Experiences in Integrating Internet of Things and Cloud Services with the Robot Operating System

Stamatis Karnouskos, Nadine Gaertner, Nemrude Verzano, Frank Beck, Andre Becker, Santo Bianchino, Daniel Kuntze, Miguel Perez, Rupam Roy, Serge Saelens, and Michael Schmut

SAP, Walldorf, Germany
Email: {stamatis.karnouskos, nadine.gaertner, nemrude.verzano, f.beck, andre.becker, santo.bianchino, daniel.kuntze,miguel.perez, rupam.roy, serge.saelens, m.schmut}@sap.com

***Abstract*—New Internet of Things open source technologies, middlewares, and programming languages, make the quick integration of devices, systems and cloud services easier than never before. With their utilization, complex tasks such as object detection, tracking and tracing, can be easily realized, even by embedded devices in a fraction of time. The interplay of highly heterogeneous IoT devices and open source software, has been utilized in this work as a learning tool, in order to train developers and enhance their IoT skills. By designing, implementing, testing and deploying a rapid prototype, new knowledge is acquired, assessment of technologies and concepts is carried out, and the end-result, although developed in a constraint timeframe, is technologically promising, cost-effective and feature-rich. This work sheds some light on the prototype implemented and discusses the developer experiences and benefits of this IoT integration hands-on approach.**

***Index Terms*—Internet of Things (IoT), Robot Operating System (ROS), Open Computer Vision (OpenCV), Lego Mindstorms EV3, Arduino Robot Arm, Augmented Reality, Cloud Services**

I. Introduction

The era of Internet of Things (IoT) [1] has empowered application and service developers with new capabilities that blur the borders of the physical and cyber worlds. Enterprise software can significantly benefit from this, as the representation of the physical world in backend systems can be done almost in real-time. The result is that decision-making processes can be utilized based on real-world up-to-date data, and as a consequence actions can be simulated in the cyber world, and decisions taken there can now be enforced in the real-world (management). Such dynamic systems that encapsulate both the power of the edge (physical world) and backend enterprise processes (e.g., ERP running on the cloud), have the possibility to enhance enterprise services and lead to more effective business processes. From a business viewpoint, these disruptive technologies can improve operational efficiency, empower outcome economy, enable human-machine collaboration and give rise to new connected ecosystems that blur traditional industry boundaries [2].

Autonomous machines will revolutionize industrial applications in the future. As an example, future warehouses are envisioned to strongly rely upon autonomous robots, that will embed themselves in the lifecycle of the relevant business processes in a transparent manner and optimally execute them. Real-time communication, cross-layer interaction among systems including, as well as the collaboration with human personnel, are expected to transform tomorrow's warehouse operations. Typical tasks that could be overtaken by robots include transport of products, visual quality inspection, optimization of available storage, real-time monitoring, predictive maintenance etc.

Open Source Software (OSS) today are in the forefront of efforts to research and develop innovative applications. A combination of several frameworks available, can easily be integrated to a solution that provides sophisticated capabilities with low effort and in a timely manner. One way to assess the suitability of such technologies, is the empirical one, i.e., a hands-on approach that aims realizing a prototype utilizing a large spectrum of the envisioned capabilities and software functions in a constrained scenario. In such a prototype, concepts can be explored, and the capabilities of the available software (in this case OSS) can be experienced by the developers.

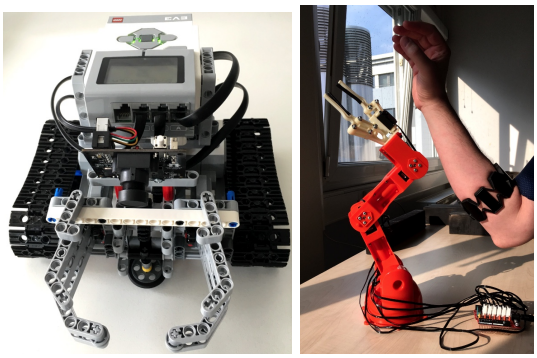
For a small group of developers, the challenge set was to demonstrate that relative simple autonomous tasks can already be realized easily by available IoT devices and empowered by OSS. As such, a prototype has been proposed, that would feature demonstration of capabilities, that show robot interactions while in parallel depicting autonomous behavior and enterprise integration. The following goals have been defined:

- Integrate with the enterprise system (backend)
- Demonstrate autonomous behavior of robotic devices
- In-sync monitoring of the real-world and backend information & management (digital twin paradigm)
- Distributed interaction among all systems and devices
- Utilize (learn & assess) OSS related to IoT device functionalities (integration & computer vision)

The prototype ought to be developed in less than eight weeks, while in parallel it would serve as a mean to learn the new OSS technologies and IoT devices and assess them. This paper provides insights on the outcome of this work i.e., the prototype and experiences acquired within the prototyping and demonstration period.

II. Scenario

The goals set in section I imply track and trace of moving assets with computer vision, relocation of physical objects with the help of IoT devices, integration with enterprise services and demonstration of the prototype. The IoT hardware available was: Lego Mindstorms *EV3* [3] (shown in Figure 1a), *Braccio* Robotic Arm [4], and Myo Gesture Control Armband [5] (both shown in Figure 1b). From OSS technologies, it was decided to use the Robot Operating System (ROS) [6], [7] as a middleware and OpenCV [8] for visual object recognition and tracking. On the enterprise side, the SAP IoT services was used to connect IoT to backend services and collect data. It has to be pointed that the developers did not have previous experience utilizing such IoT technologies, and therefore the eight weeks timeframe ought to include their learning curve, hardware setup, development environment setup, identification of a scenario, architecture design, incremental prototype development and integration.



(a) Lego Mindstorms *EV3* with Pixy camera (b) Braccio robotic arm and Myo armband

Figure 1: Some of the "things" used in the prototype

The following scenario has been realized: First via a user interface (UI) an order is issued for the production of a specific setting, i.e., a selection of 3 balls with specific colors (selected from blue, yellow, and red) need to be positioned in designated locations in the arena. The balls are thrown into the arena according to the production plan, by a robotic arm. Subsequently a robotic vehicle tracks them and moves them to their designated locations.

Tasks that need to be realized (e.g., throw/remove balls in/from the arena, moving of balls) are created asynchronously and can be picked up by the available IoT devices that have the capabilities to fulfill them. For instance, a "throw ball" task can be done by a robotic arm (*Arduino Braccio*), which can deliver the task of throwing balls of selected colors into the arena. Similarly, a "move task" can be picked up by the Lego *Mindstorms EV3*, which can visually track down the specific ball within the arena, pick it up and then transport it to its designated location. The scenario is finalized, once all balls are transported to their locations (as defined by the user) while no extra balls are in the arena. In case such extra balls are detected, they have to be brought back to the robotic arm, which disposes them to its storage area.

Although the *EV3* features a camera to track objects in its vicinity, a low-cost camera is mounted on top of the arena to monitor it and provide track & trace information of all available objects. Real-time status information for all devices, current camera view, reconstructed view from backend data (digital twin) of the arena, and augmented reality information are also provided to the end-users via a web based dashboard (shown in Figure 3). Each entity acts autonomously, while the information flows in a distributed manner among all stakeholders in the scenario following the publish/subscribe paradigm provided by *ROS* [7].

III. Architecture

Integrating the *things* in IoT can be realized in multiple ways in order to address the requirements posed [9]. Of key importance when designing IoT architectures is the consideration of device capabilities, information flow, data storage & processing, as well as other aspects such as scalability and extensibility. Such guiding principles have also been adopted in the architectural design of this prototype, which is depicted in FMC notation [10] in Figure 2. IoT prototyping requires systems which comprise components for controlling and communicating with robotic devices in a fast, and versatile manner.

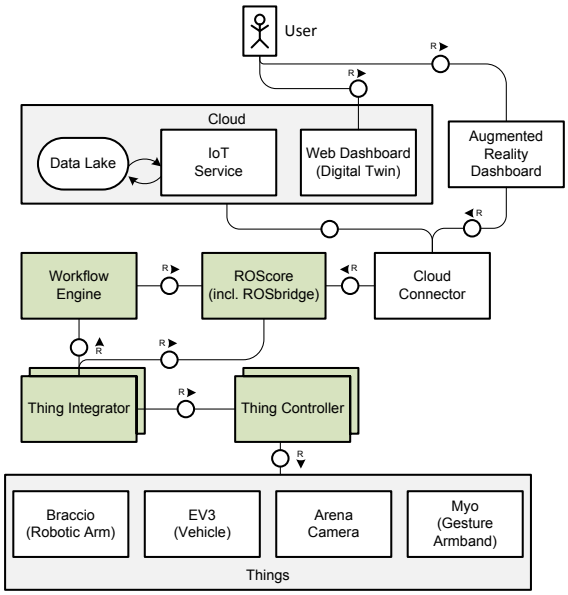


Figure 2: Main components of the system architecture; green boxes indicate *ROSnodes*

ROS [7] is a popular OSS for integrating robots, and covers well required features as such hardware abstraction, message passing, publish/subscribe capabilities etc. In *ROS* ecosystem models and reusable implementations for common physical devices are already available or can be easily extended to cover prototyping needs. ROS-based systems consist of a collection of independent processes which are called *ROSnodes*, while

Preprint of doi:10.1109/INDIN.2017.8104924

their discovery is done via the *ROScore*. As shown in Figure 2, in the realized setup one *Thing Controller* (*ROSnode*) and one *Thing Integrator* (*ROSnode*) are instantiated for each physical device, i.e., the *Braccio* (robotic arm), *EV3* (vehicle), Arena camera, and Myo (gesture armband). The *Thing Controller* handles the low-level device control, for example regulating the speed of the individual wheels of the *EV3*. The *Thing Integrator* is responsible for broadcasting and receiving messages related to the physical device, for example the device status.

Another *ROSnode* is the *Workflow Engine* which monitors the system and serves as a task administration entity. It publishes messages regarding the next tasks to be accomplished, for example to notify of missing balls. The robotics devices (*EV3*, *Braccio*) decide independently whether to bid for such tasks, and if awarded, they can execute them autonomously.

For enabling analytics, selected data generated by the activities is streamed from the physical system into the cloud. The entry point for the enterprise cloud is the IoT services [11], which accept incoming IoT data and store them in the data lake. The integration is done via the *Cloud Connector* component which interfaces with the *ROSbridge* which is a JSON API to the *ROS* system that exposes a websocket server. The *Cloud Connector* utilizes the *ROSbridge* to get access to the data available via *ROScore* and transforms the received messages into MQTT protocol [12] messages. Once available in the cloud, the data is prepared for end user consumption and can undergo e.g., analytics, dashboard visualization etc.

The end user interacts with the system via various options. An application enables the end user to start the scenario via a web interface (e.g., from a phone or tablet) by specifying and transmitting the overall ball constellation to be assembled by the robots. A web dashboard provides via the cloud a digital twin of the physical scenario along with real-time status information from the robots as shown in Figure 3. Augmented reality applications are connected and consume data streams via the *Cloud Connector* to display device-related information in place.

IV. Implementation

Driven by the architecture shown in Figure 2, all of the components have been implemented. These fully support the realization of the scenario as described in section II.

The *Web Dashboard* depicted in Figure 3 is a *SAPUI5* enabled *Spring Boot* application that shows a real-time view of the arena and the actions realized. The Dashboard realizes the "Digital Twin" approach in that it displays both an annotated image view coming directly from the *arena camera*, as well as a full reconstruction of the whole arena based on the data that comes from the messages exchanged via the devices and systems. As such both the cyber and real-worlds are in sync. In addition, information about the current goal (selected ball configuration), and status of the robots tasks, including their actions and state is visualized. An *Augmented Reality Dashboard* is also implemented and visualizes similar data via an iPad one this is pointed towards the devices or designated locations.

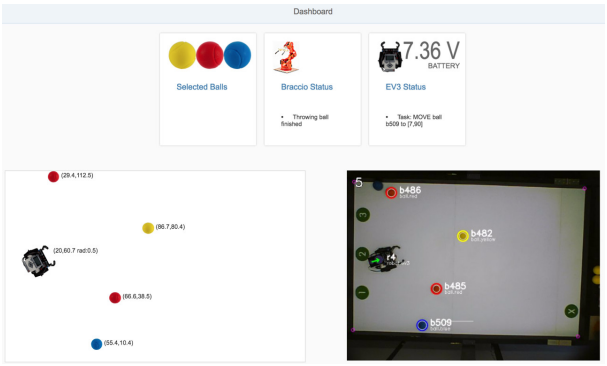


Figure 3: Digital Twin demonstration: Cyber (reconstructed) view on the left side and Real (camera) view on the right side

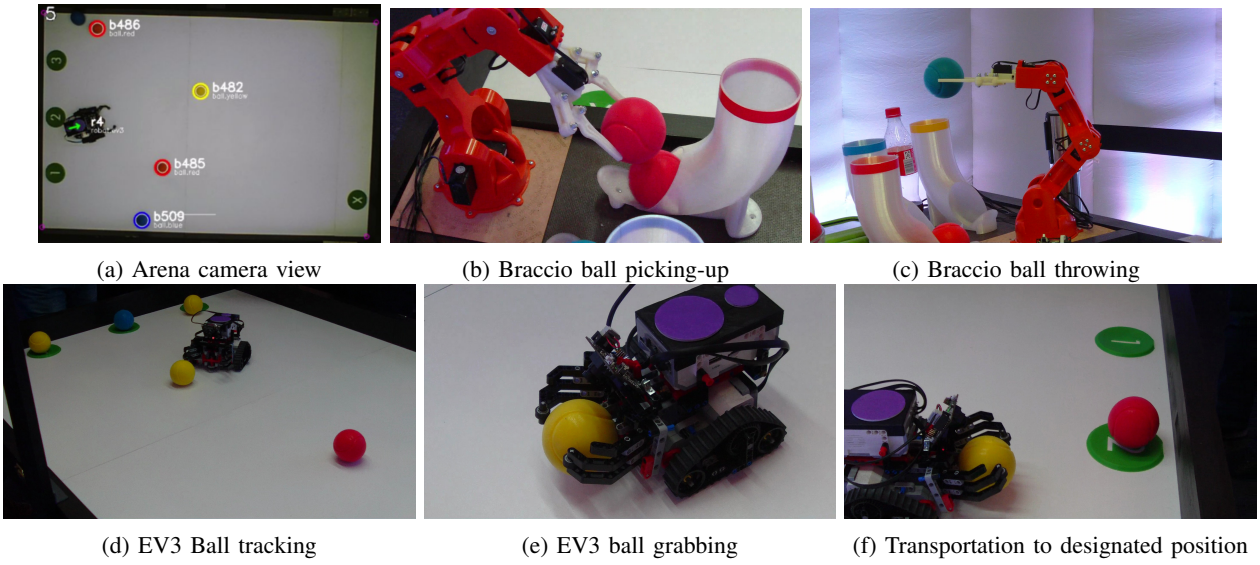
The *Workflow Engine* decides on actions that need to be carried out to satisfy a specific scenario i.e., the positioning of the balls to their selected positions. Once the information is available it creates a workflow and issues notification via *ROScore* about the available tasks. Such tasks are picked up by the *Thing Integrator* who acts on behalf of the available robot and bids for that task. Once a task is awarded, it has to be carried out by the respective robot who won the bidding. For instance if there are more than one EV3s capable of moving the balls around, and both bid, one of them will win and will get the task of actually moving the ball. The *Workflow Engine* is made aware of changes in the arena via an event-triggered method, either from the *arena camera* for changes in the arena pertaining devices and balls, or the *Thing Integrator* for bidding and acquiring tasks.

Once the *Thing Integrator* has acquired a task for a specific robot, the information is propagated to that robot's *Thing Controller*. For the *EV3* that implies the ball color and the designated position that it has to be transported to. For the *Braccio* this implies the color of the ball that needs to be thrown into the arena. The *Thing Controller* is a *ROSnode* that encapsulates the functionalities of the specific robot, and makes available a service that can be called upon it. Additional communication is happening over *ROS* topics.

The *arena camera* is a low-cost PS3 Eye USB camera, mounted on top of the arena, in order to identify and locate all items such as robots and balls, as shown in Figure 4a. The camera publishes in *ROS* topics a list of tracked objects available in the most recently processed camera image, and an annotated version of the input image where all detected balls and the robot, if detected, are marked and tagged with a discrete id. Everytime a new image is received, image processing starts and the results are published to the respective *ROS* topics. The camera has the capability to autodetect the corners of the arena via a Hough-Lines detection followed by a line intersection computation. The detected corners are then stored in the *ROS* parameter server.

The *EV3* is the robot equipped with a gripper to pick-up balls and a Pixy (CMUcam5) camera [13] for tracking the balls in its view field as shown in Figure 1a. In order to easily

Preprint of doi:10.1109/INDIN.2017.8104924



(a) Arena camera view (b) Braccio ball picking-up (c) Braccio ball throwing

(d) EV3 Ball tracking (e) EV3 ball grabbing (f) Transportation to designated position

Figure 4: Arena overview and example functionalities

track the robot and its pose by the arena camera, a cover with two circles is put on top of the *EV3*. There is a part running on the *EV3* itself, and a part running on a server as the *EV3* computational power is not adequate to run the *ROScore* itself, and therefore a separate CPU is needed. Services running in *EV3 ROSnode* enable the instruction of the robot via the *Thing Controller* which calls the different functions of *EV3* such as move, open/close the gripper, robot status (e.g., battery level, and gripper info), and identify via the *Pixy* camera an object and pick it by the gripper. Once a task is bid by the *Thing Integrator* e.g., move the blue ball to position 1, the *EV3* is instructed via the *Thing Controller* and autonomously attempts to fulfill the task as shown in Figure 4d – Figure 4f. The *EV3* acquires info from the *Arena Camera* in order to acquire the ball position and also track its own position. The *EV3* then utilizes its front camera (*Pixy*) once it is in the vicinity of the ball to follow the targeted ball and eventually capture it with its gripper as shown in Figure 4e.

The *Braccio* is a robotic arm with the capability of (i) picking up balls stored in the three 3D-printed containers and throwing them into the arena, and (ii) picking up balls from the arena and storing them back to the respective containers depending on their color, as shown in Figure 4b and Figure 4c. Similarly to *EV3*, tasks acquired by the *Thing Integrator* are passed to the *Thing Controller*. Services exist that expose the key functionalities of the *Braccio* arm in a modular fashion e.g., lift, pick, throw, and recover ball. The *ROSnode* controls the underlying *Arduino Uno* board via a serial interface.

The *Myo* gesture control armband is used as an alternative method of controlling the *Braccio* manually by a user as shown in Figure 1b. As such, tasks are then guided i.e., picking and throwing of the ball. The respective *Thing Controller* maps the Pose, Gyroscope and Acceleration data from the Myo Armband to device commands.

Finally, selected data and messages in the device world is propagated to the enterprise backend. This is realized via the SAP Cloud Platform IoT service [11] that acquires and stores the data for further analytics and visualization. To make that possible a *Cloud Connector* had to be developed that connects *ROS* to the SAP Cloud platform. The *Cloud Connector* subscribes to the websockets of the *ROSbridge* by the use of the roslibjs javascript library and sends received messages to the IoT Services via MQTT protocol [12].

V. Experiences and Lessons Learned

The development of the prototype achieved its goal of getting the developers acquainted with new OSS technologies pertaining IoT integration. As a follow-up, interviews have been realized in order to deep-dive to the experiences during the development and demonstration of the prototype, and acquire new insights on the process itself as well as the newly acquired knowledge. In addition, with respect to the prototype itself, several issues were raised in its design, development and operation. Here we discuss on some of the experiences and lessons learned from this prototyping exercise.

As the majority of the developers had no experience with most of the IoT technologies used (ROS, python, openCV, Arduino, etc.) the learning curve played a key role in the progress of the project. The high quality of documentation available for OSS, and especially online tutorials targeting beginners were of great help towards realizing the first steps. It has to be noted that all developers had several years of expertise with software engineering, hence online examples could be easily understood and integrated in the first steps of understanding what and how things are done in the IoT integration. For instance, the *ROS* examples available in the *ROS* web site [6], as well as code snippets pertaining key functionalities available in public web sites such as stack-overflow or github, enabled quick grasping of common ways things can be realized. Similar experiences were observed with

Preprint of doi:10.1109/INDIN.2017.8104924

openCV, Arduino, python etc. Even for specialized tasks such numerical computations with NumPy, the width and depth of online examples was more than adequate to help with specific issues that were raised during development. Being able to get easily help minimizes the risk of the developer being stuck in problem solving which is the most major factor in developer's unhappiness [14].

The prototype utilized a multitude of programming languages, including python, C++, JavaScript etc. All of the developers were familiar with compiled languages such as Java, Scala, C++ etc. which come with their respective Integrated Development Environment (IDE) and tools. When the option arose on the selection of tools e.g., to program *ROSnodes*, of either sticking to what they know (i.e., use C++) or go for something new (i.e., use Python), a split was observed. Some decided to follow the C++ path, as they were familiar with the language and tools that accompany it, and would only need to learn the *ROS* concepts. This was done in the hope of progressing more rapidly with the tasks and functionalities envisioned in the prototype. Others chose to follow python, since being a scripted language promised more interactive usage, that is fit for experimentation with IoT and has a low learning curve. Coming from a compiled language background, implementing prototypes with a scripted language like Python was a pleasant surprise for all of the involved developers. Key appraisals for it, were the high level of code readability, compactness of code (in comparison to Java or C++), the functional programming paradigm, the extensibility empowered by a wide range of libraries available, etc. The lack of an advanced IDE was a bit unusual at the beginning, but was soon overcome by setting up a coding environment easily with the utilization of vim, atom and utilization of style guides such as PEP8 that enabled homogeneous coding style. The latter limited the potential of bad code quality and code practice, which is one of the major reasons for developer unhappiness [14].

Linux environments are the predominant choice for IoT tools, incl. *ROS* and edge device integration. For some of the developers, this was the first time they had to set-up and use non-Microsoft Windows environments. The learning curve was steeper than with pure programming languages, especially due to the multitude of ways that things can be done, including command-line utilities. However, once the workflows were a bit clearer, such issues were quickly tackled and integrated into the everyday work. Of paramount importance in overcoming the difficulties, were other developers with Linux knowledge, that could guide the newbies.

As the development was done in a distributed manner, git repos were created, that enabled the developers to easily develop and interact with eachother. Of key appraisal is here the strong support for distributed and non-linear development, the easy merging of changes, the tracking of modifications in the code, the pull requests with code review, comments and compilation confirmation via continuous integration (CI) tools. Especially the automated builds via CI tools, helped tremendously in reducing conflicting changes, easing integration and testing. For those not used in this mode of development, the effect was very positive, as collaborative coding, integration, testing and deployment processes were transparent, simplified and timely utilized.

With respect to the design of the prototype itself, decisions taken focusing on autonomous task fulfillment, message-passing communication via *ROS*, and distributed architecture were good choices and fit for the intended purpose. The distributed workflow, the clear separation of duties and functionalities, and the independence from a central controlling point, enabled easy addition of new devices (scalability), while the local decision making meant that even without cloud connectivity, all tasks would be carried out. In case of connectivity loss with the cloud, data sync would be deferred to a later point, which had no operational influence, apart from the UI for monitoring. Another backup solution would be to instantiate a local copy of the dashboard which would tap to the messages provided via the *ROSbridge* and could depict a large portion of the monitoring functionality. Due to the high level of expertise with cloud technologies, integrating the prototype with cloud applications and services unproblematic. The integration was eased also with the development of connectors such as the *Cloud Connector* (shown in Figure 2), which enabled data to flow into IoT services and the web dashboard.

Testing of the whole system was done in a hybrid manner, meaning that some tools simulating behaviors of the individual components, especially with focus on the interactions among them, were developed. These were used to guarantee the interoperable interaction among the different *ROSnodes*, and for debugging of potential problems, while developing more in-depth functionalities per component. Holistic integrations of all components were done in order to detect additional issues in "live" mode. It has to be pointed out that *ROS* has several tools that ease testing and simulation, such as the "bags", that enable data logging (via subscription to *ROS* topics), which can then be used for data playback, and therefore greatly enhance the testing of the individual parts of the system. This can simplify individual testing and shorten the respective development, testing and integration lifecycles. As IoT systems scale and get more complex, simulation and modelling tools will be increasingly needed to develop and test them sufficiently [15].

Not all devices utilized the same technology, and that created additional work to homogenize them. As an example, the *EV3* utilized python 3.4 while the *ROS* deployment in *EV3* was in python 2.7. That resulted in some recomplilation in order to support the utilization of *ROS* in the *EV3* environment. At this stage, the *EV3* does a track and trace of the balls, the position of which however is pinpointed by the top mounted camera. A more sophisticated object detection at the *EV3* level is possible; however that would result in more processing time due to the limited resources. Complementing the *EV3* with an external board e.g., a Raspberry Pi to handle computationally costly tasks would be a good solution to increase on-device intelligence and autonomous behavior.

Preprint of doi:10.1109/INDIN.2017.8104924

In *Braccio* robotic arm, although several controlling actions could be realized, what was missing were sensors for accurate self-positioning of the robot (and reporting of deviations). As such, the developer had to assume the exact position of the robotic arm after the execution of a command (e.g., rotate 45°), which might be slightly different each time due to the mechanics involved. The presence of sensors such as gyroscope etc. could have enabled the robot to provide accurate pose and as a result the developer would have to make less assumptions/checks in the code. In addition, as the *Braccio ROSnode* depended on the underlying *Arduino* board to execute the hardware movements, sometimes serial buffer overflows were detected which led to unpredicted robotic arm behavior. As such, better synchronization between high level (*ROS* messages) and low level (*Arduino* control) is needed, including potential prediction of overflows and corrective measures. As there is no safety-control on the movements of the robotic arm, quick and uncontrollable movements of the robot may lead to partial destruction of the arm itself, or the motors that power it. Hence, safety mechanisms ought to be integrated, to avoid damage and enhance operational safety.

During a two full-day demonstration of the prototype to the public, and the continuous operation of the robots and other tools, some issues were detected, that could be enhanced in the future. For instance, flickering in the top-camera view (e.g., due to shadows, or because of the *EV3* partially covering a ball), resulted in a ball being detected as missing from the arena, which subsequently started other events, such as the throw of another ball of the same color by the *Braccio* in the arena. Although excessive balls were removed by having the *EV3* put them in a designated spot near the *Braccio*, so that the robotic arm could fetch and put them back in their bins, this process could have been better optimized. Another aspect is that the actions of the robots could have performed better if they had some prioritization, that would enable them to more wisely select their tasks e.g., avoid changing goals in the middle of tracking a specific ball, or prioritize bringing the balls to their designated locations and then attempting to clear the arena from excessive balls. Such issues could be potentially enhanced with introduction of reinforcement learning.

In some cases, the robot would approach a ball with high speed and an angle, which would result to the robot being turned upside-down, something that would require human intervention. Such issues could be avoided with adjusting the speed and more careful approaching of the target. Similarly, the *Workflow Engine* could be enhanced with different work strategies e.g., prioritize balls, parallelize robot tasks based on hypothetical routes they will follow in order not to collide, prioritize robots that perform better over time, try to be energy-aware by selecting robots in a way that would not deplete their battery, detect misbehaving or out-of-order robots and reassign their tasks etc. Such considerations are however seen as future work, and could be potentially done with machine learning, in order to attempt the creation of generalized strategies that the workflow can utilize depending on the situations that arise in the arena.

VI. Conclusion

Learning by doing, especially in the world of IoT, is powerful and has proven to be a fruitful approach for getting acquainted with new technologies and concepts. Especially in the rapidly evolving domain of IoT, that features a plethora of heterogeneous devices, software, development approaches and challenging integration, learning by doing is witnessed to be beneficial even for newbies. The developed prototype utilized several robots (*EV3*, *Braccio*) and other IoT devices, and focused on their interplay in order to realize a scenario that brings forward aspects of autonomous behavior, decentralization, decision-making, object detection, track & trace, integration with enterprise systems etc. Its goal of training the developers to new technologies, and having them assessing a variety of OSS and development approaches was a success. As discussed, the experiences acquired in a limited amount of time, have enhanced the developer skills and how problems pertaining physical devices and enterprise software can be approached in the future. In addition, the feedback from the two full-day demonstration of the prototype at a company internal event was very positive, and demonstrated that the integration of state of the art technologies can be done relative easy, even by non-experts in the domain, who can utilize it for solving complex tasks. The experiment also made obvious what many strictly-software developers usually ignore: the dynamics introduced by the hardware of the devices, and the unpredictability of the real-world that impacts the devices, their sensors and eventually how software application perceive the real world via those sensors and their data.

References

[1] J. Höller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle, *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Academic Press, Elsevier, 2014.

[2] WEC, "Industrial Internet of Things: Unleashing the Potential of Connected Products and Services," World Economic Forum (WEC), Tech. Rep., 2015. [Online]. Available: http://www3.weforum.org/docs/WEFUSA_IndustrialInternet_Report2015.pdf

[3] Lego Mindstorms EV3. [Online]. Available: https://www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313

[4] Braccio Robot Arm. [Online]. Available: http://www.arduino.org/products/tinkerkit/arduino-tinkerkit-braccio

[5] Myo Gesture Control Armband. [Online]. Available: https://www.myo.com/

[6] Robot Operating System (ROS). [Online]. Available: http://www.ros.org/

[7] A. Koubaa, Ed., *Robot Operating System (ROS)*. Springer International Publishing, 2016.

[8] OpenCV. [Online]. Available: http://opencv.org/

[9] S. Karnouskos, V. Vilaseñor, M. Handte, and P. J. Marrón, "Ubiquitous Integration of Cooperating Objects," *International Journal of Next-Generation Computing (IJNGC)*, vol. 2, no. 3, 2011.

[10] Fundamental Modeling Concepts (FMC). [Online]. Available: http://www.fmc-modeling.org

[11] SAP Cloud Platform IoT Service. [Online]. Available: https://cloudplatform.sap.com/capabilities/iot/iot-service.html

[12] MQTT specification. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

[13] Pixy (CMUcam5) camera. [Online]. Available: http://cmucam.org/projects/cmucam5/wiki

[14] D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson, "On the Unhappiness of Software Developers," *ArXiv e-prints*, Mar. 2017.

[15] G. Kecskemeti, G. Casale, D. N. Jha, J. Lyon, and R. Ranjan, "Modelling and Simulation Challenges in Internet of Things," *IEEE Cloud Computing*, vol. 4, no. 1, pp. 62–69, Jan. 2017.

Preprint of doi:10.1109/INDIN.2017.8104924