Updating Industrial Automation Software in Cloud-Native Environments

Master Thesis

By

Shardul Sonar

Matriculation No - 11011792

October 31st, 2020

SRH University Heidelberg

ABB Corporate Research Center, Ladenburg

Supervisor 1 (SRH University Heidelberg)	Prof. Christoph Hahn
Supervisor 2 (SRH University Heidelberg)	Prof. Dr. Gerd Moeckel
Supervisor 3 (ABB Corporate Research)	Dr.-Ing. Heiko Koziolek
Supervisor 4 (ABB Corporate Research)	Dr.-Ing. Julius Rückert

Affidavit

Herewith I declare:

- that I have independently composed the chapters of the Master Thesis for which I am named as the author.
- that I did not use any other sources and additives, but the ones specified.
- that I did not submit this work for any other examination procedure.

_____

Heidelberg, October 26, 2020.

Acknowledgements

I would like to express my deep gratitude to Dr.-Ing. Heiko Koziolek and Dr.-Ing. Julius Rückert, my research supervisors from ABB Ladenburg, for their constant support and encouragement, and valuable suggestions provided during my research work. I would like to acknowledge Dr. Andreas Burger for his valuable technical support on this project. I would also like to thank the staff of ABB Corporate Research Center, Ladenburg for enabling me with all the necessary technical equipment whenever required.

I wish to show my gratitude to Prof. Dr. Gerd Moeckel and Prof. Christoph Hahn, my supervisors from SRH University Heidelberg for their assistance and constructive feedback on my research. I wish to acknowledge the help provided by my colleagues and friends at SRH University Heidelberg, for their valuable suggestions on my work.

Finally, I would like to offer special thanks to my parents, Mr. Chandrashekhar Sonar and Mrs. Vandana Sonar, and my family in India for their constant support during my research tenure.

Abstract

With the development of internet-of-things and initiatives like the Industry 4.0 the need for inter-connected cyber-physical systems in industrial automation has emerged. Programmable logic controllers (PLC's) and industrial control applications which are the key components of an industrial automation system have been updated to support these technologies, but they still use dedicated hardware. With the increase in the use of internet-based technologies in industrial automation components, the frequency of updates (most of which are security updates) to these systems has also increased. Updating software for PLC's running in dedicated hardware is a complex and error-prone task. Cloud-native technologies, especially container orchestration systems like Kubernetes provide functionality for disruption-free software updates. Therefore, using cloud-native software for running control applications and PLC's can help to deliver faster, easier and error-free updates. Currently, there are different concepts implemented for running industrial automation software in application containers, but none of these concepts are tested with container orchestration systems.

This thesis presents a study of how different update mechanisms in container orchestration system can be used for providing manual or automatic updates to industrial control applications. A, so far missing, state transfer application that is required to transfer the internal state between control applications to enable a seamless execution during updates in a container orchestration system was designed, developed and tested as a part of this thesis. The state consists of internal variables, (for example, counter variables) in a control application which must be retained during the update. Extensive experiments for representative use cases to test the performance of the state transfer application and determine its limits were performed. The results indicate that industrial control applications with cycle times as low as 200 ms and having up to 23,000 variables can be updated without any disruptions using the state transfer application designed. These results show that the technology is promising and could be used in production for a wide range of relevant use cases. However, the study also shows that today's software-based PLC runtimes need to be prepared and optimized for such a setup.

Keywords : IIoT, Industry 4.0, Container Orchestration, Kubernetes, OPC UA, OpenPLC

Table of Contents

1 Introduction

1.1 Topic Overview and Motivation

The use of internet-based technologies is becoming common among industrial automation systems. With initiatives like Industry 4.0, factories have started to develop into intelligent systems where the gap between the digital world and physical world is becoming smaller (Okano, 2017). Due to this, more and more industrial devices are connected in an industrial environment and as a result the term Internet of Things (IoT) is also applied to the industrial domain which is also referred as Industrial Internet of Things (IIoT) (Gilchrist, 2016).

The field level devices like sensors and actuators and the control level devices like programmable logic controllers (PLC's) run on a dedicated hardware which is designed to sustain harsh industrial environments. With the introduction of industrial communication protocols like OPC UA[1], automation component manufacturers have started integrating the support of these protocols in the field level and control level devices, so that they can be connected to the network via Ethernet-based communication (Bruckner et al., 2019).

The field devices, machines and production systems are comprised as cyber-physical systems (Weyer et al., 2015). With the increase in the use of such interconnected cyber-physical systems, the need for keeping such systems up to date is necessary as some of these updates are also security patches which protect these systems from cyber-attacks (Mugarza et al., 2018). These updates are mostly done by the respective domain experts who may or may not have a detailed understanding of the industrial automation systems. For industrial automation components, the updates must be done without any disruption to the physical processes in the field.

For example, in an industrial boiler, different parameters are monitored like fluid level, pressure, temperature etc. and the boiler control process, based on these parameters controls actuators like pneumatic valves, in order to maintain a steady supply of the heated fluid. During an update the boiler must not be shut down and so this control

[1] OPC UA : Protocol for Industrial Communication (https://opcfoundation.org/about/opc-technologies/opc-ua/) Accessed 22.10.2020

process must not be affected as it might cause catastrophic events like rupturing of boiler tubes or over heated fluid etc. Updating the software for these hardware components is a complex and error-prone task and requires experienced field engineers who are experts in the industrial automation domain.

Cloud-native platforms provide support for automated disruption-free software updates and on-demand memory and computing resources to applications running in them. There has been a lot of work related to running industrial control applications in application containers and distributed computing systems. The results of these works show that it is a feasible solution, but there is no work in running these applications in a full fledge container orchestration system yet. Using cloud native platforms and container orchestration systems for running industrial automation systems like industrial control applications and PLC's could make the updating of control applications and will reduce the cost of the entire system as they can entirely replace the hardware controllers.

The thesis presents a study of existing update strategies used for updating applications in container orchestration systems and how these strategies can be used for updating stateful industrial control applications. During the update of an industrial control application state restoration or state transfer in which the existing state (internal variables like counter values which are supposed to be retained) in the currently running control application are transferred to the updated control application, is a critical event. If this is not done during an update, then the control process can get disrupted.

For example, if the running control application has an internal counter variable with value 10, then if the state transfer is not done, the internal counter variable of the updated control application is initialized at its default value which is generally 0, and this could cause the updated control application to send incorrect signals to the actuators which can result in failure of the entire system.

Automatic and manual update of an industrial control application in a simulated environment was implemented as a part of this thesis. Using the automated deployment strategy, the engineers can deploy security or OS-related updates to the automation system without the need for having expert knowledge in industrial automation. Extensive tests were performed to measure the state transfer time for different state sizes so that

the maximum state size that can be transferred in a single cycle can be identified. Typically control applications industrial processes have a cycle time ranging from 0.5 ms – 500 ms (Gangakhedkar et al., 2018) and use 10-650,000 (0.1kb – 5MB) variables of which on an average 30% of the variables can be assumed to be retained (Krause, 2007) (Muslija, 2017). These numbers vary depending upon the scale and the sector of the industry. The results show that the control applications with an estimated cycle times equal to or more than 200 ms and an internal state size of less than 36 kb (which is equivalent almost 15,000 different variables including integers, booleans, long and double integers) can be updated without affecting the execution of the control process. The results provide a basis for further studies and several new challenges like the fine-tuning of the software-based PLC runtimes and the state transfer application which were identified can be studied in future works.

1.2 Research Questions

As explained in the overview, the primary contribution of this thesis is studying different update strategies in cloud native systems and to use the existing features of such systems for updating industrial automation software. The main research questions that drove the research work in this thesis are:

Research Question 1 (RQ1)

What additional steps must be carried out to update an industrial control application which is running in a container orchestration system when compared to updating existing stateful and stateless web applications?

Research Question 2 (RQ2)

How can the internal state of a running control application be transferred to the updated control application in a cloud-native environment where the existing and the updated control applications might run on same or different physical nodes?

Research Question 3 (RQ3)

How can the existing features of container orchestration systems be used to support the updates of industrial automation software without any disruptions?

Research Question 4 (RQ4)

What are the different update strategies used for providing disruption-free updates to applications running in container orchestration systems like Kubernetes, and which of these strategies can be used for updating industrial control applications?

Research Question 5 (RQ5)

How are manual and automatic updates for industrial control applications implemented in a container orchestration environment?

1.3 Thesis Structure

The outline of this thesis is explained here. The next chapter discusses the background information about industrial automation systems and cloud native concepts. In Chapter 3, existing work on update strategies in cloud-native technologies and stateful applications in Kubernetes is discussed. This Section also discusses the related work on software containers for industrial control and updating industrial control applications with zero-downtime or without any disruption. The different aspects of updating different types of industrial automation software especially industrial control applications in a container orchestration system are explained in Chapter 4 in a generic way. Chapter 5 explains how some of the key use cases explained in the concept chapter are implemented in Kubernetes. The next chapter shows state transfer speeds of different state transfer mechanisms and evaluates their overhead. In the final chapter conclusions are derived based on the results and the future works that could be studied further are discussed.

2 Basics of Cloud-Native and Industrial Automation

This chapter gives background information on cloud-native environments and industrial automation software. Cloud-native environments are now widely used while deploying software applications. A computing cloud provides elasticity to software applications so that they can easily be scaled up to, e.g., serve more users. Industrial automation software, generally, runs on dedicated hardware within a production plant, which is not elastic, as the hardware is tightly coupled with the software. If these two worlds are combined, then it would result in elastic industrial control applications which can be up-scaled whenever required. Since industrial automation software applications have different requirements, some changes and modifications are required in the update strategies of such applications.

2.1 Cloud-Native Environments

Cloud-native environments allow development and delivery of software applications in a virtual scalable infrastructure rather than on bare-metal servers (Garrison and Nova, 2017). Cloud-native concepts enables companies to develop software in-house without worrying about issues like infrastructure and scalability.

The cloud-native computing foundation (CNFC) defines cloud-native as: "*Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach*". (Cloud Native Computing Foundation, 2018).

A cloud-native environment consists of three main components – containerization, container orchestration and a microservices architecture of applications. Containerization and container orchestration are a part of the infrastructure and the microservices architecture is a requirement of the applications that are deployed in cloud-native environments.

One of the key features of a cloud-native environment is that it is designed in such a way that it allows developers to develop, deploy and test applications without disrupting production systems(Gannon et al., 2017). This feature of cloud-native environments is

useful for updating industrial automation software applications without affecting the functioning of the physical process in the industry, e.g. a manufacturing plant.

2.1.1 Virtualization and Containers

Virtualization is an important concept used for deploying software applications. "*Virtualization is a method for running a virtual instance of a computer system on a layer abstracted from physical hardware*" (Opensource, 2020). Virtualization is often associated with using virtual machines (VM) which run on the top of a hypervisor (Kasireddy, 2016) that runs in a host operating system on physical hardware.

Virtualization can also be achieved using software containers. Containers provide virtualization at the level of operating system where as virtual machines provide virtualization at the level of hardware (Felter et al., 2015). Since containers provide OS-level virtualization, they share the kernel of the host system with other containers. A container does not include a complete OS, but uses kernel features to provide isolation between containers, for example, Docker uses kernel features like Process ID namespaces, Network namespaces etc. to provide a layer of isolation (Docker, 2020). On the other hand, virtual machines are heavy and run as separate applications in the host operating system so upscaling and downscaling virtual machines is a tedious task. Figure 2.1 shows the architecture of software containers and virtual machines. Since containers are more lightweight as compared to VMs they are generally used in cloud environments for running many software applications on single hosts with the overhead for many VMs. Linux Containers (LXC) and Docker container are two major ways to create containers in an OS. Docker is the most popularly used container engine as it provides features like

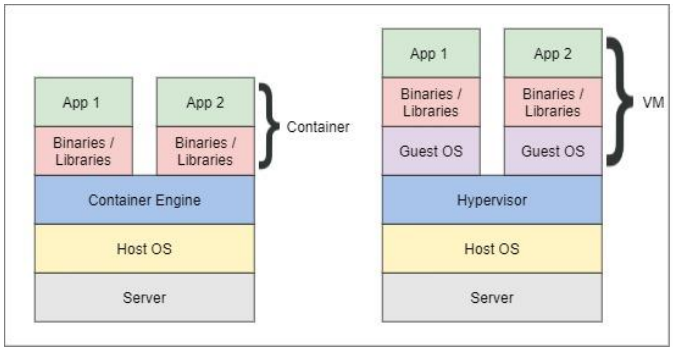

Figure 2.1 : Containers and Virtual Machine (VM)
Image based on : (Kasireddy, 2016)

portable deployments, versioning support, component reuse by using a base image etc. to manage the container lifecycle (Baukes, 2020).

2.1.1.1 Docker Containers

Docker is a container engine which provides a setup for running application containers on an operating system. Docker isolates the CPU, memory, input-output (i/o) and network using a kernel and an application level API (Bernstein, 2014). Docker containers are setup using a so called Docker file which includes a step-by-step instruction of which dependencies (Libraries, Applications, Service etc.) are required for an application. Large-scale web applications like Netflix or Amazon are built using a microservice architecture and, thus, need 100's or 1000's of containers running the different microservices on such Docker containers (Nair, 2017). Since there are so many Docker containers running in parallel, updating and deploying new applications is a complex task if all these containers are manually maintained. This is the primary reason for the need of a container orchestration system to simultaneously manage multiple containers and their lifecycles.

2.1.2 Container Orchestration Systems

A container orchestration system provides a framework for dynamically managing containers and applications at scale. (Redhat, 2020c). Using such a system, lifecycle event can be managed more easily and, e.g., updates can be delivered more frequently and with less manual efforts.

A container orchestration system typically provides the following features (Casalicchio, 2019) (Redhat, 2020c):

1. Resource Allocation – To limit the minimum and the maximum resources (CPU and Memory) allocated to a container and an application
2. Scheduling – Scheduling is done for starting the containers on certain nodes, based on factors like node affinity.
3. Load Balancing and Traffic Routing – The load balancer is required to evenly distribute the requests coming from the clients(internal and external to the cluster) among different container instances.
4. Health Checkup – Continuously monitor the health of the containers.

5. Fault Tolerance – Support node or application failure, and automatically switch-over of the traffic to the working nodes.
6. Auto-Scaling – Container orchestration system must support auto up-scaling and down-scaling features so that the resources can be optimally utilized in case of a higher or lower than average demand.
7. Zero-Downtime Updates – Container orchestration systems provide features to update the applications without any disruption, for example, rolling updates in Kubernetes. This can be achieved by having multiple containers of same application and updating each container one by one so there is no downtime to the application.

Figure 2.2 shows the architecture of a typical container orchestration system. On the left side the orchestration engine is depicted which contains the scheduler, controller, and the

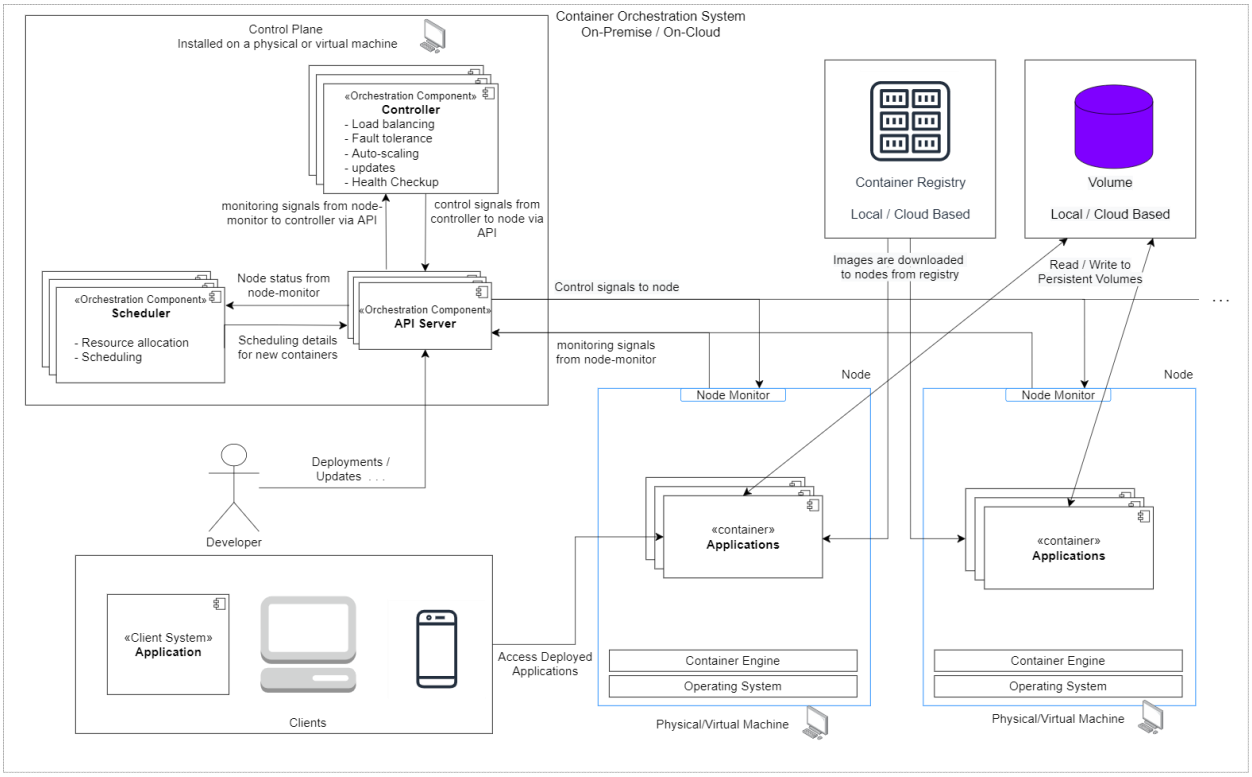

Figure 2.2 : Container Orchestration System
Image based on (Casalicchio 2019)

API Server. The scheduler is responsible for scheduling the containers on different hosts and the controller continuously monitors the cluster and is responsible for tasks like Load

Balancing, Health Checkup, Updates etc. The API server is responsible for all the communication between the orchestration engine and the nodes.

The Controller receives monitoring data from the node monitor via the API Server. This data is then processed by the controller and the controller sends the control actions which are to be performed to the nodes via the API Server. Similarly, the scheduler receives details about a resource which is to be created. The scheduler then sends the API the details about in which node the container must be scheduled.

The nodes are physical or virtual machines with an underlying operating system and a container engine like Docker. The node monitor is an application which runs on a node to monitor and control the containers in a specific node. The node monitor sends monitoring information like container failure events to the control plane via the API server. Based on the monitoring data, the control plane answers with the details about what actions are to be executed in such a case.

The applications run on containers in nodes and they are controlled and monitored by the orchestration engine. The container images used in the nodes are downloaded from either a public/private cloud-based container registry or a local registry. The images are uploaded by developers and the names and tags of these container images are provided during the deployment or update of an application. Volumes are used for storing stateful data in a cluster. The containers can be mounted to these volumes so that they can read/write data on the volumes.

There are multiple container orchestration systems that offer on-premise container orchestration: Docker Swarm is a container management and orchestration system provided by Docker, Kubernetes is another container orchestration system developed by Google and later released as an open-source project, Mesosphere Marathon, a container orchestration framework for Apache Mesos[1]. Kubernetes is the most popular container orchestration system. Container Orchestration is also offered as a cloud service by Amazon Web Services (AWS), Google Cloud Platform (GCP), Redhat OpenShift,

[1] Apache Mesos : https://mesos.apache.org/, Accessed 22.10.2020

Microsoft Azure Cloud, but most cloud service providers use Kubernetes as the container orchestration engine.

2.1.2.1 Docker Swarm

Docker swarm is an open-source container orchestration platform which creates collection of Docker containers to form a cluster. These Docker containers can be managed by using the usual Docker commands, but when running in swarm the commands apply to a collection in the cluster instead of just a single container (Sumo Logic, 2020). Docker swarm provides all the orchestration services like load balancing, fault tolerance, health checkup etc. Since Docker swarm provides the same API as Docker, all the features of Docker are supported by swarm. Docker swarm is currently supported by Docker, but the community support for Docker Swarm currently seems lower as compared to Kubernetes. Docker swarm also provides very little or almost no support for extending the core functionality with custom implementation, for example, the feature of automatic state transfer of a control application during an update. Docker swarm is ideal for users who have to quickly get started with a container orchestration system without taking much efforts and who don't need to customize or extend the existing functionality (Mangat, 2019).

2.1.2.2 Kubernetes

Kubernetes is the most popular open source container orchestration system which was initially developed by Google as a part of their Borg Project (Eldridge, 2018). Kubernetes is widely used by organizations to manage their services and applications in production. Kubernetes supports all the features of a container orchestration framework mentioned in Section 2.1.2 and along with that provides customization options to users by adding custom resource controllers for performing tasks which are not currently supported. For example – Adding an update controller to transfer state from existing version to an updated version during an update. Kubernetes is a well maintained open-source project with frequent releases, stable documentation and great community support was observed during the thesis.

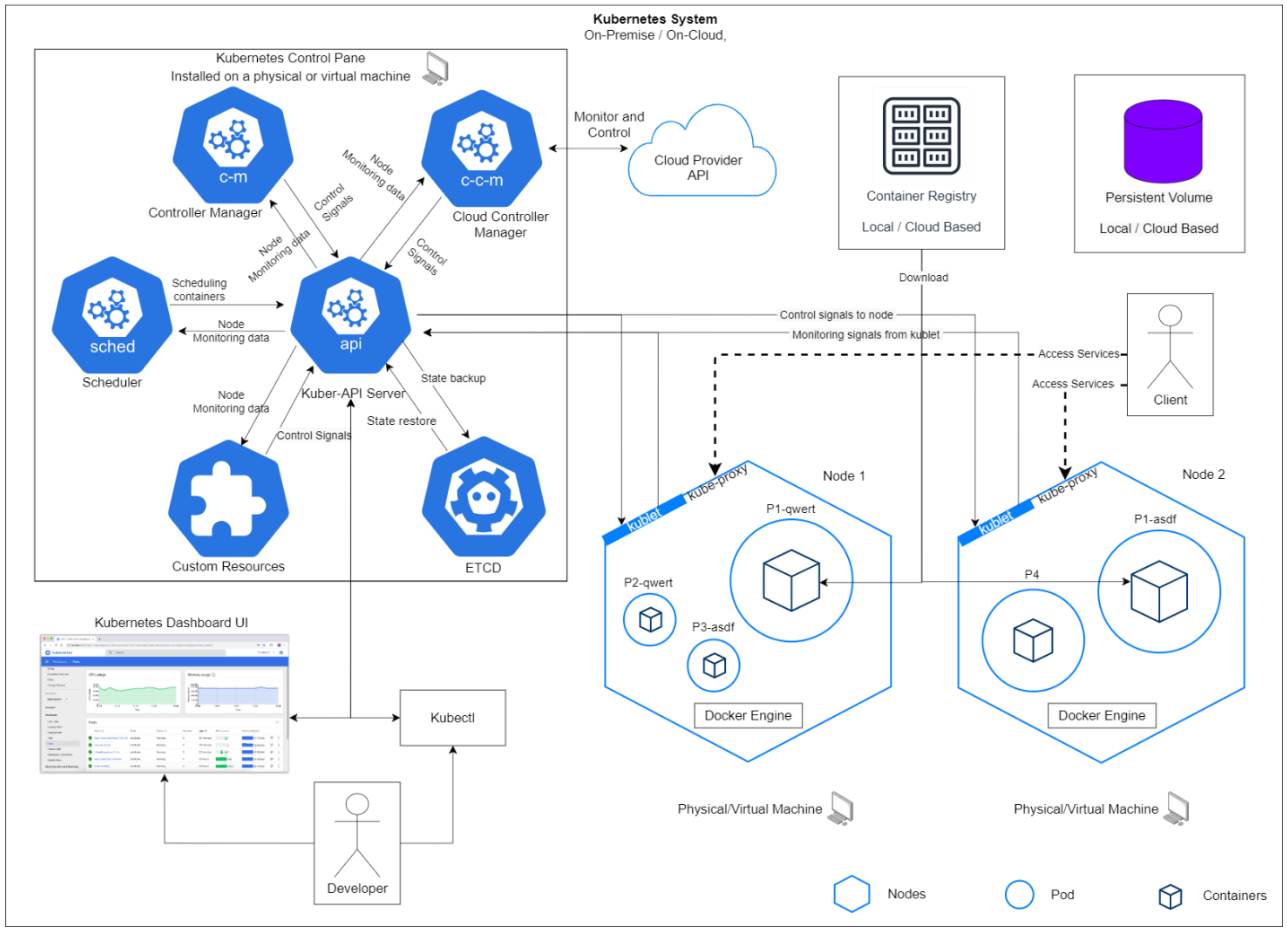

Figure 2.3 : Kubernetes Architecture
Image based on (Kubernetes 2020c)

Kubernetes Architecture

The Kubernetes architecture is based on a standard container orchestration architecture with a few additional features, which are not common in all container orchestration systems. This includes the support of custom resources, a Kubernetes dashboard, kubectl – Command line client to access kubernetes API, cloud controller manager and kubelet and kube proxy (Kubernetes, 2020c). Figure 2.3 shows the Kubernetes architecture. The Kubernetes control plane is the core orchestration framework of Kubernetes. The core components of Kubernetes communicate with the nodes through the API server. The Kubernetes API Server, Controller Manager, Scheduler, Custom Resources and ETCD are the primary components of the control plane.

The controller manager and custom resources component receive monitoring data from the Kubelets (Node Monitors) of the nodes. The kubelets are node monitors of a node in a Kubernetes cluster and send signals like pod failure or container health status to the respective controllers. The controller manager and the custom resource components send control signals based on the inputs to the kubelets through the API Server.

The Kubernetes dashboard (UI for interacting with Kubernetes) and the kubectl (command line tool to interact with the Kubernetes API) is connected to the kube-apiserver for interacting with the cluster using a user interface. These tools are used by the developer to deploy, update or delete a resource in Kubernetes. The container registry is storage for all the container images (for example, Docker images).

The developers upload the container image to the registry and the Kubernetes nodes pull the image from this registry. The registry can be local or cloud-based. To store stateful information, the cluster consists of a Persistent Volume (PV). PV's are the volume mounts for Kubernetes. A PV can also be local storage or an external hard disk or a cloud-based storage solution. Kubernetes nodes host the applications containers.

The control plane acts as a Kubernetes master which controls the cluster and the nodes are like workers which execute the container applications. The nodes and the control planes run in a physical or a virtual machine which is hosted on the cloud or on premise.

Kubernetes Control Plane Components

The control plane is responsible for making critical decisions about the cluster like scheduling of containers and for monitoring the different resources deployed on the cluster and responding to new cluster events, for example monitoring the application health and setting up new containers for applications when they are upscaled (Kubernetes, 2020c).

- Controller Manager: A controller in Kubernetes is responsible for maintaining the state of the cluster. A controller runs in a loop and monitors the parameters of a specific resource (like pods) and acts accordingly restore the desired state. The controller manager manages the different built-in controllers like node controllers, endpoints controller, service account and token controllers.

- Cloud Controller Manager controls and interacts with the API of the cloud provider. This is only applicable in the case of using a cloud-based provider like Amazon Web Services or Google Cloud Platform and is not required when using an on-premise Kubernetes cluster.
- Scheduler is responsible for scheduling the application pods on the nodes, based on factors like node affinity, resource availability etc.
- etcd is a key value data store which backs up all the cluster configuration data
- API server is a REST API which is exposed across the cluster and it interacts with the other internal components. All the actions like create, read, update, delete are done via the Kubernetes API server.
- Custom resources – Kubernetes offers extension and customization features to implement custom functionality which is not supported by the default Kubernetes resources. These custom resources run in the Kubernetes control pane. Kubernetes cannot provide complex implementations for stateful applications and remain generic and adaptable at the same time. For example, the replication controller is a generic controller for all applications which have multiple pods running, so this functionality is provided by Kubernetes. But transferring the state of a control application during an update is a very specific requirement for control applications and not a generic solution. For such implementations a custom resource is required.

Kubernetes Node Components

Kubernetes node are physical or virtual machines which run the actual applications (pods) which will be running in Kubernetes. The node components are like worker nodes in Kubernetes (Redhat, 2020b).

- Kubelet monitors and controls the containers running in a pod. The kublet is responsible for staring up new containers, killing them, notifying about container health and failures to the controller etc.
- kube-proxy also runs on every node and is a network proxy which allows communication to other nodes and the external network.
- container engine is responsible for running containers on the nodes. The most popular approach is using Docker containers and the Docker engine with Kubernetes. The

container engine downloads the Docker images from a Docker registry which can be within the network (local) or public Docker registry.

- A pod is the smallest execution unit in kubernetes. Each pod can run single or tightly coupled multiple containers which collectively run as single unit of execution of a specific application which is deployed on the Kubernetes cluster. All the containers in a pod are always scheduled in the same localhost network and share the same memory. The containers within a can communicate with each other via network using localhost or can use inter process communication in case a faster communication is required. Pods have their own IP address, name and a port range, so the containers running within a pod must be carefully configured to avoid port clashes (Ibryam and Huß, 2019).

Other Resources in Kubernetes

Apart from the core components in the Kubernetes architecture, developers use different resources to deploy the pods and access them via the network. The Kubernetes control plane controls these resources. Ibryam and Huß have explained in their book about the different use cases (patterns) in Kubernetes and which resource to use for which pattern (Ibryam and Huß, 2019). A few of the relevant resources are explained below. Pods are the basic execution workloads, which run in a node. Pods are one of the primitive components and other components are services, deployments, stateful sets, replication controllers, daemon-sets, jobs, volumes etc. Only the relevant workload components which are related to the thesis are discussed in the following section.

- Deployments: The deployment resource is used for deploying pods, and, during an update, taking care of the automatic scheduling of new version pods and then complete the update process with without any manual effort and no downtime. The pods which belong to a specific deployment are by default named as the <deployment name>-<random id>. When a deployment is started in Kubernetes, it starts a Replica Set which starts the scheduling of pods. A Replica Set is responsible for the scaling of pods and ensuring that the specified number of pods are running in the cluster.
- Stateful Sets: The Stateful Set resource is used for managing stateful applications. Stateful Sets are also responsible for managing the pods, but in stateful sets the pods

maintain a sticky identity which is not in the case of deployments. Stateful sets also offer features like ordered deployment and scaling, ordered automated rolling updates, stable persistent storage and stable IP address for pods for running stateful applications.

- Services: Services are used to expose pods to the internal Kubernetes network as well as the external network. Services represents a named entry for accessing the pods of a workload. For example, if a web application deployed on the cluster has multiple pods, and if a web request is initiated from a client, the request goes via the service to either of the pods. In this case the service acts as an internal load balancer for the pods and makes the decision about to which pod the request must go. The endpoint controller is responsible for configuring and maintaining services in Kubernetes.
- Volumes: Volumes are the storage provisions provided by the Kubernetes cluster. Containers are stateless in nature, and when the container or pod crashes, the data within the container is also deleted. In Docker the concept of volume is used to store critical data on a disk which is outside the container. In Kubernetes a persistent volume (PV) is used to store data which can then be used and accessed by other pods in the cluster. To connect a persistent volume and a pod, a persistent volume claim (PVC) is required in Kubernetes. This PVC can either be created earlier for Replica Sets or can be dynamically created for Stateful Sets.

2.2 Industrial Automation Components

Industrial automation is the process of controlling industrial equipment and industrial processes by using different programmable logic controllers and supervisory control systems. The automation components can be classified into four categories 1. Operations Management Level, 2. Control Level, 3. Field Level, 4. Communication Interface (Vyatkin, 2013).

The operations management consists of the engineering station which consists of the simulation environment and the integrated development environment (IDE) for building control applications. The control level consists of the programmable logic controller and the human machine interface (HMI) for monitoring the processes and making any

changes to industrial parameters if required. The field level consists of the field devices like sensors, which collect data from different sections of the industry (for example, a temperature sensor for monitoring temperature of a boiler of a manufacturing plant) or actuators which control the equipment. The communication interface is required for the communication between the field devices and the control applications. In the next sections, the basic concepts for industrial automation software, field devices and the communication between them is explained.

2.2.1 Industrial Automation Software

Industrial automation software consists of all the different types of software required to control the hardware equipment in an industry, analyze the data and monitor the entire process. These are 1. Supervisory Control and Data Acquisition Systems (SCADA), 2. Programmable Logic Controllers (PLC's), 3. Human Machine Interface (HMI) and 4. Distributed Control Systems (DCS) (Plant Automation Technology, 2020).

1) A SCADA system collects the system operation data, then makes decisions about the system processes based on the data, and controls the entire system (for example, a manufacturing plant).
2) PLC's are programmable controllers which run a control application in loop to monitor and control an industrial process so that it remains stable. Control applications are one of the critical sections of the industrial automation as they control critical processes of an industry. For example, in a manufacturing plant, a PLC is responsible for controlling the level of a fluid in a boiler at a certain level.
3) The HMI components display the information of different processes in the industry. Modern day HMI components are displays with touch screen interfaces or have a mouse and keyboard to change and monitor different parameters for the industry. For example, if the level of fluid to be maintained by the PLC in the previous example can provided by a field engineer via the HMI component.
4) A DCS is an alternative to SCADA system and the primary functions of both the systems are almost similar.

In this section, the basics of different types of industrial automation software are explained. Since the industrial control applications are critical components, they are

discussed in detail. Some details about other type of automation software like industrial analytics and HMI components are explained.

2.2.1.1 Control applications and the Programmable Logic Controller (PLC)

A Programmable logic controller is a device which runs a control application in a cyclic execution to control the electrical and mechanical components in a production plant. For example, in a manufacturing plant, a PLC is used to monitor and control the critical processes for example – maintaining the temperature of a furnace to a specific level by controlling the amount of gas that flows into the furnace. In this example the control application for controlling the gas flow is running in a loop.

Architecture of a Programmable Logic Controller

The architecture of a PLC is based on a standard computer architecture, but is designed to sustain harsh industrial environments and has a large number of input output ports (i/o) to connect to different peripheral devices. Figure 2.4 shows the basic PLC architecture with all its components: 1. Central Processing Unit (CPU), 2. Memory, 3. Input-Output Modules, 4. Communication Interface and 5. Control Application (Petruzella, 2005). The

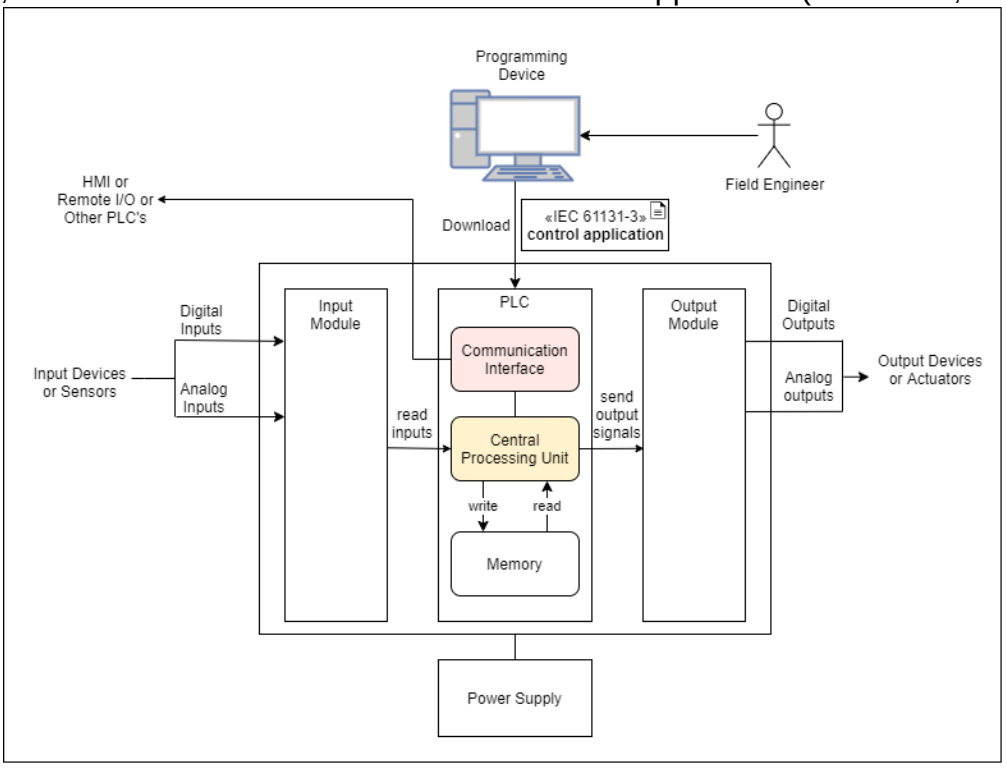

Figure 2.4 : PLC Architecture
Image based on (Petruzella 2005)

CPU collects the signals from the input module, runs the logic in the control application and then sends the output signals to the output modules.

The CPU also stores important state information in the memory of the PLC. The internal state information of the control application and the PLC is stored in the memory (for example, if a counter is used in the control logic, then the counter value is stored in the memory).

The input and output modules are required for connecting peripheral devices like sensors and actuators to the PLC. Depending on the I/O ports, a PLC can be of type Fixed I/O or Modular I/O. Fixed I/O PLC's have a fixed set of input and output ports and it cannot be changed whereas in a Modular I/O PLC the I/O ports can be externally attached to the PLC (Petruzella, 2005).

In addition to the I/O devices, a PLC needs to communicate with other applications as well and communication interface is responsible for such connections. These applications can be remote I/O devices [1]or a human machine interface (HMI) or other PLC's or even to send data to a SCADA system. A PLC system is user programmable and the control application is the PLC program which contains a set of instructions for the PLC. The IEC 61131-3 standard defines the standard for the control applications (PLCopen, 2013).

As per the standard, there are five different programming languages in which a PLC program can be written: 1. Ladder Logic – Based on circuit diagrams used to run relay logic hardware, 2. Functional Block Diagram (FBD) – FBD's visualize the signal and data flows between the components and show the logic between the input and output as functional blocks, 3. Structured Text – Is a high level language with its roots in Pascal, 'C' and Ada, 4. Instruction List – Low level programming language similar to assembly code and 5. Sequential Functional Charts – Graphical oriented language which describes the chronological order of different actions in a program.

The architecture shown in Figure 2.4 is a general architecture of an embedded PLC. A PLC can also run on general purpose hardware a software environment like a Windows PC or a Linux based PC or Raspberry Pi. To run a control application on general purpose

[1] Remote I/O: Remotely located IO devices (https://realpars.com/remote-io/), Accessed 22.10.2020

hardware, a PLC runtime is required. A PLC runtime can interpret an IEC 61131-3 (PLCopen, 2013) control application and execute it on the host system's processor. The interface between the software-based PLC's and the field devices is an important part of such systems. This connection is established using an interface card that communicates between the fieldbus[1] devices and the host system (Windows/Linux PC or a Raspberry Pi) using a software interface (Magro and Pinceti, 2007). OpenPLC is an example of an IEC 61131-3 compliant software-based PLC (Alves et al., 2014). is a general architecture of an embedded PLC. A PLC can also run on general purpose hardware a software environment like a Windows PC or a Linux based PC or Raspberry Pi. To run a control application on general purpose hardware, a PLC runtime is required. A PLC runtime can interpret an IEC 61131-3 (PLCopen, 2013) control application and execute it on the host system's processor. The interface between the software-based PLC's and the field devices is an important part of such systems. This connection is established using an interface card that communicates between the fieldbus devices and the host system (Windows/Linux PC or a raspberry pie) using a software interface (Magro and Pinceti, 2007). OpenPLC is an example of an IEC 61131-3 compliant software-based PLC (Alves et al., 2014).

Control applications and Typical cycle times

Control applications contain a set of instructions for the PLC and these instructions are executed in a closed loop. At the beginning of the loop, the control application collects all the input signals from the sensors and based on these inputs, the control application generates output signals for the actuators in an industrial process.

The control application is executed at fixed cycle time (also called as deadline). The cycle time is different than the time required for execution of the control application. The time required for execution of the control applications should be less than the cycle time, and if the execution takes longer time, then there is a delay in starting the control application in the next cycle.

[1] Fieldbus : A set of connection protocols used for communication between the field devices and PLC (https://realpars.com/fieldbus/) . Accessed 17.09.20

Gangakhedkar et al. have presented a list of different categories of control applications and their average cycle times (Gangakhedkar et al., 2018). Motion control applications for machine tools have cycle times of less than 0.5 ms which is the minimum cycle time in the list of control applications presented by the authors. On the other hand, closed control loop applications in process monitoring have a cycle time in the range of 10-100 ms.

Standard robot operations and traffic management applications have a cycle time range of 40-500 ms which is the maximum cycle time in the list of control applications presented by the authors. It can be observed that the average cycle times are around 10-100 ms for process control applications like industrial boiler control in a manufacturing plant and precise robotic machine control and motion control applications have very low cycle times which are less than 1 or 2 ms.

Hard and Relaxed Real-Time Control Applications

Programmable logic controllers are critical components in an industrial automation system. For some use cases even missing a single cycle can cause catastrophic effects. For example, nuclear reactor safety systems are designed to do complete shutdown the nuclear reactor when there is an emergency. When a nuclear reactor is turned off, the reactor continues to generate radiated heat. To do a graceful shutdown of a reactor, such safety systems are required. If the PLC responsible for performing such a safety shutdown breaks down or misses even single cycle of execution of the control application, there is a possibility of a nuclear meltdown (Thomson, 2015).

There are also systems which are not affected if a few execution cycles of the control application are missed by the programmable logic controller. Based on the cycle time requirements of a Real-Time application, Goldschmidt et al. have classified the Real-Time applications in three categories:

1) Hard Real-Time applications – These are critical applications and missing any deadline in execution can cause a system failure. For example, a boiler control application or a nuclear reactor safety system,
2) Firm Real-Time applications – Missing a deadline will not cause a system failure but will degrade the quality of the final product. The system can tolerate minor

delays in the execution cycle times. For example, a robotic assembly line, or an electric vehicle charging station

3) Soft Real-Time control applications – These also have relaxed requirements of the execution cycle times for the control applications. Missing a few deadlines does not cause a system failure, but greatly affects the quality of service of the product. A video streaming system is an example of soft Real-Time system where if a few packets are delayed but the average speed is good, the system still runs.

For Real-Time industrial control applications, there are two main categories – Hard Real-Time industrial control applications and Relaxed Real-Time industrial control applications (Goldschmidt et al., 2015).

Real-Time Operating System Requirements

It is explained in the earlier section that the performance and quality of service is adversely affected when there is a delay in the execution and cycle times of the control applications. To ensure that the cycle times are constant and there is no jitter and delay in the execution of control applications, there should be a mechanism in which a high priority can be assigned to the control applications so that if any other task is being executed by the CPU, it will be paused and the control application will be executed at its cycle time. This concept is called as Pre-emption and it is one of the key features of a Real-Time Operating system. Magro and Pinceti have tested the performance of PC-based industrial control systems in real-time systems (Magro and Pinceti, 2007). The results show that there is a maximum variance 12.39 ms when using no priorities for a control application of cycle time of 10ms. When the same control application was assigned the highest priority, the variance dropped to 0.01 ms. Based on these results it can be observed that using real-time operating systems and assigning priorities to control applications reduces the variance in their cycle times.

2.2.1.2 Stateful Components other than industrial control applications

Along with industrial control applications, industrial automation software also consists of other stateful components like industrial analytics tools and software for analyzing the data collected from different systems. Such analytics systems have different implementations for different analytics use cases. Harper et al. have discussed about

different types industrial analytics pipelines in their article (Harper et al., 2015). Most of the pipeline types discussed by the authors are stateful applications which need to store data in a permanent volume. Most of these pipelines can be executed in a container orchestration system like Kubernetes – For example, Hadoop[1] applications can run within a Kubernetes environment. (Kim, 2020). Using container orchestration systems can be very useful for running industrial analytics software because of the distributed and highly scalable capabilities of such systems. The features like stateful sets and persistent volumes allow users to easily deploy stateful applications on Kubernetes a cluster.

2.2.1.3 Stateless Components in Industrial Automation

Stateless applications are applications which don't need the previous state of the application for processing current data. SCADA systems are used in industrial automation systems consist of a Human-Machine-Interaction (HMI) component. These HMI components display an overview of the entire automation system. The key functions of an industry standard HMI system are – 1. Displaying the process sequences and statuses in an automation system, and 2. Providing a visual interface for the field engineers to operate any field device in the entire automation system. These are stateless functions in an HMI component as these functions don't need any historic data of the system. Apart from that logging and displaying archived data, reporting of events, archiving data and alarms to a database, user administration and access rights etc. are also the functions of an HMI component (Siemens AG, 2009). Lathi et al. have presented a review and evaluation of the use of a web-based HMI system in a SCADA system which can be accessed by the clients web browser (Lahti et al., 2011). Such a system can be deployed on a container orchestration system.

2.2.2 Field Devices (Sensors and Actuators)

Field devices are sensors and actuators attached to industrial components like pipes, motors, tanks etc. These field devices are designed to withstand harsh industrial environments like excessive heat or pressure. The sensors are used to monitor the parameters of an industrial component. For example, a temperature sensor is used to

[1] Hadoop: Framework for distributed processing of large data sets. https://hadoop.apache.org/, Accessed 22.10.2020

measure the temperature of a furnace or a fluid inside a boiler. Actuators are mechanical or mechatronic or electrical devices which are responsible for controlling an industrial process. For example, a valve which controls the flow of a fluid in an industrial boiler in a manufacturing plant is an actuator. These actuators can be controlled manually or electrically by sending a signal. During every cycle PLC's collect data from the sensors, process the data and then generate results which are sent to the actuators. A fieldbus network or an industrial ethernet network is used to can be used to connect the sensors and actuators with the PLC's. These networks are required to have real-time capabilities since they are a part of the control loops and any delays in the communications can be catastrophic.

2.2.3 Communication in Industrial Automation

Industrial automation is based on hierarchical structure shown in Figure 2.5 which is pyramid structure with the field devices and the factory machinery at the bottom and the sophisticated ERP system at the top. A general architecture consists of the 1. Field Devices – Sensors and Actuators connected to industrial equipment, 2. The PLC Systems

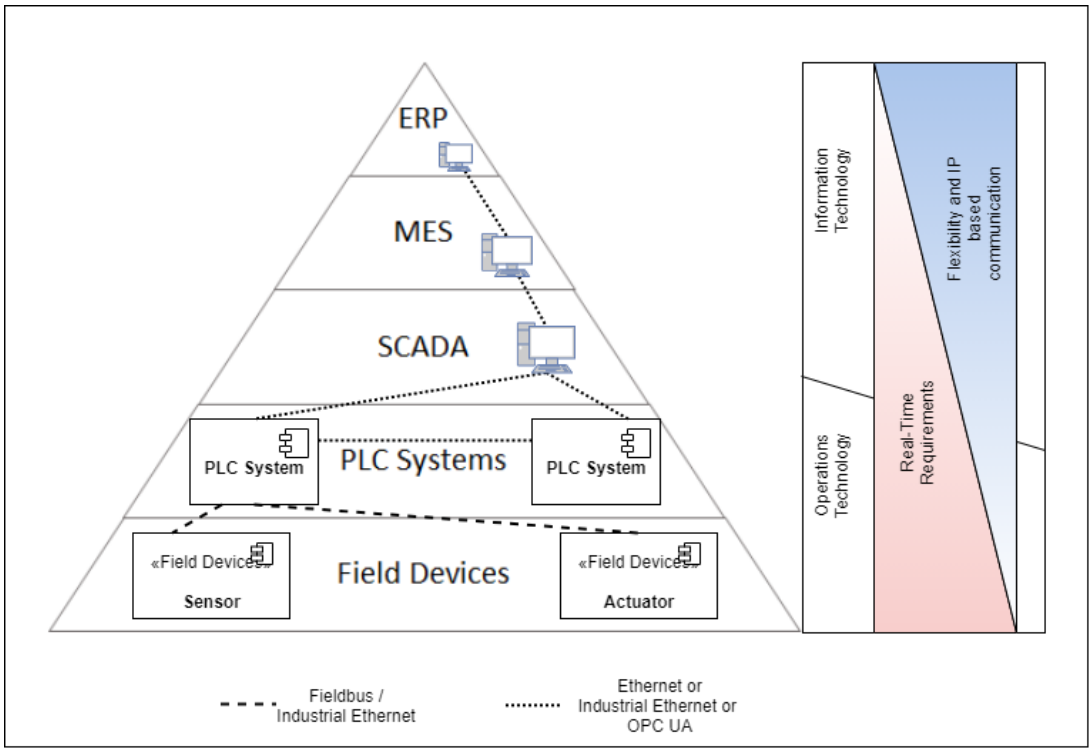

Figure 2.5 : Industrial Automation Hierarchy.
Image based on: (D. Bruckner et al. 2019) and (P. Drahoš et al. 2018)

– Control systems for controlling the industrial equipment, 3. SCADA System – A single system which can be used for monitoring and control of the entire factory or plant 4. Manufacturing Execution System (MES) – A system used in manufacturing to track the process of converting the raw materials in finished goods and 5. Enterprise Resource Planning (ERP) system – A software to monitor business processes (Bruckner et al., 2019). This is a general architecture of an automation system, but there are different variations to this approach especially in recent time with the introduction of Internet of Things. As seen in the figure when we move towards the top of the pyramid, the real-time processing requirements reduce and the flexibility increases.

For the bottom layers speed is a high priority as they are used to perform control operations with strict cycle time requirements. Fieldbus or Industrial Ethernet are suitable options in such requirements. Fieldbus is a technology which consists of multiple communication systems like Modbus[1] or Profibus[2]. The main advantage of the fieldbus technology is speed and reliability.

Some of the disadvantages include 1. Fieldbuses are complex networks, so users are needed to have a detailed knowledge of the network, 2. Fieldbus components are costly and 3. There are multiple implementations available for fieldbuses which are not fully compatible with each other, and this is a problem when components from different vendors are used (KUNBUS GmbH).

Industrial ethernet was introduced as a generic solution for industrial communication, but currently, industrial ethernet also has the same problem of having multiple incompatible protocols which use industrial ethernet for example PowerLink[3], Profinet[4], EtherCat[5] and Sercos[6] (Drahoš et al., 2018). This led to the need for a unified specification which can be used for industrial communication.

- OPC UA

[1] Modbus: https://www.modbus.org/, Accessed 16.08.2020
[2] Profibus: https://www.profibus.com/, Accessed 16.08.2020
[3] PowerLink: https://www.ethernet-powerlink.org/, Accessed 16.08.2020
[4] Profinet: https://www.profibus.com/technology/profinet/, Accessed 16.08.2020
[5] EtherCat: https://www.ethercat.org/default.htm, Accessed 16.08.2020
[6] Sercos: https://www.sercos.org/, Accessed 16.08.2020

OPC UA is a platform independent service-oriented architecture designed for communication between industrial automation software (Leitner and Mahnke, 2006). OPC UA is only focused on industrial automation and is independent of different vendors. This means that the field devices and PLC's from different vendors can be easily connected without any issues. OPC UA specification provides features like built-in security, monitoring and alarms, historical values, server discovery etc.

OPC UA offers two types of communication models 1. Client-Server and 2. Publish-Subscribe. In the client server communication, the client accesses the data from the server using the OPC UA features and in the Publish-Subscribe communication (PubSub) the published broadcasts the data in the network and the subscriber can connect to the desired published to get information (Drahoš et al., 2018).

In the client server communication, the server needs to maintain a session of the clients connected, and in some cases where multiple clients are connected to a single low powered server, then the CPU utilization is increased. For PubSub the CPU Utilization does not increase with the increase in the number of subscribers as there is no client session maintained at publisher's end (Burger et al., 2019).

Time-Sensitive Networking is the part of IEEE 802.1 standard which states that a TSN should guarantee data transport with a maximum bounded latency, low variations in the delay and extremely low loss (Bruckner et al., 2019). Brukner et al. have proposed the use of OPC UA combined with TSN for connecting field devices and PLC Systems (Bruckner et al., 2019). Alexander Gogolev has presented the results of adding basic TSN support to OPC UA. The results show that the OPC UA data exchange latency is significantly reduced when TSN is enabled (Gogolev, 2020). OPC UA aims to provide a unified information model for communication between all the layers of industrial automation.

3 Related Work and Literature

This chapter contains existing work on software-container-based industrial control applications and the available functionalities supported by Kubernetes which can be reused for running industrial control applications in Kubernetes. To run industrial automation software in cloud-native environment, the primary requirement is to run industrial automation software in application containers. Existing work for the same is summarized in Section 3.1. Industrial control software, nowadays, consists of stateful applications in which the state must be maintained during software updates at runtime. Section 3.2 explains the existing functionalities provided by Kubernetes for running stateful applications and the different approaches involved in the update of stateful applications. Sections 3.3 and 3.4 discuss the different update techniques and strategies used for updating industrial control applications running in a PLC and updating all applications in Kubernetes environment respectively.

3.1 Using Software Containers for Industrial Control

Running industrial control software in virtual environments like virtual machines and containers is a challenging task as industrial control software has time-critical requirements of processing. An interesting approach for industrial control using containers was proposed by Goldschmidt et. Al. in their paper (Goldschmidt et al., 2015). The authors provided an approach for running multi-tenancy based horizontally scalable software-based programmable logic controllers (soft plc's) as a stateless application. The architecture explained in the paper aligns with the motivation of this thesis and the state-less architecture used for soft-plc's is also the preferred approach in cloud-native applications. Figure 3.1 displays the proposed architecture of Soft PLC instances in a cloud environment as Docker containers.

As shown in the architecture, the control application runs in the cloud and the field devices (Sensor and Actuators) are connected to the control application via different connectors and data collectors. The connection of the field devices to the cloud connectors is achieved using field devices. Each instance of the Soft-PLC runs a different control application which are stored in a NoSQL database. After every cycle the input and output variables to the control application are stored in externally in the cache component. To

make the control application state less these input / output variables are sent to the respective control application, and it might be possible that the next cycle is executed in another instance of the Soft-PLC.

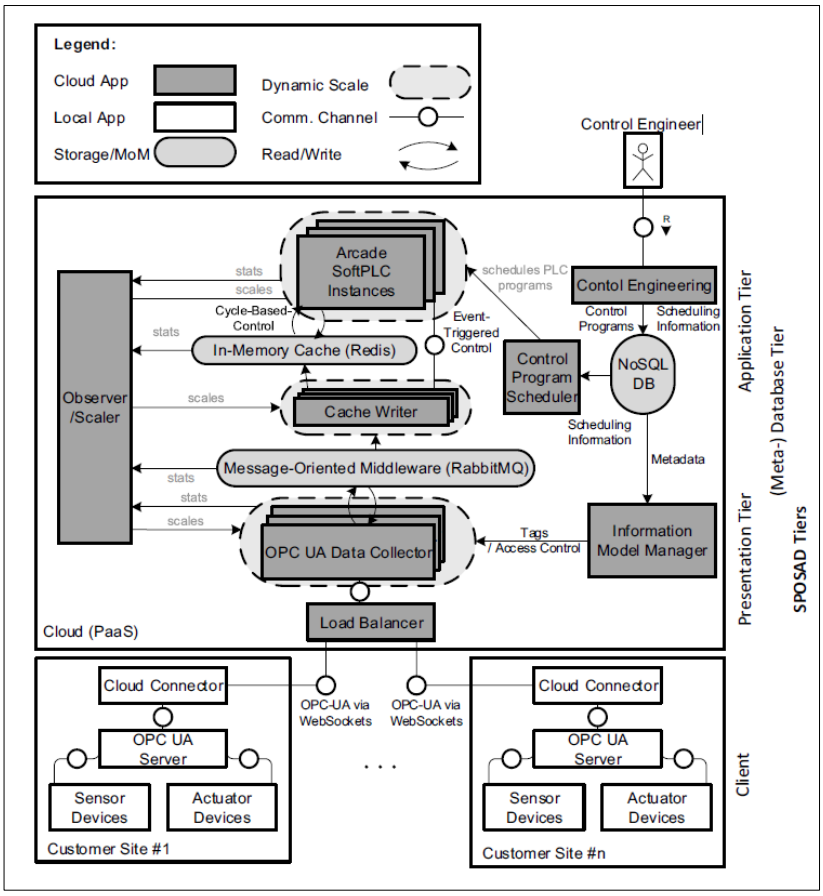

Figure 3.1 : Architecture overview of cloud-based Soft PLC
Image Downloaded from : (T. Goldschmidt et al. 2015)

The control program scheduler is responsible for scheduling the control programs in the Soft-PLC instances. Although this application is stateless and could efficiently run in a cloud-native environment, the caching of the i/o variables adds latency to the control application and so the approach only covers Soft Real-Time and Firm Real-Time control applications. With this approach, running control applications which require Hard Real-Time capabilities (a single cycle miss might cause a system failure) is not possible as there are chances of missing cycles during the execution. The results show a round trip time of below 1000 ms in 99.72% of the times for a system of 30 tenants.

In their later work, Goldschmidt and Hauck-Stattelmann have presented an architecture for running an industrial controller in a virtual container and then analyzed the impact of using containers on real-time constraints (Goldschmidt and Hauck-Stattelmann, 2016). The architecture presented by the authors is for a multi-purpose controller is based on a Real Time Container OS (OS capable of running a wide range of applications from low power field devices to high performance PLC's) and the application containers are a part of the controller itself. This means that the control applications run within a container which is a part of the multi-purpose controller, but other components like the i/o interface, container deployer etc. are also a part of the multi-purpose controller.

This concept of multi-purpose controller can also be replaced by a container orchestration system which manages the container scheduling of control applications. The authors conducted a cyclic test[1] from within and outside a Docker container, with and without additional load on the system. The cyclic test was equivalent to the running of a control application which runs repeatedly after a fixed cycle time. Based in the preliminary results the authors have suggested that overhead of using containerization in industrial control systems is low and the results are promising, but the solution needs to be tested and evaluated for real world control applications instead of a cyclic test.

Hofer et al. have tested the task latencies and the starting delays of real-time tasks on real-time patched operating systems on a containerization engine (Hofer et al., 2019). The tests were conducted on two different operating system setups – Xenomai[2] which is a co-kernel extension for Linux-based operating systems and PREEMPT-RT (The Linux Foundation, 2016) patched Linux and compared them with the results in a standard Ubuntu Server LTS.

These tests considered all the combinations for isolation, load balancing IRQ affinity and system stress etc. The results also show that the startup times of the PREEMPT-RT setup is better than the Xenomai setup. The results also show that the startup latencies for the PREEMPT-RT setup are better on the AWS cloud setup (AWS HVM Type 1 hypervisor-

[1] Cyclic Test: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start, Accessed 28.09.2020
[2] Xenomai: https://gitlab.denx.de/Xenomai/xenomai/-/wikis/home, Accessed 28.09.2020

based T3.xlarge generic and a C5.xlarge) when compared with the latencies measure on bare metal.

It can be concluded that by using the proper configuration and a Real Time operating system or an RT Patched kernel and using the appropriate cloud machine, the latencies and the startup times can be improved, but this needs to be tested extensively with real world industrial control applications. The results also confirm that migration to an IaaS (Infrastructure as a service) based virtualized cloud machine is possible from bare metal servers. In future the authors also plan to test the setup on a container orchestration platform.

There are different architectures proposed for running industrial control applications in containers and the concepts of these architectures can be reused for running industrial control applications on container orchestration platform. Based on the latency results calculated from the cyclic tests, it can be concluded that the control applications show comparable results when compared to using bare metal servers or physical hardware for running industrial control applications. When configured appropriately, using real-time operating system or real-time patches on operating systems can also improve the performance of the containerized control applications. Referring to Research Question 1 (RQ1), using real-time OS or a patched real-time Linux system could be an additional requirement during the execution and updates of if industrial control applications when compared to updating a pure software-based web applications.

3.2 Stateful Applications in Kubernetes

Stateful applications are applications which require the previous state of the application for processing current data. The term state refers to any element which changes like the internal variable values, interactions with other services, environment variables, threshold values etc. (Wootton, 2018). For industrial control applications this state refers to the values of the inter states of the control loops like counters. In this section the different use cases and approaches for deploying different types of stateful applications on Kubernetes are discussed.

The stateful applications like NoSQL databases have their own load balancing and replication algorithm which conflicts with the services and replication concepts of the

container orchestration systems. For example, Kubernetes provides a service workload for multiple pod replicas and by default the round-robin algorithm is used by the service to redirect the requests to the individual pods. This approach does not work for stateful applications like distributed NoSQL databases as the load balancer in the database cluster (for example, Mongo DB) have a different load balancing technique which is based on how the data is distributed across the cluster (Truyen et al., 2018). To solve such problems Stateful Sets have features like

1. Stable and unique network identifiers – Separate service objects for each application container running a stateful application. One IP address corresponds to only one container instead of acting as a load balanced service for the entire set of containers.
2. Stable persistent storage – In Stateful Sets the persistent volume can be accessed by dynamically creating a new persistent volume claim (PVC) during the creation of a pod. This is not supported by the Replica Set controller in Kubernetes and for the Replica Set controller the PVC must be created before starting up a pod. This is an issue during the upscaling of the application as creating a new PVC for every new pod is a time-consuming task.
3. Ordered graceful deployment and scaling – Stateless application instances running in a Stateful Set are created in an ordered manner, as in general, distributed stateful applications like MongoDB have a master slave structure and the master nodes are deployed before the slave nodes.
4. The stateful applications are hosted on nodes which are physically close to the location of the persistent volume to reduce the latency of data transfer between the container and the volume.

E. Truyen et al. have evaluated and extensively tested the execution of a NoSQL database – MongoDB on container orchestration systems. They have presented the overhead of using the different features of a container orchestration system as compared to using a VM Based or Docker only setup. They calculated the latencies of MongoDB database cluster in 4 different environments 1. Non containerised VM based deployment exposed via cloud provisioned (stable VM's IP address) IP address, 2. Docker based

containerised deployment exposed by a cloud provisioned end point, 3. Container Orchesteration based deployment exposed by a cloud provisioned endpoint and 4. Container Orchesteration based deployment exposed by a cluster provisioned endpoint (using a service in the container orchestration system). Based on the results it can be observed that the performance overhead of Kubernetes and Docker Swarm both is higher by 22% and 28% for simple read and update operations when compared with the VM-based deployment of MongoDB. Despite of the performance overhead, container orchestration systems provide better scalibility and high availability as compared to the VM-based setup. In the experiments the local volume was used by the container orchestration systems – Docker Swarm and Kubernetes (Truyen et al., 2018). The use of persistent volume for data storage also adds up a performance overhead to the execution of the basic operations of the Mongo DB cluster in Kubernetes.

Thomas Phelan at BlueData has presented a product called KubeDirector which simplifies the peocess of deploying complex stateful applications on Kubernetes (Phelan, 2018). KubeDirectory makes the process of deploying applications like cassandra[1], apache kafka[2], hadoop[3], spark[4] and other distributed stateful applications generally used for data science and machine learning use cases easier. KubeDirectory is a custom resource which is deployed on the Kubernetes control plane and provides a generic implementation for deploying stateful application by using a custom resource (CR) specification. Joel Baxter has explained the architecture of KubeDirector which consists of a 1. Application Catalog – internal cache of application metadata, 2. Validator – Validates the properties of the stateful application provided for the deployment of the application to the KubeDirector, 3. Informer – Watches the Kubernetess cluster for creation, modification or deletion of the custom resources types and informs about the changes to the reconciler, 4. Reconciler, Observer, and Executor. – The reconciler uses the observer to build a picture of the entire Kubernetes Cluster and determines the desired state of the cluster after the deployment of the stateful application. The exectuor is used for then creating the necessary components for deploying the stateful application on the

[1] https://cassandra.apache.org/, Accessed 15.09.2020
[2] https://kafka.apache.org/, Accessed 15.09.2020
[3] https://hadoop.apache.org/, Accessed 15.09.2020
[4] https://spark.apache.org/docs/latest/, Accessed 15.09.2020

cluster. This includes creation of stateful sets with the correct number of configured replicas, setting up of the persistent volumes and creation of the services for each pod of the stateful sets (Baxter, 2020). So at the end internally the KubeDirectory uses stateful sets and persistent volumes for managing the stateful applications and the developer doesnot need to worry about the management and configuration of these stateful sets and persistent volumes during deployment, updates or scaling. The core problem of adding overhead latency due to such a setup which was explained in the previous section would still persist.

Distributed databases like MongoDB and applications like Apache Spark and Hadoop have their implementation to handle data concurrency between replicas. For the stateful applications which do not have a specific mechanism for managing the concurrent concurrency across replicas to use the shared data efficiently, a specific logic must be implemented in the application to support that. When such type of stateful applications are deployed on a Kubernetes cluster, they access shared data in a persistent volume.

Netto et al. have presented a generic container replica coordination approach which manages the access concurrency between replicas accessing the backend data store using a lightweight service called as Koordinator (Netto et al., 2018). To explain the architecture, consider an example of a stateful application with three replicas deployed in Kubernetes. The Koordinator service is installed as a container layer for the stateful application and is inserted between the client and the replicated containers.

When a request is made from an external client to write or update the state in the application, the request is directed via the load balanced service to the coordination layer where the Koordinator executes a consensus algorithm based on RAFT (Ongaro and Ousterhout, 2014) on the requests. Then these requests are forwarded to the application (container running the application) in the order defined by Koordinator. This step ensures that all the 3 replicas have achieved an agreement on the data value in the write or update request. When a read request is made to the application, the request is directly forwarded to any one of the replicas.

This approach might be useful to run stateful control applications in Kubernetes a distributed manner. This approach can also be used in the solution proposed by

Goldschmidt et. Al. in their paper (Goldschmidt et al., 2015) in the cache writer section (Figure 3.1) as it presents a distributed PLC system running in application containers.

Kubernetes provides workloads and features for running stateful applications. As explained in this section that using a container orchestration system adds some overhead in the latency. Despite of this latency the applications like distributed NoSQL databases (MongoDB) and other distributed applications like (Apache Spark and Hadoop) can run in a cluster. Control applications have strict cyclic time requirements and using persistent volumes for data storage and stateful sets might add overhead and cause delay in the execution of the control applications but their performance must be evaluated for control applications. Creating a custom resource in Kubernetes for running industrial control applications which would take care of the updates and redundancy without using persistent volumes could solve the problem.

3.3 Updating Industrial control applications with zero downtime

The frequency of updating industrial control application software has increased due the advancement in industrial internet of things technology as it increases the need for security patches and updates in the control applications. With the increase in the update frequency, there have been more and more advancements in the field of dynamic software updates (or zero-down time software updates or disruption free software updates) (Mugarza et al., 2018). This section discusses about the different techniques used for dynamically updating software for industrial control applications.

Wahler et al. have presented a solution for delivering dynamic software updates for real-time systems in their paper (Wahler et al., 2009). The authors propose an approach for real-time updates to industrial control application components. The approach consists of three main steps: 1. Scheduling Updates – Identify a point at which the update must happen, 2. Updating Components – Loading updated components and 3. Real Time State Transfer – State transfer from old to updated component.

The update must happen at a point when the execution of the control application is complete, but the cycle is not complete at this time CPU is idle. For example, if the control application cycle time is 5 ms, and the control application takes on an average 200 µs for

execution. The CPU is idle after 200 µs (or 0.2 ms) and the update can be scheduled at this point. The CPU remains idle until the next cycle.

The authors also state that the updating of the components and the state transfer must happen during this time (5ms – 0.2 ms = 4.8 ms) as if the update takes longer, then it will affect execution of the next cycle. This time is when the CPU is idle is also called as slack time. It can be possible that the update is done in multiple cycles so that it does not disrupt the execution of the PLC.

Figure 3.2 displays a single cycle of a control application. In the figure, the red section is the time during which the CPU is busy, and the control application is being executed. The points S and E are the start and the end points between which the update should be completed. Based on the results the authors state that for a state size of 1000 floating point variables which correspond to a state size of 4000 byte, the dynamic software update can be completed successfully for applications having a cycle time of 3.3 ms or more. The requirement that the update must be done only during the slack time is a critical factor in the dynamic software updates of real-time industrial control applications.

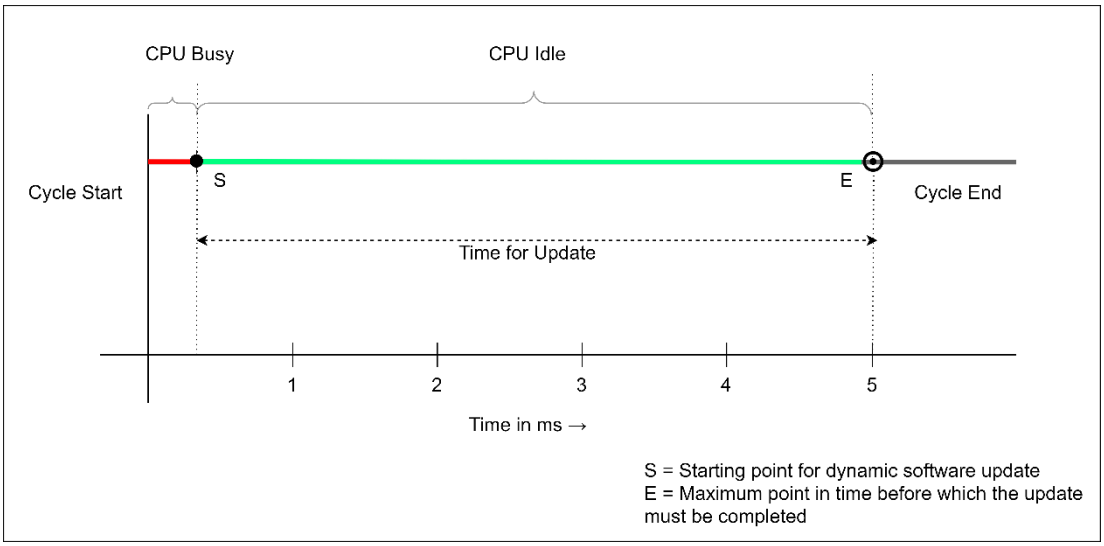

Figure 3.2 : Scheduling a dynamic software to real-time control applications running in a PLC

Wahler and Oriol have presented the state transfer mechanism in FASA in their paper on disruption free software updates in industrial automation systems (Wahler and Oriol, 2014). The authors describe the steps involved during a dynamic software update of a control application in FASA system. These steps are already discussed in earlier

paragraphs. Ideally the entire state must be transferred from the existing application to the updated application in the slack time when the CPU is idle. But sometimes the state is large, and it is not possible to transfer the entire state during a single slack time. To solve this problem, the authors suggest that state transfer can occur in across multiple cycles. They introduce an approach for state transfer and synchronization across multiple cycles by adding two function blocks – Teach and Learn. The teach block maintains an array of the memory structure in which all variables are initially marked dirty. The teach block clears the variables for which the state transfer is complete. After every cycle if there has been a change in the state of a variables which are cleared, the teach block again marks it as dirty, so that their updated state can be transferred again. This continues until all the variables are cleared. At this point the state of the existing and the updated application is synchronized. The learn block is responsible for processing the state of the existing component and then updating the state in the new component. The concept used in the teach and learn approach is a great approach for the state transfer and synchronization between the existing and the updated control application during dynamic software updates of control applications.

Load-Evaluate-Go is a method used to deliver zero-downtime updates in ABB's 800xA System (ABB, 2015). In this process, the existing application is running in the production environment. An 800xA system consists of a production environment and an engineering environment. The production environment is the environment which sends the output signals to the actuators whereas an engineering system only receives the input signals but does not send any output signals to the actuators. The engineering environment is for testing. A production environment is receiving signals from sensors, and then the processed response is sent to the actuators. If the field engineer wants to update the logic of the control application, then the field engineer could execute a load evaluate and go session. The "Load" part loads all the state of the variables from the production environment to the engineering environment. In the evaluate part, the field engineer evaluates the functionality by testing its behavior while receiving sensor data. Only sensor data is being sent to the engineering environment application and it is still not connected to any actuators. Once the evaluation is complete, the field engineer can either send go or revert the changes back to the way it was earlier. If the engineer selects "Go", then the

control application with the updated logic is switched and it runs in production environment.

FASA – (Future Automation System Architecture) is a framework for real-time control application (Wahler et al., 2015). FASA provides an architecture for executing control applications in a distributed system. The FASA system provides flexibility at runtime by supporting dynamic software updates without disrupting the execution of the application. There are three steps involved in the dynamic software updates 1. Loading of new components – Loading of the updated component or entire control application. At this point the updated code is loaded in the system, this does not affect the execution of the existing program 2. Create new configuration – As FASA supports distributed execution of control application, the components of a control application require a configuration which contains a schedule of the blocks to be executed and a list of communication channels between the components. When the application is updated, a new configuration must be created. The newly created configuration is also responsible for the state transfer and state synchronization of the existing and the updated control application. 3. Atomic Switchover – At this step the configurations are switched. This must be done during the slack time explained in earlier paragraph. When the updated components are ready to switch-over, the updated is completed. The authors successfully demonstrated the execution of a dynamic software update for a control application with cycle time of 1ms running in FASA system.

Mugarza et al have presented an analysis of dynamic software for industrial control software in their paper (Mugarza et al., 2018). Dynamic software updates ensure that the update of an application is completed without the need stopping or rebooting the application. The authors explain the three main steps involved in a dynamic software update: 1. Code transformation – Updating of the code, 2. State Transformation – Copying of the existing state to the updated application, 3. Update Point – Point at which the switchover from old to new application occurs .To perform such updates the redundant hardware (secondary device running in parallel) is used. The new version is installed to the secondary device and then the state transfer and switchover to the new application is completed. The Real time oriented dynamic software updates are most

relevant for industrial control applications. These Real time oriented dynamic software updates for industrial control applications are needed to be certified according to different standards like IEC 61508 (International Electrotechnical Commission, 2000) which states that if the software is of such a system Is updated then the entire system must be recertified before using it. As per the authors, the techniques presented by Wahler et al. (Wahler et al., 2009) and FASA (Future Automation System Architecture) (Wahler et al., 2015) are most relevant and safety compliant. Also, the software update mechanism in these two techniques run on the top of the operating system, so there is no need to recertify the entire operating system during an update.

3.4 Update Strategies in Kubernetes

Kubernetes by default provides a feature of rolling updates (Kubernetes, 2020d) which is useful to deliver disruption free software updates. Apart from that there are multiple different update strategies which are used by different organizations. Tremel and Signorini have mentioned about, tested and compared different update strategies that are used by industries in Kubernetes (Tremel, 2017) (Signorini, 2019). Container solutions in their github project have also provided a study and comparison of different update strategies in Kubernetes used by different organizations world-wide (Gold et al., 2020). These articles have described different update strategies used in Kubernetes – Recreate, Ramped or Rolling Update, Blue-Green, Canary, A/B Testing, Shadow. The update strategies like Ramped or Rolling updates, Canary and A/B Testing are relevant to microservice-based web applications and need multiple instances (replicas) of the same application running on multiple number of pods. Since industrial control applications run on a single instance, they need only one pod. Given below is the explanation of the important update strategies.

3.4.1 Rolling updates in Kubernetes

Rolling updates (also called as ramped update strategy) rollout updates to multiple replicas (pods) of the same applications step by step. Consider an example of a webserver having 4 pods which server the traffic coming from the network. If a rolling update is triggered, then the Kubernetes API will update one pod at a time until all 4 pods are updated as shown in Figure 3.3. This ensures zero-downtime since at any instant of

time during the update, only one pod is being shut down and restarted. Figure 3.4 shows the webserver requests in percentage and as the pods are gradually updated, the webserver requests for the older version gradually reduce too. The rolling update strategy provides smooth and lossless updates, but in case of testing the deployed application, it does not provide enough control to update only a few pods, test them and then update the remaining pods to the new version. Once the rolling update is started it continues until all the pods are updated and this increases the rollback time. Due to the nature of rolling updates requiring multiple pods of the application, the rolling update strategy is not suitable for the industrial control applications which do not support distributed computing, or have a separate data store apart from the execution units, as they cannot run in parallel on multiple pods. The cloud cost is low as at a time only one resource is deleted and updated and there is no need to create a new set of pods for all the instances.

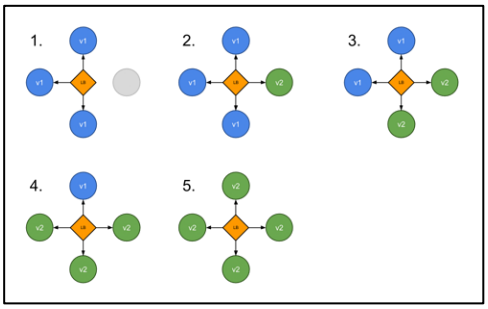

Figure 3.3 : Rolling or Ramped Update Strategy. Image downloaded from (Etienne Tremel 2017)

Figure 3.4 : Web server traffic in Prometheus for Rolling Update. Image downloaded from (Felipe Signorini 2019)

3.4.2 Blue-Green Update Strategy

In the Blue-Green strategy, a new updated set of pods which runs the updated version of an application is created using deployments or stateful sets while the existing pods are still running in production. The new set of pods running the updated version of the application are tested and then the web traffic is switched by the load balancer to the new version. Figure 3.5 explains the update step by step and Figure 3.6 shows the plot of the web-server requests during a Blue-Green update. It can be observed in the figure that the switch of the versions is happening at around 12:43 and during that window, both the old and the new instance exist. Blue-Green deployment strategy provides more advanced control to the users to update their existing applications to new versions. These advanced

strategies are not supported by Kubernetes by default and must be implemented externally. The cloud cost of the Blue-Green strategy is high, as a new set of pods are created during the update. Once the update is complete the old version pods are deleted. The Blue-Green strategy can be configured for industrial control applications, since it provides the functionality to verify and test the updated application before making the switchover.

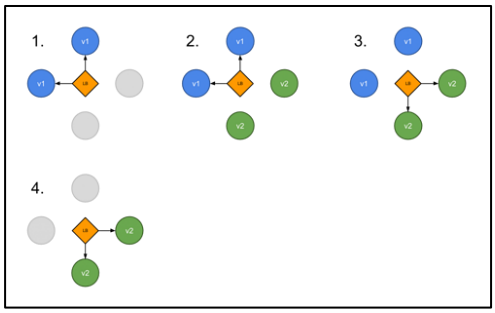

Figure 3.5 : Blue Green Update Strategy in Kubernetes. Image downloaded from (Etienne Tremel 2017)

Figure 3.6 : Web server traffic in Prometheus for Blue/Green Update. Image downloaded from (Felipe Signorini 2019)

3.4.3 Primary-Secondary Strategy in Kubernetes

Vayghan et al. have provided an excellent approach maintaining high availability in stateful applications in Kubernetes using a State Controller which allows automatic redirection to the healthy services (Vayghan et al., 2019). Consider the example as shown in Figure 3.7 where at a given time, four pods will be running in parallel and two one of which will be used in production (i.e It will be Active or primary) and the other two will be standby (or secondary). In case of a pod failure for the active pods, the state controller will detect that and will divert the production traffic to the secondary pod. This strategy is also called Primary-Secondary strategy. This concept of having a state controller and two sets of identical pods (Active and Standby) can also be used for updating applications in Kubernetes. This solution provides an architecture using both Deployments (Kubernetes,

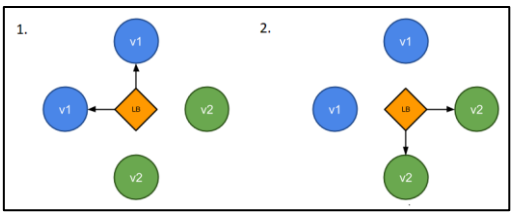

Figure 3.7 : Primary-Secondary Strategy in kubernetes. Image structure referred from (Etienne Tremel 2017)

2020b) and StatefulSet (Kubernetes, 2020f). Ali Kahoot has provided a detailed comparison of deployments and StatefulSet (Kahoot, 2019). Based on the comparison, using Deployments will be a better option for updating industrial control software containers as it provides more control to the user for managing the old and updated pods. This approach the cloud cost is high as at any instant of time one of the setups is on standby and the other is running.

3.4.4 Recreate

The recreate is the simplest update strategy supported by Kubernetes. In this strategy, during an update, all the instances of the original application are terminated. Then the new version is deployed, and the instances are started again. This strategy adds up a downtime and is not used by any critical applications. Figure 3.8 shows the web server traffic during the update of nginx server in Kubernetes. As shown, this startegy introduces a downtime in the execution of the update. As the recreate strategy doesnot support zero-down time updates, this strategy is not very relevant for industrial control applications.

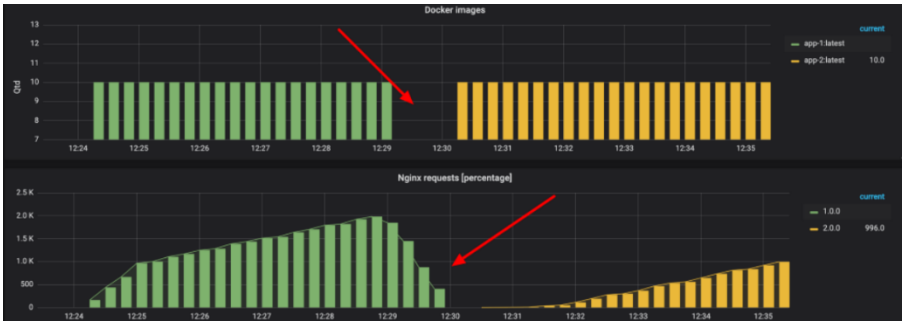

Figure 3.8 : Web server traffic in Prometheus for Recreate Update Strategy. Image downloaded from (Felipe Signorini 2019)

3.4.5 Canary Release

In a canary release, only a subset of the total instances is updated to new version and then tested. Canary release can be implemented by creating a new replica set which is running the updated version (Ibryam and Huß, 2019). This new replica set runs only a fraction of instances of the total number. Once the results for the updated instances are tested with the live data, gradually the updated number of instances is increased and the instances running the existing version are deleted. Figure 3.9 shows the canary update strategy and how the pods running the nginx server version 1 are replaced by the version

2 pods. Figure 3.10 shows the web traffic of the requests and it can be observed that the version 2 requests are gradually increasing. Canary releases are a viable option for systems where the effect of the new release is unknown. The traffic distribution management will be inconvenient with the default load balancers in Kubernetes, and so better managed load balancers are required.

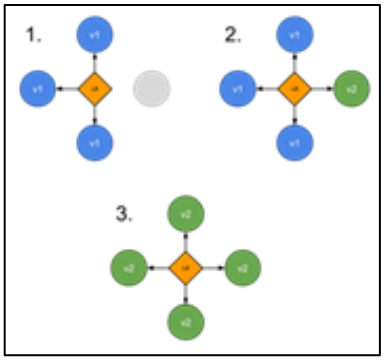

Figure 3.9 : Canary Update Strategy in Kubernetes. Image downloaded from (Etienne Tremel 2017)

Figure 3.10 : Web server traffic in Prometheus for Canary Update Strategy. Image downloaded from (Felipe Signorini 2019)

A/B testing can be implemented using canary release, but it is more of a technique for making user specific testing. Using A/B strategy, a specific feature can be tested only in certain regions, or certain group of users. Using this strategy specific users can be targeted.

Th A/B testing and canary strategies are more relevant to web applications where there are multiple users in different regions and demographics and the update is tested with only a small subset of users. If there is a problem with the update, the small subset of users is unable to access the application. For industrial control applications, the updated version needs to be thoroughly tested and verified, and the state must be transferred and only after that the switchover is done. A selective switchover without verification could be problematic as if the control application sends the incorrect signals to the actuators, it could be catastrophic for some cases.

Table 3.1 shows an overview of update strategies based on different factors. The Strategies marked in yellow are the most relevant update strategies for updating industrial

control applications. Blue-Green and Primary-Secondary strategies allow the testing of updated control applications before doing the switchover and support single instance applications. Referring to Research Question 4 (RQ4), the Blue-Green and the Primary-Secondary Strategy can be used for updating industrial control applications.

Strategy	Zero Downtime	Real Traffic Testing	Targeted Users	Cloud Cost	Rollback Duration	Kubernetes Support Inbuilt	Complexity of Setup	Single Instance Application Support
Ramped or Rolling	Yes	No	No	Low	High	Yes	Low	No
Blue-Green	Yes	Yes	No	High	None	No	Moderate	Yes
Primary-Secondary	Yes	Yes	No	High	Low	No	High	Yes
Recreate	No	No	No	Low	High	Yes	Low	Yes
Canary	Yes	Yes	No	Low	Low	No	Moderate	No
A/B Testing	Yes	Yes	Yes	Low	Low	No	High	No

Table 3.1 : Overview of update strategies. (Relevant strategies highlighted)

Table Referenced From : (Gold et al., 2020)

4 Concept

4.1 Overview

In the previous chapters the need for zero down-time updates in industrial control applications and the requirement to provide a state transfer mechanism was explained. This thesis discusses different use cases for the update of software for industrial automation and provides a concept to update industrial automation software automatically or manually without any interruption of the physical processes.

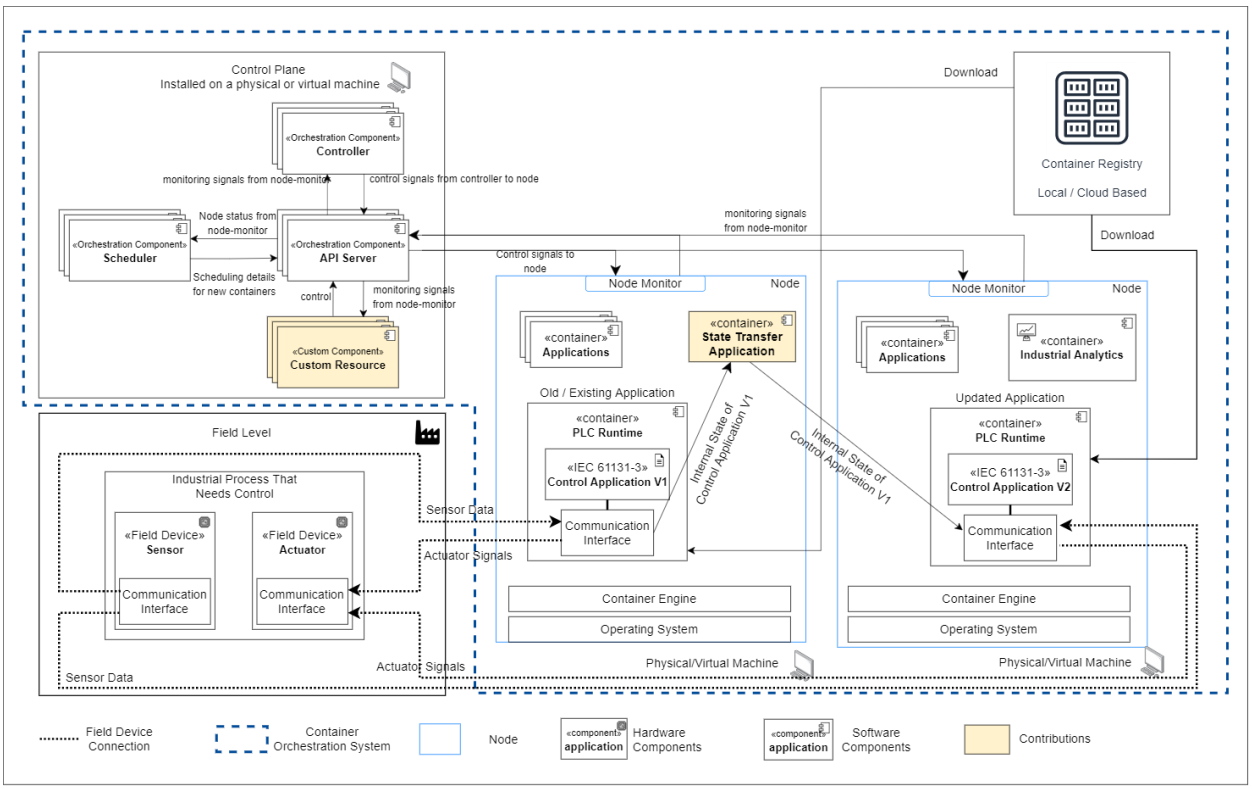

Figure 4.1 : Overview of Updating Industrial Control Applications in a Container Orchestration System

Figure 4.1 shows the entire concept of updating industrial automation software using a software container orchestration framework. The control plane manages a cluster of nodes and the pods running on them. The functionalities of the Controller and Scheduler components were discussed in Sections 2.1.2 and 2.1.2.2. A Custom Resource is a component providing functionality to extend existing features and implement additional logic (for example, performing the state transfer of a stateful application during an update)

which is not supported by a specific container orchestration system. Kubernetes supports the use of custom resources.

The field level block contains field devices, i.e. sensors and actuators attached to industrial equipment, such as pipes, tanks, pumps or motors, which automate a production process. The communication interface is responsible for connecting the field devices with the control applications. In the future, many of these field devices will contain OPC UA servers to enable communication with control applications. The nodes are physical or virtual machines hosting all the containers and, within the containers, the industrial automation software. The control applications consist of an IEC 61131-3 (PLCopen, 2013) control application and a communication interface which connects with the field devices. The state transfer application is responsible for restoring the state of the control application during an update. Along with industrial control applications other industrial automation software like the industrial analytics dashboard is also running in the cluster.

The blocks highlighted in yellow are the contributions of this thesis. These concepts are explained in the following

- State Transfer Application

As explained in Sections 2.2.1.1 and 3.3, state transfer is a critical requirement during a dynamic update of an industrial control application. The existing systems like System 800xA (ABB, 2015) have an engineering environment and a production environment running in the same physical hardware. In this case state transfer is done using shared memory. Since the goal is to execute control applications within a container orchestration environment inside a container, shared memory communication is not possible in case the old and the updated control application run in different physical nodes. In such cases a network transfer is required.

Primarily, two approaches – direct memory transfer (inter-process communication via shared memory) and network transfer were implemented, and the average time required for state transfer in each case was measured for different data sizes.

Another possible solution is to use persistent volumes for state transfer, but this approach was not tested because using persistent volumes for storing states of control applications might cause high latencies but needs to be tested. A generic state transfer application was developed to enable state transfer of between control applications. The state transfer application can run as a stand-alone application or it can also be deployed on the cluster as a containerized application. The containerized state transfer application on the cluster can be used during the update of all the control applications in the cluster and can be easily accessed across the cluster. The state transfer application includes a user interface (UI) to invoke a state transfer manually. To invoke the state transfer automatically, the UI is not required, and in such cases the state transfer application can also be invoked using any HTTP client.

- Custom Resource for Industrial Control

Container orchestration systems like Kubernetes provide workloads for different types of applications but providing generic workloads and at the same time handling specific use cases (for example, completing state transfer for industrial control applications during an update) is not possible in such systems.

To solve this problem, Kubernetes provides an option to customize the deployment and updates of applications using custom resources which are added to the Kubernetes control plane. Custom resources are explained in detail in Section 2.1.2.2. Custom resources can be used for delivering automatic updates for industrial control applications.

This thesis proposes using custom resources to perform automatic industrial control application updates. These updates include creation of a new pod with the updated application, doing the state transfer to the new application and then switching the connection of the field devices to the new control application. These custom resources can be implemented by different automation vendors for their specific requirements and use cases. This makes the industrial control applications industry ready and allows them to run on a Kubernetes system or any other container orchestration system.

4.2 Updating Industrial Control Applications

To update a control application, the State Transfer Application carries out four basic steps:

- Initialization: Create and setup new updated control application in another container. In some cases, the redundant (also called as a secondary or standby) application running in a container is used. If there is a change in the logic of the control application, the automation engineer changes the IEC 61131-3 functional block diagram or the structured text file for the updated application. The variables which store the critical state information of the control application are marked as “retained values”. Once a structured text file is uploaded, a new container image is created with the updated application and uploaded to the container registry with a new version tag. In some cases, the same control application is used, but there is an update in the communication protocol or the operating system of the container image of the control application. In this case also a new Docker image is created and uploaded to the container registry. The updated applications are created using the images with the new version tags. Once the updated containers are up and running the initialization phase is complete.
- State Synchronization: In the state synchronization step, first the output from the sensors is connected to the control application. The control applications running in the containers must have an interface to provide the connection details of the IO devices and these connections should be configurable at runtime. For example, if the communication protocol used is OPC UA client/server, then the updated control application should have an interface to provide the OPC UA server IP addresses of the sensors. Once the sensor devices are connected to the new updated control application, the state transfer process is initiated. To start the state transfer process, first the execution engines of the existing and the updated control applications are paused to avoid state overwrites during state transfer. The state of the existing control application is transferred to the newly initialized and updated application using the state transfer application. As soon as the state transfer is complete, the control applications are resumed. This will synchronize the state of the existing and the

updated control applications as the sensor output is now connected to both the existing and the updated control application.

- Verify: Once the states are synchronized, an automation engineer starts monitoring select output values. For example, especially such values that are affected by the updated application logic. If the automation engineer deems the outputs of the updated and state-synchronized control application appropriate, a change-over of the engine output data to the field devices (i.e., actuators) can be manually invoked. If there are minor changes there is no change in the control application, manual intervention is not required as the verify phase can be done automatically as well by comparing the variable values the existing and the updated application.
- Switch: If the application is working as expected, the connection between the actuators and the existing control application is terminated and a new connection between the updated control application and the actuators is established.

4.2.1 Concept of State Transfer

The state transfer is a complex task due to the requirement of the transfer within the slack time of the control application as explained in Section 3.3. The state transfer is done by extracting the state of the variables of the old control application and then copying their values, and then pasting these values to the updated control application. These variables can include information like counter values, timer data, or other information, which is calculated or provided by the field engineer.

Figure 4.2 shows an example of the control application of an industrial boiler. The requirement is to maintain the level of an industrial fluid inside the boiler. The PLC runs on a hardware platform or a container and an IEC 61131-3 control application is running on the PLC. The logic of the existing control application is to maintain the level below or equal to a certain threshold by controlling the input and output flow valves which are connected to the industrial boiler. In the updated control application, the requirement changes to maintaining the value of the boiler below the threshold. The value `LevelToMaintain` is provided by the field engineer.

During an update, the if condition of the control application is updated as shown below. If the field engineer now decides to switch the existing PLC with the updated PLC, the

desired level of 600 will not be maintained in the boiler as in the updated PLC the value of the parameter `LevelToMaintain` is still 0 which is the default value. To maintain the level of 600 in the boiler, the value of the parameter `LevelToMaintain` should be set to 600. Since this is only one value the field engineer can manually set the value, but in a large scale production level control application there are hundreds of parameters (Krause,

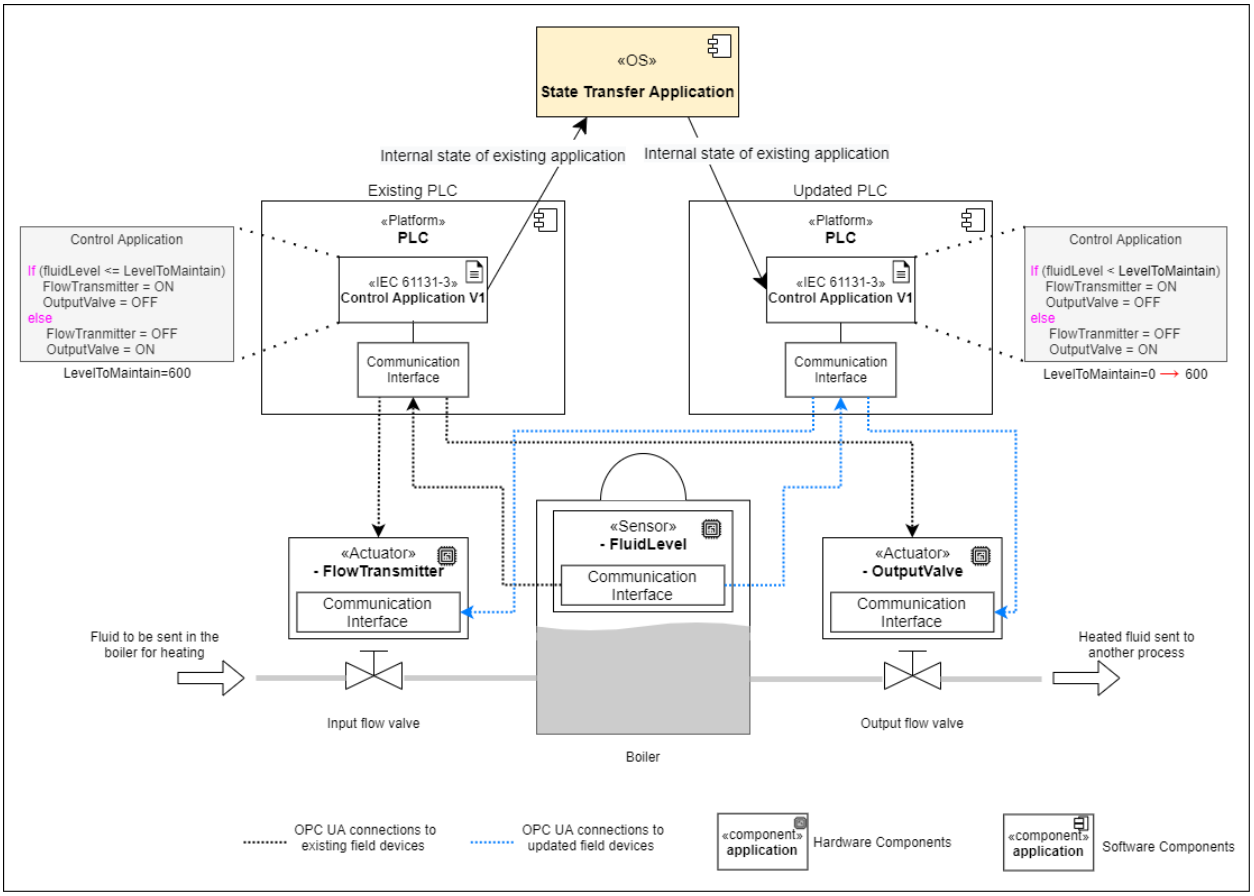

Figure 4.2 : Update Control Application for an Industrial Boiler

2007) which are supposed to be retained and some of these parameter like value of counter variables and these values cannot even be set manually as they are updated after every cycle of the control application. That is the main reason that the state transfer mechanism is required while updating industrial control applications. The state transfer application can restore the state of the updated control application. Once the states of both applications are in synchronization the connections to the field devices are switched. The state transfer can be done via the communication interface or by a direct memory transfer using the state transfer application.

Based on the existing literature (Section 3) on update strategies in Kubernetes and updating industrial control applications, the following state transfer approaches were considered.

- Transfer via Persistent Volumes
- State Transfer via network using OPC UA
- Transfer using Inter Process Communication (IPC) via POSIX memory.

4.2.1.1 Transfer via Persistent Volumes

Generally, in Kubernetes Stateful Sets (Kubernetes, 2020f) are used to preserve the state of an application with the help of a persistent volume. A Persistent Volume (PV) (Kubernetes, 2020e) is like a Hard-Disk in which the data is preserved even if the execution units i.e. pods are shut down.

During an update the current state of the control application can be stored in the persistent volume as a file and the updated application can then access the same persistent volume and get the snapshot of the state from the file. The existing (old) application and the updated (new) application are running in two separate pods managed by two separate deployments. The state transfer process is shown in the Figure 4.3.

Mercl and Pavlik have presented a performance analysis of different storage (persistent volumes) mechanisms in Kubernetes. The authors evaluated different parameters about

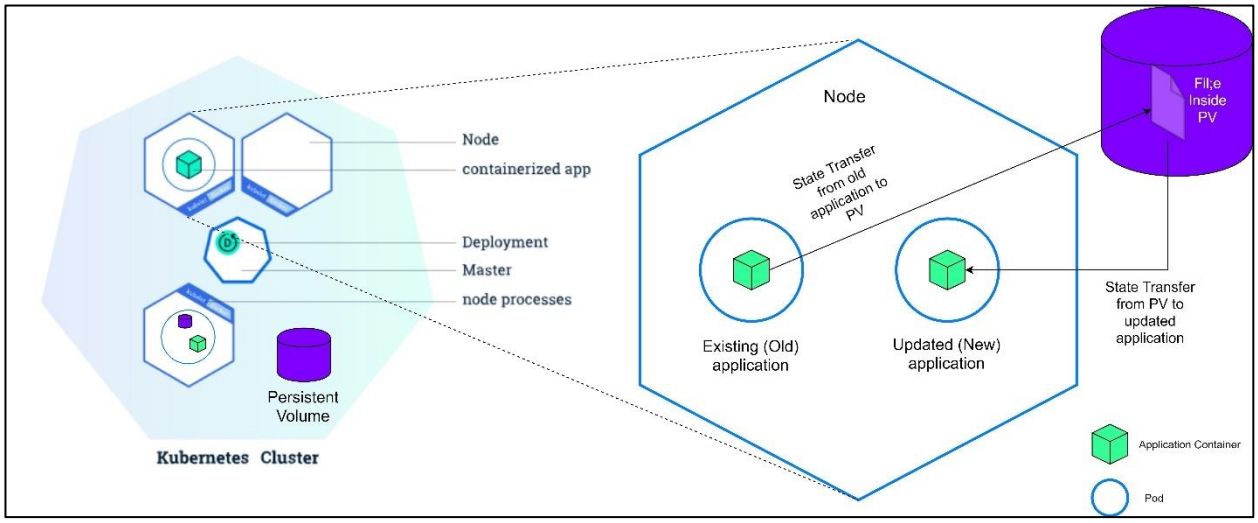

Figure 4.3 : State transfer using Persistent Volume in Kubernetes. Kubernetes Cluster Image referred from (Kubernetes 2020h)

the different storage mechanisms, of which the most relevant for industrial control applications are random read-write bandwidth and the read-write latency. Portwox[1] had the highest read and write bandwidth of 749 MiB/s and 21.6 MiB/s respectively. At the same time, the native Azure PVC[2] and OpenEBS[3] had read latencies of 29.9 MiB/s and 23.1 MiB/s and write latencies of 28.5 MiB/s and 2.6 MiB/s respectively. The latency test shows a latency of 0.3 ms for read and 4.5 ms for write operations. At the same time the Azure PVC and OpenEBS show high latencies of 4.4 ms and 37.1 ms for read and 38 ms and 466 ms for write operations. The latency and bandwidth numbers vary to a great extent for different storage mechanisms. (Mercl and Pavlik, 2019).

It was explained in Section 3.2 that running a NoSQL database – MongoDB - in a container orchestration system which uses a persistent volume adds up 28% performance overhead for insert or update operations and 20% overhead for ready heavy workloads as compared to a VM-Based non-orchestrated deployment of the same database. The tests performed in that article were tested with 23MB of data, which is a huge data size when compared to the state size of a control application in which the sizes are approximately in the order 8 kb – 4 Mb (1000 – 500,000 Variables of 64 bits each), so it cannot be stated that there will be a performance overhead for control applications while using persistent volumes. The state transfer times for industrial control applications were not tested using this approach, but if the right and optimized storage mechanism is used, the state transfer times could be comparable existing approaches like using shared memory using IPC.

4.2.1.2 State Transfer via network using OPC UA

This approach uses OPC UA network communication for the data transfer. As explained Section 2.2.3 OPC UA is a standard protocol for industrial communication which uses a Transmission Control Protocol/Internet Protocol (TCP/IP) connection internally for the client-server communication model. Using a network transfer makes it easy for data

[1] Portworx – Container-native storage for kubernetes (https://portworx.com/), Accessed 31.08.2020
[2] Azure PVC - Attaching a volume to the pod via Persistent Volume Claim in Azure Kubernetes Cluster
[3] OpenEBS - Container Attached Storage (https://openebs.io/), Accessed 31.08.2020

transfer in a cloud environment as all the execution units – pods are accessible by their IP address.

In this approach, the existing and the updated application are running within different pods managed by two separate deployments. These pods could be also running on a different Kubernetes node, which might add some delay in the transfer. The state transfer happens via an external state transfer application. The state transfer service will retrieve the snapshot of the current state of memory from the old application and then push that data to the memory of the updated application. The memory transfer is sone using the OPC UA “method-call” feature, which allows the triggering of methods in the source (existing application) to extract a snapshot of the current state of memory and in the destination (updated application) push the extracted data to its memory. In the OPC UA server for receiving data on request, complimentary OPC UA methods are used. Figure 4.4 explains the process of state transfer via OPC UA.

Network transfer is often a faster way of data transfer as compared to storing files to disk, but if the data increases, then the time required for data transfer will increase as well. Network transfer will not be as fast as direct memory transfer since it has the overhead of the network, but for use-cases where the option for direct memory transfer is not available, this is the fastest and most reliable option.

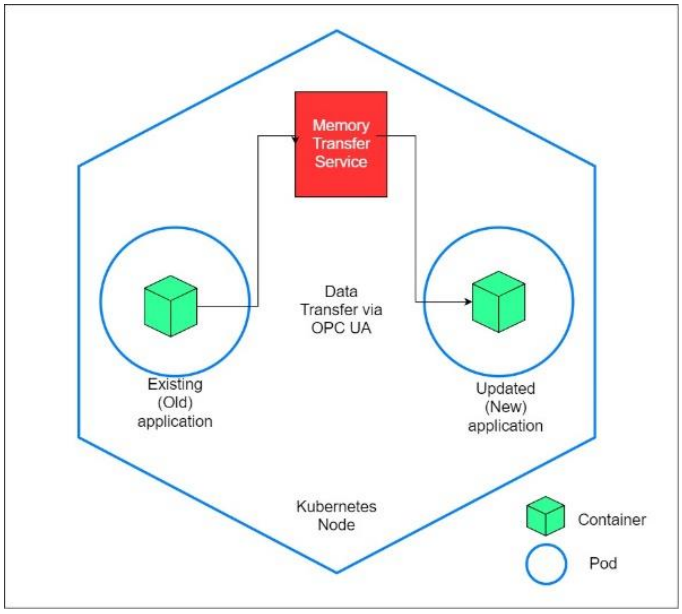

Figure 4.4 : State Transfer of control application via OPC UA in Kubernetes

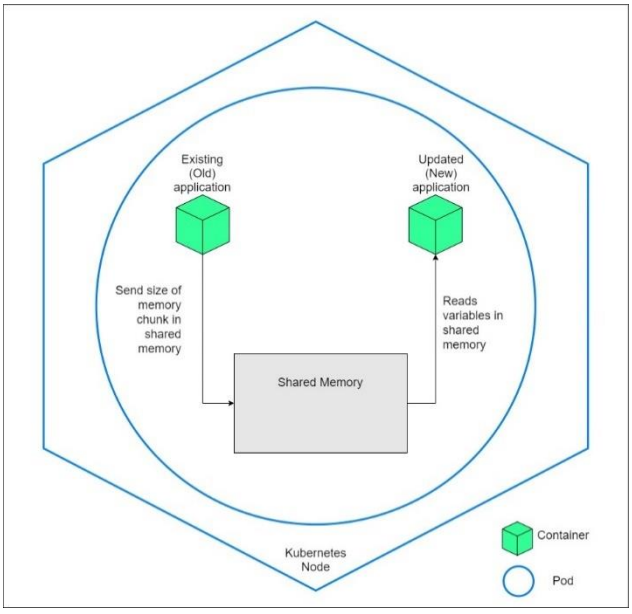

Figure 4.5 : State Transfer of control application using IPC via POSIX in Kubernetes

4.2.1.3 Transfer using Inter Process Communication (IPC) via POSIX memory

This approach refers to the transfer of the data using inter process communication (IPC) via the POSIX shared memory in an operating system. Venkataraman and Jagadeesha have presented the performance of IPC mechanisms like shared memory, pipes and sockets for different data sizes. The results show that the transfer latency for IPC via shared memory was the lowest. The maximum latency was around 100 micro-seconds for a data size if 32 kB and around 125 micro-seconds for 64 kB which are very low (Venkataraman and Jagadeesha, 2015). This method is faster than the OPC UA transfer as it does not have the overhead of the network traffic. The POSIX shared memory is present in the operating system and in the case of Kubernetes, it can be assessed only within a pod, because only containers within a single pod share the same process namespace. To achieve this the updated and the existing applications both need to run on the same pod as two separate containers.

The state transfer using IPC via POSIX memory is similar to the state transfer using OPC UA with one major difference. Both the existing (old) and the updated application (new) application run on the same pod generally managed by a deployment. When the application needs to be updated, the control application in the updated container is uploaded and then the state transfer is carried out. The state transfer is explained in Figure 4.5

A custom resource CloneSet developed by the OpenKruise project is a good alternative for deploying the pods while using IPC via POSIX memory transfer (OpenKruise, 2020). CloneSet allows to update the container image within the pod without restarting a pod. This is not the case with deployment – if there is any change in the Docker image of the control application, then the entire pod will be restarted.

The only disadvantage of this transfer is the requirement of having the source and the destination in one single pod and thus a single node. This approach will not for work for state transfer across control applications running on separate nodes or even pods.

4.2.2 Update Strategies for Updating Industrial Control Applications

The update strategy is the way in which the steps mentioned in Section 4.2 are executed in a Kubernetes environment. Industrial control applications are stateful applications

which also run on only single pods and cannot run on multiple pods and this limits the available options for choosing an update strategy for such applications. As explained in Section 3.3 the most relevant update strategies for control applications are the Blue-Green and Primary-Secondary strategies. One major difference between both strategies is that both two parallel instances of the application (of which one is active and the other is standby) continuously run forever in the primary secondary approach, and in the Blue-Green approach the old and the updated applications run only during the update and after the update the old application is shutdown.

4.2.2.1 Blue-Green Update Strategy

The Blue-Green update strategy is widely used for updating web applications. The main reason for the use of this strategy is because in the Blue-Green strategy, the updated application can be tested with live production data without affecting the working of the existing application. This strategy drastically reduces the rollback period and if the updated application is not functioning as expected, then the update process can be terminated, and it does not affect the existing application.

Four basic steps are carried out in Blue-Green Update typically for web applications in Kubernetes:

1) Initialization – Create a new Kubernetes deployment of the updated application,
2) Send a copy of input requests to the updated application – Send a copy of all requests (live production data) to the updated application so that the updated application can be tested, and the outputs can be compared with the old application. The input requests (live production data) correspond to the sensor data in an industrial control application. The output of the web application corresponds to the signals that are sent to the actuator.
3) Test if updated application is working – Perform a smoke test to test if application is working as expected. This can be an automatic or manual test. The old application is still serving all the production data and the updated application is only being tested.
4) Switch the traffic – Once the updated application is working as expected the live production data traffic is shifted to the updated application.

- Why Blue-Green Strategy?

The industrial control applications are needed to be updated without any disruption and here the Blue-Green update strategy in Kubernetes can be applied for updates. Blue-Green update strategy provides a setup for testing the updated application with live data without affecting the flow of the industry if there is something wrong in the updated application. The rollback is generally not required as the application is already tested with live data. For the update of an industrial control application, almost similar steps are involved with one additional step – State transfer and synchronization from old to new application. Control applications are stateful application which generally run on a single pod, and the other update strategies in Kubernetes do not work in this case.

Figure 4.6 shows the working of a Blue-Green update strategy for an industrial control application. Since the Blue-Green strategy is not supported by Kubernetes, the update must be completed manually or by adding extra features to Kubernetes (extending

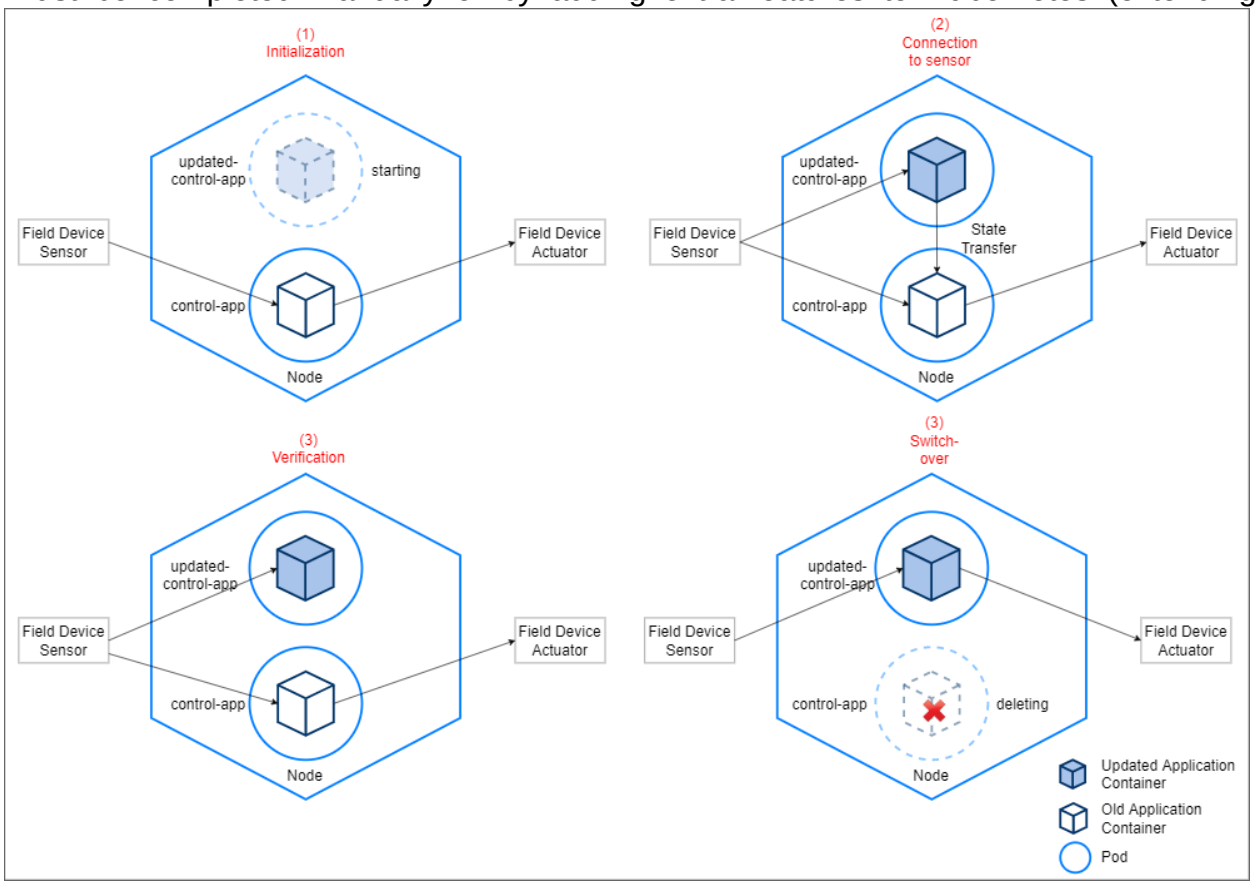

Figure 4.6 : Blue/Green Update Strategy in Kubernetes for Updating Industrial Control Application

Kubernetes) which could enable automatic updates for control applications with Blue-Green strategy.

4.2.2.2 Primary-Secondary Strategy

As explained in Section 3.4.3, the Primary-Secondary is a deployment strategy defined in by Vayghan et al. in their article in which they propose a high availability solution for stateful applications in Kubernetes (Vayghan et al., 2019). In this article, the authors suggest maintaining two sets of same application running in parallel of which one is active (primary) and another is on standby (secondary). When the active application stops working due to some error, the standby application (secondary) takes over and production data traffic is processed by the secondary application.

This ensures high availability in the applications. In this approach a state controller is configured to monitor the primary application and when the primary application stops working makes the switch-over of the production data traffic to the secondary application. This switch-over corresponds to the switch-over of the actuator connection from the active controller to the standby controller. This concept of high availibility can be used for delivering zero-downtime updates in industrial control applications. Given below are the steps and Figure 4.7 displays these steps:

1. Initialization: Initially the primary(active) and the secondary(standby) application are running in parallel with the same control application. Only the output of the primary (active) application is sent to the actuators, but the states of both the application are synchronized. When the control application needs to be updated, the secondary application is updated by shutting down and then restarting with the updated application. Since the active and passive applications are independently deployed updating the secondary application with downtime does not affect the primary control application.
2. State Synchronization: Due to the down time in restart the secondary application needs to be synchronized with the state of the primary application. This step can be done using the state transfer concept defined in earlier section.
3. Switch: Once the state synchronization is completed successfully, the primary application which is the older version as displayed in Figure 4.7 and the secondary

application which is the updated version are in sync, and the switch can happen after testing for a few cycles. This verification and switch both can be done manually as well as automatically. At this point the updated application becomes primary application (active), and the old application becomes the secondary application (stand by) and so the update is complete as the updated application starts sending signals to the actuator.

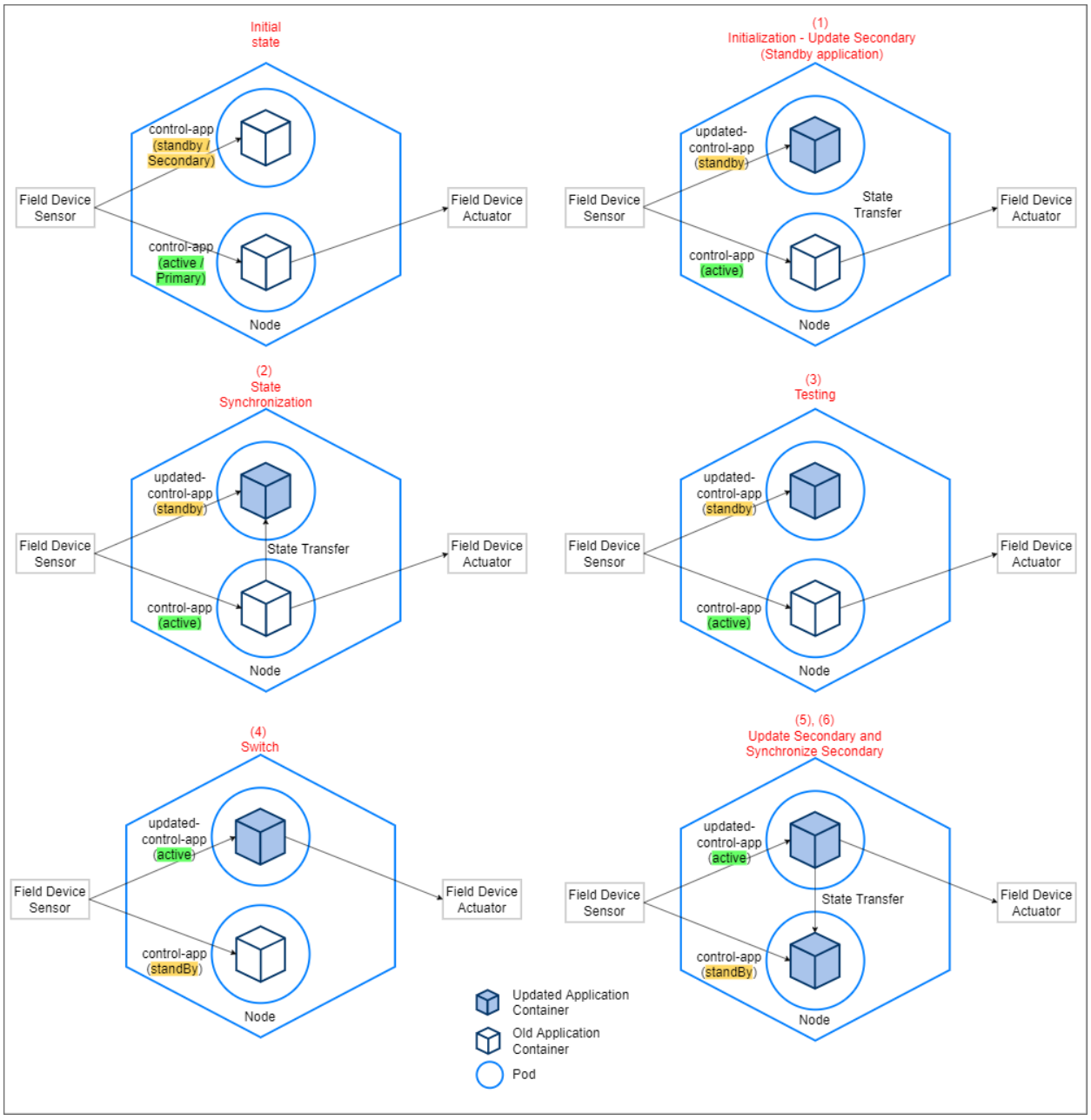

Figure 4.7 : Primary-Secondary Update Strategy in Kubernetes For Industrial Control Application

4. Update secondary: After the switch is complete the update is complete, but the secondary (stand by) application still needs to be in a synchronized state with the primary (active) application as in case of a shut-down, the secondary can replace the primary without any delay. The secondary application is supposed to be updated before the states are synchronized. That is the reason the secondary application version is also updated to be the same as the primary version.
5. State Synchronization: In this step the state of the secondary application is synchronized from the primary application. State synchronization is mandatary in this approach as the secondary application acts as a standby device and the internal states at any instant should be identical.

4.2.3 Type of Update (Manual or Automatic)

Based on the verification phase mentioned in the update steps in Section 4.2, the update can be classified in two different types, automatic and manual verification

4.2.3.1 Manual Updates

In a manual update all the above-mentioned steps are executed manually. Manual update requires a manual verification of the working of the updated control application . Manual verification of the updated control application is done when there is a major change in the control application, and a mere comparison of the variables in the old and the updated application is not enough.

The manual update is based on the Load-Evaluate-Go concept explained in Section 3.3. In the manual verification a field engineer can assess the variables of the old and updated application and then observe the variables for a few cycles and then decide if the updated application is behaving as expected to make the switch. A major change means something which cannot be automatically verified, for example, the addition of a new variable in the updated application. Since there is nothing to compare the newly added variable in the old control application, it cannot be verified automatically. The manual verification approach was proposed implemented and tested in a Kubernetes cluster in this thesis.

Figure 4.8 shows the steps involved in a manual update. These steps are the same as explained earlier. The major difference in this approach as compared to the automatic

approach is in step 3 – Verify. It can be seen from the image that the field engineer manually verifies the working of the updated control application.

1. Initialization: Manually create a new container for the updated control application with the updated Docker image of the control application. For example, it can be seen in Figure 4.8 that a new pod is initialized with an updated version of the control application (Control Application V2). In this example Kubernetes is the container orchestration system.

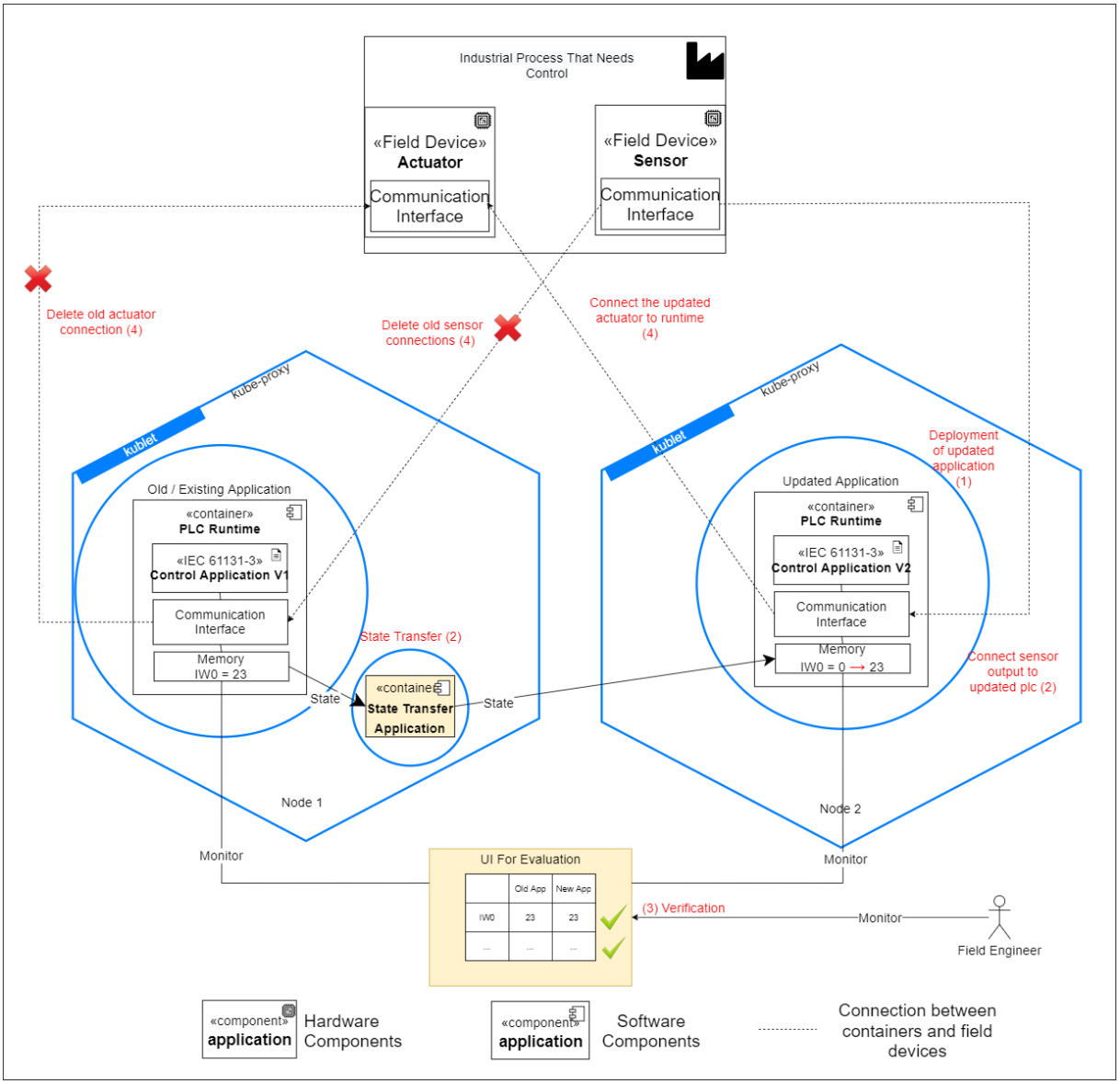

Figure 4.8 : Manual software of industrial control applications in kubernetes

2. State Synchronization (Load): In this step, first the communication interface in the updated control application is connected to the communication interface of the sensor (only sensors or only the inputs to the control application). This can be done by providing the URL or IP address or connection details of the sensor in the PLC runtime using a user interface for the control application. The state transfer application is used for doing the state transfer. The state transfer application is an executable application which copies the state from the old application to the updated application. This executable application can be executed from a user interface (UI) or by any other application automatically. In this case the state transfer can be completed by using the UI for the state transfer application. When the connection between the sensor and the control application is established and the state transfer is complete, the state of the old and the updated application is synchronized.
3. Verify (Evaluate): The verification is done by the field engineer manually by observing the old and the updated application variables on an UI or an external OPC UA Client like UA Expert[1] (In case if OPC UA is used as a connection between the field devices and PLC). The variable values can be side by side and based on the values the field engineer can decide if the updated application is working as expected or not.
4. Switch (Go): The switch step is for switching the actuator connection from the old to the updated control application. This can be done by providing the connection details of the Actuator to the updated control application and removing the connections of the old control applications with both sensor and actuator using a user interface for the control application or a script.

4.2.3.2 Automatic Updates

In automatic updates only the user will trigger the update and then all the following steps will run automatically. If there is a change in the control application, then the field engineer must create the updated IEC 61131-3-based control application and then add it to the PLC runtime. A new image with the updated changes or control application is built and

[1]UA Expert – A Windows or Linux OPC Client (https://www.unified-automation.com/downloads/opc-ua-clients.html), Accessed 22.10,2020

then uploaded to the container registry Once the updated image uploaded to the registry, then the automatic update can be triggered.

Automatic update can be done when the verification of the working of the updated control application can be done automatically. Automatic update is a better alternative to manual update as it saves up a lot of time and resources. Also, there a chance of manual error during manual verification. For small changes to the control application or to the environment variables of the control application, an automatic verification of the working of the updated control application can be done. Since these are small changes, the old and the updated variables can be compared and if their values are within the same range with acceptable error, then the updated application is working as expected. Small changes mean minor changes made in the control application, or minor changes made in the environment variables or field devices.

Automatic updates are necessary when there is a change/update in the infrastructure implemented by the infrastructure team which generally has less knowledge of the control application running. In this case since there was no change in the control application, an automatic update can be easily implemented. This way the infrastructure team can continue with the development activities in Kubernetes without having detailed knowledge of updating control applications.

Two approaches are designed for doing an automatic update of pods running control application:

1. Using Custom Resource Definition (CRD)
2. Using a pre-stop script in the pods

Automatic Update using Custom Resource Definition (CRD)

Custom resources are extension to the functionalities of a container orchestration system. Kubernetes, Apache Mesos and some other container orchestration systems provide the option to extend the existing features of a container orchestration system. Apache Mesos provides the feature of Mesos Modules[1] to extend the functionality of Apache Mesos and

[1] Mesos Modules: A way to extend the functionality of Apache Mesos - https://mesos.apache.org/documentation/latest/modules/, Accessed 21.10.2020

Custom resources (CR) are extensions to the Kubernetes. Custom Resource Definition (CRD) defines a CR. The automatic update is explained with respect to Kubernetes, but a similar concept can be used for any other container orchestration system which supports the use of custom resources. Custom resources are used for adding any functionality which is not supported by default in Kubernetes. For example, to fulfil the requirement of automatic update of control application a custom resource can be used for managing pods and doing a state transfer. Any changes in the CR are detected by a custom Kubernetes controller which controls the creation deletion and updates of resources in a CR. A Kubernetes operator includes either one or multiple custom Kubernetes controllers. (Dobies and Wood, 2020). Kubernetes operators simplify the process of managing complicated stateful applications. Xavier Coulon in his article has mentioned a step-by-step explanation for creating a Kubernetes operator using the Operator SDK [1] (Coulon, 2019) . Consider an industrial control application which is maintained by a custom resource. The pods of the control application are controlled by the custom resource controller.

An automatic restart and state transfer of control applications require four main steps.

1. Initialization: When there is an update in the control application or a security update, an updated container image is pushed to the container registry. To trigger an automatic update, the updated image name and other parameters are provided in the Kubernetes yaml file. This update command is received by the Kubernetes API and the custom resource controller detects an update action has been triggered. Immediately the existing pod is switched to a terminating state with a configurable grace period (default value = 30 sec). Also, a new pod with the updated Docker image is created in the cluster.
2. State Synchronization: Once the newly initialized pod is in the running state, the custom resource controller is configured to initiate the state synchronization step by creating a connection between the sensor devices and the updated control application. The custom resource controller then initiates the state transfer by executing an HTTP post request between the old and the updated application. Once

[1] Operator SDK : https://github.com/operator-framework/operator-sdk, Accessed 31.08.2020

the connection between the updated application is established and the state transfer is complete, the state synchronization is complete.

3. Verify: In the verification phase, the old and the updated variables can be compared for a few cycles and if the values match or are within the expected error margin, then the verification step is complete. If the values do not stop, then the update process is

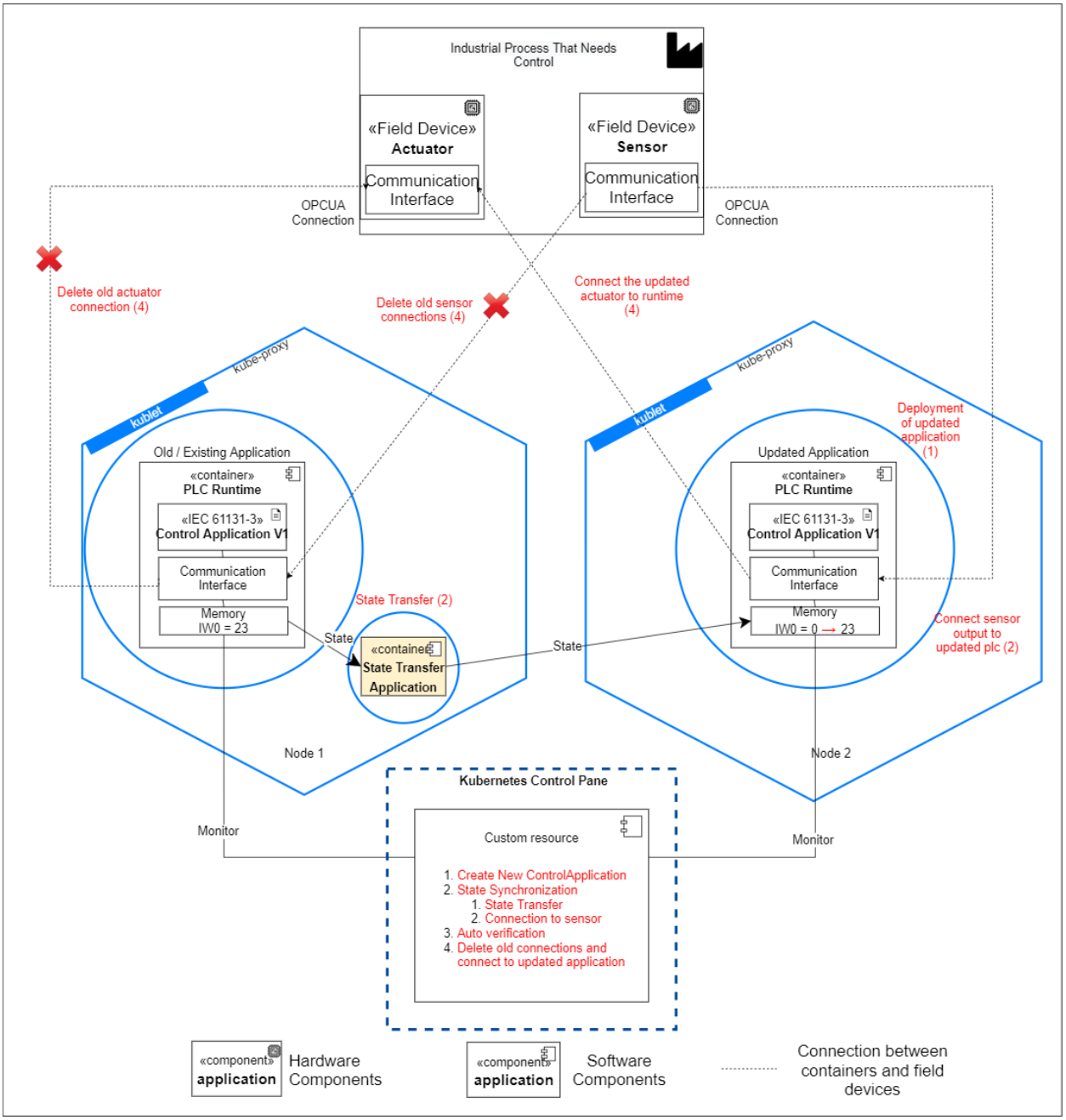

Figure 4.9 : Automatic update of control application in kubernetes using Custom Resource Definition (CRD)

terminated. Since the automatic update is only done for minor changes, it would rarely happen that the old and the updated variables don't match.

4. Switch: The switch step is for switching the actuator connection from the old to the updated control application. This can be done by providing the connection details of the actuator to the updated control application and removing the connections of the old control applications with both sensor and actuator using an HTTP request for the control application. If there is anything wrong in the verification phase, the switch will not happen, and the pod running the existing application will not be terminated.

Figure 4.9 displays an automatic update of industrial control application using a CRD. The step described above are executed by the custom resource controller implemented in Go-Language. CRD's provide a sophisticated approach for automatic restart of pods running control application. Also, it gives more control to the developers as the Kubernetes operators provide a better access and control of the Kubernetes API.

Automatic Update using a pre-stop script hook in the pod

Pre-Stop script is a specific functionality provided by Kubernetes and other container orchestration systems like Apache Mesos[1] also have similar concepts implemented for container's lifecycle. A pod in a Kubernetes has a specific lifecycle. When a pod creation request is initiated by the Kubernetes API, the request is forwarded to the Kubernetes scheduler. The scheduler then assigns schedules a new pod in any Kubernetes node, based on the scheduling algorithm and conditions provided in the pod configuration file. The kubelet inside the node starts a new Docker container of the image specified in the pod configuration file. Once the Docker image is running the application is ready (Beda, 2017).

There are two main container-lifecycle hooks involved during the pod lifecycle – postStart hook and preStop hook (Kubernetes, 2020a). These hooks are parts of the pod start and pod termination lifecycles, respectively.

- Pod Start Lifecycle

[1] Container Life Cycle in Mesos: https://mesos.apache.org/documentation/latest/nested-container-and-task-group/, Accessed 21.10.2020

A pod is the smallest execution unit in Kubernetes. When a pod is started, internally the container defined in the pod is started. After starting the container, a post-start hook is executed in the container. The command executed with the post-start hook can be added in the Kubernetes yaml file. The container also starts running along with the post-start hook. The postStart hook is executed after the entrypoint script is started in Docker.

- Pod Termination Lifecycle

If a delete pod request is initiated by the user, then the Kube API makes a terminate request to the pod. In the Kubernetes yaml file, a configurable grace period of 30 sec is given to the pod before it shuts down completely to close all the processes running in the container. During the grace period a pre-stop hook is executed in the container before the container shutdown. If the applications are still running after the grace period, a signal-kill command is executed for forcefully shutting down the application. This is the stop lifecycle of a pod. These steps are not executed when the pod is forcefully shutdown, for example in case of a node failure.

The automatic update of a control application can be done using the preStop hook in a container lifecycle. Whenever the control application needs to be updated, a delete pod or delete deployment command must be run in Kubernetes. This changes the state of the pod from `running` to `terminating` and the termination lifecycle is initiated.

The steps mentioned earlier are carried out in a preStop script. The only difference here is that in the earlier approach these steps were executed from a custom controller, and in this case the same steps are executed from within pod which runs the old container. Figure 4.10 shows the update of a control application in Kubernetes using pre-stop hook.

1. Initialization: When there is an update in the control application or a security update, an updated container image is pushed to the container registry. To trigger an automatic update, the updated image name and other parameters are provided in the Kubernetes yaml file. The updated yaml file is then sent to the Kubernetes API manually using kubectl or the Kubernetes UI dashboard and this initiates the automatic update process. The new pod is started in the cluster with the updated container image. As soon as the new pod is in a running state, the old pod goes to a

`terminating` state and this triggers the pre-stop hook in the terminating pod. The main function of pre-stop script is to get the IP address of the updated pod so that the state transfer application can be run from the source application (existing pod) to the target application (newly created pod). To get the IP of the target application, any Kubernetes client or kubectl can be used. Since the containers don't have a direct access to the Kubernetes API, a service token can be used to access the API.

2. State Synchronization: First sensor is then connected to the updated application. Once the updated application is up and running, the state transfer is executed from the preStop script using an HTTP POST request on the state transfer application deployed on Kubernetes or as a standalone application on the node.
3. Verification: An automatic verification is done using an OPC UA client (if the PLC uses OPC UA for communication) which monitors and compares the values of the old and the updated application for a few cycles.
4. Switch: Once the verification is complete, the updated control application is connected to the actuators, and the connections to the old applications is terminated either by

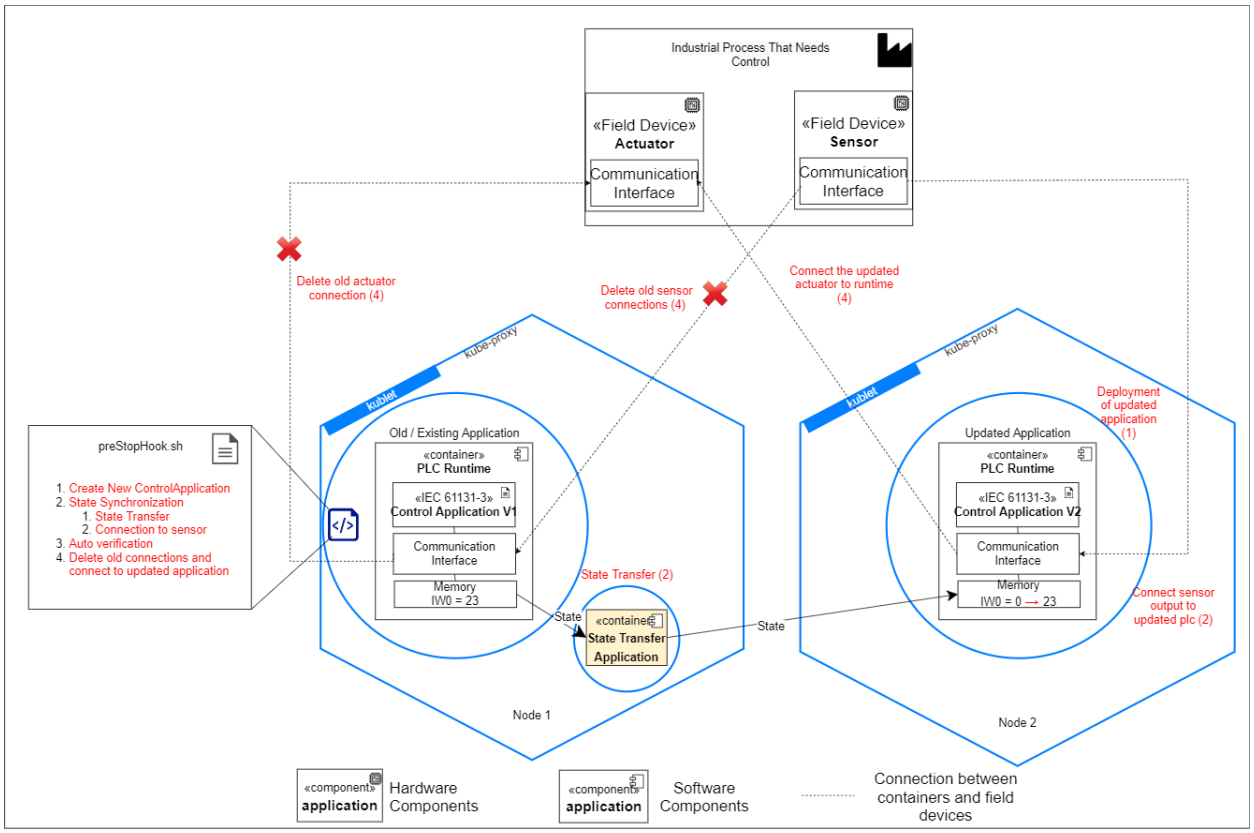

Figure 4.10 : Automatic update of control application in kubernetes using Pre-Stop Hook

forcefully killing the plc application pod or by disconnecting the connections using and HTTP request to the control application. If there is an issue in the verification phase and the updated application is not behaving as expected, skipping the switch step rolling back to the existing pod might be difficult, as the existing pod is already in the terminating state. There is no control of the script to change the state of the pod from `terminating` back to `running` again which could be possible in a custom resource definition.

4.3 Updating Kubernetes nodes

Kubernetes nodes are physical or virtual machines with their dedicated hardware and operating systems. Kubernetes node upgrade includes the update in operating system running on the node or in the Kubernetes version in the node. Since the pods run in a Kubernetes node, the process of a Kubernetes node update is complex as before the update all the pods are needed to be shifted to other nodes and then perform an update. The node update process is standard and most of the cloud platforms like google cloud, AWS cloud, Redhat OpenShift, Starling X and many more provide functionality for automatic node upgrades.

4.3.1 Update Strategies for Kubernetes Node Updates

Sandeep Dinesh has presented a solution for updating Kubernetes clusters using rolling updates as well as update using node pools in Google Cloud Platform (Dinesh, 2018). There are two major ways in which all the Kubernetes nodes can be updated in a cluster – Rolling Node Updates and Updates using Node Pools. The basic steps involved in both the approaches are the same – Drain, Update and Repeat but, the order of execution of these steps is different in both steps.

4.3.1.1 Rolling updates

In rolling updates, each node is drained and then updated one by one to the latest version. The steps involved in the rolling update of a Kubernetes node as defined in the Kubernetes documentation (Kubernetes, 2020g) are given below.

1. Drain node
2. Update node

3. Uncordon node
4. Repeat same process until all nodes are updated
5. Rebalance cluster

The rolling node updates approach requires some extra steps since the update is completed in the same cluster, and no new nodes are assigned to the cluster for the update. Figure 4.11 shows how the cluster of 2 Nodes looks while performing a rolling node update. The steps are mentioned in detail in the following chapters. Rolling node update is the most cost-effective solution as no new node is required during the update, but there is a risk of running out of resources as during the entire cluster update the node being updated is unavailable. Also rolling back to previous version is also very complex and time consuming. At the end of step 4 in the figure, there is a need to rebalance the cluster as even after all nodes are updated, the cluster is not balanced, and no pods are

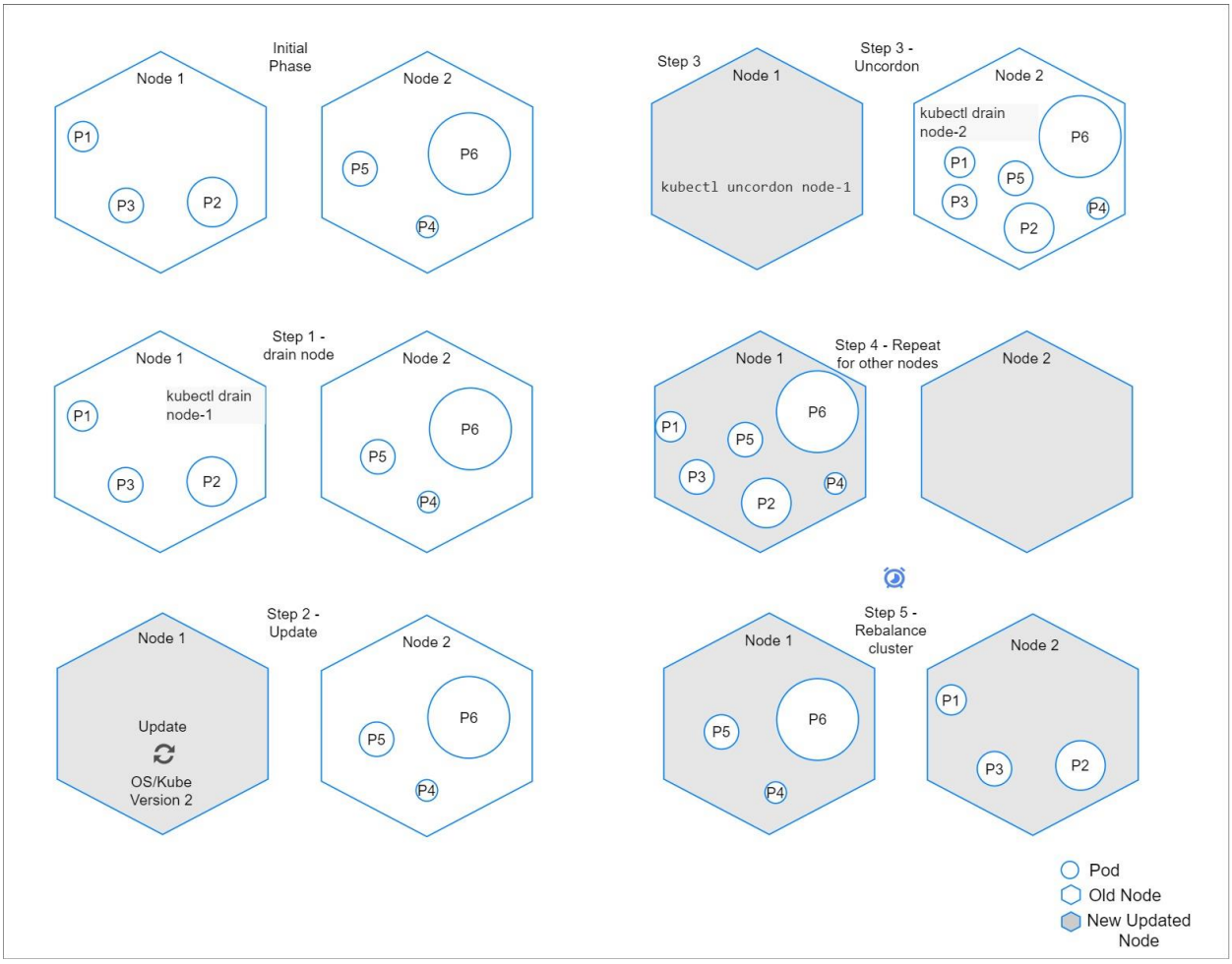

Figure 4.11 : Rolling Node Updates in Kubernetes

scheduled in the node which was last updated. In case if there are many pods and when one node is shutdown, it might happen that there are not enough resources is the cluster to run these pods. When there is a risk of happening that, generally just for the update the cluster is scaled up by adding one more node to the cluster. This reduces the risk of the cluster running out of resources during a node update.

4.3.1.2 Updates Using Node Pools

In this approach instead of updating the nodes one by one, a set (pool) of new nodes is commissioned during the update and then the newly created nodes are updated and the pods running in the old nodes are drained so they are rescheduled in the newly updated ones. The steps are similar to the rolling update process except for the part of commissioning of new nodes. The steps are mentioned below.

1. Commission new set of nodes.
2. Update all the newly commissioned nodes one by one them.
3. Cordon all the old nodes
4. Drain old nodes one by one.
5. Delete old nodes

Figure 4.12 displays the process of updating Kubernetes clusters using node pools as explained by Sandeep Dinesh in Google Cloud Platform (Dinesh, 2018). The first step is commissioning of new nodes which is easily possible in a commercial cloud provider like google cloud platform (GCP). When there is a requirement of commissioning new nodes in a cloud service with local infrastructure setup maintained locally, then the creating od now nodes might take some time. Once the new nodes are updated then the OS or Kubernetes Version of the nodes can be updated easily. After the update is complete, the newly updated nodes are ready for scheduling the existing pods. Before draining the old nodes, the old nodes are marked as cordon – disabled from scheduling any new pods. This does not affect the execution of the existing pods. When the drain command is executed, then the pods running in the old node are scheduled on the new nodes. This process is repeated until all the nodes are updated, and then the old nodes are shut down and decommissioned.

This is a costly process and during the update extra nodes are required during the update. This process less prone to risk of running out of resources during an update as extra nodes are used during the transfer. The node pool update is suitable for platforms which use a professional cloud service for Kubernetes like google cloud platform or amazon web services or any other. The rollback to the older version is faster and less complex in this process.

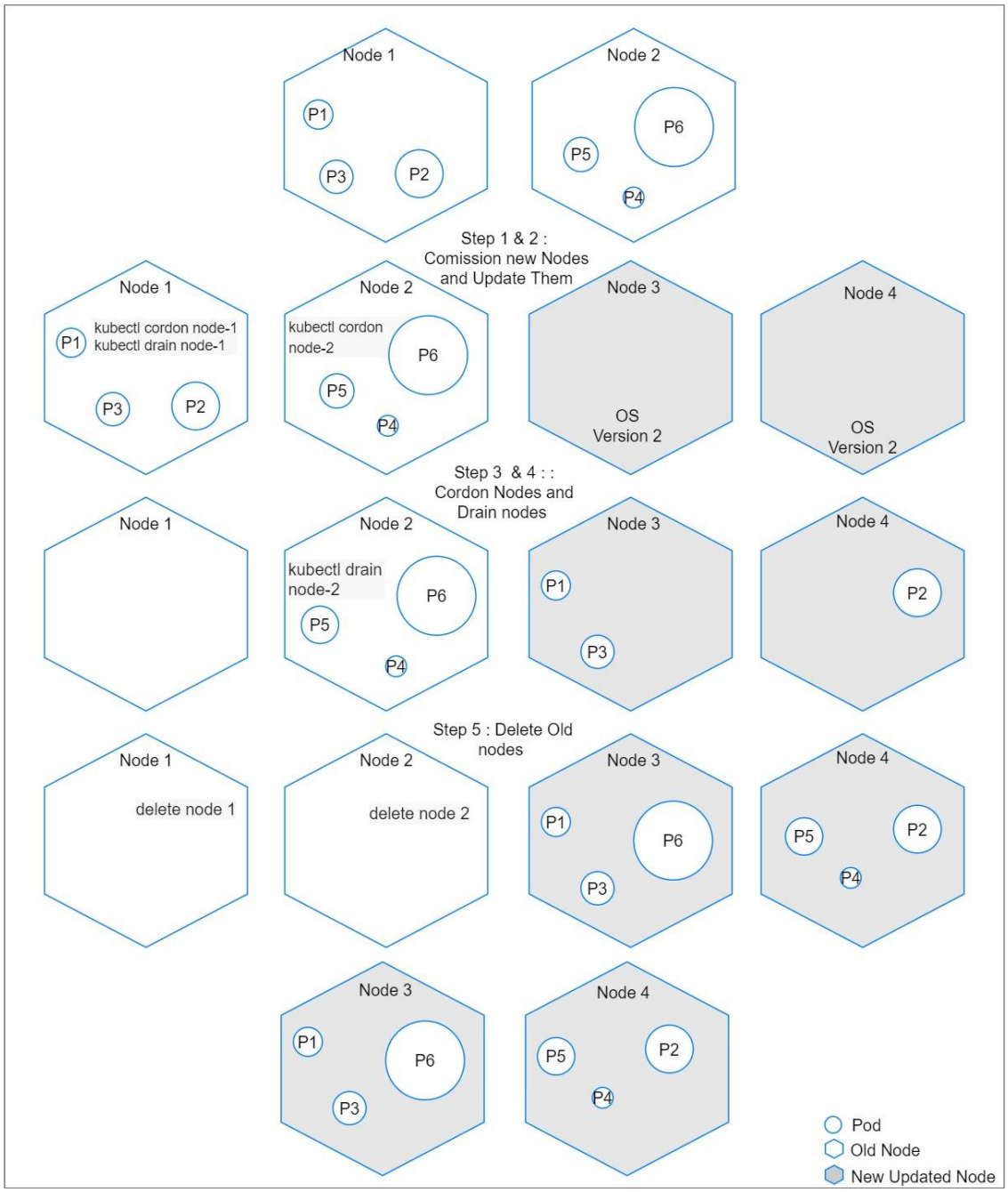

Figure 4.12 : Kubernetes node update using Node Pools

4.3.2 Steps Involved in Kubernetes Node Update

4.3.2.1 Drain Node

This step gracefully shuts down all the pods in a node and then the Kubernetes API (Kube API) creates these pods in other nodes. A Kubernetes drain command also internally cordons the node – prevents it from scheduling any new pods on the node. This will avoid scheduling of new pods on the node during the update.

For control application there is an extra requirement of restarting these pods without any downtime. In the earlier use cases, the concept of load-evaluate-go was used for updating the control applications running these pods. The evaluate part was required as there was an update in the control application or a field device and so a field engineer is supposed to observe the control application before making the switch from the old device to the new device. During a Kubernetes drain there is no change in the control application, but the same application is restarted in another node. Therefore, this process can be done automatically without any verification from the field engineer. Also, there are multiple pods running on a node and manually doing the process of restarting the pods on another node is time consuming and error prone.

Section 4.2.3.2 explains the concept of automatically updating the pods to a new version. The concept of automatic updates can be used for automatically restarting the pods in another node without any disruptions. The same steps – Initialization, State Synchronization, Verify, Switch are used for automatic restart of pods. A major difference is in the Initialization step in which instead of deploying a new version in the updated control application, the same application is redeployed as there is no change in the control application. Also, the verify step is optional as there are no direct changes in the control application. The same two approaches are available for automatic restart of control applications.

1. Using Custom Resource Definition (CRD)
2. Using a pre-stop script in the pods

The cluster update was implemented using a preStop hook in this thesis.

4.3.2.2 Update Node

Node updates include updating the Operating System of the node and/or updating the Kubernetes version installed in the node. During an operating system update, the active running processes might get affected so there should not be any critical applications running on the node. In a Kubernetes cluster there are two types of node – master node (control plane) and worker nodes. The worker nodes run all the pods and the master node runs the Kubernetes critical applications like Kubernetes API Server and etcd. Due to this reason during the update of a master node, the Kubernetes API and etcd are unavailable. This means that all the activities that handled by Kubernetes cannot be done during the update which includes updating mechanism, failover mechanism of pods. The services are back again when the update is complete. To solve this issue, it is recommended to setup a cluster having at least 2 master nodes which are accessed by a load balancer, so that when one of the master nodes is unavailable, other master node is available and so is the Kubernetes API

4.3.2.3 Un-cordon Node

Un-cordon node makes the node available for scheduling pods again after the update is complete. In the case of rolling node updates, the same nodes are updated and reused so it is necessary to make the nodes available for rescheduling pods. When an update during node pools is done, then a new set of pods is used which are already updated. The old nodes are decommissioned after the update is complete.

4.3.2.4 Rebalance Cluster

During a rolling node update, when the final node is updated and un-cordoned, it is available for new pods to be scheduled. Even after making the node available for scheduling, no pods are scheduled from the existing nodes as they are already running in other nodes. Only when a new pod is added it will be scheduled in the last updated node. This makes the cluster unbalanced as the last updated node does not host any pods and the other nodes host all the pods in the cluster. This example is shown in Figure 4.11 in step 4. To solve this problem César Tron-Lozai has presented an excellent approach for rebalancing the cluster using chaos engineering (Tron-Lozai, 2018). The de-

scheduler[1] application which will run like cron job periodically to forcefully delete the pods which are not distributed correctly in the cluster. The Kubernetes scheduler can then uniformly schedule the newly created pods across all the nodes. This issue does not happen in the node updates using node pools as all the nodes in a cluster are already available for scheduling at the beginning.

4.4 Overview of Update Scenarios in Industrial Automation Software

Different update use cases are considered during the update of different types industrial automation software, for example, updating industrial control applications or updating industrial analytics applications etc. The solution for automatic or manual software updates was designed after considering these different use cases. Table 4.1 shows an overview of the different update scenarios in industrial automation.

Use Case	Example	State Transfer Required	Docker Image Change	Occurrence (Frequency)	Type of Update Strategy Supported	Type of update (Manual / Automatic)
Updates in Control application	Update in logic / functionality of the control program	Yes	Yes	Development Phase – Highly Frequent (Weekly) Production Phase– Rare (2-3 Years)	Blue-Green Primary-Secondary	Manual / Automatic depending on the type of logic change
	Minor Updates in underlying operating system of the container running Control Applications (Security Updates)	Yes	Yes	Frequent (2-3 months)	Blue-Green or Primary-Secondary	Automatic

[1]Descheduler : https://github.com/kubernetes-sigs/descheduler, Accessed 18.08.2020.

	Update/Change in the OPCUA Version of the PLC runtime (Security Updates)	Yes	Yes	Frequent (2-3 months)	Blue-Green or Primary-Secondary	Automatic
	Change in the environment variables (OPCUA URL's for field devices) or the parameters provided via command line to the control application	Yes	No	Rare (Every Year)	Blue-Green or Primary-Secondary	Automatic
Change in the field device.	Replacement of a field device due to malfunctioning	No	No	Rare (1 – Several years)	Physical device update, no strategy required.	Manual
Update in Kubernetes nodes	Update in Kubernetes Version of the node	Yes	No	Rare (Every year)	Rolling Node Updates or Update using node pools	Manual (Requires automatic restart for control applications)
	Update in host operating system	Yes	No	Rare (1-3 Years)	Rolling Node Updates or Update using node pools	Manual (Requires automatic restart for control applications)
Update in other stateful components in the cluster	Update in the time series database used for industrial analytics	No	No	Rare (Every Year)	Blue-Green	Automatic
Change in stateless	Update in HMI component of	No	No	Development Phase –Frequent	Rolling updates	Automatic

components of industrial automation software	industrial automation			(Weekly / bi-Weekly) Production Phase– Rare (every year)	or Canary or Blue-Green (supports all strategies)	

Table 4.1 : Overview of Update Scenarios for Industrial Automation Software

4.4.1 Different Update Scenarios for Industrial Control Applications

In this section the different update scenarios for industrial control applications are discussed. Control applications are rarely needed to be updated, but due to the increase in the use of internet-based technologies in PLC's the control applications need more frequent security updates

4.4.1.1 Change in logic of the control program

This use-case is relevant when there is a requirement of changing some logic in the control application. These changes might include

- Adding or removing a new variable to existing application.
- After changing a field device, there might be a requirement to change the logic or adjust the parameters of the controller so that it can function properly with the new device parameters
- Changes made during the initial setup phase.

Most of these changes occur rarely – about once or twice in two years except the third one which might occur more frequently, but only during the initial setup phase of a new machine or plant. Once the control values are at optimum levels then there are very minor changes in the logic which occur rarely. As explained in 3.3 , any change in the control application deployed in a Kubernetes cluster needs a zero-downtime update which involves a state transfer. These "on-the-fly" or zero-downtime updates are required when the production plant is functioning regularly. Production plants undergo regular (e.g. annually) maintenance phases during which the plant is shut down. During a shut down, the zero-downtime updates are not required as the physical process is not running, only the retained variables must be read in the updated application during startup. Since the

control application is shipped with the Docker image, a Docker image change is also required every time the logic of the control application changes. The update strategies that work best for this approach are Blue-Green strategy and Primary-Secondary strategy.

In the experiments carried out in the thesis, the use case of change in number of variables while updating the control application were not considered, but as the logic for that part was not implemented. This logic can be added later when the commercial product is being developed.

4.4.1.2 Updates in underlying operating system of the container running PLC runtime

The control application runs on a PLC runtime inside a Docker image. These Docker images are referenced from the base images different Linux distributions like Debian, CentOS, Ubuntu etc. For example, the OpenPLC runtime Docker image uses Debian as its base image. There are minor updates in the Operating System(OS) released almost every quarter and security updates are added almost 4-5 times a week (Debian, 2020). Not all these updates are critical, but some of the security are critical and the system needs to be up-to-date or the application might be vulnerable to major security threats. So, if not every week or month, the security updates must be installed at least every quarter i.e. every 3 months which is quite frequent.

Since these updates are a part of the Docker image, the Docker needs to be updated every time these security updates are installed. The Blue-Green and the Primary-Secondary update strategies can be used for updating.

4.4.1.3 Updates in the OPC UA Version of the PLC runtime

Every year new features are added to the OPC UA specification. As explained in Section 2.2.3 that there are different implementations of OPC UA available. Once the OPC Foundation makes a new release of the OPC UA specification, then the projects implementing the specification start making updates to the source code. So generally, the implementation is updated after the release in the specification. Apart from these major releases, different small features and security updates are released frequently – for example, every 2-3 months a new update is released by the open62541 project (open62541, 2020).

These updates include:

- Adding a new feature like OPC UA Publish Subscribe (PubSub) to the Specification and implementation.
- Adding security features and authorization services.
- Adding new features related to the data model of the specification, etc.

Since this involves changes in the source code of the OPC UA implementation, it changes the Docker image of the PLC runtime. The update strategies supported are again the Blue-Green update strategy or the Primary – Secondary update strategy or any other strategy which ensures zero-downtime.

4.4.1.4 Change in the environment variables or the parameters provided via command line to the PLC runtime

The environment variables which are provided to the control application are generally the field device Ids and the OPC UA URL's of the field devices. In this use case there is an update or change in an external device like a field device (sensor or actuator) and not in the control application or the PLC runtime itself. Example – Whenever the field device is changed, then the field device id and the OPC UA URL of the field device might change. To connect the control application with the new field device the environment variables for the field device id and the OPC UA URL are needed to be updated. This can be done by changing the environment variable values in the Kubernetes configuration yaml file. The control application needs to be restarted with the updated environment variables. As the field devices are changed rarely, the occurrence of this use case is also rare. Most of these environment variables can be updated in the control application without even the need for restarting the control application, but not all PLC runtimes support this feature.

In this use case a Docker image change is not required, since there are no changes in the control application itself, but in the environment variables provided to the control application. There is still a requirement of disruption free update and therefore state transfer is still required. Blue-Green and Primary-Secondary Update strategy can be used for updating.

4.4.2 Change in the field device.

Field devices include the sensors and actuators in an industrial control environment. Field devices are sturdy and long-lasting devices and are designed to function in the adverse industrial conditions like high temperature or pressure. Generally field devices are changed rarely – almost once or twice in two years. Given below are some use cases where field devices are required to be changed

1. Malfunction or defect in the field device.
2. Update an outdated field device which supports the latest features.

If the field device is changed, there is no direct change in the control application, but it affects the application as the device details and other parameter change when the device is replaced. There is no Docker image change required. The field device can be replaced without the need for shutting down the control application if the field device is at a non-critical position. Once the device is changed, the updated parameters are then sent to the control application by updating the environment variables. This in turn needs a restart of the control application.

4.4.3 Installing Updates to the Kubernetes nodes

Kubernetes Node Updates are primarily of two types – Host Operating System Update and Updating Kubernetes Version. Since both the updates require a node drain i.e. deletion of all the pods in the node, the steps for completing this update are almost same.

4.4.3.1 Operating System Update

This is done very rarely, as all the applications run in Docker containers which have their own operating systems, so it rarely affects the applications. The OS updates are carried out only in case of a high security vulnerability or a major change in the updated version. The Kubernetes node OS update occur once or twice in two years. OS updates also include updates to a real-time patch which is applied to the OS for making it a real time system. Generally, applications having strict cycle times like industrial control applications are run in real-time systems to avoid any unwanted latencies in the processing of the application. For example, the PREMPT RT project provides a real-time patch to apply on existing Linux operating systems (The Linux Foundation, 2016). The PREEMPT_RT patch is updated almost every 2-3 years, so the OS update also occurs rarely. The OS

update is relevant for the organizations which use their own hardware for Kubernetes rather than using a commercial cloud provider as commissioning new nodes with the latest software installed is easy in such commercial cloud platforms.

4.4.3.2 Update Kubernetes Version

Kubernetes version upgrade includes updates to the Kubernetes API, Kubelet and other components of Kubernetes that run on a node. Kubernetes updates are different for master nodes and worker nodes. During the update of a master node, the Kubernetes API is unavailable. The pods continue to run as expected, but since the master node is being updated, tasks like rescheduling of a pod on failure, updating or deploying new applications not possible during the update. Most of the commercial cloud providers like AWS and Microsoft Azure provide functionalities for automatic Kubernetes version updates, but they don't cover the industrial control application scenario, so the update needs to be applied manually. Kubernetes receives regular version updates almost 2-3 times every year. The minor version (1.X) is released every 3-4 months. Generally, there is support provided by Kubernetes for up to three minor versions simultaneously. This means if version 1.3 is released, Kubernetes stops the support for version 1.0, versions 1.1 and 1.2 are still being supported (Kubernetes, 2019). So, there is a need to cluster updated at least every year. So, there is a need to cluster updated at least every year.

4.4.4 Update in stateful components other than control applications

Apart from control applications there are other stateful applications in the industrial automation software. These are mostly non-critical applications from the industry operation point of view. This means that even if these applications don't function for small window of time, it does not affect the working of the machinery within the industry. For example, a time-series database used for industrial analytics comes under this category. Generally, stateful applications are deployed in stateful sets in Kubernetes along with persistent volumes. Also, in most cases the database providers provide configuration files for deploying and updating the database in a Kubernetes cluster.

4.4.5 Update in stateless components in industrial automation software

Stateless components include all the web applications and Human Machine Interfaces in industrial automation. For example, during the update of a control application, if the state

transfer and verification is done manually, then the application has a user interface which internally completes the state transfer and displays the variables. So, the state transfer application is an example of a stateless component of an industrial automation software. Depending upon the criticality of the application, and the use case of the application, different update strategies can be used for updating such applications. Generally, the default rolling update strategy can be used for small applications like the state transfer application. Complex strategies like canary updates can be used for applications like an industrial analytics dashboard.

5 Implementation

The previous chapter discusses about the different concepts in updating industrial automation software, especially control applications and in this chapter the specific implementation details of a few of these concepts are discussed. Most of the update solutions discussed in the previous chapter were generic and could be implemented using any container orchestration system, any mode of communication between sensors and control applications and any PLC runtime software. For the implementation, Kubernetes is used as a container orchestration system, OPC UA client/server is used as the mode of communication, and OpenPLC is used as a PLC runtime for running IEC 61131-3 control applications.

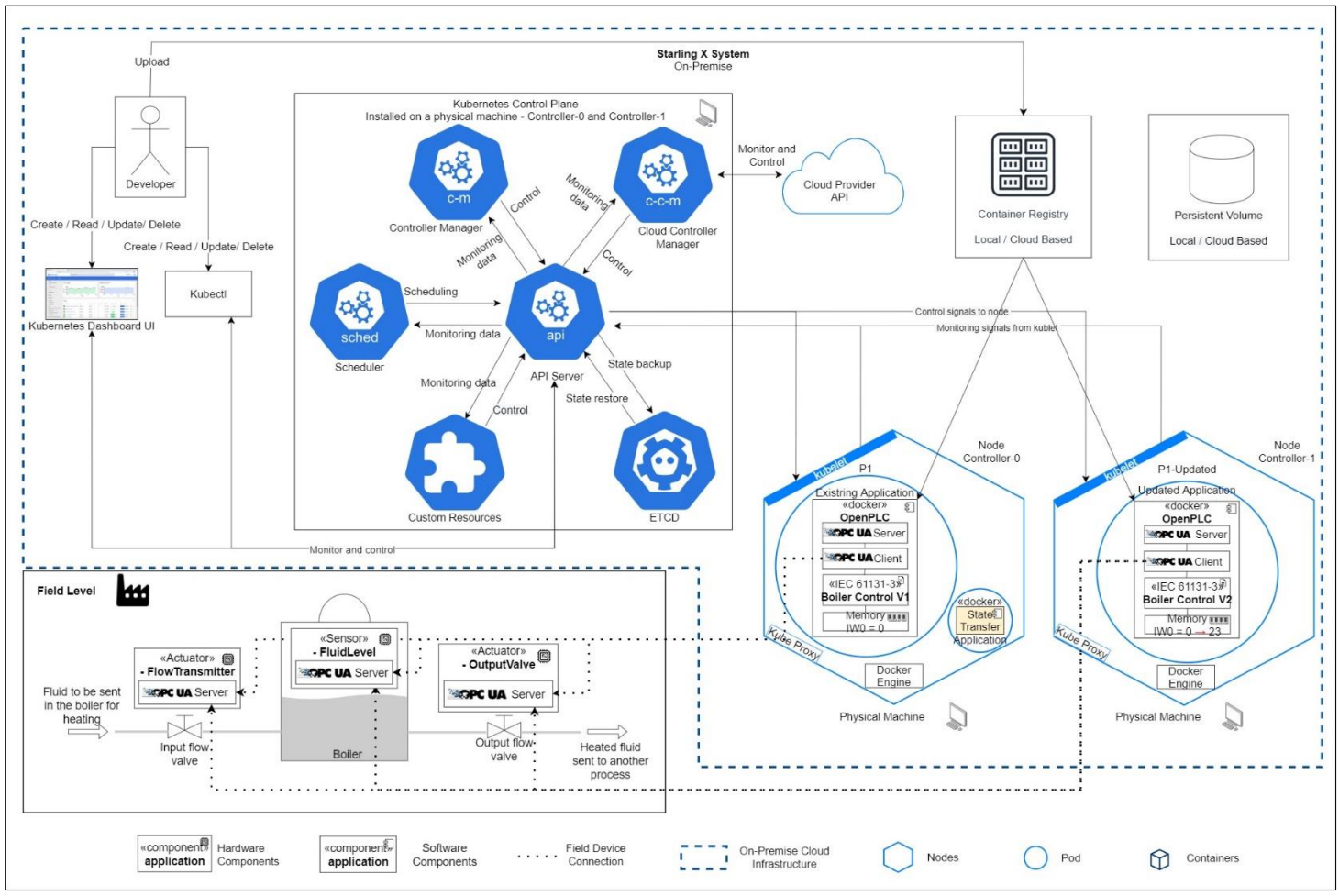

Figure 5.1 : Overview of Implementation of Industrial Automation Software in StarlingX

5.1 Overview

Figure 5.1 shows the overview of the implementation. The cloud infrastructure used for hosting the automation software was StarlingX which internally uses Kubernetes for container orchestration. StarlingX is a cloud infrastructure software-based for the edge. In this case, the edge is a specially secured area on premises in proximity of the field level where all the field devices and the industrial equipment are present. The edge serves as a gateway to plant-specific cloud computing services in public data centers.

The entire infrastructure was hosted on two physical nodes. The OpenPLC runtime was used as the PLC runtime for testing the execution of IEC 61131-3 control applications in Kubernetes. The source code of OpenPLC runtime was modified to add the support for OPC UA client/server. OPC UA was used for communication with the field devices. The field devices were simulated by an OPC UA server acting as an input-output simulator application running on the same cluster. The next section discusses how the different components were deployed on the Kubernetes cluster.

5.2 Components in the Cluster

The different components that were deployed on the cluster as a part of the implementation are 1. Cloud Infrastructure, 2. Programmable Logic Controller and Control Application 3. Field Device Simulator and 4. State Transfer Application

5.2.1 Cloud Infrastructure Software Used – Starling X

The cloud infrastructure used for the deployment of the control applications is StarlingX. StarlingX is based on the concept of a distributed control plane (Cohen et al., 2020) which allows most of the computing at the edge rather than on cloud. The main reason for hosting the cloud infrastructure in the edge is that it reduces the latency for processing the control actions on the industrial components as the field level and control level components have more strict real-time requirements. StarlingX internally uses different opensource components like OpenStack[1] – Cloud Infrastructure for Virtual Machines and Containers, Kubernetes – Container orchestration system, Ceph[2] – Distributed storage solution for cloud infrastructures and Linux. The key features of StarlingX are ultra-low

[1] OpenStack: https://www.openstack.org/, Accessed 15.09.2020
[2] Ceph: https://ceph.io/ceph-storage/, Accessed 15.09.2020

latency, reliability, and security (StarlingX, 2020). Most of the experiments were done with control applications running in containers, and no experiments were performed in the edge platform with physical devices as the field device operation was simulated using software containers so the ultra-low latency features of StarlingX were not tested in the implementation.

The hardware infrastructure consisted of two servers with Intel Xenon CPU E5-2640 v3 with a frequency of 2.60 GHz with 2x8 Cores, 16 Threads and an L3 Cache of 20 Mbyte. The total RAM for each server is 64 GB. CentOS 7.6.1810 was installed on the two servers on top of which StarlingX version 3.0 was installed. Kubernetes version 1.16.2 is bundled with the StarlingX installation. Both the nodes – Controller-0 and Controller-1

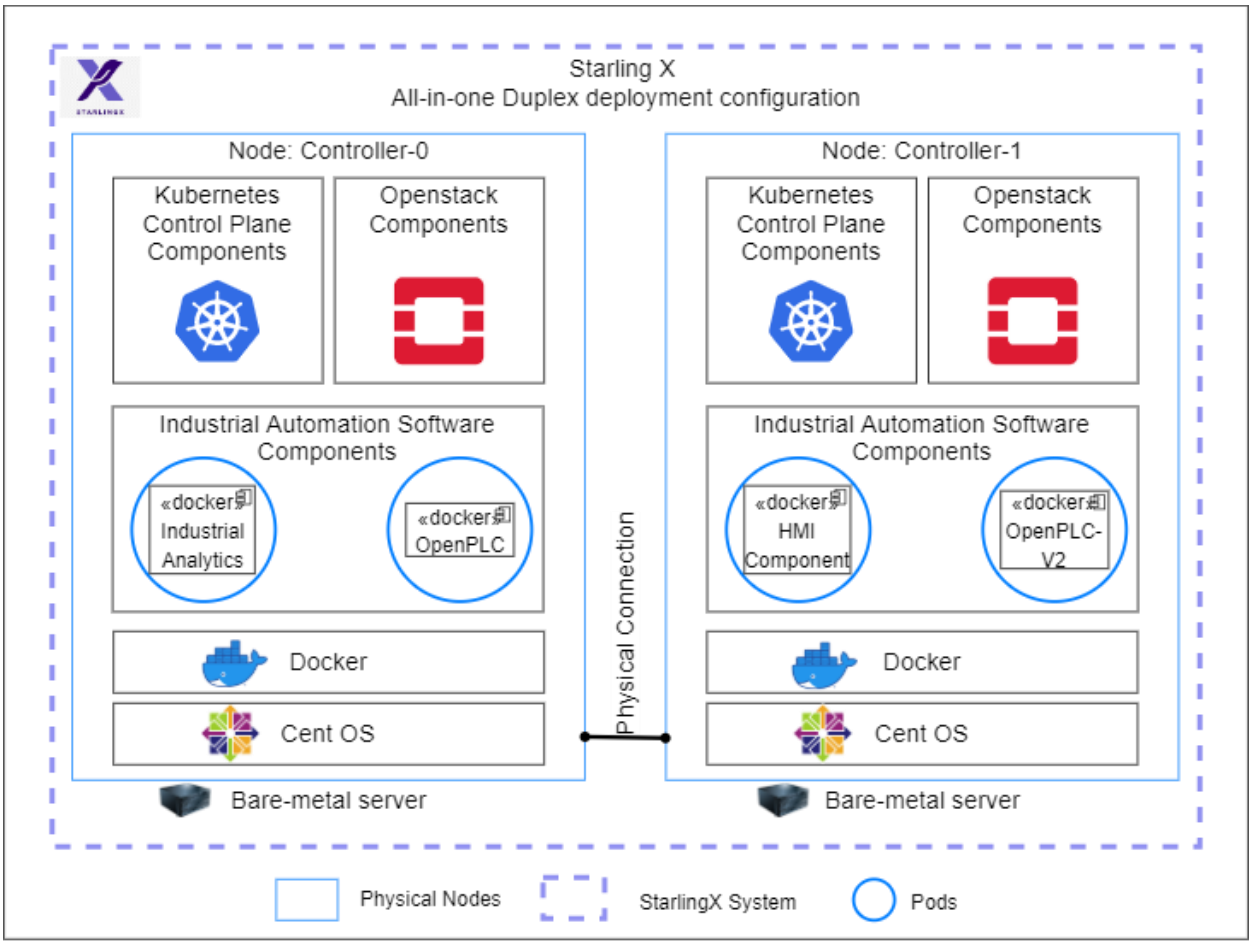

Figure 5.2 : Cloud infrastructure setup with StarlingX

were master nodes of Kubernetes and hosted the control plane components as well as other application components like control applications. Figure 5.2 shows the all in one

duplex infrastructure setup of the StarlingX cluster with the two machines Controller-0 and Controller-1. Each physical node acts as a Kubernetes master and hosts the Kubernetes control plane components. These nodes also host the OpenStack components. Generally, in a cloud infrastructure setup, the master nodes only host the Kubernetes control plane components and the compute nodes host the applications (in this case industrial automation software), but in this case the applications are also hosted on the same nodes. In future, the plan is to add worker or compute nodes to run the control applications and other industrial automation software. The system uses a local Docker registry and an internal on-cloud private container registry.

5.2.2 Programmable Logic Controller and Control Application

PLC's are generally hardware components which run on dedicated hardware designed to sustain in industrial environments. To execute control applications in a Kubernetes cluster there is a need to use a software-based PLC which can run in software containers. OpenPLC runtime was used as the PLC software for implementing and testing the concepts explained in Chapter 4. OpenPLC runtime is a part of the OpenPLC project. The OpenPLC project is an opensource PLC project which consists of three parts: 1. OpenPLC Runtime – The core PLC engine which executes the IEC 61131-3 control applications, 2. OpenPLC Editor – The software which is used for writing PLC Programs for the OpenPLC runtime according to IEC 61131-3 standard. 3. HMI Builder – Scada BR is a SCADA system which can be used to create interactive screens (HMI) for several automation projects (Alves).

5.2.2.1 OpenPLC Runtime

OpenPLC runtime is a software-based PLC runtime which can be installed on embedded platforms like Raspberry Pi, Free Wave Zumlink Radio, PiXtend and others. The runtime can also be installed on software environments like Windows and Linux and as a Docker container in any of these environments. The runtime uses Modbus TCP protocol for communication with external devices. The source code of the OpenPLC runtime was modified to support the use of OPC UA based communication client server communication.

Architecture of OpenPLC Runtime

OpenPLC runtime consists components of a which are similar to a typical PLC explained in Section 2.2.1.1 and few additional components. These components include – Memory, CPU (In this case the CPU of the Host Machine), Communication Interface, Web User Interface for uploading control applications as well as operating and monitoring the PLC runtime. Figure 5.3 shows the architecture of OpenPLC Runtime. OpenPLC consist of more components, but in this figure the relevant ones are shown.

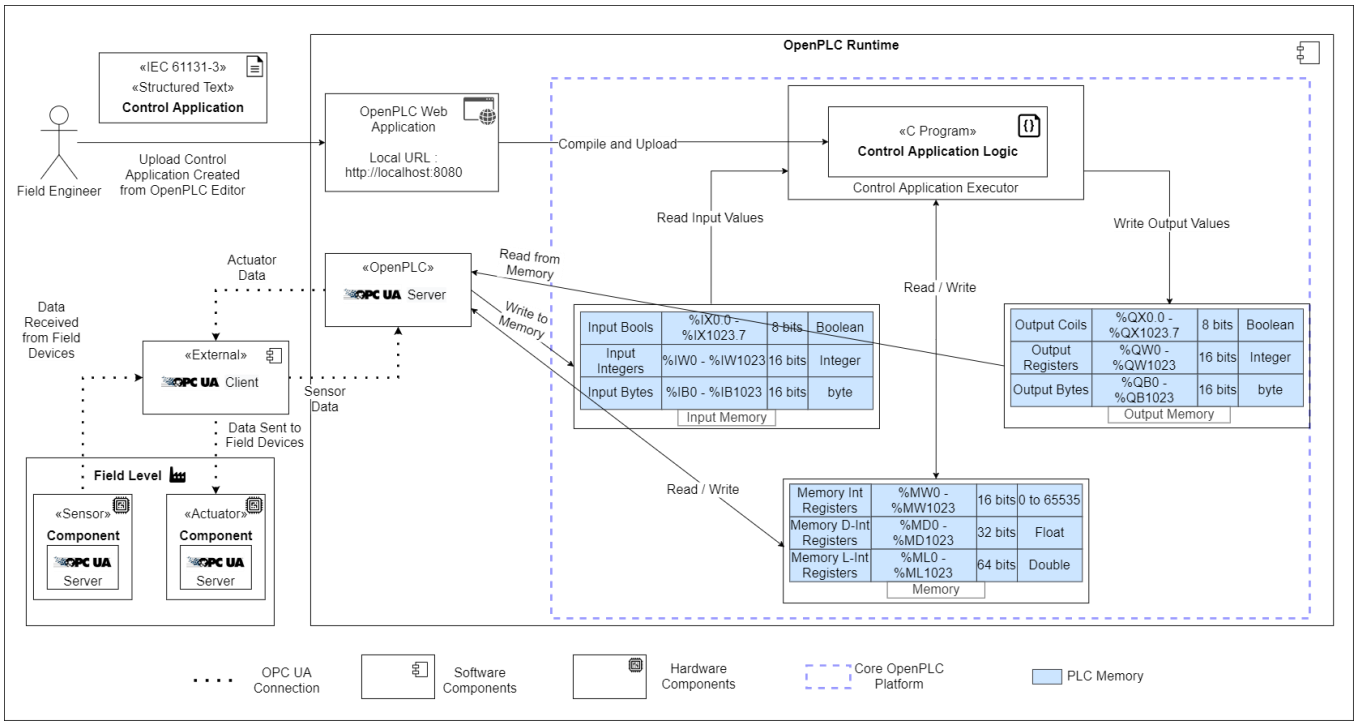

Figure 5.3 : OpenPLC Runtime Architecture

- Memory

As shown in the figure the input and output memory each contain arrays of Integer, Boolean and Byte types. In this example the buffer size of the array is 1024. The sensor data from field devices which is the input for the PLC is stored in the input arrays while the data to be sent to actuator which is the output of PLC is stored in the output arrays. Along with the input-output memory, OpenPLC also contains a separate memory for storing variables other than I-O variables. The memory block contains arrays of integer, double and long variables each of size equal to buffer size.

- Web Based User Interface

OpenPLC runtime also contains a User Interface for uploading monitoring and controlling the execution of control applications in the runtime. The web-based UI runs on a Python Server and allows the users to upload IEC 61131-3-based control application, start and stop the control application execution, modify Modbus, OPC UA ports and monitor the variables.

- OPC UA Server and Client

The OPC UA server is linked to the memory of the runtime and maintains a separate address space corresponding to the entire memory of the runtime. A C-based opensource library – open62541 (open62541, 2020) was used for implementing the OPC UA part in the OpenPLC. A separate thread continuously (every 50 ms) polls the memory and compares it with the OPC UA Server address space for detecting 1. Changes in the variable values of the OpenPLC Memory updated by the control application executor and 2. Changes in the OPC UA address space updated by the field devices. If there is a change in the OpenPLC memory, the corresponding variable is updated in the OPC UA server address space and if there is any change in the OPC UA Server address space, the corresponding variable is updated in the OpenPLC memory. The external OPC UA client is connected to the OPC UA server of the OpenPLC runtime. The purpose of the OPC UA client is 1. Read PLC input data from the field devices (Sensors) and write to the OPC UA server in OpenPLC, 2. Read PLC output data from the OPC UA server in OpenPLC and write to the field devices (Actuators). As a part of the concept (Figure 5.1), the external OPC UA client was supposed to be a part of OpenPLC runtime, but due to ease of implementation, the client was kept external to the OpenPLC runtime.

- Control Application Executor

OpenPLC editor allows users to create IEC 61131 Complainant control application and these applications can be uploaded to the web UI of the OpenPLC runtime. When the control application is uploaded to the runtime, the runtime converts this IEC 61131-3-based control application to a C program, and then compiles the entire execution code

as a single executable file. The control application executes this compiled code in loop to perform the control actions.

5.2.2.2 Control Application Used

As explained in previous section, the OpenPLC editor allows users to create a control application, which can be designed in any of the five PLC programming languages defined by IEC 61131 Standard. Figure 5.4 shows an example of a Functional Block Diagram (FBD) based control application for maintaining the level of a fluid in an industrial boiler. The FBD can be created in the OpenPLC editor and then exported as a Structured Text (.st) file. This file is then uploaded to the web interface of OpenPLC runtime. The level sensor value (`FluidLevel`) from the boiler is sent to the control application as input, and the desired level (`LevelToMaintain`) is also provided as input by the user. The less than (LT) block compares the two values, and if `FluidLevel` is less than `LevelToMaintain`, the input flow transmitter valve which allows the fluid to flow inside the boiler is switched on (`FlowTransmitter=True`) and the output flow valve is switched

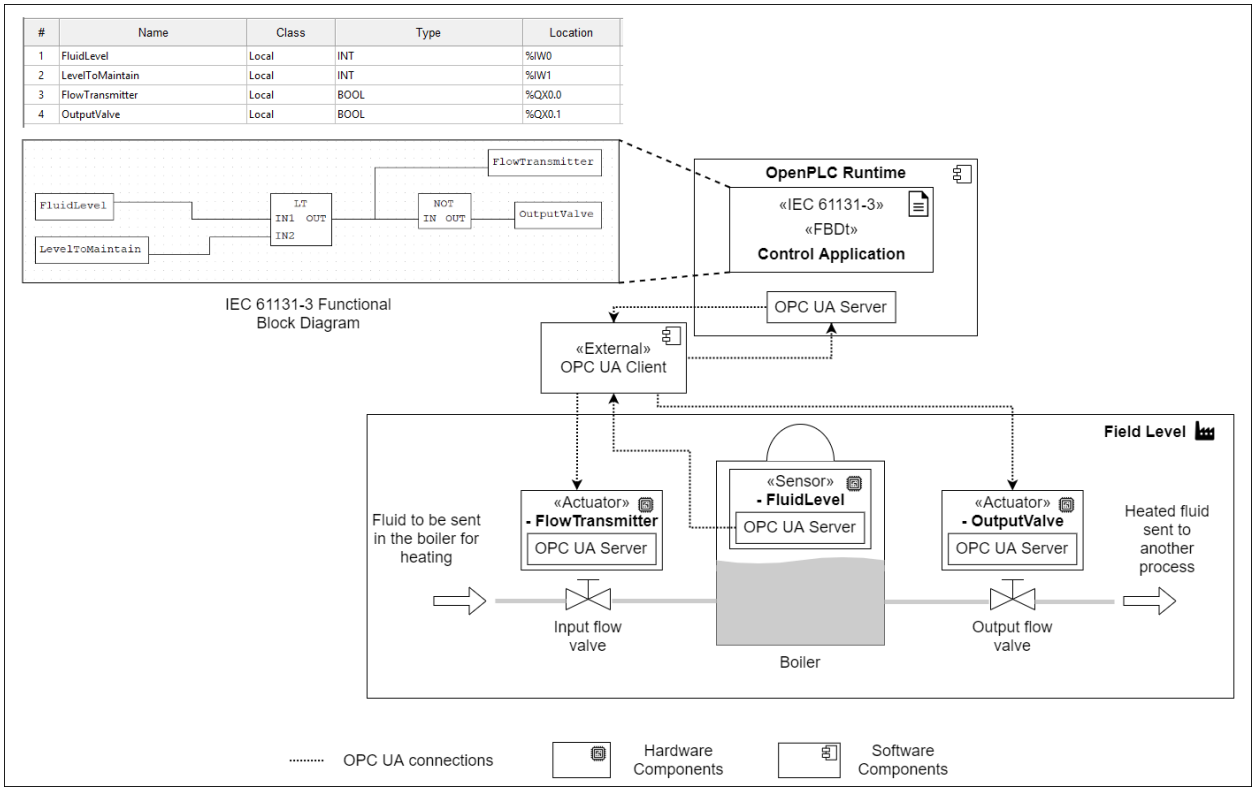

Figure 5.4 : Functional Block Diagram (FBD) based control application for controlling level of Fluid in an Industrial Boiler

off (`OutputValve=False`). If `FluidLevel` is greater than `LevelToMaintain` then the exact opposite is done (`FlowTransmitter=False` and `OutputValve=True`).

5.2.3 Field Device Simulator

For testing if the zero-downtime updates work for OpenPLC, there was a need to simulate the sensor and actuator devices enabled with OPC UA servers. Using physical field devices at such an initial stage was not practical and so an application called IO Simulator was developed to simulate the functioning of an Industrial Boiler. The IO-Simulator application consists of an OPC UA server which contains three variables – `FlowTransmitter` (Bool), `FluidLevel` (integer) and `OutputValve` (Bool). An automatic counter which would randomly increment or decrement the values of the `FluidLevel` variable every second withing a specific range. This would create a simulation of an industrial boiler environment with the fluid level steadily fluctuating.

5.2.4 State Transfer Application

The state transfer application is one of the key aspects of updating industrial control applications. Initially state transfer application was designed as a standalone executable application which can be executed from command line directly from the host machine. Later, the state transfer application was modified and exposed by a python-based web server and a user interface so that the field engineer can initiate the state transfer process manually using the UI or automatically through an automatic update script or process. Underneath the state transfer application was still an executable application, but instead of executing from a command line, the UI or the backend will be used to execute this application by receiving parameters from the users. The main purpose of the state transfer application is to get the existing state of the variables which are marked as RETAIN (the variables which need a state transfer during an update) from the source control application and send it to the target application. To implement the state transfer between two PLC's, the source PLC must have a functionality to create a snapshot of their existing state of memory and the target PLC must have a functionality to update its variable values with the values received from the snapshot of existing state of the source PLC. So, in order to implement the state transfer between two PLC's, some modifications are required to the existing PLC's.

The state transfer application was implemented using two different approaches: 1. Transfer using Inter Process Communication (IPC) via POSIX memory and 2. State Transfer via network using OPC UA

5.2.4.1 Transfer using Inter Process Communication (IPC) via POSIX memory

As explained in Section 4.2.1.3 state transfer in this approach is done by using Inter Process Communication (IPC) via the POSIX memory in the operating system. Figure 5.5 shows a detailed implementation of the state transfer working. To share the current state of the variables in the PLC memory, the memory is first serialized and archived as a single object. Before serializing and archiving the current state of the memory, the execution of the PLC (control application) is paused to avoid any changes to the current state due to the control application while the state is being archived. Then this archived object is added to the shared POSIX memory using inter process communication. Once this is done, the

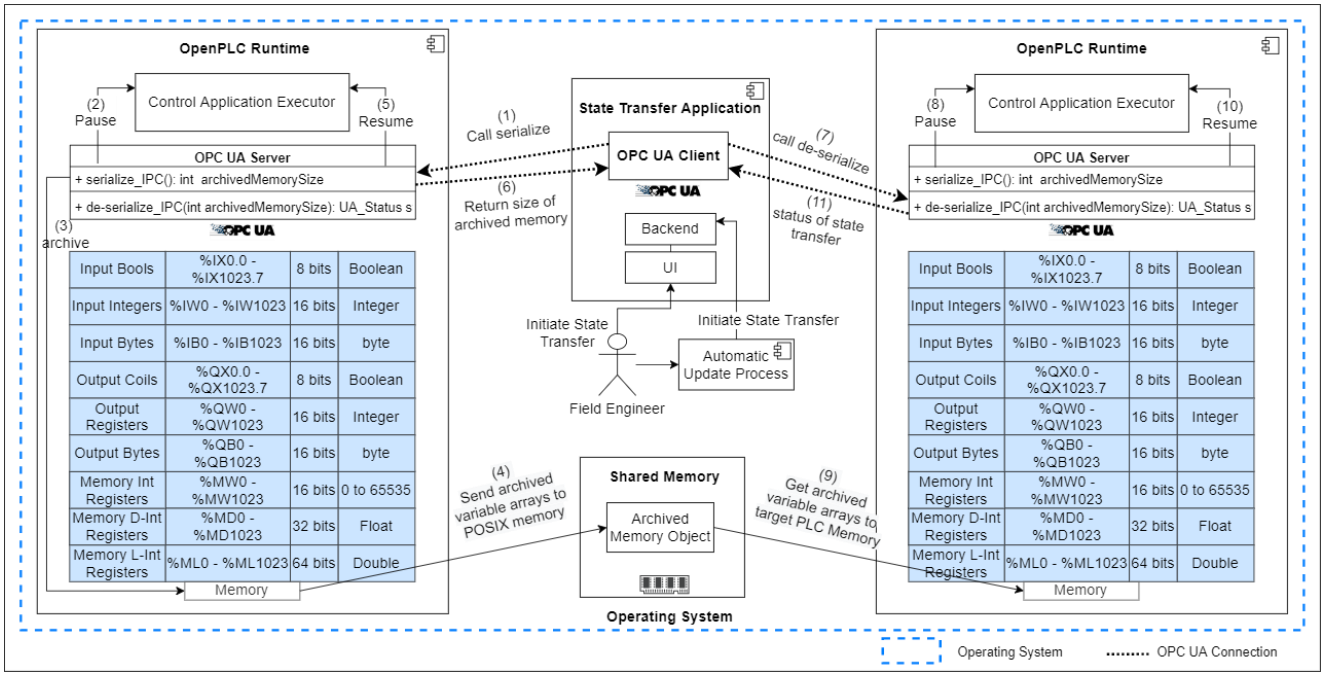

Figure 5.5 : Implementation of State Transfer using Shared POSIX Memory

execution of the PLC (control application) is resumed. The target PLC application then accesses the archived object, and the variable values are updated with the corresponding values in the archived object. The same process pausing the execution of control application and then resuming is done in the target PLC while updating the values of the variables in the target PLC to avoid the control application making any changes to the state of the memory when the variable values are being updated.

To serialize and archive the state of the PLC memory, a serialize function which can be triggered by the state transfer application was added in the source PLC. The serialize function was exposed as an OPC UA method so that the function can be called externally from the OPC UA client in the state transfer application using an OPC UA method call. The serialize function returns the size of the memory which is serialized and archived. Similarly, a de-serialize function is also implemented in the target PLC to access the archived memory, de-serialize it and then update the values of the variables in the target PLC. The de-serialize function in the target PLC is called by the state transfer application when the serialization and archival of the memory at the source PLC is complete. The size of the archived memory received from the serialize method is sent as a parameter to the de-serialize method. The size of the archived memory is required as for every control application, the number of variables that are supposed to be retained are different, and the size is necessary while reading the archived object from the shared memory. The deserialize method sends the status (success or failure) of the state transfer to the state transfer application. The OpenPLC runtime implementation was modified to support the use of serialize and de-serialize OPC UA methods. The shared POSIX memory cannot be used if the source and the target application are on two separate physical nodes or in the case of Kubernetes on two separate pods. Therefore, this approach only works when both the PLC's are running on the same pod as two separate containers.

5.2.4.2 State Transfer via network using OPC UA

As explained in Section 4.2.1.2, state transfer in this approach is done via network using OPC UA. Figure 5.6 shows a detailed implementation of this approach. This approach is similar to the earlier approach and concept, but the key difference is the mode of transport used in both the approaches. Similar to the earlier approach, the current state of the variables in the source PLC memory is first serialized and archived. The serialize function in this case instead of storing the archived state in shared memory of the OS, it sends the archived memory object as a response back to the state transfer application. An OPC UA method call is used for calling the serialize method in the source PLC. The archived memory object is then sent to the target PLC as a parameter in the de-serialize method. The de-serialize method, pauses the execution of the control application, de-serializes the received archived memory object and copies the values of the variables to the target

PLC memory and then resumes the execution of the control application. Unlike the state transfer approach using IPC via POSIX memory, this approach does not need the source and the target PLC applications in the same nodes as the transfer happens via the network.

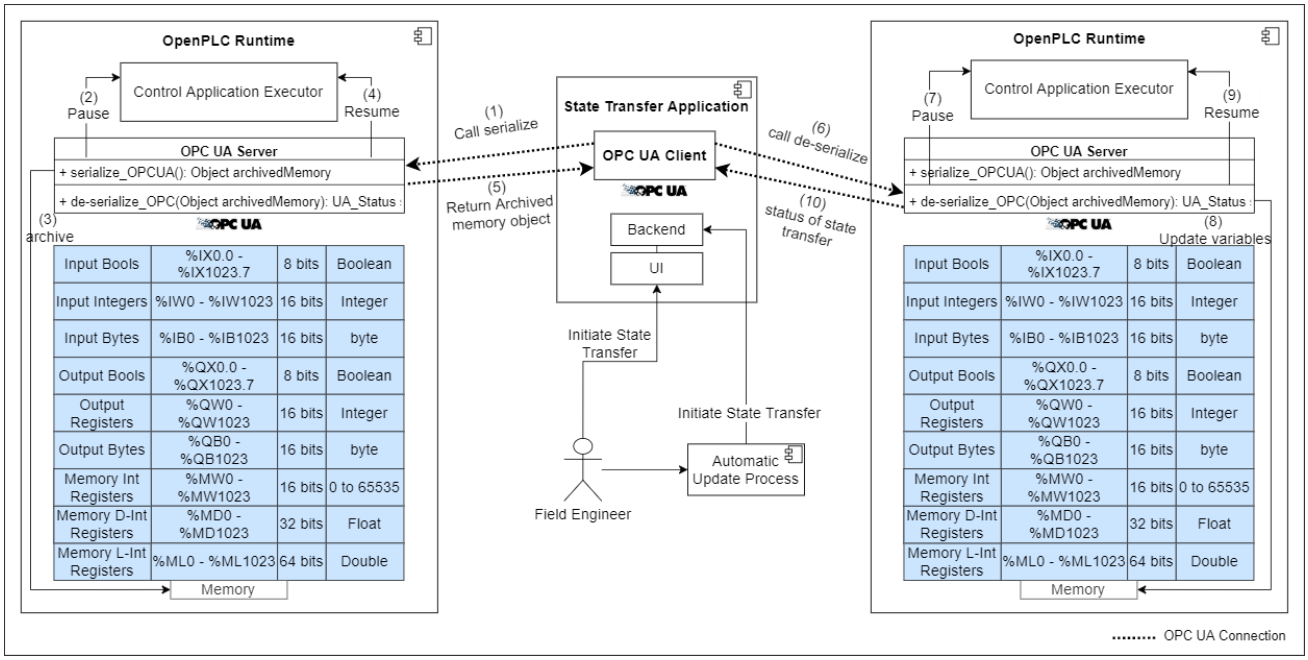

Figure 5.6 : State Transfer using OPC UA

5.3 Running components on Kubernetes

The components discussed above were deployed on Kubernetes as deployment resources. This section discusses about how these components were deployed and how they were connected with each other in the cluster. To execute, the OpenPLC runtime on Kubernetes, its Dockerfile was modified when so that the container starts running, the runtime would automatically start the execution a control application which is already present in the container. The control application name was provided in the environment variables. The OPC UA client shown in Figure 5.3 was deployed as a separate application on Kubernetes for ease of development. Ideally it should be a part of the runtime itself. Along with that the field devices simulator application (io-simulator) was also deployed as a part of the setup. Three different use cases were implemented as a part of the implementation. These include 1. Manual Update of control application, with state transfer

using OPC UA, 2. Manual Update of control application using IPC Via POSIX transfer and 3. Automatic Restart of pods in another node during a node update.

5.3.1 Manual Update of control application, with state transfer using OPC UA

The concept of manual updates was discussed in Section 4.2.3.1 and in this section the implementation of that concept using OpenPLC and the state transfer application in Kubernetes are discussed. Figure 5.7 shows the implementation of the manual update use-case. All the resources in the figure are running in pods and deployed using the deployments (explained in detail in Section 2.1.2.2) workload in Kubernetes. The applications which are accessed by an internal applications withing the cluster or external applications outside the cluster, must be exposed using a service (explained in detail in Section 2.1.2.2) in Kubernetes. Therefore, the applications following applications were exposed by services – 1. OpenPLC v1 to access web interface running at port 8080 and OPC UA server at port 4840, 2. OpenPLC v2 to access web interface running at port 8080 and OPC UA server at port 4840, 3. IO-Simulator to access OPC UA server running on port 8080 and 4. State Transfer Application to access the web UI running at port 8080.

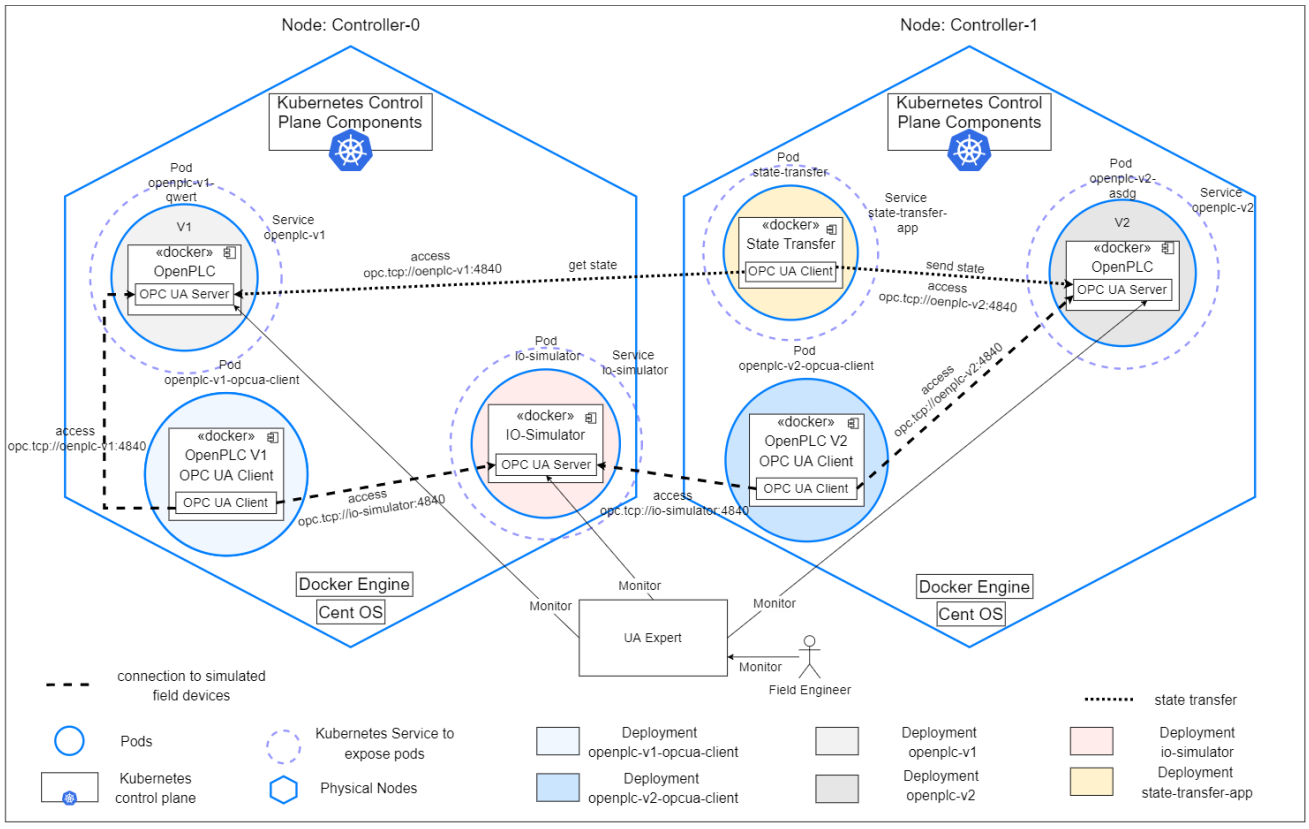

Figure 5.7 : Manual Update of control application (State Transfer using OPC UA)

By exposing these applications using a service, the other applications can access the applications with their service name and port. For example, the URL opc.tcp:/io-simulator:4840 is used to access the OPC UA server in io-simulator. The boiler control application discussed in Section 5.2.2.2 is used as the existing control application, and for the updated control application, the Less-Than block is replaced by Less-Than-Equal-To block in the same application. Once the updated ICE 61131-3 control application is created in the OpenPLC editor, the exported structured file (.st file) is added to the OpenPLC runtime, and the path to build the updated file is provided in the environment variables. The Docker image for version 2 is created with a tag of version 2 and pushed to container registry. This can alternatively be done by uploading the updated control application to the newly created deployment. To perform a manual update, as explained in Section 4.2.3.1 the first step is initialization. In this step a new deployment is created using the updated Docker image which contains updated version of the control application. The newly created deployment starts up a new pod and a container in the cluster. In this example, the newly initialized pod for v2 is in a different node, but it can be in the same node as well. Since the state transfer happens via network, it does not matter where the target pod is initialized. After initialization is complete, the next step is State Synchronization. Once the updated application is running, then the io-simulators connected to the updated control application using the external OPC UA client (OpenPLC v2 OPC UA Client). The state transfer application is then accessed using the web UI or by running the executable state transfer application via command-line from outside the cluster. The command line approach was used in this case, as there were a lot of experiments to be done for different data sizes and it was easier using the command-line version. The next step – Verification was done using the UA Expert which was running outside the cluster on the local machine. Since the control applications and io-simulator were exposed using services, they can directly be accessed from outside the cluster using IP addresses. The variables in existing and the updated applications were monitored in UA Expert to verify that both control applications are running in synchronization. In the next step – Switch, If the updated application was behaving as expected, the existing application (OpenPLC v1) was disconnected from the IO-simulator by deleting the deployment of OpenPLC v1 OPC UA Client from kubectl or Kubernetes dashboard so

only the updated application (OpenPLC v2) would send signals to the io-simulator. In future, the plan is to integrate the verification interface and the switch option in the state transfer application's user-interface. So as soon as the user initiates the state transfer, in the next screen the user can monitor the existing and the updated application variables in at the same time side by side. If the updated variables are being updated as expected, then the updated application is working fine, and the user can make the switch directly from the state transfer UI.

5.3.2 Manual Update of control application using IPC Via POSIX transfer

In this approach, the source and the target OpenPLC runtime are running in same pods. In Kubernetes, the shared POSIX memory is accessible by containers which are in the same pod (Redhat, 2020a). So, to implement the solution of state transfer using IPC via POSIX memory, the prerequisite was to have both the containers running on the same pod. Along with shared memory, containers running in the same pod also share the local network namespace (Unofficial Kubernetes, 2020). Therefore, if two OpenPLC runtimes

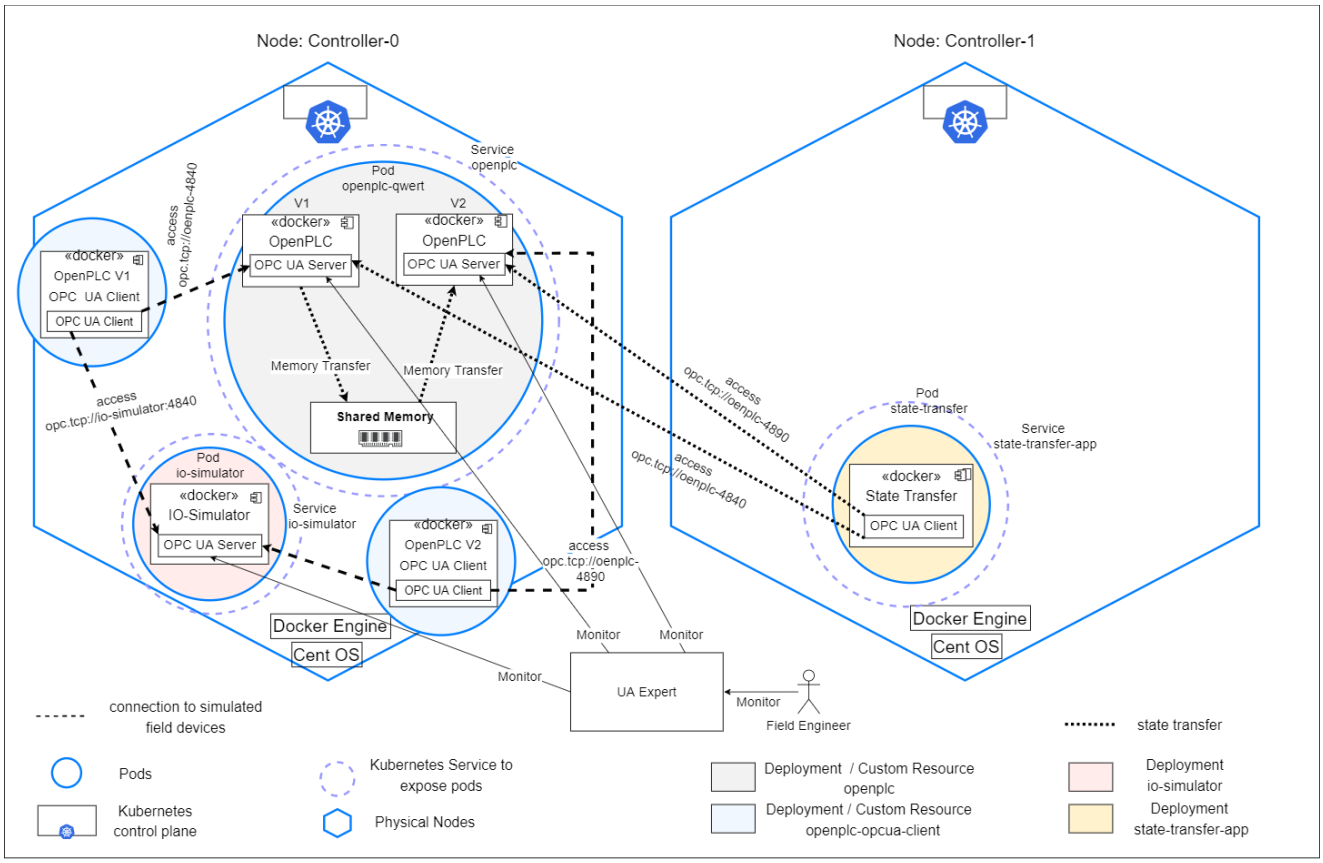

Figure 5.8 : Manual Update of control application (State Transfer using IPC via POSIX)

are running on the same pod, the internal OPC UA and webserver ports must be changed for one of the runtimes, otherwise it will cause a port conflict. Initially the same version of OpenPLC and control application is deployed in the cluster as a deployment resource with two containers.

Figure 5.8 shows the overview of the Kubernetes cluster during a manual update of control application running on OpenPLC runtime. The deployment yaml file for the OpenPLC consisted two containers, the first container was setup with OPC UA Server on the port 4840 and the second one was initialized with OPC UA server on port 4890. Even if the OpenPLC runtimes were running on same pods, they had different ports to access them. Two separate OPC UA clients which connect the io-simulator and the OPC UA server in the OpenPLC runtimes were also setup, of which only one client (OpenPLC V1 OPC UA client) was running initially and the other one is not running. The OpenPLC deployment was exposed by a service and both the runtimes can be accessed by their service names and port. To perform a manual update in the cluster, as explained in Section 4.2.3.1 the first step is initialization. In this step an updated IEC 61131-3 control application was uploaded to the OpenPLC runtime which is standby (not connected to io-simulator). The next step is State Synchronization, and, in this step, the io-simulator is connected to the OpenPLC running the updated control application (OpenPLC V2) using the OpenPLC V2 OPC UA client. Then the state transfer application is executed in a similar way as explained in the previous section. Once the state synchronization is complete, the next step – verification is done by monitoring the existing and updated variables in UA Expert client. If the verification is complete, and the application is ready to be switched, the final step of Switch is done by deleting the OpenPLC V1 client so only the updated application is connected to the io-simulator.

If there is a change the OpenPLC source code or the OPC UA version of the OpenPLC runtime, the Docker image needs to be changed. In the deployment workload in Kubernetes, when there is a change in one Docker image of a multi-container deployment, the entire deployment is updated and so the entire pod is restarted. Due to the pod restart, the container which is running steadily is also restarted, and it will cause a disruption in the execution of the control application. Only the control application can be

updated by the UI for such a scenario. To solve this issue, CloneSet – A custom resource developed by OpenKruise was used to manage pods. In a CloneSet workload, the containers in a pod can be updated without the need for restarting a pod (OpenKruise, 2020). This concept was useful while updating control applications which are running on the same pod without the need for restarting the pod.

5.3.3 Automatic Restart of pods running control application in another node

In this use case a node update (4.3) scenario is simulated, and during a node drain the automatic restart and state transfer of pods running control application is tested. As explained in Section 4.3.2.1, during the node update, the first step is to do a node drain which deleted all the pods in the specific node. In this implementation a node drain event is simulated, to test the disruption free automatic switch-over of the pod running control application to another node. To simulate a node-drain scenario, node selector option is used. If the existing control application pod is hosted on node – controller-1, the node

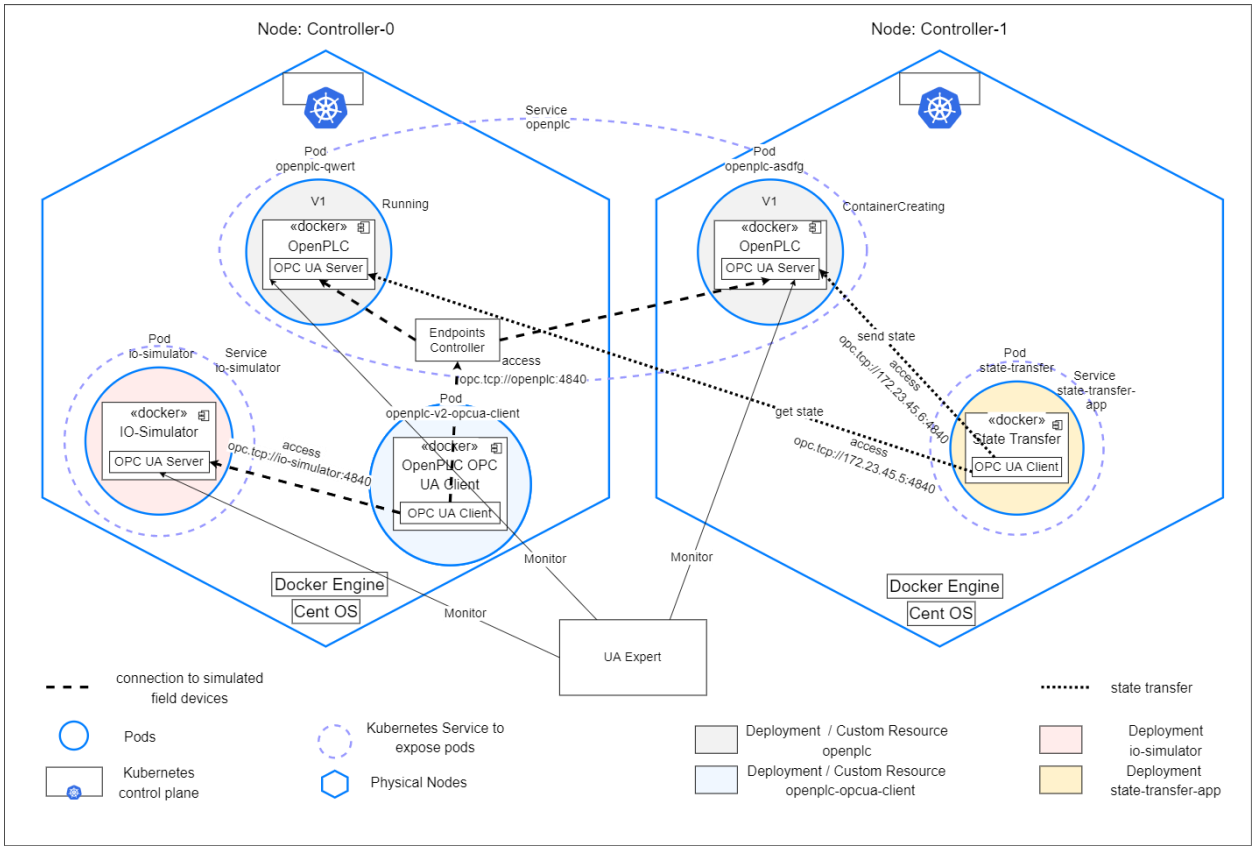

Figure 5.9 : Automatic Restart and state transfer of pod running control application during node update

selector option for the deployment yaml file was set to controller-0. This would gracefully terminate pod in controller-1 and restart the pod in controller-0.

Figure 5.9 shows the automatic restart overview of a pod running control application during a node update. The node drain was simulated by updating the deployment yaml file. When the new deployment file is sent to Kubernetes, Kubernetes initiates the pod termination sequence and restarts the pod in another node. The concept of automatic restart is similar to the concept of automatic update discussed in Section 4.2.3.2, but in this case instead of an update, the same image is deployed on any other node except the existing one. In this case a pre-stop script is added to the container while building the Docker image, and this script does the switch over and calls the state transfer application. For the ease of deployment, the state transfer application was also added to the Docker container, so the script does not need to execute an external HTTP post request to initiate a state transfer. Initially the pod openplc-qwert is running on the cluster, and it is exposed by a service. The OpenPLC OPC UA client is connected to the OPC UA server in the control application running in the given pod. The io-simulator is also connected to the OpenPLC OPC UA client.

The steps that are involved in an automatic restart are similar to the steps discussed in Section 4.2.3.2. The first step is Initialization and, in this step, the new pod is initialized already by Kubernetes, so no action is required. The next step is state synchronization and in this step the pre-stop script which is initiated by the pre stop hook when the existing pod goes into terminating state. The actions executed in the pre stop script are 1. To get the IP address of the source and target OPC UA servers running in OpenPLC runtimes, 2. State Transfer and 3. Terminate the Pod existing pod. The source IP address can be easily obtained using environment variables as it is the IP address of the same machine from which the script is being executed. The pod IP address must be added as an environment variable in the deployment yaml file[1]. To get the IP address of the destination (newly initialized pod) OpenPLC's OPC UA server the kubectl was used within the pod. In-order to use kubectl from within a container, kubectl must be installed in it and a service

[1] Know The Pods Own IP Address : https://stackoverflow.com/questions/30746888/how-to-know-a-pods-own-ip-address-from-inside-a-container-in-the-pod , Accessed 15.09.2020

account must be created for the pod in which the container is running(Stringer, 2020). Once the IP address for the source and target OpenPLC's OPC UA Server is available, the state transfer is initiated. For the implementation, the state transfer application was also installed in the container, so that it can be executed locally as an executable application instead of an HTTP request as discussed in the concept. The OpenPLC OPC UA client is connected to the OPC UA server in the existing application, and the switchover of the connections is done automatically by the service workload (endpoints controller), and it is not in the control of the pre-stop script. Once the state synchronization is done, the verification step is skipped in this case as there is no change in the updated and existing control application. The final Switch is done by shutting down the existing pod and by doing this, the endpoints controller automatically switches the connection to the updated OpenPLC's OPC UA server. Once the automatic restart of all the pods running control applications is complete, the node is drained, and an update can be easily performed on the node.

If the same concept is used for updating control applications, the verification phase is required, and if the verification fails (the updated application is not behaving as expected) the rollback is quite difficult, as the pre-stop script cannot make the pod in the terminating state back to running again. So, using the pre-stop script approach for update is not the best idea for updates. Another problem is that the switchover of the io-simulator connections from existing OpenPLC runtime to the newly initialized OpenPLC runtime is not in the control of the user or the pre-stop script. This problem will not occur if the OPC UA client is also a part of the OpenPLC runtime instead of an external application. In that case there will be two OPC UA clients, one for each OpenPLC runtime, and these clients can directly establish connections with the io-simulators. This makes the switch over easier and more controllable to the user instead of depending on the endpoints controller in Kubernetes to do the switch. Using custom resource definition for industrial control applications to do the automatic update or restart including the state transfer could be a more mature approach, as the custom resource will be specifically designed to do updates considering the state transfer and it definitely has more control over the execution of pods and services as compared to a pre-stop script.

6 Results and Evaluation

The state transfer application discussed in Section 5.2.4 is a critical component during the update of an industrial control application running in kubernetes. The total time required for transferring the state from one control application to another was measured to test the feasibility of the solution and to observe the limits in terms of data sizes for two approaches of state transfer – Using IPC via POSIX Shared Memory and Using Network transfer using OPC UA. Later, these results are analyzed and other factors causing delays in the transfer times are discussed.

Industrial control applications are one of the critical applications in industrial automation, and due to a combination of its unique requirements – State Transfer, Zero Downtime Update, Strict Real-Time Processing, they must be thoroughly tested, because fulfilling all three requirements on cloud platforms is challenging. Updating control applications involves an important step of state transfer and as explained in Section 3.3, the state transfer must be completed in the slack time as the current implementation does not support multi-cycle state transfers. This feature is expected to be only relevant for extremely large applications or short execution cycles. Here, the focus is on the feasibility of the main concept and defer the multi-cycle approach to future work. By measuring the total time required to complete the state transfer (serialize, transfer and de-serialize), using the two approaches – IPC via POSIX Transfer and OPC UA Transfer for different data sizes, the maximum limit for the data which can be transferred without affecting the execution of the control application (state transfer time < slack time) is determined. Based on these results, the general feasibility of the approach is accessed.

As explained in Section 4.2.1.3, state transfer using shared POSIX memory promises to be a fast and reliable approach to transfer data. Also, OPC UA is evaluated as alternative as it is a well-known standard for industrial communication and is becoming popular in the control layer in for horizontal communication between PLC’s (Drahoš et al., 2018). To quantitatively assess them, the state transfer mechanism using these two approaches were implemented and tested. The third approach for state transfer using persistent volumes as mentioned in Section 4.2.1 was not implemented in the course of the thesis. It is left for future work.

6.1 State Transfer Time Experiments and Main Results

Heavy industries like manufacturing, Oil and Natural Gas, power generation have complex automation and control systems with more than 100,000 variables to monitor and control the different machinery and equipment in the plant. Such a huge number of variables makes the state size of the control applications very large and potentially more challenging to transfer. Yet, also small-scale industries and smaller systems exist, which have a smaller number of variables in the range of 10-1000 variables. To define a realistic range of data sizes to be tested, example systems can be considered.

For example, Herbert Krause has discussed the control and automation software for a Liquified Natural Gas (LNG) Plant in Norway. As per the author, the system contains 94,000 function blocks, 360,000 commissioning parameters, 650,000 variables in the control units, 350,000 global variables for communication and a cycle time of 200 ms (Krause, 2007). By assuming that almost 30% of the variables out of the 650,000 variables are retained (the state for these variables needs to be transferred), around 195,000 variables must be transferred during an update. If a maximum size of these variables is considered, i.e., each variable of 8 bytes or 64 bits, the state size to be transferred would approximately be 1.5 MB.

Another example is of Sadara – world's largest petrochemical complex where the control system consisted of 18 distributed control systems, 70 operator consoles, and around 150,000 I/O foundation fieldbus connections (ABB, 2017). In such huge systems the state sizes are in the range a few mega-bytes which is a large size for in industrial control system which have cycle times in the order of a few hundred milli-seconds. Smaller automation systems use fewer variables, and their state sizes are smaller.

Muslija has presented a case study on the complexity measurement of a train control management system by Bombardier Transportation Sweden AB. They analyzed 122 control applications and the number of variables used in these applications were in the range of 4-84 (Muslija, 2017). This number is very small when compared to the earlier examples, so, extensive experiments for a wide range of state sizes were designed. Experiments for state transfer for data sizes in a range of 36 kb to 2.7 MB were performed

6.1.1 Control Application Used and Nature of Variables Transferred

The OpenPLC runtime executing a simple 61131-3-based boiler control application (shown in Figure 5.4) with a cycle time is deployed on the Kubernetes cluster. The control application uses only four variables – two Boolean and two integer, but the buffer-size was 1024 which is equivalent to 1024 Input and Output Integer (2 bytes each) variables, 1024*8 Input and Output Boolean (1 byte each) variables, 1024 Input and Output byte (2 bytes each) variables, 1024 Memory Integer (2 bytes each) variables, 1024 Memory Double (4 bytes each) variables and 1024 (8 bytes each) Long variables. This adds up to a total size of 36kB. For testing purposes, instead of only copying the state of the variables in the control application, all the variables of the entire memory were marked for state transfer.

6.1.2 How was the state transfer time calculated?

The state transfer experiments were done for (1) OPC UA state transfer use case with pods running on the same physical node, (2) OPC UA with pods running on the different physical nodes and (3) IPC via POSIX on the same physical node. For the OPC UA approach, the total time after the initiation of the state transfer (when the OPC UA client calls the serialize method in the source OpenPLC) until the end of the transfer (when the OPC UA client in the state transfer application receives a success status code from the

Data Size	Buffer Size	Input Integers	Output Integers	Input Booleans	Output Booleans	Input Bytes	Output Bytes	Memory Integers	Memory Double	Memory Long	Total Variables
36 kb	1024	1024	1024	1024*8	1024*8	1024	1024	1024	1024	1024	23,552
180 kb	5000	5000	5000	5000*8	5000*8	5000	5000	5000	5000	5000	115,000
360 kb	10000	10000	10000	10000*8	10000*8	10000	10000	10000	10000	10000	230,000
540 kb	15000	15000	15000	15000*8	15000*8	15000	15000	15000	15000	15000	345,000
720 kb	20000	20000	20000	20000*8	20000*8	20000	20000	20000	20000	20000	460,000
1.08 MB	30000	30000	30000	30000*8	30000*8	30000	30000	30000	30000	30000	690,000
1.44 MB	40000	40000	40000	40000*8	40000*8	40000	40000	40000	40000	40000	920,000
1.88 MB	50000	50000	50000	50000*8	50000*8	50000	50000	50000	50000	50000	1,150,000
2.7 MB	75000	75000	75000	75000*8	75000*8	75000	75000	75000	75000	75000	1,725,000

Table 6.1 : Breakdown of experimental parameters used in state transfer experiments

target OpenPLC) was calculated (Refer Figure 5.6). The transfer time for IPC via POSIX approach was calculated without considering the OPC UA overhead. The time for serializing, archiving and writing the state in the shared memory from the source OpenPLC was measured. This time was added with the time required for the target OpenPLC to de-serialize, read and then copy the values to the PLC memory (Refer Figure 5.5). The state transfer in each case was repeated 1000 times to achieve 1000 distinct values which can be averaged out to remove any fluctuations.

These experiments were repeated for buffer sizes from a range of 1024 (36 kB) to a buffer size of 75000 (2.7 MB). Table 6.1 shows the different data sizes the state transfer experiment was repeated for. The first column represents the data size used in each experiment and the second column represents the buffer size. All the variables in the memory are stored in the form of an array of size equal to the buffer size. For the state transfer, all the variables were transferred, but at the sending and the receiving end in the OpenPLC runtime, only the input-output booleans and input-output integers were processed, and the other variables were initialized at default values. Processing all the variables instead of a limited number of variables is not expected to cause a huge impact

Transfer → / Data Size ↓	OPC UA Same Node		OPC UA Different Node		IPC via POSIX Transfer	
	Mean ms	Median ms	Mean ms	Median ms	Mean ms	Median ms
36 kb	9.2616	8.6945	7.7406	7.3915	3.4984	3.4250
180 kb	27.3745	25.3585	25.2997	24.2310	16.9567	15.8450
360 kb	52.7567	50.0225	50.6416	49.8085	43.0406	40.9390
540 kb	377.1343	322.4325	407.5989	350.9120	199.8418	164.2325
720 kb	593.3057	509.3335	626.2894	554.6625	261.4155	218.9420
1.08 MB	938.2413	822.3500	1091.7047	957.5265	436.8433	365.9560
1.44 MB	1409.1815	1251.9585	1444.2984	1321.6830	621.8367	505.6830
1.8 MB	1694.9999	1500.5175	1647.6269	1480.8970	796.8021	665.2220
2.7 MB	2530.9898	2357.4535	2565.5177	2586.2150	1198.8593	987.5870

Table 6.2 : Mean and Median State Transfer Time in milli-seconds for different use cases and data sizes

on the readings. The detailed results in CSV format are published on GitHub (Sonar, 2020).

6.1.3 Results

Table 6.2 shows the average and median values of the state transfer time for the three different use cases mentioned before – (1) OPC UA Same Node, (2) OPC UA Different Node and (3) IPC via POSIX transfer for data sizes ranging from 36kb to 2.7 MB. Each of the configurations were repeated 1000 times and the average and median were calculated. The results indicate that the mean/median state transfer time for IPC via POSIX transfer is less than that of OPC UA transfer times. This could happen because for the IPC approach there is no network overhead involved. In the shared memory approach, no network connection is established, and simply the state is written on the shared memory and then it is read from another process. The results show that the performance of the OPC UA based state transfer when the source and the target applications (pods) are on the same node and when they are on different nodes, is almost similar. Ideally, the state transfer times for the same node should be lower than the time for different node. The applications were tested in a kubernetes cluster shared by other components and due to the external factors like testing for any other component might affect the performance of these timings. But this depicts a real-life use case where a different applications are hosted on a single cluster. Due to the smaller data size and a gigabit network, the state transfer times could be almost same. Also, the performance of the OPC UA implementation could be slow, and the network performance compared to that might not affect the latency results.

The control application execution time was measured for the first three data sizes (36 kb, 180 kb and 360 kb), and they were very low (0.0833 ms for memory of 36 kb) as compared to the 200ms cycle time used. The measurements are shown in Table 6.3. For this control application it can be assumed that the slack time is almost equal to the cycle time - 200ms. The average state transfer time exceed the cycle time for state sizes above 360kb for OPC UA use case and above 540kb for IPC via POSIX use case. So, for higher state sizes, the state transfer must be done in multiple cycles.

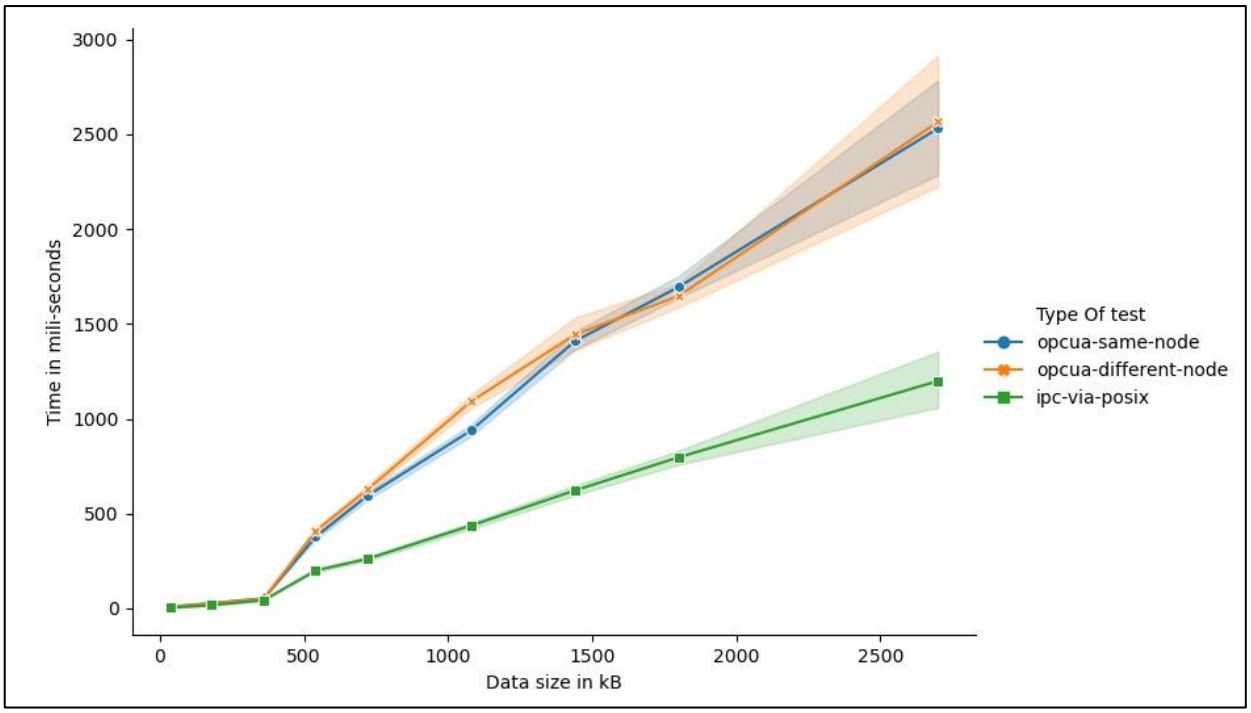

Figure 6.1 : Line chart with mean time for state transfer and a 95% confidence interval around the mean for different state sizes

Figure 6.1 shows a line graph with the time taken for state transfer on the y-axis and the data size in kB on the x-axis. The line is plotted with mean values of the state transfer time and a confidence interval[1] of 95% around the mean value. It can be seen from the graph that the state transfer time increases drastically after the data size increases above 360 kb. Also, for the lower data sizes the confidence interval is not that widespread as compared to the larger data sizes.

Figure 6.2 shows the box plot[2] of state transfer time plotted for different data sizes. The box plot gives an idea of the outliers (extreme values) and the symmetry of the data. If the box plot is very wide, then the data is less symmetrical and there are more variations in the data as they are more widespread from the median value. In the figure, the values which are more than or less than 1.5 times the interquartile range (the length of the box in each respective box plot) are considered as outliers. It can be observed that higher the value of data size, more are the outliers in the results. For lower values state sizes, the box plots are less widespread.

[1] Confidence Interval : https://www.mathsisfun.com/data/confidence-interval.html, Accessed 22.10.2020
[2] Box Plot : https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51, Accessed 22.10.2020

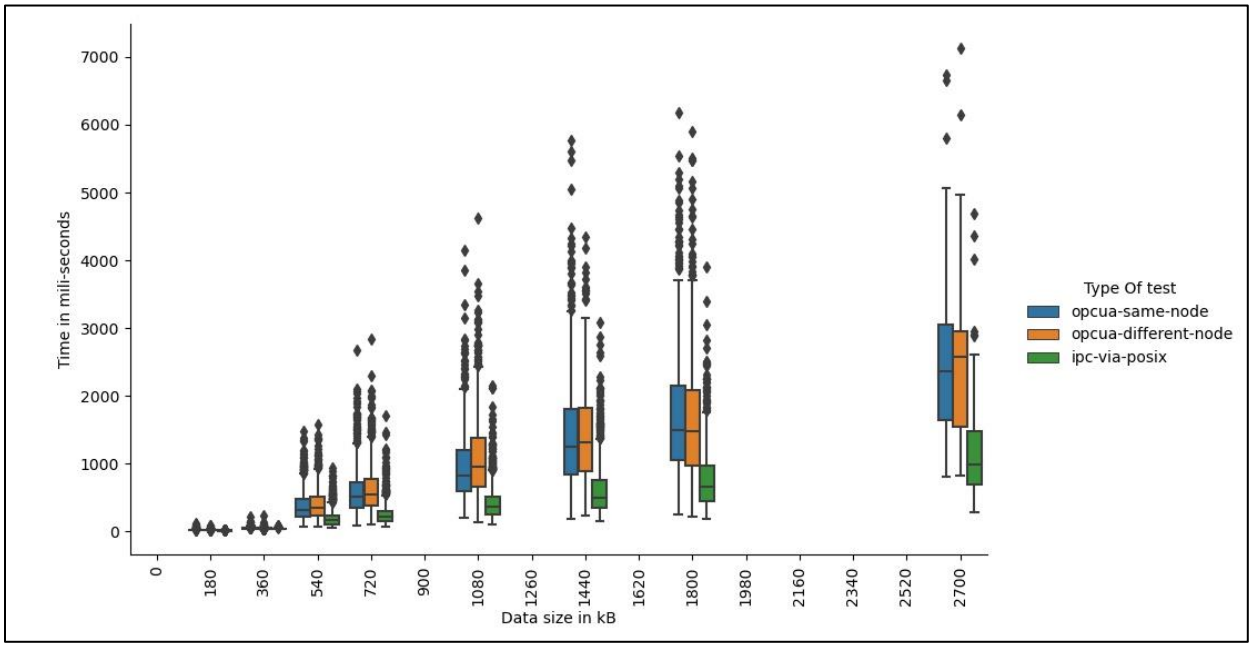

Figure 6.2 : Box Plot with median time for state transfer and outliers beyond 1.5 times the interquartile range for different state sizes

6.2 Detailed Analysis of Results

The average state transfer time results for IPC via POSIX transfer were much lower than the average time for OPC UA transfer. Based on the experiments performed by Venkataraman and Jagadeesha, the average communication latency for IPC transfer using shared memory for a data size of 32kb is around 6-7 microseconds (µs) (Venkataraman and Jagadeesha, 2015). Whereas the state transfer time required for transferring a state of 36kb is around 3 milliseconds (ms). Although the state transfer process involves more steps than just the state transfer – 1. Reading the memory from source OpenPLC memory, 2. Serializing it, 3. At the receiving end, de-serializing the state transfer object and 4. Writing the data to the target OpenPLC's memory. Based on these numbers for the IPC Transfer it can be said that most of the time is spent in the above discussed steps rather than the actual state transfer. Similarly, the average round trip time for OPC UA client/server communication for a data size of 32 kb in the range of 1500 – 2000 µs (1.5 – 1.8 ms) for different conditions like 100% CPU load, Idle CPU and 100% Network load. The state transfer time for the OPC UA transfer was in the range of 9 ms for 36 kb data (Profanter et al., 2019). For the case of OPC UA as well more amount time is spent on the remaining four steps explained earlier.

6.2.1 Analysis of State Transfer Time, Control Application Execution Time and OPC UA Thread execution time

To analyze the results and to understand why more time was spent in the setup steps rather than the actual state transfer, the control application execution time and the OPC UA thread execution time to read and write from the memory was calculated. The cycle time of the control application was 200 ms and the OPC UA thread was executing in parallel to the control application after every 50 ms. The task of the OPC UA thread was to update the OpenPLC memory with input values coming from sensors and to update the OPC UA namespace of the server with output values written by the control application and supposed to be sent to the actuators. Although in the control application only 4 variables were uses, but the entire memory was updated by the OPC UA thread. During any interaction with the memory of the OpenPLC, a mutex lock was used so two processes cannot simultaneously access the memory as it might corrupt the memory. Sometimes, the control application execution or the state transfer gets delayed as the OPC UA thread is updating a variable in the memory. The mutex lock and unlock was placed around the update of each variable individually. So, when a variable is updated by the OPC UA thread, the mutex lock was removed and so other processes can access the memory. In order to make the process of updating variables in OPC UA thread faster, the

```
for(variable in variables){
      pthread_mutex_lock(&bufferLock); // Lock Memory
      updateVariableInOPCUAandMemory (variable);
pthread_mutex_unlock(&bufferLock); // Unlock Memory
}
```

Code Snippet 6.1: OPC UA thread with multiple locks

```
pthread_mutex_lock(&bufferLock); // Lock Memory
for(variable in variables){
      updateVariableInOPCUAandMemory (variable)
}
pthread_mutex_unlock(&bufferLock); // Unlock Memory
```

Code Snippet 6.2 : OPC UA thread with single lock

multiple locks for updating each variable individually were replaced by a single lock. There are two use cases here 1. OPC UA thread with multiple mutex locks (The default implementation in which all the latency experiments were conducted) and 2. OPC UA thread with a single lock. Code Snippet 6.1 and 6.2 show the sample code for the multiple and single locks in the OPC UA thread for updating the variable values.

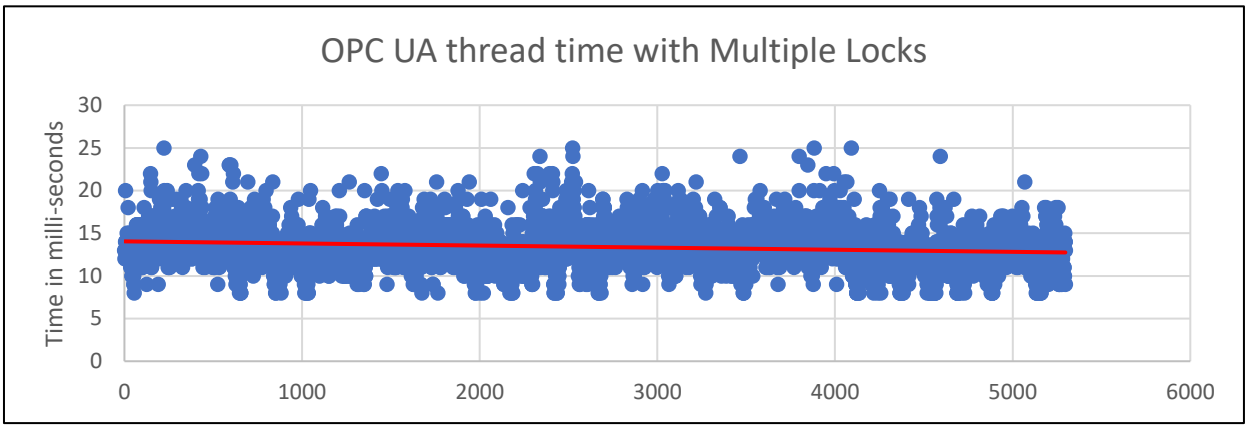

Figure 6.3 : Scatter plot for time required for updating input and output variables of OpenPLC using multiple locks plotted against number of OPC UA thread cycles

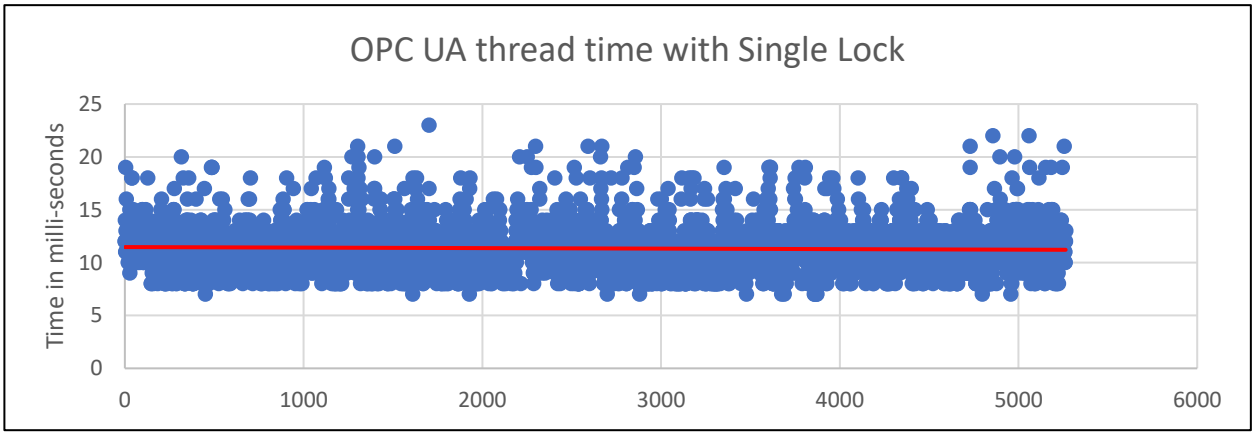

Figure 6.4 : Scatter plot for time required for updating input and output variables of OpenPLC using single lock plotted against number of OPC UA thread cycles

Figure 6.3 and Figure 6.4 show the scatter plot for the time required by the OPC UA thread to update the variables in the memory as well as in the OPC UA server address space with using multiple mutex locks and single mutex lock respectively. The total memory size used was 36 kb. It can be seen from the figures that the trendline for the single lock scatter plot is at around 12 ms and the same for the multiple locks scatter plot starts at around 15 ms. This means that the average time required for updating the

variables in the OPC UA thread is lower for the single lock use case, as the time required for locking and unlocking the memory is saved.

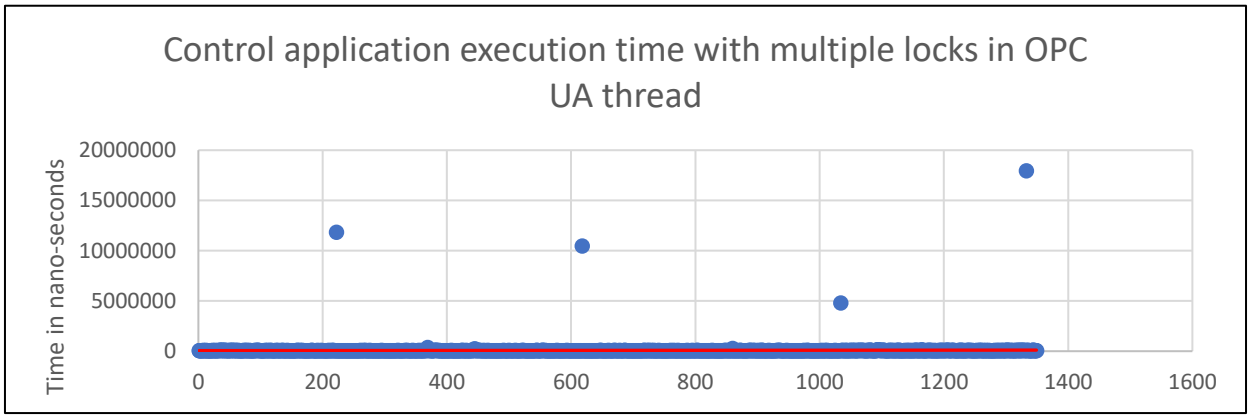

Figure 6.5 : Scatter plot for Control application execution time with multiple locks in OPC UA thread plotted against number of PLC cycles

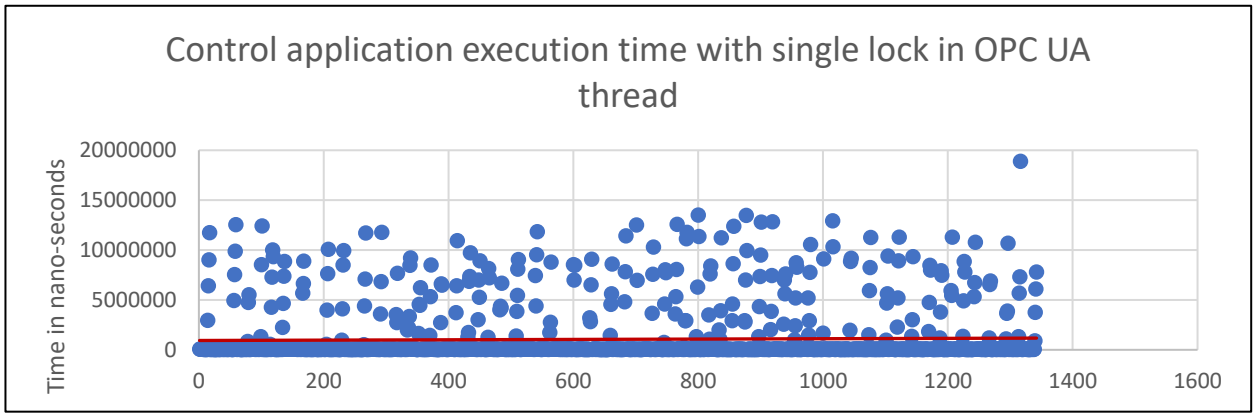

Figure 6.6 : Scatter plot for Control application execution time with single lock in OPC UA thread plotted against number of PLC cycles

Now, the effect of using single lock and multiple lock mechanism on the execution of the control application is checked. Figures 6.5 and 6.6 show the time required for the execution of the control application when multiple and single mutex locks are used in the OPC UA thread respectively. The graphs show that the execution time for the control application which is running with the OPC UA thread with multiple locks is almost constant except for a few outliers (4 out of 1347 records) while on the other hand, the execution time for the control application which is running with the OPC UA thread with single lock has a number of outliers.

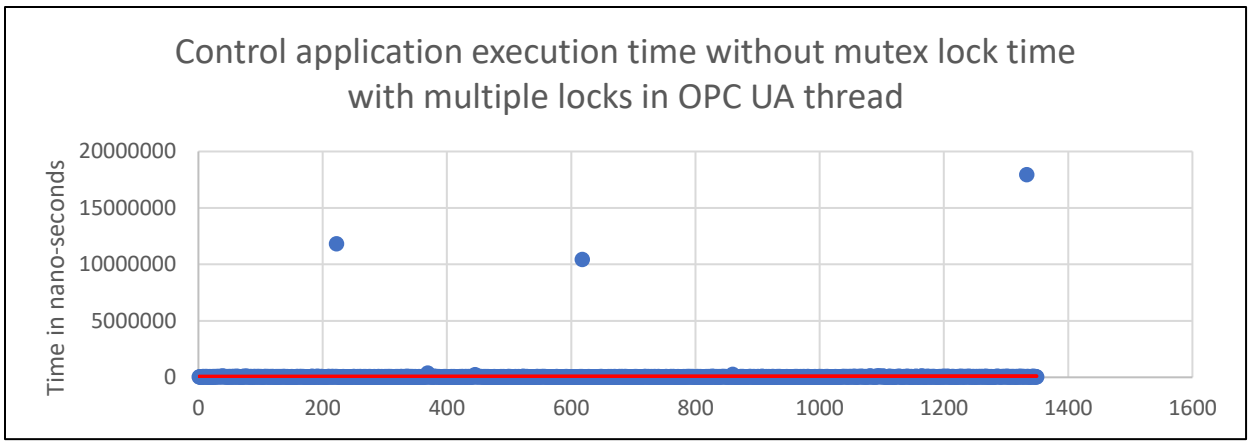

Figure 6.7 : Scatter plot for control application execution time without mutex lock time with multiple locks in OPC UA thread plotted against number of cycles

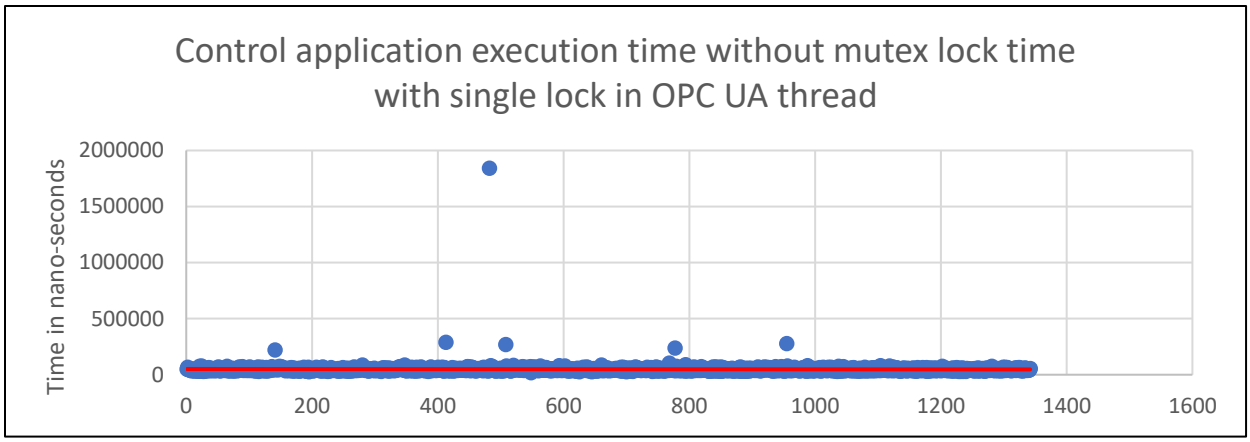

Figure 6.8 : Scatter plot for control application execution time without mutex lock time with single locks in OPC UA thread plotted against number of cycles

To verify that the outliers are caused due to the OPC UA thread which blocks the entire memory with a single lock, the control application execution time for both use cases (single and multiple locks) by removing the time required for waiting for the lock and unlock of the memory was calculated. So, if t_{CA} is the total time required for the execution of the control application and t_{ML} is the time required for execution mutex lock command in the control application, a graph for $t_{CA} - t_{ML}$ was plotted. Figures 6.7 and 6.8 show the scatter plot of control application time without the mutex lock time for multiple and single locks in OPC UA thread respectively. The scatter plot for the multiple locks in OPC UA thread looks the same as it was earlier in Figure 6., but the plot for the single lock now does not contain as main outliers as they were before in Figure 6.6. The results without the mutex lock time show that using a single lock in the OPC UA thread for updating the

input-output variables makes the process faster, but at the cost of making the control application loops slower. The single could also make the state transfer process wait, adding unnecessary delay.

The time taken for the state transfer is high because of the continuously polling of the variables in the OPC UA thread which adds locks to the (multiple) memory. Due to the continuous polling the read and the write speeds are affected. Continuous polling is not the best option while updating the input-output variables. Rather the input variables must be updated before the beginning of the PLC cycle and the output variables at the end of the PLC cycle. Also, only the variables which are updated or modified must be polled at the end of each cycle, this will also optimize the state transfer time. Another way to update the input variables in the PLC memory from the OPC UA address space is to have callbacks for on-write events. Whenever a variable is updated in the OPC UA address space, the on-write callback must trigger a function which can update the respective variable in the PLC memory. The open62541-based implementation supports this but when the OPC UA logic is running a parallel thread, the functionality does not work[1]. To solve this issue a commercial implementation of OPC UA – Unified Automation OPC UA Client server SDK can be used instead of the opensource version.

6.2.2 Analysis of Sudden Steep Curve in Line Plot

The line plot in Figure 6.1 shows the mean values of the state transfer times for different data sizes. In the graph as the memory increases, there is a steep increase in the state transfer timings after the first three data sizes (36 kb – 360 kb). The time taken for OPC UA thread to update the input-output variables of the plc was calculated for the data sizes of 23 kb, 180 kb and 360 kb. Table 6.3 shows the OPC UA thread times in milliseconds and from the table it can be clearly seen that with the increase in the data size the time taken by OPC UA thread to update the IO variables also increases. The cycle time for the control application is 200 ms and for 360 kb of data the OPC UA variable thread itself takes on an average of 316 ms which is more than the cycle time. Generally, the OPC UA thread time must be less than the cycle time so that the OPC UA thread does not

[1] Github Issue For OPC UA Callback functions : https://github.com/open62541/open62541/issues/2831, Accessed 22.10.2020

affect the execution of control applications and the state transfer. If the data size is increased more than 360 kb, the state transfer time could be almost double. So, the state transfer also measures the waiting time during which the OPC UA thread is accessing the memory and this time increases with increase in data size.

Data Size	OPC UA thread update variables average time in ms	Average PLC execution time in ms
36 kb	13.3953181	0.0833
180 kb	95.31878032	0.0593
360 kb	316.8290698	0.1245

Table 6.3 : OPC UA thread update variables average time for different data sizes

The sudden rise in the state transfer times higher data sizes (> 360 kb or 504 kb) is caused because of the increase in the OPC UA thread update variables time beyond the cycle time of the application. There could be also chance that variables are updated by the sensor and the control application receives the signals after 4 or 5 cycles (or after 800 – 1000 ms) which might be catastrophic in complex industrial automation systems. So, running the control application in which, the cycle time is less and the OPC UA thread update variable time is more makes very little sense.

7 Conclusion and Outlook

With the push towards Industry 4.0, the involvement of IOT technologies in industrial automation is increasing. Using bare metal hardware for programmable logic controllers which run industrial control applications is the most popular approach used by industries for automation even today. These hardware-based PLC's are equipped with ethernet connectivity so that they can be connected to the IOT infrastructure. Large scale industries use multiple PLC devices which are all interconnected to each other Maintaining the hardware and updating software for each of these devices is a complex costly and time-consuming task, especially security patches and updates which occur very frequently and are critical updates. Cloud-based control applications could save a lot of cost, time, and effort in automation systems.

A study of existing update strategies used in container orchestration systems is presented in Section 3.4. Based on the existing literature discussed in Section 3.3, it can be said that the update of an industrial control application involves four main steps – Initialization, state-synchronization, verification and switch-over of field device connections. Chapter 4 describes a novel technique for performing disruption-free software updates to industrial control applications running in container orchestration systems using a state transfer application. The concept of a standalone state transfer application to transfer the internal state from existing control application to the newly initialized control application is discussed, designed, developed, and tested using two different approaches – Transfer via Network and Transfer via Shared Memory. Since state transfer is a crucial part, the state transfer time was calculated for different state sizes to find the maximum size of the data that can be transmitted from source to destination without affecting the execution of the control application. The results show that a state size of up to 360 kb can be transmitted by the state transfer application for a control application having a cycle time of 200 ms running in OpenPLC runtime with OPC UA Client-Server for communication with field devices. Based on the results, it can be stated that the solution is feasible and can work in the long run. The state transfer time for the shared memory approach was much lower than the network transfer approach as the network transfer approach has a network overhead of creating and maintaining a network connection.

Apart from the state transfer application, the concept of automatically updating control application using container lifecycle hooks or custom resources in container orchestration systems is an important contribution as it automates the process of updating the control applications. Automatic updates save a lot of time, cost and the risk of human error which is involved in manual updates. The use of custom resources for deploying and updating industrial control application to cater specific requirements like the state transfer requirement can revolutionize the industrial automation sector.

7.1 Discussion on research questions

RQ1 : What additional steps must be carried out to update an industrial control application which is running in a container orchestration system?

Section 3.3 and Section 4.2 discusses about the steps involved in the update of a control application, and the requirement that the state transfer must be done within the slack time is a very unique requirement for industrial control applications and container orchestration systems like Kubernetes don’t support such a requirement. So, the state transfer application must be developed to perform the update of industrial control applications.

RQ2 : How the internal state of the existing control application will be restored in the updated control application in a cloud-native environment where the existing and the updated control applications might run on same or different physical nodes ?

To restore the internal state of variables, the concept of a state transfer application which can copy the state of the existing control application and send it to the updated control application using three different approaches is discussed in Section 4.2.1. The state transfer application was implemented using OPC UA transfer via network and IPC via the POSIX shared memory. The OPC UA approach transfers the state via the network, so, it does not matter if the nodes are not on the same physical node. The implementation of the state transfer application is discussed in Section 5.2.4.

RQ3 : How can the existing features of container orchestration systems be used for updating industrial automation software?

Initially, the plan was to use the rolling update feature in Kubernetes to update control applications, but as explained in Section 3.4.1, rolling updates work for only stateless applications or stateful applications which support distributed processing, or have a

common data store external to the execution units. So, the concept of rolling updates cannot be used by all the use cases since very few PLCs' support distributed processing of control applications. Another important functionality provided by container orchestration systems are the container lifecycle hooks which control the container termination and container startup. Section 4.2.3.2 discusses about the use of pre-stop hook in Kubernetes to trigger a pre-stop script which can transfer the state of the existing control application to the updated control application and then to switch the field device connections to the updated application.

RQ4 : What are the different update strategies used for providing disruption free updates to application running in container orchestration systems like Kubernetes, and which of these strategies can be used for updating the stateful industrial control applications?

Section 3.4 discusses about the different update strategies which are used in Kubernetes, and some of which are supported by Kubernetes and some are additional. In Section 4.2.2 it is discussed that the Blue-Green strategy and the Primary-Secondary strategy are relevant for updating industrial control application with zero-downtime, as both these strategies support testing with live sensor data without affecting the control process.

RQ5 : How are manual and automatic updates for industrial control applications implemented in a container orchestration environment?

Section 4.2.3 discusses about the concepts of manual and automatic updates for updating control applications. The manual update approach is generally used when there is a major change (for example, adding new variables) in the control application and needs a manual verification of the functioning of the updated application. On the other hand, automatic update approach must be used in most of the use cases (for example, security update in the underlying OS of the container), which do not need a manual verification of the functioning of the control application. As discussed in Section 4.2.3.1, Section 5.3.1 and Section 5.3.2, manual updates are implemented by manually executing all the steps involved in the update of a control application including running the state transfer application via command line or the state transfer UI and the verification of the functioning of the control application. Referring to Section 4.2.3.2, an automatic update

can be implemented by using two approaches – Using the pre-stop hook in Kubernetes and Using a custom resource for performing an automatic update.

7.2 Future Work

7.2.1 Using Custom Resource Definition

The use of Custom Resources which can be integrated in the control plane of a container orchestration system for delivering automatic updates to industrial control applications is discussed in this thesis. Although the pre-stop hook approach worked for the automatic update use case, but it lacks the functionality of an automatic rollback if anything goes wrong, on the other hand, custom resources have direct access to the control logic and the developers can modify the existing container and pod creation lifecycles. So, the rollback functionality can be added to the custom resources. Custom resources provide features like automatic updates and high availability. For example, custom resources can be used provide high availability for the control applications in case of a node failure by restoring the last saved state of the control application and starting a new instance of the application another node. Implementation of a custom resource for deploying and updating control applications in container orchestration system like Kubernetes can be the future of software-based programmable logic controllers and industrial control systems. Instead of manufacturing the hardware components for programmable logic controllers, automation manufacturers can directly ship the custom resource configuration file which can run in a specific container orchestration system like Kubernetes at the costumer's end.

7.2.2 State transfer across multiple cycles

The results in Section 6.1 and Table 6.2 show that the average state transfer time for OPC UA approach above the data size of 360 kb is more than the cycle time (almost equal to slack time) which is 200 ms. As discussed by Wahler and Oriol in their article and in Section 3.3, the state transfer must be done within the slack time and if that is not possible, then the state transfer needs to be done across multiple cycles, otherwise it might affect the execution of the control application (Wahler and Oriol, 2014). The authors discuss an approach in which they suggest the use of a dirty bit vector for each variable in the memory, and once all the variables in the dirty bit vector are marked as not dirty,

then the state transfer is complete. In future, the existing state transfer application can be modified so that it can support the feature of doing a state transfer between the existing and the updated control application which can be distributed across multiple cycles. Such a multi cycle state transfer application can be used in large scale industrial projects which have a state size of more than 1-5 MB, which is a huge size considering the cycle time of such applications which is around a few milli-seconds.

7.2.3 Creating a User Interface for state transfer application

Developing a user interface for the state transfer applications would make manual update process very easy. As a part of the implementation, the state transfer was executed from the command line, and then to monitor the updated variables a UA Expert OPC UA client was used. Having a UI for executing the state transfer and then viewing the variables in immediately the next screen can definitely make the process more user friendly and easy for the users of this applications - field engineers who might not be experts in executing command line applications.

List of Figures

List of Tables

References

ABB (2015) *System 800xA: Engineering and Production Environments* [Online], ABB (3BSE045030-510 C). Available at https://library.e.abb.com/public/7de2b5acd288480695245919ceedc03b/3BSE045030-510_C_en_System_800xA_Engineering_5.1_Engineering_and_Production_Environments.pdf (Accessed 25 July 2020).

ABB (2017) *Sadara – enabling world's largest petrochemical complex built in a single phase: Improving efficiency, cutting costs and mitigating risks* [Online], ABB. Available at https://search.abb.com/library/Download.aspx?DocumentID=9AKK107045A4028&LanguageCode=en&DocumentPartId=&Action=Launch (Accessed 29 September 2020).

Alves, T. R. *OpenPLC Project* [Online]. Available at https://www.openplcproject.com/ (Accessed 23 September 2020).

Alves, T. R., Buratto, M., de Souza, F. M. and Rodrigues, T. V. (2014) 'OpenPLC: An open source alternative to automation', *IEEE Global Humanitarian Technology Conference (GHTC 2014)*, pp. 585–589.

Baukes, M. (2020) *LXC vs Docker: Why Docker is Better* [Online], UpGuard. Available at https://www.upguard.com/blog/docker-vs-lxc (Accessed 13 October 2020).

Baxter, J. (2020) *KubeDirector Architecture Overview* [Online]. Available at https://github.com/bluek8s/kubedirector/wiki/KubeDirector-Architecture-Overview (Accessed 6 September 2020).

Beda, J. (2017) *Core Kubernetes: Jazz Improv over Orchestration* [Online], Heptio. Available at https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca (Accessed 20 August 2020).

Bernstein, D. (2014) 'Containers and Cloud: From LXC to Docker to Kubernetes', *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84.

Bruckner, D., Stănică, M., Blair, R., Schriegel, S., Kehrer, S., Seewald, M. and Sauter, T. (2019) ‘An Introduction to OPC UA TSN for Industrial Communication Systems’, *Proceedings of the IEEE*, vol. 107, no. 6, pp. 1121–1131.

Burger, A., Stomberg, G., Rückert, J., Koziolek, H. and Platenius-Mohr, M. (2019) ‘OpenPnP: A Plug-and-Produce Architecture for the Industrial Internet of Things’, *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 131–140.

Casalicchio, E. (2019) ‘Container orchestration: A survey’, in *Systems Modeling: Methodologies and Tools,* Springer, pp. 221–235.

Cloud Native Computing Foundation (2018) *CNCF Cloud Native Definition v1.0* [Online], CNFC (Accessed 27 August 2020).

Cohen, B., Csatári, G., Huang, S., Jones, B., Lebre, A., Paterson, D. and Váncsa, I. (2020) *Edge Computing: Next Steps in Architecture, Design and Testing* [Online], Openstack. Available at https://www.openstack.org/use-cases/edge-computing/edge-computing-next-steps-in-architecture-design-and-testing/ (Accessed 21 September 2020).

Coulon, X. (2019) *Writing Your First Kubernetes Operator* [Online], Medium. Available at https://medium.com/faun/writing-your-first-kubernetes-operator-8f3df4453234 (Accessed 11 August 2020).

Debian (2020) *Debian - Homepage* [Online], Debian. Available at https://www.debian.org/ (Accessed 6 August 2020).

Dinesh, S. (2018) *Kubernetes best practices: upgrading your clusters with zero downtime* [Online], GOOGLE CLOUD PLATFORM. Available at https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-upgrading-your-clusters-with-zero-downtime (Accessed 14 August 2020).

Dobies, J. and Wood, J. (2020) *Kubernetes Operators: Automating the Container Orchestration Platform* [Online], O'Reilly Media, Inc. Available at https://www.oreilly.com/library/view/kubernetes-operators/9781492048039/.

Docker (2020) *Docker overview* [Online], Docker. Available at https://docs.docker.com/get-started/overview/#the-underlying-technology (Accessed 13 October 2020).

Drahoš, P., Kučera, E., Haffner, O. and Klimo, I. (2018) 'Trends in industrial communication and OPC UA', *2018 Cybernetics Informatics (K I)*, pp. 1–5.

Eldridge, I. (2018) *What Is Container Orchestration?* [Online], New Relic. Available at https://blog.newrelic.com/engineering/container-orchestration-explained/ (Accessed 7 October 2020).

Felter, W., Ferreira, A., Rajamony, R. and Rubio, J. (2015) 'An updated performance comparison of virtual machines and Linux containers', *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172.

Gangakhedkar, S., Cao, H., Ali, A. R., Ganesan, K., Gharba, M. and Eichinger, J. (2018) 'Use cases, requirements and challenges of 5G communication for industrial automation', *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6.

Gannon, D., Barga, R. and Sundaresan, N. (2017) 'Cloud-Native Applications', *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21.

Garrison, J. and Nova, K. (2017) *Cloud Native Infrastructure*, O'Reilly Media, Inc.

Gilchrist, A. (2016) *Industry 4.0: The Industrial Internet of Things* [Online], APRESS. Available at https://www.springer.com/de/book/9781484220467 (Accessed 6 October 2020).

Gogolev, A. (2020) *OPC UA and TSN: enabling Industry 4.0 for end devices* [Online], ABB. Available at https://new.abb.com/news/detail/56916/opc-ua-and-tsn-enabling-industry-40-for-end-devices (Accessed 18 September 2020).

Gold, J., Foteinopoulos, S., Lübken, M., Rosequist, A. and Vuk, I. (2020) *ContainerSolutions : k8s-deployment-strategies* [Online], Container Solutions. Available at https://github.com/ContainerSolutions/k8s-deployment-strategies (Accessed 28 July 2020).

Goldschmidt, T. and Hauck-Stattelmann, S. (2016) ‘Software Containers for Industrial Control’, *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 258–265.

Goldschmidt, T., Murugaiah, M. K., Sonntag, C., Schlich, B., Biallas, S. and Weber, P. (2015) ‘Cloud-Based Control: A Multi-tenant, Horizontally Scalable Soft-PLC’, *2015 IEEE 8th International Conference on Cloud Computing*, pp. 909–916.

Harper, K. E., Zheng, J., Jacobs, S. A., Dagnino, A., Jansen, A., Goldschmidt, T. and Marinakis, A. (2015) ‘Industrial Analytics Pipelines’, *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pp. 242–248.

Hofer, F., Sehr, M. A., Iannopollo, A., Ugalde, I., Sangiovanni-Vincentelli, A. and Russo, B. (2019) ‘Industrial Control via Application Containers: Migrating from Bare-Metal to IAAS’, *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 62–69.

Ibryam, B. and Huß, R. (2019) *Kubernetes Patterns: Reusable Elements for Designing Cloud-native Applications* [Online], O’Reilly Media. Available at https://www.oreilly.com/library/view/kubernetes-patterns/9781492050278/.

International Electrotechnical Commission (2000): *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES).* [Online]. Available at http://www.cechina.cn/eletter/standard/safety/iec61508-2.pdf (Accessed 9 September 2020).

Kahoot, A. (2019) *K8s: Deployments vs StatefulSets vs DaemonSets* [Online], Medium. Available at https://medium.com/stakater/k8s-deployments-vs-statefulsets-vs-daemonsets-60582f0c62d4 (Accessed 4 June 2020).

Kasireddy, P. (2016) *A Beginner-Friendly Introduction to Containers, VMs and Docker* [Online], freeCodeCamp. Available at https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/ (Accessed 27 August 2020).

Kim, K. (2020) *HDFS on Kubernetes—Lessons Learned* [Online]. Available at https://databricks.com/session/hdfs-on-kubernetes-lessons-learned.

Krause, H. (2007) 'Virtual commissioning of a large LNG plant with the DCS 800XA by ABB', *6th EUROSIM Congress on Modelling and Simulation, Ljubljana, Slovénie*.

Kubernetes (2019) *Kubernetes Release Versioning* [Online], Kubernetes. Available at https://github.com/kubernetes/community/blob/master/contributors/design-proposals/release/versioning.md (Accessed 21 October 2020).

Kubernetes (2020a) *Container Lifecycle Hooks* [Online], Kubernetes. Available at https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/#hook-details (Accessed 20 August 2020).

Kubernetes (2020b) *Deployments* [Online], Kubernetes. Available at https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (Accessed 12 June 2020).

Kubernetes (2020c) *Kubernetes Components* [Online], Kubernetes. Available at https://kubernetes.io/docs/concepts/overview/components/ (Accessed 2 September 2020).

Kubernetes (2020d) *Performing a Rolling Update* [Online], Kubernetes. Available at https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/ (Accessed 12 June 2020).

Kubernetes (2020e) *Persistent Volumes* [Online], Kubernetes. Available at https://kubernetes.io/docs/concepts/storage/persistent-volumes/ (Accessed 27 July 2020).

Kubernetes (2020f) *StatefulSets* [Online], Kubernetes. Available at https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/ (Accessed 12 June 2020).

Kubernetes (2020g) *Upgrading kubeadm clusters* [Online], Kubernetes. Available at https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/ (Accessed 14 August 2020).

KUNBUS GmbH *Fieldbus Basics and Tutorial* [Online], KUNBUS GmbH. Available at https://www.kunbus.com/fieldbus-basics.html (Accessed 18 September 2020).

Lahti, J. P., Shamsuzzoha, A. and Kankaanpää, T. (2011) 'Web-based technologies in power plant automation and SCADA systems: A review and evaluation', *2011 IEEE International Conference on Control System, Computing and Engineering*, pp. 279–284.

Leitner, S.-H. and Mahnke, W. (2006) 'OPC UA-service-oriented architecture for industrial applications', *ABB Corporate Research Center*, vol. 48, pp. 61–66.

Magro, M. C. and Pinceti, P. (2007) 'Measuring Real Time Performances of PC-based Industrial Control Systems', *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*, pp. 540–547.

Mangat, M. (2019) *Kubernetes vs Docker Swarm: What are the Differences?* [Online]. Available at https://phoenixnap.com/blog/kubernetes-vs-docker-swarm (Accessed 31 August 2020).

Mercl, L. and Pavlik, J. (2019) 'Public Cloud Kubernetes Storage Performance Analysis', *Computational Collective Intelligence.* Cham, Springer International Publishing, pp. 649–660.

Mugarza, I., Parra, J. and Jacob, E. (2018) 'Analysis of existing dynamic software updating techniques for safe and secure industrial control systems', *International journal of safety and security engineering*, vol. 8, no. 1, pp. 121–131.

Muslija, A. (2017) *On the complexity measurement of industrial control software*, Vasteras, Sweden, Malardalen University [Online]. Available at https://www.diva-portal.org/smash/get/diva2:1113035/FULLTEXT01.pdf (Accessed 7 October 2020).

Nair, M. (2017) *How Netflix works: The (hugely simplified) complex stuff that happens every time you hit Play* [Online], Medium. Available at https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b (Accessed 7 October 2020).

Netto, H. V., Luiz, A. F., Correia, M., de Oliveira Rech, L. and Oliveira, C. P. (2018) 'Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes', *2018 IEEE Symposium on Computers and Communications (ISCC)*, pp. 58–63.

Okano, M. T. (2017) ‘IOT and industry 4.0: the industrial new revolution’, *International Conference on Management and Information Systems*, p. 26.

Ongaro, D. and Ousterhout, J. (2014) ‘In search of an understandable consensus algorithm’, *2014 5USENIX6 Annual Technical Conference (5USENIX65ATC6 14)*, pp. 305–319.

open62541 (2020) *Open62541 - Releases* [Online]. Available at https://github.com/open62541/open62541/releases (Accessed 9 August 2020).

OpenKruise (2020) *Cloneset* [Online], Alibaba Cloud. Available at https://openkruise.io/en-us/docs/cloneset.html (Accessed 4 August 2020).

Opensource (2020) *What is virtualization?* [Online], Opensource.com. Available at https://opensource.com/resources/virtualization (Accessed 13 October 2020).

Petruzella, F. D. (2005) *Programmable logic controllers*, 4th edn, Tata McGraw-Hill Education.

Phelan, T. (2018) *KubeDirector: The easy way to run complex stateful applications on Kubernetes* [Online], BlueData. Available at https://kubernetes.io/blog/2018/10/03/kubedirector-the-easy-way-to-run-complex-stateful-applications-on-kubernetes/ (Accessed 6 September 2020).

Plant Automation Technology (2020) *Different Types Of Automation Industry Tools* [Online], Plant Automation Technology. Available at https://www.plantautomation-technology.com/articles/different-types-of-automation-industry-tools (Accessed 7 September 2020).

PLCopen (2013): *IEC 61131-3: Programming Languages (edition 3.0 - 2013)* [Online]. Available at https://plcopen.org/iec-61131-3 (Accessed 7 September 2020).

Profanter, S., Tekat, A., Dorofeev, K., Rickert, M. and Knoll, A. (2019) ‘OPC UA versus ROS, DDS, and MQTT: performance evaluation of industry 4.0 protocols’, *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*.

Redhat (2020a) *Documentation - OpenShift Container Platform v 3.11: Shared Memory* [Online] (Chapter 32). Available at https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html/developer_guide/dev-guide-shared-memory (Accessed 26 September 2020).

Redhat (2020b) *Introduction to Kubernetes architecture* [Online], Redhat. Available at https://www.redhat.com/en/topics/containers/kubernetes-architecture (Accessed 2 September 2020).

Redhat (2020c) *What is container orchestration?* [Online], Redhat. Available at https://www.redhat.com/en/topics/containers/what-is-container-orchestration (Accessed 31 August 2020).

Siemens AG (2009) *SIMATIC WinCC: Process visualization with Plant Intelligence* [Online], Siemens AG.

Signorini, F. (2019) *Complete guide — Deployment strategies on Kubernetes* [Online], ITNEXT. Available at https://itnext.io/application-deployment-strategies-on-kubernetes-e3f2f63e0f31 (Accessed 4 June 2020).

Sonar, S. (2020) *Updating Industrial Automation Software In Cloud-Native Environments GitHub Repository* [Online]. Available at https://github.com/shardulsonar/updating_industrial_automation_software_on_cloud_native_environments (Accessed 23 October 2020).

StarlingX (2020) *Solve the operational problem of deploying and managing distributed networks* [Online], StarlingX. Available at https://www.starlingx.io/collateral/StarlingX_OnePager_082019_Web.pdf (Accessed 21 September 2020).

Stringer, T. (2020) *Running kubectl Commands From Within a Pod* [Online], ITNEXT. Available at https://itnext.io/running-kubectl-commands-from-within-a-pod-b303e8176088 (Accessed 27 September 2020).

Sumo Logic (2020) *Docker Swarm* [Online], Sumo Logic. Available at https://www.sumologic.com/glossary/docker-swarm/ (Accessed 31 August 2020).

The Linux Foundation (2016) *Real-Time Linux Wiki* [Online], The Linux Foundation. Available at https://rt.wiki.kernel.org/index.php/Main_Page (Accessed 14 August 2020).

Thomson, J.R. (2015) *High integrity systems and safety management in hazardous industries* [Online], Butterworth-Heinemann. Available at https://www.elsevier.com/books/high-integrity-systems-and-safety-management-in-hazardous-industries/thomson/978-0-12-801996-2 (Accessed 16 September 2020).

Tremel, E. (2017) *Kubernetes Deployment Strategies* [Online], Sontainer Solutions. Available at https://blog.container-solutions.com/kubernetes-deployment-strategies (Accessed 4 June 2020).

Tron-Lozai, C. (2018) *Keep your Kubernetes cluster balanced: the secret to High Availability* [Online], ITNEXT. Available at https://itnext.io/keep-you-kubernetes-cluster-balanced-the-secret-to-high-availability-17edf60d9cb7 (Accessed 25 August 2020).

Truyen, E., Bruzek, M., Van Landuyt, D., Lagaisse, B. and Joosen, W. (2018) 'Evaluation of Container Orchestration Systems for Deploying and Managing NoSQL Database Clusters', *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 468–475.

Unofficial Kubernetes (2020) *Pods* [Online]. Available at https://unofficial-kubernetes.readthedocs.io/en/latest/concepts/workloads/pods/pod/ (Accessed 26 September 2020).

Vayghan, L. A., Saied, M. A., Toeroe, M. and Khendek, F. (2019) 'Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes', *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 176–185.

Venkataraman, A. and Jagadeesha, K. K. (2015) 'Evaluation of inter-process communication mechanisms', *Architecture*, vol. 86, p. 64.

Vyatkin, V. (2013) 'Software Engineering in Industrial Automation: State-of-the-Art Review', *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249.

Wahler, M., Gamer, T., Kumar, A. and Oriol, M. (2015) ‘FASA: A software architecture and runtime framework for flexible distributed automation systems’, *Journal of Systems Architecture*, vol. 61, no. 2, pp. 82–111 [Online]. DOI: 10.1016/j.sysarc.2015.01.002.

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

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

Weyer, S., Schmitt, M., Ohmer, M. and Gorecky, D. (2015) ‘Towards Industry 4.0-Standardization as the crucial challenge for highly modular, multi-vendor production systems’, *Ifac-Papersonline*, vol. 48, no. 3, pp. 579–584.

Wootton, B. (2018) *Stateless vs Stateful Containers: What's the Difference and Why Does It Matter?* [Online]. Available at https://www.contino.io/insights/stateless-vs-stateful-containers-whats-the-difference-and-why-does-it-matter (Accessed 4 September 2020).

Appendix

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: openplc
  labels:
    app: openplc
spec:
  replicas  : 1
  selector:
    matchLabels:
      app: openplc
  template:
    metadata:
      labels:
        app: openplc
    spec:
      containers:
        - name: openplc
          image: 'registry.local:9001/oasys/openplc:opcua-v3'
          imagePullPolicy: Always
          resources:
            limits:
              memory: 1Gi
            requests:
              memory: 1Gi
          env:
          - name: PLC_PROGRAM
            value: BoilerExample.st
          - name: OPENPLC_PORT
            value: '8080'
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      nodeSelector:
        kubernetes.io/hostname: controller-1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

YAML File 1 : Kubernetes Yaml file for OpenPLC Deployment

```yaml
kind: Service
apiVersion: v1
metadata:
  name: openplc
  labels:
    app: openplc
spec:
  ports:
    - name: openplc-ui
      protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 31544
    - name: opcua
      protocol: TCP
      port: 4840
      targetPort: 4840
      nodePort: 31550
  selector:
    app: openplc
  type: NodePort
```

YAML File 2 : Kubernetes yaml file for OpenPLC service

```yaml
kind: CloneSet
apiVersion: apps.kruise.io/v1alpha1
metadata:
  name: openplc
  labels:
    app: openplc
spec:
  replicas  : 1
  selector:
    matchLabels:
      app: openplc
  template:
    metadata:
      labels:
        app: openplc
    spec:
      containers:
        - name: openplc
          image: 'registry.local:9001/oasys/openplc:opcua-v3'
          imagePullPolicy: Always
          resources:
            limits:
              memory: 1Gi
            requests:
              memory: 1Gi
          env:
          - name: PLC_PROGRAM
            value: BoilerExample.st
          - name: OPENPLC_PORT
            value: '8080'
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
  updateStrategy:
    type: InPlaceOnly
    inPlaceUpdateStrategy:
      gracePeriodSeconds: 10
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

YAML File 3 : Kubernetes yaml file for OpenPLC with OpenKruise - CloneSet

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: openplc-same-node
  labels:
    app: openplc-same-node
spec:
  replicas  : 1
  selector:
    matchLabels:
      app: openplc-same-node
  template:
    metadata:
      labels:
        app: openplc-same-node
    spec:
      containers:
        - name: openplc
          image: 'registry.local:9001/oasys/openplc:opcua-v3'
          imagePullPolicy: Always
          resources:
            limits:
              memory: 1Gi
          env:
          - name: PLC_PROGRAM
            value: BoilerExample.st
          - name: OPENPLC_PORT
            value: '8080'
        - name: openplc-updated
          image: 'registry.local:9001/oasys/openplc:opcua-v3'
          imagePullPolicy: Always
          resources:
            limits:
              memory: 1Gi
          env:
          - name: PLC_PROGRAM
            value: BoilerExample.st
          - name: OPENPLC_OPCUA_PORT
            value: '4850'
          - name: OPENPLC_PORT
            value: '8090'
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

YAML File 4 : Kubernetes yaml file for running OpenPLC Deployment of 2 containers on same pod for IPC Transfer

```yaml
kind: Service
apiVersion: v1
metadata:
  name: openplc-same-node
  labels:
    app: openplc-same-node
spec:
  ports:
    - name: openplc-ui
      protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 31580
    - name: opcua
      protocol: TCP
      port: 4840
      targetPort: 4840
      nodePort: 31585
    - name: openplc-ui-updated
      protocol: TCP
      port: 8090
      targetPort: 8090
      nodePort: 31590
    - name: opcua-updated
      protocol: TCP
      port: 4850
      targetPort: 4850
      nodePort: 31595
  selector:
    app: openplc-same-node
  type: NodePort
```

YAML File 5 : Kubernetes yaml file for OpenPLC service with two containers on the same pod for IPC Via POSIX transfer use case

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: opcua-service
  labels:
    app: opcua-service
spec:
  replicas  : 1
  selector:
    matchLabels:
      app: opcua-service
  template:
    metadata:
      labels:
        app: opcua-service
    spec:
      containers:
        - name: opcua-service
          image: 'registry.local:9001/oasys/opcua-service:v1'
          imagePullPolicy: Always
          command:
            - build/client-openplc-service
          args:
            - '-s'
            - 'opc.tcp://io-simulator:4850'
            - '-c'
            - 'opc.tcp://openplc:4840'
            - '-a'
            - 'opc.tcp://io-simulator:4850'
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

YAML File 6 : Kubernetes yaml file OPC UA Service (or OpenPLC OPC UA Client) for connecting field devices and OPC UA server in OpenPLC

```yaml
kind: Service
apiVersion: v1
metadata:
  name: io-simulator
  labels:
    app: io-simulator
spec:
  ports:
    - protocol: TCP
      port: 4850
      targetPort: 4850
      nodePort: 30675
  selector:
    app: io-simulator
  type: NodePort
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: io-simulator
  labels:
    app: io-simulator
spec:
  replicas  : 1
  selector:
    matchLabels:
      app: io-simulator
  template:
    metadata:
      labels:
        app: io-simulator
    spec:
      containers:
        - name: open62541-io-simulator
          image: 'registry.local:9001/oasys/open62541-io-simulator:v1'
          imagePullPolicy: Always
          command:
            - build/boilerIOSimulator
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

YAML File 7 : Kubernetes Yaml File for IO Simulator

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: openplc
  labels:
    app: openplc
spec:
  replicas  : 1
  selector:
    matchLabels:
      app: openplc
  template:
    metadata:
      labels:
        app: openplc
    spec:
      containers:
        - name: openplc
          image: 'registry.local:9001/oasys/openplc:opcua-v3'
          imagePullPolicy: Always
          resources:
            limits:
              memory: 1Gi
            requests:
              memory: 1Gi
          env:
          - name: PLC_PROGRAM
            value: BoilerExample.st
          - name: OPENPLC_PORT
            value: '8080'
          lifecycle:
            preStop:
              exec:
                command: ["/bin/sh", "/OASYS/openplc/webserver/scripts/preStopScript.sh"]
      restartPolicy: Always
      terminationGracePeriodSeconds: 60
      dnsPolicy: ClusterFirst
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

YAML File 8 : Kubernetes yaml file for OpenPLC deployment with pre-stop lifecycle

```
#!/bin/bash

# Initialize wait for updated openplc pod to setup
sleep 40

# Get IP address
DESTINATION_IP=$(kubectl get pod -o wide | grep openplc | grep Running |awk '{print $6}')

# Execute State Transfer
/OASYS/LoadEvaluateGoService/build/loadEvaluateGo -o -l opc.tcp://localhost:4840 -
g opc.tcp://$DESTINATION_IP:4840 -f -t -r 1

# Connect sensor to plc

# Verify

# Delete old connection and connect actuator to plc
sleep 1
kubectl delete pod $POD_NAME --grace-period=0 --force
```

Shell Script 1 : Pre-Stop Script to state transfer and switch connections of OPC with field devices