SIMULATING ERRORS IN WEB SERVICES

Nik Looker,
Dept of Computer Science,
University of Durham, UK
n.e.looker@durham.ac.uk

Malcolm Munro,
Dept of Computer Science,
University of Durham, UK
malcolm.munro@durham.ac.uk

Jie Xu
School of Computing
University of Leeds, UK
jxu@comp.leeds.ac.uk

Abstract: This paper details our research into creating a method and tools to perform dependability analysis of Web Services. Our method is based upon a modified version of Network Level Fault Injection and extends this technique by automatically decoding network messages, based on SOAP, and thus allowing meaningful faults to be injected. This paper also outlines our method for automating the generation of test scripts from our fault model. Our method and tools can be applied to a system to assess a wide range of Quality of Service metrics and in this paper we present a test case to show how this can be applied to a Quality of Service scenario.

Keywords: Dependability, Reliability, Fault Model, Fault Injection, Web Services, Quality of Service.

1 INTRODUCTION

Web Service technology is a key factor in the success of any e-Science or e-Commerce project. All Web Services in these systems need to achieve a high Quality of Service (QoS) to allow the production of highly reliable systems. Our research into dependability assessment [Looker and Xu, 2003a, Looker and Xu, 2003b, Looker and Xu, 2003c] has produced a method and tools that allow dependability assessment of Web Service based systems.

Standards like SOAP, UDDI, and WSDL [Curbera, et al., 2002] are being rapidly adopted by all major Web Service providers, covering a wide range of applications: financial; commercial; scientific; etc. All Web Services need to establish and adhere to standards so QoS will become an important differentiating point of these services. This differentiation may become even more important in the future with technologies such as dynamic service composition being developed, with consumers having a choice of similar services.

Fault injection is a well-proven method of assessing the dependability of a system [Lyu, 1995]. Although much work has been done in the area of fault injection and distributed systems in general, there appears to have been little research carried out on applying this to Web Services.

Previous research in the field of middleware testing via fault injection has concentrated on tightly coupled, RPC based distributed systems [Marsden, et al., 2002]. In defining a testing method for Web Services a new set of challenges are faced which require different solutions.

Our work [Looker and Xu, 2003a] has shown a different method is required for Web Service systems since existing fault injection techniques are inadequate for testing this middleware technology. Whilst Network Level Fault Injection techniques can be useful for testing network loss and message corruption [Paxson, 1997] they are ineffective at testing a system in more detail. Furthermore traditional testing methods such as evolutionary testing suffer from problems introduced by state [McMinn and Holcombe, 2003] and formal proofs are inadequate in testing distributed applications and systems due to such complications as timing constraints, etc.

Our fault injection method is a modified version of Network Level Fault Injection. This method differs from standard Network Level Fault Injection techniques in that the fault injector decodes the SOAP messages and can inject meaningful faults into a message, rather than just injecting more or less random faults into a message. This enables API-level parameter value modification to be performed as well as standard Network Level Fault Injection.

We have shown how our fault injection method can be used to test dependability aspects of Web Services [Looker and Xu, 2003b]. In this paper we will demonstrate, with the aid of test cases, how our method and tools can be applied to a typical QoS scenario. We will demonstrate how our fault model based method can be used to automatically generate test cases and how our visualization tools can aid test case construction.

2 QUALITY OF SERVICE

Our method is primarily concerned with applying dependability analysis techniques to determine QoS. QoS covers a wide range of techniques that are combined to form metrics on the quality of a Web Service offered for by a system. It will be useful at this point to define all QoS terms that we will be using in this paper, although it is also worth emphasizing that this is not an exhaustive categorisation of terms.

Availability: the quality aspect of whether a Web Service is present and ready for use. This is represented as the probability that a Web Service will be available. This may be affected by such things as

time to complete a previous operation, loading on a particular service, etc.

Accessibility: the quality aspect that represents the degree the Web Service is capable of servicing a request. A service may be available but not accessible. For instance the initial request can be accepted but it cannot be processed due to some other dependency, for instance it may depend on another unavailable service. Accessibility can be improved by improving the scalability of a system.

Integrity: the quality of the Web Service maintaining the correctness of any interaction. If a transaction fails data should remain in a consistent state. This can be achieved through mechanisms such as atomic transactions, rollback mechanisms, etc.

Performance: the quality aspect that is defined in terms of the throughput of a Web Service and the latency. Throughput is the number of requests serviced in a given period and the latency is the time taken to service a request. The aim is to produce a high throughput but low latency system. Throughput and latency can be affected by such factors as processor speed, code efficiency, network transfer time, etc.

Reliability: the quality aspect that represents the capability of maintaining the service and service quality. One measurement of reliability is the number of failures during a given period. Another aspect of reliability refers to the probability that defines the dependability that a request is correctly sent and serviced.

Regulatory: the quality aspect that the service corresponds to rules, laws, standards and specifications. This can have an effect on areas such as availability, performance, and reliability through Service Level Agreements (SLA). SLA can define minimum levels service, such as percentage of requests to be serviced in a given time, and this in turn can affect other QoS criteria such as performance and reliability.

Security: the quality aspect that defines confidentiality for parties using a service. Again this can be influenced by regulatory factors. It can also affect performance due to the extra overhead incurred in implementing security mechanisms.

3 FAULT INJECTION

Previous research with regard to QoS of Web Services indicates that fault injection is a useful technique to perform dependability analysis and works well with Web Service RPC mechanisms such as JAX-RPC [Sun 2004] over Axis SOAP [Apache 2003].

Since our method and tools will be targeted at Web Services, a logical fault injection technique to use is Network Level Fault Injection since all RPC interaction will take place by message exchange across a network boundary.

3.1 Network Level Fault Injection

Network Level Fault Injection is concerned with the corruption, loss or reordering of network messages at the network interface. It is possible to use Software Implemented Fault Injection (SWIFI) tools to inject faults by instrumenting the operating system protocol stack as in [Dawson, et al., 1997] but this runs the risk of being detected and rejected by the receiving systems protocol stack. It is therefore preferable to inject the fault at the application level before this stage [Marsden, et al., 2002]. The faults injected are typically based on corrupting message header information and injecting random bit errors.

3.2 Perturbation

Perturbation [Voas, 1997] attempts to forcefully modify program states without mutating existing code statements. This is often achieved by code insertion [Hsueh, et al., 1997]. Instrumented code (termed perturbation functions) is added to a system in the form of function calls that modify internal program values. These modified values can either be generated based on the original value, generated randomly or a fixed constant value can be used.

This technique is useful in testing such things as fault tolerance mechanisms. As stated above Network Level Fault Injection is usually based upon more or less random corruption of bytes within a network message. We extend this technique to make meaningful perturbation to a network message, for instance WS-FIT can target a single parameter within an RPC message sequence. This perturbation of parameters is also less invasive that standard code insertion techniques since only small, centralized changes are made to the system.

3.3 Fault Injection Tools

A number of SWIFI Tools have been developed to perform a variety of different types of fault injection and perturbation. A brief summary of some of them is given here.

Ferrari [Kanawati, et al., 1992] (Fault and Error Automatic Real-Time Injection) uses a system based around software traps to inject errors into a system. The traps are activated by either a call to a specific memory location or a timeout. When a trap is called the handler injects a fault into the system. The fault can either be transient or permanent. Research conducted with Ferrari shows that error detection is dependant on the fault type and where the fault is inserted.

FTAPE [Tsai and Iyer, 1995] (Fault Tolerance and Performance Evaluator) can inject faults not only into memory and registers but into disk accesses as well. This is achieved by inserting a special disk driver into the system that can inject faults into data sent and received from the disk unit. FTAPE also has a synthetic load unit that can simulate specific amounts of load for robustness testing purposes.

Xception [Carreira, et al., 1998] is designed to take advantage of the advanced debugging features available on many modern processors. It is written to require no modification of system source and no insertion of software traps, since the processor's exception handling capabilities are used to trigger fault injection. These triggers are based around accesses to specific memory locations. Such accesses could be either for data or fetching instructions. It is therefore possible to accurately reproduce test runs because triggers can be tied to specific events, instead of timeouts. Research comparing it to static analysis methods has shown that it fails to detect some classes of faults (namely those contained in infrequently executed pieces of code) [Madeira, et al., 2000] but it did prove to be effective at detecting faults in frequently executed code.

DOCTOR [Han, et al., 1995] (IntegrateD sOftware Fault InjeCTiOn EnviRonment) allows injection of memory and register faults, as well as network communication faults. It uses a combination of time-out, trap and code modification. Time-out triggers are used to inject transient memory faults. Traps are used to inject transient emulated hardware failures, such as register corruption. Code modification is used to inject permanent faults.

Orchestra [Dawson, et al., 1996] is a script driven fault injector which is based around Network Level Fault Injection. Its primary use is the evaluation and validation of the fault-tolerance and timing characteristics of distributed protocols. ORCHESTRA was initially developed for the Mach Operating System and uses certain features of this platform to compensate for latencies introduced by the fault injector. It has also been successfully ported to other operating systems.

Whilst a number of these fault injectors could be used to perform state perturbation and Network Level Fault Injection, notably DOCTOR and Orchestra, the majority of these tools are designed for general purpose protocol testing or system testing.

The WS-FIT method and tools has been designed around an engine that automatically performs SOAP decoding and state perturbation within a message. The message information is presented ready decoded at the script API so it is easily to manipulate.

3.4 WS-FIT

Our WS-FIT (Web Service – Fault Injection Technology) tool and method has been developed specifically to perform Network Level Fault Injection and parameter perturbation on SOAP based RPC mechanisms. It has been implemented to test Web Services and a detailed description of its design is given in [Looker and Xu, 2003b]. A second package (OGSA-FIT) is designed using FIT and is implemented specifically to test OGSA based GRID services. The main use of this technology is envisaged as state perturbation through the targeted modification of RPC parameters within SOAP messages, which we have found especially useful in exercising fault tolerance mechanisms.

We use a variation of Network Level Fault Injection as a means of determining system dependability. Although we intend to inject faults into network messages we cannot do this directly because of the problems of altering encrypted/signed messages after they have been constructed.

The purpose of signing a message it to: a) identify the source of the message and, b) to ensure that the message is not intercepted and tampered with in transit. Since this is what we do to inject a fault, this will be discarded by the receiving transport layer and would consequently not relay the fault to the desired destination and test the desired domain.

Encryption is concerned with ensuring that the message cannot be read and tampered with whilst in transit. Consequently the message would be unreadable once it had been encrypted so anything other than injecting random faults into the message would be impossible, and thus would not allow us the fine level of message manipulation that we require.

We therefore inject the faults at the API boundary between the application and the top of the protocol stack, this being the lowest, easily accessible point to inject faults before any encryption and signing has taken place. It overcomes these two problems and allows us the level of control that we require.

WS-FIT uses an instrumented SOAP API that includes two small pieces of hook code. One hook intercepts outgoing messages, transmits them via a socket to the fault injector engine and receives a modified message from the fault injector. This message is then transmitted normally to the original destination. There is a similar hook for incoming messages, which can be processed in the same way (See Figure 1).

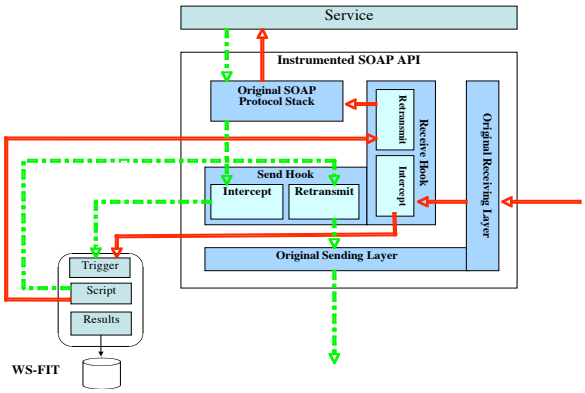

Figure 1: System Architecture

The WS-FIT method and tools implement this mechanism of fault injection and also provide a framework for the creation and execution of test cases.

The tool allows WSDL to be imported for the Web Services used in a system under test. The WSDL defines all interfaces and messages that flow between them to implement the RPC. Specification data can be added manually through the tool, for instance upper and lower bounds on parameters, but eventually our aim is to remove this manual step and import the information directly from specifications.

WS-FIT can use this information to decode an intercepted RPC and allow the user to construct triggers to fire on certain events. Test scripts can be constructed to run on the triggers and inject meaningful perturbations into RPC messages in much the same way that state perturbation functions operate using code insertion techniques.

The WS-FIT tool also provides facilities to capture RPC data and aid in the construction of test cases. There is a real time visualization facility, which allows RPC messages and parameters to be visualized in real-time. Finally there is a profiling facility to help in determining which Web Services and which methods to concentrate testing on.

4 FAULT MODEL

4.1 Fault Model for Web Services

The types of fault that can affect a Web Service can be classified as follows: 1) Physical Faults: effecting memory or processor registers, 2) Software Faults: both programming errors and design errors, 3) Resource-management faults: such as memory leakage and exhaustion of resource such as file descriptors, 4) Communication faults: such as message deletion, duplication, delay, reordering or corruption, (traditional distributed systems typically run over a LAN and it is assumed that the effect of this class of faults is negligible but Web Services will typically run over an internet which may be unreliable especially in message delivery times) and 5) Life-cycle faults: such as premature object destruction through starvation of keep-alive messages and delayed asynchronous responses.

Our previous research has concentrated on injecting communications faults [Looker and Xu, 2003a] to allow us to construct the fault injector framework and compare it to existing fault injectors, on targeted parameter perturbation [Looker, et al., 2004b], and on throughput/performance manipulation.

This paper primarily shows how our method and tools can be used to automatically generate test cases through the use of a generic, extendable fault model.

4.2 Extended Fault Model

We have enhanced our previous concept of a fault model [Looker and Xu, 2003c] and integrated it into our WS-FIT Tool. We can then use this enhanced fault model to automatically generate test scenarios.

We achieve this by functional decomposition of the top-level fault model into sub-sections and finally into concrete models that can be applied to individual parameters/messages. Structuring the fault model in this way aids in the design of the fault model and allows a more comprehensive model to be constructed. This is similar to functional decomposition used in such design methods as Yourdon [Yourdon, 1988], where three levels of decomposition are required by most projects to allow detailed designs and state machines to be constructed.

Communication Faults
- ***Duplication***
  - Request Messages
  - Response Messages
  - Fault Messages
- ***Omission***
  - Request Messages
  - Response Messages
  - Fault Messages
- ***Extra Messages (Attack Messages)***
  - Request Messages
  - Response Messages
  - Fault Messages
- ***Delay***
  - Messages within tolerances
    - *Inject a delay into the specified message that will be less that the maximum delay needed to cause a message timeout.*
  - Messages outside tolerances
  - Fault Messages inside tolerances
  - Fault Messages outside tolerances
- ***Ordering***
  - Response Message Before Request

Figure 2: Enhanced Fault Model for Message Manipulation

Our fault model uses a three level decomposition but this number could be increased of decreased as required. The levels are: 1) a general grouping, 2) a detailed grouping and there could be many of these under a top level heading, 3) defines the detailed method that can be used to generate tests cases.

An example is given in Figure 2. This shows only one section of the fault model defined in Section 4.1 decomposed down to its most detailed level.

From this detailed level it is possible to implement a model for each low level sub-section. The model will generate a script that can be run on a trigger. By making each model generic in nature and utilizing information from the WSDL and specifications it should be possible to apply this model to any parameter or message used in the system.

4.3 Failure Modes for Web Services

When applying our enhanced fault model to a system it is important to determine how the system will fail so that any defects in the system can be detected.

Here we will discuss the ways in which an Web Service can fail and the effects of each failure on the system as a whole: 1) crash of a service instance, 2)

crash of a hosting web server, 3) hang of a service, 4) corruption of data into middleware, 5) corruption of data out of middleware, 6) duplication of messages, 7) omission of messages, and 8) delay of messages.

The effect of each of these modes will depend on the fault tolerance of the system as a whole. A well-written Web Service should be able to detect and reject any corrupted data given to it and raise appropriate error responses. This is also true for duplication and omission of messages which should be handled by the middleware layer. The fault injector framework detailed in [Looker and Xu, 2003a] can detect these states via monitoring the SOAP message flow between various elements of the system.

Crashes of services/hosting environments should be detected via time-out mechanisms. For this experiment the fault injector framework has no means of checking for this. It will be addressed in later research.

More problematic is an operation that corrupts data leaving the middleware. When corrupt data is passed between Web Services internally (e.g. across a network boundary) this should be handled but when corrupt data is passed up to an application, a mechanism must be in place in the application to deal with this. The fault injector can detect this via monitoring the SOAP message exchange and detecting the invalid data being returned to the application.

Another problematic area is that of a service hang. Whilst this may superficially appear as a service crash it is indicative of corruption in the hosting environment. Since it may present the same symptoms to the application level it will be harder to detect. Differentiating between a service hang and a server crash will be left for future research.

Finally, delayed messages may cause errors due to message timeouts. These should be detected via a SOAP Fault message being sent (in cases of a Response message not being received) and the framework can detected this. Detecting a Request message being lost is more problematic since the Web Service will have no way of knowing the request has been sent.

It should be noted that WS-FIT will introduce latency into the message exchange that should be taken into account when constructing the test case. Previous research shows that under most circumstances this latency is minimal [Looker, et al., 2004a].

Our eventual aim is to automate failure mode detection in some way, and this will form the basis of future research. At present failure modes must be detected by manual inspection of data.

5 TEST CASE

This test case will demonstrate how the WS-FIT method and tools can be used to assess whether a Web Service based system can meet its timing constrains under extreme conditions. We will demonstrate the test case can be automatically generated from our extended fault model and this can be compared to previous experiments that we conducted on the same system using manual methods [Looker, et al., 2004a].

5.1 Scenario

Our test scenario will be based around an electronically controlled heater system. In our scenario a 'heater unit' is made up of a thermocouple, a heater coil and a control unit. The thermocouple and heater coil hardware are interfaced to two separate machines whilst the heater controller runs on a third machine.

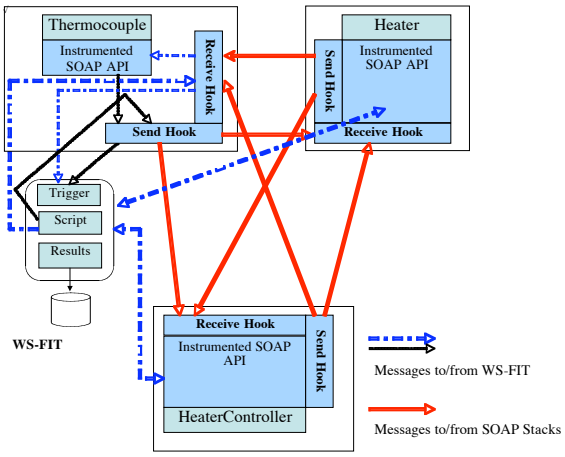

Figure 3: 'Heat Unit' Design

Driver functionality is provided by a Web Service running on each machine: *Thermocouple* to drive the thermocouple hardware; *Heater* to drive the heater coil hardware; and *HeaterController* to provide a coordination facility for both the *Thermocouple* and *Heater* Web Services. (See Figure 3)

With embedded smart devices becoming more prevalent this type of system may become more common in the future, and testing reliability and security aspects will be come crucial [Koopman, 2004]. This system could be constructed with three embedded microprocessor boards, running an embedded OS such as VxWorks and could conceivably be running a full web server connected to the outside world via an Internet connection for remote monitoring and control.

The *Heater* service allows power to be applied in small increments via two operations: *incPower* and *decPower*. The heater coil power increments are logarithmic in nature (See Figure 4), therefore if linear behaviour is required a control algorithm is required to provide this.

Thermocouple allows the current temperature to be read back via the *getTemp* operation. Since power to the heater coil can only be modified in small increments the *HeaterController* provides the *setTemp* operation that uses a time-based algorithm that issues

incPower and *decPower* operations to the *Heater* to set the correct power level. The current temperature is monitored by *HeaterController* to provide feedback into the algorithm. *HeaterController* also provides the current temperature to the client program via the *getTemp* operation.

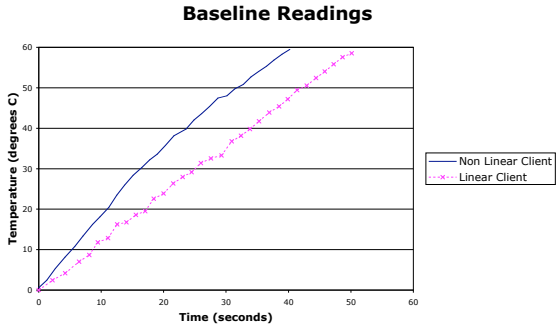

Figure 4: Baseline Readings

This 'heater unit' is to be used in a chemical process. A sample is to be heated over a precisely defined period with a precisely linear temperature rise to a temperature of 60°C. This constitutes the SLA for the *HeaterController*.

Figure 5 shows the modelled behaviour. A client program is to be used to provide this behaviour. This client will send the temperature every second to *HeaterController* via the *setTemp* operation. The *HeaterController* will then use its time-based algorithm (10 adjustments per second) to adjust the power the heater coil supplies. The small time steps used by the client should ensure that the temperature rise is linear.

This test scenario will demonstrate how WS-FIT can be used to modify latencies in a system. It will allow this to be achieved without the need for any additional test harness or test code.

Each machine in the system will be instrumented with a modified SOAP stack. This will allow us to, not only inject faults on any machine, but also capture and log all traffic to that server. This logged traffic can then be analyzed off line to determine latencies, etc.

By using the WS-FIT tool it is possible to introduce latencies into any RPC messages and also monitor messages sent/received. In this way we can assess the system and determine if the fault tolerance mechanisms included in the system are adequate, if the system is scalable, etc.

5.2 Base Line

Initially we will run the system with no fault injection triggers set. The system is executed for a length of time under normal conditions to determine baseline timings from the collected log. WS-FIT has the capability to visualize parameters in SOAP messages in real-time. The data from these visualizations can be output to file to allow offline processing.

We can assess the system by monitoring the *getTemp* operation (See Figure 5). This data shows that under normal operation, the system operates as required and the client algorithm effectively removes the logarithmic nature of the heater coil from the overall system. We can compare the operation of the system with the model to confirm this.

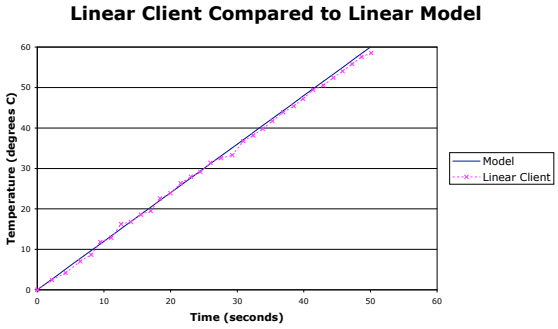

Figure 5: Model compared to Linear Client

An analysis of the log files generated by WS-FIT shows the frequency of *incPower* and *decPower* operations sent between the *HeaterController* and *Heater*. The maximum theoretical throughput of these messages is 10 per second. This data shows that the current system is running within this throughput constraint with an *incPower*/*decPower* message being sent on average every 1.07 seconds.

5.3 Test Case Generation

Previously our test cases have been constructed manually in a heuristic way, designed using experience to fit with our top-level fault model. This test case will demonstrate how the WS-FIT tool can be used to automatically generate a test script using our extended fault model.

Test case generation is based around a customizable fault model as detailed in Section 4.2 that can be used to provide automatic test case generation from a predefined set of alternatives. Our test case uses the scenario described in Section 5.1 but instead of devising a test script based on the Web Service design we will base our analysis purely on the Web Service specification which we will derive from the WSDL interface specifications and separately specified bounds for all parameters (both input and output parameters).

Our experiment will be designed to inject latency into the system at appropriate points. The injection trigger must be chosen with care since we are attempting to change the maximum throughput of the system.

To determine the effectiveness of a particular injection point we utilized the feature of WS-FIT which allows real-time visualization of RPC parameters. A monitor can be set on any RPC parameter contained in a SOAP message and these can be used to give a visual representation of the system as it is running. We can use this to determine that a test script is producing the desired results.

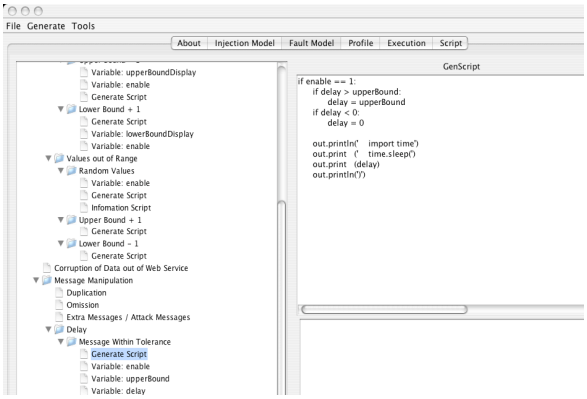

Figure 6: Constructing a Detailed Fault Model

Once we have determined which RPC to target a detailed model must be selected from our extended fault model. Since we are assuming no detailed knowledge of the implementation of the system, other than its specification, the detailed model takes general parameters and constructs a specific test case from these. If a suitable model wasn't present in our model at this point we could implement a new detailed test using our WS-FIT tools.

The fault model test is composed of a simple script which takes the various attributes associated with a WSDL defined message, for instance parameter name, parameter type, upper and lower bound, etc. and uses these to generate a piece of test code to be inserted into the main test script in the same way that a manually written script is inserted into the main script.

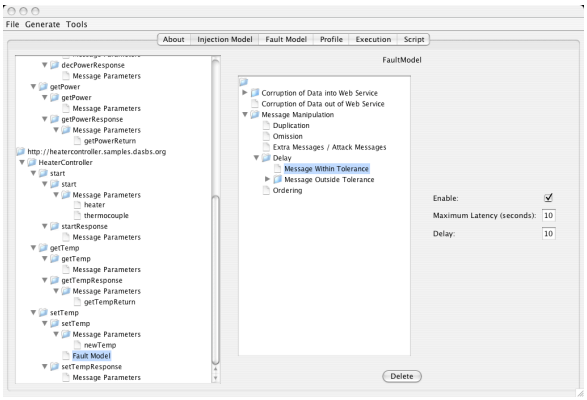

Figure 7: Applying a Fault Model to a Parameter

The fault model test doesn't actually run as part of the main test script, it mealy generates static code for the main test script. In this way it is possible to maintain a level of repeatability of testing, since any random values introduced should be statically encoded into the main test script and executed in subsequent runs.

5.4 Latency Injection

Since we are attempting to affect the throughput of the system our first injection point will introduce a latency into the *setTemp* operation sent between the client and *HeaterController*. The client sends a *setTemp* operation to the controller every second with the required temperature at that point in time. The *HeaterController* will then use a loop running in a thread to increment/decrement the *Heater*. The threading of this algorithm allows it to run asynchronously.

Our first test script was generated by adding a fault model to the *setTempRequest* message. From this fault model we selected the *Message Within Tolerance* model (See Figure 2). This detailed model requires the maximum permissible delay. This is the maximum delay that can be injected before the timeout mechanism causes the operation to be aborted. This model also requires the delay to be introduced which must be below the maximum delay.

Our first test introduces a 10 second delay into sending the *setTemp* operation to the *HeaterController*. This latency was chosen because it was large enough to be noticeable but less that the default timeout set by SOAP stack for the timeout of RPC operations. We used the visualization feature previously mentioned to monitor the thermocouple readings to determine if the test scenario had adversely affected the system.

In this case the temperature increase was not adversely effected. We postulate that this is due to the asynchronous nature of the *HeaterController*. Since the *HeaterController* is capable of increasing the power to the *Heater* independently, introducing the latency at this point caused the client to request larger temperature increases at a lower frequency rather than smaller temperature increases at a higher frequency. The simple fault tolerant nature of the *HeaterController* design allowed these requests to be correctly serviced. This is born out by the information contained in the log files once they were analyzed.

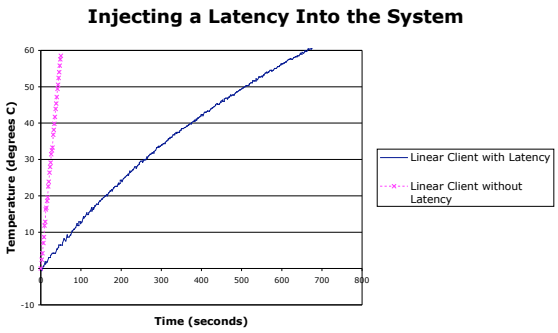

Figure 8: Injecting Latency into the System

Our second test introduced latency into the message exchanged between the *HeaterController* and the *Heater*. As explained previously the *HeaterController* runs in an asynchronous manner, largely independent of the other latencies introduced by other components. The *HeaterController* sends *incPower* and *decPower* operations to the *Heater* dependent on the temperature returned by the *Thermocouple* and the required temperature set by the *setTemp* operation.

Our script introduced a 1 second delay into sending each *incPower* and *decPower* message to the *Heater*. In this way we reduce the throughput of the *HeaterController* to *Thermocouple* message exchange, and thus alter the performance of the system.

Again we used the visualization facility provided by WS-FIT to monitor the results of this experiment. Our results are given in Figure 8 and clearly show that the SLA for this system hasn't been met. Further analysis of the log files confirms that the throughput of messages has been significantly reduced and hence the required rise in temperature can't be achieved.

6 CONCLUSIONS AND FUTURE WORK

This paper has demonstrated how Network Level Fault Injection can be used as an aid to dependability assessment of QoS constraints. We have shown how our tool, WS-FIT, can be used to achieve this. We have used our tool to perform a quantifiable experiment on a simple Web Service based system and used the results to assess the impact of unexpected latencies within the system.

We have further demonstrated how our extended fault model technique can be used to automatically generate test cases from WSDL definitions and specifications with comparable results to test cases generated manually. The user can also enhance the fault model through our WS-FIT tool.

Our future work will concentrate on three main areas. Firstly, we will conduct experiments on more complex systems. This will allow us to not only evaluate and enhance our method and tools further. It will also provide us with metrics on constructing test scripts and extending our fault model using Network Level Fault Injection techniques.

Secondly, we will examine and enhance our real-time RPC visualization method included in WS-FIT. Our preliminary experiments with visualization have provided promising results that indicate its potential usefulness.

Thirdly, we will investigate a method of automatically detecting failure modes. This may be based on a similar principle to that utilized by our enhanced fault model to create a failure model.

Whilst the method and techniques we have described in this paper provide practical dependability assessment tools for evaluating Web Services, they are only applicable to a certain restricted domain (for fault injection to occur an RPC exchange must be made). Whilst this is an important area of research there are a wide range of faults that cannot be injected at the network level, in particular faults cannot be injected into signed messages nor can they be injected into calls internal to the Web Service. We are looking at ways to apply our generic fault/failure model technique to cover these areas, possibly by applying our generic models to other fault injection techniques.

7 REFERENCES

Apache Axis Group, "Axis Architecture Guide," http://ws.apache.org/axis/, 2003.

J. Carreira, H. Madeira, and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering Se*, vol. 24, pp. 125-136, 1998.

F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, pp. 86-93, 2002.

S. Dawson, F. Jahanian, and T. Mitton, "ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations," presented at International computer performance and dependability symposium, Urbana-Champaign; IL, 1996.

S. Dawson, F. Jahanian, and T. Mitton, "Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool," *Software Practice and Experience*, vol. 27, pp. 1385-1410, 1997.

S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment for Distributed Real-time Systems," presented at International computer performance and dependability symposium, Erlangen; Germany, 1995.

M. C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, pp. 75-82, 1997.

G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A tool for validation of system dependability properties," presented at International Symposium on Fault-Tolerant Computing (FTCS'92), Boston, MA, 1992.

P. Koopman, "Embedded System Security," in *Computer*, vol. 37, 2004, pp. 95-97.

N. Looker, M. Munro, and J. Xu, "Assessing Web Service Quality of Service with Fault Injection," presented at Workshop on Quality of Service for Application Servers, SRDS, Brazil, 2004a.

N. Looker, M. Munro, and J. Xu, "Practical Dependability Analysis of SOAP Based Systems," presented at UK e-Science All Hands Meeting, Nottingham, 2004b.

N. Looker and J. Xu, "Assessing the Dependability of OGSA Middleware by Fault Injection," *Proceedings*

of the Symposium on Reliable Distributed Systems, pp. 293-302, 2003a.

N. Looker and J. Xu, "Assessing the Dependability of SOAP-RPC-Based Web Services by Fault Injection," *9th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, pp. 163-170, 2003b.

N. Looker and J. Xu, "Dependability Assessment of an OGSA Compliant Middleware Implementation by Fault Injection," presented at UK e-Science All Hands Meeting 2003, Nottingham, UK, 2003c.

M. R. Lyu, *Software fault tolerance*. Chichester ; New York: John Wiley, 1995.

H. Madeira, D. Costa, and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection," presented at Dependable systems and networks, New York, NY, 2000.

E. Marsden, J. Fabre, and J. Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection," presented at Symposium on reliable distributed systems, Osaka, Japan, 2002.

P. McMinn and M. Holcombe, "The State Problem for Evolutionary Testing," *Lecture Notes in Computer Science*, pp. 2488-2498, 2003.

V. Paxson, "End-to-End Internet Packet Dynamics," presented at ACM SIGCOMM'97 conference: applications, technologies, architectures, and protocols for computer communication, Cannes; France, 1997.

Sun Microsystems Ltd, "Java API for XML-Based RPC (JAX-RPC)," http://java.sun.com/xml/jaxrpc/index.jsp, 2004.

T. Tsai and R. Iyer, "FTAPE: A Fault Injection Tool to Measure Fault Tolerance," presented at Computing in aerospace, San Antonio; TX, 1995.

J. Voas, "Fault Injection for the Masses," *Computer*, vol. 30, pp. 129-130, 1997.

E. Yourdon, *Modern Structured Analysis*: Prentice Hall, 1988.

8 BIOGRAPHY

Nik Looker has worked in the fields of distributed systems, operating systems and embedded applications for over 15 years. He received his BSc (1993) from Oxford Brookes University, UK. He then left to pursue a career in industry focusing mainly on small footprint embedded mobile telecommunications systems and test equipment. In 1999 he moved to WindRiver Systems to work on both their embedded distributed products and also on their RTOS kernel (VxWorks). At WindRiver he made major contributions to implementing and testing several of their kernel releases. In 2002 he moved to the University of Durham to undertake research in middleware dependability analysis.

Malcolm Munro is a professor of software engineering at the Department of Computer Science at the University of Durham, United Kingdom. His research area is software engineering with a focus on how systems change and evolve over time; and how to develop and maintain systems using service oriented architectures. He has been actively involved with the IEEE International Conference on Software Maintenance and the International Workshop on Program Comprehension. He has led a number of UK EPSRC funded projects including GUSTT (Guided Slicing and Targeted Transformation), and Jigsaw (Distributed and dynamic visualisation generation); and been involved with the e-Demand project (A Demand-Led Service-Based Architecture for Dependable e-Science Applications).

Jie Xu (University of Leeds) is Professor of Computing and Director of the EPSRC WRG e-Science Centre. He is also a visiting professor at the School of Computing Science, the University of Newcastle upon Tyne. He has worked in the field of Distributed Systems Engineering for over twenty years and had industrial experience in building large-scale distributed applications. He received a PhD from the University of Newcastle upon Tyne, and moved to the University of Durham in 1998 as the head founder of the Durham Distributed Systems Engineering group. He became Professor of Distributed Systems at Durham before he joined the Multidisciplinary Informatics Institute at Leeds.

Professor Xu now leads a research team studying Grid technologies with a focus on complex system integration, dependable and secure collaboration, and evolving system architectures. He has published more than 130 edited books, book chapters and academic papers. His work in engineering dependable systems won the BCS/IEE Brendan Murphy Prize 2001. He is PI of the EPSRC/DTI e-Science Core Programme project: e-Demand. He also participates in two major EPSRC e-Science pilot projects GOLD and DAME. He is PI of the JISC fund for WR Grid, co-Leader of the EPSRC IBHIS project for distributed information integration and of the EPSRC Flexx project for software evolution. He has served as Editor of IEEE Distributed Systems Online since 2000. He is the Guest Editor of International Journal of Computer Systems Science and Engineering for a special issue on Dependable Computing, Program co-Chair of IEEE SRDS-23 on Reliable Distributed Systems, Program co-Chair of IEEE WORDS-8 on Object-Oriented Real-Time Dependable Systems, and Program co-Chair of IEEE WORDS-7.