Comparing Different Persistent Storage Approaches for Containerized Stateful Applications

Patrick Denzler
Institute of Computer Engineering
TU Wien
Vienna, Austria
patrick.denzler@tuwien.ac.at

Daniel Ramsauer
Institute of Computer Engineering
TU Wien
Vienna, Austria
daniel.ramsauer@tuwien.ac.at

Thomas Preindl
Institute of Computer Engineering
TU Wien
Vienna, Austria
thomas.preindl@tuwien.ac.at

Wolfgang Kastner
Institute of Computer Engineering
TU Wien
Vienna, Austria
wolfgang.kastner@tuwien.ac.at

Alexander Gschnitzer
Institute of Computer Engineering
TU Wien
Vienna, Austria
e1652750@student.tuwien.ac.at

***Abstract*—Cyber-physical systems are highly distributed, flexible, and closely connected to the physical world. State preservation is an essential aspect of operating cyber-physical systems. If stateful applications fail, the current state needs to be restored so that the system can operate again. With the entry of edge computing, applications can now be containerized close to the equipment and allow new use cases. The lack of persistent storage in containers to ensure statefulness requires external, centralized, or distributed solutions. This paper explores this spectrum by comparing three experimental implementations and indicates possible causes for state loss. The underlying aim is to provide a starting point for choosing which state preservation approach is most suitable for which application type. The paper concludes with future research steps.**

***Index Terms*—distributed computing, stateful applications, edge computing, cyber-physical systems**

I. Introduction

Undoubtedly, modern software systems are required to evolve towards being more distributed, flexible, autonomous, and closely connected to the physical world. Cyber-physical systems (CPSs) represent this best in all facets [1]. In such systems, several applications/components work together to achieve a goal, such as creating a product or providing a service. The complexity involved that comes along with building a CPS is not negligible [2]. Designing a distributed system where different applications interact requires careful planning and implementation, especially if the system needs to be highly reliable, scalable, and maintainable.

One essential aspect of a CPS is that all parts perform their tasks correctly. Strongly simplified, CPS parts are state machines that perform a task and wait for the next one [3]. Some state machines, will always return the same output given a specific input. Others, however, change their "state" and the output changes accordingly. Such state machines or applications are stateful. For example, if a process requires a certain sequence of state changes and the state machine crashes, the process needs to start all over again. In a CPS, where applications work concurrently together, such a state loss of one component could create complete unexpected system behavior [3]. The reasons for state loss could be due to component damage or external influence.

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

Preventing state loss is an old concern in computing or automation. For example, in production lines controlled by programmable logic controllers (PLCs), the PLCs have internal storage to save their current state [4]. In case of a power failure, such a local backup allows a recovery to the last state, and the system can proceed with its task. However, new use cases requiring statefulness appeared with the arrival of the Internet of Things (IoT) and specifically the entry of edge and fog computing in operational technology (OT) [5].

The idea behind edge/fog computing is to bring computing resources closer to the network's edge [5]. That allows applications to be hosted in virtual machines or containers running on fog devices providing various kinds of services. Examples are computation offloading, virtual PLCs, or runtime analytics [6]. Some applications are stateful and require, in case of failure, their state to be restored, which is not as straightforward in a distributed and mostly containerized environment [7].

One issue in container-based systems is the presence of volatile storage, i.e., the available storage is part of the container and is lost when there is a redeployment [8]. There are several solutions to provide persistent storage for distributed applications hosted on fog nodes [7]. Some are centralized on a server or are fully distributed and replicate the information among all nodes, while others are in-between this spectrum [9]. Nevertheless, most are use case-specific and are rarely compared with others. Therefore, this paper aims to compare different state preservation solutions along this spectrum and provide insights to choose among them.

The paper introduces possible causes for state loss in containerized distributed systems and a design rationale for persistent storage solutions. In a further step, a centralized, a hybrid, and a fully distributed prototype for persistent storage are compared. Each of the prototype implementations utilizes a compute cluster with container capabilities and a simple stateful application that provides the state that needs to be stored. The evaluation setup allows comparing the prototypes related to their scalability and state replication performance.

The contribution of this paper is twofold. Each of the implemented prototypes is openly available and the obtained measurements give a starting point for an application developer to decide which solution might be most suitable. Secondly, the paper suggests evaluation parameters to compare different persistent storage solutions.

This paper proceeds by introducing the related work. Section III sheds light on possible causes for state loss, the design rationale, and possible solutions for persistent storage solutions. Section IV presents the actual implementation details of the three prototypes. The evaluation and the results are contained in Section V. Section VI discusses the findings, and Section VII concludes the paper and points to future research directions.

II. Related work

Distributed storage systems for fog, edge, or cloud computing mainly focus on two problems: providing fault-tolerant permanent data storage and decentralized consensus [10]. Possible solutions are an optimal allocation of redundancy, reducing utilization, or introducing techniques for error detection utilizing consensus protocols based on system requirements [11]. In Jonathan et al. [12], an instance for optimal redundancy allocation in distributed storage systems is described. A limiting factor in the proposed method is that it only works efficiently with low failure rates, limiting its usability in fog applications and container-based architectures.

Concerning error detection, Chervaykov et al. [13] propose a reconfigurable data storage system based on Redundant Residue Number System (RRNS). The system aims to calculate the probability of information loss, data redundancy, and configure parameters. In a similar area, the authors in Shahaab et al. [11] were looking into 66 consensus protocols such as Raft [14], Byzantine Fault Tolerance [15], or Sieve [16]. Their study objectives were to investigate the sustainability or efficiency of the protocols, and they concluded that no consensus protocol could fulfill all requirements of a distributed system.

Narrowing down to solutions for persistent storage solutions for container-based architectures in cloud platforms, several authors look into that subject [17]–[21]. In Sharma et al. [19], the authors propose a distributed storage system using storage application deployment on Kubernetes. In their work, they do not consider any consensus algorithms or address the transient storage issue Kubernetes orchestrated pods. A similar solution is found in Kristiani et al. [22], where the authors introduce a persistent volume (PV) for container-based architectures using Openstack and Kubernetes. While the applications run on the edge resources, the PV is located in the cloud, which can cause response delays for each data access request.

In Netto et al. [21], a solution is introduced where state-machine replication is executed in all containers (and their replicas). All incoming requests are processed in all containers, but only one replica will respond to the request. The underlying protocol utilizes shared memory to project communication and persists data. In a follow-up paper [23], the authors integrate an execution layer and a firewall container between clients and containers. This so-called Koordinator receives the clients' requests and sends them to the respective application containers. Comparable solutions can be found in [24] and [25], however, with a lower level of protection.

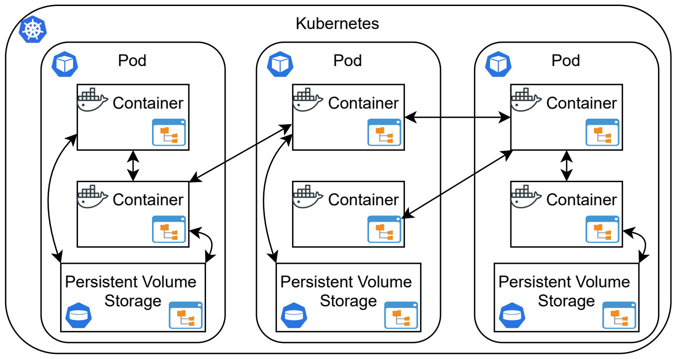
Moreover, in more recent work by Netto et al. [18], a solution is presented that incorporates the Raft protocol in Kubernetes. This setting allows a request to be sent to any replicated container to achieve better load balancing. However, the solution increases the overhead quite significantly. In the works of Bakhshi et al. [26], [27], the authors introduce a container-based local state replication solution that utilizes the Raft protocol and addresses most of the shortcomings of the papers mentioned beforehand. In summary, most of the related work focuses on distributed solutions and does not compare to other solutions that, for example, utilize centralized storage.

III. Background / Preliminaries

This section provides the necessary background information for the following sections. Particular emphasis is given to the difference between stateful, and stateless applications, possible failures and state preservation solutions in containerized distributed systems.

A. Virtualization, Docker and Kubernetes

Virtualization techniques are the preferred option for deploying applications in fog networks as it allows the allocation of isolated execution of components in heterogeneous environments [28]. Containerization is a lightweight virtualization approach for deploying applications by bundling the code and all needed dependencies [7], [29]. In order to control the communication and manage the deployment of containers, tools such as Kubernetes [30] provide a framework that facilitates the maintenance of deployed services, monitors the health of containers, and redeploys them if necessary. Additionally, various components, like an abstraction of service discovery or controllers to coordinate the replication and deployment of containers, allow for the operation of complex combinations of container setups. A vital feature of Kubernetes is the introduction of so-called pods. Pods provide a closed local environment for multiple containers, including local storage (volumes). However, since a pod is mortal, the volume inside the pod is also volatile (ephemeral). Files are stored transiently in the local system spaces. Terminating a pod leads to that data and information are not accessible anymore. Figure 1 visualizes the relations between Kubernetes, pods, and containers.



Kubernetes
Pod
Container
Container
Persistent Volume Storage
Pod
Container
Container
Persistent Volume Storage
Pod
Container
Container
Persistent Volume Storage

Fig. 1. The relations between Kubernetes, pods and containers

B. Stateful Vs. Stateless Applications

Nevertheless, before identifying possible failures that could cause state loss, stateful applications need some definition.

Figure 2 depicts the difference between stateless and stateful applications. On the left side of the figure (a), the application is stateless. As for any given input, the outputs depend only on the input, the application can easily be replaced, or in other words, such applications are interchangeable. If a failure occurs (e.g., the application crashes), a replica can take over without disturbing the system.

In stateful applications, however, such an exchange is not possible. The reason is that the application changes its current state with every input, and therefore, the output changes accordingly, i.e. the output depends on the input and on the current state of the application. In case of a failure, the state is lost, and the application does not produce the expected output, despite it being recovered and redeployed. Figure 2 depicts such a scenario on the right side (b). Therefore, for stateful applications, a state loss needs to be avoided to ensure the correct functioning. Typical means to avoid a state loss is having the state stored unaffected by any kind of failure.

C. Potential Failures Causing State Loss

The following are potential failures in containerized stateful applications that can lead to state loss. Other authors, e.g., [21], [26], found similar issues specific to containerized applications. The introduced taxonomy of errors discussed by Parker [31] provides the necessary guidance. Other types of errors, e.g., a hardware failure, as described in Kleppmann [32], are summarized as one error, i.e., a whole node stops working.

As previously described, the applications are placed in containers running within pods, and all handling is left to Kubernetes. This arrangement opens up several potential failures causing state loss:

- *Node Failure*: A node can completely fail due to disconnection or any hardware failure.
- *Application Execution Failure*: The application stops working due to an undefined internal reason.
- *Application Deployment Failure*: The application is not correctly initially deployed or re-deployed. Several reasons are possible, such as no available resources, losing communication to available nodes, or improper resource management.
- *File Access Failure*: Applications fail to access required files, states, and other necessary data to continue working. This encompasses the loss of volumes in pods, e.g., due to Node Failure.
- *Management failure*: This includes all failures caused by functions required to run a container environment, e.g., Docker Engine, Kublet, Kube-proxy [29], [30].

While all of these failures can result in state loss, the most serious is losing access to the state stored in the local pod volume. This situation can be caused by *Node Failure* and *File Access Failure* as there is no persistent storage. While Kubernetes supports stateful applications by providing PVs (a filesystem to store data in a pod), and stateful set, they have some shortcomings. According to Vayghan et al. [7], [24], when a pod fails, the recreation time plus access time to a PV is considerably high, limiting applications with high availability requirements. Additionally, in the event of a node failure, the stateful set cannot recover a pod until the specific node is back on the network. Restoring and redeploying a pod to a new node can only be done manually but involves an inevitable loss of access to PV. In addition, reliable PV can only be implemented using cloud storage, which causes higher latency in data access requests [7], [33]. Kubernetes also supports PersistentVolumeClaims, which have been analyzed by Vayghan et al. [7], [24].

In summary, as long as a solution provides highly available persistent storage for state recovery, stateful applications can run in containerized environments. A wide range of solutions adds even more relevant functionality to this type of storage.

D. Design Rationale for Persistent Storage Solutions

The design rationale for fault-tolerant persistent storage for states of stateful applications is as follows. The main objective of the solution is to guarantee that correct and up-to-date state data is available at each execution of any stateful application. In other words, any state data stored during an invocation needs to be available for the following invocation of the respective application. This goal must be met even if the application loses its current state due to an outage or is deployed to another node.

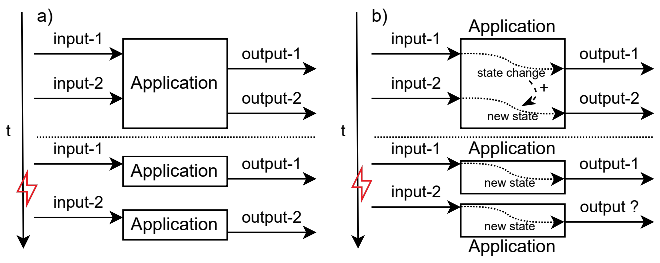


Fig. 2. Stateless (a) vs stateful (b)

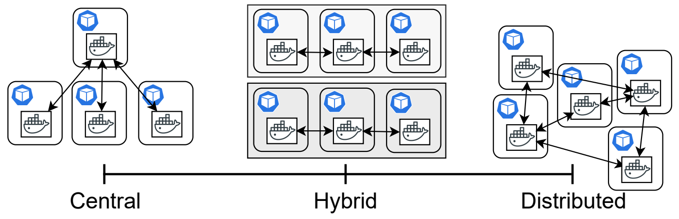


Fig. 3. Persistent storage solution categorization

In addition, the solution must support the fundamentals of distributed container systems, e.g., on-demand scalability, migration of containers between different nodes, and self-healing properties when an application fails. Using containerization already satisfies some of the additional requirements, as it tolerates configuration and task completion failures through self-healing mechanisms, most notably the ability to restart failed containers. However, a PV file system is not sufficient for persistent storage, as elaborated in [7], [24], [26].

Solutions that meet these goals are best described along the spectrum from centralized to fully distributed. Figure 3 illustrates the idea of categorizing fault-tolerant persistent storage solutions along the spectrum, particularly as some solutions combine aspects of centralized and distributed systems.

E. Centralized

A centralized fault-tolerant persistent storage provides a centralized point to save all states of the various applications. In the simplest case, the data storage is a database located on a server that stores the states when they arrive.

This type of solution allows a relatively quick state restoring of a failed node, as the restored node only requests the latest entry in the database. The downsides of centralization are, for example, having a single point of failure, more traffic on the network, or longer latencies depending on the physical connection between the nodes and the database [32]. Nevertheless, there are several enhancements available that negate some of the drawbacks. For example, databases can be redundant, located closer to the edge on more powerful fog nodes to reduce latency, or network splitting reduces the load on a single server [6]. A side effect of extending a single data store is ensuring the consistency of a state stored in several places. The more the solution wanders on the spectrum towards fully distributed, the more relevant becomes consistency between replicated data.

F. Consistency Model

Replication means that a copy of the same data is stored on multiple machines in a network [32]. That can mean that each node has all data of all the other nodes or a small subset of it. The tricky part of replication is to ensure that all nodes have the ”right” data at the right time when data is changing. Some well-known algorithms such as single-leader, multi-leader, or leader-less are a typical means to approach this issue.

In a single leader environment, the nodes send their data writes to a single node (leader), which in turn sends the changed data events to all other nodes (followers). Reading data can happen at any node. In multi-leader, there are several leader nodes that accept data writes and exchange their updates. In leader-less data writes are sent to various nodes, and reading can happen in parallel. All algorithms have their benefits and drawbacks; while single-leader is easier to implement, multi-leader and leader-less show more robustness in the presence of faulty nodes, high latency, or network errors [32]. The cost is, however, lower consistency.

Replication can be synchronous or asynchronous, affecting the behavior of the system in case of failure. Asynchronous replication is considered fast yet more sensitive to replication lag and failing nodes. If a leader node fails and a new node is appointed, the last committed data is lost.

Replication lag in a leader-follower environment can create different types of effects concerning consistency. One example in asynchronous replication is the possibility of outdated information and inconsistency. Sending the same request to a leader and follower, at the same time, can result in two different answers, because not all writes have been processed in the follower. This inconsistency is of temporary nature and will resolve eventually. Nevertheless, such effects require approaches to ensure consistency.

A potential consistency model to avoid replication lag is *read-your-writes consistency* [32]. This model ensures that a node always gets back its latest write, but does not make the same promise for other nodes. In asynchronous replication, that can be achieved, for example, by always asking the leader for the latest write. However, a leader might be overloaded with too many requests, and if a leader fails, the write is lost before replicated, and therefore the consistency model is not fulfilled. In that case, synchronous replication and a consensus algorithm (e.g., Raft) between the nodes is the better choice [14]. Such a combination fulfills the consistency model, as it tolerates a leader failure.

Errors that cannot be identified are erroneous writes from an application. Erroneous writes can be caused by either malfunctioning applications or incorrect initialization of the application after a restart. Solving the first issue would require a traditional spatial redundancy, either with replicated nodes or with replicated applications. While improper application restart is to a certain level, prevented by the existence of a persistent storage solution.

After introducing replication, it is possible to describe the concept of a distributed persistent storage.

G. Distributed

At the other end of the spectrum are fully distributed solutions. A distributed fault-tolerant persistent storage provides storage divided over several nodes [19]. The storage can be simple local file storage, replicated to the other nodes, or more advanced solutions involving databases and a variety of consensus algorithms. In the extreme example, each node keeps the states of every application in the network. The authors in Bakhshi et al. [26] present such an approach based on PV and Raft as a consensus algorithm.

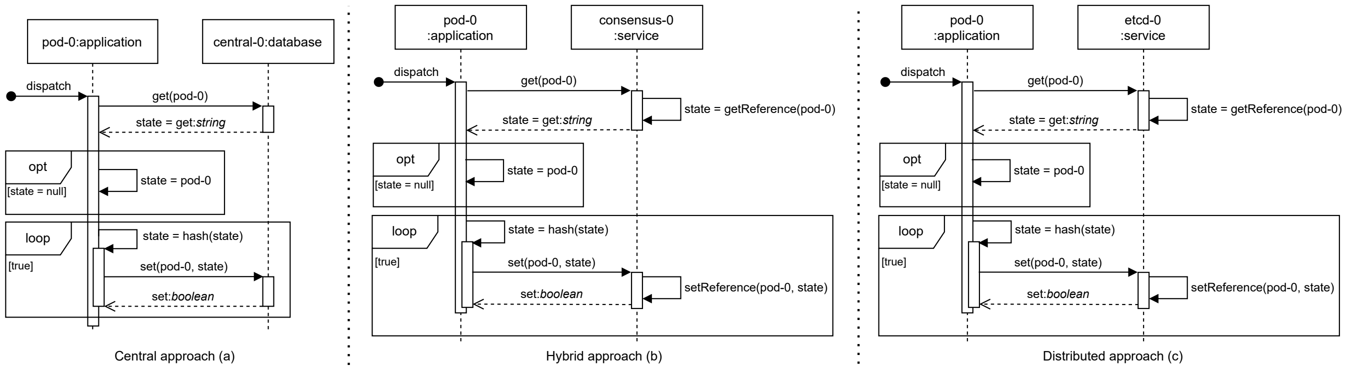


Fig. 4. Sequence diagrams depicting the interactions involved in the three prototype implementations.

The benefit of distribution is that there is no single point of failure; leader nodes that fail can be replaced quickly, increasing the system's availability [7]. A downside is the increased complexity involved in building and maintaining such systems [1]. Difficulties are, for example, imperfect networks that fail to deliver writes, timing effects that cause unexpected behavior, or partial failures. There is a wide range of solutions to counteract those difficulties; some introduce elements of centralized solutions and create hybrids.

H. Hybrid

As indicated beforehand, hybrid distributed fault-tolerant persistent storage solutions utilize elements from both ends of the spectrum. Examples of hybrid approaches are several databases that replicate states of the connected nodes, network partitioning, or nodes that host databases taking care of the replication. The significant difference is that the nodes within a group provide data replication for each other (cf. Figure 3). There is only limited communication with outside nodes.

I. Performance Evaluation Parameters

How to compare or evaluate the solutions along the spectrum is not uniquely consented, as there is no commonly agreed test setting. This situation leads to most academic papers evaluating only a specific aspect of the presented solutions. Nevertheless, potential performance evaluation parameters are: Network or data latency (including response times of the system) [21], replication lag, or created network traffic. Moreover, the authors evaluate failure, restart times of nodes, or how long the system requires to find a new leader. Some also focus on container-related parameters, such as startup time or resource consumption (CPU, memory, storage space) [18]–[20]. More specific measurements are fault tolerance (the number of nodes that can fail simultaneously), maximum node size, or the number of subgroups (partitioning).

IV. Prototype Implementations

The following three prototype implementations of a central, distributed, and hybrid approach fulfill the design rationale in Section III-D. As depicted in the sequence diagrams in Figure 4, all three approaches share the same stateful application. The application simulates the behavior of a stateful application requiring state preservation. When started (either due to a failure or startup), the application tries to retrieve a previous state from the application programming interface (API)'s *get* endpoint of the respective persistent storage approach. If there is no previous state, the application initializes its state as its instance name and enters the main application loop. In this infinite loop, the application performs a state change by hashing its last state and using the result as the new state.

Further, it stores the new state using the *set* endpoint of the respective persistent storage approach and sleeps five seconds after completion. The sequence diagrams depict each persistent storage solution. Depending on the approach, this storage is provided to the application in a central, distributed, or hybrid architecture.

A. Centralized Approach

The first prototype represents the centralized approach described in Section III-E, which utilizes an instance of a database and provides access via a service controller (cf. Figure 5). The prototype implementation uses CouchDB[1] to store non-relational data in the form of JSON documents. Each document contains a unique id, a revision number, a value, and an expiration date indicating the validity of the state. Using CouchDB's API endpoints, clients can read and write to the database structure. Figure 4 (a) illustrates the interaction between a client, i.e., an application replica, and the database.

[1]https://couchdb.apache.org/

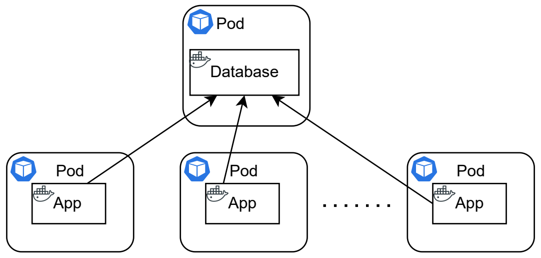


Fig. 5. Centralized approach to provide persistent storage

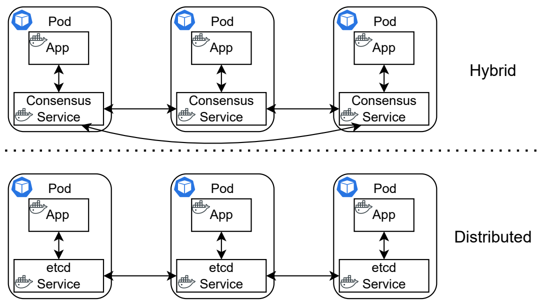


Fig. 6. Overview of the hybrid/distributed approach. In the distributed approach, etcd takes the place of the consensus service.

B. Distributed Approach

The distributed approach (see Section III-G) stores the states in the different nodes and increases the fault tolerance of the system. As discussed earlier, the solution needs to fulfill read-your-writes consistency; therefore, the nodes need to agree on every state change. The consensus is achieved by using *etcd*[2]. Each pod contains a companion container that acts as a consensus service interface. The applications in the pod can use the etcds *set* and *get* endpoints to store and retrieve its state. Etcd takes care to replicate the states to every other etcd instance in the system by utilizing the Raft consensus algorithm. Raft ensures that the state updates are only acknowledged after the cluster has reached a consensus. More than half of the nodes need to accept the state change to reach a consensus. Figure 4 (c) depicts how the application interacts with the consensus service, and Figure 6 shows the architecture of the approach.

C. Hybrid Approach

Like the distributed approach, the hybrid approach uses a deployment strategy consisting of the application container and consensus service. The consensus service provides *set* and *get* endpoints for data manipulation. In this case, the service uses a Hazelcast CP subsystem[3] as a distributed crash-fault tolerant replication system. Every service instance joins the CP subsystem cluster, which executes the Raft algorithm to achieve consensus on the list of members. The cluster assigns the members to CP groups of three to seven to store data and balance resource usage between all nodes. The CP groups run the Raft algorithm to reach a consensus every time a state should be stored. As in the distributed approach, Raft ensures that the state updates are only acknowledged after consistency has been reached in the cluster. More than half of the nodes need to accept the state change to reach a consensus. Figure 4 (b) depicts how the application interacts with the consensus service, and Figure 6 shows the architecture of the approach.

[2]Etcd (v3.5.2) is a consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.

[3]Hazelcast (v4.2) is an in-memory data grid (IMDG) solution (https://hazelcast.com/). The CP subsystem is a component of a Hazelcast cluster that provides strongly constituent data structures.

V. Evaluation and Results

This section uses some of the previously presented evaluation parameters to compare the three prototypes. Due to space limitations, the section only contains memory and CPU usage, store and retrieve latency, and start-up delay. A Raspberry Pi cluster enabled all measurements.

A. Evaluation Environment

The Raspberry Pi cluster connects 24 Raspberry Pi 4 computers over a dedicated 1Gbit network as nodes, and Kubernetes (1.21.4) provides the management interface. For further details on the cluster, the reader is directed to [34].

For retrieving metrics, the cluster uses Prometheus[4]. The application sends the evaluation parameters to the Prometheus service hosted within the cluster at regular intervals. Additionally, Prometheus queries the Kubernetes controller, which stores up-to-date information about each pod.

B. Measurement Scenario

The actual measurements on the cluster follow the same scenarios as depicted in Figure 4. Following the sequences allows obtaining the different parameters of each approach. While start-up delay, store and retrieve latency, and start-up delay are reported directly from the application, Kubernetes delivers CPU and memory consumption.

At the starting point, all nodes start at the same time. The number of nodes is variable from 5 to 21 nodes. Scaling the nodes allows for obtaining different values of the evaluation parameters. The application stores every 5 seconds a new state, while a set (number of nodes) of measurements requires 2 minutes. Moreover, the Kubernetes Customization Framework automates the entire measurement cycle.

C. Results

The graphs on the left side of Figure 7 depict the CPU and memory usage of the approaches with an increasing number of application replicas. In some ways not unexpected; the centralized approach is not directly affected by the number of nodes. The number of application replicas must be significantly higher for the database to show some effects. There is undoubtedly a need to investigate the required network traffic in this solution. Different behavior is visible in the storage latency depicted in Figure 7; as each request needs to be processed in sequential order, the time increases.

The other two approaches show different behavior regarding their CPU and memory usage. Significantly the distributed approach is affected by the increasing number of application replicas. The reason is the increased effort to find consensus between the pods. Why the hybrid approach does not show similar behavior is not completely clear. One assumption is that Hazelcast uses many resources out of the box as it is built upon JAVA and is instead not affected later. On the other hand, the storage latency is constant in both approaches, as the underlying Raft algorithm is better optimized.

[4]https://prometheus.io/docs/introduction/overview/

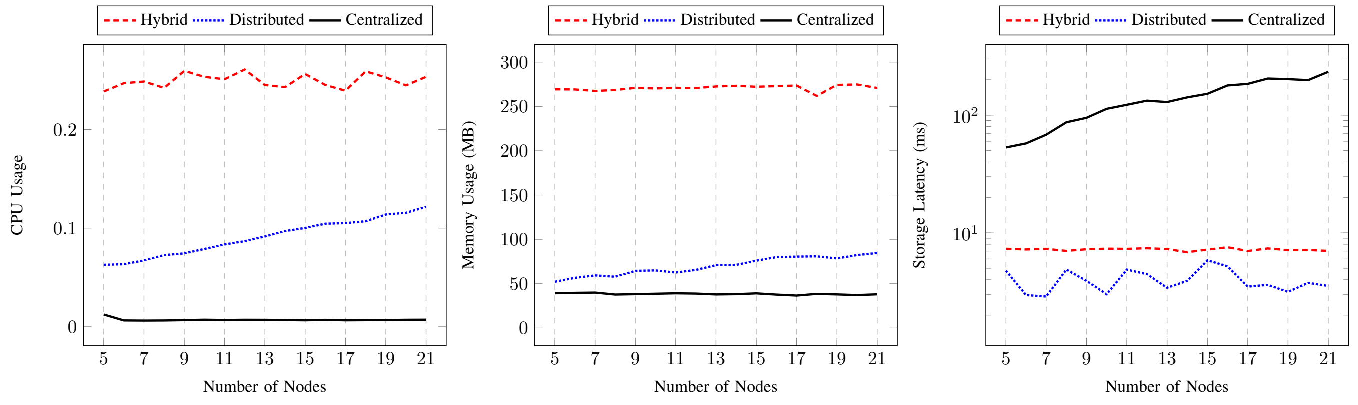


Fig. 7. The two graphs on the left show the three approaches' CPU and memory usage measurement results depending on the number of application replicas. On the right side, the latency is depicted to store a state in each approach with an increasing number of application replicas.

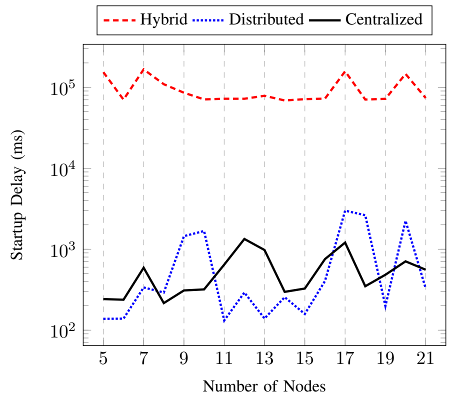


Fig. 8. Startup delay depending on the number of replicas

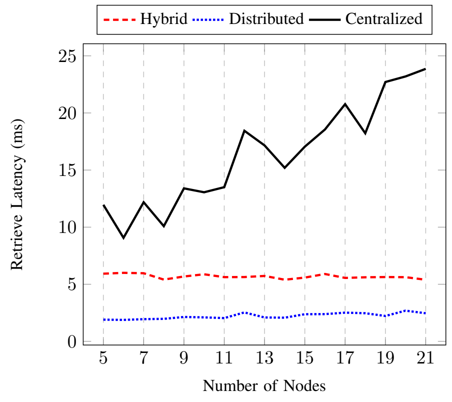


Fig. 9. Retrieve latency depending on the number of replicas

The results in Figures 8 and 9 show some interesting behavior. While the startup time in the centralized and distributed approach is somewhat unreliable, the hybrid approach is constantly high. The latter effect is explainable as the CP subsystem always waits until all nodes are connected, while the other approaches do not have an ordered startup sequence. oo many starting nodes lead to traffic jams, supplemented by the network, especially in the centralized approach. The startup time is the same for a new start or after a node failure.

The effects visible in Figure 9 are similar to the storage latency. The hybrid and the distributed provide a constant retrieve latency, while the centralized shows an increase. Again, the startup sequence could be the reason. A deeper analysis is required of why the storage and retrieve latency show significant differences in their absolute timing measures.

VI. Discussion

The paper compares different state preservation solutions applicable for containerized stateful applications distributed at the edge of the network. As introduced in Section III, there are several causes for state loss in such an environment. Notably, the volatile storage in containers or simple PV in pods is not sufficient to counter, for example, node failures. Only persistent storage solutions, either centralized or distributed, can fulfill the design rationale in Section III-D.

The three prototype implementations represent examples of persistent storage solutions. While the chosen implementation technologies are still specific, the obtained results show apparent differences in their performance. Other authors such as Bakhshi et al. [26], [27], or Netto et al. [21] reported similar behaviors in their solutions. In addition, the evaluation identified some effects that require further investigation.

There is a need to create an overview of which approach on the spectrum can be used for specific use-cases, supported by clear decision criteria. The findings indicate that there will not always be a one-by-one relation between a persistent storage solution and a use case. One reason is that the requirements allow several persistent storage solutions along the spectrum for some use cases. Therefore a systems architect needs criteria to decide which approach is most suitable. Potential criteria also employed in other studies [7] are, for example, failure tolerance, the maximum amount of nodes, resource constraints on the solution (e.g., edge devices), or the involved design complexity when building the system.

Supporting this ambition could be the identified evaluation parameters in Section III-I. The chosen parameters in the evaluation have shown the prospect of building the foundation for comparing other solutions. Further relevant parameters could be the network traffic and the systems fault tolerance.

VII. Conclusion

The paper presents insights into persistent storage solutions for stateful applications in distributed environments such as edge computing or CPS. A design rationale for such systems is introduced based on identifying possible causes of state loss in containerized applications. This rationale led to implementing three prototype instances, one centralized, one distributed, and one hybrid. The evaluation of the prototypes shows no one-to-one relationship between a particular persistent storage solution and an intended use case. In addition, the evaluation parameters used are suitable for comparisons between different solutions. Future research steps should expand the comparisons between different persistent storage solutions by implementing further prototypes. Another goal is to examine new evaluation parameters that categorize the respective solution better. Such categorization allows a system architect to choose the right solution for their application based on the given requirements.

References

[1] E. A. Lee, "Cyber Physical Systems: Design Challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 363–369.

[2] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1997.

[3] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[4] S. A. Boyer, *SCADA Supervisory Control and Data Acquisition*. USA: International Society of Automation, 2010.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 13–16.

[6] W. Steiner and S. Poledna, "Fog computing as enabler for the Industrial Internet of Things," *e & i Elektrotechnik und Informationstechnik*, vol. 133, no. 7, pp. 310–314, Nov. 2016.

[7] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *Journal of Systems and Software*, vol. 175, p. 110924, 2021.

[8] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[9] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How To Make Your Application Scale," in *Perspectives of System Informatics*, A. K. Petrenko and A. Voronkov, Eds. Cham: Springer International Publishing, 2018, pp. 95–104.

[10] 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.

[11] 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.

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

[13] N. Chervyakov, M. Babenko, A. Tchernykh, N. Kucherov, V. Miranda-López, and J. M. Cortés-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.

[14] 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.

[15] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, nov 2002. [Online]. Available: https://doi.org/10.1145/571637.571640

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

[17] 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.

[18] H. Netto, C. P. Oliveira, L. de Oliveira 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.

[19] 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*, V. Jain, G. Chaudhary, M. C. Taplamacioglu, and M. S. Agarwal, Eds. Singapore: Springer Singapore, 2020, pp. 341–351.

[20] E. Kristiani, C.-T. Yang, Y. T. Wang, and C.-Y. Huang, "Implementation of an Edge Computing Architecture Using OpenStack and Kubernetes," in *Information Science and Applications 2018*, K. J. Kim and N. Baek, Eds. Singapore: Springer Singapore, 2019, pp. 675–685.

[21] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. Sá de Souza, "State machine replication in containers managed by Kubernetes," *Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017, special Issue on Reliable Software Technologies for Dependable Distributed Systems.

[22] E. Kristiani, C.-T. Yang, Y. T. Wang, and C.-Y. Huang, "Implementation of an Edge Computing Architecture Using OpenStack and Kubernetes," in *Information Science and Applications 2018*, K. J. Kim and N. Baek, Eds. Singapore: Springer Singapore, 2019, pp. 675–685.

[23] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 00 058–00 063.

[24] 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.

[25] 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)*, 2016, pp. 202–211.

[26] Z. Bakhshi, G. Rodriguez-Navas, and H. Hansson, "Using UPPAAL to Verify Recovery in a Fault-tolerant Mechanism Providing Persistent State at the Edge," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–6.

[27] ——, "Fault-tolerant Permanent Storage for Container-based Fog Architectures," in *2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, vol. 1, 2021, pp. 722–729.

[28] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky, "Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing," in *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2014, pp. 325–329.

[29] (2022) Docker - build, ship, and run any app, anywhere. [Online]. Available: https://www.docker.com/

[30] (2022) Kubernetes documentation. [Online]. Available: https://kubernetes.io/docs/home/

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

[32] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*, 1st ed. O'Reilly Media, Inc., 6 2020.

[33] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34.

[34] P. Denzler, D. Ramsauer, T. Preindl, and W. Kastner, "Communication and container reconfiguration for cyber-physical production systems," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–8.