Provenance Issues in Platform-as-a-Service Model of Cloud Computing

Devdatta Kulkarni
devdattakulkarni@gmail.com
Rackspace, The University of Texas at Austin

Abstract

In this paper we present provenance issues that arise in building Platform-as-a-Service (PaaS) model of cloud computing. The issues are related to designing, building, and deploying of the platform itself, and those related to building and deploying applications on the platform. These include, tracking of commands for successful software installations, tracking of inter-service dependencies, tracking of configuration parameters for different services, and tracking of application related artifacts. We identify the provenance information to address these issues and propose mechanisms to track it.

1 Introduction

Platform-as-a-Service (PaaS) model of cloud computing simplifies deployment of applications to a cloud infrastructure. Application developers use a declarative model for specifying their application's source code and requirements to the platform. The platform builds and deploys the application by provisioning the required infrastructure resources. There are several such PaaS systems in existence today, such as Heroku [1], Google App Engine [2], OpenShift [3], Solum [4].

Provenance has emerged as one of the key requirement in building modern systems that support properties such as auditability and data lineage tracking [5]. In our view *provenance* is all the information about an entity that can help towards understanding how that entity got to a particular state. For instance, when a software is successfully installed on a Ubuntu host, all the packages that had to be installed towards that are part of that software's provenance. Another example is that of a service which depends on other services for its correct functioning. The dependency information, such as the versions of the dependent services, are part of that service's provenance.

In a PaaS system, the need to track provenance information is especially important given that the platform manages complete life-cycle of an application. In order for application developers to gain confidence into the platform and to gain insights into application construction process, it is important that the platform collects different kinds of provenance information and makes it available to application developers. Additionally, provenance information about the platform itself is useful for the platform developers and platform operators towards correct design and operation of the platform.

In this paper we identify different kinds of provenance information that need to be collected within a PaaS to provide better insights into the workings of the platform and the applications that it deploys (Section 2). We also propose mechanisms to collect this information (Section 3). We realized the importance of some of these through our experience of building Solum, which is an application life-cycle management system for OpenStack [6]. Application developers use declarative specification model to specify application's build and runtime requirements to Solum. Solum builds and deploys the application by creating *Docker containers* [7] with the application code, hence forth referred as *docker containers*, and deploying them on OpenStack infrastructure.

2 Provenance Issues

2.1 Platform development

While developing Solum, we frequently followed a pattern of trial and exploration when it came to using new softwares and tools, such as the *docker_registry* [8], which we use for storing application's docker containers. The typical pattern that we followed was as follows. We referred to several on-line documents, we installed different softwares and packages, changed file permissions if required, changed directories, compiled/built certain packages, and so on until we were successful.

At the end when the required software/package was installed successfully, we wanted to consolidate the ex-

act steps that we could follow if we had to install that software on a different host in the future. However, at this point we generally faced a problem. From the entire shell command history, it was difficult to find out only those commands which were essential towards successful installation of that software. We felt that we needed an automated way which, given a list of commands, will find out the exact commands that are necessary and sufficient for installing that software. Essentially, we needed the provenance for software installs (subsection 3.1).

We observe that the trial and exploration pattern is not unique to design and development of Solum. It is quite general and can be seen to be followed by developers in trying to install any new software.

2.2 Platform building

Solum depends on other OpenStack services such as Keystone [9], Glance [10], Heat [11], Barbican [12], Tempest [13]. Often times these services change their requirements, or undergo a refactoring change, which causes Solum to break. For example, recently the tempest functional testing library re-factored its code to move its *REST* module from its package to a separate library. This caused all the functional tests in Solum to stop working as the import path of the *REST* module within Solum's functional tests was no longer correct. Such a breakage results in wasted developer productivity. It also leads to wasted resources in terms of testing failures and backups on the OpenStack's continuous integration servers (*CI servers*). The problem is exacerbated if a service has had several commits. For developers of a service like Solum, it is currently very tedious to find out which commit in the dependent service broke their build. We felt that the severity of this issue could be reduced if could have information about the last commit of the dependent service that worked for Solum. Then the problem would be constrained to finding the culprit commit between the last known working commit and the latest commit. Essentially, we needed to maintain provenance information related to inter-service dependencies (subsection 3.2).

2.3 Platform deployment

Each OpenStack service has large number of configuration parameters. It is hard for developers of a service like Solum, which depends on other services, to know the parameters from dependent services that are critical to the operation of their service. We have faced this issue several times in the *devstack* [14] setup of OpenStack, which supports running all the OpenStack services within a single VM, typically for the purpose of design and development. As an example of how configuration parameters for other services matter, for Solum to function correctly in the devstack environment, following variables need to be set in Nova's [15] configuration file: *scheduler_default_filter*, *memory*, *compute_driver*. We propose in subsection 3.3 how we can use provenance of configuration parameters towards enabling choosing of appropriate configuration parameters of dependent services for the purpose of design and development of Solum.

2.4 Application provenance

Within Solum, application construction and deployment information includes, commit hash of application's source code deployed by the platform, test and run commands used in building particular version of the application, version of docker used to build the application runtime environment, and versions of dependent services used. All this information is part of the application construction and deployment provenance.

3 Provenance Mechanisms

3.1 Command list provenance

In order to find the provenance for successfully installing a software we need to consider the list of commands that eventually led to a successful installation. An entire history of user's shell interactions contains different kinds of commands, such as package installation commands (apt-get install), navigational commands (cd, pushd, popd), listing/viewing commands (ls, less, more), editor commands (opening files in vi, emacs), service start up commands (service docker start), and so on. Below we propose a method which, given a list of commands and the name of the software to install, prunes the command list to the essential commands required for installing the software.

The main issues that need to be addressed in designing this method are: (a) detecting where to start and stop in user's command history, and (b) determining whether the candidate list of commands is complete. To address the issue of where to start and stop in a long execution history we define the notion of an *anchor point (ap)* command. Example of an anchor point command on Ubuntu is *apt-get update*. The reason this command can be treated as an anchor point is because in our experience, updating packages on the host is a typical usage pattern when trying to install new software on Ubuntu hosts. The anchor point commands may also be specified by users. Moreover, instead of considering the entire command history to build the candidate list, we can use the last command that led to successfully starting up of the service as another anchor point. Lets call this as the *last successful command (lsc)*. To address the issue of

how to find out whether the candidate list of commands is complete, we propose that we need to try out that command list in an automated manner in a contained environment. Our idea for this is to build a docker container consisting of the list of candidate commands and check it with a verification script provided by the application developer which checks whether that software was correctly installed on the container or not. These ideas are captured in the following algorithm:
Input: <Command list, software-name, verification script>
Output: Pruned command list
Steps:
1) Start with the last successful command (lsc) from the command history.
2) Work backwards in the command history to collect all the commands except the listing commands. This can be done till first anchor point (ap).
3) Build a *Dockerfile* [16] consisting of all the commands from step 2. Add the verification script and set it to be run as the entry point of the container.
4) Build a docker container from this Dockerfile and run it. The docker run command will exit successfully only if the verification script exits without an error (this is by design of the verification script). The list of commands used to build the container is the required provenance list for that software.
5) If on the other hand, docker run has exited with an error then go back to step 2 till next anchor point in the command history, and repeat from steps 3 onwards with the new list of commands.

The verification script defines checks to verify that the software was correctly installed. For instance, when installing *docker*, the *docker -v* command executes successfully only if docker is correctly installed. So in this case the verification script could be to execute this command and check its return status [17]. On the other hand, if we are trying to install *tomcat*, then one way to verify that the installation was successful is to check if the *webapps* directory, which tomcat uses to run web applications, has been created at a well-known location [17]. The idea of using a verification script is inspired from Solum's custom language packs feature. In section 4 we present initial evaluation of this approach.

3.2 Service dependency tracking

As mentioned in subsection 2.2, Solum may break due to changes in dependent services. Presently we realize this when the outstanding Solum patches fail the continuous integration tests in OpenStack's CI servers. We resolve this situation by going through the commit history of the dependent service, finding the commit which is causing Solum to break, and pinning Solum to use a commit before this commit. Below we propose two approaches that can help with this process.

3.2.1 Dependent services' commits tracking

One way to find the commit(s) to pin to is to maintain *provenance information* about Solum that includes the most recent *git commits* of all the services that Solum depends on. For instance, we could maintain the following tuple, which we call as the *successful_commits_tuple*, *<Keystone=a, Tempest=1, Glance=A>* which identifies that Solum's build is known to be working correctly with *commit a* of Keystone, *commit 1* of Tempest, and *commit A* of Glance . Whenever Solum's CI server runs a Solum patch successfully, we could update this tuple to include the latest commit ids for each of the dependent services. Now consider the situation when the current commits for the dependent services are *<Keystone=c, Tempest=3, Glance=B>*, with the assumption that the commit ids increase according to the natural definition of the range from which they are picked for this example (so for instance the commits for Keystone are - a, b, c, d, and so on). Suppose that this set of commits is causing Solum to break. Then the question is how to efficiently find the next *successful_commits_tuple*.

One approach to find this is to sequentially try each previous commit of each of the dependent services to find out if that commit can be included in the *successful_commits_tuple*. If a specific commit of a service cannot be included, we remove it from consideration. When we do so, we need to also remove from consideration the commits of other services that may depend on the commit being removed. For example, Glance's *commit B* may be dependent on Keystone's *commit c*, and Tempest's *commit 3* may be dependent on Keystone's *commit b*. So if we remove Keystone's *commit c* from consideration then we need to also remove Glance's *commit B* from consideration. So the next *candidate_commits_tuple* we can try is *<Keystone=b, Tempest=3, Glance=A>*. Observe that the interservice dependency information is a directed graph with each commit representing a node and a directed edge between two nodes if the commit represented by the source node *depends on* the target. When we remove a commit from consideration, we need to remove all the commits from dependent services that are part of the *transitive closure* involving the commit being removed on the graph with direction of the edges reversed.

The order of choosing a service to try may matter in this approach. For instance, if a service *s1* is picked as the first service to try, and if its last commit *l* is required by a very old commit *o* of another service *s2*, then removing *l* from consideration would lead to removal of all the commits of *s2* that were made following and in-

cluding *o*. This might be incorrect as the commit which might be really causing the problem could be a relatively newer commit *n* of *s2*. If we pick *s2* instead of *s1* as the first service to try, we could find this. It remains to be evaluated what would be a good strategy to select the order of considering different services.

3.2.2 Dependent services' merge tracking

Another approach for determining commit(s) to pin to is as follows. In OpenStack, there is a system named *Zuul* [18], which orchestrates execution of continuous integration runs for all the OpenStack projects. It maintains a yaml based project definition for each project. We propose that this project definition be enhanced by including the list of projects that depend on that project, and a list of *trigger events*. For example, below we show how the project definition for *Barbican* might look like with this enhancement. The *Triggers* attribute identifies the list of triggering events and the *Used_by* attribute identifies the list of projects that depend on *Barbican*.

```
Barbican:
  Triggers:
    OnMergeToMaster
  Used_by:
    Solum, Murano, Mistral
```

Whenever a patch is merged into the master branch of *Barbican*, the *OnMergeToMaster* event will be generated. This event will cause a "recheck no bug" comment to be added to the *Used_by* project's outstanding patches. This comment has a special meaning within Zuul. It causes Zuul to trigger a CI run on those patches. If the patch passes then we know that the change to *Barbican* is non-breaking. However, if the patch fails, then we would have proactively caught the breaking change.

In this approach how to manage updates to the dependent service's project configuration can become an issue. Specifically, we cannot expect the dependent services' developers to know about all the other projects that are using it. We propose that the team members of the project that depends on another project propose a patch to modify the dependent service's project configuration in Zuul. The dependent service's team members need to approve this patch before it can be merged, thus changing the dependent service's project configuration.

Some of the outstanding questions in this approach are: what modifications are required to Zuul to enable event generation? if there are no outstanding patches, could we make *Zuul* to generate a new patch and submit it to the *Used_by* projects review queue and trigger a CI run it? what will be the nature of such a patch?

3.3 Infrastructure configuration tracking

Towards maintaining provenance information of configuration parameters of dependent services, we propose to version control the configuration parameters for different OpenStack services that led to successful deployment of Solum. For instance, we would version control entries in nova.conf, heat.conf, tempest.conf corresponding to successful deployments of Solum. Then, when a bug such as [19] is encountered, which stems from not having enough memory on the host, and whose resolution is to set the value of *scheduler_default_filter* configuration parameter in nova.conf to *AllHostsFilter*, having this parameter and its value tracked in version control would help create provenance for successful future deployments of Solum.

3.4 Application provenance API

In subsection 2.4 we identified the provenance requirements from the point of view of an application. For application developers to be able to extract provenance information about their applications, we propose to define an object model and an API in Solum. Through this API it will be possible for application developers to extract information such as the git commit hash used to deploy their applications' code to different environments (testing, staging, production), the version of docker used to build the application's docker containers, and the versions of other OpenStack services that are being used.

4 Evaluation

We implemented the command line pruning algorithm and used it to find provenance of installing two different softwares (*docker* and *tomcat*). The implementation, the command histories that we used, and the verification scripts that we defined are available at [17]. Aided with some changes in the way we operated on the host, the algorithm was able to successfully generate provenance for both. We performed these experiments on a Ubuntu 14.04 host with docker version 1.6.

Here are our observations from this experimentation. When building a docker container with a subset of commands from the command history, docker build may fail for several reasons. Specifically, we observed that this may happen if, (a) the command is not installed on the container, or (b) the command is a non-existent command (a typo, or a valid command but not on the path, or a service name instead of a command). We experienced the first case when we included *curl* as part of the command list to be built into a docker container (*curl* was part of our execution history). The docker build failed in this case because *curl* was not installed on the current

container layer or on any of its parent layers. This led to the realization that a command that works on a host may not work on the container unless it is explicitly installed. This led to the conclusion that when including a command *c* to be tried within a docker build, we may have to first include another command *i_c* which installs *c* on the container. We experienced the second case when we included *tomcat* as part of the command list (executing the command *tomcat* was part of our execution history). The docker build failed in this case because *tomcat* is not a valid Ubuntu command. This led to the conclusion that before including a command into the Dockerfile, we should run it on the host and check if the output contains a well-known string, such as *command not found:*. If so, we should not include it in the Dockerfile.

Certain commands may need one or more flags to be tried on the command line. We observed this when installing docker. One of the commands to install docker is to *wget* the docker package. This command worked fine on our host but failed in docker build. Upon investigation we realized that we had to use the *–no-check-certificate* flag for the command to work successfully from within the container. The general case of finding out which flags are missing from a command's execution might be difficult to figure out. But if the command history contains commands with different flags that are clustered together, which may happen if the user is trying out different flags, then we can hope to find out the correct flags as follows. We could maintain a *similar_command_map* which maintains a map of <command, [similar command list]>. We try each command as part of the docker build and keep the one which works. We also observed that piped commands, such as *wget -qO- https://get.docker.com/ | sh* did not seem to work as expected when specified within the RUN command within a Dockerfile. To address this, we ended up breaking such a command into its constituent parts and executing each sub-command separately.

Some other observations and conclusions were as follows. The navigation commands (cd/pushd/popd) need to be combined with their immediate successor (for cd/pushd) and predecessor commands (for popd) to be included as part of a single docker RUN command. Otherwise they won't take effect. For commands that modify a file, we can add the modified file from the host into the container at the appropriate location. Another part of provenance is all the packages that were installed on the host when the software was successfully installed. On Ubuntu this information can be found using the *dpkg* command (*dpkg -l*).

More experimentation is needed to determine the efficacy of this approach. Some theoretically interesting questions in this regard are: whether the algorithm is sound? (only correct provenance is generated, if it exists), whether it is complete? (provenance is always generated if a software was installed successfully). As mentioned by one of the reviewer, the output of the algorithm can be considered to be an hint towards steps for successfully installing a software. Even this could be helpful in determining the steps for installing a software on different hosts.

5 Related Work

Provenance issues in cloud computing have been considered in [5, 20]. In [5] provenance properties are defined and protocols are presented that use cloud-based storage for storing the provenance information. In [20], the importance of linking provenance information across different layers of a cloud infrastructure is stressed. The provenance information related to inter-service dependencies identified here is similar to the provenance property of *multi-object causal ordering* defined in [5]. Similar to [20], we argue for collecting provenance information that will ease designing, building, and deploying PaaS systems. In [21] a methodology based on tracking system calls is presented to enable repeatable execution of commands on different target platforms. The command list provenance approach presented here complements that by addressing the issue of how to repeat *installation* of a command on different hosts. Checking validity of configuration parameters for dependent services is an important issue [22]. Collecting provenance information for such parameters, as presented here, is a first step towards enabling such checks.

6 Conclusion

In this paper we have identified provenance issues in designing, building, and deploying PaaS systems and the applications deployed by them. We have proposed mechanisms to address these issues through collection of provenance information. Our next step is to implement these mechanisms and perform detailed evaluation of their effectiveness.

7 Acknowledgments

The author would like to thank anonymous reviewers for providing helpful comments and feedback on the paper.

References

[1] https://www.heroku.com/.

[2] https://cloud.google.com/appengine/.

[3] `https://www.openshift.com/`.

[4] `https://github.com/stackforge/solum`.

[5] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 15–14, Berkeley, CA, USA, 2010. USENIX Association.

[6] http://www.openstack.org/.

[7] `https://www.docker.com/`.

[8] `https://github.com/docker/docker-registry`.

[9] `http://docs.openstack.org/developer/keystone/`.

[10] `http://docs.openstack.org/developer/glance/`.

[11] `http://docs.openstack.org/developer/heat/`.

[12] `https://wiki.openstack.org/wiki/Barbican`.

[13] `http://docs.openstack.org/developer/tempest/`.

[14] `http://docs.openstack.org/developer/devstack/`.

[15] `http://docs.openstack.org/developer/nova/`.

[16] `https://docs.docker.com/reference/builder/`.

[17] `https://github.com/devdattakulkarni/cmdlist-provenance`.

[18] `http://ci.openstack.org/zuul/`.

[19] `https://bugs.launchpad.net/solum/+bug/1390246`.

[20] Imad M. Abbadi and John Lyle. Challenges for provenance in cloud computing. In *3rd USENIX Workshop on the Theory and Practice of Provenance*, 2011.

[21] Philip J. Guo. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 25th International Conference on Large Installation System Administration*, LISA'11, Berkeley, CA, USA, 2011. USENIX Association.

[22] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. Confvalley: A systematic configuration validation framework for cloud services. In *Proceedings of the European Conference on Computer Systems (EuroSys'15)*, 2015.