A Fault Injection Method for Testing Dependable Web Service Systems

Khaled Farj, Neil A Speirs and Yuhui Chen
A Fault Injection Method for Testing Dependable Web Service Systems

K. Farj, N.A. Speirs, Y. Chen

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.
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.

About the authors

Khaled Farj received his MSc in Computing Sciences from Newcastle University in 2004. Khaled is currently a PhD student working with Dr. Neil Speirs at Newcastle University. Khaled’s PhD studies is about a research in Web service dependability analysis.

Neil Speirs received a first class honours degree in Mathematics from the University of Newcastle upon Tyne in 1980. Between 1980 and 1982 he worked as a Systems Analyst and Computer Programmer for the Sage Group. Neil wrote the initial versions of many of Sage's well known commercial programs including the first Sage Payroll, Time Recording and Job Costing packages. He received a Ph.D. degree in Theoretical Physics in 1985 from the University of Durham for research work on Composite models of Weak Gauge Bosons. For 2 years Neil worked as a Consultant for Mari Advanced Microelectronics Ltd. He was the Project Leader of the Esprit projects Delta-4 and Concordia and a member of Delta-4 Project Technical Design Board. Delta-4 and Concordia were projects concerned with the design and development of fault tolerant distributed computing systems. Neil joined the Computing Science Department at the University of Newcastle upon Tyne as a lecturer in 1987. I was promoted to Senior Lecturer in 2002. My research interests lie in the area of fault tolerant distributed systems.

Yuhui Chen has just completed his PhD study at Newcastle University (UK). He received a MSc in Computing Science in 2003 from Newcastle University. He started his PhD from April 2004 under the supervision of Prof. Alexander Romanovsky. His research focuses on applying software fault-tolerance in the context of Web Services for improving the dependability of Web Services

Suggested keywords

FAULT INJECTION
TESTING WEB SERVICE
NETWORK EMULATION
A Fault Injection Method for Testing Dependable Web Service Systems

Khaled Farj
School of Computing Science,
University of Newcastle, 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,
University of Newcastle, 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 are becoming progressively more important in the world today. Web services are software programs that operate independently and which offer services over the Internet to other software programs, 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 their ability to solve integration problems in Internet applications. By using Web services, Internet applications can communicate with each other regardless of using different programming language or platforms.

Web services can be adopted to develop information systems through 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 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. Services are inherently distributed and heterogeneous, and are often invoked with little understanding of their reliability and their performance.

In these applications, service composition is typically dynamic so that services are discovered, selected, and composed, possibly 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 run over the Internet, there is no guarantee that all parts of the 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 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 assessing the reliability of a system [1]. In this paper we describe a tool for testing the performance and fault tolerance of either a single Web service or composed service, without any modification to the system 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 other 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 uses a technique termed Mediation [4] to provide fault tolerance. We evaluate the performance and the fault tolerance mechanisms of the service. Section 6 describes our conclusions and future work.
2. Related work

There are many Software Fault Injection tools for testing distributed systems in general and other tools for testing Web service systems in particular. Some well known fault injection tools, such as DOCTOR [5] and Orchestra [6] which both of them support network level fault injection and could potentially be used to inject faults into Web services, these two 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 systems. In [7] a tool is developed for generating and validating test cases. Tools start from the WSDL schema types and introduce some operator to generate a request with random data and a test script that manipulates the request parameters. In [8] a technique for testing Web services using mutation analysis is proposed. A mutant WSDL document is generated by applying mutant operators to the original WSDL document. A test tool called WSDLTest [9] tool generates Web service requests from the WSDL schema and tunes them in accordance with the pre-conditions written by the tester and verifies the response 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 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. WS-FIT tools [13] inject faults by modifying SOAP messages using scripts. The function parameters are modified by using the value boundaries specified by the tester. WSInject [14] tool inject both communication and interface faults and can be used to inject faults for testing a single Web service or composed service system.

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

In [16] faults are injected at the IP level to investigate the effect of retransmission mechanisms of TCP on Web services. This allows them to examine the relationship between TCP and WSRM time-out and retransmission mechanisms. By contrast, our tool 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 examine whether fault tolerant mechanisms at the application level can handle such faults.

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 which implements our fault injection method for testing Web service performance and FTMs. It lies between the Service Client and the Service Provider and requires the target system to be distributed in a modular fashion of services interacting via messages, so messages can be manipulated to emulate incorrect behavior of faulty services. It basically intercepts the request from the Service Client, provides a network emulator service and injecting appropriate fault (if any) and then forwards the request to the Service Provider. Similarly, it intercepts the response from the Service Provider, provides a network emulator and injects a fault (if any) and then forwards it to the Service Client. NetFIS gives Web service applications the sense of running over a WAN without any modification to the application. It requires no modifications to the underlying operating system, networking libraries or the Web service applications under test. It is able to emulate WAN behavior and injecting network faults such as dropping, delaying, randomly corrupting bytes of the body of exchanged 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 services parameters definitions (including data types) from WSDL files, however this class of faults will not be detailed in this paper.

The emulation is configurable and gives the ability to control every property of the target emulated network. The graphical user interface of the emulation also provides the ability to control the emulator at runtime 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 fault tolerance mechanisms can be measured and tested in a controlled emulation of a target WAN. As composed service systems can be subjected to network delays, errors, drops, reorders
and partitions, and also software faults, in this paper we restrict the use of our tool to emulate only drops, errors and delays to the messages exchanged between the system components in order to measure the performance and the fault tolerance applied to client side.

Before going through more details about the proposed system architecture, it is worthwhile to go over 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. This way the Web service systems are tested on virtual WANs that 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 [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, 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 affects 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 scale well for networks with hundreds or more of nodes while maintaining a limit on the overhead of the emulation. It is intended that the emulation is hosted over a local area network (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 wide area networks to be emulated are large enough that the overhead introduced by the emulation is negligible 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 failure modes affecting distributed systems have been classified. For example, in [15] failure modes which can be occurred in CORBA applications have been classified, and in [11] failure modes affecting Web service systems are also explained. Based on the above failure modes classifications, we have summarized 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 failure modes can be classified into one of the above major classes of 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 behavior. 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 signaled 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 accept 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. 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, 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 analyzing 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 affect 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 was 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 we describe in this paper is for SOAP based composed service systems, the architecture is generally applicable 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. It is also independent of the hosting environment for portability. Finally, it gives the applications the sense that there are other -synthetic- applications running at the same time and sharing the networking resources without a perceivable emulation overhead.

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 fault into the system under test by its sub-components.

The FIS role depends upon where it is deployed. At the client side, its role is generating 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 actual 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 services, where the service has to act as both a service and 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 java 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 message 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 script 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 of running over a LAN or WAN. It 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.). This work has been modified so that only Web service technology is used. 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

Figure 1. NetFIS Architecture
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 generated 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 start the FIS for every node which will generate a proxy service for each service needed to be called, and finally, 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 configuration. In addition, a trace file also must be provided for each node describes its traffic load. It shows packet counts per unit time and can be 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, 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. Example experiment

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 [18]. We deployed the WS-Mediator [4] at the client side to invoke three identical Bioinformatics Web services [18] 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 analyzed 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 2. 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.

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 [19], 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 [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. The 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.

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: This configuration represents the other extreme. 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 simulated traffic described in previous section, ranges between 10% 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 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.
5.1.2 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 3) and utilizes the built-in fault tolerance and logging mechanisms of the framework. The WS-Mediator claims to offer comprehensive off-the-shelf fault tolerance mechanisms to cope with various kinds of typical Web service application scenario. 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.

In our client application, we chose to use the N-version programming mechanism offered by the WS-Mediator to invoke the NetFIS proxies simultaneously and choose the first valid response to 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.

5.2 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 logs generated by the NetFIS and the client application have been used for quantitative result analysis.

Section 1: the 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 Web service before the 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. A system being tested by the tool

In Figure 3, a system being tested by the tool is shown. The 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 Web service before the 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 4. Client invocation RTT

Figure 4 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 a Web services and between different Web services are indeed insignificant compare 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 the NetFIS were added between the client and the Web services, and the client made 1000 invocations in each setting.

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 the 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 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).

<table>
<thead>
<tr>
<th>Network</th>
<th>Bandwidth (Mb/s)</th>
<th>Max, ms</th>
<th>Min, ms</th>
<th>Average, ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAN</td>
<td>N/A</td>
<td>102</td>
<td>8</td>
<td>57</td>
</tr>
<tr>
<td>Fast WAN</td>
<td>4000</td>
<td>488</td>
<td>35</td>
<td>59</td>
</tr>
<tr>
<td>Slow WAN</td>
<td>512</td>
<td>698</td>
<td>110</td>
<td>190</td>
</tr>
<tr>
<td>Hetero.WAN</td>
<td>Fast and Slow</td>
<td>870</td>
<td>99</td>
<td>104</td>
</tr>
</tbody>
</table>

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.
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 indicating the response was lost. When errors are injected, “Cannot find dispatch method for {http://webservices.callbayes.ncl.ac.uk/}getAvailableSimMethods” exception messages were thrown by the client indicating corrupted SOAP messages were received but the JAX-WS framework was unable to correctly deal with the responses. Figure 5 and Figure 6 show the plots of the results of some test cases, which clearly demonstrate the effectiveness of the tool. The tool simulates real work network conditions and faults to help on robust client application development (in this case, by applying the WS-Mediator).

**Section 2:** The NetFIS injects various types of faults between the client and the Web services in different emulated network conditions. The combinations of the injected faults are shown in Table 2. The client invoked the Web services via the NetFIS 1000 times in each setting.

**Table 2. DROP AND RANDOM ERROR INJECTED**

<table>
<thead>
<tr>
<th>Network Emulated</th>
<th>Injected Drop rate</th>
<th>Injected Error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Target %</td>
<td>Achieved (total messages)</td>
</tr>
<tr>
<td>Fast WAN</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>Slow WAN</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>Hetero.WAN</td>
<td>0.1</td>
<td>1</td>
</tr>
</tbody>
</table>

The figure shows the comparison of the RTT of the three Web services and the final responses delivered to the client by the WS-Mediator. 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 N-version programming fault tolerance mechanism in the WS-Mediator successfully dealt with the faults in most test cases and masked the reliability problems to the client. The client application only threw exceptions when the three Web services failed simultaneously.
6. Conclusions and Future Work

We have introduced a methodology and built a tool that can inject faults into any Web service application without touching the code of the application. There is great flexibility in the number and type of fault that can be injected and, furthermore, we can control the network that is used and can add background traffic.

The 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 from a real network can be used in the emulation as well as self-similar traffic patterns.

Our experiment has clearly demonstrated the network simulation and fault injection capacities of the NetFIS and an example of how to use the functionalities of the tool for testing the fault-tolerance mechanisms of the client application. In this case, the WS-Mediator has demonstrated its fault tolerance capacity with service diversity and dynamic service composition reconfiguration.

In the near future we are concentrating on two issues. Firstly, the tool does not have the capability to deal with Callback asynchronous invocations yet. In this stage dealing with Callback asynchronous invocations is left for future work as Dispatch has become a standard. Secondly, as the goal of WS-ReliableMessaging [21] is to allow applications to send and receive messages simply, reliably, and efficiently even in the face of application, platform, or network failure at middleware level, our future plan is to test our tools with WS-RM powered Web service systems.

7. References