A Dependability Layer for Large Scale Distributed Systems

V. Cristea, C. Dobre, F. Pop, C. Stratan, A. Costan, C. Leordeanu, E. Tirsa

Department of Computer Science,
University POLITEHNICA of Bucharest,
Spl. Independentei 313, Bucharest, Romania
Fax: 004021-318 10 01
E-mails: {*valentin.cristea, ciprian.dobre, florin.pop, corina.stratan, alexandru.costan, catalin.leordeanu, eliana.tirsa}@cs.pub.ro
*Corresponding author

Abstract: Ensuring dependability in large scale distributed systems represents an important research subject today. Despite the fact that many projects obtained valuable results in this domain, no acceptable solution was yet found that could integrate all the requirements for designing a dependable system and that could exploit all the capabilities of modern systems. We present a unitary and aggregate approach to ensuring reliability, availability, safety and security of distributed systems. Starting from the proposed architecture, we present implementation details for two solutions designed to ensure fault tolerance, using virtualization and container-based replication of services. We also present an approach to enhance security using combined modern security models in large scale distributed systems. The results and implementation details can serve as a methodology to assist distributed infrastructures in adopting such a middleware layer designed to enforce dependability in large scale distributed systems.

Keywords: dependability; large scale distributed systems; virtualization; security; fault tolerance.

Reference to this paper should be made as follows: Cristea, T., Dobre, C., Pop, F., Stratan, C., Costan, A., Leordeanu, C., and Tirsa, E. (2010) 'A Dependability Layer for Large Scale Distributed Systems', *International Journal of Grid and Utility Computing (IJGUC)*, Vol. 1, Nos. 3/4, pp.197–212.

Biographical notes: V. Cristea is a professor of the Computer Science and Engineering Department of the University Politehnica of Bucharest (UPB). He teaches courses on Distributed Systems and Algorithms. As a PhD supervisor he directs thesis on Grids and Distributed Computing. Valentin Cristea is Director of the National Center for Information Technology of UPB and leads the laboratories of Collaborative High Performance Computing and eBusiness. He is an IT Expert of the World Bank, Coordinator of national and international projects in IT, member of program committees of several IT Conferences (IWCC, ISDAS, ICT, etc), reviewer of ACM. He directs R&D projects in collaboration with multinational IT Companies (IBM, Oracle, Microsoft, Sun) and national companies (RomSys, UTI).

C. Dobre received his PhD in Computer Science at the University POLITEHNICA of Bucharest in 2008. He received his MSc in Computer Science in 2004 and the Engineering degree in Computer Science in 2003, at the same University. His main research interests are Grid Computing, Monitoring and Control of Distributed Systems, Modeling and Simulation, Advanced Networking Architectures, Parallel and Distributed Algorithms. He is member of the RoGrid consortium and is involved in a number of national projects and international projects. His research activities were awarded with the Innovations in Networking Award for Experimental Applications in 2008 by the Corporation for Education Network Initiatives (CENIC).

F. Pop, PhD, is assistant professor of the Computer Science and Engineering Department of the University Politehnica of Bucharest. His research interests are oriented to: scheduling in Grid environments (his PhD research), distributed system, parallel computation, communication protocols and numerical methods. He received his PhD in Computer Science in 2008 with Magna cum laudae distinction. He is member of RoGrid consortium and participates in several research projects in these domains, in collaboration with other universities and research centers from Romania and from abroad developer. He has received an IBM PhD Assistantship in 2006 (top ranked 1st in CEMA out from 17 awarded students) and a PhD Excellency grant from Oracle in 2006-2008.

S. Corina finished her PhD studies at the Computer Science and Engineering Department of the University Politehnica of Bucharest. Her research fields are Grid Computing, Workflows and Service Oriented Architectures. She is participating in several research projects in these domains, in collaboration with other universities and research centers from Romania and from abroad (CERN, Caltech, the Computer Science Research Institute from Bucharest, TU Delft). She received an IBM PhD Fellowship in 2006 and 2007.

A. Costan is a PhD student and Teaching Assistant at the Computer Science department of the University Politehnica of Bucharest. His research interests include: Grid Computing, Data Storage and Modeling, P2P systems. He is actively involved in several research projects related to these domains, both national and international, from which it worth mentioning MonALISA (in collaboration with Caltech and CERN), MedioGRID, EGEE, P2P-NEXT. His PhD thesis is oriented on Data Storage, Representation and Interpretation in Grid Environments. He has received a PhD Excellency grant from Oracle in 2006 and was awarded an IBM PhD Fellowship in 2009.

C. Leordeanu received his MSc in Computer Science in 2009 and is currently a PhD student at Politehnica University of Bucharest. His research interests include Distributed Systems, Security, Intrusion Detection. He is involved in a number of national and international research projects on these subjects.

E. Tirsa received her MSc in Computer Science in 2009 and is currently a PhD student at Politehnica University of Bucharest. She is involved in a number of national and international research projects on subjects such as Distributed Systems and Fault Tolerance.

1 Introduction

Both in the academic and industrial environments there is an increasing interest in large scale distributed systems, 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 ẗraditionalD̈istributed systems do not satisfy. Together with the extension of the application domains, new requirements have emerged for large scale distributed systems. Among these requirements, reliability, safety, availability, security and maintainability are needed by more and more modern distributed applications.

Although the importance of dependable systems is widely recognized and many research projects have been initiated in this domain, there are no mature implementations of these concepts available yet; the existing systems offer only partial solutions, and often the approaches separate the issues of reliability, availability, security etc. In this we present solutions to enabling dependability in service-based distributed systems. We present implementation details of several of the models, methods and techniques to enable dependability in large scale distributed systems previously proposed in (Cristea et al., 2009). We present solutions to enable fault tolerance using virtualization and container-based replication of services. We also present solutions to enhance security using combinations of several security models. Such solutions are based on the use of tools for communication, monitoring, scheduling, virtualization, management and accessibility (retrieval, efficient transfer) of data, and security, in order to provide increased levels of dependability.

The rest of this paper is organized 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 a virtualization-based approach to dependability. Section 5 presents solutions designed to secure services, using modern security models, in large scale distributed systems. Section 6 presents details on the implementation of a fault-tolerant Proxy service designed to mask possible faults occurring in a set of replicated services. In Section 7 we conclude and present future work.

2 Related Work

Most of the approaches to the dependability problem are mainly focused on fault detection and fault tolerance. The authors of (Jin et al., 2006) introduce an adaptive system for fault detection in Grids and a policy-based recovery mechanism. Fault detection is achieved by monitoring the system, and for recovery several mechanisms are available, among which task replication and checkpointing. In (Sommerville et al., 2005) the authors present a solution for web services, based on the implementation of a fault-tolerant container; the fault-tolerant containers manage a set of replicated services, which can be offered by external providers as well. The

containers can be configured for various fault tolerance strategies (either an equivalent service is invoked when a service fails, or multiple equivalent services are invoked from the beginning and a voting mechanism is applied etc.).

Another approach less related to large scale distributed systems is presented in (Cox et al., 2006), which addresses the tolerance to hardware faults through virtualization; the proposed solution is named Loosely Synchronized Redundant Virtual Machines (LSRVM). In (Grimshaw et al., 2005) the authors emphasize on the idea of survivability, which consists in providing, in case of a error, an alternative service, even if the latter doesn't fully satisfy the initial user requirements. For requirements specification, the authors propose DESL (Dependability Exchange and Specification Language), an XML based language; DESL isn't fully specified yet.

Other two papers in this field are (Sonnek and Weissman, 2005), which proposes a reputation based classification of Grid services, similar to the one in peer-to-peer systems, and (Rilling, 2006), which presents a Grid operating system architecture, with self-healing properties. In (Neocleous et al., 2006) the authors carry out a detailed study on errors in Grid systems, providing case-studies from the EGEE (Enabling Grids for E-Science) European project.

Another aspect of dependability is security. Currently there are several solutions for providing security for Grid environments and for distributed systems in general. In case of Grid systems, in (Foster et al., 1998) the authors identify a set of base requirements (single authentication, credentials protection, interoperability with local security solutions, etc) and proposed both an architectural model for security and a public key cryptography infrastructure based on a reference implementation (Grid Security Infrastructure - GSI).

The existing solutions for security in large scale distributed systems have various unresolved issues (Arenas, 2006). The majority of Grid systems were initially designed for scientific collaboration between participants that knew each others. This implies an implicit trusting relationship, all partners sharing a common goal - for example to carry out a scientific experiment - and it is implicitly assumed that the resources are provided and shared according to some well defined and respected rules. But when used in industrial environments such systems must satisfy the need to share resources with unknown groups, which could generate certain risks. A possible solution is to impose certain mechanisms to preserve the identity even in this situation.

3 Architectural Model

3.1 Functional Aspects

The proposed dependability layer is based on the architectural model previously proposed in (Cristea et al., 2009). The general approach to ensuring fault tolerance in large scale distributed systems consists of an extendable architecture that integrates services designed to handle a wide-range of failures, both hardware and software (Grimshaw et al., 2005; Cox et al., 2006). These services can detect, react, and confine problems such that to minimize damages, can predict and learn such that to increase the characteristic of the distributed system to survive. They include services designed to reschedule jobs when resources on which they execute fail, services capable to replicate their behavior such that 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.

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 the 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. It also orchestrates the survivable and redundancy mechanisms described below. The modularity of this component facilitates its adoption in real-world existing Grid systems. Through this component the other component scan interact with other technologies specific to large scale distributed systems.In this we present implementation details for such solutions designed to assure, at this level, a dependable execution environment.

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 developing 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.

We also adopted a survivability approach in which we provide several services such that when one cannot tolerate faults or security attacks another one takes its place. The system survives by learning the conditions which eventually led to the fault of a service or resource.

The architecture contributes to the security aspect using combined modern security models in large scale distributed systems. The results and implementation details can serve as a methodology to assist distributed

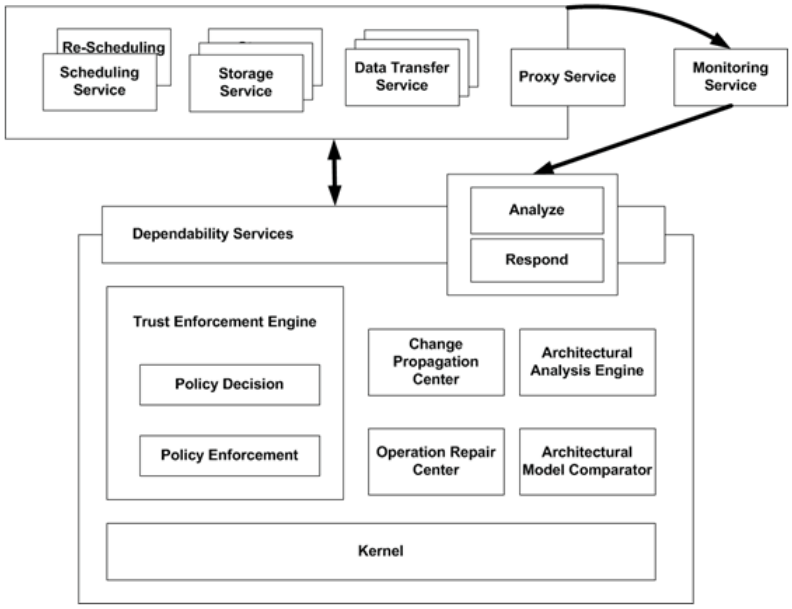

Figure 1 The architecture.

infrastructures in adopting such a middleware layer designed to enforce dependability in large scale distributed systems.

3.2 Components of the Architecture

In order to offer technologies capable to automatically correct problems and that are capable of taking intelligent decisions we designed a solution based on the use of architectural patterns for correcting faults. Such patterns have a special role in the context of monitoring, being used in a process of automatically detect faulty behavior of resources or various other problems and take higher-level decisions to start adequate repairing steps. For that, the *Analyze and Respond* services (see Figure 1) continuously analyze real-time data collected from an external monitoring system. The data is online processed and compared against a set of patterns (existing and learned as the system adapts to changing environment).

The component responsible with the real-time analysis of monitored data is the *Architectural Analysis Engine*. The analyzer uses the services provided by the *Architectural Model Comparator*. When faults are detected the *Operation Repair Center* further initiates the appropriate correction operation. This can be, for example, the instantiation of an alternative service that can supplement the activity offered by the faulty service or could represent an instruction sent to a *Scheduling Service* to reschedule faulty jobs. Also, by integrating various existing services, if the underlying distributed environment is capable of executing checkpointing operations (as in case of Condor) then the *Operation Repair Center* can initiate an action to recover the last state using the checkpointing data. Next, once the repair action is determined it must be correctly propagated throughout the distributed system. This is accomplished by the *Change Propagation Center* component.

The architecture involves communication with services providing functionalities in various parts of a Grid system. The data is collected by a monitoring service. The checkpointing capability involves dealing with a storage service. The state of the system is received from a central indexing service of the underlying middleware. The fault of a job can initiate a dialog between the services provided by the presented architecture and a scheduling service. The detection of faults, as well as propagation of recovery actions means using a data transfer service.

In the following sections we present several solutions developed within the presented architecture. They provide the capability to construct specialized containers, to monitoring the distributed system, to detect faults, as well as bringing the capability to re-schedule applications for execution when failures do occur.

4 A virtualization-based approach to dependability

Fault tolerance represents a critical requirement for modern distributed systems. 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 (Andrei et al., 2010) we previously proposed a solution designed to help better detect various types of failures occurring in different parts of a distributed system. Fault recovery generally relies on some form of replication, checkpointing or other techniques. We 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.

The dependability architecture previously presented consists of solutions designed both at the middleware (or application) and at the OS layers of the infrastructure. At the latter layer, the proposed approach 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, of course, 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. Examples are presented in the next Section.

The virtualization layer includes two components (presented in Figure 2). A first type of node assumes the role of activity coordinator. It is responsible with triggering checkpointing, restoring and load balancing actions. The rest of the virtual nodes host services pertaining to the distributed system. The virtual environment is based on the use of OpenVZ or LXC virtual environments, reinforced with security models assured by technologies such as SMACK or SELinux.

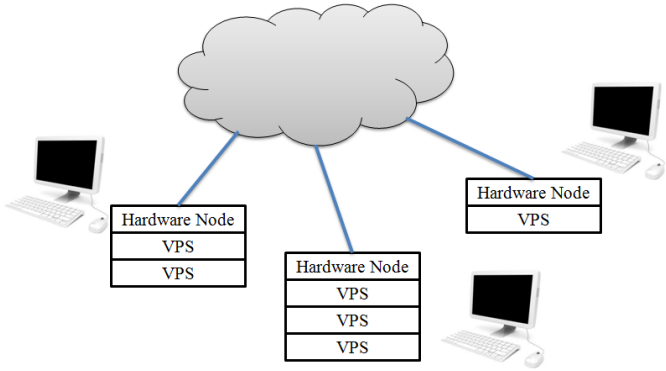

Figure 2 The architecture of the system as composed by virtual environments.

OpenVZ (Walters and Chaudhary, 2009) 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. For consistency, the implementation uses a modified coordinated checkpointing algorithm (the coordinator acts as mediator and initiator of such an algorithms), based on a Two-Phase Commit protocol. In this approach the coordinator sends a $VOTE\_REQUEST$ to all nodes. If it receives from all clients a $VOTE\_COMMIT$ than a consistent global checkpointing is possible and, thus, it sends $GLOBAL\_COMMIT$. If any of the 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 will try to determine what happened to the nodes that did not respond. Depending on the answer, it will decide whether or not they are further kept in the list of nodes.

When a 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, the coordinator initiates the restore and if all nodes agree the system is restored to a consistent state.

5 Securing Services in Distributed Systems

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) (Danger et al., 2006) or Mandatory Access Control (MAC) (Lindqvist, 2006). 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. We present a case study of security enforcement solutions that we believe, as a result of our analysis, are 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 (Danger et al., 2006). 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. The DAC model lacks enforcements of the least privilege principle and domain separations for users logged into the system (Danger et al., 2006).

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 (Lindqvist, 2006). 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 (Wright et al., 2002).

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 (Smalley, 2002). The MLS model (Haines, 2010) is an implementation based on Bell-La Padula (BLP) access levels, each level having constraints for the operations of read and write. In this model a level is allowed to read information from the levels having lower or equal security priorities and write information in a file belonging only to levels with a higher or equal security priority.

An alternative implementation of a mandatory access control model is AppArmor (Bauer, 2009), a security software maintained and released by Novell under GPL. While SELinux confines any active entity inside a domain, allowing it to access only specified resources of a particular type within the system, AppArmor confines programs to a limited set of listed files. AppArmor lacks the type enforcement model, and one cannot have access on files of a particular type but can only access the files directly enumerated.

Like SELinux, Smack also implements the MAC security model (Shaufler, 2008). 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 three 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 file 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. In this we propose a solution that 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 3. In this example the three services, seen as processes, run in a computer system. Each of these processes has associated resources (files, sockets, etc). For example, the *Pi* process has permissions to access the resource *Ri*. 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.

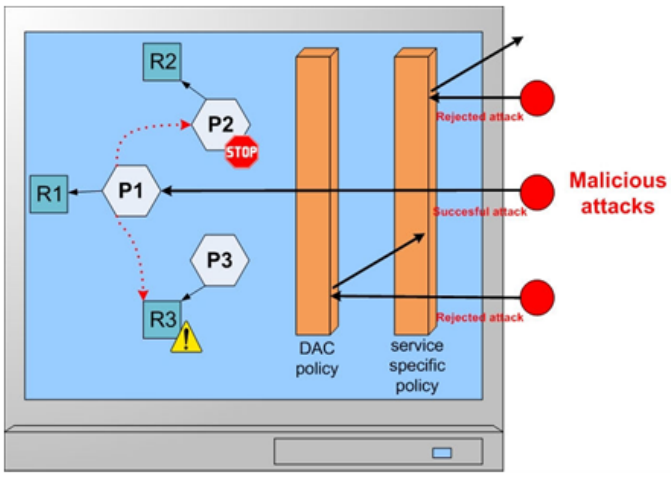

Figure 3 Malicious attacks against services.

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 (Haines, 2010). Based on the provided security functions, we developed protection mechanisms for two services. We illustrate how to further enhance security for various other distributed 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.

The implications of using a Proxy service for fault tolerance together with the described security mechanisms are few. The Proxy service is designed to forward all requests to a replicated service instance, where the security mechanisms are verified. If an exception occurs, the Proxy simply forwards the condition back to the client. The protection of the Proxy service itself is kept to a minimum, because no direct invocation is needed to this particular service.

We evaluated these mechanisms on two services: an ApMon service already used in many real-world distributed infrastructures at the monitoring layer (Legrand et al., 2009), and the Proxy service designed to enhance the fault tolerance capability of distributed systems (Nastase et al., 2009). 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 security mechanisms applied to protect a service container.

We first closed each service inside a SELinux sandbox 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 must specify how a process can be 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 is illustrated in Figure 4). 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.

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. 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 implementation was based on Tomcat as a web container for services) is allowed to do. 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 (Hger, 2008).

6 A Container-Based Service Engineering Approach to Fault Tolerance

Fault tolerance in distributed systems can be achieved using replication of services. When such a service fails to execute its function correctly, another one of its replicas can take its place. The problem is how to better access the replicas.

The problem is generally solved using a catalogue of services. This approach has several disadvantages. The catalogue should reflect as accurately as possible the situation of the services. If services accidentally leave the system the catalogue should be updated appropriately. For the application using the replicated services, without a specialized monitoring service, it is hard to decide which replica to chose. Then, the application accessing the service has to be aware about replication. Its code should implement the algorithm to choose a replica service, possible using a monitoring service.

We present an approach based on the use of a Proxy service. This service intercepts requests from clients and increases fault tolerance by directly addressing replicated services. It is composed of several components (Figure 5). The *SOAP Listener* handles incoming requests. It connects itself to the service container, intercepts all requests and can redirect them to other services.

The *Proxy Service* implements the mechanism to address the replicated services. For load balancing the service uses data gather by a monitoring service. For fault tolerance it uses a specialized failure detection service. The service forwards the request to one replica and the response back to the client. The *Monitoring* service is responsible with monitoring the status of services and resources. Instances of this service run in each container where replicas exist. The data about services and resources is further collected and provided to the Proxy service.

In this approach the client invokes the real service (step 1 in Figure 5). The request is sent to the service container where the client considers the real service is executed. In the service container the SOAP Listener intercepts the request (step 2) and redirects it to the Proxy service (step 3). This service implements the mechanism to choose the best replica. For load balancing the decision is based on the data obtained from the monitoring service (step 4). The failure detector also can instruct the Proxy service when a service replica is no longer available (step 5). After deciding which

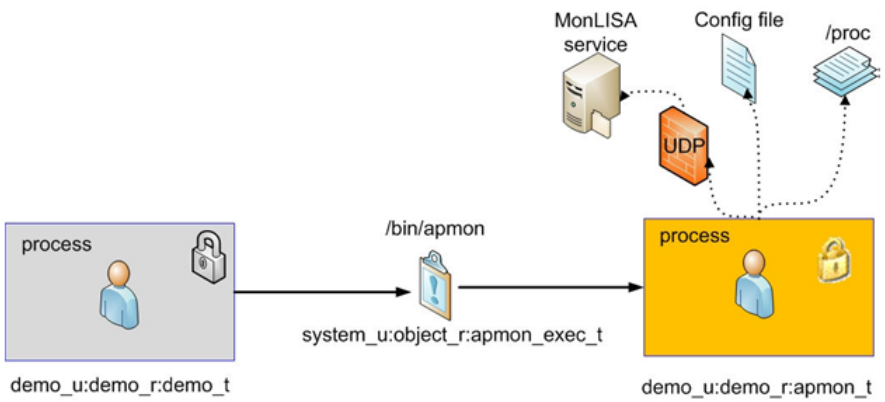

Figure 4 Security mechanisms applied to an ApMon service.

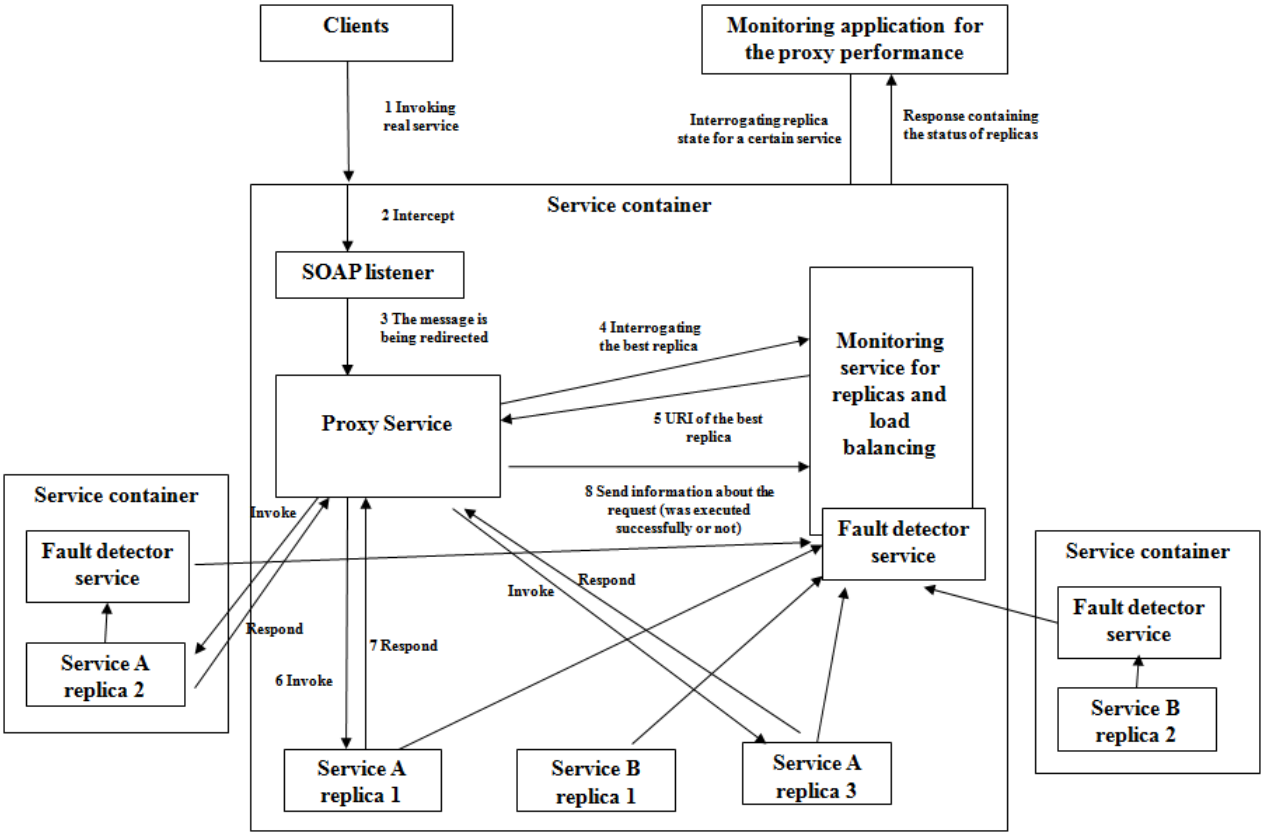

Figure 5 The Proxy Service Engineering Approach.

replica to access, the Proxy service redirects the request to the real service (step 6). The response (step 7) is forwarded back to the client. The Proxy service can send to the monitoring service data about possible exception being generated in this process (step 8). It also sends information about the quality of the community with that particular service. This data is used by the monitoring service to better analyze the capacities of the replicated service to handle request in future requests.

In the implementation the monitoring service is integrated in the Axis service container. This approach allows it to collect data from the container itself about other services. The service is also capable to collect data about Grid services. For that, it uses the Index, Java WS Core and MDS2 components from the Globus Toolkit. The monitoring of resources is handled by MonALISA farms (Legrand et al., 2009). The service registers to MonALISA and collects data about the resource usage.

The SOAP Listener component is implemented as a filter registered on the general request chain of an Axis service container. In its default implementation the Proxy service implements a round-robin rule to address the replicated services. However other addressing rules can easily be added. This service engineering approach has several advantages. The Proxy service ensures load balancing and fault tolerance. Load balancing can increase the performance of the system. The fault tolerance increases the availability of the system. The Proxy also ensures scalability of the applications being served. The implementation of the Proxy service does not require the invoking application or the invoked service to be modified.

In order to demonstrate the advantages of using the Proxy service we implemented a pilot testbed implementation. For testing the service we first implemented and deployed a service, *TestService*, which simulates the behavior of a real computational service. This service is used to emulate serving a particular request in a certain amount of time. It also includes the capability to trigger exceptions in a controllable manner. We also developed a simulator designed to mimic the client's behavior. This works by generating requests to the service. The requests are intercepted by the Proxy service and redirected to a replica service. In these experiments the client simulator runs inside the same container with the Proxy service and with the *Monitoring* and *Load-balancing* services.

Using this scenario we conducted a number of experiments. In order to demonstrate the capability to mask errors we conducted an experiment were we sent 100 requests to the *TestService*, with a frequency of three requests per second, with one replica being set as an error generator. We monitored the number of errors reported back to the client and it constantly, throughout the entire experiment, was equals to zero. All retransmission to other replicas of requests due to faults occurring with the faulty replica were completely masked to the client.

In the next experiment we demonstrated that the rate of error masking depends on the number of requests being received by the service. In this experiment the client sent requests with a frequency of five requests per seconds. Again one replica was set to continuously generate errors. Because in this experiment the number of requests exceeded the processing capacity of the two replicas working correctly these requests were occasionally sent to the faulty replica. That replica generated errors for all incoming requests and the Proxy service was then trying to retransmit them as a consequence to other replicas. When all replicas became overloaded the Proxy service sent back the generated error to the client. When at least one replica was not fully loaded with requests it could still serve the requests forwarded from the faulty replica and the client did not again see the generated error. In this way the error masking, although not completely transparent, was better than in the case when not using the proxy service at all.

Another obtained observation is that the time complexity of the data processing does not affect the perf-ormance of the Proxy service. This is due to the fact that the Proxy does not process the entire content of a packet; it stops at the SOAP envelope.

When using the proxy system the number of errors transmitted to the client decreases considerably. For the experiments where the replicas produce errors they are masked by redirected requests to other replicas, all transparently for the client. In this case the number of requests processed by replicas increases. In Figure 6 we represented the capacity to mask errors of the proxy system. We executed each experiment by varying the number of requests being sent per second. The horizontal axis represents the experiment being performed, their complexity increasing from left to right. In the figure, with green is the number of errors experimented by the client. With red we represented the number of errors experimented by the replica services. With blue we represented the number of requests being sent per second. As seen, in all experiments the client receives few or no errors. An important factor for masking errors is represented by the number of requests received by the proxy system per second, as well as the number of replicas generating errors for a particular request.

The obtained results prove that the proposed solution ensures a high degree of availability and reliability for a wide range of service-based distributed systems.

The next step will consists in the evaluation of the proposed solution using real settings. We will conducted tests on the NCIT testbed NCIT (2010), a large-scale experimental Grid platform, with advanced scheduling and control capabilities, part of the national Grid collaboration covering several sites geographically distributed in Romania. For our experimental setup, we will use the 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, will be used to evaluate the overhead of the solutions.

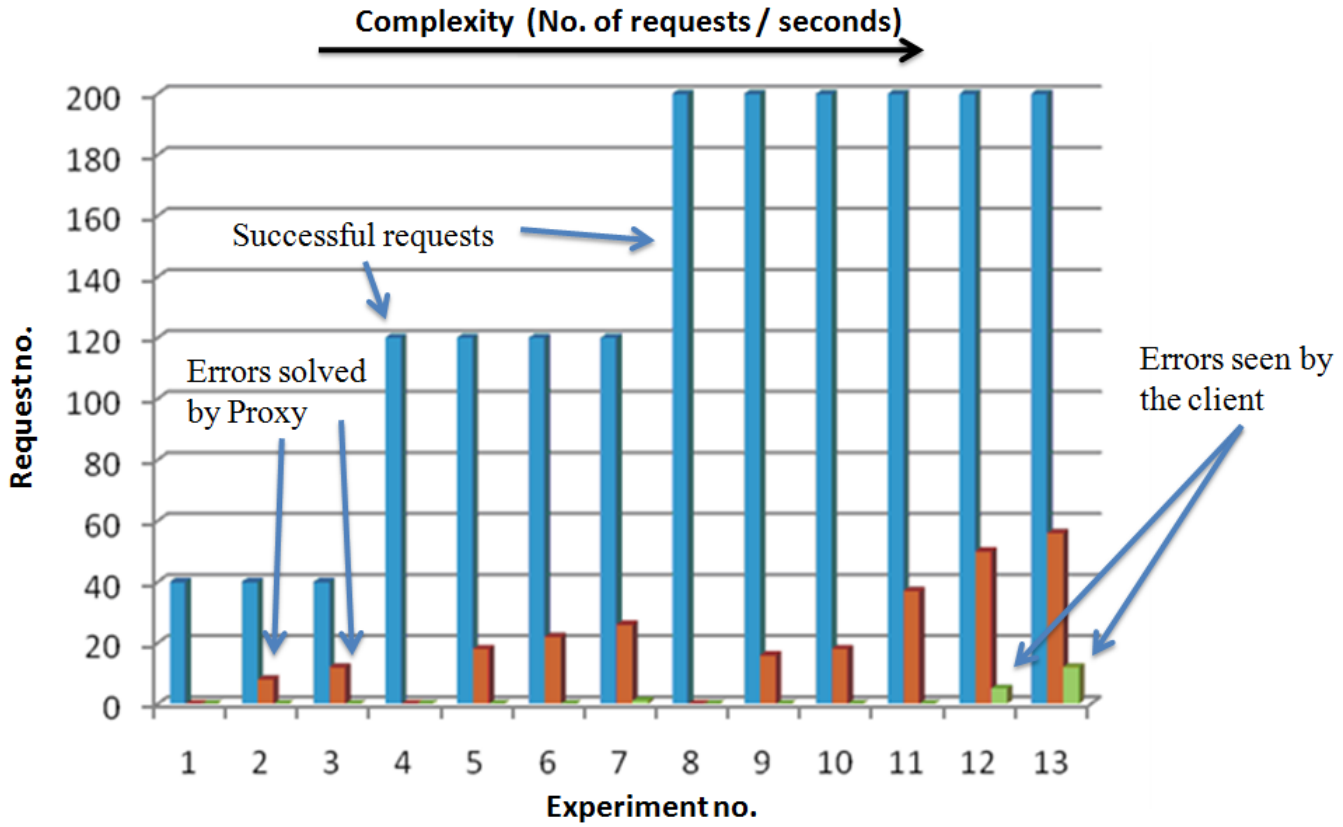

Figure 6 Masking of errors.

For monitoring we will use the MonALISA framework, the first results being presented in Costan et al. (2010).

7 Conclusion

In this paper we presented several solutions designed to increase dependability in large scale distributed systems. 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 work is concerned with increasing reliability, availability, safety and security, particularly in Grids and Web-based distributed systems. The characteristics of these 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 large scale distributed systems.

We presented implementation details of the models, methods and techniques to enable dependability in large scale distributed systems. Such solutions are based on the use of tools for communication, monitoring, scheduling, management and accessibility (retrieval, efficient transfer) of data, and security, in order to provide increased levels of dependability. Starting with this design, we proposed several solutions to increasing fault tolerance and enforcing security. The fault tolerance is based on the use of containers, either in the form of virtual sandboxes running on top of the operating systems, or 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.

8 Acknowledgements

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 (618/15.01.2009).

References

Cristea, V., Dobre, C., Pop, F., Stratan, C., Costan, A., and Leordeanu, C. (2009) '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.

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

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

Cox, A., Mohanram, K., and Rixner, S. (2006) 'Dependable $\neq$ Unaffordable', *1st workshop on Architectural and system support for improving software dependability (ASID'06)*, San Jose, California, 2006, pp.58–62.

Grimshaw, A., Humphrey, M., Knight, J.C., Nguyen-Tuong, A., Rowanhill, J., Wasson, G., and Basney, J. (2005) 'The Development of Dependable and Survivable Grids', *2005 Workshop on Dynamic Data Driven Applications*, Emory University, Atlanta, GA, May 22-25, 2005. pp.729-737.

Sonnek, J. D., and Weissman, J.B. (2005) 'A Quantitative Comparison of Reputation Systems in the Grid', *in Proc, of the 6th IEEE/ACM International Workshop on Grid Computing*, Seattle, Washington, USA, 2005, pp.242–249.

Rilling, L. (2006) 'Vigne: Towards a Self-Healing Grid Operating System', *in Proc, of Euro-Par 2006*, Dresden, Germany, Lecture Notes in Computer Science, Springer, vol. 4128, August 2006, pp.437–447.

Neocleous, K., Dikaiakos, M.D., Fragopoulou, P., and Markatos, E. (2006) 'Grid Reliability: A study of Failures on the EGEE Infrastructure', *in Proc. of the CoreGRID Integration Workshop*, Krakow, Poland, 2006, pp.3–11.

Foster, I., Kesselman, C., Tsudik, G., and Tuecke, S. (1998) 'A Security Architecture for Computational Grids', *in Proc. of the Fifth Conf. Computer and Communications Security*, ACM, San Francisco, CA, USA, 1998, pp.83–92.

Arenas, A. (2006) 'State of the art survey on trust and security in Grid computing systems', *Technical Report (RAL-TR-2006-008)*, Council for the Central Laboratory of the Research Councils, UK, March 2006.

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

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

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

Smalley, S. (2002) 'Configuring the SELinux policy'. *NAI Labs Report #02-007*, June 2002.

Haines, R. (2010) 'The SELinux Notebook. The Foundation', 2010.

Bauer, M. (2009) 'Paranoid penguin: AppArmor in Ubuntu 9'. *Linux J. 2009*, 185(9), Sep. 2009.

Shaufler, C. (2008) 'The Simplified Mandatory Access Control Kernel', *Whitepaper*, Last accessed July 14, 2010, from http://schaufler-ca.com/data/SmackWhitePaper.pdf, 2008.

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

Nastase, M., Dobre, C., Pop, F., and Cristea, V. (2009) '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, September 26-29, 2009, pp229–236.

The NCIT Cluster, http://cluster.grid.pub.ro/, Retrieved October 14, 2010.

Hger, C. (2008) 'Security Enhanced Linux - Implementierung und Einsatz', *Technical Report*, Technical University Berlin, Complex and Distributed Systems, 2008.

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

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

Costan, A., Dobre, C., Pop, F., Leordeanu, C., Cristea, V., 'A Fault Tolerance Approach for Distributed Systems Using Monitoring Based Replication', In *Proc. of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing*, Cluj-Napoca, Romania, pp. 451–458, August 2010.

Note

[1]http://www.kakadusoftware.com