Poster: Improving Cloud-based Continuous Integration Environments

Alessio Gambi, Zabolotnyi Rostyslav and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, Vienna, Austria
Email: name.surname@infosys.tuwien.ac.at

***Abstract*—We propose a novel technique for improving the efficiency of cloud-based continuous integration development environments. Our technique identifies repetitive, expensive and time-consuming setup activities that are required to run integration and system tests in the cloud, and consolidates them into preconfigured testing virtual machines such that the overall costs of test execution are minimized. We create such testing machines by reconfiguring and opportunistically snapshotting the virtual machines already registered in the cloud.**

I. Continuous Integration Environments in the Cloud

Continuous integration environments automatically execute integration and system tests on remote computing resources, and move the burden of test execution away from developers machines. Remote resources are shared among all the developers and might easily become a bottleneck when the number of developers increases, thus limiting the effectiveness of the continuous integration environments.

Some address this problem by defining techniques that filter and prioritize the tests scheduled for execution such that developers could get interesting results within acceptable time also in the presence of resource shortage [1]. Others instead leverage cloud platforms to access increasingly large and *elastic* pools of computing resources to reduce the risk of incurring in bottlenecks [2].

Current solutions mostly focus on improving the efficiency of cloud-based continuous integration by automating repetitive activities to setup and execute integration and system tests [3], which account for the coordinated deployment of several virtual machine instances and their configuration by installing software components, restoring system state, and configuring test drivers. As an example, system testing of a two-tiered Web service requires at least the deployment of two virtual machines, the installation of the application server code *and* the business logic on the 'front-end' server, and the installation of the database server *and* its content in the 'back-end' server.

This setup process repeats for all the instances that are started by the tests; therefore, automated solutions have the potential of speeding up the overall test execution. However, blind automation in the cloud might result in surprisingly high costs and long execution times that can easily overpass the potential benefits of automation and jeopardize the use of cloud-based continuous integration environments. In fact, cloud providers charge the usage of any resource, including network communications, and the set up of virtual machines might involve the download of large amount of data and the re-installation of software components whose costs accumulate over test executions.

We argue that consolidating repetitive setup actions into prepackaged testing virtual machines that better match the required testing environment might reduce the effort required to set up integration and system tests in the cloud, thus improving the overall efficiency of cloud-based continuous integration environments. Therefore, we propose to leverage snapshotting, a standard feature of cloud platforms, and opportunistically create the required testing virtual machines by executing partial updates of the virtual machines already registered in the cloud.

II. Opportunistic Snapshotting

To efficiently execute a given set of integration and system tests in the cloud, cloud-based continuous integration environments need to decide which virtual machines should be reused as-they-are and which ones should be updated. This requires to understand for each test the needed setup and the cost to implement it using available virtual machines, potential snapshots, or any combination of the two. In other words, an efficient test execution in the cloud needs to balance the trade-off between using the current virtual machines and incurring in variable costs to repeatedly set up them, and creating new virtual machines and incurring in fixed costs for the snapshotting.

We propose the following approach to address this problem: re-formulate the opportunistic snapshotting problem as an optimization problem using integer linear programming and find the optimal solution with standard techniques. In particular, we re-formulate our original problem as a variation of the well known *minimum cost flow problem*, where a given flow must be sent through a capacity constrained flow network in the cheapest possible way.

The intuition behind this choice is that we can represent instances of testing virtual machines as the flow, and build a flow network that reflects the status of the cloud, the requirements of the tests to be executed, and the act of performing various actions such as virtual machine setup, deployment, and snapshotting. Given such a network, we can associate a cost to each setup action and quantify the total cost of executing a given list of tests as the aggregated cost for sending the flow corresponding to the testing instances started by the tests through the network. Under constraints of flow conservation

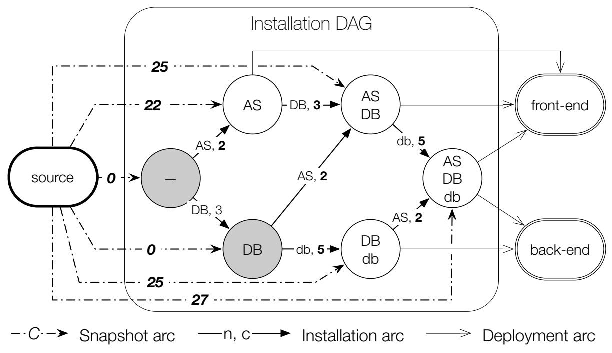

Fig. 1. Flow model for the Web service example

and balance we can find the distribution of the flow that has the minimum cost. We decide to create snapshots by observing the presence of the flow through the corresponding arcs in the network model.

We describe the construction of the flow model using the pedagogical example of a two-tiered system. Figure 1 depicts the resulting model. The system under test is composed of an application server (AS) acting as a front-end, and a database (DB) loaded with the test data (*db*) acting as a backend. The example assumes that the front-end and the backend always run on separate virtual machines.

We execute two types of test: `T1` that deploys each component once, akin to integration tests, and `T2` that deploys multiple times the front-end but only one time the backend, akin to scalability tests.

We start by extracting the list of required software components to install (AS, DB and *db*), the types of virtual machines to use (*front-end* and *backend*), and the number of virtual machine instances to deploy (1 front-end and 1 backend for `T1`, 3 front-ends and 1 backend for `T2`). Then, we build the *Installation DAG*: a directed acyclic graph that represents all the possible virtual machines that we can build using the required software components. Installation nodes represent configured virtual machines and are labeled with the names of installed components; arcs represent the act of installing new components and are labeled with component names and installation costs.

We augment the installation DAG by introducing an *instance node* for each type of virtual machine to be used in the tests and by connecting installation nodes to instance nodes by means of *deployment arcs*. In details, we introduce a new deployment arc whenever the set of components listed in the installation node is a superset of the components required by the instance node. In the example, we connect the nodes labelled $\{AS\}$, $\{AS, DB\}$ and $\{AS, DB, db\}$ to the front-end node, which requires AS, and the nodes labelled $\{DB, db\}$ and $\{AS, DB, db\}$ to the backend node, which requires both DB and *db*. We introduce a synthetic *source* node and link it to the installation nodes via *snapshot arcs* to account for the existing virtual machines, the potential snapshots that we can create, and the costs for creating the snapshots. Snapshot costs account for the variable costs for updating existing virtual machines ($\{-\}$ and $\{DB\}$ in the example) and the fixed cost for creating new snapshots (e.g., 20). In particular, only snapshot arcs that do not correspond to existing virtual machines are labeled with such costs.

We complete the formulation of the problem with the flow constraints. The source generates a flow that corresponds to the total number of the instances used by the tests (6 in the example); instance nodes absorb an amount of flow compatible with the deployment information (e.g., 4 units flow in the front-end and 2 in the backend); and, the flow is balanced.

By solving this optimization problem we identify the need for new snapshots and an optimal strategy to create them. In the example, if we execute two times `T1` and one time `T2`, our approach suggests to create no snapshots; however, if execute more times the tests (e.g., six times `T1` and four times `T2`), our approach suggests to create the $\{AS, DB, db\}$ virtual machine by updating $\{DB\}$.

III. Related Work and Outline

The definition of open standards and automated solutions for managing complex systems in the cloud enabled the creation of novel testing environments [3]. For example, Van der Burg and Dolstra [4] proposed a declarative approach for setting up complex applications during integration testing of complex systems in virtualized data centers, while Hanawa et al. [5] introduced D-Cloud for automated dependency analysis of distributed systems in the cloud. In this work we address a different problem, that is, opportunistically leveraging cloud-specific functionalities to improve the efficiency of cloud-based testing and propose a complementary solution.

Currently, we are focusing on refining the optimization problem to include setup times and additional setup actions, automatically constructing the flow model from design time artifacts such as test code and the history of commits, and developing a prototype tool to enable extensive evaluation of the proposed approach. Future work include extending the approach to deal with continuous *on-line* test executions, and using meta-heuristic techniques for scaling the approach to complex test settings [6].

References

[1] S. Elbaum et al., "Techniques for improving regression testing in continuous integration development environments," in *Proc. of the Intl. Symp. on Foundations of Software Engineering (FSE'14)*, 2014, pp. 235–245.

[2] RightScale, Inc., "Dynamic scaling Jenkins in the cloud," http://www.rightscale.com/blog/cloud-management-best-practices/dynamic-scaling-jenkins-cloud.

[3] K. Incki et al., "A survey of software testing in the cloud," in *Proc. of the Intl. Conf. on Software Security and Reliability Companion (SERE-C'12)*, 2012, pp. 18–23.

[4] S. van der Burg and E. Dolstra, "Automating system tests using declarative virtual machines," in *Proc. of Intl. Symp. on Software Reliability Engineering (ISSRE'10)*, 2010, pp. 181–190.

[5] T. Hanawa et al., "Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems," in *Proc. of the Intl. Conf. on Software Testing, Verification, and Validation Workshops (ICSTW'10)*, 2010, pp. 428–433.

[6] M. Harman et al., "Cloud engineering is search based software engineering too," *Journal of Systems and Software*, vol. 86, no. 9, pp. 2225–2241, Sep. 2013.