Newcastle University e-prints

Date deposited: 1st November 2010

Version of file: Author final

Peer Review Status: Unknown

Citation for published item:


Further information on publisher website:

http://www.ieee.org/portal/site

Publisher’s copyright statement:

Copyright notice: © 2010 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Use Policy:

The full-text may be used and/or reproduced and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not for profit purposes provided that:

- A full bibliographic reference is made to the original source
- A link is made to the metadata record in Newcastle E-prints
- The full text is not changed in any way.

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.
A Tool for Testing Fault Tolerance of Web Service Systems

Khaled Farj
School of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, UK.
k.a.s.farj@ncl.ac.uk

Abstract

Testing the Fault Tolerance Mechanisms (FTMs) is crucial for the development of today’s Web Service applications. In this work, we propose a methodology for assessing the performance and the efficacy of FTMs applied to Web services applications. We present a tool that uses application level fault injection techniques to inject communication faults by using a network Emulator Service without any modification to the system. The tool also generates additional workload on the tested system in order to produce more realistic results. The tool offers full control over the emulated environments in addition to the ability to inject application specific faults.

1. Introduction and Related work

Web services are used to build composed systems. Web Services can be adopted to develop information systems through integrating services to obtain complex composed system. These services are usually developed and administrated by different service providers and distributed over the Internet in different locations. In reality, there is no guarantee that all services are highly reliable. Distributed services over the Internet means they are subjected to many network faults, unlike traditional distributed systems such as CORBA typically runs over a LAN and it is assumed the effect of this class of faults is negligible. Researchers found that the unstable Internet environments and faults in server connections can lead to unreliability of Web services [1].

In a composed service, Web service interfaces are usually known at composition time, in turn the reliability of a composed service may depend on its participated services ability to respond to unforeseen unexpected situations and faults [2]. In addition, the performance of a composed service is very difficult to measure at design time as composed service usually runs over the Internet which involves a high cost of using it for the sake of testing. A runtime environment such as a network emulator, which our tool provides, is needed to estimate the contribution of each contributed service to the system.

Testing the performance and FTMs of web services have become an active research area by using Software Fault Injection. Software Fault Injection is a well-proven method of assessing the reliability of a system [3]. Although much work has been done in testing single web services, there appears still more research to be carried out on composed services.

There are many Software Fault Injection testing tools for testing software systems and their reliabilities. In [4] a tool is developed for generating and validating test cases. Tools start from the WSDL schema types and introduce some operator to generate a request with random data and a test script that manipulates the request parameters. In [5] a technique of using mutation analysis is proposed. A mutant WSDL document is generated by applying mutant operators to the original WSDL document. A test tool called WSDLTest [6] generates Web service requests from the WSDL schemas and tunes them in accordance with the pre-conditions written by the tester and verifies the response against the post-conditions offline. In [7] a tool is proposed based on some rules defined in XML schema or DTD. The tool modifies parameter values of requests by using boundary value testing, and on interaction perturbation, using mutation analysis. Another tool in [8] introduces a framework intercepting and perturbing SOAP messages by injecting faults by corrupting the encoding schema address, dropping messages, and inserting random text in the SOAP Body. The work described in [9] helps service requesters create test cases to select suitable and correct Web services from public registries. It proposes a method where faults are injected into SOAP
messages to test boundaries of the parameters, as specified in the WSDL document. WS-FTT tools [10] inject faults by modifying SOAP messages using scripts. The function parameters are modified by using the value boundaries specified by the tester.

A common characteristic of previous work is that their focus is almost on testing single services in isolation; furthermore, most of their focus is on injecting faults by modifying the SOAP message, since they do not emulate additional workload in the system which could give rise to different results. Moreover, most of the previous work focus is on testing the service provider not on the service requester. In a composed service where the service provider needs to be a service requester to other service provider in order to serve a request, which means it is so essential to test the service requester to prevent the whole system from failing to provide the required service.

This paper sets out an approach method for testing the performance and FTMs of either a single Web service or a composed service and of either a service requester or a service provider, without modifying, recompiling or patching the target system. The proposed method also generates a background workload in the system. Preliminary results have been obtained and both the overhead and the functionality of the tool have been tested and evaluated.

2. Network Fault Injector Service

Network Fault Injector Service (NetFIS) is an intermediary composed service which implements our fault injection method for testing Web service performance and FTMs. It lies between the Service Client and the Service Provider and requires the target system be distributed in a modular fashion of services interacting via messages, so messages can be manipulated to emulate incorrect behavior of faulty services. It basically intercepts the request from the Service Client, provides a network emulator service and injecting appropriate fault (if any) and then forwards the request to its internal subcomponent which is the Fault Injection Controller (FIC) to inject faults and then it is sent to another FII deployed where the actual Web service is running. When the FII at the client side receives a response from the other FII (at service side) it forwards it to the Service Client.

NetFIS gives Web service applications the sense of running over a WAN without any modification to the application. It requires no modifications to the underlying operating system, networking libraries or the Web service applications under test. The tool is for injecting network faults and software faults at application level. Network faults such as dropping, delaying, randomly corrupting bytes of the exchanged SOAP messages. An addition, software faults are also injected into individual RPC parameters based on obtaining the relevant web services parameters definitions (including data types) from WSDL files (due to limits in space more details about this class of faults will be available at a later date).

As the tool offers injecting many faults, it gives the user the flexibility to inject appropriate faults as required and also gives the applications a configurable emulated network which gives the system the sense it is running over a network without any modification.

2.1 Tool Architecture and Roles

Figure 1 shows our tool emulating a simple two-node network (A and B). It consists of three main components as follows:

- **Fault Injection Interceptor (FII):** FII is a Web service which has the capability to be a proxy Web service to one or more Web services. The FII role depends on where it is deployed. At the client side, its role is to generate a proxy WSDL from the actual Web Service WSDL to be called by a client. As a result all client requests are processed by the FII. Thereafter, the FII sends the request to its internal subcomponent which is the Fault Injection Controller (FIC) to inject faults and then it is sent to another FII deployed where the actual Web service is running. When the FII at the client side receives a response from the other FII (at service side) it forwards it to the client.

At the Web service side, the FII Web Service role is different. Request messages received from other FII (deployed at client side) are forwarded to the Web service then when the response is received it is redirected to the internal FIC for fault injection, and then the response (if any) is sent back to the other FII.
deployed at the client site. In the case of composed services, where the service has to act as both a service and a client, a single FII can perform both of the roles explained above. Using this way of intercepting messages, no modification is made to the system.

- Fault Injection Controller (FIC): FIC is a java application inside the FII which is regarded as the main core of the tool. Its responsibility is to control the tool and to inject the proper faults into the messages. Faults are injected into messages based upon decisions coming from two other components of the tool, the Network Emulation Service (NES) and the Script Fault Model (SFM). These components can either be turned on or off at the choice of the user. The SFM is a java fault model script written by the user (in this stage). When both SFM and NES are active, the SFM decision can only be applied if the decision from NES is not to drop or corrupt the message. The FIC gives network faults higher priority, however it can be controlled. The FIC also logs SOAP messages to be analyzed offline to assess the tested system. The message, if it has not been dropped, is sent back to the interceptor to complete its journey to the corresponding FII.

- Network Emulator Service (NES): NES is an extended; modified version of a WAN Emulator for CORBA Applications [11]. NES gives the applications the sense of running over a LAN or WAN, and also generating other workload traffic running at the same time and sharing the networking resources. In addition, it is used by FIC to help in injecting network faults (loss, delay, corruption, reordering, etc.). NES only uses Web service technology to emulate networks.

The system is deployed and exposed as composed service. Any emulated network can be controlled by only one centralized Network Controller Service (NCS), controlling the emulated network, and a set of FIIs and NES’s in pairs that each FII and NES pair represent and emulate only one node in the target network.

3. Example experiment

A simple experiment was conducted by injecting some network-related faults for evaluating the overhead and the functionality of the tool.

The target network topology is very simple two-node network setup, as Figure 1 shows, in order to make it easier to monitor its behavior. Various types of faults examined, however the synthetic traffic of the NES’s were disabled. There were no sources of traffic other than the normal traffic in the Newcastle University LAN so as to simplify evaluating the tool.

Firstly the response time delay of system under test was measured without using the tool. Secondly the performance and the functionality of the tool were measured in two emulated network configurations; a fast WAN and a slow WAN. The fast WAN configuration was as follows: the propagation delays are fixed to 2ms which is typical of inter-city links within the UK, and the bandwidth of each link is 4mb/s. The slow WAN parameters were as follows: the propagation delays are fixed to 20ms which is typical of far apart locations and links that span the oceans (e.g. between London, UK and Tripoli, Libya); and bandwidth of each link is 1000Kb/s. The system being tested was a Web service client-server pair. The server simply offers a service that returns an array of characters of fixed length (100 bytes). The server is running on one node in the network while the client is running on the other node.

In order to emulate our simple two-node network, 2 NES’s services and 2 FIIs services were required (Each node needs 1 FII for injecting faults - if any- and 1 NES for emulating the network of the node) and also 1 NCS for controlling the WAN. All services of the tool and the client run on 6 machines from a cluster of 24 Linux machines (Fedora Core 5) running on the JBoss Application Server [12] connected by a 100Mb Ethernet LAN. The Web service to be tested runs on a different machine in the same Newcastle University LAN. The system application pair was tested for a sample runs of 1000, 5000 and 10000 requests where drop, errors (injected to the system) and the response time were recorded.

The test has been done in two different setup configurations. The first setup was for testing the delay overhead introduced by the tool. The second setup was for testing the functionality of injecting network faults such as dropping, randomly corrupting the messages exchanged between the client and server.

Table 1 shows the response time delay in milliseconds collected by running the system without using the tool.

<table>
<thead>
<tr>
<th>Network Emulated</th>
<th>Injected Delay, ms</th>
<th>Num of requests</th>
<th>Response time</th>
<th>Avg, ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max, ms</td>
<td>Min, ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Without tool</td>
<td>None</td>
<td>1000</td>
<td>44.87</td>
<td>7.15</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>56.93</td>
<td>6.67</td>
<td>7.465</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>112.93</td>
<td>6.13</td>
<td>7.095</td>
</tr>
<tr>
<td>Fast WAN</td>
<td>2</td>
<td>1000</td>
<td>281.80</td>
<td>29.23</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>559.77</td>
<td>31.42</td>
<td>41.30</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>281.80</td>
<td>29.23</td>
<td>38.56</td>
</tr>
<tr>
<td>Slow WAN</td>
<td>20</td>
<td>1000</td>
<td>198.89</td>
<td>80.45</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>199.31</td>
<td>79.27</td>
<td>88.96</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>217.96</td>
<td>78.59</td>
<td>87.79</td>
</tr>
</tbody>
</table>

The response time delays were also measured by running the same experiment under the control of the
tool with two different network configurations (fast and slow) without generating any additional traffic. The averages show that the overhead of running the tool ranges between 36 and 45 milliseconds (when subtracting the injected delay) when emulating the fast WAN, as well as it ranges between 85 and 89 milliseconds (when subtracting the injected delay) when emulating the slow WAN. This is a negligible overhead that conforms to the design assumption that the target WAN to be emulated exhibit much greater delays than this small overhead.

In Table 2 the tool accurately injects the target network drop and error faults by achieving the exact (or almost) rates targeted.

<table>
<thead>
<tr>
<th>Num of requests</th>
<th>Emulated Network</th>
<th>Injected drop</th>
<th>Achieved, total messages</th>
<th>Injected error</th>
<th>Achieved, total messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>Fast</td>
<td>0.001</td>
<td>5</td>
<td>0.001</td>
<td>5</td>
</tr>
<tr>
<td>10000</td>
<td>WAN</td>
<td>0.001</td>
<td>9</td>
<td>0.001</td>
<td>10</td>
</tr>
<tr>
<td>5000</td>
<td>Slow</td>
<td>0.001</td>
<td>5</td>
<td>0.001</td>
<td>5</td>
</tr>
<tr>
<td>10000</td>
<td>WAN</td>
<td>0.001</td>
<td>11</td>
<td>0.001</td>
<td>12</td>
</tr>
</tbody>
</table>

For example, we tried to emulate a fast and a slow network links with a drop rate of 1 message per 1000 messages which were achieved for 5000 and 10000 requests. More addition the target injected random error into the messages were also been achieved in both network cases (fast and slow).

It is important to point out that the underlying layers (middleware and networking stack layers) will be most likely capable to mask and recover from errors and drops of the messages. However, this method is for propagating such faults to the application so as to enable the application developers to test the performance of their applications under such conditions.

4. Conclusion and Future work

We have introduced a methodology and built a tool that can inject faults into any Web service application without touching the code of the application. There is great flexibility in the number and type of fault that can be injected and, furthermore, we can control the network that is used and can add background traffic. The proposed system will be used to detect problems in existing systems based on a logging mechanism, for measuring the performance and the efficacy of Fault Tolerant Mechanisms applied to Web service systems.

We are currently testing the tool with some Web service applications which uses FTMs called WS-Mediator [13] so as to check that the enhancement in reliability of the systems make its deployment worthwhile. As well as our fault injection method will be improved in a way that it allows the user to generate a fault model script and the ability to inject application specific faults by analyzing WSDL documents which will enhance the flexibility of the tool such as manipulating function parameters and return values. A prototype version of the tool may become available for academic use in the near future.

5. References