Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 12, Number 3, pp. 339–352. http://www.scpe.org

A VIRTUALIZATION-BASED APPROACH TO DEPENDABLE SERVICE COMPUTING

CIPRIAN DOBRE*, FLORIN POP†, VALENTIN CRISTEA‡, AND OVIDIU-MARIAN ACHIM§

Abstract. Dependability represents a critical requirement for modern distributed systems. Today new requirements emerged. Among them reliability, safety, availability, security and maintainability are needed by more and more modern distributed service computing architectures. In this paper we present an approach to ensuring dependability in distributed infrastructures using virtualization for fault-tolerance, augmented with advanced security models. The proposed solution is part of a hierarchical architectural model that allows a unitary and aggregate approach to dependability requirements while preserving scalability of large scale distributed systems. In this context we propose dependability solutions based on the use of virtualization and modern security models, combined together to automatically protect services and applications belonging to the distributed system. We also present evaluation results for several scenarios, involving applications and services with different characteristics.

Key words: virtualization, dependability, large scale distributed systems

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction. Dependability represents a critical requirement for modern distributed systems. Both in the academic and industrial environments there is an increasing interest in large scale distributed systems (LSDS), which currently represent the preferred instruments for developing a wide range of new applications. While until recently the research in the distributed systems domain has mainly targeted the development of functional infrastructures, today researchers understand that many applications, especially commercial ones, have complementary necessities that the "traditional" distributed systems do not satisfy. Together with the extension of the application domains, new requirements have emerged for LSDS. Among these requirements, reliability, safety, availability, security and maintainability are needed by more and more modern distributed applications.

In systems composed of many resources the probability of a fault occurring is higher than in traditional infrastructures. When failures do occur, the system should limit their effects and possible even initiate a recovery procedure as soon as possible. Dependability, therefore, depends on the possibility to detect failures, and successfully recover from them. By failures we mean both hardware and software, permanent or transient, but also failures to execute operations as well as security breaches in the LSDS. Thus dependability also includes the use of adequate security models, policies, and technologies to detect and limit the effect of possible entities not obeying well-established rules.

The main contributions of this paper are: (1) *a solution designed to detect application- or service- specific failures occurring in different parts of the system*, (2) *a virtualization-based solution designed to facilitate fault recovery by freezing services running in good states and which further uses these virtual images for future state recovery or automatic migration of entire file systems, or sets of processes, for increased LSDS redundancy*, and (3) *a Mandatory Access Control (MAC) security layer designed to encapsulate services in layers with highly well-protected security access control rules.* Used together, these contributions can increase dependability in case of traditional service-based LSDS. We propose a virtualization approach to ensuring dependability by using checkpoint strategies and protection domains using virtual hosts, coupled with a proactive replication strategy necessary to maintain consistent states of the system in case of failures. The solution is part of the DEPSYS dependability architecture ([1]). It assumes that on top of a typical operating system the services run in specialized virtual environments. Such virtual environments are saved and, in case of failures, moved and re-executed on another workstation in the distributed system. The re-execution also uses appropriate consistency algorithms.

The virtual environments running on top of the operating systems and hosting the services running inside

*Computer Science Department, Faculty of Automatic Controls and Computers, University POLITEHNICA of Bucharest, Romania(ciprian.dobre@cs.pub.ro).

†Computer Science Department, Faculty of Automatic Controls and Computers, University POLITEHNICA of Bucharest, Romania(florin.pop@cs.pub.ro).

‡Computer Science Department, Faculty of Automatic Controls and Computers, University POLITEHNICA of Bucharest, Romania(valentin.cristea@cs.pub.ro).

§Computer Science Department, Faculty of Automatic Controls and Computers, University POLITEHNICA of Bucharest, Romania(ovidiu.achim@cs.pub.ro).

the distributed system form a separate layer. It allows better fault tolerance, by separating faults in different containers, or replication of virtual sandboxes to multiple nodes. It also allows quick integration with advanced security policies, a second requirement for dependability. We present examples of complementing the virtualization approach with security policies directly at the level of the operating system.

The rest of this paper is structured as follows. Section 2 presents related work. Section 3 presents the architectural design on which the dependability layer is based. In Section 4 we present the virtualization-based approach. Section 5 presents solutions designed to secure services and virtual containers, using modern security models, in large scale distributed systems. Section 6 presents experimental results. In Section 7 we conclude and present future work.

2. Related work. *Fault tolerance* includes detection and recovery. *Fault detection* in LSDS was approached in [2] through an adaptive system. The detection of faults is achieved by monitoring the system and dynamically adapting the detection thresholds to the runtime environment behavior. The prediction of the next threshold uses a Gaussian distribution and the last known timeout samples. The solution has several limitations. It cannot differentiate between high response times that are due to the transient increase of the load in the communication layer and those due to service failures, so that both are interpreted as service failures. We solve these problems and propose a solution which adapts to both the failures in the infrastructures, but also to the different requirements imposed by applications (for example, real-time application require a higher response time, instead of failure detection accuracy).

For *failure recovery* virtualization has lately enjoyed a great surge of interest. Still, with few exceptions, current solutions in this space have been largely ad-hoc. Authors of [21] analyze methods of leveraging virtualization for addressing system dependability issues. Their analysis, based on combinatorial modeling, show that unless certain conditions (e.g., regarding the reliability of the hypervisor and the number of virtual machines) are met, virtualization could in fact decrease the reliability of a single physical node. In fact, motivated by this observation, the authors propose a reliable Xen virtual machine monitor (VMM) architecture, called R-Xen. It consists of a hypervisor core and a privileged virtual machine called Dom0. Dom0, being much bulkier than the hypervisor core, is the weak link for Xen reliability. We extend such results and present a solution which considers replication inside the virtual machines sets of virtual containers, each one hosting sets of services. Thus, we protect the services as replicated containers, and the containers as part of replicated virtual machines. The virtual machine monitoring architecture is completely shielded inside the VM, thus our solution solves the problem of weak links regarding reliability.

An alternative solution proposed in [3] groups several server nodes into a set that appears virtually to clients as a single node. Upon receiving a request from a client, a node forwards the request to the nearest neighbor that offers the service. The service discovery is performed using an amended version of anycast routing scheme by using the properties of the Mobile IPv6 protocol. The disadvantage of this solution is that it works only on nodes running the XtreemOS operating system. Also, the system assumes the clients support the Mobile IPv6 protocol. The solution concentrates on the mechanisms to detect a working service from a set of replicated services. However, it does not include mechanisms to recover a request when a service fails. We present a more generic failure recovery mechanism that masks the replication of virtual nodes (combined with a container-based solution we previously demonstrated in [1]), works with a wide range of transport protocols, detects failures with higher accuracy, and takes recovery decisions that are adequate to be used with various SOA middleware.

A solution to handle fault tolerance in Grid systems is presented in [4]. The paper describes a resource failure detector, together with a fault manager, that guarantees that tasks submitted are completely executed using the available resources. The solution uses the Intra-cluster and Intra-grid load balancing model [5]. It assumes a Grid architecture that is mapped on a tree structure, where several fault managers collect failure information from fault detectors running on lower level nodes. The idea is similar to the one used in DIGS [6], which aims to increase fault tolerance of web services using the model of a fault-tolerant container. A container is a logical set consisting of several service instances. All requests to these services are mediated by specialized entry points. This is used to enforce access policies, and to increase fault tolerance of service accesses. The *security* is therefore approached at the level of the container, much like in our case. For fault-tolerance each container manages a set of replicated service instances. The containers can be configured with various fault tolerance policies. For example, an equivalent service instance is invoked when another one fails, or multiple equivalent instances are invoked with the same request and a voting mechanism is applied. However, the proposed solution uses one proxy for each service. The client accesses the service through a proxy, not directly,

so that the use of the Proxy server is not transparent to the user. To access the service container customers must know the URI of the Proxy service. In addition, the replicated business services invoked by the proxy are not necessarily deployed in the same container as the proxy service, which claims for the use of the URIs of replicas for invocations. The solution proposed in our paper eliminates these disadvantages. Each container is also protected against unauthorized accesses through a dedicated MAC security layer, designed with well-protected security access control rules.

3. An architectural model for dependability in large scale distributed systems. The proposed dependability approach is part of the architectural model that we proposed in [1]. The general approach to ensuring fault tolerance in LSDS consists of an extensible architecture that integrates services designed to handle a wide-range of failures, both hardware and software. These services can detect, react, and confine problems in order to minimize damages. By learning and predicting algorithms they are able to increase the survivability of the distributed system. They include services designed to reschedule jobs when resources on which they execute fail, services capable to replicate their behavior in order to increase resilience, services designed to monitor and detect problems, etc. The proposed architecture is also based on a minimal set of functionalities, absolutely necessary to ensure the fault tolerance capability of distributed systems.

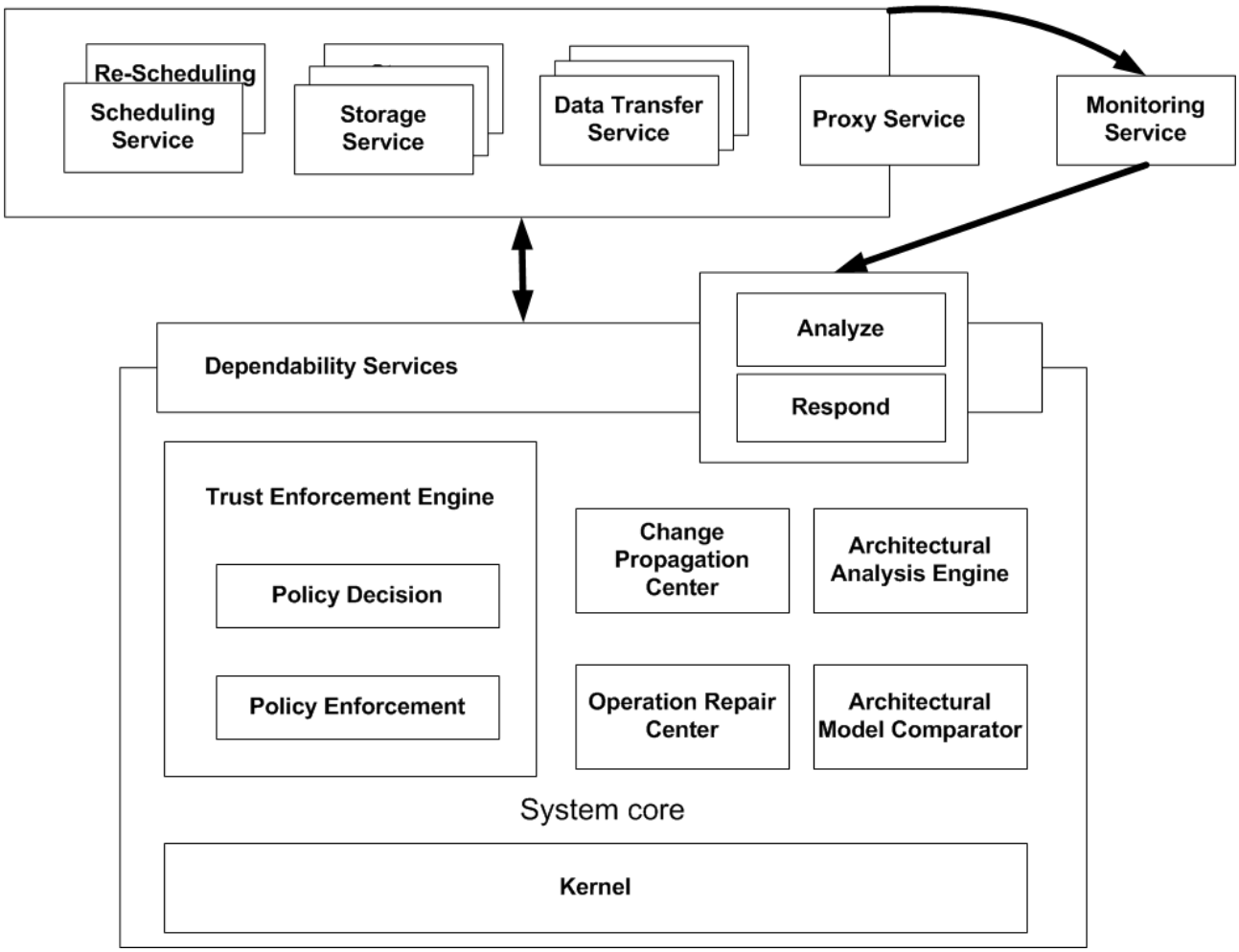

Fig. 3.1: The dependability architecture for LSDS.

An abstract model of the components making up the architecture at the middleware layer of the distributed system is presented in Figure 3.1. These components are designed to ensure fault tolerance between different hosts composing the system. At the bottom of this architecture is the core of the system, designed to orchestrate

the functionalities provided by the other components. Its role is to integrate and provide a fault tolerant execution environment for several other components.

The architecture also includes mechanisms for ensuring resilience based on replicating components of the system, such as the ones responsible for communication, storage and computation. It also considers combining the replication mechanisms with solutions to ensure survivability of the system in the presence of major faults. The solution to develop an architecture in which the system survive by adapting in the presence of fault arise naturally by explicitly acknowledge the impossibility to include a complete solution to ensure reliability considering the resulting resources and technologies. Because of this we adopted a strategy based on using replication only for the most basic core functionality of the system. We use replication in the form of fault-tolerant containers; the fault-tolerant containers can easily manage a set of replicated services, an approach presented in the next Sections.

4. An accrual failure detection service. Failure detection is an essential service of any fault tolerant distributed application that needs to react in the presence of failures. The detection of a failed process is difficult because of the unpredictability of the end-to-end communication delays. To solve the agreement problem of distinguishing between a failed process and a very slow one, various authors proposed the use of local failure detectors ([16], [17]). These are modules attached to the monitored processes and which gather data about their current status.

A failure detector (FD) combines two important functions: monitoring and interpretation. This distinction is most clear in case of accrual protocols. The family of accrual failure detectors consists in error detection processes that associate, to each of the monitored processes, a suspicion value. Previous proposed failure detectors are poorly adapted to very conservative failure detection because of their vulnerability to message loss. In practice message losses tend to be strongly correlated (i.e., losses tend to occur in bursts).

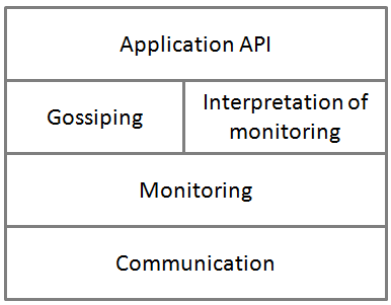

Fig. 4.1: The layers of the failure detection system.

We propose the use of an FD service that improves previously proposed solutions with several capabilities: (1) a scalable communication infrastructure used between remote failure detection processes, (2) the use of a proposed estimation function based on sampling previous responses and a formula for the computation of the suspicion level of a process being failed which more accurately reflect the dynamic nature of LSDS, (3) the addition of gossip-like protocols for failure detection which leads to the elimination of wrong suspicions because of the varying network traffic conditions, and (4) the capability of providing the failure detection function as a service to various distributed applications.

The proposed failure detector is based on the results previously presented in [18]. Several important results were further demonstrated in [19]. The service provides the detection capabilities of a distributed system composed of several monitoring and detection processes. The failure detection processes run inside the distributed environment, each being responsible for the monitoring of a subset of other processes or applications. The system is composed of four layers (see Figure 4.1).

The *Communication* layer handles a scalable, fault tolerant and dynamic communication infrastructure for the upper-layers. The failure detection processes are grouped in clusters. A cluster contains detectors that are geographically close, and also experience small communication delays. The failure detectors monitor only other processes inside the same cluster. The communication involves both heartbeat messages and gossip updates. The clustering minimizes the time needed to send messages between failure detection processes.

Each cluster is under the management of a cluster coordinator process. This process is responsible for the management of the local failure detection processes, as well as for intra-cluster and inter-cluster communication management. The coordinator handles the cluster management for example when processes enter or exit the system. Inside each cluster there are several failure detection processes, capable of exchanging messages between each others to identify possible failures. At cluster level the coordinator acts as a communication gateway. The clusters are interconnected through these gateways. The approach forms a hierarchical interconnecting communication infrastructure. Such an approach ensures scalability because the failure detectors communicate only with the processes inside the same cluster, and all intra-cluster communication is channeled through dedicated network links.

Within the second layer, *Monitoring* (see Figure 4.1), each detection process is responsible with the monitoring and logging of the data. Each FD process is responsible for monitoring several other FD processes from the same cluster. Periodically, each monitored process must issue a heartbeat message announcing it is alive. Periodically, every $T_{monitor}$ seconds, each FD process scans the list of monitored processes. It sends a heartbeat message to the each of the remote monitoring processes. Based on the receive heartbeat message, FD process updates the corresponding suspicion level.

It is difficult to determine the nature of a failure that affects a process. In an unstable network, with frequent message losses or high process failure rates, any detection algorithm is almost useless, because the processes cannot distinguish between permanent and transient failures. The third layer of the architecture attempts to solve this drawback by employing an approach based on *gossiping*. The role of gossiping is to reduce the number of false negative (wrong suspicions) and false positive (processes are considered to be running correctly even though they have failed) failure decisions. For this, each FD process periodically exchanges local failure detection information with other FD processes within the same cluster.

At this layer, we propose using a component responsible with the interpretation of monitored data (see Figure 4.1). The component analyses and further processes the monitored data. The data processing involves the use of a function for the estimation of the next heartbeat message (adaptive threshold), as well as a function for the computation of the probability that a remote process experienced failure (see Figure 4.2).

We call the probability value a *suspicion level*, as it represents the suspicion associated with the possible failure of a remote process. The suspicion level represents the degree of confidence in the failure of a certain process. While in case of accrual detection the suspicion level increased on a continuous scale without an upper bound, in this case the suspicion level takes values in the $[0, 1]$ interval so that a zero value corresponds to an operational process, and the probability of failure increases while the value approaches 1. Each failure detector maintains a local suspicion level value $\mathrm{sl}_{qp}(\mathrm{t})$ for every monitored remote process.

The computation of the suspicion level is based on the sampling of past heartbeat arrival times. The arrival times of heartbeat messages are continuously sampled and used to estimate the time when the next heartbeat is expected to arrive. The estimation function uses a modified version of the exponential moving average (EMA) function called KAMA (Kaufman's adaptive moving average) ([19]). This function ensures a more accurate prediction of the arrival time for the next heartbeat message using a trend of recent timestamps. The predicted value is further used to compute the suspicion level of the failure in case of each remote process. The suspicion value increases from 0 to 1. In case of a heartbeat H in the beginning, while H is not yet expected, the current suspicion level is 0. As time passes and H does not arrive, the suspicion increases to 1, this value being associated with the certainty in that H is lost. The suspicion value $\mathrm{sl}_{qp}(\mathrm{t})$ is computed as:

$$sl_{qp}(t) = \frac{t-1}{t+1}, \; where \; t = \frac{t_{now}}{t_{pred}} \tag{4.1}$$

The proposed function returns values in the $[0, 1]$ interval. It has a relatively quick evolution in the $[0, 0.8]$ interval and a slow one in the $[0.8, 1]$ interval. The function leads to a high probability of failure recognition in a reasonable amount of time, aspect previously demonstrated in [19].

The last layer of the FD system is represented by the interface with the applications. The failure detector is composed of several distributed processes. The failure detection capability is provided to application in the form of a *Web service*. This allows for standardized methods to access and communicate with the failure detectors, with advantages such as interoperability, flexible integration in various technologies, etc. The service provides operations such as the registration for specific failure detection events (a notification mechanism), and the interrogation for failure suspicion values.

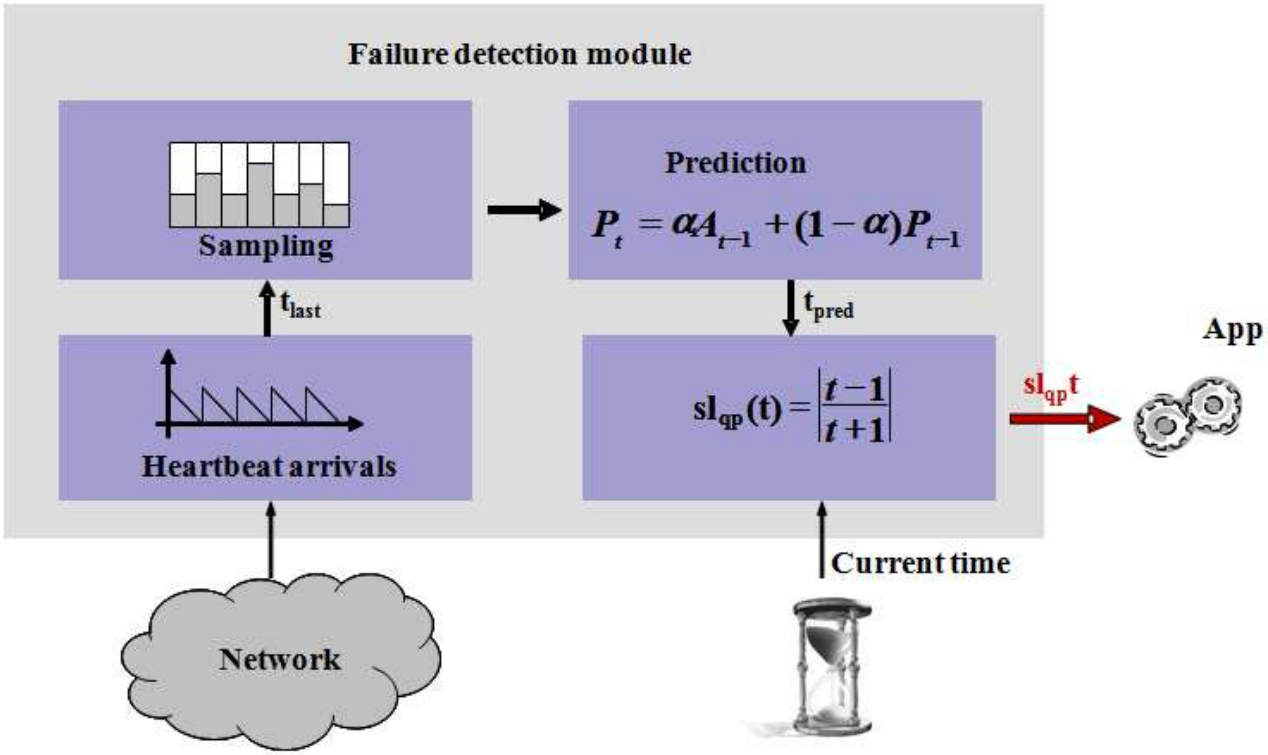

Fig. 4.2: Information flow within the failure detector.

5. A virtualization-based approach to dependability. The failure detector previously presented is complemented with a recovery solution that uses virtualization. The approach consists in the use of several type I virtual images (hosted directly on the computer hardware) running the same services [23]. These are native virtual images that are smaller than an entire operating system snapshot (unlike VMWare images, our implementation uses OpenVZ images which host only subsets of processes - an aspect crucial for communications costs involved by migration and similar operations). Such a virtual server hosts a small set of processes. These processes are generally the web containers hosting the services of the upper-layer LSDS middleware. Therefore, at this layer, we are interested in protecting the processes rather than the services themselves. A solution to protect the services themselves from LSDS-specific failures was also previously demonstrated in [14].

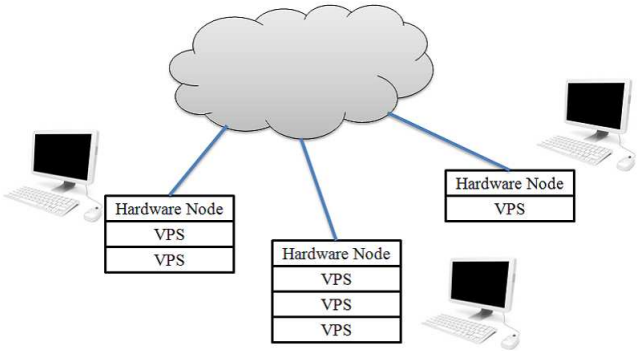

Fig. 5.1: The architecture of the system as composed by virtual environments.

At this layer the virtual environment is developed with support for both OpenVZ and LXC virtual en-

Fig. 5.2: The algorithm run by the coordinator.

```
procedure coordinator {
  send VOTE_REQUEST to all nodes
    log SEND VOTE REQUEST
  if timeout OR receive wrong answer {
    send GLOBAL_ABORT to all nodes
    call ESTABLISH_CONNECTION_STATUS
  }
  if all sent VOTE_COMMIT {
    send GLOBAL_COMMIT to all nodes
  }
}
```

Fig. 5.3: The algorithm run by the client.

```
procedure client {
  wait for VOTE_REQUEST
  receive VOTE_REQUEST
    log VOTE REQUEST
  send VOTE_COMMIT
    log VOTE COMMIT
  start timer
  wait GLOBAL_COMMIT or GLOBAL_ABORT
  if timer expired {
    /* Clientul lost GLOBAL_COMMIT or GLOBAL_ABORT */
    call ESTABLISH_NEIGHBOUR_STATE
  } else if received GLOBAL_COMMIT {
    initialize checkpointing
  } else if received GLOBAL_ABORT {
    continue
  }
}
```

vironments, and it is reinforced with security models assured by technologies such as SMACK or SELinux. The virtualization layer includes, therefore, several virtual servers (VPS) hosting services belonging to the distributed system (see Figure 5.1). OpenVZ ([7]) uses a type I hypervisor (running native on top of the physical machine). Multiple OpenVZ environments can share the same operating system's kernel. The overhead is lower than in alternative virtualization approaches such as Xen or VMware. LXC (Linux Containers) is a Linux native approach similar to OpenVZ. Except for creating the virtual environment in which services runs separately, fault tolerance is achieved using checkpointing.

There are two types of VPS nodes. A first type of node assumes the role of activity coordinator. So inside such nodes special processes (called coordinators) implement the detection algorithm (which monitors the processes running inside the virtual servers), as well as manage specific recovery actions. A coordinator is responsible with checkpointing, restoring and load balancing of virtual images. The coordinator can initiate a migration process of virtual images working with the hypervisor, and using the secured processes (such as ssh) on the remote workstation.

The other virtual nodes host the services belonging to the distributed system. They are under the control of the coordinator, which can detect and initiate repairing actions using the last known working snapshot of the virtual server. The virtual servers are also protected by various security policies.

For consistency, the implementation uses a modified coordinated checkpointing algorithm based on a Two-

Phase Commit protocol. The coordinator acts as mediator and initiator of the algorithm (see Listing 5.2). It starts by sending a VOTE_REQUEST to all nodes. If the coordinator receives from all clients a VOTE_COMMIT than a consistent global checkpointing is possible and, thus, it further sends a GLOBAL_COMMIT message. If any of the slave nodes is not responding, or it simply sends VOTE_COMMIT, then the coordinator responds with GLOBAL_ABORT. After sending a GLOBAL_ ABORT message, the coordinator tries to determine what happened to the nodes that did not respond. Depending on the answer, it can decide whether or to further keep the node in the list. The algorithm is presented in Listing 5.2. For fault tolerance, the coordinator itself is replicated (as a VPS node). When it fails, another coordinator can initiate a recovery action consisting of a retry to restore the failed node using a checkpoint image and synchronizing the differences.

When a slave node receives a VOTE_REQUEST message it responds with VOTE_ COMMIT and starts an internal timer. If it receives GLOBAL_COMMIT before the timer expires, the node simply starts its checkpointing action. If the timer expires and the node does not receive GLOBAL_COMMIT, then it sends to all other nodes a GLOBAL_ABORT. The algorithm is presented in Listing 5.3.

State restoring is similar to the distributed checkpointing approach. Again, a coordinator initiates the restore and if all nodes agree the system is restored to a consistent state.

For load balancing the coordinator monitors the load of the clients. When the load exceeds a predefined threshold the coordinator migrates VPS nodes on other station. The migration process is done without interrupting the connection with the node. The same principle is applied for fault tolerance. When the coordinator detects failures inside a VPS in uses the last saved checkpoint to initiate a recovery procedure and spawn new VPS nodes. These new nodes take over the faulty ones, and are initiated for failure tolerance on different adjacent nodes.

In our experiments we managed to automatically correct full-stop failures (i.e., complete shutdown of a node inside LSDS, or the lost of network connectivity) (see [14]). This is possible because the coordinator can recover a VPS, possibly on a completely different host inside LSDS, starting from a snapshot saved on the hard drive. The approach can also lead to further research towards the automatic corrections of transient failures. The virtualization can lead to diversity, which in turn means we can run the same VPS possible under different hosts. Or gossiping algorithms can be used to mediate between possible outcomes of operations.

The costs involved with the use of virtualization for fault tolerance was considered a problem by previous authors ([20]). For each service a XEN- or VMWare-based solution requires several virtual images (each one containing its own operating system, memory occupied by the processes, stacks, file systems, etc.) cloned inside a Cloud. This increases the expenses at the benefit of fault tolerance. Our solution does not rely on the cloning of an entire virtual machine for fault tolerance. In our case the virtual image can host several VPSs, so that if one fails another one can take its place. For full fault tolerance (to protect also the operating system for example) one needs to deploy two virtual machines inside the Cloud. But they are used for entire collections of services, each possibly running inside its own VPS. Thus, our solution decreases the expenses while still providing extensive fault tolerance.

6. The security layer. Many times reliability (and, hence, dependability) of a LSDS depends not only on fault tolerance, but also security. If a control network is compromised due to poor security policies (lack of patches, slow patch cycle, etc), the reliability of the network is decreased. If an attacker can perform attacks (e.g., man-in-the-middle) and send commands that disrupt the functionality of the LSDS, its reliability is decreased. Nearly all security threats can be seen as threats to the systems reliability.

For modern LSDS the security features provided by the operating system are simply not enough. Various authors argue that access control mechanisms for safe execution of untrustworthy services should be based on security models such as Discretionary Access Control (DAC) [8] or Mandatory Access Control (MAC) [9]. Such models are at the basis of our security layer, which is designed to augment the virtualization infrastructure previously presented with security features. For fault tolerance each VPS contains replicas of the same processes. We next assumed that each process represents a service container. We augmented these containers with access control and policy enforcement mechanisms (see Figure 6.1).

In the DAC model the access to information is determined by the identity of subjects or groups [8]. In addition, the model assumes that subjects can pass permission to other subjects. The model suffers from various limitations. Users authorized to access some information may not be the owners of that information. This leads to situations where a compromised application can control resources far beyond the needs of that application. In information security the principle of least privilege requires that in a particular abstraction layer of a computing

environment, every module must be able to access only the information and resources that are necessary for its legitimate purpose. The DAC model lacks enforcements of the least privilege principle. It also lacks domain separations for users logged into the system.

The MAC model was considered more adequate for our purpose. In this model the access is restricted based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization of subjects [9]. This model is more adequate to be used for securing distributed services: it allows the domain separations of users and to enforce the least privilege principle. There are currently several research-level implementations of this model at OS level: SELinux, Smack or AppArmor are among the most advanced solutions for Linux-based systems [10].

Security Enhanced Linux (SELinux) implements for Linux a security model that combines Type Enforcement (TE) model, with Role-Based Access Control (RBAC) and Multi-Level Security (MLS) models. The Type Enforcement model provides fine-grained control over processes and objects in the system while the RBAC model provides a higher level of abstraction to simplify user management as stated in [11]. Similar to SELinux, Smack also implements the MAC security model [12]. It was intended to be a simple mandatory access control mechanism but it purposely leaves out the role based access control and type enforcement that are the major parts of SELinux. Smack is geared towards solving smaller security problems than SELinux, requiring much less configuration and very little application support. Smack permits creation of labels according to the security requirements of the system.

These solutions implement the MAC model. The security mechanisms in SELinux and Smack are based on inodes, while in AppArmor are based on file paths. If a program is restricted from accessing a particular le using AppArmor, a hardlink to that file would still provide with access, since the protection states only the original path of the file. From these three, SELinux has the most complex implementation, because it combines complex mandatory access controls such as those based on type enforcement (TE), roles and levels (RBAC) of security (MLS). Because SELinux provides more security mechanisms, because of its flexibility in design, we selected it for the case study of securing enforcement in case of several distributed services.

Security can be applied at various levels. Security mechanisms applied within the application layer have the advantage of high granularity, while protecting sensitive information, and permit construction of complex policies. The disadvantage of such mechanisms is the high overhead. The operating system (OS) is the one that mediates accesses initialized by applications to hardware components. Therefore, access control mechanisms applied at the OS layer can provide high level of granularity for protecting processes, files, sockets, etc. At this layer there are also various DAC mechanisms that are already used to protect services.

Our solution creates an orthogonal security policy which, together with the existing security mechanisms provided by the OS, can be used to enhance the security characteristic of a distributed system.

Services can be secured as other processes running inside a computer system. By reducing the services to a simple process we can better localize a security issue from the OS point of view. This is illustrated in Figure 6.1. The example considers three services. Each service runs in a separate service container (P1-3). The security policies and access rules are applied at the level of each container (in our example permission to access resources are specified as R1-3). Several such resources can be applied for each container, or several services can be protected by the same rule. In the example also we show the relation with the fault tolerance solution - the access control is performed at the level of each process, while the fault tolerance is applied at the level of sets of processes, running in virtual VPSs.

So each Pi process has permissions to access the resource Ri. The example illustrates the use of several security layers. The processes are first protected by the DAC mechanism provided at the OS mechanisms. If this layer is compromised, each service is further protected by an individual security policy. Furthermore, each service is enclosed inside a unique sandbox that does not include any other process. In this case, even if one service is compromised, the other services are still intact and further protected.

These security mechanisms were implemented using SELinux. Over the DAC security layer provided by many Linux flavors, we used the MAC security layer assured by SELinux. The SELinux solution already confines twelve daemons inside specific domains. Based on the provided security functions, we developed protection mechanisms for several services. The result is a system having an enhanced security characteristic because beside been protected from the damage made by the twelve daemons it further offers protection guarantees against damage made by the web container and the monitoring service. These services are confined in specific sandboxes according to their activities.

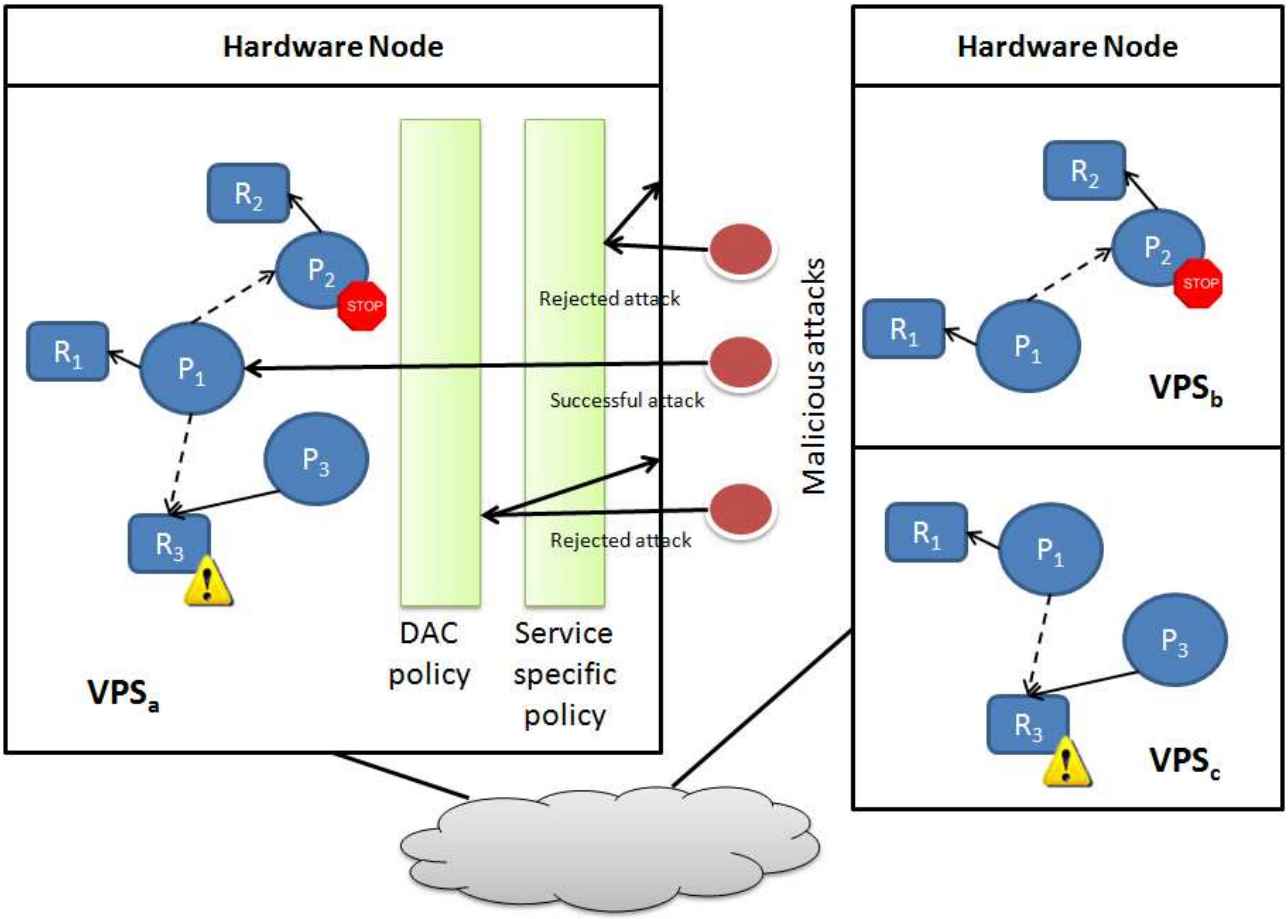

Fig. 6.1: Malicious attacks against services.

7. Evaluation results. In the sequel we present a series of results demonstrating the capabilities and performances of the proposed solutions. We conducted our tests on the NCIT testbed [22], an experimental Grid platform with advanced scheduling and control capabilities. The testbed is part of the national Grid collaboration and it includes several sites geographically distributed in various places in Romania. For the experimental setup, we considered the use of 10 nodes belonging to the NCIT Cluster at University Politehnica of Bucharest. The nodes are equipped with x86 64 CPUs running at 3 GHz, 2 GB RAM, interconnected via a 10 Gigabit Ethernet network. An additional set of 5 nodes situated at CERN, Geneva, were used to evaluate the overhead of the solution.

We evaluated the dependability mechanisms by combining virtualization with the use of the security mechanisms. The scenario involved an ApMon component, which is already used in real-world distributed infrastructures at the monitoring layer [13], and then the Proxy service previously used in evaluations, and that is designed to enhance the fault tolerance capability of distributed systems [14].

Monitoring services are encountered in many distributed systems, and, especially for fault-tolerance, one needs to have guarantees that the monitoring information is unaltered by malicious attackers. On the other hand, the Proxy service illustrates the mechanisms applied to protect a service container.

We first closed each service inside a SELinux sandbox, running on a virtual VPS that is its own domain. This domain confines inside any possible damage. For example, the ApMon is running within the *apmont* domain. To allow ApMon the access to monitor in this domain we specified how a process is allowed to run in the *apmont* domain, and what it is allowed to do. The entry point of the *apmont* domain is any executable file labeled with the type $apmon_exect$. Once a process executes this file, it transitions in the *apmont* domain and runs only under the allow rules of the domain. This example is illustrated in Figure 7.1. In the example

the executable file is */bin/apmon* and the allowed actions are reading the configuration file, accessing */proc* contents and sending monitoring information to a MonLISA service.

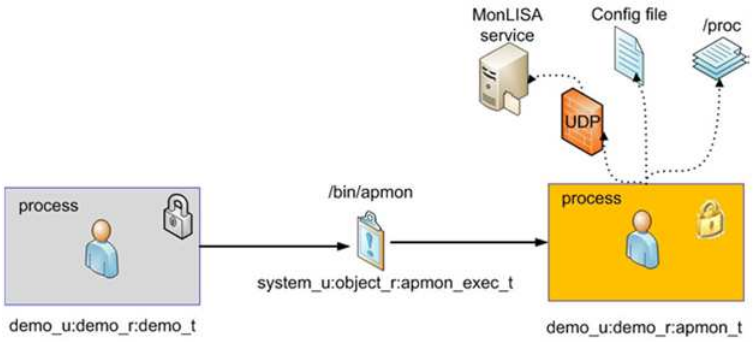

Fig. 7.1: Security mechanisms applied to an ApMon service.

A different situation is illustrated by the security mechanisms applied to the Proxy service. In this case we extended the enhancing security mechanisms to service containers, themselves running on different VPS nodes. What a service is allowed to do and what not sometimes differ greatly when treating security for services unitarily and independent from the container. A Proxy service may need access to some type of files, while a monitoring service wants for example access to other types. They can both run as applications requiring searching and loading dynamic libraries, memory execute permissions and network access. But one service might require some type of restrictions, while another might require completely different security limitations. In this case, we first developed the SELinux policies for specifying what the process represented by the service container (the current implementation is based on Tomcat as a web container for services) is allowed to do.

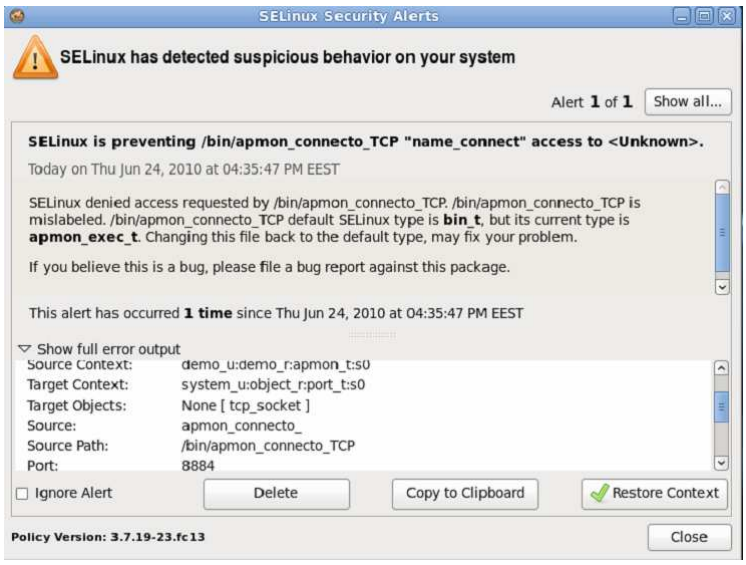

Fig. 7.2: The malicious ApMon application is denied connect access to port 8884.

We defined the domains and types of the policy that help to sandbox the restraint Tomcat container. For example, the Tomcat process only runs in the local system as daemon, started from the *initrct* domain, the SELinux domain for the *init* processes. This was further augmented with policies for the individual services running inside the container. The policies are enforced using two domain transitions before reaching the domain of a usual Java application. This idea was first introduced in [15].

We evaluated several situations in which failures and attacks from malicious code can be contained and avoid the damage of the whole system though validating imposed policies. We used experiments using from strict policies, where nothing is allowed to run and one has to define specific allow rules for each actions, to more relaxed ones, such as every subject and object can run uncontrolled except specific targeted daemons that are constrained by proposed rules.

A first experiment uses a verification that an ApMon application still works. In this scenario the ApMon application sends information about the system inspecting the */proc* directory. The monitored information is sent using datagrams to a MonaLISA farm. Next we tested what happens when ApMon is compromised and it tries to connect on another port. As expected, the system detects an attempt to break the imposed execution policy and interrupts the malicious activity (see Figure 7.2).

In another experiment we evaluated a Tomcat web service container augmented with the proposed dependability solutions. The results of these show experiments that the solution is able to preserve the defined policy. For example, in an experiment we were interested if the Tomcat web services container is able to contain the damage, in case of a situation where the monitoring service itself inside Tomcat suffers an attack and, as a consequence, it behaves maliciously and tries to send valuable information about requests received from the Proxy service somewhere else than it is supposed to. The reaction of the security policy is as expected thus the attempt of connecting on an incorrect port fails because the action was not specified as an allowed action (see Figure 7.3).

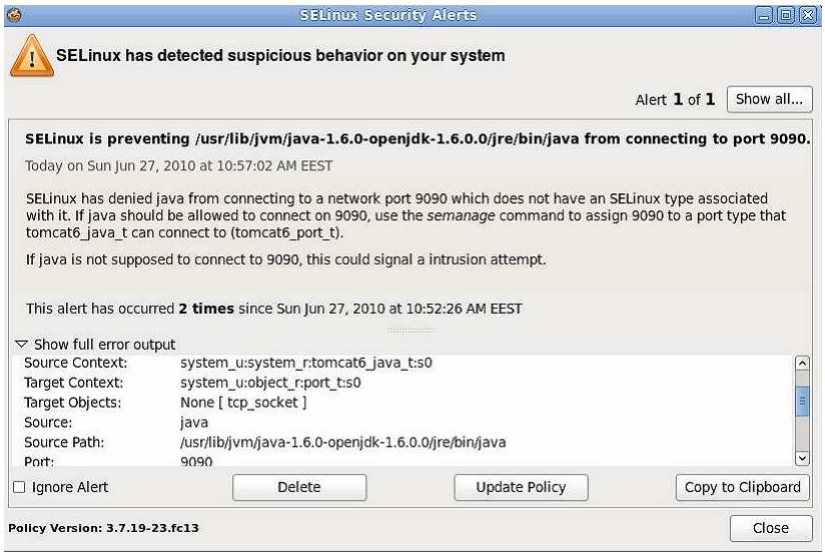

Fig. 7.3: Test service is denied to connect on port 9090.

We next continued with a series of experiments using LISA [24], a lightweight dynamic service that provides complete system and application monitoring. In this case the objective was to evaluate the overhead of the proposed solutions in terms of network traffic and the load of the systems on which services are running. The testbed involved three stations, two located in CERN, Switzerland and one in Romania. Each station hosts a VPS. We first assumed two services running on each of the nodes in Switzerland, and on the node in Romania we started the Jini service. At one point, a process fails in order to see how the system reacts to its failure.

During these experiments we measured a sustained additional traffic varying between 5 and 20 Kbps, while the machines load did not change significantly (see Figure 7.4). The measured values demonstrate that the overhead caused by running the processes is very low, both in terms of network traffic and CPU usage or system load.

These experiments reveal a good potential for integrating the services into various distributed environment for increasing dependability. The failure detection service is able to recognize various errors and, in the same time, assures good running performances. The detection is further augmented with the VPS solution that is able to provide transparent failure recovery, by resubmitting faulty requests to still-running service replicas. In

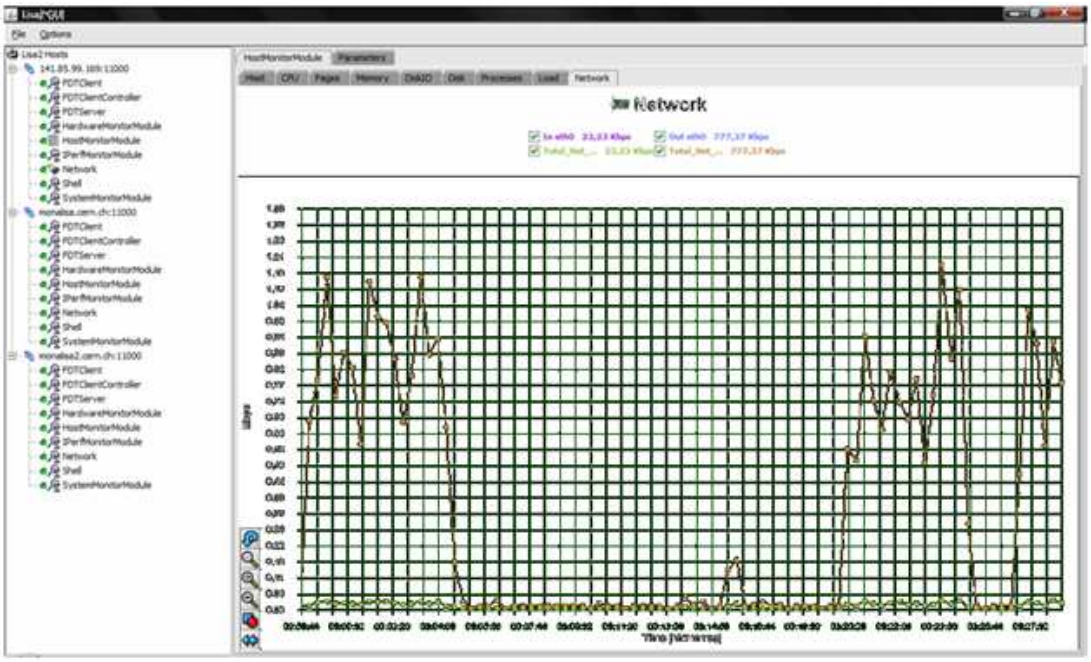

Fig. 7.4: The results for the evolution of the network traffic.

the end, security attacks are correctly recognized by the MAC mechanisms. Combined, these services provide increase reliability and availability properties.

8. Conclusions and future work. In this paper we presented an approach to ensuring dependability in LSDS using virtualization for fault-tolerance, augmented with advanced security models. Today dependability remains a key element in the context of application development and is by far one of the most important issues still not solved by recent research efforts.

Our research work is concerned with increasing reliability, availability, safety and security in LSDS. The characteristics of such systems pose problems to ensuring dependability, especially because of the heterogeneity and geographical distribution of resources and users, volatility of resources that are available only for limited amounts of time, and constraints imposed by the applications and resource owners. Therefore, we proposed the design of a hierarchical architectural model that allows a unitary and aggregate approach to dependability requirements while preserving scalability of LSDS.

We presented implementation details of such proposed methods and techniques to enable dependability in LSDS. Such solutions are based on the use of tools for virtualization and security, in order to provide increased levels of dependability. We proposed several solutions to increasing fault tolerance and enforcing security. The fault tolerance is based on the use of virtual containers, either in the form of virtual sandboxes running on top of the operating systems, but we are currently also working on proposing logic containers composed of various replicated services served by an intermediary Proxy service. We also presented solutions to introduce modern security models, such as MAC, to various distributed services. The security policies in this case are applied at various levels, by offering protection at operating system level, at service containers or further to the individual service.

Acknowledgments. The research presented in this paper is supported by national project "DEPSYS - Models and Techniques for ensuring reliability, safety, availability and security of Large Scale Distributes Systems", Project CNCSIS-IDEI ID: 1710. The work has been co-funded by national project "TRANSYS - Models and Techniques for Traffic Optimizing in Urban Environments", Contract No. 4/28.07.2010, Project CNCSIS-PN-II-RU-PD ID: 238, and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/89/1.5/S/62557. The contributions from all authors to this paper are equal.

REFERENCES

[1] V. Cristea, C. Dobre, F. Pop, C. Stratan, A. Costan, and C. Leordeanu, *Models and Techniques for Ensuring Reliability, Safety, Availability and Security of Large Scale Distributed Systems*, in Proc. of the 3rd International Workshop on High Performance Grid Middleware, the 17th International Conference on Control Systems and Computer Science, Bucharest, Romania, May 2009, pp. 401–406.

[2] H. Jin, X. Shi, W. Qinag, and D. Zou, *DRIC: Dependable Grid Computing Framework*, IEICE - Transactions on Information and Systems. Volume E89-D, Issue 2, February 2006, pp. 126–137.

[3] P. Guillaume, *Design of an Infrastructure for Highly Available and Scalable Grid Services*, D3.2.1. Technical Report, Vrije Universiteit, Amsterdam, 2006.

[4] J. Jayabharathy, and A. Ayeshaa Parveen, *A Fault Tolerant Load Balancing Model for Grid Environment*, Pondicherry Engineering College, Pondicherry, India, International Journal of Recent Trends in Engineering, Vol 2, No. 2, November 2009.

[5] B. Yagoubi, and M. Medebber, *A Load balancing Model for Grid Environment*, in Proc. of the 22nd International Symposium on Computer and Information Sciences (ISCIS 2007), Ankara, Turkey, 2007, pp. 1–7.

[6] I. Sommerville, S. Hall, and G. Dobson, *Dependable Service Engineering: A Fault-tolerance based Approach*, Technical Report, Lancaster Univ., 2005.

[7] J.P. Walters, and V. Chaudhary, *A fault-tolerant strategy for virtualized HPC clusters*, J. Supercomput., 50(3), Dec. 2009, pp. 209–239.

[8] S. Dranger, R.H. Sloan, and J.A. Solworth, *The Complexity of Discretionary Access Control*, in Proc. of the International Workshop on Security (IWSEC 2006), Kyoto, Japan, October 2006, pp. 405–420.

[9] H. Lindqvist, *Mandatory Access Control*, Master's Thesis in Computing Science, Umea University, Department of Computing Science, SE-901 87, Umea, Sweden, 2006.

[10] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, *Linux Security Modules: General Security Support for the Linux Kernel*, USENIX Security, Berkeley, CA, 2002, pp. 17–31.

[11] S. Smalley, *Configuring the SELinux policy*, NAI Labs Report #02-007, 2002.

[12] C. Shaufler, *The Simplified Mandatory Access Control Kernel*, Whitepaper, 2008.

[13] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, and C. Stratan, *MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems*, Computer Physics Communications, Volume 180, Issue 12, December 2009, pp. 2472–2498.

[14] M. Nastase, C. Dobre, F. Pop, and V. Cristea, *Fault Tolerance using a Front-End Service for Large Scale Distributed Systems*, in Proc. of 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, 2009, pp. 229–236.

[15] C. Hger, *Security Enhanced Linux - Implementierung und Einsatz*, Technical Report, Technical University Berlin, Complex and Distributed Systems, 2008.

[16] C. Fetzer, M. Raynal, and F. Tronel, *An adaptive failure detection protocol*, in Proc. of the 8th IEEE Pacific Rim Symp. on Dependable Computing, Lanzhou, Gansu, 2001, pp. 146–153.

[17] X. Defago, N. Hayashibara, and T. Katayama, *On the Design of a Failure Detection Service for Large-Scale Distributed Systems*, in Proc. of the Intl. Symp. Towards Peta-bit Ultra Networks (PBit 2003), Ishikawa, Japan, 2003, pp. 88–95.

[18] C. Dobre, F. Pop, A. Costan, M.I. Andreica, and V. Cristea, *Robust Failure Detection Architecture for Large Scale Distributed Systems*, in Proc. of the 17th International Conference on Control Systems and Computer Science (CSCS 17), Bucharest, Romania, 2009, pp. 133–141.

[19] L. Andrei, C. Dobre, F. Pop, and V. Cristea, *A Failure Detection System for Large Scale Distributed Systems*, in Proc. of 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2010), Krakow, Poland, 2010, pp. 482–489.

[20] L.M. Silva, J. Alonso, and J. Torres. *Using Virtualization to Improve Software Rejuvenation*, IEEE Trans. Comput. 58, 11 (November 2009), pp. 1525–1538.

[21] B. Jansen, H.V. Ramasamy, M. Schunter, and A. Tanner. *Architecting Dependable and Secure Systems Using Virtualization*. In Architecting Dependable Systems. Lecture Notes In Computer Science, Vol. 5135. Springer-Verlag, Berlin, Heidelberg (2008), pp. 124–149.

[22] The NCIT Cluster, *http://cluster.grid.pub.ro/*, Retrieved September 14, 2011.

[23] J.R. Douceur, and J. Howell. *Replicated Virtual Machines*. Technical Report MSR TR-2005-119, Microsoft Research, Sep 2005.

[24] I.C. Legrand, C. Dobre, R. Voicu, C. Cirstoiu. *LISA: Local Host Information Service Agent*. Proc. of the the 15th International Conference on Control Systems and Computer Science (CSCS-15), Bucharest, Romania, (2005).

Edited by: Dana Petcu and Jose Luis Vazquez-Poletti
Received: August 1, 2011
Accepted: August 31, 2011