# Synchronous Reconfiguration of Distributed Embedded Applications during Operation

Kilian Telschig
Corporate Technology
Siemens AG, Munich, Germany
kilian.telschig@siemens.com

Alexander Knapp
Institute for Software and Systems Engineering
University of Augsburg, Germany
knapp@informatik.uni-augsburg.de

***Abstract*—Speed of adaptation to changing demand is a critical success factor in factory automation. The key to speed is to enable agile development by independent engineer offices and equipment producers with industrial-grade microservice architectures. The expensive drawback is: While software components evolve over time, manufacturers have to integrate and deploy more and more updates during costly production stops. To avoid production stops as much as possible, we propose reconfiguration extensions to a real-time container architecture proposed earlier. The original container approach addresses both the functional and non-functional aspects of integrating embedded software components in late engineering phases. The extended approach allows modifications of the running distributed embedded application even during operation, while continuously ensuring reactivity of the system. The agents running on each node prepare the reconfiguration in background and then synchronously perform the required modifications according to a detailed reconfiguration plan. We demonstrate our concept by describing a synchronous API change between two distributed software components of a running gesture recognition system. An evaluation shows the feasibility of the concepts, but also calls for further research.**

## I. Introduction

The first thoughts towards safe dynamic updates of distributed embedded applications based on real-time containers were published in [1]. The authors claimed: *Speeding up adaptivity also leads to an increased demand for automatic assurance and enforcement of quality attributes such as real-time constraints, including consistency properties to be maintained during reconfiguration.* Consistency and quality of distributed embedded applications have been addressed for the static case (without dynamic updates) by the real-time container architecture described in [2]. First, the architecture isolates each embedded component in a container. Second, a real-time container agent (agent) on each node configures the containers and the distributed system appropriately and actively triggers computation, network communication and I/O at specific points in time. This enables the isolated components to communicate with each other and with the equipment exactly as declared in their interfaces. Proposed for late integration by third parties, this approach also builds a good basis for reconfiguration, as the agent completely controls the distributed embedded application.

In this paper we extend the real-time container architecture to enable deterministic concurrent reconfiguration of distributed embedded applications during operation. The major challenge we address is that some reconfiguration steps such as downloads or container initializations take a hardly predictable amount of time and resources, while other steps interfere with the application in a critical way and must be performed as a synchronized action of the distributed system. Therefore, the agents synchronously perform modifications using detailed instructions from a reconfiguration plan consisting of synchronization steps, background steps and real-time steps as needed for the application. Using these primitives a reconfiguration plan can change the set of component interfaces, how they are connected with each other, how they are allocated to nodes and which resource requirements they have with regard to execution and communication. We describe the structural and operational aspects of our approach and required execution facilities added to the real-time container architecture. As motivation and running example we describe a concrete synchronous update of a gesture recognition component and a reporting gateway component during operation. We evaluate the approach by applying an adapted version of this reconfiguration to a similar application described in [2] without breaking the embedded functionality.

The paper is structured as follows. Section II covers related work. Section III describes the gesture recognition system used as running example throughout this paper. After a brief overview of the real-time container architecture in Section IV we describe our reconfiguration approach in Section V including the structural and operational details of reconfiguration plans as well as the proposed extensions of the real-time container architecture. Section VI presents the evaluation results followed by a conclusion in Section VII.

## II. Related Work

We refer to work related to dynamic reconfiguration of distributed real-time systems. Many approaches to dynamic reconfiguration in the context of component-based software engineering are summarized by Hammer [3]. The original quiescence approach [4] blocks the complete dependency graph during reconfiguration. The amount of blocking has been reduced by numerous approaches, most prominently by the tranquility approach [5]: It minimizes blocking by fading over, i.e., sending new transactions to new components and inhibiting new transactions by the old components. However,

we want to achieve zero blocking in real-time systems to maintain reactivity of the embedded application.

Sha et al. [6] proposed the first dynamic reconfiguration approach for real-time systems based on in-field testing and atomic switch-over. The factory automation standard IEC 61499 [7] for distributed control systems enables dynamic updates as demonstrated by Zoitl et al. [8]. Kirsch et al. [9] propose a similar approach for HTL [10], a real-time runtime system, which has also been extended for distributed systems. While these approaches aim at dynamically reconfiguring real-time systems, distributed dependencies cannot be addressed during operation. This also applies to more recent approaches, e.g. using Design Space Exploration to identify configuration optimization potential [11], automatic updates based on Erlang features [12] and the exploitation of the dynamic slack to save energy [13]. Finally, service-oriented approaches (e.g. Kothmayr et al. [14]) and agent-based approaches (e.g. Chaplin et al. [15]) to evolvable cyber-physical systems do not solve this issue, either.

When interfaces change, these approaches are not sufficient for dynamic reconfiguration of distributed compositions with continuous reactivity. The problem is: To satisfy end-to-end real-time requirements, the deterministic timing of the data flow between interacting components must be maintained. We address this problem based on the real-time container architecture [2], which combines three elements: A model-based engineering approach similar to AUTOSAR [16] extended with non-functional interface descriptions; enforcement of these interfaces by a container-based runtime architecture using Linux Containers (lxc) [17]; and temporal decoupling of the components using the logical execution time paradigm (LET) [18] originating from Giotto [19], implemented with a globally synchronized cycle. We give a more detailed description of the real-time container architecture in Section IV, as it is the basis of our reconfiguration approach.

Summing it up, we propose a novel approach for dynamic reconfiguration, which is the first to roll out even breaking changes to distributed applications with real-time requirements without any blocking of components.

## III. Running Example

We describe the embedded system *CamSys* (see Fig. 1) used both as motivation for synchronous reconfiguration and as running example throughout this paper. The system consists of two micro-controllers connected via Ethernet. The first node has a USB camera installed, while the second node has an additional network connection to external observers via an engineering system. A distributed embedded application implements the following functionality. A statistics-based software component at the first node constantly checks the video stream for gestures. No relevant gesture shall be missed, which can be ensured with a sample rate of five frames per second due to usual human behavior, i.e., 200 ms period. Each time a gesture appears, this event is reported to a remote service beyond the system scope. The system shall not report the same gesture multiple times, even though it is probably visible in multiple frames. Therefore, the event stream shall be further filtered from hardly possible sequences of gestures.

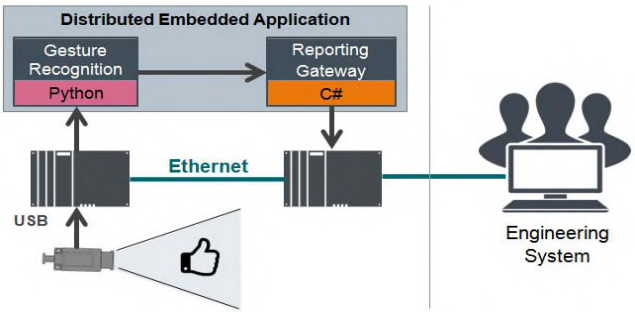


Figure 1. The running example *CamSys*: Two micro-controllers are connected via Ethernet. A gesture recognition component processes the video stream produced by a camera installed on the first node every 200 ms. Detected gestures are sent to a gateway component located at the second node, which is responsible for further reporting to network nodes beyond the system scope. Via the plant engineering system, updates of the distributed embedded application shall be rolled out during operation.

During operation, an update of the distributed embedded application is triggered from the engineering system. In this scenario the gesture recognition component has been improved by the third-party solution vendor to detect more types of gestures. Unfortunately, the output interface had to be evolved in an incompatible way to cover the new entry types. Consequently, the reporting component has to be adapted to understand the new data format. We want to roll out enhanced functionality without stopping the application, but still without loss of data caused by faults such as buffer overflows or serialization exceptions. Due to resource constraints it is not possible to run both versions of the reporting gateway component in parallel for a blue-green deployment.

Hence, both components need to be adapted in the following manner using appropriate reconfiguration extensions:

1) Prepare reconfiguration in background (i.e., download and initialize added components).
2) Synchronize across nodes.
3) Switch synchronously so that all messages are processed by the compatible component versions.
4) Finalize reconfiguration in background (i.e., shutdown removed components).

## IV. Real-Time Container Architecture

We build our reconfiguration approach on the real-time container architecture. We briefly discuss the benefits of using this architecture for our concept. Then we describe the real-time container architecture and operational details based on the *CamSys* model. Both the structural and operational details are important for the reconfiguration described later.

### A. Architecture Concerns

The architecture supports engineering and execution of distributed embedded applications made of containerized time-triggered software components. The major feature is separation

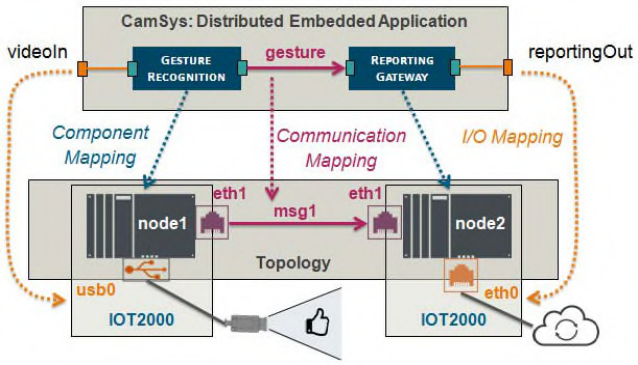


Figure 2. Overview of the *CamSys* model: Two software components are assembled to a distributed embedded application. The elements of the distributed embedded application are then mapped to the distributed system, which consists of two connected micro-controllers (cf. Deployment in [2]).

of application logic from deployment concerns despite end-to-end real-time requirements. From outside of the containers, the execution and interaction of software components are completely in the hands of the execution environment. This finally leads to the possibility to reconfigure the distributed embedded application by means of the runtime environment in a generic and predictable way. An additional benefit for reconfiguration from a software engineering perspective: It simplifies late integration of software components with or without real-time requirements implemented for different platforms by independent third-party solution vendors. Therefore, the real-time container architecture is a technical enabler for software ecosystem scenarios in industrial domains. Table I classifies the goals and benefits related to selected conceptual features of the real-time container architecture.

### B. CamSys Architecture Model

The real-time container architecture comes with a model-based engineering methodology and uses the resulting model to configure and execute an application. The reconfiguration approach presented in this paper modifies the distributed system based on instructions which refer to this model. We briefly describe the *CamSys* architecture model (see Figure 2).

A distributed embedded application is an assembly of software components, which is defined independently from any deployment aspects. The *CamSys* application consists of two software components connected by one port connector: The *Gesture Recognition* component and the *Reporting Gateway* component. They communicate with their environment via asynchronous cyclic messages (sender-receiver interaction) – including inter-component communication. The transport and delivery of messages is handled from outside of the containers based on the component interface, which describes the data and the non-functional requirements of required and provided messages. The *Gesture Recognition* component has the port *videoIn* via which it cyclically expects a camera frame. Its cyclic output provided via its *gestureOut* port is a message containing the information about recognized gestures. The *gesture* port connector effectively makes this message an input of the *Reporting Gateway* component via the *reportEvent* port. The *Reporting Gateway* filters duplicate reports and sends new gestures via the *reportingOut* port. Finally each software component is assigned a cyclic execution time during which it processes received messages and produces the provided messages. All such tasks of the software components are executed synchronously in logical execution time (LET), i.e., during each system-wide cycle all inputs are injected to the respective container, then all containers are provided with execution time as requested, finally all outputs are extracted and propagated to the receiver as configured.

Also important for reconfiguration is the logical deployment of the *CamSys* to the distributed system as follows. The two nodes are SIMATIC IOT 2040 micro-controllers modeled with a USB port and two Ethernet ports. For the USB port a camera driver has to be offered by the board support package, represented by an IO port information. A generic gateway protocol API for Ethernet ports is needed, too. The two nodes are connected via their *eth1* network interfaces. The *Gesture Recognition* component is mapped to *node1* and its *videoIn* software port is mapped to *usb0*, which is also associated with the camera driver in this step. The *Reporting Gateway* component is mapped to *node2* and its *reportingOut* software port is mapped to *eth0*, for which an association with the gateway protocol driver is added. Finally, the gesture connector is mapped to a network message from *node1* to *node2*.

In this way the structure and the non-functional requirements and properties of software components, the distributed embedded application and the deployment-dependent final system are modeled as input to the runtime configuration and execution.

### C. Application Execution Environment

Figure 3 shows an overview of the concrete system architecture resulting from the *CamSys* model. Each software component is wrapped in an lxc container on the corresponding node. Drivers are employed and triggered by the agent as implied by the I/O mapping. Inter-component communication is controlled via means of the network stack (firewall and traffic control). The barrier queuing discipline (qdisc) is used at each virtual ethernet adapter: outputs are caught at egress of the interface eth0 inside the container's network namespace, while inputs are caught at egress of the veth pair device enslaved by the network bridge lxcbr0 in the host. Each cycle the agent "moves" the barrier qdisc to release all messages enqueued in the previous cycle. After processing outputs and then inputs of each local component, the agents start the execution phase by triggering all processes in each container's cgroup sending interrupts. In this way the agents coordinate the containers across nodes by ensuring synchronous execution and managing interaction and access to installed equipment as suggested by the LET paradigm.

Listing 1 shows the agent routine in pseudo code. Figure 4 shows a schematic overview of the resulting timing behavior of the *CamSys*. Both agents are scheduled with EDF and run

| conceptual feature | original goal | additional benefit for reconfiguration |
|---|---|---|
| functional isolation | inhibit undeclared side effects | independent component life-cycles |
| resource control | ensure non-functional properties | deterministic timing of reconfiguration steps |
| synchronous execution | decouple functional from temporal aspects | cycle turnover as moment of tranquility |
| managed interaction | ensure isolation and control network capacity | enables interception of messages |
| managed I/O | ensure isolation and revoke capabilities | intercept access to equipment |

Table I
THE ORIGINAL GOALS OF THE REAL-TIME CONTAINER ARCHITECTURE AS DESCRIBED IN [2] AND BENEFITS FOR RECONFIGURATION RESULTING FROM THE CONCEPTUAL FEATURES.

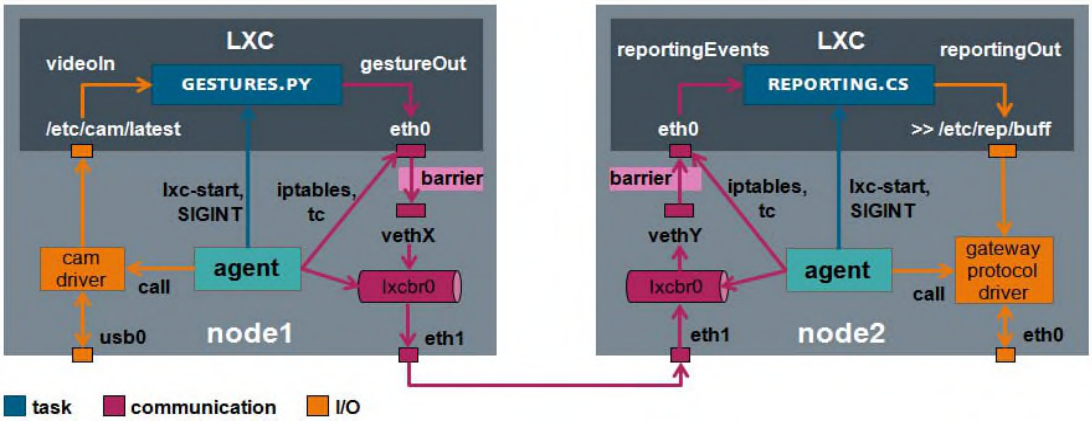


Figure 3. Overview of the lxc-based system architecture resulting from the *CamSys* model.

phase-aligned. The clocks are synchronized in background using PTP and the cycle counters are synchronized by the agents. The container execution is mostly managed by Linux's scheduler as configured via lxc (CPU time, network quota, etc.). However, the real-time tasks have to yield at the end of each cycle to align with the period and to indicate successful cycle completion. The agent triggers these suspended real-time tasks at the start of each execution phase. In the administration phase between each execution phase the agent checks the system state (e.g. successful termination of all real-time tasks). Then it extracts the components' outputs of the previous cycle by either moving the barrier or by triggering a driver. Finally the agent injects the inputs in the same manner before triggering the next execution phase. The three rcHook calls in the pseudo code belong to our reconfiguration extensions.

## V. RECONFIGURATION APPROACH

We describe a concept to perform a deterministic reconfiguration of a distributed embedded application during operation based on the real-time container architecture. A reconfiguration can be (cf. [1]):

- Add software component to running system
- Move running software component to another node
- Update running software component
- Remove running software component
- Sequential and parallel combinations of these modifications

```
every period:
    if not (health):
        trigger fault reaction;

    rcHook(before output processing);

    for each output o:
        if I/O then trigger o.driver;
    move output barrier;

    rcHook(between I/O); // logical cycle turnover

    move input barrier;
    for each input i:
        if I/O then trigger i.driver;

    rcHook(after input processing);

    for each component c:
        interrupt processes of c;
```

Listing 1. The agent routine performed on each node as pseudo code. Three calls of the reconfiguration hook function rcHook are added by our reconfiguration approach. In these hooks the applicable reconfiguration steps of the present reconfiguration plan are executed.

To prevent damage caused by a failing system, these modifications must not cause unintended effects on (the rest of) the application, i.e., processing, communication and control must be performed within the defined deadlines to keep up quality and consistency. Regarding consistency we cannot use the common blocking approach to dynamic reconfiguration of

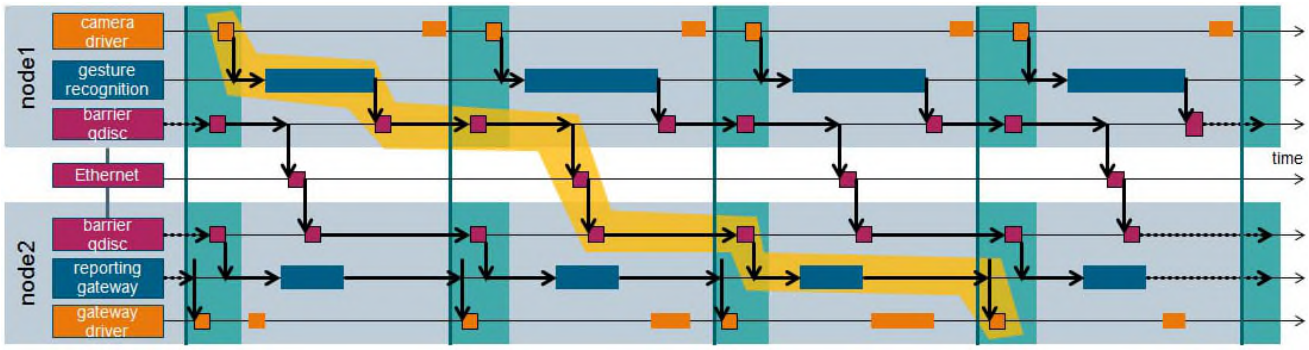


Figure 4. A schematic timing diagram of four cycles of the *CamSys*: The image data flows through the system parts so that each entity processes it at a point in time with a deterministic cycle offset across nodes. One end-to-end chain is highlighted in yellow: In the first administration phase, the agent on node1 injects the camera input to the gesture recognition component and triggers it. In the subsequent execution phase this image is processed and the result is sent out, but blocked at the barrier qdisc. The agent moves the barrier in next output processing phase, so the message is sent to the outside of the container in the second cycle. On node2, the message is enqueued in the barrier qdisc of the reporting gateway component. In the next input processing phase, this message is released to the inside of the container. After triggering the reporting gateway it processes the gesture information during the third execution phase and creates the output. In the next output processing phase, agent2 triggers the gateway protocol driver providing the extracted output.

distributed systems, because we want to keep up reactivity of the embedded system. Instead, our approach exploits the cyclic behavior of the real-time container architecture: The cycle turn-over is an opportunity for reconfiguration, as it is in fact a moment of tranquility (no task is performed and no message arrives/leaves without active triggering by the agent). To temporally coordinate the reconfiguration steps across nodes we basically use the cycle number, which is already synchronized by the agents. However, the cycle number cannot be defined when starting the reconfiguration, because some tasks take unpredictable execution time, such as the start of an additional component in a new container.

Therefore, the reconfiguration operations mentioned above can only be offered by rolling out appropriate reconfiguration instructions to the agents running on each corresponding node and employing a synchronization protocol. Each agent has its own node-local instruction list, the reconfiguration plan, which it executes step-by-step. The reconfiguration plan consists of three types of reconfiguration steps: Real-time steps, synchronization steps and background steps (see Figure 5). The agent checks the reconfiguration plan in the reconfiguration hooks shown in Listing 1 performing only applicable steps. In the rest of this section we describe these steps in detail and then show how they can be applied to reconfigure the *CamSys*.

### A. Synchronization Steps

Synchronization steps are used to determine a reference cycle – the synchronization point – to then perform blocks of real-time steps in a timely coordinated manner across nodes. There are three different kinds of synchronization steps:

**notifyAndWait:** Create and send notification messages to observer agents containing the synchronization point id and a feasible continuation cycle number at which the next block of reconfiguration steps starts. The absolute cycle number is calculated based on the worst-case-communication time to the observers known from the

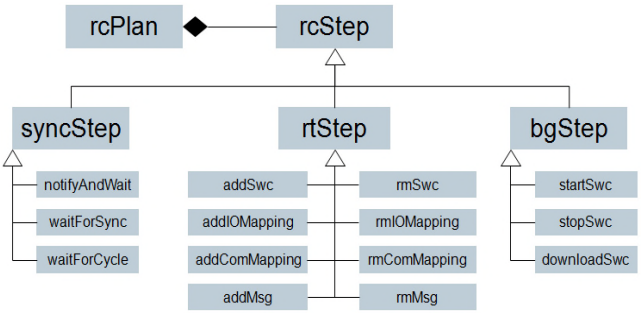


Figure 5. The structure of a reconfiguration plan: Each agent gets its node-local instructions as reconfiguration steps (rcStep) to be performed sequentially. A step is either for temporal synchronization with other nodes (syncStep), for instantaneous local system modification (rtStep) or for a background operation (bgStep).

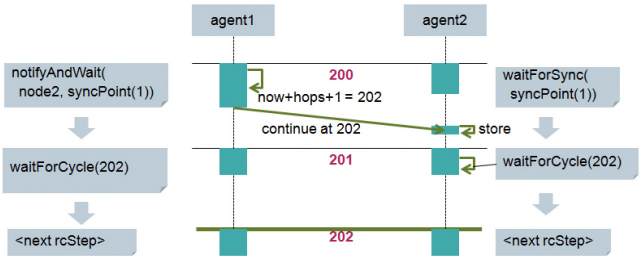


Figure 6. The synchronization protocol: In cycle 200 the next rcStep at node1 is notifyAndWait, so agent1 creates a notification message containing a feasible continuation cycle number calculated based on the worst-case communication known from the network topology model. agent2 is located at node2 with one hop distance. The next rcStep is waitForSync, which does not apply until the arrival of the notification associated with synchronization point 1. After the notification arrived the rcStep applies in cycle 201. The result: Both agents wait for the calculated cycle number 202 and only then continue synchronously with subsequent reconfiguration steps.

network topology model such that all agents receive the message at latest one cycle before the continuation, i.e., $(max_{WCCT}/period)+1$ (cf. [2]). After sending the notification(s), the agent replaces this step with waitForCycle.

**waitForSync:** Wait for a notification associated with a certain synchronization point id. On receipt, this step is replaced with waitForCycle as prescribed by the notification message.

**waitForCycle:** Simply wait for a certain absolute cycle. This step is only applicable in the given cycle and then instantly terminates, leading to the next step.

Figure 6 shows the synchronization protocol resulting from a notifyAndWait on one node and a waitForSync on the other node. When two nodes have to continue synchronously after background tasks, a two-way handshake is needed. For this use case, one of the two nodes has to notifyAndWait and then to waitForSync afterwards. The other node performs waitForSync and then notifyAndWait. Then the first node completing its background task will still wait for the continuation cycle reported by the other node. The continuation cycle calculated in the first synchronization point might have passed already when the second node finishes its background task. However, the first node then still waits for the continuation cycle delivered in the synchronization notification message of the second node. Consequently, the agents must accept and store notification messages even when not waiting for it or when the reported continuation cycle has already passed. Another aspect is network consumption: The system model used by the real-time container architecture contains a constant $M_{adm}$ for over-estimating the agent communication per cycle, which has to be modified accordingly. This way we avoid traffic control for inter-agent communication outside of the containers.

### B. Real-Time Steps

Real-time steps modify the node-local parts of an application during its operation. Table II shows an overview of the real-time steps currently supported in our approach. As visible from the table, real-time steps are simple changes such as for example the rmComMapping (remove communication mapping), which deletes two iptables firewall rules. Obviously, these modifications can completely break an application, if not applied carefully and with accurate timing across nodes.

For temporal coordination of the reconfiguration across nodes, each real-time step is associated with a synchronization point, a cycle offset and one of the three reconfiguration hooks. The agent executes the real-time step in the dynamically determined continuation cycle related to the synchronization point plus the cycle offset. Depending on the specified hook a real-time step can be placed in the administration phase either before output processing, between I/O or after input processing as indicated in Listing 1. The choice of the hook impacts data flow, i.e., when specific inputs or outputs shall be started to be treated differently. Figure 7 shows the effect of removing a communication mapping from the *CamSys* between I/O.

One remark on the feasibility of a reconfiguration plan w.r.t. the execution time of real-time steps: Real-time steps are assumed to take zero execution time (ZET) like the treatment of drivers in the logical execution time paradigm [18] (and thus in the real-time container architecture). For the scope of this paper we compensate this assumption by over-estimating the required agent reservation, which is possible due to the limited number of reconfiguration steps per cycle. In practice, it must be checked whether the agent reservation suffices for an rcPlan by summing the worst-case execution times of steps possibly performed in one cycle.

| rtStep | realization |
|---|---|
| addSwc | start triggering the component's processes in the next execution phase |
| addMsg | add tc class for rate control |
| addComMapping | add firewall rules to mark and re-direct packets from/to a specific port |
| addIOMapping | start conveying information between an I/O and a software component using a specified driver (driver-specific) |
| rmIOMapping | do no longer handover the hardware signal between the driver and the software component |
| rmComMapping | do no longer forward specific UDP packets |
| rmMsg | remove associated tc class |
| rmSwc | do no longer trigger the component's processes |

Table II
OVERVIEW OF THE SUPPORTED RTSTEPS AND THE CORRESPONDING ACTIONS TAKEN BY THE AGENT. RECONFIGURATIONS REGARDING I/O AND COMPONENTS MODIFY THE AGENT ROUTINE BY ADDING OR REMOVING TRIGGERS. RECONFIGURATIONS REGARDING NETWORK MESSAGING ARE PERFORMED VIA MEANS OF IPROUTE2 AND IPTABLES.

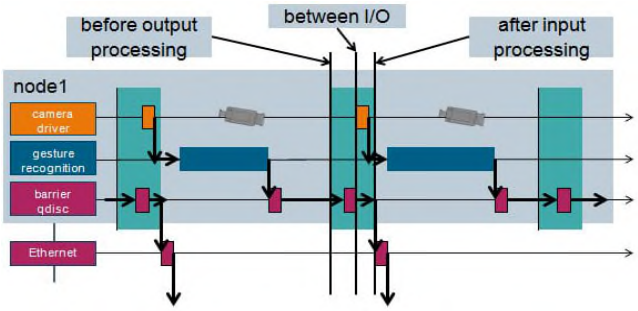


Figure 7. The execution of the rtStep rmComMapping in the agent hook between I/O: the output has already been granted by marking, forwarding and moving the barrier qdisc in the output processing phase, so the gesture output is sent once more. Starting in the third visible cycle, the gesture information is not sent to the gateway component anymore.

### C. Background Steps

The third and last kind of reconfiguration steps are background steps. They address the problem that some operations necessary for dynamic reconfiguration are long running and also unpredictable in duration. Such operations cannot be assumed to take ZET and put into the agent routine, but instead they have to be run in background, while the application continues. This applies to the following operations:

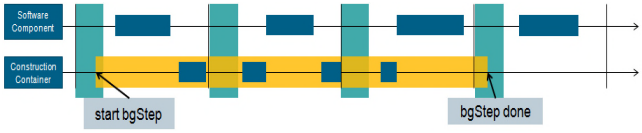


Figure 8. The execution of bgSteps: An additional construction container is running on each node during the complete system life-time to perform bgSteps within resource control as triggered by the agent during cycle turn-over.

**downloadSwc:** A software component (i.e., a container image and a description) is downloaded to the target node.

**startSwc:** An additional software component is initialized by ramping up the corresponding container.

**stopSwc:** A running container is stopped.

For these operations an additional construction container is running on each node during the complete system life-time. The construction container is not an lxc container, but a background worker running in a dedicated cgroup called bgc. The construction container is assigned a static percentage of the system resources on the node (network and cpu) to run side by side with the embedded application without interfering with it (see Figure 8). The background worker is a non-real-time component of the agent and waits for background steps – most often idling. If the next reconfiguration step is a new background step, the agent routine triggers it inside the construction container between I/O processing by sending a message containing the job information and marks the step as started. Between I/O processing in each subsequent administration phase the agent checks if a completion message from the construction container has been received. Only then the agent consumes the background step and continues with the next step.

Triggering and execution inside the construction container depend on the kind of background step. For example, if a new container is started, the container image must have been installed. A component-specific start command is transmitted to the background worker, which starts the container using lxc-start providing the given start command. As the background worker runs in the resource-controlled bgc cgroup and this cgroup membership is inherited by lxc-start, the container starts in background. During start, the component is moved to the component's own cgroup which is initially configured such that bgc and the new container together do not exceed the background reservation. We suggest that after initialization the component indicates readiness and waits for the first agent trigger to start a cycle by sleeping. Only when the background worker notices that the component is ready it reports completion to the agent routine. The component is now ready to be wired and activated using real-time steps and the construction container is ready for the next background step.

### *D. Reconfiguration of the CamSys*

Figure 9 shows an architectural overview of the required reconfiguration of the *CamSys* system, which is motivated in Section III. The two software components shall be updated during operation using our reconfiguration approach so that

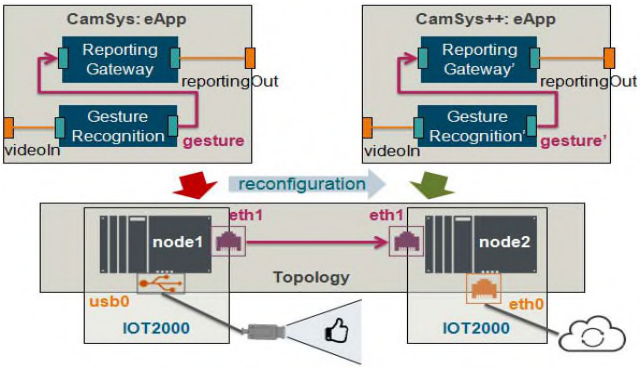


Figure 9. Reconfiguration from *CamSys* to *CamSys++*: Logically the two software components are replaced with their newer versions which have an updated interface gesture'. An appropriate rcPlan has to specify in detail how and when to change the Component Mapping, the Communication Mapping and the I/O Mapping to reconfigure this application during operation without stuttering.

no gesture is missed by the distributed application. For the required update the agents first have to download and start the new components in background. Then they have to synchronize using a two-way handshake to rewire the new components in a timely coordinated manner. Finally, the old components need to be stopped in background.

To let the agents do so we propose the reconfiguration plan given in Table III, which is composed of one reconfiguration plan for each node. The most important application-specific reconfiguration aspect is the timing of the real-time steps: As the gesture information reaches the reporting gateway component with two cycles delay and thus still has the old API version used in the old gesture recognition component, the two components are replaced timely shifted along the data flow by appropriate cycle offsets as shown in Figure 10.

Given synchronization point two is reached in cycle $c_0$. Up to this cycle, the new components have been downloaded and started in background followed by the two-way handshake. Then both agents perform the real-time steps in the associated cycle offset relative to $c_0$. On node1 the first step is performed directly in $c_0$ as the step's cycle offset is zero. The agent applies the step rmIOMapping between IO processing. Therefore, in the subsequent input processing phase, the video input is not injected to the old gesture component anymore as it was in the red data flow of the old application version. Instead, the usb0 is injected to videoIn of the new gesture component version due to the subsequent real-time step addIOMapping – which is still performed in the same reconfiguration hook. Performing the eight rtSteps for node1 the agent completely takes out the old component and puts the new version in place.

On node2, the first real-time step is performed with one cycle offset after input processing, because the old-formatted message still needs to be routed to the old gateway version. The agent prepares the system so that the expected messages from the new version will not be enqueued in the old container's barrier qdisc, but in the new one. In the cycle

with offset two, the agent performs the final switches to the new gateway component. The resulting new data flow of the updated application is highlighted in green. Both agents trigger the stopping of the old versions in background in the cycle turnover after their last real-time steps, which completes the reconfiguration.

## VI. Evaluation

We evaluate our approach for synchronous reconfiguration during operation. The evaluation is based on our prototype of the real-time container architecture, which is extended to perform modifications during operation as described in Section V. The results shown here are gathered using the onBtnSwitch system described in [2], which is almost equal to the gesture recognition system from a runtime environment perspective: The mapping and the resulting runtime behavior are equal, so the agent has to do exactly the same steps to run and reconfigure the application.

The two components are installed in two (identical) versions on the respective nodes in pre-configured lxc containers. Both nodes are SIMATIC IOT 2040 microcontrollers with a custom Yocto image installed on industrial-grade SD cards. The real-time container agent is implemented in C/C++ and installed on each node. The agents are configured for running onBtnSwitch 1.0 via corresponding yaml files. Besides executing the distributed embedded application, the agents evolve the application to onBtnSwitch 2.0 using the reconfiguration plan described in Table III, with a few extensions as follows. To gather runtime statistics of reconfiguration executions, the agent additionally applies the same steps to roll back to the old version and restarts the reconfiguration plan. We do not include the download step in this evaluation, but use pre-installed images for all components. We present the results of this experiment in this section.

### A. Reconfiguration Success

First of all we address the concern: Can the proposed approach successfully reconfigure the distributed system during operation? Indeed, reconfiguration can be performed as described without breaking the running application: The synchronization protocol works, so the two nodes always manage to agree on a continuation cycle using the two-way handshake when the cycles are aligned. Additionally, the proposed real-time steps are sufficient in so far as in this evaluation system the components can be exchanged and rewired during operation so that each message is processed by the corresponding version. And finally, the background approach leads to stronger isolation of the components as shown in Figure 11: Even when starting the CPU-affine workload `cat /dev/zero > /dev/null` the regular components are not interfered during startup nor during execution nor during shutdown.

While the reconfiguration works, we point out a few non-functional aspects, which may lead to problems in other use cases. Using the synchronization protocol a two-way handshake between two nodes costs four cycles of time in our setup. In other setups with more nodes and hops, the delay increases and also requires more synchronization messages. Thus without multi-cast synchronization it seems unlikely that this approach scales with regard to the network topology. Depending on the configured quota for the background operations it takes indirectly proportional more time to start a container than usual. With no limit lxc-start takes approximately $200\,\mathrm{ms}$; with $10\,\%$ background quota it takes approximately two seconds; with $5\,\%$ approximately four seconds and so forth. Consequently, for use cases that require fast adaptations, a high percentage of over-provisioning must be carried out to enable a sufficiently high background quota.

### B. Reconfiguration Runtime Overhead

We measured the runtime of the agent and the real-time steps performed (see Table IV and Table V). The concept assumes zero execution time and compensates this assumption with an increased agent reservation. This is reasonable for the real-time steps which do not need to change the network configuration, i.e., for addSwc, rmSwc, addIOMapping and rmIOMapping. These steps take a fraction of a millisecond and thus should be fast enough for many cases. However, as the measurements of our prototype show, the networking-related real-time steps have too high execution times: addComMapping, rmComMapping, addMsg and rmMsg take at least around $40\,\mathrm{ms}$ and can take up to over $50\,\mathrm{ms}$. Our prototype even has problems running agent phases not involved in reconfiguration. While the health check and the triggering are limited to below $2\,\mathrm{ms}$, output processing and input processing phases can take up to $80\,\mathrm{ms}$. Thus the originally demanded period of $50\,\mathrm{ms}$ for onBtnSwitch cannot be met by our prototype – to make the reconfiguration work reliably, we had to increase the period to $500\,\mathrm{ms}$ and the agent reservation to $300\,\mathrm{ms}$.

These high execution times can be explained with four reasons: First, the sophisticated Netfilter firewall feature of the Linux kernel consumes much time for modifying a qdisc and the firewall rule set. Second, there does not exist an official library or Netlink documentation for programmatic firewall configuration from user space, so our prototype has to use the indirection of the command line interface (including argument parsing etc.). Third, there were still background processes outside of the application, which were not moved to the background cgroup and thus demanding time. Forth, we have observed that the peaks in input and output processing phases are caused by interleaved qdisc modifications, which interfere with each other even though they target different devices.

Thus, we consider the runtime overhead of the real-time steps and of the other peaks to be an issue with the underlying operating system and the employed frameworks. Although the background processes have to be further isolated and the real-time steps purified by moving them closer to the kernel, these first results still look promising.

## VII. Conclusion

We proposed a concept to reconfigure distributed embedded applications during operation based on a real-time container

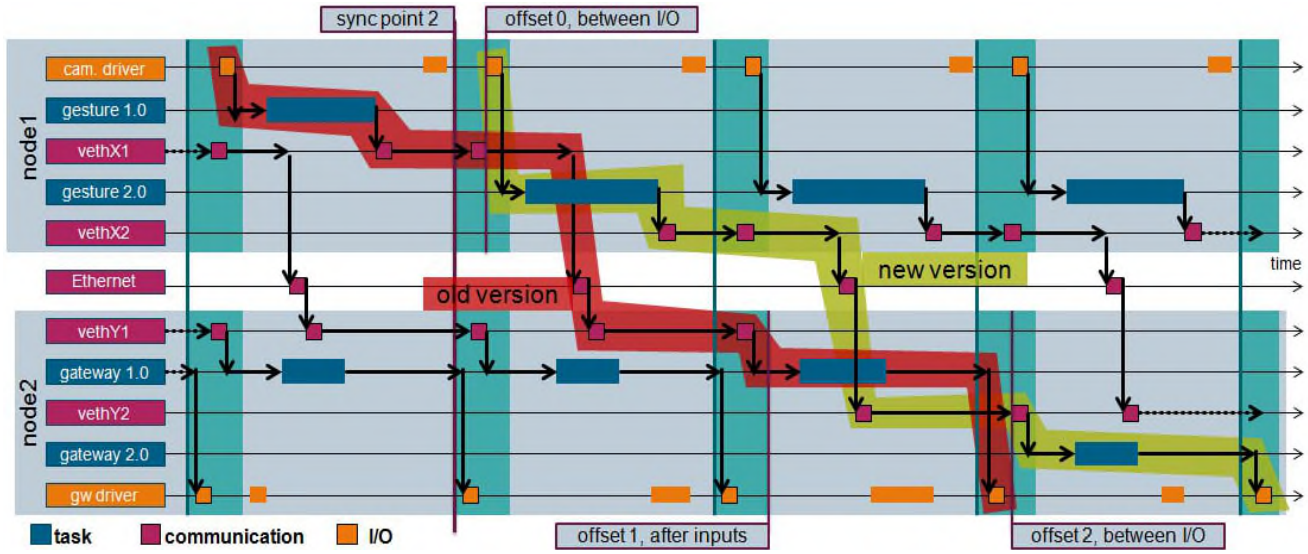


Figure 10. A schematic timing diagram of a partial execution of the proposed reconfiguration plan for the *CamSys*: After synchronizing, the gesture recognition component is updated one cycle earlier than the gateway component to cope with an API change.

| **rcPlan node1** | **rcPlan node2** |
|---|---|
| download(gestureCtrl, v2.0)<br>start(gestureCtrl, v2.0) | download(gatewayCtrl, v2.0)<br>start(gatewayCtrl, v2.0) |
| notifyAndWait(node2, syncPoint(1))<br>waitForSync(syncPoint(2)) | waitForSync(syncPoint(1))<br>notifyAndWait(node1, syncPoint(2)) |
| rmIOMapping(videoIn, usb0) – offset 0, betweenIO<br>rmComMapping(gestureOut, msg1) – offset 0, betweenIO<br>rmMsg(msg1) – offset 0, betweenIO<br>rmSwc(gesture 1.0) – offset 0, betweenIO<br>addSwc(gesture 2.0) – offset 0, betweenIO<br>addMsg(msg1', clsid(1:1), rate(10kbps)) – offset 0, betweenIO<br>addComMapping(gestureOut, msg1', src(gesture-v2-0:2000),<br>to-dst(node2:2000)) – offset 0, betweenIO<br>addIOMapping(videoIn, usb0, camDriver) – offset 0, betweenIO | rmComMapping(reportingEvents, msg1) – offset 1, afterInputs<br>rmMsg(msg1) – offset 1, afterInputs<br>addMsg(msg1', src(node1:2000), rate(10kbps)) – offset 1, afterInputs<br>addComMapping(reportingEvents, msg1', src(node1:2000),<br>to-dst(reporting-v2-0:2000)) – offset 1, afterInputs<br>rmIOMapping(reportingOut, eth0) – offset 2, betweenIO<br>rmSwc(gateway 2.0) – offset 2, betweenIO<br>addSwc(gateway 2.0) – offset 2, betweenIO<br>addIOMapping(reportingOut, eth0, gwpDriver) – offset 2, betweenIO |
| bgStep(stop(gestureCtrl, v1.0)) | bgStep(stop(gatewayCtrl, v1.0)) |

Table III
THE RECONFIGURATION OF *CamSys* AS RCSTEPS FOR BOTH NODES. HORIZONTAL LINES INDICATE THE IMAGINARY SEPARATION INTO THE STEPS PREPARE, SYNCHRONIZE, RECONFIGURE, CLEAN. THE IMPORTANT TRICK DUE TO INCOMPATIBILITY OF THE NEW GESTURE MESSAGE: THE GATEWAY COMPONENT IS REPLACED LATER, SO THAT THE LAST GESTURE MESSAGE FROM THE OLD GESTURE COMPONENT IS PROCESSED BY THE OLD VERSION.

architecture. The concept employs a reconfiguration plan consisting of three kinds of steps: background steps to download and start new containers in background, synchronization steps to agree on a continuation cycle, and real-time steps to perform critical modifications of the running system in a timely coordinated manner. We elaborated on conceptual extensions of the real-time container architecture needed to execute reconfigurations accordingly, e.g. the construction container, the synchronization protocol and real-time step timing. The reconfiguration steps are implemented by the agent using features of Linux and lxc, but also custom inter-process communication means between agent components on one node and inter-agent communication across nodes. The evaluation shows that the reconfiguration concept works in principle, but some agent steps take more time than desired. Therefore, development efforts towards more direct interaction with the kernel is needed to achieve periods below 100 ms, which is not only required by the onBtnSwitch system.

Besides that, further research is required towards scalability of the synchronization protocol with the network topology. The recent improvements in the field of software-defined networking should be considered in the future. Another aspect is state transfer between the old and the new component version for time-consuming cases, i.e., when the state is large, has

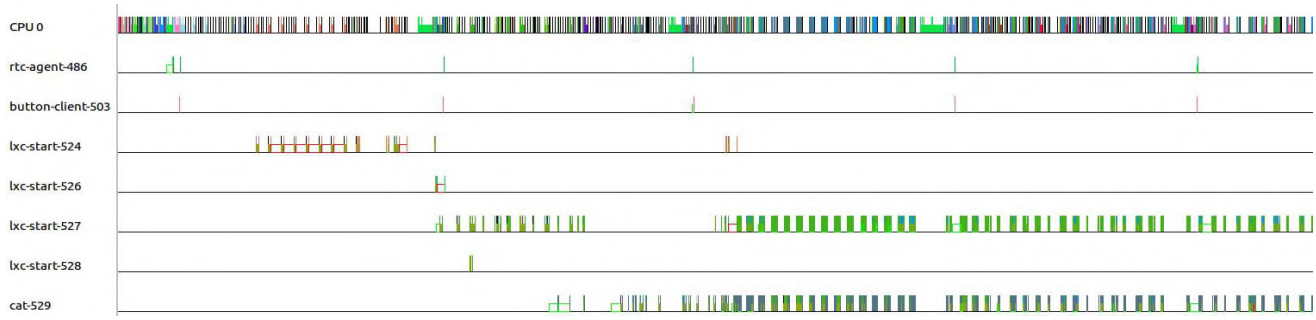


Figure 11. A trace of starting the cat component in background: while the agent and the *ButtonController* component keep their period a new container is ramped up.

| agent phase | min | max | avg |
|---|---|---|---|
| health check | 0.097 | 0.709 | 0.372 |
| before outputs | 0.452 | 14.487 | 0.544 |
| output processing | 0.083 | 78.081 | 8.438 |
| between I/O | 0.411 | 191.282 | 4.594 |
| input processing | 0.484 | 23.209 | 10.369 |
| after inputs | 0.219 | 13.839 | 1.025 |
| trigger | 0.031 | 1.242 | 0.661 |
| agent overall | 3.552 | 213.512 | 30.459 |

Table IV
THE NON-FUNCTIONAL BEHAVIOR OF THE REAL-TIME CONTAINER AGENT W.R.T. ITS EXECUTION TIME PER CYCLE IN MILLISECONDS.

| rtStep | min | max | avg |
|---|---|---|---|
| addSwc | 0.050 | 0.069 | 0.052 |
| rmSwc | 0.049 | 0.075 | 0.062 |
| addComMapping | 45.936 | 47.388 | 46.332 |
| rmComMapping | 49.927 | 56.512 | 51.215 |
| addMsg | 40.809 | 42.215 | 41.461 |
| rmMsg | 41.491 | 42.888 | 41.993 |
| addIOMapping | 0.050 | 0.077 | 0.063 |
| rmIOMapping | 0.050 | 0.266 | 0.088 |

Table V
THE EXECUTION TIMES OF THE REAL-TIME STEPS IN MILLISECONDS.

to be transformed, or when network transmission is required. In general, if the complexity of a reconfiguration increases, the question arises, how to come up with a reconfiguration plan – can it be derived automatically from simpler primitives? Additionally, the current real-time container architecture does not cover fault handling beyond stopping the application: *To make reconfiguration feasible in practice it must be possible to permit temporary quality degradation* [1]. All in all, we are confident that the concepts presented in this paper are useful and build a good basis for future work in various directions.

## REFERENCES


[1] K. Telschig and A. Knapp, "Towards safe dynamic updates of distributed embedded applications in factory automation," in *22nd IEEE Intl. Conf. Emerging Technologies and Factory Automation*, 2017.

[2] K. Telschig, A. Schönberger, and A. Knapp, "A Real-Time Container Architecture for Dependable Distributed Embedded Applications," in *CASE*, 2018.

[3] M. Hammer, "How to touch a running system: Reconfiguration of stateful components," Ph.D. dissertation, Ludwig-Maximilians-Universität München, 2009.

[4] 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.

[5] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 856–868, Dec. 2007.

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

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

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

[9] C. M. Kirsch, L. Lopes, E. R. Marques, and A. Sokolova, "Runtime programming through model-preserving, scalable runtime patches," in *Application of Concurrency to System Design (ACSD), 2011 11th International Conference on*. IEEE, 2011, pp. 77–86.

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

[11] T. Terzimehić, "Optimization and reconfiguration of iec 61499-based software architectures," in *ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS) - Doctoral Symposium*, Oct. 2018.

[12] L. Prenzel and J. Provost, "Dynamic software updating of iec 61499 implementation using erlang runtime system," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 12 416–12 421, 2017.

[13] A. Lenz and R. Obermaisser, "Global adaptation controlled by an interactive consistency protocol," *Journal of Low Power Electronics and Applications*, vol. 7, no. 2, p. 13, 2017.

[14] T. Kothmayr, A. Kemper, A. Scholz, and J. Heuer, "Schedule-based Service Choreographies for Real-Time Control Loops," in *ETFA*, 2015.

[15] J. Chaplin, O. Bakker, L. de Silva, D. Sanderson, E. Kelly, B. Logan, and S. Ratchev, "Evolvable assembly systems: A distributed architecture for intelligent manufacturing," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 2065–2070, 2015.

[16] AUTOSAR, "AUTOSAR - Technical Overview," online - last visited 26 Feb 2018, 2015.

[17] Canonical Ltd., "LXC," 2018, Internet: https://linuxcontainers.org/lxc/ [May 23, 2018].

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

[19] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, 2003.