Fault-tolerant Permanent Storage for Container-based Fog Architectures

Zeinab Bakhshi, Guillermo Rodriguez-Navas, Hans Hansson
Mälardalen University, Sweden, {zeinab.bakhshi, guillermo.rodriguez-navas, hans.hansson}@mdh.se

***Abstract*— Container-based architectures are widely used for cloud computing and can have an important role in the implementation of fog computing infrastructures. However, there are some crucial dependability aspects that must be addressed to make containerization suitable for critical fog applications, e.g., in automation and robotics. This paper discusses challenges in applying containerization at the fog layer, and focuses on one of those challenges: provision of fault-tolerant permanent storage. The paper also presents a container-based fog architecture utilizing so-called storage containers, which combine built-in fault-tolerance mechanisms of containers with a distributed consensus protocol to achieve data consistency.**

I. INTRODUCTION

Fog computing aims to deploy distributed applications closer to the edge of the system, and can bring important benefits for time-sensitive applications.

Containerization is a novel technique to implement efficient and adaptive distributed applications, which is widely used in Cloud computing [1]. Containers are also very appealing for fog computing because of their flexibility and self-healing mechanisms. However, the suitability for critical applications has not been investigated in enough detail. We are interested in understanding to what extent container-based fog architectures can be applied to cyberphysical systems, and to identify the main shortcomings with respect to dependability, and to propose techniques to overcome these shortcoming.

In this paper we discuss the integration of a ROS robotics application [2] with a container-based architecture. We perform a preliminary study of the system's behavior in the presence of faults and identify the benefits and the limitations of integrating ROS and containerization at the fog layer. We also build a Fault Tree Analysis to find the critical points of failures. As a result, we identify a number of important limitations of existing container-based virtualization architectures, including the lack of persistent storage for *stateful* applications at the fog layer. To overcome this limitation, we propose using storage containers to provide persistent storage at the edge of the network. Additionally, we suggest a series of improvements, in the form of new services or adaptation of existing services, in order to build a dependable containerized fog architecture.

Our main contributions are (1) that we identify key limitations of directly applying existing containerization and orchestration solutions in the resource constrained fog layer (presented in Section II), and (2) that we propose remedies to address related dependability challenges by proposing a new architecture inspired by Kubernetes (Section IV), and a persistent storage solution using storage containers, addressing one of the identified challenges (Section V).

The remainder of this paper is structured as follows: We study an integration of containerized robotic application and analyze the behaviour of a container orchestration solution in presence of faults at the fog layer in Section II. We continue with a description of the main problem adapting container orchestration at the fog layer for robotic applications in Section III. In Section IV, we propose a new container orchestration architecture inspired by Kubernetes. Persistent storage solution for container-based architecture is presented in Section V. Related work to our study are briefly reviewed in Section VI. Finally, we conclude the study in Section VII.

II. FAULT ANALYSIS OF A CONTAINERIZED ROBOTIC APPLICATION AT THE FOG LAYER

Our study of container-based architecture for fog computing is based on a use case that integrates the Robot Operating System (ROS) [3], Docker Containers [4], and Kubernetes [5]. We use a robotic application which is developed in ROS and containerize it using docker containers and then deploy it at the edge of the network. (The files and instructions to access the use case are uploaded in a github repository)[1]. The implemented robotic application is a navigator robot that constantly moves towards newly set goals, while avoiding obstacles. The behavior is illustrated in Fig. 1, the robot first aims to reach Goal 1, while it should avoid the obstacles. It plans the optimum path to the goal. This process is repeated every time the robot reaches the goal and a new goal is set for it, as shown in Fig. 1B and 1C.

A. ROS and Kubernetes Design

Our ROS application consists of several ROS nodes. Nodes in ROS are processes that perform computation [2]. We design our ROS application to be decomposed into three different containers: (A) the ROS Core application; (B) the navigation application, containing ROS nodes that process the parts related to navigation and map; and (C) the Simulator application, containing nodes related to simulation.

To containerise the ROS application, we use so-called *dockerized images* of ROS from Open Source Robotic Foundation (OSRF) [6], which simplifies the installation and setup of the ROS system. We built a Kubernetes cluster (Version 1.16) in our local network without using any cloud platform. The cluster we implemented has three nodes. One Kubernetes master node and two worker nodes. The Master node and

[1] https://github.com/ZeinabBa/Fog-Robotic

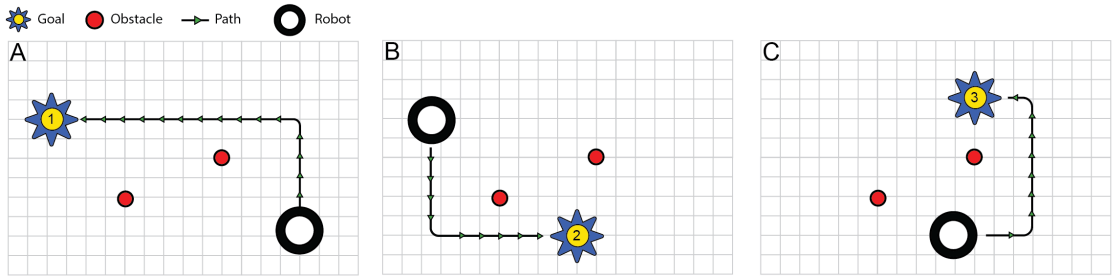

Fig. 1. How the Robot works

one worker node are running Linux Ubuntu 18.04. The other node is a Raspberry Pi (RPi) running Rasbian OS, which is a Linux based OS.

B. Fault model

The described robotic system might be affected by faults that in turn might disrupt the whole navigation application. In our work, we are going to consider the taxonomy of faults discussed by Parker [7] but we will review the system in light of the built-in fault tolerance mechanisms provided by containers. In particular, containerization already tolerates configuration and task completion faults, both mentioned in [7], by means of self-healing mechanisms; notably the ability to restart failed containers. To investigate and analyze different points of failure we built the Fault Tree shown in Fig. 2. Since the number of states in which failure of one or multiple components results in service failure was far more than the states in which the desired service is delivered, and due to the intricacy of Kubernetes design, we first depict the points in which the system will function as desired and then we use a logic "NOT" gate to show system failure states. As shown in Fig. 2, we can see that Volume has an important role in delivering a correct service; particularly for applications that exhibit dependencies, require access to files or are stateful (i.e. should keep memory of the current state).

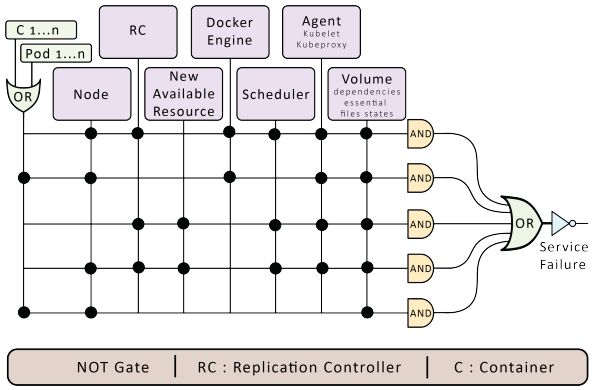

Fig. 2. Fault Tree Analysis

C. Fault Experiment Scenarios

We evaluate the impact of the failures related to orchestration of containerized robot applications over our use case, which is a containerized robotic application deployed on Kubernetes nodes. We inject faults to generate the following failures: 1) Application Execution failure, 2) Kubernetes Node failure, 3) Application deployment failure, and 4) File access failure.

Application Execution Failure: To test the impact of application failure using Kubernetes and evaluate how Kuberenetes can tolerate this type of failures, we killed (terminated) the Pod containing the core ROS application container. We observed that the Pod was automatically recreated after injecting this intentional fault and terminating the Pod. The whole navigation process continued working.

Kubernetes Node Failure: To investigate the impact of Kubernetes node failure, we disconnected the Kubernetes node in which the containerized robot application was executing, to check how Kubernetes would respond to this type of failure. We first deployed the Pods to the RPi and then joined the other worker node (Linux device) to the cluster. To ensure that at least one instance of the whole application (three containers of the ROS application) will work together at the same time, we unplugged the RPi node to inject the node failure and automatically the Pods were recreated and deployed in the other available node, the Linux device.

Application Deployment Failure: Deploying an application or containerized robot application in our scenario means assigning available resources to the containers for execution. There are different possible causes that a system fails to deploy an application, for instance, when there are no available resources, losing communication to available nodes, or improper resource management. We study how Kubernetes does the resource management and if it is effective for preventing application deployment failure. To test this, we rejoined the nodes to check when both the nodes are available, on which node the Pods will be automatically deployed. The experiment showed that the Kubernetes scheduler will select the node which has the maximum resources at deployment. Therefore the Linux device was selected. This means that, when an application is ready to get deployed and all the nodes are free then it will be assigned to the node with the maximum capacity. If, later on, another application with higher demand wants to be executed on the system, it is possible that suitable resources are not available for deployment. Although this is not a failure *per se*, it potentially decreases service availability due to lack of load balancing and resource management of the system.

File Access Failure: ROS applications require access to

files, states and other necessary data to continue working. In Kubernetes, files are stored in volumes of each Pod. When an application needs to access a file, it directly accesses the files from the volume inside the Pod. However, since the Pods are mortal, the volume inside the Pods are also volatile (ephemeral). This is because Pod Volumes transiently store the files if their local system spaces (not mounted to clouds). Therefore, data and information of the applications is not accessible once the Pod is terminated. In a pure ROS system implementation, there is a mechanism named *Parameter server* [8], through which system parameters are saved for further references and decision makings upon failure. Parameter server is always implemented in the ROS master node. When a ROS master node is containerized it shares its state, files and data in the Pod volume. When a Pod is terminated, its volume will no longer exist. Therefore, the parameter server would not be accessible for fault recovery or reconfiguration purposes when a ROS application fails.

III. Problem Statement

Our case study provides us with two interesting findings, (1) Docker and Kubernetes are suitable as a distributed platform for execution of ROS applications; (2) by injecting faults to our use-case we identified that using the Docker and Kubernetes platform for ROS applications suffers from limitations that are threats to dependability as discussed in Section II-C.

Putting the experiment beside the fault tree analysis we find that the most important issue that requires further attention in our use case is lack of persistent storage for stateful applications. Kubernetes has already implemented support for stateful application by providing Persistent Volume (PV) and stateful set to improve relaiability [5]. However, Vayghan et al. [9] identified two main issues with the stateful set and PV. First, when a pod fails, recreation time plus access time to PV may be too large for applications requiring high availability. Second, node failure cannot be recovered in stateful set, hence if a node fails in this type of deployment (stateful set) then the pod will not be recreated until the node is back in the network. It is possible to manually recreate and redeploy the pods suffering from node failure but the access to the PV will be lost in this case.

In addition, PV can only be implemented using cloud storage and our interest is to investigate what can be done at the network edge in the fog layer to reduce data latency and provide immediate response to data access requests.

As a remedy, we take advantage of the lightweight, self-healing characteristics of containers and propose container-based storage applications which can implement a distributed, fault-tolerant and persistent storage system. However, to fully realize the potential of the storage container that we are going to propose, we need to extend Kubernetes error-handling capabilities. Therefore, we first propose an improved fault-tolerant container orchestrator architecture, inspired by Kubernetes, and then we explain how the storage system we propose can fit into this new architecture.

IV. A container-based architecture for fog

In this section we propose an architecture that extends Kubernetes to achieve better dependability properties by delivering the following services: (a) Configuration and Application Deployment, (b) Monitoring, (c) Failure Detection and Recovery, (d) Communication, and (e) Persistent Storage. These services are delivered using different components in the proposed architecture, illustrated in Fig. 3.

A cluster in our architecture consists of nodes, called *Fog node* or *Node*, with each node having the following attributes:

$$Node_i = \{CPU, RAM, Storage, Cost, App\_List\_on\_Node, Status, Desired\_Status\}$$

Thus specifying the resources of the fog node as *CPU*, *RAM* and *Storage* capacities, the cost of execution on the node as *Cost*, the list of applications currently running on the node as *App_List_on_Node*, current status of a node as *Status* and the desired status of a node as *Desired_Status*. Both, Status and Desired_Status, are used for checking whether a node is up and running.

Fog nodes host containerized applications, which are called *application* or *App* in our architecture.

Let $A = \bigcup_{a=1}^{n} App_a$ be a set of applications to be deployed in fog nodes. Each application is characterised by its specification:

$$App_a = \{CPU, RAM, Storage, Data, Storage\_Tag, Run, Label, Status, Desired\_Status\}$$

These specifications indicate the resources the application requires as *CPU*, *RAM* and *Storage* capacities, the data and states of an application as *Data*, the persistent storage requirement of an application as *Storage_tag*, where if an application is stateful then its storage_tag will be equal to *persistent*. It also indicates the run time of an application as *Run* and the label of the applications as *Label*, where the latter identifies the type of the application and is used for grouping containerized applications. Finally, the specification indicates the current status of the application as *Status* and the desired status of the application as *Desired_Status*. Status and Desired_Status of the applications are used to check whether the application execution process has its desired status. We also define storage containers (SC) for delivering persistent storage for stateful applications as $SC = \bigcup_{j=1}^{n} SC_j$. We explain the detailed specifications that characterise each SC in Section V.

To have a complete and working cluster, these services are delivered through various components. Fig. 3 depicts a block diagram of our architecture.

Figure 4 is an illustration of the overall functionality of each components of the proposed architecture. We will briefly explain the functionality of the whole architecture and the role of each components. All the internal communications in the fog layer are done through the agents, but, for simplicity, we do not explain the interaction of agents while describing the connection between modules of fog nodes and the administrator node. Here we also assume that communications through agents are reliable and fault-free.

When application requests are received in the fog layer through the communication module, they will be listed

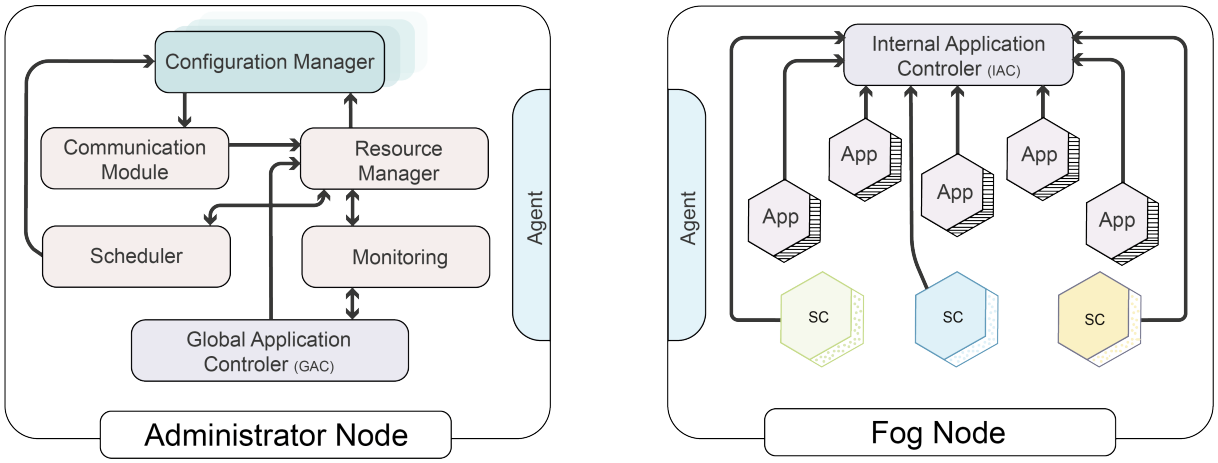

Fig. 3. Proposed Container Orchestration Fog Architectures

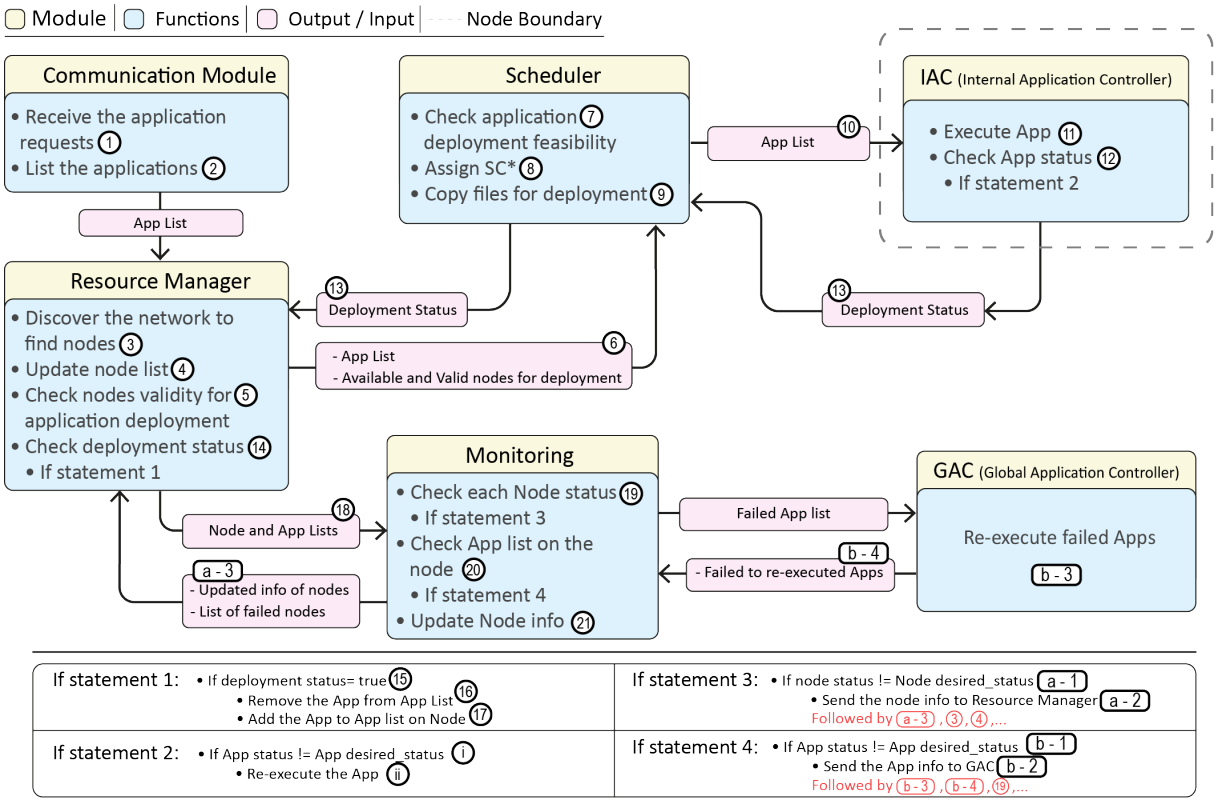

Fig. 4. Overall Functionality of the Architecture

and sent to the Resource Manager, which then discovers the network to update the list of available resources and performs a validity check for each of the resources $R = \{CPU, RAM, Storage\}$, as shown in Equation 1:

$$"Validity\_Check" = (R_{Ni} - \mu R_{Aa}) \geq 0 \qquad (1)$$

R indicates the resources capacities of Nodes and Applications. Therefore R_N and R_A denote CPU, RAM and $Storage$ capacities of fog nodes and applications respectively. μ is a percentage added to guarantee that application resource consumption on a node does not exceed a given threshold μ. Validity check is performed for each application a in the App_a subset which are the applications waiting for deployment. If a node passes the validity check for all the resources, its specifications will be sent to the Scheduler along with the list of applications to be deployed on the node. The Scheduler will perform a feasibility check as shown in Equation 2 and then if the deployment is feasible the application will be copied on the node for deployment. k denotes the number of applications currently running in a node and are members of the App_List_on_Node subset for each node. The remaining available resources considering the running

application(s) on each node is calculated in feasibility check. If the current available resources on the node suffice the new application request in terms of resources then the application will be copied on the node for deployment. If the application a in the App_a requires persistent storage, (indicated in its specifications as $Storage\_Tag$) the Scheduler will assign a storage container as well. The details of storage container assignment are given in Section V-B.

$$\textit{"feasibility check"} = (R_N - \sum_{i=1}^{k} R_A(i)) > 0 \qquad (2)$$

Afterwards, it is the role of the Internal Application Controller (IAC) to do the deployment. IAC is located inside each node and another responsibility of IAC is to check the status of each application to verify that they are working as desired. If they are not, IAC will re-execute the non-working applications inside the node without notifying the Administrator node for taking action for application re-deployment. This works is as an internal audit and control for checking if applications inside a node are always working as desired. The Monitoring module will also monitor the status of both the nodes and the applications running on the nodes, and in case of node failure it will call the Resource Manager to find a new suitable node. In case an application fails and IAC cannot re-execute the application, the monitoring module will call the Global Application Controller to do the application re-execution.

Equation 3 shows how the Monitoring module calculates the remaining resources of each node, which is needed in order to update the node resource information. The Monitoring module send this information to the Resource Manager. The variable R_{rN} is the remaining capacity of each fog node, again calculated for each resource $R = \{CPU, RAM, Storage\}$.

$$R_{rN} = R_N - \mu \sum_{i=1}^{k} R_A(i) \quad \forall k \in App\_List\_on\_Node \quad (3)$$

Figure 4 illustrates the overall functionality of our architecture, with the number 1-21 representing the sequence of actions in normal (non-faulty) operation. Steps number 1 to 5 represent receiving application request and sending it to Resource manager and Validity check in the Resource manager. Steps number 6 to 12 show the application deployment process. Steps number 13 to 18 indicate the process in which an application from the App_a subset (applications waiting for deployment) are removed from application list and added to the $App\_List\_on\_Node$ list. Steps number 19 to 21 show the functionality of monitoring module that constantly checks the status of applications and nodes in the cluster. Steps (i) and (ii) represent the behaviour of IAC when an application fails and IAC recovers it internally inside the node. Steps a-1 to a-3 followed by steps 3, 4, etc. show the process in which a node failure is detected and recovered whereas steps b-1 to b-4 followed by steps 18, 19, etc. show the case in which the monitoring and GAC components recover an application failure that could not be recovered internally.

V. Fault-tolerant Persistent Storage in container-based fog architectures

The *state data* of a stateful application is the memory content that needs to be kept from one invocation to the next. The objective of our fault-tolerant persistent distributed storage is to guarantee that correct and up-to-date state data is available at each execution of any stateful application supported by our fog architecture. This means ensuring that any state data stored during an invocation must be readily available for the subsequent invocation of said application, and this holds even if the application was restarted, for example due to a failure, or deployed on a different node.

Additionally, our solution must adhere to the basic principles of containerization: it has to be scalable on demand, support migration between different nodes in the cluster and implement self-healing whenever a fault occurs. This will be realized by means of a container-based mechanism that creates and handles consistent replicas of stateful data across the fog layer.

A. Design rationale

In a container-based architecture, state data is normally stored in a local volume that can be read in every invocation, but which is lost after re-initialization of the container. Our solution still uses storage in the local volume but extends it by means of a specific container, called the *Storage Container* (SC). The SC keeps a copy of the state data and also shares it within a cluster of synchronized SCs, such that several consistent copies of the data exist and can be retrieved if needed. The consistency within the SC cluster is guaranteed by means of a consensus protocol. In our case, we rely on RAFT, but other consensus protocols might be applicable as well [10] .

Therefore, we can distinguish two processes concerning storage in the SC. There is a process called *Internal synchronization* in which the application container reads (fetch) or writes (commit) from/to the shared environment in the node which is a local storage space on the node created by the Storage Container, and a process called *External synchronization* in which the Storage Container reads (fetch) or writes (commit) from/to the shared environment in the node. When SC fetches the states from the shared environment which are related to the applications working inside the same node it sends the states to the SC cluster (leader in the cluster) in a consistent manner. The states from applications running on other nodes in the cluster are received from the SC cluster (leader in the cluster) to be committed to the shared environment by the local SC in the node.

The Storage Container is deployed transparently like any other container, but there are some specific steps that need to be taken by the Scheduler. These will be described next. Note that SC also benefits from the fault-handling mechanisms discussed in Section IV.

B. Architecture and operation

Like any containerised application, a storage container is characterised by its specification:

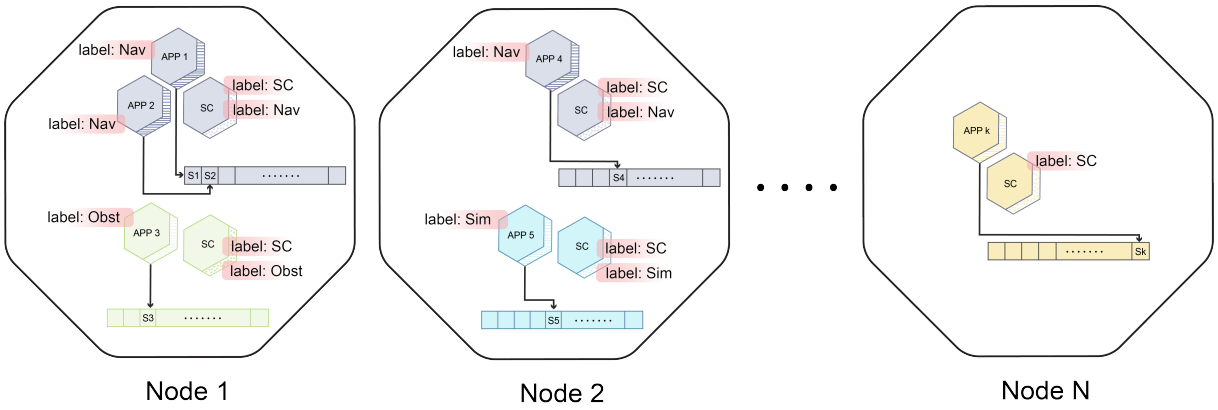

Fig. 5. Proposed Storage Container Schema

$$SC_j = \{CPU, RAM, Storage, RAFT\_Status, Run, Label, Status, Desired\_Status\}$$

A specification indicates the resources that a SC requires in terms of *CPU*, *RAM* and *Storage*. It also indicates the status of the SC in every time unit of the RAFT protocol, which can be one of the following: (1) follower, (2) candidate and (3) leader. The *Label* of a SC is vital for grouping containerized storage and applications correctly and it takes two parameters, (1) SC_label and (2) App_label. Finally, Status and Desired_Status are used to check whether SC execution process has the desired status.

The number of SCs in a node depends on the number of different application labels in the node. Figure 5 depicts a system of fog nodes with the containers (applications and SCs) inside them. Note that one or more SC have been assigned to the applications. It is the role of Scheduler to assign SCs with the same labels to the corresponding applications. For instance, in Node 1 there are three applications. Two of them have a label "Nav" (navigation application) and one has a label "Obst" (Obstacle detection application), thus, two different SCs are deployed in this node.

Each instance of a containerized application generates a state and updates it after each execution. First, the state data is stored locally in the application container volume and then the data is duplicated to be stored and committed in the shared environment created by the SC. The duplicated state is accessible by SC, means that only when SC is not in the external synchronization mode it can fetch the state from shared environment. When SC fetches the state created by local applications inside the node from the shared environment it goes to external synchronization mode and send the states to the SC leader in the SC cluster. This is the role of SC leader to send and commit the states in a consensus manner to/from all SC members in the cluster.

Reintegration of SCs occurs every time the SC is executed for the first time, which can happen after the first deployment or as a consequence of a restart due to failure. In such a case, the SC first collects the states from the applications with the same labels and later, in the first external synchronization, it reconciles its data with the synchronized set of states kept by the cluster.

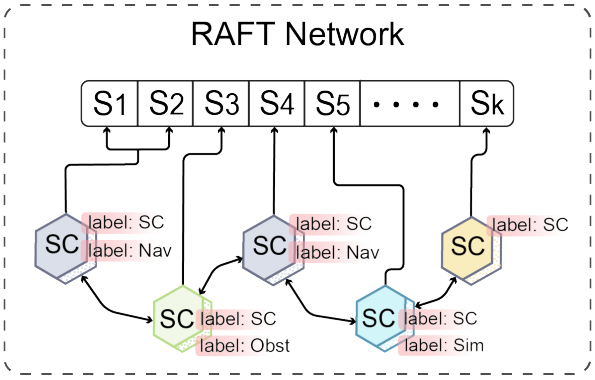

Fig. 6. RAFT Consensus Network for SCs

Figure 6 shows the RAFT network created by a set of SCs and shows how each SC contributes to form the whole state set. All the storage containers having the label SC are members of this RAFT network.

The aforementioned RAFT_Status determines how SCs have different status in the RAFT network and how they interact to synchronize in the network. A newly created SC has the Raft_status *follower*. As it is a new entity in the SC RAFT network, it must follow the leader SC to be consistent with the state changes. The details of the RAFT protocol are explained in [11].

C. Achieved fault tolerance

Let us now review how faults will be handled in this new architecture.

Omission and Crash Failures: Omission and crash failures both in application and at node level can be detected and recovered in this architecture.

- Omission and Crash in application: When an application fails, IAC restarts the application. If the application restart fails, the monitoring module will notify GAC to redeploy the application. The application redeployment mechanism works also in case the IAC fails.
- Omission and Crash in node: Whenever a node does not provide a response (also if caused by a communication failure), the monitoring tool will detect this failure and

notify the Resource Manager. The Resource Manager will find an alternative to the node, normally a new node, and then the applications from the faulty node will be deployed on the new available node.

Response Failure: This failure occurs in a distributed system when an incorrect response is provided. It can be caused by a problem with the value or because of an error in the logical flow control. Regardless of cause, the error manifests as a State failure and will be discussed in the next point.

State failure: State failure can be caused by two issues: i) application malfunctioning that changes the value of the state data and ii) state failure due to incorrect initialization of the application after restart. The first type of failure cannot be detected nor tolerated by our architecture, and would require redundancy of the application containers. An example of this type of failure in the context of our ROS application is if the function responsible to process some sensor, like e.g. the camera or the lidar, fails, resulting in the generation of incorrect values. The same would happen if the sensor itself is faulty. This can be tolerated with traditional spatial redundancy, either with replicated nodes or with replicated application containers.

The second type of failure, improper application restart, is prevented by the existence of our permanent storage. In our architecture, the application containers synchronize with the corresponding SC after restart, and obtains the correct state from the SC cluster.

VI. Related Work

Existing solutions for distributed fault-tolerant distributed storage systems in fog, edge and cloud computing have primarily focused on two fundamental problems in distributed systems: providing fault-tolerant, permanent data storage [12] and achieving a decentralised consensus. The first set of solutions focuses on distributed data storage systems and optimal allocation of redundancy, to reduce utilization and techniques for error detection and reconfiguration upon failure and the latter one is applying consensus protocols based on system requirements [10].

An instance for optimal allocation of redundancy in distributed storage systems is provided by Jonathan et al. [13]. The issue in leveraging the method proposed in [13] is that it is only efficient when the failure rate is low. For large scale fog applications and container-based architecture we cannot directly apply such a method as redundancy mechanism.

For error detection in distributed storage system, Chervaykov et al. [14] propose a reconfigurable data storage system based on Redundant Residue Number System (RRNS). This work provides a theoretical basis to calculate probability of information loss, data redundancy and configure parameters, but the efficiency and effectiveness of their solution is not measured for large scale systems.

Shahaab et al. [10] investigates 66 consensus protocols such as RAFT [11], Byzantine Fault Tolerance [15], Sieve [16], etc. are studied and analysed based on different objectives, like, sustainability, efficiency, etc. The conclusion is that, there is no single consensus protocol to apply as a solution for all the requirements in a distributed system. A consensus solution must be leveraged on the network, types of nodes, and the whole system requirements. In our proposed solution, we use the RAFT protocol to achieve consensus in our distributed storage system.

There are also a number of recent works proposing the leverage of persistent storage solutions for container-based architectures in cloud platforms. For instance, Shrama et al. [17] proposed a distributed storage system using storage application deployment on Kubernetes, however, they did not address the transient storage issue on the Pods in Kubernetes orchestration solution. This work also lacks a consideration of any consensus algorithm. In our solution we focus both on data persistent storage locally on the node and also we consider a consensus protocol to recover data in case a storage container in a node fails. Kristiani et al. [18] also proposed persistent volume for container-based architectures using Openstack and Kubernetes. Although in this work the container-based applications are running on the edge devices in the network, the persistent storage solution is still located in the clouds which increases delay in response for each data access request by the application.

To ensure that operations are executed on all the containers and their replicas, state-machine replication in containers is proposed by Netto et al. [19]. Authors in this work, use a protocol which uses shared memory to project communication and persist data. The main focus of this work is on latency and authors did not consider resource consumption in this work.

In another recent work by Netto et al. [20] the incorporation of RAFT protocol in Kubernetes has been proposed. In this work, execution requests can be sent to any replicated container, matching the cloud nature in regard to load balancing. However, the overhead of this solution on containers change the light weight nature of using containers and increase the container image size. Compared to other works, for instance, [9] and [21], the level of protection provided in the work by Netto et al. [19] is stronger. However, this comes at the price of increasing the resource usage. In this work handling one failure, for each container that provides a service requires two extra containers. In our solution, however, one extra container is enough for recovering the service after a single failure.

Studying the literature provided us with a better view of the possible existing solutions as well as broadening our knowledge about our system requirements. The need for persistent storage comes from the stateful application we are dealing with in our use case. In addition, we needed to investigate what could be done at the network edge in the fog layer, to reduce data latency and provide immediate response to data access requests in the fog layer. To the best of our knowledge data recovery/reintegration upon node failure in Kubernetes had not been investigated yet for fog architectures. Our work extends current state of the art by taking advantage of the light weight and self-healing characteristics of containers and proposing a novel container-based applications that provides consistent distributed storage.

VII. Conclusion

This article proposes a novel container-based architecture for fog layer orchestration, especially suited for dependable cyberphysical systems. Our architecture adapts a cloud-native solution for container-based application orchestration and management, and extends it to tackle a number of dependability limitations that were identified in a case study involving a containerized robotics application. This includes improved error-handling mechanisms and a container-based subsystem for fault-tolerant permanent storage. We see this architecture as a first step towards the design of a complete fault-tolerant fog architecture, which will have to be extended as new dependability limitations are revealed in other applications domains. The next steps of our work are to build a formal model of the architecture to formally verify the storage subsystem, and to evaluate our first prototype on a more complex use case.

ACKNOWLEDGMENT

This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 764785, and by the Swedish Foundation for Strategic Research (project FiC).

We thank Dr. Johan Relefors for providing us with the ROS application used for the experiments in this work.

References

[1] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015. DOI: 10.1109/MCC.2015.51.

[2] J. M. O'Kane, *A gentle introduction to ROS*, 2014.

[3] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: An open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[4] *Docker Hub, Docker Description*, https://www.docker.com/resources/what-container.

[5] *Kubernetes Foundation, Kubernetes Documentation*, https://kubernetes.io/.

[6] R. White and H. Christensen, "ROS and Docker," in *Robot Operating System (ROS): The Complete Reference (Volume 2)*, A. Koubaa, Ed. Cham: Springer International Publishing, 2017, pp. 285–307, ISBN: 978-3-319-54927-9.

[7] L. E. Parker, "Reliability and fault tolerance in collective robot systems," *Handbook on Collective Robotics: Fundamentals and Challenges*, 2012.

[8] *Parameter Server, ROS*, http://wiki.ros.org/ParameterServer/.

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

[10] A. Shahaab, B. Lidgey, C. Hewage, and I. Khan, "Applicability and appropriateness of distributed ledgers consensus protocols in public and private sectors: A systematic review," *IEEE Access*, vol. 7, pp. 43 622–43 636, 2019.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[12] M. Itani, S. Sharafeddine, and I. ElKabani, "Dynamic multiple node failure recovery in distributed storage systems," *Ad Hoc Networks*, vol. 72, pp. 1–13, 2018.

[13] A. Jonathan, M. Uluyol, A. Chandra, and J. Weissman, "Ensuring reliability in geo-distributed edge cloud," in *2017 Resilience Week (RWS)*, Sep. 2017, pp. 127–132.

[14] N. Chervyakov, M. Babenko, A. Tchernykh, N. Kucherov, V. Miranda-López, and J. M. Cortes-Mendoza, "AR-RRNS: Configurable reliable distributed data storage systems for internet of things to ensure security," *Future Generation Computer Systems*, vol. 92, pp. 1080–1092, 2019.

[15] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[16] C. Cachin, S. Schubert, and M. Vukolić, "Non-determinism in byzantine fault-tolerant replication," *arXiv preprint arXiv:1603.07351*, 2016.

[17] A. Sharma, S. Yadav, N. Gupta, S. Dhall, and S. Rastogi, "Proposed model for distributed storage automation system using kubernetes operators," in *Advances in Data Sciences, Security and Applications*, Springer, 2020, pp. 341–351.

[18] E. Kristiani, C.-T. Yang, Y. T. Wang, and C.-Y. Huang, "Implementation of an edge computing architecture using openstack and kubernetes," in *International Conference on Information Science and Applications*, Springer, 2018, pp. 675–685.

[19] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. S. de Souza, "State machine replication in containers managed by kubernetes," *Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017.

[20] H. Netto, C. Pereira Oliveira, L. d. O. Rech, and E. Alchieri, "Incorporating the raft consensus protocol in containers managed by kubernetes: An evaluation," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, no. 4, pp. 433–453, 2020.

[21] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2016, pp. 202–211.