A Fault Injection Method for Testing Dependable Web Service Systems

Khaled Farj
School of Computing Science,
Newcastle University,
NE1 7RU, UK.
k.a.s.farj@ncl.ac.uk

Yuhui Chen
Wellcome Trust Centre for Human Genetics,
University of Oxford,
OX3 7BN, UK.
Jeff.yuhui.chen@gmail.com

Neil A. Speirs
School of Computing Science,
Newcastle University, NE1 7RU, UK.
neil.speirs@ncl.ac.uk

Abstract

Testing Web services performance and their Fault Tolerance Mechanisms (FTMs) are crucial for the development of today's applications. Testing the performance and FTMs of composed service systems is hard to measure at design time because service instability is often caused by the nature of the network. Using a real internet environment for testing is difficult to set up and control. We have developed a fault injection toolkit that emulates a WAN within a LAN environment between composed service components and offers full control over the emulated environments in addition to the ability to inject network-related and application specific faults. The tool also generates background workloads on the tested system for producing more realistic results. We describe an experiment that has been carried out to test the impact of fault tolerance protocols deployed at a service client by using our fault injection toolkit.

1. Introduction

Web services technology is becoming progressively more important in service oriented computing and applications. Web services are software programs that offer services over the Internet to other software programs, including web applications and other Web services. They have changed the way we look at the Internet from being a repository of data into a repository of services.

Web services play critical roles in developing information systems that rely on integration of heterogeneous data and autonomous component 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 [1]. These services are usually developed and administrated by different service providers, running on different platforms, and distributed over the Internet in different geographic locations.

The quality of such complex systems depends both on the quality of the network environment and 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 assessing their overall quality. Web services are inherently distributed and heterogeneous, and in real-world practice, are often invoked with little understanding of their reliability and performance.

In many applications, service composition often relies on dynamic service integration, meaning that very possibly the component services are discovered, selected, and composed at runtime. Thus it becomes more difficult to assess the behaviour and performance of these component services, especially in the presence of transient faults. Therefore, there is no guarantee that all parts of the composite service are highly reliable. In [15] it is reported that communication faults such as message loss, duplication reordering, or corruption have an effect on traditional distributed systems such as CORBA applications. Moreover it has been found that unstable Internet environments and server connections can lead to unreliability of Web service applications [3].

Web services are subject to many network faults such as delaying, dropping, damaging, and reordering messages and also to software faults within the services.

Testing the performance and fault tolerance of Web services has become an active research area. Software Fault injection is a well-proven method of assessing the reliability of a system [2]. In this paper we describe a tool for testing the performance and fault tolerance of either a single Web service or composed service, without modification to the component services being tested. No recompiling or patching is necessary. Furthermore, the tool will generate background workload to more accurately emulate real networks. Our tool is also independent of the hosting environment for portability. In Section 2, we describe existing tools for injecting faults into web services. In Section 3 and 4, we describe the design and implementation of our Network Fault Injector service (NetFIS). In Section 5, we describe an example experiment that uses NetFIS to assess the dependability of a Bioinformatics Web service.

This Web service employs an off-the-shelf Mediation [4] framework to provide fault tolerance. We demonstrate how to apply the NetFIS tool in evaluating the performance and the fault tolerance mechanisms of a service. Section 6 describes our conclusions and future work.

2. Related work

There are many Software Fault Injection tools for testing general distributed systems. Some of these are specifically designed for testing Web services. Well-known fault injection tools, such as DOCTOR [5] and Orchestra [6], support network level fault injection and could potentially be used to inject faults into Web services. These two tools however have been designed for testing network protocols without decoding complete middleware message sequences. Therefore they are not wholly suitable for debugging Web services in which message integrity is a big concern.

In Almeida and Vergilio [7] a tool is proposed for generating and validating test cases. The tool interprets WSDL schema, and then introduces some operator to generate a request with random data and a test script that manipulates the request parameters. Paper [8] proposes a technique for testing Web services using mutation analysis. A mutant WSDL document is generated by applying mutant operators to the original WSDL document. A test tool called WSDLTest [9] generates Web service requests from the WSDL schemas and tunes them in accordance with the pre-conditions written by the user, and then verifies the responses against the post-conditions offline. In [10] a testing tool is proposed based on some rules defined in XML schema or DTD. The tool modifies the value of the parameters in requests by using boundary value testing, and on interaction perturbation, using mutation analysis. Another tool [11] 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 [12] helps service requesters to create test cases so as 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. The WS-FIT tool [13] injects faults by modifying SOAP messages using scripts. The user may define the value boundaries that the tool uses for manipulating the function parameters. The *WSInject* [14] tool injects both communication and interface faults and can WS-RM WS-RM be used for testing a single Web service or composed service system.

A common characteristic of the work discussed above is that their focus is largely on testing single services in isolation (*such as WSInject*); furthermore, most of them focus on injecting faults by modifying the SOAP message. We argue that such a strategy is only sufficient for debugging simple Web services and communication between a client and a Web service, but is hard to extend to testing composite services. Many Web service applications employ dynamic service integration and service redundancy, and may involve multiple autonomous component services. In order to test the dependability of such application as a whole, many more parameters have to be considered. For instance, the workload of the component systems and networks, the quality of the network environment etc. may greatly influence the overall dependability of a Web service application. Different workloads could lead to different testing results, due to cause different system activation patterns [15]. System testing becomes an even more difficult task when dealing with composed Web services, in which a component service may also act as a client of other services.

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

3. Network fault injection method

The Network Fault Injector Service (NetFIS) is a Web service system that implements our fault injection methods for testing Web service performance and FTMs. The tool acts as a fault injection proxy and network emulator between the Service Client and the Service Provider. It manipulates the invocation and response messages to emulate incorrect behaviours of faulty networks and services. It intercepts the request from the Service Client, injects appropriate faults (if any), and then forwards the request to the Service Provider. Similarly, it intercepts the response from the Service Provider, injects faults (if any), and then forwards it to the Service Client.

NetFIS is able to emulate WAN behaviours and inject network faults such as message dropping and delaying, and randomly corrupting SOAP messages. There are existing tools for inject networking faults in Web service applications. For example, [16] proposes a tool to inject faults at the IP level to investigate the effect of the TCP retransmission mechanisms on Web services. This allows examining the relationship between the time-out and retransmission mechanisms implemented in the TCP and the WS-ReliableMessaging protocol. By contrast, our tool can drop a whole SOAP message which may consist of more than one packet. In this way the consequences of faults injection can be propagated to the application level so as to examine whether the application level FTMs can effectively handle such faults. In addition, the tools can interpret Web services parameters definitions (including data types) from WSDL files to inject software faults into RPC parameters based, however this class of faults will not be detailed in this paper.

The network emulation mechanisms implemented in the NetFIS are configurable. This gives the ability to control every property of the emulated networks. The tool implements a graphical user interface to allow users to control the emulator at runtime for simulating dynamic networking environments. The tool uses the network topology configuration file, network traffic trace files and the GUI to measure and test the performance and FTMs of composed service systems in an emulated WAN.

3.1 Application level and network level

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

Networking faults typically involves packet corruptions, reordering, and dropping. Faults may be present in the physical media and all the layers in the network stack. It is logical to inject intentional network related faults at the network level in networking emulation. Traditionally, it has been done for assessing the reliability and performance of networking protocol stacks. Such faults are normally automatically dealt with by the actual underlying network protocols [15]. Consequently, the application or middleware will not notice the presences of the faults, and therefore will not be tested. Moreover network level fault injection tampers with packets, but not application level messages.

NetFIT is intended to inject communication faults and tests their impact on the performance and FTMs of a composed service system. Therefore it is more efficient and desirable to inject faults at the application/middleware level instead of the network level. Communications between the component 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

In composed services, the component services in the system are usually deployed over the Internet. The performance and fault tolerance of an application with composed services are very difficult to be measured at design time. In order to test such systems, a distributed testing environment is required, such as a WAN or the Internet. However, it is usually impractical to use the real-world Internet or WAN for system tests. It is very costly and time consuming to set up a WAN or use 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 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 will help the testers to test the performance and fault tolerance of a system 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 be measured, and a bottleneck service can be discovered. Such runtime environment should also be able to inject faults to the system under test.

Based on the discussion above, the proposed solution we present in this paper is emulating customizable and controllable WANs over LANs. This way the Web service systems are tested on virtual WANs that are very similar and comparable to a real world WAN environment. Testing over these virtual WANs will not be suffering the problems of the real WANs as previously discussed. The assumption made in this work is that the actual LANs used for hosting the virtual LANs are very reliable and very fast thus making uncontrolled faults and delays negligible. We argue this assumption applies to most well maintained LANs.

Our network emulation is based on the architecture of a fault injection testing method with successful results for testing CORBA Applications [17]. The original testing method is for emulating the behavior 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 [17]. In CORBA interceptor level, the messages are already coded in binary code. The method does not target any particular elements in the message to inject faults, such as function parameters, in the case of RPC. Also message corruption and dropping faults are only injected by throwing exceptions. That means the CORBA fault injection method can only inject the mentioned faults to test only the system's ability of dealing with such exceptions, whereas it is more logical to inject explicitly the faults and observe the 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 CORBA approach cannot help in distinguishing between message delays and message losses. 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's performance will suffer.

All the discussed issues above have been taken into account in order to produce a WAN Emulation design 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 (dropping or corrupting messages). The time out period setting issue is tackled by testing the system under different real delay rate and drop rate scenarios. By explicitly assigning the best timeout period, the risk of confusing between the normal network delays and the message losses can be minimized.

3.3 Scalability and overhead

There are some other 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 scale well for networks with hundreds or more of nodes while maintaining an acceptable emulation overhead. It is intended that the emulation be 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 emulated WAN is large enough so that the emulation overhead is negligible compared to the actual network delays.

3.4 System monitoring (failure detection)

Many failure modes affecting distributed systems have been classified. For example, in [15], failure modes, which can occur in CORBA applications, have been classified. In [11], failure modes affecting Web service systems are also explained. Based on the those failure modes classifications, we summarize the failure modes of composed service systems as follows: 1) crash of a service instance/hosting environments, 2) service hang, 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.

The effect of the above failure modes depends on the capability of the FTMs of the system that detects the faults and prevents the system from deviating from its specified behavior. Corrupted data coming into the system should be detected by the middleware (or the Web service application), rejected, and then raise an appropriate error exception as a response. Corruption of data coming out of the system should be handled by the middleware at the client side. Undetected corrupted data can cause failures when being propagated from the middleware to the application level. In such cases, a mechanism must be deployed at application level to deal with the issue.

Duplication and omission of messages should also be handled by the middleware layer of the service and raise appropriate exceptions. 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.

If an application server is crashed, it will not be able to accept invocations and the client will get exceptions from the transport layer. If the application server hangs, it may not respond to invocations, making it hard for the client to discover what has happened.

Delayed messages may cause timing faults. Timing faults should be detected by the middleware at client side when a response message is not received in a specified time. However at the service client there is a problem of distinguishing between a lost request message and the message experience a long delay in the network. 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. In addition, some other failure modes are also difficult to detect. For example, it is difficult to distinguish if a client request is lost before reaching to the service provider or the acknowledgement packet is lost before reaching to the client.

To face these problems, we rely on the logging mechanism of the proposed methods and the logging of the client as well. The failure modes mentioned before can be observed by analyzing the log files to detect exceptions caused by corruption and omission of messages. Crashes of services/hosting environments and hanging services can be detected by receiving exceptions or via time-out mechanisms applied at the client side.

4. NetFIS Implementation

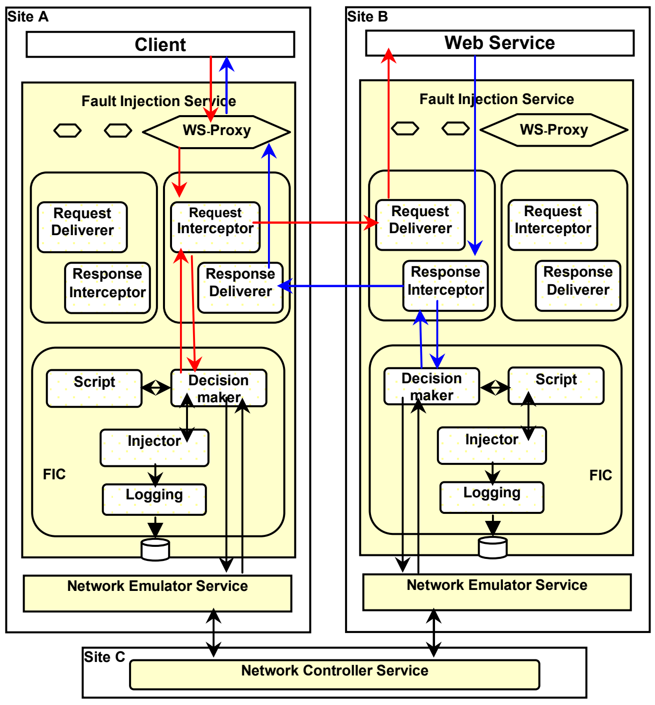


Figure 1. NetFIS Architecture for a 2 Node System

The NetFIS tool has been implemented for SOAP based composed service systems. However, the architecture is generally applicable to most SOA distributed computing platforms (e.g. Open Grid Services Architecture). The emulator is transparent to the applications and requires no modifications, recompiling or patching to the middleware. It is also independent of the hosting environment for portability. Finally, it emulates other synthetic application traffic running at the same time and sharing the networking resources. The architecture of NetFIS is shown in Figure 1 for a 2 node system

4.1 Fault injection service

The Fault Injection Service (*FIS*) is a Web service which has the capability to generate a proxy Web service to one or more Web services of the system under test. More importantly, it injects the proper faults into the system under test with its sub-components.

At the client side, FIS generates a proxy WSDL from the actual Web service WSDL needed to be called by client. As a result, all client requests are processed by the *FIS*. Thereafter, the *FIS* sends the request to its internal subcomponent, the *Fault Injection Controller (FIC*) to inject faults. Then the request is sent to another *FIS* that is deployed on the site where the actual Web service is running. When the client side *FIS* receives a response from the Web service, it forwards it to the client.

At the Web server side, messages received from the client side *FIS* 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 services, where a component service acts as both a service and a client in the same system, a single *FIS* can perform both of the roles explained above. By using this way of intercepting messages, no modification is made to the system under test.

4.2 Fault injection controller

The Fault Injection Controller (*FIC*) is a component inside the *FIS* which is responsible for controlling the tool and injecting the proper faults into the messages. Faults are injected into the SOAP messages based upon decisions coming from two other components of the 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 program written by the user. The function parameters may be modified by using the value boundaries specified by the tester. 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. The *FIC* also logs SOAP messages to be analyzed offline. 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 that there are other synthetic applications running at the same time and sharing the networking resources. In addition, it provides the ability to inject network faults (loss, delay, corruption, reordering, etc.). All the generated workload traffic and the faults injected use SOAP messages.

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

4.4 Setting up the tool

The first stage consists of building a description of the target network using a topology file and to describe the traffic load to generate on all network nodes. The next stage is to start the *NCS* and load the topology. The third step is to start the *NES's* for all the network nodes. Then to start the *FIS* for every node which will generate a proxy service for each service to be called, and finally, to order the *NEC* to start the emulation and then start the system to be tested.

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 configurations. In addition, a trace file also must be provided for each node to describe its traffic load. It shows packet counts per unit time and can either be created by hand, captured from real traffic traces or produced using network traffic modelling algorithms. Then, the *NCS*, which is a Web service itself, is started. *NCS* is used by *NES's* to provide node configuration parameters and locations of neighbouring *NES's*. Each node of the 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. 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, 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. Example experiment

In this section we describe an example experiment that injects a number of networking faults (delaying, dropping and randomly corrupting SOAP messages) into a collection of Bioinformatics BLAST Web services [18]. BLAST is an algorithm which is commonly used in *silico* experiments in bioinformatics to search for gene and protein sequences that are similar to a given input query sequence. We needed a fault tolerant mechanism to test so we deployed the WS-Mediator [4] at the client side to invoke a three times replicated Bioinformatics Web services [18] via NetFIS. The performance and the fault tolerance protocols of the system under test have been examined.

5.1 Experiment setup

The topology of the target network that we emulated is a four-node fully connected network. Since studies of network traffic suggest that it is self-similar in nature [19], we chose to emulate continuous self-similar traffic in our network. We used a mean packet rate of 30 packets/second on each link, and the self-similarity parameter value is 0.8. The packet size distribution follows measurements taken from Internet backbones [20]. The link utilization varies based on the generated packet size and the link configuration.

5.1.1 Network configuration

We measure the performance of the protocols in four network configurations:

i). LAN Configuration: The LAN was used without deploying NetFIS to test the base performance of the system.

ii). 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 link given this bandwidth, and the simulated traffic described in previous section, ranges between 10% and 20%.

iii). Slow WAN configuration: All services are located in far apart geographical locations and connected by slow links. The propagation delays are fixed at 50ms which is typical of far apart 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 link given this bandwidth and the traffic ranges between 20% and 40%.

iv). Heterogeneous WAN configuration: This configuration represents a case somewhere between the two extremes. One of the services was placed in a far away location (connected by slow WAN links) while the other servers and the client were close 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.

5.1.2 Client configuration

The functionality of the client is as follows. 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.

In our client application, we chose to use the 3-version, first response programming mechanism offered by the WS-Mediator i.e. requests are sent to all three service replicas and the first response obtained is accepted. There is no voting of responses. During the invocations, all request and response messages are logged.

5.2 Experimental Results

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

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

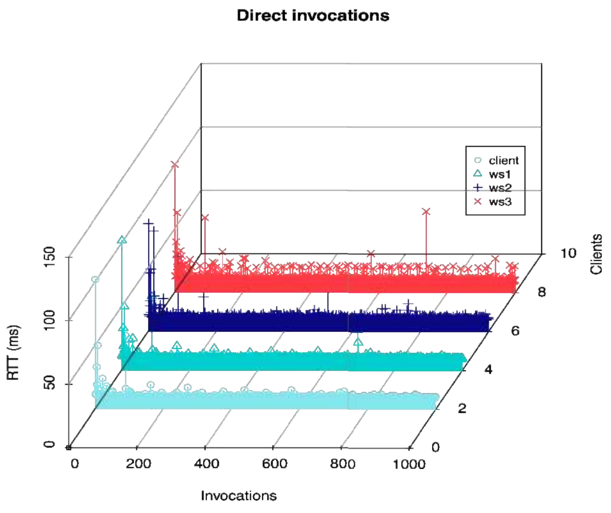


Figure 2. Client invocation Round Trip Response Times

Figure 2 shows the RTT to the three Web services logged by the WS-Mediator. It is interesting to note that the three Web services had much longer RTT at the very beginning of the test. It may suggest that the RTT could have been optimized by some kind of caching mechanisms 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 a Web service itself and between different Web services are indeed insignificant when comparing with the delays to be injected by the NetFIS, and 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 NetFIS was 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

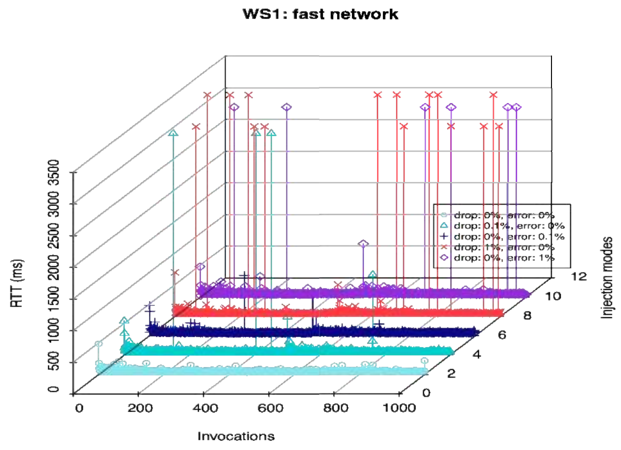
The statistics of the results shown in Table 1 clearly indicated the effectiveness of the emulation on the three different networks between the system components. The average RTT of the Fast WAN, without injecting drop and error faults, is 59ms, where the average RTT of the LAN, without using our tool, is 57ms. That means the delay overhead introduced by NetFIS to the system under test is clearly insignificant. However the differences of the average response time between the Fast WAN and the Slow WAN is indeed large. With the 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).

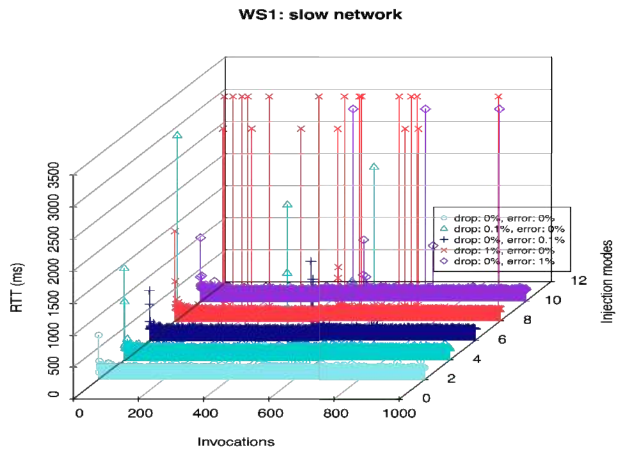
Section 2: The NetFIS injects various types of timing faults between the client and the Web services in different emulated network conditions. The combinations of the injected faults are shown in Table 2.

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

The client invoked the Web services via NetFIS 1000 times in each setting. Table 2 shows the statistics of the results in each test case. The results indicate that the tool coped well with the settings and injected expected faults correctly. When "drop" is injected, the client threw a "timeout" exception after 10 seconds waiting; this indicates the response was lost. When errors are injected, "*Cannot find dispatch methodfor{http:%/webservices.calibayes.ncl.ac.uk/}getAvailableSimMethods*" exception messages were thrown by the client which indicates corrupted SOAP messages were received but the JAX-WS framework was unable to correctly deal with the responses.





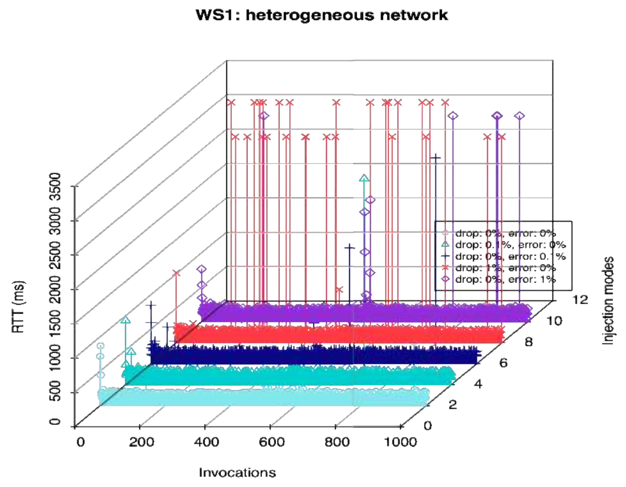
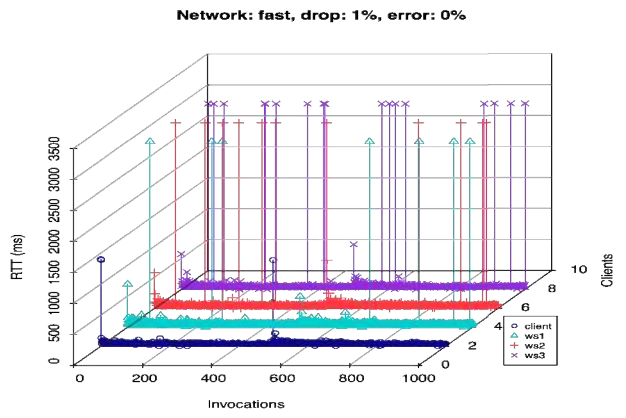
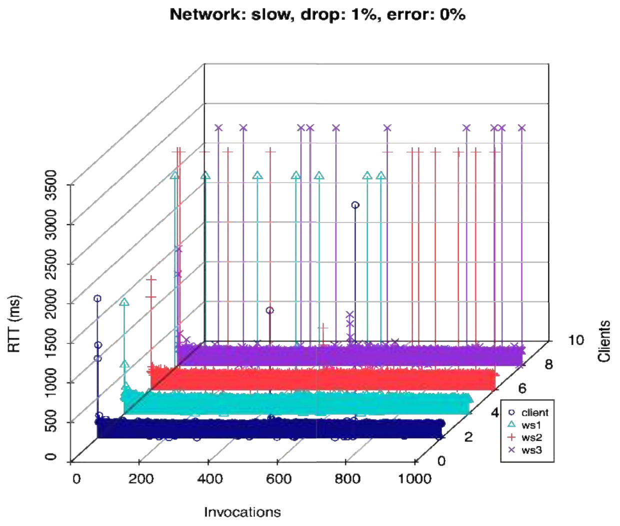


Figure 3. RTT to Web Service 1 for various network configurations

Figure 3 shows the RTT of Web Service 1 (monitored at the WS-Mediator client) at different drop and error rates and network conditions. The 'injection modes' axis represents the invocation Round Trip Time of each injection mode. As expected, 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 in the heterogeneous network than in the slow network. The timeout value has been bounded at 3000ms to make the plots more readable.

Figure 4 shows the comparison of the RTT of all three Web services where the fastest response message is counted. The 'clients' axis represents the invocation RTT monitored at each client thread (which respectively deals with WS1, WS2, WS3) and the client application that employs the WS-Mediator to deal with the results received by the client threads. We chose the 1% drop rate injection scenario to show the comparison since this test case affects the RTT most. As the faults were injected arbitrarily into the three Web services, the 3-version first response mechanism in the





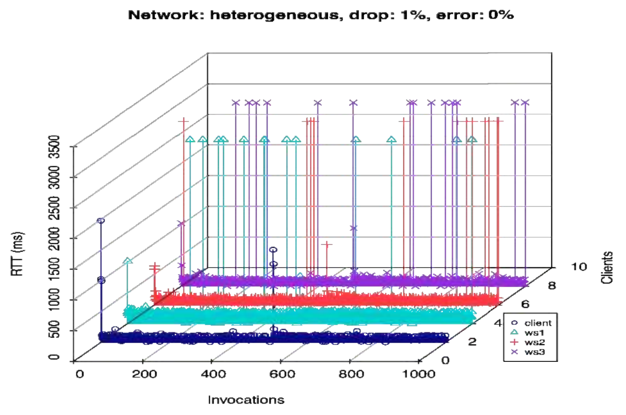


Figure 4: Round Trip Response times from the fastest of the 3 replicated web services under different network configurations

6. Conclusions and Future Work

In this paper, 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. The tool can inject value or timing faults into application. Furthermore, we can control the network that is used and can add background traffic. The network emulation may not exactly mirror the real world environment. However, it is a significant advance on testing using a single machine or a LAN. In particular, sample traffic recorded from a real network can be used in the emulation as well as self-similar traffic patterns. Our experiment has clearly demonstrated the network emulation and fault injection capacities of NetFIS and an example of how to use the functionalities of the tool for testing the Fault Tolerance Mechanisms of the application. In this case, the WS-Mediator has demonstrated its fault tolerance capacity with service diversity. The client application only threw exceptions when the three Web services failed simultaneously. WS-Mediator successfully dealt with the faults in most test cased and masked the reliability problems from the client.

7. References

[1] 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.

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

[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. Exploring 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 Evolution, 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. Eng. 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] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.