A Fault Tolerance Approach for Distributed Systems Using Monitoring Based Replication

Alexandru Costan, Ciprian Dobre, Florin Pop, Catalin Leordeanu, Valentin Cristea
University Politehnica of Bucharest
Computer Science Department
313, Splaiul Independentei, 060042 Bucharest, Romania
Email: {alexandru.costan, ciprian.dobre, florin.pop, catalin.leordeanu, valentin.cristea}@cs.pub.ro

***Abstract*—High availability is a desired feature of a dependable distributed system. Replication is a well-known technique to achieve fault tolerance in distributed systems, thereby enhancing availability. We propose an approach relying on replication techniques and based on monitoring information to be applied in distributed systems for fault tolerance. Our approach uses both active and passive strategies to implement an optimistic replication protocol. Using a proxy to handle service calls and relying on service replication strategies, we effectively deal with the complexity and overhead issues. This paper presents an architecture for implementing the proxy based on monitoring data and the replication management. Experimentation and application testing using an implementation of the architecture is presented. The architecture is demonstrated to be a viable technique for increasing dependability in distributed systems.**

I. Introduction

Reliability and availability, also referred as dependability, are the main concerns when dealing with large scale distributed systems and applications. These requirements are continually increasing both in scientific and commercial applications such as finance, booking reservation, industrial control and telecommunication. One solution for achieving fault-tolerance is to build software on top of fault-tolerant (replicated) hardware. Although this may indeed be a viable solution for some domains, it is tightly coupled to the applications' use patterns and can hardly be reused or scale. Moreover, economic factors have prevented the large adoption of such approaches. In contrast, more efficient software based fault-tolerance techniques rely on replication enhanced with some management support for control and consistency. While this principle is readily understood, the methods required to implement replication raise non-trivial problems.

From a fault-tolerance perspective, distributed systems have a major advantage: they can easily be made redundant, which is at the core of all important fault-tolerance techniques. Unfortunately, distribution also means that the imperfect and fault-prone physical world cannot be ignored, so that as much as they help in supporting fault-tolerance, distributed systems may also be the source of many failures. Usually the difficulties in dealing with dependability in such environments arise from the geographical distribution of resources, the high frequencies of user updates and data transfers, volatility of resources, conflicting application constraints, etc.

To discuss dependability one need to specify the types of failures that are considered. In our research we assume the following general model of a distributed system: the system consists of a set of processes which run different services, connected through communication links, which are used by the processes to exchange messages. We consider several types of faults: processing site failures, communication media failures and transmission delays.

With our approach, the process of dealing with faults is split into a series of distinct activities. In error detection phase we identify that the system is in an invalid state, meaning that some component in the system has failed. To this end, we rely on real-time monitoring of the distributed system, using custom built modules within the MonALISA framework [1]. The modules perform background audits that determine whether a service is functioning correctly or not. To ensure that the effects of the error are limited, it is necessary to isolate the failed component so that its effects are not propagated further. This is achieved by a Proxy service that can choose from several replicas of a specific service the best suited to answer a client call. In the error recovery phase, the error and, more importantly, its effects, are removed by restoring the system to a valid state. Hence, the replicas are maintained consistent when return from the faulty state. In a large scale distributed system, it is typical for a single fault to cause many cascading errors that are reported independently. Correlating and tracing through a potential multitude of such error reports often requires sophisticated reasoning.

The remainder of this paper is organized as follows. Section 2 surveys the existing work on dependability and argues the originality of our approach. In Section 3 we give a brief overview of the context of our research, presenting the architecture of the system incorporating our fault-tolerance modules. Section 4 details the mechanism for ensuring dependability based on a set replicated services running in a fault-tolerant container and a proxy service able to mask possible faults, completely transparent for the client. The Proxy's decisions are based on monitoring information provided by a custom monitoring system presented in Section 5. In Section 6 we illustrate the services designed to cope with security threats, extending the GSI middleware services. These services include the set of mechanisms necessary to guarantee the security needs, based on mechanisms for taking security related decisions, using monitored data gather using

MonALISA. Section 7 presents the performance evaluation of our solution while Section 8 concludes this paper.

II. Related Work

Most of the approaches for the dependability problem are mainly focused on fault detection and fault tolerance. [2] introduces 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. Another approach less related to large scale distributed systems is presented in [3], which addresses the tolerance to hardware faults through virtualization; the proposed solution is named Loosely Synchronized Redundant Virtual Machines (LSRVM).

Another aspect of dependability is security. In case of Grid systems, [5] identifies a set of base requirements (single authentication, credentials protection, interoperability with local security solutions, etc) and proposes both an architectural model for security and a public key criptography infrastructure based on a reference implementation (Grid Security Infrastructure - GSI). However, all existing solutions for security in large scale distributed systems have unresolved issues [6]. The majority of Grid systems were initially designed for scientific collaboration between participants that knew each others. In that case one can imply 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 such systems are to be used in an industrial environment, there apears the need for sharing resources to unknown groups, which could generate certain risks. A possible solution is to impose some mechanisms to preserve the identity, and the architectural model proposed in this project will integrate many of those, providing trust mechanisms for service and resource provisioning, and providing guarantees for the delegation and context services.

Unlike similar research projects, in this paper we propose a unified approach to the aspects concerning dependable systems, by analyzing and designing methods and techniques for improving the resiliance and security in case of large scale distributed systems. We are focusing on Grid systems, but the proposed solutions can be easily applied to other types of distributed systems. These services are extensions of existing research projects which only propose partial and separated solutions for these problems.

III. Architectural Overview

The proposed architecture is based on a minimal set of functionalities, absolutely necessary to ensure the reliability, availability, safety and security. Currently, the problem of developing highly dependable large scale distributed systems is technological prohibited by the various technical problems arising in distributed systems. An example of such a problem is the weak resilience of the operating systems, known for offering security vulnerabilities. The solution to this consists in the inclusion of a complete set of mechanisms necessary to guarantee the various security needs: the trust in service and resource provisioning, guarantees to the delegation and context mechanisms. The architectural model includes for these purpose two specialized sets of components.

The first architectural component set is responsible with ensuring dependability at the operating system level. At the heart of the operating system lies the kernel. A *Reference Monitor* component resides within the kernel space. It cannot be circumvented and/or modified and must be simple and compact enough to be readily understood. This component can validate all attempts to access the system resources based on the input provided by the second component, the *Security Policy*. This component provides complete mediation of verification of the validity of requests, based on sets of permissions. All possible problems are reported and stored for further inspection and even for automatically pattern recognition of errors in future cases.

A second set of components act on the middleware layer of the distributed system, ensuring dependability between different hosts composing the distributed 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 dependable execution environment for several other components. It also orchestrates the survivable and redundancy mechanisms described below. This architectural layer is built around a Proxy service which implements the mechanisms for ensuring dependability based on a set replicated services running in a fault-tolerant container. The Proxy service is able to mask possible faults completely transparent for the clients. The modularity of this component facilitates the adoption in real-world existing Grid systems. This architecture allows the generic integration with various technologies specific to large scale distributed systems, and the development and evaluation of various new models and technologies. The architecture's components can be interconnected with services available in such systems, and can cope with QoS specifications coming from higher-level applications. These middleware layer components include a set of mechanisms necessary to guarantee security needs in case of large scale distributed systems. They are capable of taking security decisions based on monitored data. To ensure that these decisions are enforced, the components use several other existing services and security infrastructures (such as GSI).

In this paper we focus on the second set of (middleware) components, as the kernel level components were previously discussed in [4].

IV. Proxy Service

One solution to incresing resiliance in case of large scale distributed systems consists on a set of replicated services running in a fault-tolerant container and a Proxy service able to mask possible faults, completely transparent for the client. In a distributed architecture this service can not only masks possible faults, but also optimizes the access to the distributed

services and their replicas using a load-balancing strategy, whilst ensuring a high degree of scalability.

The proposed solution masks possible failures and optimizes the access to the distributed services and to the replicas of these services. The messages sent by a client when invoking a service are intercepted by the Proxy service, implemented inside a container, which in return forwards them further to a particular replica (chosen based on an optimality criterion) of the invoked service and also re-forwards the request to another replica when the first one fails. The approach is presented in Figure 1.

The approach has the advantage that is does not require the alteration of the invoked service or the client. In most cases the service provider does not have access to the source code of the service in order to add its own fault tolerance mechanisms. The existence of a mechanism that can separate the service implementation from the fault tolerance policy represents an improved approach in the SOA context.

The proxy service also considers balancing the load between service replicas. The implementation of a load balancing policy ensures a more efficient use of resources and leads to a smaller response time. The algorithm used to choose a replica to receive a particular request is based on the number of errors previously generated by a replica. In this way we added a mechanism for fault avoidance that considers the history of failures.

The architecture of the system implementing the Proxy pattern is represented in Figure 2. In this architecture a proxy is situated between the client and the real service. This has the advantage that it can control the access to the real service and inside can implement various actions that are triggered each time a service is being invoked.

The service container also implements the proxy functionality for the replicated services. The messages sent to a real service are intercepted by the container (with the help of a SOAP listener) and are redirected to a replica of the real service (with the help of a Proxy Service). All actions executed inside this container are transparent to both the client and the real service. The client considers, throughout the entire communication, that it is connected with the real service and not with the proxy. Also all replicas respond to all requests as if they are coming from real clients and not from another service.

The monitoring and load balancing service centralizes all information regarding the service replicas and allow choosing the optimum replica to be invoked by the Proxy service. The service was designed as a separated component so that the functionalities of the application are distributed in small units, in accordance with the SOA specifications. It can be invoked by other Proxy services or even by replicas of the Proxy service, being implemented in more containers as a replicated service. Also, because it runs as a completely separated component, it can easily be replaced by another component providing the same interface, so it also allows for better extensibility.

In the context of the Axis architecture, the SOAP listener component was implemented in the form of a filter registered on the general request chain of Axis. This component receives all incoming messages and redirects them to the real service or to the Proxy service. The filter can be configured to the deployment of the entire container or to just the deployment of certain services.

The proxy service was implemented as a *request* Web service, implemented inside an Axis container. This type of service assumes the instantiation of an object for each incoming request. The approach allows the parallel processing of incoming requests without the need to synchronize the access to the internal information of the service. The advantage of this approach consists of lower response times and the increased capability of processing requests. The Proxy is further replicated at its turn, in order to avoid a single point of failure.

V. Monitoring Service

The Proxy service takes decisions based on real-time information received from a monitoring system. The role of the monitoring service is crucial both in terms of information provided and high availability requirements. In this section we motivate the decisions taken regarding the monitoring approach and detail its architecture and implementation.

In our approach towards a dependable distributed system we use service replication and we guarantee that all processing replicas achieve state consistency, both in the absence of failures and after failure recovery. We achieved consistency in the former case by implementing a module that ensures that the order of monitoring tuples is the same at all the replicas. To achieve consistency after failure recovery, we rely on checkpointing techniques. he optimization problem is addressed of the replication architecture by dynamically monitoring and estimating inter-replica link throughputs and real-time replica status.

Replication is a widely used technique to guarantee the availability and dependability of large scale distributed systems in the presence of faults is replication and implies the use of more services or components performing the same function. Whenever a replicated entity encounters a failure another replica is switched on and takes its place. Data and service replication is a reliability improvement technique used in many types of distributed systems. By replicating the data and services over multiple nodes, the system can support the failure of some of these nodes, without losing its ability to function correctly. Moreover, data and service replication are employed for load balancing reasons. In a typical distributed environment that collects or monitors data, useful data and services may be spread across multiple distributed nodes, but users or applications may wish to access that data through a central location (a proxy service). A common way to ensure centralized access to distributed data is by means of maintaining replicas of data objects of interest at a central location. However, when data collections are large or volatile, keeping replicas consistent with remote master copies poses a significant challenge due to the large communication

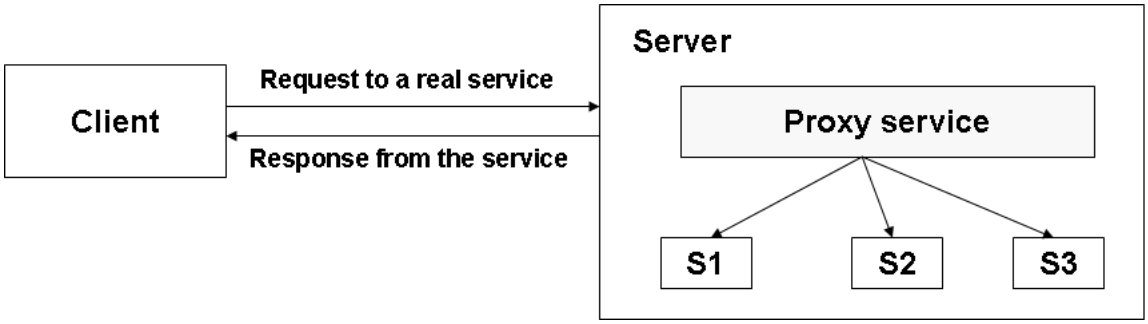

Fig. 1. Communication between the client and a replica of the invoked service using the Proxy service as intermediary. S1, S2, S3 are replicas of the same invoked service.

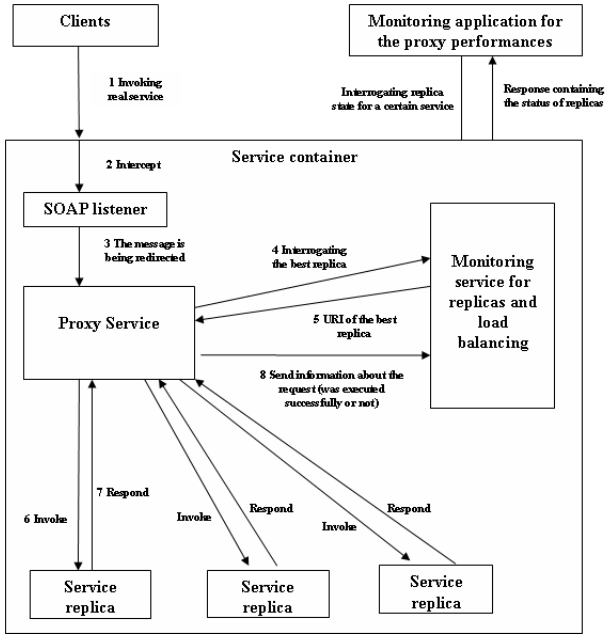

Fig. 2. The architecture of the system for fault tolerant access

cost incurred. In this paper we address these challenges by proposing a service and data replication architecture that relies on the MonALISA distributed monitoring framework.

A. The MonALISA Framework

MonALISA is a distributed monitoring system, relying on JINI and WebServices technologies, able to provide complete monitoring, control and global optimization services for complex systems. An agent-based architecture provides the ability to invest the system with increasing degrees of intelligence, to reduce complexity and make global systems manageable in real time. The scalability of the system derives from the use of a multi threaded engine to host a variety of loosely coupled self-describing dynamic services, the ability of each service to register itself and then to be discovered and used by any other services, or clients that require such information. The system monitors and tracks site computing farms and network links, routers and switches using SNMP, and it dynamically loads modules that make it capable of interfacing existing monitoring applications and tools (e.g. Ganglia, MRTG, LSF, PBS, Hawkeye). Within this framework we developed a repository system, able to collect data from replicated services and present global views from the dynamic set of services running in the distributed environment to higher level services, namely the proxy service. The system subscribes to a set of parameters or filter agents to receive selected information from the farms, and stores this monitoring information locally, using space and time optimizations. Data is collected in order to enable real-time and retrospective analysis, thus dealing with large amounts of information and introducing reduction and aggregation mechanisms. A servlet engine is used to dynamically present customized views of the jobs, resource usage, system and network key metrics in a flexible way. It provides general visualizations at various aggregation levels of the actual set of resources available, it permits evaluations of how the global service offered by resource providers fulfills the service level agreement demanded, it allows to analyze data retrospectively to understand how to improve the effectiveness of applications running and relying on the replicated services,

also helping to detect fault situations and possibly preventing them by taking automated actions.

B. The Monitoring Architecture

The monitoring architecture relies on the MonALISA system for the underlying monitoring of replicated services. We augmented the existing system with a layer able to deal with replicated services in terms of concurrently collecting data from them, maintaining consistency between replicas when failures occur and evaluating the best replica to serve a client's request via the proxy service. The monitoring architecture consists of 3 modules: the *Replication Manager* - a module that manages the incoming queries from the proxy service, *Replicas* - entities that monitor locally the replicated services and communicate with the Replication Manager in order to answer to the proxy's requests and perform monitoring data updates, and the *Monitoring Module* - a module that helps the replicas to trigger decisions when a file transfer between them should occur.

A. The Replication Manager

The main module of the monitoring architecture implements a *Replication Manager* (RM). Its role is to intercept all queries coming from the monitored farms and clients and passes them to the replicas whenever it is queried. The RM saves those queries that modify data (update, insert or delete SQL commands) until all the replicas receive them. The queries are tagged with sequence numbers, which are used by replicas when they want to update their databases or in the answers given by the RM to the queries coming from the replicas. The Replication Manager consists of 3 sub-modules: the *file remover* module, the *replica listener* module and the *init* module.

The File Remover module removes files with queries from its local storage. It receives confirmations from replicas that certain intervals of queries have been executed, inserts those intervals in an interval tree and it queries this interval tree to see if there is any interval that have been completely received by all replicas. If there is any, the RM deletes the queries with sequence numbers in that interval, removing also those intervals from the interval tree. The Replica Listener module asynchronously listens for connections from replicas and solves the requests. Taking into account the fact that the RM holds the queries until all the replicas receive them, every request to the RM should be successful. The Init Module receives the queries and based on the type of the query (a select or a non-select query) it looks through the online backends and selects the most updated replica or it logs the query, respectively. The most updated replica is the one that has executed the most queries.

B. Replica

This module receives queries from the RM and saves them into files. At predefined interval of times, it reads the queries from these files and executes them. It also sends to the RM some messages in which it requires the sequence number of the last received query. If this differs from the last sequence number received locally, then the replica tries to get the lost queries from the other replicas or, if the attempts to take files from the other peers fail, it gets them from the RM. A Replica consists of 3 modules: the *File Remover* module, the *Query Demander* nodule and the *Query Sender* module.

The *File Remover* module periodically deletes files from which queries were already read and executed. The period of time at which the rescheduling does occur influences the overall performance of the architecture: thus, in an unstable architecture, if the period of time is big enough, it increases the chances that a replica finds some files with queries at a peer, thus alleviating the RM which is the core of our architecture. The *Query Demander* module connects periodically to the RM and requires the sequence number of the last received query. If it is greater than the last query's sequence number received locally, then it tries to get the lost queries. It will try to retrieve the queries from the RM, but also from the other replicas. The *Query Sender* module accepts connections from the other peers and serves the requests with files with queries. Thus it receives messages in which it is asked to send some files. It sends the requested files if these are found in their local storage or send null instead.

C. The Monitoring Module

The third module monitors the link states between the replicas and between the replication manager and replicas. Whenever a replica wants to update its database, it first queries the monitor module to see from which replicas it should transfer the missing files. The monitor module finds the target replicas doing some heuristics on some parameters like the load on all the replicas, the CPU usage, the physical memory usage, the availability and the available bandwidth between the demander replica and the other replicas. After computing a score for each of the replicas, this module returns to the demander replica a list with the most unloaded and available replicas from which it should begin copying the files. The monitoring module periodically receives some parameters from the replicas. These parameters are collected by means of ApMon, which is a set of flexible APIs that can be used by any application to send monitoring information to MonALISA services. Furthermore, it is very light-weight and non-intrusive and it has the advantages of flexibility, dynamic configuration and high communication performance. Based on these parameters, it assigns a score for every replica (doing a weighted mean) and, whenever it is queried, it sends the most suitable replicas from whom the demander replica can copy its missing files and to preserve consistency. The system administrator can control the replication procedure by means of the replication manager. Thus he can send some messages to the RM telling it to draw a replica from the replication architecture or to add a new one. The only details that must be sent are the Internet Address and the listening port of the replica.

The scores computed by the Monitoring Module are sent to the Proxy Service, which selects the best replica service to serve a client's request based on the calculated hierarchy. Heuristics that make up the scores can be changed in order to meet a targeted application's specific requirements.

VI. Enhancing Security

Using replication to achieve a fault tolerant environment leads us to major security concerns. We must ensure the integrity of the data which is replicated, the confidentiality of the service users and also the resistance in the case of external attacks. Obtaining an increased resistance to attacks is an important goal for us since we are focusing on resource availability, as well as fault tolerance. This means that we must detect any threats as soon as possible and not allow any of them to diminish the functionality of the entire distributed system.

To reach a necessary level of security for such an architecture, when designing dependable distributed systems, we used the basic security mechanisms present in large scale distributed systems. The most useful for our architecture are the ones provided by the Globus Security Infrastructure, used in Grids. This infrastructure meets very important security requirements like single sign-on, which enables the user to authenticate only once, the protection of credentials and interoperability with local security solutions. It uses X.509 certificates or username/password for authentication and X.509 proxy certificates for delegation.

The proxy certificates are valid for a short period of time and they are generated from the user's certificate. Transport Level Security (TLS) can be applied to encrypt all communication including SOAP messages. Once mutual authentication is performed, the communication between different entities communication can occur without the overhead of constant encryption and decryption. A related security feature is communication integrity. By default, the integrity of the messages is also ensured. Communication integrity introduces some overhead in communication, but not as large as the encryption process.

In our architecture the same low-level security mechanisms are present as in a general Security Oriented Architecture(SOA) [7]. For example, WS-Security provides low-level security mechanisms like message integrity and confidentiality; WS-Trust provides additional primitives for delegation, along with X.509 certificates. The basic security mechanisms are also provided by GSI (Grid Security Infrastructure) which provides low-level mechanisms [8]. We extedned these mechanisms with more complex ones.

A major security problem in dependable distributed systems is the authorization process, which poses many challenges in large scale distributed systems such as Grids. This deals with the correct identification of the active users. However, because of their size and complexity, in large scale distributed systems there may be a large number of security domains. A security solution must use the actual attributes of each entity in its valid security context. The security service also needs to manage a set of policies for the attributes of each entity. The authorization step must be done at the service provider level so that only valid users may have access to the rest of the services of the distributed system. For that a complex authorization scheme is used, based on other factors besides the simple identity of the user, such as its role. In RBAC (Role Based Access Control) each user has a set of roles which determine his attributes. Another acces control method which is used is based on the context of the request (CBAC). This is useful for example if we wish to allow access only in certain time intervals.

The solution that the security service proposes for the authorization of the involved entities uses these described mechanisms. This restricts the access to the shared resources using not only the identity of the user, but also its attributes and other available information. These attributes are defined by a trusted entity and are published in a secure way. This model also supports delegation and the use of proxies which is essential for large scale distributed systems as the identity of a user may not be known in every local environment.

Another considered security risk is the existence of outside attackers. The only efficient way to protect against a DDoS or other complex attacks is to use high level intrusion detection mechanisms to identify and block the potential attacker. Due to the number of available resources in such a large scale distributed system the intrusion detection system which operates on the entire service level must rely on lower-level network intrusion detection systems(NIDS) each detecting possible attacks on a local domain. Integration with the local NIDSs is done using the IDXP protocol. This approach is compatible with many existing distributed systems since the IDXP protocol is supported by the existing local intrusion detection systems, such as Snort.

The detection of attacks is performed using simple XML patterns which add to the modularity of the system. The patterns can be easily modified to describe new types of attacks and are also easy to interpret and understand by the system administrator. There are two main types of attacks that a high-level intrusion detection system can identify:

- *Resource based attacks*. The attacker tries to compromise a certain set of resources from the same administrative domain.
- *Application based attacks*. The Intrusion Detection System receives alerts from various resources which are running parts of the same application which the attacker may be trying to target.

After identifying the type of attack we can block the user if the source of the attacks is known and also we can determine an attack trail by agging the resources which have a high risk of being attacked.

We also need to consider the fact that a user may become compromised in time and start sending damaging requests. To prevent this we need to consider the previous actions of each user to determine a trust level which may then be used to take certain decisions regarding the security of the entire data sharing system. This method takes into account the actions that were made by authorized users but may or may not benefit the entire system. When a user begins its activity and there is no known activity log for him, he is attributed a neutral trust level which is continually modified according to the actions he takes. As a result of these checks to determine the trust level we can detect if a user is spamming, if he is repeating certain

actions, or if he is a security threat and should be blocked. For example, previous actions can be used to increase the penalty for bad queries if those queries seem to be part of an attack, until the user's permissions are completely revoked if it is considered that he is a security risk.

VII. Performance Evaluation

We present a series of results demonstrating the capabilities and performances of the proposed solutions. We conducted our tests on the NCIT testbed [10], 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 used 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 our solution. Nodes were used to deploy the Proxy, the storage and security services and the MonALISA monitoring trackers.

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 emulates the functionality of serving a particular request in a certain amount of time, but also includes the possibility to be controlled if and when it triggers exceptions. We also developed a client simulator which generates requests to this service and runs inside the same container with the Proxy service and with the Monitoring and Load-balancing service. The requests are intercepted by the Proxy service and redirected to a replica service. The used topology is presented in Figure 3.

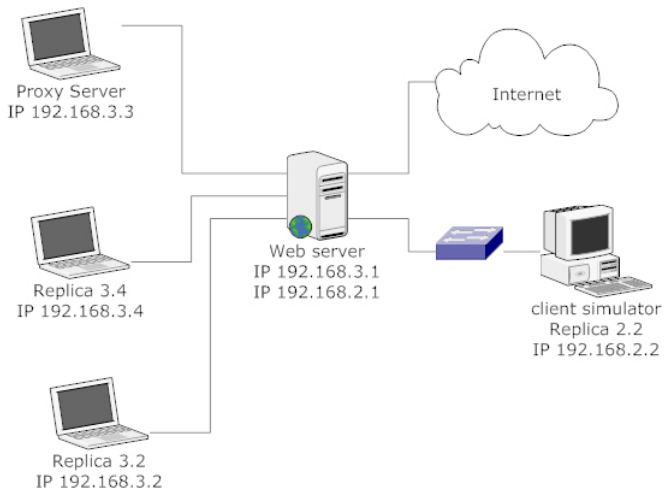

Fig. 3. The topology used for the evaluation experiments.

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 then in the case when not using the proxy service at all.

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 4 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.

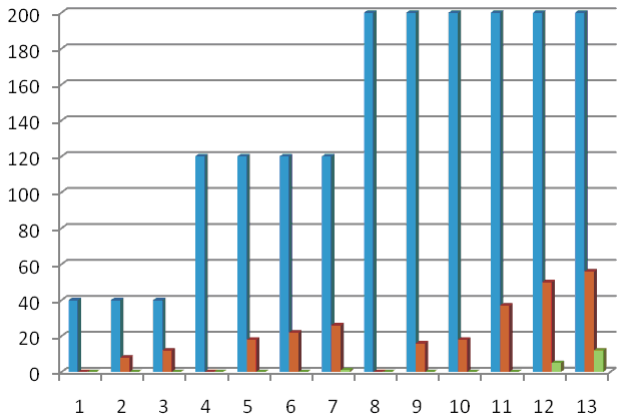

Fig. 4. The masking of errors.

In the figure with blue is presented the number of requests per seconds, with red the number of errors generated by replicas and with grren the total number of errors as seen by the client. As seen, an important factor for masking errors is represented by the number of requests received by the proxy system per second, as well as the unumber 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 second series of experiments assumed the use of LISA [9], a lightweight dynamic service that provides complete system and application monitoring. In this case the objective was to evaluate the network traffic and load of the system on which the services are running in order to demonstrate the non-intrusive characteristic of the proxy-replication system. The testbed involved three stations, two located in Switzerland and one in Romania. In the beginning we 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, an agent was broken to see how the system would react to its failure.

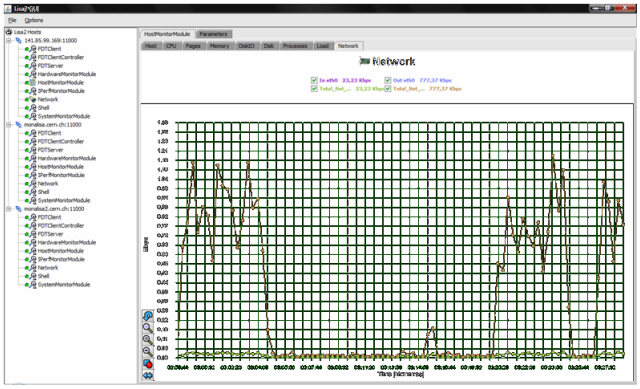

Fig. 5. The results for the evolution of the network traffic.

During the tests that were performed we measured a sustained additional traffic varying between 5 and 20 kbps while the machines load didn't change significantly. The measured values demonstrate that the overhead caused by running the agents is very low, both in terms of network traffic and CPU usage or system load.

VIII. Conclusion

In this paper we presented an architectural approach for satisfying dependability requirements 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.

The original contribution addresses the architectural model and the combination of existing monitoring, data management, security and fault tolerance solutions. The system is under development and will be used in Romania EGEE clusters. The future work considers the development of mechanism for fault prediction and avoidance.

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.

References

[1] I.C. Legrand, H.B. Newman, R. Voicu, C. Cirstoiu, , MonALISA: An agent based, dynamic service system to monitor, control and optimize grid based applications, *CHEP 2004*, Interlaken, Switzerland, September 2004

[2] H. Jin, X. Shi, W. Qinag, D. Zou, DRIC: Dependable Grid Computing Framework, *IEICE - Transactions on Information and Systems*, E89-D(2), 2006.

[3] A. Cox, K. Mohanram, S. Rixner, Dependable ??? Unaffordable, *1st workshop on Architectural and system support for improving software dependability*, San Jose, California, 2006.

[4] V. Cristea, C. Dobre, F. Pop, C. Stratan, A. Costan, C. Leordeanu, E. Tirsa, Models and Techniques for Ensuring Reliability, Safety, Availability and Security of Large Scale Distributed Systems, *In Proc. of the the 17th International Conference on Control Systems and Computer Science, HiperGrid09*, Bucharest, Romania, 2009

[5] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A Security Architecture for Computational Grids, *Proc. Fifth Conf. Computer and Communications Security*, ACM, 1998.

[6] A. Arenas, State of the art survey on trust and security in Grid computing systems, *Technical Report*, Council for the Central Laboratory of the Research Councils, UK, 2006.

[7] C. Geuer-Pollmann, J. Claessens, Web services and web service security standards, *Information Security Technical Report*, 10(1):15 24, 2005.

[8] The Globus Security Team, *Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective*, Retrieved May 14, 2010, from http://www.globus.org/toolkit/docs/4.0/security/key-index.html.

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

[10] The NCIT Cluster *http://cluster.grid.pub.ro/*, Retrieved May 14, 2010