Autoconfiguration of Enterprise-class Application Deployment in Virtualized Infrastructure Using OVF Activation Mechanisms

Fermín Galán, Miguel Gómez, Fernando de la Iglesia, Ignacio Blasco, Daniel Morán
Cloud Infrastructure Area – Product Development and Innovation Unit
Telefónica Digital
Madrid, Spain
{fermin, miguelg, fim, ibl, dmj}@tid.es

Abstract— **IT-based services existing today, such as the ones supporting e-commerce systems or corporate applications, demand complex architectures to address enterprise-class requirements (high availability, vast user demand, etc.). In particular, most enterprise-class applications are multi-tiered and multi-node, i.e. composed of many independent systems with complex relationships among them. In recent years, virtualization technologies have brought many advantages to enterprise-class application implementation, such as cost consolidation and ease of management. However, even when using virtualization, the configuration operations associated to the deployment of enterprise-class applications are still a challenging task since they constitute a mostly manual, time-consuming and error prone process. In this paper we propose a solution to that problem based on the automation of that procedure by means of the OVF activation mechanism. Our work focuses on a practical case, which has been used to assess the feasibility of our solution and to extract valuable lessons that we expose as part of the article.**

Keywords-component; enterprise-class applications; virtualization; multi-tier architectures; OVF

I. Introduction

Most of the current enterprise-class applications can be categorized as *transactional applications*, meaning applications based on the client-server paradigm where the user interacts as a client with the server in order to obtain a specific result. This interaction is what we call a *transaction*. These transactions can be very simple (e.g., requesting and obtaining a web page or a static image) or very complex (e.g., an online shopping process where the user selects several products and pays for them using a credit card, interacting with a data base that manages the product catalogue and the payment system). An especially interesting subset of transactional applications is *web applications*, where users interact with the server using a web browser (in the context of the Internet or in a corporate intranet) through web technologies such as HTTP.

The transactional applications that we typically find in corporate environments present a multi-tier architecture [1]. These architectures are based in creating the application as a set of components, each of them specialized in a task, and structured in several tiers that interact with each other in order to solve the transactions requested by the users. Typically, three tiers are considered:

- Presentation tier (or front-end). It is the most external application tier, in charge of the direct interaction with the users. It receives the requests from them and progresses them to the application logic tier. It also receives the results of the transactions and presents them back to the users.
- Application logic tier (or back-end). It is the tier that implements the transactional processes themselves, usually in direct interaction with a persistence tier where the data needed for the processes reside.
- Persistence tier. It is the tier where the data needed by the application are stored, typically using a database.

The technologies used in each tier are different and suited for the functions the tier has to execute. Taking an on-line shopping web service as example, we could have an Apache web server acting as the front-end (to present users a form to send their purchase orders and show the web page with the results), a JBoss application server in the back-end where the application logic would be installed (in charge of processing the data filled-in by the users in the form, querying for product availability, generating the bill, etc.) and finally a MySQL database in the persistence tier (storing the information about the products, prices, availability, users' purchases, etc.).

Although it is possible to place all these tiers in the same system, in enterprise environments where the amount of application users can be very high the common use is to separate them in different nodes. Moreover, each tier can be scaled horizontally with different nodes in order to increase the throughput of the application. To distribute the workload between the different nodes that compose the tier, a *Load Balancer* (LB) is typically used as the single entry point to the tier. In addition to workload allocation across the different nodes, load balancers can implement additional features such as session stickiness, node health monitoring, etc.

As can be deduced from the description above, the deployment of a transactional application with several nodes per tier and load balancing elements typically comprises a significant amount of systems. However, the adoption of

978-3-901882-48-7 ©2012 IFIP

virtualization technologies [2] in the last years has allowed to increase system utilization by consolidating several logical nodes in one physical node, thus making possible to keep the amount of physical systems low even in the case of application architectures like the one described above. Virtualization also improves the operation and management of the logical systems (virtual machine migration for physical system maintenance, etc.).

Another key advantage introduced by virtualization is the possibility for ISVs to package and distribute their applications in the form of *virtual appliances* (VAs). A VA can be defined as a pre-configured software stack comprising one or more virtual machines (or virtual systems) that provide a self-contained service. However, a new problem space appears: for VAs to be bundled once and deployed in any possible target environment, a mechanism to customize and adapt the VA's elements to peculiarities of such environment and the desired deployment configuration needs to be provided.

This paper focuses on that problem space, presenting how complex enterprise-class transactional applications can be made deployment-agnostic by means of the parameterization mechanisms offered by the Open Virtualization Format (OVF) standardized by the Distributed Management Task Force (DMTF), and then customized at deployment time (either automatically or by requesting user input) by means of the OVF activation mechanism. We would like to stress that the main goal of this paper is to analyze the practical use of OVF to automate the deployment of a key type of VAs in corporations (i.e. enterprise-class applications) taking into account the characteristics of such systems. Thus, rather than in proposing new enhancements to OVF, the contribution and novelty of this paper lie in the lessons learnt from that process and the shortcomings identified. Additionally, part of those shortcomings could be solved by future contributions to OVF, as identified in the conclusions and future work section.

In order to address this topic, the remainder of this paper is structured as follows. First, section II introduces the problem to be addressed, presenting the main challenges in VA parameterization and customization. Next, section III describes the OVF activation mechanism. Section IV exposes a use case for the OVF activation mechanism, detailing the experiments undertaken and the results obtained. Finally, section V introduces the main lessons learnt from the study and experiments and section VI outlines the main conclusions and next steps of this work.

II. Problem Statement

As described in the previous section, the deployment of an enterprise-class transactional application typically comprises a large number of systems. Normally, a minimum of nine nodes is required: two nodes on each of the three tiers (i.e., presentation, application and persistence) for high-availability purposes plus a load balancer per tier.

In a traditional environment, such deployment implies the physical installation of the nodes, their interconnection with the local area network and the storage network, operating system installation and configuration and, finally, the deployment and configuration of the application-specific software stack on each node. It is not difficult to determine that this manual process is very time-consuming and error prone, normally requiring the intervention of different groups (network and storage administrators, data base administrators, system administrators, application vendors, etc.).

The introduction of virtualization technologies reduces significantly the complexity of that deployment process. Firstly, virtualizing the application nodes as virtual machines makes the deployment requirements much more flexible, allowing the deployment of the application in a lower number of physical nodes while retaining all the high-availability and load sharing features found in physical deployments. In a second evolution stage of virtualized application deployment, applications are directly bundled as Virtual Appliances, containing all the required nodes, their configuration and, in a certain way, the relationships among nodes (e.g., the network segments where the nodes are to be deployed).

Although the advantages introduced by virtualization technologies are undeniable, plain virtualization is still not capable of supporting the "bundle once, deploy anywhere" paradigm. On one hand, there are many different virtualization technologies in the market, and thus choosing one of them to bundle the application would limit its deployment to virtualized computing platforms of that sort, being that a limitation of the virtualization solution and not of the application itself. On the other hand, applications need to be deployed according to the requirements of the target virtualization environment, mainly in three different areas:

- VM resource allocation. Fundamentally CPU and RAM requirements.
- Network configuration. Depending on the network topology and addressing of the target environment, the different virtual interface cards of each node need to be configured.
- Application-level parameters. Complex enterprise-class transactional applications typically support several configurations to cater for different deployment and/or usage scenarios and to optimize application performance. Thus, the application-level configuration of each application node needs to be adapted, typically by acting on a series of configuration files.

Virtual Appliances failing to address these three issues will still constitute an improvement in deployment terms when compared to physical installation or individual node deployment in virtualized environments, but they will still require the complex and cumbersome customization procedures mentioned earlier in this section.

Fortunately the IT industry has created a standard that helps solving these problems: standardize the packaging and deployment of multi-tier complex applications and ease and automate as far as possible the deployment process in order to minimize or even eliminate manual intervention and thus reduce the possibility of error. This standard is the Open Virtualization Format (OVF) from the Distributed Management Task Force (DMTF) standardization organization, which will be introduced in the following section.

III. OVF Activation Mechanism

A. Introduction

OVF [3] is a vendor and platform neutral portable format to package one or more virtual machines composing a VA in a single *OVF package*. The main elements of the OVF package are an *OVF descriptor* (a XML file enclosing information regarding the VA and its constituting virtual machines or *Virtual Systems* -VSs- in OVF parlance, conforming to several XML Schemas included as normative documents in the specification) and the set of resources used by the VA (virtual disks, ISO images, internationalization resources, etc.). The OVF package can either be a single OVA file (in TAR format) or a set of separate files "linked" by the OVF descriptor.

The OVF descriptor is composed of three main parts: a description of the files included in the VA, as `<Reference>` tags for virtual disks, ISO images, internationalization resources, etc.; metadata about all virtual systems contained in the package (e.g., `<DiskSection>` describes virtual disks data and `<NetworkSection>` specifies the logical network interconnecting the systems in the VA); and a description of the different VSs, in `<VirtualSystem>` tags. OVF allows specifying the virtualization technology (e.g., "kvm" or "xen-3") and describing the virtual hardware required by each VS (`<VirtualHardwareSection>`) in terms of CPU, memory and hardware devices (including disks and network interfaces) to be provided by the underlying hypervisor running the VS. Additional information can be provided in the form of metadata sections (as the `<ProductSection>` element, described in next subsection).

Finally, VSs can be grouped in collections (`<VirtualSystemCollection>` or VSC) which may be used to group common information regarding several VSs. Within the collection, the `<StartupSection>` defines the virtual system boot sequence.

B. OVF Activation Mechanism

By *activation* we mean the process to configure on its first boot each one of the VSs in the VA (included in the OVF package as "frozen" images) for the context in which that VS will run. In the current version of OVF the mechanism for activating the VA is based on communicating to each of the VS that sum up the VA the parameters to be configured locally. The values to be adopted on each parameter are passed on from the deployment platform to the VS.

OVF defines a specific section in the OVF descriptor called `<ProductSection>` that is defined at VSC or VS level and where the parameters to be communicated to the VSs shall be placed in the form of key-value entries. The definition of "product" is quite vague in OVF. It could be the operating system, the middleware, the application or anything else that is inside the VS. The structure of the `<ProductSection>` includes a list of configuration properties in the form of key (*ovf:key*) and value (*ovf:value*) attributes. Depending on the way the value of the properties is set, two different types of configuration properties are possible: values fixed at OVF package build time that cannot be modified at deployment time (*ovf:userConfigurable=*
="false"), or values that can be fixed or modified at deployment time (*ovf:userConfigurable="true"*). In the latter case, the value included in the *ovf:value* attribute indicates the default value fixed at build time. The properties can take the value of a property in the VSC that contains the VS/VSC (i.e., the "parent" VSC) via macro *${property_name_in_the_parent}*.

A quite simple example of a `<ProductSection>` would be the following:

```
<ProductSection ovf:class="load_balancer">
  <Property ovf:key="port" ovf:type="uint16"
    ovf:value="80" ovf:userConfigurable="false"/>
  <Property ovf:key="mode" ovf:type="string"
    ovf:value="http" ovf:userConfigurable="true"/>
</ProductSection>
```

Considering the `<ProductSection>` presented above, the corresponding OVF activation process workflow is shown in in Figure 1. First, when deploying the VA, the deployment platform prompts the user or deployment system (step 1) to introduce the values for the keys that are marked as *ovf:userConfigurable="true"*. Next, an XML document (called *OVF Environment*) is generated containing the key and values with the following structure (step 2). Note that the key name is prefixed by the *ovf:class* attribute.

```
<Property ovf:key="load_balancer.port"
    ovf:value="80"/>
<Property ovf:key="load_balancer.mode"
    ovf:value="http"/>
```

The OVF Environment generated shall be given to the VS via the transport specified in the OVF descriptor by the *ovf:transport* attribute of the `<VirtualHardwareSection>` at VS level (step 3 and 4). The specification sets that any platform must support at least the "iso" transport, which consists of including a virtual CDROM in the VSs, creating an ISO image in which the OVF Environment file is included and then attaching the ISO image in the CDROM of the VS at boot time.

Once the OVF Environment file is made accessible to the

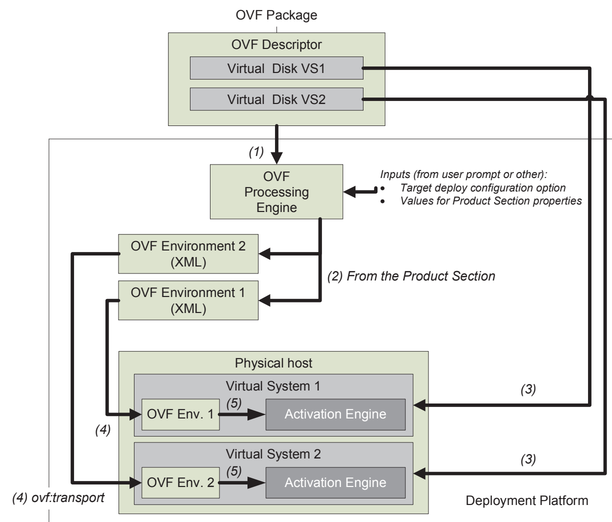

Figure 1. Sample OVF activation workflow

VS any software inside it can make use of the information contained on it during the first boot in order to activate or configure any product, from the operating system to any element of the application software stack (step 5). Typically, the software in charge of activating the products is known as Activation Engine (AE).

C. *Related Work*

Some tools address a problem space similar to that of the OVF activation mechanism, such as Puppet [4] or Chef [5]. These tools aim at automatic configuration of the managed systems, e.g. virtual machines of enterprise-class applications. However, they need a piece of software (i.e. an agent) already running and network-accessible in the managed system. Thus, OVF activation and these automatic configuration tools are complementary, the first for the automatic configuration at deployment stage (e.g. network configuration, bringing up the Puppet/Chef agent, etc.), and the second for post-deployment configuration management operations.

Closer to OVF activation is the IBM Activation Engine [6], which also addresses automatic configuration at deployment time. OVF provides the same functionality but in a more flexible and standardized way, which is a definitive advantage in an environment in which VA builders, virtualization providers and VA consumers are independent agents. In fact, the IBM Activation Engine (proposed 5 years ago) can be seen as an ad-hoc script-based solution, the main concepts of which have been incorporated to the OVF standard.

IV. USE CASE

In order to evaluate the feasibility of auto-configuring enterprise-class application deployment in virtualized infrastructure using the OVF activation mechanism, we have conducted a practical use case. In this use case, we deploy an OVF package containing a benchmark product resembling an enterprise-class application in an OVF-compliant virtualization platform. The purpose of this experiment is to asses that the approach discussed in this paper is valid and flexible enough to cope with different deployment environments (i.e. different network topologies existing in the virtualization platform) and configurations.

A. *Target Application*

As target application we use RUBiS [7], which is a widely used benchmark to experiment with multi-tier applications. RUBiS implements a sample online auction service and, among the different tier configurations supported, we have chosen the alternatives that lead to higher setup complexity, in order to resemble enterprise-class applications in the most accurate way. Firstly, we consider three separate tiers (i.e., presentation, application logic and storage). Secondly, we consider several nodes on each tier (detail follows below). Accordingly, LBs have to be included as part of the application.

All nodes are based on Debian GNU/Linux 6.0 with the following particularities:

- The presentation tier is based on Apache web server 2.2.16 using mod_jk module and containing RUBiS static content. There are two stateless nodes in this tier.
- The application logic tier is based on JBoss EAP 5.1 containing the servlets (encapsulated in a .war package) that implement the RUBiS functionality. There are two nodes in this tier, also stateless.
- The storage tier is based on MySQL Cluster 7.1. The cluster is composed of a single Management Node (DBM) for cluster synchronization, and two DB worker nodes, each one containing a logical API Node

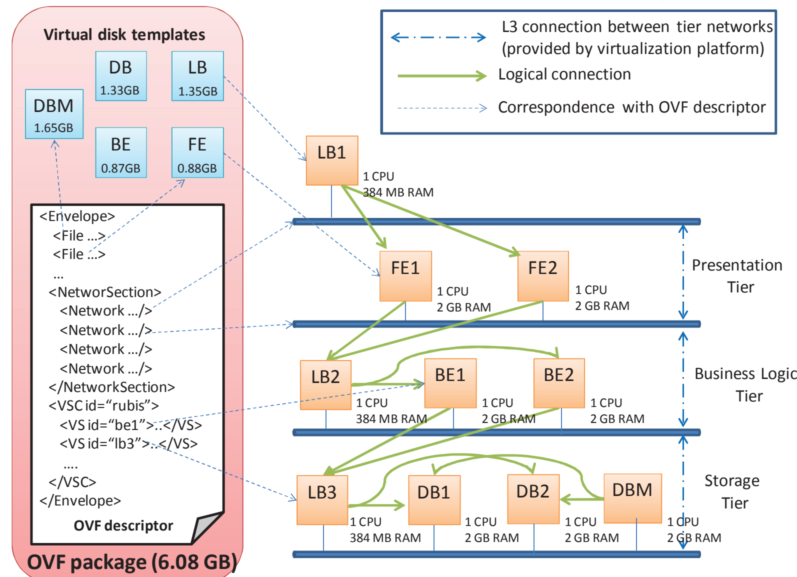

Figure 2. Use case application logical architecture (right) and outline of the corresponding OVF package (left)

and a logical Data Node. The data model is replicated in both Data Nodes, and clients (i.e. application logic nodes) can read and write to any instance.

- LBs are based on HAProxy 1.3.15, which is able to balance TCP connections in any port (so suitable for HTTP or MySQL traffic) and includes keepalive features (so it can detect dynamically if the balanced nodes are up and running). There is one load balancer per tier, all of them stateless. Load balancers work in one-arm configuration (i.e. the load balancer has a single network interface and source NAT is applied to requests to force return traffic to traverse the LB).

The application is encapsulated within an OVF package. This package is composed of 5 files providing the virtual disk for the different instances (.vmdk files are used, as vSphere was chosen as virtualization platform, as will be described in Section IV.B) and the OVF descriptor file containing metadata information. We could have packaged all the files in a single OVA file, but considering the large size of the virtual disk files we preferred the option of keeping a set of files in order to have greater flexibility. All virtual disk nodes include an AE capable of processing OVF Environments, the details of which are provided in Section IV.D.

The service itself is represented by a VSC and the different nodes by VS. The main information elements in the OVF descriptor are summarized below:

- The 4 networks that interconnect the different service nodes, implementing the topology as shown in Figure 2. Note that they are logical networks so, depending on the deployment environment, several (or all) of them could be "collapsed" into the same real network at virtualization platform level.
- The hardware resources that each node needs. All of the nodes have the same resource requirements: 1 vCPU and 2GBs of RAM (except the load balancers, which did not require so much capacity to perform their duty, so they only use 384MBs of RAM each).
- `<StartupSection>`, specifying the order in which instances have to be powered on. This order is defined as the inverse of the tier dependency relationship.
- `<ProductSection>`s (as described in Section III.B), containing information to configure each node. Given the importance of this information in our use case, it is described in detail in Section IV.C, along with how each configuration element is processed.

In is worth noting that our RUBiS deployment results in a 10-nodes OVF package with an overall footprint of 10 vCPUs and 15.12 GBs of RAM. Thus, this deployment involves enough size and complexity to be considered a representative case of enterprise-class application.

B. *Virtualization Platform*

We have used VMware vSphere v4.1 as virtualization platform. The testbed setup is composed of a vCenter server which manages a physical host ("ESXi" in VMware parlance) with the following hardware profile: PowerEdge 1950, 4 Intel Xeon E5310 CPU @ 1.6GHz, 4GB RAM, 150GB HD.

In order to be able to test different alternatives regarding network configuration on deployment, we have configured four virtual networks within the ESXi and an additional VM acting as router, implemented with a lightweight GNU/Linux system (Figure 3). We have assigned a /24 subnet to each one of the networks (172.16.1.0/24, 172.16.2.0/24 and 172.16.3.0/24), except the external one which uses a /25 range (10.1.0.0/25).

The requirements for the virtualization platform in this experiment are that it has to be able to import OVF packages, generate OVF Environment correctly (in the format specified in the OVF standard) and present them to the VMs in a ISO file that is mounted as a CDROM at first boot time. All these features are implemented by the virtual platform of choice as far as we have checked in our experiments.

C. *Product Sections*

The OVF descriptor includes the following `<ProductSection>`s (outlined in Figure 4):

- An *ovf:class="rubis",* at VSC level which addresses the configuration of the RUBiS service as a whole. It includes the following properties: IP addressing (in CIDR notation) for each one of the 10 VS, gateway IP for each one of the 4 networks and 2 general properties for specifying the DNS server and a DNS domain. In sum, it contains 16 properties.
- An *ovf:class="network_config",* at VS level which includes the properties to configure the network of the given VS. In particular, it includes properties for that node's IP address and GW. It appears in every VS, as all them need to configure the network.
- An *ovf:class="load_balancer",* at VS level which includes properties to configure the load balancer. In particular, it includes up to 6 properties to configure its different options (e.g. port, protocol type, etc.) and 2

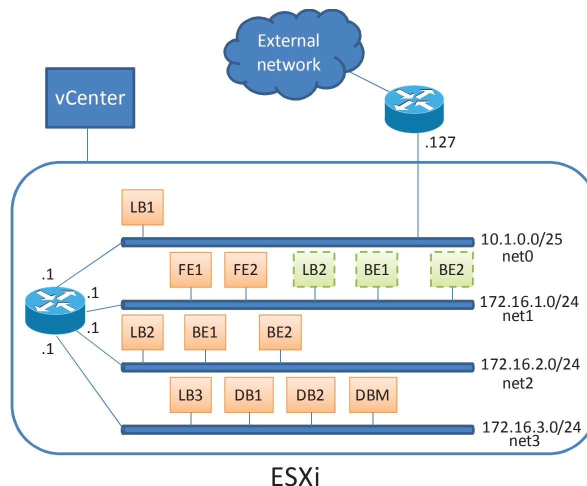

Figure 3.Virtualization platform and example of different application logic tier deployment alternatives (dashed-line and continuous line)

additional ones to specify the IP addresses of the two balanced nodes. Note that for the chosen setup all load balancers are balancing the same number of nodes (i.e. two). Otherwise, we would have needed to choose a different approach (e.g. a single property with space-separate values, each one the IP of a balanced node). Only the three VS implementing the load balancer function incorporate this section.

Among the different sections, only the *"rubis"* one has *ovf:userConfigurable* properties. The properties in *"network_config"* are derived from the IP and gateway properties in *"rubis"* using the *${..}* macro (see Section III.B). The properties in *"load_balancer"* are also either derived from the ones in *"rubis"* (IP of the balanced nodes) or defined in the OVF itself (configuration options that depend exclusively of the LB tier, e.g. *port* is 80 for presentation tier LB and 5000 for storage tier LB). In other words, the user has to provide 16 configuration values, all others (36 properties) are either already defined within the OVF descriptor or derived from the ones provided by the user.

D. Activation Engines

An AE is installed in the disk images of each virtual machine, composed of three main components:

- A service script (*/etc/init.d/runOvfAe*) which is configured in the system init (*/etc/rcS.d/*) to be run at boot time at the proper moment, i.e. after the mountall service (because otherwise the AE cannot write in the VM's filesytem) but before the network init service starts (because the AE has to generate the proper network configuration required by that service). It mounts the CDROM containing the OVF Environment and invokes the *ConfigFileCustomizer* program for each one of the files found in the templates directory, putting the final configuration file produced as output in the right place (usually within the */etc* directory).
- The *ConfigFileCustomizer* program (written in Java and installed in the virtual disk as a JAR package) takes a given template configuration file and the OVF Environment as input. The template configuration file includes text macros in the form *{$Environment.product_section.property}* which are interpreted as keys in the OVF Environment and replaced by the corresponding value, thus generating the final configuration file as output. Some modifiers can be applied to macros, e.g. *{$Environment…@ip}* to get only the IP part ("A.B.C.D") of an IP in CIDR notation.
- Template directory, where are located all the template configuration files that apply to the particular VS. Figure 4 shows which templates are used in each node.

E. Running the Experiment

Considering the use case described earlier in this section, a typical experimentation cycle resembling the deployment of an enterprise-class application in a real world situation would be as follows. The process starts with the activation-capable OVF package containing the target application. This package could come from different sources, e.g. an internal software development department, bought from an ISV vendor which distributes his software packages as VAs, etc.

The user (e.g. IT staff) accesses the virtualization platform from a management console and invokes the "Deploy OVF package" feature. Among other platform specific information that needs to be provided (e.g., ESXi node that will host the VMs, etc.), the platform asks the user for the information that will be used in the application autoconfiguration process,

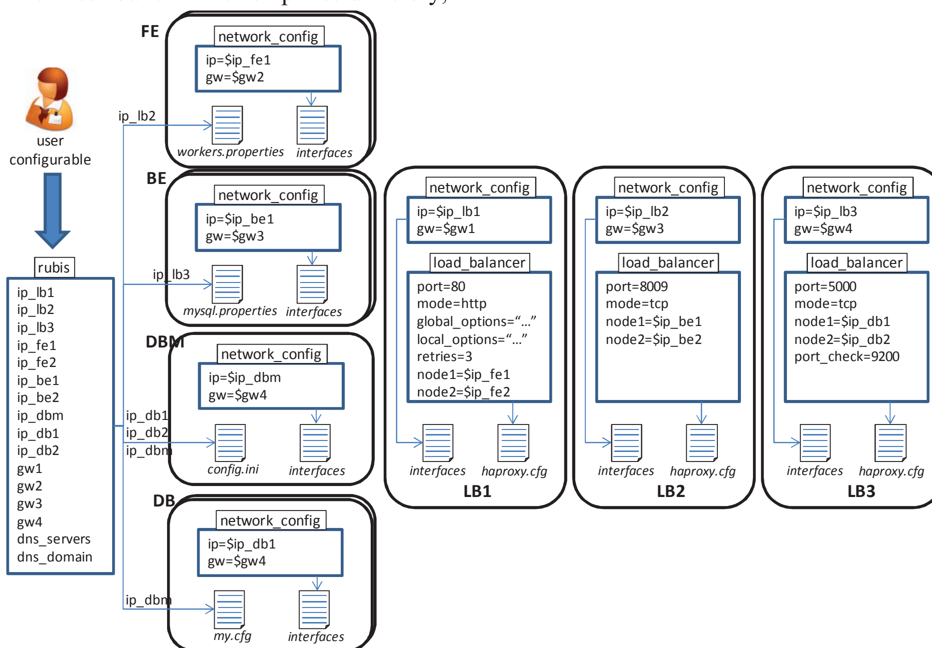

Figure 4. `<ProductSection>` structure and derived configuration files (simplified)

usually through a user friendly GUI dialog. In particular, in our RUBiS case, how the four logical networks (in the OVF package) are mapped to the real networks implemented by the virtualization platform and the different values for the 16 properties in *"rubis"* `<ProductSection>`.

The steps above are the only ones that the user has to perform manually. Once all the information has been gathered, the remainder of the process is performed automatically. The OVF package contents are transferred from the management console to the virtualization platform. In the case of RUBiS, that implies transferring approximately 6.08 GBs of disk images, which can take a long time depending on the bandwidth available between the management console and the virtualization platform. In our tests (using our corporate intranet), this time used to take around 2.5 hours, which could be seen as a long time but which is actually negligible compared to the usual deployment timeframe for an enterprise-class application (months). In our case, a single physical host was used, so all the VMs were deployed on it; in the case of a multi-host infrastructure, the VM distribution across hosts would depend on the policy configured on the vCenter server.

Once the virtualization platform receives all the disk templates, it instantiates the corresponding VMs as described in the OVF descriptor and connects them to the different networks as specified by the user. Then, the newly deployed VMs are powered on (in the order specified by the `<StartupSection>`) and provided with a CDROM containing the OVF Environment file (as described in Section III.B). The AE in each VM is started as part of the boot process and the VM gets properly configured.

We have repeated this process several times (more than ten times with a success ratio of 100%) with different topologies (considering the networks shown in Figure 3, from all nodes running in "net0" to each tier running in a separate network). In all cases, once the deployment process ends, the RUBiS service is up and running, accessible at the web port of the presentation tier load balancer. Thus, our tests assess that the OVF-based activation mechanism is a valid solution for the deployment and autoconfiguration of enterprise-class applications.

V. LESSONS LEARNT

Several remarkable conclusions can be derived from the experiments conducted in our use case:

- The separation between logical networks (at OVF descriptor level) and actual networks (at virtualization platform level) and the ability to freely associate the formers to the latters at deployment time, allows a great flexibility, so the same OVF package can adapt to several network setups in the virtualization infrastructure of an organization.
- Regarding the AE in our use case, the *ConfigFileCustomizer* program is the same in every VS. The differences are in the particular template configuration files being used and in the *runOvfAe* script manipulating them. Thus, we have achieved a quite general solution for implementing AEs, avoiding to make a specific development for each one of the 5 types of VS. However, we have taken advantage of the fact that every aspect in RUBiS is configured through text files, which could not be the case in other applications (e.g. registering a product license, etc.) and that would require more specific AE programs.
- In our AE, the *runOvfAe* script runs always when the CDROM is mounted successfully and an OVF Environment is found (otherwise it silently exits without doing anything). That means that the activation process could be unnecessarily launched every time the VM is rebooted. In the case of RUBiS, that would not be a problem (as the configuration file generation process is idempotent), but could be problematic for other enterprise-class applications. If that case, the AE should "remove itself" as a final step after activation.
- Our deployment is based in a fixed number of nodes in each tier (see the detail in Figure 2). However, multi-tier applications enable the possibility of adding or removing nodes to each tier after the initial deployment to cope with variations in application workload. Although we have focused on the initial application deployment and we have not addressed scale up/down scenarios in depth, our use case could also cope with that. In order to scale up, we could define a "delta" OVF package for each tier, composed by the VS template of that tier with the subset of properties needed for that node. In order to manually scale up a given tier, we should just deploy the "delta" OVF package corresponding to that tier, which is autoconfigured in the same way the initial deployment was (e.g. user prompting, AE activation, etc.). Manual scaling down is simpler and not related with OVF at all, consisting in just removing the VM from the virtual platform. Automatic scaling up/down (as described in [8]) could be also implemented, although in this case how to obtain *ovf:userConfigurable* properties (that inherently needs user interaction) has to be solved.
- Although our use case has used a "classical" virtualization platform (as vSphere is) as deployment platform, Infrastructure-as-a-Service (IaaS) cloud is currently a strong trend to which our approach could be used. As far as the IaaS cloud platform allows OVF-based deployment with a full support for activation (e.g. a way of introducing *ovf:userConfigurable* properties, OVF Environment transport to VM, etc.) our approach will work. OVF has been proposed some time ago for application deployment in clouds [9] and, in fact, is considered in major IaaS cloud management interfaces such as VMware vCloud [10], OCG's OCCI [11] and DMTF's CIMI [12].
- OVF states that virtual disk file references (*ovf:href* attribute in `<File>` at `<References>`) must be URIs (see [3]). Typically, these URIs point to files included in the OVA package and, in applications like the one exposed here, these can be several virtual disk files of several GBs each, thus making difficult the distribution of the application. Because it is already included in the standard and in order to ease application distribution,

Internet/intranet HTTP URLs could be used as file references. With the advent of cloud storage services (see for example [13]), virtual disk files can be kept in the cloud storage services. This enables distributing the OVF descriptor only and letting the deployment platform retrieve the disk image files at deployment time. SNIA has recently released the Cloud Data Management Interface (CDMI) [14] specification with the aim of providing a standard interaction mechanism for cloud storage services. By using CDMI a standard URI could be specified for the *ovf:href* attribute in order to make application distribution more portable, in addition to the packaging and deployment benefits already introduced by OVF.

Additionally, the following OVF shortcomings were identified in the course of our experiments:

- Each node within the same tier is represented by an independent VS element in the OVF descriptor's XML, although they are using the same configuration. This constitutes a problem from the point of view of maintainability (a modification in one VS has to be manually synced in the other peer-VS of that tier). Fortunately, the DMTF has addressed this issue, solved with the `<ScalingSection>` in OVF 2.0 [15], which allows representing homogeneous systems with the same prototypical VS.

- OVF property typing (provided by the *ovf:type* attribute in `<Property>` at `<ProductSection>`) is quite limited, restricted to basic types (e.g. strings, integers, etc.). Thus, it is difficult to specify for example that a given property has to be a string in CIDR notation so that it can be enforced by the deployment platform at user prompting time. Note that, although the AE could check the format, if an error is detected at AE activation time the only action that can be adopted is to raise an error (since the user cannot interact with the platform at this stage). Thus, the introduction of a more powerful *ovf:type* typing (e.g. based in regular expressions) would be highly beneficial.

- Our case fits very well when static addressing is used, since the user can specify the same set of IP addresses to be used both for configuring the VMs at OS level and the RUBiS components. The use of IP autoconfiguration (e.g., DHCP) removes the need of configuring IP addresses at OS level (so simplifying the AEs) but introduces the problem that the user can not typically know which IP addresses will be assigned to each VS, so he/she cannot provide the right IP addresses at prompting time to configure the RUBiS application. To solve this problem, a more sophisticated activation mechanism than the one described in Section III.B would be required, so some properties can be "captured" by certain AEs at deployment time (e.g. the AE running in the tier 3 LB should capture the IP address obtained through DHCP) and passed on to other AEs (e.g. the AEs running in the tier 2 nodes, requiring that IP address to connect to tier 3). Note that the current OVF 1.1 transport mechanism is unidirectional (from deployment platform to VS). In order to cover this AE-AE communication, specific protocols would have to be defined (probably with the deployment platform acting as a mediator for these communications).

VI. Conclusions and Future Work

The deployment of large and multi-tiered enterprise-class applications is *a complex* and difficult process. In our paper, we have proposed a solution based on the OVF activation mechanism that enables the automatic deployment of such applications. Therefore, the work of IT staff in charge of such deployments is clearly eased. We have shown the feasibility of our approach using a real world deployment platform (i.e. not a prototype or experimentation platform) with a benchmark muti-tiered application (RUBiS) composed of 10 nodes.

We are considering several future work lines to continue our work. First, take into account other deployment platforms apart from VMware's ones, to analyze if their OVF support level matches the requirements of our use case. This analysis could include IaaS clouds. Second, to expand the use case with scaling up/down operations on the application tiers. Third, to address the problem of autoconfiguration when some parameters have to be taken from the application deployment environment (typi*cally, I*P addresses dynamically assigned using DHCP).

References

[1] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, A. Tantawi, "An Analytical Model for Multi-tier Internet Services and Its Applications", ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems, pp. 291-302, Alberta (Canada), June 2005.

[2] R. Figueiredo, P. A. Dinda, J. Fortes, "Resource Virtualization Reinassance", IEEE Computer, vol 38(5), pp. 28-31, May 2005.

[3] DMTF, "Open Virtualization Format (OVF)", Specification DSP0243 1.1.0, January 2010.

[4] Puppet, http://puppetlabs.com

[5] Chef, http://www.opscode.com/chef

[6] L. He, *et al.*. "Automating deployment and activation of virtual images", Technical Report 0708, IBM WebSphere Journal, 2007

[7] RUBiS, http://rubis.ow2.org/

[8] L. Rodero-Merino, *et al.*, "From Infrastructure Delivery to Service Management in Clouds", Future Generation Computer Systems", vol. 26(8), pp. 1226-1240, October 2010.

[9] F. Galán, *et al.*, "Service Specification in Cloud Environments Based on Extensions to Open Standards", Fourth International Conference on COMmunication System softWAre and middlewaRE (COMSWARE 2009), June 2009, Dublin (Ireland).

[10] VMware, "vCloud API specification v1.5", http://www.vmware.com/pdf/vcd_15_api_spec.pdf, 2011.

[11] OGF, "Open Cloud Computing Interface - Core", GFD.183, April 2011.

[12] DMTF, "Open Virtualization Format (OVF)", Specification DSP0243 2.0.0c Work-in-progress (WIP), June 2012.

[13] Amazon Simple Storage Service (Amazon S3), http://aws.amazon.com/es/s3/

[14] SNIA, "Cloud Data Management Interface (CDMI) v1.0.1", SNIA Technical Position, September 2011.

[15] DMTF, "Cloud Infrastructure Management Interface (CIMI)", Specification DSP0263 1.0.0e Work-in-progress (WIP), July 2012.