TOWARDS THE NEXT GENERATION OF MODEL DRIVEN CLOUD PLATFORMS

Javier Esparza-Peidro and Francesc D. Muñoz-Escoí
Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Camino de Vera s/n, Valencia, Spain

Keywords: Cloud, Cloud computing, Cloud platform, Scalability, Model driven, Model.

Abstract: Current cloud platforms are based on two mainstream models: Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). Both approaches entail strenghts and weaknesses that we collect and present in this paper and we conclude the need to devise a new approach, based on graphical models, to overcome the imposed limitations. This model driven approach is introduced and slightly described, highlighting the importance of a comprehensive scalable modeling language and uncovering new research lines for designing self-manageable cloud platforms.

1 INTRODUCTION

Current cloud platforms are based on two mainstream models: Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) (Vaquero et al., 2008), (Foster et al., 2009), (Lenk et al., 2009) . Both approaches entail strengths and weaknesses that we will try to collect and present in the following paragraphs.

The IaaS model (Amazon, 2011) is aimed at professionals with system administration skills since the provider does typically supply a set of services for managing virtual machines and virtual networks. Defining machine settings, operating system configurations and network architecture is the users' responsibility, and requires expertise in all these domains. Deploying an application in this scenario is not an easy task either, since each application component must be distributed to the correct machine, installed and started, and the whole process must be correctly orchestrated to have the overall application properly working. Furthermore, managing application scaling usually involves developing some additional software to measure and detect heavy traffic conditions and scaling out some key application components. Lately some cloud offers include tools for automatically detecting these conditions and scaling out without user intervention.

On the other hand, the PaaS model (Google, 2011), (Azure, 2011) hides all the above mentioned complexities and provides a consistent high level software platform for developing scalable applications. The intricate details of configuring the underlying layers, connecting components and on-demand scaling are automatically managed by the platform and covered by a collection of APIs which provide high level constructs to assist in the application development. This approach typically imposes a specific programming language, and usually a new application model must be followed, forcing the developer to organize the application components in certain way. Consequently legacy systems are difficult to migrate to the new platform, since reengineering processes are required. Moreover, this model typically focuses on the development of new management applications, resulting useless in systems-oriented solutions where the different components are interconnected network servers, as in a corporate network.

Therefore, no approach is suitable for every case, and each one has its specific shortcomings. IaaS is adequate for legacy systems or system-oriented applications provided that the user has deep understanding of systems administration and assuming some semi-automated scaling procedures. Conversely, PaaS is appropriate for new business applications tailored for a particular programming platform where all scalability aspects and much of the application configuration is automatically carried out by the platform.

To summarize, IaaS is more open and versatile, in exchange for extra management effort, while PaaS is more closed and limited, but it automates nearly any aspect of the platform, letting the developer to focus on the real application development, which will be irrevocably bound to the platform on which it was developed.

In this paper we try to conciliate both worlds, promising to obtain the same versatility and platform independence as IaaS solutions, and reaching automation levels very similar to the PaaS counterparts. To this end, we devise a new approach that merges the best characteristics of both solutions, and assists the developer with graphical models, inspired by the Model Driven Engineering (MDE) principle. Besides helping in the specification steps, our methodology will be able to automatize application deployment and to build scalable self-administering (i.e., autonomous) applications.

The rest of the paper is organized as follows: section 2 presents our model driven approach and introduces an overview of the overall methodology. Section 3 discusses a key element in this methodology, the language used for modeling scalable applications. We list some of its most important characteristics and we show a preview of a platform-independent language which may meet such characteristics. Section 4 suggests some issues which should be solved in order to use such language in a concrete environment. Section 5 outlines the most important features that a platform should provide in order to support the presented methodology. Finally, conclusions of this work are presented in section 6.

2 MODEL DRIVEN APPROACH

As it is known, a software product can become a very complex creature, so much so that thousands of developers may be involved in the construction of a single prototype. To address such a hard job we need very powerful tools and well defined procedures. The Software Engineering discipline (DeMarco, 1979) emerged to give solution to these problems. It defines a set of best practices and specifies a concrete sequence of steps that any application development process should go through in order to guarantee a minimum quality level. This process can be roughly broken into the conceptualization phase and the implementation phase. The first stage involves modeling any aspect of the application, including static and dynamic characteristics, so that the designer can define with great detail the entire application in advance, before initiating the implementation tasks. To this end the designer needs a strong modeling language, capable of capturing any software property. Most of the times, the chosen language is the Object Management Group's Unified Modeling Language (UML) (UML, 2011), (Booch et al., 2005).

Nowadays any serious software project begins with a robust requirements specification and a complete collection of models describing in detail the structure and behavior of the software. The latest software development methodologies take advantage of this approach, taking as input the software models and trying to automatize the next part of the process, that is to say, the software construction. This methodology is coined as Model Driven Engineering (Kent, 2002), (Schmidt, 2006) and the Object Management Group's Model Driven Architecture (MDA) (MDA, 2011), (Kleppe et al., 2003) is one of the best known initiatives in this field. In MDA the designer defines a Platform Independent Model (PIM) by using UML, which describes the architecture and functionality of the software, including action semantics. This model is not bound to any specific platform, rather it describes in an abstract way all aspects of the modeled software. Then the PIM is translated into one or more Platform Specific Models (PSMs), which are models targeted to specific technological platforms. PSMs are finally translated into executable code. Translations between models are considered as model transformations, and a language called Query, Views, Transformations (QVT) (QVT, 2011) is provided for defining such mappings.

Working with abstract models has a number of advantages: (1) models allow engineers to reason about the relevant application properties, ignoring minor details usually linked to specific platforms, (2) they increase productivity, since developers work with high level languages and concepts they are comfortable with, avoiding low level and error-prone details, and making easy communication between engineers, and (3) they provide platform-independence, portability and cross-platform interoperability, due to the abstract nature of the model, which avoids any binding with a particular platform, and increases the lifespan of the solution.

Our aim is to follow the presented model driven approach for designing and deploying scalable architectures on a cloud platform. The user captures every aspect of the application architecture in the model, specially the scalability features. The platform is responsible for everything else, including distribution, deployment, execution, monitorizing, on-demand scaling, etc. The application just works. In this sense, the application model is some sort of executable specification. To the best of our knowledge, today no cloud platform requires the specification of an abstract model in order to deploy a complete solution, or at least not in the sense we are proposing in this paper.

In Figure 1 we illustrate this methodology. The software architect defines the application model along with the application bits. Both elements may be pack-

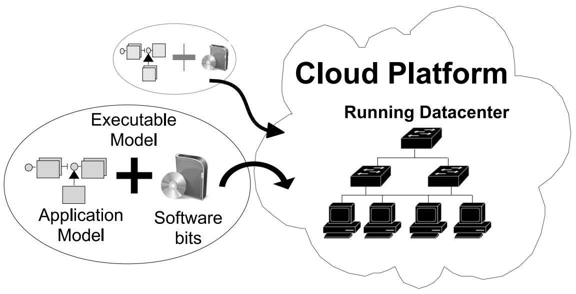

Figure 1: Model driven methodology.

aged in just one bundle that will be called executable model. The executable model is then injected into the platform. The platform analyzes the model and creates a plan for deploying the different parts of the application on the appropriate physical machines, according to the application properties described in the model. This plan introduces several advantages: (1) application deployment can be fully automatized, (2) the plan is able to devise a components monitoring scheme, making possible a fast reaction of the platform and/or the application components when different events occur (workload variations, component failures, etc), and (3) either the plan or the model may hold different rules that allow the implementation of self-managed components.

3 SCALABLE COMPONENT LANGUAGE

The key component of the methodology advocated in this document is the language used to model the scalable application. To keep the language platform-independent it should not focus on intracomponent details, rather it should be positioned at the architectural level, being able to define abstract components and connections between those components. Furthermore, the language must be expressive enough to catch all the scalability properties related with the application components. The minimum and maximum number of instances of one component, or the circumstances under which the component gets replicated are some examples of scalability features.

Therefore we must find a modeling language which meets the above-mentioned requirements. Literature searches have only given a fairly good result which loosely matches our expectations. It is the UML component and deployment diagram defined by OMG, which targets the definition of an application at the architectural level. This model is comprehensive enough for defining abstract components and connections between components, but unfortunately the resulting diagrams do not collect characteristics related with scalability. Trying to capture these properties in the model is ardous and confusing, since the language was not designed for that purpose.

For example, in a component diagram there is no way to specify that a component may replicate in n instances, or that the set of instances may grow or shrink within some limits and under certain circumstances. Also, connecting scalable components, which at runtime may expand in many instances, can not be solved with simple interface connections. Rather, complex connection elements should be considered, such as load balancers, proxies, etc.

For the above reasons, we believe that a new modeling language must be designed. The language must be rich enough to describe the individual components of any distributed application, as well as its connections. Regarding scalability, the language must provide a means for defining when, where and how many replications of each component may take place. Also, the language must supply tools for accurately specifying different types of connections between components, considering the multivalued nature of such relationships in scalable architectures. Another goal to achieve is to design a language which software architects feel comfortable with.

To attain the discussed objectives the most suitable approach involves taking features from a widely spread modeling language - with similar purposes - and redefining some of its graphical components and introducing some new ones. We are currently working on the definition of such a modeling language, though some extra work is needed to have a precise specification.

In Figure 2 we present a diagram written with such hypothetical modeling language, and we present some of the challenges the language will need to face to properly design scalable architectures. The diagram closely resembles the UML component and deployment diagram, facilitating its understanding and reducing the learning curve required by software architects. The example depicts an elaborated business application, which is composed of four connected components: a simple HTTP front end which forwards requests to the next component, a web application implementing the application GUI, a business application containing the application logic and a back end SQL database server. Each component may expose a public interface, represented by the familiar ball symbol.

The first three components are scalable, that is, there may be several instances at runtime. It is represented by a multiple box. The minimum and maximum number of instances of each component is de-

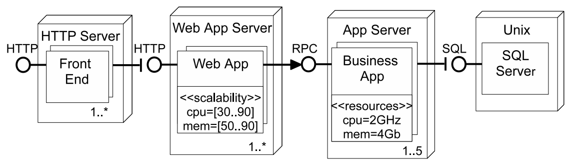

Figure 2: Application model example.

fined at the bottom right corner. For example, at execution time one or more instances of the web application may be concurrently running.

In a real environment, each component runs within a concrete execution environment, which provides some facilities and imposes some requirements to the embedded component. The execution environment is represented by a 3D box which surrounds the component. In the example, the web application component must run inside a web application server, whereas the business application requires an application server to successfully run.

Each component can define scalability properties, which describe how the component instance set should grow and shrink at runtime. This feature is specified inside a ≪scalability≫ compartment in the component. In the example, the web application component should be replicated when the average processor consumption of all its instances reaches the 90%. Similarly, when the average processor utilization of all instances reaches the 30%, some instances should be destroyed.

Each component requires specific resources, such as processor and memory, to successfully run. This aspect is defined inside a ≪resources≫ compartment in the component. In the example, the business application component requires a 2GHz processor and 4Gb of main memory to run.

Finally, components must be able to connect, in such a way that one component consumes the interfaces published by other components. Connections do not entail any problem in 1 to 1 runtime scenarios, that is to say, when one instance of a component connects to another single instance of a second component. Troubles arise when considering contexts with multivalued connections, in other words, when one or more instances of a component connect to one or more instances of another component. In such cases, we consider two types of connections: anonymous connections and named connections. The first type of connection, represented by a socket-like connector, as the connection between the front end and the web application components in the example, does forward requests from the client to either of the different server instances, without knowing the real identity of the actual replying server. This type of connection could be easily implemented by a load balancer element. In turn, a named connection, represented by an arrow, like the connection between the web application and the business application components in the example, describes a connection where the client knows all the server instance identities at all times, being able to choose the most appropriate instance to serve its requests.

4 CLOUD OPERATIONAL DETAILS

A language which looks like the one introduced in the previous section seems sufficient to define any scalable architecture with a reasonable detail level. However, it needs some additional declarations in order to be executable by a concrete cloud platform. Describing these features in the modeling language would probably bind the language to the specific platform requiring those features. That is why we present these operational features in a different section. In the following paragraphs we discuss some of the most important issues which should be considered in order to turn an abstract model into an executable model inside a particular platform. It is not an exhaustive enumeration, but the reader will get an idea about the extensions needed to that end.

The proposed modeling language allows to describe components and execution environments in an abstract fashion, explicitly avoiding dependencies on a concrete platform. Determining the components abstraction level, as well as the relationship between components and execution environments depends on the modeled system type and is the architect's responsibility. For example, in a corporate network model, a component may be a web server running within a particular operating system, whereas in a business web application, a component may be a website running within a web server which in turn runs inside a particular operating system. Therefore, the same element, the web server, may be considered a component or execution environment depending on the context. This flexibility, though desirable at the modeling level, introduces confusion and an extra degree of freedom very difficult to manage at the operational level. Therefore, each particular cloud platform should define a fixed set of commonly accepted execution environments within which the application components will be deployed, installed and executed.

On the other hand, the proposed model defines the application composition but it does not define its behavior. The behavior is provided by the actual application bits, which are typically broken down into several pieces, each one implementing a particular component of the overall system. Each component bits may be provided as a package or as a set of URLs from which the actual bits could be automatically downloaded and assembled. If URLs are supplied, the platform could provide automatic application upgrades when URL content updates are detected. Furthermore, if URL contents are properly packaged and the platform provides some means of automatically installing new software, then the deployment, installation and upgrade of applications could be fully automatized by the platform.

The combination of the application model and the software bits (or URLs) could be packaged inside a bundle which makes application distribution easy. If the bundle contains the software bits it will take up much space. However, if it does only contain URLs, the file could take just some kilobytes. This last approach, together with the automatic deployment, installation and upgrade facilities mentioned before could revolutionize today's system administration area. Following this line of thought installing a new full-fledged corporate network could be as simple as downloading from some place a properly configured bundle file and feed the platform with it, getting in return a scalable, high available and fully functional corporate network with all the corporate servers automatically deployed and ready to accept new requests. The platform will automatically download the required software, distribute it among several machines and install it on each of these machines. Then it will start the overall application and will make it available to the outside world. If any update is detected in the original URLs, the platform will automatically undertake the upgrade process, constantly maintaining an up-to-date system.

Yet another aspect to consider is the application component's lifecycle. The platform must publish a stable and robust lifecycle which will be enforced for all the deployed components. The platform will typically notify the components about lifecycle changes by sending events. To that end, the components will have to publish a specific interface, so that the platform may convey the appropriate information at all times. In order to avoid specific platform dependencies, such interface should be defined in some neutral manner, using open and standard protocols, such as HTTP or web services. These events will be sent to the particular instances of each component at runtime. For example, when a new instance of a component gets created, the platform should send an event to the rest of the instances, since they may need to perform some particular operations in order to accommodate the new instance. Similar considerations should be taken when an old instance gets destroyed or fails, or when the application must be updated, etc.

To conclude this section, we would like to suggest that in some cases the platform will have to provide special components to fill some gaps left by the scalable modeling language. These components will not be standard application components, but part of the platform instead, even though they could be made available as standard components. We will support this thesis with an example. As presented above, the scalable modeling language provides a way to define the resources that each component needs for its successful execution. One of these resources should be the storage. In this way, when a component instance gets created it receives the specified amount of storage, probably mounted as a local partition or similar. This storage may be used by the instance at will while it is active. When the instance is destroyed, the associated storage should also be destroyed, along with the rest of allocated instance's resources. With this approach, data could not persist across different system executions, or be shared by all the instances of the same component. To solve this type of shortcomings, the platform should provide special components which offer the required functionality. In the storage case, the special component could be modeled either as a stand-alone component which provides shared and persistent storage to a set of client components, or as a special feature of the platform which should be specified by using some sort of platform-dependent notation. Notice that the persistent storage can not be included as an out-of-the-box property in the proposed abstract modeling language, since depending on the platform this feature will be provided with a different system implementation, with diverse communications protocols and special properties regarding scalability, fault tolerance, efficiency and other aspects.

5 THE CLOUD PLATFORM

So far, we have explained our model driven approach, setting the model as the cornerstone of the whole process. Needless to say, such an executable model would not make sense without a full-fledged platform which supports the exposed methodology. Our approach demands a highly automated platform, capable of managing all the details of running scalable applications without requiring any user intervention. Modern platforms should go one step beyond though, and implement the concept of autonomic computing (Kephart and Chess, 2003), (Kramer and Magee, 2007), which aims to obtain self-managed systems. In the following paragraphs we present some of the features a model driven platform should implement.

First and foremost, the platform will cover a real datacenter. Whether the datacenter is homogeneous or heterogeneous, or if it must comply with certain network architecture or not is not relevant to this discussion and does only have to do with the platform versatility and applicability. The datacenter may be specified by using a particular computer network diagram, like Cisco-alike diagrams, which have become the de facto standard for describing network topologies. On the other hand, the datacenter physical infrastructure may also be dynamically discovered (Breitbart et al., 2004), (Jin et al., 2006), relieving the administrator of the burden of providing such information and ensuring up-to-date data, though some inaccuracies may arise. This autodiscovery facility is extremely useful when new machines are added to the pool, since no specific configuration protocols are required.

Secondly, the platform should understand perfectly the scalable modeling language, as well as the required extensions used for defining the scalable application properties. Once the application model has been submitted, the platform automatically analyzes its structure, properties and requirements and tries to conciliate them with the physical infrastructure. This process will typically result in a deployment plan for distributing the different elements of the application to the physical machines. Then most of the elements of the application will be virtualized and transferred to the planned locations. The devised plan must be dynamic, since the datacenter infrastructure conditions may change in the course of the application execution. For example, a machine may fail due to a power supply crash, or most of the datacenter machines may be busy because of occasional heavy demands coming from other applications running in the same cloud. Furthermore, when the concrete machines for deploying the application must be chosen, the deployment plan should take into account high availability, energy consumption and workload prediction issues, in addition to the application specific requirements.

Regarding high availability (Cristian, 1991), if a component must be replicated, the different instances will tend to be transferred to machines located in different fault domains. A fault domain includes all the machines and devices which are likely to fail at the same time, maybe because they belong to the same computer rack, or because they are powered by the same power line, or because they are located inside the same network segment. Also, the platform will take the required measures to ensure no single point of failure exists on the actual application layout, and if a component fails, it will undertake all the needed actions to recover the failed component and restart it transparently.

Also, while deploying the different components of the application, the platform should try to do its best on power consumption savings (Berl et al., 2010), (Beloglazov and Buyya, 2010), taking into account not only single machines consumptions, but network devices and cooling systems consumption as well. To that end, when a component must be transferred to a new machine, busy machines will typically be chosen first, since they are already working and consuming power and running a new component on them would consume less power than switching on a new machine. Other similar measures should be implemented for the sake of saving energy.

Finally, predicting workload behavior (Akaike, 1969) (Smith et al., 1998) could significantly improve the performance and response time of the deployed applications, helping to allocate resources in advance and reducing the delay inherent in the distribution and launching of new elements.

6 CONCLUSIONS

In this paper we have introduced a new methodology for designing cloud platforms, due to the difficulties found in current approaches. This methodology follows the model driven paradigm, where the whole process starts with a user-defined application model which is injected into the platform. To that end, we list the most important features that such a language should have, and we present a preview of a familiar modeling language which meets all requirements for modeling scalable architectures. We are actively working on the formal definition of such language and we will make the results available in the short term.

Our approach requires a platform capable of automatically handling almost every aspect of both the underlying infrastructure and the applications deployed on it. We collect our thinkings about all the characteristics that a cloud platform should take into consideration in order to support our fully automatized model driven vision.

ACKNOWLEDGEMENTS

This paper has been partially supported by EU FEDER and Spanish MICINN under research grant TIN2009-14460-C03-01.

REFERENCES

Akaike, H. (1969). Fitting autoregressive models for prediction. *Annals of the Institute of Statistical Mathematics*, 21:243–247.

Amazon (2011). Amazon Web Services web site. http://aws.amazon.com/.

Azure (2011). Microsoft Azure web site. http://www.microsoft.com/windowsazure/.

Beloglazov, A. and Buyya, R. (2010). Energy efficient allocation of virtual machines in cloud data centers. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:577–578.

Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M. Q., and Pentikousis, K. (2010). Energy-Efficient Cloud Computing. *The Computer Journal*, 53(7):1045–1051.

Booch, G., Rumbaugh, J., and Jacobson, I. (2005). *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.

Breitbart, Y., Garofalakis, M., Jai, B., Martin, C., Rastogi, R., and Silberschatz, A. (2004). Topology discovery in heterogeneous IP networks: the NetInventory system. *Networking, IEEE/ACM Transactions on*, 12(3):401–414.

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78.

DeMarco, T. (1979). *Structured Analysis and System Specification*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Foster, I., Zhao, Y., Raicu, I., and Lu, S. (2009). Cloud Computing and Grid Computing 360-Degree Compared. *ArXiv e-prints*, 901.

Google (2011). Google App Engine web site. http://code.google.com/appengine/.

Jin, X., p. Ken Yiu, W., Member, S., Member, S., h. Gary Chan, S., Member, S., and Wang, Y. (2006). Network topology inference based on end-to-end measurements. *IEEE JSAC*, 24:2182–2195.

Kent, S. (2002). Model driven engineering. In Butler, M., Petre, L., and Sere, K., editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer Berlin / Heidelberg.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36:41–50.

Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kramer, J. and Magee, J. (2007). Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, Washington, DC, USA. IEEE Computer Society.

Lenk, A., Klems, M., Nimis, J., Tai, S., and Sandholm, T. (2009). What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, Washington, DC, USA. IEEE Computer Society.

MDA (2011). OMG Model Driven Architecture web site. http://www.omg.org/mda/.

QVT (2011). OMG Query, View, Transformation spec. http://www.omg.org/spec/QVT/1.0/.

Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, 39(2).

Smith, W., Foster, I., and Taylor, V. (1998). Predicting application run times using historical information. In Feitelson, D. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 122–142. Springer Berlin / Heidelberg.

UML (2011). OMG Unified Modeling Language spec. http://www.omg.org/spec/UML/.

Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. (2008). A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39:50–55.