Fault Tolerance using a Front-End Service for Large Scale Distributed Systems

Marieta Nastase, Ciprian Dobre, Florin Pop, Valentin Cristea

Faculty of Automatics and Computer Science, University “*Politehnica*” of Bucharest, Romania
Emails: marietanastase@yahoo.com, ciprian.dobre@cs.pub.ro, florin.pop@cs.pub.ro, valentin.cristea@cs.pub.ro

Abstract—**In this paper we present a solution to ensuring dependability in service-based large scale distributed systems. The proposed solution is based on a set of replicated services running in a fault-tolerant container and a proxy service able to mask possible faults, completely transparent for a client. We demonstrate an architecture which 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 advantages of the proposed architecture were evaluated using a pilot implementation. 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.**

Keywords

Fault Tolerance, Dependability, Proxy Service, Replication, Load-Balancing

I. Introduction

The domains of large scale distributed systems have been extending during the past years from scientific to commercial applications. Both in the academic and industrial environments there is an increased interest in large scale distributed systems, which currently represent the preferred instruments for developing a wide range of new applications. The Grid computing domain has especially progressed during the last years due to its offered technological opportunities.

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 the commercial ones, have some complementary necessities that the ”traditional” distributed systems do not satisfy. Among the requirements that have emerged for large scale distributed systems dependability is needed by more and more modern distributed applications, not only by the critical ones.

The service oriented architecture and the use of Web services in large scale distributed systems led to the ease of developing larger applications because of several advantages [**?**]: the standardizations of the messages exchanged between clients and services, as well as the way services are described and discovered. But with the increasingly interest in large scale distributed systems, clients pose new requirements: they are interested in invoking services that can deliver correct responses in a limited amount of time. The availability time is becoming more important: the longer time a service is available the more requests it can serve. Services that consider such requirements are called dependable [**?**].

There are several methods to improve the quality of the services [**?**]: fixing bugs before deployment, the adoption of various fault tolerance mechanisms and solutions to recover from errors, etc. All of them can be generally implemented inside a service by the developer, leading to an increase in the development time and to higher costs. Most service developers choose the less costly and the fastest solution, not necessary the one providing the highest quality. Unlike them, the service providers are committed to offering dependable services to their clients, which can cope with various negotiated Quality of Services.

Although the importance of dependable services is widely recognized and many research projects are currently well underway, no general solution exists that could be easily adopted by service providers and which could lead to an increase in the dependability of the provided services, without the support coming from service developers. In this paper we present an efficient solution to satisfying the dependability requirements of services in large scale distributed systems. The solution masks the faults occurring in a set of replicated services (at least functional similar set of services) by providing a container layer between them and various clients. The architecture also provides optimal access to the distributed services, without influencing the scalability characteristic of the applications running in such distributed environments. The solution is specifically adequate to provide support to service providers, and do not rely on the inclusion in the service development stage or any auxiliary functionalities or costs.

The rest of this paper is structured as follows. Section 2 presents related work to the problem of ensuring fault tolerance in large scale distributed systems. Section 3 introduces the proposed service-based architecture. In the next section we present the implementation details of the pilot application, together with the considered balancing polity. In Section 5 we present several results obtained in the evaluation of the solutions. Finally, in Section 6 we present some conclusions and future work.

II. Related Work

Although the importance of dependable services and systems is today widely recognized and many research projects have been initiated recently 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. Most researches in this area are based on the idea of modifying the service implementation such that to ensure fault tolerance.

An example of such an approach is described in [?]. The authors propose the use of a primary-backup mechanism, maintaining several synchronized replicas of a service, in order to ensure fault tolerance. When the main service fails one of the secondary replicas becomes principal. When the client receives an error from the service it must resubmit the request, which is then processed by the new primary replica. The approach does not mask the errors and assumed the modification of the stub used by the client.

Other approaches consider the masking of faults in the communication level [?]. Such approaches represent the starting point in obtaining a fault tolerance system at application level. The difference here consists in the fact that, in the context of services, at application level there is also a context for the received message, containing meta-information related to the request, information that can be useful for the fault tolerance mechanism.

In [?] the authors present a system to ensure dependability of services based on making a group of nodes appear to clients as one single node. Once a request is received from a client the system chooses the ”‘closest”' node for treating the request. The choosing of the node is realized based on a modified approach of the any-cast routing scheme, using the properties offered by the Mobile IPv6 protocol [?]. Although this represents a solution that targets many of the problems of dependability, if works only on nodes running the Xtreem OS. From the client point of view the operating system must also support the Mobile Ipv6 protocol.

The solution proposed in [?] considers the development of a system composed of alternative, differently implemented, set of services. Each available service is designed such that to consider another set of errors. The solution is based on DESL (Dependability Exchange and Specification Language), a language for describing specifications for each service. But the DESL language itself is not yet fully mature.

In [?] the authors showed, similarly to our own work, how distributed computations that operate with an essentially functional parallel programming model can tolerate faults with relatively simple mechanisms, i.e. without global snapshots as required in message passing programs. The authors also implemented a system to demonstrate the validity of the idea, based on the work presented in [?].

DIGS (Dependability Infrastructure for Grid Services) [?], [?] is a project having as objective the design and implementation of a framework at application level for developing dependable Grid services. The project describes the structure of a ”proxy” that combines multiple identical services in a ”better” (having a higher dependability level) unitary service. The starting point in reaching the desired objectives in this case consisted in the design of a mechanism that can transparently intercept the SOAP messages exchanged between client and service. A client sends a message to the proxy as this would be the service providing the desired functionality. The proxy intercepts the message and process it based on a fault tolerance model, prior to forwarding it to the real service behind. The messages coming from the service to the client are also forwarded by the proxy.

In order to implement this system the authors used the idea of a container of components: the development of a service container that incorporates the mechanisms for fault tolerance. Unlike our solution, this one considers the use of a proxy for each service, and the client accesses the real service by sending requests to the proxy. The real replicated services invoked by the proxy are not necessary deployed in the same container as the proxy service. Also, unlike our solution, for accessing the service container the clients must be aware of the URI address of the proxy service.

III. Service-based Architecture

The starting point in developing the solution for achieving fault tolerance using replicated services was represented by the analysis of the SOA and the currently used protocols. A typical service is represented by an application identified by URI and whose interfaces can be identified, described and discovered using XML-based protocols. This scheme supports the direct interaction with other applications using XML messages over Internet protocols [10]. Typically an engine, such as Axis, is furthermore responsible with processing the messages coming from a client and invoking the services (Web or Grid service). A typical service invocation is presented in Figure **??**.

In order to increase the dependability of its services a service provider can use several techniques. For example, when a replica fails another one takes over and responds to all future requests instead. In this way the dependability of services increases, but the requests already sent to the failed replica will not be recovered/answered anymore, and clients will not correctly receive the answers to these requests.

In this we propose a solution that 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 a 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 proposed approach is presented in Figure **??**.

The approach has the advantage that is does not require the alteration of the service or 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 being presented in this paper also considers the balancing of 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.

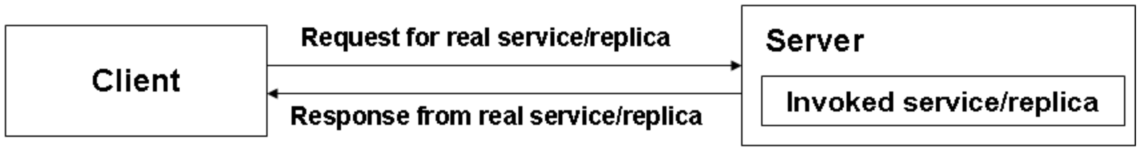
Client
Request for real service/replica
Response from real service/replica
Server
Invoked service/replica

Fig. 1. Communication between client and the invoked service

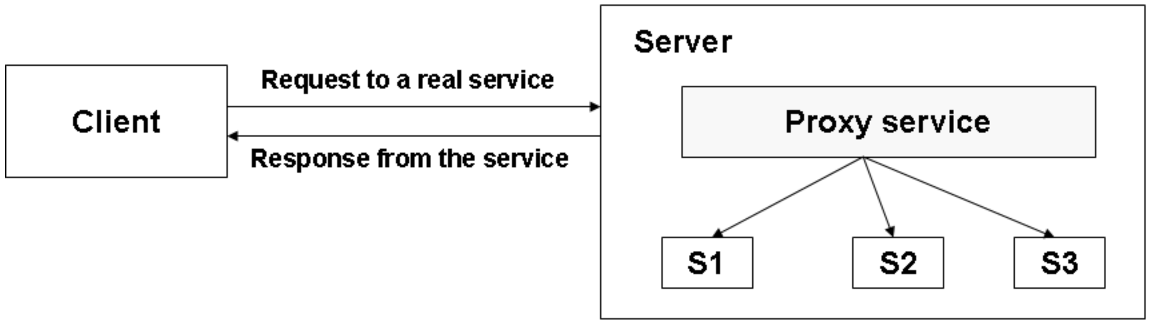

Fig. 2. Communication between the client and a replica of the invoked service using the Proxy service as intermediary. S1, S2, S3 are replicas of the same invoked service.

The algorithm used to choose a replica to receive a particular request is also based on the number of errors previously generated by a replica. In this way we also add a mechanism for fault avoidance that considers the history of failures.

The proposed architecture is represented in Figure **??**. The architecture implements the proxyPattern design pattern. The objective of this pattern consists in the definition of a proxy object, situated between the client and the real service. This object 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 SOAP listener is the components responsible with the monitoring of the SOAP message received by the service container. This component receives the SOAP messages and verifies if these messages should be redirected to the proxy service or to the real invoked service. There are several situations when the message should be directly passed on to the real service and should not be processed first inside the proxy service. For example, when the messages being redirected by the proxy to a replica of an invoked service are captured by the SOAP listener they should be directly forwarded. Or in the case of special services: the AdminService service is used by the service providers to deploy services and the corresponding messages should not be handled by the proxy.

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. This 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. This service can also 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.

When deployed this service first reads the list of replicated services and the information regarding each replica from a configuration file having the following structure:

```
<services>
  <service name="TestService">
    <replica url="http://192.168.2.2:8080/
        axis/services/TestService"
        MaxConnections="20" />
    <replica url="http://192.168.3.2:8080/
        axis/services/TestService"
        maxConnections="30" />
    <replica url="http://192.168.3.4:8080/
        axis/services/TestService"
        maxConnections="50" />
  </service>
</services>
```

Each replicated service is represented by a ”service” element, where the name of the service is specified by the ”name” attribute. The replicas are specified as elements of type ”replica”. For each replica one can specify the URI address and the maximum number of allowed connections. For each replica the service initializes the total number of processed

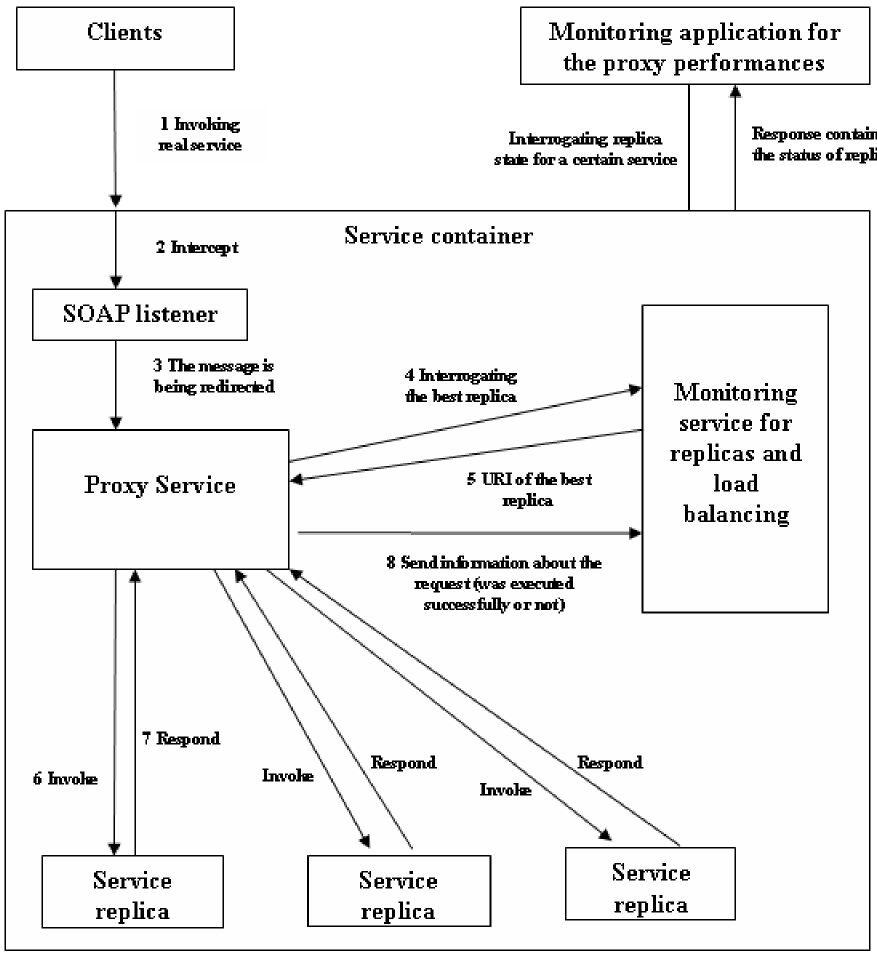
Clients
Monitoring application for the proxy performances
1 Invoking real service
Interrogating replica state for a certain service
Response containing the status of replicas
Service container
2 Intercept
SOAP listener
3 The message is being redirected
4 Interrogating the best replica
Monitoring service for replicas and load balancing
Proxy Service
5 URI of the best replica
8 Send information about the request (was executed successfully or not)
6 Invoke
7 Respond
Invoke
Respond
Invoke
Respond
Service replica
Service replica
Service replica

Fig. 3. The architecture of the system for fault tolerant access

requests, the number of active requests as well as the total number of exception being thrown by the replica service. The statistical information is used by the algorithm in order to find the optimum replica to serve a certain request.

When choosing the optimum replica to serve a certain request the service also modifies the number of processed requests and the number of active requests for that particular replica. If for a replica the number of active requests is equaled with the maximum number of requests it can process then the replica is considered as functioning at full capacity and can not be considered as adequate to process any other incoming requests. When this situation happens another replica is chosen instead. If at some time no replica is available a corresponding exception is returned to the client.

The Proxy service acts as intermediary between the SOAP messages received from the Listener, and the invocation of the adequate replica responsible for processing the request, is responsible with sending back any returned answer and masking of any occurred error. When it receives a SOAP message from the Listener, the Proxy service makes a request to the Monitoring and Load-balancing Service to find out the optimum replica that can serve the requests. The replica is then used as the destination to which the Proxy forwards the SOAP message. When the replica generates an exception the monitoring service is again interrogated, and a new replica is selected and the SOAP message is redirected to the new replica.

IV. Pilot Application

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 filter itself is an object of type ProxyHandler, a class that extends the org.apache.axis.handlers.BasicHandler class and which overrides the invoke method. The entire processing chain of the message from the client to the proxy service is represented in Figure **??**.

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

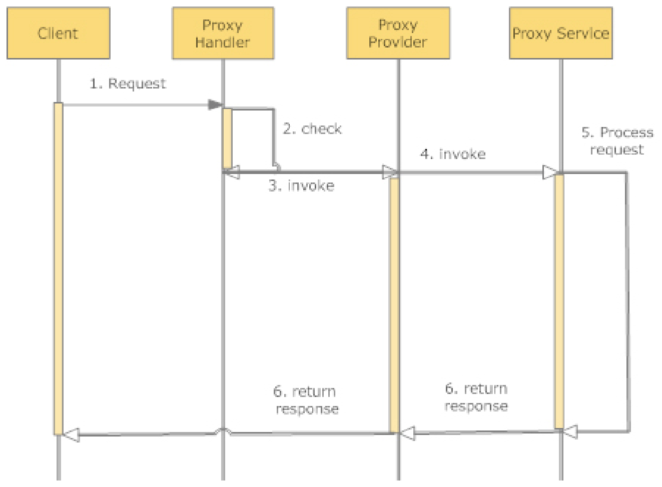

Fig. 4. From Client to the SOAP message to the Proxy service.

of this approach consists of lower response times and the increased capability of processing requests.

The service exports the function processMessage. Each time this function is called a ProxyMessage object is created on which we call the method processMessage. ProxyMessage is a class having as members the context of the request, a list with all the invoke replicas for that requests, the received response from the last invoked replica and the exception received from it. This information is then used by the proxy to choose a replica.

Figure **??** presents the chain of actions being executed by the Proxy service inside the processMessage function, when a SOAP message is being received from Listener.

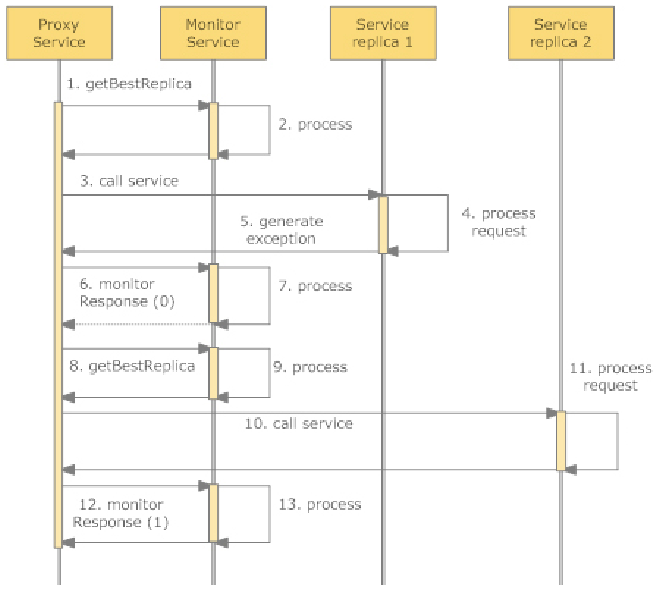

Fig. 5. The chain of actions being executed by the Proxy service.

When a replica is invoked the Proxy service modifies the SOAP message, adding a supplementary header containing meta-information. Inside the header it adds a field specifying if the message should be forwarded to the real service and not sent over again through the Proxy.

The Monitoring and Load-balancing Service is developed as a Web service (called MonitorService) of type application inside the Axis container. For this type of service Axis instantiates a single service object. Consequently, all accesses to the methods of this object are synchronized. The advantage of this approach is that we have a centralized control of monitoring information.

The replicated services are stored in a hash of ServiceInstance objects. A ServiceInstance object maintains information regarding the service, as well as a list of Replica objects. A Replica object maintains information about the replica of a particular service: the total number of requests that can be processes by the replica, its accessing URI. Based on this information for each replica internally we further compute a value representing the level of trust or confidence we have in that particular replica.

In order to offer high availability, the monitoring service also runs a load-balancing algorithm. This is used when choosing the replica that should serve a request. The load-balancing functionality of replicas has the role of optimizing the usage of resource and to decrease the response time needed to serve a request.

The level of trust associated with each replica is computed based on the number of active requests at a certain moment of time, the number of requests that resulted in a valid response, as well as the number of requests that resulted in exception being thrown. Figure **??** presents the parameters that define the level of trust for a particular replica.

Formula used to compute the level of trust for each replica is as follows:

$$Level = \frac{noOk + 1}{noErrors * 2 + noActive + 1}$$

when $noActive < noMaxim$, or 0 otherwise. In this formula Level represents the level of trust, $noOk$ is the total number of requests correctly processed, noErrors is the total number of thrown errors, noActive is the total number of active requests (the difference between the number of received requests and the number of received responses, valid or errors), and noMaxim represents the maximum number of requests that can be concurrently processed by the replica.

V. Tests Scenarios and Experimental Results

For testing the system we 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. In order to evaluate the performance of the proxy system we also developed a client simulator which generates requests to the TestService service running 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 the TestService replica

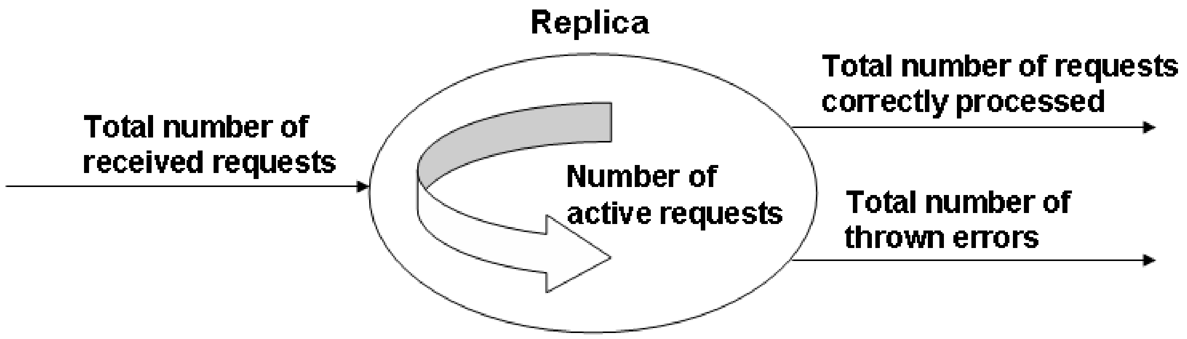
Replica
Total number of received requests
Number of active requests
Total number of requests correctly processed
Total number of thrown errors

Fig. 6. The information associated with each replica.

TABLE I
THE RESULTS OBTAINED IN THE EXPERIMENTS

Total number of requests sent by the client	The total number of requests processed by replicas	Number of errors generated by replicas	Number of errors as seen by the client
9	10	1	0
111	118	7	0
200	236	47	10
266	266	0	0
147	129	0	18

having the higher level of trust. The used testing topology is presented in Figure **??**.

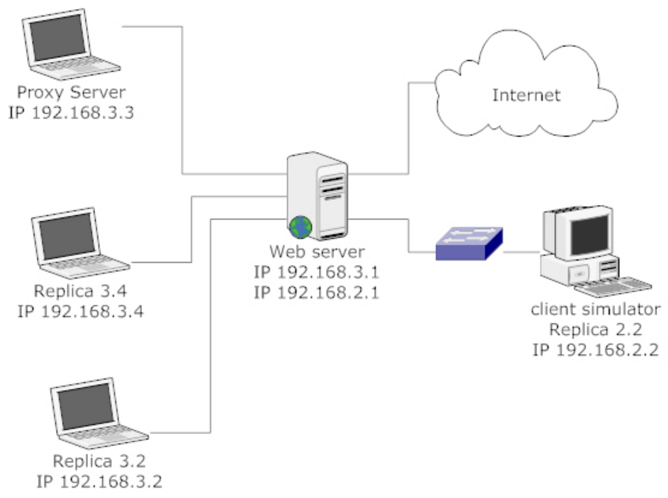

Fig. 7. The topology used for the evaluation experiments

Using this scenario we conducted a number of experiments. First we evaluated the capability of the system to mask possible errors. In the first steps of the experiment the load balancing algorithm behaved in a Round-robin manner because no replicas encountered errors, the replicas being circularly and repeatedly selected to process each new incoming request. In this situation the level of trust associated with each replica is 0.33. At a certain moment the client set one replica to generate errors and then sent several more requests to be processed. The first requests were processed by replicas without any errors. Then the next request resulted in an error being generated. When the Proxy service received the error retransmitted the request to another replica.

In order to demonstrate the capability to mask errors the next experiments considered the sending of 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.

The client simulator sent only three requests per second in this experiment, so in 10 seconds 30 requests are send, while the maximum limit for concurrent processing of replicas is set to 50. Because the number of requests received per second never exceeded the processing capacity of the other two replicas the client never received back any errors.

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.

The results obtained running all these evaluation experiments are presented in Table **??**.

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 **??** 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.

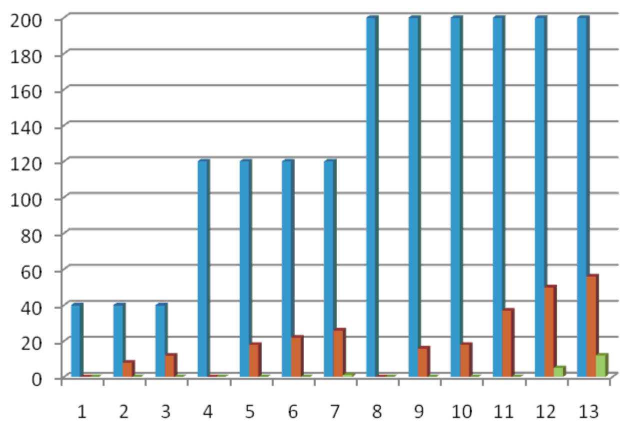

Fig. 8. The masking of errors.

In this figure with blue we represented 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 in this diagram 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.

VI. Conclusion

In this paper we presented a solution to ensuring fault tolerance in large scale distributed systems. The presented solution is based on the idea of encapsulating replicated distributed services inside a container with the purpose of masking possible errors. In this case the client with a high degree of confidence do not sees the error and the proxy service tries various automatic solutions to recover from occurred errors, such as forwarding the original requests to another non-faulty replica. The presented solution also uses a load-balancing system to ensure an efficient use of resource and the decrease of the response time.

We demonstrated that the solution is viable and do not necessitate the addition of any supplementary mechanisms in the service implementation or the stub known by the client. We used existing technologies that are already used by service providers. This can easily be deployed in the context of both Web and Grid service, as the architecture follows completely the SOA architecture.

The use of the proxy system by the clients assumes only the knowing of the address of the service implemented inside the same container as the Proxy service. This address remains unaltered even if other replicas of the same service are dynamically added or removed from the system. The transparent use of replication ensures a high degree of scalability.

The system is implemented in accordance with the SOA specifications and each component can be used by other systems. For example the monitoring service can be used by other services outside the Proxy service, and the Proxy service can be used independently in the context of any other monitoring service.

In the future we aim to extend the presented system by implementing a replication mechanism for the Proxy service itself, currently this being the single point of failure for the system.