Automated Dependability Planning in Virtualised Information System

Marco D. Aime
m.aime@polito.it

Paolo Carlo Pomi
paolo.pomi@polito.it

Marco Vallini
marco.vallini@polito.it

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, 10149 Torino, Italy

Abstract

Virtualisation technologies widely affect information dependability practices, and suggest novel approaches for dependable system configuration. We analyse how to exploit these peculiarities within a tool for semi-automated configuration of virtual information systems. We present the general architecture of the tool, which is the product of previous work focused on traditional information systems, and discuss how to update its configuration strategies when moving to virtual environments. We present the stepwise process that, starting from the model of the target service, computes a set of configuration plans that determine which virtual machines should be deployed, their internal configuration, and their intended interactions. Our tool also generates re-configuration templates to switch between different configuration plans in case of dependability problems or changed requirements. Actually, reaction plans are heavily affected by virtualisation technologies, which permit fast re-allocation and reconfiguration of virtual machines. We finally discuss the integration of our process with virtual machine management systems, in order to perform configuration and re-configuration in fully automated way.

Keywords: automatic system configuration; dependability ontology; virtualization.

1 Introduction

Virtualisation technologies are changing traditional architecture in data center environments [8, 17, 26]. They widely affect the trade-off between costs and benefits of standard dependability mechanisms and suggest the introduction of novel approaches. In previous work [6] we addressed the policy-driven, automatic configuration of information systems, taking care of dependability requirements at the business service level. [6] describes process, algorithms, and tools for semi-automatic generation of dependability configurations for information systems not relying on modern virtualisation technologies. In [7], we have first analysed how to adapt our approach to virtualised information systems. In this paper, we investigate how virtualisation changes the best practices of dependability planning, and how we have updated our configuration framework accordingly. In particular, after a quick reference to virtualisation concepts, we summarise the relevant background from previous works in Sec. 2, discuss virtualisation-based dependability configuration strategies in Sec. 3, describe the configuration generation algorithms implemented by our tool in Sec. 4, and give guidelines to integrate with virtual machine management systems in Sec. 5.

The heart of modern virtualisation technologies is the Virtual Machine Monitor, or hypervisor, that allows multiple operating system instances to run on the same physical host. The first technology class is *full virtualisation*, like VMware[28], which introduces a layer that traps and emulates all privileged instructions. On the other hand, *para virtualisation* [9] uses a modified operating system to let virtual machines directly use hardware resources. This technique introduces lower overhead and best fits data centres.

Hypervisor offers isolation among hosted virtual machines: if a service running on a guest machine is tampered, security threats are confined and do not affect the other virtual machines. Virtual machines (VMs) are connected together by virtual connections, managed by the hypervisor, with advantages in terms of performance and security [16]. Actually, process and connection isolation assumes the hypervisor as trusted. The set of techniques to achieve trusted hypervisors, e.g. Trusted Computing [25], are out of the objectives of this paper.

Hypervisor can also freeze and restore VMs, but the migration of a VM requires taking a snapshot of *all* its resources at a given time, sending this snapshot to the destination hosting device, and rebuilding there the VM from the snapshot. Although migration is extremely promising for dependability, migration functions, as currently provided by

the hypervisor, make hard assumptions: source and destination hosts must share the same hardware architecture; the source and destination hosts must share the same network segment; the VM image and virtual storage must be already available at the destination host. Moreover, management of failures in the migration process is generally incomplete [19]. Therefore, safe VM migration requires in practice a complex process to be carefully planned.

Our process and tools allow minimising the risks of virtual systems by planning hot/cold standbys for critical VMs, configuring distributed storage, and protecting data transfer across the system. Starting from service-level models of the target dependability requirements, our tools generate multiple configurations featuring different dependability solutions. Configurations may be further selected according to other constraints (e.g. performance, cost, energy). We instead exploit these alternative configurations to construct re-configuration templates, intended in terms of switching between different configurations to react to dependability problems or changed requirements.

2 Planning Framework

Our framework implements a model-based approach to progressive policy refinement and configuration generation. We developed a tool, able to generate set of alternative configurations for complex ICT systems, considering the dependability requirements at business level.

Approaches for automatic configuration were proposed by [23], using policy refinement techniques widely adopted also in our work. The author focused his attention on configuration of devices, but did not consider in detail the chance to have different deployment plans. Another interesting work on policy refinement is [10] but it is mainly focused on authentication and authorisation rules.

In [11] authors present an approach very similar to ours, but limited to J2EE based applications, whereas ours presents a technology independent solution, in fact, in this paper we study its application at virtualized environments. Furthermore, they do not present a clear methodology for modelling information system nor a classification of the system components from the dependability point of view. [22] shows a promising model-driven approach to achieve security requirements, but it does not consider dependability problems. Another interesting project is [12], but is more concerned on software development rather than on configuration and management of an existing system.

This tool generates an allocation plan, for alternative hosting of service components, then for every plan generates the configuration for the devices involved in the hosting, in the communication enabling and protection, and the protection of hosts.

Our approach is based on a system of rules that act on the ontology representing the system. Add the beginning the ontology is separated in service and requirements on one side, and on the other side the hardware and software able to host and protect the service. Every rule reads the information from the ontology and adds some information, reducing the gap among the two side of the ontology. The rules executed after can use all the information already generated, since some rules can add the connection between the two sides. Such approach permits to fully track the automatic generation process, and insert other rules to exploit different scenarios or dependability solutions.

More details of the framework are presented in [6]: we resume here its building blocks as long as this work is layered upon them. Three steps compose the process: model construction, configuration generation and configuration ranking. During *model construction*, we describe the information system, composed by business services, virtual machines, physical resources, and dependability requirements. We model business services using a profile of W3C's *Web Service Choreography Description Language* [2]. We consider services as a set of interactions between different service components, from a choreographic point of view. This approach handles every business service component as a black box, and focuses on the role inside the global business process. To model the virtual infrastructure, we use the *System Description Language* [3] based on DMTF's CIM[24]. This model is based on a multi-layered graph of physical and software elements in the network (nodes), and their structural and logical dependencies (edges). All our models have an XML serialisation. These models are then represented as ontology, to be read by the generation process.

The *configuration generation* process produces a set of alternative virtual network topologies (composed by virtual machines and virtual networks) driven by templates plus corresponding per-device abstract configurations. Every configuration provides the business services with varying degrees of compliance with the given dependability constraints. We perform two main refinement steps in cascade: (1) generate the virtual machines to host and protect the business service components, considering the required capabilities; (2) permit and protect interactions among service components, generating an abstract configuration for every involved network node (e.g. firewalls, load balancers). The tool is designed as an ontology-based model transformation engine. Transformation rules are written using XSL and executed by a standard XSL engine. This technological choice has been taken because XSL allows writing rules using set operation discouraging the usage of internal models: since its support to procedural programming approach is minimal. In this manner, rules are naturally forced to be simple and have to exploit ontology models as intermediate data repository, by adding artifacts instance to be used by the rule ex-

ecuted after. Finally, the integration that we did within an XML database permits a distributed approach with few efforts. The engine retrieves the model processed by the previous steps from the XML database and executes the set of transformations for the current level extending the internal model. In this paper, we analyse the main changes required to make the configuration rules virtualisation aware.

3 Dependability Techniques in virtualisation environment

3.1 Service provisioning

This section describes how virtualisation technology impacts the dependability techniques used for configuring servers and technical services to maintain business service continuity.

Our tool originally computed allocation of services on physical machines. When migrating to the virtualisation world, we calculate the allocation of virtual machines while considering the security requirements determined by the hosted services. In other words, the maximum of the hosted service for every dependability requirements is considered, in order to guarantee the correct level of support for all them. This choice must consider that, for example, internal resources are scarce (few machines available) and less efficient (provider is supposed to have a better connection to the Internet), but grant higher trust levels. Outsourcing VMs instead of services implies that the configuration of VMs' internals is not delegated to the hosting provider. Our tool computes such configurations, allowing stronger control on the system.

Depending on the service allocated, we classify VMs in two classes: *working machines*, performing operations on external data (e.g. application servers), and *storing machines*, saving data on their own disk images e.g. DBMS. In accordance to this classification, the configurations also describe whether each VM should be spawned in 'persistent' (their disk images are modified) or 'non-persistent' mode (when we shut down and restart the VM all the modification on disk images are lost). As detailed in the next sections, this approach reduces the impact in case of vulnerability, and helps in backup strategies.

Service isolation and packaging The configuration rules first isolate different classes of services to limit fault propagation, including exploitation of vulnerabilities. In traditional systems, isolation implies different servers, generating big additional costs. When our tool produces plans for virtual environment, it enforces isolation more aggressively due to reduced costs of VMs. Nevertheless, it considers the different strength between such logical isolation and a traditional physical isolation.

Splitting a service onto different machines allows minimising permissions. Imagine to have two services differentiated either by roles (one for restricted users, one for public), permissions (one just reading some contents, the other modifying them), or resources (one accessing the stock, the other using customer data). With no alternative allocation, we must expose both services to each other, delegate isolation to the operating system, and increase risks of violations of the more restricted service. In similar cases, our tool spawns two different machines: a machine accessible from the public, the other implementing session-level access control (TLS or VPN); a machine accessing read-only data, the other with read-write access to data. Similarly, imagine allocating two web applications on the only available Apache web server, one requiring a module that presents a well-known vulnerability. Our tool generates two different VMs, each one with the proper set of software dependencies.

In traditional systems, our tool considers service isolation costly from the communication performance point of view, due to the additional network interactions. When adopting VMs, the tool decreases this cost, since the interactions between VMs on the same physical device are generally faster than network communications. Naturally, to exploit this feature our tool calculates the VM groups, in order to minimise the communication delay w.r.t the business service model. As result, typically, these groups collect service components belonging to the same business service [13].

Some services reside on isolated hosts by design (e.g. , firewalls of different brands, front-ends and application servers). Our planner seeks server consolidation, implementing them as different VMs on the same physical machine, to speed up network communications, while preserving most of the security advantages deriving from isolation.

Virtual machine replication Replication of resources, typically with the aid of a central manager (e.g. load balancer), is a pillar of dependable configuration. Our planning tool exploits the chance to spawn dynamically virtual machines to implement countermeasures and mitigations to problems like vulnerability exploitations, denial of service attacks, failures and QoS degradation due to misuse or spikes in demand.

Menaces deriving from possible vulnerabilities are quite common in information systems. Typically, there are three different approaches to such problems: change the software implementation with an immune one, apply a security patch, or take the risk of a possible exploitation. We model software packages with their dependences and we can tag whether a software package is vulnerable or not. Thanks to this model, the planner mitigates the impact of vulnerabilities affecting service availability exploiting "diversity implementation" concepts [18]. In other words, our tools plan

a replication of critical service components, using different operating systems, with different software implementations, and subject to different vulnerabilities. Since in many cases the vulnerability affects a library imported and not directly the software package, this approach is effective when the two implementations do not share any library. Actually, this approach is most effective when service availability is crucial, but problematic for confidentiality requirements. In fact, the chance to get confidential data increases, due to the possibility that at least one replica has unknown vulnerabilities. In this case, the planner considers a solution based on cold/hot standby as more effective: the spare system with the alternative implementation is exposed only in case the standard one has become vulnerable. In particular, the tool prescribes the storage of disk images of VMs with alternative implementations (e.g. a Linux firewall and an OpenBSD one, a Linux web server and a Windows one).

Data management In a standard system, DBMS servers provide persistent data services to applications. In virtual environment, our tool determines at configuration time how many DBMS virtual machines are needed and why. In fact, since our models manage also the nature of the data of the applications, our tool plans to spawn different DBMS instances and split data belonging to different applications. This isolation allows enforcing access control also at network level by selectively authorising service component interactions. For example, we can suppose that applications X and Y need two different data sets, L and K. Instead of allowing X and Y to communicate with the same DBMS (that contains L and K), our tools spawn two different DBMS, hosting L and K respectively. Our tool will then compute the network configuration rules allowing interactions $X <->L$ and $Y <-> K$ (see 3.2). This 'defence in depth' mechanism can be expensive, but, if correctly balanced (e.g. right grouping of data applications), impacts positively on the overall system security.

Unfortunately, virtual machine technologies do not solve the problem of data loss. On the contrary, due to increased data splitting, we need to perform more backup operations. For this problem, our planning tool exploits the same approach of non-virtual environment. It assumes the use of separated or external file systems (e.g. NFS, SAN), storing data in dependable way (e.g. RAID), and performing replication/backup services. We classify virtual machines in working machines (e.g. application servers, web servers) that offer an execution environment and operate on remote data, and storage machines that use their disk image as permanent storage (e.g. DBMS, file servers). We spawn only the storage virtual machines in persistent mode, whereas working machines are spawned in non persistent mode. Therefore, the planner applies backup services only for storage virtual machines, spawned in persistent mode. When critical information is not easily separable from the other, it is less expensive than a full backup.

3.2 Enabling and protecting communications

For authorising service interactions in non-virtualised environment, we base on network topology and equipment capabilities; e.g. we look for packet filters placed between two interacting applications and select appropriate filtering rules to permit their traffic. Virtualisation adds more flexibility in redesigning network topology (at least internally to physical hosts), and in spawning capabilities where needed (e.g. displace a virtual firewall in front of a service). A basic security advantage of virtualisation is ameliorating service interaction protection in respect of traditional local networks. Integrity and confidentiality of internal traffic are often neglected and, as a result, the local network is exposed to several security risks: unauthorised network attachment or host compromise easily lead to jeopardise interactions (traffic sniffing, ARP poisoning, host impersonation, session hijacking etc.) and services (vulnerabilities, poor controls etc.). If we deploy two interacting services in two virtual machines on the same host connected by a dedicated virtual link, their traffic is no more exposed to external attacks. Actually, in virtualised environment, most of the security guarantees fall if the hypervisor misbehave (e.g. because of unintended design or tampering). In the following we basically assume the hypervisor as trustworthy.

Filtering, proxing, balancing In local area networks, we configure Virtual LANs (VLANs) to achieve host/service separation. This solution has some issues: (1) it requires VLAN aware switches, not always available in local networks (hosts connected to the same VLAN unaware switch are necessarily part of the same VLANs); (2) it specifies which hosts are part of the same virtual network but does not control which services are reachable from whom. Using local firewalls, i.e. firewalls collocated with server equipment, has several drawbacks too: (1) decreased performances due to hardware resource sharing with the application processes; (2) software dependencies problems and incompatibility with other installed software; (3) imperfect security due to application vulnerabilities and their propagation (if one of provided services is vulnerable, and privilege escalation is possible, the attacker could easily circumvent the local firewall). For virtualised environments, our tool designs alternative ad-hoc virtual networking to enforce host isolation. It creates dedicated virtual links for interacting virtual machines, it spawns separate virtual machines to host filtering and proxy services when needed, and uses VPNs to protect traffic flowing in/out a physical host (VPN configuration is discussed in the next section). The sim-

plest configuration template contains two virtual machines within a physical host: one for the application service and another for firewall services. Adding a separate virtual machine for firewall services solves problems related to dependencies and vulnerabilities, because application and firewall are now isolated. On the other side, performance problems do not increase significantly thanks to the efficiency of modern virtualisation technologies. The firewall performance depends on hardware capabilities (CPU and memory), operating system, network stack, and filtering mechanism implementation. Several improvements have been proposed to enhance virtualised networking performances [16]. The hypervisor analyses each data packet that arrive on the network interface and transfers it to the proper virtual domain. The packet transfer operation is quite complex and requires several memory operations (data allocation/deallocation). The [16] work improves this process modifying packet transmission and reception path. For high assurance, a common practice is using two or more firewall devices in cascade: if their platforms and operating systems are different, it is reasonable to assume that they have different vulnerabilities. However, this configuration increases service latency, energy consumption, and costs. For external network attachment, our tool deploys and configures two firewall virtual machines with different OS and filtering software on the same physical host, reducing energy consumption and saving costs. However, two physical hosts are still required for achieving resiliency against hardware failures. More complex virtual networking is needed when placing several application virtual machines on the same physical host. Virtual traffic filtering can be enforced either at the hypervisor or by spawning dedicated firewall virtual machines. Hypervisor has privileged access to network traffic, does not depend on virtual network configuration, and is much more efficient. However, as the hypervisor is the single point of failure it should be kept as simple as possible: e.g. packet-filtering rules can be enforced, but extended firewalling through reverse proxies does not fit. Our tool adopts the dedicated firewall virtual machine technique. First, the tool computes the feasible configurations (e.g. single or cascade virtual firewalls) in respect of security requirements. It then performs a trade-off reusing the same firewall virtual machines for several service interactions (resources optimization). Finally, firewall virtual machines are configured with a virtual network interface per each application machine that should be filtered. This approach requires configuring a specific IP subnet for each application virtual machine, but allows defining more specific packet filtering policies (per interfaces). The increased configuration complexity is masked by the planning capabilities of our tool, that computes the virtual interfaces to adopt, and generates the filtering rules for each of those interfaces. This task is achieved by extending the *communication authorisation* module and the *reachability* algorithm described in [6] to manage virtualised networks. Practically, the reachability algorithm finds the firewall interfaces interested by each communication between network nodes by traversing the network topology graph. Then the *communication authorisation* module selects and generates filtering rules for each virtual interface.

The strategies applied for firewalling can be extended to balance load among service replicas. In fact, common load-balancing techniques use a mix of routing, network address translation, reverse proxing, and name services. All but the last one works at the same level and uses mechanisms close to the firewall services. In the last case, DNS services are configured to use a pool of IP addresses, which correspond to the replicated services. Name resolutions have a validity that can be configured to implement rude load-balancing mechanisms. With a long validity, clients keep sending requests to the same server and the load-balancing is ineffective. However, if the validity period is too short, every client request may hit a different replica making session handling trickier. To achieve load-balancing via DNS (reverse-proxy) mechanisms our tool first deploys a DNS (reverse-proxy) virtual machine associated to a set of service replicas virtual machines; then it configures the virtual interfaces and virtual machines' IP addresses, and generates the appropriate DNS address pool (proxy rules). The DNS approach is also more flexible and scalable when the load-balancing task should be performed by two or more data centres. The primary DNS could balance the request addressing the data centre site and a secondary DNS service, located into selected data centre, could address the request to the available virtual machine. We are most interested in how balancing strategies interact with security mechanisms such as TLS and VPN channels, as discussed in next paragraph.

Channel protection and Virtual Private Networks A common solution to protect the confidentiality and integrity of interactions across trust boundaries is authenticating and ciphering data using TLS or IPSec technology. Both solutions require more computational resources, increase energy consumption, and introduce traffic overhead. In virtualised environments, we can allocate services that require channel protection in distinct virtual machines on the same physical host: as already explained in the previous paragraph, traffic isolation can be granted without traffic encryption. However, when one of the endpoints is outside the administrative domain (e.g. customers, 3rd party services), or when the endpoints could not be aggregated on a single host (e.g. for resource consumption), their traffic should be ciphered. To protect this traffic, our best practice is performing aggressive service isolation and creating a separate logical ciphered channel for each interaction. Our tool proposes a set of alternative configurations with different

trade-offs between level of isolation and resource optimization. Consider a three-tier web application interacting with two user categories, for example gold and platinum users. To address availability requirements and cost savings the presentation and application servers have to be outsourced to a 3rd hosting service. Instead, to meet stringent security requirements, the database should be kept internally to the company. The most common solution for protecting the data flow between application and database across the Internet (or Intranet) is to configure a Virtual Private Network (VPN). Often, in non-virtualised environments, IPSec is configured to tunnel the traffic between two gateways. The first problem is that the traffic between hosts and gateways is not protected and might be attacked. In addition, IPSec may be difficult to configure, especially key exchange services that may collide with firewall and address translation policies. Another solution is to configure a TLS based VPN, e.g. using OpenVPN[1], that is more flexible than IPSec and easier to configure. However, as for IPSec, the TLS VPN is often configured between two concentrators. Following the rules described in Sec. 3.1, in similar cases, our tool first provides users role isolation by splitting both the application and the database in a pair of virtual machines: one for gold users and another for platinum ones. The tool then computes two alternative solutions that rely on TLS VPN concentrator virtual machines. The first adds two TLS VPN concentrator VMs per each application-database pair. This configuration performs strong interaction separation, allows configuring a different IP subnet for every TLS VPN, and provides IP reachability only between interacting endpoints. In fact, if a VPN is compromised, only its traffic can be attacked, while other interactions are not affected. The second solution provides resource optimization by adding only two VPN concentrator VMs shared by both the application-database pairs. The VPN machines are always placed on the same physical host of the VMs originating the traffic to protect. As already discussed for local firewalls, isolating the concentrator in a separate virtual machine, limits attack propagation. To implement the configuration rules above, we extended our previous algorithms, originally described in [6], for the configuration of end-to-end TLS VPNs. In practice the tool: (1) computes the required VPN concentrator VMs; (2) computes the virtual interfaces to adopt; (3) generates VPN configuration rules for each of those interfaces. Finally, we reuse the *communication authorisation* module to permit TLS VPN communications generating the proper filtering rules. Note that, in non-virtualised environments, the proposed solution and security level would have unreasonable costs as we need to configure a host for each virtual machine and two TLS VPN concentrators for each interaction. Our tool can also configure balancing services for increasing dependability of the VPN concentrators. To achieve this, we can configure DNS to resolve the VPN service to the address pool of the available concentrators. In a more complex configuration, we also add an additional layer of reverse-proxy services acting as NAT-level load balancers and forwarding client requests to one of the available concentrators. In both cases, we assume that the load sharing mechanism is performed rarely: once an endpoint has joined the VPN, the concentrator does not change. If the current VPN concentrator fails, the client should rejoin, and the DNS (and reverse-proxy) resolves to another concentrator. At last, to better secure traffic between the company and the provider's data centre (e.g. prevent traffic analysis), our tool suggests encapsulating the TLS VPNs into an IPSec tunnel. This step uses the same configuration rules originally described in [6].

We now focus on how managing secure connections (e.g. HTTPs) in case of virtual service replicas. As presented in 3.2, our strategies for placing virtual firewalls offer reverse-proxy and balancing capabilities. In fact, the use of a TLS capable reverse-proxy is the simplest solution. Every client contacts the reverse proxy that forwards the client request to one of the application virtual machines configured in the pool. Our tool generates a set of alternative configurations using a TLS reverse-proxy virtual machine. One of the simplest configurations is to displace, on the same physical host, the reverse-proxy machine and two or more application virtual machines. In this case, only the traffic between the proxy and the external world is ciphered. The interactions between the proxy and the replicas are configured as logical links using virtual interfaces as described for firewalls. When replicas are allocated to multiple physical hosts, the traffic between proxy and remote replicas is ciphered via the VPN techniques described above. Unfortunately, the reverse-proxy solution does not scale to large installations and multiple systems. We must introduce DNS-based load sharing mechanisms that requires additional issues to be solved. Even if the DNS resolution validity period is correctly configured, when it expires, a new TLS negotiation may be required. Most notably, this happens with HTTPs sessions and the use of TLS session ID. To efficiently solve the TLS balancing problem we must address two phases: negotiating a new TLS session, and resuming pre-negotiated sessions through the session ID mechanism. The straightforward solution is using a backend engine that performs TLS negotiation and/or caches session data. For example, a relay like [20] allows a server to resume a TLS session negotiated by any other one. A similar solution that implements a centralised session cache is the "distcache project"[2], supported for example by the Apache web server. This tool can be installed directly on application server replicas (e.g. distcache is supported by the Apache web server), or on reverse-proxies / TLS relays

[1] http://openvpn.net

[2] http://distcache.sourceforge.net

to load-balance TLS interactions. We also consider more distributed solutions like the TLS session tickets specified in [21]. They provide a mechanism to store session data on the client side, in the way web cookies work. Basically, the server seals the session state into an encrypted ticket and forwards it to the client, which uses this information to resume the session against any server in the pool. Similarly to the previous solutions, session tickets are directly supported by some application servers (e.g. Apache), and can be exploited at TLS proxies to enable distributed session resumption. When selecting a server-side approach, our tool allocates replicated virtual machines for DNS services, and one or more VMs for TLS engine. Instead, for the client-side approach, the tool configures the shared keys, to protect the TLS session ticket, on each virtualised replicas.

Monitoring and logging Monitoring and logging are relevant tasks to trace system behaviour and to highlight problems. For security purposes, they are useful to check host and service availability, to detect network attacks, to ensure non repudiation. In non-virtualised environments, monitoring tasks often rely on dedicated hosts and networks for better performance of application services and better isolation of monitor ones. However, to report host/service status, monitoring systems also install agents on each monitored host (e.g. Nagios[3]). To contain costs and management activities, every agent or network probe collect and sends a subset of information to a centralised host, the aggregation point that analyses the reported data. This approach has its dependability and scalability limits in the aggregation point. In virtualised environments, the monitor and logging tasks can be distributed more accurately to achieve a multi-level and dependable architecture with reduced costs. Best practice for securing monitoring and logging tasks is using stealth components, to hide the presence of agents and network probes and preserve them from security attacks. In virtualised systems, placing monitoring functions at the hypervisor can hide and protect monitoring services. On the other side, as already discussed for firewalls, the trade-off is the increased complexity of the hypervisor.

Our strategy is to install an agent on every virtual machine in order to collect information like network reachability, received network packets, and service status. A set of virtual machines able to collect reported data, are located on different physical hosts and configured in load-balancing mode to achieve high availability. These virtual machines, to better address administrator objectives, are configured to perform all monitoring/logging tasks or a specific task: for example, collecting only network reachability data. We can deploy another set of virtual machines (in load-balancing configuration) able to analyse the reported data, in order to build the network model, trace network behaviour, and evaluate security risks.

[3]http://www.nagios.org

4 Configuration Algorithms

4.1 Configuration and Re-configuration

The configuration generation process is based on the requirements (functional and non-functional) of the service(s) to be hosted by the information system. It refines these requirements, exploring the alternatives exploitation of available resources. These computations originate a Directed Acyclic Graph (DAG), where every node represents a partially refined configuration. Every step of the process, performs the refinement for every partial configuration, reading all the information generated by previous phases. Some steps generate directly element to be used for the final configuration, whereas others compute artefacts that are useful only for the steps executed after. When all the refinement steps are completed, the final leaves of the configuration DAG represents the final configurations to be enforced while configuring the information system. Some configuration will presents different level of residual requirement to be managed by the actual devices. If the requirements are high, the system recommends the usage of high availability devices; ignoring these recommendations causes a configuration with lower dependability of the system.

In order to react to emergency states, some rules perform a linkage between configurations. In other words, on a configuration, if occurs an event that could compromise the achievement of the requirements. Reaction consists in performing a configuration switch to alternative refinements, where such risky events do not occur or their effects are negligible or tolerable. In fact, faults or vulnerability exploitation make affected components unusable. Such links are named “Reaction” and links the problem in a configuration to the novel configuration to be adopted. The target (emergency) configuration is chosen among alternatives as the one with less differences compared with the current one, to minimise the reaction time. This retrieval is performed, searching the configuration that is not excluded by the fault, with the maximum number of choice of refinements in common: this operation is simple since the configuration DAG maintains tracks of every choice.

The class diagram in Fig. 1 represents the data used in the process. The dashed boxes discriminates the different supplier of such data.

Administrators supply as input the list of the service components, “Service” in the diagram, and, for each of them, the different implementations of the service, “Implementation”. Every implementation is composed by one or more software packages, with their dependencies. We refer to functional requirements to indicate the implementa-

tion together with the Service Access Point and the need to permanently store data or not. Furthermore, administrators assign values to non functional (dependability) requirements, classified on a scale of LOW, MEDIUM and HIGH: *availability* (how much is important that a service is available?), *confidentiality* (does the service manage classified data? How much it is important to preserve their confidentiality?) *integrity* (how much is serious if the service or the service data are tampered?). Other requirements could be added, but the extension must supply also additional rules to the process to refine them. Since the requirements are assigned to all the elements, we created an abstract class that is extended by every object subject to requirement. The adoption of a coarser scale for such requirements is supported and possible, if the rules are changed to understand it. Our concern is the benefit of a more detailed scale would not confuse the network administrator, without an actual gain in terms of more precise configurations.

The process exploits also the *SoftwareDatabase* that contains the data on the dependencies and conflicts among software packages. It is built on the base of package managers' repository, like RPM.

Finally, the box *Refinements* collects the elements generated during the process, described in the following sections. During the description of the process the meaning and the usage of the classes will be clarified. Such elements are grouped together to create different configuration (in the diagram we connected only the superclass Component for readability purposes). Furthermore, some components are subject to event that causes a reconfiguration tasks (a change from a configuration to another).

4.1.1 Service provisioning

The purpose of this step consists in defining the compatible software group, using the implementation of the different service components. Each of them presents ore or more implementation whose software requirements are solved by means of the software database.

First, the algorithm generates different software packages groups (containing the application software and its dependencies) for every service. If the dependency can be satisfied in different manners, alternative groups are generated. They derive the software requirements of the services implementation. Then, additional groups are added merging such basic groups, if the software packages composing them, including the dependencies are is compatible, i.e. there are not packages in conflict. Actually, if the confidentiality or integrity is HIGH or they have a different value, or if their availability is HIGH, groups are not merged, otherwise the group takes the maximum for every dependability requirement.

Moreover, requirements are inter-related. The tool propagates the requirements of availability to the software implementation (and, of course, its dependencies) into requirements of confidentiality and integrity. We propagate the requirement of integrity as confidentiality, and vice-versa. Such requirements are propagated with a level lower that the originating one, for example an availability=HIGH will become a confidentiality=MEDIUM and integrity=MEDIUM, since they are derived requirements.

Listing 1 presents the implementation of the task that is composed by three main parts. In *group initialisation*, a different group for each implementation is created. In *dependency retrieval* the tool completes the dependency by means of software database, and in case of different alternatives to satisfy, it creates different groups, without conflicting software packages. In *merging groups*, if compatible originates additional composed groups.

Listing 1. Allocation computation

```
//Group initialization
foreach service in Service.getInst()
  foreach impl in service.impl()
  Group g = new Group(service, impl)
  foreach sw in impl.getSw()
    g.add(sw);

//Dependency retrieval
foreach g in Group.getInst()
  foreach sw in g.getSoftwares()
  Impl alternativeDeps = SoftwareDB.getDeps(sw);
  foreach alternativeDep in alternativeDeps
    // if the software to add do not presents any
         conflict ...

SoftwareDB.getConflicts(alternativeDep.getSoftwares()).
     intersect(g.getSoftwares)
==null
    Group gNew = g.clone();
    gNew.addAll(alternativeDep.getSoftwares);

//Merging groups
foreach g1 in Groups.getInstances()
  foreach g2 in Groups.getInstances()
  if (g1 != g2)
    if (g1.isCompatible(g2))
      Group g = Group.merge(g1,g2);
```

The generated groups of software packages are aggregated together to compose different solutions at application level, with at least one group for every service. If the service requires MEDIUM or HIGH availability, it is possible get more groups, with a lower level of availability, implementing a redundant solutions or maintaining only one group that will be than achieved or with a VM duplication or with HA devices.

In this phase, also groups with different implementation or dependencies for the same service are generated, to implement a diversity design strategy. In this case, the propagation from the availability to the other requirements is computed subtracting two levels, since to definitely compromise the availability of the service, it is needed to find a way to compromise every implementation. On the contrary,

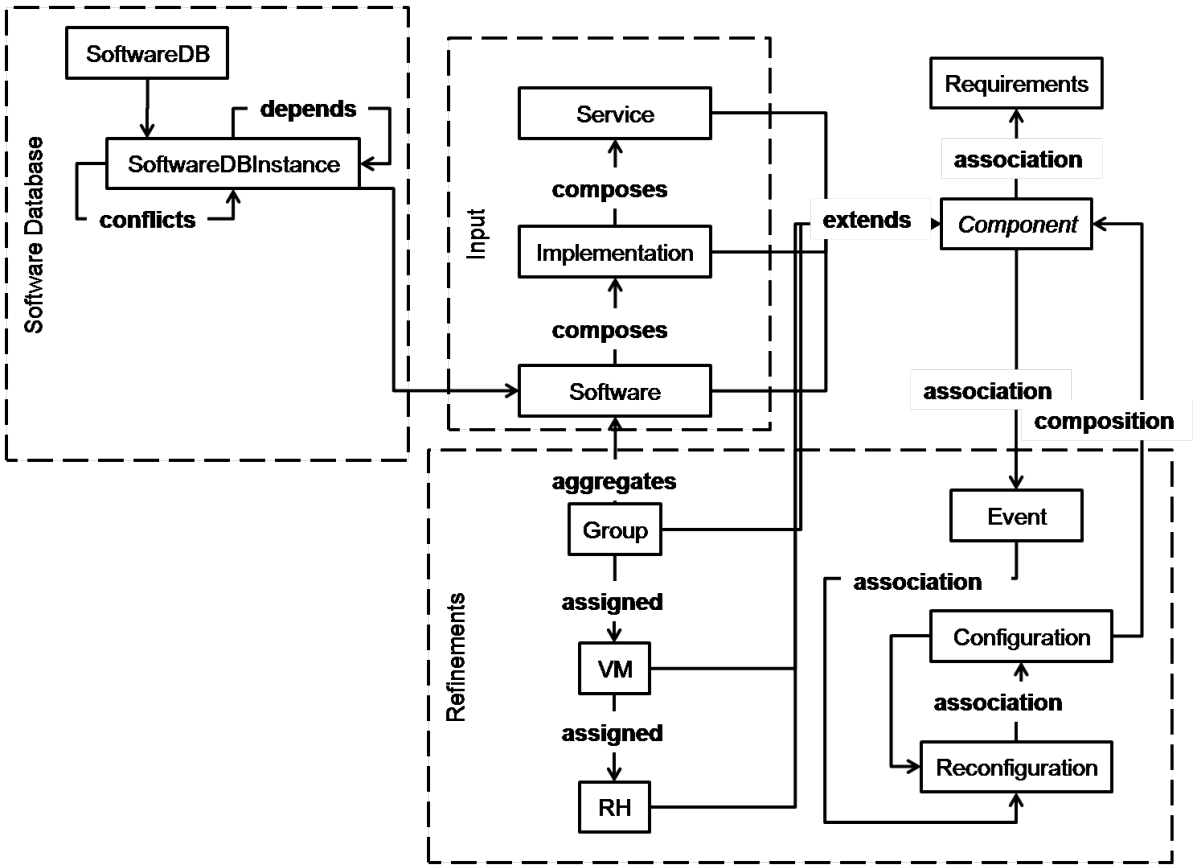

Figure 1. Class Diagram for the refinement algorithms

confidentiality requirement is augmented of one level, since the violation of single software is sufficient to access to critical condition.

A more complex analysis is necessary for integrity issues, in fact, a different implementation may help in recognising when integrity is violated (same service returning different results) but for understand the correct results it is necessary having at least three implementation, to masquerading an integrity violation on one group, as explained in [14]. Actually, to implement this measure, it necessary having ad-hoc modules to implement a voting system. Since such modules are not common, integrity requirement propagation is not modified by diversity implementation.

Additional configurations are generated, to be used in emergency conditions. Our tool generates configurations with only the service with availability=HIGH, configuration with all the service with availability=HIGH and some or all service with availability=MEDIUM and configuration with the entire MEDIUM and HIGH availability service and some low availability. Indeed, when a configuration is chosen, the best choice is the one that offers as more service as possible, but the reactions could use also others to maintain at least the critical service available.

Listing 2 brings the results of Listing 1. It performs the configuration generation, providing every possible combination of the group, collecting in conf instances.

Listing 2. Configuration generation

```
// Configurations Initialisation
if (Configuration.getInst()==null)
  foreach service in Service.getInst()
    foreach impl in service.getImpl()
      foreach g in impl.getGroup()
        Configuration conf = new Configuration(g)

//Configurations generation
foreach service in Service.getInst()
  foreach conf in Configurations.getInst()
    foreach impl in service.getImpl()
      foreach g in impl.getGroup()
        Configuration conf1 = conf.clone()
        conf1.add(impl);
        //supposes that conf is inserted into the
Configuration register only if no other conf with the
     same impls is already
available
```

4.1.2 React to new vulnerabilities discovered

It is common that new vulnerabilities are discovered when the system is already online. Typically, such vulnerabilities are available on vulnerability databases like National Vulnerability Database [4]. The tool matches the vulnerability with the software available in every system and, on the base of the impact they have, plans reactions. In fact, vulnerabilities entries present also a Common Vulnerability Scoring System [5] evaluation of the impact in term of confidentiality, integrity and availability.

In practice, the configuration generator retrieves all the vulnerabilities affecting every software package in the system and crosses the menaces of exploitation with the dependability requirements. Even if, it is generally a good practice having as few open vulnerability as possible in the system, the tool is concentrated to maintain satisfied dependability requirements. In other words, if a software package presents a vulnerability affecting its integrity, and the requirements for the group of software packages are for availability it does not consider vulnerability exploitation a problem. A natural objection to this method is: "What? Such requirements are linked. Integrity gives confidentiality, integrity and confidentiality gives availability...". We agree with this assertion, in fact we consider these issues in the propagation of the requirements throughout the configuration phases, and, they are already evaluated w.r.t. the interdependency among them.

Typical reaction to vulnerability consists in switching (when possible) to an alternative implementation, updating/patching menaced software packages, or turn off the dangerous application. If there is an alternative configuration that is not affected by the vulnerability in object, the reaction implies a configuration change. Unfortunately, in many cases, such solution is not feasible and we have not any alternative implementation. Consequently, the model of software and software dependencies needs to be updated with the last version of the products, which hopefully will be not affected to this. Then, the tool generates additional configurations, which are added to the configuration tree. At this point, we plan a configuration swap to one of the new configuration generated. In case of none of the previous solution is possible, it remains or tolerating the situation (if the availability of the service is crucial) or switching to a configuration that does not use to software, even if some service are not available. This last approach is to be considered transitional, and the update of the software model is performed regularly, in order to generate a configuration with all the services available again as soon as possible.

Listing 3 computes the reaction in case of vulnerability discover. For each requirements associated to a software, an event of type VULNERABILITY is set. If there is a different implementation of sw, the reaction performs the configuration swap to the corresponding configuration; otherwise, it excludes the sw from the usage (again as configuration change) or tolerates the vulnerability on the base of the availability requirement.

Listing 3. Reaction to vulnerability discover

```
//Reaction to vulns
foreach conf in Configuration.getInst()
  foreach impl in service.getImpl()
    foreach g in impl.getGroup()
      foreach sw in g.getSoftware()
      reqs = sw.getReqs()
        foreach req in reqs
          if (req.getValue() == HIGH
||req.getValue() == MEDIUM)
        Event vt = new Event(sw, VULNERABILITY
,req);
       //finds an alternative configuration, with the
service implemented by the sw available, but without the
      sw vulnerable
      alternativeConfs =
ConfigurationDag.filter(!contains(sw)).filter(!contains(
     service)).getCloser(conf
);
      if (alternativeConfs!=null)
        Reaction r = new Reaction (vt,
alternativeConf);
      else
      // if the requirement availability is not HIGH,
whereas the others are HIGH or MEDIUM, or the
     availability is HIGH, while the
it is we plan to switch off the service
        if (reqs.getAvailability().getValue !=
HIGH || (reqs.getAvailability().getValue != MEDIUM &&
     req.getValue()== MEDIUM))
         //finds an alternative configuration,
but without the sw vulnerable (but this will not have
     the service availabel)
          alternativeConfs =
Configurations.getInst().filter(!contains(sw).filter()))
     ;
          Reaction r = new Reaction (vt,
alternativeConf);
```

4.1.3 Virtual machine architecture

VM generation considers the groups of software of every configuration generated by the previous step, together with their dependability requirements. First, for every group is created a separated VM. Second, if a group presents a high level of availability, more than one VM can be assigned, with a lower availability requirement. Depending by the service, it is indicated if the machine needs to store permanently on its own file system or not, to discriminate between working machines and storage machines.

The dependability requirements of the VM are used to eventually choose among different VM templates. If the availability requirement is high, the performance of the VM must be coherent. If the confidentiality is high, it will require VM templates with additional protection, like personal firewalls or disk-image encryption. VM integrity will imply, again, protection from network, like personal firewalls, but also digital signature of disk-images. The adoption of trusted computing techniques will be also useful to

satisfy such requirements.

The VM resulting from this computation derives the requirements from the software packages of the corresponding group, which are used to plan the deployment of the machine on virtualisation environment. Also VM configurations for emergency conditions are generated, to be linked by reactions, as we see in the following sections.

For this work, we consider having different templates of virtual machines, classified on the base of their capability to satisfy non-functional requirements. First, the machine can present a classification on the base of their performances, to answer to availability requirements. Then, they can have an operating system implementing Mandatory Access Control (MAC) that benefits confidentiality and integrity. They can offer the chance to spawn only digitally signed disk images. They can have one or more personal filtering software packages (e.g. proxy, firewall) to limit the access from external. They can offer cryptographic primitives to stores only ciphered data that increase the confidentiality, but decrease the availability. Unfortunately, the match between these capabilities and the dependability requirements is not one-to-one, and, furthermore, some mechanisms useful for requirements are counter-productive for others. For this reason, over dimensioned configurations are generated, and, only after considering all mechanisms, the configuration are chosen.

Actually, the configurations candidates to be applied are the ones with the lower requirements remaining. In fact, machines with higher specifications cost more. Another important factor is the number of machines, the lower it is, and the cheaper is the solution. More expensive, but promising, configurations are kept into account to perform reactions in case of problem.

Listing 4 presents the generation of VM starting from groups. In this part is crucial the match between the dependability requirements of the groups and the service offered by the VM. In the first part, *generation of vm* , the availability is mitigated generating different configurations with a varying number of VM assigned. Availability requirements are coherently adjusted, to consider the redundancy. In the second part, *mitigation of requirement thought VM capabilities*, creates different configuration, for every VM, adopting a different technical solution. Note that, since this process is executed more times, also combinations of different solutions are possible. MAC means mandatory access control, TC trusted computing techniques, CRYPTO cryptographic primitives, FILTERING a personal firewall system.

Typically, the best configurations are the one whose requirements on VMs are lower, since it implies

Listing 4. VM generation

```
//Generation of vm
foreach conf in Configuration.getInst()
  foreach impl in service.getImpl()
    foreach g in impl.getGroup()
      //Maximum number of machines to be created
     if (g.getReqs.getAvailability().getValue == HIGH)
          limit
= 4;
     else if (g.getReqs.getAvailability().getValue ==
          MEDIUM)
limit = 3;
     else limit =2;

     for (n_machines=1; n_machines++; n_machines<=limit)
       conf_new = conf.clone();
       for (i=1; i++; i<=n_machines)
         //get the correct VM template
        vm = new VM (group, conf_new);
        vm.setReqs (group.getReqs());
         // decreases the availability, one level
every additional virtual machine
        vm.decreaseAvailability(n_machines)

//Mitigation of requirement thought VM capabilities

foreach vm in VM.getInstances();
conf = vm.getConf();
if (group.getReqs().getIntegrity = HIGH)
  vmt = vm.clone();
  vmt.add(MAC);
  vmt.getReqs().decrease(INTEGRITY);
if (group.getReqs().getIntegrity = HIGH)
  vmt = vm.clone();
  vmt = vm.clone();
  vmt.add(TC);
  vmt.getReqs().decrease(INTEGRITY);
if (group.getReqs().getConfidentiality = HIGH)
  vmt = vm.clone();
  vmt.add(CRYPTO);
  // increases the integrity, since the requirement of
       confidentiality is
mitigated if the VM is integer
  vmt.getReqs().increase(INTEGRITY);
  // increases the availability requirements, since
       cryptography decreases
performance
  vmt.getReqs().increase(AVAILABILITY);
  vmt.getReqs().decrease(CONFIDENTIALITY);
if (group.getReqs().getIntegrity > MEDIUM or group.
     getReqs().getConfidentiality
> MEDIUM)
  vmt.add(FILTERING);
if (vmt.is(CRYPTO))
  avail = vmt.getAvailability ();
if (avail == MEDIUM)
  vmt.setAvailability (HIGH);
if (avail == LOW)
  vmt.setAvailability (MEDIUM);
```

4.1.4 React to service performance degradation

When performance for a service is not satisfactory, due to peaks in workload or a Denial of Service Attack, the reaction consists in adding computational resources to the stressed service. Another solution consists in using VM with higher performance, instead of the original ones. Alarms derive from sensor put on the VM, and could, for example, high CPU or memory consumption. Another alarm is typically required, to advertise that the solution is oversized (for example low CPU load) to react back to normal

usage conditions.

To generate this reaction plan, the tool generated additional configuration for virtual machines that uses a higher number of VMs then in normal condition, or VM with more computational resources, also for the configuration that do not have all the service active. At this point, depending by the availability needs of the VM, we generate a reaction to the proper configuration: if availability needs is set to HIGH tools prefer a configuration with an additional VM, also if this implies losing some service (with an availability less than high); if availability is MEDIUM it generates a configuration switch to use an additional VM only if it has the same available services of the starting one, otherwise it plans the usage of a more powerful machine; if the availability is LOW, it plans the usage of a more powerful virtual machine, if available, otherwise it performs no reactions.

Listing 5 presents the computation of reaction to low performance on a virtual machine. If the availability is HIGH, we switch to another configuration with a higher number of machines, also decreasing the number of machines without considering other machines. When the availability is MEDIUM, it switches only to configuration with the same number of service available.

Listing 5. reaction to VM performance degradation

```
// React to performance degradation
foreach vm in VM.getInstances()
  g = vm.getGroup();
  impl = g.getImpl();
  // for each configuration in which the vm is involved
    foreach conf in vm.getConfs()
      /// in case of performance low, pass to a
           configuration
with an higher number of machines
     if (vm.getReqs().getAvailability == HIGH)
       Event e = new Event(PERFORMANCE_LOW, vm);
       alternativeConf =
ConfigurationDag.filter(.getImpl().getGroup().getVM().
     count >
.getVM().count()).getCloser(conf);
       Reaction r = new Reaction (e, alternativeConf);
     if (vm.getReqs().getAvailability == MEDIUM)
       Event e = new Event(PERFORMANCE_LOW, vm);
       alternativeConf =
ConfigurationDag.filter(conf.getImpl() ==
.getImpl()).filter(.getImpl().getGroup().getVM().count >
.getVM().count()).getCloser(conf);
       Reaction r = new Reaction (e, alternativeConf);
```

4.1.5 React to VM integrity violation

As already said, integrity violation implies more complex reactions, but virtualisation environment helps.

First, the tool distinguishes in storing and working machines. For working machines that are spawned in non-persistent mode, a first measure consists in rebooting the machine.

On the contrary, for storing machines, we can react swapping to a configuration in which the violated machine is not longer used. If this violation is caused by vulnerability exploitation, and the integrity is high, the system could also consider switching off the machines affected, until a reaction for the vulnerability is not put in act. Another option offered by the tool consists in changing the configuration to another one with a stronger integrity protection for the machine.

Listing 6 pseudo-code shows the implementation of this step. It creates an event of type INTEGRITY_VIOLATION for every VM with an integrity grater or equal to MEDIUM, and attaches to it a reaction to configuration where the integrity requirement is lower. In case of lack of proper alternatives, if the requirement of availability is lower than the integrity one it react passing to a configuration that does not use the tampered machine, otherwise it tolerates the violation, without any change to configuration.

Listing 6. reaction to VM integrity violation

```
foreach vm in VM.getInstances()
  g = vm.getGroup();
  impl = g.getImpl();
    foreach conf in vm.getConfs();
     if (vm.getReqs().getIntegrity() > MEDIUM)
     /// in case of integrity violation, search a vm,
          with
the same group but with a lower integrity requirements,
     Event e = new Event(INTEGRITY_VIOLATION, vm);
     alternativeConf =
ConfigurationDag.filter(vm.getGroup().getVM().
     getIntegrity() <
group.getVM().getIntegrity()).getCloser(conf);
     if (altenativeConf!=null)
       Reaction r = new Reaction (e, alternativeConf);
     else if (vm.getIntegrity()>vm.getAvailability)
       // if not available and integrity is greater
than availability, switch to a configuration that does
     not use the vm (switch
off the vm)
       alternativeConf =
ConfigurationDag.filter(!.contains(vm)).getCloser(conf);
```

4.2 Enforcing security controls

The refinement process, as defined previously, is represented as a directed acyclic graph. The general workflow to enforce security controls is composed by the following steps: (1) evaluation of the suitable and available technologies; (2) refinement of directly and indirectly security controls; (3) generation of alternative configurations. Each step refines the information provided by the previous ones and populates the graph adding a new level for each step. Each path between the root and a leaf represents an enforceable solution. Alternative configurations are represented as different nodes of the same level.

The first step analyses the security requirements to derive the suitable technologies to protect communication traffic.

Our model associates CIA properties (confidentiality, integrity and availability), expressed as LOW, MEDIUM or HIGH to a set of suitable technologies and related modes. For example, if confidentiality and integrity are set to MEDIUM the model suggests that the available technologies are TLS, IPSec and TLS VPN. Otherwise, if confidentiality and integrity are set to HIGH, the model suggests adopting IPSec or TLS VPN. On the contrary, the availability property identifies when a service has to be replicated adopting load-balancing techniques. The association between security properties and technologies is defined into technology templates, expressed using an XML based language. Such templates also contain technology-specific modes to support alternative configuration. For example, the IPSec technology-specific template contains three different configurations: tunnel, transport and remote access.

After analysing suitable technologies, the tool analyses network components capabilities to evaluate which technologies can be directly or indirectly enforced. A technology is directly enforceable if exist a set of services or devices which support the related capability (e.g. IPSec, TLS, filtering). Otherwise, if no service or device is able to support the technology we should enforce it indirectly installing a new feature or adopting virtualisation techniques. Our choice consisting providing a virtual machine with the required features in order to make this process more flexible, secure and feasible. The benefits derived from this strategy are described in details in the next sections. Our tool supports filtering and channel protection security controls which can be both directly or indirectly enforceable. This process is divided into two sub-steps: network component analysis and communication policy analysis. The first step takes in input the network description and classifies hosts, routers, firewalls and services information to discover network components features and capabilities. The second step analyses communication components to identify network components involved in the policy. More precisely the algorithm transforms the network in a graph, finds all paths between policy elements (source and destination) and identifies the security components (which have security capabilities, e.g. filtering, channel protection) involved in the communication. When filtering or channel protection capabilities are not available, our tool adopts a set of virtual machines with configurable security controls to enforce the policy. Such VMs have to be added to the output VM architecture for deployment on virtualised providers.

The adoption of virtual machines allows enforcing a security control on demand, satisfying policy requirements. First of all it is necessary understanding which security controls should be enforced using the virtualisation technology. For example, considering the objective to enforce IPSec in tunnel mode for protecting traffic between two servers. The tunnel mode requires the adoption of two IPSec gateways to enforce the tunnel (as defined in the IPSec technology-specific template). Evaluating security components involved in the policy the algorithm find only one suitable gateway. To enforce the policy two general solutions are available: (a) deploy a virtual machine as IPSec gateway; (b) deploy two virtual machines, one for each server. However the problem is quite complex. When an administrator configures IPSec in tunnel mode, the traffic that belongs on untrusted network (e.g. Internet) must be ciphered. This simple consideration entails that each gateway is displaced internally to a trusted network or at its border. Therefore, the use of a gateway displaced in untrusted network partially invalidates the benefit of ciphered channel. However, if an IPSec gateway is displaced in untrusted network, a possible solution is to deploy two virtual machines (in the trusted networks) and not configure the other gateway. The deployment of a virtual machine requires a host able to support virtualisation. To identify in the network which are the available hosts we define the virtualisation feature as a capability. Therefore, when the algorithm performs network description analysis identify the hosts which support virtualisation. At previous step the algorithm decides which virtual machines are necessary then it evaluates, using a set of strategies, which hosts are selected to guest VMs.

This approach requires to analysing allocations finding which hosts guest allocated VMs. To maximize performance a physical host supports a limited set of virtual machines depending on its hardware features. Hence if the host supports the required VMs to enforce security controls the algorithm plan to add the new VMs. Otherwise our strategy is to find the nearest suitable host.

Once the tool decides which security controls should be adopted to protect communication traffic it is necessary to identify their displacement. Our approach is to group together services and security controls as indivisible block using a predefined template. Practically a template contains a set of predefined VMs to perform security controls which can be enabled or disabled depending on requirements (e.g. TLS VPN VM and firewall VM). In addition this strategy allows configuring the related virtual network in more simple way.

4.2.1 Security controls templates

Once the tool derives the required security controls, the related VMs templates (depicted in Fig. 2) are used to generate alternative configurations which should be finally deployed on virtualisation platforms. The simplest template is *ServiceTemplate* that contains three virtual machines: *VM_Service*, *VM_FW* and *VM_VPN*. The first virtual machine is adopted to host the service and it is mandatory. The *VM_VPN* instead can be activated on demand to protect ser-

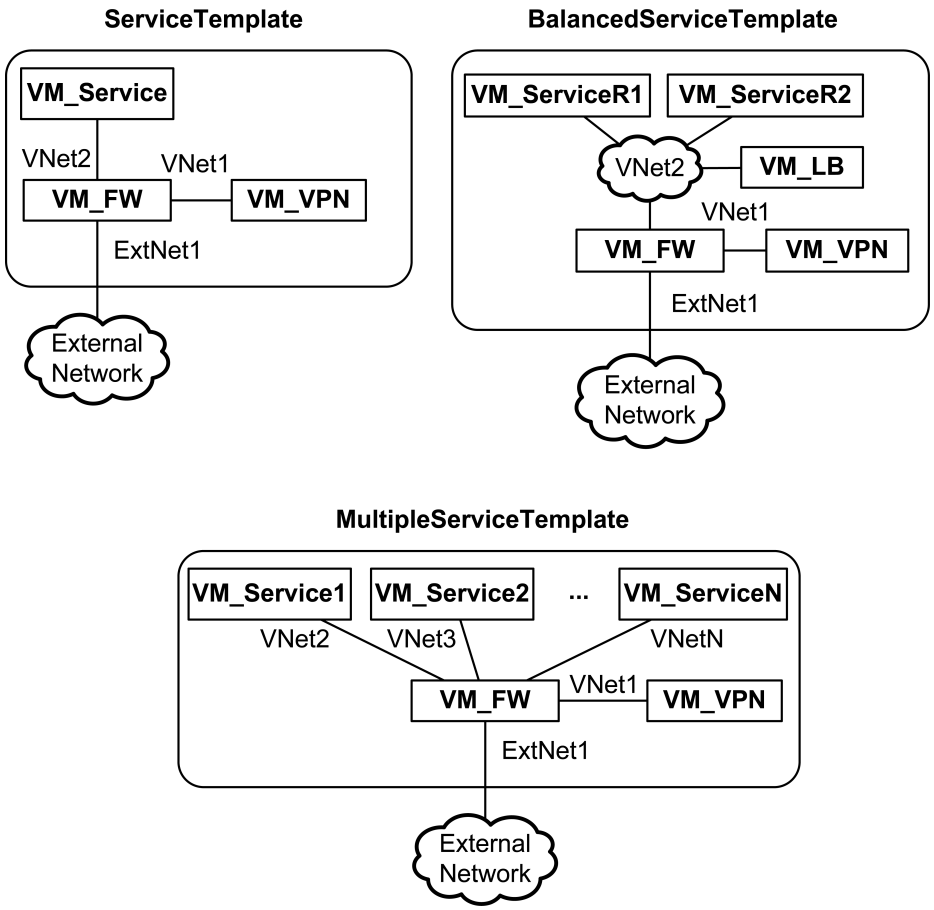

Figure 2. Security controls VMs templates

vice traffic. It can be implemented using IPSec or TLS VPN technologies as alternative solutions. The *VM_FW* is also mandatory and it is responsible to filter traffic among service, VPN and external network. As depicted in Fig. 2 this virtual machine is connected in bridged mode to the external network (*ExtNet1*), the local sub-network of the host, and to other VMs using two virtualised networks (*VNet1* and *VNet2*). This logical configuration allows separating and masquerading external and internal (virtualised) networks. For example, to allow traffic between the external network and the service, on the filtering virtual machine we can add a port forwarding rule, masquerading internal network. In addition, when *VM_VPN* is active, it is possible to derive a set of rules to: (a) allow only ciphered traffic between external and *VM_VPN*; (b) allow unprotected traffic only between *VM_VPN* and *VM_Service*. The security requirements, defined as input are refined using templates. For example, if the confidentiality requirement of service is set to HIGH, in the resulting template the *VM_Service* and *VM_VPN* are set to MEDIUM.

The *BalancedServiceTemplate* introduces the load-balancing functionality adopting the *VM_LB*. This, often implemented as a reverse-proxy, is in charge to dispatch traffic (protected or not protected) among replicas to enhance service availability. The application of this template is mandatory when in the configuration there are more software package groups for one implementation. In that case the availability requirement on *VM_LB* is set to HIGH. An interesting extension to ameliorate service availability is to add another load-balancer VM. This can be placed in standby and resumed as soon as the first load-balancer fails. In order to decrease the number of virtual machines and to aggregate functionalities the filtering and reverse-proxy could be grouped together as a *VM_PRX-FW*. An interesting extension of this template is to introduce load-balancing feature for VPN traffic. In a simple scenario we can substitute the service replicas with VPN virtual machine replicas enhancing VPN service availability. In more complex scenario the objective could be to improve availability of application and VPN services. In that case, we can modify the template to insert two different load-balancers: one for application and another for VPN replicas.

Finally the last template *MultipleServiceTemplate* is suitable to displace different service virtual machines on the same physical host. In this configuration each service is directly linked to the *VM_FW* using a virtual sub-network with specific network address and the VPN gateway is shared to different services. However it is possible configure the virtual machines to belong to the same sub-network. Grouping together the service VMs allows optimizing their allocation enhancing general performance introducing a security drawback. In fact, each service is protected using the same VPN channel and if the VPN is compromised, all traffic could be jeopardized. To reduce the related risk the template could be modified to support different VPN gateways. Practically we can displace a shared VPN gateway for non critical service and a specific VPN VM for each critical service.

4.2.2 Security controls transformations

Once the tool selects the set of templates to adopt, it performs a set of transformations to refine the provided information. A transformation T is an operation that takes as input a set of services, a template and security requirements and produces a set of alternative configurations. A subset of security controls transformations is depicted in Fig. 3. A configuration is expressed using an XML based language and contains: (a) the VM components; (b) the virtualised and physical network description; (c) the policies to enforce. The VM components and the virtual network links are derived from the set of services and template, then the virtual and physical network configuration are computed consequently. The firewall virtual machine is directly connected using a bridge to the physical network. The related external IP should be assigned accordingly to physical host subnet evaluating network description. The algorithm, after a network analysis, proposes to the administrator a set of available IPs. For each virtual links the algorithm computes the network configuration for each virtual machine. In practice it selects the addresses range from a table that contains the allocations for private networks (populated considering the [1]) and generates other network information (gateway, DNS). Finally, the tool generates the related policies for the provided solution. This process is composed by two steps: (1) generation of channel protection policies; (2) generation of filtering policies. Considering the selected security technology and the service description, the algorithm generates the related properties for each virtual machine related to a protected communication. For example, if the selected technology is HTTP over TLS, the TLS properties (e.g. cipher suite, server authentication or mutual authentication, etc.) are attached to the service virtual machine component configuration. In similar manner, when the tool selects TLS VPN or IPSec, the algorithm attaches the related properties to the VPN component. Finally the algorithm evaluates the components security properties and virtual network configuration generates the filtering policies. In practice, the firewall VM is configured to forward traffic between external world and internal services according to security properties. For example, if the internal service is protected using HTTP over TLS, the firewall policy forwards only HTTPS traffic between external and internal networks. To clarify the security controls refinement we discuss the transformations of Fig. 3. The most simple transformation is identified by (a) and generates a configuration that contains a service and a firewall virtual machines. The tool (1) defines the properties of the virtual network that links service to the firewall; (2) generates a port forwarding policy to allow external traffic reaching internal service using specific protocol and port. The algorithm analyses the table of private addresses and provides the network configurations for the virtual machines shown in listing 7. For example, let us consider that the service is a web server that uses HTTP protocol on port 80: the firewall (*VM_FW*) is configured to forward traffic from external network to the internal service as described in listing 7.

Listing 7. Network configuration and policies for (a)

```
<VM id='VM_Service'>
  <network-interface value='eth0'>
    <addr type='IPv4' value='10.10.10.2' />
    <netmask value='255.255.255.0' />
    <gateway value='10.10.10.1' />
    <dns value='ask to admin' />
  </network-interface>
</VM>

<VM id='VM_FW'>
  //bridged interface
  <network-interface value='eth0'>
    <addr type='IPv4' value='ask to admin' />
    <netmask value='ask to admin' />
    <gateway value='ask to admin' />
    <dns value='ask to admin' />
  </network-interface>
  //VNet1
  <network-interface value='eth1'>
    <addr type='IPv4' value='10.10.10.1' />
    <netmask value='255.255.255.0' />
    <gateway value='ask to admin' />
    <dns value='ask to admin' />
  </network-interface>
  <policies>
    <filtering-policy id='1'>
      <type='forwarding' />
      <src ip='*' proto='tcp' port='80' />
      <dst ip='10.10.10.2' proto='tcp' port='80' />
      <action value='allow' />
    </filtering-policy>
  </policies>
</VM>
```

The (b) example demonstrates service load-balancing. In this case the tool generates network configurations for: (1) the sub-network *VNet2* that contains the load-balancer

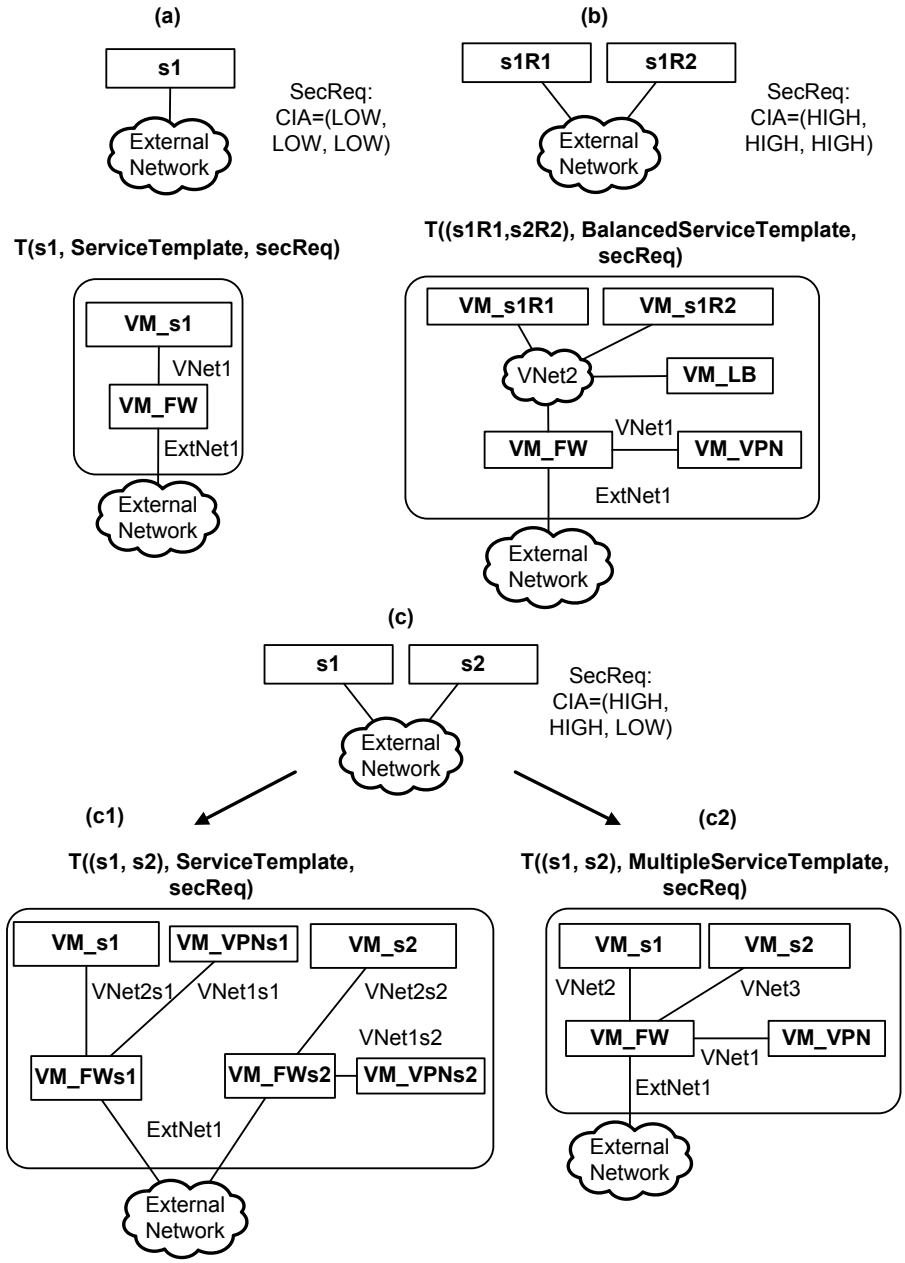

Figure 3. Security controls transformations

(*VM_LB*) and the replicas (*VM_s1R1* and *VM_s1R2*); (2) the sub-network *VNet1* that contains the VPN gateway (*VM_VPN*). The virtual firewall (*VM_FW*) policies are: (1) port forwarding for load-balanced services; (2) port forwarding for VPN service (if the gateway is based on TLS VPN technology). In addition, the tool configures the *VM_LB* to balance traffic among replicas and the *VM_VPN* to protect communications. The load-balancer configuration depends on mechanism selected to implement its functionality. The VPN gateway, often implemented as TLS VPN (for flexibility purposes), can be configured as client or server. For example, it is often configured as server on the service side and in client mode on external clients (any-to-one interaction model). In other cases, VPN gateways could be configured as one-to-one, i.e. to tunnel traffic between two different services or any-to-any, i.e. to tunnel

Listing 8. VPN and balancer network configuration and policies for (b)

```
<VM id='VM_VPN'>
  <network-interface value='eth0'>
    <addr type='IPv4' value='10.10.10.2' />
    <netmask value='255.255.255.0' />
    <gateway value='10.10.10.1' /> <!-- to VM_FW -->
    <dns value='ask to admin' />
  </network-interface>
  <vpn-network-conf>
    <vpn-pool type='IPv4' lowValue='10.10.10.5'
highValue='10.10.10.10'/>
      <vpn-service-addr type='IPv4' value='10.10.10.2' />
      <vpn-service-interface type='tap' value='tap0'/>
      <vpn-service-protocol type='tcp' port='1194'/>
  </vpn-network-conf>
</VM>
<VM id='VM_LB'>
  <network-interface value='eth0'>
    <addr type='IPv4' value='10.10.20.2' />
    <netmask value='255.255.255.0' />
    <gateway value='10.10.20.1' /> <!-- to VM_FW -->
    <dns value='ask to admin' />
  </network-interface>
  <balancer-conf>
    <balancer id='balancer-VM_LB'>
      <addr type='IPv4' value='10.10.20.2' />
      <proto type='tcp' port='80' />
    </balancer>
    <replicas>
      <replica id='replica-VM_s1R1'>
        <addr type='IPv4' value='10.10.20.3' />
        <proto type='tcp' port='80' />
      </replica>
      <replica id='replica-VM_s1R2'>
        <addr type='IPv4' value='10.10.20.4' />
        <proto type='tcp' port='80' />
      </replica>
    </replicas>
  </balancer-conf>
  <balancer-policies>
    <balancer-policy id='1'>
      <balancer-fronted value='balancer-VM_LB'/>
      <balancer-replica value='replica-VM_s1R1'/>
      <balancer-replica value='replica-VM_s1R2'/>
    </balancer-policy>
  </balancer-policies>
</VM>
<VM id='VM_s1R1'>
  <network-interface value='eth0'>
    <addr type='IPv4' value='10.10.20.3' />
    <netmask value='255.255.255.0' />
    <gateway value='10.10.20.1' />
    <dns value='ask to admin' />
  </network-interface>
</VM>
<VM id='VM_s1R2'>
  <network-interface value='eth0'>
    <addr type='IPv4' value='10.10.20.4' />
    <netmask value='255.255.255.0' />
    <gateway value='10.10.20.1' />
    <dns value='ask to admin' />
  </network-interface>
</VM>
```

traffic among different services. In listings 8, 9 we propose a configuration to balance *VM_s1R1* and *VM_s1R2* replicas. In this case, we adopt the TLS VPN approach and the listing 8 contains useful information for *VM_VPN* configuration. First of all, the definition of the network addresses which belong to the VPN and assigned to clients, in that case, the tool allocates 5 IPs from 10.10.10.5 to 10.10.10.10. In addition, the tool generates, using the specific TLS VPN template, the virtual interface (tap0) and the protocol and port of the VPN service (tcp/1194). To allow external clients accessing the internal services (*VM_s1R1* and *VM_s1R2*) they must join to the VPN. For that purpose, the virtualised firewall is configured to forward ciphered traffic (using TCP protocol) to the virtual host 10.10.10.2 on port 1194. The load balancer configuration is quite simple and contains the description of adopted replicas and a set of policies to balance traffic. In practice for each replica the tool defines an IP address that belongs to the *VNet2* and adds *VM_s1R1* and *VM_s1R2* as members of balancing pool.

Listing 9. Firewall network configuration and policies for (c)

```
<VM id='VM_FW'>
  //bridged interface
  <network-interface value='eth0'>
    <addr type='IPv4' value='ask to admin' />
    <netmask value='ask to admin' />
    <gateway value='ask to admin' />
    <dns value='ask to admin' />
  </network-interface>
  //VNet1
  <network-interface value='eth1'>
    <addr type='IPv4' value='10.10.10.1' />
    <netmask value='255.255.255.0' />
    <gateway value='ask to admin' />
    <dns value='ask to admin' />
  </network-interface>
  //VNet2
  <network-interface value='eth2'>
    <addr type='IPv4' value='10.10.20.1' />
    <netmask value='255.255.255.0' />
    <gateway value='ask to admin' />
    <dns value='ask to admin' />
  </network-interface>
  <policies>
    //allow access to VPN
    <filtering-policy id='1'>
      <type='forwarding' />
      <src ip='*' proto='tcp' port='1194' />
      <dst ip='10.10.10.2' proto='tcp' port='1194' />
      <action value='allow' />
    </filtering-policy>
    //allow access to balancer
    <filtering-policy id='2'>
      <type='forwarding' />
      <src ip='10.10.10.*' proto='tcp' port='80' />
      <dst ip='10.10.20.2' proto='tcp' port='80' />
      <action value='allow' />
    </filtering-policy>
  </policies>
</VM>
```

The last example (Listing 9) describes the alternative solutions (c1 and c2) to configure a set that aggregates two or more services. The major difference between the provided solutions is that in (c1) each service is protected by a dedicated firewall and VPN gateway, on the contrary, the (c2) adopts shared firewall and VPN virtual machines. The different approaches have pros and cons. The dedicated fire-

wall and VPN allow protecting the services in more fine grained way. For example, if a VPN channel is compromised, the other services are not involved. On the contrary, if we adopt a shared VPN gateway, in case of attacks, each service could be compromised. However the use of shared resources enhances the entire system performance. In addition, the configuration of dedicated firewall and VPN, often requires assigning an external IP address for each firewall (*VM_FWs1* and *VM_FWs2*) to correctly perform the port forwarding. Consider for example that *VM_s1* and *VM_s2* are web servers hosting *s1* and *s2* applications and each one communicates with external world using tcp protocol on port 80. In this situation we need to assign an external IP address for each firewall to distinguish *s1* and *s2* requests.

5 Virtual machines generation and management

Once the VM configurations are generated, it is necessary create and configure the related VMs to implement a solution. This goal requires the creation of the software environment that hosts the services, the configuration of networks, services, policies and the deployment of the VM to a physical host. In addition, to react on fault events, it is also important monitor VM services, manage startup, shutdown and migration of the virtual machines.

5.1 Building software environment and specific configurations

Several approaches could be followed to build the software environment, for example it is possible create manually a base system template, building a VM from a common GNU/Linux distribution. Then, the base system template could be modified in automatic way, generating a set of specific templates, adding the required software packages to implement services. For example, the base system could be transformed into a VPN VM template adding IPSec or OpenVPN software packages. Similarly adding the Apache Tomcat package we can implement a web server VM template. The next step requires translating the services, network configuration and policies from a technology specific and device/service independent language (previous described models) to the device/service specific language (e.g. network configuration for a GNU/Linux system, rules for a netfilter firewall). This task can be performed using a set of adapters, one for each device/service category. For example, an adapter for VPN should be able to translate an IPSec configuration into the racoon specific language and a TLS VPN into the OpenVPN language. Similarly, for filtering, the adapter should translate the device/service independent configuration into a set of netfilter or PF (OpenBSD Packet Filter) rules. The adapter output is a device/service specific configuration that must be deployed into the VM.

5.2 Deploying and Managing VMs

Finally, the configured VMs should be deployed into the physical machine that will host the services. This task requires to: (1) identify which physical machine are able to host the VMs; (2) define a set of mechanisms to transfer the VMs to physical hosts. Parsing the system description model allow to identify which physical hosts support virtualisation and which are their performance. This information is useful to know how many virtual machines can be deployed on a particular host. The deployment process, or in other words the task that transfers the VM to a physical host, is quite simple but it depends on the virtualisation technology adopted (e.g. Xen, KVM, VMware, etc.). On the contrary the VM managing tasks (startup, shutdown, migration, network and disk management) are not simple to address. To handle these tasks and reducing the problem complexity a possible solution is to adopt a toolkit, like *libvirt* [15], able to deal with different virtualisation technologies, hiding details. The *libvirt* toolkit offers a set of API to interact with the virtualisation capabilities of physical hosts to deploy and manage VMs.

5.3 Our approach

In this section we describe our approach to generate and manage the virtual machines accordingly to the previous defined models.

5.3.1 Architecture layers

To perform the tasks described before we introduce in our architecture three different layers: *VM software*, *VM configuration*, *VM management*. The VM software layer contains the activities to build a specific VM template like *VM_FW*, *VM_VPN*, *VM_LB* and VM for specific services, for example to host Apache Tomcat web application. More practically we start from a common GNU/Linux distribution, built manually, to derive automatically a set of predefined templates, adding the required packages, e.g. OpenVPN for *VM_VPN*. To perform this process automatically we adopt a tool (*ubuntu-vm-builder* [27]) able to add new software packages to a virtual machine. The predefined templates could be generated only once, when the tool is run for the first time. On the contrary, a specific template that hosts a particular service must be generated when needed. The build process is performed on a particular physical host, the builder machine that is also used to deploy and manage VMs. The VM configuration layer performs the following tasks: (1) VM internal network configuration; (2) service

configuration; (3) policy configuration; (4) VM configuration. The VM internal network configuration is generated accordingly to the security controls transformation, starting from network configuration and policies model, defined before. The information is translated using a specific adapter that transforms the model to a specific network configuration language. For example, considering the listing 7 and suppose that the VM is implemented using an Ubuntu Linux operating system, each VM network configuration is translated into '/etc/network/interfaces' language. For bridging interfaces, the tool asks to admin the IP address that should be defined accordingly to physical host and external network. The service configuration, for example an Apache Tomcat web application, is translated using a specific template and a service dependant adapter. The policies, similarly to network configuration are translated from network configuration and policies model to a specific security control language, e.g. netfilter, using a specific adapter. Finally it is necessary to build the configuration of the virtual machine as XML model. To perform this step the tool takes as inputs (a) the VM template generated before (e.g. *VM_FW*); (b) network configuration and policies model. Considering the listing 10, the template is used to define every properties of the XML model except for the network interface tags, that are generated using network configuration and policy model. In addition, this model is used to create the virtual network properties, as shown for *VNet1* in listing 11. The specific configurations, except for virtual network properties, derived from previous tasks, are deployed into the VM. More practically, the VM disk is mounted on local file system of the builder machine, and specific configuration are copied accordingly. On the contrary the virtual network properties are used in the next step to configure the virtualisation environment.

The VM management layer is able to setup the virtualisation environment, deploy, migrate, start and stop a virtual machine. In order to create and deploy a VM on the physical host, the first activity is setup the virtualisation environment. In practice, our tool, using the *libvirt-java* API, creates the network environment on the remote host, accordingly to the network properties described in listing 11. In the next step, the tool takes as input the VM configuration model of 10 and using the API creates the new domain on remote physical host. Finally, the last step copies the VM disk on the remote host, and the VM is ready to start.

Listing 10. *libvirt* **VM configuration model**

```
<domain type='kvm'>
  <name>vm-fw1</name>
  <uuid>0cc4736f-2568-241d-c610-2e7ba90002f5</uuid>
  <memory>262144</memory>
  <currentMemory>262144</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='i686' machine='pc-0.11'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/kvm</emulator>
    <disk type='file' device='disk'>
      <source file='/home/vm-deployment/vm-fw1-disk.img'
          />
      <target dev='hda' bus='ide'/>
    </disk>
    <disk type='file' device='cdrom'>
      <target dev='hdc' bus='ide'/>
      <readonly/>
    </disk>
    <interface type='network'>
      <mac address='54:52:00:60:ef:e5'/>
      <source network='VNet1'/>
    </interface>
    <interface type='bridge'>
      <mac address='54:52:00:04:3b:c0'/>
      <source bridge='br0'/>
    </interface>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target port='0'/>
    </console>
    <input type='mouse' bus='ps2'/>
    <graphics type='vnc' port='-1' autoport='yes' keymap=
        'it'/>
    <video>
      <model type='cirrus' vram='9216' heads='1'/>
    </video>
  </devices>
</domain>
```

Listing 11. *libvirt* **virtual network properties for** *VNet1*

```
<network>
  <name>VNet1</name>
  <uuid>3e3fce45-4f53-4fa7-bb32-11f34168b82b</uuid>
  <bridge name='virbr1' stp='on' forwardDelay='0' />
  <ip address="10.10.10.254" netmask="255.255.255.0">
    <dhcp>
      <range start="10.10.10.100" end="10.10.10.200" />
      <host mac="54:52:00:60:ef:e5" name="VM_FW" ip="
          10.10.10.1" />
    </dhcp>
  </ip>
</network>
```

6 Conclusion

We have analysed how the best practices for service and network dependability change when exploiting modern virtualisation technologies. Based on the results of this study, we have updated the process and tools we had developed for semi-automatic dependability planning of information systems.

In particular, this paper has detailed the algorithms that our tools exploit to automatically compute allocation and reaction plans for virtualised information systems. Each algorithm is defined in terms of the relevant modelling ontology and the pool of transformation rules working on this ontology. The actual implementation exploits XML representations and transformation languages. We have also developed prototype integration of our tools with popular virtual machine management software, which is an essential step towards automatic deployment of the generated plans.

During the selection of the relevant configuration strategies, we have focused on the point of view of the virtual data center customer: our approach can, and should be, extended taking in further consideration the point of view of the hosting provider, which may partially conflict with the customer's one.

Acknowledgment

This work was developed in the framework of IST-026600 DESEREC, "Dependability and Security by Enhanced Reconfigurability", an Integrated Project partially funded by the E.C. under the Framework Program 6, IST priority.

References

[1] Network Working Group, Address Allocation for Private Internets. IETF RFC 1918, February 1996.

[2] W3C Consortium, Web Services Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/, 2005.

[3] POSITIF Consortium, The POSITIF System Description Language (P-SDL). http://www.positif.org/, 2007.

[4] National Vulnerability Database. http://nvd.nist.gov/, 2008.

[5] Common Vulnerability Scoring System. http://www.first.org/cvss/, 2009.

[6] M. D. Aime, P. C. Pomi, and M. Vallini. Policy-driven system configuration for dependability. *Emerging Security Information, Systems, and Technologies, The International Conference on*, 0:420–425, 2008.

[7] M. D. Aime, P. C. Pomi, and M. Vallini. Planning dependability of virtualised networks. *Dependability, International Conference on*, 0:46–51, 2009.

[8] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[10] N. Damianou, N. Dulay, E. C. Lupu, and M. Sloman. Ponder: a language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC 2000/1*, 2000.

[11] T. Eilam, M. Kalantar, A. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in internet data centers. *Communications Magazine, IEEE*, 44(3):166–177, March 2006.

[12] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Requirements engineering meets trust management - model, methodology, and reasoning. In *In Proc. of iTrust '04, LNCS 2995*, pages 176–190. Springer-Verlag, 2004.

[13] M. Israel, J. Borgel, and A. Cotton. Heuristics to perform molecular decomposition of large mission-critical information systems. In *SECURWARE '08: Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies*, pages 338–343, 2008.

[14] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[15] libvirt. http://libvirt.org/.

[16] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. Proceedings of the USENIX Annual Technical Conference, 2006.

[17] Microsoft Corporation. Azure Service Platform. http://www.microsoft.com/azure/.

[18] D. M. Nicol, W. H. Sanders, and K. S. Trivedi. Model-based evaluation: From dependability to security. *IEEE Transactions on Dependable and Secure Computing*, 1(1):48–65, 2004.

[19] J. Oberheide, E. Cooke, and F. Jahanian. Empirical exploitation of live virtual machine migration. In *In Proceedings of the BlackHat DC convention*, 2008.

[20] E. Rescorla, A. Cain, and B. Korver. SSLACC: A Clustered SSL Accelerator. In *Proceedings of the 11th USENIX Security Symposium*, pages 229–246, Berkeley, CA, USA, 2002. USENIX Association.

[21] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. IETF RFC 5077, January 2008.

[22] F. Satoh, Y. Nakamura, N. K. Mukhi, M. Tatsubori, and K. Ono. Methodology and tools for end-to-end soa security configurations. In *SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I*, pages 307–314, Washington, DC, USA, 2008. IEEE Computer Society.

[23] J. Strassner. *Policy-Based Network Management: Solutions for the Next Generation (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[24] The DMTF Technical Committee. The Common Information Model (CIM). http://www.dmtf.org/standards/cim, 2008.

[25] Trusted Computing Group. https://www.trustedcomputinggroup.org, 2009.

[26] Ubuntu. Ubuntu Enterprise Cloud. http://www.ubuntu.com/cloud/private.

[27] Ubuntu. ubuntu-vm-builder. https://help.ubuntu.com/8.04/serverguide/C/ubuntu-vm-builder.html.

[28] VMware. http://www.vmware.com.