Kubernetes Orchestration of High Availability Distributed Control Systems

Bjarne Johansson[1,2], Mats Rågberger[1], Thomas Nolte[2], Alessandro V. Papadopoulos[2]
[1] ABB, Västerås, Sweden, {bjarne.johansson, mats.ragberger}@se.abb.com
[2] Mälardalen University, Västerås, Sweden,{thomas.nolte, alessandro.papadopoulos}@mdu.se

2022 IEEE International Conference on Industrial Technology (ICIT) | 978-1-7281-1948-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICIT48603.2022.10002757

***Abstract*—Distributed control systems transform with the Industry 4.0 paradigm shift. A mesh-like, network-centric topology replaces the traditional controller-centered architecture, enforcing the interest of cloud-, fog-, and edge-computing, where lightweight container-based virtualization is a cornerstone. Kubernetes is a well-known container management system for container orchestration in cloud computing. It is gaining traction in edge- and fog-computing due to its elasticity and failure recovery properties. Orchestrator failure recovery can complement the manual replacement of a failed controller and, combined with controller redundancy, provide a pseudo-one-out-of-many redundancy. This paper investigates the failure recovery performance obtained from an out-of-the-box Kubernetes installation in a distributed control system scenario. We describe a Kubernetes based virtualized controller architecture and the software needed to set up a bare-metal cluster for control systems. Further, we deploy single and redundant configured containerized controllers based on an OPC UA compatible industry middleware software on the bare-metal cluster. The controllers expose variables with OPC UA PubSub. A script-based daemon introduces node failures, and a verification controller measures the downtime when using Kubernetes with an industry redundancy solution.**

I. Introduction

Distributed control systems (DCS) are large-scale control systems with multiple Distributed Controller Nodes (DCN) interconnected. A traditional DCN consists of dedicated hardware running the real-time controller firmware (FW). A high availability DCN is often achieved with hardware duplication – i.e., two DCNs, an active (primary) and a passive (backup). If the primary fails, the backup takes over the primary role, providing a one-out-of-two (1oo2) redundancy. The controlled process dictates the critical upper bound takeover time, which translates to around 500 ms for DCS in process automation [1]. Manual replacement of a failed DCN is required to restore redundancy.

The Industry 4.0 [2] data thirst drives DCS towards a network-centric architecture with an increased possibility of information and data retrieval. Figure 1 shows a simplified view of a traditional controller-centric system and network-centric system. The interconnectivity provided by a network-centric architecture allows data exchange between all devices connected to the network. Access to data produced near the process, i.e., the I/O, sensors, and actuators, does not need to involve the DCN.

This work is funded by the Knowledge Foundation (KKS), projects ARRAY and SACSys, by The Swedish Foundation for Strategic Research (SSF), project FuturAS, and by the Swedish Research Council (VR), project PSI.

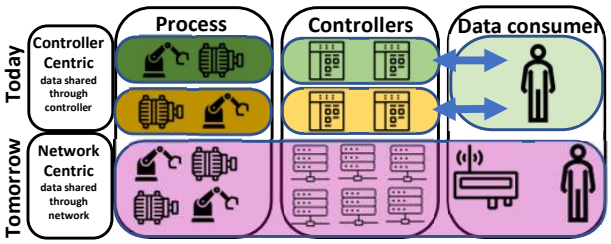

Fig. 1. A simplified view of a controller-centric and network-centric system.

Interconnectivity and interoperability are key concepts in the Open Process Automation™ Standard [1] (O-PAS). The O-PAS standard for DCN communication utilizes the OPC-UA [2] model making OPC UA suitable as communication means for our virtualized controller.

Virtualization is a cornerstone in realizing the computational elasticity provided by cloud-, fog-, and edge-computing. Containers are a lightweight and more performant virtualization alternative to Virtual Machines (VM) [3].

The widespread use of containers has led to container orchestration management systems such as Docker Swarm, Marathon on Mesos, and Kubernetes. The central functionality provided by the orchestrator is situation-aware scheduling and deployment of containers on the available resources.

We study the failure recovery properties provided by a vanilla out-of-the-box Kubernetes installation in a DCN context and the additional plugins needed to set up a bare-metal cluster hosting Virtualized DCN (VDCN). Kubernetes failure recovery, combined with 1oo2 VDCN redundancy, provides a pseudo-one-out-of-N (1ooN) VDCN redundancy and complements manual replacement of failed DCNs.

II. Related work

DCNs are embedded real-time systems, i.e., the temporal aspect of function output is as important as the output itself. Therefore, container performance is of primary concern. Struhár et al. [4] survey the usage of real-time containers and conclude that tool support, communication, and shared resources are open challenges. Even though challenges remain, ongoing research on real-time containers has been developed over the past few years [5], [6]. Felter et al. [3] show that the container overhead for CPU and memory utilization is negligible, but there can be a performance impact on I/O intensive applications. A similar conclusion is reached by Watada et

[1]https://publications.opengroup.org/p190
[2]https://opcfoundation.org/

al. [7], who also identify several challenges, i.e., persistent storage, complex networking, and orchestration management.

Fog computing addresses the inherent communication latency with geographic distant cloud computing by utilizing computational resources geographically closer [8], implying that the temporal aspect is vital in fog computing. Bellavista et al. [9] show that Single Board Computers (SBC), such as Raspberry Pis, are viable as fog computing nodes. They emphasize that providing real-time guarantees in a system with complex temporal utilization resource patterns is challenging. In a virtualized environment, resource inference can occur even if the utilized resources are different. Kim et al. [10] show that a network-bound application can saturate the CPU with `softirq` processing induced by the network communication.

Domain-specific container scheduling prerequisites have driven scheduling-related research. Eidenbenz et al. [11] evaluate three different Kubernetes scheduling integration alternatives to reduce communication latency and conclude that the Kubernetes native scheduling is the better alternative. Further, Eidenbenz et al. evaluate failover times, similar to our work, but just with Kubernetes native approach, and conclude that it is not fast enough. Vayghan et al. [12] propose an enhanced Kubernetes controller that gives a shorter downtime if a stateful application fails by having a redundant, passive instance ready to resume when told so by the controller.

Struhár et al. [6] introduce monitoring of real-time properties utilized by a Kubernetes scheduler extension to strengthen the temporal aspects. Großmann et al. [13] develop a resource utilization measurement tool with a small footprint. Using this tool, they compare the resource utilization between Kubernetes and Docker Swarm. Docker Swarm is less resource-demanding, but they also highlight that the comparison is not fair since Kubernetes provide more functionality.

A DCN in a network-centric context relies on network connectivity for various purposes such as communication with field devices, i.e., the network is fundamental. In a containerized context, the network is typically partly virtualized. Container Network Interface (CNI) and Container Network Model (CNM) are two specifications, with corresponding libraries, plugins, and interfaces that a container runtime can utilize to configure the network. Kubernetes supports CNI, and CNI is a *de-facto* standard [14]. There exist many CNI plugins, and researchers have studied the performance of some of them. Qi et al. [15] categorize a selection of CNI plugins in four categories and perform a benchmark, measuring throughput and Round-Trip Time (RTT). Depending on the plugin and the type of communication, the performance degradation ranges from a fraction of a percent up to 30% [14], [16], [15], [17].

As for automation-related controller virtualization research, Hegazy et al. [1] show that automation as a Service (AaaS) is feasible with a latency compensating control algorithm. However, recouping for the latency is impossible when an RTT shorter than communication time to the remote cloud is required, for example, a quick reaction to a discrete event. Goldschmidt et al. [18] presented and benchmarked a containerized controller architecture, concluding that the introduced overhead is insignificant.

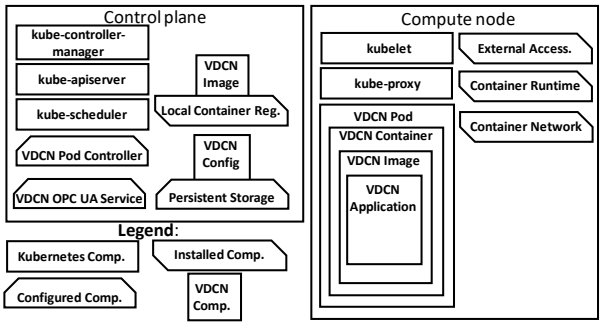

Fig. 2. A Kubernetes-based VDCN cluster architecture.

To the best of our knowledge, no related work combines orchestrator failure recovery with VDCN redundancy. The combination results in a pseudo-1ooN VDCN redundancy. Our contribution is the description of the components needed to realize a Kubernetes orchestrated cluster for hosting single and redundant VDCN combined with the measurement and evaluation of failure recovery performance.

III. System description

Docker is a well-known container runtime; examples of other alternatives are rkt[3] and LXC[4]. We select Docker as the container runtime, mainly due to its popularity and performance [3], [5], [18]. Rodriguez et al. [19] presented an overview of orchestration systems. We chose Kubernetes since it is relatively mature with a large open-source community. Marathon on Mesos is a relevant alternative due to its high-availability properties. We identify Marathon on Mesos and Kubernetes dependability evaluation as potential future work and focus solely on Kubernetes in this work.

A. Kubernetes components and architecture

Control plane is the name for the logical consolidation of the cluster control logic, i.e., the brain of the orchestrator. Compute node is the name for the nodes, physical or virtual, doing the actual work. Kubernetes offer a high-availability setup that prevents a single point of failure to bring down the control plane functionality. This work focuses on compute node failure and notes that evaluating control plane failure in a DCS context is relevant future work.

The central component in Kubernetes is the *Pod*. A Pod is a collection of one or more containers that are co-scheduled and co-located. Kubernetes do not schedule or deploy containers directly; Kubernetes operates on Pods.

The main components in the Kubernetes architecture, divided into control plane and compute node components, are shown in Figure 2, and briefly described below. Control plane components:

- `kube-apiserver`: the frontend of the cluster, all cluster interaction, including configuration, takes place through the `kube-apiserver`.

[3]https://github.com/rkt/rkt/
[4]https://linuxcontainers.org/

- `kube-scheduler`: assigns pods to nodes based on scheduling constraints and node resource availability.
- `kube-controller-manager`: control loops driving the actual state towards the desired state.

Compute node components:

- `kubelet`: Kubernetes node agent that monitors the node and the Pod deployed containers.
- `kube-proxy`: maintainer of node network rules, allowing inter-pod communication.

B. Kubernetes DCN cluster architecture

A VDCN Kubernetes cluster requires additional components, configuration of Kubernetes components, and VDCN specific components. Figure 2 show the architecture. We divide the components into three categories and give a short overview below and a more detailed description in Section IV.

The installed components are: (i) the Container Runtime (CR) and Local Container Registry (LCR), (ii) the Container Network, for intra-cluster container communication, and (iii) the External Access, for inter-cluster communication.

The configured Kubernetes objects are: (i) Persistent Storage, (ii) VDCN Pod, containing a container instantiated from the VDCN Image, (iii) VDCN Pod Controller, for managing VDCN Pod instances, and (iv) VDCN OPC UA Service, for OPC UA Server endpoint lookup amongst VDCN Pods, i.e., intra-cluster OPC UA traffic routing.

VDCN components include: (i) VDCN Application, i.e., the controller software, (ii) VDCN Image, containing the VDCN Application image, and (iii) VDCN Configuration, containing the specific VDCN configuration.

IV. Components

In this section, we describe the cluster components to provide a holistic view of a bare-metal cluster capable of hosting redundant and single VDCNs, that together with Kubernetes, constitute the VDCN cluster.

Docker runtime and registry: The Docker runtime pulls the container image from the LCR and starts the containerized process, the VDCN Application. Upon termination, the runtime cleans up the allocated resource.

The LCR serves as a cluster repository for container images, i.e., the VDCN images.

Container network: Container Network (CN) is the network that connects containers, intra-, and inter-node. The CN can be the physical network directly, set up with IP address routing and Ethernet switching, i.e., the underlay network, e.g., a traditional switched Ethernet network. A CN can also be a virtual network built upon the underlay using tunneling protocols such as VXLAN[5], i.e., an overlay network.

An example of an underlay network is a network created using the `macvlan` driver. The `macvlan` driver creates a virtual Ethernet interface, with an additional MAC address tied to a physical Ethernet interface. By making the virtual `macvlan` interface accessible from the container network namespace, the container gets access to the network.

A more common approach to allow the container to partake in network communication is to use a Virtual Ethernet Device (`veth`) adapter pair, `veth` are always created in pairs. One of the adapters resides in the container namespace and is therefore visible from the containerized application. The other `veth` lives in the host namespace. Together, the `veth` pair form a tunnel from the container namespace to the host namespace.

CNI Plugins are software components that comply with the CNI specification and provide a CN. For example, Flannel[6], Weave[7] and Cilium[8] create a VXLAN based overlay while Calico[9] and Kube-router[10] uses IP-in-IP[11] [15].

To summarize and relate CNI to Kubernetes context, each Pod has an IP address. The CNI plugins are responsible for providing the IP address and realizing the Pod-to-Pod communication within the cluster.

Qi et al. [15], evaluate the performance of Flannel, Cilium, Weave, Calico, and Kube-router and conclude that there is no all-around winner performance-wise. Cilium has the best intra-host performance, while Kube-router and Calico are more performant in inter-host communication.

The VDCN utilizes UDP multicast, described in the following. Searching the internet and available CNI-plugins project pages tells us that Calico has multicast support on the roadmap, but it is currently not implemented. Weave is the only plugin we found with multicast support; hence Weave is the plugin we use for the VDCN cluster.

External access: The CN setup the intra-cluster communication. For inter-cluster ingress traffic, Kubernetes provides three alternatives[12]:

- NodePort: expose the service on a statically allocated port on each node's IP. I.e., a node IP address combined with the static port is the externally exposed access point.
- Load balancer: load balances and directs the traffic to the service endpoint. The load balancer specifies the external access point, and typically the cloud provider provides the load balancer.
- External IPs: ingress traffic reaching a cluster node, on an IP address that matches the IP address specified in the Kubernetes external IP service specification, is routed to the service endpoint by Kubernetes Services.

A VDCN cluster use case where external access is needed is when an OPC UA Client outside the cluster requests services from a cluster VDCN OPC UA Server. The OPC UA Client should always reach the same VDCN on the same IP address, provided that the VDCN is available.

How feasible are the different alternatives for realizing the above? NodePort requires that the client outside the cluster re-connects to a new IP address in case of failure of the node

[5]https://tools.ietf.org/html/rfc7348

[6]https://github.com/flannel-io/flannel

[7]https://www.weave.works/oss/net/

[8]https://cilium.io/

[9]https://www.projectcalico.org/

[10]https://www.kube-router.io/

[11]https://tools.ietf.org/html/rfc1853

[12]https://kubernetes.io/docs/concepts/services-networking/service/

owning the IP address the client currently uses. NodePort also requires mapping between the original port and the port used for exposing the service. NodePort does not ensure that the OPC UA Client only needs to know one IP address per DCN. Hence, NodePort is not an alternative.

External IP is not per se managed by Kubernetes; the cluster administrator must ensure that the external IP address exists and routes to a node in the cluster. Kubernetes forwards cluster ingress traffic with a destination IP matching the external IP to the endpoint designated for the port. For example, external IP could ensure that an OPC UA Client only needs to know one IP address per VDCN; however, it would require the cluster administrator to set up a solution tolerant to node failures.

Load balancers, as mentioned, are typically provided by the cloud provider hosting the cluster. However, an on-site, bare-metal cluster does not necessarily utilize the cloud, and it is not desirable to route time-critical traffic through the cloud provider. Hence, the load balancer alternative requires a bare-metal load balancer.

We have been able to identify three bare-metal load balancers, MetalLB [13], PorterLB [14] and PureLB [15]. The selection and evaluation of the load balancer is a potential work on its own. For the work presented here, we conclude that using a load balancer with network redundancy capabilities and IP Address Management (IPAM) would make the load balancer alternative the better of the three presented alternatives. MetalLB provides both; hence the load balancer alternative with MetalLB as the bare metal load balancer is the one we use.

MetalLB supports two modes: (i) layer 2 mode, and (ii) Border Gateway Protocol (BGP) mode. In layer 2 mode, all incoming traffic pass through one of the cluster nodes kube-proxy, the leader node. From kube-proxy and onward, it is the internal Kubernetes service endpoint handling. MetalLB elects a new leader node if the leader node fails, and MetalLB will send gratuitous ARP packets, announcing that the IP address association changed to the MAC address of the new leader.

MetalLB BGP mode requires a router; MetalLB uses BGP to announce multiple routes, routes leading to different nodes in the cluster, i.e., the load balancing is the multipath handling in the router. When the traffic reaches the cluster node, the handling is the same as in layer 2 mode.

We use MetalLB in layer 2 mode, leaving load balancing related questions as possible future work. MetalLB provides IPAM, and the IP address managed are provided to MetalLB as an IP address pool. The specification of an externally accessible Kubernetes Service contains an IP address from the MetalLB IP address pool.

Persistent storage: Traditionally, memory on the DCN provides the DCNs persistent storage, for example, a non-volatile RAM or an SD card. The DCN stores the configuration, application, and current state in the persistent storage, which allows the DCN to resume operation after a failure, such as a power failure. A VDCN in a Kubernetes cluster is deployable on multiple nodes. Hence the persistent storage needs to be accessible from the nodes that host the VDCN.

Kubernetes provides the possibility to use various storage solutions. Volume is the Kubernetes term for file storage. A Volume, from a Kubernetes Pod perspective, is just a directory. Kubernetes do not care how that directory comes into existence. Setting up the storage is the cluster administrator's responsibility.

Kubernetes manages the lifetime of the Volume. There are two types of Volumes, Volume and Persistent Volume (PV). A Volumes lifespan is the same as the Pod', i.e., when the Pod ceases to exist, Kubernetes destroy the Volume. On the other hand, the PV lifespan is independent of the Pod. A Persistent Volume Claim (PVC) is the mean for a Pod to claim a PV. The PVC specifies the Pods requirements on the PV, such as size, access modes, etc.

In our experiment setup, we use a Network File System (NFS)[16] hosted on the control plane node to provide storage. The storage is not redundant – but that is not crucial for the evaluation since the control plane reschedules the Pods, i.e., no control plane, no Pod rescheduling.

VDCN Pod: The VDCN Pod is the Kubernetes Pod encapsulation of the VDCN Container. The VDCN Pod claims a PV using a PVC; the VDCN Pods in the test setup claim 100 MB that is read and writable. Due to multicast not being supported by MetalLB, the VDCN Pod has access to the node (host) network directly.

VDCN Pod controller: The VDCN Pod controller is the name we have given to denote the functionality we achieve by utilizing Kubernetes for controlling the VDCN Pods. An application running on a Kubernetes cluster is a workload, e.g., the VDCN is a workload. Workload resources are Kubernetes objects that specify the desired state for a Pod or Pods. Kubernetes controllers, executed in the context of the kube-controller-manager, strive to maintain the workload resource desired state.

A Kubernetes Deployment is a workload resource type for managing Pods. A Deployment strives to ensure that at least as many Pods as specified in the Deployment description are available in the cluster. In addition, if the Pods use PV, all the Pods created by the same Deployment share the same PV. Thus, Deployments are well suited for stateless applications.

Statefulset is another workload resource type that ensures that, at most, the number of Pods specified in the Statefulset description is available in the cluster. The Statefulset creates the Pods in a predetermined order with a known identity. If the Pods use PV, each Pod gets its PV.

The VDCN Pod Controller is the Kubernetes controller with a Statefulset describing the desired state. We use Statefulset as the workload resource since we want stricter control of the number of VDCN instances running than the Deployment can provide to avoid situations where two or more VDCN with the same identity are active but in different states.

[13]https://metallb.universe.tf/

[14]https://porterlb.io/

[15]https://gitlab.com/purelb/purelb

[16]https://tools.ietf.org/html/rfc3010

Our testbed cluster uses two separate VDCN Pod Controllers, i.e., Statefulsets, one for the single VDCN and one for the redundant. For the single VDCN, the number of Pods is one. The number of Pods in the redundant VDCN is two since the redundant VDCN is a pair. We use Pod anti-affinity to ensure that Kubernetes does not schedule both VDCN of the redundant pair on the same node.

VDCN OPC UA Service: Kubernetes Services is the front-end of a cluster-hosted application function. The containers running inside Pods are the *Service endpoint*. Pods' IP addresses and whereabouts are not static; they can change from one moment to another. Kubernetes Services is the mechanism to find the Pod that offers the Service for the requested function, independent of the current deployment. Kubernetes Services is the intra-cluster solution to find the endpoint. The kube-proxy handles the Service endpoint lookup on each node, watches the control plane for Service and endpoint updates through the kube-api, and updates the node iptables accordingly.

The VDCN has three network communication dependent functions: (i) the cyclic exchange of variables over OPC UA PubSub, (ii) acyclic communication using OPC UA Client Server, and (iii) the redundancy communication. OPC UA PubSub and the redundancy communication use UDP multicast and do not need a Kubernetes service. The network IGMP support provides the means to match publishers with subscribers.

OPC UA Connection Protocol (UACP) is the abstract protocol that describes the full-duplex communication channel between client and server. OPC UA supports TCP, HTTPS, and WebSocket as the UACP underlying transport protocols, and the VDCN OPC UA Client-Server uses TCP. In other words, the OPC UA Client-Server communication is unicast-based, point-to-point.

A request addressed to a VDCN OPC UA Server can originate from an OPC UA client inside or outside the cluster. The external handling described above ensures that the request reaches a cluster node. When the request has reached a cluster node, the Kubernetes Service handling provides the endpoint reaching means.

Our example setup consists of three VDCNs, the single configured and the redundant pair. We use two VDCN OPC UA Services, one for the single VDCN and one for the primary VDCN. The redundancy state of the redundant VDCN is application-specific. To allow Kubernetes to redirect the traffic to VDCN in primary mode, we need Kubernetes to update the routes depending on the application state. A Kubernetes mechanism for that is the probes, probes that probe the application's state. The application tailors the application end of the probe for its need.

Kubernetes provides three types of probes. The Liveness-probe determines if the application is responsive (alive) or not. If not, Kubernetes can restart the container. The Startup-probe tells Kubernetes that the container application has started, and the Readiness-probe tells if the container application is ready to accept traffic. If the Readiness-probe result is negative, the probed application is removed from the list of potential service endpoints. The VDCN Application uses the readiness probe to direct traffic to the primary VDCN in the redundant VDCN configuration; the backup VDCN Application reply negatively to Kubernetes Readiness-probe requests.

VDCN Application: In a traditional DCN, the VDCN Application is the FW capable of executing the control loop logic. The VDCN Application used in our testbed is an ABB proprietary software, i.e., a modern DCN FW. It consists of three main parts, an OPC UA stack for industrial use, a middleware, and the control loop logic. The OPC UA stack provides the OPC UA communication means, and the middleware offers functionality to the control logic. The middleware functionality relevant for this testbed is redundancy-related. Finally, the control logic in the VDCN Application consists of a cyclic task with a configurable interval time. The cyclic task updates configured variables each iteration and exposes the updated variables externally using OPC UA PubSub.

In addition to the cyclic OPC UA PubSub communication, the VDCN Application also contains means for OPC UA Client-Server request-based acyclic communication. An OPC UA Server is the VDCN Application side of the request-based, acyclic OPC UA communication, exposing Remote Procedure Calls (RPC) callable from an OPC UA Client.

As the name implies, OPC UA PubSub is a publisher-subscriber-based solution. The publishers do not directly connect to the subscribers, and vice versa. Two models are supported, broker-based and broker-less. A broker-based publisher sends messages to a central broker from which subscribers subscribe. Two concrete broker-based solutions are supported, Message Queue Telemetry Transport (MQTT) and Advanced Message Queuing Protocol (AMQP). The broker-less model relies on properties provided by the network, specifically multicast and broadcast possibilities. UDP multicast is the supported realization of the broker-less model. A network infrastructure supporting IGMP ensure that published message only is forwarded to the subscribers. Network infrastructure without IGMP support broadcast the messages, i.e., published messages reach the whole broadcast domain. The VDCN application uses the broker-less OPC UA PubSub model realized with UDP multicast.

The OPC UA PubSub publishing and subscribing function run in a task of its own - unsynchronized with the producer/consumer of the exchanged variable values. Figure 3 shows a conceptual view of the data flow between the tasks.

The VDCN Application redundancy mode is configurable as single or redundant. In single mode, there is no backup ready to resume operation in case of failure. The single configured

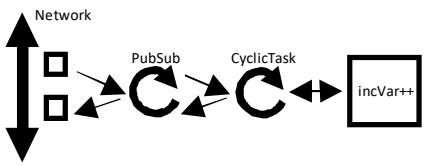

Fig. 3. The VDCN Application is involved in the cyclic exchange.

VDCN Application stores the dynamic state (variable values etc.) on a file located in the PV, allowing a re-deployed single VDCN to resume operation from the last stored state.

The redundant VDCN Application runs in a one-out-of-two (1oo2) setup. One VDCN Application is active, publishing updated variable values using OPC UA PubSub, and the other is passive, ready to resume operation in case of failure of the active. We refer to the active as the primary and the passive as the backup.

Two mechanisms are fundamental in a redundant setup where only one is active, the state transfer and the failure detection. The state transfer provides the backup with the primary's latest checkpointed dynamic state, allowing a backup to resume the role as primary, without historic signal values outputted. The VDCN application utilizes an ABB proprietary state transfer mechanism based on UDP multicast. Heartbeat Bully [20] over UDP multicast constitutes the failure detection and role selection mechanism.

VDCN Image: The container image. When instantiated by the container runtime, the VDCN image of the VDCN Application becomes the VDCN. The VDCN image is built with Docker and pushed to the LCR.

VDCN Configuration: The VDCNs are configurable, and VDCN Pod PV holds the configuration files, ensuring that they are accessible from each node that hosts the VDCN. Section V describes the specific configuration used in the test setup, such as task cycle times and the variables exchanged.

V. Execution and Result

The purpose is to measure the failure recovery time of single and redundant VDCNs, deployed in VDCN Pods orchestrated by Kubernetes. First, we let Kubernetes deploy the VDCN Pods on the cluster compute nodes while bringing down the nodes hosting the primary or single VDCN after a random time. Then, Kubernetes failure detection and rescheduling re-deploy the VDCN affected by the node failure. In the redundant VDCN case, the backup VDCN resumes operation as primary, while Kubernetes re-deploy a new backup VDCN. A Verification DCN sample the signal values and gather statistic related to the cyclic exchanged variables, see Section V-A3, i.e., it checks the cyclic communication. The Verification OPC UA Client test the acyclic communication, see Section V-A4.

A. Testbed

Four main parts constitute the testbed, the cluster, a failure daemon, a cyclic communication verification node (Verificator DCN), and an acyclic verification client (OPC UA Client Verificator). Figure 4 shows the testbed deployment and Table I list the used software.

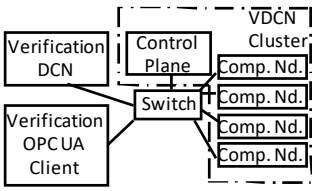

Fig. 4. Testbed deployment.

TABLE I
Software used.

Name	Version	Comment
Ubuntu Server	20.04	Control plane OS
Raspberry Pi OS	10	Compute node OS
PREEMPT_RT	4.4	Compute node kernel patch
KubeAdm	1.21	Kubernetes installer
Kubernetes	1.21	Kubernetes version
Docker	20.10.1	Container runtime
Weave	2.8.1	CNI plugin
MetalLb	0.9.6	Bare-metal load balancer

1) Cluster: The compute nodes in the testbed consist of four Raspberry Pi 4B, with four GB RAM. The control plane runs on a 2GHz Intel I7 I7-9700T PC, with 16 GB RAM.

2) Failure daemon: The failure daemon is a `systemd` daemon installed on all compute nodes that check if a VDCN Pod runs on the node. If it does, and the VDCN Application is in single or primary mode, the failure daemon shuts down the node after a random time of 5 minutes, $\pm$10s.

3) Verification DCN: The Verification DCN hardware is a 2GHz Intel I7 I7-9700T PC, with 16 GB RAM, running VxWorks 7.0 and the verification application (VA). The VA checks VDCN output values and measures the time between updates. The VA is the same application as the VDCN Application but with a different cycle time configuration.

4) Verification OPC UA Client: The Verification OPC UA Client runs on a Windows 10 PC. Every 10s, it establishes a connection to the VDCN OPC UA servers, one to the primary and one to the single VDCN, measuring the time between two successful connection attempts.

B. Exchanged variables

The variables published by the VDCN and monitored by the VA in the Verification DCN are: (i) $incVar$, 32-bit unsigned integer, incremented by the VDCN each iteration, and (ii) $nodeId$, a string identity of the node currently hosting the specific VDCN. We expect application size to affect a VDCN similar to a DCN since container overhead is neglectable [3]. Furthermore, we deploy one VDCN per node. Hence variable handling and communication come down to resource utilization and prioritization within the VDCN and node, and the critical VDCN task has real-time priority. Ensuring real-time properties when running multiple VDCN in the context of containers/pods on one node, with more extensive use of virtualized networks, is future work.

C. Task interval and updating period

We base the calculation on the simplification of the VDCN Application shown in Figure 3. In reality, the span will be larger due to task interleaving patterns with other high-priority tasks in the VDCN Application. The PubSub and Cyclic task in the VDCN Application has 5 and 20ms cycles. The corresponding cycle time in the VA is 1ms for both.

With the cycle times above, the VDCN publishing interval of updated variable values is in the range $PubUpdIntv \in (20, 25)$ms. The interval in which the Cyclic task in the VA can receive and check variable values is $VaCheckIntv \in$

$(1, 2)$ms. We denote the VA measured update interval of $incVar$, $VaUpdIntv = PubUpdIntv \pm VaCheckIntv$, thus being in the range $VaUpdIntv \in (18, 27)$ms. In relation to interval times used, the network propagation time is deemed negligible and not included.

The redundant VDCN failure detection mechanism is Heartbeat Bully [20]. The backup VDCN expects heartbeat messages regularly from the primary VDCN. The time between primary VDCN failure to the backup VDCN resuming the primary role depends on the heartbeat interval and the number of missing heartbeats allowed. The heartbeat interval used is 10ms, and the number of missed heartbeats allowed is two. The VDCN Application polls the heartbeats receiving status, adding a heartbeat interval to the detection time, resulting in the primary failure detection time range $PriFailDetTime \in (30, 50)$ms. On top of that, the VDCN has a resume primary role functionality with an execution time interval of $BePriExec \in (12, 34)$ms that contributes to the total failover time. Resulting in failover time range $FoT \in (42, 84)$ms.

The $UpdIntvFail$ is the longest time between VA observed updates of $incVar$ when a failover happens we have that:

$$UpdIntvFail = 2 \cdot PubUpdIntv + FoT \pm VaCheckIntv$$

resulting in the range $UpdIntvFail \in (80, 136)$ms.

D. Kubernetes settings

Kubernetes reschedule Pods on failure and the failure detection and reaction time are configurable, and those setting impacts the single configured VDCN $UpdIntvFail$ and redundant VDCN timeframe without a backup. The Kubernetes `kubelet` monitors the Pod and updates the status of the Pod and node to the `kube-apiserver` on the control plane. By default, the `kubelet` reports the node status every 10s, and the `kube-apiserver` has a grace period of forty seconds before setting the node status to not-available.

The default Pod eviction timeout is five minutes, and when a node failure is detected, Kubernetes reschedules the Pod after the eviction timeout expiration. Statefulsets are used for Pod management, i.e., the VDCN Pod controller. We use a Pod eviction timeout of 3s, specified in the Statefulset specification. A VDCN Pod hosted in a failing node is rescheduled after 43s; hence $UpdIntvFail$ for the single VDCN is higher than 43s.

When a node running stateful Pods fails, Kubernetes do not schedule a new stateful Pod by default. Since missing updates from that node's `kubelet` could be a consequence of network partitioning. Kubernetes cannot be confident that the node and Pod are gone. Hence, Kubernetes requires the cluster administrator to delete the failed node stateful Pods manually in a node failing situation.

We assume that the network is redundant and reliable with minimal risk of network partitioning. Even though our testbed network is not redundant, the network on an actual site is likely to be. Hence, we configure Kubernetes to reschedule stateful Pods by setting their termination grace period to 0s.

TABLE II
MEASURED INTERUPDATE TIME.

Mode	Primary				Single			
	Min	Max	Avg	SD	Min	Max	Avg	SD
Normal	3.0ms	36ms	20ms	1.3ms	3.0ms	224 ms	20ms	1.8ms
Failure	83ms	129ms	110ms	10.5ms	43.5s	74.7s	55.4s	8.1s

E. Result

The test ran for ten hours and accumulated 200 node failures, one hundred failures each on primary and single VDCNs hosting nodes and 3.6 million interupdate measures without node failures in between.

Table II shows the interupdate times per VDCN and mode. The Normal mode row shows the interupdate time measured during the failure-free periods and the Failure mode row when two updates span a node failure of the node hosting the VDCN publishing the variable. The Normal interupdate time is a reference for update time in a normal situation. The measured interupdate (or update interval) times without failure are on average 20ms, and that is in the expected range $VaUpdIntv$ from Section V-C. The single VDCN interupdate time standard deviation is higher than from the primary. The NFS-based PV state storage induces a longer execution time for the single VDCN. Max and min interupdate measured from the primary are outside the theoretical limit due to task interleaving patterns in the VDCN Application used that we do not address in the theoretical simplification. The SD tells that the vast majority is within the expected interval.

The interupdate time during a primary node failure, i.e., the failure recovery time, is within the expected range with an average of 110ms. As a comparison, the Kubernetes controller-based recovery time archived by Vayghan et al. [12] is in the range of 1.5 seconds. The single VDCN minimum interupdate time during node failure is 43.5 seconds, reached when the Pod is scheduled on the same node again. That can happen since the failure injection is a reboot. After the reboot, the node is failure-free again. In that case, the container image is not pulled from the LCR, reducing the time. The max of 74.7s includes pulling the image from the LCR.

OPC UA Client connection reestablishment to the new primary VDCN took a max of 41 seconds with an average of 40 seconds, which is the pod eviction time. For the single VDCN, the connection reestablishment times are the same as the failure interupdate times, which is feasible since that is the time it took Kubernetes to replace the failed single VDCN.

F. Availability discussion

We base the replacement scenarios in this section on input from experienced engineers working close to end-users. When a DCN fails today, it needs manual replacement and the replacement time depends on the situation. The best circumstance is when the failed DCN is close to both a spare DCN and maintenance personal that can exchange the broken DCN. In that case, a replacement within an hour is optimistic but realistic. A less favorable scenario is a failure in a remote location. It could take time for maintenance personnel to reach the site, causing repair time to range from several hours to

many days. With an orchestrated VDCN cluster, the orchestrator could redeploy the failed VDCN on compute nodes with enough available capacity and reduce the replacement time.

A commonly used measure in reliability context is Mean Time Between Failure (MTBF), a statistical measure of the probability of failure within a specific period. DCN MTBF depends on the hardware components used, temperature, and more. Based on public product information[17,18], we use a mid-range MTBF approximation of twenty years for the DCN and compute node hardware in the following discussion. Depending on the product and equipment type, it can be higher and lower.

Availability is the proportion of time that a system is available, often denoted in the number nines in the availability percentage; for example, 99.99% has four nines. A DCN replacement mean-time, Mean Time To Repair (MTTR) of one hour with MTBF of twenty years translates to an availability of five nines and an MTTR of twenty-four hours to four nines.

The average time for a redeployment of VDCN upon failure on the compute node currently hosting it is 55 seconds, see Table II, which translates to an availability of seven nines for the single configured VDCN. A four nines availability level gives a yearly downtime of roughly 52 minutes, five nines and seven nines correspond to 5 minutes and 3 seconds downtime respectively and annual.

A redundant DCN has high availability, with the twenty years MTBF and the average takeover time from Table II the availability level is nine nines. The orchestration benefit is the automated and quicker backup VDCN return, resulting in a pseudo-1ooN redundancy, with N being the number of compute nodes in the cluster capable of hosting a redundant configured VDCN.

An orchestration benefit is the increased availability that follows the quicker replacement of a failed DCN. Other potential benefits are flexible maintenance. For example, a VDCN can be moved to another compute node while upgrading the base software of the former. Even if one compute node fails, there might still be enough spare capacity to counter further failures, allowing process operation until the next scheduled maintenance with high confidence in the availability.

VI. Conclusion and Future work

By describing the components needed when setting up a bare-metal Kubernetes cluster for VDCN, we provide a holistic view of the system and show the multitude of component alternatives available. We created a testbed consisting of compute nodes, on which we deployed two VDCN configurations, a single and a redundant. Outside the cluster, we had a DCN verifying the VDCN OPC UA PubSub published variables and Windows application on a PC confirming the OPC UA client-server communication.

The result shows that Kubernetes hosted VDCN are feasible for both single and redundant VDCN. The measured redeployment of the single VDCN is too long to be a redundancy replacement. As stated in Section I takeover time below 500 ms can be needed for DCS in process automation. Nevertheless, it can still serve as a faster alternative to a manual replacement of a failed node. If shorter downtimes are required, a 1oo2 setup is feasible, where Kubernetes also ensure a quicker reinstatement of a backup VDCN than manual replacement, resulting in a pseudo-1ooN VDCN redundancy.

Kubernetes provide much more extensive customization alternatives than we have utilized. Further work could evaluate the feasibility of further customization of Kubernetes for VDCNs, such as the faster discovery of failed nodes. Kubernetes scheduling in the DCS context is another natural extension of this work, for example, finding and deploying the VDCN to a suitable node capable of fulfilling the dependability needs dictated by the VDCN.

We showed the plurality of different network virtualization alternatives. Reliable, deterministic communication is essential for DCS. VDCN in a containerized Kubernetes managed context relies on virtualized network functions provided by CNI plugins and load balancers. Ensuring dependability when utilizing virtualized network is a future challenge, especially when sharing the underlying resources between VDCN or other entities on the compute node.

[17]https://search.abb.com/library/Download.aspx?DocumentID=3BSE091397&Action=Launch

[18]https://support.industry.siemens.com/cs/attachments/16818490/mtbf.zip

References

[1] T. Hegazy *et al.*, "Industrial automation as a cloud service," *IEEE Trans. Par. and Distr. Syst.*, vol. 26, no. 10, pp. 2750–2763, 2015.

[2] R. Drath *et al.*, "Industrie 4.0: Hit or hype? [industry forum]," *IEEE Industrial Electronics Magazine*, vol. 8, no. 2, pp. 56–58, 2014.

[3] W. Felter *et al.*, "An updated performance comparison of virtual machines and linux containers," in *IEEE Int. Symp. Perf. Analysis of Syst. and Software*, 2015, pp. 171–172.

[4] V. Struhár *et al.*, "Real-Time Containers: A Survey," in *Fog-IoT*, 2020.

[5] A. Moga *et al.*, "Os-level virtualization for industrial automation systems: Are we there yet?" in *SAC*, 2016.

[6] V. Struhár *et al.*, "React: Enabling real-time container orchestration," in *ETFA*, 2021.

[7] J. Watada *et al.*, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, 2019.

[8] F. Bonomi *et al.*, "Fog computing and its role in the internet of things," in *Mobile Cloud Comp.*, 2012, pp. 13–16.

[9] P. Bellavista *et al.*, "Feasibility of fog computing deployment based on docker containerization over raspberrypi," in *ICDCN*, 2017.

[10] E. Kim *et al.*, "On the resource management of kubernetes," in *ICOIN*, 2021, pp. 154–158.

[11] R. Eidenbenz *et al.*, "Latency-aware industrial fog application orchestration with kubernetes," in *FMEC*, 2020, pp. 164–171.

[12] L. A. Vayghan *et al.*, "A kubernetes controller for managing the availability of elastic microservice based stateful applications," *J. Syst. and Soft.*, vol. 175, pp. 110 924–, 2021.

[13] M. Großmann *et al.*, "Monitoring container services at the network edge," in *ITC*, 2017, pp. 130–133.

[14] J. Yoon *et al.*, "A measurement study on evaluating container network performance for edge computing," in *APNOMS*, 2020, pp. 345–348.

[15] S. Qi *et al.*, "Understanding container network interface plugins: Design considerations and performance," in *LANMAN*, 2020, pp. 1–6.

[16] N. Kapočius, "Performance studies of kubernetes network solutions," in *eStream*, 2020, pp. 1–6.

[17] H. Zeng *et al.*, "Measurement and evaluation for docker container networking," in *CyberC*, 2017, pp. 105–108.

[18] T. Goldschmidt *et al.*, "Container-based architecture for flexible industrial control applications," *J. Syst. Arch.*, vol. 84, 2018.

[19] M. A. Rodriguez *et al.*, "Container-based cluster orchestration systems: A taxonomy and future directions," *Soft., Pract. & Exp.*, 2019.

[20] B. Johansson *et al.*, "Heartbeat bully: Failure detection and redundancy role selection for network-centric controller," in *IECON*, 2020.