Dynamic Updates of Virtual PLCs Deployed as Kubernetes Microservices

Heiko Koziolek[1(✉)], Andreas Burger[1], P. P. Abdulla[2], Julius Rückert[1], Shardul Sonar[1], and Pablo Rodriguez[1]

[1] ABB Research Germany, Ladenburg, Germany
heiko.koziolek@de.abb.com
[2] ABB Research India, Bangalore, India

Abstract. Industrial control systems (e.g. programmable logic controllers, PLC or distributed control systems, DCS) cyclically execute control algorithms to automated production processes. Nowadays, for many applications their deployment is moving from dedicated embedded controllers into more flexible container environments, thus becoming "Virtual PLCs". It is difficult to update such containerized Virtual PLCs during runtime by switching to a newer instance, which requires transferring internal state. Former research has only proposed dynamic update approaches for single embedded controllers, while other work introduced special Kubernetes (K8s) state replication approaches, which did not support cyclic real-time applications. We propose a dynamic update mechanism for Virtual PLCs deployed as K8s microservices. This approach is based on a purpose-built K8s Operator and allows control application updates without halting the production processes. Our experimental validation shows that the approach can support the internal state transfer of large industrial control applications (100.000 state variables) within only 15% of the available cycle slack time. Therefore, the approach creates vast opportunities for updating applications on-the-fly and migrating them between nodes in a cloud-native fashion.

Keywords: Software architecture · PLC programs · Kubernetes · Docker · Microservices · Kubernetes Operator · Performance evaluation · Stateful applications · Dynamic software updates · OPC UA

1 Introduction

PLCs and DCSs are at the heart of many industrial production processes, such as power generation, mining, chemical production, or paper production [6]. The DCS is market size is at 13.4 BUSD in 2020 and expected to grow significantly in the next few years [3]. PLCs and DCSs receive telemetry data from sensors and cyclically run control algorithms that produce output signals for various actuators, such as motors, pumps, mixers, reactors, heat exchangers, etc. This typically relies on embedded controllers running on purpose-built hardware for high reliability.

S. Biffl et al. (Eds.): ECSA 2021, LNCS 12857, pp. 3–19, 2021.
https://doi.org/10.1007/978-3-030-86044-8_1

However, in recent years, more and more automation customers are starting to adopt server-hosted PLC programs running in container frameworks, due to their reduced costs and easier application management.

Updating PLC/DCS programs shall ideally not require a production stop, which can incur high costs for the associated machinery and processes. Patching PLC runtimes or applications is therefore undesired and only rarely done. Container orchestration (e.g., Kubernetes) allows switching to an updated virtual runtime or application in another container "on-the-fly", by transferring the signal input subscriptions and output publications. However, the newly started runtime or program need to work with the same internal state (i.e. a set of variables storing intermediate calculations) as the former runtime. Due to the short execution cycles (e.g., every 100 ms), the required state transfer from container to container needs to be fast, so that the control actuators continuously receive their control signals without interruption.

Researchers have formerly proposed dynamic update approaches for control applications (e.g. [15,19,20]), but these works were limited to single embedded controllers and could not utilize the advanced orchestration concepts of a container systems. Other researchers validated that PLC programs can achieve their real-time behavior if deployed as software containers (e.g. [5,11,15]), but did not investigate updates involving state transfers. Specifically, for container orchestration systems, there are approaches for state replication (e.g., [13,14,18], which however do not involve PLC programs with short execution cycle times.

We propose a novel state-transfer approach for dynamic updates of Virtual PLCs deployed as K8s microservices. The contributions of this paper are 1) a conceptual architecture for a state transfer method that utilizes the capabilities of a container orchestration framework, 2) a procedure for state transfer across network nodes while adhering to industry standards, and 3) a rationalization of the design decisions for the entire architecture. The approach allows to update both PLC programs and runtimes during system execution. A K8s Operator monitors a running PLC engine, starts an updated PLC engine in parallel, issues the internal state transfer, and then switches over the input/output signal handling to the updated engine.

To validate the approach, we have implemented the conceptual architecture exemplary based on open-source components (i.e., OpenPLC, Open62541, Cereal, Kubernetes, Docker). In a series of experiments, we simulated updating large control applications derived from existing power production and mining plants with up to 500K internal state variables. For an application with 100K state variables the approach was able to transfer the internal state in less than 15 ms, which is significantly lower than the required cycle time. This validates that the approach can update large running applications as desired.

This paper is structured as follows: Sect. 2 provide an introduction to PLC and DCS controllers, as well as containers and the communication framework OPC UA. Section 3 analyzed related work. Section 4 presents the conceptual architecture and the rationale for the design decisions. Section 5 describes the prototypical implementation and testbed, before Sect. 6 explains the experimental results. Finally, Sect. 7 lists assumptions and limitations underlying the approach.

2 Background

PLCs and DCS controllers are used to automate industrial processes, such as power, paper, or chemical production, as well as mining applications or steel plants [6]. Such controllers often rely on real-time operating systems, such as Embedded Linux, FreeRTOS, or VxWorks. A set of five different programming languages for these controllers was standardized as IEC 61131-3 in the early 1990th, including function block diagrams, structured text, and ladder logic [6]. IEC 61131-3 control runtimes execute algorithms such as 'PID controllers', cyclically, usually with cycle times between 10 and 1000 ms. For safety-critical applications such controllers feature redundant processing units with special purpose hardware failover mechanisms. So-called SoftPLCs (e.g., controller runtimes deployed on workstations or servers) are today mostly used in development and testing. However, a growing market of server-deployed Virtual PLCs is expected thanks to the constantly increasing computing capabilities.

Containers (e.g., LXC, Docker) provide an operating-system-level virtualization layer for Linux processes using namespace isolation and resource limitation with cgroups [7]. They can be used to package applications with their required libraries and are a preferred deployment target for microservices. Applications with many containers can be managed with a container orchestration engine, such as Kubernetes. Stateless microservices in Kubernetes are preferred, since they can be easily horizontally scaled-up to support workloads of large internet applications. Virtual PLCs are stateful services, which are also supported by Kubernetes. Usually they have comparably constant workload and do not require horizontal scaling. Industry analysts speculate that software containers will replace embedded software to a large extent in the future, since they significantly improve managing and updating services in an efficient and less error-prone manner [4].

OPC UA provides a middleware and information modeling for industrial applications [10]. It is designed for monitoring industrial devices from workstations, but was lately extended to also support fast, deterministic controller-to-field device communication [2]. OPC UA includes a client/server protocol on top of TCP/IP, as well as a publish/subscribe mechanism on top of UDP. OPC UA address spaces may hold both configuration and sensor data. The Open Process Automation Forum [12] has identified OPC UA as the core communication mechanism in future open and interoperable industrial control systems. Controllers and certain field devices shall be equipped with OPC UA clients and servers, while legacy field buses shall be integrated via OPC UA gateways.

3 Related Work

Numerous authors have surveyed the field of **dynamic software updating**, which includes many approaches in the last 20 years [1,16]. Specifically, for real-time systems, Wahler et al. [20] proposed a component framework and update

algorithm based on shared memory transfer. In experiments the framework was able to transfer a 4000 byte internal state of a cyclically executing control program below a 5 ms deadline. Later, they extended the work to allow iterative state synchronization over multiple execution cycles [19]. Their mechanism provides more flexibility but can lead to a never terminating state transfer in case of large or very volatile states. Prenzel et al. [15] discussed dynamic updates of IEC 61499 applications but did not transfer internal variables in their experiments. None of these works assumed a container deployment or state transfer across different nodes.

Another line of research is concerned with the deployment of **Virtual PLCs in container environments**, which shall offer more flexibility for updating and portability [4]. Moga et al. [11] compared virtual machines and containers as deployment targets for industrial applications. They ran the microbenchmark 'cyclictest' inside a Docker container on an Intel Xeon E5 and found that the jitter introduced by Docker was below 20 ms, therefore negligible for most industrial applications. Goldschmidt et al. [5] proposed different use cases for containerized Virtual PLCs, among them dynamic updating. They also executed 'cyclictest' inside a Docker container on a Raspberry PI 2 and showed that the average overhead was below 100 ms. Sollfrank et al. [17] deployed a PID controller based on Simulink C-code in Docker onto a Raspberry PI 4 and concluded that it can meet soft real-time requirements, since the container overhead was below 150 ms. These works were mostly concerned with characterizing the overhead introduced by Linux containers, but did not investigate dynamic updates in detail.

Outside the industrial automation domain, researchers have proposed approaches for **state replication in container orchestration engines**. Netto et al. [13] introduced the DORADO protocol to order requests in Kubernetes that are saved to shared memory (i.e., etc.d) and can be exchanged between container replicas. In experiments, they showed that this increases latency and lead to a leveling out throughput but can tolerate container failures. This work assumes a continuous state transfer between redundant replicas, instead of a one-shot transfer to an updated application. Vayghan et al. [18] introduced a State Controller for Kubernetes that can replicate internal state between containers to enable fail-over scenarios. State is stored into a persistent volume and a standby container can take over serving clients in case of the primary container failing. Experiments with a video streaming application, where the transferred state was the client current streaming position, showed that the approach could reduce the service outage time significantly. Oh et al. [14] proposed a checkpoint-based stateful container migration approach in Kubernetes. None of these works is concerned with cyclically executing control applications that need to adhere to short deadlines.

4 State-Transfer Approach

This section describes the architecture of our proposed state-transfer approach. First, a simple example explains the required internal state transfer in more

detail, followed by a description of the static architecture. The dynamic procedure of the state transfer follows next, before the section concludes with a discussion of the included architecture decision points, alternatives, and rational for the made decisions.

4.1 Example

Figure 1 depicts a typical control program in IEC 61131-3 Structured Text (i.e., Pascal-like syntax) from the industrial domain. The function block FT_PIWL refers to a proportional integral (PI) controller with dynamic anti wind-up (WL). It can for example be used to regulate the filling level in a tank or the pressure in a pipe. The program cyclically computes its outputs Y based on the internal state variables *t_last* and *in_last*, e.g., every 100 ms. Its actual execution runtime is usually well below the cycle time, e.g., below 5 ms. For a dynamic update of the program, where for example another host node shall take over the execution, these state variables needs to be transferred within the cycle slack time (e.g., 95 ms) in order not to interrupt the underlying production process.

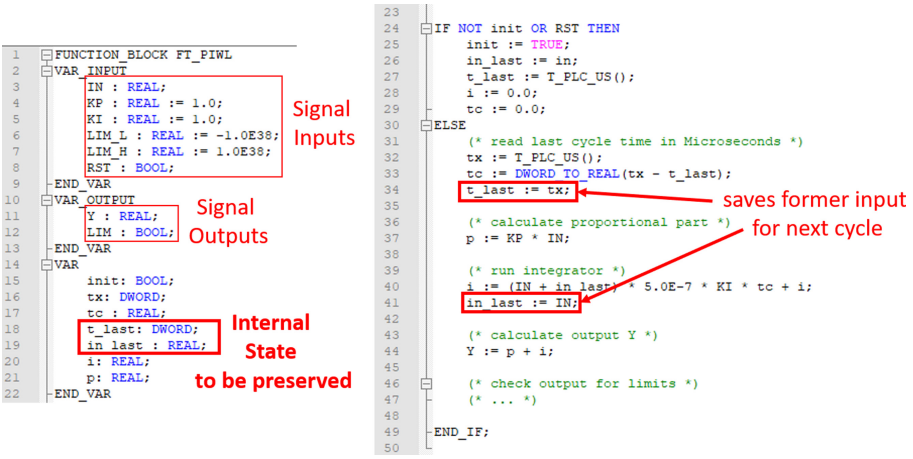

Fig. 1. Example PLC program with internal state to be preserved. The variables *t_last* and *in_last* need to be transferred in case the runtime is updated.

Typical control applications in process automation may contain thousands of such state variables, since they regulate thousands of sensors and actuators within a single production plant. During programming, engineers mark specific variables as "retained", indicating that they need to be saved and restored in case a PLC runtime needs to restart. These retained variables are also used by our approach for dynamic updates at runtime.

4.2 Static View

Figure 2 shows a component and deployment view of our state transfer approach. The figure shows a single master node and two worker nodes as a minimal example. In a full-scale system, many worker nodes and redundant masters would be used. The system features a container orchestration engine (e.g., Kubernetes) in order to flexibly deploy and update Virtual PLCs. Today's physical embedded controllers typically need to be shut down, patched, and restarted in case the runtime system needs updates. Figure 2 depicts typical Kubernetes components required on the nodes in light gray (e.g., Kube API server, kubelet, kube-proxy).

Worker 1 contains a **PLC Runtime System** deployed as a "virtual-plc-pod", a K8s custom resource including a container. The custom resource allows the user to parameterize the pod for real-time specifics. This pod can be considered as a microservice of an entire DCS. Preferably, the PLC Runtime System pods are deployed on a Linux node with the PREEMPT_RT kernel patch to achieve soft real-time behavior and minimize jitter in the cycle periods. The PLC Runtime System can execute IEC 61131-3 applications by consuming input signals and sending output values via the OPC UA protocol, to ensure interoperability with IO devices from different vendors. These OPC UA signals are either directly sent by field devices with OPC UA servers or come from field gateways that translate traditional fieldbus protocols or analog signal connections [8]. Such devices are not depicted in the component view for brevity, as

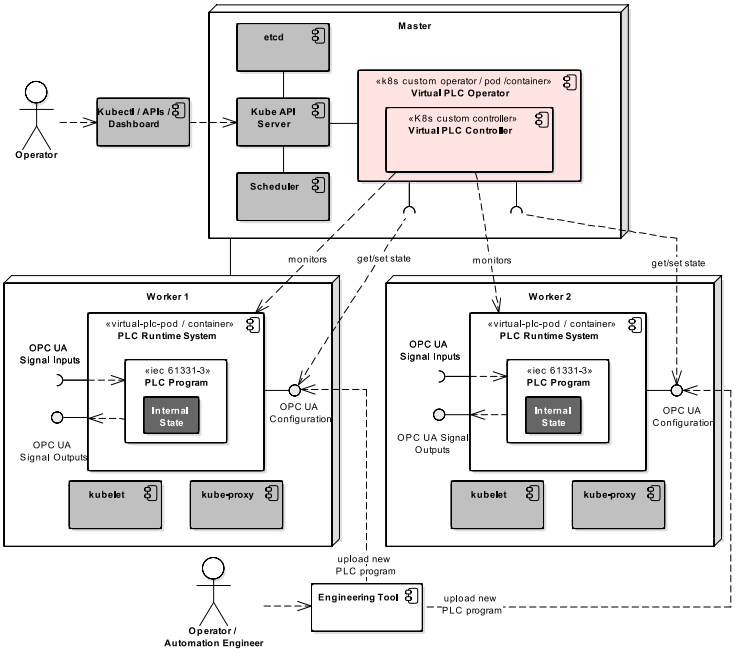

Fig. 2. Virtual PLC state transfer (combined component and deployment diagram)

a single PLC runtime could control hundreds of devices connected via sophisticated network topologies.

An automation engineer can upload a **PLC Program** from an Engineering Tool via the PLC Runtime Systems OPC UA Configuration interface. After start-up, the PLC Program carries **Internal State** in the form of numerous state variables as illustrated in Sect. 4.1. The PLC Runtime System here provides a mechanism to serialize the current values of this Internal State and send it via its OPC UA Configuration interface to any OPC UA client.

Our approach features a custom K8s Operator called **Virtual PLC Operator**. This relies on a standard extension mechanism for K8s[1]. The operator contains a custom K8s controller called **Virtual PLC Controller** that monitors **Virtual-PLC-Pods** and handles their lifecycle via the K8s API as well as their OPC UA configuration interfaces.

The **Virtual PLC Controller** has three options to detect a dynamic update request triggering a state transfer:

1. **Updated PLC program:** The controller can detect if an automation engineer uploaded an updated PLC program via the Engineering Tool. Updated programs often include only minor changes of the internal state structure; thus the internal state of the former program can be largely mapped to the updated program. Newly introduced variables are set to default values, removed variables are discarded.
2. **Updated PLC runtime:** The controller can detect if an operator has pushed a new container image of the **PLC Runtime System** to the container registry. This can be caused by a changed configuration of the runtime system or a new compiled version due to bug fixes or security patches.
3. **Updated host:** The controller can detect if K8s scheduled restarting a PLC Runtime System pod on another node for carrying out maintenance on the former node. For example, the administrator may have selected a node for an operating system update or a hardware replacement.

If such an update request is registered, the **Virtual PLC Controller** follows the procedure depicted in Fig. 3.

4.3 Dynamic View

After detecting and validating the update request (Step 1), the Virtual PLC Controller starts the pod of the PLC Runtime System 2 (Step 2), possibly on a separate node (here: worker 2). The controller connects to both PLC Runtime systems via OPC UA and uploads the desired application to the PLC Runtime System 2. It also connects the OPC UA IO *input* signals to this runtime by setting up according subscriptions, so that it can start executing based on live inputs. *Output* signals will be computed but not published to the field devices, as long as the state transfer has not been executed and the correct execution is verified.

[1] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

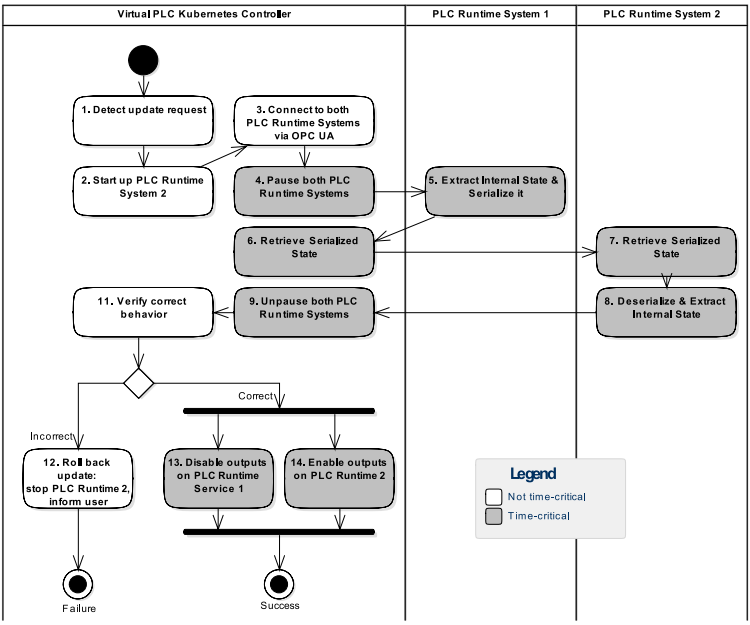

Fig. 3. Dynamic Flow of the state transfer (UML activity diagram)

After this less time-critical initialization, the controller enters a time-critical phase in step 4–9 that needs to complete within the cycle slack time. The controller pauses the execution of both runtimes (Step 4) in order to prevent updates to the internal state. The controller extracts the internal state (e.g., the values of *t_last* and *in_last* in Fig. 1) from runtime system 1, which includes serialization to a binary large object (step 5). Via the OPC UA configuration interface and the method `GetState`, the controller retrieves the internal state (step 6) and sends it the runtime system 2 (step 7) using its `RetrieveState` method. This runtime system deserializes and extracts the state, overwriting the corresponding internal state variables (step 8). Finally, the controller resumes both runtime systems, so that they continue executing their control algorithms to compute outputs.

With both engines running, but only runtime 1 publishing output signals to the devices, the controller verifies the correct behavior or runtime 2 by comparing its computed output values to the ones from runtime 1. If runtime 2 produces incorrect values (e.g., by drifting beyond a predefined threshold), the controller rolls back the entire update, stops runtime 2 and informs the user (step 12) of an update failure. If the values are correct and no substantial drift between the values of both runtimes is detected, the controller disables output signal publishing of runtime 1 (step 13) and activates the output signal publishing by runtime 2 (step 14). In this case the update was successful. PLC runtime 1 can be terminated and archived to allow for rollbacks later on.

4.4 Decision Points

The state transfer architecture includes a number of decisions, which we rationalize in the following.

Transfer Mechanism: The transfer mechanism is a core component to enable a reliable, flexible and fast state transfer. As state transfer shall be possible through node/pod boundaries and fit into the open process architecture defined by OPAF, we selected OPC UA as transfer mechanism. OPC UA is purpose-built for industrial applications and provides interoperability as virtual control engines will require to have built-in OPC UA servers. Additionally, by using the already required OPC UA and not exposing additional HTTP, MQTT, or proprietary sockets, the VirtualPLC attack surface is limited. OPC UA can be combined with TSN to provide deterministic real-time communication.

Serialization Mechanism: We decided to transfer the application state as a binary large object (BLOB). A BLOB allows to abstract the internal application memory structure, which is important as the internal memory state structure might be vendor-specific. Hence, using a BLOB instead of structured data, allows to keep the interface of the state transfer service stable for different virtual PLCs. Additionally, it gives the ability to use data compression techniques or encryption.

State Transfer Service as K8s Extension: The state transfer service is a K8s Operator (Virtual PLC Operator) running in an own pod. It should not directly be built into the virtual PLC runtimes, as this would require code duplication in each engine. Ideally, the PLC runtime is not aware of the Virtual PLC Operator and simply exposed the OPC UA interfaces anyhow required by OPAF. Hence, the state transfer is seamlessly integrated in the Kubernetes infrastructure and complexity is hidden from the user by not having a separate interface in the Kubernetes microservice architecture.

Container Virtualization: The state transfer service is designed to be used in a virtualization context by exchanging the application state between different virtual PLCs, deployed as docker container or Kubernetes pods. The architecture and process of the state transfer service can also be used without containers, but this scenario would need an orchestration instance/component which handles the network configuration for the virtual PLC engines and enable the startup and shutdown of those. In general, the K8s system eases the state transfer, the orchestration of virtual control engines and the scheduling significantly.

5 Prototypical Implementation

We prototypically implemented the architecture using open source components. This shall ensure repeatability of the experimental evaluation, which is thus not constrained to commercial software. The concepts behind the architecture also apply to commercial components (Fig. 4).

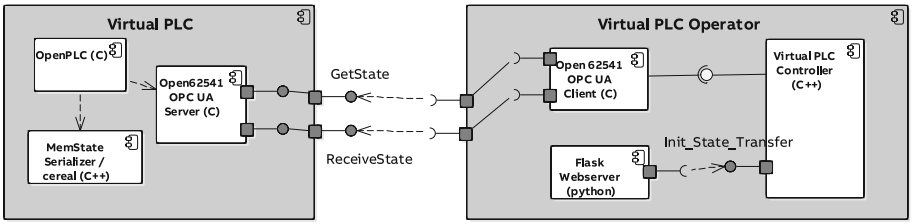

Fig. 4. Prototypical implementation of the state transfer approach

Virtual PLC: We selected OpenPLC[2] as the PLC Runtime System of our prototype. As OpenPLC did not have an integrated OPC UA server to receive and sent signal values, we enhanced the engine by adding an OPC UA server based on the open62541 stack[3]. The OPC UA server was integrated in parallel to the control engine and other components. It shows all variables which are available for the actual running control program as well as the required interface methods for the state transfer. Additionally, we implemented functionality for retrieving the application state out of OpenPLC's internal memory as well as writing back an application state. Therefore, the functionality to pause and resume the entire control engine was introduced and implemented. We also added a new component to OpenPLC, the so called MemState Serializer. This component takes the retrieved application state and serializes it to an array of bytes as well as deserializes it to the necessary internal memory structure. Keeping this component separate from the control engine allows us to exchange the underlying serialization library without touching the main functionality of the Virtual PLC. In the prototypical implementation we used a fast, open source C++ serialization library called Cereal[4].

Virtual PLC Operator: In order to evaluate the proposed state transfer approach, we implemented the state transfer service in C++. The operator contains three main components, a webserver, an OPC UA client and the Virtual PLC Controller. The state transfer service is implemented in C++ by using an open source OPC UA stack, open62541 and a webserver implemented in python, based on Flask[5]. The small webserver allows the user to communicate with the service through a web interface as well as through a REST interface. Two main functions are realized by the integrated Open62541 OPC UA server, a functionality to get an serialized application state from a virtual engine (here called GetState) and another function to send back an serialized application state to a virtual engine (here called ReceiveState).

[2] https://www.openplcproject.com/.
[3] https://open62541.org/.
[4] https://uscilab.github.io/cereal/.
[5] https://flask.palletsprojects.com/.

Virtual PLC Controller: The core component of the state transfer takes all necessary inputs from the webserver or the REST interface and starts initiating the state transfer by using the OPC UA client and the required interfaces. The controller receives the serialized application state, which should be transferred from one virtual engine to the other, via the GetState interface and then send it to the receiving engine by using the ReceiveState interface. Hence, this component ensures that all data is transferred properly and the sending and receiving virtual PLCs confirm the successful transfer. In order to guarantee real-time requirements, the controller is implemented as a separate POSIX thread with high real-time priorities. This ensures a deterministic execution of the state transfer, if deployed in a real-time capable operating system.

Software Platform: Finally, the Virtual PLC Operator as well as the Virtual PLC are put into Docker containers and K8s pods. We used the open source StarlingX[6] cloud infrastructure software stack that includes Kubernetes and Docker. StarlingX includes CentOS as Linux OS and nodes can be configured to use a PREEMPT_RT patched kernel for deterministic execution using real-time priorities. However, the results should be similar on other K8s platforms.

Hardware Environment: Our testbed consisted of two master servers ("controllers") in a high availability deployment as well as two worker nodes. All servers were connected using Gigabit Ethernet. Experiments utilized the worker nodes, which were made up of HPE ProLiant DL380 Gen10 Rack Servers with Dual Intel Xenon Silver 4110 CPUs (40 cores) and 256 GB of RAM.

6 Experimental Evaluation

6.1 Test Application Sizing

To make our experimental setting realistic, we reviewed several larger industrial control applications to determine typical cycle times and application sizes.

The control application for one of the largest Liquid Natural Gas (LNG) plants was reviewed by Krause [9] and characterized to comprise of 650.000 variables distributed to 18 different hardware controllers and 10.000 IO devices (≈30.000 variables per controller and 550 IO per individual controller). The cycle time of these controllers was 500 ms.

A different project from the area of mining also used around 500 IO-devices per and almost 200.000 variables in total. Here, the configured cycle time was 250 ms. A third project for a chemical plant had 350 IO-devices per controller and around 100.000 variables in the control programs, while running on a 1000 ms cycle time.

[6] https://www.starlingx.io/.

From these applications it becomes clear that the application sizes and cycles times vary from domain to domain. Furthermore, the number of variables may reflect the sophistication of computations needed for a given application. We decided to follow a conservative approach and assume a cycle time of 100 ms for our experiments. In addition, we aimed a state transfer of around 100.000 variables within the cycle slack time. We also assume that all variables are marked as "retained", meaning that they are included in the state transfer, although in practice a much lower number of variables is usually retained. If these conditions are met, then most of the typical industrial application should be compatible with our state transfer approach.

We assume each of the 100.000 variables encoded with 32 bits, thus aiming at a internal state size of around 400 KByte plus metadata. Assuming that the control algorithms need 10% of the cycle time for their computation (i.e., 10 ms), then these 400 KByte need to be transferred in less than 90 ms, resulting in a minimum transfer rate of about 4.5 MByte/sec. Typical RAM, bus, and network bandwidths are far beyond this threshold, so the main bottleneck is expected to be the CPU and possibly the network latency.

6.2 Jitter Characterization

To characterize the real-time properties of our hosts used for running the experiments, we executed the tool 'cyclictest' of the Linux Foundation[7]. It is commonly used to benchmark real-time systems. Cyclictest measures the time between a thread's intended and actual wake-up time, which can include latencies in the hardware or operating system. To simulate meaningful real-time stress conditions, we also used the Linux tool 'stress'[8] as follows:

```
stress -i 40 -c 40 -d 40 --hdd-bytes 20M -m 40 --vm-bytes 10
cyclictest -l100000 -m -Sp99 -i200 -h400 -q >output
```

The tool thus starts as many CPU-intensive and IO-intensive threads as given CPU cores and also uses the hard-disk, resulting in a 100% utilized system. Cyclictest is repeated 100K times with the highest thread priority 99. Figure 5 shows a histogram of the results.

The worst-case latency was 315 ms, which can be compared to other platforms in the OSADL Real-time QA Farm[9]. Since our experiments use cycles times at 100 milliseconds derived from practical application cases, we deem the jitter of our test host node negligible and not interfering with our state measurement experiments.

[7] https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start.
[8] https://manpages.ubuntu.com/manpages/eoan/man1/stress.1.html.
[9] https://www.osadl.org/Latency-plots.latency-plots.0.html.

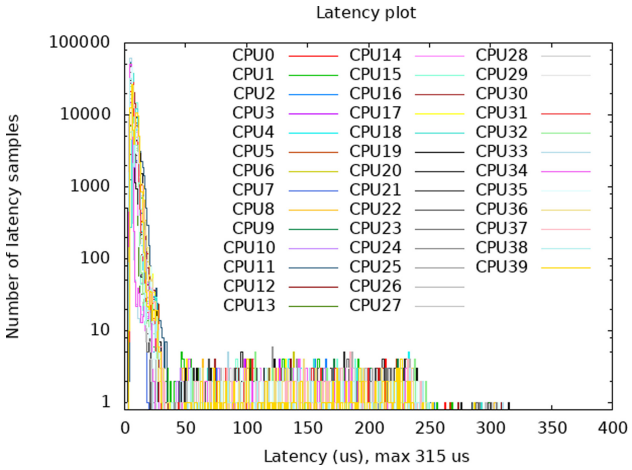

Fig. 5. Cyclictest results (histogram)

6.3 State Transfer Time

To determine the feasibility of transferring a realistic internal state within a cycle slack time of 90 ms, we deployed the prototypical implementation of our dynamic update approach on two worker nodes within our K8s cluster. One OpenPLC runtime ran on the first node, while the Virtual PLC Controller and the second OpenPLC runtime ran on the second node. We uploaded a generated test application with a 100 ms cycle time to the OpenPLC runtime with a varying size of internal state variables. In our experiments, we measured the time for the critical serialization, state transfer, and deserialization (step 5–8 in Fig. 3) from the Virtual PLC Controller.

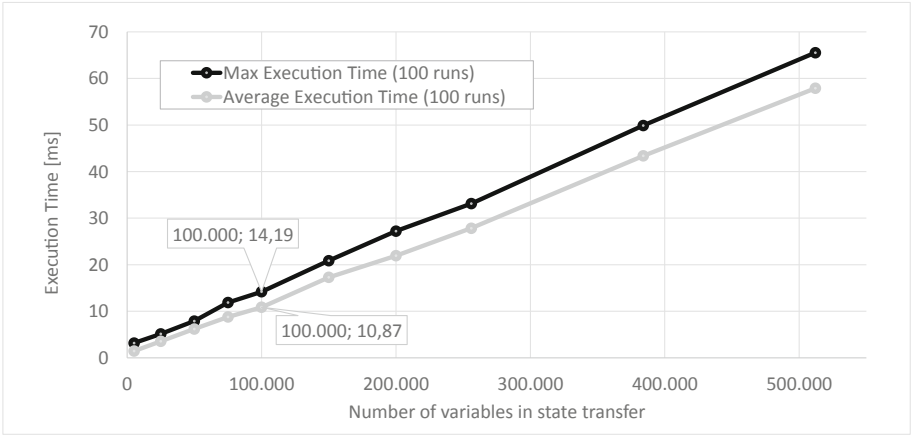

Fig. 6. State transfer times via OPC UA for different number of variables: 100K variables can be transferred in less than 15 ms, well below the cycle slack time of 90 ms.

For each state size from 5000 to 500.000 variables, we executed 100 experiment-runs to account for outliers and distortions in the K8s cluster. Figure 6 shows these aggregated state transfer measure, including both the maximum measured time and the average time for 100 runs. The results show that our dynamic update approach could transfer the internal state of 100.000 variables (400 KByte) between the two runtimes and nodes via OPC UA in 10.87 ms on average. The longest run took 14.19 ms. This duration is well below the target cycle slack time of 90 ms (approx. 16% of the cycle slack time).

Within 90 ms, approx. 600.000 variables could be transferred via network boundaries within the cycle slack time, which thus provides room for very large application. However, not fully utilizing the slack time is a safe way to assure meeting soft real-time guarantees, also under less optimal conditions (e.g., interfering workloads).

To characterize the jitter and find out where the state transfer execution time is spent, we ran an additional experiment of 1000 runs for the state size of 100.000 variables. Figure 7 shows the execution times for the serialization (step 5), the retrieval of the state by the Virtual PLC Controller (step 6), the retrieval of the state by the second OpenPLC runtime (step 7), and the deserialization (step 8).

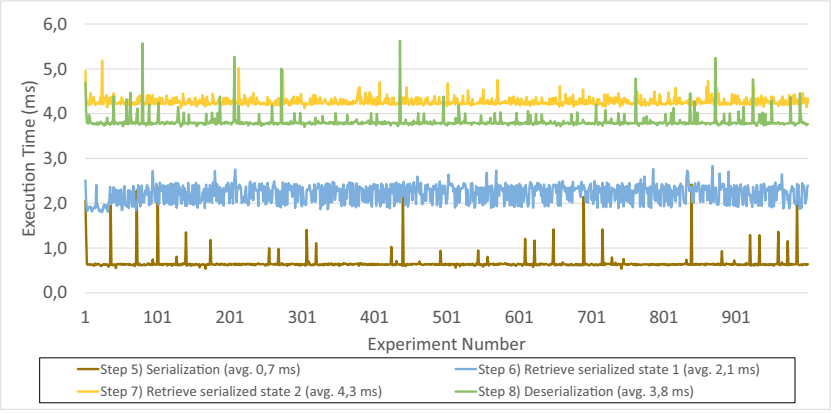

Fig. 7. Execution times for step 5) to step 8)

Serialization and deserialization show a mostly constant execution time with occasional outliers that contribute to the overall state transfer times. It is assumed that the processes may have been preempted during these outliers, which could possibly be addressed with better fine-tuned real-time thread-priorities. The deserialization (step 8) was on average more than 5 times slower than the serialization (step 5), 3.8 ms vs. 0.7 ms. We profiled the code using Valgrind and Callgrind and traced the different times to the external library Cereal used for serialization and deserialization, which is inefficiently implemented for

the deserialization of arrays by using string stream-buffers. Other serialization libraries could be tested in this context.

The network transfer times in Fig. 7 are 4.3 ms and 2.1 ms on average. The figure shows that these transfers are much less constant than serialization/deserialization and introduce a additional jitter around one millisecond to the overall state transfer time. This could be attributed to the non-prioritized TCP/IP stacks in the real-time operating system as well as the missing prioritization of network packets in the TCP/IP connection. Using time-sensitive networks (TSN) with appropriate priority classes could be applied to reduce this jitter for even more deterministic state transfer times.

7 Assumptions and Limitations

The following assumptions and limitations are underlying our approach:

- The approach is not applicable and not meant for fast machine control applications in discrete automation requiring sub-millisecond cycle times.
- We used comparably fast server hardware, which currently are often not be available in many smaller production plants, but which is however also required for running non-trivial container workloads in an orchestration system.
- Our experiments included only simulated IO devices, but not real devices communicating with the OpenPLC runtime.
- Our prototypical implementation relied only on open source components. Other OPC UA SDKs, serialization libraries, or PLC runtimes coming from open source or commercial software could potentially achieve better or worse results.
- We restricted the approach to an OPC UA based state transfer. We used client/server communication via TCP/IP in our setup, while pub/sub communication via UDP could be faster. Alternative communication protocols could streamline the state transfer.
- We set all processes in the implementation to the highest real-time thread priorities, which can result in suboptimal jitter.
- The prototypical implementation did not include the verification step 11), which has been shown in other works (e.g., [19], see monitoring component).

Some of these limitations remain conceptual, others could be addressed in future work and additional experiments.

8 Conclusions

We have introduced a novel approach that allows updating industrial control applications at runtime. Automation engineers can much more flexibly change parameters of their control algorithms in order to optimize the automation of many production processes. These algorithmic optimizations are nowadays based

on large-scale data analytics and can contribute to major production cost savings. The operators do not need to stop the production processes but can perform the updates on-the-fly, which lowers the barrier to consider updates significantly.

In the Industrial Internet-of-Things (IIoT), more and more sensors, actuators, and controllers are equipped with IP connectivity, requiring continuous security updates. The approach thus also enables updating the PLC runtimes systems to fix bugs and security issues, since the K8s Operator can deploy an updated container image and transfer the state. Furthermore, using the mechanism the Virtual PLC can be moved in a cloud-native fashion between nodes, so that operating systems and container engines can be updated, or hardware can be replaced. This was previously impossible on-the-fly using embedded controllers.

As a next step, we plan to address several of the approach's assumptions, namely testing with other PLC runtimes and using more resource-constrained and thus less costly hardware. The approach could in principle also support fail-over scenarios to provide redundancy, which however require a continuous state transfer. Furthermore, the approach could be extended to optimally select a target node for the updated PLC runtime or to transfer the internal state in smaller chunks to be applicable for special control applications with very low cycle times.

References

1. Ahmed, B.H., Lee, S.P., Su, M.T., Zakari, A.: Dynamic software updating: a systematic mapping study. IET Softw. **14**(5), 468–481 (2020)
2. Burger, A., Koziolek, H., Rückert, J., Platenius-Mohr, M., Stomberg, G.: Bottleneck identification and performance modeling of OPC UA communication models. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, pp. 231–242 (2019)
3. Forbes, H., C.D.A.A.G.: Distributed Control Systems Global Market 2017–2022. ARC Market Analysis (2018). https://www.arcweb.com/market-studies/distributed-control-systems
4. Forbes, H.: The end of industrial automation (as we know it), December 2018. https://www.arcweb.com/blog/end-industrial-automation-we-know-it
5. Goldschmidt, T., Hauck-Stattelmann, S., Malakuti, S., Grüner, S.: Container-based architecture for flexible industrial control applications. J. Syst. Architect. **84**, 28–36 (2018)
6. Hollender, M.: Collaborative process automation systems. ISA (2010)
7. Kocher, P.S.: Microservices and Containers. Addison-Wesley Professional (2018)
8. Koziolek, H., Burger, A., Platenius-Mohr, M., Rückert, J., Stomberg, G.: Openpnp: a plug-and-produce architecture for the industrial internet of things. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 131–140. IEEE (2019)
9. Krause, H.: Virtual commissioning of a large LNG plant with the DCS 800XA by ABB. In: 6th EUROSIM Congress on Modelling and Simulation, Ljubljana, Slovénie (2007)
10. Mahnke, W., Leitner, S.H., Damm, M.: OPC Unified Architecture. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-68899-0

11. Moga, A., Sivanthi, T., Franke, C.: 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 (2016)
12. Montague, J.: OPAF draws nearer to interoperable process control, March 2020. https://www.controlglobal.com/articles/2020/opaf-draws-nearer-to-interoperable-process-control/
13. Netto, H.V., Lung, L.C., Correia, M., Luiz, A.F., de Souza, L.M.S.: State machine replication in containers managed by kubernetes. J. Syst. Architect. **73**, 53–59 (2017)
14. Oh, S., Kim, J.: Stateful container migration employing checkpoint-based restoration for orchestrated container clusters. In: 2018 International Conference on Information and Communication Technology Convergence (ICTC), pp. 25–30. IEEE (2018)
15. Prenzel, L., Provost, J.: Dynamic software updating of IEC 61499 implementation using erlang runtime system. IFAC-PapersOnLine **50**(1), 12416–12421 (2017)
16. Seifzadeh, H., Abolhassani, H., Moshkenani, M.S.: A survey of dynamic software updating. J. Softw. Evol. Process **25**(5), 535–568 (2013)
17. Sollfrank, M., Loch, F., Denteneer, S., Vogel-Heuser, B.: Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. IEEE Trans. Industr. Inf. **17**(5), 3566–3576 (2020)
18. Vayghan, L.A., Saied, M.A., Toeroe, M., Khendek, F.: Microservice based architecture: towards high-availability for stateful applications with kubernetes. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), pp. 176–185. IEEE (2019)
19. Wahler, M., Oriol, M.: Disruption-free software updates in automation systems. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), pp. 1–8. IEEE (2014)
20. Wahler, M., Richter, S., Oriol, M.: Dynamic software updates for real-time systems. In: Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades, pp. 1–6 (2009)