AURA: Recovering from Transient Failures in Cloud Deployments

Ioannis Giannakopoulos*, Ioannis Konstantinou*, Dimitrios Tsoumakos§ and Nectarios Koziris*
* *Computing Systems Laboratory, School of ECE, NTUA, Athens, Greece*
{*ggian, ikons, nkoziris*}*@cslab.ece.ntua.gr*
§ *Department of Informatics, Ionian University, Corfu, Greece*
dtsouma@ionio.gr

***Abstract*—In this work, we propose AURA, a cloud deployment tool used to deploy applications over providers that tend to present transient failures. The complexity of modern cloud environments imparts an error-prone behavior during the deployment phase of an application, something that hinders automation and magnifies costs both in terms of time and money. To overcome this challenge, we propose AURA, a framework that formulates an application deployment as a Directed Acyclic Graph traversal and re-executes the parts of the graph that failed. AURA achieves to execute any deployment script that updates filesystem related resources in an idempotent manner through the adoption of a layered filesystem technique. In our demonstration, we allow users to describe, deploy and monitor applications through a comprehensive UI and showcase AURA's ability to overcome transient failures, even in the most unstable environments.**

I. Introduction

The advent of the cloud computing era generated new perspectives regarding the deployment of applications. The virtualized nature of the compute and storage resources, allocated in an entirely dynamic, pay-as-you-go manner, enables the cloud users to call services, exported as a set of APIs, to allocate Virtual Machines and storage devices and deploy their applications on top of them. The concept of programmatic management of resources allows for the automation of complex deployment tasks that entail resource initialization (e.g., formatting virtualized block devices), execution of the necessary installation and configuration scripts (e.g., software dependencies installation and configuration files editing), etc. Simultaneously, the ever increasing complexity of applications architectures imposes the utilization of a synchronization mechanism among different resources so as to guarantee that the deployment tasks happen in a particular order (without hindering parallelism) and, upon their execution, leave the newly deployed application in a functional and consistent state.

For the execution of the aforementioned deployment tasks, several systems have been proposed. Be it a component of the cloud ecosystem itself, such as Openstack Heat [1] for Openstack, AWS CloudFormation [2] for Amazon, etc., or an external software module that communicates with the provider as a client, such as Vagrant [3], Juju [4], etc., these systems share a similar view of the deployment process as they express it as a Directed Acyclic Graph (DAG), the nodes of which represent the deployment states of the application modules and the edges represent the deployment scripts. The DAG itself expresses the order of execution and the dependencies among different states. Based on this model, the aforementioned systems first communicate with the cloud provider in order to allocate the necessary resources, traverse the deployment DAG and execute the deployment scripts.

Albeit the correctness of the deployment is guaranteed if the scripts terminate successfully, the aforementioned systems do not take into consideration the unstable and error-prone nature of the cloud [5]. The complexity introduced by the virtualization and cloud software, often leads to *transient* failures that appear for a short period of time and then vanish: Network glitches, temporal unavailability of network services such as DNS, read-only file systems attributed to random errors in the storage backends, etc. are some of those. Although most of these failures are harmless since they instantly disappear, they are capable of leading a script execution into failure and, hence, fail an entire application deployment. For example, assume that a deployment script installs specific packages through a package management system (such as apt) when a network glitch occurs; The script will fail since the packages will not be retrieved and, as a consequence, the entire deployment will also fail. Such failures can prove expensive, both in terms of budget and time. Taking into consideration that a deployment failure may also leave stale resources (e.g., VMs that were successfully provisioned) that require manual handling, it is apparent that the management of transient failure can prove extremely beneficial for the deployment process.

In order to tackle the aforementioned limitation of the existing deployment solutions, we provide *AURA*[1] a system used to perform cloud deployments, attempting to overcome transient failures through re-executing the failed scripts with the incentive that when the script is re-executed, the failure will have vanished. AURA is the implementation of the methodology described in our previous work [6] and it is

[1]According to Greek mythology, Aura was the goddess of breeze, commonly found in cloudy environments.

responsible to: (a) traverse the deployment DAG and execute the deployment scripts, (b) monitor the deployment and isolate the scripts that need to be re-executed if an error occurs and (c) guarantee that the deployment scripts will always have the same effects when executed multiple times. The property of having the exact same effect through re-executions makes a script idempotent and this is an essential precondition that needs to be fulfilled before attempting to re-execute it; If the script is not idempotent, a second execution may not lead to the same results as the first, possibly failed, execution. For example, if a network glitch appears during the installation of software packages (through apt), the package management system may end up in an inconsistent state if the configuration of a package is interrupted and, hence, manual handling may be required (e.g., forcefully reconfigure the appropriate packages). AURA guarantees that any script that affects file system related resources (e.g., files, databases, etc.) are idempotent through an elegant file system layered mechanism, also adopted and extensively used by the Docker [7] community.

The concrete contribution of this work is twofold. Firstly, we present AURA, an open source cloud deployment system which is capable of deploying applications in cloud environments, identifying transient failures and re-executing part of the deployment scenario in order to overcome the errors. Secondly, we present an indicative demonstration of our system, using two widely popular real-world applications: (a) RUBiS [8], which is a three-tier Web Application that emulates the functionality of auction sites like eBay and (b) Hadoop, which is the base platform for processing and executing analytic tasks over Big Data. These two applications, formed using fundamentally different architectures, are chosen to depict AURA's applicability to deploy both layered applications and cluster-based, distributed applications. Apart from these applications, the users will be able to utilize AURA's user-friendly UI to describe and deploy new applications according to their custom deployment scripts.

II. Related Work

The problem of application deployment in a cloud environment resembles the well researched configuration management problem [9]. The added property in the cloud field is that any software configuration must take place upon the successful resource allocation from the cloud provider and the configuration process is not limited to a single execution environment (e.g., a single VM) but the necessity for coordination spans to multiple hosts. From this viewpoint, we can categorize the deployment tools in two broad categories: (a) software based configuration management tools and (b) cloud based deployment tools. The former category comprises of tools that express the deployment DAG as a set of recipes, written either in a procedural or declarative format. Puppet [10] is one of the most popular tools of this category. It is based on a set of recipes which are published to a Puppet Server. New instances fetch the appropriate recipes from the server and execute them accordingly. Chef [11] is a similar tool, that adopts a declarative language; The user does not describe what needs to be done, but describes a desirable state that the application must reach. Finally, Ansible [12] is a configuration tool that follows the exact opposite route: A user dictate commands which are spawned in parallel to the Ansible cluster. All these tools do not handle resource allocation, as they operate on top of previously allocated resources or on bare metal environments.

The later category consists of systems that operate in parallel with a cloud system, allocate and manage virtualized resources and then traverse the deployment graphs. Openstack Heat [1] is responsible for the deployment of applications in the Openstack cloud. The users submit application description in a YAML format, describing an application blueprint that receives a number of parameters (e.g., VM flavors, keypair names, etc.). Upon the definition of those parameters, the user can deploy their application in the cloud. Vagrant [3] and Juju [4] are two popular deployment tools with different user bases: The former is mainly used by developers who want to generate reproducible environments in different providers, while the later is mainly used by Ubuntu users that want to deploy software packages, the so called *charms*. Wrangler [13] is a similar system, that utilizes *plugins* in order to deploy the necessary components, whereas [14] is an interesting parallel approach that bases its functionality on Chef: Multiple configuration commands are spawned in parallel in order to speedup the configuration process. In this work, the necessity of coordination between the parallel workers is also identified and tackled. From the error recovery viewpoint, in [15], a data-centric approach is presented. The authors formulate the application deployment as a set of database transactions and issue *undo* scripts for failed script executions, something that may not always be possible, since not all actions are undoable. Finally, [16] focuses on analyzing Puppet scripts in order to identify any idempotence and convergence related issues.

III. Methodology and System overview

A. Deployment Model

AURA utilizes the deployment model presented at [6]. Assume an application that consists of different software modules; The term software module refers to any software component (package, daemon, etc.) that can be deployed with a set of deployment scripts. In Figure 1, we depict the deployment timeline of an application that consists of three modules. We assume that during the deployment, each module executes its deployment scripts (denoted with the dotted vertical line) and can send or receive messages to other software modules (denoted with the black horizontal arrows). These messages may contain useful information for the other modules (e.g., IP addresses, network ports, etc.) or they can be ACK messages that a deployment script

has been executed successfully. When a module waits for a message (e.g., points *A'*, *B'*), it blocks until the message is received. On the contrary, when a message is sent by a module, the sender posts it into a queue and proceeds with the next installation script, e.g., upon sending a message in point *A*, module *(1)* proceeds with the script execution between points *A*-*C'*.

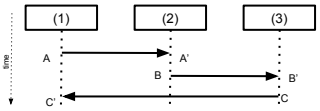

Figure 1. Application Deployment Timeline

As the deployment evolves, the blocking mechanism when waiting for a message achieves the essential coordination between different software modules. With this simple and elegant mechanism, a module can wait until another module has reached a specific point, something that ensures that the deployment script will be serialized when necessary, and scripts will be executed in a particular order. For example, a Web Server will start only when the Database Server is successfully configured. As described in the following section, this blocking scheme is also crucial for the error recovery mechanism.

B. Error Recovery and Idempotence

Error recovery entails three discrete but tightly coupled operations: (a) error identification, (b) isolation of the part of the deployment scenario that needs to be repeated and (c) execution of the scenario. When a transient error occurs, AURA must first identify that an error occurred. When a module remains blocked (waiting for a message) for more than a pre-defined waiting threshold, it spawns a new "error-alert" to AURA, translating this absence of message as a possible failure of a deployment script of the sender module. At this point the error recovery mechanism is launched, that determines whether an error did occur, or if the sender module needs more time to finish its execution. To facilitate the error recovery process, it is convenient to express the deployment timeline presented at Figure 1 as a DAG, as depicted in Figure 2 (a). Again, the vertical edges represent script executions and the horizontal edges represent messages between different modules. The nodes of the DAG represent the states that the modules reach after the successful execution of the deployment scripts. Furthermore, one may notice that when all the deployment scripts are successfully executed, each module broadcasts a message to the rest of the modules, informing them about its ready state.

The direction of the edges of the deployment graph demonstrated in Figure 2 (a) indicates the order of the operations (scripts executions and message flows). If we reverse the directions of the edges, we obtain the graph of Figure 2 (b), that represents the dependencies between the states of the deployment graph, called the Dependency Graph of the deployment. If this graph is traversed with any topological ordering algorithm, starting from any of its nodes (say u), the outcome would inform us about the states that must be reached before u is reached. When an "error-alert" is received by AURA, it traverses the Dependency Graph in a breadth-first manner, starting from the node in which a module was blocked and scans for the health of the neighbor nodes. For example, if an "error-alert" message was issued by *(1)* in *C'*, AURA will first evaluate whether *C* state has been reached. If *C* was not reached, then AURA will continue with *B'*, *B*, etc. until a set of healthy nodes frontier is found. When such a frontier is identified, the essential scripts are re-executed and the respective messages are re-sent.

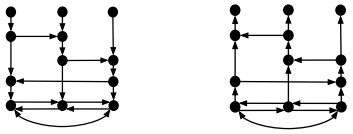

Figure 2. Deployment and Dependency Graphs

One hypothesis implied by our analysis so far, is that script re-executions will always have the same impact. However, this may not always be the case; For example, take the example of a deployment script that reads from a file and deletes it from the filesystem. This script can only be executed once; During a second execution, the script will fail since it will try to read from a non-existent file. To handle these circumstances, AURA utilizes a layered filesystem architecture in order to nullify any modifications made by a failed script. Specifically, prior to a script execution, AURA mounts an AUFS [17] layer, on top of the directories that the script will modify (`/var/lib`, `/etc`, `/opt`, etc). If the script successfully terminates, AURA will append a new AUFS layer on top of that layer and it will continue with the next deployment script. If the first script fails, AURA will remove the AUFS layer and it will replace it with a new empty layer, starting from the beginning. This mechanism, already used with success by the Docker [7] community for image distribution, guarantees that any script that modifies filesystem related resources remains idempotent.

C. System architecture

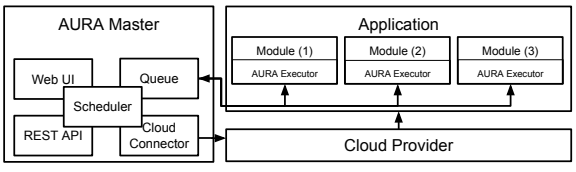

Figure 3. AURA Architecture

In Figure 3 we provide AURA's architecture. The left part of the Figure represents the AURA Master, which is deployed in a dedicated VM into the cloud. The right part of the Figure consists of two components: the Application and the Cloud Provider components. The Application consists of multiple modules; For convenience, we assume that

each module is deployed in a dedicated VM. The Cloud Provider represents the provider endpoint, that manages the virtualized resources. AURA consists of two components: the AURA Master and the AURA Executors. The AURA Executors are responsible to execute the appropriate deployment scripts for the module they are deployed to, setting up the AUFS filesystem layers prior to the script execution. Each software module is deployed by a dedicated AURA Executor. On the other hand, the Master operates as an endpoint; It exposes a REST API and a user-friendly UI through which the user can describe new applications, issue new deployment requests, obtain real-time monitoring statistics, etc. The Master contacts the cloud provider through the Cloud Connector, which is responsible to translate the high level resource allocation commands (e.g., create new VMs, etc.) to IaaS specific commands. The Queue submodule represents the communication channel between the Executors, as any message from the modules is posted into it. The Executors subscribe to specific queues and consume messages, as these arrive from the other modules. Finally, the Scheduler is responsible for monitoring the entire deployment process and, in case of error, intervene, traverse the Dependency Graph and require the re-execution of specific scripts from the respective Executors.

IV. Demonstration

To demonstrate AURA's functionality, we chose an indicative cloud application for deployment: a Hadoop Cluster, which is the state-of-the-art Big Data processing and analytics platform. Hadoop is a typical example of a distributed application that consists of a master node and multiple slave nodes. During the demonstration, AURA's user will be able to browse the application description and trigger new deployment requests. AURA will then allocate new VMs from the Openstack cluster and deploy the AURA Executors accordingly. Each Executor will run the necessary deployment scripts and the user will be able to monitor the deployment status through a real-time monitoring UI. Specifically, the user will be able to view each script's logs, a real-time view of the Deployment Graph, the status of the respective script execution and the number of AUFS layers that are deployed to each software component. A screenshot of AURA's UI is demonstrated in Figure 4, in which the Deployment Graph is depicted: Blue edges represent running scripts, green edges represent completed scripts, red edges represent failed executions and gray edges represent pending scripts. To test AURA in unstable and error-prone environments, the user will be able to assign a "script failure probability" that denotes the probability of a script to fail. If the probability is high, the scripts will be killed in a random point throughout their execution.

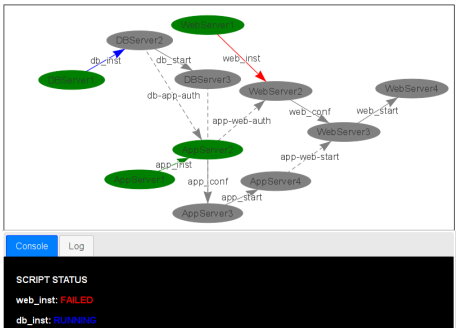

Figure 4. Deployment Graph as depicted by AURA

V. Conclusions

In this work we presented AURA, a cloud deployment system used to deploy applications over providers that present transient failures (e.g., network glitches, temporal storage medium errors, etc.). Our demonstration is indicative of AURA's ability to handle any application description, deployed over the most error-prone infrastructures with a minimum number of script re-executions.

Acknowledgment

This work was supported by the TREDISEC project (G.A. no 644412), funded by the European Union (EU) under the Information and Communication Technologies (ICT) theme of the Horizon 2020 (H2020) research and innovation programme.

References

[1] "Openstack Heat," https://wiki.openstack.org/wiki/Heat.
[2] "AWS CloudFormation," http://aws.amazon.com/cloudformation/.
[3] "Vagrant," https://www.vagrantup.com/.
[4] "Juju," https://juju.ubuntu.com/.
[5] "Google App Engine Incident," https://goo.gl/ICI0Mo.
[6] I. Giannakopoulos, I. Konstantinou, D. Tsoumakos, and N. Koziris, "Recovering from Cloud Application Deployment Failures through Re-execution," in *Algorithmic Aspects of Cloud Computing*. Springer, 2016.
[7] "Docker Container," https://www.docker.com/.
[8] "RUBiS," http://rubis.ow2.org/.
[9] D. B. Leblang, "The CM challenge: Configuration management that works," in *Configuration management*. John Wiley & Sons, Inc., 1995, pp. 1–37.
[10] "Puppet," http://puppetlabs.com/.
[11] "Chef," https://www.chef.io/chef/.
[12] "Ansible," http://www.ansible.com/home.
[13] G. Juve and E. Deelman, "Automating Application Deployment in Infrastructure Clouds," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. IEEE, 2011, pp. 658–665.
[14] Y. Katsuno and H. Takahashi, "An Automated Parallel Approach for Rapid Deployment of Composite Application Servers," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 126–134.
[15] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernandez, "Cloud Resource Orchestration: A Data-Centric Approach," in *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, 2011, pp. 1–8.
[16] O. Hanappi, W. Hummer, and S. Dustdar, "Asserting reliable convergence for configuration management scripts," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016, pp. 328–343.
[17] "AUFS," http://aufs.sourceforge.net/.