A NOVEL METHOD TO TEST DEPENDABLE COMPOSED SERVICE COMPONENTS

Khaled Farj[1] and Adel Smeda[2]

[1]Faculty of Science, University of Al-Jabel Al-Gharbi, P.O. Box 64200, Gharian, Libya
[2]Faculty of Accounting, University of Al-Jabel Al-Gharbi, Gharian, Libya

ABSTRACT

Assessing Web service systems performance and their dependability are crucial for the development of today's applications. Testing the performance and Fault Tolerance Mechanisms (FTMs) of composed service components is hard to be measured at design time due to service instability is often caused by the nature of the network conditions. Using a real internet environment for testing systems is difficult to set up and control. We have introduced a fault injection toolkit that emulates a WAN within a LAN environment between composed service components and offers full control over the emulated environment in addition to the capability to inject network-related faults and application specific faults. The toolkit also generates background workloads on the system under test so as to produce more realistic results. We describe an experiment that has been performed to examine the impact of fault tolerance protocols deployed at a service client by using our toolkit system.

KEYWORDS

Web Services, Fault Tolerance Methodologies, and Software Fault Injection & Composed Web service.

1. INTRODUCTION

Web services are becoming progressively more essential in the information systems. Web services are software applications that operate independently and that offer services over the Internet to other software applications, including web applications and other Web services. Web services have changed the way we look at the Internet from being a repository of data into a repository of Services [1], because of its capability to solve integration problems in Internet applications. By using Web services, Internet applications can communicate with other Internet applications regardless of using different programming language or platforms.

Web services can be adopted to develop information systems by integration of services to obtain complex composed services. Web service technology is being used to allow the creation of complex systems, composed of simple Web services, which exchange messages to form complex conversation schemas [2]. These services are usually developed and administrated by different service providers, running on different platforms and also distributed over the Internet in different locations.

The quality of such complex systems depends on the quality of the network environment and, of course, on the quality of the Web service applications participating in forming such systems. One of the obstacles of the adoption of the Web service paradigm in such composed systems is the problem of measuring their overall quality. Services are inherently distributed and heterogeneous, and are often invoked with little knowledge of their reliability and their performance.

In such applications, service composition is typically dynamic so that services are discovered, selected, and composed, probably at runtime. In such services it is hard to assess behaviour and performance in the presence of faults. As Web services are usually distributed and operated over the Internet, there is no guarantee that all the components of the service are highly reliable. In [15] it is reported that network faults such as message duplication, reordering, loss or corruption have an effect on traditional distributed systems such as CORBA applications. In addition, it has been concluded that unstable Internet environments and server connections can lead to unreliability of Web service systems [3].

As Web service systems are subject to many network faults such as delaying, dropping, damaging, and reordering messages and also to software faults within the service, testing the performance and fault tolerance of Web services have become an active research area. Software Fault injection is a well-proven method of testing and assessing the reliability of a system [1]. In this paper, we describe a toolkit for assessing the performance and FTMs of either a single Web service or composed services, without modifying the system being tested. No recompiling or patching is necessary. Furthermore, the toolkit will generate a background workload for more accurately emulate real networks. Our toolkit is also running independent of the hosting environment for portability.

In Section 2, other tools for injecting faults into web services are described. In Section 3 and 4, we explain the design and implementation of our Network Fault Injector service (NetFIS). In Section 5, an example experiment are described which uses NetFIS to test and assess the dependability of a Bioinformatics Web service system. This examined Web service system uses a technique termed Mediation [4] to provide fault tolerance mechanisms. We evaluate the performance and the fault tolerance mechanisms of the service. Section 6 describes our conclusions and our future work.

2. RELATED WORK

There are many Software Fault Injection tools for testing and assessing distributed systems in general and other tools for testing Web service systems in particular. Some well-known fault injection testing tools, such as DOCTOR [5] and Orchestra [6] that both of them support network level fault injection and could be potentially used to inject faults into Web service systems, both tools have been designed to test network protocols and therefore don't decode complete middleware message sequences. There are many other fault injection tools for testing Web service applications. In [7] a testing tool is developed for generating and validating test cases. Tools start from the WSDL schema types and introduce some operators to generate a request with random data and a test script which manipulates and modifies the request parameters. In [8] other technique for testing Web service systems using mutation analysis is proposed. A mutant WSDL document is generated by applying mutant operators to the elements of the original WSDL document. A test tool called WSDLTest [9] testing tool generates Web service requests from the WSDL schemas and tunes them in accordance with, what's called, the pre-conditions written by the user and verifies the response against the post-conditions offline.
In [10] an assessing tool is proposed based on some rules which defined in XML schema or DTD.

The testing tool modifies the parameter values in requests by using boundary value testing, and on interaction perturbation, using mutation analysis. Another testing tool [11] introduces a toolkit framework intercepting and perturbing exchanged SOAP messages by injecting faults by corrupting the encoding schema address, dropping messages, and inserting random text in the body of SOAP messages. The work described in [12] helps Web service requesters to create test cases scenarios in order to select suitable and correct Web service application from public repositories. It proposes a testing method tool which injects faults into SOAP messages to test the parameter boundaries, as specified in the WSDL document. WS-FIT testing tools [13] inject faults by modifying exchanged SOAP messages using scripts. The function parameters are manipulated by using the value boundaries specified by the tester. WSInject [14] testing tool injects both communication and interface faults, and the ability to inject faults for testing a single Web service or composed service system.

A common characteristic of previous testing method tools is that their focus is mostly on testing single Web services in isolation (except for a few such as WSInject); furthermore, most of their focus is on injecting faults by only modifying the exchanged SOAP messages, since they do not emulate additional workload in the system which most likely could give rise to different results. Different workloads could lead to different assessment and testing results, due to the cause of different system activation patterns [15]. Moreover, most of the previous testing methods focus is on testing only the Web service provider not on the Web service requester. In a composed service where the Web service provider needs to be a Web service requester to other Web service provider in order to serve a request, which means it is so essential to test the Web service requester to prevent the whole system from failing to provide the required service.

In [16] testing method, faults are injected at the IP level in order to investigate the effect of retransmission mechanisms of TCP on Web services. This allows them to assess the relationship between TCP and WSRM time-out and retransmission mechanisms. By contrast, our testing toolkit method will drop the whole exchanged message which may consist of more than one packet. In this way, the consequences of injecting such faults can be propagated to the application level so as to assess and examine whether FTMs at the application level can handle such faults. In [17] a methodology, for assessing and testing the performance of FTMs applied to Web services applications, have been introduced and the overhead of the tool has been tested.

In this paper, we propose a fault injection testing method that, adopting the architecture of a Wide Area Network emulator used for testing and assessing other distributed systems, extends it to test and assess composed Web service systems. In addition, two classes of faults are injected, communication faults and software-specific faults without modification to the system under test. The method also generates additional workload on the tested system in order to produce more realistic testing results.

3. NETWORK FAULT INJECTION METHOD

The Network Fault Injector Service (NetFIS) is a Web service system which implements our fault injection method for assessing Web services performance and fault tolerance mechanisms.

It is deployed between the Web service client and the Web service provider and requires the system under test to be distributed in a modular fashion of services interacting via messages, therefore, messages can be manipulated and modified to emulate incorrect behaviour of faulty services. It basically intercepts the request from the Web service client, provides a network emulator service and injects appropriate fault (if any) and thereafter, forwards the request to the Web service provider. Similarly, it intercepts the response from the Web service provider, provides a network emulator and injects a fault (if any) and then forwards it to the Web service client. NetFIS gives Web service applications the sense of running over a WAN without any modification to the applications. Our testing method requires no modifications to the underlying operating system, networking libraries or the Web service to be tested. NetFIS is able to emulate WAN behaviour and injecting network faults such as dropping, delaying, randomly corrupting bytes in the body of the SOAP messages, network faults have been used to carry out experiments reported hereafter. In addition, software faults can also be injected into individual RPC parameters based on obtaining the relevant Web service parameters definitions (including data types) from WSDL files, however this class of faults will not be injected or detailed in this paper.

The network emulator is configurable and has the ability to control every property of the target emulated network environment. The graphical user interface (GUI) of the network emulator also gives the ability to control the emulator at runtime so as to achieve the dynamic nature of networks. By using the network topology configuration file, network traffic trace files and the GUI, composed service systems performance and FTMs can be measured and assessed in a controlled emulation of a target WAN. As composed service systems can be subjected and affected to network delays, drops, errors, reorders and partitions, and also software faults, in this experiment we restrict the use of NetFIS to emulate only drops, errors and delays to the messages exchanged between the system components in order to measure the performance and the FTMs applied to Web service client side.

Before going through more details about the proposed system architecture of the NetFIS, it is worthwhile to elaborate on some of the major design issues.

3.1 Application Level And Network Level

Network related faults like corruptions, reordering and dropping packets occur at the network level. This includes the physical media and all the layers in the network stack. It follows logically that intentional network related faults are injected at the network level. This is done traditionally to assess the reliability and performance of networking protocol stacks. However there is a very high probability that the fault injected (e.g. corruption) will be detected by the actual underlying network protocol stack at the other end [15]. Consequently, the application or middleware being tested will not notice the occurrence of an error and the reliability measures built there will go untested. Moreover injected faults at network level are based on tampering with packets not on application messages.

Our fault injection method is for injecting communication faults and tests their impact on composed service system performance and their impact on the fault tolerance mechanisms applied to such systems. Therefore it is more efficient and desirable to inject faults at the application/middleware level instead of at the network level. Communications between the system services are intercepted at application level and faults are injected by using proxies. We elaborate on the architecture of this choice in later sections.

3.2 Network Emulation

Since, in composed service, the services participating in the system are usually running over the Internet, the performance and fault tolerance of a composed service are very difficult to be measured at design time. Testing such systems a distributed testing environment is required such as a wide area network (WAN) or the Internet. Therefore performance and fault tolerance of the system can be tested by deploying the system and run it over a WAN or the Internet. However using the Internet or WAN for the sake of testing is usually impractical. It involves a high cost in terms of time consuming and setting up a WAN or using the Internet for the sake of testing. It is impossible to control such dynamic environment as networks such as putting more stress, load, or errors. Moreover, errors and faults may take a long time to occur. Some errors may not occur without applying a certain chain of events.

A realistic approach is to run the system in one machine or over a LAN using a WAN emulation system which can provide the sense that the system is running over a WAN and provides all the properties of a dynamic WAN like the Internet. That is, will help the testers to test the performance and fault tolerance by running the system under different circumstances such as different network traffic load, delays, loss rate, and so on. By using network emulation, not only the performance of the whole system can be measured under different circumstances, but also the contribution of each service to the overall composed service system can also be measured and a bottleneck service can be discovered. Such runtime environment should also be able to inject faults to the system under test which this research proposes so.

Based on the discussion above, the proposed solution we propose in this paper is emulating customizable and controllable WANs over LANs. Therefore, the Web service systems are assessed on virtual WANs which are very similar and comparable to the actual target WAN environments. Testing over these virtual WANs will not be suffering the problems of the real WANs discussed above. The assumption made in this thesis is that the actual LANs used in hosting the virtual LANs are very reliable and very fast thus making uncontrolled faults and delays negligible.

Our network emulation is based on the architecture of a fault injection testing method with successful results for testing CORBA Applications [18]. The original testing method is for emulating the behaviour of WAN and injecting network faults at the application level. The messages exchanged between CORBA components are intercepted (using CORBA Interceptors), and then network faults are injected.

However, there are some shortcomings of the CORBA fault injection approach [18]. In CORBA interceptor level the messages are already coded in binary code, therefore this method does not target any particular elements in the message to inject faults such as function parameters in the case of RPC. Also the corruption and dropping messages are only injected by throwing exceptions. That means, the CORBA fault injection method assumption is for injecting the mentioned faults to test only the system ability to deal with such exceptions, whereas it is more logical to inject explicitly the fault and observe its effects on the system. Injecting faults such as dropping and delaying messages can help developers to assign a reasonable time period before the system times out. The problem of the CORBA approach cannot help in how to distinguish between in which a message (or its acknowledgement) is simply experience a delay in the network from those in which a message has actually been lost. If the time-out interval is made too short, then there is a risk of duplicating messages and also reordering in some cases. If the interval is made too long, then the system becomes unresponsive.

All the discussed issues above have been taken into account in order to produce a WAN Emulation for our fault injection method. As discussed in the previous section, the messages are intercepted at application level by using proxies, so at this level the complete message entities are captured and any particular part of the messages can be manipulated. In addition the network faults may be injected explicitly (dropped or corrupted messages). The proposed solution for choosing a reasonable time out period is tackled by testing the system under different real delay rate and drop rate scenarios, then monitoring the system in order to assign the best timeout period that can minimize the risk of confusing between the normal network delays and the message losses.

3.3 Scalability And Overhead

There are some key issues have been considered in order to design our fault injection method. In order to emulate a large multi-hop network, scalability and overhead issues need to be addressed. The emulator must has to scale well for networks with hundreds or more of network nodes, in the meantime maintaining a limit on the overhead of the emulation. It is intended that the network emulator is hosted over a LAN where every physical node is responsible for a clique of virtual nodes. This reduces the chances of uncontrollable faults caused by the underlying hardware or networking devices and allows accurate emulation of other traffic sources. The design assumes that the WAN, needed to be emulated, are large enough that the overhead introduced by the network emulator is negligible when compared to the actual network delays.

3.4 System Monitoring (Failure Detection)

Here we will discuss many ways in which distributed systems and in particular Web service systems can fail and the effects of each failure on the system.

There are many system failure modes, affecting distributed systems, have been identified and classified. For example, in [15] system failure modes which can be occurred in CORBA applications have been identified, and in [11] system failure modes affecting Web service systems are also classified. Based on the above system failure modes classifications, we have summarized system failure modes of composed service systems as follows: 1) crash of a service instance/hosting environments, 2) hang of a service, 3) corruption of data coming into the system, 4) corruption of data coming out of the system, 5) duplication of messages, 6) omission of messages and 7) delay of messages.

This list is very general. Every Web service system has its own specific failure modes. However, the majority, if not all, of these system failure modes can be classified into one of the above major classes of system failure modes.

The effect of the above failure modes will depend on the capability of the fault tolerance of the system to detect them and prevent the system from deviating from its specified behaviour.

Corrupted data coming into the system should be detected by the middleware (or the Web service application) and rejected then raising appropriate error exception as a response. Corruption of data coming out of the system should be handled by the middleware at the service client, however Corruption of data coming out of the system can cause failure when it is not signalled by the system and propagated from the middleware to the application level. In such case a mechanism must be deployed at application level to deal with this.

Duplication and omission of messages should also be handled by the middleware layer of the service and raise appropriate exception. However omission of messages from client to service must be detected by the middleware of the client since the service would have no mechanism for knowing the message had been sent so it could not generate an exception.

If the application server crashed, it will not be able to accept the invocation and the client will get an exception from the transport layer. If the application server hangs, it may either accepts the invocation but does not respond so the client will not know what's happening or the application server may not be able to accept the invocation at all making it more similar to crash.

Delayed messages may cause timing faults. Timing faults should be detected by the middleware at service side when a response message is not received in a specified time slot. However, at the service client there is a problem of distinguishing between a lost request message and the message experience a long delay caused by the network. Therefore, a reasonable time span should be deployed before raising timeout exception at service client to minimize this issue.

Because of all the problems above, some of the failure modes are very difficult to detect. For example as discussed above, it is difficult to distinguish between crash and hang failure modes in some cases, when the testing run by service client that do not have access to the application server logs where the Web service is running. In addition some other failure modes are also difficult to detect when the tester have no access to the service client logs. For example omission of requests when client request is lost before reaching the service provider. As a result of that a mechanism (such as timeout mechanism) deployed to detect omission of request messages at service client cannot be tested.

To face these problems we rely on the logging mechanism of the proposed method and the logging of the client as well. The failure modes mentioned before can be observed by analysing the tool logging mechanism to detecting exceptions caused by corruption of data and detect omission of messages from service to client. In addition for this experiment we use service client logs to detect omission of message failure from the client to service and also the effect of delays of messages. Moreover, as crashes of services/hosting environments and hang of services can be detected via receiving exceptions or via time-out mechanisms applied in service client, client logs are also used to detect such failure in this experiment. For this experiment our fault injection method has no means of detecting duplication of message failure. It will be addressed in later research.

4. NETFIS IMPLEMENTATION

The fault injection method is designed to overcome the shortcomings of earlier fault-injection tools. It minimizes the dependence on the underlying OS and distributed computing platform.

Although the implemented version of our testing method is for SOAP based composed service systems, the architecture is generally applicable to be deployed to any SOA distributed computing platform (e.g. Open Grid Services Architecture) that allows the installation of proxies into the communication subsystem. The emulator is transparent to the applications and requires no modifications, recompiling or patching. NetFIS is also independent of any hosting environment for portability. Finally, NetFIS gives the applications under test the sense that there are other -synthetic- applications running during the test and sharing the networking resources without a perceivable emulation overhead. Our NetFIS Architecture tool consists of three main components as follows:

4.1 FAULT INJECTION SERVICE

Our Fault Injection Service (FIS) is a Web service application that has the ability to generate a proxy Web service to one or more Web service applications of the tested system. More importantly, it injects the proper fault into the system under test by its sub-components.

The FIS role depends upon where it is deployed. At the client side, FIS role is to generate a proxy WSDL from the actual Web service WSDL needed to be called by client. As result, all client requests are processed by the FIS. Thereafter, the FIS sends the intercepted request to its internal subcomponent, the Fault Injection Controller (FIC) to inject the required faults. Then the request is sent to another FIS that is deployed on the site where the actual requested Web service is running. When the client side FIS receives a response from the requested Web service it forwards it to the client. At the actual requested Web service side, the FIS role is different. Request messages received from the FIS, deployed at client side, are forwarded to the actual Web service by the FIS. 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 FIS deployed at the client site. In the case of composed service systems, where the service has to be acting as both a service and client in the same system, a single FIS can perform both of the roles mentioned above. Because of using this way of intercepting messages, no modification need to be made to the system under test.

4.2 FAULT INJECTION CONTROLLER

The Fault Injection Controller (FIC) is a java component resided in the FIS that is responsible for controlling the testing tool and injecting the required faults into the intercepted massages. Faults are injected into the message based upon decisions coming from two other components of the testing tool – the Network Emulation Service (NES) and the Script Fault Model (SFM).

These two components can either be turned on or off at the choice of the user. The SFM is a java script which written by the user. The function parameters can be modified by using the value boundaries specified by the tester. When both SFM and NES are activated, the SFM decision can only be applied if the decision from NES is not to drop or corrupt the exchanged messages. In other words, The FIC gives network faults higher priority. The FIC also logs all the exchanged SOAP messages so as to be analysed offline. Thereafter, the message, if it has not been dropped, is sent back to the interceptor to complete its journey to the corresponding FIS.

4.3 NETWORK EMULATOR SERVICE

The Network Emulator Service (NES) is a WAN Emulator Web service, which gives the applications the sense of running over a LAN or WAN. It gives the applications under test the sense that there are other -synthetic- applications running during the test and sharing the networking resources. In addition, it provides the capability of injecting network faults (loss, corruption, delay, reordering, etc.). This work has been modified and extended by using only Web service technology. All the generated workload traffic and the injected faults use only SOAP messages.

The network emulator service system is deployed and exposed as composed Web services. It consists of a centralized Network Controller Service (NCS), controlling the target emulated network and a set of NES's deployed at every node in the system which emulates the nodes of the target emulated network. The NCS and every NES communicate with each other by exchanging SOAP messages and also communicate with the FIC using also SOAP messages as required.

4.4 SETTING UP THE TOOL

The first stage of setting up the tool consists of building a description of the target emulated network using a topology file and to describe the traffic workload generated on all network nodes.

The next stage is to start the NCS and load the topology. **The third stage** is to start the NES's for all the network nodes. Thereafter, starting the FIS for every node which will cause generating a proxy Web service for each Web service required to be called in the system under test, and finally, order the NEC to start the emulation and then start the system need to be tested and assessed.

The Topology file is a simple configuration xml file that describes the target network topology. It lists the nodes in the network together with their configuration. In addition, a trace file also must be provided for each node describes its traffic workload. The trace file contains the packet counts per unit time and can be either created manually, captured from real traffic traces or produced by using network traffic modelling algorithms. Then, the NCS, which is a Web service itself, is started. NCS is used by NES's in order to provide node configuration parameters and locations of neighbouring NES's. Each node of the target emulated network is represented by one FIS and one NES.
Each FIS at the client side needs to be provided with an xml file containing the URL(s) of the Web service(s) under test. The client needs to call this, in order to generate a Web service proxy which will be called by the client instead of the actual Web service under test. In addition, the xml file also contains the URL of the NES emulating the same node.

As the tool does not require any modifications to the system under test, unless the only job for the client is to start calling the proxy service generated by the FIS instead of calling the actual Web service.

5. TEST CASE

In this section we describe an experiment that injects a number of network-related faults (delaying, dropping and randomly corrupting SOAP messages) into a Bioinformatics Web service [19]. We deployed the WS-Mediator [4] at the client side to invoke three identical Bioinformatics Web services [19] simultaneously via the NetFIS. The performance and the fault tolerance protocols of the system under test have been examined. Moreover the overheads introduced to the system by using our tool are also measured. The results obtained from logging files are analysed and discussed. The setup of the experiment is explained in the next section.

5.1 EXPERIMENT SETUP

The topology of the target network that we emulated is a four-node network setup as shown in Figure. 1. In the experiments various types of faults were injected into the emulated network. The Network Emulation Service is enabled to generate synthetic traffic through the network.

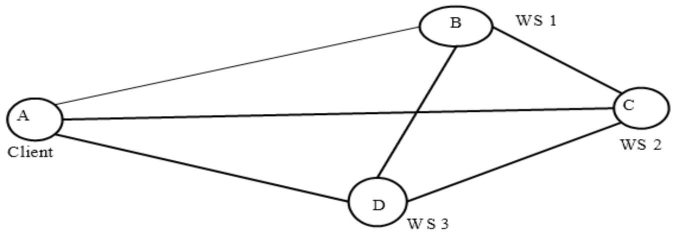


Figure 1. Network topology

For a real application deployed on a WAN, there is a significant variation in performance due to other traffic occupying the network resources. NetFIS supports various simulated traffic models including, but not limited to, self- similar, random, constant and even replaying previously captured traffic traces. Since studies of network traffic suggest that it is self-similar in nature [20], we chose to emulate continuous self-similar traffic in our network. The mean packet rate is 30 packets /second on each link, and the self-similarity value is 0.8. The packet size distribution follows measurements taken from Internet backbones [21]. The link utilization varies based on the generated packet size and the link configuration.

5.2 NETWORK CONFIGURATION

We measure the performance of the protocols in four network configuration scenarios:

5.2.1. LAN CONFIGURATION

The LAN was used to test the base performance of the target system without deploying NetFIS. The Web service client issuing the requests was loaded on machine A in Figure. 2. The 3 services participating in the test are run on machines B, C and D respectively.

5.2.2 FAST WAN CONFIGURATION

The propagation delays are fixed at 2ms which is typical of inter-city links within the UK. The bandwidth of each link is 4mb/s. The average utilization of each network link given this bandwidth, and the simulated traffic detailed in previous section, ranges between 10% and 20%.

5.2.3 SLOW WAN CONFIGURATION

This configuration represents the other extreme network environment. All services are located in far apart geographical locations and connected by slow links. The propagation network delays are fixed at 50ms which can be typical of far apart geographical locations and international links (e.g. between Newcastle, England and Tripoli, Libya). The bandwidth of each link is 512Kb/s. The average utilization of each network link, given this bandwidth and the traffic, ranges between 20% and 40%.

5.2.4 HETEROGENEOUS WAN CONFIGURATION

This configuration represents a case somewhere between the two extremes. One of the services was placed in a faraway location (connected by slow WAN links) while the other servers and the client were closer to each other (connected by fast WAN links). The links and loads used here are similar to those used for the slow and fast WAN configurations.

5.3 CLIENT CONFIGURATION

We have developed a special client application implementing several test cases corresponding to the fault injection configurations applied during the experiment. The client application is implemented on the WS-Mediator framework (as shown in Figure. 2) and utilizes the built-in FTMs and logging mechanisms of the framework. The WS-Mediator claims to offer comprehensive off-the-shelf FTMs in order to cope with various kinds of typical Web service application scenarios. It also includes a monitoring mechanism to benchmark a collection of candidate Web services that would be used during service composition and generate their dependability metadata for dynamic composition reasoning. The framework allows the client to submit a number of candidate Web services for service composition and define a reconfiguration policy to specify how to make use of the candidate Web services, and thus to reduce the development cost of a dependable client application.

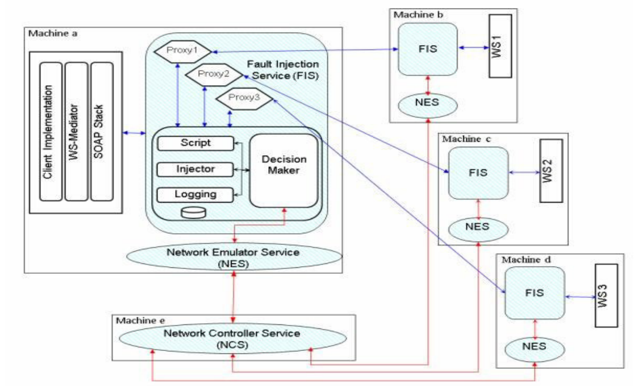


Figure 2. A system being tested by NetFIS

In our Web service client application, we chose to use the deployed N-version programming mechanism offered by the WS-Mediator in order to invoke the NetFIS proxies simultaneously and choose the first valid response from the service a client's request. During the invocations, all request and response messages are logged using the built-in monitoring mechanism of the WS-Mediator. The complexity and processing overheads of the WS-Mediator have been minimized with these settings. It is worth noting that classic N-version programming approach normally requires voting for result validation.

However in Web service applications, although similar Web services may return semantically identical responses they are not usually exact matches. The WS-Mediator framework allows a policy-based response mapping, however since the NetFIS tool injects random faults into the SOAP messages especially with random timing, the response mapping and voting mechanisms are of little value in our test cases. Nevertheless, the late responses are also logged for further analysis.

Besides the fault tolerance mechanisms deployed in the client application, the functionality of the client is fairly simple. It invokes the three replicated Web services repeatedly with or without the NetFIS. The number of invocations and the delay interval between invocations can be configured dynamically.

5.4 Experimental Results

The experiment comprises several test cases for validating the NetFIS approach. All the events have been logged (SOAP requests and responses, injected faults, round trip response times and exception messages) during the experiment. Those saved logs generated by the client application and the NetFIS have been used for quantitative result analysis.

Section 1: the NetFIS emulates different types of network with simulating various traffic load. Details about the settings are shown in Table 1. A preliminary assessment test was carried out to check the network condition and the Web service before the performing other test cases. The client invoked the three Web services directly 1000 times (interval: 1000ms) without the NetFIS. The overall maximum, minimum, and average round trip response time (RTT) received by the client application are 102ms, 8ms and 57ms respectively.

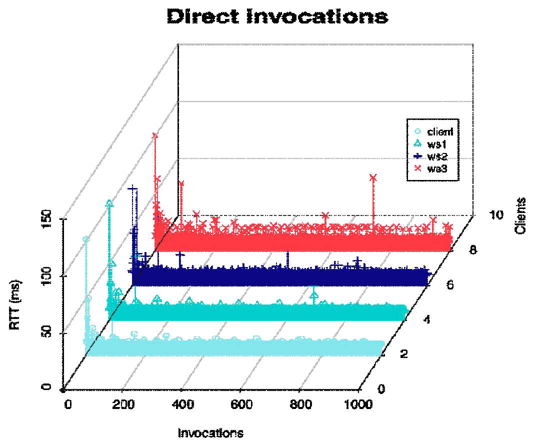


Figure 3. Client invocation RTT

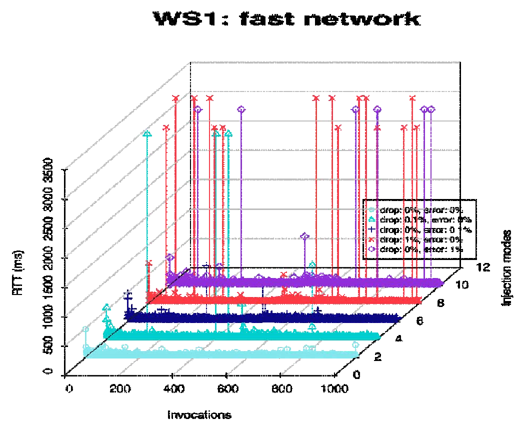
Figure. 3 shows the RTT to the three Web services logged by the WS-Mediator. It is very interesting to see the three Web services had much longer RTT at the very beginning of the test suggesting the RTT could have been optimized by some kind of caching mechanism employed in the Web services. It is also worth noting although the three replicated Web services have identical hardware, operating system, middleware, etc., WS3 constantly had longer delays than WS1 and WS2. However, the RTT variations of Web services and between different Web services are indeed insignificant compared with the delays to be injected by our toolkit. Therefore can be safely ignored. The average RTT of WS1, WS2 and WS3 are 10ms, 11ms and 12ms. The average client RTT was slightly smaller than 10ms, because it always uses the quickest response from the three Web services.

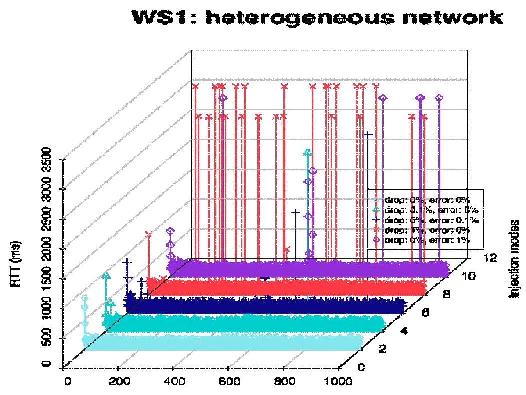
The preliminary test results provided the benchmarking information of the physical LAN and Web services involved in the experiment. Then the NetFIS were added between the client and the Web services, and the client made 1000 invocations in each setting.

Table 1. Response time overhead

Network	**Bandwidth (Mb/s)**	**Response time**		
		Max, ms	***Min, ms***	***Average, ms***
LAN	N/A	102	8	57
Fast WAN	4000	488	35	59
Slow WAN	512	698	110	190
Hetero.WAN	Fast and Slow	870	99	104

The result statistics shown in Table 1 clearly indicated the effectiveness of emulating the three different networks between the system components. The average RTT - without injecting drop and error faults - when emulating Fast WAN is 59ms, where the average RTT without using our tool in LAN is 57ms. That means the overhead delay introduced by our toolkit to the tested system is clearly insignificant. However the differences of the average response time between the Fast WAN and the Slow WAN is indeed big. That is related to the configurations of the two emulated networks, specifically the propagation delays and the bandwidths where in Fast WAN are 2ms, 4mb/s and in Slow WAN are 50ms, 512Kb/s respectively. When considering Heterogeneous WAN, the average response time is almost between the average response times of the Fast and Slow WANs. That is due to Heterogeneous WAN is configured of a combination of the other two WANs (Fast and Slow).





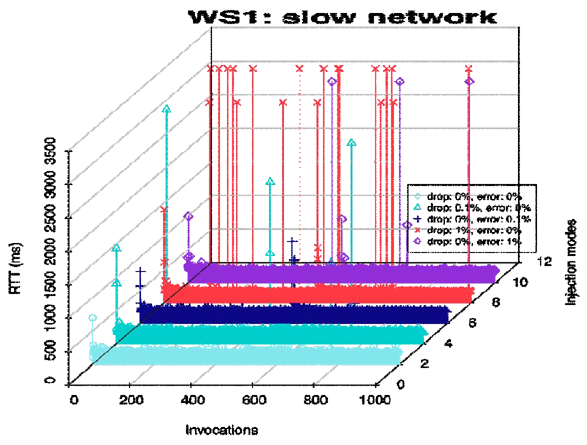


Figure 4. RTT of the test cases (Web service 1)

The Figure shows the RTT of WS1 (monitored at the WS-Mediator client) at different drop and error rates and network conditions. The 'injection modes' axis represents the invocation RTT of each injection mode shown in the Figure legend.

The overall average RTT of the fast network is much smaller than of the other two network conditions. The Figure clearly shows greater RTT variations of the heterogeneous network than the slow network. The timeout value has been regulated to 3000ms in the Figure to make the plots more readable.

Section 2: Our tool injects different types of faults into the exchanged messages between the client and the Web services in different emulated network conditions. The combinations of those injected faults are detailed in Table 2. The client invoked the Web services via the NetFIS 1000 times in each testing scenario setting.

Table 2. Drop and random error injected

Network Emulated	**Injected Drop rate**		**Injected Error rate**	
	Target %	***Achieved (total messages).***	***Target %***	***Achieved (total messages).***
Fast WAN	0.1	1	0.1	1
	1	9	1	10
Slow WAN	0.1	1	0.1	1
	1	10	1	10
Hetero.WAN	0.1	1	0.1	1
	1	9	1	10

Table 2 indicates the statistics of the results in each test case scenario. The results show that the NetFIS coped well with the settings and injected the expected faults correctly. When "drop" fault is injected, the client threw a "timeout" exception after 10 seconds waiting indicating that the response was lost.

When errors are injected, “Cannot find dispatch method for{http:%/webservices.calibayes.ncl.ac.uk/}getAvailableSimMethods”., exception messages were thrown by the client indicating that corrupted SOAP messages were received but the JAX-WS framework was not capable to correctly deal with the responses. Figure. 4 and Figure. 5 indicate the plots of the results of some test scenario cases, which obviously demonstrate the effectiveness of the. The NetFIS simulates real work network conditions and faults in order to help on robust client application development (in this case, by applying the WS-Mediator).

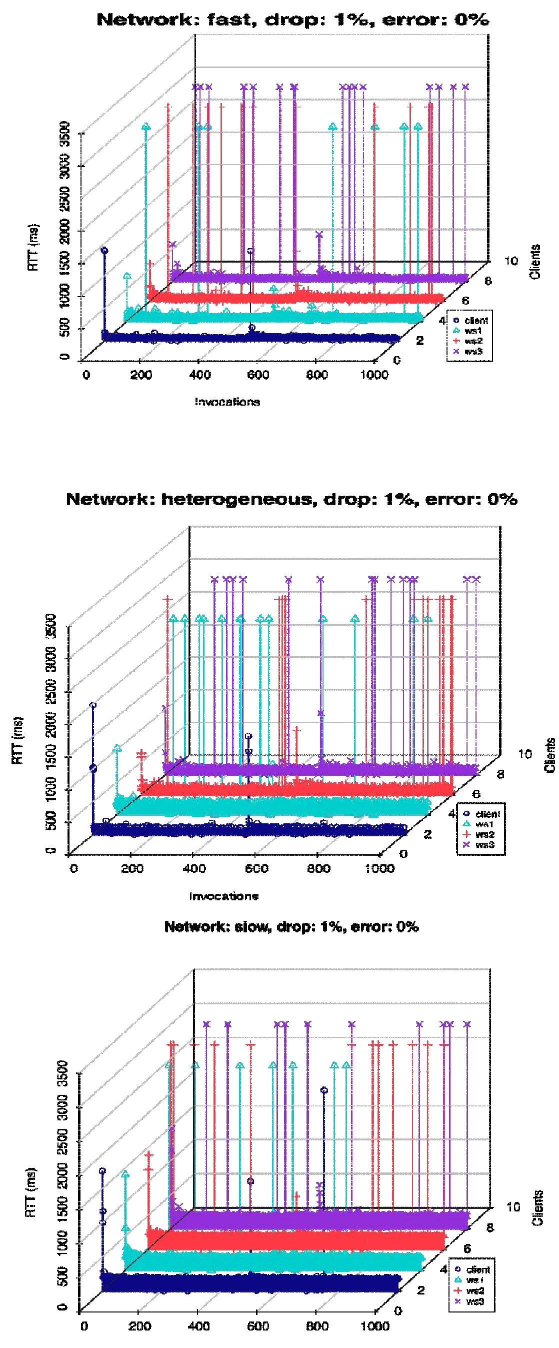


Figure 5. RTT of the test cases (Client)

Figure 5 indicates a comparison of the RTT of the three Web services and the responses delivered to the client through the WS-Mediator. The ‘clients’ axis represents the invocation RTT monitored at each client thread (respectively deals with WS1, WS2, WS3) and the client application that deploys the WS-Mediator to deal with the results received by the client threads.

We chose the 1% drop rate injection scenario to illustrate the comparison since this test case scenario affects the RTT most. As the faults were injected arbitrarily into the three Web services, the N-version programming FTM in the WS-Mediator successfully dealt with the injected faults in most test cases and masked the reliability issues to the client. Moreover, the client application only threw exceptions when all the three Web services failed simultaneously.

6. CONCLUSION

We have introduced a testing methodology and built a toolkit that is capable to inject faults into any Web service application without any modification to the code of the application. There is great flexibility in both the number and type of faults that can be injected into the system under test. Furthermore, we can control the emulation of the network that is used and the ability to add background traffic.

The network emulation may not exactly mirror the real world of the network environment, however, it is a significant advance on testing a system using a single machine or a LAN. In particular, a sample traffic from a real workload of a network can be used in the network emulator as well as self-similar traffic patterns.

Our experiment has clearly demonstrated the network emulation and fault injection capacities of the NetFIS and an experiment of how to deploy and use the functionalities of our toolkit for testing the FTMs of the client application. In this experiment, the WS-Mediator has demonstrated its FTMs capacity with service diversity and dynamic service composition reconfiguration.

In future work, we are concentrating on two main issues. Firstly, the toolkit does not have the capability to deal with Call-back asynchronous invocations yet. In this stage dealing with Call-back asynchronous invocations is left for future work as Dispatch has become a standard.

Secondly, as the goal of WS-ReliableMessaging [22] is to empower applications to exchange messages simply, reliably, and efficiently even in the face of application, platform, or network failure at middleware level, our future plan is to test and assess NetFIS with WS-RM powered Web service systems.

REFERENCES

[1] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, Fault Injection Techniques and Tools. IEEE Computer, vol. 30, pp 75-82, 1997.

[2] Papazoglou, M., & van den Heuvel, W. J. (2006). Service-Oriented Design and Development Methodology. International Journal on Web Engineering and Technology, 2(4), 412–442.

[3] Z.Zheng and M. R. Lyu. Ws-dream: A distributed reliability assessment mechanism for web services. In DSN, pages 392–397, Anchorage, Alaska, USA, June 2008.

[4] Y. Chen and A. Romanovsky. Improving the dependability of web services integration. IT Professional, 10(3):29–35, 2008.

[5] S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment for Distributed Real-time Systems," International Computer Performance and Dependability Symposium, Erlangen; Germany, pp.204-213, 1995.

[6] S. Dawson, F. Jahanian, T. Mitton and T.-L. Tung, Testing of Fault- Tolerant and Real-Time Distributed Systems via Protocol Fault Injection, in Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26), Sendai, Japan, 1996, pp.404-414.

[7] de Almeida, L.F. and S.R. Vergilio. E xploring Perturbation Based Testing for Web Services. in Web Services, 2006. ICWS '06. International Conference on. 2006.

[8] Siblini, R. and N. Mansour. Testing Web services. in Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on. 2005.

[9] Sneed, H.M. and H. Shihong. WSDLTest - A Tool for Testing Web Services. in Web Site E volution, 2006. WSE '06. Eighth IEEE International Symposium on. 2006.

[10] Jeff, O. and X. Wuzhi, Generating test cases for web services using data perturbation. SIGSOFT Softw. E ng. Notes, 2004. 29(5): p. 1-10.

[11] Looker, N. and J. Xu. Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection. in Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. The Ninth IEEE International Workshop on. 2003.

[12] Zhang, J. and R.G. Qiu. Fault injection-based test case generation for SOA-oriented software. in 2006 IEEE International Conference on Service Operations and Logistics, and Informatics, SOLI 2006. 2006.

[13] Looker, N., M. Munro, and J. Xu. WS-FIT: A tool for dependability analysis of web services. in Proceedings - International Computer Software and Applications Conference. 2004. Hong Kong, China.

[14] F. Bessayah, A. Cavalli, W. Maja, E. Martins, A. Willik Valenti, A Fault Injection Tool for Testing Web Services Composition, Proc. 5th Testing Academic and Industrial Conference -- practice and research techniques (TAIC PART 2010), Windsor, UK, (LNCS 6303), 2010, pp. 137-146.

[15] E. Marsden, J.-C. Fabre, J. Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection", 21st IEEE Int. Symp. on Reliable Distributed Systems (SRDS-2002), (Osaka, Japan), pp.276-285, IEEE CS Press, 2002.

[16] P. Reinecke, A. P. A. van Moorsel, and K. Wolter. The Fast and the Fair: A Fault-Injection-Driven Comparison of Restart Oracles for Reliable Web Services. In QEST '06: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems, pages 375–384, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Khaled Farj, Adel Smeda, "A Methodology for Evaluating Fault Tolerance in Web Service Applications", Proceedings of the 15th International Conference on Applied Computer Science (ACS '15), p. 188-191, Konya, Turkey, May 20-22, 2015.

[18] Mohammad Alsaeed, Neil A. Speirs, "A Wide Area Network Emulator for CORBA Applications", Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, p.359-364, May 07-09, 2007.

[19] Chen, Y., Lawless, C., Gillespie, C.S., Wu, J., Boys, R.J., Wilkinson, D.J., 2009, CaliBayes and BASIS: integrated tools for the calibration, simulation and storage of biological simulation models. Briefings in Bioinformatics.

[20] Mark E. Crovella and Azer Bestavros, "Self-similarity in World Wide Web traffic: Evidence and possible causes," IEEE/ACM Transactions on networking, vol. 5, no. 6, pp. 835-846, Dec. 1997.

[21] C. Fraleigh and S. Moon, et al. "Packet-Level Traffic Measurements from the Sprint IP Backbone", In: IEEE Network, Volume 17, Number 6: November 2003.

[22] OASIS. (2010). 'OASIS Web Services Reliable Messaging (WSRM) TC'. [Retrieved: 30 Jun 2010]; Available from: http://www.oasis open.org/committees/tc_home.php?wg_abbrev=wsrm.

AUTHORS

Dr. Khaled Farj is a specialist in distributed systems, web and cloud applications. Khaled gained a PhD in Cloud Computing, and MSc in Computing Sciences at Newcastle University (UK). Also a Diploma in e-Commerce, UMIST (UK). He is involved in numerous professional and academic activities related to distributed systems and some government systems. He was the Technical Project Manager for IT Libyan Customs team; the project is for automating all Libyan ports based on what's called ASYCUDA application. Thereafter, worked as a Head of IT Department at the Ministry of Higher Education (Libya). Khaled also assigned as the Ministry Coordinator of the national e-government long-term strategic project plan. In addition, involved in the academic higher education sector, by teaching some modules for MSc students and supervising some student projects and researches at University. He is interested and involved in some research areas, such as Cloud and Web Computing, Architecture for distributed Web applications, Enterprise Computing and Middleware and Web Services.

Dr. Adel Smeda is an associate professor graduated from université de Nantes, France. He is a staff member of university of al-jabal al-gharbi, Libya, adjunct professor at the Libyan academy in Tripoli, and an active member of the research centre LINA of université de Nantes in France. His area of interest includes software architecture, cloud computing, modelling, E-technologies and published more than 40 articles in these areas.