A Virtualization-based Approach to Dependability for Service Computing

Ciprian Dobre[*], Florin Pop[*], Valentin Cristea[*], Ovidiu-Marian Achim[*]
[*] University POLITEHNICA of Bucharest, Romania
E-mails: { ciprian.dobre, florin.pop, valentin.cristea}@cs.pub.ro, ovidiu.achim@oracle.com

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.**

Keywords- virtualization, dependability, large scale distributed systems

I. 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. In [1] we previously proposed a solution designed to detect failures occurring in different parts of a distributed system. However, excFault recovery generally relies on some form of replication, checkpointing or other techniques. In this paper we extend our previous results and present a virtualization-based solution designed to facilitate fault recovery by freezing services running in good states and using virtual images for future recovery or migration of file systems, or processes.

We propose a virtualization approach to ensuring dependability by using checkpoint strategies and protection domains using virtual hosts, coupled with a proactively replication strategy necessary to maintain consistent states of the system in case of failures. The solution is part of the DEPSYS dependability architecture ([2]). Our solution assumes that on top of a typical operating system the services can run in specialized virtual environments. Such virtual environments can be easily 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 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 extended 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.

II. RELATED WORK

Other authors approached the problem of ensuring fault tolerance in LSDSs in different ways. Authors of [3] propose the use of an adaptive system for fault detection in Grids, together with a policy-based recovery mechanism. 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.

An alternative solution to fault tolerance is based on grouping several server nodes into a set that appears to clients as a single node [4]. 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 concentrate 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 [2]), 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 load balancing and fault tolerance in Grid systems is presented in [5]. 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 [6]. The model 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 [7], 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 fault-tolerant 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 such disadvantages.

III. AN ARCHITECTURAL MODEL FOR DEPENDABILITY IN LARGE SCALE DISTRIBUTED SYSTEMS

The proposed dependability approach is part of the architectural model that we proposed in [2]. The general approach to ensuring fault tolerance in LSDS consists of an extendable 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.

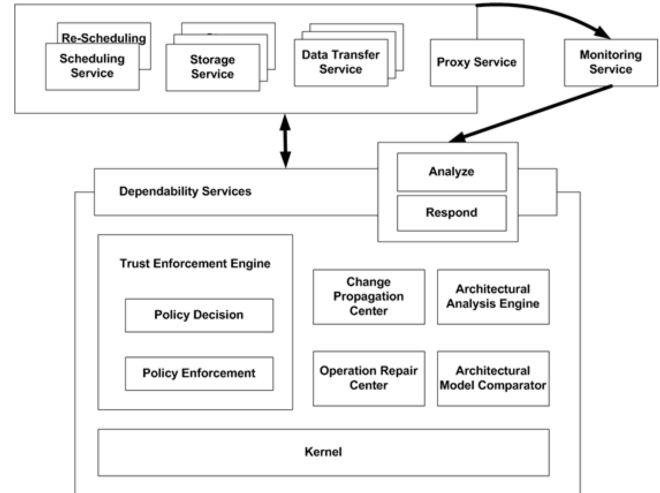

Fig. 1. The dependability architecture for LSDS.

An abstract model of the components making up the architecture at the middleware layer of the distributed system is presented in Figure 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.

IV. A VIRTUALIZATION-BASED APPROACH TO DEPENDABILITY

The virtualization layer includes several virtual servers (VPS) hosting services belonging to the distributed system (see Figure 2). There are two types of VPS nodes. A first type of node assumes the role of activity coordinator. This

node is responsible with triggering checkpoint, restore and load balancing actions. The other virtual nodes host the services belonging to the distributed system.

The virtual environment is developed with support for both OpenVZ and LXC virtual environments, and it is reinforced with security models assured by technologies such as SMACK or SELinux.

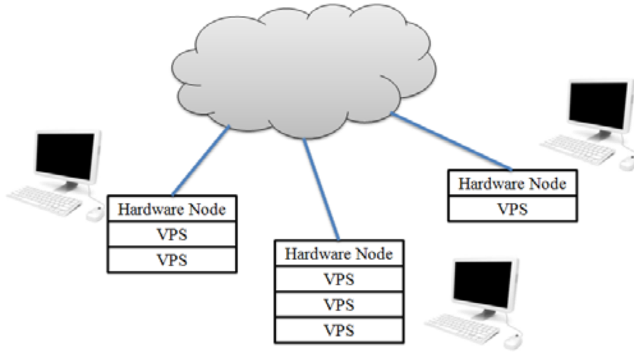

Fig. 2. The architecture of the system as composed by virtual environments.

OpenVZ ([8]) 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.

Table 1. The algorithm run by the coordinator.

```
procedure main {
        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
        }
}
```

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 such the algorithm. 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 Table 1. 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*.

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 can migrate VPS nodes on other station. The migration process is realized without interrupting the connection with the node. The same principle is applied for fault tolerance. When the coordinator detects failures with VPS, using the saved checkpoint it can initiate a recovery procedure and form new VPS nodes. These new VPS nodes will take over the faulty ones, and are initiated for failure tolerance on different adjacent nodes.

V. THE SECURITY LAYER

For modern distributed systems the security features provided by most operating systems are not enough. Because of their characteristics, assuring safe execution of untrustworthy distributed services requires the use of security features currently poorly used in operating systems, such as Discretionary Access Control (DAC) [9] or Mandatory Access Control (MAC) [10]. We analyzed existing security models and available technologies to support them. We were interested in how to best support the security enforcement requirements of modern distributed systems, as an extra layer added to the virtualization approach presented in this paper.

We analyzed several security enforcement solutions to use the one best fit to protect services even in the presence of various security threats. The DAC model states that the access to information is determined by the identity of subjects or groups [9]. In this model subjects can pass permission to other subjects. This model suffers from various limitations. For example, 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 such 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.

In the MAC model access is restricted based on sensitivity (as represented by a label) of the information contained in the objects and the formal authorization of

subjects [10]. 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 [11].

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 [12].

Like SELinux, Smack also implements the MAC security model [13]. 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.

All 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 4. In this example the three services, seen as processes, run in a computer system. Each of these processes has associated resources (files, sockets, etc).

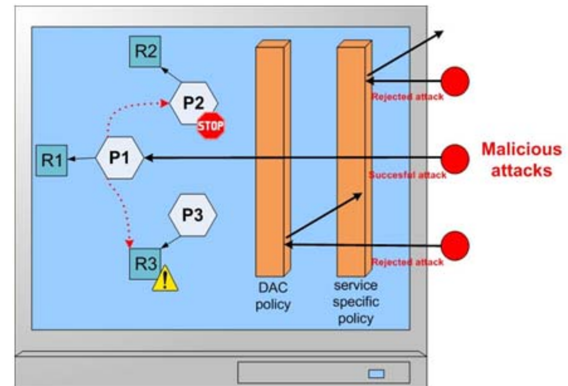

Fig. 4. Malicious attacks against services.

The P_i process has permissions to access the resource R_i. In this approach we propose 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.

VI. EVALUATION RESULTS

We evaluated the proposed dependability mechanisms by adding virtualization and security policies for an ApMon component already used in real-world distributed infrastructures at the monitoring layer [14], and then for a Proxy service designed to enhance the fault tolerance capability of distributed systems [15].

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 *apmon*$_t$ domain. To allow it

to monitor in this domain we specified how a process is allowed to run in the $apmon_t$ domain and what it is allowed to do. The entry point of the $apmon_t$ domain is any executable file labeled with the type $apmon_exec_t$. Once a process executes this file, it transitions in the $apmon_t$ domain and runs only under the allow rules of the domain. This example is illustrated in Figure 5. 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.

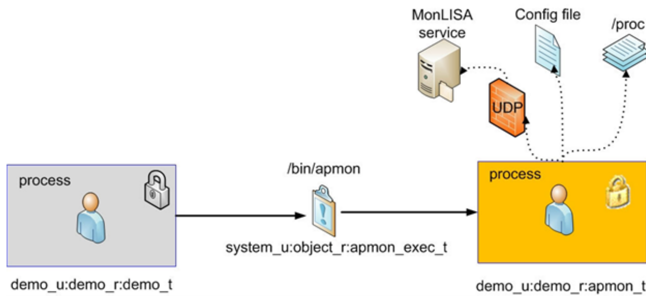

Fig. 5. Security mechanisms applied to an ApMon service.

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.

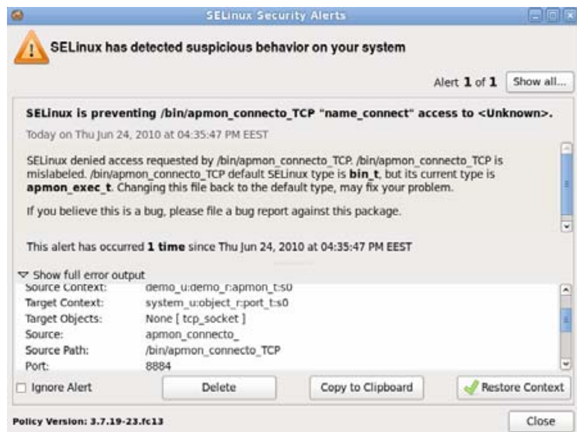

Fig. 6. The malicious ApMon application is denied connect access to port 8884.

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 $initrc_t$ 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 [16].

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 6).

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).

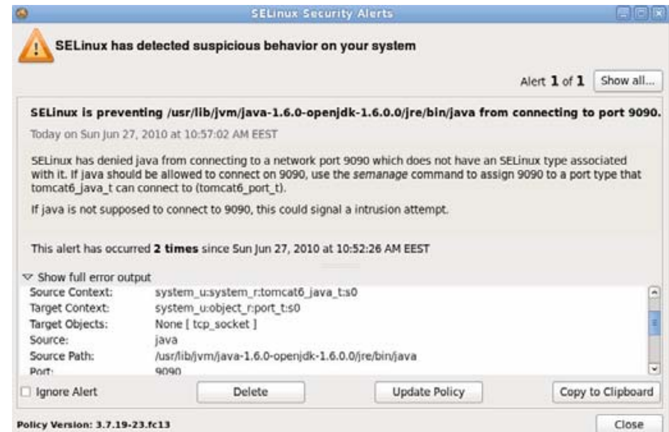

Fig. 7. Test service is denied to connect on port 9090.

VII. 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.

ACKNOWLEDGMENT

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] Andrei, L., Dobre, C., Pop, F., and Cristea, V.: A Failure Detection System for Large Scale Distributed System. in Proc. of The Fourth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2010), Krakow, Poland, February 15th - 18th, pp.482--489.(2010)

[2] Cristea, V., Dobre, C., Pop, F., Stratan, C., Costan, A., and Leordeanu, C.: Models and Techniques for Ensuring Reliability, Safety, Availability and Security of Large Scale Distributed Systems. 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.(2009)

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

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

[5] Jayabharathy, J., Ayeshaa Parveen, A.: 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).

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

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

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

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

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

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

[12] Smalley, S.: Configuring the SELinux policy. NAI Labs Report #02-007.(2002)

[13] Shaufler, C.: The Simplified Mandatory Access Control Kernel. Whitepaper.(2008)

[14] Legrand, I., Newman, H., Voicu, R., Cirstoiu, C., Grigoras, C., Dobre, C., Muraru, A., Costan, A., Dediu, M., and Stratan, C.: 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.(2009)

[15] Nastase, M., Dobre, C., Pop, F., and Cristea, V.: 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, pp229--236.(2009)

[16] Hger, C.: Security Enhanced Linux - Implementierung und Einsatz. Technical Report, Technical University Berlin, Complex and Distributed Systems.(2008).