Verifying the timing of a persistent storage for stateful fog applications

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

***Abstract*— In this paper, we analyze the failure semantics of a persistent fault-tolerant storage solution for stateful fog applications. This storage system is a container-based solution that provides data availability and consistency in a distributed container-based fog architecture. We evaluate the behavior of this storage system with a formal model that includes all the important time parameters and temporal aspects of the solution. This allows us to verify data consistency and other fault-tolerance properties of our system model while considering application startup latency, together with synchronization intervals and delays. We prove that the solution can tolerate failures at application, node, communication and storage level with the ability to automatically recover from failures and provides data consistency within the synchronization delay defined as t time units, which we can calculate for a given system configuration.**

I. INTRODUCTION

Providing persistent storage in container-based architectures is an ongoing research challenge [1]–[3]. This service is crucial for stateful applications, in which the behavior of the application depends on the state history, and therefore, the state needs to be stored reliably and must be readily available.

Deploying stateful applications in container-based and container orchestration architectures can be achieved relying on different methods and tools, such as a *deployment controller* [1] or even using third-party solutions like *Ceph storage* [4]. These solutions are among the few cloud-native approaches to solve the stateful application deployment issue.

However, these solutions also suffer from certain limitations when deploying applications in the vicinity of edge devices, like it happens in fog and edge platforms. The main issues are that: (1) it is challenging to access states when a component fails in these solutions because the states are kept outside of the cluster [5]; (2) the latency might be high since they are implemented in the Cloud; and (3) they do not consider data consistency between different nodes in the cluster nor different replicas of the applications.

A solution to implement local persistent storage in container-based fog architectures, with proper support to data consistency, was recently published [6]. This solution provides a distributed data storage called Replicated Data Structure (RDS) that is instantiated locally in each node of the cluster. There is also a Storage Container (SC) in each node. The SC is a dedicated storage application that provides data synchronization and consistency between nodes. The consistency protocol used in SCs is the RAFT protocol [7].

This solution was verified to fulfill fault tolerance and eventual data consistency properties, with some simplifications of the timing performance [8]. In this work, we build on that work and question how the delay and latency parameters of the solution (application startup latency, data synchronization delay, etc.) will affect very critical attributes like data accessibility and data consistency. For that purpose, we extend the modeling of the persistent storage solution for container-based stateful applications [6] by adding the indicated temporal aspects. The aim of this evaluation is to verify whether the system fulfils (1) data availability after component (Application, SC, Node) failure, considering application and SC startup latency, and (2) data consistency in the cluster within a tolerable synchronization delay t.

The remainder of this paper is structured as follows: In Section II we provide a brief background of the application in our use-case. In Section III we elaborate on the problem statement according to the system model, our persistent storage solution and timing parameters. In Section IV, we study the related works on how concurrency and delay issues are addressed in the literature. We specify our system model parameters and specifications in Section V. In Section VI, we verify our system model considering the temporal aspects. We conclude our work in Section VII.

II. BACKGROUND OF THE APPLICATION

The stateful application that we study as a use-case in our container-based fog architecture is a robotic application implemented with the Robot Operating System (ROS) [9], as presented in [6]. This application is a navigator robot that constantly moves towards newly generated goals, while avoiding obstacles.

The essential components of a ROS application are ROS packages, ROS nodes and topics. Communication among ROS nodes is implemented with a Publish/Subscribe mechanism via shared topics; where a *topic* in ROS is the communication module over which nodes exchange messages [9]. There are three different design options to containerise our robotic application that are explained in [6], namely (1) putting all ROS nodes in one container; (2) allocating the ROS nodes to a number of different containers; and (3) putting each ROS node in a single container. Design model (1) will turn into a huge and heavy container and design model, whereas (3) results in a huge number of containers, especially when the robot system is complicated with a large number of nodes. Therefore, we choose design model (2) and put ROS nodes that are working together in the same container. Containers are the applications in our system model since our ROS application is encapsulated in containers.

III. Problem Statement

In order to understand the conceptual construct of the persistent storage solution, we need a suitable representation of the involved entities. For this, we provide a model of the solution which comprises related components, resources and their interactions as well as their data consistency model in Section III-A. Then in Section III-B we formulate the timing problem and define the properties that we need to check. For the sake of clarity, in Table I we summarize the notation used throughout the paper.

A. Container-based Persistent Storage System Model

In this section, we explain the system model of the container-based persistent storage solution. This solution provides persistent and fault-tolerant data storage for containerized stateful application that require local and distributed persistent data storage [8]. To better understand our system model, we explain the concept of stateful application and data eventual consistency in the context of our system model.

Stateful vs stateless application: the most important reason why we proposed our solution is dealing with stateful applications. Such as the majority of control applications, stateful applications are reliant on their history as captured by their states. These states need to be stored for future use by the same and/or other applications. To better understand the difference between stateless and stateful applications, we define an application as a function. In stateless applications the behavior of the function[1] for a given value of the input variable x, no matter how many times the function is called, the result of the function will be the same. This can be defined as

$$\forall x \in X, \text{ and } t, t' \in T : f_t(x) = f_{t'}(x) \tag{1}$$

Where in Equation 1, x is the variable (like the request) and t and t' are the time instants of calling the function. This is like a read request to a fixed identity number or scanning barcode of an object.

However, in stateful applications, the result of the function is reliant also on previous calls to the same function (or even on the time when the call is made). This history is captured by state variable. This can be defined as

$$f(x) = f(x.state)) \tag{2}$$

Where in Equation 2, x is the variable (like the request), and $state$ is the current state value of the system; for instance, the current location of a moving object. In stateful applications, the current state and the time when the function is called affect the result of the function and therefore the behaviour of the application.

System model: A cluster in our system consists of a set of $N = \{N_1, N_2, ..., N_n\}$ distributed nodes. In each cluster there is an administrator node AN which works as an orchestrator in the cluster. Each node is a virtual or physical computing node. Nodes are characterized by their computing capacities $R^C_{N_i}$, Memory $R^M_{N_i}$, Storage $R^S_{N_i}$. Nodes host one or more applications. We consider that the number of all applications in the cluster is k. A node also contains an updated list of applications running locally on it. We use App or A interchangeably to refer to an application.
Let $\mathcal{A} = \bigcup_{i=1}^{k} A_i$ be a set of applications to be deployed in fog nodes. An application is characterized by its demands for computational resources, like, $R^C_{A_i}$, Memory $R^M_{A_i}$, Storage $R^S_{A_i}$. In addition, an application has a unique id and a storage tag ${A_i}^{S^t}$. Each application has a set of tasks to execute, $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$.

Applications in our system are fault-tolerant and are automatically restarted upon failure [6]. However, they suffer from a volatile storage issue. i.e., their volatile storage cannot be retrieved after application failure.

Persistent Storage Solution: To solve the volatile storage issue, we introduce the use of a local data storage system in each node, called Replicated Data Structure (RDS). After execution of each of the tasks, the application writes its state in the volatile storage and in the RDS, respectively. Applications working in the same node can directly and locally access RDS. However, if a node fails, the local data might not be retrievable. As a remedy, we need to provide a mechanism to distribute data in the cluster. For communication purposes and data synchronization throughout the cluster, we propose the use of separate Storage Containers (SC). This is to reduce the load of synchronization from the application.

The role of SC in each node is to keep the data in the RDS of each node synchronized and consistent. SC is a container itself, so it supports the built-in fault tolerance mechanism provided by containerization. Each node has at least one SC that works as a storage application. SCs in the cluster use the RAFT consensus protocol [7]. RAFT is an election-based consensus protocol that provides data consistency and high availability.
An SC is characterized by its computational demand $R^C_{S_i}$, Memory $R^M_{S_i}$, Storage $R^S_{S_i}$. An SC also has an unique id and RAFT_Status. This is explained in detail in [6].
Figure 1 is an illustration of our system model architecture. This figure shows the different entities of the system and their interactions at a high-level view.

Data Consistency: In a distributed system consisting of N nodes, if all concurrent read actions of a specific data element from any given number of nodes is guaranteed to returns the same value, then the system satisfies the property of strong consistency [10]. This means that when a data value is changed in one node all other read requests from other nodes must be blocked until the changes are propagated to all the nodes. This is to guarantee that the next read operation returns the same value. Figure 3 shows how strong data consistency can be obtained in our system. In this figure, we can see that a state value, which is currently X, is changed to X' (by application App_1). The new state X' cannot be read by other applications until a full synchronization (Send, Receive, Commit) occurs in the system. The main advantage of strong consistency is that it guarantees consistent data. However, this comes at the cost of blocking read operations

[1]We assume that applications can be represented as single or multiple iterative calls to a function, i.e. that we have read-execute-write semantic.

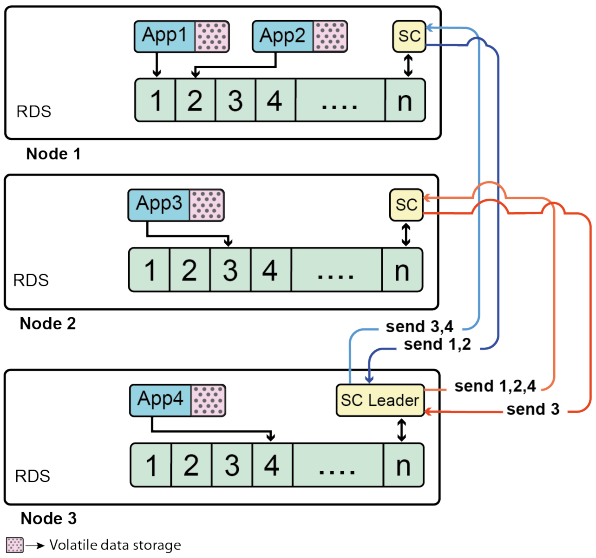

Fig. 1. System model artchitecture

TABLE I
SYSTEM NOTATIONS

Type	Parameter	Description
Application	$\mathcal{A}$	Set of applications to be executed
	A_i	Application
	$R^C_{A_i}$	CPU demand of an application
	$R^M_{A_i}$	RAM demand of an application
	$R^S_{A_i}$	Storage demand of an application
SC	$R^C_{S_i}$	CPU demand of a Storage application
	$R^M_{S_i}$	RAM demand of a Storage application
	$R^S_{S_i}$	Storage demand of a Storage application
Node	N	Set of fog nodes
	N_i	Fog node
	AN	Administrator Node
	$R^C_{N_i}$	CPU capacity of a fog node
	$R^M_{N_i}$	RAM capacity of a fog node
	$R^S_{N_i}$	Storage capacity of a fog node
	$R^C_A N$	CPU demand of an administrator node
	$R^M_A N$	RAM demand of an administrator node
	$R^S_A N$	Storage demand of an administrator node
Time	t^{app}	Average Container (App or SC) startup/restart latency T_{ad} and T_{sd} in the UPPAAL model
	t^S	Interval periods SC needs to contact SC leader
	d^N	Communication delay between different nodes

while synchronizing.

In contrast, eventual data consistency is non-blocking. In eventual data consistency, when no updates are executed on the data, all read operations to that data will eventually return the same value, yet it is possible that in some intermediate states, simultaneous read operations in different nodes will return different values (normally a value previously written). The non-blocking behavior of eventual consistency offers high availability (low latency) at the risk of returning older data in terms of time and version. Time refers to the time when the latest write is accessible for a read request and version refers how many versions of the written data a read request could have access to, i.e. in the worst case, how many versions back in time from the most recent write the value read could be.

Figure 2 shows eventual data consistency in our model. In

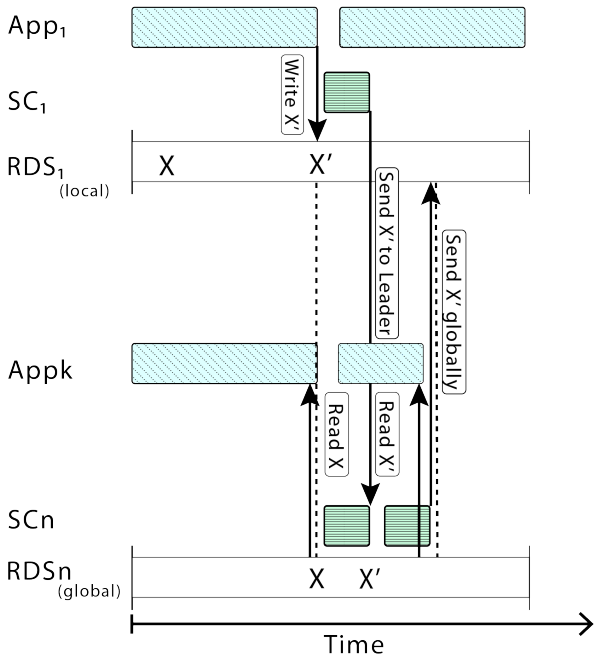

Fig. 2. Eventual data consistency in our system model

this case we can see that data are queued to be processed and read. By leveraging eventual data consistency, completion of application execution is fast and for systems like ours, with multiple fault-tolerant levels, it helps to protect data and reduces time loss.

Figure 3 shows an abstraction of strong data consistency in our model. As shown in this figure, strong consistency comes at the cost of high latency due to its blocking behaviour. In addition, when the number of applications and nodes scale up in the system, this blocking time will increase. The other disadvantage of leveraging strong data consistency in our system model is that when a failure occurs, i.e., SC or Node failure, the system stops working until the failed component is back and the new state is committed to all the nodes in the system.

Still, in some other systems it might be required to ensure strong data consistency. This derives from the application requirements.

B. Timing Problem Formulation

To perform timing analysis of our system model we need to account for timing requirements of the different system entities, i.e., application start-up latency, application execution time, SC startup latency, SC synchronization time, etc. These parameters makes it possible to calculate an upper bound (i.e. worst case estimation) of the synchronization time. We consider synchronization time as the time to fetch, send, commit, receive data between different distributed SCs plus network latency.

We also need to know how the application, SC and node failures affect the upper bound time of the synchronization and identify problematic scenarios. For instance, circumstances in which a SC fails several times and the

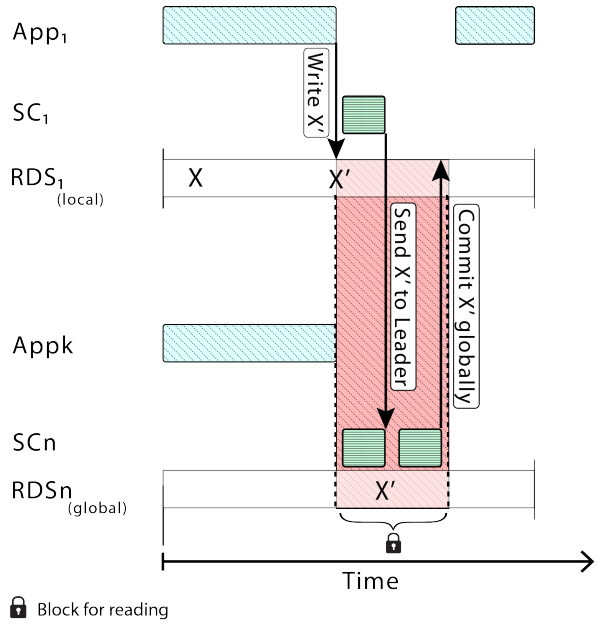

Fig. 3. Strong data consistency in our system model

synchronization cannot happen. Another example can be when the application execution is much faster than the storage container. In this case the system might face an issue of outdated data. Combination of these failures with other application failures may lead to more complicated scenarios that also need to be understood with the help of formal analysis.

In our analysis, we categorize the timing aspects into two different classes: (i) the effect of adding temporal delays on data synchronization and data consistency, and (ii) the effect of adding temporal delays on data accessibility after different failures. Specifically, our analysis intends to prove that our solution satisfies the following properties:

P1 (Eventual) Data consistency is guaranteed within time t.

P2 Application, SC and node startup latency do not affect data availability in the cluster.

Note that in order to show that our model satisfies each one of the stated properties, we need to further break them down into more descriptive formulas that can be comprehended by our modeling and verification tool (UPPAAL). This also helps us get a complete understanding of the behaviour of the model in different states, times and events.

IV. Related Works

Delays caused by component failures may lead to data consistency issues by means of two main mechanisms: concurrency problems of read/write actions and delay in synchronization [11], [12]. In this section we study these two causes, to investigate how delay and concurrency are dealt with in the literature and by other technologies. Since this topic in distributed data storage systems is a broad concept, we narrow our study down to investigate more and study about the application requirements and the RAFT consistency protocol [7].

First, we briefly explain how RAFT manages delay and concurrency in processing states. Then, considering our application explained in Section II, we discuss how concurrency and delay are dealt with in ROS.

A. Concurrency and Delay in RAFT

We need to know what happens when an application writes two or more different states locally in the replicated data structure (RDS) before data synchronization between nodes via SC happens.

In this case, when SC sends the states to the SC leader, the states are not the latest version or there might be a case in which some states are missed since the latest one will overwrite them. Studying this type of situation in RAFT, helped us figure out that there is no single state that can be lost in RAFT, and all logs are stamped with terms and will be processed eventually. However, on occasions when there is a delay in data transfer due to communication failure, node partitioning, or the concurrency of receiving different states from the client the RAFT leader will put the states in a queue based on the time of the state log. Message queuing is to ensure that all the states, no matter if they are in the latest state or outdated, are processed.

B. Concurrency and Delay in ROS

Let us study the issue of data consistency due to delay or concurrency in ROS. When using pure ROS in an implementation of a Robotic application, the topics are queued in stack of messages named *'Callback'* [9]. The callback is a function responsible to handle the topics so, different nodes can share their messages using subscribe and publish topics method. When the execution time of the callback function is short enough (shorter than the execution time of a task before publishing the next topic) the callback can handle all the topics in order before the next one is received in the queue. However, this ideal scenario does not always take place in complicated multi-node ROS applications. When different nodes want to access topics in the callback at the same time, the callback function needs to execute topic sharing for different nodes that their execution times overlap to another [13]. In this type of situation, the callback function executes the topics based on their priority in the callback message queue.

Processing messages in the callback based on their state in the queue helps to manage all the messages that overlap with other messages. However, for processing critical topics, like observing an obstacle in the vicinity of the robot, this method pushes processing of critical messages back in the queue and this might result in hazardous behavior of the whole application. To solve this issue, the message priority based on the topic criticality is introduced that prevents the critical messages to stay back in the queue. The critical messages will be processed with higher priority and their execution will be highly prioritized. This is implemented using *'NodeHandles'* [14] in ROS that create different message queues according to the tasks' priority.

C. Discussion

In modeling our solution we consider that the complexity of ROS application implementations underlies inside the containers. This means that since we are encapsulating our ROS application in a container we are going to use the same methods in ROS to manage concurrency and delay in message exchanges. However, we also need to consider that by adding SC as another layer to our system to manage storing and distributing state data in the cluster, this extra layer is also prone to failure and probable delay in communication and synchronization. Meaning that even if we consider that messages with higher priority based on their critical state (i.e., observing an obstacle) are going to be processed faster in the callback queue, if the SC, the node, or communication between the nodes fails, the critical messages are not going to be received and processed on time.

For this type of failure and state loss, we can implement a fail-safe mode as follows: if ROS nodes do not receive a topic from the node with the critical state for a defined threshold, then the Robotic application will stop working and notify a higher entity, which will handle the fail-safe mode.

According to this discussion we draw a conclusion regarding the synchronization method: after reviewing the methods used in RAFT and ROS, we decide that we continue the modeling to be a non-blocking synchronization method. We are aware that it is not going to provide us with strong consistency, however, we need to confirm our system model will satisfy the consistency properties defined in **P1**.

V. System Model Specifications

There are some temporal aspects that are crucial and that we need to incorporate into our system model: (1) startup latency period; (2) synchronization/communication delay; (3) accumulative application execution time; and (4) synchronization frequency. We further explain each of these parameters as follows:

(1) The startup time latency refers to the sum of two parameters: container creation and deployment time. The average startup latency differs for each type of container and, for this reason, application containers and SCs have different startup latency.

(2) The synchronization delay refers to the total time required for an SC to communicate to the RAFT leader and send/receive states and commit/fetch them. The average time of communication between nodes (and RAFT protocol delay) is also considered along with this time.

(3) The accumulative application execution time refers to the sum of the execution time of each task in an application, and is denoted as C.

We have two approaches for measuring deployment (creation plus startup latency) and execution time of the containers. The first one is using a Python3 code than measures deployment and execution time for n rounds and provides min, average and max time. The second one, is using Docker Container Inspect to measure start and end time of a container. In both cases we need to terminate and remove the container after execution to see the results. However, there are other methods like using the "Sysbench" tool that measures execution time, CPU, RAM usage of the container as well.

There are different parameters that affect the result of execution time of an application. For instance, the input load/size, computational resources of the device the application is running on, etc. Our measurement is simple. We make an assumption that all the nodes in the cluster have the same computational resources. We also consider that input of the application has a fixed load/size. This is to simplify the current measurement since the objective of this work is not measuring performance parameters and we just need an estimation of startup latency and execution time.

After running the application for n rounds in one device, we measured the minimum, average and maximum execution time. We consider the application execution time as the *maximum time* calculated in our measurements.

(4) The synchronization frequency period is the interval period between each synchronization occurrence. By using this parameter, we can evaluate the level of data consistency in our model.

VI. Verification of the Proposed Solution

We used the UPPAAL [15] verification tool to model, analyze, validate and formally prove the functionality of our proposed solution in [8]. We stick to the same verification tool and investigate the effect of adding temporal aspect to our model using UPPAAL and by defining a new set of properties.

A. Expanded Criteria

In [8], we defined In-scope and out-of-scope parameters and metrics to narrow the borders of our verification. Here we want to add the parameters and measurements that were excluded as out of scope in our previous work to have a thorough evaluation of our solution. The expanded criteria in this work are as follows:

- Data consistency level considering the synchronization time between SCs and RDSs
- Application and SC startup latency and their impact on the accessibility of states
- Timing effect of synchronization delay to application and SC functionality

It still remains to evaluate the scalability of our solution. This shall be covered in future work.

B. Temporal aspects added to the application model

The first state of the application is the deployment state where the application creation and deployment time are considered. We use t^{app} to denote the sum of these two times.

When an application is deployed, it starts to execute the set of tasks as τ explained in Section III. The complexity of the tasks are packaged into the containers so, we consider the accumulated task execution for each application. To be on the safe side we consider the maximum of all, denoted C.

TABLE II

RESULT OF EXPERIMENT WITH PROPERTIES

Property	Formula	Result
P1-1	A[] O.SendtoSC imply (O.t<=T)	Satisfied
P1-2	E<>O.SendtoSC and (O.t>T)!	Satisfied
P1-3	A[] O.SendtoSC imply (O.t>0)	Satisfied
P2-1	A[] A1.App_execution imply rep_data_struct[1] ==last_committed [1] or rep_data_struct[1] ==pre_last[1] and x<C	Satisfied
P2-2	A[] A0.App_execution imply rep_data_struct[0] ==last_committed [0] or rep_data_struct[0] ==pre_last[0] and x<C	Satisfied

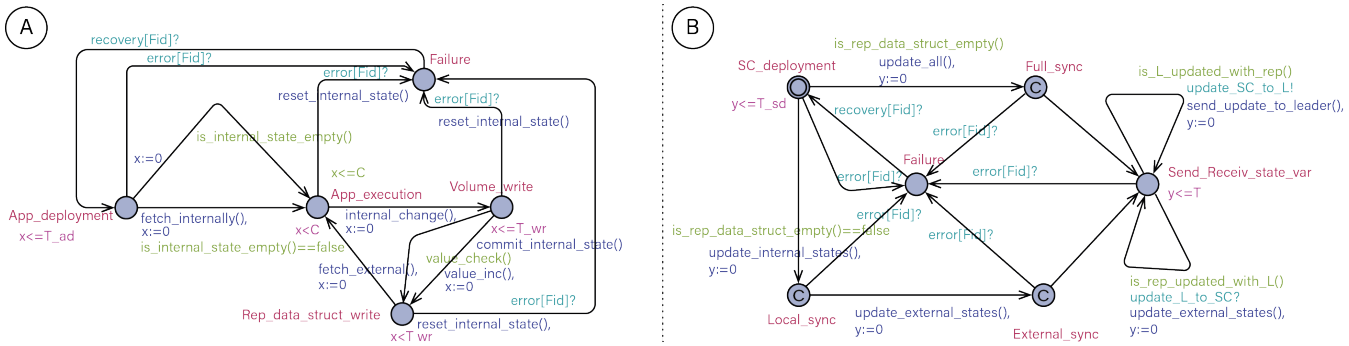

Fig. 4. Application and Storage Container Automata

The next stage of an application after application execution is writing its state locally both on the volume and the RDS. The average writing time for an application is also added to our system model design.

Algorithm 1. Application Functionality Pseudocode

```
τ = {τ1, τ2, ..., τn}  //Task set
J = {j1, j2, ..., jn}  //External states
id := App_id
Fetch internal_state ()

While Running Tasks()
{
for τ = 0; τ <= n − 1; τ++{
  Run τ()
  Wait () <= C
  if τ >= n or t <= C
Commit self_state()
//Commiting Application's state locally in RDS
 }
 if (read_external(true)){
Fetch external_state()
   for j = 0; j <= n − 1; j++{
       } else{} }
```

C. Temporal aspects added to the SC model

The SC is created to serve two purposes (1) creating RDS and (2) data synchronization between RDS on different nodes. The first one is a one-time task and the second one is a constant task to perform synchronization. We use t^{SC} to denote the sum of creation and deployment time for the SC. For the synchronization time, we have two parameters, synchronization delay and synchronization frequency, shown as T and F respectively in the UPPAAL model.

Algorithm 2. Storage Container Functionality Pseudocode

```
if (rep_data_struct.empty())
  Full_synchronization() = {};
//Fetch data from SC leader
}else{
  for (id=0; id<=num_apps−1; id++){
    Local_synchronization (id)}
//Fetch data from RDS
 for (j=0; j<=num_ext_apps−1; j++){
      External_synchronization()}
//Fetch external states from SC leader
While SC_working (true){
 for (id=0; id<=num_apps−1; id++){
  if (rep_data_struct[id]=!l_rep_data_struct_log[id
      ]){
//send update to leader by appending the log
    Synchronize_internal_state(id)
    time_out() < t }
}for (j=0; j<=num_ext_apps−1; j++){
  if (l_rep_data_struct[j]=!rep_data_struct[j]){
//update external state
    Synchronize_external_state(j)
    time_out()<t  } }
```

D. Application and Node Failures Behavior and Temporal Aspects

In order to model the behavior of the system in presence of failures we added two automata to the system modeling: (1) Application failure and (2) Node failure.

(1) Application Failure: Using the Application failure automaton, we want to verify that Applications and SCs on each node are mortal and still fault-tolerant. Meaning that they are prone to failures but they will be recovered automatically. Figure 5-A shows the UPPAAL automaton, modeling the application failure. As is shown in this figure, activation of Application and SC failure is done through channels in UPPAAL. This automaton starts from the initial location `Working` that shows that both Applications and SCs are working.

When an error occurs for any application (including SCs) the automaton moves to `App_Failure` location. The

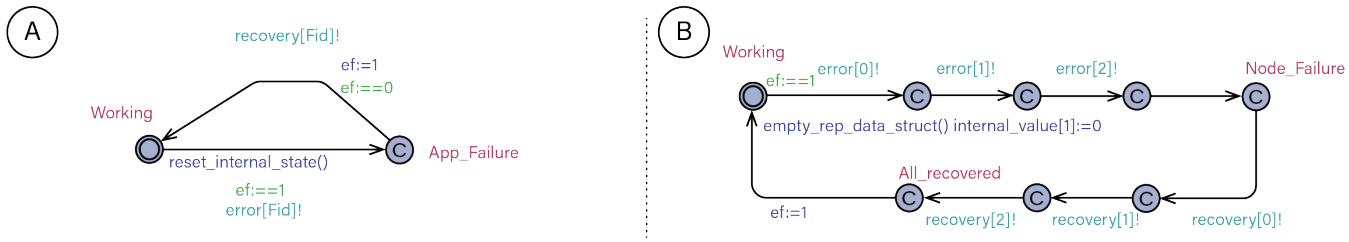

Fig. 5. Application and Node Failure Automata

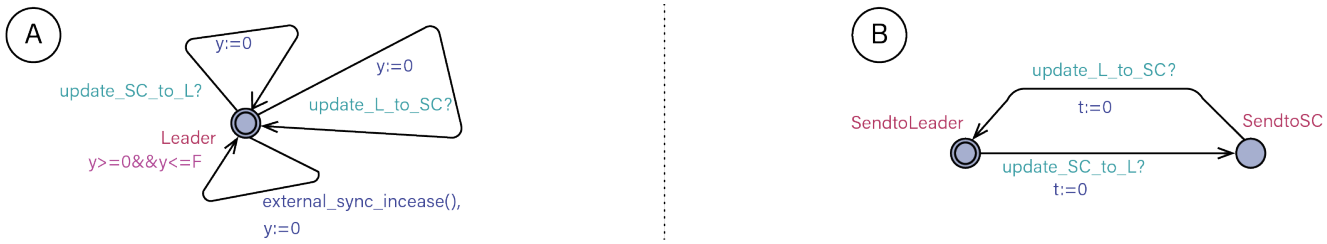

Fig. 6. Leader and Observer Automata

`error!` signal is then sent to an application or SC. This forces the automaton in which the signal `error!` is received to move to the `Failure` location. Figure 4-A and 4-B show the application and SC automata respectively, including the failure states.

For each of the Apps and the SCs, a separate automaton is defined, since failure of one App/SC (automaton) should not affect the other App/SC automata. As shown in figures 4 and 5 (a), when a failure is triggered in the application failure automaton, it is the `id` indicated in the channel that determines which container (App or SC) must transit to the failure location. When an App or a SC fails and reaches the failure state it must transition to its initial state afterwards. However, when an application fails, its own state in its local variable or `internal state` must be turned to 0 (empty) as a consequence of the failure.

(2) Node Failure:

The node failure automaton is to verify that when a node fails (i) all applications and SC(s) working on it will fail as well and (ii) RDS as the local storage will also fail and all local states/data will be lost.

Like the Application failure automaton the node failure automaton also starts from the `Working` location. When a node failure is activated with the `error` signal, each of the App(s) and SC(s) in the same node will reach the failure state in their own automaton. In the node failure automaton this is done through committed locations (which prevents any delay before taking the next transition) to avoid time elapsing. Since when a node fails all working entities on it will fail. These committed locations activate a sequence of failures for App(s) and SC(s), one by one with no time elapsed between them.

When all the App(s) and SC(s) in the node fail, then they all will start from their initial state, after recovery. The recovery is also done using committed locations and each recovery channel activates a recovery channel in the relevant automata (App or SC).

E. Temporal Aspect added to the Leader model

The Leader automaton is a simplified single location automaton that models the RAFT leader with clock `y`. Figure 6-A shows the UPPAAL model of this automaton. There are two synchronization channels in this automaton `update_L_to_SC!` and `update_SC_to_L?`. The first one is triggered when sending update states to the `SC` and the second one is triggered when receiving state updates from `SC`. There is also an update edge in `Leader` automaton that is activated when the value of external(s) states are updated.

Time `F` is defined to show the lower bound and upper bound of staying in the location `Leader` in Leader automaton.

F. The Observer Automaton

In order to verify the timing property **P1**, we need to introduce an additional automaton (see Fig 6-B) which is composed with the rest of the system. This automaton acts as an external observer and does not interfere with the model previously described.

This automaton starts from location `SendtoLeader` with clock `t` set to `0`. When the `update_SC_to_L?` signal is received from the `SC` automaton the automaton moves to the `SendtoSC` location and reset the clock `t`.

From this point on, the clock value again increases with time. When the signal `update_L_to_SC?` is received from `Leader`, the Observer automaton switches to the initial location `SendtoLeader` and reset the clock `t`.

To measure the synchronization time in the SC automata we use an Observer automata which acts as a clock precision monitoring entity. This helps us measure the upper bound and lower bound of the synchronization clock of the SC automata. The Observer automata is defined as O in our modeling system declaration. **P1-1** is a property in the first category of properties (P1) we want to examine in this work.

G. Properties

Table II gives the formula and the status of each property in our extended model. In this subsection we explain each of these properties with an interpretation of their result status.

P1 refers to the data consistency properties within the defined synchronization time.

In **P1-1** captures that T is an upper bound of clock t in the synchronization process.

P1-2 captures that it is not possible that the upper bound increase more than T. This property is not satisfied, which captures the impossibility of the clock t to go beyond the synchronization time defined as T.

To capture that we defined synchronization time not as an immediate action, we defined **P1-3**. Intuitively this property implies that the synchronization time is always greater than 0. Intuitively, it shows that clock t in the Observer automata is always greater than 0 while synchronizing.

P2-1 and **P2-2** are driven from the second category of properties. Intuitively these properties indicate that data is available after failure before the application execution.

VII. Conclusion

In this paper, we extended the modeling and evaluation of our previously proposed persistent fault-tolerant storage solution for container-based architectures [6]. We added temporal aspects related to our work [8] to improve the evaluation of our system. These temporal aspects refer to application startup latency and synchronization delay between nodes in a cluster. We evaluated our solution in presence of these newly introduced timings. The results indicates that our solution provides fault-tolerance and data availability in case of application and node failure. In addition, we verified that the data consistency property is satisfied within t time units in our system model. Where t is the tolerable synchronization and communication delay of the proposed SC. In our future work we will target the scalability and performance of our solution.

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 also from the VINNOVA project 2018-02437.

References

[1] 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.

[2] 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.

[3] 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)*, IEEE, 2018, pp. 00 058–00 063.

[4] L. Mercl and J. Pavlik, "Public cloud kubernetes storage performance analysis," in *International Conference on Computational Collective Intelligence*, Springer, 2019, pp. 649–660.

[5] Z. Bakhshi Valojerdi, "Persistent fault-tolerant storage at the fog layer," Mälardalen University, 2021.

[6] Z. Bakhshi Valojerdi, G. Rodriguez-Navas, and H. Hansson, "Fault-tolerant permanent storage for container-based fog architectures," in *Proceedings of the 2021 22nd IEEE International Conference on Industrial Technology (ICIT)*, 2021.

[7] 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.

[8] 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.

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

[10] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[11] N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei, "Delay-free concurrency on faulty persistent memory," ser. SPAA '19, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 253–264, ISBN: 9781450361842.

[12] O. T. Lee, S. M. Kumar, and P. Chandran, "Erasure coded storage systems for cloud storage—challenges and opportunities," in *2016 International Conference on Data Science and Engineering (ICDSE)*, IEEE, 2016, pp. 1–7.

[13] N. Valigi, "Lessons learned building a self driving car on ros," in *Robot Operating System (ROS)*, Springer, 2021, pp. 127–155.

[14] M. CHIABERGE and S. RAPISARDA, "Ros-based data structure for service robotics applications," 2019.

[15] *UPPAAL Model Checker, UPPAAL Official Website*, `https://https://uppaal.org/`.