The Journal of Systems and Software 211 (2024) 112004

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems ☆

Heiko Koziolek [a,*], Andreas Burger [a], Abdulla Puthan Peedikayil [b]

[a] ABB Research, Wallstadter Str. 59, Ladenburg, D-68526, Germany
[b] ABB Research, ITPL Main Rd, Bangalore, 560048, India

ARTICLE INFO

Keywords:
Software architecture
Control systems
Software containers
Dynamic software updates
Docker
Kubernetes

ABSTRACT

Many industrial processes (e.g., steel, fertilizer, paper production) utilize programmable logic controllers or distributed control systems, which cyclically execute control applications. Dynamic updates of such applications at runtime are desirable for optimizing the production while avoiding production stops. However, such updates are challenging since they require fast switching to an updated instance and executing an internal state transfer within the control cycle times. Existing dynamic update approaches for industrial applications can for example not update controller runtimes or operating systems, since they do not utilize modern container virtualization technologies. Furthermore, existing state transfer methods for container orchestration systems do not support fast cyclic applications. We propose a novel state transfer method for industrial control applications deployed as containerized Kubernetes microservices on server hosts or resource-rich industrial personal computers. The method enables dynamic updates and live migration of applications to others hosts without interrupting the underlying production facilities, which avoids costly downtimes. We have implemented our method as a Kubernetes extension and found that it can transfer large application states (i.e. 100.000 variables) via OPC UA in less than 42 ms across different hosts, which can fit into the available cycle slack time, therefore enabling a wide range of bumpless updates at runtime.

1. Introduction

Industrial control applications deployed in programmable logic controllers (PLC) or distributed control systems (DCS) automate chemical refinement, power production, or mining (Hollender, 2010). Automation engineers often program such applications in one of the five IEC 61131–3 programming languages (Tiegelkamp and John, 2010). They classically deploy these applications to embedded real-time controllers, where they cyclically execute to process sensor values and control actuators. Software container virtualization could make such applications more portable, flexible, and updatable, but are not yet widely utilized (Forbes, 2018).

Automation customers desire frequent updates to control applications, for example to fine-tune the automation (e.g., tuning parameters of a PID loop (Hollender, 2010)), to address changing customer needs, or to apply security fixes (Forbes, 2018). However, today's embedded systems often require a shutdown for updates and thus expensive production stop to apply updates (Hollender, 2010). Production downtimes prevent plant owners from utilizing their costly production equipment. Container technology induces a cloud-native approach towards software updating, where updated applications inside containers replace running application instances on-the-fly (Burns et al., 2019). Control applications are stateful and must adhere to short execution cycles (e.g., 100 ms, see Section 6.1).

Researchers have devised methods for dynamic program updates already for decades (Gupta et al., 1996; Hicks and Nettles, 2005; Baresi et al., 2016; Ahmed et al., 2020), but rarely addressed cyclic control applications, nor considered container orchestration systems. While several researchers validated the feasibility of running control applications in software containers (Moga et al., 2016; Goldschmidt et al., 2018; Sollfrank et al., 2020) without inducing undesirable jitter, the existing dynamic updates methods for real-time applications (e.g., Wahler et al. (2009), Wahler and Oriol (2014), Prenzel and Provost (2017)) still assume deployment in dedicated embedded systems and thus only updates within a single node. This for example does not enable operating system updates or hardware modification by moving an application to a different node.

☆ Editor: Uwe Zdun.

* Corresponding author.
E-mail address: heiko.koziolek@de.abb.com (H. Koziolek).

https://doi.org/10.1016/j.jss.2024.112004
Received 21 December 2022; Received in revised form 28 December 2023; Accepted 12 February 2024
Available online 16 February 2024

Furthermore, special state transfer methods for container orchestration systems (Netto et al., 2017; Vayghan et al., 2019; Oh and Kim, 2018) do not support cyclic applications, nor safe application updates, and target less challenging web applications. Existing methods for container live-migrations (e.g., Ma et al. (2019), Machen et al. (2017), Yu and Huan (2015), Qiu et al. (2017) work with comparably large container checkpoints unsuited for a fast state transfer within the cycle times in industrial automation.

The contribution of this paper is a novel state transfer method for updating industrial control applications deployed as microservices in a container orchestration system. The method exploits characteristics of cyclic applications to facilitate the state transfer, but unlike previous approaches for embedded systems enables live-migration to separate host nodes. In comparison to existing container live-migration approaches, it only requires significantly smaller *application* states instead of entire *container* states. The method allows state transfers both via a network protocol and alternatively via local shared memory. Compared to other methods, it utilizes special features of IEC 61131–3 (predictable cycle slack time), OPC UA (Open Platform Communications Unified Architecture) (Mahnke et al., 2009) Pub/Sub communication (decoupled signal senders and receivers), container orchestration (custom Kubernetes Operator for taking memory-saving internal state snapshots) to support a wide range of updates.

To validate our method, we have implemented the state transfer method in a Kubernetes Operator and tested it on a commercial ABB runtime as well as an open-source runtime, both deployed as Docker containers and connected to simulated sensor/actuators devices. The experiment setup was derived from actual large chemical plants (Krause, 2007). Upon an update request, our Kubernetes Operator serializes the internal application state and sends it to an updated control application container. We found that we can transfer the internal state of large control applications (i.e., 100.000 variables) via the network without losing execution cycles, thus leading to bumpless dynamic updates.

This paper's structure is as follows: Section 2 introduces the background on the domain and technologies the method is rooted in. Section 3 provides a minimal running example to illustrate the concepts and constraints in the respective application domain. Section 4 introduces our method in form of a static reference architecture and a state transfer algorithm. Section 5 describes the prototypical implementation and the testbed used for experimentation. Section 6 validates the method with a series of experiments on multiple deployment platforms and discusses threats to validity. Section 7 surveys related work and finally, Section 8 concludes the paper.

2. Background

2.1. Domain-specific technologies

Our approach is designed for distributed control systems (Hollender, 2010) (DCS), supports dynamic updates specifically for the IEC 61131–3 programming model (Section 2.1.1), and utilizes OPC UA for networking between different nodes (Section 2.1.2).

2.1.1. IEC 61131-3 programming

IEC 61131-3 is an international standard dealing with the basic software architecture and programming languages for PLCs. Automation engineers often write the control logic running on controllers in one of the **61131-3 programming languages** or comparable variants (Tiegelkamp and John, 2010). The standard specifies the syntax and semantics of five functionally equivalent languages: Sequential Function Charts (SFC), Function Block Diagrams (FBD), Ladder Diagrams (LD), Structured Text (ST), and Instruction Lists (IL). The syntax of ST resembles the syntax of the Pascal programming language. For example, ST is better suited to express mathematical operations, while SFCs provide a convenient way of specifying control sequences.

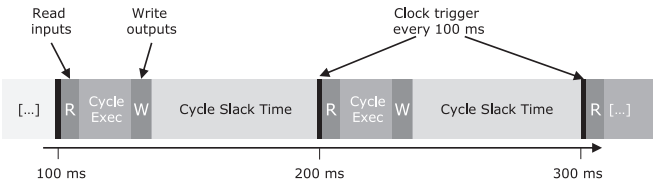

Fig. 1. IEC 61131-3 Cyclic Execution: the cycle slack time waiting for the next cycle is predictable.

IEC 61131-3 programs consist of multiple tasks , each having a defined cycle execution time . Tasks are again structured in programs, which in turn can contain several function blocks. Many pre-defined function blocks cover for example mathematical operations, timers, or proportional–integral–derivative (PID) controllers (Levine, 2018). Automation engineers can readily connect them in sequences, parametrize them, and wire them with input and output signals.

The IEC 61131-3 programming model is intentionally simple compared to other programming languages (Tiegelkamp and John, 2010):

- It does not support threading (beyond the basic cyclicly executing task model) or side effects avoiding many concurrency issues.
- Each data type has a defined initial value, either by default or specified by the user. This is important for our approach since variables added during an update are always correctly initialized.
- Configurations and resources can be started and stopped via IEC 61131-1 defined functions. Starting a resource causes the initialization of all variables and an enabling of all tasks. Stopping of a resource causes the disabling of all tasks. This mechanism is used in our dynamic update mechanism to pause the execution of the control application.

An IEC 61131-3 runtime traditionally runs on the highest priority level on a real-time operating system (e.g., VxWorks, QNX). A recent trend are mixed-criticality systems, where such runtimes may be combined with other functionalities of different safety and/or time-criticality (Cinque et al., 2022). IEC 61131-3 tasks within the runtime are scheduled periodically by timer interrupts (Fig. 1). First, the runtime reads all inputs for the function blocks, scanning all I/O devices (e.g., sensors). Then, all tasks from highest to lowest priority are executed with the included control logic to compute the output values, which are then written as outputs to the I/O devices (e.g., actuators). After writing the outputs, the runtime scheduler waits for the next timer interrupt. This period of time is called "cycle slack time". Other, less time-critical software, such as web servers, may execute in the cycle slack time in parallel to runtime, but will be preempted by the operating system, when the next timer interrupt triggers the runtime scheduler. On a multi-core node, such less time-critical software can also be executed by other cores.

The occurrence of an application's cycle slack time is predictable, since the control cyclic control systems are stable as the system structure rarely changes (Wahler et al., 2009) and there is no dynamic scheduling for the control application tasks. The cycle slack time of an IEC 61131-3 runtime can be detected by other processes or even explicitly signaled by the runtime itself via notification mechanisms and be utilized to trigger update procedures. During the cycle slack time, the function blocks' internal state is quiescent, i.e. it does not change, since no algorithms execute on it.

2.1.2. OPC UA middleware

While I/O signals traditionally arrived at controllers via analog connections, fieldbuses or industrial Ethernet, supervision components used the OPC client/server protocol running on top of TCP/IP for monitoring and issuing operator commands. Our approach is built around its successor, the **OPC UA** standard (IEC 62541) for data exchange (Foundation, 2017; Koziolek et al., 2019). The OPC foundation has recently

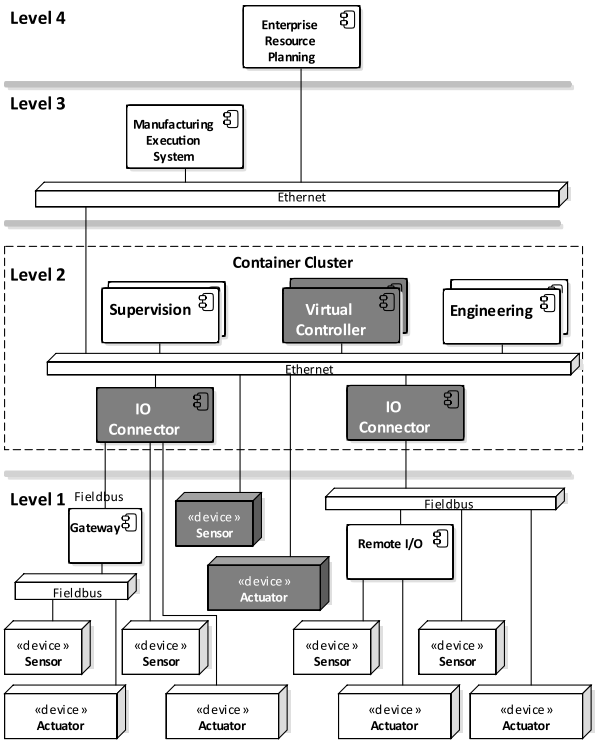

Fig. 2. Revised reference architecture: virtual, containerized controllers hosted on servers, directly connected sensors and actuators, and IO connectors for existing devices. All software on level 2 can be hosted in an on-premise cluster.

extended the standard with a pub/sub mechanism[1] allowing fast, cyclic communication between I/O devices and controllers. This affects the traditional ANSI/ISA95 architecture[2] as shown in Fig. 2. Native OPC UA sensors and actuators can be directly connected to the Level 2 network that both serves controllers and supervision, possibly employing Time-sensitive Networking (TSN) (Finn, 2018). As existing cabling and gateways may continue to get used for a long time, traditional connections and fieldbuses are integrated via "IO connectors", which translate the I/O signal data into OPC UA pub/sub messages. OPC UA servers built into control runtimes provide an object-oriented address space that may expose the internal state of control applications. This is used by our state transfer method.

2.2. Virtualization

Furthermore, automation customers are promoting **software container** technologies into distributed control systems (Forbes, 2018). Software containers are lightweight, stand-alone, and executable software packages, based on resource isolation features of the Linux kernel, i.e., cgroups and kernel namespaces. In many situations even time-critical, deterministic control execution runtimes can be packaged into containers and hosted on servers running real-time enabled Linux operating systems, as former research has demonstrated that the added jitter from containerization is negligible in many practical cases (Moga et al., 2016; Goldschmidt et al., 2018; Sollfrank et al., 2020).

Industry experts on industrial automation have elaborated (Forbes, 2018) that using a container orchestration system, such as Kubernetes, can be beneficial for control software to simplify system management and self-healing. The basic scheduling unit in Kubernetes is called 'pod'

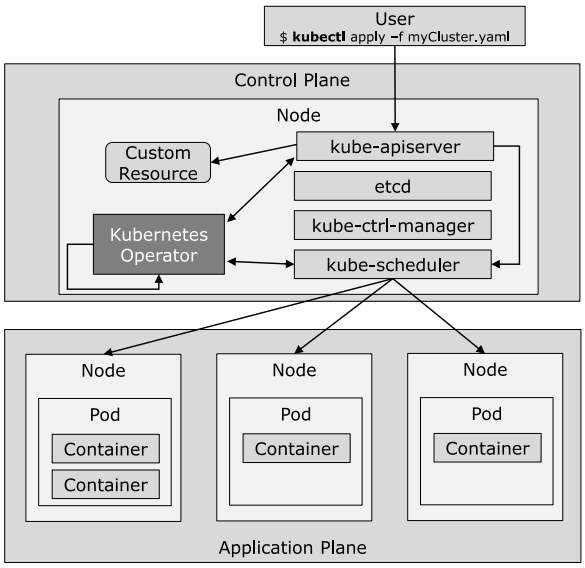

Fig. 3. Kubernetes operator.

and usually contains a single software container (Fig. 3). Kubernetes users can deploy multiple pods for the same application in the same cluster on different nodes, so that Kubernetes automatically provides load balancing and self-healing (i.e., restarting failed pods).

Fig. 3 shows that the kube-apiserver manages all Kubernetes resource configurations which are stored in a clustered key–value store called etcd. The kube-ctrl-manager compares the user-desired state of the cluster with the actual state and instructs the kube-scheduler to create new pod instances if needed. The kube-scheduler matches a pod's configured resource requirements (e.g., CPU load, memory) against the available nodes and autonomously decides to schedule new pods on appropriate nodes.

Kubernetes users usually deploy resources (e.g., pods, services) or change resource configurations manually using the *kubectl* command line tool or web-based K8s dashboards, which send REST queries to the central kube-apiserver in the Kubernetes Control Plane. Kubernetes foresees custom extensions following the operator pattern[3] (Fig. 3). A Kubernetes *Operator* extends the functionality of the Kubernetes API to manage applications with special orchestration requirements (Dobies and Wood, 2020). It runs as a separate pod in Kubernetes and controls custom-defined Kubernetes resources (i.e., specialized pods, for example a Virtual PLC runtime) with developer-defined interfaces and configuration parameters. A Kubernetes operator is a continuously running process and can take autonomous actions without user involvement.

At runtime, a Kubernetes operator monitors specific custom-defined Kubernetes resources and takes application-specific actions, such as scaling, upgrading, or utilizing specialized hardware. Our method utilizes this mechanism for control application updates.

3. Running example

Many industrial automation applications use a control loop mechanism as in our *running example* (Fig. 4, left-hand side). A heat exchanger warms up a process medium (e.g., a liquid chemical) for a specific reaction. It provides process medium input and output, as well as an heating medium input and output. The temperature transmitter (TT202) monitors the temperature in a pipe and sends a signal to a temperature controller (TIC201), which cyclically runs a control

[1] https://reference.opcfoundation.org/Core/docs/Part14/
[2] http://www.isa-95.com/
[3] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

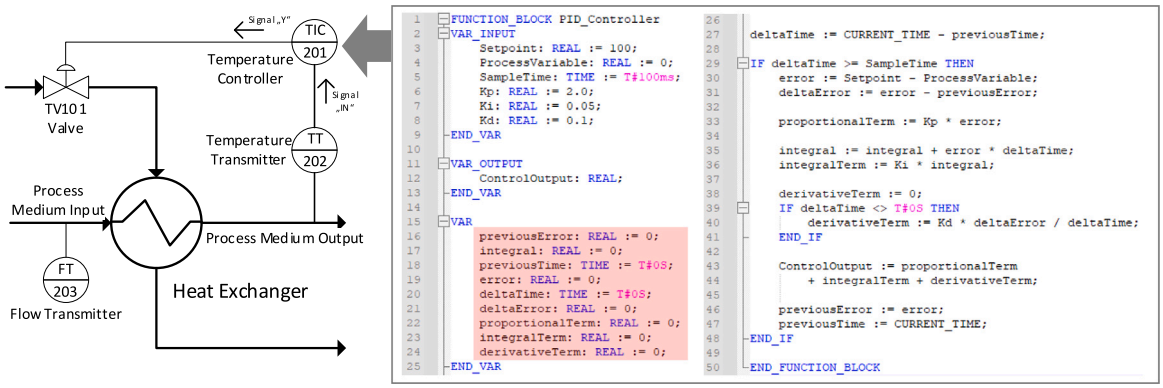

Fig. 4. Left-hand side: Piping and Instrumentation Diagram (P&ID) of a typical Heat Exchanger used in many process plants. Right-hand side: The temperature controller TIC201 uses the depicted algorithm to control the temperature valve TV101 and regulate the heat of the process medium. The algorithm's internal state is highlighted in line 16–24.

algorithm. This algorithm computes values for opening or closing a valve (TV101, e.g., 60 percent opening), so that the temperature of the process medium can be regulated to a pre-defined set point (e.g., 70 degrees Celsius).

The temperature controller runs a proportional–integral control (PID) algorithm (Fig. 4, right-hand side). The algorithm is depicted in IEC 61131-3 ST. It computes the next output in each cycle based on the current inputs and its *internal state*. Specifically, in line 43 it computes the output signal for TV101 by adding the proportional, integral, and derivative terms. The integral term is calculated based on the integral value from the previous control cycle, and the derivative term is calculated based on the error from the previous cycle. Thus, both of these values depend on the PID controllers internal state based on previous executions.

Assuming that a control engineer came up with an updated PID algorithm that checks with the computed output is within certain limits, the goal is now to update the application. The update requires transferring the internal state variables to the new deployment, so it can produce the correct control outputs considering the history of execution. As the cyclic execution (e.g., every 100 ms) shall not be interrupted to avoid the heat exchanger running uncontrolled, the state transfer needs to execute within the cycle slack time (e.g., 90 ms).

As more and more control functions are deployed on powerful server hosts or industrial-grade personal computers (Industrial PCs), then denoted as "*Virtual PLCs*". There are vast computing resources for parallel executions. Cloud-native updates of application software replaces patching software with deploying updated instances in parallel and gradually switching new client requests over to these instances, which avoids downtimes.

4. State transfer method and architecture

4.1. Overview

We propose a novel dynamic updated method for industrial automation applications as in Fig. 4. To allow for robust, bumpless updates the method starts an updated application in parallel to original application, transfers the internal state of the original application to the updated application, and then allows an algorithm or human user to validate whether the update was successful by comparing the outputs of the original and updated application. Only after successful validation, the method hands over control to the updated application and deactivates the original application.

The method utilizes the IEC 61131-3 programming model, which, via the cycle slack time, provides time periods when the internal state will not be changed. The method is built around container technology to extend the scope of possible dynamic updates from application updates to library, operating system, and even hardware updates. These kind of updates are enabled since the method uses a network protocol to transfer the internal state and thus can start the application on a separate updated host node.

The method supports the following kinds of updates:

- Add, modify, and remove externally supplied variables: referring to the running example, this would affect the VAR_INPUT and VAR_OUTPUT sections in line 2–13. For example, a control engineer could add an input variable so that the operator can set a limiting value to the control output in the user interface. For an update, added variables would be initialized with a mandatory default value. For these externally supplied variables no state transfer is required, since they are read in each control cycle either from the I/O devices or the graphical user interface.
- Add, modify, and remove local variables: these would affect the VAR section in line 15–25 of the running example. For example, a control engineer could decide to add or remove the derivative part of the PID controller to account for measurement noise. Added variables would need to be initialized with default values, for modified variables the values of the original application would be overwritten. Renaming variables is possible, but such renamed values will be treated as new variables and initialized with default values, as our approach does not perform any code analysis that could detect renaming. Modifying the values of local variables may lead to deviating output values between the original and updated version, so it must be checked whether this deviation is within acceptable thresholds given the application at hand.
- Add, modify, and remove program code: these changes would affect the main program code, in the running example in line 27–48. For example, a control engineer could add an if/then/else clause to check whether the computed control output is within specified limits and issue an alarm otherwise. Again this may intently change the computed output values and thus needs careful validation.
- Add, modify, and remove function blocks: these changes would again affect the main program code, where separately specified function blocks are used. For example, a control engineer may add a timer function block to introduce a time delay before writing the control output. Changes to commonly used function blocks in many programs can however lead to situations where an unfeasible amount of outputs would need to be validated.
- Add, modify, and remove control libraries: these changes could affect a number of function blocks bundled in a control library. For example, a control engineer could decide to upgrade a referenced control library to a newly released version. Again in this case, a validation of the change may be difficult.
- Modify control runtime: such a change may affect the control runtime itself, which may be updated to a newer version to

remove bugs or address security issues. In a containerized system, a new container image would be started to then run an identical version of the control application. In most cases, such an update would not affect the output values of the control functions and is therefore comparably simple to validate.
- Modify software libraries: the control runtime may use external libraries, e.g., for encryption, XML parsing, or OPC UA communication. Updates to these libraries would again require building a new container image, but often not affect the control application functionality. Updating these software libraries is often desirable and usually does not affect the internal state of the application, but rather affects the application execution engine itself.
- Modify operating system: an operating system on a physical host or a virtual machine may be updated to add new features or remove security vulnerabilities. For example, a control engineer may decide to migrate an entire host node to a newly released OS version. In this case, the node would first be prepared with the new OS, then integrated into the container orchestration, and then our approach would be used to "move" the containers to this new host.
- Modify hardware: updates to Industry PC or server hardware may be required to avoid hardware failures. In this case our proposed method would allow to move the PLC software to a new host node.

The method does not allow changes to the application task cycle times, which would complicate the output validation. The method requires an appropriate amount of cycle slack time to able to perform the state transfer, which will be further characterized in the experimental evaluation. If the update involves two separate host nodes, the method assumes that both nodes are clock synchronized using the Precision Time Protocol (PTP, IEEE 1588). The targeted application cases do not require a lockstep execution of the old and updated applications, since the underlying industrial processes can tolerate small deviations of the clock cycles, although not missing an entire control cycle.

While the method can support many different kinds of changes, the extent of changes to the application functionality is in practice constrained by the ability to validate the intended updated functionality. A human operator either needs to compare the behavior of the original application and the updated application manually by observing the live updating values of both applications running in parallel or needs to encode the desired behavior into thresholds for our state transfer algorithm to automatically make the comparison over a pre-specified period of time. For example, the comparison could yield that the values of the PID controller output after the update would be out of the desirable ranges, which would then lead to an abort of the update procedure.

Large control applications may have thousands of output values to control many connected I/O devices. If a change would affect the values of for example more than 10 output values, then the output validation may get impractical, since too many dynamically changing outputs would need to be compared. Therefore, regarding application updates the method is intended for fine-tuning changes to individual variables or function blocks, while larger algorithmic changes that affect more than 10 output values should still go through offline engineering and testing procedures. However, in practice most changes intended by control engineers during runtime are in fact small and can often be isolated in specific function blocks. In addition, this limitation does not constrain the methods ability to update libraries, OS, and hardware in case these do not affect the application output.

The remainder of this section first describes our method's component architecture to execute the state transfer (Fig. 5) and then the state transfer algorithm itself (Alg. 1).

4.2. Component view

A core idea of our method is to deploy PLC runtimes within software containers (Sollfrank et al., 2020) and introducing a custom-built Kubernetes Operator (Dobies and Wood, 2020) to manage the state transfer between original and updated containers. The remote state transfer of our method utilizes the OPC UA network protocol (Mahnke et al., 2009), while a local "fast-mode" utilizes interprocess communication (IPC) via POSIX (Portable Operating System Interface) shared memory on a single node. The former method covers most automation applications and allows switching nodes during updates, while the latter method can additionally be used to address the small percentage of very short-cycled control applications (e.g., <50 ms) in process automation. The "fast-mode" does not allow to move a PLC application between host nodes.

Fig. 5 shows the reference architecture of our approach with the required components. The figure depicts only two worker nodes for brevity, whereas a real cluster would contain many worker nodes. The figure does not show the connection of the cluster to sensors and actuators via IO connectors or native OPC UA devices, as this was already shown in Fig. 2.

The cluster contains the hosts **Control Plane Node**, **Worker Node 1**, and **Worker Node 2**. A PLC runtime (e.g., CodeSys, TwinCAT, OpenPLC, etc.) executes IEC 61131-3 applications in a cyclic fashion using an internal task scheduler (Tiegelkamp and John, 2010). Here, PLC runtimes are embedded into software containers and managed using a Kubernetes custom resource (VirtualPlcPod) in the cluster. A custom resource is an extension of the Kubernetes API that can be created and accesses by cluster operators, just as regular Kubernetes pods. For the running example of Fig. 4, the control application for the temperature controller TIC 201 among other controllers would be hosted in a PLC runtime in a pod and managed by a VirtualPlcPod. With the custom resource, it can be assured that the PLC runtime pod runs on a real-time enabled node, e.g., a Linux PREEMPT_RT node. The containerized PLC runtime also requires an OPC UA interface and respective Kubernetes service that exposes the I/O signals used by the PLC runtime via the OPC UA pub/sub protocol.

Different Kubernetes pods can by default only communicate via network sockets or volumes but not via shared memory.[4] While it is possible to configure multiple pods to share host interprocess communication, this is strongly discouraged since it yields a security vulnerability. However, a single Kubernetes pods can include multiple containers, which can perform interprocess communication in a secure manner.

The pod managed by a VirtualPlcPod may thus optionally include *multiple* virtual PLC runtime containers to be able to utilize shared memory communication for fast state transfer (see in Worker Node 1 in Fig. 5). Having a mechanism providing shared memory communication allows loading an updated control application in one of the virtual PLC runtime containers and doing a fast, local state transfer inside the pod. Coupled with an in-place pod update strategy from OpenKruise,[5] a VirtualPlcPod can also update one of the contained, but I/O detached virtual PLC runtime containers to a newer version *without restarting the entire pod*.

As seen in Fig. 5, the **VirtualPlcOperator** is directly interfacing with the kube-apiserver and the VirtualPlcPod services. It manages the update procedure and state transfer and is hosted on the control plane node. This is a Kubernetes operator (Dobies and Wood, 2020) featuring a custom Kubernetes controller that monitors all deployed custom resources VirtualPlcPods using the Kubernetes API and detects different kind of changes:

- Control application updates, coming from engineering applications.

[4] https://kubernetes.io/docs/concepts/workloads/pods/
[5] https://github.com/openkruise/kruise

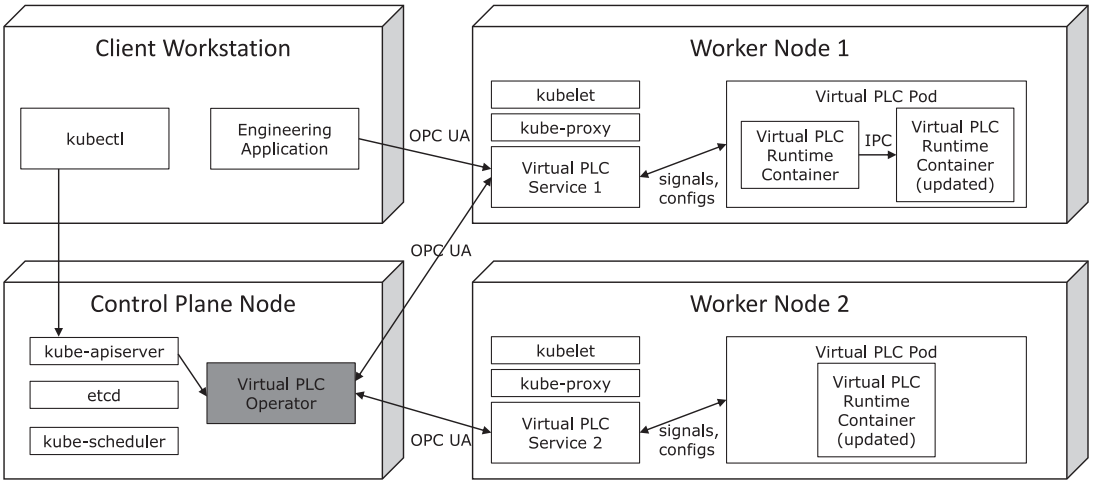

Fig. 5. Static Architecture showing our proposed VirtualPlcOperator (highlighted in dark gray color) arbitrating the state transfer between the VirtualPlcPods via the OPC UA communication protocol within a minimal Kubernetes cluster.

- Container updates, e.g., update of the PLC runtime from v2.2 to v2.3
- Infrastructure updates, e.g., node1 needs to be drained from pods for platform or hardware updates

All these changes are detectable via the Kubernetes API and need separate monitoring probes.

4.3. Dynamic view

First we describe the high-level interactions of the VirtualPlcOperator with Kubernetes (Fig. 6), before providing a detailed walkthrough of its state transfer algorithm.

As an example update scenario, after an automation engineer has updated the control logic via an engineering tool, the Kubernetes custom resource for the running VirtualPlcPod gets modified in the kube-apiserver. As the VirtualPlcOperator watches these modifications via an integrated custom Kubernetes controller (Dobies and Wood, 2020), it gets notified by the kube-apiserver to start the update procedure. The VirtualPlcOperator creates a separate VirtualPlcService in the kube-apiserver and schedules a new pod for the PLC runtime via the kube-scheduler, which in turn contacts the kubelet of another node.

After the kubelet has started the pod and created a Kubernetes Service for access, the VirtualPlcOperator connects it to the input signals by setting up OPC UA subscriptions. Due to the OPC UA pub/sub communication, the I/O devices are not bound to a particular VirtualPlcPod instance but publish and subscribe to signals on the network independent of the actual senders and receivers. Using a clock reading from the original PLC runtime container and utilizing the synchronized clocks using the PTP protocol, the VirtualPlcOperator can start the updated PLC runtime, so that their I/O scan cycles are aligned, although a lockstep execution is not required. The VirtualPlcOperator then performs the state transfer from the old VirtualPlcService to the new VirtualPlcService as described in detail using a pseudo-code algorithm in the following. Afterwards, it checks the outputs of both services for differences, and switches over the publishing of outputs to the new VirtualPlcService if the difference is within acceptable limits. This step may be supervised and approved by a human operator.

In the successful case, the I/O devices are not aware that their input signals are now published by another VirtualPlcPod and thus their controlled production process is not interrupted.

4.4. State transfer

Algorithm 1 describes the actual state transfer more formally, assuming that the source runtime $R1$ and the target runtime $R2$ are already started as separate VirtualPlcPods or separate containers in the same pod, while having the respective applications loaded. $R2$ already receives the current inputs from the sensors via equal subscriptions as the source runtime $R1$. It however computes the output values based on the internal state initialized from the default values on program startup. Therefore, the application A' computes invalid output values given the current process situation, and the system prevents the application from publishing them back to the actuators.

The VirtualPLCOperator first connects to the OPC UA endpoints (e.g., $ip1$=192.168.0.1:4840, $ip2$=192.168.0.2:4840) of both runtimes using an internal OPC UA client (line 3). The client reads from and writes into the OPC UA address spaces of both engines, which contain the current application configuration, telemetry data, as well as the internal state variables. Let A and A' be the original and updated application, while α and α' store the structure of the original and updated applications' internal state. In a first, not time-critical step (line 4), the operator extracts the state structure from the source application A into α. The structure is an ordered list of variable name and type tuples, excluding the actual variable values. The temperature control algorithm from Fig. 4 would for example yield α = [(TIC201.PID_Controller.previousError,REAL) (TIC201.PID_Controller.integral,REAL)]. As the names and types of the internal state variables do not change dynamically, this step does not require fast completion and can be executed over multiple application cycles in case of large internal states.

The Operator also extracts the state structure of the target application A' from its OPC UA address space via the internal OPC UA client (line 5). Due to updates to the application, this state may have variables added or removed from the previous version. Added variables are irrelevant for the state transfer and are initialized with their default values. Removed variables are excluded from the state transfer to make it more efficient. In many practical cases however (Patrick, 2007), there are only few isolated changes in the state structure, even to single variables. Nevertheless, the values of all internal variables need to be transferred. The Operator creates a *transfer structure* β (line 9) by comparing α and α'. It removes all variable references that are no longer in the target application. In our running example, no variables get removed or altered for brevity, so that $\alpha = \alpha' = \beta$. Afterwards, β is the ordered list of variables and types, whose values are subject to the state transfer.

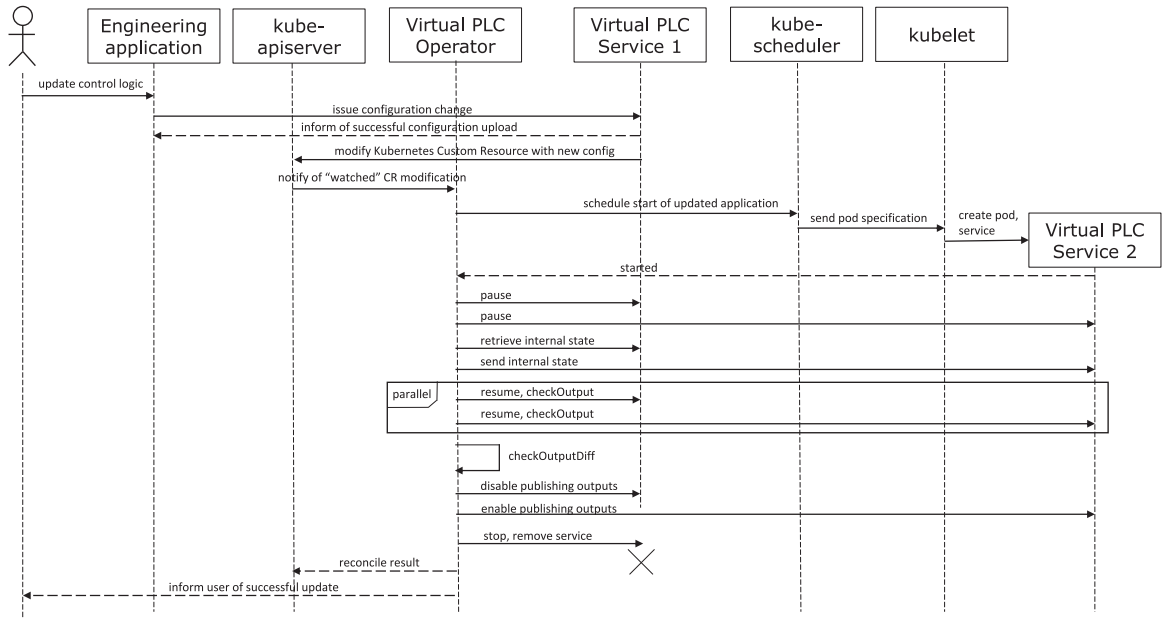

Fig. 6. VirtualPlcOperator: high-level UML Interaction diagram of a successful update procedure.

Prerequisite: Source runtime $R1$ executing application A,
Target runtime $R2$ executing application A'
but not publishing outputs,
States of A and A' unsynchronized

1 function TransferApplicationState $(m, ip1, ip2)$;
Input : Transfer method m: IPC || Network,
IP address $ip1$ source runtime $R1$,
IP address $ip2$ target runtime $R2$
2 **begin**
// not time-critical
3 connect to $ip1$, $ip2$ with OPC UA client;
4 $R1.extractStateStructure(A) \rightarrow \alpha$;
5 $R2.extractStateStructure(A') \rightarrow \alpha'$;
6 createTransferStructure$(\alpha, \alpha') \rightarrow \beta$;

// time-critical
7 wait for cycle slack time of A on $R1$, pause execution;
8 wait for cycle slack time of A' on $R2$, pause execution;
9 $R1.serializeStateValues(A, \beta) \rightarrow \gamma$;
10 **if** m =*Network* **then**
11 retrieve γ from $R1$ via OPC UA;
12 send γ to $R2$ via OPC UA $\rightarrow \delta$;
13 **else if** m =*IPC* **then**
14 $R1.sharedMemWrite(\gamma) \rightarrow address, size$;
15 $R2.sharedMemRead(address, size) \rightarrow \delta$;
16 $R2.deserializeStateValues(\delta, \beta) \rightarrow A'$;
17 resume execution $R1, R2$;

// not time-critical
18 $checkOutputDiff(A, A') \rightarrow diff$;
19 **if** $diff \leq userThreshold$ **then**
20 disable A publishing outputs;
21 enable A' publishing outputs;
22 stop A and shutdown $R1$;
23 **else**
// diff too high, rollback update
24 stop A' and shutdown $R2$;
25 provide message to user: "update aborted";
26 **end**

Algorithm 1: Provides a mechanism to transfer the internal state of cyclicly running control applications between two different runtimes.

The algorithm now waits for the source runtime $R1$ to complete its execution cycle (e.g., requiring 10 ms of the 100 ms cycle time) and then pauses its execution via an OPC UA method call to prevent any changes to the internal state during the cycle slack time (i.e., 90 ms, line 7). The same is performed for $R2$ (line 8). Pausing the runtimes is an extra safety measure to avoid unwanted state changes in rare cases the network transfer would take longer than the cycle slack time. Then, the Operator instructs $R1$ via a method call into the OPC UA address space to serialize the internal state *values* according to the transfer structure β. In the running example, this could for example yield γ = [7.3, 10.5], with the values simply being concatenated.

In this case, the state shall be transferred between two different hosts ($ip1, ip2$), so the Operator sends γ as a binary object via OPC UA to the runtime $R2$ (line 12). In case of a requested *local* state transfer, the operator can alternatively apply inter-process communication using shared memory to avoid the overhead of network marshaling. In that case, the operator instructs $R1$ to write the internal state to a memory location and retrieves only the memory start address and size (line 14). The runtime $R2$ is then instructed to read this memory location, circumventing the OPC UA communication for the value transfer.

Afterwards, $R2$ accesses the variable values as δ, e.g., δ = [7.3, 10.5]. It deserializes the values according to the transfer structure β and writes them into the application A' (line 16), which then performs identical computations of the outputs as A. For example, the values for the output $ControlOutput$ of the temperature controller should now be identical for A and A'. The operator finally resumes the execution of both $R1$ and $R2$ (line 17), but does not yet allow $R2$ to publish the outputs to the I/O devices. The procedure of pausing the runtimes and then retrieving, sending, and deserializing the state variables is time-critical and needs to complete within the cycle slack time (e.g., 90 ms).

In the final stage (line 18ff.), the algorithm monitors the outputs of both applications A and A' and measures a potential $diff$ that can occur both from unsynchronized clocks or the altered application logic.

The usage of PTP for clock synchronization rules out most problems due to unsynchronized clocks. Considering that the assumed cycle times are in the range of 100 ms, application-affecting unsynchronized clocks in this scenario are a rare case in practice. A more common case for outputs deviating from the desired values are erroneous control logic modifications attempted by the control engineer. Since the application logic may involve sophisticated mathematical algorithms, the outcome

Table 1
Variable values of the running example in four different phases: red cells indicate non-identical values between runtime R1 and R2; after the state transfer most values are synchronized between runtime R1 and R2 and the remaining deviations are as desired within the specified threshold.

	Variable values								
	Phase 1	→	Phase 2		→	Phase 3		→	Phase 4	
Variable names	Runtime R1 at initialization		Runtime R1 after 12 hours, before state transfer	Runtime R2 at initialization, before state transfer		Runtime R1 after state transfer	Runtime R2 after state transfer		Runtime R1 1 min after state transfer	Runtime R2 1 min after state transfer
previousError	0.0		0.02	0.0		0.02	0.02		0.4	0.35
integral	0.0		0.01	0.0		0.01	0.01		0.02	0.021
previousTime	T#0S		T#12h4m34ms	T#0S		T#12h4m34ms	T#12h4m34ms		T#12h5m34ms	T#12h5m34ms
error	0.0		0.5	0.0		0.5	0.5		0.6	0.7
deltaTime	T#0S		T#100ms	T#0S		T#100ms	T#100ms		T#100ms	T#100ms
deltaError	0.0		0.03	0.0		0.03	0.03		0.04	0.05
proportionalTerm	0.0		1.0	0.0		1.0	1.0		1.2	1.4
integralTerm	0.0		0.01	0.0		0.01	0.01		0.015	0.0021
derivativeTerm	0.0		0.0003	n/a		0.0003	n/a		0.04	n/a
limitHigh	n/a		n/a	100.0		n/a	100.0		n/a	100.0
limitLow	n/a		n/a	-100.0		n/a	-100.0		n/a	-100.0
ControlOutput	0.0		105.2	0.0		105.2	0.0		102.3	100.0
variable1	7		21	7		21	21		21	21
variable2	'msg42'		'msg42'	'msg42'		'msg42'	'msg42'		'msg42'	'msg42'
…	…		…	…		…	…		…	…
variableN	0.231		0.456	0.231		0.456	0.456		0.456	0.456

of a control logic update in terms of output signals is often not immediately clear to the control engineer. It can also hardly be tested offline, since it may be affected by the chemical processes controlled. For example, the control engineer may modify an advanced control algorithm to open a valve faster or to start a pump slower. The exact effect of these changes on the valve or pump behavior is often difficult to determine upfront and may rely on many dynamic variables in the process. In some cases, the control engineer can at least define thresholds for the output signals, which can then be checked by the algorithm. In other cases, the output signals need to be monitored and validated manually before the updated application can take over the process.

This step is not time-critical and being executed over multiple application execution cycles. If the drift is below the certain user-defined threshold, the VirtualPlcOperator switches the application execution to $R2$ and A' by disabling A from publishing outputs and enabling A' to publish outputs (line 20/21). $R1$ and A can then be deactivated and archived for potential late rollbacks. In case the $diff$ is too high, the algorithm simply deactivates $R2$ and issues an error message to the user. In this case the dynamic update has failed, and manual intervention is required (e.g., starting another attempt or executing a static update).

4.5. State transfer example

To better characterize our approach, Table 1 shows exemplary variable values of our running example (PID control algorithm from Fig. 4) in four different phases in a exemplary scenario. Notice that the values are chosen for illustration purposes and are not directly derived from an execution of the algorithm.

In phase 1, the source runtime R1 starts and initializes each variable with default values. In our running example (Fig. 4), the default values for the internal state variables are explicitly specified in the Structured Text code in line 16 to 24. Most of these are simply set to 0.0. An arbitrary number of additional internal state variables (variable1 to variableN) from other function blocks and algorithms in the same application may add to the complete internal state of the application (see the last four lines in Table 1). These can be more than 100.000 variables for a single application in realistic cases (Krause, 2007). If the default values were not explicitly defined, IEC 61131-3 also prescribes default values for each basic data type, which then can be used for the initialization.

In phase 1, the ControlOutput is still 0.0, since the execution has not yet started. The algorithm gets as input the ProcessVariable, which is a dynamically varying reading from a temperature sensor. The algorithm is executed with a cycle time of 100 ms, so that the internal state variables may now assume new values in each cycle based on the changing ProcessVariable input, previous executions, and the current time.

We now assume in this example scenario that the executed algorithm shall be updated after running for 12 h. The control engineer deems the derivativeTerm unnecessary for the temperature controller and removes it. The control engineer also adds new limits to the control output, which require two new internal state variable "limitHigh" and "limitLow". These changes constitute the update, and once the changes are committed and detected by the VirtualPlcOperator, the updated application is started in runtime R2.

In phase 2, most of the internal state variables in runtime R1 have assumed new values after 12 h of execution, due to the continuous calculations in each execution cycle. For example, the variable previousError now has the value 0.02 and the variable integral has the value 0.01. The newly started runtime R2 with the updated application initializes the variables with the default values, most of them again at 0.0. Notice in Table 1, that the variable derivativeTerm is not initialized in runtime R2, since the control engineer has removed it. Furthermore, the two new variables limitHigh and limitLow, which do not exist in runtime R1, receive default values. The ControlOutputs of R1 and R2 are different, since they are computed based on different internal states. However, our method assures that still only the outputs of R1 are written back into the process, thus controlling the actuators, while the outputs of R2 are computed but not published before the state transfer and a successful deviation check.

Transitioning from phase 2 to phase 3, the VirtualPlcOperator now conducts the state transfer by executing Algorithm 1 up to line 17. After this transfer, we enter phase 3 in our example. Table 1 shows that runtime R2 has now changed its default values to the same values as in runtime R1. This means that due to the state transfer more than 100.000 variables (cf. variable1 to variableN in the last four lines) have now changed their values from the default values to the current values in runtime R1. However, the internal state of both runtimes is now not identical, since runtime R2 has removed the derivativeTerm and added limitHigh and limitLow. Since the algorithm has changed in R2, in the following execution cycles, the variable values between runtime R1 and R2 may deviate.

After 1 min of parallel execution of R1 and R2, we transition from phase 3 to phase 4 in our Table 1, which corresponds to checking the output diff in line 18–25 in Algorithm 1. At this point, most of the internal state variables (e.g., variable1 to variableN) are still identical due to the same input sensor data coming from the process. Consider that in large applications these variables make up more than 99 percent

of the entire internal state. However, for the variables for the updated function block, the ControlOutput now deviates between R1 (102.3) and R2 (100.0). Due to the introduced control output limits between −100.0 and +100.0, the ControlOutput of R2 is now at 100.0, which alters the valve opening of our running example and thus has the desired effect of the update. With a specified user threshold for the difference of the control output of less than 10 percent, the VirtualPlcOperator would now execute line 20–22 in Algorithm 1. These lines disable writing the outputs from R1 and enable writing the outputs from R2, before stopping and shutting down R1. In this example scenario, the control engineer's update was successful.

4.6. Assumptions & limitations

To better contextualize our method, we briefly describe some of the underlying assumptions and limitations:

- **Cycle slack time:** the IEC 61131-3 programming model induces an uninterrupted cycle slack time after computing the algorithms, where the application is idle waiting for the next input signals. During this slack time the internal state is not altered and can be copied. Other programming models may not have these uninterrupted phases and the state would have to be extracted while it is being modified.
- **Small application state changes:** As our method requires a validation of the application output of the original and updated version, large changes of more than 10 variables at once may be impractical. The more the update alters the application's output, the harder the validation becomes. If an entire algorithm is exchanged, it is up to the control engineer to initialize the internal state of the new algorithm appropriately if needed or rely on default values. Our approach cannot map between two different internal state variables automatically if this involves additional calculations. However, it can preserve state if the same internal state variables are available before and after then update.
  Our method allows for slightly differing outputs after an update, for which an application-specific threshold can be defined or for which human oversight can be triggered. For example, if the valve from the running example in Fig. 4 received slightly higher output signals from the temperature controller, this may still be within the valid range in this specific application case.
  Practical online application updates of control applications by large automation customers such as Dow Chemical have successfully used similar state transfer methods in the past (Patrick, 2007), which were however restricted to single embedded devices. Furthermore, the assumption of small *application* updates does not affect the capability of the method for large *library, infrastructure, and hardware* updates.
- **No changes to cycle time:** the updated application must have the same task cycle times as the former application. Although it is possible to adjust the task cycle times via a control logic engineering tool this easily would make the original and updated application hard to validate and is therefore not permitted. Changes to the task cycle times an still be done through offline engineering, if desired.
- **Cycle time ranges:** our method only works for process control applications with cycle times of about 10 ms to normally 1000 ms. Faster control cycles below 10 ms or in the sub-millisecond range, e.g., used in machine control, are not supported. However, typical process control applications do not exhibit such fast cycle times.
- **Control runtime properties:** we assume that control runtime provides a mechanism to copy all internal state variables, which was already given for the two sample implementations described later. Furthermore, the control runtime must have an OPC UA interface for injecting the internal state.
- **IPC state transfer only on a single host:** by design, the state transfer via inter-process communication is only possible on a single host. In this case, a live-migration of the control application to another node (e.g., for node maintenance) is not possible.
- **No concurrent updates to multiple pods:** the method is not designed to update multiple pods in parallel, which could complicate checking the output differences. If a user wants to make updates involving different applications and pods, these need to be executed sequentially following the procedure described in Algorithm 1.
- **No safety certification:** some applications in process control require a safety certification according to IEC 61508, which requires specially tested software and redundant hardware. Our method is currently not designed for these settings, as the used operating systems and virtualization infrastructure have no safety certification.

5. Prototypical implementation

We programmed the VirtualPlcOperator as a dedicated application in C++. This VirtualPlcOperator is implemented as a lightweight operator, which ensures that the state transfer is executed properly. The core functionalities for the state exchange are implemented in the respective PLC runtimes, as the VirtualPlcOperator orchestrates that state transfer. As mentioned before, the VirtualPlcOperator is a dedicated application, hence, we deployed the VirtualPlcOperator as a docker container in a separate Kubernetes Pod within the cluster. In order to minimize jitter and guarantee a fast state transfer, this Pod is deployed on a compute node within the testbed with CentOS and the Linux PREEMPT_RT kernel patch. It interacts with VirtualPlcs via an integrated OPC UA client. The implementation supports both OPC UA and interprocess communication (IPC) state transfers. We also added a web server for manual interaction. In a production environment, the operator shall run mostly in an autonomous mode without manual input required.

We selected a commercial ABB runtime as well as an open-source runtime, namely OpenPLC.[6] The ABB runtime is written in C++ based on a predecessor that is integrated in thousands of commercial control system installations. The runtime already featured an integrated OPC UA server to retrieve and send signal values. We modified the runtime to expose the internal state serialization and de-serialization as well as pausing and resuming via OPC UA methods. Due to that it is possible to marshal and un-marshal the internal memory state of the runtime with all application relevant variables and states. This marshaled and un-marshaled memory state is used by the OPC UA methods. The VirtualPlcOperator uses these methods, to orchestrate the state exchange between the PLC runtimes.

We also integrated the same mechanisms in the freely available OpenPLC runtime written in C. As it did not include OPC UA connectivity, we enhanced it with an Open62541[7] OPC UA server. Additionally, we added the possibility to pause and resume the OpenPLC runtime. We also updated the OpenPLC runtime with the same mechanism as in the ABB runtime, to marshal and un-marshal the internal application-relevant memory state. We used the same serialization library 'cereal'[8] and method as in the commercial runtime. We deployed multiple instances of both types of PLC runtimes as Docker containers running in VirtualPLCPods in Kubernetes. The containers ran in privileged mode to support the highest POSIX real-time priorities and to minimize interruptions from other processes.

Our testbed utilized the open-source StarlingX 4.0[9] cloud infrastructure platform, which bundles Docker, Kubernetes, ceph, OpenStack,

[6] https://www.openplcproject.com/
[7] https://open62541.org
[8] https://uscilab.github.io/cereal/
[9] http://starlingx.io

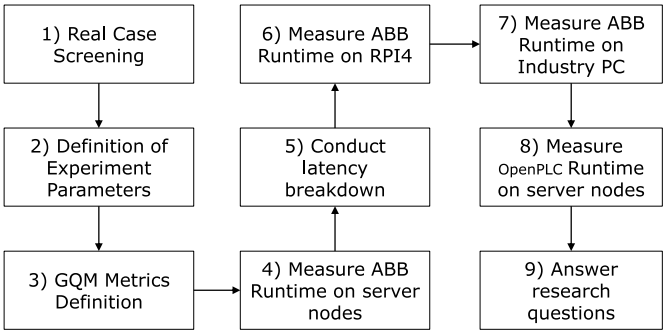

Fig. 7. Validation steps: from case screening to answering research questions.

and other OSS components. We used the StarlingX All-in-one (AIO) Duplex Configuration with redundant control plane nodes. We also set up two worker nodes running CentOS and the Linux PREEMPT_RT kernel patch The servers had Intel Xeon CPUs with up to 40 cores and 128 GB RAM each.

For smaller installations, we tested the method on a single Industrial PC (IEI Tank-870AI-E3, with an Intel Xeon Quad core CPU, and 32 GB RAM). This PC included Yocto Linux with the PREEMPT_RT patch and a Docker runtime. Here we deployed the VirtualPlcOperator as a plain docker container on the same node and used a custom mechanism to detect configuration changes and start the updated application version.

Finally, we also deployed the prototype to Raspberry Pi 4 (RPI) single-board computers, which have an ARM-based quad-core CPU and 4 GB RAM. We added a real-time clock hardware module to the RPIs to be able to get precise time measurements. We did not use real I/O devices, but instead emulated their behavior using OPC UA servers that frequently published simulated sensor data and consumed updated set points and commands. For the RPI testbed we used 2 RPI 4. The first RPI 4 host the VirtualPlcOperator and is connected via Ethernet to the other RPI 4. This RPI 4 hosted the PLC runtimes which are used for the state transfer. For both components, VirtualPlcOperator and PLC runtimes, we use plain docker containers. The RPI 4.0 were equipped with an Ubuntu 18.04.04 LTS patched with the PREEMPT_RT patch to ensure low latency capabilities.

6. Validation

To evaluate our approach, we conducted the 9 steps shown in Fig. 7.

6.1. Case study screening

To validate our state transfer method, we aimed at identifying realistic and representative test cases.

Based on literature (Krause, 2007) and feedback from ABB local engineering units regarding typical application parameters, we determined a task cycle time of 100 ms in our experiments as sufficiently fast to represent a large class of industrial applications. We assume a state transfer of 100.000 variables at a 100 ms task interval as a our goal. The controllers typically have a significant cycle slack time to allow for robustness in case of disturbances. We thus assume here a cycle slack time of 90 ms.

6.2. Goal/questions/metrics

The goal of our validation was to find out if our method can support an update of large-sized control applications (i.e., 100.000 internal variables) and to determine the maximum number of variables transferable within the defined target cycle slack time (i.e., 90 ms).

Derived from the goal, the first question was to determine the feasibility for OPC UA state transfer since this is the most flexible and demanding method, as it allows updating underlying software layers but requires network marshaling. As metric M_1 to operationalize this question, we defined the maximum OPC UA state transfer time between two server hosts for a test application with 100.000 internal state variables. Let $t_{x,y,z}$ be a measured transfer time, where x is a transfer method (OPC UA or IPC), y is the number of variables transferred, and z is the number of a measurement experiment. Then M_1 is defined as:

$$M_1 = max_{1 \le j \le m} \left(t_{opcua,100000,j} \right)$$

where m is the number of measurements for this state size.

The next question was to quantify the speed-up achieved by the IPC transfer. As metric M_2, we defined the average ratio in the state transfer times between OPC UA and IPC communication:

$$M_2 = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{1}{m} \sum_{j=1}^{m} t_{opcua,i,j} \Big/ \frac{1}{m} \sum_{j=1}^{m} t_{ipc,i,j} \right)$$

where n is the number of state size classes measured (i.e., nine classes, between 25 K and 400 K) and m is the number of measurements carried out for that state size (i.e., 100).

To find out the maximum application size for the target cycle slack time and the given platform, we defined the following metric, where again m is the number of the experiment and n is the number of state size classes:

$$M_3 = max_{1 \le i \le n} \left(t_{opcua,i,m} | t_{opcua,i,m} \le 90 \text{ ms} \right)$$

To better understand how the state transfer time is spent and where the method has potential for optimization, metric M4 is defined as the average execution times of individual steps of the state transfer.

$$M_4 = (s_1, \ldots, s_n) | \sum_{i=1}^{n} s_i = M_1$$

where s_i is the average execution time for the i'th step of the state transfer. As automation systems may utilize more or less powerful hardware, we checked the state transfer times on other typical hardware platforms. We do not define speed-up metrics here since the results depend on contextual factors such as background load. Therefore, we aimed at providing a visual representation of the transfer times. Finally, we asked how the state transfer times of the OpenPLC and the ABB runtime differed and also selected a visual representation over a formal metric.

6.3. Experiment procedure

For the experiments, we first (Fig. 7, step 4) set up a control plane node and two worker nodes as depicted in Fig. 5. These were server nodes with StarlingX/Kubernetes installed, as described in Section 5. In this testbed, we also installed our VirtualPLCOperator, as well as the OpenPLC and ABB runtimes as pods. These pods contained *generated* control applications that performed simulated test loads by continuously calculating dummy multiplications on externally supplied inputs. These test applications were thus not real customer applications, which were unavailable for our research as they belong to the customers and contain their intellectual property. However, our test applications emulated the load of such applications due to their sizes and calculations.

The control application generation allowed us to test state transfers for different application sizes as depicted in our result diagrams, including applications with 100.000 variables as derived from the case study screening in Section 6.1. For our state transfer experiments we always transferred the entire internal state from one pod to another pod. A script triggered each state transfer by invoking the state transfer method of our VirtualPLC operator.

We did not *modify* the internal state nor other parts of the application in these experiments, e.g., which could have simulated adding or removing a variable for an update. Thus, we only performed the

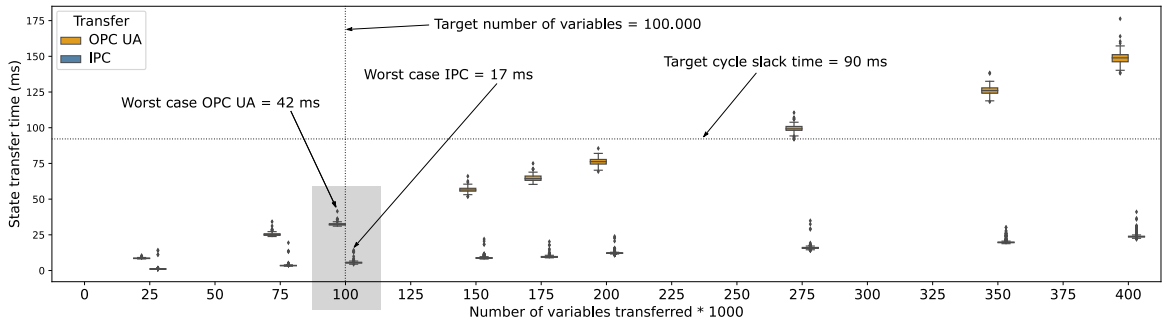

Fig. 8. Boxplot of the state transfer times for IPC and OPC UA on StarlingX server nodes. Both methods manage the state transfer for 100 K variables in less than 90 ms, the IPC-method is faster by an average factor of six.

state transfer between identical control applications. This is a valid simplification, since first our method supports infrastructure updates, where the state and the application remain unmodified. Secondly, our method is meant for application updates of less than 10 variables, as explained in Section 4.1, which would thus alter a state size of 100.000 variables by only 0.1 percent and thus affect the measured timings by an amount smaller than the error of measurement. We thus also did not check for output differences, since these are not time-critical and highly application-specific. We clarify that our approach transfers the values of 100.000 variables and alters the default values of the variables at the target application to these values. Most of the variables retain the same values as in the original application, but up to 10 variables may have changed variable values which reflect the control engineer's update.

We measured the timings reported in the following by instrumenting the code of our VirtualPLCOperator, which includes the state transfer procedure from Algorithm 1. We took timestamps before entering the time-critical zone in line 7 of Algorithm 1 and after ending the time-critical zone in line 17. Other parts of the algorithm are not time-critical and can take an arbitrary long time, thus we excluded these from our measurements.

To account for temporary timing distortions and outliers, we executed the state transfer for each given application size and infrastructure configuration 100 times through the script that invoked the state transfer algorithm. The script paused for 2 s between each state transfer to avoid caching effects.

6.4. IPC vs. OPC UA

Our method offers two variants to execute the state transfer, which each have their benefits and drawbacks and need to be selected based on a specific application context. The following characterizes the timings of both variants but should not be mistaken for a comparison of our approach to a baseline. Our methods is functional in both variants, however IPC-based state transfers by design cannot be executed between two separate hosts and therefore exclude many infrastructure update scenarios.

Fig. 8 shows a boxplot of the state transfer times in milliseconds for the different state sizes, both for transfer via OPC UA and IPC. The measurements contain a few minor outlier points, which are however not severely affecting the overall results. These outliers can be attributed to competing workloads running on the cluster during the experiments, which reflects realistic conditions.

The orange boxes indicate OPC UA communication across two different nodes, while the blue boxes indicate IPC communication within a single pod on a single node. Both mechanisms manage the state transfer in less than 90 ms for the test application with a cycle time of 100 ms. For OPC UA and 100.000 variables the operator transfers the state with a maximum of M_1=42 ms and average case of 32 ms. For local IPC, the operator only took at maximum 17 ms and on average 5.5 ms. The average ratio of OPC UA to IPC transfer time is M_2=6.55, that is, IPC is more than 6 times faster than OPC UA, since network marshaling and transfer is not required.

The operator can transfer up to M_3=200.000 variables in less than 90 ms via OPC UA, and much higher numbers via IPC. Such large applications would however be beyond typical limits used today and could run into other bottlenecks, such as the maximum number connected IO devices in parallel. The experiment series in Fig. 8 also shows that the transfer times increase linearly for both communication methods, indicating that the system did not run into a bottleneck. The workload is CPU-bound, as the required network bandwidth for the transfer is low (approx. 0.5 percent of Gigabit Ethernet). Even faster server execution times could provide more cycle slack time. However, this can hardly be utilized for much larger state sizes.

6.5. Latency breakdown

Instrumenting the operator source code yielded a latency break down for different steps of the state transfer. Fig. 9 provides a boxplot for four steps of the state transfer from an experiment with OPC UA and 100.000 variables sent from one engine to another one. The step "Read mem" maps to line 9 in Algorithm 1 and took on average 12.3 ms. The steps "Marshaling" and "Unmarshaling" map to line 11 and 12 respectively in Algorithm 1 and needed less than 5 ms in all measurements. This also included the network transfer itself (assuming an undisturbed network, or usage of TSN priority classes), which consumed negligible time in this setting, due to the low amount of data (400 KByte) and the high bandwidth (10 GBit/s).

The step "Write mem" corresponds to line 16 in Algorithm 1 and required the most time, on average 15.2 ms. The used serialization library 'cereal' showed a slightly higher execution time for the deserialization compared to the serialization. It is conceivable to decrease the time for serialization and deserialization using other libraries, such as Google Protocol Buffers. To further avoid potential jitter introduced by the network, it would be also useful to use a TSN (Time Sensitive Networking) connection with prioritized traffic for the state transfer, however optimizing the serialization itself is expected to be more effective given the results.

6.6. Server vs. Industrial PC vs. ARM

All measurements shown so far were obtained from the StarlingX server nodes. Fig. 10 compares measurements on the servers with measurements on a resource-constrained Raspberry Pi 4 and a firmware optimized Industrial PC. These experiments only used IPC communication on single nodes for better comparability. The RPI4 required compiling the ABB runtime for an ARM processor. Here we used plain Docker without Kubernetes but with a separate C++ service emulating the VirtualPlcOperator. This setup cannot yet support updates initiated by the orchestration system, which however does not distort the state transfer performance measurements.

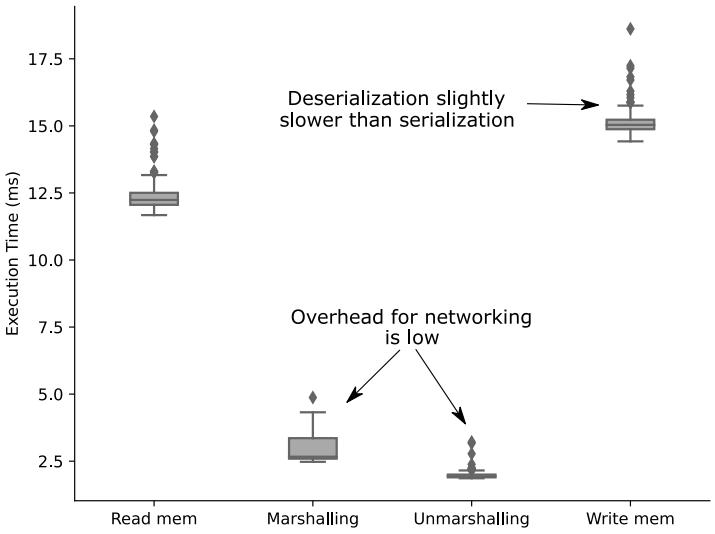

Fig. 9. State transfer time breakdown: most of the time is spent in serialization and deserialization, while the network transfer is comparably fast.

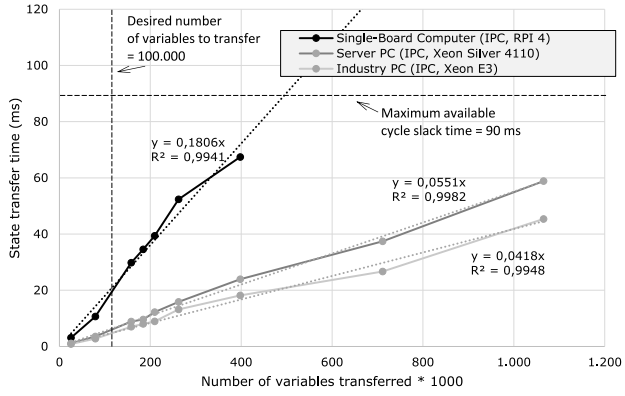

Fig. 10. Platform comparison: the Industrial PC achieves the fastest state transfer on a single node and thus could scale up to very high application sizes.

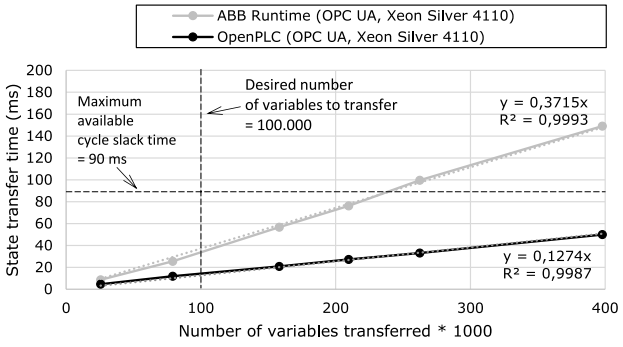

Fig. 11. The OpenPLC runtime has faster state transfer times compared to the ABB runtime. Both runtimes show a linearly increasing trend for a higher number of variables.

All three systems show a linear trend in Fig. 10. The Industrial PC managed the largest state transfers, followed by the server PC. The Industrial PC had fewer background loads and an optimized firmware, which contributed to the slightly better performance compared to the server PC, which had much more CPU cores. The RPI4 required significantly more time for the state transfer due to the lower CPU performance, but still managed to transfer 100.000 variables in much less than 90 ms. Due to the fast state transfer via IPC, we were able to conduct measurements for state sizes above 1.000.000 variables on the Industrial PC as well as the server PC. However, such application sizes are no longer realistic and well-beyond the required threshold.

6.7. Commercial vs. Open-source implementation

Fig. 11 compares the OPC UA state transfer times of the ABB Runtime and the OpenPLC runtime on the StarlingX server nodes. This validates that the state transfer method is feasible for a non-ABB runtime. In fact, the OpenPLC runtime was able to transfer the internal state faster than the ABB Runtime. Both runtimes support executing IEC 61131-3 control logic, however the OpenPLC runtime consists of a single process and is meant for resource-constrained, small-scale PLCs, whereas the ABB Runtime includes multiple processes hosted in the same Docker container, which provide rich functionality for larger DCS controllers. The latter includes optional support for redundancy as well as sophisticated facilities for monitoring.

Table 2
Metrics determined in the experimental validation to answer the research questions.

Metric	Description	Value
M_1	Maximum transfer time for OPC UA state transfer time between two nodes for a control application with 100.000 internal state variables	42 ms
M_2	Average ratio of state transfer times OPC UA and IPC	6.55
M_3	Maximum internal state size to be transferable via OPC UA in the cycle slack time of 90 ms	200K variables
M_4	Average execution times for steps of the state transfer (OPC UA, 100.000 internal state variables, two nodes)	(12.3 ms, 2.7 ms, 1.8 ms, 15.2 ms)

Table 2 summarizes the main metrics M_1 to M_4 as the outcome of our experiments.

6.8. Threats to validity

We discuss the internal, construct, and external threats to the validity of the experimental results, so that the reader can better evaluate their robustness.

The **internal validity** is the extent to which the measurements provided correct values and to which interfering variables could be controlled. We measured the execution times only by instrumenting the source code of the VirtualPlcOperator but did not systematically triangulate the measurements with other monitoring tools. However, a

few random tests with separate OPC UA clients did not indicate any large deviation. To avoid caching effects, we did not perform the state transfer in directly succeeding cycles but introduced a waiting delay after each transfer.

We did not systematically measure CPU nor network utilization, since the feasibility of the state transfer was already shown without exhausting both resources. Thus, this is deemed a negligible threat to validity. The experiments ran fully automated which excluded distortion from human intervention. Kubernetes background workloads were not fully controlled to reflect a realistic cluster workload, however we did not see serious outliers in the measurements that could have affected our conclusions. To improve the reliability of the results and control for smaller glitches caused by the platform, we repeated each test 100 times, so that we were confident in not experiencing transient effects.

The **construct validity** is the extent to which the test subjects used as proxies in the experiments were actually representative and realistic to allow conclusion for automation systems in production. We used an actual commercial control logic execution runtime, whose former versions automate thousands of large production plants. The test application was sized according to some of the largest applications published and computed comparably complex calculations (Krause, 2007). The chosen cycle time of 100 ms is typical for many process automation applications, since the processes often involve physical measurements for temperature and level, which cannot be measured in higher frequencies in a useful manner. A few processes may require shorter cycle times (e.g., 10 ms for specific pressure measurements) to avoid equipment damage, which can only be handled by our IPC-transfer method on a single node. The method is not designed for sub-millisecond cycle times used in machine control.

In our tests we assumed an internal state comprising only of 32-bit encoded basic data types. Some applications include potentially larger string types or arrays as variables, however these usually do not change after default initialization and are thus not relevant for the state transfer. IEC 61131-3 control logic is widespread in industrial automation (Koziolek et al., 2020) and has a simple execution model that avoids threading which could have complicated the state transfer. We used simulated IO devices instead of real sensors, but configured OPC UA servers to publish in the same frequencies as such devices, which is common practice for testing such runtimes. For the Kubernetes-based method, we could not use a hard real-time operating system, such as VxWorks or FreeRTOS. However, we used Linux with the PREEMPT_RT patch and thus at least soft real-time capabilities and assured a reduced jitter. We could not compare our method directly to other methods in literature since they either do not support software containers or are not geared for cyclic applications.

The **external validity** is the extent to which the findings can be transferred to other situations and settings. To improve the external validity, we tested the method both with a commercial and an open-source runtime. This demonstrates that potentially other IEC 61131-3 runtimes deployable into containers and providing OPC UA connectivity could also utilize our state transfer method. Some of the most complex applications served as template for our test applications, which provides confidence that the method also works on the vast range of smaller applications.

7. Related work

Several articles provide an overview of the field of dynamic software updates in the last 30 years (Miedes and Muñoz-Escoí, 2012; Seifzadeh et al., 2013; Ahmed et al., 2020). Numerous seminal works have explored the topic from the perspective of programming languages (Gupta et al., 1996; Hicks and Nettles, 2005; Ajmani et al., 2006; Stoyle et al., 2007; Subramanian et al., 2009), operating systems (Baumann et al., 2005; Makris and Ryu, 2007; Giuffrida and Tanenbaum, 2009) and software engineering (Chen et al., 2007; Previtali, 2007; Baresi et al., 2016).

Many of these works aim to identify the best point in time when an application is ready for a state transfer. Kramer and Magee (1990) introduced the notion of processing units being quiescent, if they are not currently engaged in transactions they initiated themselves, if they do not initiate new transactions, if they are not currently servicing a transaction, and if no other services have been or will initiate such transactions. In such a quiescent configuration state, the application state is consistent and frozen. In the simple and predicTable 1EC 61131-3 software execution model (Plaza et al., 2006), this quiescent configuration state is clearly identifiable as the cycle slack time. Furthermore, in our context, updates are triggered by container orchestration systems or human operators, who can provide additional assurance that the overall system (e.g., running power plant) is in a safe state. In line with the mapping study of Ahmed et al. (2020) we position our approach in the application domain of time-critical systems dealing with quiescent state (see Table 3).

Wahler et al. (2009) introduced an update algorithm based on shared memory transfer that was embedded in a component framework. They transferred an internal state of 4000 Bytes on a single node in less than 5 ms. This work was later extended to enable an iterative state synchronization spanning multiple execution cycles (Wahler et al., 2011; Wahler and Oriol, 2014). This is more flexible regarding the update procedure itself than our approach as it does not constrain the update mechanism to conduct the update in a single cycle slack time period. However, it is not guaranteed that this form of state transfer terminates due to potential rapidly changing states. In this sense, our approach is currently more reliable regarding the state transfer, besides being capable of supporting more sophisticated updates across multiple nodes and even to the runtime itself. Our approach can be extended with a multi-cycle update capability to expand its scope.

Mugarza and Mugarza (2021) employed a Petri-net runtime engine to conduct live updates in PLCs for robot control. State sizes are smaller and cycle times are faster in robot control compared to the approach for process automation we have proposed. As with all other approaches in this class, Murgaza's approach is tied to updating within a single node and cannot migrate PLC programs to other nodes. It also does not support updating the PLC runtime or underlying operating systems. Prenzel and Provost (2017) compiled IEC 61499 applications for the Erlang Runtime System that natively provides dynamic updating but did not demonstrate state transfers in their experiments.

None of these works on dynamically updating real-time systems utilized a container deployment or tackled state transfer across different hosts. The transfer across multiple hosts introduces a delay for marshaling and network communication, furthermore, in our approach the PLC software runs on a virtualization stack including Kubernetes. As a result, the state transfer times of the different approaches are not directly comparable. As stated in Section 4.3, our approach is inherently not meant for very fast cyclic control applications with cycle times in the sub-millisecond range (e.g., machine control).

More recently, researchers analyzed the feasibility of deploying cyclic real-time applications using software container virtualization, which is desirable for portability and flexible updating (Forbes, 2018). Moga et al. (2016) discussed the concepts and constraints for deploying real-time applications to containers and executed the microbenchmark "cyclictest" within a Docker container on an Intel Xeon CPU. They found a negligible added jitter of less than 20 microseconds. Goldschmidt et al. (2018) extended these results to more resource-constrained single-board Raspberry Pi computers, where they measured a jitter of less than 100 microseconds. In a similar manner, Sollfrank et al. (2020) measured the container overhead for a PID controller implemented in Simulink C-code at 150 microseconds. These works paved the way for the Virtual PLCs considered in our paper but did not study dynamic updates at runtime (see Table 4).

Table 3
Comparison to related work: approaches for updating cyclically executing real-time systems. None of these approaches supports a multi-node setting, which prevents seamless updates to the runtime and underlying node.

	Alonso1995	Stewart1997	Patrick2007	Wahler2009	Wahler2011	Murgaza2021	Our approach (IPC)	Our approach (OPC UA)
Computing nodes	Single node	Single node	Single node	Single node	Single node	Single node	Single node	Multi node
Processor cores	Single core	Single core	Single core	Single core	Single core	Single core	Multi core	Multi core
Hardware	MVME-135, MC 68020, 32 bit, 16 Mhz, 1 MB RAM	Ironics IV 3230 single board computer, MC 68030, 25 Mhz	MPC8270, 450 Mhz, 256 MB Ram	Freescale Lite5200B, PowerPC, 396 Mhz, 256 MB RAM	Freescale Lite5200B, PowerPC, 396 Mhz, 256 MB RAM	OMRON PLC	IEI Tank-870AI-E3, Intel Xeon Quad Core, 32 GB RAM	IEI Tank-870AI-E3, Intel Xeon Quad Core, 32 GB RAM
Operating system	? (Ada)	Chimera RTOS	VxWorks	Integrity	RT-Linux	?	RT Linux	RT-Linux
Container runtime	n/a	n/a	n/a	n/a	n/a	n/a	containerd	containerd
Container orchestration	n/a	n/a	n/a	n/a	n/a	n/a	Kubernetes	Kubernetes
Application domain	Railroad control	Robot control	Process automation	Overvoltage protection	Overvoltage protection	Robot control	Process automation	Process automation
Cycle time	200 ms	1 ms	100 ms	5 ms	1 ms	?	100 ms	100 ms
Cycle execution time	50 ms	250 us	10 ms	273 us	750 us	531 us	10 ms	10 ms
Cycle slack time	150 ms	750 us	90 ms	4727 us	250 us	?	90 ms	90 ms
State transfer approach	Single-cycle	Single-cycle	Single-cycle	Multi-cycle	Multi-cycle	Single-cycle	Single-cycle	Single-cycle
State size	16 Bytes	~200 Bytes	400 Kbyte	4 Kbyte	4 Kbyte	< 1Kbyte	400 Kbyte	400 Kbyte
State transfer time	~1 ms	~120 us	? (<90 ms)	<5 ms	600 ms (multi-cycle)	?	17 ms	42 ms

Table 4
Comparison to related work: approaches for container migration. None of the approaches targets cyclically executing control applications that shall not exhibit any visible service outages.

	Yu2015	Netto2017	Qiu2017	Machen2018	Oh2018	Vayghan2019	Ma2019	Our approach (IPC)	Our approach (OPC UA)
Purpose / use case	Container live-migration	Failover for K8s pods	Container live-migration	Container live-migration	Container live-migration	Fail-over for K8s pods	Container live-migration	Control application update	Control application update
State transfer approach	Replay recorded log on new container	Continuous state transfer	Container checkpointing (CRIU)	Container checkpointing using rsync	Container checkpointing (CRIU)	Continuous state transfer	Container checkpointing based on image layer comparisons	Application checkpointing on containerized IEC 61131 runtime	Application checkpointing on containerized IEC 61131 runtime
Container runtime	Docker	Docker	LXC	LXC	Docker	Docker	Docker	containerd	containerd
Container orchestration	n/a	Kubernetes	n/a	n/a	n/a	Kubernetes	n/a	Kubernetes	Kubernetes
Application domain	SPECweb99 Benchmark	Simple counter application	Generic "cloudlets"	Video streaming, face detection	Webserver	Video streaming	Busybox, OpenFace	Process automation	Process automation
State size	?	4 Byte	?	1-10 Mbyte	1120 Kbyte	16 Bytes?	30 Kbyte - 23 Mbyte	400 Kbyte	400 Kbyte
Service downtime	50 ms	n/a	?	2-15 secs	1.1 secs	2.8 secs	2 - 40 secs	0.0 secs (uses cycle slack time)	0.0 secs (uses cycle slack time)
Total migration time	31 secs	n/a	240-290 secs	6-15 secs	1.1 secs	4.5 secs	3 - 45 secs	17 ms + arbitrary long drift evaluation	42 ms + arbitrary long drift evaluation

shows approaches for live container migration as well as state replication in container orchestration systems. They do not tackle the migration of cyclically executing real-time application. Yu and Huan (2015) proposed a log-and-replay approach for the live migration of Docker containers which however targeted migrations lasting about 30 s, which are not applicable for control applications. The DORADO protocol (Netto et al., 2017) allows ordering requests in Kubernetes that are saved to shared memory (e.g., etcd) and can be exchanged between container replicas. This work targets fail-over scenarios and assumes a continuous state transfer between containers.

Qiu et al. (2017) presented a live migration approach for LXC containers using CRIU container checkpointing mechanism.[10] This approach showed a migration time of over 240 s. Machen et al. (2017) also investigated LXC container migration using the CRIU mechanism and conducted experiments for game servers, video streaming, and face detection applications. Using rsync for file synchronization and a layered approach, they achieved lower service downtimes and total migration times. Oh and Kim (2018) introduced a checkpoint-based stateful container migration method for Kubernetes. They experimented with state sizes around 1 MByte and achieved a total migration time of 1.1 s. Vayghan et al. (2019) proposed a state controller for Kubernetes, which replicates internal state between containers to support redundant stand-by containers and fail-over scenarios. In experiments, they persisted the client state of a video streaming application to a persistent volume and measured container restart times, which could be substantially reduced. Finally, Ma et al. (2019) investigated live-migration of containers and conducted extensive experiments, measuring service downtimes and migration times for Busybox and OpenFace. Fast, in-memory CRIU-based container snapshots have been demonstrated (Venkatesh et al., 2019), but these rely on migrating the entire container state to an exact replica and are thus not suited for dynamic updates, where the container image and/or application changes.

To better characterize the difference between container checkpointing as supported by CRIU and our application state transfer method, we have executed a test where we created a checkpoint of a Docker container of the OpenPLC runtime, which was also used in our experiments in Section 6. The container image size is 1.03 GB and the CRIU checkpoint of this container had a size of 17.7 MBytes. It contains much more data than the control application internal state used in our approach, which had a size of 0.4 MBytes and is thus much faster to transfer over the network than CRIU checkpoints. The latter contain for example the entire process state, process meta-data, file descriptors, process namespaces, cgroups, and mount points. This information does not need to be transferred in our approach and can be re-created outside a time-critical window by starting a new container.

The time for creating the container checkpoint of the OpenPLC runtime was 700 ms, whereas reading the application state variables in our approach took less than 16 ms (Fig. 9). We conservatively assume the same time for restoring the container as for creating the checkpoint. The time to transfer a 17.7 MByte snapshot over a Gigabit link as in our experiments is at least 138 ms. This means that a CRIU-based live migration would at least take 1538 ms under optimal conditions, which would violate our desired cycle slack time limit of 90 ms severely. If large applications would be loaded into OpenPLC, the

[10] https://criu.org/

container snapshot size would increase, thus delaying the state transfer time further. We therefore conclude that CRIU container checkpointing cannot address the use case we have tackled.

None of the existing works in line of container migration research explicitly supports cyclically executing control applications that utilize sub-second cycle times. Unlike typical container live-migration approaches, our approach uses comparably small, custom snapshots of application internal state, which are more fine-granular than an entire container checkpoint. This provides us the ability to speed up the state transfer, so that it can be conducted within the short cycle slack time and not interrupt the running production processes. In the successful case, our approach does not exhibit any externally visible service outage, since the I/O devices only expect new inputs every 100 ms and do not get aware that the signal publisher has changed to an updated application.

Our method is based on the method introduced by Koziolek et al. (2021). It has been redesigned and fully integrated into a container orchestration framework. We have conceptualized a novel "fast-mode" internal state transfer via shared memory within the same pod, which can expand the application areas of the approach. We have implemented the method as an extension to a commercial PLC runtime, whereas the former method used only open-source software. Moreover, we have conducted a large set of experiments including different hardware platforms, state transfer mechanisms, and PLC runtimes.

8. Conclusions and future work

We have introduced a state transfer method for dynamically updating industrial control applications without interrupting the underlying production processes. It relies on containerized VirtualPLCs deployed in Kubernetes. Experiments with large applications showed that the required state transfer between containers takes less than 50 ms via OPC UA, which is well below the target 90 ms cycle slack time. We still see potential for improving the method's performance by using alternative serialization libraries or more streamlined communication protocols.

Our method can impact the practice of developing and running industrial control applications. Our method provides human operators a safe update mechanism. Easier updates can lead to better optimizations of the automation as a whole as well as better security due to faster removal of vulnerabilities. Automation systems also may potentially increase production yield and quality via updates.

In future work, the VirtualPlcOperator could be enhanced to automatically select optimized nodes for deploying the updated application. It is also conceivable to transfer the internal state in multiple independent chunks after a static code analysis (also see Wahler and Oriol (2014)), therefore potentially expanding the range of supported applications. Another area for research is the execution and updating of containerized applications on resource-constrained devices.

CRediT authorship contribution statement

Heiko Koziolek: Conceptualization, Methodology, Software, Supervision, Validation, Writing – original draft. **Andreas Burger:** Conceptualization, Methodology, Software, Validation, Writing – original draft. **Abdulla Puthan Peedikayil:** Software, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

References

Ahmed, B.H., Lee, S.P., Su, M.T., Zakari, A., 2020. Dynamic software updating: A systematic mapping study. IET Softw. 14 (5), 468–481.

Ajmani, S., Liskov, B., Shrira, L., 2006. Modular software upgrades for distributed systems. In: European Conference on Object-Oriented Programming. Springer, pp. 452–476.

Baresi, L., Ghezzi, C., Ma, X., La Manna, V.P., 2016. Efficient dynamic updates of distributed components through version consistency. IEEE Trans. Softw. Eng. 43 (4), 340–358.

Baumann, A., Heiser, G., Appavoo, J., Da Silva, D., Krieger, O., Wisniewski, R.W., Kerr, J., 2005. Providing dynamic update in an operating system. In: USENIX Annual Technical Conf.. pp. 279–291.

Burns, B., Beda, J., Hightower, K., 2019. Kubernetes: Up and Running: Dive Into the Future of Infrastructure. O'Reilly Media.

Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.-C., 2007. Polus: A powerful live updating system. In: 29th International Conference on Software Engineering. ICSE'07, IEEE, pp. 271–281.

Cinque, M., Cotroneo, D., De Simone, L., Rosiello, S., 2022. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. Future Gener. Comput. Syst. 129, 315–330.

Dobies, J., Wood, J., 2020. Kubernetes Operators: Automating the Container Orchestration Platform. "O'Reilly Media, Inc.".

Finn, N., 2018. Introduction to time-sensitive networking. IEEE Commun. Stand. Mag. 2 (2), 22–28.

Forbes, H., 2018. The end of industrial automation (as we know it). https://www.arcweb.com/blog/end-industrial-automation-we-know-it.

Foundation, O., 2017. OPC unified architecture: The interoperability standard for industrial automation. https://opcfoundation.org/wp-content/uploads/2017/08/OPC-Foundation-Process-Automation-Whitepaper-20170823.pdf.

Giuffrida, C., Tanenbaum, A.S., 2009. Cooperative update: A new model for dependable live update. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades. pp. 1–6.

Goldschmidt, T., Hauck-Stattelmann, S., Malakuti, S., Grüner, S., 2018. Container-based architecture for flexible industrial control applications. J. Syst. Archit. 84, 28–36.

Gupta, D., Jalote, P., Barua, G., 1996. A formal framework for on-line software version change. IEEE Trans. Softw. Eng. 22 (2), 120–131.

Hicks, M., Nettles, S., 2005. Dynamic software updating. ACM Trans. Program. Lang. Syst. (TOPLAS) 27 (6), 1049–1096.

Hollender, M., 2010. Collaborative Process Automation Systems. ISA.

Koziolek, H., Burger, A., Abdulla, P.P., Rückert, J., Sonar, S., Rodriguez, P., 2021. Dynamic updates of virtual PLCs deployed as kubernetes microservices. In: Proc. 15th European Conf. on Software Architecture. ECSA 2021, Springer, ISBN: 978-3-030-86044-8, pp. 3–19.

Koziolek, H., Burger, A., Platenius-Mohr, M., Jetley, R., 2020. A classification framework for automated control code generation in industrial automation. J. Syst. Softw. 166, 110575. http://dx.doi.org/10.1016/j.jss.2020.110575.

Koziolek, H., Burger, A., Platenius-Mohr, M., Rückert, J., Stomberg, G., 2019. OpenPnP: A plug-and-produce architecture for the industrial Internet of Things. In: Proc. 41st Int. Conf on Software Engineering, ICSE. SEIP, IEEE / ACM, pp. 131–140. http://dx.doi.org/10.1109/ICSE-SEIP.2019.00022.

Kramer, J., Magee, J., 1990. The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. 16 (11), 1293–1306.

Krause, H., 2007. Virtual commissioning of a large LNG plant with the DCS 800xa by ABB. In: 6th EUROSIM Congress on Modelling and Simulation, Ljubljana, SlovÉNie.

Levine, W.S., 2018. The Control Handbook (Three Volume Set). CRC Press.

Ma, L., Yi, S., Carter, N., Li, Q., 2019. Efficient live migration of edge services leveraging container layered storage. IEEE Trans. Mob. Comput. 18 (9), 2020–2033.

Machen, A., Wang, S., Leung, K.K., Ko, B.J., Salonidis, T., 2017. Live service migration in mobile edge clouds. IEEE Wirel. Commun. 25 (1), 140–147.

Mahnke, W., Leitner, S.-H., Damm, M., 2009. OPC Unified Architecture. Springer Science & Business Media.

Makris, K., Ryu, K.D., 2007. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. pp. 327–340.

Miedes, E., Muñoz-Escoí, F., 2012. A survey about dynamic software updating, vol. 46022, Instituto Universitario Mixto Tecnologico de Informatica, Universitat Politecnica de Valencia, Campus de Vera s/n.

Moga, A., Sivanthi, T., Franke, C., 2016. Os-level virtualization for industrial automation systems: Are we there yet? In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. pp. 1838–1843.

Mugarza, I., Mugarza, J.C., 2021. Cetratus: Live updates in programmable logic controllers. In: 2021 IEEE International Workshop of Electronics, Control, Measurement, Signals and Their Application to Mechatronics. ECMSM, IEEE, pp. 1–7.

Netto, H.V., Lung, L.C., Correia, M., Luiz, A.F., de Souza, L.M.S., 2017. State machine replication in containers managed by kubernetes. J. Syst. Archit. 73, 53–59.

Oh, S., Kim, J., 2018. Stateful container migration employing checkpoint-based restoration for orchestrated container clusters. In: 2018 International Conference on Information and Communication Technology Convergence. ICTC, IEEE, pp. 25–30.

Patrick, L.M., 2007. Collaborating for success: Innovative Dow/ABB relationship provides mutual benefits for manufacturers in all industries. ABB Rev. (3), 15–17, URL https://bit.ly/3cAQFOd.

Plaza, I., Medrano, C., Blesa, A., 2006. Analysis and implementation of the IEC 61131-3 software model under POSIX real-time operating systems. Microprocess. Microsyst. 30 (8), 497–508.

Prenzel, L., Provost, J., 2017. Dynamic software updating of IEC 61499 implementation using erlang runtime system. IFAC-PapersOnLine 50 (1), 12416–12421.

Previtali, S.C., 2007. Dynamic updates: Another middleware service? In: Proceedings of the 1st Workshop on Middleware-Application Interaction: In Conjunction with Euro-Sys 2007. pp. 49–54.

Qiu, Y., Lung, C.-H., Ajila, S., Srivastava, P., 2017. LXC container migration in cloudlets under multipath TCP. In: 2017 IEEE 41st Annual Computer Software and Applications Conference. COMPSAC, vol. 2, IEEE, pp. 31–36.

Seifzadeh, H., Abolhassani, H., Moshkenani, M.S., 2013. A survey of dynamic software updating. J. Softw.: Evol. Process 25 (5), 535–568.

Sollfrank, M., Loch, F., Denteneer, S., Vogel-Heuser, B., 2020. Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. IEEE Trans. Ind. Inform. 17 (5), 3566–3576.

Stoyle, G., Hicks, M., Bierman, G., Sewell, P., Neamtiu, I., 2007. Mutatis mutandis: Safe and predictable dynamic software updating. ACM Trans. Program. Lang. Syst. (TOPLAS) 29 (4).

Subramanian, S., Hicks, M., McKinley, K.S., 2009. Dynamic software updates: A VM-centric approach. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1–12.

Tiegelkamp, M., John, K.-H., 2010. IEC 61131-3: Programming Industrial Automation Systems. Springer.

Vayghan, L.A., Saied, M.A., Toeroe, M., Khendek, F., 2019. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 176–185.

Venkatesh, R.S., Smejkal, T., Milojicic, D.S., Gavrilovska, A., 2019. Fast in-memory CRIU for docker containers. In: Proceedings of the International Symposium on Memory Systems. pp. 53–65.

Wahler, M., Oriol, M., 2014. Disruption-free software updates in automation systems. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation. ETFA, IEEE, pp. 1–8.

Wahler, M., Richter, S., Kumar, S., Oriol, M., 2011. Non-disruptive large-scale component updates for real-time controllers. In: 2011 IEEE 27th International Conference on Data Engineering Workshops. IEEE, pp. 174–178.

Wahler, M., Richter, S., Oriol, M., 2009. Dynamic software updates for real-time systems. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades. pp. 1–6.

Yu, C., Huan, F., 2015. Live migration of docker containers through logging and replay. In: 2015 3rd International Conference on Mechatronics and Industrial Informatics. ICMII 2015, Atlantis Press, pp. 623–626.

Heiko Koziolek is a Corporate Research Fellow with ABB Research in Ladenburg, Germany. His research interests include software architecture, performance engineering, and empirical software engineering.

Andreas Burger is Cluster Lead Distributed Systems at Bosch Research in Renningen, Germany. His research interests include embedded systems engineering, IoT systems, and software architecture.

Abdulla PP is a software engineer for ABB Process Automation. His research interests include machine learning, software design, and automation engineering.