Design and Implementation of Fault Tolerance Techniques to improve QoS in SOA

Edvard Martins Oliveira*, Júlio Cézar Estrella*, Bruno Tardiole Kuehne *,
Dionísio Machado Leite Filho *, Lucas Junqueira Adami*, Luiz Henrique Nunes *,
Luis Hideo Nakamura *, Rafael Mira Libardi *, Paulo Sérgio Lopes Souza* and Stephan Reiff-Marganiec †
*University of São Paulo (USP)
Institute of Mathematics and Computer Science (ICMC), São Carlos-SP, Brazil
Email: {edvard, jcezar, btkuehne, dionisio, lhnunes, ljadami, nakamura, mira, pssouza}@icmc.usp.br
† University of Leicester
University Road, Leicester, LE1 7RH - UK
Email: srm13@le.ac.uk

***Abstract*—Fault tolerance techniques can improve the trust of users in service oriented architectures as they can ensure data availability. This paper presents an implementation of a novel fault tolerance mechanism in a SOA architecture which simultaneously provides increased availability and better quality of service. In addition to this mechanism, a service selector using reputation ratings of the architecture components is discussed. The selection is based on information from past transactions of the components of the architecture, which allows to identify the best web services able to meet the requests of customers. The mechanisms are tested and a performance evaluation is presented to validate the results.**

***Keywords*-QoS; Service Oriented Architecture; Fault Tolerance; Web Services;**

I. Introduction

With an increasing uptake of services users now have a choice of getting the same functionality from different providers. Other criteria, collectively referred to as non-functional properties or quality of service (QoS), are used to differentiate services based on their quality. The resulting recognised problem is to identify the best service based on a user's demands. However, according to [19], the efficient selection of web services with guaranteed quality of service has become a critical problem. The problem occurs due to the difficulty of obtaining values for service quality – service providers can improve their QoS intentionally in a dishonest way by providing unrealistically positive values in the registry. There are no good evaluation methods available to identify this behaviour, so there is a need to automatically determine values for non-functional properties based on service execution histories. Such a method will further help with fluctuations of quality that naturally occur during service use, based on request loads.

Fault tolerance allows a system to continue operating in the presence of one or more failures [15]. Systems that require security and fault tolerance mechanisms can use models based on reputation of system components, using only the best components in the execution of the system. Component faults are automatically detected and treated by these mechanisms [6]. Reputation is considered a collaborative model of services classification [19]. The feedback obtained from users can complement the automatic ranking mechanisms.

In the same way that fault tolerance is useful to ensure the availability of services in SOA, reputation mechanisms are important tools to support the service selection. It is possible to classify the web services to meet the requests of customers through analysis of data collected from previous executions, i.e. the collection history. For this to be reliable, the web services reputation needs to be regularly and dynamically recalculated based on QoS parameters such as response time, availability and security to select web services allowing to predict probability levels of QoS in the next invocations [12].

In this paper, we propose a novel mechanism for selecting services based on data mining, a logging module that bridges UDDI failures and a set of new service selection algorithm which ensure good selections in the presence of component failures. These elements are applied in the WSARCH architecture, where information from past transactions is collected, a neural network is applied to help determine providers and ultimately the best provider to meet a request is selected.

The remainder of this paper is organized as follows: Section II presents the WSARCH architecture. Section III shows related work. Section IV presents the contributed mechanisms. Section V details settings and the testing environment. Section VI presents the results. Section VII concludes the paper and identifies future work.

II. WSARCH

The architecture used in this work is called *Web Services Architecture* (WSARCH) [7] [1] and it will be briefly presented to allow system visualization and put the work in context. WSARCH has five distinct modules: the client application, the providers, the Broker, the UDDI registry and the Log Server. Figure 1 shows the relationships between these components.

Requests from clients arrive at the *Broker*, specifying the desired quality of service. The *Broker* is responsible for finding

[1] http://wsarch.lasdpc.icmc.usp.br

978-3-901882-48-7 © 2012 IFIP

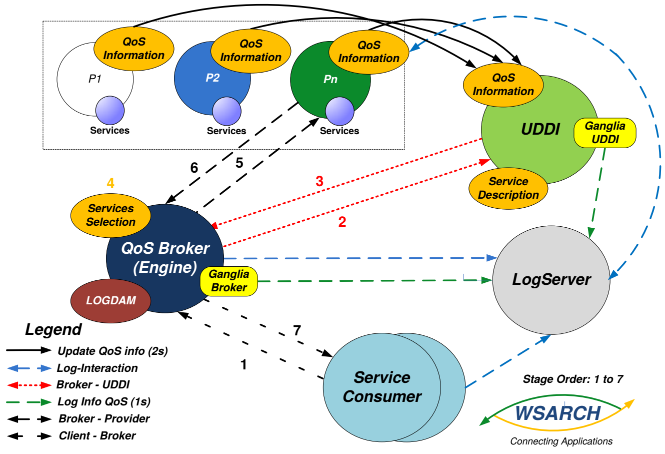

Fig. 1: *Web Service Architecture* - WSARCH. The Fault Tolerant Mechanisms are inside the Broker and in the LOGDAM. [7]

a specific service to meet the request. Services will be available in one of the providers [9]. Providers are hosts for services based on *Apache Axis 2* technologies. Axis 2 is responsible for processing the request and response messages [7].

The location information of service providers and their qualifications and characteristics are provided by the *Universal Description, Discovery and Integration (UDDI)* registry. The UDDI registry is used to look for published providers and services and here also provides QoS values. The UDDI repository could provide reputation data as well, enabling the *Broker* to rank the providers based on their historic reliability. Having access to QoS data and reputation allows the broker to select services such that the service level agreements (SLAs) negotiated between client and system are maintained.

The architecture also has the *Log Server*, which is a database responsible for storing all the data transactions made between components. Besides, information of quality of service offered by the various modules are updated every second, collected by a Ganglia monitor [14] and transmitted from each module to the Log Server under *Broker* management – ensuring that most up-to-date QoS data is avaialble for selection.

WSARCH was developed with a standard service selector (*Default Selector*), working directly in the *Broker*. This selector uses the user's QoS requirements as parameters to identify the best provider, always looking for best performance.

When a provider receives many requests, its QoS values are likely to decrease. This indicates that it may be getting overloaded. So, clients may have problems of not receiving expected answers, either by an increase in waiting time, or incomplete or incorrect responses. Clearly these lead to violations in SLAs in the extreme and can be costly for providers. The *Default Selector* avoids these situations by redirecting requests to another provider, least occupied at the given moment. This is possible as the QoS values are updated continuously at very small time intervals. Traffic is observed by a *Ganglia* monitor running in each service provider which transmits the data in real time to the *Broker*. Thus, the selector is able to make the most appropriate choice for users.

The Default Selector works well for balancing workload of the services, thus avoiding overloads that may lead to a decline in some components, and could cause further disorders. When a specific service provider has an increase in workload, the provider will be avoided until the number of requests decreases and it can work efficiently again. Here SLAs are assumed to be grouped in classes of service level (specifically "Gold", "Silver" and "Bronze"). The pseudo-code of the *Default Selector* is presented in Code 1:

```
retrieve the list of service providers offering the service

iterate over all providers:
    saves the largest and the smallest load
calculates the average load

computes the reference load by class:
if class == "gold":
    reference load <- lowest load
else if class == "silver":
    reference load <- average load
otherwise (class == "bronze"):
    reference load <- biggest load

iterates over all providers:
    calculates the Eucledean distance of each load to the
     load reference

return the provider with the shortest distance
```

Code 1: Outline of Default Selector Algorithm

III. Related Work

QoS aware service selection has been investigated for a number of years, with work focusing on modelling and using QoS properties to make good choices [8]. But generally to collect data at execution time and use this data is not considered. In our work the run-time behaviour is utilized to determinate the reputation of providers.

The idea that a client may have problems when seeking services only based on QoS is discussed in [10]. This is particularly true when users have access to a wide range services offering equivalent functionality. [18] suggest that selection of services should provide the best performance, according to the non-functional requirements. Although providers are able to guarantee a certain quality, they may end up failing to deliver on that promise, degrading the operation and user experience. Thus a reputation measure is crucial for business applications, especially for ensuring the safest choice of components and in order to highlight the most relevant and reputable services.

A reputation scheme is based on feedback related to services according to [2]. Some schemes may be unitary, based on a single value such as response time. Others are multi-layered considering more and more complex factors such as performance or reliability. The feedback from clients is the key to understand the reliability of web services with the model that captures the data being implicit or explicit. The explicit model presented is more accurate, but also computationally expensive and hence somewhat slow. It is also important to observe that each client can make a different assessment of the same aspects of a system according to their preferences [13]. However, while this seems bad, it actually creates a heterogeneity that is crucial to select services that meet a wider range of user expectations. The reputation models are based on in the E-commerce domain and thus do not consider the requirements of the web services domain adequately [8]. The main consumers of web service are other computers that must be able to evaluate their needs mechanically and should ignore minor fluctuations in QoS occurring due to dynamics in the network.

Considering some of the work that relies on user input in ore detail, [16] presents a QoS prediction system that uses the users reputation to determine the data needed for collaborative filtering. This system avoids untrustworthy user contributions and focuses on good feedback. This model depends highly on client acting, and might have weak results in unfriendly environments. [4] presents a system to preserve users privacy in service composition. It uses reputation to rank reliable mechanisms in order to avoid unreliable modules. It allows to quantify the risk of unauthorized disclosure of user information. The evaluation is not complete, as the authors propose to do in future works. In [11] user feedback is utilized as parameter to a reputation system, that combines credibility and sensibility of raters. However, again evaluation is not complete and dishonest raters are not considered in the work.

[17] presents an aggregated overall QoS based on *Broker* interaction rates. The experimental evaluation in this paper is quite weak, since does not present any details of the tests performed, rather it only states that the *Broker* achieved better results than before.

Even P2P systems have been using reputation to improve the user experience. Since they do not have a central autority, it is very difficult to coordenate the quality and security of the services available. In [5] is presented a model that encapsulates the behaviour of the peers and use it to predict the future. The focus of this work is to improve the bandwidth consume, in such a way that does not meet the needs of web services. And the work of [3] tries to ensure the quality of connection in networks with malicious peers. The scheme distinguish between the quality of service and the rating of the client. Despite of showing that the algorithms are robust, the system overloads in high rates of requests.

None of these solutions use reputation directly in the QoS aware selection of services. The work presented in this paper uses reputation and automatically gathered QoS data to increase reliability of the system by adding a fault tolerance mechanism which reduces probability of ultimate failure of system components.

IV. Fault Tolerant Transaction Mechanisms

This section will focus on the fault tolerant transaction mechanisms which are evaluated later in the paper. The last subsection explores the mechanism for introducing errors into the platform to allow for testing.

Figure 2 shows an overview of the overall contribution. On receiving a service request, the broker explores whether the UDDI repository is available or not. If the UDDI repository is available it will be used and the broker will make use of the *Risk Selector* to determine, rank and classify suitable service. If the UDDI repository is unavailable the *Intelligent Selector* will be used for the same purpose. Once a list of services with suitable services classes is available, a kind of intelligent round robin approach (based on service load) will decide on the most suitable service to handle the request.

In the following subsections, we will explore each of the selectors (*Risk*, *Intelligent* and *Round Robin*) in more detail. Specifically the subsections will describe the operation and need for the methods.

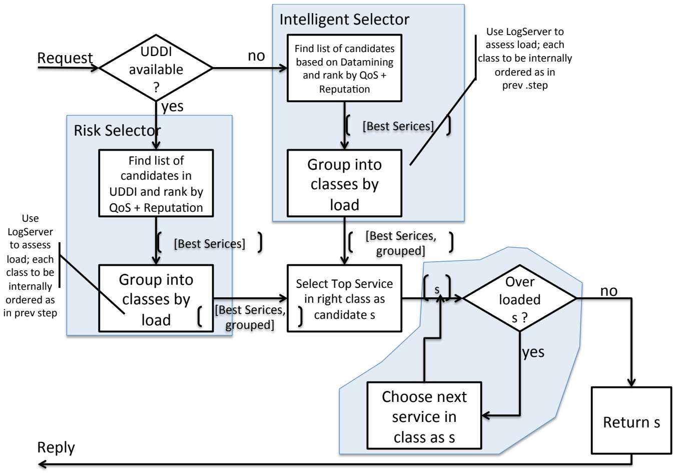
Intelligent Selector
Request
UDDI available ?
no
yes
Find list of candidates based on Datamining and rank by QoS + Reputation
Use LogServer to assess load; each class to be internally ordered as in prev .step
[Best Serices]
Group into classes by load
Risk Selector
Find list of candidates in UDDI and rank by QoS + Reputation
[Best Serices,
grouped]
Use LogServer to assess load; each class to be internally ordered as in prev step
[Best Serices]
Group into classes by load
[Best Serices,
grouped]
Select Top Service in right class as candidate s
s
Over loaded s ?
no
yes
Choose next service in class as s
Return s
Reply

Fig. 2: Overview

A. Risk Selector

In high-risk operations, whether financial or life-threatening, it is essential to ensure that the quality of service offered is as high as possible. It is imperative to ensure that only the best and most reliable components are involved in transactions to avoid large losses and at the same time to allow the user to feel confidence in the system.

The *Risk Selector* (see Code 2:) is a web services selector that unites quality of service with data reputation of the providers in order to provide both a favourable experience of use, and confidence to clients of the system. It looks for the safer providers to meet client demands. The search for providers uses QoS values to rank services, penalising providers with a high risk of failure or with a history of unsuccessful or unduly slow method calls.

This selector calculates a providers' rank, according to their behaviour to meet multiple requests. A provider that drops requests will have their risk increased, reducing the chance to be selected in the near future. The worse the behaviour of a provider, the further its risk will be increased.

Furthermore, the model developed assures that an offline provider has their risk value increased periodically (providers are tested every 1.5 seconds for availability). If the last response time from a service is more than five seconds ago, the risk is increased by one.

```

retrieves the list of service providers offering the
    service

iterates over all providers:
saves the largest and the smallest load
calculates the average load

computes the reference load according to the **class**:
    **if** "gold": the reference load <- lowest load
    **else if** "silver": reference load <- average load
    **else** ("bronze"): reference load <- biggest load

order providers based on the distance (Euclidean) of each
    load to the reference load

calculates the maximum risk in accordance with the **class**:
    **if** "gold": maximum risk <- 5
    **else if** "silver": maximum risk <- 10
    **else** ("bronze"): maximum risk <- 15

**while** the best provider (whose load is closest to the
    reference load) has a risk greater than the maximum
    risk:
removes the provider from the list of providers

    **if** the list is empty, the best provider will be the
    first of the previous ordination

returns the best provider
```

Code 2: Risk Selector Algorithm

The use of a selector that calculates the level of risk of sending a request to a particular server, makes the overall SOA (in this case implemented through WSARCH) more reliable and robust. This is achieved as services with bad response behaviours in the past who are hence high risk are not chosen as the *Risk Selector* ensures that the components with a recent history of problems are kept out of the transactions until inconsistencies are resolved.

A provider with a potentially dangerous ranking position could be chosen by the selector depending on the environmental situation at a given time. For example, all providers with good ranks start to get overloaded if large numbers of requests are made, and this will cause a reduction in the quality of service. As QoS is another selection criteria, the selector will choose one of the providers with higher risk, since its QoS will be better. However, hopefully this effect is mitigated by load balancing introduced later in this paper – unless the load for the system is so large that it cannot really fulfil requests in the first place (a situation where no solution apart from increasing resources will help).

B. Intelligent Selector

WSARCH has been enhanced with the *Data Analyzer Module for Serviced Oriented Architecture* (LOGDAM) [1] in a previous project (Fig. 1 already shows this component attached to the broker).

LOGDAM, as we will see, mines data from the log server. The *Intelligent Selector* draws on this mined data to decide on services in the case that the UDDI repository is unavailable – thus adding fault tolerance to a key component of SOA (namely the repository) by providing a suitable alternative.

Recall that all information from service interactions in the architecture is recorded in the *LogServer*. LOGDAM utilizes machine learning to determine which providers have record attendance favourable for each type of client. A trainer running in the background uses the IP address of the client, the service name, operation name and class of customer to understand good selection of providers for specific needs.

For ranking, the *Intelligent Selector* uses a *Naive Bayes* algorithm shown in Code 3. This is a simple probabilistic classifier based on Bayes' theorem. It is called *Naive* since it assumes that the attributes are independent. That is, the value of an attribute in a given class is independent of the values of other attributes. This is a reasonable assumption as the QoS parameters can independently take on any value in their range. For each request set the probability of being achieved by a provider will be calculated and the provider with the highest probability of meeting the parameters will be chosen.

```
Let $D$ be a set of training examples recovered from *LogServer*
Considering the existence of $n$ providers and a tuple $X$ to
    be classified
The provider $p$ of $X$ will be the provider that has the
    highest value of the posterior probability conditioning
    to $X$:
    $P(p_i|X) > P(p_j|X), 1 \leq i, j \leq n, i \neq j$
The goal is to maximize $P(c_i|X)$:
    $P(c_i|X) = \frac{P(X|c_i)P(c_i)}{P(X)}$
To reduce the computation of $P(X|c_i)$, the assumption *naive*
    that the attributes are independent of the tuple is
    made:
    $P(X|c_i) = \prod_{k=1}^{n} P(x_k|c_i)$
```

Code 3: Naive Bayes Algorithm

Over time, there is a concern that the data to be used by the machine learning algorithm in LOGDAM will be biased, since the same small set of services might be chosen as favourable (a risk of all machine learning algorithms). This potential problem is mitigated by the fact that a kind of round robin algorithm is used in the process to share load between a number of service candidates. Also, this mechanism has the disadvantage that it only relies on data from previous service executions and thus cannot detect newly added services. However, as the mechanism is seen as a fallback in case that UDDI is not available, new services added to UDDI will be used by the risk selector and can build up a profile and thus become available later in case of UDDI failures.

C. Load Balancing

To address the learning concern mentioned in the previous section, but more crucially to address the concern that the same service might be chosen as preferred candidate all the time a *Round Robin Selector* phase is added in the overall process. This selector provides load balancing across service candidates to ensure that we do not end up in a situation where the best service becomes overloaded by rotating request between candidates. This selector does not directly make assessments of QoS, nor does it assess the reputation of the providers. It uses the ranked lists provided by the *Risk* or *Intelligent Selector*, decides on the right service category required (e.g. "Gold" and then considers the load on the service candidates.

If the load of the most highly ranked service candidate is high, the next best will be considered until one with a reasonably low load is found and selected. (Note that the system might be so overloaded that no low load service can be found – we have not considered this, but a good solution might be to randomly assign a request to any of the candidates).

D. Fault Injection

For the experiments performed for this article, it was necessary to develop a tool to simulate faults in a service-oriented architecture.The providers of WSARCH operate from virtual machines based on *KVM*.

The risk selector was designed to deflect requests to providers with service problems. For example, once a provider is again showing as available, it is not immediately selected since its reputation rank would have been modified, deeming the provider unsafer. Only after some time in operation, the reputation will increase again. In general, the longer a provider is online the more reliable they are assumed to be.

For the experiments we created a shell script with the ability to pause virtual machines, thus emulating a failure (simulating server or network failure, as both would lead to provider unavailability). The script ensures that VMs recover to their previous state, thus making the provider available again.

The interval between failures refers to the time that a provider is available. The recovery time of failure refers to the time it takes for the provider to recover from when it is send into a failure state (that is a VM is taken offline). These intervals are randomly set, based on exponential distribution. Thus the events occur randomly, but with a set and mathematically predictable behaviour. An exponential function is used as a basis to define the ranges of failures during testing, considering the two different intervals, i.e. the interval between failure and the recovery time.

In detail, the interval between failures is generated as a random number based on a exponential function. The failure recovery time is considered to be the average of the same exponential function, but uses a fraction of the time between failure. This approach is used to ensure that recovery can occur between failures and there is time for the providers' rating to recover so that they can meet clients request again. The operation is presented in Code 4.

V. Performance evaluation

The main goal of this study is to evaluate fault tolerant techniques for Web Services selection using the reputation mechanisms. Thus, the testing environment is composed of several physical machines, each running VMs: There are 4 physical machines each running 3 client instances, again 4 physical machines each with 3 service provider instances and a further 4 machines each running 3 UDDI repository instances. Finally one physical machine is hosting the *Broker* responsible for controlling transmission of requests between the client and service provider. Each physical machine has 3 network interfaces, so that we can exclude contention on the network interface as a negative factor. This setup also dictated the maximal number of instances used in the experiments. The testing environment available is detailed in Table 1.

```
Average <= 200000
Off <= 10
On <= 40
Random <= 0
Pause <= 0
Control <= 0

Do 100 times

Random <= Random Value

If control = 0 do
    pause <= ((Random divided by Average) divided by
         Average)times 10000000) + Off
    Suspends the operation of the virtual machine
    Sleep Time = pause
    control <= 1

Else do
    pause <= ((Random divided by Average) divided by
         Average)times 10000000) + On
    Reactive virtual machine operation
    Uptime = pause
    control <= 0

End if.
```

Code 4: Fault Injection Script Algorithm

We analysed three scenarios in the test. They consider variations in the type of selector (Default or Risk), the presence (or absence) of faults and the number of UDDI repository instances. Note that we refer to the approach presented in this paper as Risk Selector in this section. The variants are presented in Table 2. The first scenario represents the base line considering the ideal environment (no faults occurring) allowing to compare the other two scenarios. Obviously since there are no faults in this scenario there was no fault treatment needed. The trial environment was configured with 1 and 12 UDDI instances representing the minimum and maximum capacity available.

TABLE II: Experiments Design

Scenario	Selector	Faults	UDDI
1	Default Selector	No	1 / 12
2	Default Selector	Yes	1 / 12
3	Risk Selector	Yes	1 / 12

The second test scenario was set with the *Default Selector*, and the environment was subjected to faults inserted into the providers, challenging the selector service to seeks to maintain good levels of QoS by selecting appropriate services. The first execution in this scenario was scheduled with a single UDDI repository, to exemplify the situation with minimal resources available and hence a higher overhead. In the second part of this test, the number of UDDI instances was elevated to 12, the maximum capacity of our test architecture to share the load between repositories. We would expect high selection times as faults will result in bad choices.

The third scenario presents the approach defined in this paper in the challenging situation it is meant to address, namely the *Risk Selector* subjected to an environment with failures to demonstrate its functionality and performance. We hope that the result should be close to scenario 1, and certainly

TABLE I: Infrastructure

UDDI		Provider	
CPU	4 Core - Intel Core 2 Quad 8400	**CPU**	1 Core - Intel Core 2 Quad 8400
RAM	4GB - DDR3	**RAM**	2 com (512MB) and 1 com (1GB) - DDR3
HD	Virtual HD - 50GB	**HD**	Virtual HD - 50GB
OS	Linux Ubuntu 11.10 Server	**OS**	Linux Ubuntu 11.10 Server
Quantity	**12**	**Quantity**	**12**
Broker		**Cliente**	
CPU	6 Core - AMD Phenom II X6 1090T	**CPU**	4 Core or 2 virtual Core
RAM	16GB - DDR3	**RAM**	8 / 4 GB (virtual
HD	500GB	**HD**	500 / 15 GB (virtual)
OS	Linux Ubuntu 11.10 Server	**OS**	Linux Ubuntu 12.04
Quantity	**1**	**Quantity**	**46**

superior to the results achieved by using the *Default Selector* (scenario 2).

VI. Results Analysis

The results of the first execution, with just one UDDI repository providing the *Broker* with the location of providers, show that the presence of failures does not degrade the work of the *Risk Selector*. The number of UDDI repositories positively influences the response time.

Considering the analysed scenarios, we observe that the *Risk Selector* is able to better withstand in a faulty environment. Figure 3 exposes the average times of changes. However, in a context with increased workload, surely this difference would be even larger, demonstrating the efficiency of the algorithm.

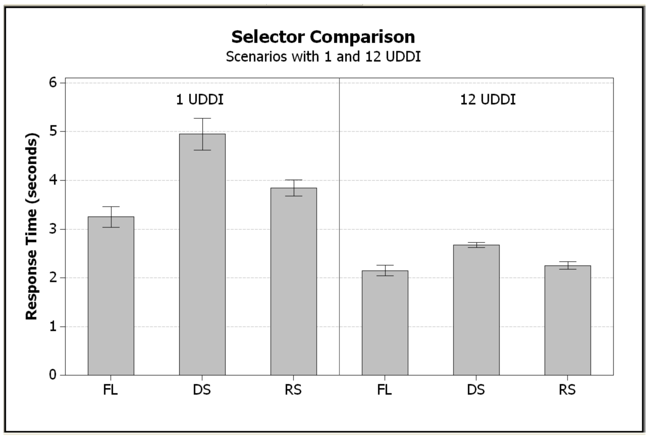

Fig. 3: FL = Faultless; DS = Default Selector with Fault Infection; RS = Risk Selector with Fault Injection.

The first half of the Figure represents the scenarios with 1 UDDI for the faultless, the *Default Selector* and the *Risk Selector* respectively. The second half shows the results of the experiments with 12 UDDIs. The faultless set represents the ideal environment, in which there are no problems and the users requests are always answered. The set with *Default Selector* represents the worst case possible, when the faults occur and there is no mechanism to avoid them. Besides, as we can see, the average response times increase significantly. In the experiment with 12 UDDIs the elevation is smoother. The scenarios in which the *Risk Selector* operated show that the algorithm is capable of avoiding faulty providers and thus getting closer to the ideal scenario. This demonstrates the efficiency of the proposed mechanism.

A. Fault Tolerance Mechanism Results

In experiments performed with the *Default Selector*, the providers were not injected with faults, since the selector is not able to treat them, and as noted earlier, the recorded times tend to be worse. However, in the tests made with the *Risk Selector*, the providers have suffered random stops to observe the behaviour of the reputation algorithm. This decision was made to show that *Risk Selector* is capable of improving the handling of requests in faulty environments.

Figure 4 (top) showns the values obtained for the two main selectors, acting to serve 16 clients and facing the static type of failure, in which the UDDI works just on the first requests. When the UDDI stops, the *Intelligent Selector* is activated, and if it causes overhead, the *Round Robin Selector* is triggered. Figure 4 (top) shows the results achieved by the selectors in the same conditions described above, but with 32 clients making simultaneous requests. These results are shown in Figure 4 (bottom).

A comparison between the *Default* and *Risk Selector*, serving 16 clients with a dynamic fault model are conducted. Here, the UDDI suffers interruptions from time to time, and made constant changes in context, activating the (*Intelligent* and *Round Robin*) selectors several times. The results are exhibited in Figure 5 (top). The same set of experiments with 32 users, lead to the results are shown in Figure 5 (bottom).

These results show that although the *Risk Selector* has slightly longer response times than the *Default Selector*, it is important to remember that the *Default Selector* one is not experiencing faults in the providers as we know it performs badly from the first experiment. And, as we can see in Figure 6, the *Risk Selector* presents a lower rate of dropped requests because of the better selection system.

Along with that, the *Risk Selector* also achieves a better distribution of the workload between the providers in the architecture, as shown in Figure 7. This is a relevant point as it will avoid overheads in the modules. With that it is possible to see a improvement in the QoS, since the clients has less problems with their requests and has to wait less to the answers.

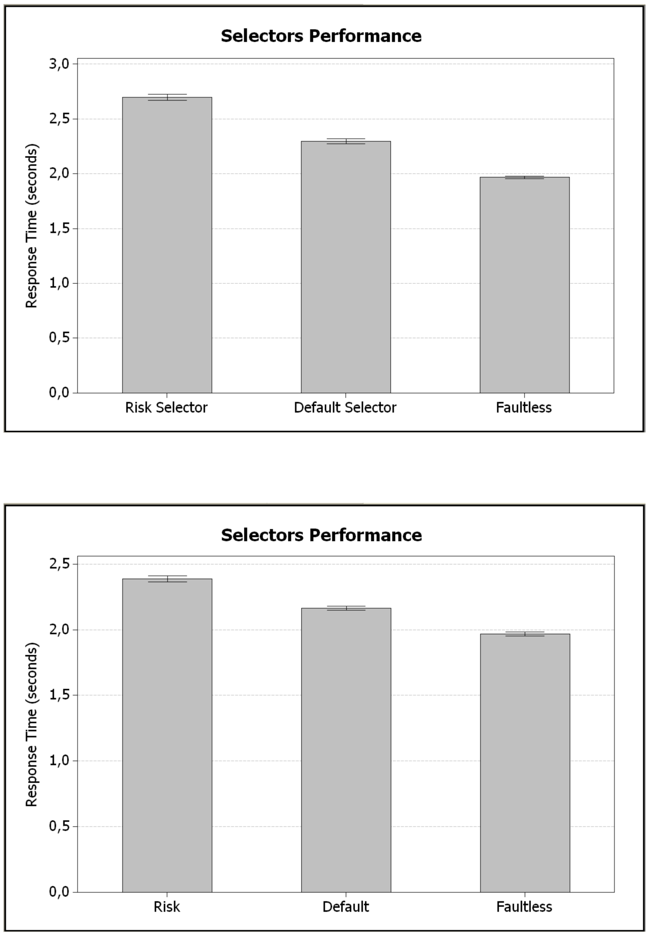

Fig. 4: Comparison of response times between the selectors, with 16 (top) and 32 clients (bottom) respectively, and static failures (UDDI repositories dropping out and remaining unavailable).

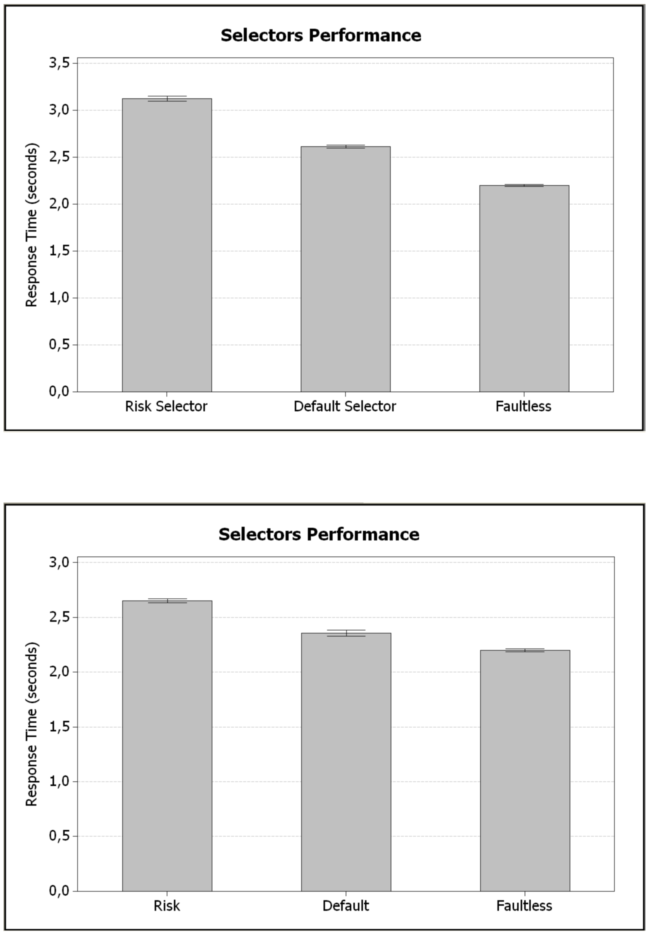

Fig. 5: Comparison of response times between the selectors, with 16 (top) and 32 clients (bottom) respectively, and dynamic failures (UDDI dropping out and coming back repeatedly).

B. *Influence of Factors*

Through an influence of factors analysis we can determine which of the factors involved in the experiments have a bigger impact on final results. Figure 8 (top) shows that the two factors most distant from normal line are the type of selector and the number of clients. It indicates that those change the behaviour of the system and altered the response. In Figure 8 (middle) is possible to observe the behaviour of the system when the factors are changed. And finally the interaction of factors (in pairs) is represented in Figure (bottom). The proposed algorithms are able to deal with the failures and hide them of the clients, in such a way that the faulty modules have time to be fixed and the interactions do not have to be stopped.

VII. Conclusions and Future Work

The results from the experiments lead to the conclusion that the *Risk Selector* has the capacity to optimize the distribution of workload between the components, even in the presence

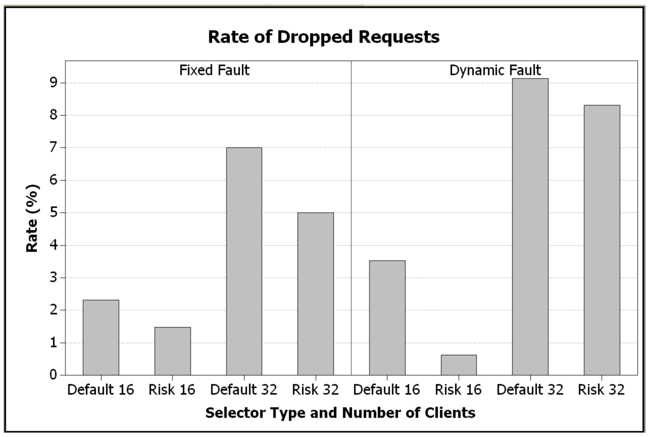

Fig. 6: Dropped Requests

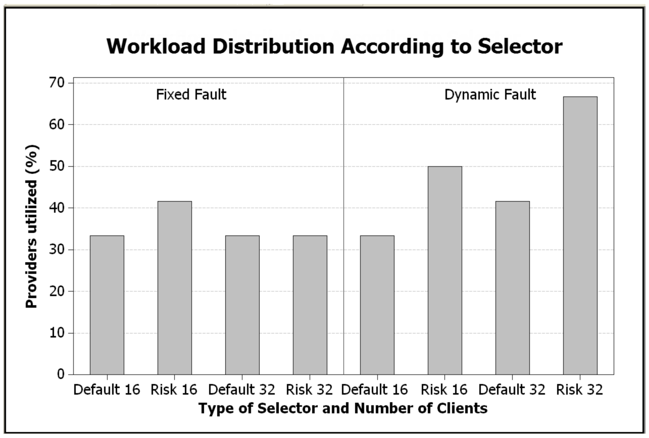

Fig. 7: Workload Distribution

of faults. The *Default Selector* has a good operating capacity, but it is blind to faults and recoveries of providers, ignoring unfavourable historical behaviour. When there are no faults, the *Risk Selector* works in the same way as the *Default Selector* and the ranking system functions in background, not causing overload.

The reputation system proved its importance by keeping providers with recent failures as undesirable until their valuation levels stabilized and they are again able to answer incoming requests reliably. Response times show the efficiency of the model, which maintains adequate service times. In some cases, the *Broker* suffered due the the high number of requests, which caused an increase in search time. Besides, the reputation mechanism showed the expected behaviour, allowing to overcome the failures in the components and thus keeping the architecture in operation.

The *Intelligent Selector* presented excellent results, being a very reliable substitute for the UDDI repository when faults occurred. It is able to suggest providers, keeping the *Broker* able to organize service requests and their responses. In order to avoid the overload of providers the *Round Robin Selector*, switching between alternative providers worked well.

Finally we conclude that the presented combination of Selectors present an important evolution of the WSARCH's standard selector, making possible the risk analysis of the components in real time. The suggested selectors would be expected to be a good addition to any broker based QoS aware selection.

In future work, we intend to extend the reputation rating to other components of the architecture such as UDDI, to obtain a pattern of scalability for web services transactions. Since the Broker is the central module of the WSARCH, there is a replication of it, and a substitution system can be installed in case of a general failure. A version of the architecture without the UDDI is in study also. Fault tolerance mechanisms will be expanded to cover other components.

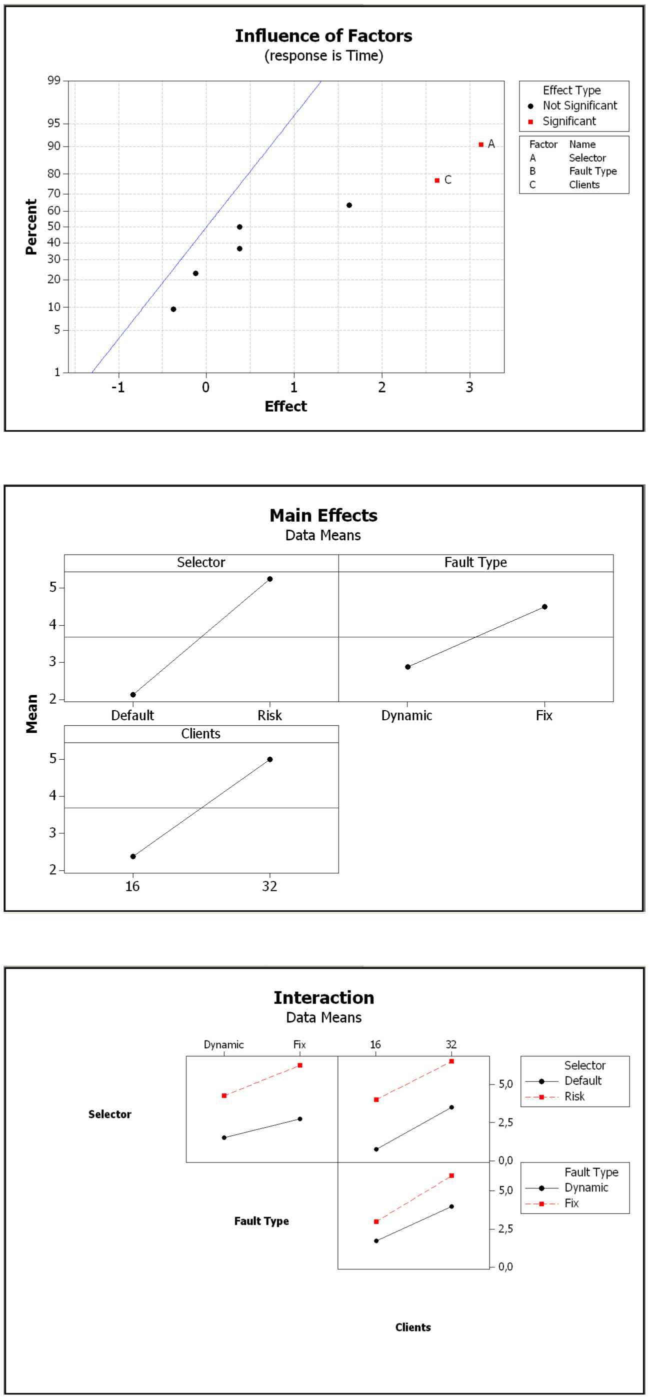

Fig. 8: Influence of factors analysis, whit main effects and interaction data

ACKNOWLEDGMENT

We thank CAPES and FAPESP (in processes 2011/09524-7, 2013/26420-6, 2011/12670-5), for the support of this research. We also like to thank ICMC-USP and the LaSDPC for offering the necessary equipments for this study. Some of this work was conducted while Stephan Reiff-Marganiec was on study leave from the University of Leicester.

References

[1] L. J. Adami and J. C. Estrella. A data analyzer module for logs of service oriented architeture. In *3rd High Performance Computing Regional School (ERAD)*, jul 2012.

[2] J. Al-Sharawneh, M.-A. Williams, and D. Goldbaum. Web service reputation prediction based on customer feedback forecasting model. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2010 14th IEEE International*, pages 33 –40, oct. 2010.

[3] E. Ayday and F. Fekri. Bp-p2p: Belief propagation-based trust and reputation management for p2p networks. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2012 9th Annual IEEE Communications Society Conference on*, pages 578–586, June 2012.

[4] E. Constante, F. Paci, and N. Zannone. Privacy-aware web service composition and ranking. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 131–138, June 2013.

[5] P. Dewan and P. Dasgupta. P2p reputation management using distributed identities and decentralized recommendation chains. *Knowledge and Data Engineering, IEEE Transactions on*, 22(7):1000–1013, July 2010.

[6] K. Echtle and A. Masum. A fundamental failure model for fault-tolerant protocols. In *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, volume 0, pages 69–78, 0 2000.

[7] J. C. Estrella, R. H. C. Santana, and M. J. Santana. *WSARCH: An Architecture for Web Services Provisioning with QoS Support - Performance Challenges*. Saarbrucken : VDM Verlag Dr. Muller GmbH & Co, 2011. http://www.amazon.com/WSARCH-Architecture-Provisioning-Performance-Challenges/dp/3639378245.

[8] X. Fu, P. Zou, Y. Jiang, and Z. Shang. Qos consistency as basis of reputation measurement of web service. In *Data, Privacy, and E-Commerce, 2007. ISDPE 2007. The First International Symposium on*, pages 391 –396, nov. 2007.

[9] P. Harshavardhanan, J. Akilandeswari, and R. Sarathkumar. Dynamic web services discovery and selection using qos-broker architecture. In *Computer Communication and Informatics (ICCCI), 2012 International Conference on*, pages 1 –5, jan. 2012.

[10] D. Janardhan and S. Devane. Web service reputation-based search agent. In *Research and Development (SCOReD), 2009 IEEE Student Conference on*, pages 184 –187, nov. 2009.

[11] G. Li, D. Song, L. Liao, F. Sun, J. Du, and K. Yang. A novel reputation model for web services selection with raters' sensitivity. In *Service Systems and Service Management (ICSSSM), 2013 10th International Conference on*, pages 708–712, July 2013.

[12] M. Li, J. Huai, and H. Guo. An adaptive web services selection method based on the qos prediction mechanism. In *Web Intelligence and Intelligent Agent Technologies, 2009. WI-IAT '09. IEEE/WIC/ACM International Joint Conferences on*, volume 1, pages 395 –402, sept. 2009.

[13] H. Liu, F. Zhong, and B. OuYang. A web services selection approach based on personalized qos prediction. In *Parallel and Distributed Computing (ISPDC), 2011 10th International Symposium on*, pages 199 –206, july 2011.

[14] M. Massie, B. Chun, and D. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[15] N. Mohamed and J. Al-Jaroodi. A collaborative fault-tolerant transfer protocol for replicated data in the cloud. In *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pages 203 –210, may 2012.

[16] W. Qiu, Z. Zheng, X. Wang, X. Yang, and M. Lyu. Reputation-aware qos value prediction of web services. In *Services Computing (SCC), 2013 IEEE International Conference on*, pages 41–48, June 2013.

[17] M. Rathore and U. Suman. Evaluating qos parameters for ranking web service. In *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pages 1437–1442, Feb 2013.

[18] S. Wang, Z. Zheng, Q. Sun, H. Zou, and F. Yang. Evaluating feedback ratings for measuring reputation of web services. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 192 –199, july 2011.

[19] X. Zhu, B. Wang, and S. Wang. Reputation-driven web service selection based on collaboration network. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 704 –705, july 2011.