Computing (2022) 104:809–833
https://doi.org/10.1007/s00607-021-00977-z

REGULAR PAPER

Self-healing trans-cloud applications

Antonio Brogi[1] · Jose Carrasco[2] · Francisco Durán[2] · Ernesto Pimentel[2] · Jacopo Soldani[1]

Received: 9 April 2021 / Accepted: 24 June 2021 / Published online: 4 July 2021

Abstract
Trans-cloud applications consist of multiple interacting components deployed across different cloud providers and at different service layers (IaaS and PaaS). In such complex deployment scenarios, fault handling and recovery need to deal with heterogeneous cloud offerings and to take into account inter-component dependencies. We propose a methodology for self-healing trans-cloud applications from failures occurring in application components or in the cloud services hosting them, both during deployment and while they are being operated. The proposed methodology enables reducing the time application components rely on faulted services, hence residing in "unstable" states where they can suddenly fail in cascade or exhibit erroneous behaviour. We also present an open-source prototype illustrating the feasibility of our proposal, which we have exploited to carry out an extensive evaluation based on controlled experiments and monkey testing.

Keywords Self-healing applications · Cloud application management · Trans-cloud · IaaS/PaaS cross-cloud applications

Mathematics Subject Classification 68M14

✉ Jacopo Soldani
soldani@di.unipi.it

Antonio Brogi
brogi@di.unipi.it

Jose Carrasco
carrasco@lcc.uma.es

Francisco Durán
duran@lcc.uma.es

Ernesto Pimentel
ernesto@lcc.uma.es

[1] Department of Computer Science, University of Pisa, Pisa, Italy

[2] ITIS Software, University of Málaga, Málaga, Spain

1 Introduction

Like any other application, cloud-based applications are unavoidably subject to failures [18]. Independently of the service providers and the kind of services used, system administrators must be continuously monitoring their applications and resources, to be able to detect such failures, and recover from them as soon as possible, and with as little impact as possible. Such errors may occur at any stage of the life cycle of applications, including their deployment and operation. Indeed, the fact that they are deployed on external resources requires some additional effort, mainly due to the limited control on such resources and the applications themselves. To alleviate such effort, most cloud vendors have in recent years added to their offerings different forms of self-healing mechanisms. For instance, AWS offers AWS OpsWorks Stacks[1] agents to monitor AWS service instances, and to replace them in case of need. Similar solutions are featured by other commercial providers, e.g., Google Cloud[2] or Kubernetes.[3] The increasing need for continuous availability have led to architectures that are able to operate, even with failing components. In parallel, after Netflix's Chaos Monkey,[4] we have witnessed how different providers have laid out services able to test the response upon the continuous injection of random failures.

The way in which service instances are replaced in the above mentioned solutions may slightly differ, depending on the type of service instances and the monitoring strategies, but they all consist in stopping the failed instance and starting a new one in replacement of the former. However, while the self-healing mechanisms featured by cloud vendors may succeed in efficiently substituting single failed service instances, they do not take into account application faults, nor the dependencies among the different components that form a multi-component application. For instance, if an application component depends, either directly or indirectly, on other application components, the fault of any of these may cause an erroneous operation of the first component, which may provide inconsistent answers or be left unresponsive.

At the same time, the virtual environment used to run application components has a significant impact on their recovery process. For instance, the time to recover a container is much lower than the time to recover a VM (i.e., virtual machine). This in turn means that, if an application component depends on another one, and if the latter fails, the time during which the former is left "unstable" (viz., the time during which it can provide inconsistent answers or be unresponsive, because some component it depends on failed) significantly varies depending on the deployment of the failed component. As a result, depending on the actual deployment of multi-component applications, we may have different durations for their possible "instability periods" (viz., time periods during which some of their application components are left unstable). Instability periods are definitely an issue, as inconsistent answers may cause inconsistent states for an application [13]. Furthermore, unresponsiveness increases

[1] AWS OpsWorks Autohealing: https://docs.aws.amazon.com/opsworks/.

[2] Google Cloud Autohealing: https://cloud.google.com/compute/docs/tutorials/high-availability-autohealing.

[3] Kubernetes: https://kubernetes.io.

[4] Chaos Monkey: https://netflix.github.io/chaosmonkey/.

the latency in answering to end-users, potentially causing client loss in the same way as underprovisioning does [3].

Despite their great recent improvement, the support provided by currently available self-healing mechanisms basically consist of destroying and restarting the VM or container instances running failed components, without considering instability induced by a failed component to the components depending on it. In addition, the currently available mechanisms can only be used on specific service offerings, and only used in specific ways. For instance, the self-healing OpsWorks-Stacks facilities offered by AWS only work on some kinds of EC2 instances with a specific configuration. Needless to say that these facilities are only intended for components deployed on their own services, ruling out cross-cloud deployments. However, the fact that these mechanisms do not take into consideration the dependencies between components, open the door to their use when services of multiple providers are used to deploy multi-component applications. One could think on the possibility of using AWS self-healing facilities for the components deployed using AWS services, Azure facilities for those deployed on Azure, and so on. This would however require to coordinate the corresponding self-healing solutions, e.g., to recover failures in interdependent components deployed on cloud offerings of different providers, hence resulting in a time-consuming and cumbersome process, which is currently to be manually performed.

This article proposes a novel platform for the self-healing of cross-cloud application deployments, which also enables reducing the instability periods in deployed applications by going beyond the classical "destroy and restart" recovery approach. We build on the recently proposed infrastructure for the management of *trans-cloud* applications [10], which enables operating multi-component applications across different cloud providers and also across different cloud service levels (viz., IaaS and PaaS). This approach uses a specification of the component-based application being managed using the OASIS standard TOSCA [25], which is used both to specify all the information on the deployment and operation of each individual component and the dependencies between them. We then propose an uniform self-healing methodology, which can be used to self-heal trans-cloud applications independently of which cloud providers or service levels are used to deploy their components. In addition, our self-healing methodology suitably considers inter-component dependencies while recovering failed components, in a way that reduces the possible instability periods of applications. Intuitively speaking, this is done by proactively self-healing components residing in unstable states, as they rely on capabilities featured by components that have failed and are being recovered.

To illustrate the feasibility of our self-healing methodology, we also present a prototype implementation, developed by extending the existing trans-cloud platform and by integrating a novel support for orchestrating the self-healing of trans-cloud applications. The presented prototype has been subjected to an extensive experimentation. In addition to a wide range of representative scenarios, a state-of-the-art chaos monkey has been developed for trans-cloud applications, which has been used for the analysis of the robustness of applications under the control of the managing system here proposed. The results of our experimentation show that our methodology enables recovering trans-cloud applications both from application failures (viz., internal malfunctioning of application components) and from cloud service failures (viz., crashing

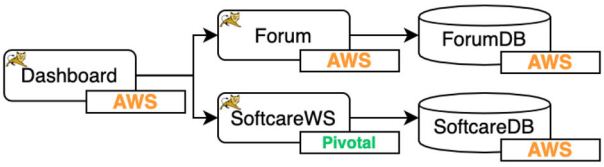

Fig. 1 Topology of the *Softcare* application

of the cloud resources used to host application components), and even if such failures happen for unforeseen reasons (e.g., because of cloud resources being destroyed by an external agent).

The rest of the article is organised as follows. Section 2 describes an example that will be later used both to illustrate the procedures in place and in the experimentation. Section 3 presents the proposed methodology to automate the self-healing of trans-cloud applications and its prototype implementation. Sections 4, 5 and 6 present and discuss the results of an experimentation of the prototype implementation of our self-healing methodology. Finally, Sects. 7 and 8, respectively, discuss related work and draw some concluding remarks.

This article is a revised and extended version of [7]. The methodology and prototype implementation presented in Sect. 3 are an improvement of that in [7], which now supports monitoring and recovering applications from both application and cloud service failures, with the prototype also being engineered to further speed up the recovery of applications. We also thoroughly expanded the discussion of related work in Sect. 7. The evaluation presented in Sects. 4, 5 and 6 is instead entirely new and first presented in this article.

2 Running example

Softcare [6] is a simple cloud application that we will use to discuss the problems of failing deployments and of recovering from faults at runtime. Figure 1 depicts the abstract topology of *Softcare*. The Dashboard component provides the main portal of the application, to which users can connect. It depends on the component SoftcareWS, which implements the core backend of the application, and on the Forum component, which implements a chat room. SoftcareWS and Forum, in turn, depend on components SoftcareDB and ForumDB, which are the databases storing application configuration data and the messages posted on the forum, respectively. Figure 1 also includes information on the technology used to develop each of the components (Dashboard, Forum and SoftcareWS run on Tomcat servers, ForumDB and SoftcareDB are MySql databases). Any available location could have been used to deploy these components, but in this case we chose to use both IaaS and PaaS services for illustration purposes, using Pivotal for SoftcareWS, and AWS for the rest.

Suppose that we wish to deploy the *Softcare* application using the trans-cloud support presented in [10]. We first must describe its topology and the locations where

to deploy its components by providing a TOSCA specification.[5] Such specification includes descriptions of the nodes forming the application (i.e., its components) and the relationships between them (by associating each requirement of each node with the node satisfying such requirement). The specification also includes the locations on which each node is to be deployed.

By feeding the trans-cloud support described in [10] with the corresponding TOSCA YAML specification, the actual trans-cloud deployment of *Softcare* can be enacted. The trans-cloud manager can indeed orchestrate the deployment of the application components over the indicated IaaS and PaaS offerings. However, its support for fault management is very basic, it is just able to show alerts about failed components in its main panel. What if the operations to install or start a component fail? What if a component unexpectedly fails while the application is up and running? What about other components depending on such failed components? They reside in unstable states, where they can fail as well or, even worse, exhibit an erroneous behaviour. We hence need a solution for self-healing failed components in trans-cloud application deployments, which also reduces the instability periods for components depending on failed components. Without such support, our only alternative is to get notified about the failure and to manually act on it.

3 Robust trans-cloud management

A key point for the proper operation of cloud applications is to ensure that all dependencies among their components are satisfied at all times [16], that is, to ensure that the application is always *stable*. Indeed, a component can work properly only if all components it depends on are up and running [5]. Thus, when deploying an application, a component should be started only after all the components it depends on have been previously started. When a component fails, other components depending on it may fail in cascade. Therefore, it is important to detect failures in applications as soon as possible, and to promptly react to failures by minimizing unstable states and bringing applications back to stable states, where all dependencies of all components are satisfied again [24].

We hereafter propose a self-healing, trans-cloud application management platform ensuring such a support, by suitably taking into account the components of the managed applications, their states, and inter-dependencies. With such information, the platform can manage the deployment and operation of applications. In the event of failures, the platform can detect the failed components and the type of failures that have occurred, and it automatically determines a plan to recover the application to its normal operational state.

The architecture of our prototypical platform consists of two components, namely the Trans-Cloud Controller and the Model Manager, coordinated using an Orchestrator. A TOSCA specification is used to represent all the required information on

[5] The TOSCA YAML specification of *Softcare* is available at https://github.com/scenic-uma/brooklyn-dist/blob/trans-cloud/trans-cloud-samples/softcare.yaml.

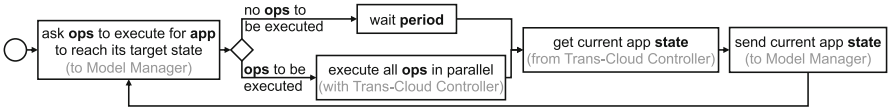

Fig. 2 Flowchart diagram sketching how the Orchestrator coordinates the Model Manager and Trans-Cloud Controller to enforce the self-healing of a trans-cloud application

the applications, while at the same time decoupling cloud applications from specific cloud providers and service levels.

3.1 Self-healing trans-cloud applications: methodology

An application administrator is expected to provide a TOSCA specification of her application and the target state, i.e., the desired state for each application component. Given these, the Orchestrator coordinates the Trans-Cloud Controller and the Model Manager in such a way that the target application state is eventually reached and maintained. The first step consists in forwarding the TOSCA specification to the Trans-Cloud Controller and the Model Manager, with the latter being also provided with the target application state.

The Orchestrator then coordinates the Trans-Cloud Controller and the Model Manager to enforce the self-healing trans-cloud management of the application as sketched in Fig. 2. The Orchestrator starts managing the application by asking to the Model Manager whether any management operation is to be executed for the application to reach its target state. This is obviously the case when first deploying an application, as none of its components is there. The Model Manager hence returns the first set of operations to be executed for the application to get closer to its target state. Such operations can all be executed in parallel: The Model Manager indeed ensures that their executions do not interfere, as the components to which such operation are applied do not depend on one another, neither directly nor because they depend on components that depend on components to which some other operations are applied. The Orchestrator then instructs the Trans-Cloud Controller to execute all operations returned by the Model Manager in parallel. Once they are all completed, the Orchestrator asks the Trans-Cloud Controller to provide the current state of the application, which the Orchestrator then forwards to the Model Manager. The Orchestrator does so because the executed operations may have not been completed successfully and since a component may fail while the step is being executed (even if such step is affecting other components). The Orchestrator then asks to the Model Manager whether any operation is to be executed for the application to reach its target state, and it repeats the process.

When the current state of the application coincides with the target one, no more operations need to be executed, i.e., the Model Manager returns an empty set of operations. The Orchestrator then only monitors the application, in order to maintain it in the target state. After waiting for a given period, the Orchestrator asks the Trans-Cloud Controller to provide the current state of the application and it forwards such state to the Model Manager. The Orchestrator then asks to the Model Manager whether

management operations are to be executed: if nothing happened to the application components (e.g., if no failure happened), this will not be the case. The Orchestrator will then periodically repeat the monitoring process.

When a component fails, some differences between the application state and its target state will eventually be identified. If this is the case, the Orchestrator will request the operations to be executed for restoring the target application state from the Model Manager. The Orchestrator can then instruct the Trans-Cloud Controller to concretely execute such operations and to monitor the actual state of application components. By repeating this process in the very same way as it was done for the first deployment of the application, the Orchestrator can coordinate the Trans-Cloud Controller and the Model Manager to enact a recovery plan that restores the target application state.

As we will illustrate in our experiments in Sects. 4, 5 and 6, our methodology also covers the special cases of failures occurring *while* management operations are being enacted, both on components to which operations are applied and whose execution fails, and on components that fail while other components are being managed, i.e., during a recovery procedure. This is because the methodology only asks for operations that can be executed in parallel in the current state of a supervised application for the latter to get closer to its target state. Once such operations are completed, the actual state (possibly including the effects of some unexpected failures) is again monitored and communicated to the Model Manager, which returns a brand new set of operations allowing the supervised application to get closer to its target state. This ensures that the application eventually reaches its target state, even if some of its components unexpectedly fail while deployment/recovery plans are being enacted.[6]

3.2 Self-healing trans-cloud applications: prototype

We present in this section details on the current implementation of our failure-aware application management platform. In order to obtain a modular platform, and following the separation of concerns design principle [19], we implemented it by realising the two-tiered architecture above described.

The Trans-Cloud Controller. We realised the Trans-Cloud Controller by extending the trans-cloud deployment tool proposed in [10], which is based on Apache Brooklyn.[7] Brooklyn is a multi-cloud deployment platform enabling the deployment of the components forming an application across multiple heterogeneous IaaS clouds. Brooklyn was extended in [10] with mechanisms to manage PaaS services, fully-supporting in this way trans-cloud applications, i.e., allowing the deployment of applications not only without a specific knowledge about the target cloud providers, but also independently of their service level.[8]

[6] Intuitively, the target state can always be eventually reached because (in the very worst case) the recovery plan would destroy the IaaS/PaaS resources on which application components are running and create new resources where to deploy such components afterwards.

[7] Apache Brooklyn: https://brooklyn.apache.org/.

[8] The resulting, engineered trans-cloud extension of Brooklyn is open-source and publicly available at: https://github.com/scenic-uma/brooklyn-dist/tree/trans-cloud.

The trans-cloud approach is based on the OASIS standard TOSCA, which allows to specify the topology of a cloud application in a vendor-agnostic way, but also declaratively indicating configurations for all components forming the application. The combination of Brooklyn and TOSCA used in [10] enables enforcing an homogeneous access to cloud services. For instance, *policies* are used to indicate the locations of components, without giving details neither about the vendor nor the used services. All these issues are automatically managed by the trans-cloud extension of Brooklyn, together with other issues (as, e.g., the management of certificates, etc.). The TOSCA specification indeed enables abstracting from the specificities of cloud vendors. Then, to manage the selected cloud services when applications' components have to be deployed, the trans-cloud extension of Brooklyn defines a common interface, ensuring an homogeneous treatment of different cloud vendors and their services. The latter includes, for instance, operations for starting a component in a given location, stopping it, and for starting the component again. All these functionalities can get accessed through a REST API.

The above makes the trans-cloud extension of Brooklyn a natural candidate to realise the Trans-Cloud Controller in our solution. The only missing piece for actually implementing the Trans-Cloud Controller is full and uniform support for the monitoring of PaaS and IaaS services, as well as mechanisms for the notification of the successful completion of the enacted operations. The extension basically amounts to a periodic REST request on the status of applications and on the status of the associated resources. The access to the resources is decentralized and asynchronous. Following Brooklyn's approach, which controls the pooling frequency and multi-threaded execution of the monitor objects, an object is in charge of monitoring each component or service. This approach provides an elastic monitoring system that can scale to support a large amount of IaaS and PaaS resources.

In addition, and to enable the envisioned orchestration approach, we encapsulated the extended trans-cloud deployment support behind a Java-based REST API that enables POSTing the TOSCA specification of an application, as well as the management operations to be enacted on the application components. Our REST API wraps the original REST API of the trans-cloud deployer in [10] by also enabling GETting the actual state of the application components, with such GET exploiting the newly added monitoring support.

The Model Manager. We realised the Model Manager as a Java-based web application, which is open-source and publicly available.[9] Our Model Manager relies on the modelling and analysis support given by management protocols [5]. Management protocols enable modelling the management behaviour of an application component as a finite state machine (such as that in Fig. 3, for instance). States model the possible states of the component, while transitions model the effects of performing management operations on the component. To model inter-component dependencies, states, and transitions are enriched with conditions on the requirements and capabilities of the component. Conditions on states define which requirements must be satisfied in a state and which capabilities are provided by a component in a state. Conditions on transitions instead define which requirements must be satisfied to actually execute a

[9] Model Manager: https://github.com/di-unipi-socc/trans-cloud-model-manager.

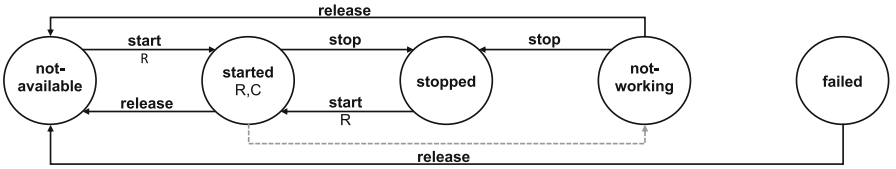

Fig. 3 Management protocol of a Brooklyn entity. Circles represent states. Solid and dashed arrows represent transitions corresponding to operation execution and failure handling, respectively. Label R is associated with the state and transitions where the entity is needing its requirements to be satisfied, while label C is associated with the state where the entity is actually providing its capabilities to satisfy other entities' requirements

management operation in a state. A second transition relation then models how components react when failures occur, e.g., when a component stops providing a capability satisfying a requirement needed by another component.

Management protocols can then be composed according to the relationships in the topology of an application (with each relationship connecting a requirement of a node with a capability of another node). This enables to automatically derive the overall behaviour of an application, which is essentially given by a labelled transition system over the possible states for the application. The possible transitions for such a transition system correspond to the execution of management operation on an application component, or to the handling of a failure affecting an application component.

This makes management protocols a natural solution for determining a plan allowing an application to move from its current state to a target state. The Model Manager indeed relies on management protocols to model the management behaviour of the components forming an application and to derive the overall application management behaviour. The Model Manager also features the logic for automatically determining a plan allowing the application to reach its target state. The latter is implemented as follows. A planner first finds the shortest sequence of operations allowing to reach the target state from the current application state, with a breadth-first-like search over the labelled transition system modelling the overall application management behaviour. It then collapses the determined operation sequence in a sequence of parallel steps, with each step corresponding to the parallel execution of operations that cannot impact one other, as applied to components pertaining to separate portions of the application topology, i.e., such that none of the components in a portion depend on a component in the other portion, nor on components that depend on components in that portion, and so on.

The Model Manager encapsulates the above logic behind a REST API. The API provides a method to POST a new TOSCA application specification, whose topology and management protocols are parsed to create an internal representation of the application itself. If no management protocol is associated with a component, the management protocol modelling the default lifecycle of Brooklyn entities is considered (Fig. 3). The Model Manager also features methods to PUT the actual and target application state and to GET the management operations to be executed in parallel to allow an application to get closer and eventually reach its target state from the actual one.

The Orchestrator. The Orchestrator is realised as a Java-based REST API.[10] The REST API provides a method to POST the TOSCA specification of an application (including the description of the management protocols of its components) and the target state for such application. Once POSTed, the Orchestrator suitably interacts with the REST APIs of the Trans-Cloud Controller and Model Manager to forward them the TOSCA application specification and to provide the Model Manager also with the target application state. The Orchestrator then starts invoking the APIs of the Trans-Cloud Controller and Model Manager to coordinate them and realise the deployment and recovery approach described in Sect. 3.1.

4 First experiments: robust deployment and operation

The proposed self-healing, trans-cloud application management platform is intended to supervise the application during its entire lifecycle. In this section we illustrate its functioning both during the deployment and during the operation of applications. Specifically, in this section we present two scenarios using the Softcare application (Sect. 2). In the first one, we show how the system responds when a failure occurs during its deployment. Specifically, we show how failures are detected and how a new sequence of actions is executed to recover the application from its failure state to the state in which it is fully operational. In the second scenario, failures are injected on specific components during the normal operation of the applications. This enables to provide details on the interactions between the components and the actions executed until the application recovers its normal activity. Both scenarios use the locations depicted in Fig. 1, viz., SoftwareWS is deployed using Pivotal (PaaS) services, while all other components of *Softcare* are deployed on AWS (IaaS).

The scenarios in this section highlight the main features of our proposal: (i) the Trans-Cloud Controller enables operating applications across different providers (AWS and Pivotal in this case), (ii) the Trans-Cloud Controller provides a monitoring mechanism that allows the system to promptly detect failures, (iii) the Model Manager keeps a model of the application that includes inter-component dependencies, and it suggests actions to minimize the instability periods of the application.

Deployment-Time Recovery. The sequence diagram in Fig. 4 shows a fragment of a deployment of the Softcare application where the service on which the Forum component is to be deployed fails. The sequence diagram illustrates the interactions between the Orchestrator, the Model Manager and the Trans-Cloud Controller. The diagram was automatically excerpted from the logs produced by the three components while enacting the considered deployment.

The Orchestrator first sends to the Model Manager the application topology and target state (steps 2–3), and sends to the Trans-Cloud Controller the application topology. If both of them accept the topology, then an accept message is submitted to the user (step 5). The status of the components is then read and submitted to the Model Manager (steps 4–6)—the initial status of all components is the not-available state (see Fig. 3). The Orchestrator then starts (step 9) components ForumDB and Soft-

[10] The Orchestrator: https://github.com/scenic-uma/lifecycle-orchestrator.

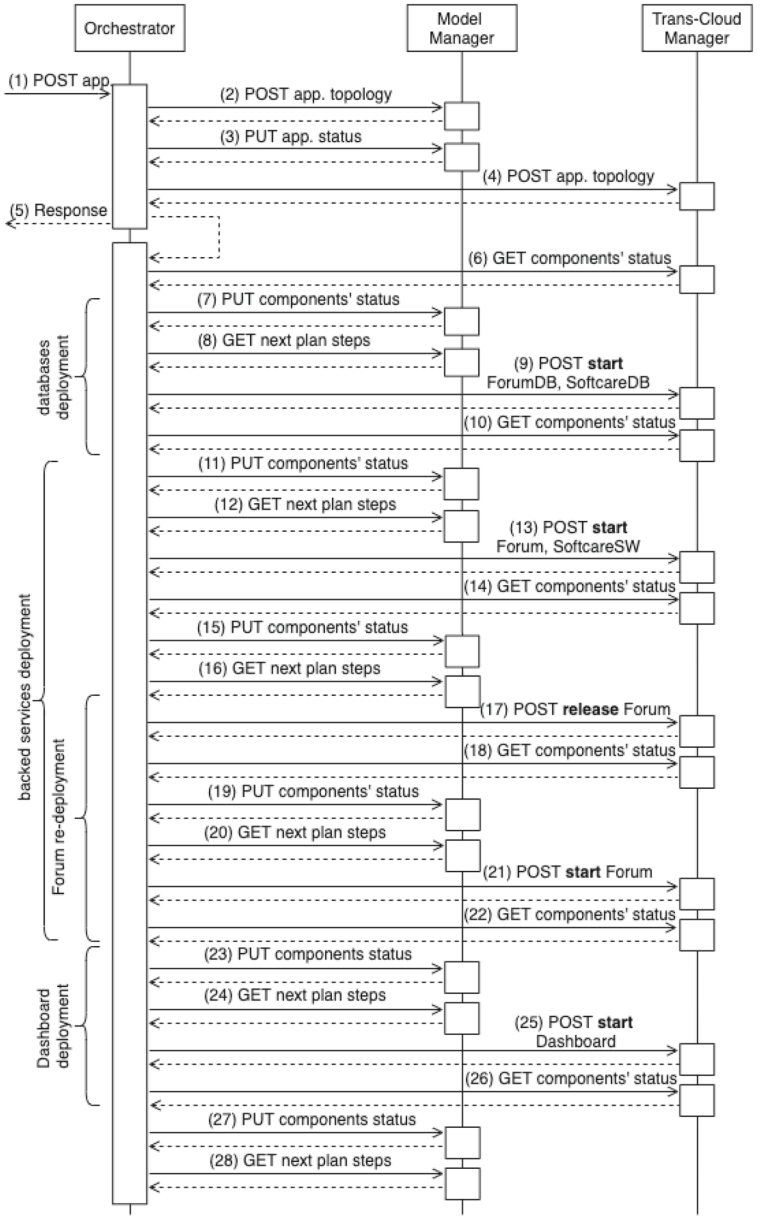

Fig. 4 Fragment of the interaction between the Orchestrator, the Trans-Cloud Controller and the Model Manager during a deployment of the Softcare application

careDB, which do not depend on others, as suggested by the Model Manager (steps 7–8). Then the Orchestrator gets from the Trans-Cloud Controller the status of all components (step 10), forwards it to the Model Manager (step 11), and gets from the latter the next recommended actions to take (step 12).

At step 13, the Orchestrator asks the Trans-Cloud Controller to start the Forum and SoftcareWS components. The Forum component fails while being started (because of an "unforeseen" failure of the cloud resources requested for it). The Orchestrator gets the actual state of the application (step 14) and forwards it to the Model Manager (step 15). The latter detects a mismatch between the actual state and the expected state of the application. It hence suggests the Orchestrator to release the resources requested for the Forum component and to start a new instance of Forum (steps 16–21). The deployment then proceeds with no further failures: the Dashboard component is deployed (steps 23–25), and the Orchestrator then continues monitoring the operation of the application (steps 26–28).

Operation-Time Recovery. In this scenario, during the normal operation of the application, failures are injected on the AWS VMs on which the ForumDB and SoftcareDB components are deployed. It shows a full scenario, where components at the bottom of the dependency hierarchy of the application fail and need to be re-deployed on IaaS services. The Model Manager suggests to stop the components depending on them to minimize application instability. After the failed components have successfully been recovered, stopped components are restarted by following a bottom-up strategy. As a result, even if the two components experimenting failures are deployed at IaaS level, each on a different VM, their recovery involves several other entities at both IaaS and PaaS level that must be stopped and restarted.

The sequence diagram in Fig. 5 shows the sequence of messages interchanged by the Orchestrator, Model Manager, and Trans-Cloud Controller to recover the application from failures once they have been detected. First, the Dashboard component is stopped (step 4), then the Forum and the SoftcareWS components (step 8). Once all components depending on the failed ones are stopped, the ForumDB and SoftcareDB components are released (step 12) and started (step 16). The procedure continues by ensuring that no component is started until those it depends on are already up. Note that the Trans-Cloud Controller is in charge of additional operations in background, e.g., allocating VMs and deploying binaries.

To get an idea of the execution times dispersion of each of the operations performed, the scenario, like all other scenarios in this paper, was run 10 times. Figure 5 also shows its boxplot. Time zero in this and following figures represent the time at which the failure is detected. In this scenario, the two failures were simultaneously injected, but, in general, depending on the frequency of the monitoring polling mechanism, it may happen that even if failures do not happen simultaneously, they are detected at the same time. The red and green vertical lines highlight, respectively, the time at which the Dashboard component is stopped and restarted. In this and the rest of the boxplot diagrams in this article, the time elapsed between these lines show the downtime of the application. Notice also that the parallelization of operations speeds up the procedure. In this case, the start of ForumDB and SoftcareDB components implies the allocation of VMs and deployment of binaries. The most time-wise-expensive operations take place concurrently to save time while executing the procedure. Although this is a very simple scenario, it is easy to foresee potentially significant gains in more complex scenarios.

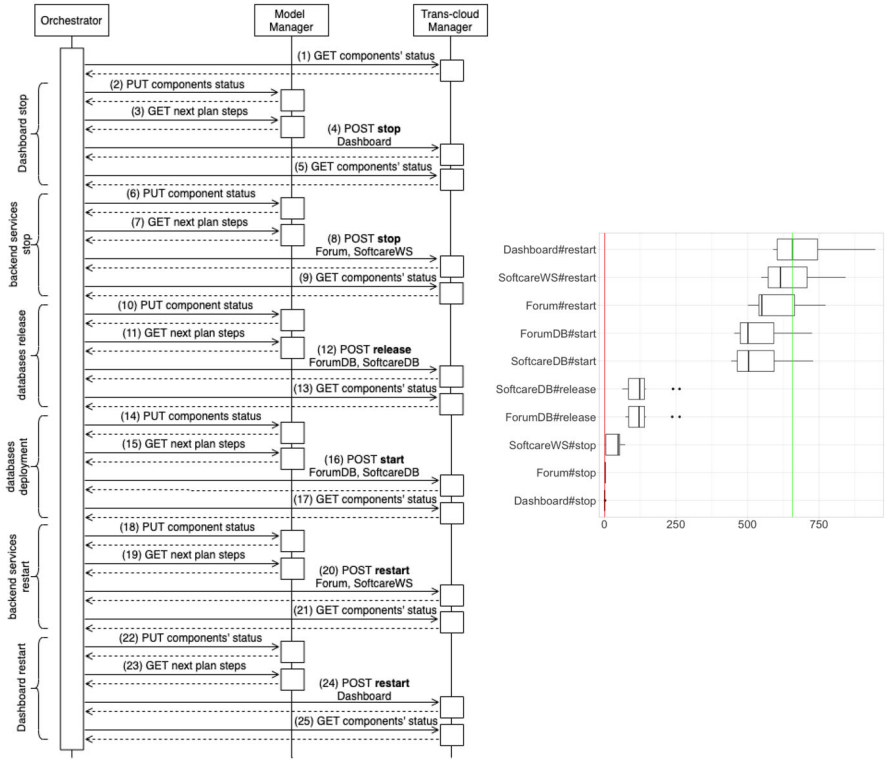

Fig. 5 ForumDB and SoftcareDB fail: sequence diagram (left) and box plot (right)

With a recovery mechanism as those available in providers such as AWS, Azure, or OpenStack, upon the detection of a failing service instance, such instance would be released and a new one provisioned to substitute it. Note however, that these solutions do not take into account the topology of the application to identify dependencies and to minimize the time the application is unstable. Our claim is that such technologies could still be improved if they used the topology of the applications, and appropriate on-the-fly strategy computation, like in our proposal. In our case, the components that depend on the failed ones are immediately stopped, and their restart is performed also taking into account components' inter-dependencies.

5 Controlled experiments

The trans-cloud monitoring infrastructure may report two types of failures. A component may be unresponsive because it is experiencing some internal problem, or because it is running on some failed infrastructure—a component deployed on a PaaS service or on a VM may fail or stop. On the other hand, a VM may also *crash*, or a Tomcat server running on a PaaS service may crash, leaving the components deployed on them unresponsive. In both cases, the application service fails, but in the second

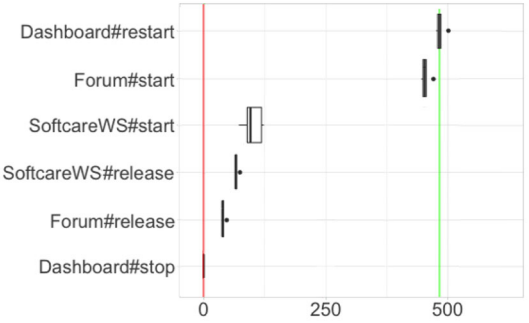

Fig. 6 Forum and SoftcareWS crash

one the underlying VM or server is also unresponsive. Although from an operational point of view both situations are very similar, from a recovery point of view they are quite different. If a VM crashes, a new VM must be allocated and the application binaries loaded before the service is restarted. However, if an application service fails, it may be enough to restart such a component.

We hereafter present various different situations that occurred on a running deployment of the Softcare application. All such situations are intended to enable analyzing cases in which components in different positions in the dependency hierarchy and locations fail, stop, or crash, and others in which a failure occurs while the application is recovering from a previous failure.

Repeated failures: Crashes of Forum *(AWS) and* SoftcareWS *(Pivotal).* The second scenario in Sect. 4 presents a situation in which the failure of two components is detected simultaneously. However, failures may occur at any time, also while recovering from a previous failure. In this scenario, the Forum component crashes, and then, while the system is recovering from such failure, the system detects a failure of the SoftcareWS component. After the release of the Forum component, the Orchestrator is notified of a failure in the SoftcareWS component, which leads to its release. In this case, the plan consists of the following sequence of operations: stop Dashboard, release Forum, release SoftcareWS, start Forum, start SoftcareWS, and restart Dashboard. Figure 6 shows the times for this scenario.

This is an interesting case, in which two components in the middle level of the dependency hierarchy, respectively deployed on IaaS and PaaS services, fail, but where one of them fails while the recovery of the other one is ongoing. Note that although the start of the SoftcareWS and Forum components is initiated simultaneously, the start of SoftcareWS involves the reallocation of a PaaS service, which requires a much shorter time than the allocation of a VM necessary for the Forum component, which is deployed using IaaS services. Once both components are fully recovered, the Dashboard component is restarted. Note that, as expected, the procedure begins by stopping any components that depend, directly or indirectly, on the failed components. In this case, it is only the Dashboard component, which just needs a restart, which is performed very quickly.

ForumDB *(AWS) and* SoftcareDB *(AWS) Stop.* This scenario presents a similar situation to the one in Sect. 4, but with ForumDB and SoftcareDB components stopping

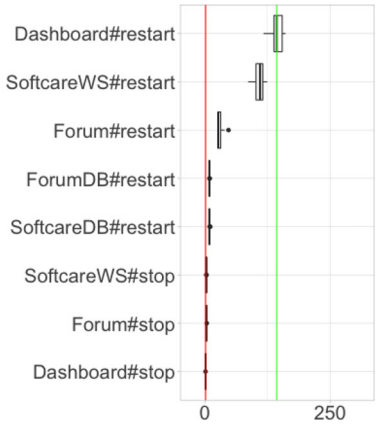

Fig. 7 ForumDB and SoftcareDB stop

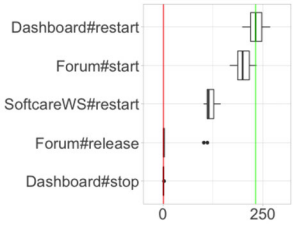

Fig. 8 Forum crashes, SoftcareWS stops

because from internal problems (instead of crashing), and with the VMs where they operate remaining responsive. In this case the recovery process involving the rest of the dependency hierarchy is the same, but these two nodes can get back to normal just by restarting them. Figure 7 shows the boxplot for this scenario.

By comparing the boxplot charts in Figs. 5 and 7, we can observe that the fact that new VMs do not need to be allocated makes the second recovery much faster. Specifically, the start operations of the ForumDB and SoftcareDB take several minutes in the scenario depicted on the right, while their restart take just a few seconds in the scenario depicted in Fig. 7. The recovery time is much shorter in this case, and clearly exemplifies the impact of the ability of distinguishing between component crashes and failures.

Forum *(AWS) crashes and* SoftcareWS *(Pivotal) Stops.* In this scenario, the PaaS entity SoftwareWS is stopped and a failure is injected to the IaaS entity running Forum to make it crash. The scenario illustrates how IaaS entities are stopped and restarted if the VMs they run on are unresponsive, how PaaS services are released and re-deployed in case of errors, and how Tomcat servers must be re-deployed or restarted on both IaaS and PaaS.

Figure 8 shows the boxplot for this scenario. As soon as the failure is detected, the only component depending on the failed components is stopped. Then, concurrently, the IaaS entity Forum is released and the SoftcareWS component is restarted. Once both operations have finished, the Forum component is started. Finally, the Dashboard component is restarted.

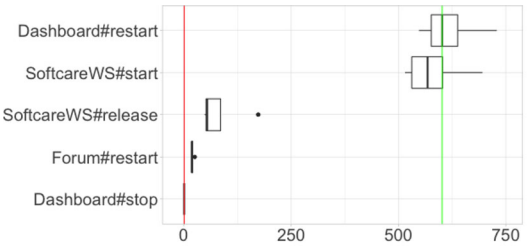

Fig. 9 Forum stops, SoftcareWS crashes

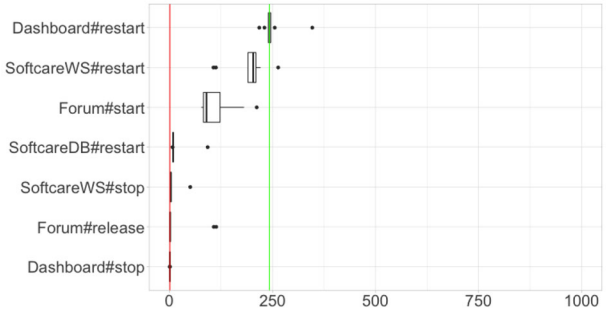

Fig. 10 Forum crashes and SoftcareDB stops

Forum *(AWS) Stops and* SoftcareWS *(Pivotal) Crashes.* In this scenario the PaaS entity SoftwareWS crashes and the IaaS one Forum stops. This scenario is very similar to those in Sects. 4 and 5. However, it shows that even though the SoftwareWS is allocated on PaaS services, its crash implies the restart of a Tomcat server, which may take a significant amount of time. Figure 9 shows the boxplot for this scenario. In the figure, we can see how most of the time of the recovery is taken by the start operation on the SoftcareWS component. Again, notice that the Dashboard is stopped right after detecting the failure, which guarantees that it is not operating while the components it depends on are failed/being recovered.

Forum *(AWS) crashes and* SoftcareDB *(AWS) stops.* In all previous scenarios the failed entities were at the same level in the dependency hierarchy. In this scenario the IaaS entity SoftwareDB stops and the PaaS one Forum crashes. In this case, errors were injected to entities on the second and third levels of the dependency hierarchy. It shows how PaaS entities are released/redeployed, how IaaS entities are restarted because of an error, how Tomcat servers must be redeployed on PaaS in case of need, and how MySQL must be restarted on IaaS. Figure 10 shows the boxplot for this scenario.

6 Monkey testing

The previous two sections show various different scenarios in which stop and crash failures were injected to show different interesting situations, with the aim of illustrating the operation of the system and providing precise information on its operation. In this

section, we describe a stress-test of the prototype implementation of our self-healing methodology, based on Monkey Testing [4].

Monkey or random testing basically consists on providing random inputs to check the behavior of applications and the infrastructure they operate on. Depending on the context, monkey testing can be carried out in different ways, but it can always be reduced to feeding the systems under test with random events for long enough to be able to ascertain their reliability. In the cloud context, monkey testing became popular with Netflix's Chaos Monkey and with AWS's Fault Injection Simulator.[11] The Chaos Monkey was designed with the goal of testing system stability by introducing failures via the pseudo-random termination of instances and services. By pseudo-randomly making their own hosts fail, they could realize weaknesses and validate that their automated remediation system worked correctly. In the case of Netflix, all instances are allocated on AWS. Therefore, the Chaos Monkey randomly terminates VM instances and containers that run inside a production environment. The principles of the testing tool have inspired developers to implement similar tools for different technologies, for example, Kubernetes clusters,[12] Azure Service Fabric,[13] Docker,[14] or private cloud infrastructures.[15]

Following the above ideas, we have developed a tool for monkey testing our self-healing, trans-cloud application management platform. Our tool randomly introduces stop and crash failures on any of the components of the application under test, which is running on a trans-cloud environment, so that components may be running on IaaS and PaaS services of different providers. The recovery system detects such failures and takes the necessary steps to recover the application to its normal operating state. Since the monkey is continuously operating, it may indeed happen that new failures are injected during the recovery of a previous failure. The monkey operates with no control, and in fact these failures may even occur in cascade. This is indeed the most interesting part, to see how these recovering stages overlap and accumulate.

The monkey is controlled by a pseudo-random number generator that guides failure injection. Figure 11 shows the operation of the recovery system under the attack of the trans-cloud monkey for an execution of 24 hours, guided by a normal distribution with mean 8 and variance 5 (minutes). The chart shows the number of active components along time. With these parameters, the monkey is being quite aggressive, although not so much that the application cannot recover after some time. We can see that although the system is completely operational in some periods of time, there are times in which up to four components are down. Vertical bars represent failure injections.

Above and below the main chart in Fig. 11, we can see magnified excerpts of the evolution of the system. Let us begin by focusing on the interval (a) 46462–47009, zoomed out at the bottom of the figure. This is one of the simplest situations, that initiates when an internal failure on the Forum component is simulated by injecting a

[11] AWS failure injection simulator: https://aws.amazon.com/fis/.

[12] The Chaos Toolkit: https://chaostoolkit.org/.

[13] Failure Analysis Service: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-testability-scenarios.

[14] Gremlin's Failure as a Service for Docker: https://www.gremlin.com/.

[15] Muxy: https://github.com/mefellows/muxy.

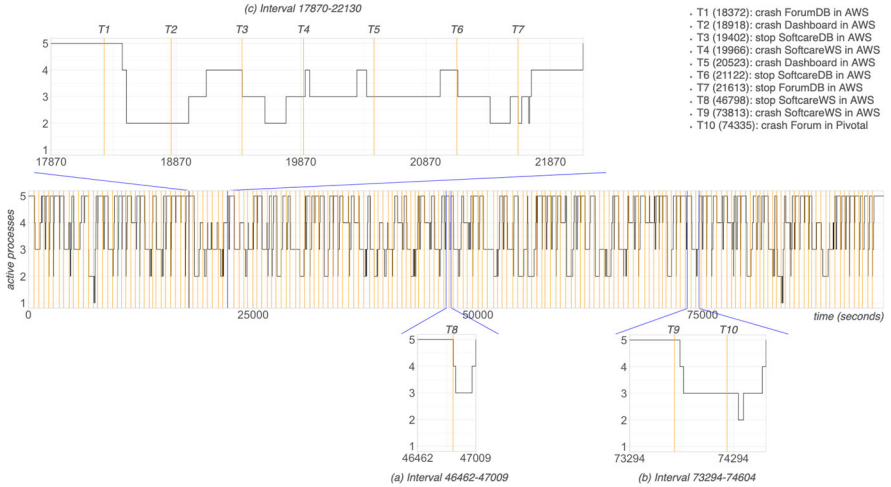

Fig. 11 Evolution of the Softcare application under the control of the monkey

stop failure. When the problem is detected, the Dashboard is stopped and the *Softcare* application is then recovered without any further incidents.

The second excerpt at the bottom, viz., interval (b) 73294–74604, shows a situation that starts with the five components normally running. The system takes a few seconds to detect the failure, an injected crash on the SoftcareWS component. When the problem is detected, the Dashboard is stopped. Then, while the SoftcareWS component is being started a new failure is injected by the monkey, in this case a stop on the Forum component. The system completes its recovery without further failure injection.

An even more interesting case is extracted at the top of the figure. The fragment corresponds to the interval 17870–22130, which begins with five active components. All components remain active until a crash failure is injected on the ForumDB component. The monitoring polling system takes some time to notice the failure, but, when it does, it initiates the recovery by first stopping the Dashboard and the Forum. When the ForumDB component is started, a new failure is injected. In this case it is a crash failure on the Dashboard, which was still down. After the ForumDB and Forum components are back running, and while the Dashboard was being started, a new failure is injected, in this case a stop on the SoftcareDB. This failure forces the system to also stop the SoftcareWS component, leaving only two active components. The next failure is a crash on the SoftcareWS component when it was starting. The graph shows how several other failures are injected on different components until the system can finally come back to normal at the end of the depicted interval. The specific injected failures are shown on the right top corner of the figure.

Other similar experiments have been run for longer periods of time. The transcloud support always recovered all components independently of when, which, or how components were failing, demonstrating the effectiveness of our approach in any

of the considered tests. The code is publicly available and the interested reader is welcome to further test it.[16]

7 Related work

The need for self-configuring and self-healing cloud applications is well-known. Dai et al. were among the firsts investigating self-healing solutions for cloud computing. In [14], they proposed a solution for consequence-oriented diagnosis and healing. More precisely, they combined multivariate decision diagrams and Naïve Bayes classifiers to predict consequences (rather than causes) from the symptoms affecting applications and to enact countermeasures to avoid degradation of applications in the event of foreseen failures. Our approach is instead more recovery-oriented [9], as we enable detecting failures and recovering applications, even if failures were not foreseen or happened while an application is first deployed. In addition, while Dai et al. focus on single cloud applications, our approach is designed to support the failure-aware operation of applications over multiple clouds and at different service levels.

SeaClouds [6] and MODAClouds [2] are two other noteworthy attempts to master the need for multi-/trans-cloud application deployments. They both enable specifying multi-component applications, together with their SLA (Service Level Agreement) and QoS (Quality of Service) requirements. Based on this, SeaClouds and MODAClouds discover the best distribution for an application across multiple clouds and service levels, by also enabling to concretely enact such application deployments. Unfortunately, neither SeaClouds nor MODAClouds provide a full-fledged self-healing solution like ours. They can monitor the environment, analyze the current status, and validate the non-functional requirements. They also can propose recovery plans in case some foreseen failures happen, but they always require some kind of human intervention in some point of their processes.

In this perspective, there exist various solutions for fully-automated self-healing of cloud applications. However, to the best of our knowledge, none of such solutions can handle the recovery of both foreseen and unforeseen failures affecting both application components and the IaaS/PaaS resources used to host them. We hereafter discuss such solutions and position ours in their respect. We first focus on self-healing solutions outcoming from research works and then we discuss how self-healing is realized in commercial solutions.

Research on Self-Healing for Cloud-based Applications. In [29], Singh Gill et al. present RADAR, which integrates cloud infrastructure with a comprehensive monitoring system to detect and predict failures in cloud resources, together with mechanisms for the execution of corresponding recovery processes. For this, RADAR needs details on the characteristics and restrictions of an application, which need to be updated whenever updates are applied to the applications, e.g., usage of new resources or migration of some components. This means that the automatic changes by the self-healing system and the manual updates of these restrictions need to be synchronized. The decoupling of these two parts is one of the main challenges addressed by our proposal.

[16] Trans-cloud Monkey Testing: https://gitlab.com/kiuby88/robust-monkey/.

In [30], Stack et al. propose a distributed solution for self-healing based on layers: lower layers monitor the status of system's components, which they then forward to a higher, centralized layer. The latter processes the received data to detect failures. If errors are diagnosed, then recovery operations are applied. This solution can only manage information and process the resources from an infrastructure's point of view, and it cannot operate to solve errors maintaining the integrity of applications. Moreover, the proposal by Stack et al. is tailored to work only with applications deployed on IaaS infrastructures.

The above-mentioned proposals, as well as those in [12,15,17,20,23], can determine and enact recovery plans, based on the analysis and simulation of the foreseen failures that can affect application components when they are up and running. Our approach goes a step further by exploiting monitoring to detect failures and by enabling to automatically recover application components from failures, even if such failures are unforeseen or happen while an application is being deployed.

In [28], Ray et al. propose VM re-allocations to manage failures in the cloud, similar to other proposals, such as [15]. However, authors go one step further in [28] since they propose to have re-allocations in federated clouds having a failure tolerance system that is able to preempt failures proactively. Moreover, using different metrics, such as the CPU temperature, failures can be predicted. Then, the federation is requested to find idle resources in the associated vendors in order to migrate failing resources or workloads. Once candidates for the movement are found, the cost of the re-allocation is analyzed in term of the cost and the impact in the federation before redistributing the resources. This solution is able to manage and distribute resources using different providers, but it is focused to work on federated environment, while our proposal can manage IaaS and PaaS resources offered by different (non-federated) providers. Moreover, this approach does not have mechanisms to recover failing components, self-healing, and redistribution of services are applied even if only a rolling-restart would be enough.

Other solutions, such as those proposed in [22,33], follow a less intrusive approach, by applying a passive inspection based on log analysis. These solutions are not predictive and they can only react to errors that have already happened, like our proposal, to apply some recovery operations. Indeed, logs analysis reduces the effort necessary for system analysis, since it does not require adding to applications or to the infrastructure a custom status producer or collector to have an overall view of the current system's status. At the same time, log analysis-based solutions are intended to work with IaaS-based or on-premise infrastructures. To adapt them to work with multi-cloud environments, we would need some centralized log collector storing the log traces sent by the multiple clouds forming the cross-cloud system, hence resulting in a highly impacting, costly, and network consuming solution.

In [1], Alhosban et al. propose a solution in which runtime events are stored and analyzed to assess the likelihood of possible failures to occur in a cloud-based application, as well as to determine beforehand plans for recovering the application from such failures. When foreseen failures occur, recovery plans are enacted for recovering the application. In addition, to adapt to runtime application changes, recovery plans can also be dynamically determined if the statically determined ones are not enough to recover the application from a foreseen failure. While this results in a self-healing

solution similar to ours, it is worth noting that the solution by Alhosban et al. copes with foreseen failures (already occurred in some of the stored runtime events), while our solution can recover applications also from unforeseen failures, first happening to some application component. In addition, while the solution by Alhosban et al. works with applications deployed in IaaS clouds, our solution is agnostic to the cloud service level.

In [21], Li et al. propose the use of a cloud management platform to decouple application administrators from infrastructure providers. Applications and infrastructures are instrumented and registered in a centralized system. Then, information is collected and processed according to the configurable monitoring rules, to generate recovery plans when failures are detected. A solution for OpenStack is provided, where monitoring is considered only after deployment has been concluded. By taking advantage of the trans-cloud's monitoring capabilities, our proposal can observe the earliest stages of the applications' lifecycle, e.g., enabling to recover applications from failures also during their deployment. Moreover, Li et al. focus on application management and recovery in OpenStack-based IaaS clouds, whereas our solution is independent from the actual techonology or service level used to deploy applications.

Several works have recently proposed failure-tolerant approaches based on the replication of resources. In [31], Tomás et al. propose a Disaster Recovery Layer built on the OpenStack platform by extending some of its components. The layer allows to have cloud resources, such as workload, applications, virtual machines, and storage systems, in a primary datacenter whose status is synchronized with backups in an secondary datacenter. Authors have also developed a robust and distributed disaster detection system to identify the status of the resources across different datacenters, allowing failures to be detected in order to redirect the traffic to backup copies. Moreover, they propose a robust protocol for the automatic synchronization and replication of resources and data based on snapshots that minimize the traffic that is able to work over different kinds of network profiles with a variable latency and bandwidth. So, the platform can detect and react automatically to failures maintaining the reliability of the systems. The amount of backup resources limits the tolerance to the amount of chained failures that can be managed. This solution requires a large number of replicated resources, which implies huge costs for having a backup system for massive workloads. Scalability is also an important problem of this solution, since the increment of size and complexity of systems and applications also increments the transient failures, for example, in the infrastructure, what would require even more duplicated resources to avoid SLA violations, increasing the redundant cost. In [32], Xie et al. propose a solution also based on replication, but using a model to describe workflows and the cost of the resources, which is used to reduce the duplicity of resources, but maintaining the QoS and the reliability of large workflows focused on the IaaS level. These solutions can have a better QoS than our proposal, because they would not need to restore resources that can become a time-consuming task. However, they do not apply self-healing techniques, what means systems cannot auto-recover when failures happen. They can only use a different set of non-failing resources to work, what allows users to assume the cost of the redundant resources and the network usage.

Self-Healing in Commercial, Production-Ready Solutions. Some current commercial clouds natively feature self-healing to recover cloud service instances. Some of them have also recently released mechanisms allowing users to incorporate self-healing techniques to monitor and repair their own applications. For example, AWS allows to set up CloudWatch[17] alarms to notify users or trigger recovery plans upon the occurrence of some event, specified in terms of predefined EC2 instance metrics. However, only some limited operations can be performed to stop or restart the corresponding machines, and this feature is only available for some kinds of instances. Azure goes one step further since it does not only allow reacting to failures, but also storing them to learn about the actual behaviour of a deployed application. Then, heuristics can be applied to predict imminent failures. Google Cloud and Alibaba Cloud[18] also provide health check mechanisms to determine if VM instances are responding as expected. They are used to maintain the high availability of instance groups, but they also allow detecting failures in VM instances and apply some repairing operations. Even if all such solutions effectively enable to self-heal failed components and cloud resources in application deployments, they are bound to specific cloud providers or cloud service levels.

Container-based deployment automation technologies also feature mature self-healing solutions, e.g., Kubernetes and Swarm[19] natively include observability hooks to check the container status and to operate on them if needed. For example, the manifest of a Kubernetes deployment allows to describe operations to check when the container deployment has been completed (readiness) and check periodically if the container is running correctly (liveness). Some retry or recreation policies can be specified if these indicators show errors. In this case, Kubernetes will be in charge of checking the containers status and if failures happen during building, or once the application is running, Kubernetes can remove the failed container and create a new one.

However, like some authors have already pointed out (see e.g., [8,26], or [27]), Kubernetes and other container-based technologies present some problems to provide comprehensive topology of applications because the information on which components are allocated in each container is not explicit, which makes difficult to identify relations between containers to preserve the relations of the application's components. Then, although Kubernetes and Swarm can easily recover failures in a concrete part of the system, carrying out rolling-restart or even roll-out, the impact on the rest of the application is not considered, hence possibly inducing instability to the components depending on the rolled-out components. Indeed, this is one of the questions we solved in the domain of our work.

To summarise, commercial clouds and container-based deployment automation technologies natively features valuable solutions for self-healing and self-configuration of cloud applications. At the same time, such solutions are bound to some cloud, cloud service level, or technology, hence not supporting the failure-aware management of applications in trans-cloud scenarios. In addition, they currently do not

[17] CloudWatch: https://aws.amazon.com/es/cloudwatch/.

[18] Alibaba Cloud: https://alibabacloud.com/.

[19] Docker Swarm: https://docs.docker.com/engine/swarm/.

account for components depending on failed components, which are left in unstable states where they can suddenly fail or, even worse, deliver erroneous behaviour. Our claim is that they could still be improved with suitable, on-the-fly recovery strategies taking into account the topology of deployed applications, like that we propose in this article. Our proposal indeed provides what—to the best of our knowledge—is the first solution for managing trans-cloud applications, enabling to self-heal their components from both foreseen and unforeseen failures, and reducing instability periods for components depending on failed components.

8 Concluding remarks

This article presents a self-healing methodology supporting the automated management of foreseen and unforeseen faults in trans-cloud scenarios, where application components are deployed on different platforms and at different service levels (IaaS or PaaS). We have also shown a prototype implementation of a platform implementing such a methodology, which we exploited to carry out exhaustive experimentation, including monkey testing for long periods of time. The tool requires a TOSCA description of the application to manage, which is used to automatically deploy and monitor its operation, and to construct recovery plans. Upon the occurrence of a failure, the tool is then able to recover from application and underlying services failures, minimizing instability periods.

As future work, we plan to further improve the performance of the current prototype. Potential aspects to be improved include the time needed to determine plans for recovering failed application components, and the overhead and footprint induced by the monitoring activities. We are currently studying how performance can be improved by better tuning the scheduling of monitoring actions, and by introducing heuristics in the planning (possibly also taking into account the actual "cost" of executing management operations on application components, e.g., in terms of their execution time or resource consumption). We also plan to strengthen the reliability of our solution and to extend it so as to robustly manage applications involving components developed and managed by third-parties. Finally, we plan to extend the recovery mechanisms to be able to recover from failures occurring during the live migration of trans-cloud applications, e.g., by adapting it to work with the trans-cloud migration approach proposed in [11].

Funding Open access funding provided by Universitá di Pisa within the CRUI-CARE Agreement.

References

1. Alhosban A, Hashmi K, Malik Z, Medjahed B (2013) Self-healing framework for cloud-based services. In: International conference on computer systems and applications (AICCSA). IEEE, pp 1–7. https://doi.org/10.1109/AICCSA.2013.6616511
2. Ardagna D, Di Nitto E, Casale G, Petcu D, Mohagheghi P, Mosser S, Matthews P, Gericke A, Ballagny C, D'Andria F, Nechifor C, Sheridan C (2012) MODAClouds: a model-driven approach for the design and execution of applications on multiple clouds. In: International workshop on modeling in software engineering (MiSE). IEEE, pp 50–56. https://doi.org/10.1109/MISE.2012.6226014
3. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53(4):50–58. https://doi.org/10.1145/1721654.1721672
4. Basiri A, Behnam N, de Rooij R, Hochstein L, Kosewski L, Reynolds J, Rosenthal C (2016) Chaos engineering. IEEE Softw 33(3):35–41. https://doi.org/10.1109/MS.2016.60
5. Brogi A, Canciani A, Soldani J (2018) Fault-aware management protocols for multi-component applications. J Syst Softw 139:189–210. https://doi.org/10.1016/j.jss.2018.02.005
6. Brogi A, Carrasco J, Cubo J, D'Andria F, Di Nitto E, Guerriero M, Pérez D, Pimentel E, Soldani J (2016) SeaClouds: an open reference architecture for multi-cloud governance. In: Software architecture. Lecture notes in computer science, vol 9839. Springer, pp 334–338. https://doi.org/10.1007/978-3-319-48992-6_25
7. Brogi A, Carrasco J, Durán F, Pimentel E, Soldani J (2019) Robust management of trans-cloud applications. In: International conference on cloud computing (CLOUD). IEEE, pp 219–223. https://doi.org/10.1109/CLOUD.2019.00046
8. Brogi A, Rinaldi L, Soldani J (2018) TosKer: a synergy between TOSCA and docker for orchestrating multicomponent applications. Softw Pract Exp. https://doi.org/10.1002/spe.2625
9. Candea G, Brown AB, Fox A, Patterson D (2004) Recovery-oriented computing: building multitier dependability. Computer 37(11):60–67. https://doi.org/10.1109/MC.2004.219
10. Carrasco J, Durán F, Pimentel E (2018) Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud. Comput Stand Interfaces 58:167–179. https://doi.org/10.1016/j.csi.2018.01.005
11. Carrasco J, Durán F, Pimentel E (2020) Live migration of trans-cloud applications. Comput Stand Interfaces. https://doi.org/10.1016/j.csi.2019.103392
12. Chen G, Jin H, Zou D, Zhou BB, Qiang W, Hu G (2010) Shelp: automatic self-healing for multiple application instances in a virtual machine environment. In: International conference on cluster computing (CLUSTER). IEEE, pp 97–106. https://doi.org/10.1109/CLUSTER.2010.18
13. Cotroneo D, De Simone L, Liguori P, Natella R, Bidokhti N (2019) How bad can a bug get? An empirical analysis of software failures in the openstack cloud computing platform. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2019. ACM, pp 200–211. https://doi.org/10.1145/3338906.3338916
14. Dai Y, Xiang Y, Zhang G (2009) Self-healing and hybrid diagnosis in cloud computing. In: International conference on cloud computing (CloudCom). Lecture notes in computer science, vol 5931. Springer, pp 45–56. https://doi.org/10.1007/978-3-642-10665-1_5
15. Dewangan BK, Agarwal A, Venkatadri M, Pasricha A (2019) Design of self-management aware autonomic resource scheduling scheme in cloud. Int J Comput Inf Syst Ind Manag Appl 11:170–177
16. Endres C, Breitenbücher U, Falkenthal M, Kopp O, Leymann F, Wettinger J (2017) Declarative vs. imperative: two modeling patterns for the automated deployment of applications. In: International conference on pervasive patterns and applications (PATTERNS). Xpert, pp 22–27
17. Etchevers X, Coupaye T, Boyer F, de Palma N (2011) Self-configuration of distributed applications in the cloud. In: International confernce on cloud computing (CLOUD). IEEE, pp 668–675. https://doi.org/10.1109/CLOUD.2011.65
18. Gupta P, Gupta C (2014) Evaluating the failures of data centers in cloud computing. Int J Comput Appl 108(4):29–34
19. Laplante P (2007) What every engineer should know about software engineering. CRC Press, Cambridge
20. Li W, Zhang P, Yang Z (2012) A framework for self-healing service compositions in cloud computing environments. In: International conference on web services (ICWS). IEEE, pp 690–691. https://doi.org/10.1109/ICWS.2012.109

21. Li X, Li K, Pang X, Wang Y (2017) An orchestration based cloud auto-healing service framework. In: International conference on edge computing (EDGE). IEEE, pp 190–193
22. Lin Q, Zhang H, Lou JG, Zhang Y, Chen X (2016) Log clustering based problem identification for online service systems. In: 38th international conference on software engineering companion (ICSE-C). IEEE, pp 102–111
23. Mfula H, Nurminen JK (2018) Self-healing cloud services in private multi-clouds. In: International conference on high performance computing and simulation (HPCS), pp 165–170. https://doi.org/10.1109/HPCS.2018.00041
24. Nygard M (2007) Release it! Design and deploy production-ready software. Pragmatic Bookshelf, Raleigh
25. OASIS (2016) TOSCA Simple profile in YAML, Version 1.0
26. Pahl C (2015) Containerization and the PaaS cloud. IEEE Cloud Comput 2(3):24–31
27. Quint P, Kratzke N (2018) Towards a lightweight multi-cloud DSL for elastic and transferable cloud-native applications. In: International conference on cloud computing and services science (CLOSER), pp 400–408
28. Ray B, Saha A, Khatua S, Roy S (2020) Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment. Trans Cloud Comput
29. Singh Gill S, Chana I, Singh M, Buyya R (2019) RADAR: self-configuring and self-healing in resource management for enhancing quality of cloud services. Concurr Comput Pract Exp. https://doi.org/10.1002/cpe.4834
30. Stack P, Xiong H, Mersel D, Makhloufi M, Terpend G, Dong D (2017) Self-healing in a decentralised cloud management system. In: CloudNG@EuroSys. ACM, pp 3:1–3:6. https://doi.org/10.1145/3068126.3068129
31. Tomás L, Kokkinos P, Anagnostopoulos V, Feder O, Kyriazis D, Meth K, Varvarigos E, Varvarigou T (2017) Disaster recovery layer for distributed openstack deployments. Trans Cloud Comput 8(1):112–123
32. Xie G, Zeng G, Li R, Li K (2020) Quantitative fault-tolerance for reliable workflows on heterogeneous IAAS clouds. Trans Cloud Comput 8(4):1223–1236
33. Yuan Y, Shi W, Liang B, Qin B (2019) An approach to cloud execution failure diagnosis based on exception logs in openstack. In: 12th international conference on cloud computing (CLOUD). IEEE, pp 124–131