Testing Web Services for Robustness: A Tool Demo

Nuno Laranjeiro, Marco Vieira

Testing Web Services for Robustness: A Tool Demo

Nuno Laranjeiro, Marco Vieira
CISUC, Department of Informatics Engineering
University of Coimbra – Portugal
cnl@dei.uc.pt, mvieira@dei.uc.pt

Abstract

Web services represent a powerful interface for back-end systems that must provide a robust interface to client applications, even in the presence of invalid inputs. However, developing robust services is a difficult task. In this paper we demonstrate wsrbench, an online tool that facilitates web services robustness testing. Additionally, we present two scenarios to motivate robustness testing and to demonstrate the power of robustness testing in web services environments.

1. Introduction

Web services are supported by an infrastructure that typically includes an application server, the operating system and a set of external systems (e.g., databases, payment gateways, etc). The Simple Object Access Protocol (SOAP) [1] is used for exchanging XML-based messages between the consumer and the provider over the network. A web service may include several operations (in practice, each operation is a method with one or several input parameters) and is described using WSDL (Web Services Definition Language) [1], which is a XML format used to generate server and client code, and for configuration.

Robustness testing is an effective technique to characterize the behavior of a system in presence of erroneous input conditions. It has been successfully used to assess the robustness of operating systems [3], [6] and recently has been extended to the evaluation of web services [4], [9]. Robustness tests stimulate the system under testing through its interfaces ("black-box" testing) by submitting erroneous input conditions that may trigger internal errors.

wsrbench [4], an online tool based on the work presented in [9], is a powerful instrument in three key scenarios: 1) help providers in evaluating and improving the robustness of their web services implementations before deployment; 2) help consumers to select the web services that best fit their requirements by testing different alternatives; and 3) help providers and consumers to identify the need for wrappers to perform the required validations before execution.

The goal of this paper is to demonstrate how *wsrbench* can be used in practice. The goal is not to present technical details (those are available in [4], [9]), but to demonstrate the tool and present scenarios on how to use it to develop robust web services. This way, we discuss two hypothetical scenarios and two real case studies on the usage of *wsrbench*. The hypothetical scenarios serve as motivation to apply robustness testing in web services environments and the real case studies show that robustness testing can be applied in practice to test and compare real web services.

The structure of the paper is as follows. Next section presents motivation scenarios. Section 3 summarizes the robustness testing approach and introduces *wsrbench*. Section 4 presents two case studies. Section 5 briefly describes the demo to be performed during the workshop and concludes the paper.

2. Motivation scenarios

In this section we present two hypothetical scenarios of the usage of robustness testing.

2.1. Scenario #1: Selecting a web service

SurveyMe is a medium size company that performs customer satisfaction surveys for large companies. As surveys are typically conducted by phone, it is of utmost importance to guarantee the correctness of the phone numbers registered in the database. A frequent situation is to find phone numbers in the database with incorrect country codes, area codes, or prefixes, due to mistakes during data collection or storage.

Mr. Simms, the head of the division in charge of analyzing the results of the surveys, is concerned with the number of surveys that are not conducted, or are delayed, due to incorrect phone numbers.

Mr. Simms decided to meet with Mr. Pinto, the head of the division in charge of managing and improving the software used by the company, to discuss this situation and understand if the phone number verification rules used by the application could be improved. While surfing in the Internet for potential solutions, Mr. Pinto browsed through a web page that called his attention. WebX, the owner of that web page, announces a web service that can be used to ver-

ify phone numbers around the world. Amazingly, the use of this web service is free of charge. "This is very good and it can be easily integrated in our application" thought Mr. Pinto, while searching for other services providing similar functionality. After some minutes, Mr. Pinto found two other options, both free of charge.

Mr. Pinto decided use an external web service to perform phone numbers validation. However, in order to maintain the robustness of the application they needed to select a robust web service. The problem was then how to choose the web service to be used. Some time ago, Mr. Pinto read a research paper on a tool for web services robustness testing [4]. He recalled that the tool (*wsrbench*) was online and free to use, and decided to try the tool to test the web services.

The results were impressive. With a simple configuration procedure the tool was able to automatically test the services and detect potential robustness problems. In fact, Mr. Pinto only provided the tool with the URL of each service and configured the expected domain for each input parameter. Table 1 summarizes the results.

From the robustness point-of-view, TelefX should be the selected service, as it presents no robustness problems. However, this was the first step in the selection process as Mr. Pinto decided then to test the services for performance. In any case, the information gathered about these services was of utmost importance. In fact, even if one of the services with robustness issues is selected due to performance reasons the development team knows its robustness problems in detail. This information can be used to implement wrappers that perform parameter validation before web service invocation.

2.2. Scenario #2: Developing a web service

Electrics.com is a large company specialized in producing electrical components that recently decided to develop a new database application to manage daily operations. A service-oriented approach in which a set of web services provides functionalities that can be invoked by client applications in a transparent manner.

The plan was to follow an interactive lifecycle during the project. The web services would be developed first and the different user interfaces implemented afterwards. The web services would be tested immediately after their development phase.

During the development of the interface to register new customers, Sanders, a junior programmer, was struggling with a problem. When inputting some values in this web page, the application returns an unexpected exception. After several hours analyzing his code Sanders was not able to find any problem and decided to perform some tests in the web services using *wsrbench*, an online tool he had recently found while surfing in the web [4].

Table 1 – Robustness-testing results for Scenario #1.

Service Name	Operation	Parameter	# Abort Failures	# Hindering Failures
PhoneNumbers	ValidatePhone	PhoneNumber	3	0
PhonesWS	CheckPhone	Phone	0	1
TelefX	CheckNumber	PhoneNumber	0	0

After reading some documentation about the tool, Sanders started the tests by configuring some parameters. The result obtained was not a surprise for Sanders: the *newCustomer* web service had a problem. When a long *contactEmail* was provided the web service returned a *StackOverflowException* exception. This way, Sanders decided to analyze the web service code in order to identify the source of the problem. However, after some hours, he was not able to identify any problem, not even with help of some other developers.

Sanders then decided to ask for the help of a senior developer – John, not involved in the project, but an expert in web services development. After some minutes analyzing the code, John identified the source of the problem: although the code targeting the validation of the *contactEmail* parameter was in place, too large email addresses caused the web service to throw a *StackOverflowException*. After some analysis of the code they concluded that the problem resided in the external API that was being used to validate email addresses (Jakarta Commons Validator 1.3.0). This shows that robustness problems may occur even when programmers pay great attention to the code correctness. In fact, the use of third party software can raise problems that are not obvious for programmers.

3. *wsrbench*: A web services robustness testing tool

Web services robustness testing is based on erroneous call parameters [9]. The robustness tests consist of exceptional and acceptable input values of parameters of web services operations that can be generated by applying a set of predefined rules according to the data types of each parameter.

The robustness benchmark includes the following key components: **workload** (represents the work that the service must perform during the benchmark run); **robustness tests** (set of invalid call parameters that is applied to expose robustness problems); and **failure modes classification** (characterize the behavior of the web service while executing the workload in the presence of the robustness tests). The testing procedure is based on the following generic set of steps:

1. Tests preparation
   1.1 Analysis of the WSDL of the web service under testing in order to gather information about the

relevant operations, parameters, and data types.

1.2 Workload generation.

2. Tests execution

2.1 Execution of the workload generated in step 1.2. The goal is to understand the expected correct behavior for the service.

2.2 Execution of the robustness tests in order to trigger faulty behaviors, and in that way disclose robustness problems.

3. Service characterization, including failure modes identification (using the data collected in step 2).

The robustness of the web services is classified according to an adapted version of the CRASH scale [3] (the wsCRASH scale) that distinguishes the following failure modes: **Catastrophic** (the application server used to run the web service under testing becomes corrupted or the machine crashes or reboots), **Restart** (the web service execution hangs and must be terminated by force), **Abort** (abnormal termination of the web service execution), **Silent** (no error is indicated by the application server), and **Hindering** (the error code returned is not correct or the response is delayed).

wsrbench, publicly available at http://wsrbench.dei.uc.pt, implements the web services testing approach proposed in [9] and provides a web based interface that allows users to perform configurations and visualize the results of tests. Note that, anyone can use *wsrbench* as it is free and very easy to use. Only a very simple registration and posterior authentication process is required. In the following paragraphs we introduce the key functionalities of *wsrbench* (technical details can be found in [4]).

After registration and authentication three key options are available for users: Configuration; Add WSDL; and My Tests. The **Configuration** option allows several configuration aspects to be defined, such as the user's email, number of finished tests to show on screen, etc. The **Add WSDL** option allows users to add the WSDL file describing a web service to be tested for robustness. After submitting the WSDL file the user can visualize the set of operations and parameters provided by the service.

Operations not conformant with the WS-I Basic Profile [2] will be grayed out and not tested. This standard is an industry effort to clarify the ambiguous parts of the WSDL specification and is accepted by the main service providers, including Microsoft's .NET framework version 3 [5] and Sun Microsystem's Java 6 Web Services stack (JAX-WS) [7].

For each testable operation the user may define the valid values for each parameter (i.e., the domain of the parameter). When these are not defined, the tool considers that the parameter domain is the domain of the corresponding data type. After defining the domain of the parameters the tests can start (by clicking the Start Test button). The user will be informed by email when the tests conclude.

The **My Tests** option allows the user to visualize the tests previously performed along with information on currently ongoing tests. For each operation of a web service, the results for the individual faults applied to each parameter are shown. Clicking the 'XML' link opens a popup where more details are provided. These include the list of requests sent and the service responses received.

The user can mark the service interaction as a robustness problem, a correct interaction (no problem detected), or simply leave it unmarked. It is important to emphasize that, after testing a given web service, the tool performs an automatic analysis of the responses obtained in order to distinguish regular replies from replies that reveal robustness problems in the service being tested. However, in some cases the tool is not able to decide if a given response is due to a robustness problem or not. That is why the tool also allows users to perform this analysis manually.

4. *wsrbench* in practice

In this section we present two case studies on using *wsrbench* to test and compare real web services.

4.1. Case Study #1: Testing public web services

More than 100 publicly available web services were tested for robustness (most of these services are listed at *http://www.xmethods.net/*, a web site that references publicly available web services). The full list of web services tested can be found at [4]). About 35% of the web services tested presented robustness problems.

Some silent failures were observed for two web services: *Web-Service Documentation* (owned by westwind.com) and *Code 39 Bar Code* (owned by webservicex.net). For the former, robustness tests consisting in the replacement of any parameter by a null value always leads to an absence of response from the server. For the latter, an overflowed string in the Title parameter led the server to report a null reference exception. However, web service requests submitted immediately after that abort failure remained unanswered.

A total of 61 Abort failures (not counting similar errors triggered by different faults for the same parameter in a given operation) were detected. 30% were marked as null references, 30% as SQL problems, 13% conversion problems, 7% as arithmetic problems, and 21% as others.

4.2. Case Study #2: Comparing web services

Two implementations of a subset of the TPC-App web services [8] were submitted to robustness testing (each web service implemented a single operation).

As shown in Table 2, robustness problems related to abort failures were observed for both solutions. An interesting robustness problem was observed for the *newCustomer* service in implementation A. In fact, although the code targeting the validation of the *contactEmail* parameter was in place, too large email addresses caused the web service to throw a *StackOverflowException*. After analyzing the source code we concluded that the problem resided in the external API that was being used to validate email addresses (Jakarta Commons Validator 1.3.0). This shows that robustness problems may occur even when programmers pay great attention to the code correctness. Furthermore, this type of errors can easily appear or disappear when an apparently harmless update is done to the external libraries commonly required by projects. However they can be easily detected with the help of robustness testing, which once more highlights the importance of robustness testing.

The analysis of the results in Table 2 suggests that, from a consumer standpoint, the best option would be to choose the *newProducts* and *productDetail* services from implementation A, and the *changePaymentMethod* and *newCostumer* services from implementation B. From the provider point-of-view, it is clear that some software improvements are needed in order to resolve the robustness problems detected.

5. Conclusion

The goal of this paper is to demonstrate a tool for web services robustness testing. This tool builds on solid scientific concepts proposed in previous works, and seals the space between research and industry practice. In fact, *wsrbench* fills a gap in current development tools, providing an easy interface for robustness testing of web services. The tool is available online requiring no installation and little configuration effort. Its effectiveness will be demonstrated during the workshop and has already been proved by testing a large number of public and custom made web services. The results show that many services are deployed with robustness problems that, in some cases, may also represent security issues. For example, some of the problems uncovered show that some services may be vulnerabl to SQL injection attacks, which highlights the importance of testing services for robustness. This is something that could be easily avoided if these services were tested for robustness by *wsrbench*.

Table 2 – Abort failures for TPC-App services.

Web Service	Impl. A	Impl. B
changePayment method	3	3
newCustomer	15	6
newProducts	0	2
productDetail	0	1

The *wsrbench* demo during the workshop will be organized as follows. First we will discuss basic concepts on robustness testing and present the two hypothetical scenarios for the use of robustness testing in web services environments. Then we will introduce the key functionalities of the tool. Finally, we will show the power of the tool in practice by applying it to several web services publicly available in the Internet and to some of the web services from the TPC-App performance benchmark. In the latter case, we will demonstrate how the robustness problems disclosed can be fixed by adding appropriate input validators.

It is important to emphasize that we are planning a hands-on demo. In fact, the tool will be effectively demonstrated and explained during the talk. Participants will have the possibility to try the tool during or after the workshop. Our goal is to motivate researchers and practitioners working on web services to try it and to foster research on this important topic.

6. References

[1] Curbera, F., et al., “Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI,” Internet Computing, IEEE, vol. 6, 2002, pp. 86-93.

[2] Ferris, C., et al., “Basic Profile - Version 1.2,” 2007; http:// www.ws-i.org/Profiles/BasicProfile1_2(WGAD).html.

[3] Koopman, P., DeVale, J., "Comparing the robustness of POSIX operating systems," Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999.

[4] Laranjeiro, N., Canelas, S., Vieira, M. , "wsrbench: An On-Line Tool for Robustness Benchmarking", IEEE International Conference on Services Computing (SCC 2008), Honolulu, Hawaii, USA, July 2008.

[5] Microsoft, “.NET Framework Developer Center,” 2008; http://msdn2.microsoft.com/en-gb/netframework/default.aspx

[6] Rodríguez, M., et al., "MAFALDA: Microkernel Assessment by fault injection and design aid", 3rd European Dependable Computing Conference, EDCC-3, 1999.

[7] Sun Microsystems, Inc., “JAX-WS Reference Implementation”; https://jax-ws.dev.java.net/.

[8] Transaction Processing Performance Council, “TPC BenchmarkTM App (Application Server) Standard Specification, Version 1.1”, 2005, available at: http://www.tpc.org/tpc_app/.

[9] Vieira, M., Laranjeiro, N., Madeira, H., “Benchmarking the Robustness of Web Services,” 13th IEEE Pacific Rim Dependable Computing Conference (PRDC 2007), Melbourne, Australia, 2007.