arm down, open gripper, move arm up, return arm to initial position

[1] https://github.com/Lynxmotion/LSS_Library_Python

The RTCA framework with the serial I/O extension presented in section IV allowed for a seamless communication with the robotic arm by the containerized robot controller component. This was achieved by using the Serial I/O extension in order to have a */ttyUSB0* device inside the container with which the robot component could communicate. The agent then took the I/O in the cycle turnover from the paired pseudo terminal and read/wrote them into the corresponding device. The RTCA framework provides dynamic reconfiguration, but allows I/O operations only before/after each cycle. In order to meet this requirement the robot component used the state pattern [18], which contained the states *Init*, *UDP*, *Wait*, *Move*. This allowed the robot component to wait until the robotic arm was at the expected position before sending the next command or receive the information from the camera or button components via UDP. Since the information was received via UDP continuously from the camera or button component, no reconfiguration was needed for this component in the showcase setup. In order to speed up the development process, a robot simulation was used. It used pseudo terminals for I/O, one of which was connected to *USB0*, where the robot is expected to be according to the RTCA configuration. The RTCA framework did not require any reconfiguration in order for this to work, i.e., it could not differentiate whether it was talking to the simulation or the robot.

VII. Discussion and Future Work

We demonstrated the feasibility of applying the RTCA to a smart factory use case by implementing *Cube Bot* as an example taken from an industrial context. The *Cube Bot* system can be seen as a simple version of an adaptable intelligent pick and place unit that sorts workpieces for further processing (e.g. reject workpieces). Thus, this use case demostrates applicability of RTCA to a wide range of systems. We extended the existing RTCA architecture by two new communication methods, the serial I/O as well as camera input, which enable even more applications by providing a common communication interface with physical devices and allowing for real-time camera input as a basis for machine learning algorithms. These features were implemented using Linux native facilities without violating the deterministic real-time requirements, which are fundamental for the RTCA. Our showcase demonstrates that the architecture is flexible and fast enough to support the two new categories of applications we implemented for our showcase. The machine learning model is able to process the webcam image and make a decision based on the color of the cubes. The result is used by the robot component to instruct the robotic arm to correctly sort the cubes. Thus, the RTCA concepts can be applied to anticipated smart factory technologies, including webcam input, machine learning, and serial robot I/O. In particular, this makes the RTCA applicable to cyber-physical systems that act in a more open decision space based on software-defined equipment. In the future we want to implement more sophisticated machine learning models to test the limits of the hardware and software architecture. It is also possible to extend the serial interface to communicate via more protocols beyond RS-232 to allow for the integration of more industrial hardware. For better observability of the internal system behavior both on application and platform level – especially of the reconfiguration progress – we are working on integrated monitoring features. We also want to improve on the dependability of the inter-component communication and enable usage of industrial communication standards and technologies in compliance with the RTCA concepts. Further extensions like these will make the RTCA even more suitable for industrial smart factory systems.

References

[1] Y. Koren, *The global manufacturing revolution: product-process-business integration and reconfigurable systems*. John Wiley & Sons, 2010.

[2] K. Telschig, A. Schönberger, and A. Knapp, "A real-time container architecture for dependable distributed embedded applications," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2018, pp. 1367–1374.

[3] K. Telschig and A. Knapp, "Synchronous reconfiguration of distributed embedded applications during operation," in *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2019, pp. 121–130.

[4] A. Zoitl, W. Lepuschitz, M. Merdan, and M. Vallée, "A real-time reconfiguration infrastructure for distributed embedded control systems," in *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. IEEE, 2010, pp. 1–8.

[5] M. Hammer, "How to touch a running system: Reconfiguration of stateful components," Ph.D. dissertation, München, Univ., Diss., 2009, 2009.

[6] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on software engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[7] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, 2007.

[8] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th international conference on Software engineering*. IEEE, 1998, pp. 177–186.

[9] J. Kramer and J. Magee, "Dynamic configuration for distributed systems," *IEEE Transactions on Software Engineering*, no. 4, pp. 424–436, 1985.

[10] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1. IEEE, 1996, pp. 335–346.

[11] *IEC 61499-1:2012. Function blocks - Part 1: Architecture*, International Electrotechnical Commission Std., 2012.

[12] C. M. Kirsch, L. Lopes, E. R. Marques, and A. Sokolova, "Runtime Programming through Model-Preserving, Scalable Runtime Patches," in *International Workshop on Formal Aspects of Component Software*. Springer, 2010, pp. 290–294.

[13] T. A. Henzinger, C. M. Kirsch, E. R. Marques, and A. Sokolova, "Distributed, modular HTL," in *2009 30th IEEE Real-Time Systems Symposium*. IEEE, 2009, pp. 171–180.

[14] L. Prenzel and S. Steinhorst, "Automated dependency resolution for dynamic reconfiguration of iec 61499," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2021, pp. 1–8.

[15] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[16] K. Telschig and A. Knapp, "Time-critical state transfer during operation of distributed embedded applications," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1. IEEE, 2019, pp. 516–523.

[17] pty - pseudoterminal interfaces. [Online]. Available: https://man7.org/linux/man-pages/man7/pty.7.html

[18] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides *et al.*, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.