Deploying large-scale service compositions on the cloud with the CHOReOS Enactment Engine

Leonardo Leite, Carlos Eduardo Moreira dos Santos, Daniel de Angelis Cordeiro, Marco Aurélio Gerosa, Fabio Kon
Department of Computer Science, University of São Paulo, Brazil
{leofl, cadu, danielc, gerosa, fabio.kon}@ime.usp.br

***Abstract*—In recent years, service-oriented systems are becoming increasingly complex, with growing size and heterogeneity. Developing and deploying such large-scale systems present several challenges, such as reliability, reproducibility, handling failures on infrastructure, scaling deployment time as composition size grows, coordinating deployment among multiple organizations, dependency management, and supporting requirements of adaptable systems. However, many organizations still rely on manual deployment processes, which imposes difficulties in overcoming such challenges. In this paper, we propose a flexible and extensible middleware solution that addresses the challenges present in the large-scale deployment of service compositions. The CHOReOS Enactment Engine is a robust middleware infrastructure to automate the deployment of large-scale service compositions. We describe the middleware architecture and implementation and then present experimental results demonstrating the feasibility of our approach.**

I. Introduction

Researchers and developers have been conceiving novel middleware systems for next-generation distributed applications on the Future Internet [1]. These novel middleware solutions must support systems composed of a very large number of distributed services, running on platforms hosted on a variety of mobile and cloud-based infrastructures. In this Future Internet scenario, distributed, decentralized business processes are implemented by interacting services provided by multiple participating organizations in a structure known as *choreography*.

Service choreographies are composition models where the knowledge about the control flow is distributed among the participants. Each service acts autonomously, holding the knowledge of when to execute its operations and with whom to interact [2]. This "control-flow distribution" is present in cross-organizational compositions where each organization defines its own business flow. Choreographies were designed, for example, to automate processes in a futuristic airport scenario [3], supporting interactions among a high number of actors[1], including passengers, airline companies, airport authority, etc.

This paper addresses the challenges encountered when deploying a large number of services, which can happen either in the case of a few compositions with a large number of services or in the case of a large number of compositions with a few services. The deployment process starts after the software is developed, packaged, and published, and finishes when the service is running [4].

The airport scenario mentioned before involves a large number of services within a single composition. On the other hand, a large suite of automated tests for a small service choreography is a good example of running a large number of services compositions.

When running a suite of automated acceptance tests, the following properties are desirable: (1) tests run in parallel, each one in an isolated environment, so a large number of tests is executed in a small amount of time, and (2) setting up the test environment is easily reproducible, so tests can be executed frequently, enabling continuous integration and continuous delivery [5].

The large number of services to be deployed and the high level of distribution worsen the already existing difficulties present in any deployment process. The challenges in such large-scale deployment of service compositions are the following:

- *Process:* Manual deployment is time-consuming and error-prone [6]. Deployment must include automated environment and service provisioning, turning deployment into a quick, safe, and reproducible process [5].
- *Third-party faults:* Components of distributed large-scale systems must expect and handle faults of third-party components [7], [8], [9]. Even if the probability of a failure in each component is small, the large number of components and interactions increases the likelihood of failures somewhere in the system [9].
- *Scalability:* The effective use of concurrency is a key to provide good scalability in the deployment process. However, the power of concurrency is seldom leveraged in commonly used *ad-hoc* deployment scripts. It is, thus, desirable to hide the complexity of concurrency from deployers in an effective middleware layer.
- *Heterogeneity:* Although web services emerged to reduce heterogeneity problems among systems and organizations, nowadays there are multiple mechanisms to implement the concept of services, including SOAP, REST, and others. Therefore, supporting heterogeneity is still important to service-based systems.

[1]Heathrow [http://www.heathrowairport.com] in London, for example, deals with more than 80 airlines, 190,000 passengers per day (peaks of 230,000), 6,000 employees, 1,000 take-offs and landings per day, and 40 catering services.

- *Multiple organizations:* Multiple partners in a single business process not only increases the heterogeneity problem, but also presents challenges to deployment coordination, as services of multiple organizations must be bound together, i.e., they need to discover dynamically the location of their dependencies.
- *Adaptability:* To accommodate dynamic changes in the execution environment, in usage patterns, and in business requirements, modern systems must be self-adaptive [10]. Therefore, the deployment process should consider the requirements of self-adaptive systems, accommodating their needs and being itself adaptive to support the continuous change of business requirements and maintain quality of service.

These challenges could be handled with *ad-hoc* solutions tailored to specific applications. However, this approach leads to low reuse within a single organization and across partners, yielding a very large cost for development and maintenance. A *middleware-supported* solution can address the common problems of deploying compositions, providing sophisticated and well-tested mechanisms. By using such middleware, deployers can write less and simpler code and contributors interested in the deployment problem can work together in a single code base to strengthen a common infrastructure.

The CHOReOS Enactment Engine (EE) is a novel, extensible, open source middleware system, developed in the context of the CHOReOS project[2], that provides a fully automated deployment process for service compositions. The Enactment Engine offers developers access to service deployment using the Platform as a Service (PaaS) model [11], relying on an Infrastructure as a Service (IaaS) [11] provider to support the virtualized target environment. In the cloud PaaS model, application developers do not need to care about the virtualized infrastructure and can focus on application development. The relationship between the cloud models and our proposed architecture is depicted in Fig. 1.

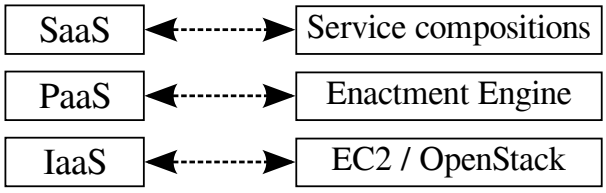

Fig. 1. Enactment Engine and cloud computing models.

Unlike current PaaS solutions, our EE is designed to support service compositions rather than web applications. Our solution extends past research on deployment of distributed component-based systems, including new mechanisms to support the dynamism of current environments by considering the new requirements of cloud deployment and large-scale service compositions.

II. Related work

The use of procedural languages (via scripts) is a flexible method to configure and deploy applications. Tools such as TakTuk [12] and Chef[3] provide scripting languages that can be used to configure each node and deploy, build, and test applications on individual nodes. Nevertheless, learning a new scripting language is an additional burden to deployers. Declarative languages can describe the application to be deployed in terms of its components and how they are structured [13], [14]. But even declarative languages for general deployment (i.e. not restricted to specific application classes), like Puppet[4], are hard to use since they require the specification of all the deployment details.

Our Enactment Engine was designed to deploy *service compositions* described as business processes and uses a notation based on a declarative language. Using the EE deployment specification is simpler than using general deployment specifications, such as Chef or Puppet, since our deployment specification presents only the necessary elements for deploying service compositions, abstracting underlying details, such as the middleware hosting the services.

Quéma et al. [15] conducted an empirical study about the performance and scalability of the deployment process of component-based applications. They propose a decentralized, fault-tolerant mechanism that deploys the application using a hierarchical approach based on the application's architecture. This approach enabled an asynchronous and parallel execution of the components that define the application. However, their hierarchical approach can only be applied to business processes whose communications can be modeled as trees. The EE does not impose such restriction.

One of the first large-scale platforms to support total control of the software stack (including the operating system) was Grid'5000 [16]. Its *Kadeploy OS provisioning system* [16], [17] has been designed to help system administrators to install and manage clusters and to provide users a flexible way to deploy their own operating systems on nodes for their experimentation needs. The EE achieves a similar result but in the more dynamic and heterogenous cloud environment by using the information about the application being deployed, the automatically-generated scripts, and the underlying IaaS platform.

The OASIS Topology and Orchestration Specification for Cloud Applications (*TOSCA* [18]) and Canonical's *Juju*[5] are industrial solutions for the problem of configuring and deploying services. Both systems implement the same underlying method, defining an abstraction for the services (TOSCA "service templates" and Juju "charms") that specifies how they are deployed, bound to other services, and how to scale them. In these systems, lower-level implementation artifacts are still necessary. This method can be viewed as more portable, but it imposes an additional burden to the developer. With the EE, the underlying execution environment is totally abstracted.

An application created using a cloud PaaS model can be deployed more easily, since all the underlying details about deployment, resource provisioning, automatic load balancing, monitoring, etc. are in the responsibility of the platform provider. Several cloud computing providers offer access to their resources using a PaaS model, such as Amazon Web

[2]http://choreos.eu

[3]http://www.opscode.com/chef

[4]http://puppetlabs.com/

[5]http://juju.ubuntu.com/

Services (AWS) Elastic Beanstalk, Google AppEngine, and Microsoft Azure. A problem with the use of a PaaS model is that the application becomes dependent on the tools and libraries provided by the chosen cloud computing provider. This dependence might lead to vendor lock-in and may become a problem if the developer decides to change its cloud vendor. The EE solves this problem by not imposing a programming model and by relying on an extensible architecture to support new programming technologies and new cloud platforms.

Other open source frameworks also aim to solve this dependence problem. Cloud Foundry[6] provides an open source PaaS framework that can be deployed on a variety of cloud computing platforms and that manages any type of cloud services. However, that platform is targeted to simple web applications. The EE supports the execution of applications described as business processes. This feature is related to two commercial solutions: Amazon Simple Workflow Service[7] and Force.com Visual Process Manager[8]; both offer a simple coordination mechanism to the execution of each activity of a business process, *i.e.*, an orchestrator.

III. The CHOReOS Enactment Engine

In this section, we present the main ideas and the architecture used by the EE to deploy decentralized service compositions on large-scale platforms.

A. Architecture

The major components encountered in the Enactment Engine execution environment (depicted in Fig. 2) are the following.

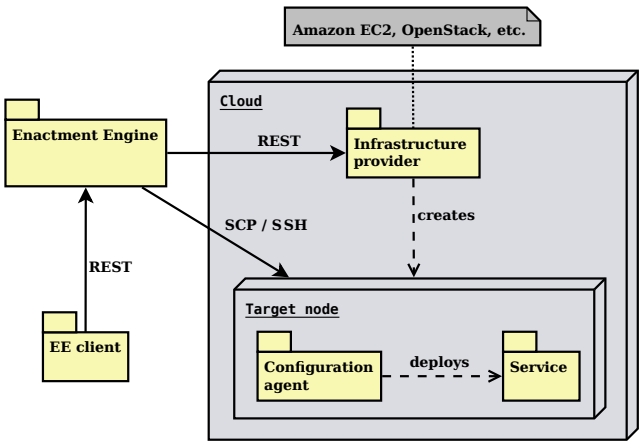

Fig. 2. CHOReOS Enactment Engine execution environment.

- **Infrastructure provider** creates and destroys virtual machines (also called *nodes*) in a cloud computing environment. Currently, there is support for both Amazon EC2 and OpenStack as infrastructure providers.
- **Configuration agent** is installed by the EE in each cloud node. It manages and runs scripts that implement the process of configuring operating systems, installing required middleware, and finally deploying the services. Chef-Solo[9] is our configuration agent.
- **EE client** is a script, written by deployers, that specifies the choreography deployment and invokes the EE to trigger the deployment process. *Deployer* is the human operator responsible for the deployment process.
- **Enactment Engine** deploys choreography services according to the specification sent by the client.

The deployment process performed by the EE, depicted in Fig. 3, encompasses the following steps: **(1)** *Client request:* receives the specification of the service composition to be deployed. **(2)** *Node selection:* selects one or more cloud nodes where each service will be deployed, creating new nodes if necessary and accommodating service non-functional requirements. **(3)** *Scripts generation:* dynamically creates scripts for environment configuration and service launching. **(4)** *Nodes update:* executes the scripts on the selected nodes, so services and their dependencies are installed and launched. **(5)** *Service binding:* injects addresses of service dependencies (other participant services in the composition). **(6)** *Response to client:* sends the response to the client, informing which services were deployed, in which nodes, and their service URIs, enabling runtime monitoring.

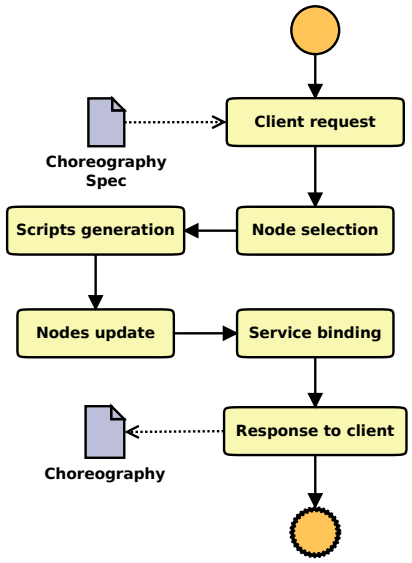

Fig. 3. Deployment process performed by the Enactment Engine.

Optionally, the EE can deploy a monitoring infrastructure, by deploying a monitoring agent (Ganglia[10]) on each node.

B. Automating the deployment process

It is imperative to make service deployment a fully automated process [7] to ensure testability, flexibility, and reliability. Deploying large-scale distributed systems is a time-

[6]http://www.cloudfoundry.org
[7]http://aws.amazon.com/swf
[8]http://www.salesforce.com/platform/process
[9]http://docs.opscode.com/chef_solo.html
[10]http://ganglia.sourceforge.net

consuming and error-prone activity [6]; the deployment process in complex systems must be automated[5]. In the Future Internet large-scale scenario that motivates this study, one can expect that service compositions will grow in complexity and scale. Thus, a scalable and resilient deployment service will be a crucial element in this context.

The EE enables deployment automation by providing a RESTful API that receives the declarative description of the service composition to be deployed and returns information describing the deployment outcome. The use of a declarative description of composition have been used previously in the context of component-based system deployment [19], [13].

The description must provide all the necessary information to deploy the composition, including, for each service, where the service package can be downloaded from and the type of the package (e.g., WAR, JAR). It may also specify existing third-party services that are already available on the Internet and should be bound to the choreography dynamically. The deployer can choose to write the specification directly in XML format or using simple Java objects. In both cases, the choreography description must adhere to the data model presented in Fig. 4.

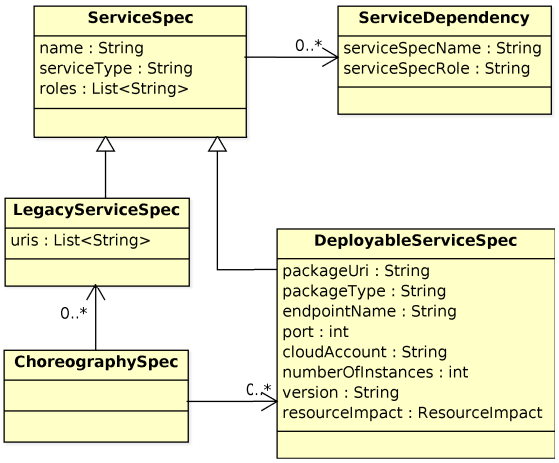

Fig. 4. Data model defining choreography specification.

Each service can consume operations from other services in the composition. Thus, the deployment process must *bind* the services, so they know how to invoke each other. When using the EE, each consumer service must implement an operation called `setInvocationAddress` that receives dependency endpoints. Service dependencies are also declared on the composition specification, so the EE can properly inject dependencies [20] on dependent services in a way similar to the one proposed by Magee and Kramer [13].

On the one hand, the current trend in large-scale system deployment is the use of elastic resources provided by cloud computing. Virtualized resources provided by the cloud leverages the automation of the deployment process [5]. Since the creation of a whole new environment can be automated by VM provisioning, each new deployment can easily create a clean environment where the deployed system will run, leveraging reproducibility and environment isolation.

On the other hand, using cloud resources brings additional challenges, since it is necessary to take into account the dynamic nature of the cloud [21]. Different from the scenarios studied in earlier works on component-based systems deployment [19], [13], the target nodes are more dynamic and it is not possible to know their IP addresses when writing the composition specification. The service binding must be performed at runtime (via `setInvocationAddress`), and the node allocation policy must be more flexible, *i.e.*, a service must not be allocated to a static IP before its VM is running.

C. Handling cloud infrastructure failures

Distributed large-scale systems must expect and handle third-party component faults [7], [8], [9]. An example of a typical failure in a cloud environment involves VM provisioning. When a new node is requested to the infrastructure provider, there is a chance that the provisioning will fail. Moreover, some nodes may take much longer than average to be operational. Other steps that may fail during the deployment process are SSH connections and the execution of scripts on cloud nodes.

Fig. 5 shows the observed distribution of node creation time when concurrently creating 100 nodes on the Amazon cloud (repeated 10 times). We count the time since the creation request until the machine is able to accept SSH connections. We observed a failure rate of 0.6% when concurrently creating 100 nodes. It is interesting to note that the creation time has a stable median, but that larger creation times are expected when creating a large amount of nodes. In our observations, failures and long provisioning times affected up to 7% of node creation requests.

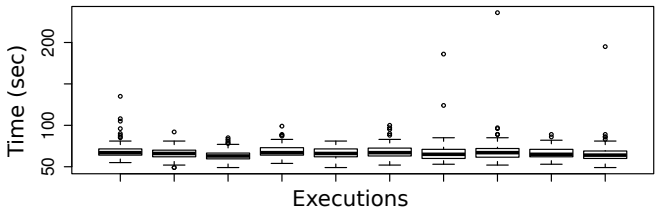

Fig. 5. Observed EC2 instances creation times.

A simple approach adopted by the EE to handle external failures was encapsulating the logic of invoking external systems in a class, called Invoker, and using it in a disciplined way throughout the system. For each kind of task (*e.g.*, file transferring), our Invoker is configured with the following parameters: a *task*, which is a routine that will invoke an external system, a quantity of *trials* to perform the task, a *timeout* for each trial, and an *interval* between trials.

The EE adopts a particular strategy to handle failures during the creation of new VMs. When a request arrives, the EE tries to create a new node. If the creation fails, or takes too long, an already created node is retrieved from a reservoir of idle nodes. This strategy avoids waiting again for another node creation. The initial reservoir capacity is defined by configuration and it is refilled every time a node is requested. If the pool size is decreased and reaches a given threshold, the capacity is increased, trying to avoid a future situation with an empty reservoir when an extra node is requested.

The reservoir approach imposes an extra cost to keep more VMs running (in an idle state). However, this problem is treated in the EE by a distributed management algorithm in each node: if the node is idle for $N-1$ minutes, where N is a threshold of time that implies additional cost, the node sends to the EE a message requesting its destruction. Thus, after a time of inactivity in the EE, the reservoir eventually becomes empty, and it will be filled again only if new requests arrive.

Another important practice related to fault tolerance is *graceful degradation* [7], [22]. In our context, graceful degradation means that if a service cannot be properly deployed, it is not acceptable to halt the entire deployment process. With the EE, if some service is not properly deployed, the deployment process continues, and the EE response details the problems encountered during deployment, enabling recovery actions. Nonetheless, it is important to point out that graceful degradation responsibility must be shared with service implementation, since each service must know how to behave in the absence of some of its dependencies.

A possible recovery action enabled by the EE after a deployment with failures is simply one more client request for the composition deployment. In such situation, the EE will deploy only the missing services. This is possible thanks to the *idempotent* design and implementation of the deployment operation. Idempotence is an important property of network available operations. Its guarantee is specified in REST APIs [23] and, in the deployment context, it is a key feature of Chef scripts[11].

D. Scalability

When deploying a large amount of services in a distributed environment, it is not desirable to perform the deployment of each service sequentially. Since the deployment of different services are independent tasks, deploying them concurrently drastically increases scalability.

We say an architecture is perfectly scalable if it continues to yield the same performance per resource, albeit used on a larger problem size, as the number of resources increases [24]. In the deployment context, it means that, ideally, the deployment time should remain constant when there is a proportional increase on the number of services to be deployed and on the number of available nodes. The number of services to be deployed increases in two situations: 1) when deploying larger compositions and 2) when deploying a larger number of compositions simultaneously.

Concurrent programming is hard and error prone and, in general, scripting tools do not provide good support for concurrency. Without proper failure handing (see Section III-C), customized scripts can also impact scalability. Therefore, handling concurrency and fault handling in the middleware layer are important steps towards facilitating the effective implementation of a scalable deployment process.

E. Extensibility

Many current PaaS solutions are known for limiting technical choices in the application architecture. The EE eases such problem by handling technological heterogeneity and customization needs offering the following extension points:

- *Service type:* service binding in technologies other than SOAP (*e.g.*, REST) requires new conventions to the `setInvocationAddress` operation, which can be accomplished by implementing the ContextSender interface.
- *Package type:* users may add support for a new package type by writing a Chef cookbook template. JARs and WARs are packages already supported.
- *Infrastructure provider:* new cloud infrastructures can be supported by implementing the CloudProvider interface. To ease its implementation, EE uses jclouds[12], an open source library with portable abstractions for many cloud providers. We already support Amazon EC2 and Openstack.
- *Node selection policy:* by implementing the NodeSelection interface it is possible to build a new dynamic node selection policy, having access to non-functional service requirements to be considered on the algorithm. A few policies are currently available, including *limited round robin*, which first creates a predefined number of nodes, and then selects nodes in a round-robin fashion for each deployed service.

This flexibility provided by the EE helps to overcome the current limitations of PaaS solutions that restrict the technological choices of application developers, such as the IaaS provider and the application programming language.

F. Supporting cross-organizational compositions

A service composition may encompass services belonging to multiple organizations. The EE has two main mechanisms to cope with this situation. The first uses a service specification attribute to define under which "cloud account" the service will be deployed (and billed).

The second mechanism is used when an organization performs the deployment of a composition encompassing both services belonging to the organization itself and also existing third-party services. This situation can be modeled in the composition specification, so the EE will deploy only the services belonging to the organization and bind them to the existing third-party services.

G. Coping with adaptability

The EE supports the development of self-adaptive compositions by providing the following features: service composition updates, service migrations, service replications, and monitoring infrastructure deployment.

Service replication associated with load balancing, in particular, is a common strategy to provide system scalability [21]. The monitoring infrastructure consists of *Ganglia probes* deployed by the EE on target nodes to collect virtual machine metrics such as CPU, memory consumption, etc. These features make the EE a suitable option for researching service composition self-adaptation. They ease the implementation of adaptive systems by letting researchers be more focused on high level adaptation problems instead of highly specific deployment details.

[11] http://docs.opscode.com/chef_why.html

[12] http://jclouds.apache.org

IV. Deployment with EE versus *ad-hoc* deployment

Middleware-based solutions for deployment automation brings several benefits over *ad-hoc* approaches. To better understand such benefits from the deployer point of view, we investigated how the CHOReOS Enactment Engine improves the deployment process by providing a middleware-supported solution.

We conducted an evaluation by developing an *ad-hoc* solution dedicated to the deployment of a single choreography. The "Airport choreography" is an example provided by Airport domain experts [3] containing 15 services. We also deployed the same choreography using the EE. Both solutions are publicly available online[13]. Comparisons with other middleware solutions were not performed since, to the best of our knowledge, the EE is the first comprehensive middleware solution for choreography deployment on the cloud.

To deploy the Airport choreography with the EE, we wrote the choreography specification and a program to invoke the EE, launching the deployment. The choreography specification was assembled with simple Java objects consisting of 162 lines, with 11 lines of code per service on average, in 40 minutes. The deployment launch program uses the EE Java API and has only 22 lines of code. After this code was written, deploying the choreography over three nodes with EE took only 4 minutes.

The development of the *ad-hoc* solution required nine hours from one developer. Plus, 60 additional minutes were needed (by the same developer) to actually execute the deployment, distributing the 15 services over three different nodes. This solution required the writing of 100 LoC of shell scripts, 220 LoC of Java, and 85 LoC of Ruby (for Chef).

The execution of the *ad-hoc solution* has several steps, including some manual ones. For each target node, the deployer must log into it, install git, checkout the cookbooks, execute the `install_chef` script, edit some configuration files defining which services will be deployed on the node, and run chef-solo. After deploying the services, the deployer must edit the IP tokens within the `bind_services` script, and finally execute the `bind_services` script. This *ad-hoc* solution has several drawbacks:

- It requires three different technologies: shell script, Java, and Chef. Command line expertise was also necessary. It suggests that a large set of skills is required from the developer of deployment solutions. Some of those skills, such as using Chef or understanding SOAP, are known not to be easy to learn.
- For each target node, the deployer must perform time-consuming and error-prone manual steps, like editing configuration files. Missing commas or mistyping service names are likely to happen. More automation would be provided by using more advanced deployment tools, such as Capistrano[14], but it would be one more skill to learn.
- There is very little concurrency in the process. With the provided scripts, the deployer could somehow enhance parallelism by using tools like Byobu[15] to enter the same command in multiple machines. But this requires yet another deployer's skill and it is a very limited way to scale the process.

[13] http://ccsl.ime.usp.br/EnactmentEngine#source – v2014-07

[14] http://www.capistranorb.com/

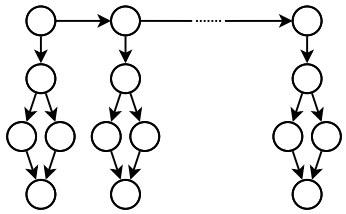

Fig. 6. The topology of the compositions used in the experiments.

In this example, we used a service composition with only 15 services. Using large-scale compositions would increase much more the complexity of the *ad-hoc* solution. To reach a complete solution in the *ad-hoc* approach, an extra development effort would be required to implement features already provided by the EE, such as third-party failure handling, composition update, dynamic node allocation policies, concurrent deployment, etc. In addition, to produce the *ad-hoc* solution we used some code already available in the EE. Deployers would have to code them from scratch.

We recognize that this assessment has its limitations, since results depend strongly on the deployer's technical skills. Conducting a rigorous software engineering experiment with several developers assuming the deployer's role would bring stronger evidence. However, we believe that the assessment described here is enough for expanding our understanding of the value added by a middleware-supported solution such as the EE.

V. Performance and scalability analysis

It is expected from a middleware-supported solution for automated deployment the ability of executing the deployment process with acceptable performance and scalability, especially in scenarios with a large number of services to be deployed. Therefore, we conducted experiments to evaluate the performance and scalability of the proposed EE in terms of its capability to deploy a significant number of compositions in a real-world cloud computing platform.

Our experiments use a synthetic workload modeled based on a common pattern found on many business processes: a main service that uses other sub-services (potentially provided by different organizations) to deliver its result. These services are modeled using a *fork-join* service composition. This workload is depicted in Fig. 6.

Initially, we conducted a multi-variable analysis of the EE performance by deploying service compositions in the following scenarios: 1) a small set of small compositions; 2) a small set of larger compositions; 3) a larger set of small composition; 4) a larger ratio of services per node. Table I quantifies each scenario.

In our experiments, the node allocation policy was the "limited round robin". The idle node reservoir size was five, and

[15] http://byobu.co/

TABLE I. DEPLOYMENT SCENARIOS.

Scenario	*Compositions*	*Size*	*Nodes*	*Services/Nodes*
1	10	10	9	11 or 12
2	10	100	90	11 or 12
3	100	10	90	11 or 12
4	10	10	5	20

TABLE II. EXPERIMENTAL RESULTS.

Scenario	*Time* (s)	*Successful compositions*	*Successful services*
1	467.9 ± 34.8	10.0 ± 0	100.0 ± 0 (100%)
2	1477.1 ± 130.0	9.3 ± 0.3	999.3 ± 0.4 (99.9%)
3	1455.2 ± 159.1	98.9 ± 0.8	998.5 ± 1.3 (99.9%)
4	585.2 ± 38.1	10.0 ± 0.1	100.0 ± 0.1 (100%)

the node creation timeout was 300 seconds. We used Amazon EC2 as the cloud computing service and the VMs were EC2 small instances (1.7 GiB of RAM and one vCPU equivalent to 1.0–1.2 GHz), running Ubuntu GNU/Linux 12.04. The EE was executed on a machine with 8 GB of RAM, an Intel Core i7 CPU with 2.7 GHz and Linux kernel 3.6.7. The EE version used, and raw data collected from the experiments are available online[13], for reproducibility purposes.

Table II presents, for each scenario, the time necessary to deploy all compositions plus the time to invoke them to make sure they were correctly deployed. Each scenario was executed 30 times and their averages are presented with a 95% confidence interval. The table also shows how many compositions and services were successfully deployed.

The results show that the EE scales well in terms of the number of services being deployed. In scenarios 2 and 3, when the number of services was increased by 10 times, the deployed time increased only 3 times approximately. This time increment was caused mainly by the fact that the higher the number of services, the higher the likelihood of a fault triggering the re-execution of some routine.

The results also show that when the number of services per node was doubled (scenario 4), the deployment time increased nearly 25% (comparing with the first scenario). Part of this overhead was caused by the increase of the number of Chef scripts that must be executed (sequentially) on the nodes.

During our experiments, we observed that, thanks to the EE fault tolerance mechanisms, the amount of failures was low: all the services were successfully deployed in more than 75% of the executions. By a failure, we mean that one service was not properly deployed. In scenario 1 we got no failures, whereas in scenario 4 we had only one failure. In scenario 2, the worst situation had 3 failures out of 1,000 services. In scenario 3, we got one execution with 20 failures, but it was an exceptional event, since the second worst situation had only 3 failures.

Finally, we observed that 80% of the executions did not use the node reservoir. When used, there was a maximum of six uses but, most of the time, there was only one use. We also observed that the deployment time was not significantly affected when the failures on the cloud environment occurred, because new nodes were immediately retrieved from the reservoir.

We also conducted experiments to evaluate the performance and scalability of the EE in terms of its capability to deploy a large number of services in a single composition. These

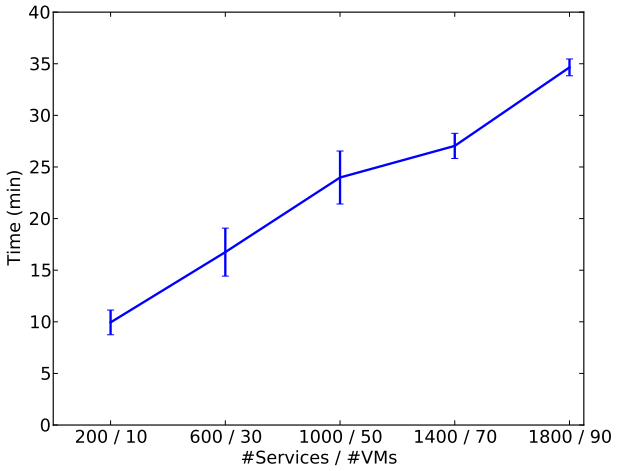

Fig. 7. Average deployment times for increasingly larger compositions.

experiments were conducted in 5 scenarios by varying the deployed composition size and the number of used target nodes, while fixing 20 deployed services per node. Each scenario was executed 10 times.

The composition topology used was the same as before (Fig. 6) and the environment used to run the EE was a virtual machine (8 GiB of RAM and 4 vCPUs) hosted in our University infrastructure. The created nodes were Amazon EC2 small instances and node creation timeout was set to 250 seconds. The average deployment times with 95% confidence intervals are shown in Fig. 7.

These results show a good scalability in terms of deployed services. After increasing 9 times the number of deployed services, the deployment time increased only 3.5 times. In absolute numbers, each increase in 400 deployed services was responsible for increasing the deployment time from 180 to 460 seconds. As in the previous experiment, part of deployment time increase can be explained by the higher occurrences of failures in large-scale scenarios. Concerning service deployment failures, the worst executions of each scenario had 1, 1, 2, 2 and 4 services not successfully deployed out of 200, 600, 1000, 1400 and 1800 services, respectively. This can be considered a very low failure rate. Moreover, the failed services could be easily deployed after the client simply trigger again the deployment operation (EE would not redeploy the already deployed services).

We performed also a final experiment to asses the EE failure handling mechanisms (Fig. 8). Enabling the invoker and the reservoir, a deployment of 100 services in 10 nodes succeeded in the 10 performed executions. But when disabling them, the deployment succeeded completely only in 3 executions. In other executions we got 10 or 30 failed services. The failures were multiple of 10 because they were caused by failures on EC2 node provisioning (each node should host 10 services).

VI. CONCLUSION AND FUTURE WORK

Sophisticated distributed applications of the Future Internet will be composed of a large number of highly-distributed services executing on heterogeneous mobile- and cloud-based environments, interacting at runtime with millions of users.

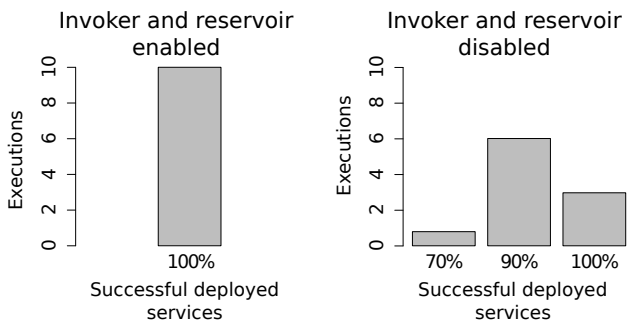

Fig. 8. Assessing the EE failure handling mechanisms.

To enable the easy deployment and execution of such complex service compositions, flexible, robust, and adaptive middleware systems will need to be devised. In this paper, we introduced a novel middleware supporting the adaptive enactment of complex service compositions on the cloud. This middleware highly facilitates the deployment, on the cloud, not only of large-scale service choreographies but also of large quantities of service compositions. It also provides runtime support for monitoring and adaptation of the compositions.

Experimental results demonstrate that the proposed architecture is feasible and that acceptable performance and scalability can be obtained. One lesson learned about achieving scalability in large-scale systems is that scalability is not obtained by a single special architectural decision, but rather by a set of several small design and implementation decisions, such as the ones we list in Section III.

Our solution also encourage other researchers to take advantage of our flexible architecture and open source software[13] to perform empirical evaluation of service-based systems. The EE, for example, can facilitate research about QoS-based adaptation of service compositions on different infrastructure providers, since researchers can implement new node allocation policies for the Enactment Engine very easily. Developers of middleware systems targeting large-scale environments can also benefit by studying our design and implementation decisions.

Ongoing and future work include (1) improving failure handling algorithms, making them more adaptive by learning from the environment how often each kind of failure occurs, and (2) improving the current support for cross-organizational composition deployment by automatically federating multiple EE instances.

ACKNOWLEDGEMENTS

The authors would like to thank the CHOReOS project (FP7 European program #FP7-ICT-2009-5) and FAPESP (#2012/03778-0) for their financial support.

REFERENCES

[1] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadist, M. Autili, M. A. Gerosa, and A. B. Hamida, "Service-oriented middleware for the Future Internet: state of the art and research directions," *Journal of Internet Services and Applications*, vol. 2, no. 1, pp. 23–45, Jul. 2011.

[2] A. Barker, C. D. Walton, and D. Robertson, "Choreographing Web Services," *IEEE Transactions on Services Computing*, vol. 2, no. 2, pp. 152–166, 2009.

[3] P. Chatel and H. Vincent, "Deliverable D6.2. Passenger-friendly airport services & choreographies design," Available online at: http://choreos.eu/bin/Download/Deliverables, 2012.

[4] OMG, "Deployment and configuration of component-based distributed applications (DEPL)," Apr. 2006.

[5] J. Humble and D. Farley, *Continuous Delivery*. Addison-Wesly, 2011.

[6] E. Dolstra, M. Bravenboer, and E. Visser, "Service configuration management," in *Proc. 12th Intl. Workshop on Software Configuration Management*. ACM, 2005.

[7] J. Hamilton, "On designing and deploying Internet-scale services," in *Proc. 21st Large Installation System Administration Conference (LISA '07)*. USENIX, 2007.

[8] P. Helland and D. Campbell, "Building on quicksand," *CoRR*, vol. abs/0909.1788, 2009.

[9] B. Pollak, Ed., *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, Jun. 2006.

[10] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*, vol. 15, no. 3, pp. 313–341, 2008.

[11] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[12] B. Claudel, G. Huard, and O. Richard, "TakTuk, adaptive deployment of remote executions," in *Proc. HPDC'09*. ACM, 2009, pp. 91–100.

[13] J. Magee and J. Kramer, "Dynamic structure in software architectures," in *Proc. 4th Symposium on foundations of software engineering (SIGSOFT '96)*. ACM, 1996.

[14] J. Magee, A. Tseng, and J. Kramer, "Composing distributed objects in CORBA," in *Proc. 3rd International Symposium on Autonomous Decentralized Systems*, 1997.

[15] V. Quéma, R. Balter, L. Bellissard, D. Féliot, A. Freyssinet, and S. Lacourte, "Asynchronous, hierarchical, and scalable deployment of component-based applications," in *Component Deployment*. Springer, 2004, vol. 3083, pp. 50–64.

[16] R. Bolze et al., "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *Intl. J. of High Performance Computing Applications*, vol. 20, no. 4, 2006.

[17] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum, "Kadeploy3: Efficient and scalable operating system provisioning," *USENIX ;login:*, vol. 38, no. 1, pp. 38–44, Feb. 2013.

[18] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, "Integrating configuration management with model-driven cloud management based on TOSCA," in *Proc. of the 3rd Intl. Conference on Cloud Computing and Service Science*. SciTePress, 2013, pp. 437–446.

[19] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury, "Architecturing and configuring distributed application with Olan," in *Proc. of Middleware*, 1998.

[20] M. Fowler, "Inversion of control containers and the dependency injection pattern," Jan. 2004, http://martinfowler.com/articles/injection.html.

[21] M. Tavis and P. Fitzsimons, "Web Application Hosting in the AWS Cloud: Best Practices," Amazon, Tech. Rep., Sep. 2012.

[22] E. A. Brewer, "Lessons from giant-scale services," *Internet Computing, IEEE*, vol. 5, no. 4, pp. 46 –55, 2001.

[23] S. Allamaraju, *RESTful Web Services Cookbook*. O'Really Media / Yahoo Press, 2010.

[24] M. Quinn, *Parallel Computing: Theory and Practice*, 2nd ed. McGraw-Hill, 1994.