ContAv: a Tool to Assess Availability of Container-Based Systems

Stefano Sebastio, Rahul Ghosh, Avantika Gupta, Tridib Mukherjee

ContAv: a Tool to Assess Availability of Container-Based Systems

Stefano Sebastio*, Rahul Ghosh†, Avantika Gupta‡ and Tridib Mukherjee§
*Inria Rennes, France 35042
Email: stefano.sebastio@inria.fr
†American Express Big Data Labs, Bangalore, India 560103
‡Conduent Labs India, Bangalore, India 560103
§Data Science, Play Games24x7, Bangalore, India 560103

***Abstract*—The momentum gained by the microservice-oriented architecture is fostering the diffusion of operating system containers. Existing studies mainly focus on the performance of containerized services to demonstrate their low resource footprints. However, availability analysis of densely deployed container-based solutions is less visited due to difficulties in collecting failure artifacts. This is especially true when the containers are combined with virtual machines to achieve a higher security level. Inspired by Google's Kubernetes architecture, in this paper, we propose `ContAv`, an open-source distributed statistical model checker to assess availability of systems built on containers and virtual machines. The availability analysis is based on novel state-space and non-state-space models designed by us and that are automatically built and customized by the tool. By means of a graphical interface, `ContAv` allows domain experts to easily parameterize the system, to compare different configurations and to perform sensitivity analysis. Moreover, through a simple Java API, system architects can design and characterize the system behavior with a failure response and migration service.**

***Index Terms*—software containers, virtualization, cloud computing, distributed system availability, simulation tools**

I. Introduction

Operating system (OS) containers (e.g., Docker) are widely adopted by large providers such as Amazon [1], Google [5], Microsoft [2], and Netflix [3] for building cloud-based solutions. Containers are created by OS level virtualization [26] through the UNIX-like kernel resources isolation and management features. The lightweight and portable runtime environment built with containers enables the foundation for implementing microservice-oriented architectures [29]. A container-based system is composed by a *container manager* (or daemon) supporting *container instances* (more simply referred as containers) generated from *images* available in a *repository*. Containers spawned in multiple copies, for load balancing, scaling or availability purposes, are generated from the same image (containing e.g., database management systems or web servers) and thus belong to the same type.

Recent works [12], [19] have benchmarked and compared OS containers versus Virtual Machines (VMs) with respect to processing, storage, memory, and network aspects. While in general containers have better performance and lower overhead, the main drawbacks for container-based solutions are related to security issues. To this end, a common practice is to deploy the containers inside VMs for creating a layer of *security by isolation* [8].

The work was done when the authors were working at Xerox Research Centre India (XRCI), Bangalore, Karnataka, 560103, India.

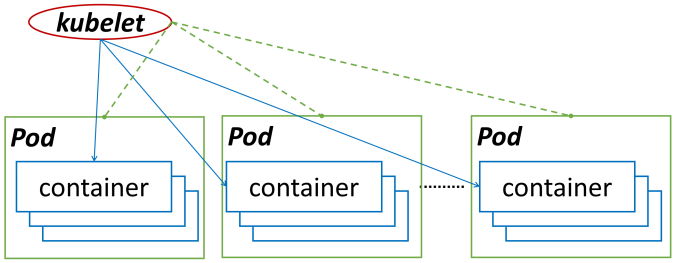

Fig. 1: Google Kubernetes architecture.

Google Kubernetes, Apache Mesos, Docker Swarm and Spotify Helios are examples of orchestration platforms for container-based systems. Note that, Kubernetes has a High-Availability version [6] with two components: *pods* and a *kubelet* agent (see Figure 1). The *pods* are groups of containers, co-located, and co-scheduled, running in a single logical host (i.e., a VM or a physical host). The *kubelet* manages the pods through a health check monitor, and can spawn new containers or restart existing containers when needed.

As more enterprises implement container-based systems, the availability assessment and management aspects of these solutions are becoming important. Availability studies can be classified in two main categories: (i) data-driven analysis and (ii) model-driven analysis. Datasets are populated with failure artifacts collected through measurements of a deployed system [15], [23], [27] or from a system undergoing stress tests [14], [18]. In the first case, the data reflect characteristics and phenomena of a real environment, whereas, in the latter, stressful conditions can be generated in laboratory having a controllable environment. Failure prediction and the evaluation of mitigation techniques can be realized by modeling the system behavior and by analyzing the associated availability. Formalisms for studying the system availability can be classified in state-space (e.g., Markov chains and Petri nets - PN) and non-state-space (e.g., Reliability Block Diagram and Fault Trees Diagram - FTD) models [13], [17]. These models are solved: analytically (even relying on the support of software

tools e.g., [10], [30]) or by simulation whenever the models are highly complex.

In this paper, we take a model driven approach for availability analysis of container-based systems, primarily due to the difficulties in collecting or finding publicly available failure artifacts. However, we conduct laboratory experiments to collect real data (as reported later in Section V) and to parametrize our analytical models. In our recent work [22], we proposed, designed, modeled and analyzed some container-based solutions for studying system availability through FTD and stochastic PN. The models in the aforementioned work become too complex to be solved analytically or with the aid of stochastic modeling software tools such as SPNP [10] due to increasing number and type of containers which eventually lead to state space explosion problem.

Contributions. This paper presents an open-source simulator `ContAv` to support availability analysis of container-based solutions. Striking a balance between the ease-of-use and flexibility, `ContAv` provides two key features: (i) through its GUI, the tool enables the domain experts to perform sensitivity analysis and to compare different deployment configurations (hiding the models complexity); (ii) system architects can use a simple Java API to have the desired flexibility in designing, testing and evaluating failure mitigation and migration strategies. `ContAv` is based on distributed statistical model checking for performing quantitative analysis on the system properties [25]. By using `ContAv`, a cloud practitioner is relieved from the hurdles encountered in generating and solving the analytical models of container-based deployment environments. Other than automatically generating the models, compared to the existing stochastic modeling tools (e.g., SPNP), `ContAv` provides model checking and distributed simulation functionality, and a higher abstraction of the functions for managing the containers behavior via a Java API.

Structure of the paper. The paper is structured as follows. In Section II the considered container-based configurations are discussed. The `ContAv` architecture is presented in Section III, whereas some implementation details and the validation results are reported in Section IV. Section V shows a case study in which a sensitivity analysis is performed and some of the `ContAv` APIs are described. Finally, Section VI concludes the work with some final remarks and outlines our future efforts on improving the tool.

II. Container-based Systems

Containers can be deployed in multiple instances for load balancing, scalability or availability purposes. To overcome the well-known security risks, exacerbated in densely deployed environments, a common practice contemplates the use of VMs for creating sandboxes around group of containers [8]. From this observation and studying the Google's Kubernetes architecture [6], in our recent work [22], we discerned three configurations named `consolidated`, `homogeneous` and `heterogeneous` therein. A pictorial representation of these three configurations is depicted in Figure 2.

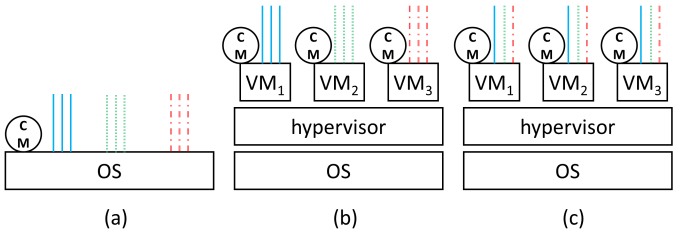

Fig. 2: OS Containers configurations. CM is the container manager (daemon), the lines represent the container instances (e.g, spawn for replication), whereas each line style represent a different container type (i.e., generated from a different image).

The simpler setup is represented by the `consolidated` configuration (Figure 2-a), in which all the containers are grouped in a single environment. A single container daemon is thus in charge to manage all the containers. This setup has the benefit of requiring less computational resources than the other configurations at the price of losing the security isolation provided by the VM. It is thus recommended to use the `consolidated` configuration in scenarios without security concerns or with scarce hardware resources.

The other two setups envisage a security via VM isolation approach. In the `homogeneous` configuration (see Figure 2-b), containers are grouped within VMs according to their type. This setup is thus suitable for a public cloud in which users run several instances of the same container in their own VM, for load balancing or availability, but are not willing to share their VMs for security concerns or pricing issues. An isolation by container type is achieved. Therefore, this setup is advised for all the scenario in which security is a key feature and can not be assured in the same way for all the container types.

Lastly, the `heterogeneous` configuration (see Figure 2-c) groups the containers in VMs according to the instance number. This setup can be adopted in a corporate cloud in which all the containers comply with the same security policy. With respect to the `homogeneous` configuration, in this case a VM failure does not affect the service availability, and the system can be kept working even if a VM reboot is required (e.g., to apply security patches).

As any researcher with hands-on experience on cloud containers could recognize, all the widely used container based platforms [1]–[3], [5], [6], such as Google Kubernetes, from a deployment perspective are organized around the `heterogeneous` or `homogeneous` configuration presented in the paper (others configurations are subcases of these and can be easily obtained through the `ContAv` API by selectively removing containers from the VMs).

Having multiple instances of the same container running, it is possible to define the system available when at least k *out-of* N instances are working for each container type (where k and N could be different for each container type). In this regard, it is worth noting that the container operation is subjected to the functioning of its manager. Containers can

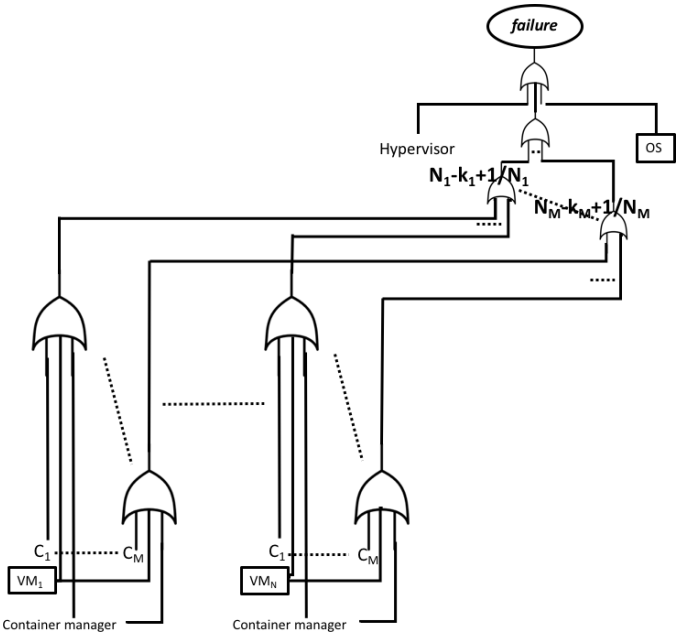

Fig. 3: Fault level diagram for the `heterogeneous` configuration (from [22]).

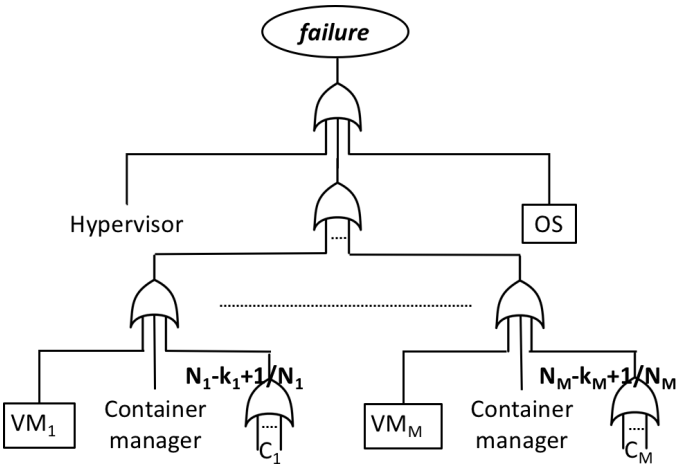

Fig. 4: Fault level diagram for the `homogeneous` configuration (from [22]).

thus fail for an internal problem but also due to problems affecting the environments that execute and control them i.e., manager, hypervisor and VM (if present), OS, or hardware. In [22] we present the Fault Tree Diagrams for all the three configurations, to highlight their substantial difference from an availability point of view. E.g., the availability of a given container type depends, in the `heterogeneous` configuration on the status of all the VMs (see Figure 3), whereas the `homogeneous` configuration has a single point-of-failure represented by the VM running all the containers of such a type (see Figure 4).

In case of container failures, it is possible to instruct the manager to automatically restore the container e.g., through the "autorestart" flag of the Docker implementation. Whereas, to

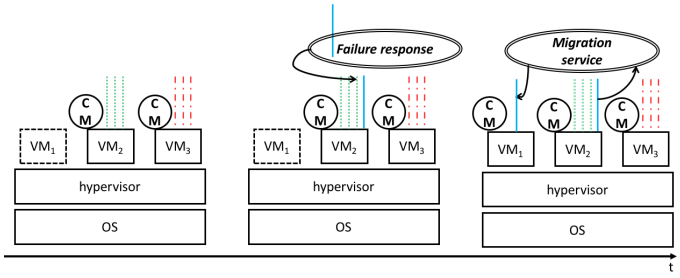

Fig. 5: Failure response and migration service.

recover the system in case of failures of any other component, the only feasible approach is to resort to an external component similarly to the Kubernetes in high availability configuration. We thus envisage a kubelet-like agent running in a different machine. Its action is limited to restart the manager, in case of failure of such a component. Whereas, since the VM restart is a moderately slow process, the kubelet-like agent has the chance to carry out several *failure response and migration service* strategies in the event of VM failures. In particular, another VM can be selected to temporary spawn and host a certain number of container instances to overcome the lack of the failed VM. At a later time, when the original VM is restored (together with its container manager), thanks to the orchestration of the kubelet-like agent, those backup containers can be migrated back resorting to the container live migration functionality [4], [7], [11]. The supporting VM will manage the backup containers as its own, thus restarting them in case of failures. A pictorial representation of the failure and migration support is presented in Figure 5.

Since several strategies can be exploited for the failure response and migration service, our simulator allows to design and test new approaches through a simple Java API, as described in the following Section V.

III. ContAv Architecture

In this section the `ContAv` architecture is illustrated. First, the tools underlying `ContAv` are presented, then the `ContAv` functionalities are described through an Unified Modeling Language (UML) use case diagram, and finally an UML activity diagrams make explicit the internal workflow of `ContAv`.

`ContAv` is built on top of DEUS [9], an open-source, Java-based, general-purpose discrete event simulator (DES). Among the different freely availably DES, our choice fell on DEUS for its flexibility on supporting analysis of every kind and size of complex systems. DEUS is constituted by three basic elements (reflected in an equivalent number of founding classes): (i) nodes, the entities interacting in the system; (ii) events, defining the internal actions and interactions among the nodes or with the environment; (iii) processes, determining the occurrence of the either stochastic or deterministic events. DEUS foresees a model described in Java and a scenario (i.e., the parametrization of the processes) in XML.

We enhanced DEUS with MultiVeStA [24] an efficient distributed statistical model checker (SMC) which can be

easily integrated with any DES. System properties of interest can be compactly asserted by means of the MultiQuaTEx quantitative temporal expressions. MultiQuaTEx queries are evaluated by the SMC running simulations distributed on multiple cores or physical machines, and evaluated against the significance level and the Confidence Interval (CI) size computed with the Student's t-test.

Through the SMC, it is possible to formally specify system properties such as the total outage time in a year, the number of container, container manager, VM, hypervisor and OS failures, and the number of failure and migration requests. For all the specified metrics, MultiVeStA runs independent simulations until the first of the following conditions is attained: reaching the required CI for all the MultiQuaTEx queries and achieving the maximum number of runs. Whenever the system metrics are gauged at different points in time, a comma-separated value (CSV) file is also automatically produced. The `ContAv` GUI allows to easily setup the analysis of interest and then compare the results graphically (e.g., after a sensitivity analysis). The analysis output can be exported as gnuplot, PNG or CSV file.

All in all, we built a toolchain coinciding with the one recently proposed for the volunteer cloud simulator AVo-Cloudy [21].

A. `ContAv` use case diagram

The UML use case diagram in Figure 6 describes the `ContAv` functionalities and how the different actors interact with the tool. Despite the open-source nature of the tool, in `ContAv` actors could interact with the core (the simulator itself) simply through the abstraction provided by its services. `ContAv` envisages three possible actors interacting with it. *Experts* on distributed and container-based systems are in charge of defining the OS containers configurations (*Setup*) and parametrize the scenario (*Configuration*) according to the properties of the system under study. *Decision makers* will compare the system performance (*Comparison*) against several configurations or after having performed a sensitivity analysis. *System architects* can work at a lower level designing and implementing the *failure response and migration service* strategies (as described later in Section V) that will be subsequently analyzed by the other actors. The *Analysis* use case depends on the simulator model and in turn on the way it has been designed and configured.

The interaction with all the functionalities shown in the use case diagram can be abstracted at a higher level taking advantage of the GUI provided by `ContAv`. The latter presents the use cases: setup, configuration, analysis and comparison with a tabbed interface (see Section IV). As already mentioned, the only feature that could require code writing is the failure response and migration service, if design and test of novel solutions are deemed.

B. `ContAv` activity diagram

In this section the internal workflow of `ContAv` is explained by means of the UML activity diagram in Figure 7,

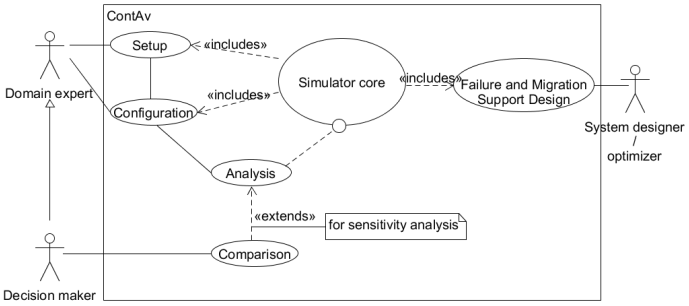

Fig. 6: `ContAv` UML Use Case diagram.

in case the tool is used through its GUI.

At first, the user must setup the configurations of interest among the ones presented in Section II or user-specified through the Java API. Then, the input parameters for configuring the scenario and the SMC features must be provided. For the scenario parameters such as Mean Time to Restart (MTTR) and Mean Time to Failure (MTTF), `ContAv` provides default values computed during experiments in our laboratory or from literature, as later reported in Table I of Section V. The parameters required by the distributed SMC are instead: number of simulations performed in parallel, batch size, maximum number of simulation runs, significance level and confidence interval size.

Once the setup is ready, the SMC can start the quantitative statistical analysis loading the scenario and the system properties of interest. According to the specified degree of parallelism MultiVeStA will start independent and distributed simulations exploiting the cores parallelism or the available machines. In the latter case, users need to fill in the `serverlist` file with a list of IP address - TCP port pairs running the DES engines. Simulations are performed until the first of the following conditions is reached: the statistical analysis has an accuracy less or equal to the requested CI, or the maximum number of runs is obtained. Readers interested about the performance scaling achievable integrating MutliVeStA with a DES can refer to [20] in which a deeper analysis is presented.

`ContAv` users can perform sensitivity analysis by simply choosing the parameter(s) of interest from a drop down menu and specifying range, maximum and minimum values, whereas the tool will be responsible for automatically generating all the corresponding scenarios. Several sensitivity analyses can be requested at a time.

Finally, it is possible to graphically compare results from different configurations and sensitivity analyses through an interactive plot widget. Results for each scenario are saved as CSV, whereas any additional data and plot produced during the comparison can be exported as PNG, gnuplot, SVG or CSV file.

IV. `ContAv` Implementation

`ContAv` models are based on the Stochastic Reward Nets (SRNs) and FTDs designed in our recent work [22]. The FTDs provide an overview on the interdependencies of the

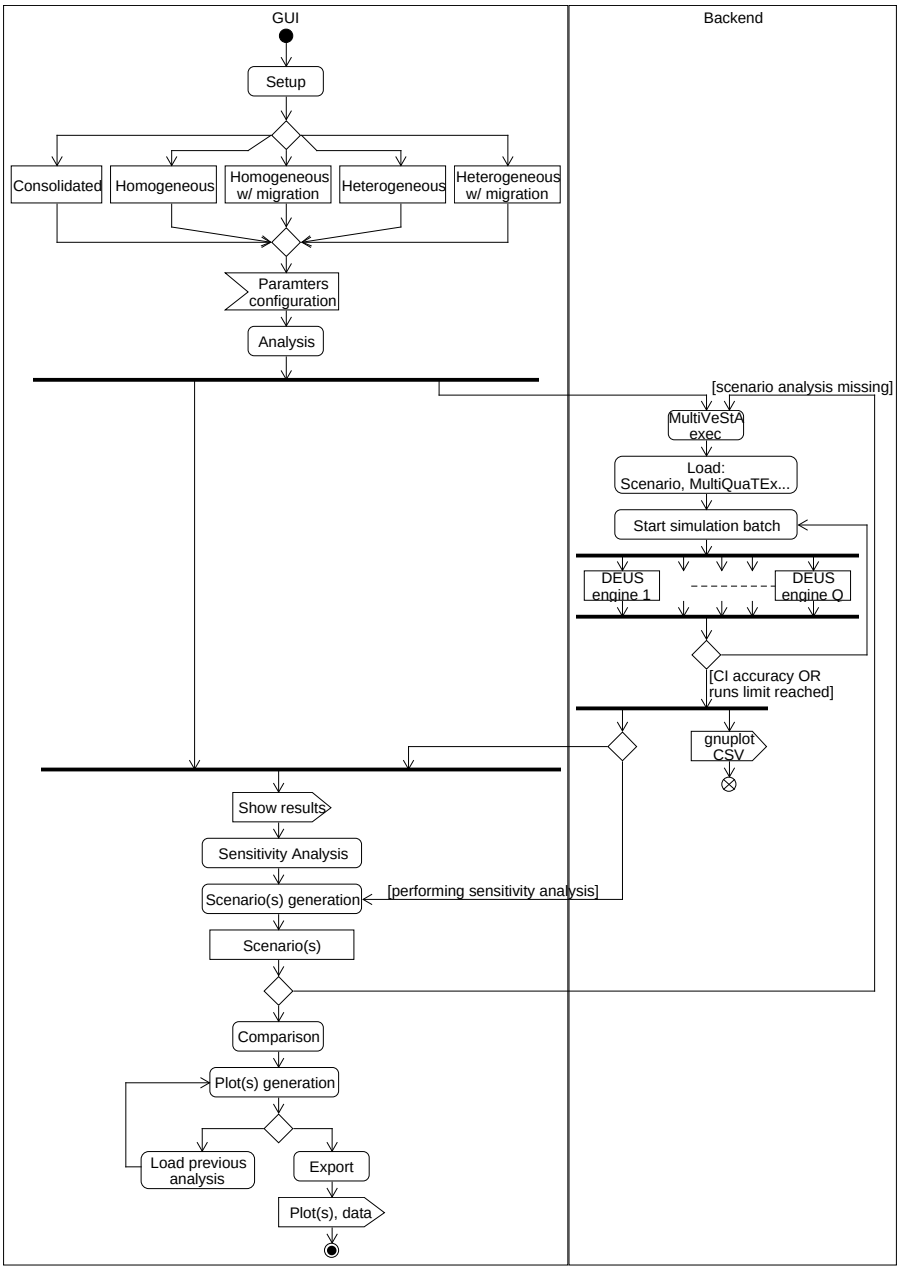

Fig. 7: `ContAv` UML Activity diagram.

availability of the components and show how a component failure affects the system availability. Instead, the SRNs model the system behavior by means of stochastic processes.

For instance, in the FTD of the `heterogeneous` configuration (see Figure 3 in Section II and [22]) it is possible to identify three layers. In the bottom layer, availability of container manager, VM and the one of all the containers of the same type are OR-ed (i.e., in this configuration an OR gate is required for each container type running in each VM). The central layer is constituted by as many *voting OR-gate*s as the number of container types in the system (to obtain the availability of a *k out-of N* configuration), in which the input parameters are the gates of the lower layer. The topmost layer considers instead the availability of containers (originated from the aforementioned gates), hypervisor, OS and can be easily extended to consider also hardware components.

An example of the SRN for a single VM in `heterogeneous` configuration (without the failure response and migration service) and constituted by only two types of container, is reported in Figure 8. The SRN places and transitions required for each container type are identified by the dashed selection in Figure 8. As the number of containers and/or their type increases, the model will incur in a state space explosion and even tools such as

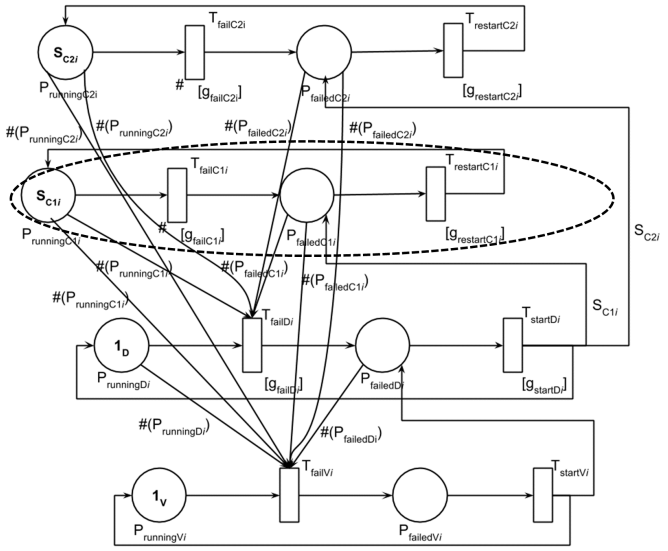

Fig. 8: SRN Model for a VM hosting two type of containers in `heterogeneous` configuration. From the bottom, availability of: VM, daemon, and container type $C1$ and $C2$.

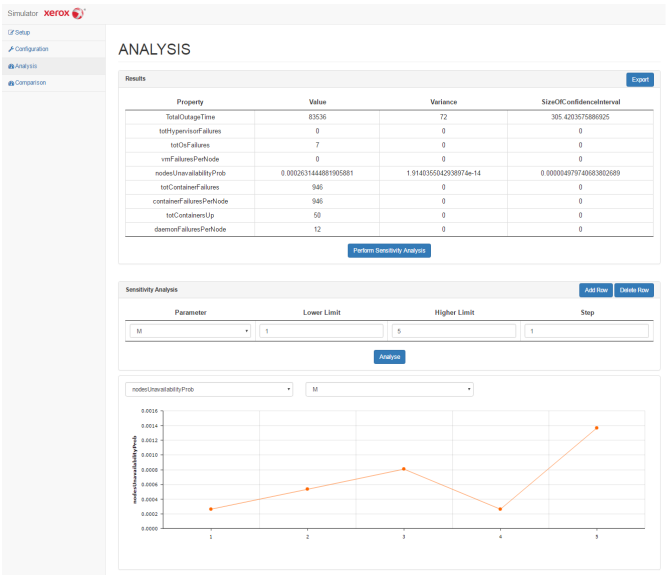

Fig. 9: `ContAv` GUI: Analysis tab.

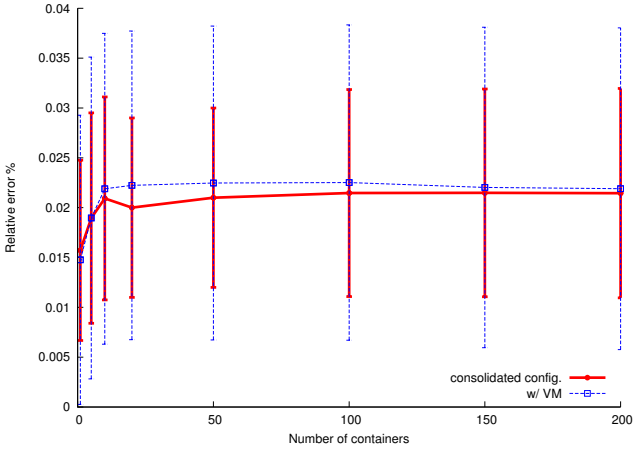

Fig. 10: `ContAv` validation: Percent (relative) error on system availability.

SPNP can not be of any help (since the reachability graph needs to be built in memory). For the `homogeneous` and `heterogeneous` configurations in which a failure response and container migration service is implemented, the underlying SRN models are very complex (see [22]), a node-by-node (i.e., VM) decomposition can not be adopted and thus the system turns out to be analytically intractable. The only viable solution in these cases is resorting on the Monte Carlo simulations on which the SMC is based on. `ContAv` is the first tool tailored for studying availability of container solutions extracting away all the hurdles faced by a cloud practitioner in generating these models.

In Figure 9 a screenshot of the tabbed interface designed around the use cases discussed in the previous Section III is presented. The system availability and the number of tasks lost are two fundamental availability metrics for a container-based cloud (see Google's Borg [31]) captured by `ContAv`. Other measures e.g., downtime, charge-back for customers, loss of productivity, and operational cost of re-running the tasks are derived from the first. Noteworthy, `ContAv` users can easily study and compare container-based configurations answering to questions such as: *"Having a certain configuration, how many container instances are required to achieve an availability of at least x%?"* or *"Which configurations are the best wishing to spawn a given number of containers?"*.

To validate the simulator, we compare the `ContAv` solutions with the analytical solution using SPNP [10] and the SHARPE tool [30] for certain configurations. Figure 10 shows the relative error of `ContAv` with respect to the analytical solution varying the number of containers for the monolithic models in which a single container type is present and no failure response and migration service is available. The `ContAv` percent relative error is of about the 0.02% running only 100 simulations. Obviously, the time taken by the simulator is greater as higher is the required accuracy. The beforehand discussed model complexity prevents us to derive the analytical solutions for more complex scenarios.

V. Case Studies

To provide a starting point for the model parameterization, we populate the configuration tab of the GUI with some default values (see Table I). These parameters are either computed from experiments on our enterprise private cloud system or acquired from public data [16], [17]. In particular OS, hypervisor and VM values have been extracted from [17]. The container availability has been computed from the Google dataset [16] considering how Google maps each task in a Linux cgroup-based resource container [31] (on top of which the Docker containers are built). Each value computed in our laboratory is the average of 1000 experiments on a server equipped with

TABLE I: Default input parameters of ContAv

Parameter	Value
OS steady state availability (*os*)	0.99977
Hypervisor steady state availability (*hypervisor*)	0.99989
mean time to container failure ($1/\lambda_{Ci}$)	1258 hours
mean time to container restart ($1/\mu_{Ci}$)	0.238 secs
mean time to container manager failure ($1/\lambda_d$)	2516 hours
mean time to container manager restart ($1/\mu_d$)	0.255 secs
mean time to VM failure ($1/\lambda_v$)	2880 hours
mean time to VM restart ($1/\mu_v$)	5 mins
mean time to orchestrate the failure support for node i on node j ($1/\delta_{iSendReqToj}$)	1 sec
mean time to load container of type i on node j ($1/\delta_{loadiToj}$)	2 mins
mean time to run container of type i on node j ($1/\mu_{runiToj}$)	0.275 secs
mean time to migrate back container of type i from node j ($1/\omega_{migrateCiFromj}$)	0.140 secs

2 Intel Xeon CPU E5-2698 v3, 322 GB of RAM running Ubuntu 14.04 LTS (Linux 3.13.0-24-generic x86_64) and Docker 1.10.1 (API 1.22). The container restart corresponds to the Docker `START` command, whereas the failure response assumes the `RUN` of a new container once, if needed, an estimated time of 2 mins to `LOAD` the Ubuntu container image available on "The Docker Hub" (e.g., downloading its 120 MB file size with an ADSL 8 Mbit/s) is elapsed. Being the migration a feature currently under development for Docker, we computed the sum of `PAUSE`, `COMMIT` (on the original node) and `UNPAUSE` as the migration time.

Since in the aforementioned table the container manager failure rate is due to an arbitrary estimation, in this section we performed a sensitivity analysis on this parameter through `ContAv`. The *k out-of N* availability is instantiated assuming that the system is considered *up* if, for each of the two types of container, at least the 80% of its initial 100 containers are running. Seeing as how the Google's Borg [31] runs tasks in containers, in the following analysis we are interested on knowing the dependency of the tasks lost (i.e., any OS process wrapped under a container) from the container manager failure rate. Results of the sensitivity analysis are shown in Figure 11. From the plot it is possible observing that, increasing the mean time to failure for the container manager, for each of the three container-based configurations, the tasks lost are reduced. Performance is stabilized when the container manger has a mean time to failure of about 2000 hours. Over this values the performance improvement is only marginal. For a mean-time-to-daemon-failure of 5000 hours, the `homogeneous` configuration has lost almost 1000 tasks more than the other configurations. This is an example of how `ContAv` makes a deployment decision *failure-aware*.

In the beforehand considered analysis, we assumed that all the mean time to failures were referred to a Poisson distribution. Exploiting the `ContAv` toolchain (as described in Section III) we can easily modify the process definition just by calling a different handler. E.g., in Listing 1 is reported a snippet of the XML file defining the simulated scenario, and particularly the process definition of the container manager failure. The distribution can be changed

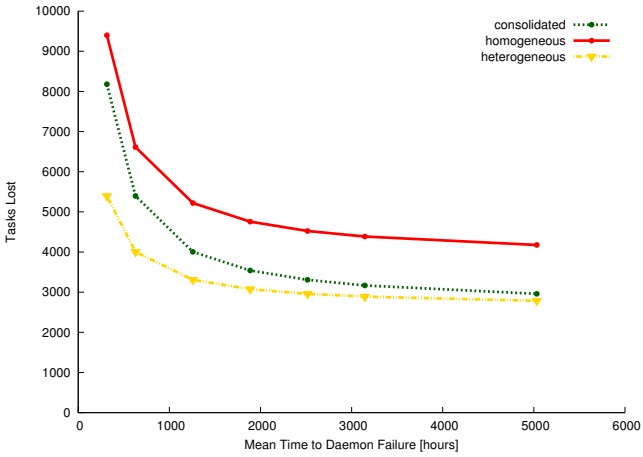

Fig. 11: Sensitivity analysis on container daemon failure rate.

Listing 1: Process definition of the container manager failure

```
<aut:process id="processDaemonFailure" handler="Poisson">
  <aut:params>
    ...
    <aut:param name="meanArrival" value="%MTTF_DAEMON%"/>
  </aut:params>
  <aut:nodes>
    <aut:reference id="containersAvailNodeModel" />
  </aut:nodes>
  <aut:events>
    <aut:reference id="failureDaemon" />
  </aut:events>
</aut:process>
```

in the `handler` key choosing among: `Exponential`, `Lognormal`, `Uniform`, `Weibull`, `Periodic`, `Poisson`, `Rectangular`, and `Pareto`. Any change made on the XML configuration file is automatically managed by `ContAv`.

In a densely deployed container-based system, VM nodes can cooperate for realizing failure response and container migration strategies. As described in Section III, system architects can design and test such strategies with `ContAv` thanks to its Java API.

Failure response strategies can be designed implementing the `failureSupport()` function. Such a function is called by the kubelet-like agent on a VM failure. As a basic example, the implementation for a uniform random selection is shown in Listing 2. A list containing all the working nodes (i.e., with running VM and container manager) is created (see line 3). Then, a working node is randomly selected calling the `selectNodeForSupport()` function (see line 13). Finally, the request is forwarded to the support node through the `askFailureSupport()` `ContAv` API function (line 8). More complex strategies can be implemented following these steps.

Similarly, migration back policies can be implemented through the `migrateBack()` `ContAv` API. A Javadoc attached to the tool provides a full description of all the functions available in `ContAv`.

Listing 2: Failure mitigation with random selection startegy

```
public void failureSupport() {
  ArrayList<Node> workingNodes = new ArrayList<Node>();
  Model.nodes.parallelStream().filter(n -> (!n.vmOut &&
     !n.daemonOut).forEach(n -> workingNodes.add(n));

  if (workingNodes.size() > 0){ //start the failure resp.
    Node node = selectNodeForSupport(workingNodes);
    askFailureSupport(containerTypeToSpawn,
       numOfContainersToSpawn, node);
  }
}

public int Node selectNodeForSupport(ArrayList<ModelA>
   workingNodes){
  int nodeToAsk = Engine.getDefault().getSimRandom()
    .nextInt(workingNodes.size());
  return workingNodes.get(NodeToAsk);
}
```

VI. Conclusion and Future Work

This paper introduced our distributed statistical model checker ContAv. ContAv is a first-of-a-kind availability analysis tool for containers capturing enterprise grade deployment configurations and providing a scientific modeling approach while hiding the details to users. The simulation models have been designed by means of Stochastic Reward Nets and Fault Tree Diagrams recently proposed by us in [22], and validated against the analytical solution.

ContAv is publicly released as an open-source tool, enriched with a full set of default parameters computed from experiments on our enterprise private cloud, and has a twofold use. System architects can use the simple Java API to design new failure response and container migration strategies, whereas domain experts can interact with the GUI and parametrize the system to quantitatively compare system configurations and perform sensitivity analysis. Through a what-if analysis performed with ContAv, cloud administrators can judiciously use costly resources (e.g., servers, VMs) that can increase redundancy and create careful deployment strategies.

We plan to extend ContAv in many directions. The model can be extended to systems consisting of many physical nodes in the same network, and considering resource constraints in the case in which containers have associated resource requirements. Moreover, performability aspects can be considered in the model. We are also evaluating approaches to enhance the ContAv statistical model checking capability: including the steady-state analysis with the batch means method [28], and managing rare events during simulation through importance sampling or splitting.

Acknowledgment

The first author would like to thank Prof. Axel Legay of Inria Rennes Bretagne Atalantique (France) for his support.

References

[1] Amazon EC2 Container Service. https://aws.amazon.com/ecs/. Accessed: 2016-05-15.

[2] Azure container service. https://azure.microsoft.com/en-gb/services/container-service/. Accessed: 2016-05-15.

[3] Containers at Netflix - an evolving story. https://qconsf.com/system/files/presentation-slides/containersnetflix.pdf. Accessed: 2016-05-15.

[4] CRIU integration in Docker. https://criu.org/Docker. Accessed: 2016-06-23.

[5] Google cloud platform - container engine. https://cloud.google.com/container-engine/. Accessed: 2016-05-15.

[6] Kubernetes high-availability. http://kubernetes.io/docs/admin/high-availability/. Accessed: 2016-06-21.

[7] P.Haul an engine for containers live migration. https://criu.org/P.Haul. Accessed: 2016-06-23.

[8] Introduction to container security. Technical report, Docker, March 2015.

[9] M. Amoretti, M. Picone, F. Zanichelli, and G. Ferrari. Simulating Mobile and Distributed Systems with DEUS and ns-3. In *HPCS*, 2013.

[10] G. Ciardo and K. S. Trivedi. Spnp: The stochastic petri net package (version 3.1). In *MASCOTS*, 1993.

[11] P. Emelyanov. Live migrating a container: pros, cons and gotchas. http://www.slideshare.net/Docker/live-migrating-a-container-pros-cons-and-gotchas. Accessed: 2016-06-23.

[12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *ISPASS*, 2015.

[13] R. Ghosh, F. Longo, F. Frattini, S. Russo, and K. S. Trivedi. Scalable analytics for iaas cloud availability. *IEEE Trans. Cloud Comput.*, 2:57–70, Jan 2014.

[14] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. Analysis of software aging in a web server. *IEEE Trans. Rel.*, 55:411–420, Sept 2006.

[15] M. Grottke, A. P. Nikora, and K. S. Trivedi. An empirical investigation of fault types in space mission system software. In *DSN*, 2010.

[16] J. L. Hellerstein. Google cluster data. Google research blog, Jan. 2010. Posted at http://googleresearch.blogspot.com/2010/01/google-cluster-data.html.

[17] D. S. Kim, F. Machida, and K. S. Trivedi. Availability modeling and analysis of a virtualized system. In *PRDC*, 2009.

[18] R. Matias and P. J. F. Filho. An experimental study on software aging and rejuvenation in web servers. In *COMPSAC*, 2006.

[19] R. Morabito, J. Kjllman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *IC2E*, 2015.

[20] D. Pianini, S. Sebastio, and A. Vandin. Distributed statistical analysis of complex systems modeled through a chemical metaphor. In *HPCS*, 2014.

[21] S. Sebastio, M. Amoretti, and A. Lluch Lafuente. AVOCLOUDY: a simulator of volunteer clouds. *Softw.: Practice and Experience*, 46(1):3–30, 2016.

[22] S. Sebastio, R. Ghosh, and T. Mukherjee. An Availability Analysis Approach for Deployment Configurations of Containers. *IEEE Trans. Services Comput.*, 2017.

[23] S. Sebastio, K. S. Trivedi, and J. Alonso. Characterizing machines lifecycle in google data centers. *Performance Evaluation*, 2018.

[24] S. Sebastio and A. Vandin. MultiVeStA: Statistical model checking for discrete event simulators. In *ValueTools*, 2013.

[25] K. Sen, M. Viswanathan, and G. Agha. On Statistical Model Checking of Stochastic Systems. In *CAV*, 2005.

[26] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41:275–287, Mar 2007.

[27] I. Stefanovici, A. Hwang, and B. Schroeder. Battling borked bits. *IEEE Spectr.*, 52:34–53, December 2015.

[28] N. M. Steiger, E. K. Lada, J. R. Wilson, J. A. Joines, C. Alexopoulos, and D. Goldsman. Asap3: A batch means procedure for steady-state simulation analysis. *ACM Trans. Model. Comput. Simul.*, 15(1):39–73, Jan. 2005.

[29] J. Thnes. Microservices. *IEEE Softw.*, 32:116–116, Jan 2015.

[30] K. S. Trivedi and R. Sahner. Sharpe at the age of twenty two. *SIGMETRICS Perform. Eval. Rev.*, 36:52–57, Mar 2009.

[31] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.