Recovering from Cloud Application Deployment Failures through Re-execution

Ioannis Giannakopoulos[1], Ioannis Konstantinou[1], Dimitrios Tsoumakos[2], and Nectarios Koziris[1]

[1] Computing Systems Laboratory, National Technical University of Athens, Greece
{ggian,ikons,nkoziris}@cslab.ece.ntua.gr
[2] Department of Informatics, Ionian University, Greece
dtsouma@ionio.gr

Abstract. In this paper we study the problem of automated cloud application deployment and configuration. Transient failures are commonly found in current cloud infrastructures, attributed to the complexity of the software and hardware stacks utilized. These errors affect cloud application deployment, forcing the users to manually check and intervene in the deployment process. To address this challenge, we propose a simple yet powerful deployment methodology with error recovery features that bases its functionality on identifying the script dependencies and re-executing the appropriate configuration scripts. To guarantee the idempotent script execution, we adopt a filesystem snapshot mechanism that enables our approach to revert to a healthy filesystem state in case of failed script executions. Our experimental analysis indicates that our approach can resolve any transient deployment failure appearing during the deployment phase, even in highly unpredictable cloud environments.

1 Introduction

The evolution of cloud computing has aroused new needs regarding task automation, i.e., the automatic execution of complex tasks without needing human intervention, especially in the field of application deployment. The users need to be able to describe their applications, deploy them over the cloud infrastructure of their choice without needing to execute complex tasks such as resource allocation, software configuration, etc., manually. Through automation, they can instantly build complex environments fully or semi automatically [18], enhance portability through which they can easily migrate their applications into different cloud providers, facilitate the deployment of complex applications, by decomposing the complex description into simpler and easier to debug components, etc. Nevertheless, automated resource orchestration and software configuration practices act as a prerequisite towards cloud elasticity [25], which dictates that resources and software modules must be configured automatically as in [28].

To automate the application deployment, a number of systems and tools has been proposed such as Heat [13], Sahara [14], Juju [12], CloudFormation [3], BeanStalk [4], Wrangler [21], etc. The behavior of these systems is similar: They

first communicate with the cloud provider so as to allocate the necessary resources and then execute configuration scripts in order to deploy the software components into the newly allocated resources. The configuration process can be facilitated through the utilization of popular configuration tools such as Chef [8], Puppet [16] and Ansible [1]. Since each script may depend on multiple different components (resources and software modules), each of the aforementioned systems creates a dependency graph in order to execute the configuration actions in the correct order and synchronize the concurrently executed configuration tasks.

A shortcoming of the aforementioned approaches, though, is that they do not take into consideration the dynamic and, sometimes, unstable nature of the cloud. Services downtime due to power outages, hardware failures, etc. [5], unpredictable boot times [6] closely related to the provider's load are some of the factors that contribute to the instability of the cloud. Furthermore, unpredictability is exaggerated by the fact that the current cloud infrastructures have increased the complexity of the utilized software and hardware stacks [20], something that contributes to the appearance of *transient* failures, such as network glitches [26], host reboots, etc. These failures are short-but but can produce severe service failures [11]. Furthermore, since the application deployment demands cooperation and synchronization between multiple parties including the cloud provider, the VM's services and other external parties, it becomes apparent that these transient errors are commonly found and can lead application deployment to failure, leaving, in the worst case, stale resources that need manual handling, e.g, a partially deployed application may have successfully created some VMs or volumes that need to be deleted before triggering a new deployment.

To tackle the aforementioned challenges, we propose an application deployment methodology which is able to recover from transient infrastructure errors through re-execution of the necessary scripts. Specifically, the deployment is modeled as a set of scripts concurrently executed for different software modules and exchanging messages, forming a directed acyclic graph. If a script execution fails, our approach traverses the graph, identifies the scripts that need to be re-executed and executes them. Since a script execution may be accompanied with implicit side effects, we adopt a filesystem snapshot mechanism similar to the mechanism used by Docker [9] which guarantees that a script execution remains idempotent for all the filesystem-related resources, enabling our approach to re-execute a script as many times as needed for the deployment to complete. Through an extensive evaluation for complex deployment graphs, we showcase that our approach remains effective even in unpredictable cloud environments. Our contributions can be, thus, summarized as follows:

- we present an error recovery methodology that can efficiently restart partially completed application deployments,
- we provide a powerful methodology, that allows to snapshot the filesystem and easily revert into a previous state, in case of failure,
- we demonstrate that our approach achieves the deployment of complex applications in times comparable to the failure-free cases, even when the error rate is extremely high.

2 Deployment and Configuration Model

Assume a typical three-tier application consisting of the following modules: a Web Server (rendering and serving Web Pages), an Application Server (implementing the business logic) and a Database Server (for the application data). For simplicity's sake, we assume that each module runs in a dedicated server and the application will be deployed in a cloud provider. If the application deployment occurred manually, one should create three VMs and connect (e.g., via SSH) to them in order to execute scripts that take care of the configuration of the software modules. In many cases, the scripts need input that is not available prior to the resource allocation. For example, the Application Server needs to know the IP address and the credentials of the Database Server in order to be able to establish a connection to it and function properly. In such cases, the administrator should manually provide this *dynamic* information.

In order to automate the deployment and configuration, we need to create a communication channel between different application modules in order to exchange messages containing such dynamic information. Such a channel can be implemented via a simple queueing mechanism. Each module publishes information needed by the rest of the modules and subscribes to queues, consuming messages produced by other modules. The deployment scripts are executed up to a point where they expect input from another module. At these points, they block until the expected input is received, i.e., a message is sent from another module and it is successfully transmitted to the blocked module. The message transmission is equivalent to posting a new message into the queue (from the sender's perspective) and consuming this message from the queue (from the receiver's perspective). In cases that no input is received (after the expiration of time threshold), the error recovery mechanism is enabled. To provide a better illustration of the deployment process, consider Figure 1. In this Figure, a de-

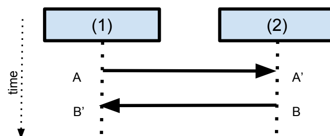

Fig. 1. Message exchange between modules

ployment process between two software modules named (1) and (2) is depicted. The vertical direction represents the elapsed time and the horizontal arrows represent message exchange[3]. At first, both (1) and (2) begin the deployment process until points A and A' are reached respectively. When (1) reaches A, it sends a message to (2) and proceeds. On the other side, (2) blocks at point A' until the message sent from (1) arrives and, upon arrival, consumes it and

[3] Note that message transmission might not be instant (as implied by the Figure) since consumption of a specific message might occur much later than the message post, but the arrows are depicted perpendicular to the time axis for simplicity.

continues with the rest of the script. If the message does not arrive, then the recovery procedure is triggered, as described in Section 3.

From the above description it is obvious that when a module needs to send a message, there is no need to wait until this message is successfully received by the other module. However, in some cases, this blocking send may be desired. For example, take the case where two modules negotiate about a value, e.g., a randomly generated password assigned to the root account of the Database Server. Assume that module (1) (the Application Server) decides on the value and sends it to module (2) (the Database Server). Module (1) must ensure that the password is set, prior to trying to connect to the database. To this end, (2) can send an acknowledgment as depicted between points B and B'. In this context, it becomes apparent that the message exchange protocol can also function as a synchronization mechanism. This scheme represents a dependency graph between the application's modules, since each incoming horizontal edge (e.g., the one entering at point A') declares that the execution of a configuration script depends on the correct execution of another. Schemes like the one presented at Fig. 1 present a circular dependency, since both modules (1) and (2) depend on each other, but on different time steps. Various state-of-the-art deployment mechanisms do not handle circular dependencies (e.g., Openstack Heat [13]) since they do not support such a message exchange mechanism during the application configuration. Furthermore, such a circularity could easily lead to deadlocks, i.e., a case in which all the modules are blocked because the wait input from another (also blocked) module. We assume that the user is responsible for creating deadlock-free deployment descriptions.

This message exchange mechanism can easily be generalized so as to be applied when an elasticity action is applied, i.e., a dynamic resource allocation during the application's lifetime. A simple *elastic* action can be decomposed into three steps: (a) the preparation step, where scripts are executed in order to prepare the application for a resizing action, (b) the IaaS communication step, so as to allocate/deallocate resources and (c) the post action step, where the application is configured in order to function properly after the IaaS action. Steps (a) and (c) can be expressed with a similar messaging/synchronization mechanism described before; Step (b), on the other hand, entails the communication with the cloud provider to orchestrate the application in the infrastructure level. Through the composition of such resizing actions, it is possible to express a composite resizing action affecting many application modules in parallel. Finally, the same mechanism is applied in cases where a module consists of multiple module instances (e.g., different nodes of a NoSQL cluster). In these cases, each module instance is addressed as a new entity and interacts with the rest modules separately.

3 Error detection and recovery

We now describe the mechanism through which the deployment errors are identified. During the deployment process, a module instance may send a message

to another module. This message may contain information needed by the later module in order to proceed with the its deployment, or it could just be a synchronization point. In any case, the second module blocks the deployment execution until the message arrives, whereas the first module sends its message and proceeds with the execution of its deployment script. In the case of an error, a module will not be able to send a message and the receiver will remain blocked. To this end, we set a timeout period that, if exceeded, the waiting module broadcasts messages to the other modules informing them that an error has occurred. From that point on, the error recovery mechanism takes control, evaluates the error, as we will shortly describe, and performs the necessary actions in order to restore the deployment procedure. Many modules may only consume messages and not create any. In such cases, a possible failure is identified with the same mechanism in the final part of the deployment. When a module finalizes its deployment, it broadcasts a message informing the rest modules that the process is finished. When all the modules receive these messages from the rest modules, the process is considered finished. Eventually, even when a module does not need to send a message, a possible failure of its execution will be identified at the final state of its execution, since it will not send a termination acknowledgment.

Error recovery is based on the repetition of the execution of scripts/actions which led the system to a failure. Given the previously described error identification mechanism, at some random point a module stalls waiting for a message from another module. It then broadcasts a special message informing the other modules that the deployment may have failed. A *master* node is then elected (among the existing modules) and this node undertakes the responsibility to execute the health check algorithm. When it identifies that a script execution has failed, it *backtracks* and informs the responsible module that it should repeat the execution of the scripts before the failure. For example, assume the more complex example provided in Figure 2.

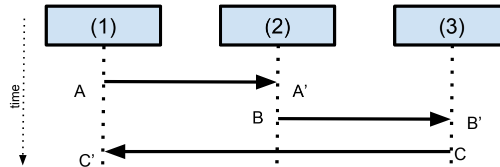

Fig. 2. More complex deployment example

Assume that an error is identified in B', meaning that (2) did not send a message to (3). Since (2) had received a message from (1) in A', this means that the problematic script is the one executed between A' and B. Let's repeat the same example now, but assume that the problem is now found in C'. After point A, (1) will remain blocked until (3) sends a message when point C is reached. If the timeout is exceeded, then (1) triggers the Health Check mechanism. In that case the master (any of (1), (2) or (3)) module will backtrack until all the dependencies for each state are resolved, and finally identify whether a failure occurred or the scripts needed more time to finish. If a problem did occur, the master node informs the responsible module and this module, accordingly, re-

executes the necessary scripts. From the above analysis, it becomes apparent that the absence of a message is indicative of a possible failure. The reason behind this absence, though, remains obscure: A script might have finished its execution and attempted to send a message but the message never reached the communication channel due to unreliable network. Our approach views all the possible alternatives in a unified manner since we make the assumption that the communication channel is also unreliable and address the channel's errors as deployment errors.

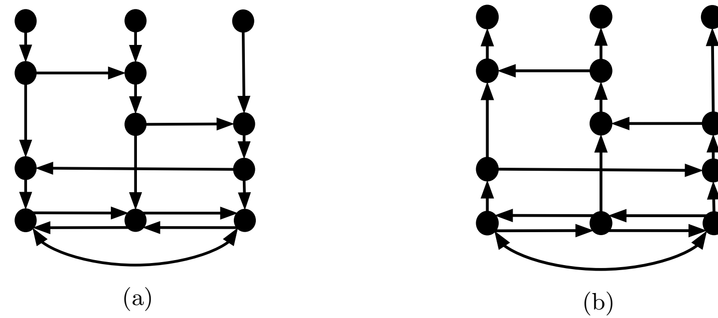

Fig. 3. Deployment and dependency graph of Figure 2

In order to identify the dependencies between the different software modules, we address the problem of error recovery as a graph traversal problem. The idea is that the messaging/synchronization scheme is translated in a graph representing the dependencies between different modules. For example, the deployment represented in Fig. 2 can be translated as shown in Fig. 3 (a). The arrows in this Figure represent either a script execution (vertical edges) or a message exchange (horizontal edges). The top nodes represent the state of each VM exactly after the VM bootstrapping has finished. The bottom graph nodes represent the final state of the deployment for each module. The intermediate nodes correspond to intermediate states of the modules, where messages are sent and consumed. At the end of the process, all the modules exchange messages once more, to verify that the process is successfully finished. Since receiving and sending messages in the same module may lead to a deadlock, we serialize the modules' actions and enforce them to first send any messages and then block to receive them.

Assume now that, at some point, an error occurs. The master node first parses the deployment graph as presented in Fig. 3 (a) and creates the dependency graph, as depicted in Fig. 3 (b), which is essentially the deployment graph with inverted edges. Then, it executes Algorithm 1 where the dependency graph is traversed in a Breadth First order, starting from the node where the failure was detected and the healthy states are, then, identified. Specifically, if a visited node is not healthy, then the graph traversal continues to its children, else the traversal stops. In this context, *healthy* states are considered to be the inter-

mediate configuration states (nodes on the graph) that have been reached by the deployment execution without an error. This way, the algorithm manages to identify the most recent healthy intermediate states reached by each module. With this information, one can easily identify which steps should be repeated.

Algorithm 1 Health Check Algorithm

Require: transpose deployment graph T, failed node n
Ensure: list of *healthy* nodes $frontier$

```
 1: failed ← {n}
 2: frontier ← ∅
 3: while failed ≠ ∅ do
 4:     v = pop(failed)
 5:     for t ∈neighbors(T, v) do
 6:         if failed(t) then
 7:             failed ← failed ∪ {t}
 8:         else
 9:             frontier ← frontier ∪ {t}
10:         end if
11:     end for
12: end while
13: return frontier
```

In various cases, script re-execution means that a module must publish a new message and its content may be different for each execution. For example, if a module creates a random password and broadcast it to the rest modules, each script re-execution generates a new password. In other cases though, the content of the message might be the same: e.g., when a module sends the IP address of the VM that is hosted into, the content of the message remains the same since it is independent from the script execution. In the former case, the modules that depend on the script-dependent messages should be re-executed as well. For example, if the Application Server sends a new password to the Database Server, this means that the Database Server should be reconfigured so as to reflect this change (even if the Database Server presented no errors), otherwise the Application Server will not be able to establish a connection with the later. Our approach views all the messages that are exchanged between different modules as script-dependent and forces the re-execution for each module that depends on a specified message. The script-independent messages should be transmitted at the first steps of the deployment so as to avoid pointless script re-execution. Such a practice is also followed by other popular deployment tools such as Openstack Heat, since all the script-independent information is considered known to every software module, prior to the execution of the configuration scripts.

To further clarify the algorithm's execution, we provide an example in which we demonstrate the way that the Health Check algorithm traverses the dependency graph when an error occurs in Fig. 4. Fig. 4 (a), (b) and (c) demonstrate the iterations of the algorithm, until the healthy nodes (green nodes) are iden-

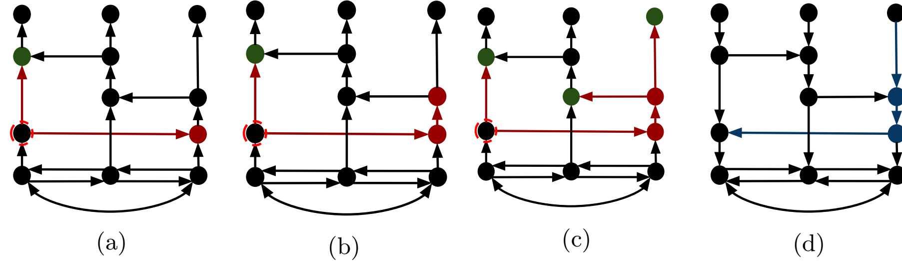

Fig. 4. Example of Health Check algorithm

tified. Fig. 4 (d) demonstrates the scripts that are re-executed so as to continue with the deployment (identified as blue edges). Two notes must be made at this point. First, the incoming vertical edge to the node in which the error was identified need not be re-executed. This is attributed to the fact that the script execution must have been successful, else the module would never have reached to the point where it expects a message, thus the Health Check mechanism would not have been triggered. Second, it is obvious that script-dependent messages need to be resent and demand script re-execution only when they are sent and not when they are consumed by a script. When a module need to re-consume an message generated by another module, we assume that the message is already available through the communication channel and the sender does not need to resend any information.

4 Idempotent script execution

The idea of script re-execution when a deployment failure occurs, is only effective when two important preconditions are met: (a) the failures are transient and (b) if a script is executed multiple times it always leads the deployment into the same state, or, simpler, it always has the same side effects, i.e., it is *idempotent*. In this, section we discuss these preconditions and describe how are these enforced through our approach.

First, a failure is called "transient" when it is caused by a cloud service and it was apparent for a short period of time. Network glitches, routers unavailability due to a host reboot and network packet loss are typical examples of such failures caused by the cloud platform but, in most cases, they are only apparent for a short time (seconds or a few minutes) and when disappeared the infrastructure is completely functional. Various works study those types of failures [26] and attribute them to the complexity of the cloud software and hardware stack. Although most cloud providers protect their customers from such failures through their SLAs [17], they openly discuss them and provide instruction for failures caused by sporadic host maintenance tasks [7] and various other reasons [19]. Since the cloud environment is so dynamic and the automation demanded by the cloud orchestration tools requires the coordination of different parties, script

re-execution is suggested on the basis that such transient failures will eventually disappear and the script will become effective.

However, in the general case, if a script is executed multiple times it will not always have the same effect. For example, take the case of a simple script that reads a specific file from the filesystem and deletes it. The script can only be executed exactly once; The second time it will be executed it will fail since the file is no longer available. This failure is caused by the side effects (the file deletion) caused by the script execution, which lead the deployment to a state in which the same execution cannot be repeated. To overcome this limitation, we adopt a *snapshot* mechanism to capture the VM's filesystem prior to script execution, allow the script to be executed and in case of failure revert it to the previous healthy state. This mechanism is based on building the VM's filesystem through the composition of different layers. The base layer consist of the VM's filesystem. Any layer on top of this contains only the updated files along with their new content. The *revert* action is equivalent to removing the top layer of the filesystem and the script's side effects are vanished. To ensure the idempotent property, we generate a new layer prior to each script execution. This way, the side effects of each script are "undoable" and, in case of failure, we guarantee that the VM's filesystem will be identical to the one before the failed script execution.

From a technical perspective, there exists various solutions which can be utilized to achieve this snapshot mechanism. In our approach, we utilize *AUFS* (Another Union FileSystem) [2], which is a special case of a Union Filesystem. AUFS allows different directories to be mounted on the same mountpoint where each directory represents a layer. The ordering of the directories determines the order of the layers into the filesystem. The addition of a new layer or the removal of an unwanted layer is equivalent to creating and removing a directory respectively. Different solutions such as overlayfs [15] and btrfs [27] could also be utilized, but all these solutions are equivalent in terms of practicality and only differ in terms of available features and performance. Since this snapshot mechanism is adopted by Docker [9], a popular Linux Container implementation, we utilized AUFS since it is considered to be the most stable and achieves the highest performance among the proposed solutions [10]. Finally, we must note that through the previously described mechanism we guarantee that only filesystem-related actions remain idempotent. Any actions that have side effects towards non disk based or external resources, e.g., external API calls, memory state updates, etc., cannot be addressed by our approach and, thus, cannot be reverted.

5 Experimental Evaluation

A deployment can be expressed in multiple different ways for one application and the execution time can vary greatly according to the used scripts. We test our methodology for various deployment graphs, as the one depicted in Figure 3, for different sizes (number of nodes and edges) and different failure probabili-

ties. The number of vertical paths represents the number of modules (or module instances). The number of states inside a vertical path is expressed as the number of *levels* of synchronization. For each graph, we create a random number of messages, i.e., two nodes belonging to the same level of different module are connected with an edge with a specified probability. This measure expresses the cohesion of the graph that is defined as the degree of dependency between different modules. Finally, during the deployment each vertical edge (script execution) may fail with a specific probability. We will study how these parameters affect the duration of the deployment process measured not in absolute time, since time is dependent to the duration of each resizing script and it greatly varies between different applications, but in number of edges.

Each module begins from an initial node of the deployment graph (for each module) and terminates in a terminal node (for the respective module). The longest path traversed, is equal to the duration of the deployments. Each module stalls when an error occurs, as described in the previous sections. We divide this path with the number of steps needed for the deployment in the optimal case, that is if no failures occurred. The results are presented in Figure 5. The left most Figure describes the relationship between the Error Rate of the scripts and the Relative Execution time for increasing number of deployment levels, whereas the right Figure depicts the same relationship for an increasing number of application modules. The cohesion factor defined as the possibility for one module to have an outgoing edge in a specific node is set to 0.2. For each graph size we created 100 random graphs and for each graph we executed the deployment algorithm 5 times. The graphs depict the averages of those runs.

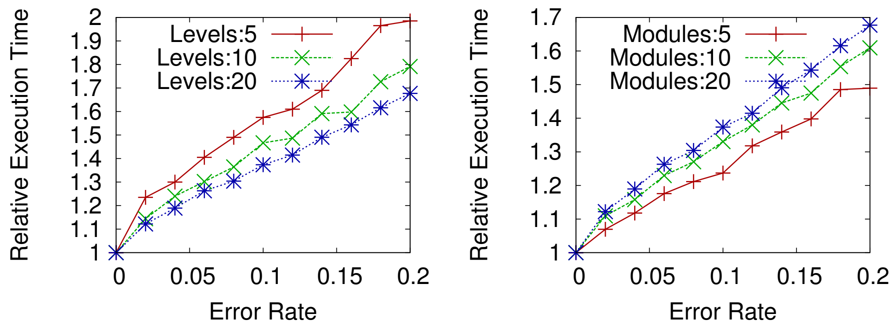

Fig. 5. Relative Execution Time vs Error Rate

In both graphs it is obvious that an increase in Error Rate leads to increased Relative Execution Time. The left graph shows that deployments described with less steps per module are higher affected by an increase in the Error Rate than the ones with a large number of levels per modules. More levels lead to execution of more scripts, thus more scripts will probably fail. However, each failure costs less, since the repetition of one script will traverse a smaller portion of the graph

and eventually, as seen from the comparison with the optimal case, the overall overhead inserted by the re execution declines with the number of levels. The deployments depicted in this figure consist of 20 modules. On the contrary, applications that consist of many modules are affected more by an increase in the Error Rate (on the rightmost Figure). The five modules case is deployed 1.5 slower than the optimal case (when the error rate is 0.2), while simultaneously, the 20 modules case is deployed 1.7 times slower. This means that when the number of modules quadruples, the increase in Relative Execution Time is less than 15%, making our methodology suitable for complex applications, consisting of multiple modules and many synchronization levels per module. Comparing with the traditional deployment tools, where no error recovery mechanisms are employed, it is obvious that the termination of the deployment occurs orders of magnitude faster, since the probability for a successful deployment is $(1 - ER)^{(M \cdot N)}$, where ER is the Error Rate, M equals the number of application modules and N is the number of deployment levels for each module. This quantity grows exponentially with the application's complexity making these tools unsuitable for complex applications deployed in unpredictable environments.

6 Related work

Automated cloud application deployment and configuration is a widespread problem, evolving with the expansion of cloud computing. Various systems, both in industry and academia, have been proposed in order to handle application deployment and scaling. In the Openstack ecosystem, Heat [13] and Sahara [14] have been proposed. Heat is a generic deployment tool used to define and configure deployment *stacks* that involve different resources (e.g., VMs, volumes, networks, IP addresses, etc.) and target to automate their management and orchestration. Heat retains a simple deployment model where it assumes that any script-independent information is available to each VM prior to the script execution and it does not support cyclical dependencies between different application modules. It also retains autoscaling capabilities, by monitoring the allocated resources and performing simple rule based actions in order to scale the running applications. Sahara (the ex Savannah project) is targeted to the deployment and configuration of Data Processing systems, such as Hadoop and Spark. Both tools integrate solely with Openstack and do not provide error recovery features.

The AWS counterparts of these tools are [3] and [4]. CloudFormation is a generic tool, relevant to Openstack Heat and it retains a similar deployment model where the user defines the different resources that should be allocated along with configuration scripts that are responsible for the software configuration. Elastic BeanStalk is primarily used for deploying specific applications (e.g., Web Servers, Load Balancers, etc.) into the Amazon cloud and also provide elastic capabilities. Finally, Canonical Juju [12] is another system used for deployment and scaling. Different modules are organized as *charms*. A charm is, essentially, a set of configuration files that determine the way through which a software module is configured and deployed and how it interacts with other

modules. The user can choose from a set of preconfigured charms and compose complex applications structures. All the aforementioned systems target to provide a level of automation through which a user can easily deploy and configure their application. However, the dynamic cloud nature is not taken into consideration since in case of failure the deployment is aborted and the user must manually resume it or cancel it. On the contrary, our approach allows the re-execution of failed scripts so as to automatically resume the deployment.

Similar to the previous systems, there exist standalone tools specializing in creating an identical application environment, mainly for development reasons. The most representative is [18]. It cooperates with popular configuration management tools, such as [8], [16] and [1] and creates an identical virtualized application environment, supporting multiple hypervisors and cloud providers. Runtime orchestration and scaling are not considered, though. Furthermore, in [21] *Wrangler* is proposed, a system that bases its functionality in script executions (called plugins). The philosophy of this system is similar to our approach; However error recovery is not considered. In [23], a data-centric approach is presented formulating resources as transactions, exploiting their ACID properties for error recovery. However, each action should have an "undo" action in order to participate in a transaction, which is a stronger hypothesis than the idempotent property enforced by the layered filesystems proposed in this work. In [22] a synchronization framework for Chef [8] is introduced: The user can introduced "breakpoints" into the execution of Chef recipes so as to synchronize the application configuration. This work identifies the need for synchronization when a cloud application is deployed; However, it does not handle transient deployment failures. Finally, in [24] an approach through which the cloud APIs are overridden in order to obtain knowledge about the status of their commands. This work is not focused on the application configuration as the current work is; However, it identifies the error-prone nature of the synchronous cloud providers and suggests a solution for transforming the unreliable cloud APIs into reliable calls with predictable outputs.

7 Conclusions

In this paper, we proposed a cloud deployment and configuration methodology, capable to overcome transient cloud failures through re-executing the failed deployment scripts. Our methodology resolves the dependencies among different software modules and identifies the scripts that should be re-executed so as to resume the application deployment. To guarantee that each script will have the same effect, we utilize a layered filesystem architecture through which we snapshot the filesystem before the script execution and, in case of failure, revert it to the previous state and retry. Our evaluation indicates that our methodology is effective even for the most unstable cloud environments and it is particularly effective for application that consist of many different modules exchanging multiple messages throughout the process.

References

1. Ansible. http://www.ansible.com/home
2. AUFS. http://aufs.sourceforge.net/
3. AWS CloudFormation. http://aws.amazon.com/cloudformation/
4. AWS Elastic BeanStalk. http://aws.amazon.com/elasticbeanstalk/
5. AWS Incident. https://goo.gl/f959fl
6. AWS Instances Boot Times. http://goo.gl/NQ1qNw
7. AWS Maintenance. https://aws.amazon.com/maintenance-help/
8. Chef. https://www.chef.io/chef/
9. Docker Container. https://www.docker.com/
10. Docker: Select a storage driver. https://goo.gl/o383To
11. Google App Engine Incident. https://goo.gl/ICI0Mo
12. Juju. https://juju.ubuntu.com/
13. Openstack Heat. https://wiki.openstack.org/wiki/Heat
14. Openstack Sahara. https://wiki.openstack.org/wiki/Sahara
15. Overlay Filesystem. https://goo.gl/y0H76w
16. Puppet. http://puppetlabs.com/
17. Rackspace SLAs. https://www.rackspace.com/information/legal/cloud/sla
18. Vagrant. https://www.vagrantup.com/
19. VMware vCloud Automation Center Documentation Center. http://goo.gl/YkKNic
20. Jennings, B., Stadler, R.: Resource management in clouds: Survey and research challenges. Journal of Network and Systems Management 23(3), 567–619 (2015)
21. Juve, G., Deelman, E.: Automating Application Deployment in Infrastructure Clouds. In: Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on. pp. 658–665. IEEE (2011)
22. Katsuno, Y., Takahashi, H.: An Automated Parallel Approach for Rapid Deployment of Composite Application Servers. In: Cloud Engineering (IC2E), 2015 IEEE International Conference on. pp. 126–134. IEEE (2015)
23. Liu, C., Mao, Y., Van der Merwe, J., Fernandez, M.: Cloud Resource Orchestration: A Data-Centric Approach. In: Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR). pp. 1–8 (2011)
24. Lu, Q., Zhu, L., Xu, X., Bass, L., Li, S., Zhang, W., Wang, N.: Mechanisms and architectures for tail-tolerant system operations in cloud. In: 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14) (2014)
25. Mell, P., Grance, T.: The NIST Definition of Cloud Computing (2011)
26. Potharaju, R., Jain, N.: When the network crumbles: An empirical study of cloud network failures and their impact on services. In: Proceedings of the 4th annual Symposium on Cloud Computing. p. 15. ACM (2013)
27. Rodeh, O., Bacik, J., Mason, C.: Btrfs: The linux b-tree filesystem. ACM Transactions on Storage (TOS) 9(3), 9 (2013)
28. Tsoumakos, D., Konstantinou, I., Boumpouka, C., Sioutas, S., Koziris, N.: Automated, elastic resource provisioning for nosql clusters using tiramola. In: Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on. pp. 34–41. IEEE (2013)