A Comparison of Network Level Fault Injection with Code Insertion

Nik Looker, Malcolm Munro, Jie Xu,
Department of Computer Science, Department of Computer Science, School of Computing,
University of Durham, UK University of Durham, UK University of Leeds, UK

n.e.looker@durham.ac.uk malcolm.munro@durham.ac.uk jxu@comp.leeds.ac.uk

Abstract

This paper describes our research into the application of fault injection to Simple Object Access Protocol (SOAP) based Service Oriented-Architectures (SOA). We show that our previously devised WS-FIT method, when combined with parameter perturbation, gives comparable performance to Code Insertion techniques with the benefit that it is less invasive. Finally we demonstrate that this technique can be used to compliment certification testing of a production system by strategic instrumentation of selected servers in a system.

Keywords: Dependability, SWIFI, SOA, SOAP.

1. Introduction

Web Services form the basis, not only of standard web based eCommerce applications but also the Globus implementation of Open GRID Service Infrastructure. Given the prominence of this technology, test methods and tools are required to ensure that robust, fault tolerant software services are deployed. Testing is required not only to uncover existing problems with systems but also to provide users with metrics to compare similar service based solutions.

Fault Injection is a well-proven method of dependability assessment. 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 [1]. Previous research in the application of fault injection to distributed systems has concentrated on tightly coupled, Remote Procedure Call (RPC) based systems [2]. In defining an assessment method for SOAs a new sets of challenges are faced which require different solutions [1, 3]. Key differences that are encountered are: 1) greater chance for network failure; 2) higher levels of security and encryption; 3) more generic nature of the platform and the need to support multiple programming languages; 4) timing constrains and the asynchronous nature of Web Service operations. This is due to the loosely coupled nature of typical SOAs.

This paper demonstrates our WS-FIT method that applies fault injection to SOAP based SOA. We first review dependability assessment techniques and then outline our WS-FIT method. Finally we compare WS-FIT to a well-established method of fault injection (Code Insertion) and demonstrate that for the given domain WS-FIT can produce comparable results to Code Insertion whilst being less invasive and not requiring access to source code.

2. Dependability Assessment

System Dependability can be assessed using either model or measurement techniques [2]. Both techniques have their merits. Modeling can be used in the design stage to predict potential faults and errors in algorithms. Measurement can be applied to existing systems to provide metrics on dependability. Modeling can only make predictions of the dependability of a system since it is derived from system designs and specifications. Once a system has been implemented, actual measurement techniques can be applied to obtain specific metrics and allow data on dependability to be derived from them. Measurement techniques are useful because they can be applied to existing systems without requiring access to source code or design documentation. There are two main measurement techniques:

Observation: measurements are performed by the observation of failures in a large set of deployed systems. Existing logs are used and analysis of this data can obtain information on the frequency of failures and the activity that was in progress when they occurred. Since failures may occur infrequently it is unlikely that this technique will catch rarely seen errors [4] and data must be collected over a long period of time and from many systems.

Fault Injection: since it may take a very long time for some errors to occur fault injection attempts to speed up this process by injecting faults into a running system. There are a number of different methods of fault injection but all methods attempt to cause the
execution of seldom used control pathways within a system. By doing this either a failure will be observed or the system’s fault tolerance mechanism will handle the error.

There has been much research in developing Software Implemented Fault Injection (SWIFI) tools, rather than those based on Hardware Implemented Fault Injection. This is partly because a SWIFI tool does not require any expensive hardware. SWIFI also allows specific systems running on target hardware to be effectively targeted without injecting faults into other parts of the system.

SWIFI does have some drawbacks: 1) Faults cannot be injected into memory that is protected by the operating system; 2) Instrumentation of the code to be tested may perturb the operation of the system; 3) Timing of events may be inaccurate because the timers available to a software system on some hardware platforms may not have a high enough resolution to trigger short latency faults; 4) SWIFI may introduce additional latency into the system under test.

SWIFI techniques can be categorized into two types: Compile-Time Injection and Runtime Injection. For the purpose of this paper we treat Code Insertion as a third technique for reasons we will describe below.

2.1. Compile-Time Injection

Compile-Time Injection is an injection technique where source code is modified to inject simulated faults into a system. A simple example of this technique could be changing:

\[ x = x + 1 \text{ to } x = x - 1 \]

This technique has the advantage that it can be used to simulate both hardware and software faults. It has been shown to simulate faults in a system that are very close in nature to those produced by programming faults [5]. Since the faults are coded into the executables it is possible to emulate permanent faults as well as transient faults. Finally this system is very simple to implement.

The main drawbacks of this technique are that it requires the modification of the actual source code and therefore the source code must be available to the test team that may not be the case for COTS systems for example. Also since the code is being modified there is the chance that unintended faults will be introduced. Lastly as a consequence of the source code being altered this technique cannot be used as part of a certification processes since the system under test will be a different system to that which is shipped.

2.2. Code Insertion

Code Insertion involves inserting code into the target system source just before an event is to occur. This code performs the fault injection and then the original statement can execute with the fault present in the system. This method differs from compile-time injection in that it does not modify existing code but adds fault-generating code. There is still a static element of Compile-Time Injection since the code must be modified but it is added to rather than corrupted.

Its main advantages are that it is easy to implement and will run on processors which do not have the required debugging hardware to use other Runtime Injection techniques. It also requires no special access to protected areas of memory. Its main disadvantage is again that it requires the system source code to be modified making it hard to use this method for system certification.

2.3. Runtime Injection

Runtime Injection techniques use a software trigger to inject a fault into a running system. Faults can be injected via a number of techniques. Triggers can be implemented in a number of ways. The main types are:

- **Time based triggers**: When the timer reaches a specified time an interrupt is generated and the interrupt handler associated with the timer can inject a fault. Since this trigger method cannot be tied with any accuracy to specific operations it produces unpredictable effects in a system. Its main use is to simulate transient and intermittent faults within a system.

- **Interrupt based triggers**: These use hardware exceptions and the software trap mechanism to generate an interrupt at a specific place in the system code or on a particular event within the system, e.g. access to a specific memory location. This method of trigger implementation is capable of injecting a fault on a specific event and has the advantage that it requires no modification to the system code.

A large number of Runtime Injection mechanisms are based upon debugging features present in most modern processors [6]. This method of implementation has the advantage of providing low latency triggers and allowing fault injection tools to be written in a non-invasive way, but has the disadvantage that it ties a tool to a particular machine architecture.

2.4. Network Level Fault Injection

Network Level Fault Injection is a runtime technique concerned with the corruption, loss or reordering of network packets at the network interface. It is possible to use SWIFI tools to inject faults by instrumenting the operating system protocol stack as in Dawson et al [7] 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 [2]. The faults
injected are based on corrupting packet header information and injecting random byte errors.

3. WS-FIT

WS-FIT (Web Service – Fault Injection Technology) is a fault injector that allows Network Level Fault Injection to be used to test SOAP based SOA and a detailed description of its design is given in Looker et al [8]. The main use of this technology is envisaged as state perturbation through the targeted modification of RPC parameters within SOAP messages.

WS-FIT implements a variation of Network Level Fault Injection as a means of determining dependability. Although we intend to inject faults into network packets we cannot do this directly because of the problems of altering encrypted/signatured packets after they have been constructed.

The purpose of signing a packet is to: 1) identify the source of the packet; and 2) to ensure that the packet is not intercepted and tampered with in transit. Since this is effectively what we would do to inject a fault, this would 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 packet cannot be read and tampered with whilst in transit. Consequently the packet would be unreadable once it had been encrypted so anything other than injecting random faults into the packet would be impossible, and thus would not allow us the fine level of packet manipulation that we require.

We therefore inject 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. This overcomes these two problems and allows us the level of control we require. This allows us to use our method from previous research [1, 3] to inject targeted faults into specific parts of an RPC message, i.e. individual parameters.

3.1. State Perturbation

State perturbation [9] attempts to forcefully modify program states without mutating existing code statements. This is often achieved by Code Insertion. 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 be generated based on the original value, randomly or be a fixed constant.

This technique is useful in testing such things as fault tolerance mechanisms. As stated above Network Level Fault Injection is usually based upon the more or less random corruption of bytes within a packet. WS-FIT extends this method to make meaningful perturbation to a packet, e.g. our method can target a single parameter within an RPC message sequence.

3.2. WS-FIT Method

WS-FIT uses an instrumented SOAP stack 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 back 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: WS-FIT Architecture

By instrumenting the SOAP stacks on strategic machines this method can be used as part of the certification testing for individual components within a production system without the need for a test harness, as we will demonstrate in Section 4.

Whilst a number of existing fault injectors could be used to do this, notably DOCTOR [10] and Orchestra [11], these tools are designed for general purpose protocol testing. WS-FIT was designed around an engine to decode SOAP messages and presents an interface at the script API level with the information included in an RPC easily accessible.

4. Test Case

This test case demonstrates the value of WS-FIT in providing a framework for certification testing of Web Services. Our test case simulates a complex real-world scenario and we use the tools to locate a defect in the design of the SOA.

4.1. Scenario

Our system is based on a simulation of a self-regulating heating system (see Figure 2). The system is composed of three main elements: 1) a heater coil, 2) a
thermocouple and 3) a controller.

A driver to both the heater coil and the thermocouple is provided via a Web Service. These drivers are hosted on separate servers (In a real world application these could be embedded devices). The heater coil hardware is designed to allow only small stepped changes to the power. It also has an upper limit to its power output of 100°C, if set above this limit the coil will burn out.

Figure 2: SOA with WS-FIT Instrumentation

The controller is hosted on a third server. It allows a required temperature to be set. The Controller service runs a continuous polling loop that periodically polls the Thermocouple service to check that the actual temperature is equivalent to the required temperature. If it is not the Controller increments or decrements the power supplied by the HeaterCoil, thereby increasing or decreasing the temperature.

In our simulated system the Thermocouple requests the currently set power from the HeaterCoil and calculates the temperature based on this. In the real system this information would come from the thermocouple hardware.

A client is provided to exercise the system. A simple state machine is implemented by the client to first increase the temperature to 10°C, then decrease the temperature to 5°C and finally increase and hold the temperature at 7°C.

The test case is performed using two different configurations: 1) A WS-FIT instrumented system; 2) A system using Code Insertion. By comparing the results from these two configurations we demonstrate that WS-FIT can be used to produce compatible results to Code Insertion whilst being less invasive.

4.2. WS-FIT

As shown in Figure 1 a small amount of hook code must be installed on any server on which faults are to be injected. By strategically positioning this hook code on certain machines WS-FIT can be used as part of the certification process for individual components of a system. For example if the instrumented SOAP stack is positioned on the server running the HeaterCoil service it could be used to certification test the Thermocouple or Controller since no changes are made to these servers.

Our system is set up to certification test the HeaterCoil service so we have chosen to position the instrumented SOAP stack on the machine running the Thermocouple service (see Figure 2). In this way we can monitor the output of the Thermocouple driver and inject faults into the messages received from the HeaterCoil (without modifying the HeaterCoil code or environment).

A script is required to provide both a trigger for fault injection and also the fault injection itself as described in Looker et al [1]. The WS-FIT GUI can be used to create skeleton scripts by parsing the WSDL for a Web Service and allows the user to set triggers on messages and parameters. The user can then complete the test script by entering fragments of script to perform fault injection. The GUI can then generate a complete test script from this information.

We will construct a script to monitor temperature response messages between the Thermocouple and the Controller. We will also construct a trigger to inject a fault into the getPower responses received by the Thermocouple from the HeaterCoil after the temperature has reached a certain limit. By modifying this response to give a constantly low value we will attempt to force the controller to continually increase the power emitted by the heater coil, thus causing the heater coil to exceed its maximum power.

This configuration uses two test scripts. The first is a control script that passes all messages through unaltered to their destination. It is used to monitor SOAP messages. The second script is the one described above. While running a test script, the fault injector framework logs a variety of data including unmodified and modified messages.

4.3. Code Insertion

This configuration demonstrates that Code Insertion can produce similar results to those of WS-FIT. The original code for the services was taken and perturbation functions were inserted at appropriate points to perturb parameters in a similar way to those in Section 4.2.

Two points were identified for Code Insertion in this scenario but in practice with a complex SOA many more insertion points would potentially be used, for instance where RPC calls are called from multiple places in the code. Inserted code is marked in grey on Figure 3.

The first insertion point was in the getTemp routine in the Thermocouple service (See Figure 3). The second was a Controller routine where the
controller calls the Thermocouple getTemp routine. This insertion point was constructed to modify the value returned by the call to getPower of the HeaterCoil service. This corresponds to the getTempResponse SOAP message that was logged and modified in the WS-FIT configuration. As in the WS-FIT configuration the returned power was set to a constant value once it had reached a certain value, thus attempting to force the Controller service to continually increment the heater coil power. Both the original value and the modified value were logged.

The second insertion point was implemented to log the modified temperature to the system log on the server running the Controller service so a comparison of injected temperature to actual temperature could be made.

```java
private int inject1(int power) {
    int injectPower;
    if (power > 5) {
        injectPower = 5;
    } else {
        injectPower = power;
    }
    System.out.println(power + "," + injectPower);
    return injectPower;
}

public int getTemp(int ctx) throws java.rmi.RemoteException {
    HeaterCoilServiceLocator locator = new HeaterCoilServiceLocator();
    try {
        HeaterCoil service =
            locator.getHeaterCoil(new URL(
                getHeaterContext(ctx).getUrl()));
        return inject1(service.getPower(
            getHeaterContext(ctx).getCtx()));
    } catch (MalformedURLException e) {
        e.printStackTrace();
        throw new RemoteException(e.getMessage());
    } catch (ServiceException e) {
        e.printStackTrace();
        throw new RemoteException(e.getMessage());
    }
    return 0;
}
```

**Figure 3: Instrumented Thermocouple Routine**

To obtain data similar in form to that obtained from the WS-FIT configuration the two log files were combined via a simple shell script.

5. Analysis

We collected three series of data: 1) the control experiment (Figure 4), 2) the fault injection experiment using WS-FIT (Figure 5), and 3) the fault injection experiment using Code Insertion.

The control experiment was carried out using WS-FIT running a ‘null’ script that injected no faults but captured all messages received by and sent to the Thermocouple service. These messages were analyzed to give a temperature plot of the system when running under normal conditions. The data obtained from this experiment (Figure 4) indicates that the system functions according to the state machine given in 4.1. This experiment gave us a basis for comparison with the following fault injection experiments.

![Figure 4: Control Temperatures](image)

**Figure 4: Control Temperatures**

The following metrics were extracted from the logged data: 1) the temperature returned by the Thermocouple to the Controller, 2) the power reading sent by the HeaterCoil to the Thermocouple, and 3) the power reading supplied to the Thermocouple with a fault injected into it.

![Figure 5: Fault Injection Temperatures](image)

**Figure 5: Fault Injection Temperatures**

The logged data was converted into temperature graphs, one for the temperature returned by the Thermocouple and one for the actual temperature. The
actual temperature was extrapolated from the power reading sent by the HeaterCoil to the Thermocouple.

The control data shown in Figure 4 clearly demonstrates that the system is functioning according to specification with only random variations (introduced deliberately as part of the simulation).

The data returned by WS-FIT (Figure 5) shows a problem with the design of the SOA. Once a trigger condition has been met the fault injection modifies the power sent to the Thermocouple to a power that indicates a temperature of 1°C and holds at this temperature. The controller is written in a simple fashion. According to its criteria the temperature is too low so it keeps ramping the power to increase the temperature. The heater coil soon exceeds its maximum operating temperature and in a real system would malfunction. The Code Insertion configuration yielded results identical to those obtained using the WS-FIT configuration (see Figure 5).

The scenario used here could be caused, under real world conditions, by a thermocouple malfunctioning and thus causing an invalid reading to be received. It would indicate that some form of fault tolerance mechanism is required in the system, for instance a piece of guard code in the heater coil driver service.

6. Conclusions and Future Work

This paper has demonstrated our WS-FIT method and tools. We have demonstrated how our modified Network Level Fault Injection technique can be used as part of a specification based certification strategy. The comparison of WS-FIT with Code Insertion has indicated that it is less intrusive, requiring just one set of modifications to the network stack as opposed to many potential modifications for Code Insertion.

Our proposed method allows specific components within a SOA to be certification tested, provided that strategic decisions on instrumented SOAP stacks are taken based upon which components will be certification tested.

Finally Code Insertion requires access to the service source code to allow placement of extra code where as WS-FIT tests can be based entirely on the WSDL specification since it requires no modifications to the service code. Also Code Insertion may require instrumentation of the code in multiple places to capture or perturb parameters to a single method, for instance if a remote method is called from more than one place.

Our future research will concentrate on the following areas. Firstly we will conduct experiments on more complex systems. This will allow us, not only to evaluate and enhance WS-FIT further, but also provide us with more metrics on constructing test scripts using Network Level Fault Injection techniques. Secondly we will extend WS-FIT so that it can coordinate tests across multiple servers using WS-FIT as a centralized coordinator.

Finally we intend to devise an ontology and complementary method to capture heuristic knowledge from experienced testers so that it can be used to automatically generate fault injection test campaigns for WS-FIT.

This work was supported by the EPSRC through a DTA studentship.

7. References


