Using UPPAAL to Verify Recovery in a Fault-tolerant Mechanism Providing Persistent State at the Edge

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

***Abstract*— In our previous work we proposed a fault-tolerant persistent storage for container-based fog architecture. We leveraged the use of containerization to provide storage as a containerized application working along with other containers. As a fault-tolerance mechanism we introduced a replicated data structure and to solve consistency issue between the replicas distributed in the cluster of nodes, we used the RAFT consensus protocol.**

In this paper, we verify our proposed solution using the UPPAAL model checker. We explain how our solution is modeled in UPPAAL and present a formal verification of key properties related to persistent storage and data consistency between nodes.

I. INTRODUCTION

Containerization and container orchestration solutions initially devised for cloud computing are also useful for implementation of the fog layer, as they provide good portability, scalability, and automatic application deployment, including migration between the fog and the cloud layer [1]. Containerization also saves much resources at resource constrained fog platforms due to its lightweight characteristic.

Cloud-native container orchestration solutions, like Kubernetes, also provide automatic healing for its managed containers, that is, restarting the failed containers and replacing or rescheduling them when they or their hosts fail [2]. In addition to these recovery actions that naturally improve the availability of the containerized applications deployed with Kubernetes, redundancy mechanisms implemented by replicating applications remain the most important Kubernetes feature to improve application availability [3].

Replication of stateless applications can be performed easily as they can be deployed as interchangeable instances. However, the same is not applicable for stateful applications. There are two important issues when it comes to deploying stateful applications using Kubernetes; (1) the state of the failed container is not restored, and (2) the behaviour of a stateful applications is dependent on its current state and therefore, redeploying a new instance without providing the state of the instance it is replacing could lead to unpredictable, potentially erroneous, behaviour. Hence, a synchronization mechanism is required to coordinate the different replicas when redeploying stateful applications.

In a previous publication [4], we proposed a number of strategies to overcome volatile storage issues and achieve data consistency for stateful applications in container-based fog applications. Our solution is based on three principles:

978-1-7281-2989-1/21/$31.00 ©2021 IEEE

(1) Dissociation of data storage from application processes by introducing separate storage containers. This reduces the load of data synchronization from applications by off-loading it to containerized storage applications; (2) Using a replicated data structure to increase data availability and to provide fault-tolerant persistent storage in each node of the cluster; and (3) Adding a consistency protocol to keep the replicated data structures that are distributed in the cluster of nodes synchronized.

In this work we present a formal model of our solution, using the UPPAAL [5] model checker, and formally verify the fault-tolerance and data consistency mechanisms of our solution. The rest of the paper is organized as follows. We describe the design logic of our system in Section II. We explain the limitations of the existing solutions in Section II-C. We continue with a short description of our proposed solution in Section III and then we continue to the verification of our solution in Section IV and review related work in V. We conclude our work in Section VI.

II. PRELIMINARIES

In this section we explain the difference between stateful and stateless applications, the application model we consider in this work, and the temporal aspect of our solution.

A. Stateful vs. Stateless Applications

Figure 1 illustrates the difference between stateless and stateful applications, and the consequence of their failures. As shown in Figure 1 (a) and (b), stateless applications are interchangeable, as for any given input the outputs are identical thus can easily be replaced. However, in stateful applications illustrated in Figure 1 (c) and (d), applications are not interchangeable. They are performed with the context of previous state. In this case failure of one application results in data loss (due to volatile storage) and data inconsistency. As shown in Figure 1 (d) failure of the Robot Application, results in losing state and as well as an undesired output of the Application although it is recovered and redeployed.

We now briefly analyze how this situation is handled in Kubernetes. For stateful applications, Kubernetes provides two solutions: it is possible to deploy stateful applications with "Deployment Controllers" or with "StatefulSet Controllers". In both cases, the state information is stored on a persistent storage (PV) outside of Kubernetes. However, using PV for both these stateful deployments cannot solve data access and availability issues for distributes systems when an application or a node fails.

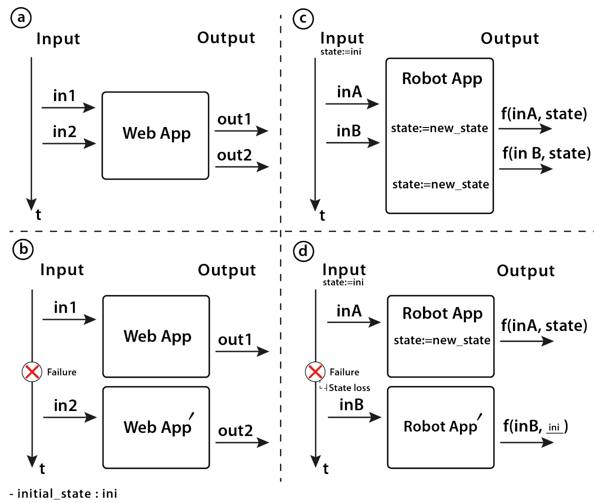

Fig. 1. Stateful versus Stateless Applications.

After a node failure, the connection to PV is lost and the only way to access it to have the failed node back to the network,and thus none of these solutions can provide state recovery. With regards to application failures, the implementation with a Deployment controller causes data loss when a pod (virtual host of a container) fails and the StatefulSet Controller suffers from longer restart time when a container fails [2], [3].

B. Design rationale

In our solution, we consider the use of Kubernetes deployment with an integration of storage containers and secondary labels (key identifier for objects in Kubernetes) together with a consensus protocol called RAFT [6]. In this way, the slow restart issue of the application in StatefulSet deployment is solved by instead using a Deployment controller, which is known to restart the application in the shortest possible time [2], [3].

To guarantee data availability and consistency between distributed nodes we rely on the RAFT consensus protocol [7], particularly because of its good fault tolerance mechanisms and because the service is guaranteed to be provided at the rate of the so-called leader heartbeat, as formally and experimentally proven [8], [9]. In principle, other consensus protocols could also be used with our scheme, but that has not been investigated yet.

Figure 2 depicts a high level block diagram of how a stateful application works in our solution. We consider that after writing the output of execution on the container's volume, the state of the application is also written to a local storage space named "Replicated Data Structure".

Each application has a set of specifications described in our previous paper [4] including id, run time, etc. In addition to those specifications, each application includes a set of tasks, defined as $T = \{t_1, t_2, t_3, ..., t_n\}$. Each of the tasks can be executed once or repeatedly. We assume that tasks defined in the tasks set of an application must be executed based on pre-defined orders. The detail of interaction between the

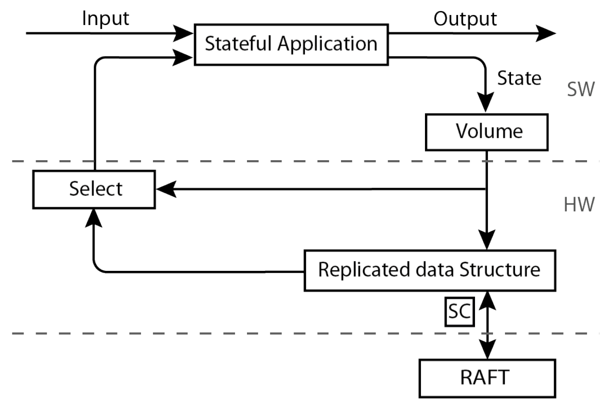

Fig. 2. Application Execution Model.

application and the storage subsystem is explained in Section III. We use the terms App and Application alternatively to refer to a containerized application in this work.

We here summarize the properties of underlying infrastructure, divided into properties that are part of the validation of the modeling, or Model Properties (MP), and properties that are verification of the fault-tolerance behaviour of designed model, which we will call Fault-Tolerance Properties (FTP).

MP1 (Volatile storage): Whenever a containerized application is deployed or restarted, the corresponding volume is empty.

FTP1 (Application failure recovery): Whenever a containerized application fails it is automatically restarted.

FTP2 (Node failure recovery): Whenever a node hosting containerized application fails, all the application running on it are restarted on another node.

C. Comparison with existing solutions

In our previous work [4], we described the problem of persistent storage for stateful applications at the fog layer. We also mentioned that there are some solutions in the literature for the described issue of storing states for stateful applications at the fog layer. However, the existing solutions have some limitations to fit in a fog infrastructure:

1) They are all implemented using cloud storage and our solution aims to provide a fault-tolerant storage system at the fog level [2], [3].
2) Solutions proposed for data consistency between different nodes and applications inside nodes require at least two replicas of each application and all the load of execution is always on one container. (All other containers in the cluster will remain standby, apart from forwarding the task and result to/from the leader [9].)
3) Node failure in a cluster has not been investigated.
4) Data consistency between pods (virtual container hosts in Kubernetes) and containers of different kinds has not been investigated.

Our approach to address the volatile storage issue in container-based architectures in fog platforms is using the advantages of containerization: scalability, self-healing and

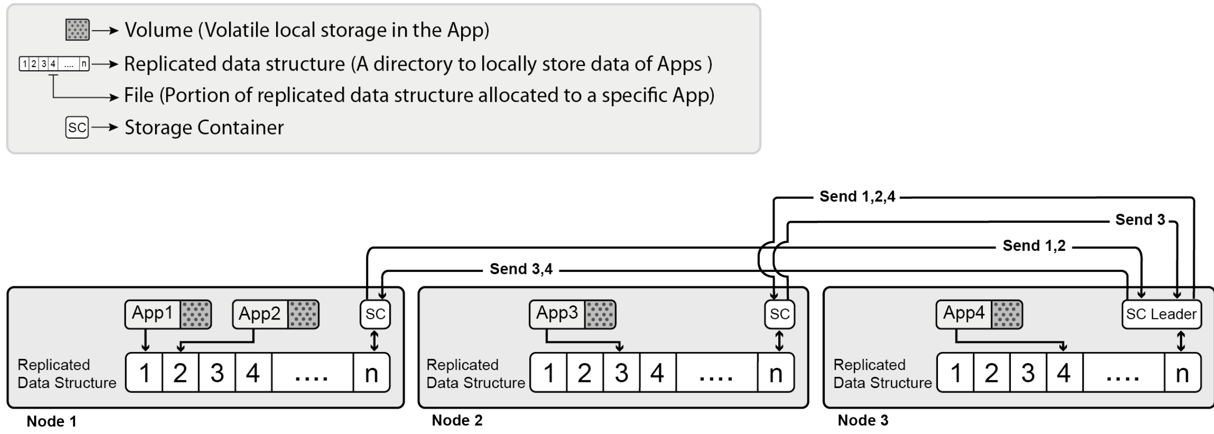

Fig. 3. Proposed Storage Container Schema

portability [4]. In this solution, the use of storage containers (SC) provides suitable mechanisms for:

(1) Outsourcing data storing and data retrieving upon failure to containerized storage mechanisms thereby reducing the application load; (2) Storing data/states of stateful application (execution) locally inside each node; (3) Tolerating failures at two levels, application (software), and node failures (Software and Hardware); and (4) Achieving data consistency between nodes and applications inside nodes.

III. Proposed Solution

Figure 3 is an illustration of the fog nodes, including the applications, SCs and replicated data structure.

Each instance of a containerized application creates a state and updates it after each execution. First, the state is stored locally in the application container volume and then it is duplicated to be stored in the replicated data structure. The replicated data structure is the local storage space or directory that is created by the SC. Each application has a specific space (file) in the replicated data structure directory, with write access to their own file and read access to other applications' file.

A data exchange mechanism is needed to provide exchange of data between different replicated data structures in a cluster. One possible solution is to use the same mechanism that orchestration solutions provide for data exchange between volumes. However, in this case, when a node fails, the replicated data structure on the failed node will be inaccessible. The applications previously working on the failed node can be restarted on another available node in the cluster, but, the replicated data structure newly created on the new node is empty. Therefore, the applications restarted on another node loses access to their previous states. This also causes data inconsistency between different applications and different nodes in a cluster. Data consistency in our work is defined as the operation of updating the latest value of states uniformly as the states change in application execution and is transferred through the network.

Each SC on each node is responsible to participate in data synchronization to achieve consensus and data consistency in the cluster using the RAFT protocol. SC reads the states of all applications located on the same node as the SC, and sends them to the SC which is labeled as the ”Leader”, which can be located in a different node. The SC which is the leader works as the RAFT leader as described in our previous work [4] and the RAFT paper [6]. The SC leader receives states from different SCs (followers) distributed over the nodes in the cluster and then it broadcasts the states to other SCs to make sure all the SCs have the same states.

When an SC receives a state (external state) from the leader, it will write the state to its instance of the replicated data structure, so the application working on the same node can directly read it from the replicated data structure.

When a SC is deployed in a node (either for the first time or if it is restarted due to a failure) it first gathers the states from the applications inside the same node that is hosting the SC. If there is no states in the local replicated data structure on the same node, the SC will contact the leader to synchronize all the states.

We rely on the RAFT consensus mechanism implemented in each SC to ensure that: (1) in case of a node failure, data is available to the applications, and (2) the distributed replicated data structures are consistent with each other.

These features derived from our novel design are summarized in the following properties:

MP2 (Node failure consequences): when a node fails, the corresponding replicated data structure crashes and its content is lost.

MP3 (Data transfer to/from leader): there is a communication channel between the SC and the leader to send and receive data to achieve consistency (CP1).

FTP3 (Data availability after application failure): when a containerized application restarts after application failure, the last data committed to the replicated data structure is available for access.

FTP4 (Data availability after node failure): when a containerized application restarts after node failure, either the last data committed to the replicated data structure or the previous one is available to access. We consider cases in which a node fails before SC sends committed data to the

SC leader.

CP1 (Data consistency in distributed replicated data structure): data stored in the replicated data structures are eventually equal.

IV. Verification and Analysis of the Proposed Solution

To visualize, analyze, validate, and formally prove that the proposed solution works as explained, we use UPPAAL to model our solution and verify the properties that we define later. (The UPPAAL file is uploaded in a github repository)[1].

A. Scopes of the verification

This subsection explains the functions/modules/parameters that we consider in our modeling as in-scope and what we did not consider as out of scope.

In-scope:

- Functional testing of the application and storage containers and their relations.
- Application and storage container failures and the effects of these failures on the replicated data structures values
- Node failure and its effect on accessing replicated data structure
- Data consistency between distributed replicas of replicated data structures

Out of scope:

- Timing effects and probable delays of application, storage container restart up on application and node failure
- Scalability of the system is not tested and analyzed (the effect of adding multiple nodes, SCs, etc.)
- Cost in terms of energy, replication of application, etc.

B. Overview of the application functionality

The main functionality of an application in failure free conditions is to (1) execute the defined tasks and (2) save data to its volume.

In addition to task execution, an Application in our proposed solution must: (1) write its state to the replicated data structure whenever there is a change on its state (this must occur after saving the state in the volume), and (2) fetch the required states from the replicated data structure whenever an App requires to read external states.

The pseudocode of the application functionality is shown in Algorithm 1.

Algorithm 1. Application Functionality Pseudocode

```
1
2  T={t_1 ,t_2 ,.. ,t_n}
3  id := App_id
4  //Fetch internal_state
5 internal_state:=rep_data_struct[id];
6 While Running Tasks()
7 {
8 for (t=0; t<=n-1; t++){
9   Run t()
10  internal_state :=new_internal_state
11 //Commit self_state
12  rep_data_struct[id] := internal_state;
13 }
14 if (read_external(true)){
15 //Fetch external_state
16   for (j=0; j<=num_ext_apps-1; j++){
17     external_state[j]:=rep_data_struct[num_apps+j]
18  } else{}
19 }
```

[1]https://github.com/ZeinabBa/UPPAAL

The variables in Algorithm 1, are explained next. Each application in the cluster has an identifier number called `id` (this is the same identifier number in the application specification explained in [4]). Each application has a set of tasks to execute. After execution of each of the tasks, the application writes its state in the volume and in the replicated data structure, respectively.

When an `internal_state` is changed in the next iteration of task execution, the updated "*internal_state*" is called `new_internal_state`. In this algorithm there are three variables, namely `internal_state`, `external_state` and `rep_data_struct` that are all used for storing state variables of applications. The variable named `internal_state` contains the state variable of the corresponding application. The applications residing in other nodes or external Apps do not have write access to the `internal_state` of other applications.

The `external_value` contains the state variable of applications that are working on any other node(s) in the cluster. We further explain the interactions between different entities in the cluster to clarify how data is written in this variable.

The array `rep_data_struct` refers to the replicated data structure of each node and it contains both internal and external state variables of all the applications. Simply put, `rep_data_struct` is a concatenation of all the `internal_state` and the `external_state` array.

Depending on the number of internal Apps and the number of external Apps the length of the array forming the replicated data structure will vary. We use two integers, `num_apps` and `num_ext_apps` to show the number of internal and external applications in our system respectively.

C. Overview of the storage container functionality

The Storage Container (SC) is responsible to (1) provide data storage, and (2) data consistency.

It is the responsibility of a SC to create the replicated data structure on each node right after the SC is deployed on the node. This is a one time task to be performed when the SC is deployed for the first time on a new node.

Each SC on each node is responsible to participate in data synchronization to achieve consensus and data consistency in the cluster, using the RAFT protocol.

We assume that all SCs in the cluster send the internal state(s) to the SC leader whenever there is a change in any of the internal states (states of the applications running on the same node). It is the role of SC leader to broadcast the data received from a SC to all other SCs.

The synchronization process for each SC has two main procedures: (a) to read (fetch) internal states from the replicated data structure, in case there is a change in the value

of any internal states, since SC must send these data to the SC leader, and (2) to receive external states from the SC leader and commit them to the replicated data structure so that internal applications can fetch and read them locally.

Algorithm 2. Storage Container Functionality Pseudocode

```
1
2 if (rep_data_struct.empty())
3   rep_data_struct[num_apps+nume_ext_apps] = {};
4   Full_synchronize(rep_data_struct){
5   rep_data_struct:=l_rep_data_struct_log;
6 }else{
7   for (id=0; id<=num_apps-1; id++){
8     rep_data_struct[id] :=internal_state[id]
9 } for (j=0; j<=num_ext_apps-1; j++){
10       rep_data_struct[num_apps+j]:=external_state[j
           ]}
11 }
12 While SC_working (true)
13 {
14 for (id=0; id<=num_apps-1; id++){
15   if (rep_data_struct[id]=!l_rep_data_struct_log[id
         ]){
16 //send update to leader by appending the log
17     l_rep_data_struct_log[id]:=rep_data_struct[id]
18  }
19
20 }for (j=0; j<=num_ext_apps-1; j++){
21   if (l_rep_data_struct[j]=!rep_data_struct[j]){
22 //update external state
23     rep_data_struct[j]:=l_rep_data_struct[j]
24   }
25 }
```

Algorithm 2 is the pseudocode of the SC functionality. When a SC is deployed on a node, it first checks if a replicated data structure already exists on the node and whether the replicated data structure is empty or not. In case the replicated data structure does not exist on the node, SC will create it and then by following the leader the SC will read all the states from the leading replicated data structure (`l_rep_data_struct`). Submitting data to leader and replicating data from the leader in the algorithm done through the replicated logs [6]. The update of logs is implemented in each of leader's heartbeat. The consensus log is shown as `l_rep_data_struct_log` in this pseudocode.

Reading all the states (internal and external) from the leader is called a *full synchronization* and it happens even if the replicated data structure already exists on a node but it is empty. Whenever the replicated data structure is not empty, SC first fetches the latest internal states from the replicated data structure and then reads the external states from the leading replicated data structure by following the leader.

In this work we consider that applications are clients to the RAFT network, the SCs are (1) the interfaces between the SC leader in the RAFT and the clients which are the applications; (2) replicas of the leader to ensure data consistency despite failures in the system.

D. Overview of the Leader Storage Container functionality

We described earlier that one of the SCs in the cluster must be a leader. According to RAFT, a leader is responsible to process the requests from clients and send the output of processing the request to all other members in the RAFT network. In case a leader fails other members of the RAFT consensus protocol will become leader candidates and the next leader will be selected based on the election and the votes that members give to the candidates. The election is limited in time, so that if a response from a candidate is not received in time then it will be excluded from the election. Based on this, we can assume that a RAFT network will always have a leader. Figure 4 shows the SC leader automaton in UPPAAL. This automaton has only one state with two synchronization channels and an update edge that whenever it is activated the value of external(s) Apps will be increased. The synchronization channels are the communications to the `Send_Receiv_state_var` state in the SC automaton.

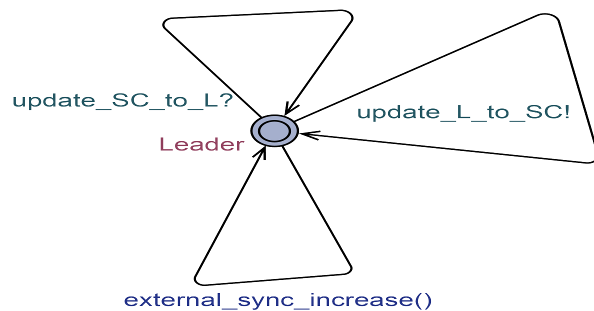

Fig. 4. Leader Storage Container Automaton

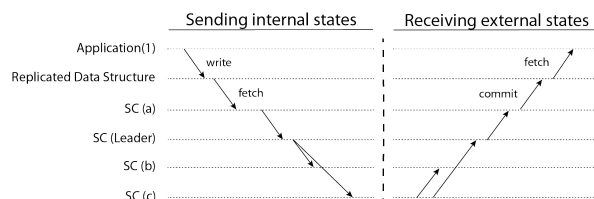

Fig. 5. RAFT and SCs Interactions

Figure 5 illustrates sending internal states and receiving external states between SCs, where arrows without label represent send actions. To make the data transaction clearer, we include an application and the replicated data structure as well.

E. Adding Application and Node Failure

The main aim of our modeling is to evaluate the behavior of the system in the presence of application and node failures, knowing that SCs are also mortal and vulnerable to failure. Therefore, we need to add two types of failure to our model: application failure and node failure. By adding these failures to our model we want to investigate if using replicated data structure together with SC and RAFT can provide our system with fault-tolerant persistent storage and data consistency. An application failure represents the failure of a container, and therefore it can be either a failure of an application container or a failure of a SC.

To add application failure to the model, we add an automaton that activates application and SC failures. Figure 7

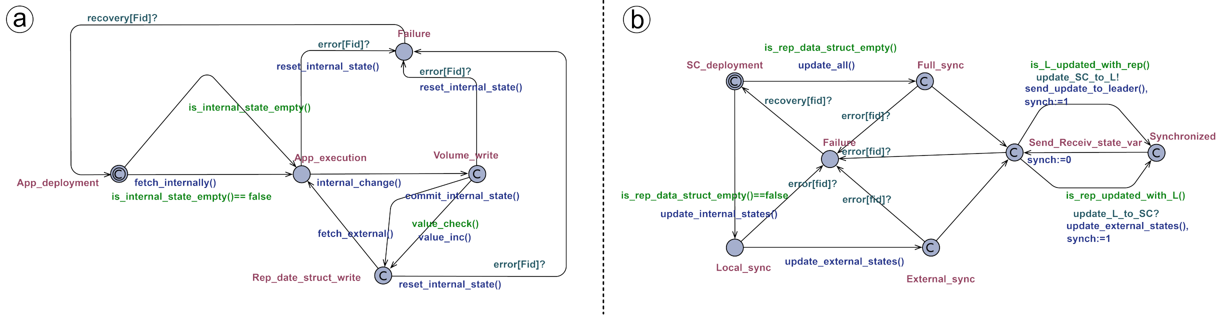

Fig. 6. Application and Storage Container Automatons

(a) shows the UPPAAL automaton, modeling the application failure. As is shown in this figure, activation of App and SC failure is done through channels in UPPAAL. When the edge having this channel is activated it forces the App or SC automaton to take an edge to its failure state. Figure 6 (a) and (b) show the application and SC automatons respectively, including the failure states.

For each of the Apps and SC a separate automaton is defined as a system, since failure of one App/SC (automaton) should not affect the other App/Sc automata. As shown in the figures 6 and 7 (a), when 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.

For activating the node failure, we create another automaton in UPPAAL, shown in figure 7 (b). However, in the App and the SC the failure state is the same location (for both failures, App and node) and they are also activated using the same channel. When a node failure is activated, 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 (delay for next transition is not possible) and activating a sequence of failures for App(s) and SC(s), one by one. 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).

F. UPPAAL Queries

We divide the properties into three categories. (i) Model properties that are used to verify that the functionality of the system is specified correctly; (ii) Fault-tolerance properties to verify that the proposed fault-tolerant mechanisms work as intended and our designed system can tolerate application and node failures and achieve data availability after these two types of failure; and (iii) Consistency properties to verify that the integration of the RAFT consensus protocol with our solution provides eventual data consistency.

1) Model Properties:

MP1 (Volatile Storage:) As stated in Section II-B, we need to make sure that we have implemented the volatile characteristic of volumes in our model correctly. Therefore we need to verify that when an App is in the deployment state, its internal state (its volume) is empty. We verify this by the following properties as:

```
A[] A0.App_deployment imply internal_value[0]==0
A[] A1.App_deployment imply internal_value[1]==0
```

In our modeling value `0` represents empty. These properties verify that MP 1(Volatile Storage) is fulfilled in our modeling.

MP2 (Node Failure Consequences): As state in Section III, we need to examine the behaviour of the system when a node fails. Specifically, we need to verify that when a node failure occurs, the data stored in the local copy of the replicated data structure is lost.

```
A[] NF.Node_Failure imply(rep_data_struct[0] == 0

&&rep_data_struct[1] == 0&& rep_data_struct[2] == 0)
```

Intuitively, this property states that, when the location `Node_Failure` in the `Node Failure` (NF) automata is reached for all possible transition sequences, always all the values of the `rep_data_struct` array are reset to `0`

MP3 (Data transfer to/from leader): As stated in Section III, we need to verify if the defined communication channel leads to data transfer in location related to data transfer to/from leader. To verify this we need to check if the edge for the channel is taken in the send and receive location. To formulate this, we need to either use an observer to monitor if the edge if taken or use a Boolean that is false and it only is true when it takes the edge with the channel and after leaving the synchronization state it turns to false again. This Boolean in our model is named `checker`. To verify this, we formulate the property:

```
A[] S.Sync imply checker==true
```

Intuitively, this property states that, when the state `Sync` in the SC automaton is reached the `checker` is always true.

2) Fault-tolerance Properties:

FTP1 (Application Failure Recovery): As stated in Section II-B, we need to verify the failure recovery ability of applications in our system. We formulate its related properties as:

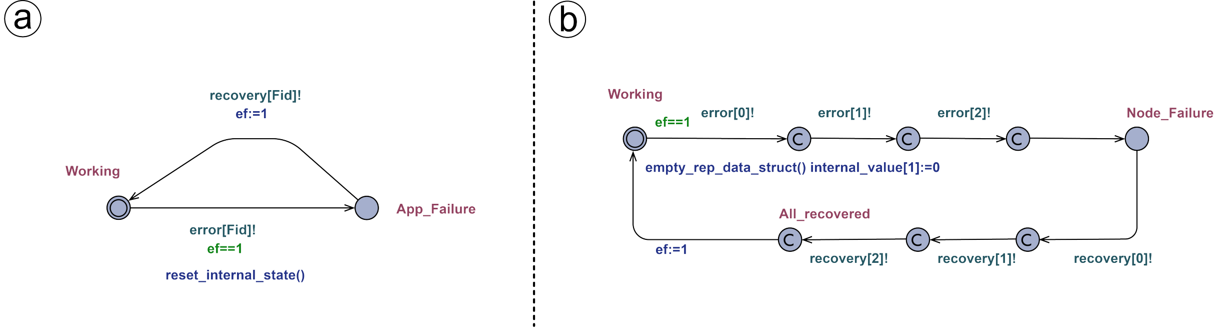

Fig. 7. Application and Node Failure Automatons

```
A0.Failure --> A0.App_deployment
A1.Failure --> A1.App_deployment
```

Intuitively, these properties state that, if the `Failure` state in `App0` (A0) is reached, the state `App_deployment` in `A0` then will eventually be reached. The same applies to `App1` (A1).

FTP2 (Node Failure Recovery): As stated in Section II-B, we need to verify that after node failure, the applications (Apps and SC) that before the failure were executing on the node will be recovered. We formulate this property as:

```
NF.Node_Failure --> NF.All_recovered
```

Intuitively, this property states that, whenever the `Node_Failure` state in `Node Failure` (NF) automaton is reached, the `All_recovered` state in `NF`, which indicate all the applications are recovered, will eventually be reached.

To complement the above property, we also examine whether the `Node_Failure` state indicates application failure and the `All_recovered` state represents that the applications are all back to deployment state. This is expressed as:

```
A[] NF.Node_Failure imply A0.Failure && A1.Failure
A[] NF.All_recovered imply A0.App_deployment &&
A1.App_deployment
```

FTP 3,4 (Data availability after application/ Node failure): Since the failure state in our modeling is the same for both application and node failure. We examine these two properties in one query.

```
A[] A1.App_execution imply
rep_data_struct[1] == last_internal_value[1] ||
rep_data_struct[1] == pre_last[1]

A[] A0.App_execution imply
rep_data_struct[0] == last_internal_value[0] ||
rep_data_struct[0] == pre_last[0]
```

Intuitively, this property states that, whenever the `App_execution` state in `App` (App0 and App1) automaton is reached, the value of replicated data structure is the last committed data to it or the previously committed data.

3) Consistency Properties:

CP1 (Data Consistency in Distributed Replicated Data Structures): As stated in Section III we need to verify data consistency between the nodes. We formulate its related property as path formula 1:

$$A[]A <> \varphi \tag{1}$$

Where φ indicates data consistency. However, since UPPAAL does not allow nesting of path formula [10] we define the equivalent propertiy in UPPAAL as follows:

```
True --> rep_data_struct == l_rep_data_struct
```

This property intuitively states that, for all possible transition sequences always eventually, the values of the replicated data structure (`rep_data_struct`) are equal to the values of leader replicated data structure (`l_rep_data_struct`).

We also verify data consistency with another property:

```
A[] S.Sync imply (rep_data_struct == l_rep_data_struct)
```

Intuitively, this property states that for all paths when the `Sync` state in a storage container (`S`) is reached, always the values of replicated data structure (`rep_data_struct`) and leader replicated data structure (`l_rep_data_struct`) are equal.

In addition to these properties we also define a safety property to show that the system will never be in a deadlock state. This property is essential to verify as the deadlock of any single state will lead to data loss or even system breakdown. We specify this property as usual in UPPAAL:

```
A[] not deadlock
```

All the defined properties are shown to be satisfied by the UPPAAL verifier. This proved that the proposed fault-tolerance and consistency mechanisms work as intended and the target properties are achieved. Additionally, satisfaction of the properties we called model properties proves that the underlying properties of the platform were correctly specified in our model.

V. Related Work and Discussion

In our previous paper [4], we discussed the main problem of distributed data storage and the existing solutions for that in the literature. We reviewed the related works focusing on two fundamental problems: providing fault-tolerant, permanent data storage and achieving a decentralised consensus.

We also reviewed the works proposing persistent storage container-based architectures in cloud platforms [3], [9], [11]–[13].

In this section, we review related work focusing on stateful application execution in container-based architectures.

Netto et al. in [13] and [14] indicated that using Kubernetes improve service availability of stateless applications. However, for stateful applications, Kubernetes faces some issues. In [13], they proposed state replication between pods, in a way that all pod replicas execute the incoming requests. However, only the one replica will respond to the request. In [14], they improved their solution by integrating an execution layer between clients and containers (called Koordinator). In this solution Koordinator received the clients requests and sends them to the application containers. There is also a layer between the Koordinator and the client which is a containeriezed Firewall. However, this firewall is a single point of failure in this model, when the firewall container fails the whole connection to nodes and other containers are lost especially in case of node failure. In our solution, we rely on the underlying Kubernetes service that is designed as a set of rules added to the IP tables of all nodes. Moreover, our solution is completely based on Kubernetes' principles, therefore, integration of it to Kubernetes can be easily managed. In another work Soenen et al. in [15] proposed a solution to provide high availability for the management and orchestration (MANO) in the Network Function Virtualization (NFV) architecture. Their solution is based on decomposing the application to functional blocks each performing a task in a workflow. The interaction between these functional blocks is through remote calls over a network. They deploy a redundant instance for each functional block to increase availability. Redundant instances check each other's availability through heartbeat. This means that each application needs to have an implementation of the availability logic which increase the application load to send and receive the heartbeats. In our solution, it is the Storage container responsibility to send and receive data and the communication between applications is offloaded to SC. The other problem in this work is the data consistency between replicas that has not been investigated. In our solution however, data consistency in the whole cluster is achieved by integrating the RAFT consensus protocol.

VI. Conclusion

In this paper we modeled and verified a persistent fault-tolerant storage solution for container-based fog architecture, which we proposed in [4]. We verified the model, fault-tolerance and consistency properties. The results indicates that using SC along with the containerized stateful applications provides fault-tolerance and data availability in case of application and node failure. In addition, integration of the RAFT protocol with SC provides eventual data consistency. In our future work we will add timing constrains to our model to examine the performance of our solution when there are tight timing requirements in the system.

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] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant IoT architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 813–818.

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal 4.0," *Department of computer science, Aalborg university*, 2006.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] T. Soenen, W. Tavernier, D. Colle, and M. Pickavet, "Optimising microservice-based reliable nfv management & orchestration architectures," in *2017 9th International Workshop on Resilient Networks Design and Modeling (RNDM)*, IEEE, 2017, pp. 1–7.