Deploying fault tolerant web service compositions

Nuno Laranjeiro and Marco Vieira

CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal E-mail: {cnl, mvieira}@dei.uc.pt

Many businesses are now moving towards the use of composite web services. These consist of a collection of web services working together to achieve an objective. Although they are becoming business-critical elements, current development tools do not provide a practical way to include fault tolerance characteristics in web services compositions. This paper proposes a mechanism that allows programmers to easily develop fault tolerant compositions using diverse services. The mechanism allows programmers to specify alternative web services for each operation and offers a set of artifacts that simplify the coding process, by automatically dealing with all aspects related to the redundant web services invocation and responses voting. The mechanism is also able to perform a continuous evaluation of the services based on their behavior during operation. The approach is illustrated using compositions based on services publicly available in the Internet and on the services specified by the standard TPC-App performance benchmark.

Keywords: web services; composition; fault tolerance; diversity.

1. INTRODUCTION

Computer or service outages can result in a huge loss of money or in an unaffordable loss of prestige for companies. In fact, due to the impressive growth of the Internet, some minutes of unavailability in a server somewhere may be directly exposed as loss of service to thousands of users around the world.

Web services (WS) provide a simple interface between a provider and a consumer and are a strategic mean for data exchange and content distribution. Ranging from on-line stores to media corporations, web services are becoming a key component within the organizations information infrastructure.

Many businesses are now using composite web services. Web services compositions [1] are based on a set of services working together to achieve an objective. This composition is normally defined at programming time as a “business process” (specified using for example the Business Process Execution Language for Web Services [1] or BPEL for short) that describes the sequencing and coordination of calls to the component web services.

In a web service composition, if a service with bad performance is used the overall performance of the composition may decrease. Additionally, when one component web service fails the entire composition may be affected. In fact, as a composition consists in a sequence of different web services invocations, where the results from the execution of one web service are passed to the next in an atomic manner, if a web service fails the entire execution may have to be aborted. This way, including fault tolerance characteristics in web services compositions is a key step to achieve high dependability and performance.

Redundancy is used in several contexts to tolerate failures and attain high availability. Design diversity is typically the solution to deal with design faults, where two or more different implementations of a component or service are used for fault-detection and fault-tolerance [2]. However, current support for web services composition does not provide the capabilities needed for programmers to easily develop compositions with fault-tolerance characteristics.

The paper proposes a mechanism that allows programmers to easily develop fault-tolerant compositions using diverse web services. Several well known techniques like redundancy, voting, etc, were studied and used taking into account the specific architectural characteristics of web services compositions. The proposed mechanism, named FTWS (Fault Tolerant Web Services) and available at [3], allows programmers to spec-
ify alternative web services for each operation and offers a set of artifacts that simplify the software design and coding process. FTWS deals with all the aspects related to the redundant web services invocation and responses voting. The programmers only need to perform a small set of configurations and provide (if necessary) some input/output adapters.

FTWS includes an evaluation functionality that allows programmers to assess and compare alternative services starting from the development phase. Afterwards, the evaluation mechanism performs a continuous assessment of the services based on their behavior during operation. This data is used by the voting mechanism to resolve impasses. Three core metrics are considered: response time, availability during operation, and correctness of the results. A composed metric based on these three can also be defined.

The structure of the paper is as follows. Next section presents some background and related work. Section 3 presents the mechanism for fault tolerant web service compositions development. Section 4 presents the experimental evaluation and Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK

In an environment based in web services a provider supplies a set of services for consumers [4]. The Simple Object Access Protocol (SOAP) is used for exchanging XML-based messages between the consumer and the provider over the network (using for example http or https protocols). In each interaction the consumer (client) sends a request SOAP message to the provider (the server). After processing the request, the server sends a response message to the client with the results.

A web service is described using WSDL (Web Services Definition Language) [5], which is a XML format used to generate server and client code, and for configuration. A broker is used to enable applications to find web services. Figure 1 presents a typical web services infrastructure.

Typically, web services compositions consist in a set of component web services that are invoked in a sequence that represents a specific business process. Component services can be provided by the same or different entities (having different providers is the most common situation). The composition is then used by client applications to perform a complex operation (a client application can also be another web service composition). Figure 2 presents an example of a typical service-oriented environment based on web service compositions.

IBM recently released FAWS [6], an application that aims to provide fault tolerance for the web services technology. FAWS consists of several independent components designed to operate in a distributed network and it is based in passive replication. In each moment, one of multiple servers is acting as primary and is responsible for all request handling. There is no state update between servers (no information is shared between primary and backup servers). The FAWS mechanism provides client-transparent fault tolerance behavior [6]. However, as only simple replication has been considered, there is no support for design diversity.

In [7] an N-version based mechanism (WS-FTM) is used along with a voting strategy to increase reliability in web services. WS-FTM utilizes a layered architecture that extends Apache Axis 1.1 [8] standard classes. It still uses RPC, and no compliance is assured to the WS-I Basic Profile standard [9] that promotes interoperability for web services development. The use of this framework also couples the developer to the Axis SOAP stack which may be undesirable in some cases and brings complexity to the development model. No support for diversity at the interface level of the component services is supplied. Note that this type of support is very useful when selecting alternatives for a service using third party components as they usually provide their functionality through different interfaces.

In [10] a mechanism is proposed that enables the creation of connectors that provide fault tolerance to unreliable web services. These connectors can be designed by clients, providers or other users (e.g., dependability experts) starting from the WSDL service description. The connectors provide runtime assertions useful for error detection, recovery actions based on several replication models and they can also be used for monitoring and error diagnosis. Results presented appear promising, however the actual development model is far from being simple.

In [11] the authors discuss vertical and horizontal composition in Service Oriented Architectures (SOA). The former is used for functionality extension and the latter for dependability improvement. The authors propose developing new workflow patterns and implementing them in different popular workflow languages. FTWS focuses on the horizontal composition introduced in [11]. A key difference between both works is that FTWS will not extend any particular workflow language in any form. In fact, FTWS is now based in a configuration scheme that is separate and independent from any workflow engine or language chosen by developers.

In [12], the authors identify relevant parameters that have impact in web services dependability. As in FTWS, [12] explores redundancy to enhance dependability. In fact, space and time redundancy methods are described and experiments are undertaken to evaluate the availability of web services. A replication manager is also used to apply several replication schemes in order to increase the availability of web services.

3. DEVELOPING FAULT TOLERANT WEB SERVICE COMPOSITIONS

Distributed applications over unreliable transport channels such as the internet often suffer from long response delays or temporary unavailability. For example, in a composition when a component web service fails or shows bad performance the entire web service composition may be affected. Although the provider of the composition may try to guarantee that both the infrastructure and software used are built in such a way that provide high performance and dependability (by doing extensive tests, code reviews, selecting the adequate infrastructures, etc.) he cannot guarantee any quality attribute in what concerns the component web services that are provided by third parties.

When dependable compositions are required, the programmers have to select the component web services and have to include in the composition all the code required for redundant invocation and responses validation and selection. This is a very difficult, high time consuming, and error prone task for several
reasons: concurrent programming (e.g., using threads) in web services compositions is complex; there are no practical evaluation mechanisms available; voting mechanisms must be embedded in the composition; the comparison of results is difficult as different services may respond using different data formats (the same is true for input parameters); voting impasses are difficult to resolve; deadlines for execution are difficult to implement and detect; etc.

To overcome the difficulties mentioned above and to facilitate the job of the programmers we propose a mechanism that automatically deals with all the aspects related to the redundant web services invocation and responses voting. Obviously, for some specific cases transparent fault tolerance may not always be the best solution. For instance, in legacy or complex applications, exceptional service behavior may be useful information for the developer to make some design or optimization decisions.
FTWS does not currently support the propagation of target services exceptions to the end client. However, these are logged by the application in order to provide some useful information for developers. FTWS usage requires the modification of the typical web service compositions environment. As shown in Figure 3, instead of invoking component web services, the compositions invoke a proxy web service (a new element in the environment). The proxy web service then executes the alternative component services in a redundant manner and deals with all the voting and evaluation issues.

Obviously, the implementation of this mechanism raises several relevant problems:

- **Evaluation of alternative web services**: as mentioned before, we believe that it is important for the programmers to have a practical mechanism to evaluate and compare alternative web services. Additionally, evaluation metrics can also be used to overcome results voting impasses and achieve low-cost redundancy (see details below). This way, the mechanism must be able to perform offline (before deployment) and online (during utilization) evaluation of the component web services.

- **Implementation of redundant invocation**: low performance overhead is obviously a major issue. Additionally, as compositions use alternative component web services that may provide slightly different interfaces and response formats we need to define an easy way for the programmers to reconcile input and output data (preferably some standard adapters should be defined). Finally, concurrent execution must be implemented and the definition and detection of deadlines must be possible (deadline violation may indicate that a given web service is unavailable).

- **Voting**: voting algorithms are well understood in the literature [13]. A major issue in this context is to resolve impasses. In fact, as typically there is a small number of alternative web services available (sometimes it is difficult to find more than two or three alternative services) impasses may become a major issue. For example, when only two web services are available and both provide different results, there is an impasse and no result can be obtained. However, in some cases it is well-known that one of the services typically provides correct results (this can be observed during testing or based on previous utilization history). Our goal is to endow the voting mechanism with an impasse solving feature (see Section 3.5).

- **Low-cost**: the use of commercial web services is typically paid. This way, an infrastructure based in redundancy implies additional costs for the provider of a web service composition. For example, if a given composition uses five alternative web services to perform a given operation, the cost of that operation is five times higher than if a single web service is used. Obviously, if the goal is high-dependability this may not be a major issue. However, in many cases the provider may want to reduce costs while providing a robust service. Our goal is to endow the mechanism with the capability of switching between an execution mode based on the simultaneous execution of all the alternative services and a basic recovery blocks [14] execution mode (sequential web services execution). This consists in the execution of the alternative web services in a sequential manner until a response is obtained (see Section 3.2). Obviously, the identification of the execution order is a problem that must be resolved.

- **Consistency in stateful web services**: consistency is a major issue in web services when the execution changes the internal state. During redundant execution, if one of the web services does not provide the result, the execution of the composition may continue as some component web services have completed the request. However, the state of the failed web service is no longer consistent with the state of the services that completed the execution successfully. This way, it is necessary to study new algorithms and to implement features that guarantee the recovery of the component web services after failures.

This paper focuses on the definition and development of a mechanism that overcomes these problems with exception of the last one. In fact, there are several challenges to overcome when considering the use of stateful services in systems like FTWS. This class of services is not handled here and this topic remains an open issue for future research work.

### 3.1 Architecture and operation

The FTWS system [3] uses a simple architecture based on the latest standards and available technologies to build and provide web services. Figure 4 presents a detailed view of its internal architecture.

FTWS uses a widely known AOP (Aspect Oriented Programming) framework to, in a transparent way, intercept each client request, and do the necessary tasks to calculate a response. The intercepted method signature and its arguments are delivered to a dispatcher (see Figure 4). This is the core component of the system and is responsible for resolving each request and coordinating the whole replication/adaptation process. The intercepted information consists of a method signature and its arguments, which correspond to our generic proxy service (this is the web service exposed by FTWS that will serve as basis for the interception of service invocation, and which will then trigger all subsequent FTWS procedures). At the dispatcher, the configuration is consulted to see what target services are related to this generic service and what invocation method applies (sequential or parallel). This target information consists in two method signatures that matches proxy classes previously generated with the wsimport tool (see a small description in section 3.6). After this information has been collected, and for each service to be executed, the generated class that represents our service provider is instantiated by reflection (this is the one that is referred in our configuration file), the service port is then obtained by calling the already configured method by reflection. Finally, the second signature (that represents our service invoker) is invoked, also using reflection, to execute the desired operation on the target service. This reflection-based mechanism enables the user with a transparent way of invoking services only by providing a few methods signatures. The user does not need to write client code, the only important task is configuring FTWS as the framework handles almost all remaining technical details. Refer to section 3.6 for full details on how to use FTWS.
The FTWS solution consists of a lightweight application that can be used without a standard application server. This is an excellent solution if low memory footprint is required, or scarce computational resources are available. Nevertheless, for other cases, the application can also be deployed in an application server in a traditional way. In addition, FTWS enables the use of services that have different interfaces through the use of convenient adapters (see Section 3.4 for details). Several configurable voting strategies are offered with the possibility of easy extension (see Section 3.5 for details on the response voting process). As in some cases voting may result in a draw, historical data are kept and used to solve these impasses, which can also be used to define a higher call priority for services that presented better behaviors in the past. A web services evaluation mechanism is also provided.

3.2 Evaluating alternative web services

The evaluation and selection of alternative web services to improve performance and dependability is an important aspect. In fact, as the web services used in a given composition may be supplied by several different providers and have different characteristics in terms of performance, availability, etc, making the right choices is important to improve the effectiveness of the composition.

The FTWS evaluation mechanism characterizes each web service using three core metrics and a composed adaptable metric:

- **Response time (RT):** characterizes the average time (in milliseconds) each component web service needs to execute.
- **Response correctness (RC):** characterizes the accuracy of the responses returned by a web service. Correctness is given as a ratio between the number of times the web service returns a correct answer and the total number of times the web service is executed. Correct responses are identified during the voting process.
- **Response availability (RA):** characterizes the web service availability during operation. The web service is considered available when it is able to respond. For example, a given web service is not available if it is invoked and returns no
answer or returns an error. Service availability is given as a ratio between the number of times the web service returns an answer and the total number of times the web service is executed.

- **Composed metric (CM):** it is defined based on the three core metrics presented above. It consists in the following formula: \( CM = RTw \times RT + CRw \times CR + RAw \times RA \), where RTw, CRw, and RAw, represent the weight of the respective core metric. This metric allows programmers to associate a different importance to each core metric.

The metrics can be collected offline or online. The offline evaluation is typically performed while developing a new composition. The results can be applied to select the best web services and are used later as the basis for the online evaluation. Obviously offline evaluation is not possible when the evaluation tests change the state of the service (e.g., change the state of the data used) and there is no parallel infrastructure that can be used only for testing purposes. In this case the programmer must define for all web services the default value for each metric (which are used latter as the basis for the online evaluation).

The online evaluation is automatically performed during the utilization of the web services. When a web service is invoked, the FTWS mechanism collects some information about the behaviour observed. This information is used to maintain the characterization metrics updated.

Although our proxy-based approach can give a biased perception of the real Quality of Service (QoS) for each target service (i.e., it does not represent an absolute and possibly accurate measure of the service’s real performance at a given moment), this perception represents the quality of the service from the client point-of-view. In fact, since clients typically do not have access...
to the target services infrastructure (except when they are also providers) such a system can extract relevant measures that are independent from using the system or not, as this biased perception will exist in both cases.

### 3.3 Operation modes

FTWS provides two modes of operation:

- **Simultaneous invocation of the alternative component web services**: for each given operation all the available web services are executed in simultaneous. The obtained responses are then voted in order to select the final result. Two voting mechanisms are provided: a unanimous voter and a consensus voter. The consensus voter can be complemented with the metrics information mentioned before to overcome impasses.

- **Sequential invocation of the alternative component web services**: available web services are executed sequentially as needed. The mechanism starts by executing one service and waits for the response. If there is no response then another service is executed. Invocations stop when a service returns a result or when all services available have been used (an exception is raised if none of the services returns an answer). Since the mechanism returns at the first available answer, no voting is done. The first option is the best and will be the one to return to the client.

In the simultaneous invocation method, several threads are created to invoke the corresponding target services. The dispatcher waits for thread termination or timeout before proceeding to response analysis and delivery. If the programmer does not define a timeout, the application assumes a default value for all services. If history on response times is available, the application can be configured to use it to calculate the timeout value, which is typically equal to the maximum response time observed excluding the potential outliers. For this, we consider that, for a given service, response times that diverge from the average more than N times the standard deviations are outliers (N is defined by the FTWS user). A keep value can be added to the maximum response time observed (excluding outliers) in order to reduce timeouts.

In the sequential invocation mode, the web services execution sequence is defined based on the information available for each service. The services are ranked based on the evaluation metrics (see Section 3.1) as selected by the programmer. For example, the programmer may specify that the web services should be ranked using the response correctness metric, the response time metric, the availability metric, or the composed metric (which allows a more elaborated ranking approach).

The sequential invocation mode is quite useful when the cost of the services is an issue. It allows a cost reduction by executing the minimum number of services needed to obtain a response. Obviously it must be applied only when the services being used provide high correctness. When a service does not provide high correctness values and is the first to be consulted, there may be a high probability of returning an incorrect answer. This highlights the importance of choosing an appropriate sorting method. Sorting services, for instance, by response time can be a good practice if the fastest services also present high correctness values (or when wrong answers are acceptable). In other cases, the sorting method will render our composition useless. All the available metrics can be collected during offline evaluation, and this preliminary evaluation can be used as basis for choosing the adequate sorting method.

In practical terms, during execution a list of target services is consulted. This list is sorted from the best to the worst service and is updated continuously. This means that each time a service is executed and its score changes the corresponding list is reordered. As this may be quite expensive in computational terms, list sorting may be performed periodically in configurable time intervals.

To reduce input/output operations, service score information is stored in memory during the application’s life cycle. This information can be written to secondary memory on demand or when service scores change by changing an attribute in the configuration file.

Our mechanism allows periodic switching between the two modes. This switching feature is important to maintain the metrics up-to-date. An aspect that needs to be considered is that, in the sequential operation mode there are services that are only used if other services fail. This may lead to the aging of the metrics for the services that are not used frequently. Thus, providers must periodically move to simultaneous invocation mode in order to keep the metrics up-to-date.

As we can see, these operation modes complicate the use of such a system, when having stateful target web services. In fact, it may not be useful to use such a system when considering the sequential invocation mode. All services not used in a given invocation would have to be updated some time later, i.e., we would be using, in fact, all of the available services, and this would no longer be a low cost operation mode. However, it would remain an interesting option when one is interested in obtaining an answer from the service that is historically better (regardless of the chosen criteria). Considering the simultaneous invocation of stateful services, such a system would have to guarantee that all services are updated, even in the presence of faults, which is an issue that has no trivial answer. Stateful service usage remains an open issue for future research.

### 3.4 Adapting web services invocation

An important characteristic of the FTWS mechanism is that it can use component web services that have different interfaces, providing developers with more options to build their solutions.

The mechanism uses an Adapter Design Pattern inspired implementation [15] to provide easy uniform access to different target services. It does so by transparently providing a generic and standards compliant web service as an interface to multiple target services. These target web services may have been built for distinct purposes, to a certain extent, but can still be used in an aggregated generic solution (provided the semantic meanings of their input and output messages are similar). An example of this can be seen in the generic Zip Service (see Section 4) created to return geographic details for a provided zip code. In fact, one of the target web services used is a weather condition reporting service, but it includes in its answer the required information for the Zip Service.
The mechanism does automatic parameter copy, based in a parameter map provided in the configuration file (see Figure 5), and allows the use of more complex transformation logic using custom adapters. For example, a converter may be used simply to convert a word to a boolean value or an adapter may be included to extract a date from a string. An important detail is that web services responses may also have different formats. Thus, adapters can also be configured to concatenate responses into a standard format. FTWS provides a set of standard converters (some quite complex) and allows programmers to define adapters that perform more complex logic that may be required to convert specific responses. The current version of FTWS supports primitive data types and complex data types in input parameters and response parameters in all possible combinations, nested complex types are not supported by the current version.

To illustrate the adaptation process, consider the case where we have 4 services that accept a temperature value as input. Three of these accept values in Celsius and the remaining service accepts values in Fahrenheit. If we wanted to model this with FTWS it would be necessary to create a generic service that accepted a temperature value, but it would also be necessary to define one Celsius to Fahrenheit converter (the other way around would be of a computationally higher cost). So, the tasks are: 1) map possible non matching parameter names (between our generic service and each target service) in the configuration file; 2) Code our converter; 3) reference our converter in the configuration file.

Figure 5 represents a partial example of a configuration file. The parameter celsiusTemp in our generic web service maps to the FahrenheitTemp parameter in the target service (one map must be built per target service if any kind of conversion is needed). Furthermore, we are indicating that class C2FConverter is the class to be used for parameter conversion in every invocation (FTWS knows which converter method to invoke as the class implements IConverter, a well known interface). Reflection is used to create the converter object and to invoke its well known conversion method in runtime. The temperature value will be transparently converted from Celsius to Fahrenheit by FTWS and this is the value that will be used as FahrenheitTemp to invoke the target web service. Obviously, a similar approach may be necessary to map and/or convert the target service’s response.

### 3.5 Selecting web services responses

When a request dispatcher determines that all (or the maximum possible) answers were collected, it transfers them to a decision manager (see Figure 4). This will be responsible for coordinating a voting process that will determine the best of the answers obtained.

The decision manager has access to a voter list (represented by the Voter Provider component in Figure 4). This list is defined and sorted by the application user and includes two predefined voters. New voters can be easily added. The standard voters include a unanimous based voter and a consensus based voter. The former votes each answer as correct if all of the collected answers are consistent. The latter marks a response as correct if a majority of consistent answers exists. The concrete process for this voter includes the evaluation of a result list of converted target responses.

In a first step, each response in the list is associated with a value that represents the number of answers that are equal to it. A relevant fact is that this ‘message’ comparison is not done directly over the SOAP messages, as that would be an extremely difficult and error-prone process. Instead, the comparison is done at object level (i.e., programming language level). This implies that complex object types implement the ‘equals’ method or, in alternative, the ‘Comparable’ Java interface (otherwise the comparison process would become more difficult). Since we are dealing with our own generic answers (i.e., we are not comparing each of the target service’s specific answers, but our already converted answers), there is no need to apply difficult comparison logic over different objects; this comparison logic is embedded once in our generic response object. After this first step, the maximum value obtained is stored. Then, all of the answers that have a cardinality value equal to the maximum are marked as correct. At the end of the process the voter returns a value that enables the decision manager to know that no correct response was found, a single correct response was found, or multiple correct responses were found. Figure 6 illustrates the basic algorithm (in source code form) of the unanimous based voter provided with FTWS.

At the decision manager, if only one response was marked as correct, this is the response to return to the dispatcher and subsequently to the client. For the other two cases we have a tie and if there are no more voters to break the tie, it is necessary to consult historical information.

If multiple correct answers were found at the decision manager, these are separated from the wrong answers and the historic information is searched using the service identifier of each of the answers. As the service list is, in principle, ordered for the configured criteria, the best services will be first in the list. This means that we can find the best service (for the chosen criteria) searching the list from the beginning and the best will be the one whose identifier is contained in the list of correct answers found (the answer object is basically a wrapper of a service response and includes a service unique identifier).

If no correct response was found, history is searched in an analogous way, but in this case, an identifier that is contained in the list of all obtained answers will be searched (opposing to considering only the correct answers).

Besides specifying one preferred criteria, the programmer may specify that impasses are solved using first the response correctness metric, then the availability metric, and finally the response time metric. This way, if an impasse occurs, the response selected is the one provided by the set of web services that present the highest correctness ratio. If the impasse still holds, then the response selected is the one provided by the set of web services that present the highest correctness and availability. And so on.

Note that this represents a best effort strategy whose effectiveness is dependent on the metric used. Obviously it should be applied only when the services being used provide high correctness (which can be observed during offline evaluation).
3.6 Using FTWS to develop fault tolerant WS

As mentioned before, one of the goals of FTWS is to reduce the developer programming effort to simple configuration tasks. In this section we briefly describe how web services programmers can use the FTWS mechanism.

The basic system requirements to build a FTWS project (using the current V1 version) are Java 6 SE [16] and Maven 2 [17]. The following steps must be followed to create an FTWS solution.

1. Create a Java project in any Integrated Development Environment (IDE).

2. Generate the component web services artifacts, including the local proxy classes for the remote services. The recommended tool for this task is wsimport. This is a tool that is included in the latest Java Development Kits and is regularly used for client code generation from each service’s WSDL. Keep in mind that this is a task that is always executed by web service developers, in normal circumstances,
This section presents a set of experiences that was conducted to demonstrate FTWS and analyze two important aspects: the average impact on web services performance and the fault tolerance gain in a typical usage scenario.

For the performance impact analysis, a 30 minutes load test was executed using a typical composition (not using FTWS) based on a sequential invocation of a set of web services specified by the TPC-App performance benchmark [18]. This composition was created with the help of soapUI, a web services test tool [19]. There was no use of specific business process engines; instead, the simple testing facilities provided by soapUI were sufficient to create the desired testing environment. After this first test, a second test was done, this time with our application acting as a proxy for the TPC-App services.

The average impact on the overall performance for the FTWS application under high stress conditions was of 1090.63 ms with a standard deviation of 117.50 ms. Although this is not a low impact, it is important to note these environments are characterized by large execution times which mitigate the relative impact in the performance. Nevertheless, as it is simply a technological aspect, an effort is being undertaken to minimize this impact in the future, via some code refactoring and optimizations.

The fault tolerance test scenario included the creation of three compositions using FTWS. Seven component web services were used: three to obtain a country list service, two to validate the format of e-mail addresses, and two to obtain Zip code geographic details, all referenced in [20]. For experimental purposes a service was considered unavailable whenever it took more than 60 seconds to respond to a client request. This is directly related to the availability results presented.

Table 2 presents the initial evaluation (offline) of the web services in terms of the core metrics considered in this work. Afterwards, both FTWS operation modes were tested. In this way 4 sets of tests were executed, one for the simultaneous invocation operation mode (referred as the 'All' criteria in Table 3) and the remaining three for the sequential invocation operation mode. A different criteria was chosen among response time, availability and correctness for each of these three tests. Results for the correctness metric are not presented as there were no wrong answers returned by the web services during the test phase. However, this metric may prove its usefulness in services not included in the tested set. Additionally, these particular results do not weaken the utility of the simultaneous invocation mode. This mode continues to be useful even if all services are correct. For instance, when using internet based services, it is frequent to have large fluctuations in response time or availability. In these cases, if the user is interested in the fastest services, this will be the method that assures that the client always gets the fastest answer (comparing to sequential invocation and assuming that there is a large variation in historic response time values). Similarly, if the user is more interested in a highly available service, he will have an answer without having to potentially wait for a sequential invocation of N services (this is useful when there is no cost associated with invoking the target services).

The following results represent the result of preliminary tests, these had a duration of 2 hours and the web services were invoked with a periodicity of 60 seconds (with a uniform variation of 30 seconds). Each test was executed three times in order to increase the results representativeness.

As we can see from the analysis of tables 2 and 3, choosing the FTWS simultaneous invocation operation mode increases the average response time as the application is always dependent on the worst performer. Note that, regardless of the high response times, the availability also increases to 100% even in presence of detected remote service failures, highlighting in this way the usefulness of the FTWS mechanism. The high standard deviation values observed are related to the fact that, although the majority of the requests had a response time that was close to the average, occasionally a few requests took much longer that the average (a typical behavior when using services over the internet), largely increasing the standard deviation values. Additionally, as mentioned before, these results were extracted from preliminary testing that does not have a high enough duration to be statistically tested. However, they completely serve the purpose of demonstrating the utility and performance of FTWS in more qualitative terms.
Table 1 Main parameters for the FTWS configuration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delegate</td>
<td>The name of the package where the proxy services classes are included</td>
</tr>
<tr>
<td>comparator</td>
<td>Specifies the metrics that should be used to solve voting impasses (i.e., response time, availability, correctness, or other function specified by the programmer). To use a function specified by the programmer the relative weights of each base metric should be defined (see Section 3.2)</td>
</tr>
<tr>
<td>invokeAll</td>
<td>Specifies the operation mode. If true, the alternative component web services are invoked simultaneously. If false, then sequential invocation is used</td>
</tr>
<tr>
<td>archiveOnDemand</td>
<td>Boolean property that specifies if historical data should be stored in persistent memory. For higher performance it should be defined as false. For higher reliability it should be defined as true (data is changed when scores change)</td>
</tr>
<tr>
<td>historyWindowSize</td>
<td>Number of items from the recent historical data to be used in the metrics calculation. Zero means that all available data should be used</td>
</tr>
</tbody>
</table>

Table 2 Baseline performance results for internet services.

<table>
<thead>
<tr>
<th>Web Service</th>
<th>Response Time (ms)</th>
<th>Standard Deviation</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country (1)</td>
<td>1509.71</td>
<td>1901.77</td>
<td>95.94%</td>
</tr>
<tr>
<td>Country (2)</td>
<td>1303.60</td>
<td>1461.76</td>
<td>97.48%</td>
</tr>
<tr>
<td>Country (3)</td>
<td>826.62</td>
<td>994.66</td>
<td>93.90%</td>
</tr>
<tr>
<td>E-Mail (1)</td>
<td>2483.42</td>
<td>46.77</td>
<td>98.20%</td>
</tr>
<tr>
<td>E-Mail (2)</td>
<td>2033.23</td>
<td>3387.68</td>
<td>92.37%</td>
</tr>
<tr>
<td>Zip (1)</td>
<td>858.63</td>
<td>27.48</td>
<td>99.30%</td>
</tr>
<tr>
<td>Zip (2)</td>
<td>402.24</td>
<td>754.17</td>
<td>99.66%</td>
</tr>
</tbody>
</table>

Table 3 FTWS services results.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Web Service</th>
<th>Response Time (ms)</th>
<th>Standard deviation</th>
<th>Detected failures</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Country</td>
<td>3762.27</td>
<td>4668.97</td>
<td>2.21%</td>
<td>100.00%</td>
</tr>
<tr>
<td></td>
<td>E-mail</td>
<td>3211.32</td>
<td>4900.63</td>
<td>4.09%</td>
<td>100.00%</td>
</tr>
<tr>
<td></td>
<td>Zip</td>
<td>1295.06</td>
<td>1303.02</td>
<td>0.70%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Response Time</td>
<td>Country</td>
<td>1929.27</td>
<td>2950.43</td>
<td>1.60%</td>
<td>100.00%</td>
</tr>
<tr>
<td></td>
<td>E-mail</td>
<td>1879.57</td>
<td>1644.48</td>
<td>0.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td></td>
<td>Zip</td>
<td>1325.28</td>
<td>4371.35</td>
<td>0.53%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Availability</td>
<td>Country</td>
<td>1852.88</td>
<td>2240.38</td>
<td>0.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td></td>
<td>E-mail</td>
<td>2154.21</td>
<td>5076.96</td>
<td>1.42%</td>
<td>100.00%</td>
</tr>
<tr>
<td></td>
<td>Zip</td>
<td>1039.57</td>
<td>3564.50</td>
<td>0.35%</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

Considering the sequential invocation mode (shown in the rows Response Time and Availability criteria in Table 3), we can see that the response times have decreased (compared to the simultaneous invocation) for all existing criteria (we empirically noticed a relation between the fastest and most available services). For this operation mode we can also see that the percentage of failures is lower compared to the simultaneous invocation method. This may indicate that a correct metric has been used to rank the services.

5. CONCLUSION

This paper presented a simple mechanism that allows programmers to easily develop fault tolerant compositions using diverse web services. The mechanism provides a set of artifacts that simplify the coding process. It deals with all the aspects related to the redundant web services invocation and responses voting and is able to perform a continuous evaluation of the services.

The FTWS mechanism uses a set of well known techniques (redundancy, voting, etc) to solve a set of problems typically faced by programmers that need to implement high dependable compositions.

The FTWS solution consists of a lightweight application that can be used without a standard application server. This is an excellent solution if low memory footprint is required, or scarce computational resources are available, however for other cases, the application can also be deployed in a full application server in a traditional way.

As future work we need to address the consistency aspect in stateful web services. The later is a major issue in web services whose execution changes the internal state. In fact, if one of the alternative web services does not provide a result, the state of the failed web service is no longer consistent with the state of services that completed the execution successfully. This way,
we need to study new approaches that guarantee the recovery of the component web services after failures.

REFERENCES