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 accounts 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 accumulates 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 the available virtual machines, their 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 send 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

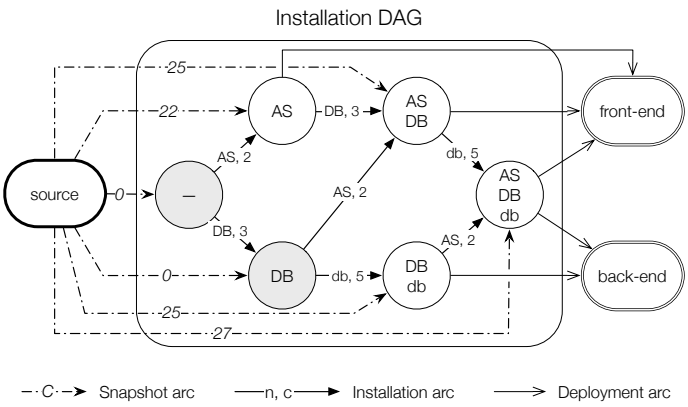

Fig. 1. Flow model for the Web service example

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

We start by extracting the list of required elements (e.g., AS, DB and db), the type of virtual machines that each test requires (e.g., *front-end* and *backend*) and the number of instances per machine that are deployed (e.g., 1 front-end and 1 backend for `T1`, 3 front-ends and 1 backend for `T2`). Given these information we build the installation DAG and create a number of instance nodes. The *installation DAG* is a directed acyclic graph that represents all the possible setups that we can build using the required elements from an empty virtual machines. Nodes represent generic setup of virtual machines and are labeled with the names of the installed components, while arcs represent the act of installing new elements and are labeled with the name of the new element with the cost for its setup. *Instance nodes* capture the types of virtual machines that are required by the tests, such as front-end and backend, and are connected to installation nodes by means of *deployment arcs* when the set of components listed in the installation node is a superset of the ones declared by the instance nodes. In the example, we connect the nodes $\{AS\}$, $\{AS, DB\}$ and $\{AS, DB, db\}$ to the front-end node that requires AS, and the nodes $\{DB, db\}$ and $\{AS, DB, db\}$ to the backend node that requires DB and db. To account for available virtual machines and potential snapshots we introduce in the flow model a synthetic *source* node and a number of *snapshot arcs* that represent the creation of specific virtual machines. Snapshotting costs are equal to zero if snapshots correspond to available virtual machines (e.g, $\{-\}$ and $\{DB\}$) and are equal to the sum of the fixed cost for creating a new virtual machine (20 in the example) plus the minimum cost for updating one of the available virtual machines. We complete the formulation of the problem with the flow constraints: The source node generates the flow that corresponds to the total number of the instances started for 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 elsewhere.

By solving this optimization problem we can identify the need for new snapshots and an optimal strategy to create them by updating the available virtual machines. For example, in case we would have to execute two times `T1` and one time `T2`, our approach would not suggest any snapshot; however, in case we would have more tests to run (e.g., six times `T1` and four times `T2`), our approach would suggest to create the $\{AS, DB, db\}$ virtual machine by snapshotting $\{DB\}$.

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] 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 construct 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.