Dependability analysis of web services

Nik Looker

DEPENDABILITY ANALYSIS OF WEB SERVICES

by

Nik Looker

Submitted in conformity with the

requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Durham University

Dedication

This work is dedicated to my beautiful daughter Caitlin Erin Coxon who keeps me on my toes with more questions than I have answers, to my understanding wife Helenmarie who puts up with me sitting in front of a computer screen until the small hours of the morning, to Grant who should share the blame, to Dicky and the evenings we spent drinking scotch, to Cathy for putting up with a southerner, to Mum for looking after the dogs, to John for all the trips in boats when I was young, and to Rene for all the good times. I know I have it right this time.

And finally to Californian Highway 154 where six years ago I remembered that I'd always wanted to do this.

Abstract

Dependability Analysis of Web Services

Nik Looker

Web Services form the basis of web based eCommerce and eScience applications so it is vital that robust services are developed. Traditional validation and verification techniques are centred around the concept of removing all faults to guarantee correct operation whereas Dependability gives an assessment of how dependably a system can deliver the required functionality by assessing attributes, and by eliminating threats via means attempts to improve dependability.

Fault injection is a well-proven dependability assessment method. 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 middleware systems and Web Services in particular. There are additional problems associated with applying existing fault injection technologies to Web Services running in a virtual machine environment since most are either invasive or work at a machine level.

The Fault Injection Technology (FIT) method has been devised to address these problems for middleware systems. The Web Service-Fault Injection Technology (WS-FIT) implementation applies the FIT method, based on network level fault injection, to Web Services to create a non-invasive dependability assessment method. It allows targeted perturbation of Web Service RPC parameters as well as more traditional network level fault injection operations. The WS-FIT tool includes taxonomies that define a system under test, fault models to apply and failure modes to be detected, and uses these taxonomies to generate fault injection campaigns.

WS-FIT has been applied to a number of case studies and has successfully demonstrated its effectiveness. It has also been successfully applied to a third-party system to evaluate dependability means. It performed this dependability assessment as well as allowing debugging of the means to be undertaken uncovering unknown faults.

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Durham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

This work has been documented in part in the following publications:

N. Looker, J. Xu, and M. Munro, "Determining the Dependability of Service-Oriented Architectures," International Journal of Simulation and Process Modelling, 2007 (accepted and awaiting publication).

N. Looker, L. Burd, S. Drummond, M. Munro "Pedagogic Data as a Basis for Web Service Fault Models," presented at IEEE International Workshop on Service-Oriented System Engineering, Beijing, China, October 20-21, 2005.

N. Looker, M. Munro, J. Xu "A Comparison of Network Level Fault Injection with Code Insertion," presented at the 29th Annual International Computer Software and Applications Conference, Edinburgh, Scotland, July 25-28, 2005.

N. Looker, J. Xu, M. Munro "Increasing Web Service Dependability Through Consensus Voting," presented at the 2nd International Workshop on Quality Assurance and Testing of Web-Based Applications, COMPSAC, Edinburgh, Scotland, July 25-28, 2005.

N. Looker, B. Gwynne, J.Xu, M. Munro "An Ontology-Based Approach for Determining the Dependability of Service-Oriented Architectures ," 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems, Sedona, USA, 2005.

P. Townend, P. Groth, N. Looker, and J. Xu, "FT-Grid: A Fault-Tolerance System for e-Science," presented at UK e-Science Programme All Hands Meeting, UK, 2005.

P. Townend, J. Xu, E. Yang, K. Bennett, S. Charters, N. Holliman, N. Looker, and M. Munro, "The e-Demand project: A Summary," presented at UK e-Science All Hands Meeting, UK, 2005.

N. Looker and M. Munro, 'WS-FTM: A Fault Tolerance Mechanism for Web Services,' University of Durham, Technical Report 02/05, 19th March 2005.

N. Looker, M. Munro, J. Xu, "Simulating Errors in Web Services," International Journal of Simulation: Systems, Science & Technology, vol. 5, 2004.

N. Looker, M. Munro, and J. Xu, 'WS-FIT: A Tool for Dependability Analysis of Web Services,' 1st Workshop on Quality Assurance and Testing of Web-Based Applications, COMPSAC, Hong Kong, 28-30 Sep 2004.

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, 2004.

N. Looker, M. Munro and J. Xu, 'Testing Web Services'. The 16th IFIP International Conference on Testing of Communicating Systems, Oxford, 2004.

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

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, 2003.

N. Looker and J. Xu, 'Assessing the Dependability of OGSA Middleware by Fault Injection,' Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems, pp. 293-302, 2003.

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, 2003.

N. Looker and J. Xu, 'Assessing the Dependability of OGSA Middleware by Fault Injection,' University of Durham, Technical Report 01/03, 27/04/03.

Copyright Notice

Acknowledgements

This thesis would not appear in its present form without the kind assistance and support of the following individuals and organizations:

Professor Jie Xu, The University of Leeds, who acted as my supervisor for the first year of my research and then as my external advisor and provided invaluable advice and guidance.

Professor Malcolm Munro, Durham University, who acted as my supervisor for the final three years of my research and provided invaluable advice and guidance.

Dr. Liz Burd, Durham University, for arranging my initial entry into research and for support and advice during my studies.

This research started as a self funded project, and during the second and third years was funded by the EPSRC through a doctoral training award. The fourth and final year was completed whilst working as a Teaching Fellow on the HEFCE funded ALiC project and then as a Research Fellow on the EPSRC/BAE funded NECTISE project.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

Web Services [23] are a heavily used middleware technology in both eCommerce and eScience applications. Web Services also make a major contribution to the Globus Toolkit [28]that is the current front running Grid technology. Given the prominence of this technology it is vital that methods are developed to ensure that dependable software services are deployed.

Dependability [8] is a discipline that provides an assessment of how much trust can be placed on a service to deliver its specified function. Conversely validation and verification [60] techniques attempt to determine that a system contains no faults, which is an important discipline and increases the overall reliability of a system, but is difficult to do with current techniques due to such problems as state explosion. Dependability is a more realistic approach to assessing a system since it measures the reliance that can be placed upon a service rather than validating it against its specification and includes methods that increase this reliance. Indeed Dependability embraces validation and verification as a fault prevention means [8].

Dependability analysis is therefore vital to aid in increasing the dependability of a system, not only to uncover existing problems with services but to also provide potential users with metrics to compare similar serviced based solutions.

A Web Service is a software service defined by a number of standards that can be used to provide interoperable data exchange and processing between dissimilar machines and architectures. A Web Service by definition has to exchange data in a machine independent form and the most popular form currently in use is SOAP [15]. Currently Web Services provide both synchronous and asynchronous models but this

research will concentrate on the synchronous RPC mechanism [13] defined by the W3C. There are two reasons for this:

1. It is a commonly used model
2. The coupling of request and response messages simplifies the detection of incorrect results in this initial work

The IFIP Working Group on Dependable Computing and Fault Tolerance defines dependability in [39] as "...the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers...". This definition does not mention the system being fault free, just that it should be able to deliver the specified service when needed. This is because it is virtually impossible to engineer a system that can be guaranteed to be fault free.

1.1 Dependability

Validation and verification techniques commonly employed to ensure that systems are fit for use attempt to remove all faults so that error conditions cannot occur. Since it is not feasible to verify all states a system can achieve it is not possible to completely test a system using validation and verification techniques. This is commonly called the state explosion problem. Further, this technique can fail to take into account the failure of third party items such as operating systems, support software such as databases, and the hardware itself.

Dependability takes a different approach. To understand dependability it is important to understand the three main concepts that it utilises:

Attributes: Measurements of how Dependable a system is

Threats: Things that may affect the Dependability of a system

Means: Ways of increasing the Dependability of a system.

Attributes are measures of how dependable a system is. Some attributes are quantifiable, for instance you can put a number to them, whilst others are not. For instance it is very hard to put a simple numeric value to Confidentiality but relatively easy to measure reliability. Dependability is therefore quantified using a loose classification of: Dependable, Very Dependable or Highly Dependable. There is a judgement applied to the classification of the system.

Threats are affects on a system that lower its dependability. Here we encounter the terms fault, error and failure.

Fault: A fault is a defect in a system.

Error: An error is an unexpected state within a system boundary.

Failure: A failure is an instance in time when a system displays behaviour that is contrary to its specification.

Faults, Errors and Failures are created according to a mechanism. This mechanism is sometimes known as a Fault-Error-Failure chain [9]. As a general rule a fault, when activated, can lead to an error (which is an invalid state) and the error may lead to another error or a failure (which is an observable deviation from the specified behaviour at the system boundary).

The purpose of Dependability means is to break fault-error-failure chains and thereby increase the Dependability of a system. There are four means of improving the

dependability of a system:

- Fault Prevention
- Fault Removal
- Fault Forecasting.
- Fault Tolerance

Therefore Dependability is measured using dependability attributes. It assumes that threats can exist in a system and employs means to break fault-error-failure chains and thus improve the overall dependability of the system. Since these means employ techniques that assume that not all faults are removed from the system it is a more realistic approach than verification and validation since it is currently not possible to guarantee all faults are removed from a system.

Further it is a more appropriate way to assess a Service based system since loosely coupled service based systems may include third party services which have been validated by that third party. There may be a lack of trust between the party constructing the service-based system and the service provider such that the validation results may not be trusted. Dependability therefore allows means other than verification and validation to be used, for instance fault tolerance, to increase the dependability of the un-trusted service.

1.2 Service Environment

There are a number of problems associated with applying existing fault injection technologies to Web Services. Currently Web Services are predominantly written in Java to run in a Java Virtual Machine. Unfortunately most fault injection tools are

engineered to operate on the physical machine rather than on virtual machines and would therefore not allow tight correlation between trigger events and elements of a Web Service affected, in effect the fault injection would affect the environment the Web Service was running in rather than the Web Service itself. This is similar in effect to comparing hardware implemented fault injection and software implemented fault injection and its application to an operating system [30].

In loosely coupled systems the Services that are selected may not have been assessed as part of the constructed system or, in systems that employ ultra late binding [11], may change from invocation to invocation. This implies that the assessment technique used could be run on individual services as part of the system binding. Further a third party who may not supply the source code could supply these Services. This precludes the use of some compile-time injection techniques or detailed knowledge of the service other than that supplied by the service description and accompanying meta-data. Indeed it precludes any fault injector that requires physical access to the machine hosting the Service that may be another trust issue with the service provider.

1.3 Objectives

Fault injection is a well-proven method of assessing the dependability of a system through exercising dependability means. Although much work has been done in the area of fault injection and distributed systems in general, there appears to have been little research with regard to the application of fault injection techniques to middleware products [52].

Most dependability assessment of middleware appears to have been conducted using observational measurement techniques [41]. Some research has been conducted using

fault injection to test the dependability of CORBA implementations using network level fault injection [52, 53], again with promising results.

Previous research in the field of service assessment via fault injection has concentrated on tightly coupled, binary protocol, RPC based distributed systems. In defining an assessment method for Web Services new sets of challenges are faced which require different solutions and are non-trivial to address using existing methods and tools. Key differences that are encountered when testing Web Services are:

1. Greater chance for network failure
2. Higher levels of security and encryption
3. More generic nature of the platform and possible virtualization of environment
4. Timing constrains of Web Service operations
5. Lack of access to physical system
6. Lack of source code

This thesis intends to utilise network level fault injection, which has been successfully applied to assess CORBA based middleware, to construct a method and tools to provide dependability assessment of Web Service systems. In particular this method will be concerned with points 2 and 3 because these are qualities of particular importance to Web Services. Traditional fault injection techniques can be used to cover points 1 and 4 although the method will also be capable of assessing these.

This method should be as non-invasive as possible and be capable of dependability analysis without access to the Web Service source code to cover points 5 and 6. The

method will also allow easy creation of fault injection campaigns though the use of an accompanying tool and provide an analysis of the results obtained. The method should be applicable to both the development phase, for example fault removal means, as well as during the testing phase to access all dependability means. Given this coverage it should also be possible to use the method and tool to assess functionally equivalent Web Services to give dependability metrics that can be used to compare and select competing Web Services.

Since middleware messages run over the physical network interface as sequences of packets, for instance one message may be split across multiple network packets, the fault injector will work at a message level rather than a packet level so it can manipulate complete middleware messages.

1.4 Criteria For Success

Given the objectives defined above a number of criteria to measure the success of the work have been defined. These are as follows:

Method: Devise a method based on network level fault injection to perform dependability testing of Web Services. This method should be comparable to other code insertion methods but with the added benefit of minimal alteration to code.

Tool: Construct a tool for use with the method. This tool will be tailored to injecting faults into SOAP packets and will handle the decoding of SOAP packets so that lightweight scripts can be written by the user to implement test cases without the complexity of decoding SOAP packets.

Test Case Construction Method: Devise a method to construct test cases for the method given above. This method should be devised to allow easy automation so that it can be incorporated into the tool.

Analysis Method: Devise an analysis method to assess dependability of result sets generated by the method. This analysis method should be applicable not only to the test phase of system development but also to the development stage.

Applicable to both development and testing phases: The methods and tools should be applicable, not only to the testing phase of a project but also to the development phase. The method should also be able to test systems without access to source code.

1.5 Major Contributions

There are three major contributions of this thesis:

1. Fault injection at a middleware message level rather than at a network message level. This allows fault injection operations to be preformed on an entire middleware message and affect a specific middleware service rather than test error recovery in a network protocol stack.

2. Runtime decoding of each middleware message combined with information on message formats derived from interface definition language definitions of a service to allow targeted triggering on specific messages corresponding to specific operations. Further, given this information, specific parameters can be perturbed in an RPC exchange allowing network level fault injection to be used to perform API level injection similar to Code Insertion techniques.

3. Non-invasive manipulation of a service via instrumentation of surrounding machines which is comparable to Code Insertion but without physical access to the service source code or the machine.

1.6 Thesis structure

This thesis is structured into three main parts.

1. A literature survey examines Web Service middleware (Chapter 2) and Dependability (Chapter 3) and examines the problems involved with applying Dependability Assessment to Web Services
2. A method (Chapter 4) and implementation (Chapter 5) are described which address the problems described in the literature review
3. A series of case studies are used to demonstrate the implementation (Chapter 6) and conclusions are made based on this (Chapter 7).

Chapter 2 - Service Terminology and Technology

This chapter will define the concepts and technology of services that is specifically relevant to this thesis. It will explore a general definition of a service and then explore how services are utilized via the use of Software-Oriented Architectures and implemented as Web Services.

2.1 Services

A service in economics and marketing terminology is defined by Boone and Kurtz [14] as *"... intangible tasks that satisfy both business and consumer needs."*. This definition originated to describe activities in the service industry such as hotels, garages, barbers, etc. In general terms a service is not owned by the customer but is something that is utilised to complete a task. The advantage of utilising a service to do this is that the customer does not have to design, maintain or run the service. This definition can readily be adapted to software services, which is how the term will be used for the remained of this thesis.

A service in computing terms is an entity that communicates with other entities via messages [18]. This definition does not specify that the services be networked or the method in which messages should be exchanged. Further it does not specify that the entities must perform tasks or satisfy a specified requirement but this is usually taken as implied.

Software services can be characterised as follows:

1) They must communicate over a network, in the case of Web Services this is usually an internet utilising HTTP as the transport protocol

2) They must provide an API that can be called by external programs to invoke service functions

3) A service should be discoverable in someway so that external programs can utilise the service. This is usually done by registering the service in a registry which can be searched by programs searching for services

4) Services should be loosely coupled which allows composed systems to be adaptable, for instance a system can be recomposed to utilise another service if needed without affecting the function of the system as a whole. These characteristics are explored in greater depth in the following sections.

Services are often used to implement client/server architectures. A client/service Architecture [66] (also called two-tier architectures) are systems that are composed of a client, which utilises a service provided by a server. The client accesses the service on the server to perform some task. A client may also be a service itself, thus allowing services to be composed from other services.

A client may be a thin client that needs to do little more than utilize services on servers to perform its task, with most, if not all business logic located on the server side. Conversely a thick client may implement its business logic local and utilize services only to perform specific tasks. Therefore the utilized services would not necessarily know any of the business logic.

Tasks provided by servers can be varied, for instance file storage, database, complex calculation, etc. In the context of Web Services a server may offer a number of services through a number of defined interfaces.

For the purposes of this thesis, when considering service composition, it will be assumed that a client can also be a service and use the term client rather than service of a service.

Since services must communicate with each other it is useful to have a model that defines this communication pattern. One of the first models defined and still one of the most used is the Remote Procedure Call (RPC) [12]. This model allows the implementation details of executing a routine on a remote machine to be hidden from the client program in such a way as the invocation looks like a normal routine call. This is done by hiding the implementation detail of the message exchange in a stub routine. A normal RPC invocation would consist of a request message being sent to the remote server, the requested processing being executed, and a response message being sent back by the server containing the result to the client program. Since data formats on clients and servers may vary according to machine architecture data must be converted into an agreed format for transfer between machines and this is known as marshalling.

To utilize a service the data required and the format of the message the service requires must be known. A commonly used technique for achieving this without having to implement this by hand on a service-by-service basis is to use an Interface Definition Language (IDL) [75]. This is a type of language used to define an interface to a service. An interface is a collection of accessor methods, and possibly required data structures, necessary to utilise the service. IDLs tend to be system specific, for instance DCOM IDL [1] and CORBA IDL [2].

While working at an abstract level the above is all that is required to implement a set of services and exchange messages between them, indeed if a common platform and common hardware is used the above is also adequate. Unfortunately most distributed systems rarely run on identical hardware and are frequently required to communicate with legacy systems and other organizations hardware/software. To overcome this problem and allow service-based systems to be constructed middleware is used.

Middleware [73] is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to communicate across a network. Middleware eliminates differences between machines in a heterogeneous environment by marshalling data and includes an agreed set of useful functions.

2.2 Processing Models

For a third party to offer a software service to a customer there is usually an implied need for the service to operate on a separate processor.

Once a number of processors are available it is possible to execute code in ways other than a simple sequential flow of control. The two main ways to achieve this are Parallel Processing and Distributed Processing and this section will give an overview of these two processing models.

Parallel Processing [62] is an approach that allows processing to be shared out between a number of CPUs so that parts of a system can be executed at the same time. This approach originated on arrays of multiple processors linked by a bus. Such systems usually have access to common shared memory.

It is also possible to implement an approach similar to parallel processing using a number of processors linked together by a computer network and this is termed

distributed processing. Although these two techniques share aspects in common they are two distinct programming models since distributed systems do not share memory, network operations are orders of magnitude slower that bus transfers and distributed systems tend to be geographically distributed as well. Further distributed systems can be independently owned so there can be trust issues involved in distributed processing.

Services allow distributed processing to be undertaken in two different ways. Firstly, a Service can be invoked by asynchronous messaging rather than by a more traditional RPC mechanism. This allows a client to continue processing whilst the Service (hosted on a different machine) processes the request. When the Service has processed the request an asynchronous response message can be sent back if required.

Secondly, if a synchronous RPC mechanism is being used parallel processing is still inherently present in the system if a number of Services are in use by a number of clients since RPC requests are sent to different Services at different times by different clients. This means that at any one time different servers are servicing requests in parallel. This is the model that will be examined in this thesis.

2.3 Service-Oriented Architectures

Distributed services must be organised into a system to be useful and a common architecture used to accomplish this is a Service-Oriented Architecture (SOA). SOA is an architecture that represents software functionality as discoverable services on a network. Channabasavaiah et al [21] define a SOA as

> *"an application architecture within which all functions are defined as independent services with well-defined invokable interfaces, which can be called in defined sequences to form business processes."*

The main principles are not new, for instance CORBA provides functionality by offering functions as components [2], but SOAs provide a number of advantages such as loose coupling and late binding. However, these advantages come with potential problems not least in the area of dependability. Given the widespread usage of these technologies methods for increasing the reliability of SOA is greatly desired.

Loose coupling [10] is the capability of services to be composed and utilized on demand, possibly using different system technologies, to create a working system. This can be accomplished using a combination of Dynamic composition [54], which is the composing of systems from existing services that are discovered at runtime, and late binding. Loose coupling implies that a service is referentially transparent from any data or other services that it requires. From a dependability point of view this makes it impossible to assess a complete system since each time the system is composed it may be composed of different services. It can also have an impact on the integrity of the system since the integrity of the services used may be in question, for instance you may bind to a malicious service.

Late Binding [10, 11] is the property of a system which allows a system to bind to a service at runtime, rather than at compile time. A new provider of a service can make a description of a new service available at runtime and an existing client can then utilize the new implementation without modification of it's code. This method implies that the client is written in such a way as to accommodate the utilization of new services in some way rather than just dynamically composing itself from already known service definitions.

Whilst late binding is a very useful facility and essential for the construction of loosely coupled systems it means that new services can potentially be deployed which

have not been assessed with the combination of services being used by an SOA, and as described above the integrity of the service could be called into question.

A directory service [32] can be used to supply an SOA with a service discovery function. A directory service is a service that keeps track of the location of available services. When a service becomes available it registers itself with a directory service. The directory service is sent the description of the service and its location. Once registered a client, or another service, can query the directory service to obtain this information and thus utilize the registered service. This process is known as service discovery.

A directory service also has the potential to provide alternate services that match a requested description, for instance if a directory service has two entries for an identical service on different servers it could transparently provide either service entry to a requesting call. This can be leveraged by such dependability means as n-voter algorithms and multi-version diversity to provide functionally equivalent services.

A common directory services is Universal Description, Discovery and Integration (UDDI) [4]. This provides a way to publish and discover web services. The UDDI server is logically centralized but physically replicated to provide fault tolerance. Once registered a service is effectively published to the whole network and is automatically replicated on all servers

Dependability is a key factor for SOAs. Many traditional distributed systems performing business-to-business (B2B) [32] operations, for instance those in the banking domain, perform computations that require very little execution time but the impact of incorrect results can have far reaching financial consequences. Conversely,

scientific Grid applications often perform tasks that require many days to complete so any failures during this time can have a considerable impact in terms of time and hence indirectly to costs thorough man hours lost. Dependability is therefore essential for both types of SOA to reduce the risk of failures.

The cost and difficulty of containing and recovering from faults in service-based applications may be higher than that for normal applications [46] whilst the heterogeneous nature of services within an SOA means that many service-based applications will be functioning in environments where interaction faults, operational faults caused by a system interacting with another system within the environment, are more likely to occur. Dependability means are therefore an advantage in these situations since they can be used to mitigate faults without the need to remove them.

Research in providing dependable SOAs falls into two main disciplines [29]:

- Increasing the Mean Time To Failure (MTTF).
- Reducing the Mean Time To Recover (MTTR).

These concepts will be discussed in more detail in Chapter 3.

2.4 Web Services

A Web Service is software service defined by a number of standards that can be used to provide interoperable data exchange and processing between dissimilar machines and architectures. For the purposes of this research it is concerned with Web Services defined by the W3C that are described by WSDL [22] and implemented using SOAP [15] and the RPC model.

Web Services are commonly used to provide the 'building blocks' of SOA and, as

such, any dependability assessment method that targets them will be of wide use, not only to Web Services but also to Globus Grid services, which utilize the same technology.

2.5 Extensible Markup Language

Both WSDL and SOAP utilize Extensible Markup Language (XML) to define and implement Web Service message exchange. XML notation and terminology is used frequently in this thesis and the following covers the basic terms used.

XML is a standard for document markup [16, 36]. It provides a document layout that allows a document to be self-describing and portable thus allowing data transfer between dissimilar systems. Its portability is largely due to it being an ASCII format document, with numeric values encoded as strings. Since it is portable it largely eliminates the need for marshalling and unmarshalling of data but this overhead is replaced by the need to construct XML documents and parse them at the receiving machine so no net gain is obtained by eliminating marshalling.

An XML document is structured from a number of Elements. An Element is composed of a Start Tag, an End Tag and Element Content. The document can have only one root element, with all other elements nested within it. An example XML document is given in Table 2-1 and this shows the basic structure and format of a typical XML document with examples of all the major features.

Table 2-1: An example XML Document

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<root>
    <element1>
        <element2 attr1="some value" attr2="some other value">
            Some element content and some &lt;escaped&gt; characters
            <element4 />
        </element2>
        <element5>
            <![CDATA[
                A CDATA section can contain unescaped characters like
                <>& etc...
            ]]>
        </element5>
    </element1>
    <element6 />
</root>
```

A start tag consists of a tag name and an optional number of attributes, for instance

```
<element2 attr1="some value" attr2="some other value">
```

The Start Tag and attribute names can be any user-defined name which may be made up from standard English letters and digits, non-English letters and ideograms as well as certain punctuation characters. The End Tag for an element must be the Start Tag name preceded by a solidus.

Element Content can be a mixture of free format text and Elements. Free format text can take one of two forms:

1. Normal characters and escaped special characters such as greater than and less than symbols
2. CDATA that is unescaped free format text enclosed in a CDATA tag.

Escaped symbols consist mainly of symbols that appear in the syntax of element tags, such as great than or less than symbols.

Namespaces are a way of grouping sets of elements and attributes together and an example is given in Table 2-2. They have two main uses:

1. Distinguish between elements and attributes from different vocabularies that share the same names
2. Group together related elements and attributes in an XML document so that applications can easily recognize them

Table 2-2: Example XML using namespaces

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <soapenv:Body>
    <ns1:test1
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                xmlns:ns1="http://test1.testsuite.samples.wsfit.org">
      <p1 xsi:type="xsd:int">0</p1>
    </ns1:test1>
  </soapenv:Body>
</soapenv:Envelope>
```

XML is extremely flexible and can be used to represent a large variety of data but most programs that utilize XML are not designed to be this flexible and are only concerned with a sub-set of XML elements. XML can be limited by defining a Document Type Definition (DTD). A DTD can be used to define which elements are permissible and the circumstances that these elements can be used under. XML parsers can validate an XML document against a DTD and determine if it is valid according to the syntax defined by the DTD.

DTDs allow the basic structure and syntax of an XML document to be defined and validated. A DTD cannot be used to limit such things as data types used within an element. For this purpose an XML Schema must be used. Schemas can be used to

describe and enforce complex restrictions on an XML document, such as type information for contents and complex data types.

It is possible to check an XML document to see if it follows the XML syntax rules and this process is called validation. Validation is performed by parsing the XML document through a parser. This is a piece of software designed to read and validate XML against a set of restrictions defined in a DTD or schema.

2.6 WSDL

Web Services Description Language (WSDL) is an XML-based IDL used to define Web Services interfaces and how to access them [22, 23]. Our research is mainly concerned with RPC message exchanges and WSDL lends itself well to providing explicit information on the structure of message exchanges between Web Services and their clients.

Table 2-3 shows an example WSDL description of a message. A WSDL message description is composed of an element that has a unique name attribute that is used to identify the message and a number of part elements. Each part defines a parameter or return value in the case of a response message. A part has an associated name that must be unique within the message element and a type that defines the parameter type. There are a number of predefined types and complex types can also be defined using a types element.

Table 2-3: Example WSDL Description of a Message

```
<wsdl:message name="unregisterServiceRequest">
      <wsdl:part name="context" type="xsd:string"/>
      <wsdl:part name="entry" type="impl:ServiceEntry"/>
</wsdl:message>
```

Once all request and response messages required to implement the RPC style interface have been defined they can be used to define the calling interface for a Web Service. This is done by use of the portType element (see Table 2-4). A portType contains a number of operation elements with each operation element corresponding to a method of the Web Service. Each operation is made up of an input element that will be the request part of the RPC and an output element that will be the response part of the RPC.

Table 2-4: Example WSDL PortType

```
<wsdl:portType name="Register">
      <wsdl:operation name="registerService"
                      parameterOrder="context entry">
         <wsdl:input name="registerServiceRequest"
                     message="impl:registerServiceRequest"/>
         <wsdl:output name="registerServiceResponse"
                      message="impl:registerServiceResponse"/>
      </wsdl:operation>
      <wsdl:operation name="unregisterService"
                      parameterOrder="context entry">
         <wsdl:input name="unregisterServiceRequest"
                     message="impl:unregisterServiceRequest"/>
         <wsdl:output name="unregisterServiceResponse"
                      message="impl:unregisterServiceResponse"/>
      </wsdl:operation>
   </wsdl:portType>
```

The above explanation briefly describes the use of WSDL to define a classic RPC style Web Service. WSDL can also be used to describe other styles of Web Service calling interface such as message-oriented calling but this is outside the scope of this research.

2.7 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) [15, 23] is a messaging protocol designed to allow the exchange of messages over a network. It is XML based to allow the exchange

of messages between dissimilar machines. It is extensively used to implement Web Services so it has been chosen as the middleware protocol to examine.

The research conducted in this thesis is primarily concerned with the RPC mechanism over SOAP. This is defined by the W3C in [33] and describes a general purpose RPC mechanism. The message types that are involved in an RPC exchange and the relevant features used by the WS-FIT method are briefly reviewed here.

A SOAP message utilizes the http://schemas.xmlsoap.org/soap/envelope/ schema which defines the namespace soapenv and this namespace is setup in the soapenv:Envelope element. Consequently all elements that unitise this namespace must be enclosed by the soapenv:Envelope element. The soapenv namespace defines a semantic framework for SOAP messages.

The body, or payload, of the SOAP message is enclosed by the soapenv:Body element. This element acts as a grouping for the body elements for different types of messages. Its primary function is to keep the body elements distinct from other grouping of elements, for instance a header block.

These two elements form the basis of a SOAP message. The soapenv:Body element is then populated with elements that make up the payload of a request, response of fault message.

A typical request message is given in Table 2-5. The request message name is defined in the wsdl:operation (see Section 2.6) but by convention the name of the message equates to the service method name but it can be defined as any valid name. In the example the message and method name are getQuote. The message element is therefore ns1:getQuote. The namespace ns1 is defined to be the urn of the service that implements

the method. If this is combined with the address of the server hosting the service this allows a specific method of a specific service on a specific server to be identified.

Table 2-5: Typical Request Message

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:getQuote
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:ns1="http://quote.stock.samples.dasbs.org">
   <symbol xsi:type="xsd:string">foo</symbol>
  </ns1:getQuote>
 </soapenv:Body>
</soapenv:Envelope>
```

The ns1:getQuote element contains parameter elements that represent the RPC parameters, for instance the getQuote method takes one string parameter called symbol so ns1:getQuote contains one element with an element tag symbol which contains the string data for that parameter. Parameters are defined in WSDL by wsdl:part elements in wsdl:message elements (see Section 2.6).

A typical response message is given in Table 2-6. The response message is similar in structure to the request message but by convention the response message name is post-fixed with the word Response although again it can be any valid name defined in the wsdl:operation element. In this example the response element name is ns1:getQuoteResponse.

A response element contains elements that represent any method return value and any parameters that are marked to be marshalled in-out or out. Method return results follow the naming convention of the method name post fixed by the word Return and are

represented in the WSDL by wsdl:part elements. In-out and out parameters follow the same conventions as parameters in a request message.

The example response message in Table 2-6 also demonstrates the format that objects and arrays are marshalled in a SOAP message. This utilizes the multiRef element. Each object or item in an array is created using a multiRef that has an id. The actual parameter/return value is then set to this reference id and the complex data can then be constructed within the multiRef element in the same way that individual parameters are constructed in a message. This technique applies to both request and response messages.

Table 2-6: Typical Response Message

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
 <soapenv:Body>
  <ns1:getQuoteResponse

soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                  xmlns:ns1="http://quote.stock.samples.dasbs.org">
   <getQuoteReturn href="#id0"/>
  </ns1:getQuoteResponse>
  <multiRef id="id0"
            soapenc:root="0"

soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xsi:type="ns2:QuoteValue"
            xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:ns2="http://quote.stock.samples.dasbs.org">
   <date xsi:type="xsd:dateTime">2004 10 30T10:54:18.511Z</date>
   <quote xsi:type="xsd:double">47.5</quote>
  </multiRef>
 </soapenv:Body>
</soapenv:Envelope>
```

Table 2-7 shows a typical SOAP Fault Message. Fault Messages are used to return failure information from a server to a client. The soapenv:Fault element contains three elements: faultcode, faultstring and detail. These elements are used to convey failure information to the client with the faultstring and detail elements being language specific, for instance using Axis 1.1 for Java if a piece of user code on a server throws a

Java exception the faultcode is set to soapenv:Server.userException to indicate that the fault originates in server side user code, then the fault string is set to the text description of exception and the detail element is not used.

Table 2-7: Typical Fault Message

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
 <soapenv:Body>
  <soapenv:Fault>
   <faultcode>soapenv:Server.userException</faultcode>
   <faultstring>
    java.rmi.RemoteException: can&amp;apos;t get a stock price for this
symbol
   </faultstring>
   <detail/>
  </soapenv:Fault>
 </soapenv:Body>
</soapenv:Envelope>
```

2.8 Summary

This chapter has reviewed the concepts, problems and commonly used technology for implementing SOA. This review has concentrated on SOA implemented using Web Services since this is currently the predominant technology. It has highlighted that, because of the duration and financial consequences of failures of SOAs, dependability is a key factor. It has also highlighted that dynamically constructed, loosely coupled SOA are particularly hard to assess because:

1. The combination of services used may vary each time the SOA is constructed
2. New services may become available that have not already been assessed within the SOA.

Since SOAs composed in this manner may not have been assessed using a particular combination of services this implies that the level of trust in the SOA will be reduced

since interaction faults may be present that have not been detected. Also since services may be included in the SOA without being assessed at all this may lead to a decreases in the integrity of the SOA.

SOA constructed from Web services utilise XML based protocols to exchange information. An RPC model allows a request to be coupled with a response, which allows easier error detection when injecting faults. Combining this with interfaces defined in WSDL, another XML based technology, then an easily decodable system definition and message exchange that can be easily manipulated is obtained.

Chapter 3 - Dependability Terminology and Fault Injection

This chapter reviews and gives a consistent definition of the field of Dependability since, particularly in the area of Quality of Service, there is variation amongst the meaning of some of the terms used, for instance contrast Avizienis et al [8] with Somerville [60]. This chapter attempts to give a standard definition of all relevant terms and they are taken to be definitive throughout this thesis.

3.1 A Definition of Quality of Service for Web Services

When assessing the reliability of a system it is useful to have some agreed measurement to determine the quality of a system. A commonly used measurement is Quality of Service (QoS) [60]. QoS is commonly used in networking where it is defined by Steinmetz et al [63] as

> *"...a concept for specifying how 'good' the offered networking services are. QoS can be characterised by a number of specific parameters."*

When applied to Web Services this definition has to be slightly modified. Haas et al [34] define Quality of Service as

> *"...an obligation accepted and advertised by a provider entity to service consumers."*

where an obligation is defined as

> *"...a kind of policy that prescribes actions and/or states of an agent and/or resource"*

The factors that go to make up the obligation cover a wide range of factors that are combined to define the QoS offered by a system but the following factors are commonly used [51] and are taken as definitive when used in this work:

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 at a specific time. This may be affected by such things as time to complete a pervious operation, loading on a particular service, etc.

Accessibility: the quality aspect that represents the degree the Web Service is capable of serving a request and a specific point in time. This is different from Availability since 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, so that the request would be queued awaiting a response from the 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 distributed commit [67], rollback mechanisms [65], etc. This also encompasses the concept of malicious tampering with a service.

Performance: the quality aspect that is defined in terms of the throughput of a Web Service and the latency. Throughput is defined asthe 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 [59].

Regulatory: the quality aspect that the service corresponds to rules, laws, standards and specifications. This can have an affect on areas such as availability, performance, and reliability through Service Level Agreements (SLA). SLA can define minimum levels of performance expected by a service that set levels for its dependability.

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

Some attributes can be quantitatively measured and others remain harder to quantify, for example Reliability can be measured by failures over time but the effectives of Regulatory cannot be measured by any simple means (See Table 3-1)

Table 3-1: Quantifiable QoS Attributes

Attribute	Quantifiable
Availability	Yes
Accessibility	Yes
Integrity	No
Performance	Yes
Reliability	Yes
Regulatory	No
Security	No

3.2 Dependability

Quality of Service attempts to give a measurement of the over all quality of a service and utilises some of the same attributes present in Dependability, whilst Dependability is concerned not only with measuring the Dependability of a system but also with means to improve the Dependability of the System.

The IFIP Working Group on Dependable Computing and Fault Tolerance [39] defines dependability as

> *"The notion of dependability, defined as the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers, enables these various concerns to be subsumed within a single conceptual framework."*

A number of factors affect the dependability of a system [8] but for the purpose of our research we are interested in factors that can be used to assess dependability via fault injection [74]. Dependability can be assessed and increased by the use of three things:

1. A way to assess the Dependability of a system
2. An understanding of the things that can affect the Dependability of a system
3. Ways to increase the Dependability of a system.

These things are known respectively as:

1. Attributes
2. Threats
3. Means

3.2.1 Attributes

Attributes are measurements that can be applied to a system to determine its overall dependability. A generally agreed list of Attributes is:

1. Availability - the probability that a service is present and ready for use
2. Reliability - the capability of maintaining the service and service quality
3. Safety - the absence of catastrophic consequences
4. Confidentiality - information is accessible only to those authorised to use it
5. Integrity - the absence of improper system alterations
6. Maintainability - to undergo modifications and repairs

As with QoS some attributes are quantifiable by direct measurements whilst others are more subjective, for instance Safety cannot be measured directly via metrics but is a subjective assessment that requires judgmental information to be applied to give a level of confidence, whereas Reliability can be quantified by physical metrics (See Table 3-2). The method devised in this work will focus on the quantifiable aspects of dependability, for instance Availability and Reliability.

Table 3-2: Quantifiable Dependability Attributes

Attribute	Quantifiable
Availability	Yes
Reliability	Yes
Safety	No
Confidentiality	No
Integrity	No
Maintainability	No

Security is sometimes classed as an attribute [60] but the current view is to aggregate it together with dependability and treat it as a composite term called Dependability and Security [8]. The reasoning behind this is that a dependable system must also be secure since otherwise its Integrity and Confidentiality could not be guaranteed. In this thesis the term Dependability will be assumed to be the composite definition of Dependability and Security even if not explicitly stated.

3.2.2 Threats

Threats are things that can affect a system and cause a drop in Dependability. There are three main terms that must be clearly understood:

Fault - A fault (which is usually referred to as a bug for historic reasons) is a defect in a system. The presence of a fault in a system may or may not lead to a failure, for instance although a system may contain a fault its input and state conditions may never cause this fault to be executed so that an error occurs and thus never exhibits as a failure.

Error - An error is a discrepancy between the intended behaviour of a system and its actual behaviour inside the system boundary. Errors occur at runtime when some part of the system enters an unexpected state due to the activation of a fault.

Since errors are generated from invalid states they are hard to observe without special mechanisms, such as debuggers or debug output to logs.

Failure - A failure is an instance in time when a system displays behaviour that is contrary to its specification. An error may not necessarily cause a failure, for instance an exception may be thrown by a system but this may be caught and handled using fault tolerance techniques so the overall operation of the system will conform to the specification.

It is important to note that Failures are measured at the system boundary. They are basically Errors that have propagated to the system boundary and have become observable.

Faults, Errors and Failures operate according to a mechanism. This mechanism is sometimes known as a Fault-Error-Failure chain [9]. As a general rule a fault, when activated, can lead to an error (which is an invalid state) and the invalid state generated by an error may lead to another error or a failure (which is an observable deviation from the specified behaviour at the system boundary).

Once a fault is activated an error is created. An error may act in the same way as a fault in that it can create further error conditions, therefore an error may propagate multiple times within a system boundary without causing an observable failure. If an error propagates outside the system boundary a failure is said to occur. A failure is basically the point at which it can be said that a service is failing to meet its specification. Since the output data from one service may be feed into another, a failure in one service may propagate into another service as a fault so a chain can be formed of the form: Fault leading to Error leading to Failure leading to Error, etc.

3.2.3 Means

Since the mechanism of a Fault-Error-Chain is understood it is possible to construct means to break these chains and thereby increase the dependability of a system.

There are four means of improving the dependability of a system:

1. Prevention
2. Removal
3. Forecasting
4. Tolerance

Fault Prevention deals with preventing faults being incorporated into a system. This can be accomplished by use of development methodologies, and good implementation techniques.

Fault Removal can be sub-divided into two sub-categories:

1. Removal During Development
2. Removal During Use

Removal during development requires verification so that faults can be detected and removed before a system is put into production. Once systems have been put into production a system is needed to record failures and remove them via a maintenance cycle.

Fault Forecasting predicts likely faults so that they can be removed or their effects can be circumvented.

Fault Tolerance [49] deals with putting mechanisms in place that will allow a system to function in the presence of faults but still deliver the required service, although that service may be at a degraded level.

Dependability means are intended to reduce the number of failures presented to the user of a system. Failures are traditionally measured over time and it is useful to understand how they are measured so that the effectiveness of means can be assessed.

Failure rate is a frequently used measurement that denotes the frequency with which failures occur and is represented by the Greek letter λ. When used in relation to software it is often expressed as failures per hour but any measurement of time can be used in place of hours, indeed the units need not be based on time but based on any measurable period, for instance revolutions and miles are commonly used in general engineering.

For mechanical systems it is common for the failure rate to increase during the lifetime of a component due, for instance due to mechanical wear, so this makes failure rate time dependent. Software systems, on the other hand, are often described as having no wear over time so failure rate can be considered constant. Whilst this is generally true there are factors which can cause failure rates to increase over time, for instance hardware failure of the environment running the software, but it is widely accepted as an approximation [37]. In this special case the failure rate is defined as the reciprocal of the Mean Time Between Failure (MTBF).

$$MTBF = \frac{1}{\lambda}$$

or

$$\theta = \frac{1}{\lambda}$$

Thus it is the mean number of failures per hour (if the failure rate is measured in hours). There are variations on MTBF that break it down into discrete components.

Mean Time To Failure (MTTF) is often used to denote the mean time for a failure to occur when a system is replaced in some way after the failure, for instance a new system is swapped in or the fault is fixed. This term does not include the time it takes to rectify the failure or replace the faulty component. This time is usually described as the Mean Time To Recovery (MTTR). So MTBF can be described as:

$$\theta = MTTF + MTTR$$

The MTTF can be used to improve the dependability of a system, for instance by making the MTTF very long, failures become infrequent and thus the reliability of the system is increased. Time and financial considerations may make this approach hard to achieve in practice so a different approach may be taken. The MTTR is small it may be acceptable to have a shorter MTTF since the system will recover quickly from a failure so failures can occur more frequently.

A long MTTF can be an issue when a assessing and debugging a system since it means that in normal operation:

1. Faults will take a long time to manifest themselves as failures, making fault removal problematic
2. Fault tolerance means may not be exercised during development, which means that they will remain unproved until the execution of a fault in a production system causes their use.

The second point can easily be assessed using fault injection since faults can be injected in such as way as to exercise fault tolerance mechanisms.

3.2.4 Fault Tolerance

Fault tolerance is a specific Dependability Means that can directly improve the Dependability of a system and, since it acts to break fault-error-failure chains, it can be evaluated using fault injection. It therefore makes a good case study for this thesis. The function of fault-tolerance has been described by Avizienis [7] as

> *"...to preserve the delivery of expected services despite the presence of fault-caused errors within the system itself. Errors are detected and corrected, and permanent faults are located and removed while the system continues to deliver acceptable service."*

There are many mechanisms available to implement fault tolerance in software systems but a popular approach when seeking to tolerate both development faults and physical machine failures is that of design diversity, which can be defined as the production of two or more systems aimed at delivering the same service through separate designs and realizations.

Redundancy has long been used as a means of increasing the availability of distributed systems, with key components being replicated to protected against hardware failure [47]. Redundancy can be achieved either through:

1. Hardware modules can be replicated to provide backup capacity when a failure occurs
2. Redundancy can be achieved using software solutions to replicate key elements of a business process.

Redundancy can be dynamic or static. Both use replication but in static redundancy, all replicas are active at the same time. If one replica fails another replica can be used

immediately with little impact on response times. Dynamic redundancy has one replica active at one time and others are kept in a standby state. If the active replica fails, a previously inactive replica must be initialized and take over operations. This type of replication is more often used in hardware replication.

Redundancy has many attractions as a technique for providing fault tolerance in a system but it also incurs a number of overheads in its use. These include:

1. The cost of replicating calls to replicas
2. An increase in the complexity of system design
3. The cost of providing and maintaining the replicas

Once redundancy has been introduced into a system it can also be used to protect against Byzantine faults through the use of the N-version model [57]. Most hardware failures result from either physical defects sustained during manufacture or develop over time as components wear out. This is not the case with software failures since software does not wear out. Software failures result from the invocation of paths that contain faults. Since software is typically more complex than hardware it can be expected to contain many more faults resulting in more failures.

The N-version model is a design pattern for implementing software fault tolerance. Physical faults, such as machine failure, can be handled by redundancy. Simply replicating a single software component n times may not eliminate a particular failure since the software fault will exist in each replica.

The N-Version model uses n independently implemented versions of a software component run in parallel. By running the components in parallel with the same input data a set of results is obtained. By using a voting mechanism on this set of results

individual failures in a component can be eliminated and the integrity of the final result is increased. The voter will guarantee, to some agreed level of integrity, to return a correct result or flag an error.

Since each version of the component is independently implemented each should contain a different set of faults and the voting process can cancel these out. In practice common-mode failures can still occur since developers tend to implement similar solutions to problems [44], and also Web Service composition can include common Web Services increasing the chance of common-mode failure [68, 70] but there is still a significant gain in reliability from using this model [47]. As stated by Knight et al in [44]

> *"...our result does not mean that N-version programming does not work or should never be used. It means that the reliability of an N-version system may not be as high as theory predicts under the assumption of independence. If the implementation issues can be resolved for a particular N version system, the required reliability might be achieved by using a larger value for N using the coincident errors model to predict reliability."*

The voting mechanism used in an N-version system can become a single point of failure within a system. This is traditionally overcome by making the voting mechanism simple and conducting extensive testing to determine its reliability. A tiered voting mechanism can be used to reduce the chance of failure through machine faults.

An N-version model has traditionally been more expensive to design and implement that a single system since each version must be designed, implemented and maintained. With the advent of SOA and Web Services this cost may be reduced since many

independent vendors may offer functionally-equivalent Web Services which can be used to construct an N-version system, thus eliminating a large part of the implementation and maintenance cost.

Some work has been conducted on constructing frameworks to facilitate N-version models using Web Services. Most prominently is the work of Townend et al [68, 70] on the FT-Grid system which allows dynamic construction of N-version systems from existing Web Services. A simpler system designed for creating static N-version systems from Web Services is WS-FTM by Looker et al [48]. This system is less complex and service discovery is left to the application software unlike FT-Grid but it does allow construction of complex N-version systems and is therefore an ideal test bed for fault injection experiments.

3.3 Dependability Assessment

This work will focus on Dependability Analysis [8] rather than more traditional validation testing [61] and one of the areas focused on will be that of Dependability Means assessment. System dependability can be assessed using either model-based or measurement-based techniques [52]. Both techniques have their merits.

Modelling can be used in the design stage to predict potential errors and faults in algorithms. Measurement can be applied to existing systems to provide metrics on dependability.

Modelling can only make predictions of the dependability of a system since it is derived from system design, specification and code documents. 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, and may not require access to source code or design documentation. There are two main measurement techniques:

1. Observation
2. Fault Injection

Observation [41] measurements can be performed by the observation of errors and failures in a large set of deployed systems. This technique uses existing logs, either logs maintained by the system administrator or logs generated automatically by the system. Analysis of the data can obtain information on the frequency of faults and the activity that was in progress when they occurred. Since failures and errors may occur infrequently data must be collected over a long period of time and from a large number of systems. Even with this it is unlikely that this technique will catch rarely seen errors [37] since the MTTF may be very large (in the order of years).

3.4 Fault Injection

Fault injection [74] is a group of techniques that attempt to induce faults into a running system to assess, not only its tolerance to faults but it can also be used to exercise seldom used control pathways within the system which would otherwise go unused for long periods of time [19]. Since the MTTF may be very large, Fault Injection attempts to speed up this process by deliberately injecting faults into a running system in an attempt to speed up the production of errors and hence either exercise dependability means or produce failures. Fault injection can be used to simulate unusual input conditions and exercise the boundaries between software components that would otherwise rely on being exercised by calls generated by other components in response to

user input. Since this input will go through a number of intermediate steps it is extremely unlikely that this testing would be able to exercise all conditions of the component using traditional testing techniques [76].

Fault injection should be used as a supplement to traditional testing techniques. Traditional testing is intended to check that a system meets the requirements of the specification using expected input and conditions, whilst fault injection techniques are intended to assess the operation of a system under exceptional conditions and invalid input. Because fault injection assesses exceptional conditions and accelerates the occurrence of errors it is not suitable for the measurement of reliability or MTBF since these statistics are related to failures and time. These must be measured by observational techniques [38].

The technique of fault injection dates back to the 1970s [20] when it was first used to induce faults at a hardware level. This type of fault injection is called Hardware Implemented Fault Injection (HWIFI) and attempts to simulate hardware failures within a system. The first experiments in hardware fault injection involved nothing more that shorting connections on circuit boards and observing the effect on the system (bridging faults). It was used primarily as a test of the dependability of the hardware system. Later specialized hardware was developed to extend this technique, such as devices to bombard specific areas of a circuit board with heavy radiation.

HWIFI is divided into two types:

HWIFI With Contact: this involves the use of specialized hardware, which makes electrical contact with a target chip. These connections can cause current and voltage changes externally to the chip. These voltage changes can be used to

simulate a number of different physical faults. By analyzing the electrical signals input into the chip and using timers it is possible to control fault injection timing with a good degree of accuracy.

HWIFI Without Contact: this involves bombarding a specific part of a target device with heavy ion radiation or placing the target device in a strong magnetic field. This has the advantage of simulating natural physical phenomena but it has the disadvantage that it is very hard to control fault injection timing since there are latencies involved in physically firing these fault devices.

It was soon found that faults could be induced by software techniques and that aspects of this technique could be useful for assessing software systems. Collectively these techniques are known as Software Implemented Fault Injection (SWIFI).

The earliest application of SWIFI was Mill's fault seeding approach [55] which was later refined by stratified fault-seeding [56]. These techniques worked by adding a number of known faults to a software system for the purpose of monitoring the rate of detection and removal. This assumed it is is possible to estimate the number of remaining faults in a software system still to be detected by a particular test methodology. True SWIFI methods of injecting faults that simulated HWIFI faults soon followed [74].

In recent years there has been more interest in developing SWIFI based tools. This is partly because a SWIFI tool does not require any expensive specialized hardware. SWIFI also allow specific systems running on target hardware to be effectively targeted without injecting faults into other parts of the system. This is difficult to do with HWIFI techniques.

SWIFI also has a number of drawbacks:

- Faults can not be injected into memory that is in protected, for instance a user process will not be able to corrupt the memory of a kernel process in a system running under the UNIX operating system
- Instrumentation of the code to be assessed may perturb the operation of the system, for instance it may introduce latencies into an operation that could cause a time-out.
- 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 capture short latency faults. This problem could be overcome by using a combination of SWIFI and hardware monitoring to record fault events but this would increase the cost of the system and also decrease the portability of the system, since such hardware may be platform specific.

SWIFI techniques can be categorized into two types. Compile-Time Injection and Runtime Injection.

3.4.1 Compile-Time Injection

Compile-Time Injection (also know as code mutation) is an injection technique where source code (or assembler code) is modified to inject simulated faults into a system. A simple example of this technique could be changing

```
a = a + 1
```

to

```
a = a - 1
```

Although these types of faults can be injected by hand the possibility of introducing an unintended fault is high, so tools exist to parse a program automatically and insert faults.

This technique has the advantage that it can be used to simulate both hardware and software faults. It has been shown to induce faults into a system that are very close in nature to those produced by programming faults [24]. It requires no expensive hardware and no additional software during runtime. Further, the faults are coded into the system and require no communication with a fault injector so it has a considerably smaller impact on the execution timing of the system under test than other techniques. Since the faults are hard coded into the system image 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. This is a drawback for the following reasons:

1. It requires that the source code must be available to the test team, which will most likely not be the case for COTS systems.
2. There is the chance that unintended faults will be introduced during the code modification, especially if the faults are being injected by hand.
3. Since the source code is 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.

3.4.2 Runtime Injection

Runtime Injection techniques use a software trigger to inject a fault into a running software system. Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways:

Time based triggers: These triggers can be based on either hardware or software timers. When the timer reaches a specified time an interrupt is generated and the interrupt handler associated with the timer can inject the 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: Hardware exceptions and software trap mechanisms are used to generate an interrupt at a specific place in the system code or on a particular event within the system, for instance access to a specific memory location. These are the same mechanisms used by most debugging tools so it is a well understood technology. This method of trigger implementation is capable of injecting a fault on a specific event. Exception based triggers are usually limited to detecting access to a specific memory location as part of the fetch-execute cycle [62], for instance memory access to a particular location which holds a variable. When the exception is raised execution is passed to an interrupt handler.

The trap mechanism (often referred to as a breakpoint) is used to transfer execution to an interrupt handler at a certain point in the software [17]. It works by inserting a special trap instruction into the code (either at compile time or

runtime). When this instruction is executed it causes control to pass to an interrupt handler.

In either case when control passes to the interrupt handler, the interrupt handler can insert the correct fault into the system dependent on the event that triggered the fault injection. This trigger technique has the advantages that it requires no modification to the system code and is possible to trigger on specific events. Its main disadvantage is that on systems that support a process protection model the fault injector may be required to run as a system level process.

Code Insertion: This technique 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. In this way it is similar to using a trap mechanism but it is implemented as a normal function call so does not incur the problems associated with using processor dependent facilities.

This method differs from compile-time injection in that it injects its faults at runtime rather than at compile time and rather than corrupt existing code it adds code to perform the fault injection. Its main disadvantage is that it requires the system source code to be modified but it has the advantage that the fault injector can be compiled into the system as a library and run as part of the system in user mode on systems that support this process protection model.

This technique can be considered a hybrid of compile-time and runtime fault injection since it can use a trigger to determine when and if a fault is to be injected, either as an explicit trigger or by virtue of the placement of the code

insertion, but it requires code to be modified prior to compilation to inject faults similar to compile-time techniques.

To determine which SWIFI techniques are most appropriate to use when assessing Web Services it is useful to consider a number of criteria as well as general criteria that would contradict our stated goals.

One of the main defining characteristics of Web Services is that they run in a heterogeneous environment, in terms of machine architecture and implementation language. Further it is common for Web Services to run in a virtual machine that precludes injecting faults directly onto the hardware of the hosting machine. Therefore any SWIFI technique cannot depend on processor dependent features and this is our main criteria to determine if a SWIFI technique is appropriate to a Web Service environment.

One of the stated aims of this research is that the developed method should be as non-invasive as possible. Consequently any SWIFI technique should make as few modifications to source code and the environment as possible.

Finally, two further criteria, which are important generally to developing a fault injection method, are that there is a close linkage between the trigger mechanism and the injected fault and the possibility of unexpected faults being injected. A close linkage between a trigger event and the specific area a fault is injected into is important in terms of the repeatability of an assessment and also allows specific pieces of code to be assessed. A method which does not generate any unexpected or unrepeatable faults is also an advantage for similar reasons.

Table 3-3: Comparison of SWIFI techniques

	Modification of Source Code	Processor Specific Features	No close linkage between injected faults and trigger events	Possibility of unexpected faults being injected	Appropriate to environment of Web Services
Compile-time	X			X	X
Runtime					
Time-based Triggers		X[1]	X	X	
Interrupt-based Triggers		X			
Code Insertion	X				X

Table 3-3 compares these various criteria and identifies both Compile-time injection and Code-insertion as appropriate techniques since they do not rely on any processor specific parameters. Of these techniques Code-insertion provides the potential for far more targeted fault injection that Compile-time injection because it corrupts existing algorithms whereas Code-insertion can be used to manipulate inputs and outputs with a fine degree of control since it inserts extra code, for instance method parameters can be corrupted prior to making a method call.

[1] Whilst it is possible to implement time-based triggers using software timers rather than hardware timers, it would be hard to implement them in a language/environment agnostic way.

Runtime injection techniques can use a number of different techniques to insert faults into a system via a trigger. Some of the more pertinent ones with regard to middleware assessment are given here but the list if not exhaustive:

Corruption of memory space: This technique consists of corrupting RAM, processor registers, and I/O map. Both permanent and transient faults can be injected. Memory and processor registers can be simply corrupted by writing new values from the trigger handler routine. There are a number of different strategies for calculating new values to inject but one of the most effective methods is to bit flip values [64].

Syscall interposition techniques: This is concerned with the fault propagation from operating system kernel interfaces to executing systems software. This is done by intercepting operating system calls made by user-level software and injecting faults into them. Traditionally this has required kernel level code to archive but recent research indicated that it can also be achieved as user level processes [40] although this research was concerned with intrusion detection. Since most software relies on the operating system for a number of services, errors injected at this level can be propagated up to the system being assessed. There are a number of ways to propagate errors from the operating system, two of the most important being returning an invalid error code from a system call and signalling an exception.

Network Level fault injection [77]: This technique is 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. [26] 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 [52]. Faults are then processed normally by the protocol stacks at both ends and can be relayed to the application layer. The faults injected are based on corrupting packet header information and injecting random byte errors.

Given these fault insertion techniques it is useful to compare them in terms of their applicability to a Web Service environment. Given the heterogeneous nature of Web Service environments the key criteria of whether it is appropriate to use a technique is closely related to its dependence on a specific hardware environment. Further, security issues must be taken into account but since techniques exist to work around these problems it is possible to apply a technique that has implications to security. Finally, as a general criteria, the technique must be able to target faults reliably in specific pieces of code.

Table 3-4 gives a comparison of fault insertion techniques. It has identified network level fault injection as an appropriate fault insertion technique to use. Although it has issues with security it is possible to circumvent them, whereas the other two techniques are tightly coupled with specific hardware platforms.

Fault insertion techniques allow faults to be inserted into a system, either as a general corruption of the environment the system is running in, for instance memory corruption, or tied to a specific event, for instance a message transfer with network level fault injection. These techniques can be used to stress a system and can therefore be used to undertake Robustness testing.

Table 3-4: Comparison of Fault Insertion Techniques

	No close linkage between injected faults and trigger	Closely related to a hardware environment	Security Issues	Appropriate to environment of Web Services
Corruption of memory space	X	X	X[2]	
Syscall interposition techniques		X		
Network Level fault injection techniques			X[3]	X

Robustness testing is the deliberate stressing of a system to determine if it can function correctly in the presence of this stress. Marsden et al states in [53]

> *"The robustness of a system is a measure of its ability to operate correctly in the presence of invalided inputs and stressful environmental conditions".*

This is achieved by corrupting data at the interface level of a component and observing its behaviour. Robustness testing can also be achieved by running the system under a heavy processor load and observing its effects.

Robustness testing is particularly useful in assessing a system since it not only covers fault tolerance means to prevent invalid input, such as input that may be received by a system when it is under malicious attack, but it also assessing its operation under load and hence can give an indication of how a scalable a system will be when in use.

[2] May not be possible without running at kernel level.
[3] Corrupted messages may be rejected by protocol stack as security failures.

3.4.3 Fault Injection Tools

A number of SWIFI Tools have been developed and a brief review of a selection of these tools is given here. Five commonly used fault injection tools are Ferrari [42, 43], FTAPE [72], Doctor [35], Orchestra [25] and Xception [20].

Ferrari [42] (Fault and Error Automatic Real-Time Injection) is based around software traps that are used 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 faults can either be transient or permanent. Research conducted with Ferrari shows that error detection is dependent on the fault type and where the fault is inserted [42].

FTAPE [71] (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.

DOCTOR [35] (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 and traps are used to inject transient emulated hardware failures, such as register corruption. Code modification is used to inject permanent faults.

Orchestra [25] 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.

Xception [20] 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.

Each fault has a fault mask and this mask defines the type of fault that will be injected from a predefined set of fault classes, for instance stuck-at-zero, stuck-at-one, bit-flip, bridging, etc. Although this method of predefining faults speeds up fault generation, over a script based tool, 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) [50] but it did prove to be effective at detecting faults in frequently executed code.

Table 3-5 compares the features and properties of these common fault injection tools. The heterogeneous environment that hosts Web Services means that any fault injection tool must not be tied to a particular hardware platform. Any tool used must therefore be able to run equally well under any environment hosting Web Services. Network Level Fault Injection offers a way of using a relatively machine independent way of injecting

faults whilst allowing tools to be run on a specific hardware platform since only messages are intercepted and relayed to the fault injector.

Table 3-5: Features of Common Fault Injection Tools

	Tied to physical machine architecture	Provides network level fault injection	Predefined fault models	Invasive
Ferrari	X			X[4]
FTAPE	X			X[5]
Doctor	X	X		X[6]
Orchestra	X[7]	X		
Xception	X		X	

From Table 3-5 both DOCTOR and Orchestra support network level fault injection and could potentially be used to implement fault injection into Web Services but these two tools have been designed to perform network protocol testing and therefore don't decode complete middleware message sequences.

3.4.4 Fault Models

A fault model [6] is a model of the types of fault that can occur in a system whilst it is running. A fault model can be categorized into the following types of faults:

[4] Trap instructions are inserted into system code.
[5] A replacement disk driver is required.
[6] Code modification is used to provide permanent faults.
[7] Ported to a number of operating systems but each variant is still tied to a particular system.

Physical Faults: These are faults that are caused by physical hardware failures, such as memory failures, processor failures, power spikes, etc. It has been found that this class of faults can be replicated effectively by using bit flipping techniques.

Software Faults: This class of faults includes both programming faults and design faults.

Resource-management faults: This class of faults includes such faults such as memory leakage and exhaustion of resource, for instance file descriptor exhaustion.

Communication faults: This class of faults is specific to systems that use some form of communication so they may not be present in all systems. These faults are concerned with simulating faulty network connections by the corruption, duplication, reordering and deletion of network messages.

Life-cycle faults: This class of fault is concerned with the mechanisms that maintain objects. This class of faults includes such faults as premature object destruction, delayed asynchronous operations (outside specified timing constraints).

This list is not exhaustive but covers commonly found classes of faults found within most systems. A specific fault model must be defined for a system before its failure modes can be defined.

3.4.5 Failure Modes

Once a fault model has been defined the ways in which a system can fail must be defined. These are know as the Failure Modes [6] and these will vary from system to system but some common failure modes are given in Table 3-6.

Table 3-6: Example Failure Modes

- Crash of an object, process or thread
- Hang of an object, process or thread
- Corruption of data into the system
- Corruption of data out of the system
- Omission or duplication of messages
- Incorrect generation of exceptions

Again this list is not exhaustive but covers the likely failures encountered by a Web Service based system.

More detailed failure modes are defined on a per-application basis. For instance The Apache Software Foundation defines a set of failure modes for Axis [5] that are shown in Table 3-7. By comparison with the example failure modes given in Table 3-6 we can see that the failure modes given can be grouped into groupings mainly concerned with communication, with only one group "500: internal Error" being concerned with programming failures and exceptions.

This is a reasonable set of failure modes from an end user perspective of Web Service middleware such as Apache Axis since its failure model is concerned with the correct function of the middleware rather than Web Services. The failure modes given in Table 3-7 assume that the middleware is fault free and these are the failure modes that will be encountered by a normal user. To test the actual middleware a far more extensive set of failure modes is required since other failures may be encountered during testing. These failure modes will be more akin to the failure modes found in any large application.

Web Services are a rapidly evolving technology and as such specialized testing has concentrated on protocol issues and general middleware validation. It is a sad fact that most general texts on the subject neglect testing all together [27, 58] the assumption being that normal test methodologies can be used to test systems. Whilst this may be the

case during development of an in-house SOA, larger SOA using external third party components would not have access to source code so validation techniques could not be applied.

As far as we are aware dependability analysis of Web Services and middleware has not be extensively applied [52] although some work is now underway to apply Interface Propagation Analysis (IPA) to Web Services [78] although this work is only concerned with applying IPA to Web Services and not with providing a non-invasive framework to undertake dependability analysis.

A more detailed set of failure modes is given by Gorbenko et al [31]. This work concentrates on providing information on constructing fault tolerant Web Services from unreliable sets of Web Services. Consequently the failure modes given are aimed at providing a basis for constructing fault tolerant solutions rather than testing for correct implementation of Web Services, for instance the taxonomy given in this report groups all suspected corrupted results under one failure mode to allow these to be dealt with by Multi-Version Diversity mechanisms. Conversely for testing requirements we may require a higher level of granularity to help assess at which point a Web Service is failing.

Table 3-7: Axis Failure Modes

Failure Mode	Cause
Connection refused	The host exists; nothing is listening for connections on that port. Alternatively, a firewall is blocking that port. Site Finder: the URL is using a port other than 80, and the .com or .net address is invalid
Unknown host	The hostname component of the URL is invalid, or the client is off-line.
404: Not Found	There is a web server there, but nothing at the exact URL. Proxy servers can also generate 404 pages for unknown hosts.
302: Moved	The content at the end of the URL has moved, and the client application does not follow the links. Site Finder: the .com or .net address is invalid, the port is explicitly -or defaulting to- port 80
Other 3xx response	The content at the end of the URL has moved, and the client application does not follow the links.
Wrong content type/MIME type	The URL may be incorrect, or the server application is not returning XML. Site Finder: a 302 response is being returned as the host is unknown
XML parser error	This can be caused when the content is not XML, but the client application assumes it is. Site Finder: this may be the body of a 302 response due to an unknown host, the client application should check return codes and the Content-Type header
500: Internal Error	SOAP uses this as a cue that a SOAPFault has been returned, but it can also mean 'the server is not working through some internal fault'
Connection Timed out/ NoRouteToHost	The hostname can be resolved, but not reached. Either the host is missing (potentially a transient fault), or network/firewall issues are preventing access. The client may need to be configured for its proxy server. This can also crop up if the caller is completely off-line.
GUI hangs/ long pauses	Client application may be timing out on lookups/connects

3.5 Summary

This chapter has defined some commonly used terms and concepts used in QoS and some of these are subjective and therefore not suitable for measurement whilst others are quantifiable and will form part of the basis of this thesis.

This chapter has also reviewed dependability and fault tolerance methods that can be used to increase dependability. It has reviewed the process of how a fault may produce a failure.

Execution of an error may or may not lead to a failure being observed at the system boundary and this may be due to the presence of a fault tolerance mechanism. The main purpose of fault tolerance mechanisms is to allow a system to continue functioning reliably in the presence of errors. One simple fault tolerance model for eliminating errors is the N-version model which compares multiple outputs from functionally equivalent services and provides a consensus to either prevent failures of indicate when an answer cannot be reliably obtained.

MTBF may be very large since control pathways containing errors may be executed infrequently so fault injection techniques can be used to speed up the process of assessing fault tolerance mechanisms since specific faults can be injected into a replicated service to check the consensus processes.

There seems to have been little research carried out using network level fault injection to assess middleware products, the notable exception to this being it's application to CORBA systems where it provided promising results. Therefore this chapter has reviewed Dependability Assessment and Fault Injection as a means of assessing the dependability of systems. It has described the different techniques available for performing fault injection and has reviewed the available tools. Of these tools most are designed for general-purpose dependability assessment of protocol stacks and not middleware products.

It has demonstrated that network level fault injection can be applied to RPC exchanges since network messages are sent to implement each call. Traditionally simple fault models are applied using network level fault injection, such as bit-flip fault models, to test network protocols. Network level fault injection also runs the risk of being detected by the protocol stack and messages rejected so it is preferable to inject faults at a higher level in the protocol stack if specific middleware functionality is to be assessed.

Chapter 4 - FIT Method for Dependability Assessment

Although much work has been done in the area of fault injection and distributed systems in general, most dependability assessment of middleware appears to have been conducted using observational measurement techniques. There also appears to have been little research carried out on applying this technique to Web Services. Some research has been conducted using fault injection to test the dependability of CORBA implementations using network level fault injection with successful results and we have used this as a starting point for defining our new novel method.

4.1 Problem Definition

There are many problems associated with using traditional testing techniques and formal proofs on distributed systems. Dependability assessment provides a useful technique for allowing a level of confidence in a system to be obtained, even though this does not assure correct operation under all circumstances.

To address this need within the middleware domain a new dependability assessment method has been developed which has been called Fault Injection Technology (FIT) method has been developed. The FIT method provides a dependability assessment method that can be applied to SOA to allow an indication of the level of reliability of a system. There are many variations possible for an SOA. For the purpose of the FIT method we will define an SOA as being a simple system, composed of a number of services and clients interconnected via a middleware layer. Clients can make use of services via the middleware layer and services may make use of other services via a middleware layer (see Figure 4-1). We further assume a homogenous middleware layer is used by each service, although the services may run on heterogeneous platforms.

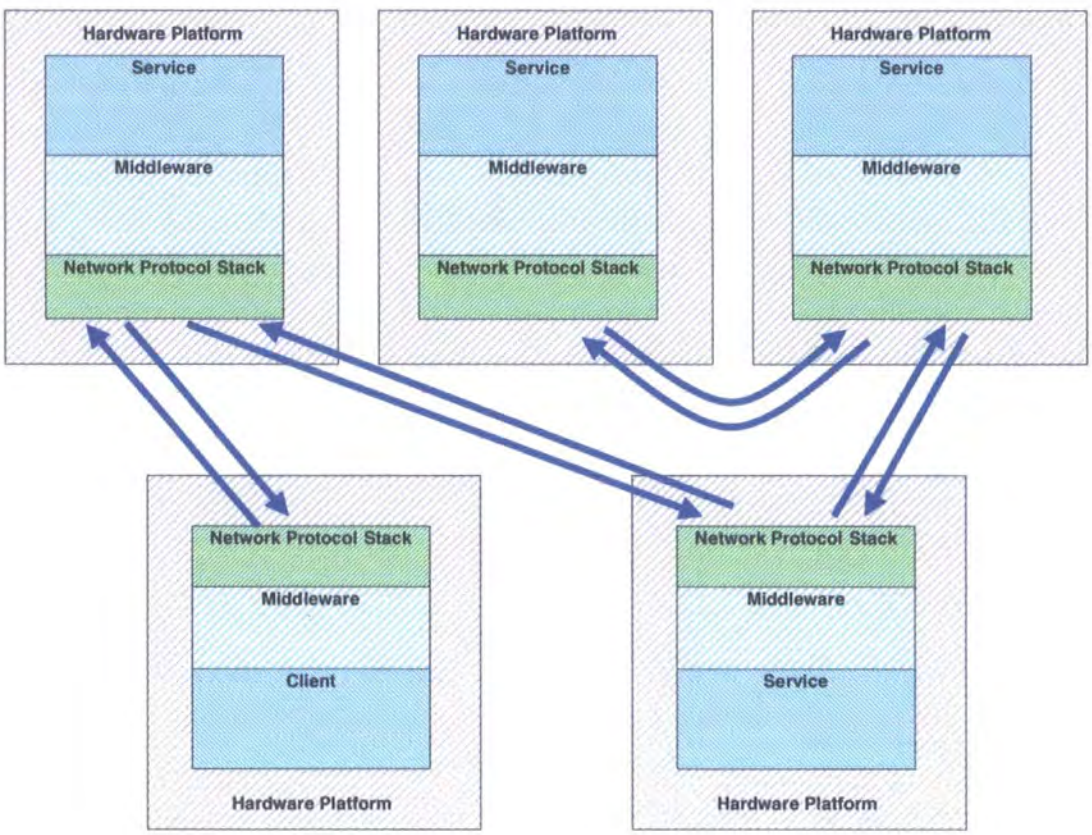


Figure 4-1: Outline Middleware System

The FIT method addresses three areas of dependability assessment:

- Fault Injection mechanism (Section 4.2)
- Automatic test generation (Section 4.3)
- Automatic failure detection (Section 4.4)

We intend the FIT method to be extendable and act as a framework, so the models given here are generic and it is intended that they should be enhanced and adapted to a specific system.

This thesis details the FIT dependability assessment method and shows how it can be applied to service based systems. Whilst the method supports ontology based automatic

test generation and failure detection the intension is to demonstrate the framework that the FIT method provides and any ontologies given are provided to illustrate this and are not intended to be definitive or complete.

The major contribution of this work is the novel fault injection mechanism that allows network level fault injection to be used to simulate Code Insertion fault injection whilst circumventing the need for modifications to the service source code. This comparison is limited to Code Insertion where it is used to perturb method input parameters or output results when an RPC is made since this generates a network request/response exchange which network level fault injection can operate on. It does not compare Code Insertion that operates on internal method calls, which do not involve an RPC.

This novel injection mechanism is achieved by intercepting middleware messages within the protocol stack, decoding the middleware message in real-time and injecting appropriate faults. By decoding the middleware message and allowing this level of targeted fault injection it is possible to perform parameter perturbation. This is detailed in Section 4.2.

The method incorporates three data structures intended to assist a user in describing a service-based system and construct triggers on specific parameters (Section 4.2), a data structure to aid a user in classifying and utilizing fault models (Section 4.3), and a data structure to classify and detect failures in a system (Section 4.4). This part of the FIT method could be implemented via any fault injection mechanism that supports parameter perturbation.

4.2 Fault Injection Mechanism

Of the techniques available we have decided to base the FIT method on network level fault injection for the following reasons:

1. Since the test domain involves transfer across a network interface it is a common factor in the design of both client and server software.
2. It is simple to implement under heterogeneous middleware because it can be implemented using small modifications to the middleware protocol stack.
3. Middleware messages can be modified to simulate a large number of fault classes, for instance transport layer failure, API parameter faults, etc.

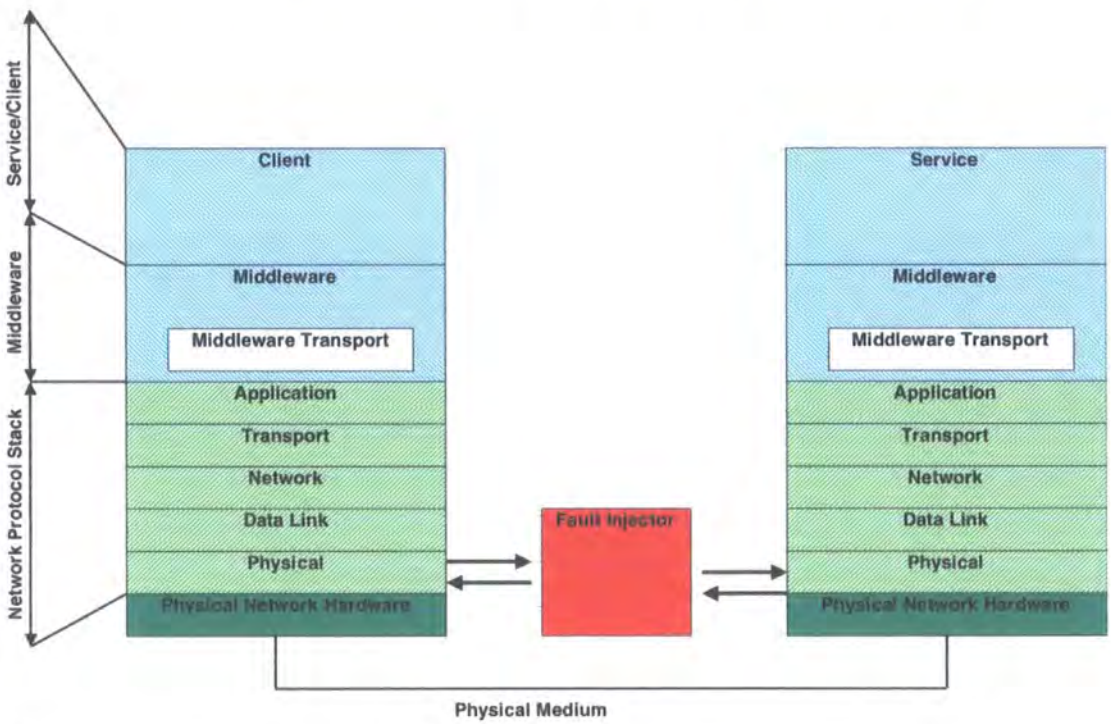


Figure 4-2: Detailed Middleware System

The FIT method uses a modified network level fault injection technique to inject faults into a service. Standard network level fault injection works by performing the

following operations on network packets at the physical network interface (see Figure 4-2):

- Corrupting
- Reordering
- Dropping

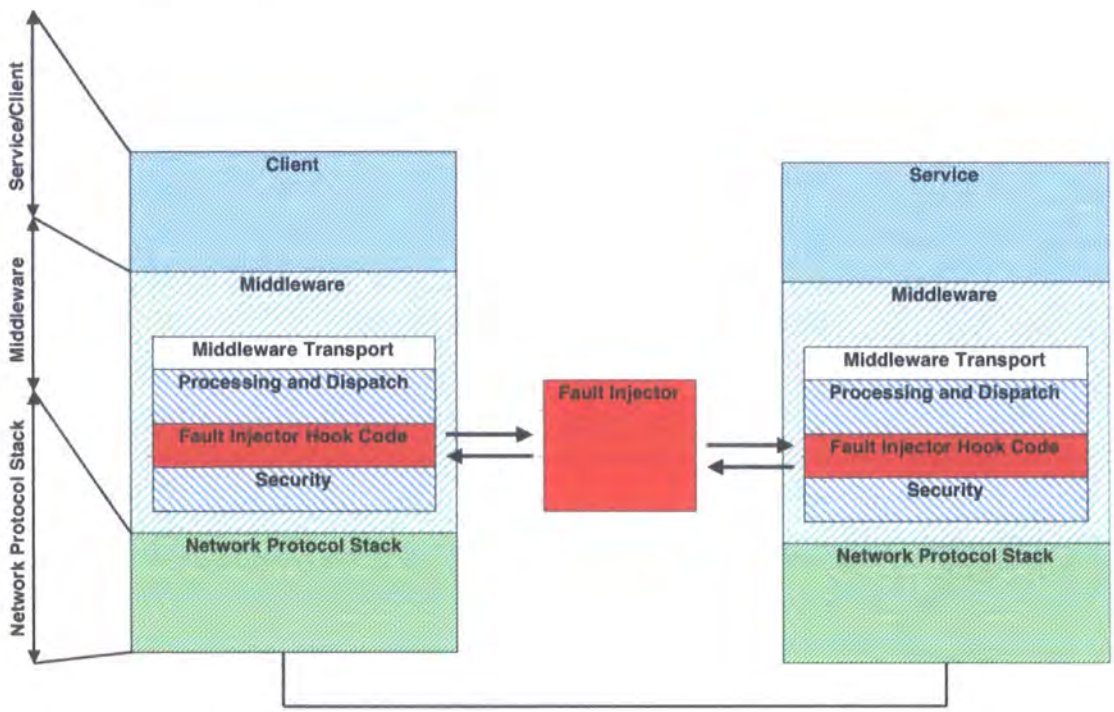


Figure 4-3: FIT Fault Injection Hook Code Placement within Middleware

Since this fault injection is done at the network interface modifications to these packets tend to only be reflected at the middleware level as random corruption of data, even reordering and dropping of packets may only result in corruption of a data stream since a middleware level message may span more than one physical network packet. Further, reordered or dropped packets may be subject to error correction such as retransmission so faults injected may not reach the middleware layer. Also packets corrupted at this level may be rejected by the network protocol stack lower layers, for

instance via mechanisms such as checksums. It is thus hard to target a particular element of a middleware message with any great certainty. Thus network level fault injection has traditionally been used only for assessing network protocol stacks, not service based systems.

The FIT method of network level fault injection takes the basic concept given above but moves the fault injection point away from the network interface and positions it in the actual middleware transport layer (see Figure 4-3). Since middleware messages are then intercepted as complete entities it is possible to corrupt, reorder and drop complete messages, rather than just part of a network packet that may be discarded before it reaches the middleware layer. Messages can thus be modified and then passed on to the lower layers of the middleware and network protocol stacks. In this way faults can be injected but not filtered out by the network protocol stack.

Further, if the messages are intercepted before they are signed or encrypted (or after they are decrypted and the signature checked in the case of incoming messages), it is possible to corrupt individual elements within a message without that message being rejected by the middleware as having been tampered with. Since we can assume we are familiar with the rules and metadata used to construct messages for the specific middleware we are using, by combining the corruption of data in a message with these rules and metadata it is possible to produce meaningful perturbations of such things as RPC input parameters and thus we can use our network level fault injection method to simulate API level fault injection.

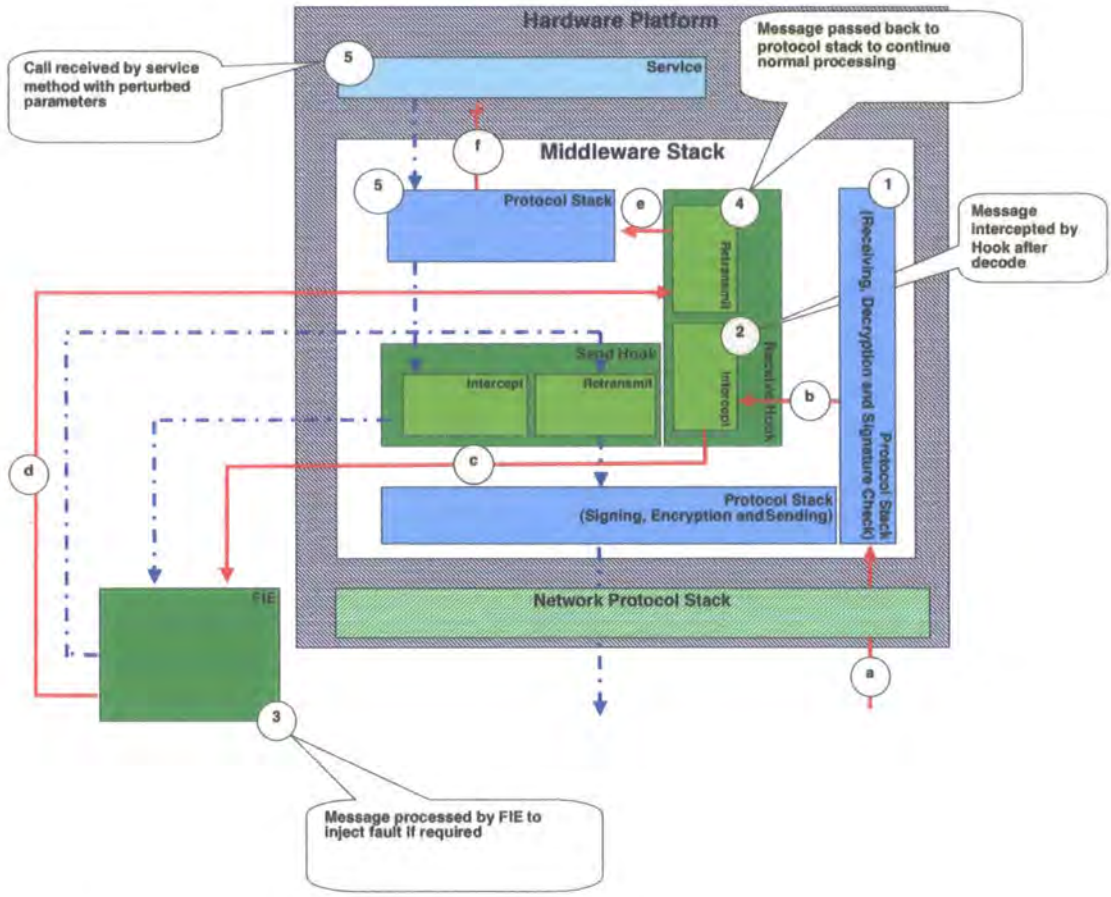


Figure 4-4: Procedure for Intercepting and Processing a Middleware Message

The procedure used to implement this process is shown in Figure 4-4. A message is processed in the following way on an instrumented machine:

1. A middleware message (a) is received by the Network Protocol Stack and is passed into the middleware stack (1).

2. This part of the middleware stack (1) receives the message from the Network Protocol Stack, decrypts it if necessary and validates any signatures.

3. The message (b) that is passed out of (1) is therefore in an unprotected and modifiable form. The message (b) is passed into the FIT hook code (2).

4. This part of the Hook Code (2) intercepts the message (b) and transmits it (c) via a socket to the FIT Fault Injection Engine (3).
5. The Fault Injection Engine (3) can then process the message and inject any faults required.
6. This processed message (d) is then transmitted back to the hook code (4).
7. The hook code (4) then retransmits the message (e) on to the remainder of the original middleware stack (5) that will complete the processing of the middleware message. At this point the message (e) may contain a fault and should be rejected by the protocol stack if the fault violates the protocol syntax.
8. If the middleware message (e) is valid and has not been rejected it will be processed as normal (5) and passed on (f) to the Service (5).

The rules and metadata used to define the interface of a service in a middleware are typically encapsulated in an IDL. Some IDLs, for instance DCE IDL, just define the interface of the web service and have a set of implicit rules built into the IDL compiler to generate the required message structures. Other IDLs, for instance WSDL, explicitly define the messages to be exchanged in the IDL definitions themselves. Which ever system is used it is possible to interpret the IDL files to decompose the service interface into method calls with their associated messages and within the messages identify specific parameters.

The FIT method decomposes this information into a taxonomy (see Figure 4-5) that we will refer to as the System Model. This taxonomy provides all the information required to construct a fault injection trigger. Although triggers can be constructed to

trigger on any node in the taxonomy the FIT method allows triggers to be constructed for either the parameter nodes or the message nodes. FIT is primarily concerned with perturbing parameters in RPC messages and this matches closely with the lowest level of node within the taxonomy. FIT also allows custom operations to be preformed at the message level so that the method can be more versatile.

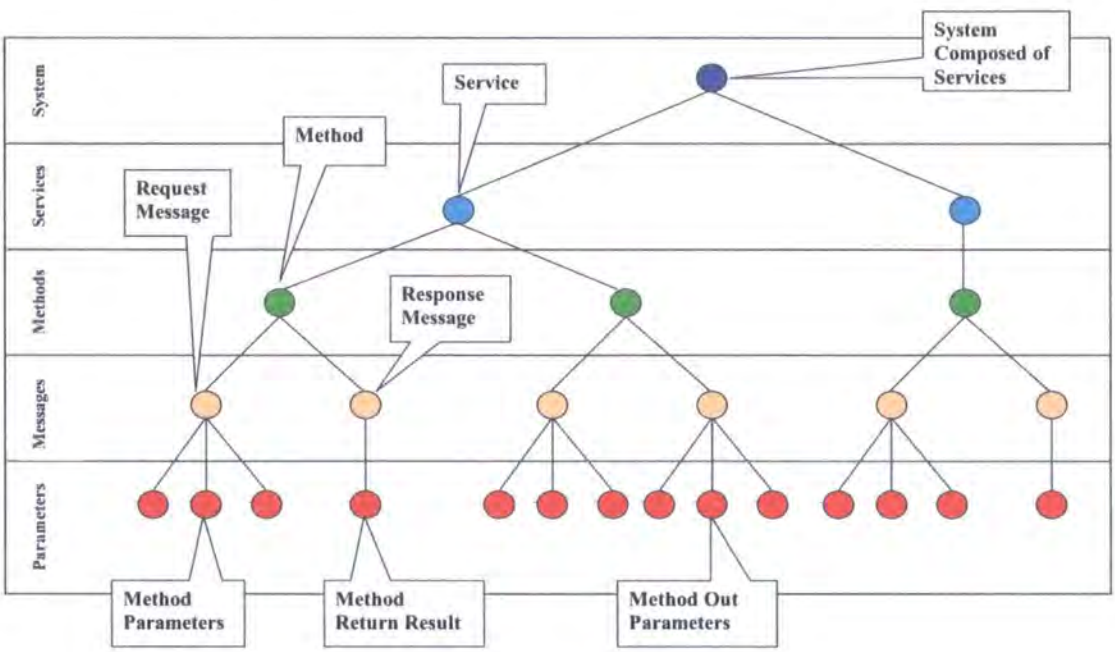


Figure 4-5: Taxonomy created from IDL

By using this taxonomy and creating triggers on specific parameters the FIT method can target individual elements of a message rather than inject random faults into middleware messages as in standard network level fault injection techniques. The method will decode the middleware message and inject meaningful faults, such as modifying RPC parameters and results, so that they are syntactically correct but may be out of specified ranges. The method builds on this framework to allow test cases to be written. These test cases can either be written manually or automatically generated using our Extended Fault Model.

4.3 Automatic Test Generation

The FIT method uses the concept of an Extended Fault Model (EFM). This is, again, structured as a taxonomy (see Figure 4-6). It is intended to act as an aid to the user evaluating a system by organising well known fault models into a structure that can be applied to the system described in Section 4.2. This takes the standard concept of a fault model and groups fault models into the taxonomy so that they can be classified and applied to an element of the System Model.

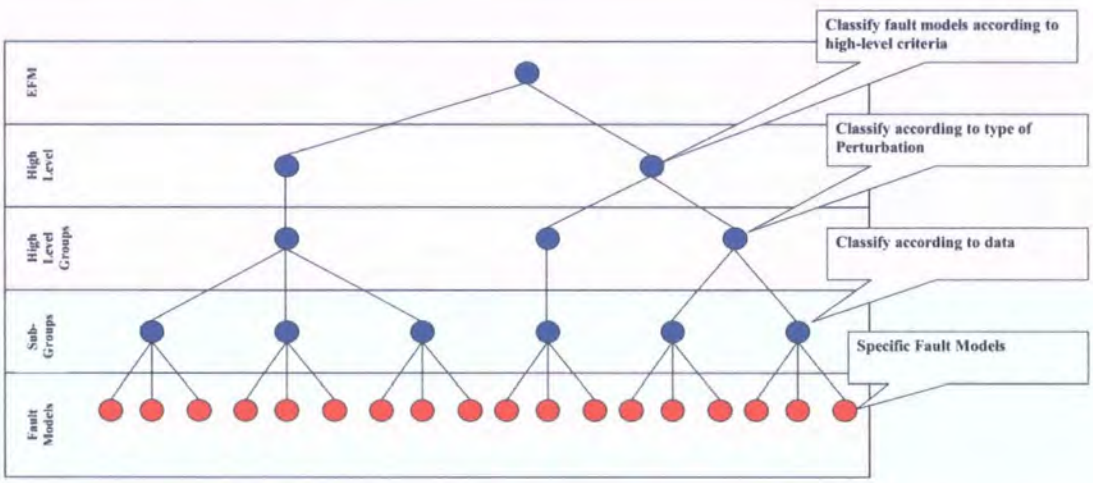


Figure 4-6: Fault Model Taxonomy

This is done by apply the technique of functional decomposition to the top-level fault model, for instance we can define a high level fault model, as show in Table 4-1, which shows a fairly standard fault model for an SOA. Then each item in the fault model would be decomposed into a more detailed sub-category (see Table 4-2) after which further decompose of each item in the model can continue until a level is reached where we have a detailed enough description to implement a specific test case (see Table 4-3).

Table 4-1: High Level Fault Model

- Physical Faults
- Software Faults
- Resource Management Faults
- Communication Faults
- Life-cycle Faults

The detailed test case should be generic enough so that it can be applied to any message/parameter of an appropriate type in the System Model, for instance a test case could be written to perturb any numeric input parameter so that it contains a random value within a specified range. The range would be obtained from the specification of that RPC and associated with the parameter's node in the System Model. The two taxonomies could then be linked together and the EFM could obtain the required range information from the System Model taxonomy.

Table 4-2: Partially Decomposed Fault Model

- Software Faults
  - Perturbation of Data into a Service
    - Values in Specified Range
    - Values out of Specified Range
    - Values in Specified Range but Logically Incorrect
  - Perturbation of Data out of a Service
    - Values in Specified Range
    - Values out of Specified Range
    - Value in Specified Range but Logically Incorrect
  - Coding Errors
    - Data Returned is the wrong type

The EFM is intended to be extendable so that a user of the system can customize it to cater to their system. The EFM described both here and implemented in the tools is therefore not extensive but is intended to provide a basic level of functionality that can be enhanced to fit users needs.

Table 4-3: Fully Decomposed Fault Model

- Software Faults
  - Perturbation of Data into a Service
    - Values in Specified Range
      - Upper Bound
        - Replace specified parameter with the upper bound value specified for this parameter.
      - Lower Bound
        - Replace specified parameter with the lower bound value specified for this parameter.
      - Lower Bound + 1
        - Replace specified parameter with the lower bound value specified for this parameter with one added to it.
      - Upper Bound – 1
        - Replace the specified parameter with the upper bound specified for this parameter with one subtracted from it.
      - Random Values between Upper and Lower Bounds
        - On test generation, generate a static sequence of randomly distributed values that lie between the upper and lower bounds inclusively. Cyclically substitute the next value from the statically generated sequence for the specified parameter. A static sequence should be used to provide a level of test repeatability although this repeatability may be effected by concurrency effects and timing.

It is possible, by careful perturbation of input parameters, to perturb the internal state of a service but this requires detailed knowledge of the design of the service and so is considered outside the scope of this method's automatic test case generation, but it can be accomplished via manual test case construction and manual inspection of the service code.

4.4 Automatic Failure Detection

To detect potential failures in a system, the method includes an Extended Failure Model (EFAM). This is similar in concept to the EFM described in Section 4.3 and is structured as a taxonomy (see Figure 4-7)

The EFAM defines a high level set of failure modes as described in Table 4-4. The FIT method applies decomposition to the failure modes in a similar way to the EFM.

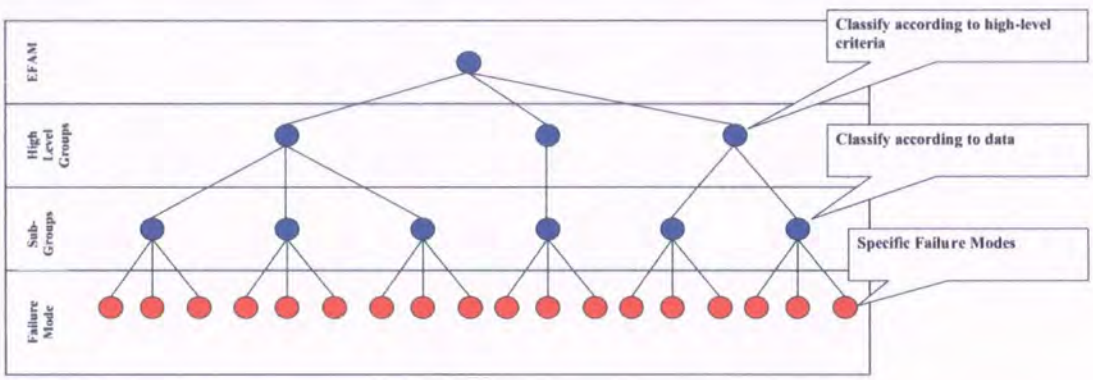


Figure 4-7: Taxonomy of Failures

Decomposition is applied iteratively to the Failure Modes to compose a number of partial models (see Table 4-5). These partial models sub-divide each failure mode into a number of sub sections until a simple enough level is reached at which point a script can be written to detect the detailed failure mode as in Table 4-6.

Table 4-4: High Level Failure Modes

- Crash of a service instance
- Crash of a hosting server
- Hang of a service
- Corruption of data into service
- Corruption of data out of service
- Duplication of messages
- Omission of messages
- Delay of messages

Table 4-5: Partially Decomposed Failure Model

- Corruption of data out of service
  - Data out of specified range
  - Data logically incorrect

Once a EFAM is available it can be applied to the whole or part of the SOA, so unlike the fault model it is active for all message exchanges so it can detect normal failures as well as failures caused by fault injection.

Table 4-6: Detailed Failure Model

- Corruption of data out of service
  - Data out of range
    - Data above upper bound
      - Check specified parameter against upper bound in specification and if it is greater than this value flag an error condition.
    - Data below upper bound
      - Check specified parameter against lower bound in specification and if it is less that this value flag an error condition.

To aid in failure detection the FIT method allows three predefined outcomes from a fault injection operation:

1. Exception
2. Response out of specified range
3. No visible effect.

These outcomes are specified when a fault model is applied to a parameter and apply to the next message exchange in that sequence after the fault model has been applied. The expected outcome is specified terms of a specific EFAM as part of the EFM. This effectively links the two taxonomies together using a many to many relationship (see Figure 4-8).

The exception outcome is the normal outcome to be expected from a fault injection since well-written code should detect parameter perturbation and reject the transaction. If we expect this outcome and do not detect the expected outcome this would be flagged to the user as a discrepancy, where as receiving the exception would be classed as being the correct outcome.

The response out of specified range outcome is typically to be expected from a system in the development or testing phase of a project when not all guard mechanisms

are present. This would typically result from a perturbed input parameter creating a perturbed result and should therefore be flagged to the user. We may expect this outcome if the result is being intentionally perturbed to allow it to be fed into another component to perturb it.

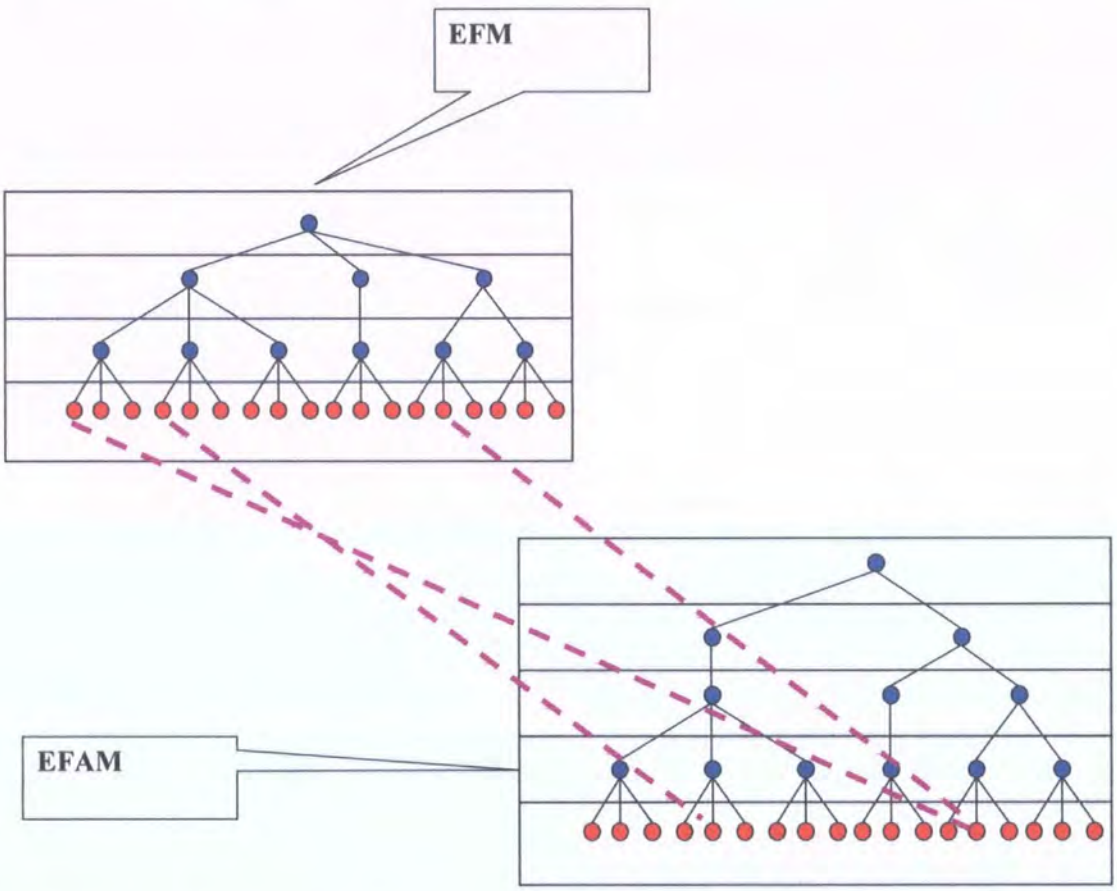


Figure 4-8: Fault Model Taxonomy Linked to Failure Model Taxonomy

The no visible effect outcome would typically be expected mainly from the message manipulation fault model class. This outcome would arise if normal message flows continued with no corruption of output data or exceptions. This is most likely to occur under two sub-classes:

1. Omission of messages from client to server, since the server would have no mechanism for knowing the original message had been sent so it could not generate an exception
2. Message manipulation involving time, since if a timeout is not reached this would result in a performance degradation rather than timeout exceptions.

In both cases a specialized test script in the failure model would need to be constructed to test for these on a per system basis.

This outcome could also be seen when parameters are perturbed within their specified range. In this case a manually constructed script would be written to detect the expected outcome on a per test basis.

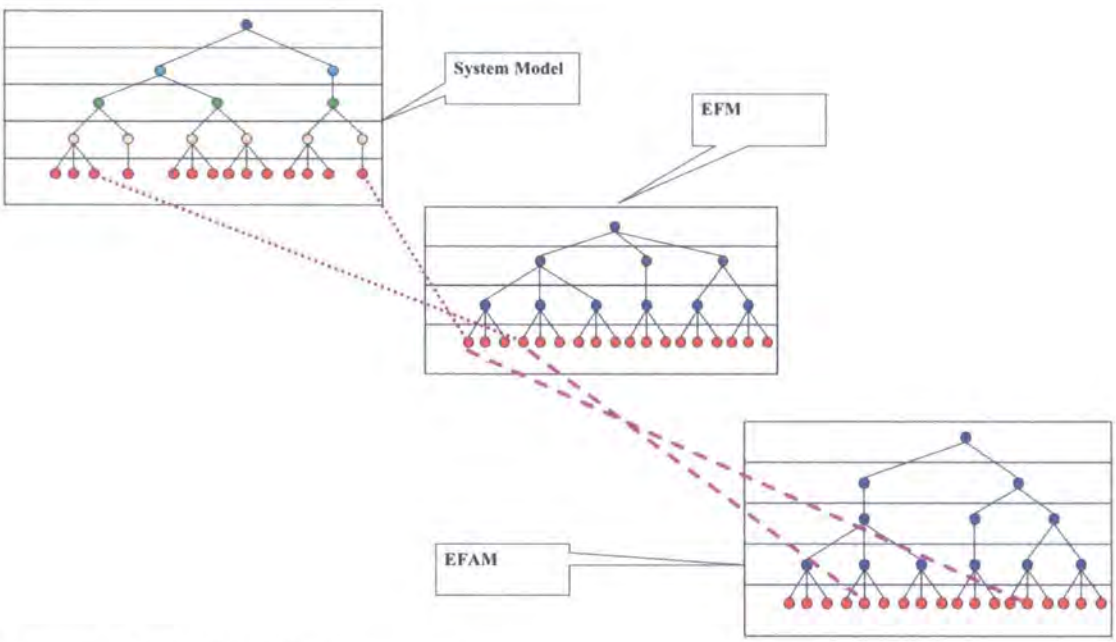


Figure 4-9: Linking Taxonomies

By applying one of the fault models contained in the EFM to the System Model taxonomy by linking the System Model to the EFM (see Figure 4-9) we not only generate a fault injection a specific place but because the EFM is linked to the EFAM it

is possible to determine any error condition that should be generated by the fault injection. This allows automated failure detection to be preformed.

4.5 Summary

This chapter has detailed the FIT method. The FIT method is composed of a novel network level fault injection technique, combined with a three of taxonomies to allow automated test generation and failure detection.

The novel fault injection technique used allows faults to be precisely injected into messages allowing perturbations of specific RPC parameters at the middleware message level. The injection mechanism also allows faults to be injected and passed to a service without being intercepted and rejected by either the network or middleware protocol stacks.

Test generation is achieved by use of the Extended Fault Model taxonomy. This is a collection of standard fault models that are categorised according to high-level criteria and each individual fault model can be applied generically to any RPC parameter.

Failure detection is accomplished through the Extended Failure Model taxonomy. This is a collection of standard failure modes that are categorised according to high-level criteria. The individual failure modes can be globally applied to the System Model to detect unexpected failures. Individual failure modes from the model can be associated with individual fault models from the Extended Fault Model that allows expected outcomes from fault injections to be detected.

Chapter 5 - WS-FIT Applied to Web Services

The FIT method detailed in Chapter 4 describes a generic method that can be applied to a range of RPC based middleware, such as DCE, DCOM, CORBA, Web Services, etc. This chapter applies the FIT method to Web Service middleware to demonstrate the method. This implementation is termed Web Service – Fault Injection Technology or WS-FIT whilst the generic method is referred to as FIT.

WS-FIT was conceived in two distinct phases with the intention of evolving the second phase from the first. The first phase is an implementation of a fairly conventional fault injector but with the enhancement that it processes middleware messages as opposed to network packets. The second phase builds on the first allowing meaningful faults to be injected into specific parts of a middleware message and implements the various taxonomies described in Chapter 4.

The first phase of the WS-FIT implementation implements the functionality of existing fault injection techniques and applies them to Web Services. The reasons for this were three fold:

1. We can draw on existing fault injection methods to select the best techniques that can be applied to Web Services.
2. Once we have a solid working method we can use it as a basis to design and implement our second phase method.
3. It should be possible to compare the FIT method to existing methods to gauge its effectiveness.

5.1 Middleware Environment

This implementation of FIT for Web Services realises the method described in Chapter 4 and provides a dependability assessment method that can be applied to SOA based on Web Service middleware (see Figure 5-1). This implementation has been applied to both Apache Axis 1.1 and Apache SOAP 2.3 for the SOAP implementation and Apache Jakarta Tomcat 4.x & 5.x reference implementation for a service container. This Web Service implementation can be deployed to form a heterogeneous distributed system comprising many different machine architectures with the middleware layer allowing interoperability between them (See Figure 5-1). WS-FIT must take this heterogeneity into account.

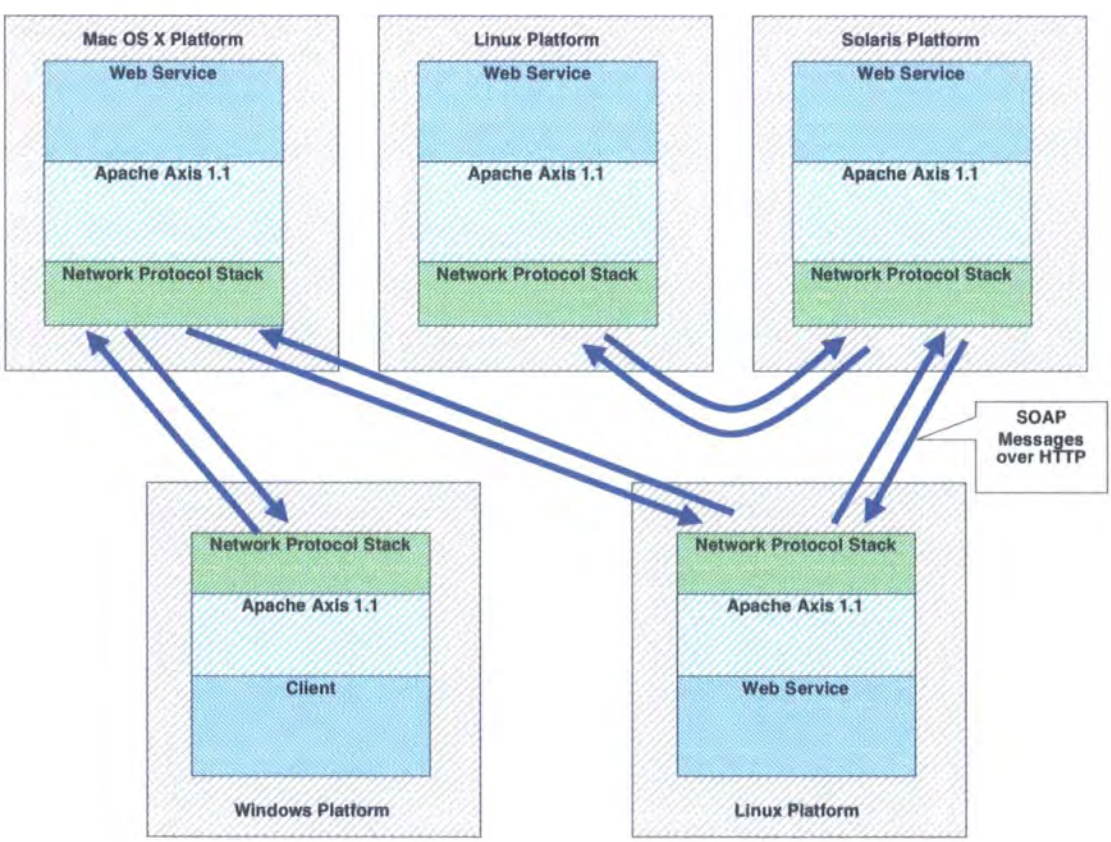


Figure 5-1: Web Service Middleware System

The WS-FIT implementation addresses the following areas which were defined in Section 4.1:

- Fault Injection mechanism (Section 5.2)
- Automatic test generation (Section 5.3)
- Automatic failure detection (Section 5.4)

This chapter shows the details of how the FIT method has been implemented and applied to Web Services and demonstrates the concepts behind the Automatic Test Generation and Automatic Failure Detection mechanisms on a real middleware product.

5.2 Fault Injection Mechanism

WS-FIT is physically divided into two parts:

1) Hook Code inserted into the SOAP stack to capture messages (see Figure 5-2 items H).
2) The Fault Injection Engine (FIE) that processes the messages and injects faults when required (see Figure 5-2 item E).

As shown in Figure 5-2 the hook code (Figure 5-2 items H) can be installed onto more that one machine, for instance on the SOAP stack of the machine running client software (Figure 5-2 machine 1) and the SOAP stack of the machine running the Web Service the client is utilising (Figure 5-2 machine 2). In this way the WS-FIT fault injection engine (Figure 5-2 item E) can intercept outgoing messages on machine 1 destined for machine 2 or incoming messages on machine 2 from machine 1. It is also possible to intercept either outgoing results before they leave machine 2 or as they are received by machine 1.

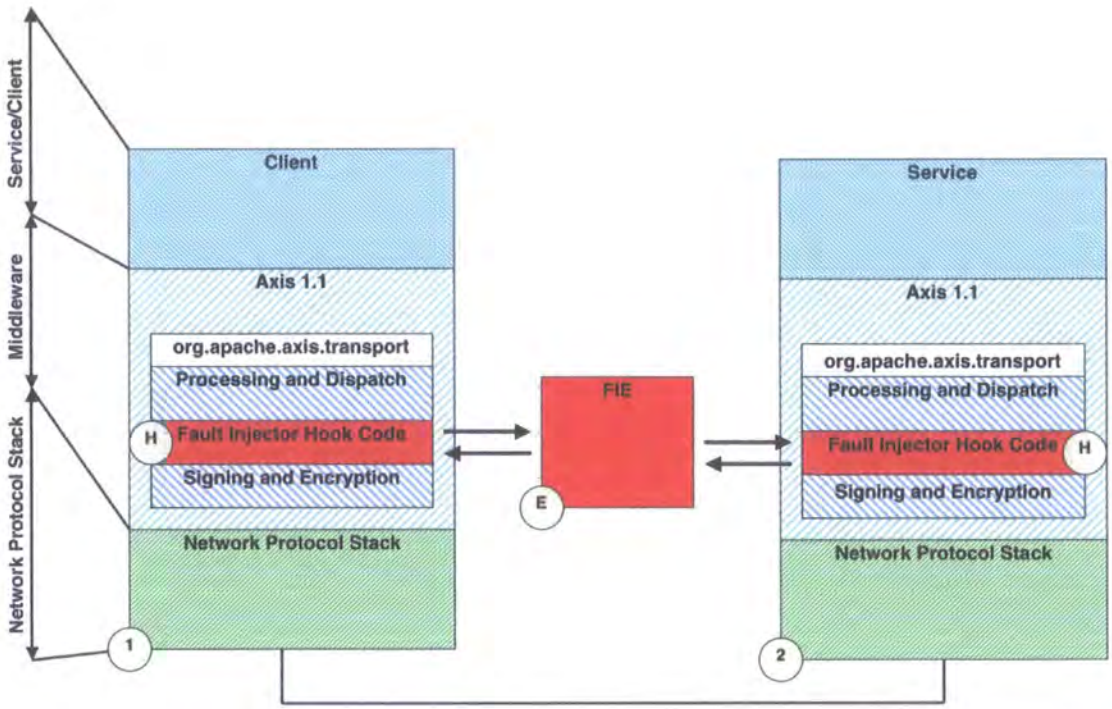


Figure 5-2: WS-FIT Fault Injection Hook Code Placement within Axis 1.1 Stack

Therefore the hook code is further sub-divided into two parts, one part that intercepts incoming messages and a second part that intercepts outgoing messages. This is partly dictated by the design of the SOAP stack (there are two distinct pathways through the code to allow processing of incoming and outgoing messages) and also to allow differentiation between the two message types. Although it would be possible to instrument only one pathway, for instance outgoing messages, and thus make the modifications to the SOAP stack less invasive capturing both incoming and outgoing message on one machine has certain advantages in terms of the flexibility of the method.

A basic requirement of certification release testing in a production environment is that the code installed and tested should be the code that is distributed to a customer since modifications to it could alter the control pathway used in production. Also compilation

of a source file can produce different binaries between compilations even when an unaltered source file is used because the compiler may have faults in it, etc.

The extra flexibility offered by WS-FIT could be used as part of a certification process. This can be demonstrated by use of Figure 5-3. This shows a distributed system made up of a number of Web Services and a client running on a heterogeneous network of platforms. In this example machine 1 and machine 2 have been instrumented with WS-FIT hook code, whilst machines 3, 4 and 5 have been left unmodified.

This configuration allows WS-FIT to inject faults into messages send and received by machines 1 and 2 without modifying the production environment of machines 3, 4 and 5. Thus faults can be propagated through the system into unaltered production machines without the need for specific test harness. This can be used to simulate transient and permanent faults in workflows.

For instance the client (on machine 5) can send a request to the Web Service on machine 1. The Web Service on machine 1 utilizes the service on machine 3, which in turn utilises the Web Service on machine 2, which makes a call on the Web Service on machine 4. Since machine 2 is instrumented with WS-FIT hook code it is possible to inject a fault into the request message going to machine 4 without altering the Web Service or environment on machine 4. If an error state is produced by the invalid request sent to machine 4 it may be propagated back to the client with no further modification to the system with the hook code on machine 1 being used to monitor the failure.

Similarly a fault could be injected into the system by modifying the return result of machine 4 as it is received from machine 4 by machine 2. This could then propagate any

error back through the workflow and be monitored by machine 1 at the system boundary. A similar effect to this could be achieved by injecting a fault into the response message sent by machine 2 to machine 3 but this would mean that the logic of the Web Service on machine 2 would have to be incorporated into the construction of the fault injected.

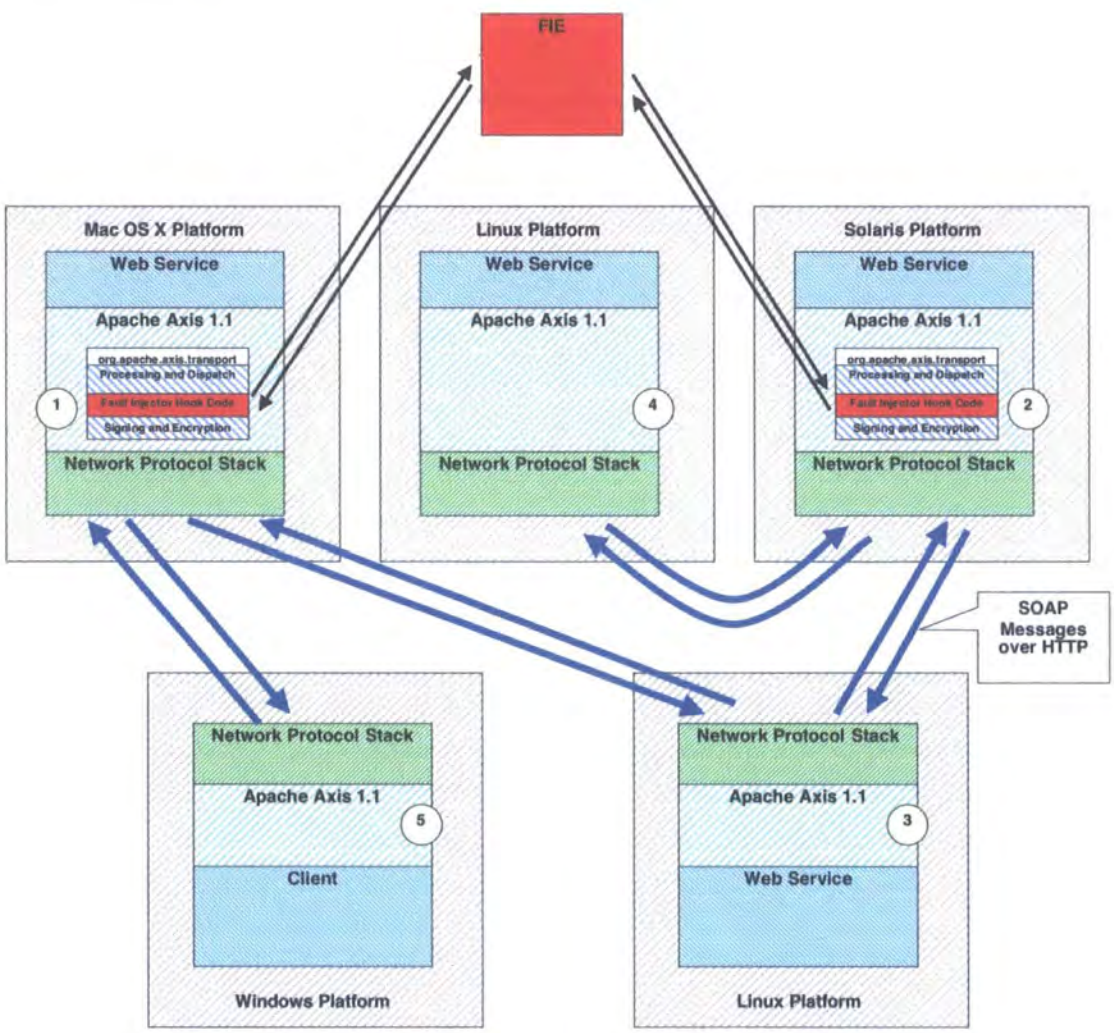


Figure 5-3: Instrumentation showing potential for certification

5.2.1 Hook Code

The hook code used to intercept message and pass them to the fault injection engine is relatively simple and the main components and data passed are shown in Figure 5-4. A message is processed in the following way on an instrumented machine:

1. A SOAP message (a) is received by the Network Protocol Stack and is passed into the SOAP stack (1).
2. This part of the SOAP stack (1) receives the message from the Network Protocol Stack, decrypts it if necessary and validates any signatures.
3. The message (b) that is passed out of (1) is therefore in an unprotected and modifiable form. The message (b) is passed into the WS-FIT hook code (2).
4. This part of the Hook Code (2) intercepts the message (b) and transmits it (c) via a socket to the WS-FIT Fault Injection Engine (3).
5. The Fault Injection Engine (3) can then process the message and inject any faults required.
6. This processed message (d) is then transmitted back to the hook code (4).
7. The hook code (4) then retransmits the message (e) on to the remainder of the original SOAP stack (5) that will complete the processing of the SOAP message. At this point the message (e) may contain a fault. If it is a SOAP schema fault the message (e) should be rejected by (5) as a protocol error.
8. If the SOAP message (e) is valid against the SOAP schema it will be processed as normal (5) and passed on (f) to the Web Service (5).

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

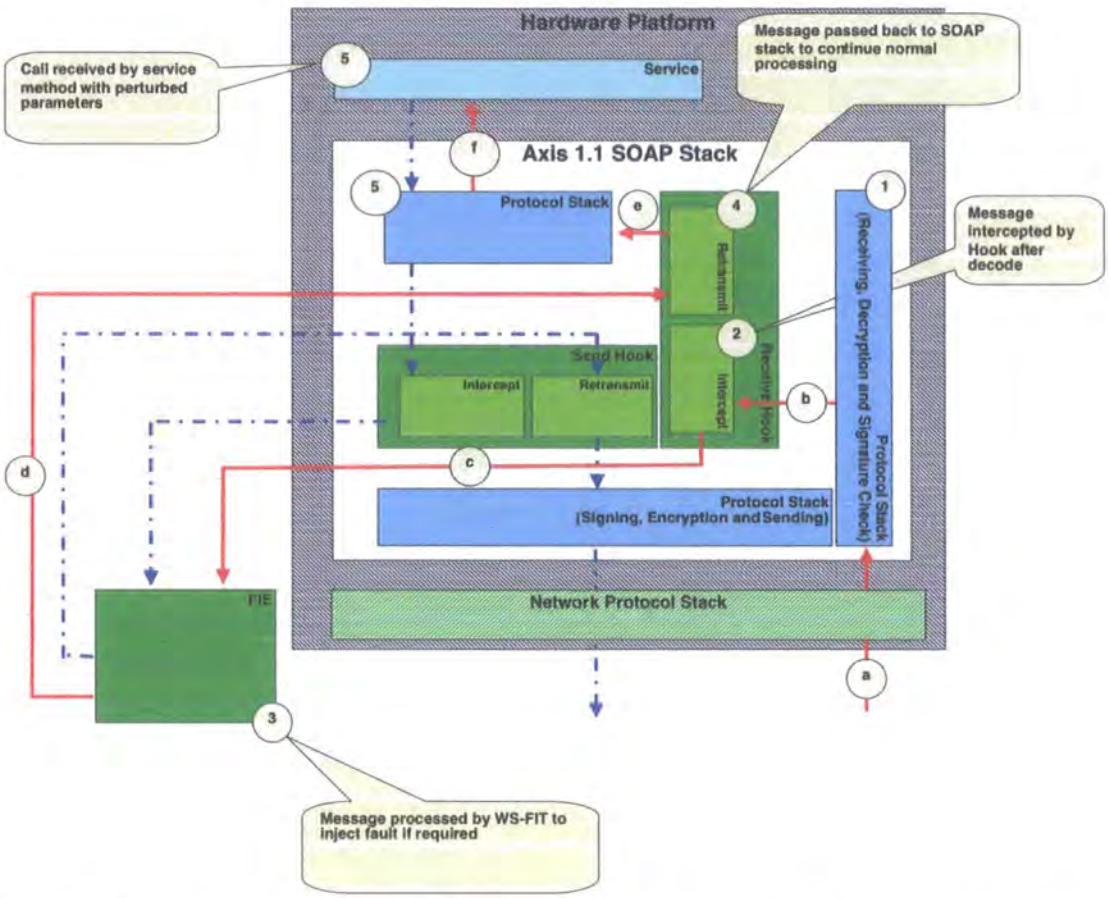


Figure 5-4: Instrumented SOAP Stack showing processing of incoming message

Outgoing messages from Web Services/clients are processed in a similar way (See Figure 5-5).

1. The service or client (5) sends information into the SOAP stack API (g). This may either be a direct call to the API from the program code or via the stub generated to handle return responses, etc.
2. The information (g) is then processed by the SOAP stack (6) into and unsigned and unencrypted message (h).
3. This SOAP message (h) is then intercepted by the WS-FIT hook code (7).
4. The hook code (7) then passes the message (c) to the WS-FIT Fault Injection engine (3).

5. The Fault Injection Engine (3) then processes the message in the same way as informing messages are processed (see above).

6. The modified message (d) is then passed back to the hook code (8).

7. The hook code (8) then send the message (j) on through the SOAP stack with can then perform any signing and encryption (9) that may be required before it is transmitted by the Network Protocol Stack to its destination.

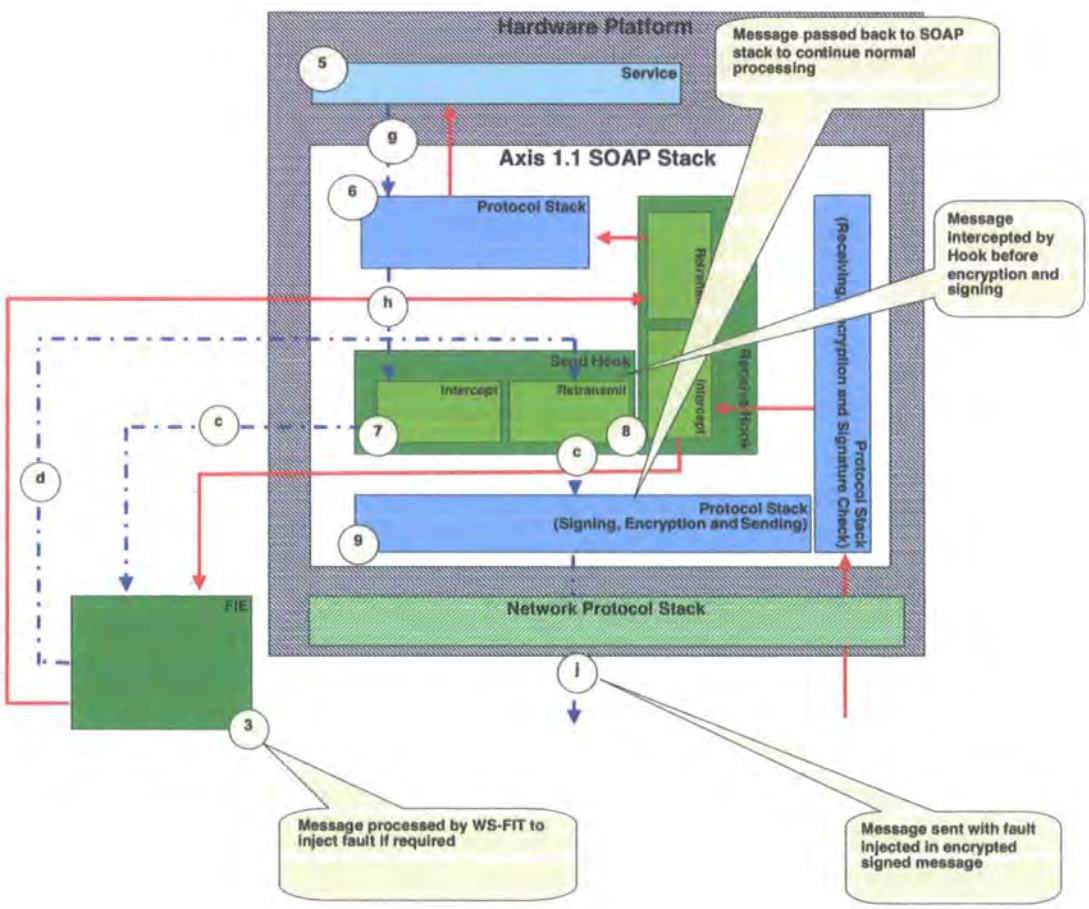


Figure 5-5: Instrumented SOAP Stack showing processing of outgoing message

FIT method is an enhancement from standard Network Level Fault injection in that it allows the targeted manipulation of specific parts of the message. This allows the FIT method to trigger on very specific RPC sequences and specific parameters contained

within them. This allows FIT to use Network Level Fault Injection to emulate Code Insertion fault injection.

The Hook Code is the only component of WS-FIT that must be implemented for each SOAP implementation that it is to be used with. The Hook Code is a simple piece of code that connects itself to the FIE via a network socket. Each SOAP message passing down the SOAP stack is intercepted by the Hook Code and transmitted to the FIE that may be running on a separate machine. The Hook Code then waits for a string reply from the FIE and substituted this message in place of the original message as described above.

Since WS-FIT is intended to be interoperable between different platforms and machine architectures it uses an XML format for transferring data between the fault injector Hook Code and the fault injector. This document is the message data passed into the FIE (c) and also the message passed out of the FIE back to the hook code (d). The SOAP message is encapsulated within an XML document as a CDATA section to circumvent the need to escape reserved characters. This makes the original message easy to receive and process since the start and end of the XML document is easily detectable. Also the encapsulating XML document can be used to carry such information as the originating machine's identity and timestamps.

The CDATA section within the document carries protocol specific information and this would change between implementations for different middleware products so the encapsulating XML document can also be used to identify the message data contained within. In this way it would be possible, although it may not be desirable for reasons of efficiency, to have on implementation of FIT handle several middleware products by

detecting the middleware type sent and switching to the appropriate parser for that product.

The Hook Code must be inserted into the SOAP stack at a convenient point, just before signing and encryption take place for outgoing messages and just after signature checking and decryption have taken place for incoming messages. In this way the injected faults in the messages can be propagated up the protocol stack to their intended destination and consequently test the domain they were intended for, rather than be rejected as network errors or a malicious attack on the system.

We have used this design, rather than something more complex because we believe that it provides the most portable solution between different SOAP implementations. For instance with our main test environment it would be possible to implement the Hook Code as a plug-in module in the processing chain. Whilst this would provide a neater solution for this particular implementation of SOAP, it would not be possible to implement the Hook Code in this way under all implementations since this facility is not available on some SOAP implementations. Further if our system used WS-Addressing or WS-MessageDelivery it would be possible to redirect the SOAP messages to the fault injector via the protocol stack, effectively removing the requirement for Hook Code on systems under test and thus removing the need to modify protocol stack code, but this would restrict the tools to working on SOAP based SOA that supported these standards and would make it harder to implement FIT on other middleware products that are not SOAP based. Also this approach would mean that each SOAP message would have to be processed by multiple SOAP stacks before being received by WS-FIT, and hence would make it harder to identify where errors have

occurred since the message must pass through intermediate nodes before reaching the fault injector.

The Hook Code is implemented so that once it had been inserted into the SOAP protocol stack it can safely be turned off and left in a normally running system via a configuration file. Whilst this should be relatively non-invasive it would not be advisable to ship a system with the Hook Code left in because of the risk of malicious attack because the Hook Code gives unsecured access to all messages sent and received by a server running it.

5.2.2 Fault Injection Engine

Once a SOAP message has been received from the Hook Code by the FIE it must be processed. The FIE is split up into a number of functional steps to aid processing of these message. These steps are:

Receive: Physically receive the XML document containing the SOAP message from the Hook Code.

Process: Extract the SOAP message from the encapsulating XML document and process and log the associated information in the encapsulating document.

First Stage Trigger: This step determines if the message supplied requires a fault injected into it. It is intended to be relatively fast so that detailed processing, and thus the inherent time overhead, can be minimised for message that do not require fault injection.

Second Stage Trigger: This is a more detailed triggering process that locates the actual position in the message where a fault needs to be injected. It relies on the

First Stage Trigger to identify messages where fault injection is required so it can be optimised with this assumption.

Fault Injection: Injects the actual fault into the message. It assumes that the Second Stage Trigger has located the correct position.

Transmit: This packages the message into an encapsulating XML document and physically transmits it back to the Hook Code.

Whilst these are the functional steps followed by the FIE design in the actual implementation some of them must be combined for efficiencies sake, for instance the Second Stage Trigger and the Fault Injection are combined. This is partly for efficiency and partly because they must be preformed iteratively if multiple faults are to be injected into the same message since the location of the faults will be different for each second stage trigger.

There are two paths through the FIE. One deals with the case where no faults are to be injected into a message, and the other deals with the case where faults must be injected. We assume that not injecting a fault is the default and have optimised WS-FIT accordingly because faults will be injected infrequently. This is shown in Figure 5-6.

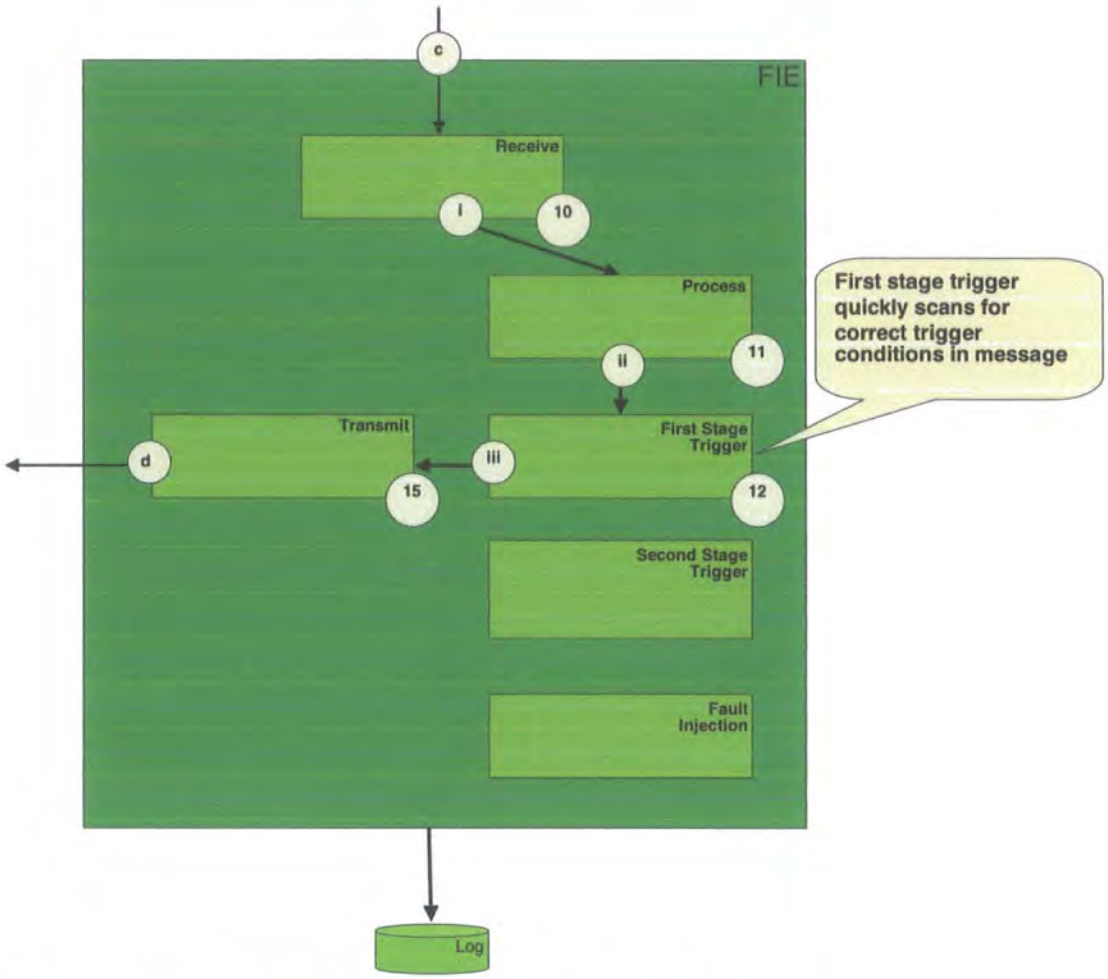


Figure 5-6: Processing of SOAP Message (No Fault Injected)

In this case the following steps are undertaken:

1. The encapsulated SOAP message (c) is received by the receive process (10).
2. The XML document (i) is then processed (11) to extract the SOAP message and the originating machine address.
3. This data (ii) is then used by the First Stage Trigger (12) to determine if the SOAP message should be triggered on and a fault injected. This is done by pattern matching specific XML tags to determine the specific message, for instance request/response message and message name, and if it contains specific RPC parameters. A match can also be done on originating machine

address so that specific Web Services on specific machine scan can be triggered upon. This stage also lends itself to a SAX parser implementation but the decision was made to implement it using pattern matching to reduce latency overheads.

4. If the SOAP message does not match the trigger criteria the data is passed (ii) to the transmit process (15) where it is packaged into an XML document for return and transmitted back to the Hook Code (d).

The other case specifies the path through the code when a message is received that does require a fault or faults injecting into it. This is shown in Figure 5-7.

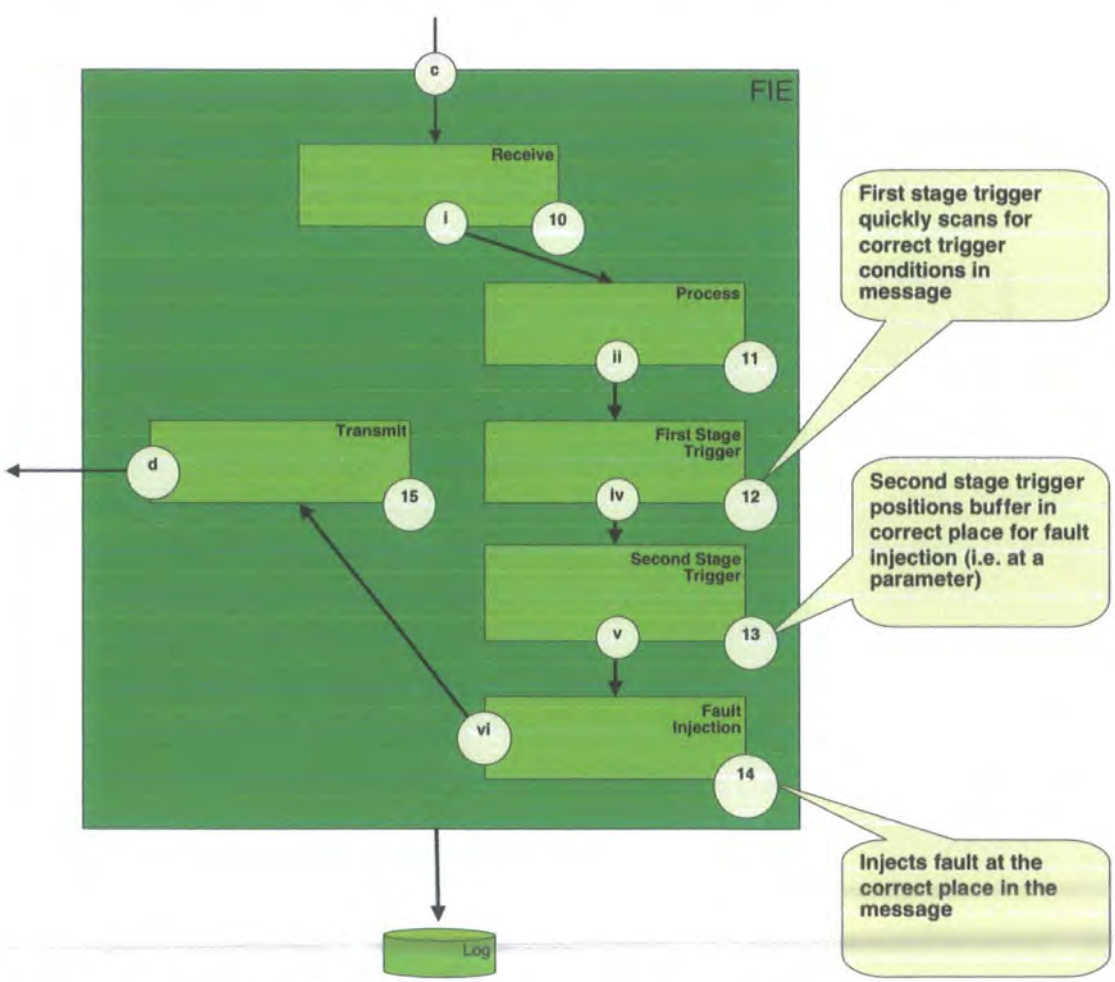


Figure 5-7: Processing of SOAP Message (Fault Injected)

In this case the following steps are undertaken:

1. The encapsulated SOAP message (c) is received by the receive process (10).
2. The XML document (i) is then processed (11) to extract the SOAP message and the originating machine address.
3. This data (ii) is then used by the First Stage Trigger (12) to determine if the SOAP message should be triggered on and a fault injected. This is done by pattern matching specific XML tags to determine the specific message, for instance request/response message and message name, and if it contains specific RPC parameters. A match can also be done on originating machine address so that specific Web Services on specific machine scan can be triggered upon. This stage also lends itself to a SAX parser implementation but the decision was made to implement it using pattern matching to reduce latency overheads.
4. If the SOAP message matches the trigger criteria the data is passed (iv) to the Second Stage Trigger (13). The Second Stage Trigger locates the precise location in the message data where the fault associated with a trigger should be inserted. In the WS-FIT implementation this lends itself to a SAX parser implementation since: a) the message is XML based; b) if multiple triggers are used on one message the SAX parser structure lends itself to iterative processing of a message.
5. Once the location in the message has been found in (13) the data (v) is passed to the Fault Injection process (14) that injects the faults into the message. Since a SAX parser was used this process was combined with the

Second Stage Trigger to allow iterative processing of a message with multiple triggers.

6. After the Fault Injection (15) process has injected all the required faults it passes the new SOAP message (vi) to the transmit process (15).

7. The transmit process (15) packages into an XML document for return and transmitted back to the Hook Code (d).

These two cases write data to a log at two main points shown in Figure 5-8. The log is structured as an XML document, with each transaction being enclosed in an XML element to aid in the analysis of the data.

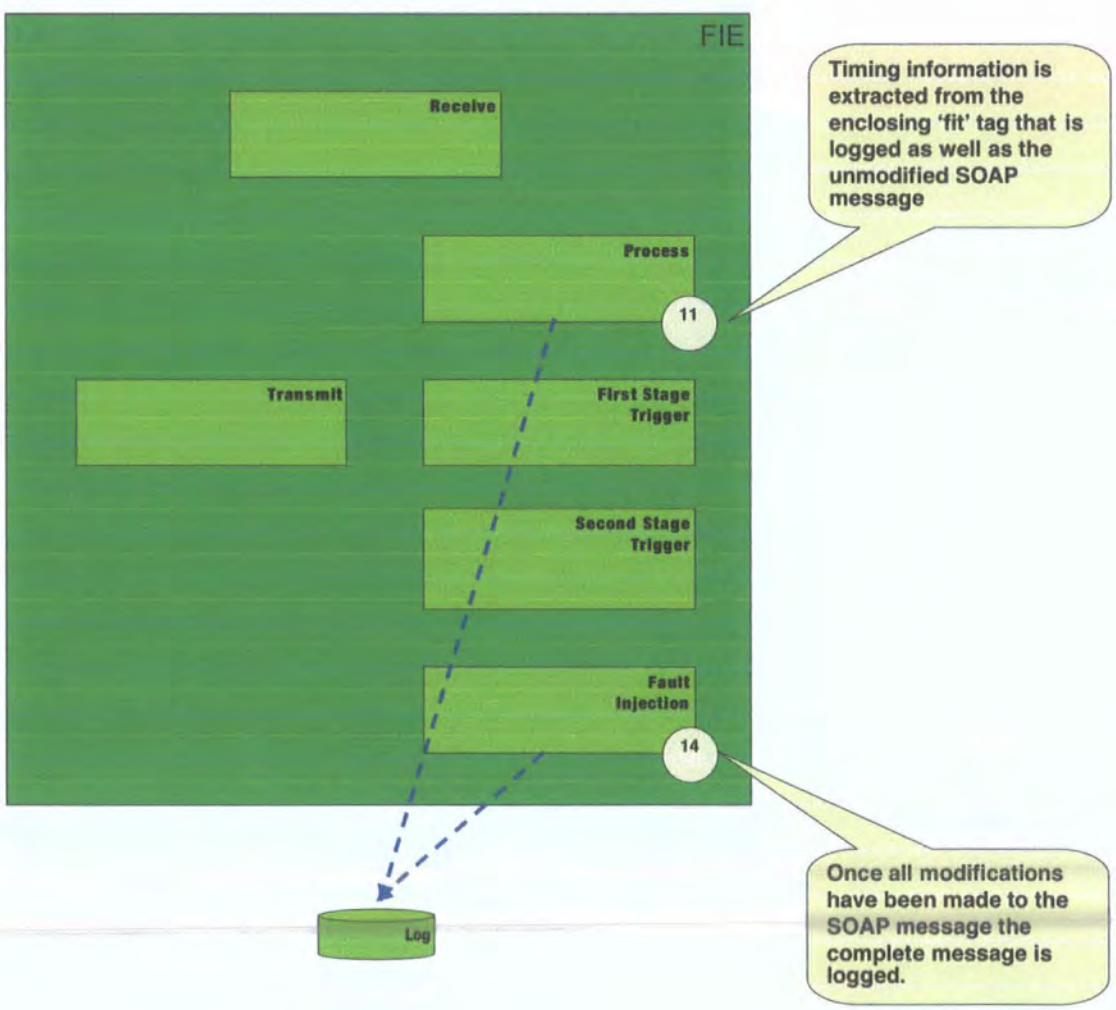


Figure 5-8: Logging Activities within WS-FIT

The first point is in the Processing of the initial XML document (11). This extracts the associated data from the encapsulating XML document, for example timestamps, and writes it to the log along with a copy of the original unaltered SOAP message. This provides a record of all data received from the Hook Code for later analysis. This also allows baseline audit data to be obtained from any system by running a 'null' script, for instance a script that does not inject any errors. This allows comparisons to be made with later fault injection campaigns.

The second point data is written to the log is at the end of the Fault Injection process (14). This logs the modified SOAP message thus allowing the exact fault injected into the message at a later data by comparison to the original message.

At various points in the Fault Injection Engine timestamp and diagnostic information is recorded so that assessments of the latency introduced by the FIE can be made but this is omitted for clarity.

5.2.3 First Phase Fault Injector

The first phase of the WS-FIT implementation implements a relatively standard fault injection mechanisms based on existing models with one major exception, WS-FIT is designed to decode each SOAP message as it is received and inject faults at a message level rather than on a network packet level. All middleware messages are supplied as complete entities regardless of the number of network packets involved in their transport. Faults can then easily be injected at this message level and supplied back to the protocol stack.

The first phase allows:

1. Triggers on particular messages to be created and executed.
2. Any part of a message to be manipulated, and thus inject a fault, including the structure of a message.

The first phase fault injector allows traditional fault injection operations to be implemented, for instance corruption of bytes within a message, reordering messages, dropping messages. This can be used for SOAP protocol stack assessment.

5.2.4 Second Phase Fault Injector

The second phase fault injector is based on the first stage and enhances it. Since SOAP messages are presented as complete entities it is possible to modify the triggers to trigger on specific parts of the message, for instance a specific RPC parameter in the message. This allows syntactically correct perturbations of input and output data to be preformed by substituting one value for another. Since these perturbations are syntactically correct as far as the SOAP scheme is concerned the SOAP stack will pass them through to their destination thus injecting a fault into a Web Service.

WS-FIT uses a simple script based trigger mechanism and is intended to be used with SOAP based RPC mechanisms so there will be a request message and a response. A uniquely named message is used by each RPC to implement both the request and response messages. This name is defined as a tag within the SOAP envelope to uniquely define each message transfer.

One of the advantages of the XML based SOAP is that it is self-describing. This means that our trigger mechanism can parse each SOAP message as an XML document and determine which RPC is being used by detecting the RPC's unique tag within the

message. In this way WS-FIT can determine a specific request or response message to a specific method of a service.

It is also possible to determine which client or server is transmitting the message because it will be known which particular SOAP API on which machine is instrumented and this information is passed as part of the enclosing XML document from the hook code to the FIE.

The first stage of WS-FIT implements a two-stage trigger mechanism. The first stage determines which message has been sent. This allowed fault injection to be targeted at a specific message and a specific method of a service. Once the first stage has detected the trigger message, the second stage runs and this injects the fault into the message.

The fault injected is generated by a second script. The script is only executed if the trigger determined that the message is of the specified type. Since our first stage model is designed to employ conventional fault injection techniques WS-FIT allows the corruption of any byte within the message but since SOAP messages are implemented as XML documents WS-FIT reflects this in its injection technique. An XML document is made up of elements; each element is composed of a start tag, a body and an end tag. WS-FIT treats each of these as a separate entity and presents each of these in turn through the model and allows each to be corrupted separately. Elements can also be nested within an element body so these are presented recursively through the method. Within each of these parts, any byte can be modified or the entire string making up this part of the message can be modified or discarded.

In this way faults can be injected so that either validly formatted SOAP messages, conforming to the SOAP schema, can be constructed or SOAP messages can be

corrupted in such a way as to present invalid SOAP/XML to the system. Invalid SOAP/XML should be rejected by the SOAP stack at a low level and would not allow us to ultimately target the domain we wished, but our fault injection method is capable of performing this type of corruption none the less.

WS-FIT is capable of processing multiple connections to multiple servers concurrently. When a server's Hook Code makes a socket connection to WS-FIT a thread is created and this thread acts as a listener on the socket. All processing of messages is then carried out asynchronously, with only shared logging and monitoring operations synchronized. In this way it is possibly to construct test cases that are coordinated across a number of machines, for instance if there are three services on three different machines and each has a trigger event associated with it, it is possible to coordinate their triggers such that trigger (C) can only happen once trigger (B) has occurred, which can only occur after trigger (A) has fired.

Each SOAP message is decoded using an Apache Xerces SAX parser. The SAX parser parses each message and calls appropriate abstract methods for each of the three sub-divisions of an XML tag, namely start, body and end. These abstract methods are implemented in the user script so that each element of the SOAP message is available in sequence to the user script. Each abstract method returns either the tag that it was supplied with, or a start containing an injected fault. Triggers and fault injection can thus be preformed on the SOAP message.

The body part of an XML element can potentially be nested and this adds extra complexity to the processing algorithm, for instance examine the SOAP message given in Table 5-1 we can see that each Body element is nested inside an Envelope tag. The

RPC message is, in turn, contained inside the Body tag and the RPC message element contains any parameters associated with this RPC call.

Table 5-1: Example SOAP Message

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
            xmlns:soapenv=http://schemas.xmlsoap.org/soap/envelope/
            xmlns:xsd=http://www.w3.org/2001/XMLSchema
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <soapenv:Body>
            <ns1:setTemp
                        soapenv:encodingStyle=

      http://schemas.xmlsoap.org/soap/encoding/
                        xmlns:ns1=

      "http://heatercontroller.samples.dasbs.org">
                  <newTemp xsi:type="xsd:double">
                        80.0
                  </newTemp>
            </ns1:setTemp>
      </soapenv:Body>
</soapenv:Envelope>
```

It is clear from this example that a mechanism is required to track triggers on messages and nested parameters. Since our input data will always be structured according to an XML Schema we can safely assume that all parameters will be nested inside the message element utilizing them. We therefore utilize a choice of triggers:

1. Trigger on a message and have the script process the entire message body
2. Trigger on a parameter within a message.

The first type is used to trigger so that the whole message body is passed to the user script. It is then the responsibility of the user script to interpret the message body and inject faults in appropriate places. This allows the user script to inject faults that are syntactically incorrect, for instance they are invalid XML or invalidate the SOAP

schemas being used. This also allows specialized test cases to be written which are not included in WS-FIT by default.

Table 5-2: Example Script

```
1:from org.dasbs.fit.exec import ServerContainer
2:from org.dasbs.fit.exec import InjectScript
3:from org.dasbs.fit.exec import ResultScript
4:from string import find
5:from operator import mod
6:
7:class UserResult(ResultScript):
8: def __init__(self,graph):
9:  ResultScript.__init__(self,graph)
10:  return
11:
12:class UserInjectScript(InjectScript):
13: def __init__(self):
14:  InjectScript.__init__(self)
15:  self.reset()
16:
17: def reset(self):
18:  self.m_msg = 0
19:  self.m_param = 0
20:
21: def create(self):
22:  return UserInjectScript()
23:
24: def injectStart(self, name, attrs):
25:  if self.checkName(name, attrs, 'ns1:setTemp',
'http://heatercontroller.samples.dasbs.org'):
26:   self.m_msg = 82
27:  return None
28:
29: def injectBody(self, name, attrs, body):
30:  self.m_param = 0
31:  if self.m_msg == 82:
32:   if name == 'newTemp' and
mod(self.getResults().getCount('newTemp', None), 10) == 0:
33:    self.m_param = 105
34:  if self.m_param == 105:
35:   print "the trigger has fired"
36:  return None
37:
38: def injectEnd(self, name, attrs):
39:  return None
40:
41:server.startServer(UserInjectScript(), UserResult(monitorGraph))
```

The second class of trigger allows the body of a parameter to be processed. The parameter body is supplied to the user script and can be manipulated by it. Since only the body of the element is supplied to the user script, whatever fault the user script

injects will result in syntactically correct XML being generated since the tags will be the responsibility of the fault injection framework. It will be possible for the user script to supply an element body that will be invalid according to the schemas being used, for instance a double with a decimal point could be supplied for a parameter that is typed to an integer via a schema.

This trigger mechanism is implemented in the user script (see Table 5-2). Each RPC message type is allocated a unique numeric value that is used to quickly identify it. Also each parameter is allocated a unique numeric id. In the start method, the start tag of the message to trigger on is pattern matched and if it is detected the unique id for that message is stored in a variable. This variable has scope whilst the SOAP message is being processed and is reset when the next SOAP message is processed. This can then be used in the start, body and end methods in the user script as a trigger for a specific message.

Parameters are triggered on in a similar way. The message variable is checked to see if the correct message body has been reached. If it has, pattern matching is preformed on the tag to determine if a particular parameter has been reached and if it has the unique ID for the parameter is stored in a second variable. This can be used in a similar way to the message ID to trigger on a specific parameter in all three script methods.

The design for the first stage utilizes a two-stage trigger mechanism. The purpose of this is two fold:

1. The first parser stage can be lightweight and can speed up the trigger process in systems with sparsely triggered events

2. Since the whole message is parsed in the trigger stage out of sequence triggers can be constructed, for instance if a test script wants to modify an element in a message that occurs before the trigger element this can be done.

This design has the disadvantage that it complicates the user script triggers. Since the trigger stage returns a simple Boolean to determine if the second stage should run, the second stage user script needs to do an amount of pattern matching as the message is parsed the second time to determine the correct place to insert the fault. Also multiple faults can be injected into the same message meaning that all pattern matching needs to be carried out on each message, since the only thing the trigger stage indicates is that the message needs a fault injecting into it, not which set of faults need to be injected.

The two-pass design allows an initial check of the SOAP message to ascertain if any faults need to be injected. Then the second pass can be preformed which locates the specific points in the message where faults need to be injected. By using a two stage trigger mechanism WS-FIT is capable of injecting multiple faults into a SOAP message whereas a one stage trigger mechanism would be far more complex, for instance if we required a trigger test to determine that a fault must be injected into a message but the specific trigger tag came after the point where the fault was required to be injected then a one pass mechanism would need to reposition the buffer and inject the fault, whereas a two stage mechanism would do an initial pass to determine if a fault injection was required and then in the second stage locate the point in the message where the injection was required and inject the fault.

Although the API used to construct scripts allows them to be written and executed manually the WS-FIT tool implements a graphical user interface that automates the construction of triggers and allows predefined fault models to be applied.

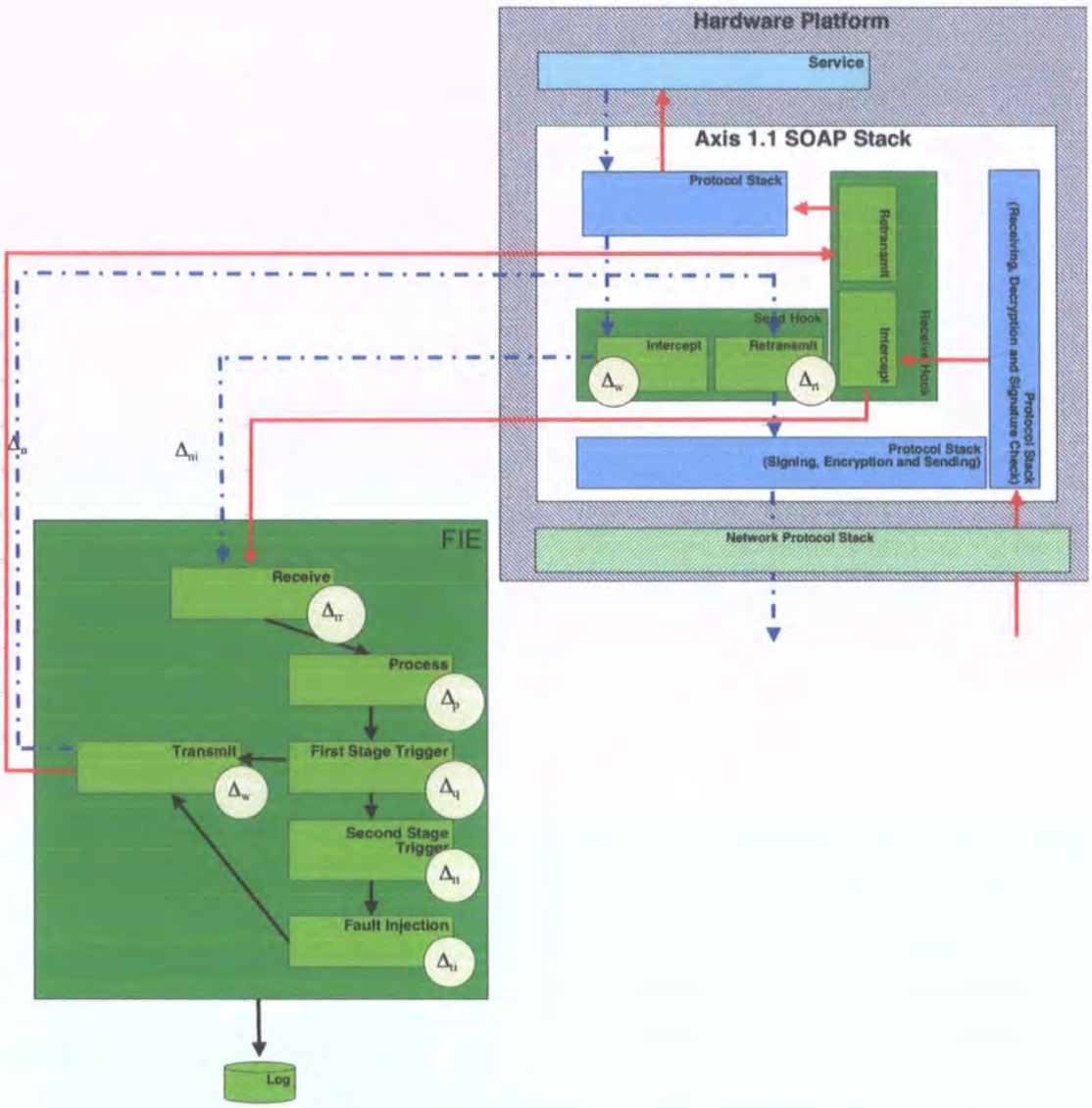


Figure 5-9: Typical System Configuration Showing Latency Terms

5.2.5 Latency Model

As described in Chapter 3 any SWIFI technique will incur some form of overhead when in use. This section quantifies the overhead introduced into a WS-FIT instrumented system under a number of typical scenarios. To achieve this we define a Latency Model that describes the temporal overheads introduced into a system under test. This model is demonstrated in Chapter 6 and is used to determine under what

conditions WS-FIT can be used with little impact on a running system and outline possible improvements to WS-FIT to further reduce its impact.

5.2.5.1 Model

To model the behaviour of the overheads introduced into a system under test, we break down WS-FIT into a number of discrete parts as shown in Figure 5-9. Each part is assigned a term which we use in our model.

As described in section 5.2.2 there are a number of discrete steps involved in injecting a fault into a SOAP message using WS-FIT:

1. Intercept SOAP message and transmit to Fault Injector.
2. Receive Intercepted message from Hook Code.
3. First Stage Trigger.
4. Second Stage Full Trigger.
5. Inject Fault.
6. Transmission of modified message to Hook Code.
7. Retransmission of modified message to remainder of SOAP Stack.

There are two main paths through the fault injection system:

1. If a fault in injected into the message all the steps listed above are followed.
2. If a message does not pass the first stage trigger then steps 1, 2, 3, 6 and 7 are executed in order.

The stages shown in Figure 5-9 have time overheads associated with them and these are shown in Table 5-3. Each of the terms given in Table 5-3 is affected to differing

degrees by the properties of the messages being processed and these are given in Table 5-4.

Table 5-3: Overhead Terms

Term	Description
Δ_{wi}	The average time taken to intercept the SOAP message in the Hook Code and write it to the socket.
Δ_{ni}	The average time taken to transmit the SOAP message across the socket from the Hook Code.
Δ_{rr}	The average time taken to read the SOAP message from the socket.
Δ_{p}	The average time taken to put it in a form that WS-FIT can process.
Δ_{q}	The average time taken to perform a first stage trigger on the SOAP message.
Δ_{tt}	The average time taken to perform a second stage full trigger on the SOAP message.
Δ_{ti}	The average time taken to inject faults into a SOAP message.
Δ_{wt}	The average time taken to write the modified SOAP message to the socket.
Δ_{nt}	The average time taken to transmit the SOAP message across the socket to the hook code.
Δ_{rt}	The average time taken to read the modified SOAP message from the socket and pass it on to the remaining part of the SOAP stack for transmission to source.

Any of the terms that involve processing a message can be affected by the triggering mechanism. There are two types of trigger:

1. A message trigger
2. A parameter trigger

There can be one message trigger per SOAP message type but there can be multiple parameter triggers for a single SOAP message type. All triggers detect certain patterns of elements within a SOAP message.

Table 5-4: Factors Affecting Overhead Terms

Term	Factors
Δ_{wi}	This step is primarily affected by message size since no processing is carried out in this step.
Δ_{ni}	This step is primarily affected by message size since no processing is carried out in this step. It may also be expected that this latency would increase in discrete steps if a fixed size packet based network is being used, since increases in latency would occur as a modulus of the packet size since packets must be transmitted even if they are not full.
Δ_{rr}	This is a function of the message size since no processing is carried in this step.
Δ_{p}	This is a function of message size since payload start and end points are being searched for.
Δ_{q}	This is a function of message size and may also be affected by the number of message triggers since each one will have to be matched.
Δ_{tt}	This is a function of message size. It may also be affected by the number of message triggers and the number of parameter triggers since these will have to be matched. Finally the number of elements within the message may also affect the result depending on the implementation used, for instance a SAX parser.
Δ_{ti}	This is a function of the number of parameter triggers on a single message, since there may be more that one, and also the time taken to actually inject a single fault into the message.
Δ_{wt}	This step is primarily affected by message size since no processing is carried out in this step.
Δ_{nt}	This step is primarily affected by message size since no processing is carried out in this step. It may also be expected that this latency would increase in discrete steps if a fixed size packet based network is being used, since increases in latency would occur as a modulus of the packet size since packets must be transmitted even if they are not full.
Δ_{rt}	This step is primarily affected by message size since no processing is carried out in this step.

5.2.5.2 Latency introduced when no fault is injected

Equation 1 gives the term Δ_{tnf} that is the overhead introduced when no fault is injected into a message. This is the latency measured from the point of entering the

Hook Code to leaving the Hook Code and returning the original message to the SOAP stack so that it can be passed down the remainder of the stack.

Equation 1

$$\Delta_{tnf} = \Delta_{wi} + \Delta_{ni} + \Delta_{rr} + \Delta_{p} + \Delta_{q} + \Delta_{wt} + \Delta_{nt} + \Delta_{rt}$$

5.2.5.3 Latency introduced when a fault is injected

Equation 2 shows the term Δ_{tif} that is the overhead introduced when a fault is injected into a message. This is the latency measured from the point of entering the Hook Code to leaving the Hook Code and returning the modified message to the SOAP stack so that it can be passed down the remainder of the stack.

Equation 2

$$\Delta_{tif} = \Delta_{wi} + \Delta_{ni} + \Delta_{rr} + \Delta_{p} + \Delta_{q} + \Delta_{tt} + \Delta_{ti} + \Delta_{wt} + \Delta_{nt} + \Delta_{rt}$$

5.3 Automatic Test Generation

Automatic test generation in WS-FIT realises the FIT method described in Section 4.3. The WS-FIT implementation must implement and populate two taxonomies to implement this method:

1. The System Model which describes the system in terms of messages exchanged to form RPCs.
2. The EFM that defines a set of fault models that can be applied to messages.

5.3.1 System Model

The first of these taxonomies is the System Model. This can be instantiated using the WSDL definitions of all the Web Services used to compose a system. Since WSDL is

XML based WS-FIT uses an XML parser to extract required information from the WSDL files. The format of these is described in Section 2.6 and Table 2-3 shows an example WSDL message. The relevant elements obtained from the WSDL are mapped onto the System Model as shown in Figure 5-10.

As described in Section 2.6 a wsdl:portType defines a collection of wsdl:operations. A wsdl:operation is defined as a wsdl:input and a wsdl:output. The wsdl:input defines the request message of an RPC and the wsdl:output defines the response message. A wsdl:message is used to define the format of the message with wsdl:part defining the parameters/return values in a specific message.

This structure maps onto the FIT System model with Services equating to wsdl:portType elements. There may be a number of Services defining the system under assesment so a number of wsdl:portType elements may have to be parsed into the system. It is also conceivable that more than one replica of a Service exists within the system (each service being on a different machine). This is dealt with by having multiple copies of the same wsdl:portType imported into the System Model and each one being tagged with the machine address of the host it is installed on. In this way triggers can be setup to only trigger on a particular wsdl:portType from a particular machine.

Each wsdl:portType is made up of a number of wsdl:operations and each wsdl:operation equates to a method in the System Model. Further each wsdl:operation contains both a wsdl:input element and a wsdl:output element and these point to the wsdl:message elements that are used at the method level in the System Model for the request and response messages.

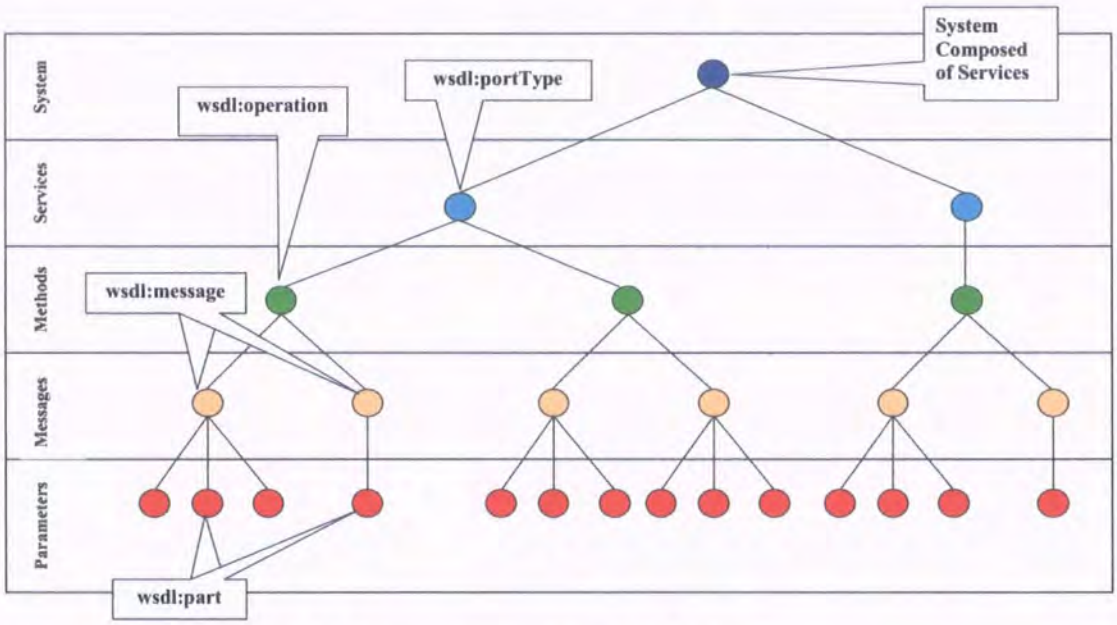


Figure 5-10: System Model Constructed from WSDL

Finally each wsdl:message is made up of wsdl:part elements. For request messages these make up the parameters and are unambiguous. For the response message we assume that the name follows the format of post-fixing the wsdl:part name with the word 'Return' to identify the return wsdl:part.

When this information is mapped onto the generic System Model given in Section 4.3 it creates a WS-FIT specific System Model. This specific System Model can be used as a basis for generating triggers described in section 5.2.2 that can identify specific RPC parameters within a SOAP message and allow faults to be injected at this level.

5.3.2 Extended Fault Model

The second taxonomy that must be implemented to realise the FIT method is the EFM defined in section 4.3. To demonstrate the WS-FIT implementation a sample EFM has been implemented. This is not intended to be exhaustive or complete but is rather to demonstrate the WS-FIT implementation.

The types of fault that can affect a Web Service based system have be classified as follows:

Physical Faults: affecting memory or processor registers

Software Faults: both programming errors and design errors. These can occur at either the application level or within the system boundary but we will only consider faults within the system boundary that may propagate to the system boundary as failures or be detected as errors inside the system boundary. The detection of errors within the system boundary assumes that the system is composed of more than one Web Service and that these services propagate the error between each other.

Resource-management faults: such as memory leakage and exhaustion of resource such as file descriptors

Communication faults: such as message deletion, duplication, reordering or corruption. Whilst in traditional distributed systems this class of errors is widely assumed to not have a large effect on middleware (since this is usually built on a reliable transport over a LAN) Web Service middleware runs over WANs that may be more unreliable that LAN based systems (especially in message delivery times)

Life-cycle faults: such as premature object destruction through starvation of keep-alive messages and delayed asynchronous responses (outside specified timing constraints).

This classification is intended to demonstrate how an EFM may be classified and is not intended to be exhaustive.

Initially WS-FIT fault models concentrated on communications faults since this fits in well with the stage one fault injector design. Subsequent evolutions of the fault models used have concentrated on Software Faults since these exercise the second stage of the WS-FIT implementation.

WS-FIT implements the EFM by allowing functional decomposition of the top-level fault model into sub-sections and finally into methods that can be applied to individual parameters. Structuring the fault model in this way aids the designing phase of the fault model and allows a more comprehensive model to be constructed. This is similar to functional decomposition techniques used in such design methods as Yourdon where three levels of decomposition are required by most projects to allow detailed designs and state machines to be constructed. This level of decomposition is arbitrary but appears to work well.

WS-FIT uses a three level decomposition:

Top level: This is a general grouping

Second level: This is a detailed grouping and there could be many of these under a top-level heading

Third level: This defines a detailed grouping group specific fault models together.

Under the third level of decomposition individual fault models can be constructed. Each fault model is defined as a name, a textual description of its function and a script to implement this function.

The script can be further decomposed into two parts:

1. A script that implements any specialized GUI and static test generation. The GUI tool executes this when the fault model is applied to a parameter. As well

as generating the GUI this is a convenient place to implement any static data, either input or randomly generated. An example of this is given in Table 5-5 in the Random Values between Upper and Lower Bounds model which requires the generation of a statically encoded sequence of random numbers. Most scripts do not require this script since only simple data is required at instantiation. For these scripts a simple GUI is used for data input that allows the entry of named types. An example of this is given in Table 5-5 Upper Bound that only requires the maximum valid value (given by a system specification).

2. A script that generates the script fragment of the fault injection script that injects the required fault. This fragment of code is inserted into the test script to form part of the test campaign. The test script is generated by 'printing' lines of text which give the lines of code that make up the test script fragment.

The scripting language used by WS-FIT is Python. In the initial prototype systems these scripts were generated by hand and run from a standard Python interpreter. In the WS-FIT implementation these test scripts are automatically generated from the EFM and can be executed natively from the GUI using the Jython package [3]. This allows scripts interoperability with the Java JVM used to implement WS-FIT and also allows them access to the GUI for entering data etc. An example of this is given in Table 5-5. This shows only one section of the fault model defined in Table 4-3 decomposed down to its most detailed level.

Table 5-5: Implemented Extended Fault Model

- Software Faults
  - Perturbation of Data into a Service
    - Values in Specified Range
      - Upper Bound
        - Replace specified parameter with the upper bound value specified for this parameter.

```
if enable == 1:
 out.println('    # Upper Bound Check')
 out.print('    return ')
 out.println(vars.getConst('upperBound'))
```

      - Random Values between Upper and Lower Bounds
        - On test generation, generate a static sequence of randomly distributed values that lie between the upper and lower bounds inclusively. Cyclically substitute the next value from the statically generated sequence for the specified parameter. A static sequence should be used to provide a level of test repeatability although this repeatability may be effected by concurrency effects and timing.

(For script see Table 5-6)

Table 5-6: Example Script

```
import string

# Check enable variable and only run generation
# script if it is true.
if enable == 1:
      # Writes a line to the generated script to
      # read the stored index variable.
      # This is used to index into the static sequence.
        out.print('    index = ')
        out.print(' self.getResults().')
        out.print('getInt(self.m_param,')
        out.println('"index")')

      # Write a line to the generated script to read
      # the static sequence
        out.print('    sequence = ')
        out.println(vars.getVarList("RandomSequence"))

      # Make sure the index for the static sequence
      # loops round at the end of the sequence.
        out.println('    if index >= len(sequence):')
        out.println('      index = 0')

      # Check for type of parameter. if it is an int
      # get rid of the floating point part.
        if vars.getConst("type").find("int") > 0:
                out.print('    value =')
                out.println('int(sequence[index])')
        else:
                out.print('    value = ')
                out.println('sequence[index]')

      # Write out a line to increment the sequence.
        out.println('    index = index + 1')

      # Write out code to save the index back into
      # stateful storage
        out.print('    self.getResults().')
        out.print('setInt(self.m_param, ')
        out.println('"index", index)')

      # Write out code to record the new value to
      # the minotor graph for this parameter/ip
      # combination.
        out.print('    self.getResults().')
        out.print('addValue("mon://Corruption of ')
        out.print('Data into Web Service/Values in')
        out.print(' Range/Random Values/%IP%", ')
        out.println('str(value))')
        out.println('    return str(value)')
```

5.4 Automatic Failure Detection

As described in Section 4.4 the FIT method defines an EFAM that is comprised of a number of discrete failure modes classified into a taxonomy. As in Section 5.3 the EFAM described here is intended to demonstrate WS-FIT rather than be an exhaustive model that can be applied to any Web Service system.

The EFAM is built upon a set of high-level failure modes. These classify how a Web Service can fail. The high-level classifications are:

1. Crash of a Web Service
2. Crash of a hosting Web Server
3. Hang of a Web Service
4. Corruption of data into a Web Service
5. Corruption of data out of a Web Service
6. Duplication of messages
7. Omission of messages
8. Delay of messages

The effect of each of these modes will depend on the fault tolerance of the system as a whole. We can assume that a well-written Web Service should be able to detect and reject any corrupted data given to it and raise appropriate error responses. Although the rejection mechanism can vary we assume that the majority of error conditions are relayed by SOAP Fault messages since these equate to Java exceptions and the prevalent language in use in the construction of Web Services in Java.

We can further assume that the detection of duplicated and omitted messages generated by fault transmission mediums will be handled by the Network Protocol Stack. Since these failure modes do not generate any traffic at the middleware layer their detection by WS-FIT will not be possible. WS-FIT can be employed to detect duplicated or omitted messages at the SOAP stack level which could result from such activity as security attacks, for instance replay attacks.

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 through a SOAP stack it can be intercepted and examined. A detailed knowledge of the logic of the Web Service and its current state is required to determine if a result returned by a Web Service is correct. The WS-FIT implementation of a EFAM uses available range data, supplied manually from the system specification, to determine if the value is in range. This will not catch all failures but it will give an indication of failure.

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 service life span issues. The Web Service middleware should present relevant error conditions to the application for

delayed messages. The application layer must ultimately handle these. WS-FIT can handle this by delaying request/response packets.

The High Level EFAM used to demonstrate WS-FIT is therefore defined in Table 5-7.

Table 5-7: High Level Failure Modes

- Crash of a Web Service
- Crash of a hosting Web server
- Hang of a Web Service
- Corruption of data into Web Service
- Corruption of data out of Web Service
- Duplication of messages
- Omission of messages
- Delay of messages

As described in Section 4.4 this set of high-level groupings is decomposed into a taxonomy until a level is reached where a detailed description of the failure mode can be written. A generic script is then written to generate code for the fault injection campaign script that can detect the defined failure mode when applied to a parameter. An example of this is given in Table 5-8.

Table 5-8: Implemented Detailed Failure Model

- Corruption of data out of service
  - Data out of range
    - Data above upper bound
      - Check specified parameter against upper bound in specification and if it is greater than this value flag an error condition.

```
If value > maxLimit:
  Flag error in log
```

    - Data below upper bound
      - Check specified parameter against lower bound in specification and if it is less that this value flag an error condition.

```
If value < minLimit:
  Flag error in log
```

This EFAM could then be applied globally to all messages defined in the System Model (see Section 5.3.1) and would provide detection of unexpected failures.

To detect failures generated by specifically injected faults, injected by linking the System Model to the EFM, a link must be made from the EFM to the EFAM script corresponding to the expected failure mode as described in Section 4.4. The expected outcome from the fault injection may be no failure detected since a fault tolerance mechanism may be in place.

In our implementation this stage of WS-FIT was implemented by post processing the generated logs manually but this facility could be integrated into the WS-FIT tool in future work.

5.5 Summary

This chapter has detailed an implementation of the FIT method detailed in Chapter 4 that has been applied to Web Services, specifically using SOAP as the middleware protocol.

It describes how the message based injection mechanism and triggering can be implemented to work with SOAP middleware and how this can be used to realise the method described in Section 4.2 by the use of Hook Code to intercept SOAP messages, decode and manipulate messages.

WS-FIT demonstrates how the EFM (Section 4.3) can be implemented using a Python based scripting language. A system model can be created by importing WSDL for the Web Services making up an SOA and mapping certain WSDL elements to certain elements within the System Model taxonomy. The System Model can then be used to create triggers which the EFM can be applied to.

Finally the EFAM (Section 4.4) has been described and how this can be implemented via WS-FIT to detect failures in an SOA. An outline EFAM is shown as a demonstration.

Chapter 6 - Case Studies

This chapter describes a number of case studies that demonstrate different areas of the FIT method through use of the WS-FIT tool. The case studies are divided into two groups:

1. Case studies demonstrating the operation of the Fault Injection Mechanism (Section 6.1)
2. Case studies demonstrating the application of the FIT method to SOA (Section 6.2)

The first set of case studies demonstrating the operation of the Fault Injection Mechanism are as follows:

- Fundamental Operation of WS-FIT (Section 6.1.1)

  This demonstrates the basic features of the method:

  1. Fault Injection into a middleware message
  2. Trigger Mechanism
  3. Simple Parameter Perturbation

- Latency Model (Section 6.1.2)

  This demonstrates the latency model outlined in Section 5.2.5.

- Comparison of FIT with Code Insertion (Section 6.1.3)

  This compares WS-FIT to another fault injection technique.

The second set of Case Studies demonstrating the application of the FIT method to SOA are as follows:

- Fault Generation and Failure Detection (Section 6.2.1)

  This case study demonstrates test campaign generation and failure detection through the:

  1. Application of the Extended Fault Model for test campaign
  2. Application of the Extended Failure Model for failure detection
  3. Parameter Perturbation

- Assessment of a Fault Tolerance Mechanism (Section 6.2.2)

  This case study demonstrates the use of parameter perturbation to assess the dependability of a fault tolerant SOA using the 'Software Faults' categorization of the Extended Fault Model and also the 'Communications Fault' categorization of the Extended Fault Model. This Case Study also shows how the Extended Failure Model can be applied.

  1. Application of the Extended Fault Model to assess a fault tolerant SOA

- Application of Communications Faults to an SOA (Section 6.2.3)

  This demonstrates how WS-FIT can be used to apply fault models which are categorized as 'Communication Faults' rather than 'Software Faults'.

  1. Application of the Extended Fault Model to a real-time scenario

6.1 Fault Injection Mechanism

The case studies in this section demonstrate the main features of the FIT method through the WS-FIT tool. The key features demonstrated are:

1. Injection into a message
2. Trigger Mechanism
3. Parameter Perturbation
4. Acceptable Latency Overhead
5. Comparison to Code Insertion

6.1.1 Fundamental Operation of WS-FIT

This case study demonstrates the basic operation of the WS-FIT implementation. The features exercised are:

1. Injection into a message (rather than individual network packets)
2. Triggering mechanism
3. Parameter perturbation

These features form the first stage fault injector on which the FIT method is based.

6.1.1.1 Scenario

This scenario is based around a simple client and server. The server exports a service constructed using simple data types that is utilized by a client. A SOAP application was written and deployed onto the server machine that provided a simple set of routines that calculated a simple integer sequence of numbers. An interface method was defined to return this sequence one value at a time. A SOAP client was written which polled back

the sequence of numbers via the Apache SOAP API. These two simple applications form the test-bed for the experiments described here.

6.1.1.2 Configuration

The middleware package used was Apache SOAP 2.2 with the container being provided by Apache Tomcat 4. Both the server and the client were hosed under Redhat Linux 8.0 running Java version 1.3.1.

6.1.1.3 Results of the First Test Script

This test was intended to check the middleware under a combination of fault conditions. A script was written to inject the following faults into the stream of SOAP messages:

1. Every 10th sequence request from the client to the server, corrupting the encoding styles schema address
2. Every 15th sequence response from the server to the client, corrupting the encoding styles schema address
3. Discarding every 3rd sequence response from the server to the client
4. Every 7th sequence request from the client to the server, inserting extra text into the <SOAP-ENV:Body> element after the method element

The number of faults injected into the system by this test is approximately 25% of the total number of packets sent and received by the fault injector (see Table 6-1). Approximately 17% were corrupted SOAP packets, with the remainder being withheld response packets.

The test was performed using a loop iteration of 1000, 15000, 45000 and 90000 and was repeated 5 times for each loop iteration. There was no variation in results for each set of results produced for a particular loop iteration value but there was a very small variation in percentage totals between loop iterations (see Table 6-2).

Table 6-1: Faults injected by test 1

Loop Iterations	Total Packets	Faults Injected (%)	Responses Discarded (%)	Total Faults Injected (%)
1000	5551	17.26	8.11	25.37
15000	83251	17.26	8.11	25.36
45000	249651	17.26	8.11	25.35
90000	499501	17.25	8.11	25.35
Variance		0.00	0.00	0.00

The fault injector recorded the following classes of faults:

1. Failure expected and generated
2. Failure expected but not generated
3. Failure generated but not expected.

A fourth category of response to injecting a fault is also inferred. This is No Failure expected, and is deemed to have occurred if a fault is injected into the system but no fault message is expected to be generated. This condition could occur when a response is sent back from the server to the client but the fault injector discards the message. The client cannot generate a fault message because it has no way of knowing that a response message is expected.

Of the failure conditions generated (see Table 6-2), the 7% of failure expected and received were generated by the schema corruption case as expected. The 31% of no failure expected failures were attributed to the discarding of response packets from the

server to the client. This indicated a level of reliability within the middleware since all these error conditions appear to have been handled correctly within the middleware and the server remained stable. The reliability of the middleware was only assessed over a period of approximately 24 hours, due to time constraints, so our data can only indicate a level of reliability during this period of time and these conditions [37].

Table 6-2: Failures Detected by test 1

Loop Iterations	Failures Generated (%)	No Failures Expected (%)	Failures Missing (%)
1000	7.10	31.96	60.94
15000	7.11	31.98	60.92
45000	7.11	31.98	60.91
90000	7.11	31.98	60.91
Variance	0.00	0.00	0.00

The 60% failures missing figure were generated by the inserting of text into the SOAP-ENV:Body element. A failure was expected because the fault would break the SOAP specification although it was syntactically correct according to XML but no fault messages were generated. Although this was a small deviation from the SOAP specification it did not appear to affect the reliability of the middleware over the lifespan of the test, which continued to function correctly in all other respects. This could be caused by the SOAP implementation not installing a handler for the specific element body, since it never expects any data there according to the specification. Consequently this explains the continued functioning of the SOAP implementation because data within the SOAP-ENV:Body tag would just be silently discarded.

This fault condition can therefore be classed as a non-catastrophic event but caused by a rare event. We have classed this event as rare because its likelihood of happening outside of our fault injection experiment is extremely remote. It would require a

corresponding bug in the SOAP implementation connected to the SOAP implementation receiving the message to generate the event. Its effects on the system as a whole seemed to be negligible during the lifespan of the test since the test case caused no server or service crashes and no application crashes.

It was noted that the client application returned error conditions when faults were injected and its performance was degraded slightly due to a combination of timeout effects when response messages were discarded and the latency introduced by the fault injector parsing SOAP messages.

Overall the middleware appeared to have a high degree of dependability since it continued to operate and service requests even when generating error conditions for which handlers had not been implemented in the SOAP implementation. Furthermore the test was executed over a period of 24 hours and no cumulative memory leaks or similar effects were seen. More data is required on determining a suitable duration for a test to determine the dependability of the middleware over long periods of time but this experiment would seem to indicate that the middleware does have a level of dependability when errors are present in the communications medium.

6.1.1.4 Results of the Second Test Script

The intent of this test is to check the dependability of the middleware with a simulated bad connection. A script was written that discarded every other request and response. This caused approximately 50% of the SOAP messages, both requests and responses to be discarded (see Table 6-3). A trigger was constructed to do this based on the message type and the frequency of the message. The message type was obtained from the WSDL definition for the Web Service.

Table 6-3: Faults injected by test 2

Loop Iterations	Total Packets	Messages Discarded (%)	Total Faults Injected (%)
1000	4001	49.99	49.99
15000	60001	49.99	50.00
45000	180001	50.00	50.00
90000	360001	50.00	50.00
Variance		0.00	0.00

The same loop iteration and run repeats were used in Test 2 as in Test 1. There was, again, no variation in the results obtained within a repeated test with the same number of loop iterations. There is a small variance in % totals between loop iterations.

Table 6-4: Failures generated by test 2

Loop Iterations	Failures Generated (%)	No Failures Expected (%)	Failures Missing (%)
1000	0.00	100.00	0.00
15000	0.00	100.00	0.00
45000	0.00	100.00	0.00
90000	0.00	100.00	0.00
Variance	0.00	0.00	0.00

Of the fault conditions detected 100% of the faults were of the class No Failures Expected (see Table 6-4). This is the expected result of the test since discarding a request message from client to server, from the server's point of view, will be the same as never having received the request. Discarding a response message from server to client generates no network fault messages since this is handled by a time out. The client application was manually monitored to verify that the correct error responses were given.

This test appeared to indicate that the middleware under test has a high degree of resilience to SOAP message loss over an unreliable network during the duration of the test, since the middleware continued to operate correctly under the presence of these

faults. No unexpected results were obtained indicating a level of dependability under this test case. Discarding the SOAP messages seemed to cause no adverse effects on either the client or the server such as server crashes/hangs.

6.1.1.5 Parameter Perturbation Modification

A test script was written to allow application return values to be changed. This experiment allowed the return data to be modified, whist still returning a valid SOAP response message. These tests generated no failures and the modified results were seen at the client program.

A trigger was set which detected first a specific message (the response message from the server) and then a specific element within the message (the return parameter). This allowed the contents of the element containing the parameter to be modified without invalidating the syntax of the message. This in turn allowed the message to be passed across the network interface and be received and processed normally by the client protocol stack.

This test was carried out to ensure that the fault injector infrastructure could reliably handle modifying SOAP parameter data, whilst still returning correctly formatted SOAP response messages.

6.1.1.6 Evaluation

This case study has demonstrated that the basic mechanisms employed to implement FIT can be applied to Web Services written using SOAP.

It firstly demonstrated that WS-FIT can inject faults at a middleware message level rather than a network packet level by instrumenting the protocol stack at the appropriate level within the stack. This case study also demonstrated that WS-FIT can be used to

injected faults which corrupt the syntax of a message which is akin to more traditional network level fault injection although it is not intended to be the main use of WS-FIT. This case study also demonstrated a potential weakness in the SOAP implementation under evaluation (not detecting extra text outside of elements in a SOAP message).

Secondly this case study has demonstrated that the triggering mechanisms can target specific messages within an RPC exchange between a client and a server by parsing intercepted messages and determining the message type. The message type can be obtained from the WSDL definition of a Web Service.

Thirdly it has demonstrated that WS-FIT can target specific parameters within an RPC exchange and inject faults into them so that the messages remain syntactically correct. This allows faults to be injected and processed by a SOAP stack as normal valid messages, thus allowing perturbed parameters to be propagated through the middleware layer to clients or Web Services. WS-FIT is therefore capable of injecting API level faults into a middleware system without the need for modification to the client or Web Service code.

6.1.2 Latency Model

These test cases will be used to demonstrate conditions under which the latency introduced by WS-FIT can be regarded as acceptable. This will indicate the types of SOA that can be assessed using WS-FIT without the instrumentation adversely affecting the system. A formal proof will not be attempted but rather demonstrate when various terms in the model detailed in Section 5.2.5 become significant.

The test cases will first establish an empirical measurement of a RPC execution time and subsequent test cases will be compared against this. We will go on to demonstrate

under what conditions various terms become significant and which terms can be disregarded.

6.1.2.1 Scenario

The test cases use a simple SOA that demonstrates types and interconnections that WS-FIT supports. The system is deliberately simplistic so that timings can easily be measured.

The scenario utilises a test Web Service called *LatencyTest*. It has two main groups of service methods:

- Ten service methods that take an integer parameter as input and return the integer parameter as the output. This group of methods are called sequentially in a loop to allow both multiple message triggers and parameter triggers to be added. Each service call is comprised of two messages (a request and a response) and each message contains one parameter, either a call parameter or a return parameter. The SOAP messages that make up these exchanges contain a relatively small number of XML elements.
- A single service method that takes an array of strings as input and returns the same array of strings as the output. This method is used to provide a message with a large number of XML elements.

6.1.2.2 Configuration

The test scenario utilizes three machines to minimise interference effects from machine loading. Two 1.2Ghz AMD Sempron based systems, each with 256MB of RAM running Fedora Core 3 Linux were used to host the LatencyTest service and the client. A separate 1Ghz G4 based system with 392MB of RAM running MacOS 10.3

was used to run WS-FIT. Various combinations of connection were used to simulate different network configurations (See Table 6-5 and Table 6-6).

Table 6-5: Configuration 1

	Client	**LatencyTest Service**	**WS-FIT**
Client		1 Mb Broadband Internet	
LatencyTest Service	1Mb Broadband Internet		100Mb Ethernet
WS-FIT		100Mb Ethernet	

Web Services were hosted using the following software:

- Apache Jakarta Tomcat 5.5.4
- Sun Microsystems Java 2 Platform, Standard Edition, Version 1.4.2.06
- Middleware was provided using:
- Apache Axis Version 1.1
- Sun Microsystems JavaMail Version 1.3.2
- Sun Microsystems JavaBeans Activation Framework Version 1.0.2
- Apache XML Security Version 1.2.0

Two network configurations were used. One to simulate interconnections between Web services over an Internet (Table 6-5) and one to simulate a more controlled network environment (Table 6-6).

Table 6-6: Configuration 2

	Client	LatencyTest Service	WS-FIT
Client		100Mb Ethernet	
LatencyTest Service	100Mb Ethernet		100Mb Ethernet
WS-FIT		100Mb Ethernet	

6.1.2.3 Internet latency

The data shown in Table 6-7 was gathered using Configuration 1 (See Table 6-5). This data is empirical and is not intended to be an exhaustive study but to give a rough baseline to compare WS-FIT instrumented services to. The first group of services was used to gather data. Each group of ten service method calls was iterated 1000 times and averages were taken of this data.

Table 6-7: Timings of RPC Across an Internet

	Average (ms)	Standard Deviation (ms)	Min (ms)	Max (ms)
Normal 1	167.72	19.64	92.07	3808.07
Normal 2	167.48	28.76	90.78	6041.78
Loaded Connection	374.16	834.13	89.95	17579.95

The total time taken to perform an RPC was obtained by recording the time just prior to making the RPC call and the time just after the RPC returned and subtracting the two. This time includes the time taken by the SOAP stack to process the RPC but this time should be a constant if the same environment is used to run subsequent test.

Table 6-7 shows three measurements of total RPC execution time across an Internet connection. The measurements labelled Normal were taken using a client and server

connected by a lightly loaded connection. Since the Internet connection between the two machines could not be regulated some variation between the results is to be expected as can be seen in the results.

The measurements labelled Loaded Connection were obtained with a loaded network connection to show how this can affect round trip RPC times. Whilst the test was running, a large file was intermittently transferred from the server running the LatencyTest service. This shows that there can be a great variation in RPC execution time according to the loading of the Internet connection. The value given by this measurement should be considered extreme and a smaller execution time should be expected. Axis 1.1 uses a default timeout value of 10 seconds.

6.1.2.4 Triggering

These tests are used to assess the impact of the WS-FIT trigger mechanism on a running system. The results were generated by calling the group of ten similar service methods in a 1000 iteration loop and applying a sequence of different trigger setups each time. The trigger setups ranged from no triggers to 20 triggers (one for each parameter and return the values). The test run by the trigger was a random corruption test as defined in the fault generic fault model given in Table 5-5. The SOA used network configuration 2 (see Table 6-6) to eliminate major timing variations introduced by network transfers.

The SOA under test was instrumented as shown in Figure 6-1. WS-FIT was instrumented to provide timings for Δ_{rr}, Δ_{p}, Δ_{q}, Δ_{tt}, Δ_{ti} and Δ_{wt}. The client side Hook Code was also instrumented to record the client side Δ_{wi} and Δ_{rt}. Since the synchronization of the clock on the machines used could not be guaranteed it was not

possible to measure Δ_{ni} and Δ_{nt} directly. We have assumed that Δ_{ni} and Δ_{nt} are approximately the same since similar amounts of data will be transferred across the same network link. We therefore approximate these values by subtracting all the other measured terms from the total RPC execution time to get $\Delta_{ni} + \Delta_{nt}$.

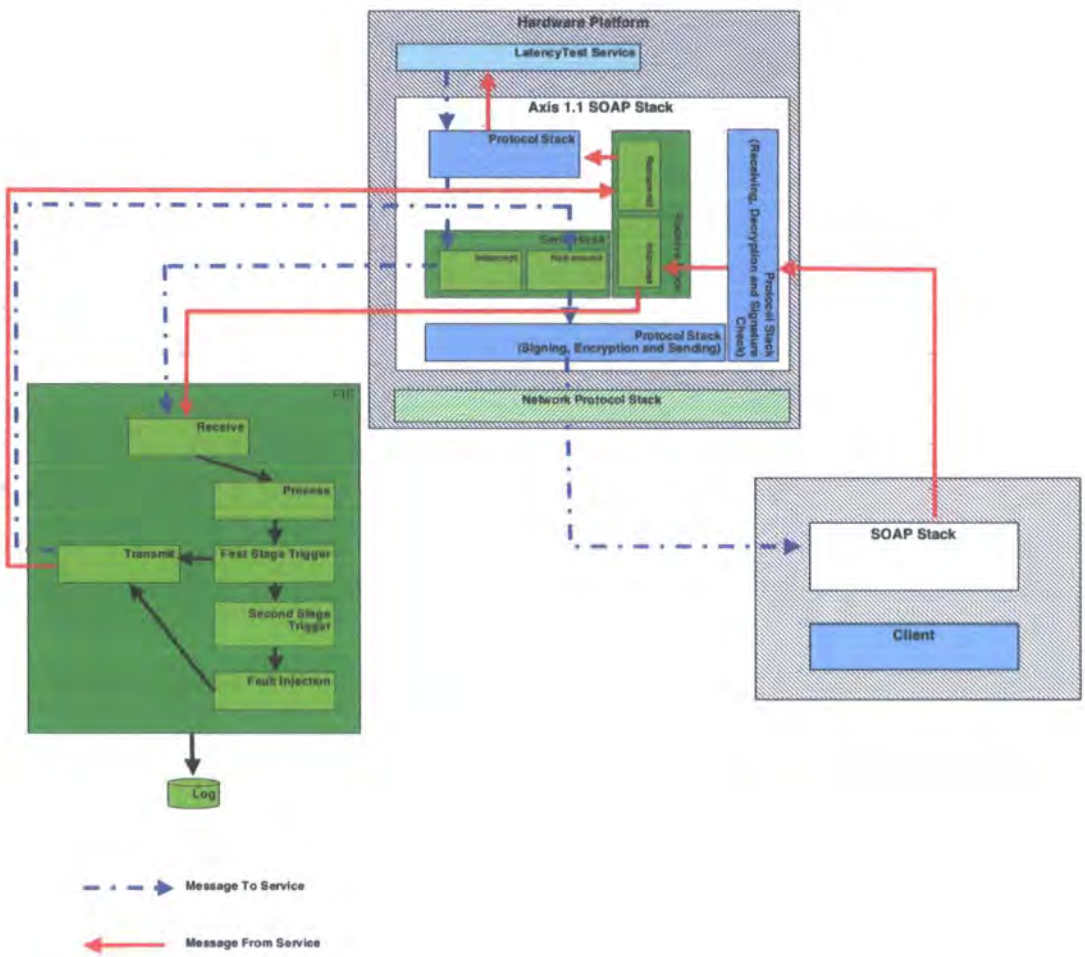


Figure 6-1: Basic Fault Injector Configuration

Figure 6-2 shows the average RPC execution time for a call to LatencyTest. The values displayed are averaged over the loop iteration. For the tested number of triggers (20) the increase in latency appears to be linear, with an equation of:

$$y = 4x/10000 + 0.0379$$

This makes the latency injected into the system less than the standard deviation for the internet transfer time given in Table 6-7 which should be acceptable for most applications since all Web Service systems should be designed to tolerate this level of latency due to the relatively unpredictable nature of Internet transfer times.

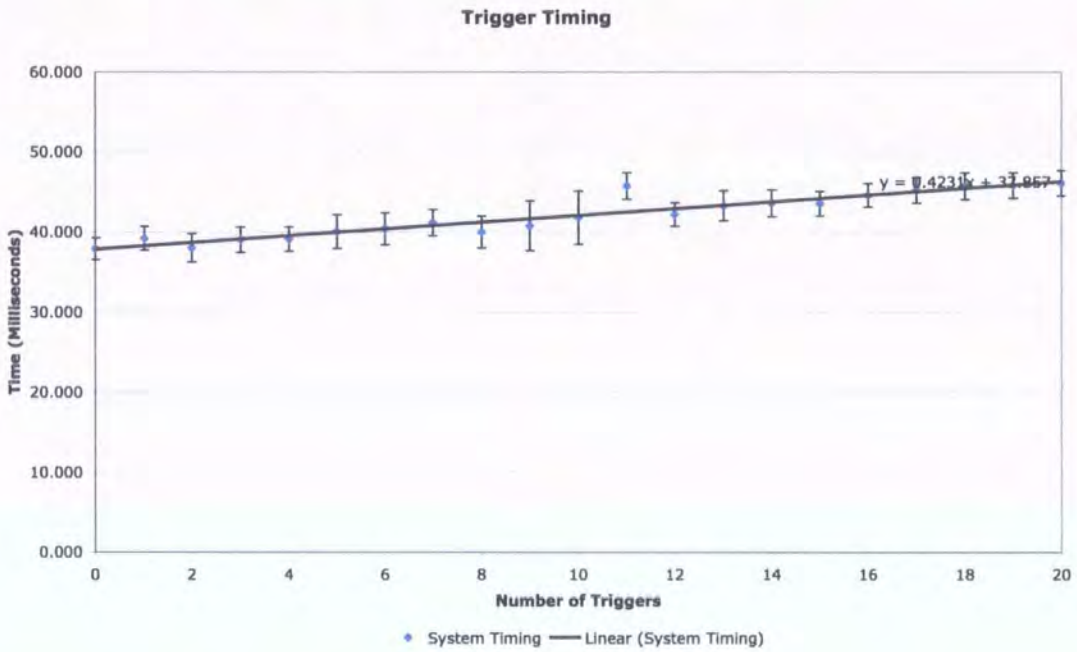


Figure 6-2: Total RPC execution timings

A more detailed breakdown of timings is given in Figure 6-3. This graph gives each term as a percentage of the total. From this we can see that Δ_p is the most significant term, ranging from 66% to 83% making it the most significant factor when processing messages with small numbers of elements. This is due to an amount of processing required after reading the data sent from the Hook Code to WS-FIT to put it into a form readable by the SAX parser. This requires several string searches to find certain points in the message and hence is relatively constant throughout this test since the size of the message in bytes is relatively constant. We can also see that whilst Δ_{tt} increases Δ_{ti} remains roughly constant.

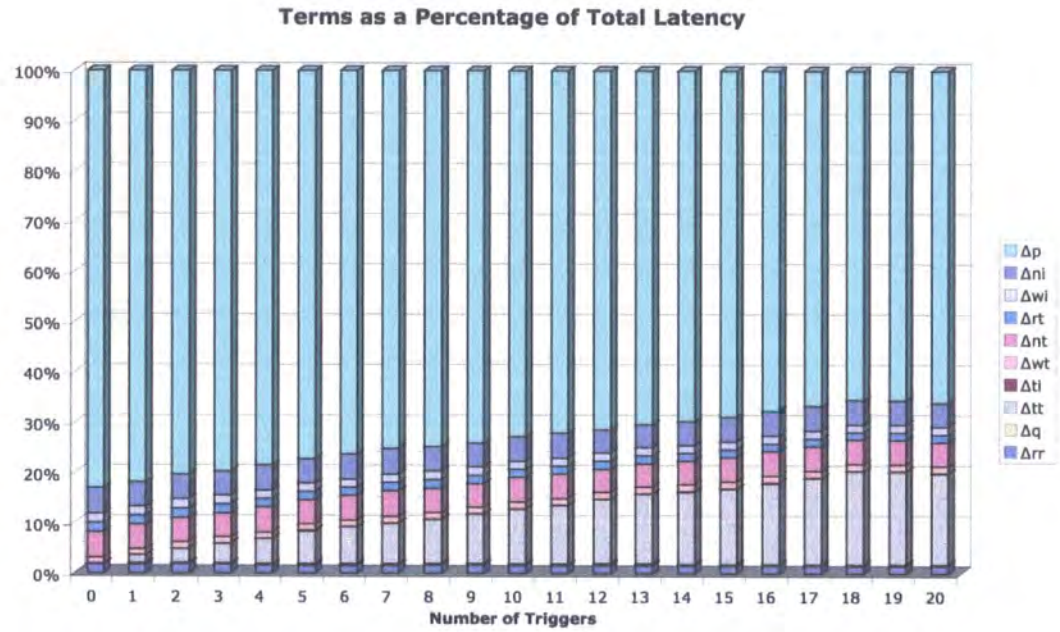


Figure 6-3: Terms as a Percentage of Total Latency

6.1.2.5 Multiple Elements

This set of results is used to assess the impact of the WS-FIT trigger mechanism on a running system when the messages contain a large number of elements. The results were generated by calling the LatencyService method that takes an array of strings and returns an array of strings in a 1000 iteration loop.

Three tests are preformed:

1. Execute test with no WS-FIT instrumentation which provides a baseline to compare the following tests against;
2. Instrument the system but do not apply any triggers which allows us to measure the latency of the first part triggering mechanism;
3. Apply a single trigger.

A range of array sizes were tested for ranging from and array containing 1 element to an array containing 500 elements. The system was setup to use network configuration 2 (see Table 6-6) to eliminate major timing variations introduced by network transfers.

The data given in Figure 6-4 shows the total RPC execution timings for the system running with no WS-FIT instrumentation. This data shows a small linear execution time increase over the given array size range.

$y = 3x/10$

The data given in Figure 6-5 shows the system running with WS-FIT instrumentation but no active triggers. The data shows a polynomial increase in RPC execution time of

$y = 3x^2/100000 + 41x/10000$

For this test the latency rise is dictated by the size of the message since it must be scanned by the first stage trigger that gives a character based latency.

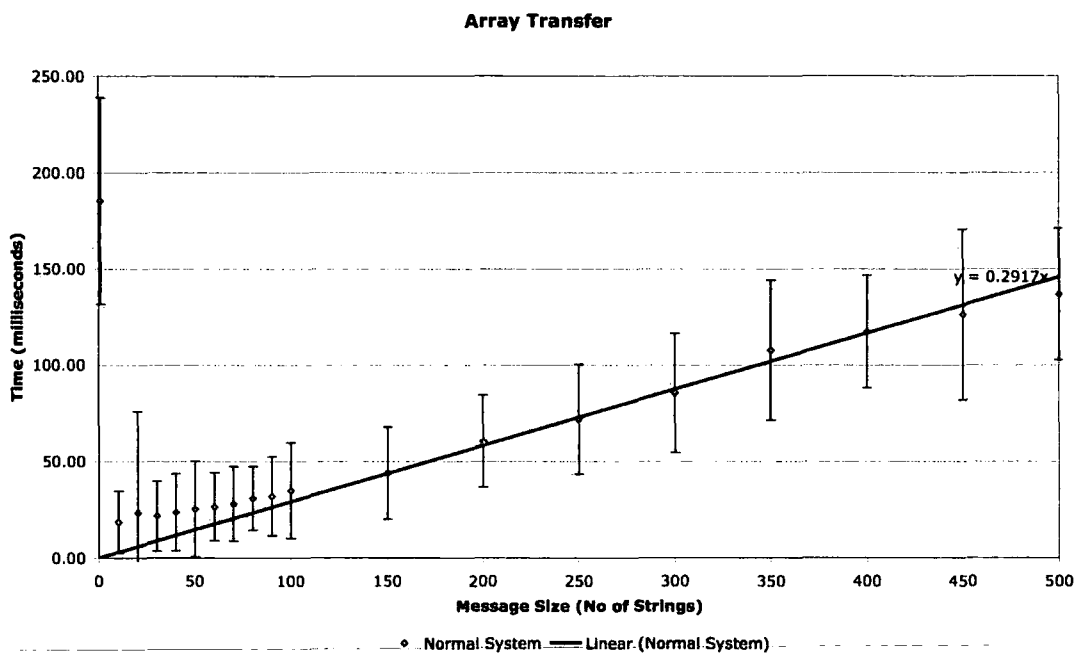


Figure 6-4: Array Transfer Timing

Figure 6-5: Array Transfer Timing with WS-FIT Instrumentation

6.1.3 Comparison of FIT with Code Insertion

This case study compares WS-FIT to Code Insertion fault injection and demonstrates that it can achieve comparable results. This case study also demonstrates the value of WS-FIT in providing a framework for certification testing of Web Services. The test case simulates a complex real-world scenario and WS-FIT is used to locate a defect in the design of the SOA.

6.1.3.1 Scenario

Our system is based on a simulation of a self-regulating heating system (see Figure 6-6). The system is composed of three main elements:

1. Heater coil
2. Thermocouple
3. Controller

A driver to both the heater coil and the thermocouple are provided via a Web Service. 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.

6.1.3.2 Configuration

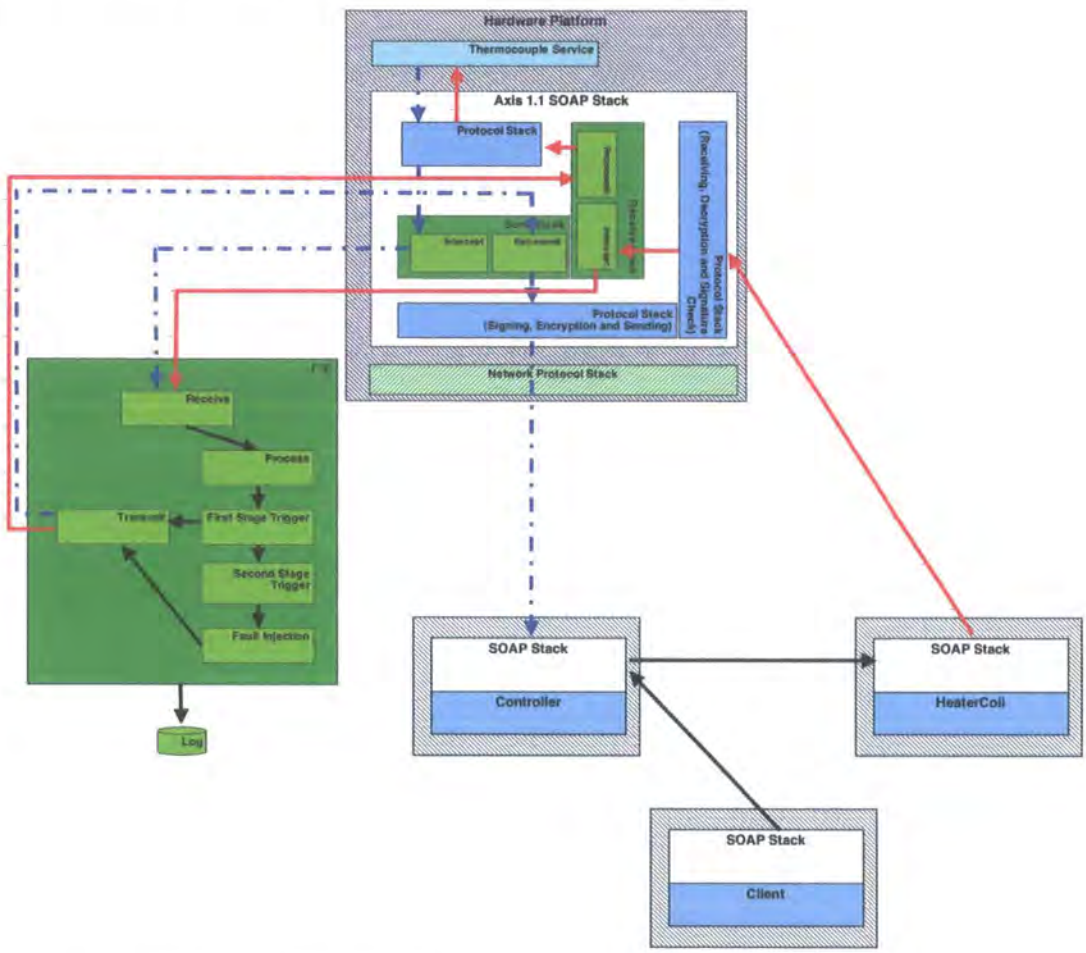


Figure 6-6: SOA with WS-FIT Instrumentation

Each Web Service is hosted on a separate server as shown in Figure 6-6. The Thermocouple service is hosted on an instrumented server whilst the HeaterCoil service is hosted on an unaltered server.

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 simple state machine is implemented by the controller 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.

6.1.3.3 WS-FIT Experiment

As shown in Figure 6-6 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 6-6). 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 was constructed to monitor temperature response messages between the *Thermocouple* and the *Controller*. A trigger was created 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 and 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.

6.1.3.4 Code Insertion Experiment

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 6.1.3.3.

Two points were identified for Code Insertion in this scenario but in practice with a complex SOA many more insertion points would potentially be needed, for instance where RPC calls are called from multiple places in the code. Inserted code is marked in grey on Figure 6-7.

```
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())) *
        POWER_TO_TEMP;
} catch (MalformedURLException e) {
    e.printStackTrace();
    throw new RemoteException(
      e.getMessage());
  } catch (ServiceException e) {
    e.printStackTrace();
    throw new RemoteException(
      e.getMessage());
  }
}
```

Figure 6-7: Instrumented Thermocouple Routine

The first insertion point was in the getTemp routine in the *Thermocouple* service (See Figure 6-7). 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.

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.

6.1.3.5 Results

Three series of data were collected:

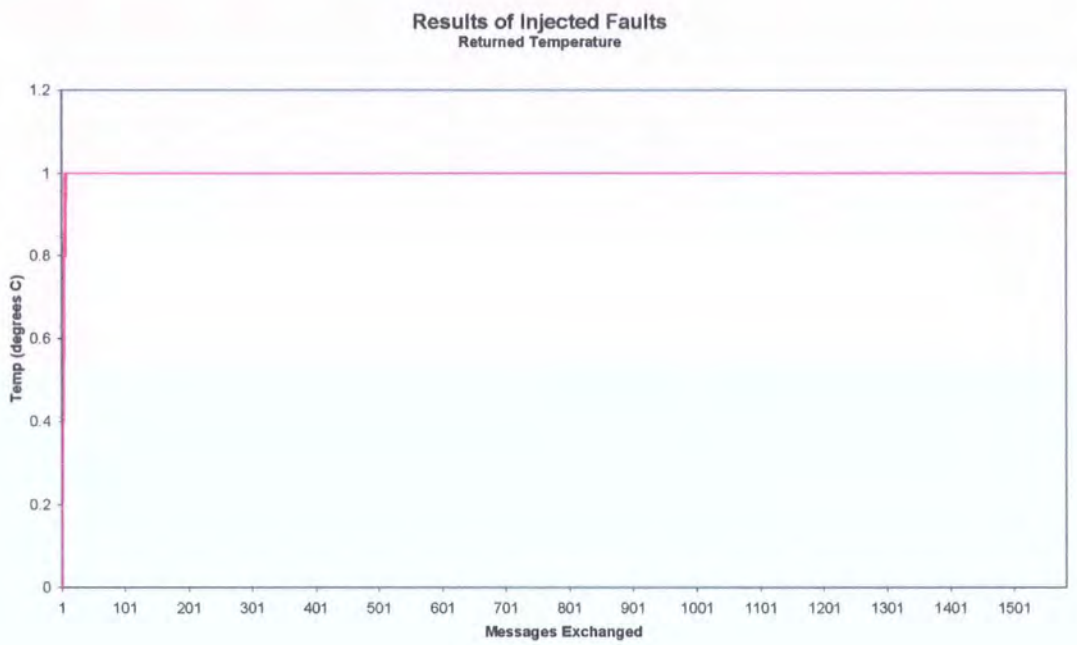
1. Control experiment (Figure 6-8)
2. Fault injection experiment using WS-FIT (Figure 6-10)
3. Fault injection experiment using Code Insertion.

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 6-8) indicates that the system functions according to the state machine given in Section 6.1.3.1. This experiment gave us a basis for comparison with the following fault

injection experiments.

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*
3. The power reading supplied to the *Thermocouple* with a fault injected into it.



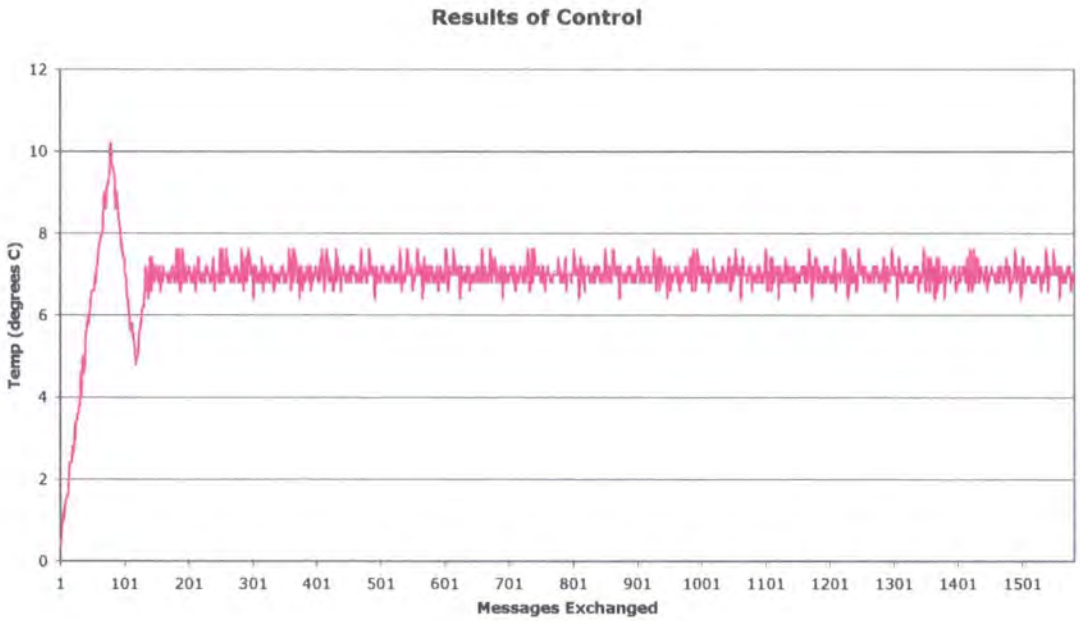


Figure 6-8: Control Temperatures

Figure 6-9: Returned Temperature with Fault Injected

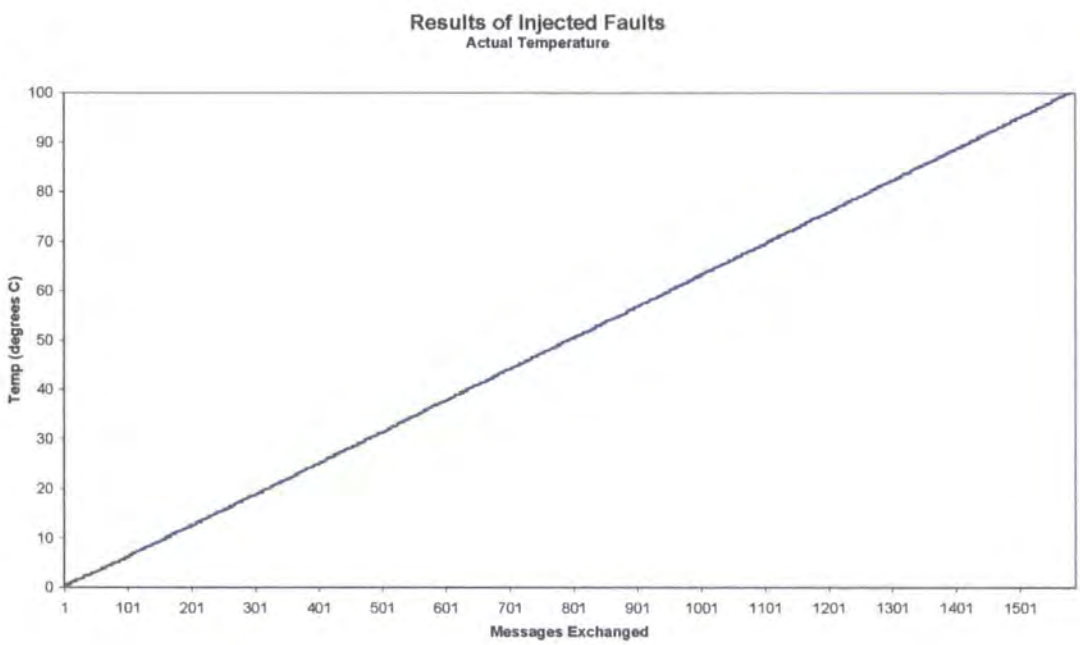


Figure 6-10: Actual Temperature of Heater Coil

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 6-8 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 6-10) demonstrates 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 6-10).

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.

6.1.3.6 Evaluation

This case study has demonstrated how the FIT method 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 that one place.

6.2 Application of FIT to SOA

The case studies in this section demonstrate the application of the method FIT method through the WS-FIT tool to SOA. The key features demonstrated are:

1. Application of the Extended Fault Model for test campaign generation
2. Application of the Extended Failure Model for failure detection

6.2.1 Fault Generation and Failure Detection

This case study demonstrates test campaign generation and failure detection through the:

1. Application of the Extended Fault Model for test campaign generation
2. Application of the Extended Failure Model for failure detection
3. Parameter Perturbation

6.2.1.1 Scenario

To provide a test bed to demonstrate WS-FIT a test system has been constructed that simulates a typical stock market trading system (See Figure 6-11). This system is composed of a number of elements:

1. A Web Service to supply real-time stock quotes
2. A Web Service to automatically trade shares
3. A bank Web Service that provides a simple interface to allow deposits, withdrawals and balance requests
4. A client to interact with the SOA

The stock quote service is implemented to use a large repeatable dataset, stored in a backend database to produce a time based real-time stock quote. Since the quote service is based around a database containing the simulated quote values it is possible to replicate a test run exactly by resetting time etc. to a set of starting conditions. The trading service implements a simple automatic buying and selling mechanism. An upper and lower limit is set which triggers trading in shares. Shares are sold when the high limit is exceeded and shares are brought when the quoted price is less than the lower limit.

The buying and selling process involves transferring money using the bank service and multiple quotes (one to trigger the transaction and one to calculate the cost of the transaction). Since these multiple transactions involve processing time and network transfer time this constitutes a race condition as the quoting service produces timed real-time quotes. Any such race condition leaves the potential for the system to lose money since the initial quote price may be different from the final purchase price.

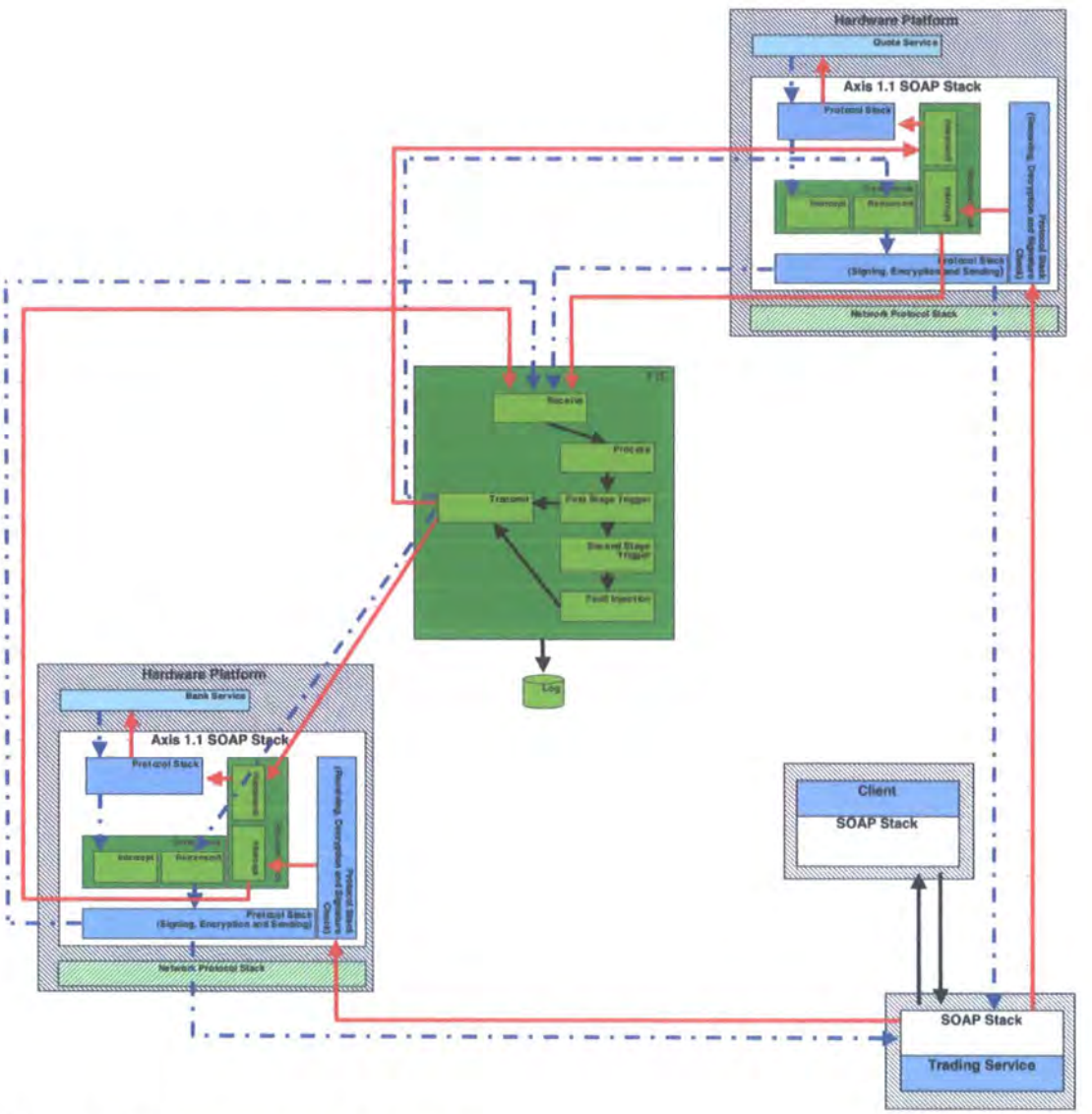


Figure 6-11: Instrumented System

6.2.1.2 Configuration

This paper details three different series of data:

1. A baseline set of data with the system running normally
2. A simulated faulty/malicious service
3. A simulated heavily loaded server

The test system was implemented using Apache Tomcat 5.0.28 with Web Services implemented using Apache Axis 1.1, hosted on Fedora Linux Core 2 running on 2Ghz IA-32 Processors.

Since the Extended Failure Model has not been implemented in WS-FIT at present it has been applied by hand. The method described by the Extended Failure Model has been implemented either by writing specialized scripts that detect failures in real time or post processing of log information to flag failures.

6.2.1.3 Baseline Test

The baseline test is designed to demonstrate the system running under normal conditions. This provides a series of data to compare further test cases against. We instrument the system for all tests. This instrumentation allows faults to be injected into the system and also monitor the RPC exchanges between Web Services (See Figure 6-15). It has been demonstrated in Section 6.1.2 that the latency introduced by this instrumentation is negligible when compared to Internet message transfer times involved in a SOAP based SOA.

Table 6-8: Baseline Test Series

	Test 1		*Test 2*		*Test 3*		*Test 4*		*Test 5*	
	Match	*Miss*	*Match*	*Miss*	*Match*	*Miss*	*Match*	*Miss*	*Match*	*Miss*
Match %	99.00	1.00	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00
Average Time	0.05	0.06	0.05		0.05		0.05		0.05	
Std Dev	0.03		0.03		0.03		0.03		0.03	

The first series of data collected from the normally running system allows us to verify that the system operates according to its specification. Table 6-8 shows a summary of

the results collected and demonstrates that the deviation between the original quoted prices and the transaction completion price is negligible 0.2%.

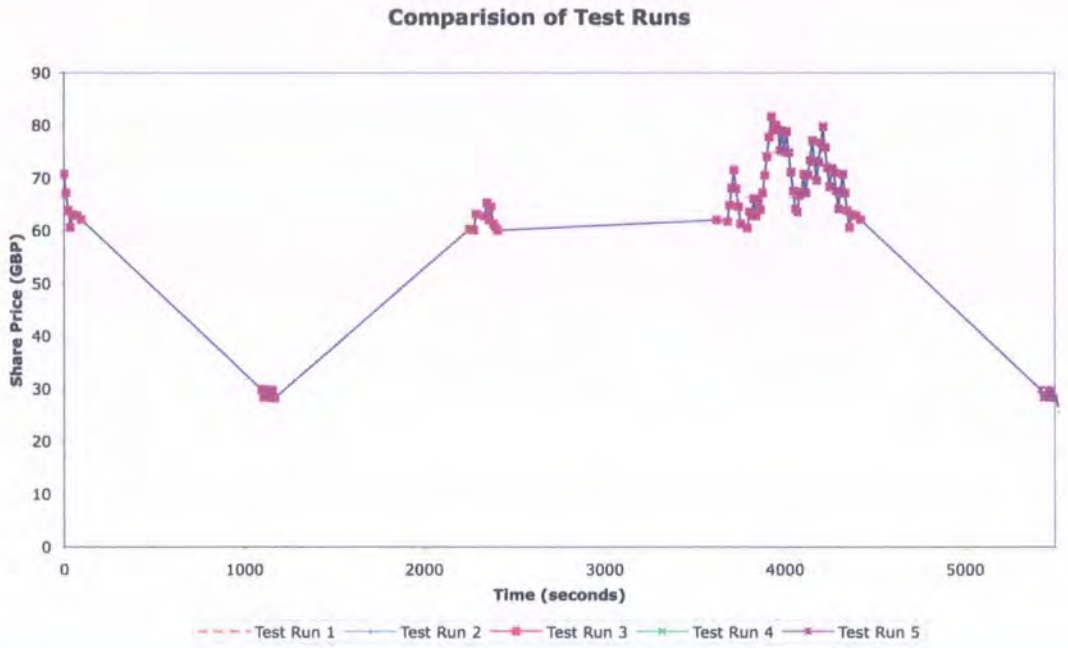


Figure 6-12: Baseline Test Series

The test case was iterated five times and the transactions from each were compared. Apart from minor timing variations the analysis showed that the test case was repeatable as can be seen in Figure 6-12. This plot is drawn as a line graph to show the trend of purchase and sales and to make discrepancies between runs clear. This also allows the sequence in which transactions occur in any run to be clearly represented. In this plot all five test runs produced similar results so the points plotted and lines drawn linking them together appear as one line.

6.2.1.4 Faulty/Malicious Service

The second test series simulated a faulty/malicious quote service by applying one of the Extended Fault Model tests to the quote service. The test chosen was a test that generated a random value that is within the specified range for the parameter it is

applied to. The random model used injects a normally distributed randomly generated value that replaces the RPC parameter specified. The same starting conditions as the first test series were used and the test was iterated five times.

Table 6-9: Attack Injection

	Test 1		*Test 2*		*Test 3*		*Test 4*		*Test 5*	
	Match	*Miss*	*Match*	*Miss*	*Match*	*Miss*	*Match*	*Miss*	*Match*	*Miss*
Match %	0.00	100.00	0.00	100.00	0.00	100.00	0.00	100.00	0.00	100.00
Average Time		0.02		0.02		0.02		0.02		0.02
Std Dev		0.01		0.02		0.01		0.01		0.01

Table 6-9 shows the results of the analysis. The data shows a clear deviation from the first test series since the quote service is returning different data. Each test run was repeatable since the randomly generated sequence was contained statically within the test script. During all test runs the system appeared to run correctly from a user perspective but by comparing it to the first test series it is clear that the system was being corrupted by the fault, with the consequence that the share trades were inaccurate (see Table 6-9 and Figure 6-13).

Figure 6-13 is drawn as a line graph to show the trend of purchase and sales and to make discrepancies between runs clear. This also allows the sequence in which transactions occur in any run to be clearly represented. In this plot all five test runs produced similar results so the points plotted and lines drawn linking them together appear as one line. The baseline plot is shown as a sixth line but since the original baseline plot has a small scale compared to the scale used in this plot the baseline appears as a single straight line near the bottom of the Y axis.

The Extended Failure Model is capable of detecting this since our specification specified a time duration for quote repeatability. Within this period the quote service

must return the same result. Since each value from the quote service is replaced by a random value and the quote that triggers the purchase/sale and the purchase/sale quote fall within this time duration, it was possible to customize the Extended Failure Model with a script to detect this from the data extracted from the WSDL and original specification.

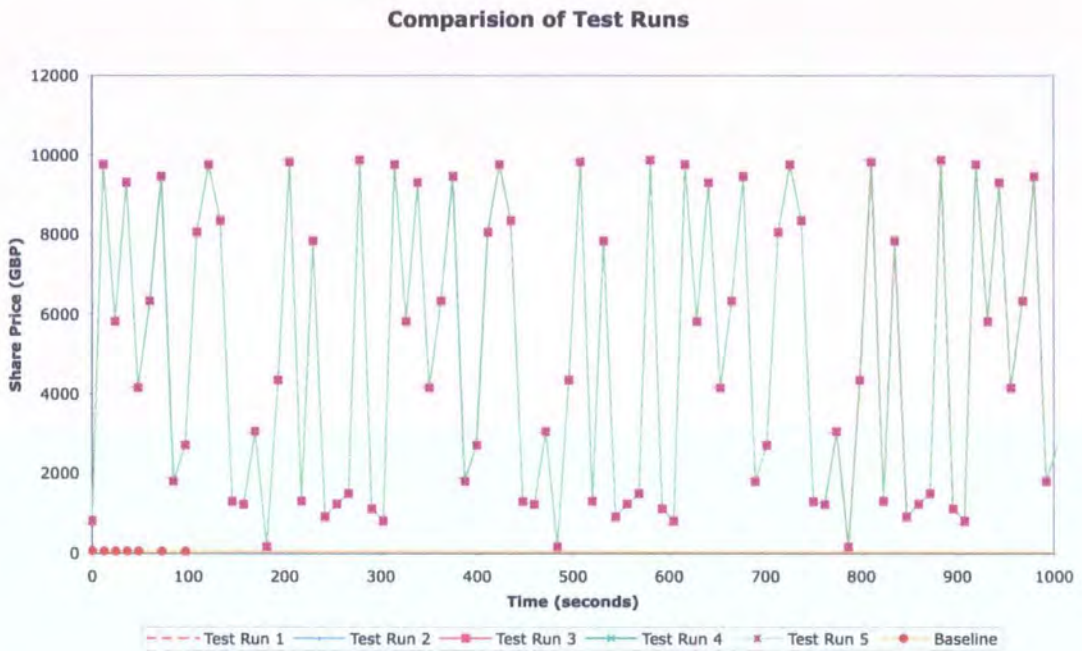


Figure 6-13: Attack Data Series

The test series showed that it was possible for the system to be corrupted/attacked without the user being aware of any such failure.

6.2.1.5 Latency Injection

The third and final series of data in this set of tests again injected a fault into the system. This fault was an increased latency induced into the quote service. This latency simulates server loading. To implement this one of the predefined tests in the Extended Fault Model. The test used introduced a delay into the system based on a possion

distribution. The distribution is statically encoded into the test script to allow for repeatability. Latency was first introduced into the quote service and data was collected. Then a second series of data was collected with the latency introduced into the bank service. The test was iterated over five runs.

Table 6-10 contains the results from the injection performed on the quote service. This clearly shows that the system is functioning differently to the baseline test series. By analysing the test data gathered it can be see that the quote value that triggers a sale/purchase of shares differs from sale/purchase price approximately 63% of the time. This is due to some quote values being delayed long enough to cause the quote to fall into the next quote period (see Figure 6-14).

Figure 6-14 is drawn as a line graph to show the trend of purchase and sales and to make discrepancies between runs clear. This also allows the sequence in which transactions occur in any run to be clearly represented. In this plot all five test runs produced slightly different results so the sequencing of the points plotted can be seen by following the line linking any given set of points for a run. This allows discrepancies in both timing and transaction to be seen. The baseline is shown as a sixth line on the plot for comparison.

Table 6-10: Latency Injection

	Test 1		*Test 2*		*Test 3*		*Test 4*		*Test 5*	
	Match	*Miss*	*Match*	*Miss*	*Match*	*Miss*	*Match*	*Miss*	*Match*	*Miss*
Match %	39.00	61.00	32.89	67.11	40.00	60.00	38.00	62.00	36.96	63.04
Average Time	3.23	24.86	2.33	23.94	3.40	25.11	3.69	24.64	3.93	24.03
Std Dev	4.22	7.63	3.40	7.82	4.29	7.43	5.38	7.75	5.20	8.52

We repeated this test by injecting the same latency into the bank service rather than the quote service. This produced results comparable to those obtained above. Since the race condition designed into the system includes both a call to the quote service and the bank service this similarity is to be expected.

This test series demonstrates not only that the system is susceptible to delays introduced by loaded servers but also that a user of WS-FIT need not have detailed knowledge of the system to use the Extended Fault Model since it is not critical where latency is introduced into this system. Again Extended Failure Model can detect this failure using the same failure model test as the second test series.

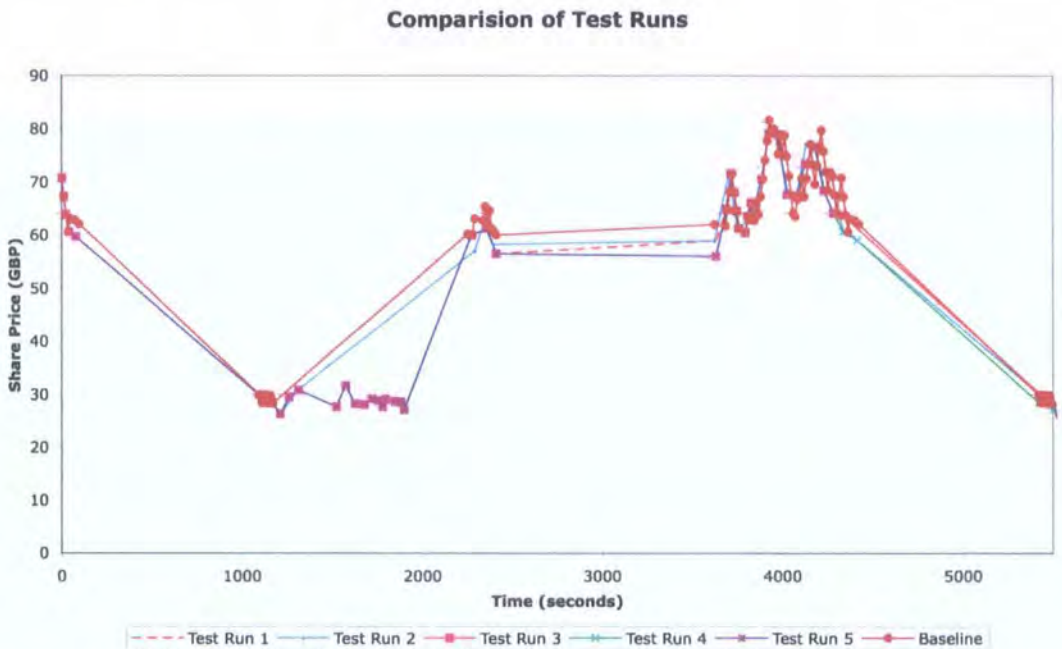


Figure 6-14: Latency Test Series

6.2.1.6 Evaluation

This case study has demonstrated that the Extended Fault Model can be used to automatically generate fault injection test campaigns. This required a minimum of user intervention, apart from constructing the ontology for the SOA by importing WSDL,

and selecting the services/methods to inject faults into. For the test scenario demonstrated that applying the same latency injection to a number of points in the system could generate a similar effect. This demonstrated that, for this test scenario, user knowledge of the injection point was not important to the test outcome.

The Extended Failure Model was applied to the test results to demonstrate its potential for detecting failures in a SOA. The current Extended Failure Model is a proof of concept and further work is required to enhance this and integrate it into the WS-FIT tool.

Finally this case study demonstrated that it is possible to create parameter perturbation using WS-FIT and use it to aid in the dependability assessment of an SOA.

6.2.2 Assessment of a Fault Tolerance Mechanism

This case study demonstrates that parameter perturbation can be applied to a non-trivial system using WS-FIT. Specifically this case study assesses a Multi-Version Design (MVD) fault tolerance mechanism intended to increase the overall dependability of an SOA. This is an important use of WS-FIT since Fault Tolerance is one of the four means of countering threats to a system and hence increasing its dependability.

6.2.2.1 Scenario

To provide a test bed to demonstrate WS-FIT we have constructed a test system that simulates a typical stock market trading system that incorporates a MVD system called FT-Grid [69]. This system is composed of a number of Web Services:

1. A service to supply real-time stock quotes
2. A service to automatically trade shares

3. A bank service that provides deposits, withdrawals and balance requests.

The trading SOA implements a simple automatic buying and selling mechanism. An upper and lower limit is set which triggers trading in shares. Shares are sold when the high limit is exceeded and shares are purchased when the quoted price is less than the lower limit. The buying and selling process involves transferring money using the bank service and multiple quotes, one to trigger the transaction and one to calculate the cost of the transaction. These multiple transactions involve processing and network transfer time, and this constitutes a race condition since the quoting service produces real-time quotes using a time-based algorithm. Any such race condition leaves the potential for the system to lose money since the initial quote price may be different from the final purchase price.

6.2.2.2 Configuration

We instrument each server in the system to eliminate any bias caused by the slight latency introduced by WS-FIT. This instrumentation allows faults to be injected into the system and monitor the RPC exchanges between Web Services (see Figure 6-15 and Figure 6-16).

There are three different series of test data:

1. A baseline with the system running normally
2. A simulated malicious service
3. A simulated heavily loaded server

There are two configurations of the SOA for each test series:

1. The SOA running with a single quote service (Figure 6-15). This will be known as configuration C_1
2. The SOA running with 5 replicated quote services using FT-Grid to provide a fault tolerance mechanism (Figure 6-16). This will be known as configuration C_2

C_1 is used to provide a baseline system to compare C_2 against. To demonstrate that our results are repeatable each test series is repeated four times.

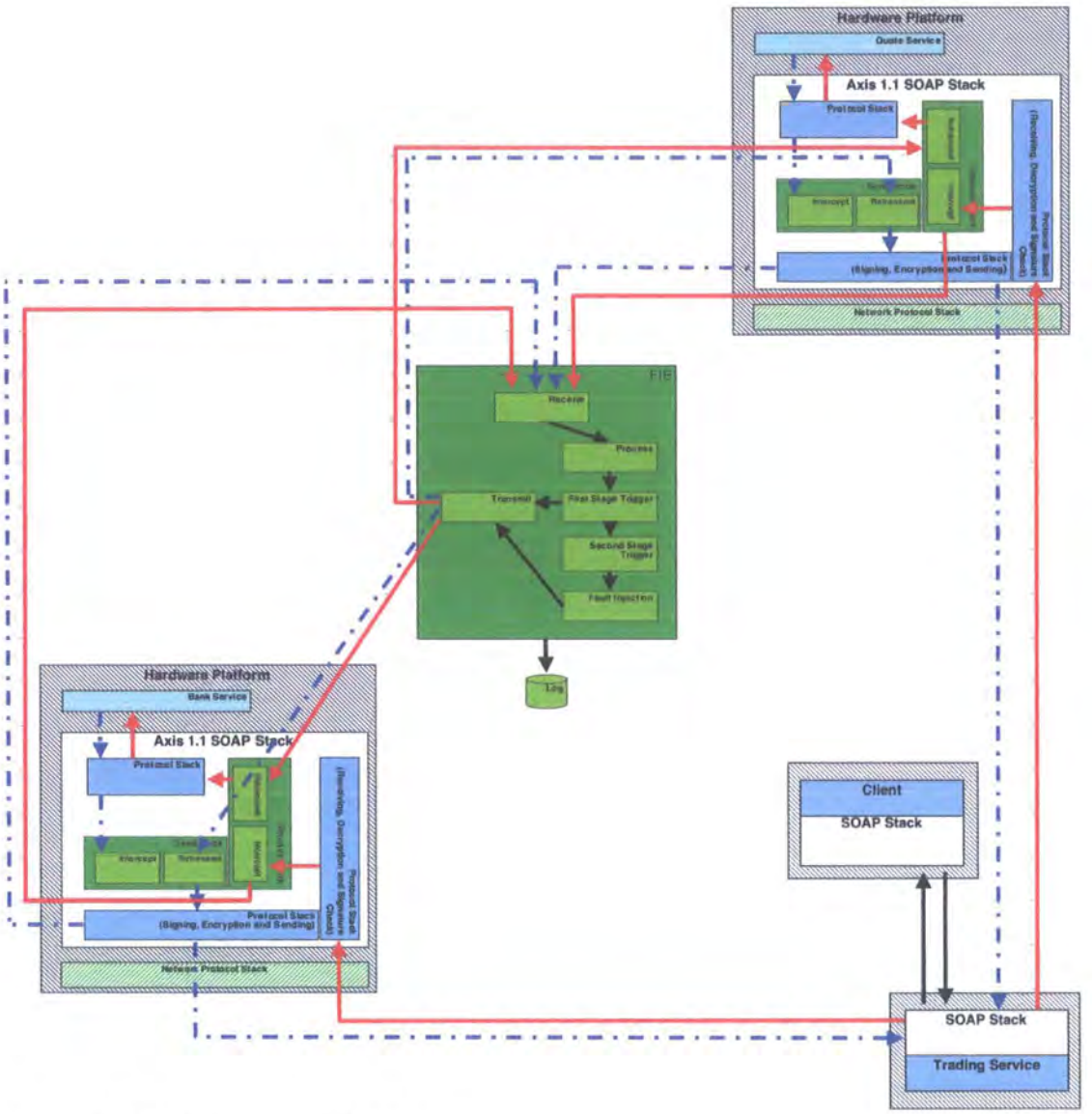


Figure 6-15: Instrumented SOA

The test system was implemented using Apache Tomcat 5.0.25 with Web Services implemented using Apache Axis 1.1, hosted on Fedora Linux Core 2 running on dual 3Ghz IA-32 Processors. Each quote service was hosed on a separate Tomcat server to avoid interaction effects.

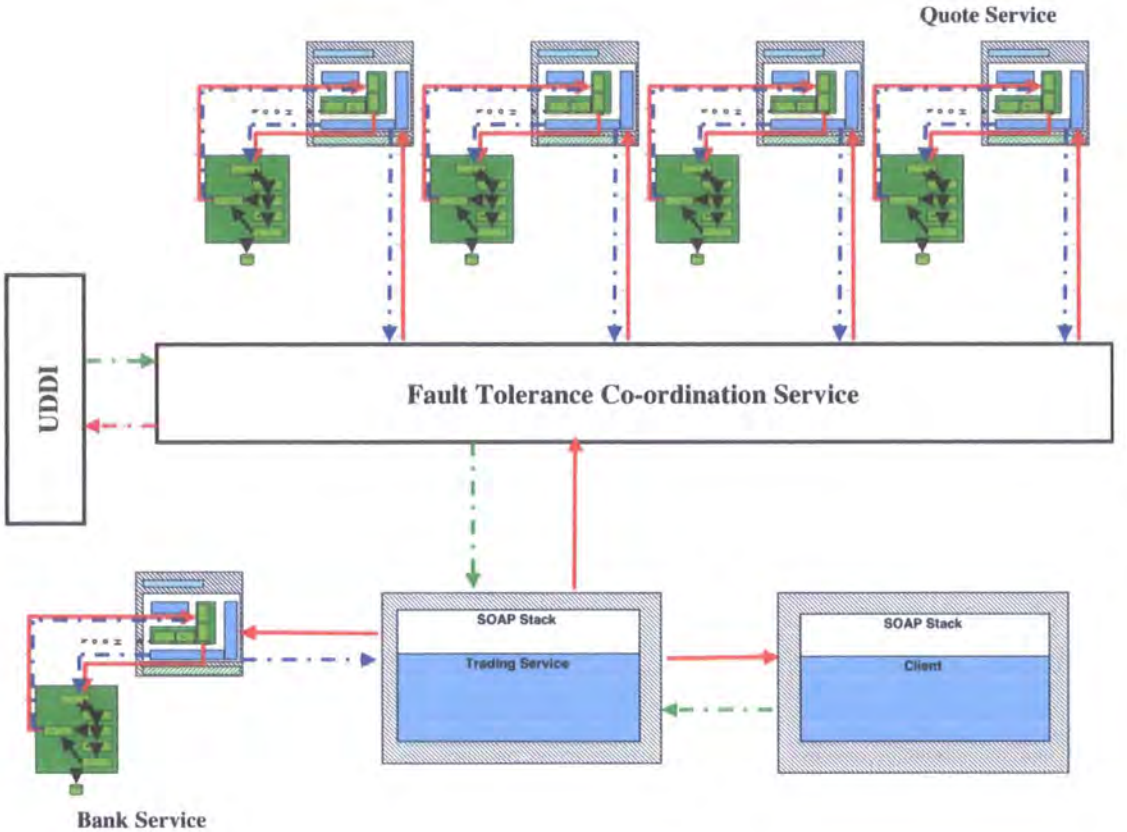


Figure 6-16: Instrumented FT-Grid SOA

This results collected from the configurations simulate three different test conditions:

1. Normal SOA Operation
2. Malicious Service
3. Server Loading

6.2.2.3 Normal SOA Operation

This test is designed to demonstrate the system running under normal conditions (C_1) and to provide a baseline to compare further test cases against. It also demonstrates that C_2 improves the system in terms of reliability whilst minimising the overhead introduced by the fault tolerance mechanism itself.

The first series of data collected from C_1 allows us to verify that the system operates according to its specification. The data given for each configuration and test type in Table 6-11 is split into three parts (Normal, Malicious and Server Loading). For each configuration and each test type four metrics are shown:

1. Percentage match between trigger price and sale/purchase price
2. Time for a successful transaction
3. Time for a failed transaction
4. Percentage of transactions that have a successful consensus.

From the data given in Table 6-11 it can be seen that under normal operation C_2 causes approximately the same number of mismatches as C_1 indicating that FT-Grid does not adversely affect the system. It should be noted the FT-Grid does cause a small latency of 0.22 seconds in overall transaction time when compared to C_1.

Table 6-11: Summary of Data From Test Cases

	Normal		*Malicious*		*Loaded Server*	
	C_1	C_2	C_1	C_2	C_1	C_2
Match %	99.50	99.00	0.00	99.71	37.37	76.33
Transaction Time (sec)	0.05	0.27	N/A	0.22	3.16	3.28
Transaction Time (Failure) (sec)	0.06	0.37	0.04	0.20	24.64	11.59
Consensus %	N/A	100.00	N/A	68.38	N/A	50.31

6.2.2.4 Malicious Service

The second test series simulates a malicious quote service by applying one of WS-FIT's predefined fault model tests to the quote service. The test chosen generated a random value that replaces the RPC parameter returned for a quote. For C_2 the same test was used and was applied to 2 of the 5 replicated quote services.

Figure 6-17 is drawn as a line graph to show the trend of purchase and sales and to make discrepancies between runs clear. This also allows the sequence in which transactions occur in any run to be clearly represented. In this plot all four test runs produced similar results so the points plotted and lines drawn linking them together appear as one line. The baseline plot is shown as a fifth line but since the original baseline plot has a small scale compared to the scale used in this plot the baseline appears as a single straight line near the bottom of the Y axis.

Figure 6-18 is drawn as a line graph to show the trend of purchase and sales and to make discrepancies between runs clear. This also allows the sequence in which transactions occur in any run to be clearly represented. In this plot all four test runs produced slightly different results so the sequencing of the points plotted can be seen by following the line linking any given set of points for a run. This allows discrepancies in both timing and transaction to be seen. The baseline is shown as a fifth line on the plot for comparison.

The malicious service data contained in Table 6-11 shows the results of the simulated attack using C_1. This data demonstrates a clear deviation from the normal system since the quote service is returning corrupt data and there is a 100% mismatch between trigger and sale/purchase quotes. By comparing the trend given in Figure 6-17 with the baseline trend shown by the crosses, it is clear that the system is under attack but since, under normal operation, this data would not be available it would not be possible to easily detect this attack.

Figure 6-17: C_1 Malicious Trend

By comparing this data to the attack data for C_2 we can demonstrate that the FT-Grid voting has rectified a proportion of the attack. This can be seen by comparing C_1 (See Figure 6-17) with C_2 (See Figure 6-18). Since FT-Grid has been configured to use consensus voting, any value returned to the trading service can be guaranteed correct since corrupted values are rejected. We see that with only 3 reliable quote services out of 5, consensus is reached in 68% of all transactions. Further this operation introduces no more latency into the system than when running C_2 without injected faults.

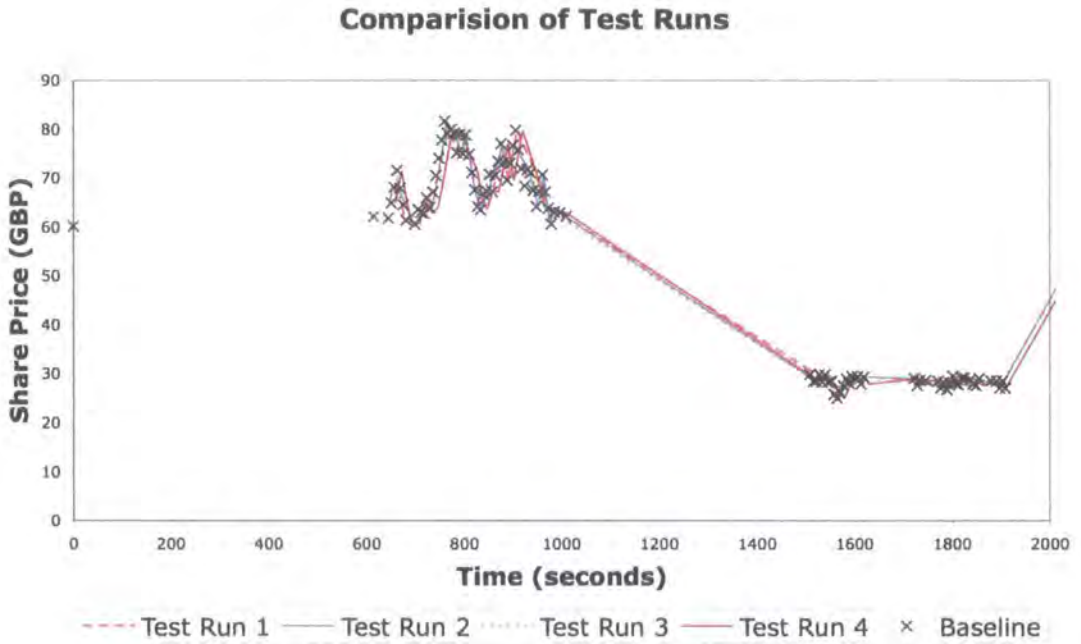


Figure 6-18: C_2 Malicious Trend

6.2.2.5 Server Loading

The third series of data injects a latency fault into the quote service and this simulates server loading since the request to the server from the client is delayed. To implement this one of the predefined tests in the Extended Fault Model that introduces a delay into the system based on a poisson distribution was applied. For C_2 the fault was injected into 2 out of 5 of the replicated quote services.

For C_1 it can be seen that the match rate between trigger and sale/purchase quotes has fallen to 37%. This indicates that the latency is causing the race condition between trigger and sale/purchase quotes to fail. This can be verified by examining the average transaction time for a mismatch, which is 24 seconds. This is double the frequency used to generate the real-time quotes and is causing the sale/purchase quote to be miss-calculated. By comparison the match rate for C_2 is 68%. Since the voting algorithm is

written to accept the first three returned values and discard the rest C_2 is not as susceptible to latency as C_1.

6.2.2.6 Evaluation

Our initial dependability analysis of FT-Grid using WS-FIT uncovered a number of problems that had gone undetected during development testing and would have been hard to detect using conventional test methods. Firstly, when running the two sets of baseline tests for C_1 and C_2 it became apparent that a 2 second latency was being introduced into the system. This was tracked to the service discovery mechanism, which was performing a UDDI lookup for every operation. This was modified to cache the initial service discovery for reuse and only request a discovery when the cache becomes invalid.

A second problem was uncovered which biased the consensus voter to favour certain services over others partly as a result of the above fix. Initially FT-Grid performed a service discovery and then invoked the services in a linear manner. It was hoped that the service discovery was sufficiently different each time to provide some different in ordering. When the service discovery was cached this then favoured the services started first, and hence biased the voting mechanism. This was eliminated from FT-Grid by randomising the service invocation order each time from within FT-Grid.

The above demonstrates that WS-FIT can be effectively applied to a fault tolerance mechanism during development to debug a system. The above faults were unexpected and had gone undiscovered during the previous development of this third party code.

An interference effect caused by excessive latency being injected into a service was also observed. If a large enough latency was injected into a server a marked degradation

in performance was seen. This is more than would be expected from the system since FT-Grid should compensate for latency effects in the system as we have demonstrated in section 6.2.2.5. This may be due to an interaction between the Web Service container and WS-FIT. Latency is injected by a single thread in WS-FIT (one per server) so if the latency injected is greater that the frequency of the service call then the server may still be blocking on the previous latency from WS-FIT, hence the degraded performance. It is believed that this effect is only present in extreme latency injection but further research is required to determine the exact cause.

The analysis of the data given in section 6.2.2.3 simulating Normal SOA Operation demonstrates that for this scenario FT-Grid works in a non-invasive manner with only a small latency introduced. Whilst this latency is not significant in the context of this demonstration, since timing constraints are in the order of 5s or greater, it could become significant in systems with tighter timing constraints. Although these results are promising more experimentation is required to determine if this latency is fixed or if it is cumulative, etc. This in turn will give an indication of the scalability of FT-Grid.

This data also shows that WS-FIT can be used in a non-invasive way when applied to a system with the given timing constraints and not adversely affect the operation of the system. The timing constraints specified are typical of a Web Service system operating over the Internet.

The data collected for a simulated SOA Operating with a Malicious Service shows a marked increase in reliability when using FT-Grid. This is due to the consensus voting mechanism utilized by FT-Grid. This ensures that single corrupted values cannot be passed to the utilizing client. Although there is a drop in the number of readings that are

accepted by the client the overall operation of the system can continue at a degraded level. The degradation was 32% for this test scenario.

This data demonstrates that WS-FIT and its EFM can be used to effectively assess the fault tolerance mechanism employed here and simulate a service who's integrity has been compromised.

The scenario that simulates an SOA Operating with Server Loading indicated a marked improvement when operating in the FT-Grid configuration when latency was introduced. It was observed that whilst C_1 was susceptible to latency and an increased number of mismatched readings were observed, C_2 saw a reduction in this due again to its consensus voting mechanism. The voting mechanism is designed to vote on the first 3 values returned and discard the other two is consensus if reached. Consequently any delayed values are discarded and the effect of any latency within the quote services is reduced.

This data demonstrates that WS-FIT and its EFM can be applied to a fault tolerance mechanism to assess its operation in the presence of delayed communications, in this case simulated server loading. This is a more traditional application of fault injection not involving any parameter perturbation or other data corruption demonstrating that WS-FIT can be used to inject standard fault models as well as message level parameter perturbation.

6.2.3 Application of Communications Faults to an SOA

This case study demonstrates how WS-FIT can be used to apply fault models which are categorized as 'Communication Faults' rather than 'Software Faults'. It 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.

6.2.3.1 Scenario

The test scenario will be based around an electronically controlled heater system. In the 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.

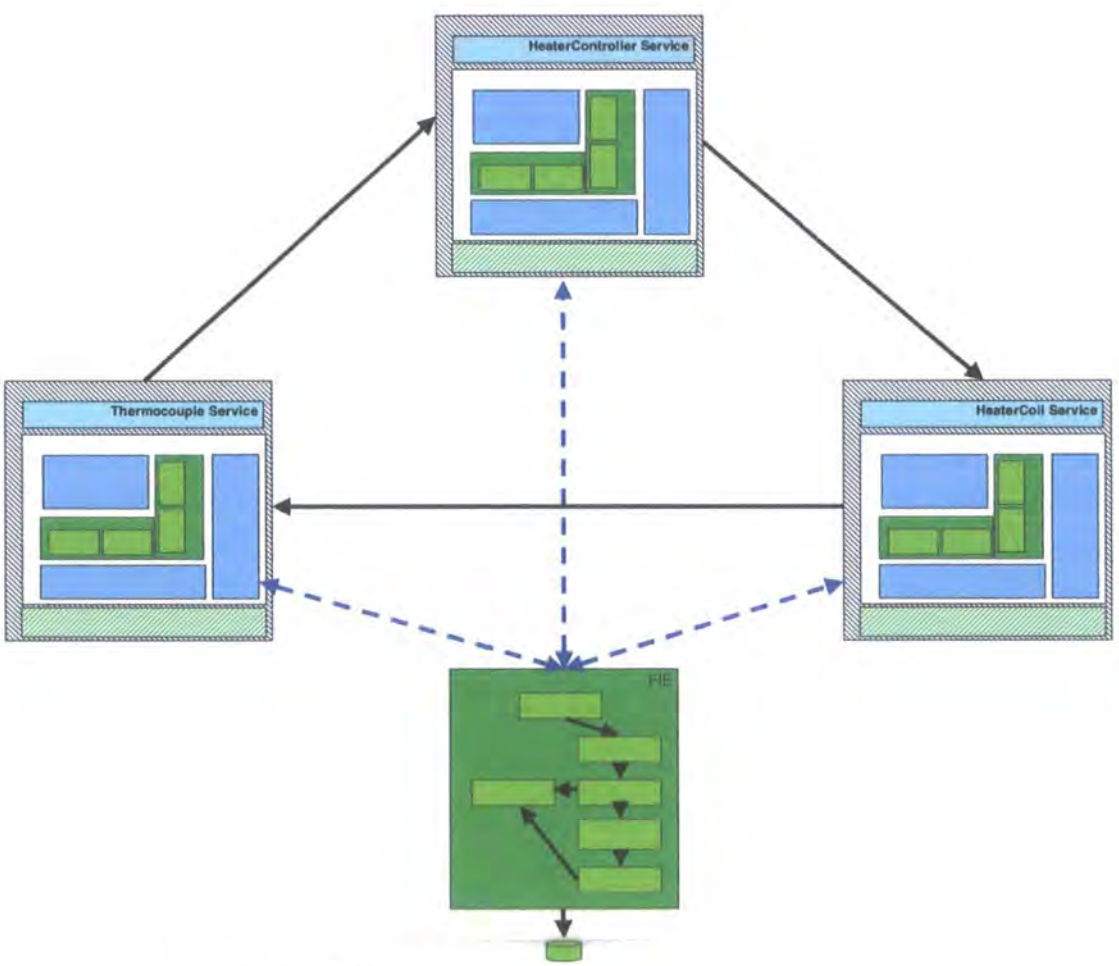


Figure 6-19: 'Heat Unit' Design

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 6-19)

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 become crucial [45]. This system could be constructed with three embedded microprocessor boards, running an embedded OS and could conceivably be running a full web server connected to the outside world via an Internet connection for remote monitoring and control.

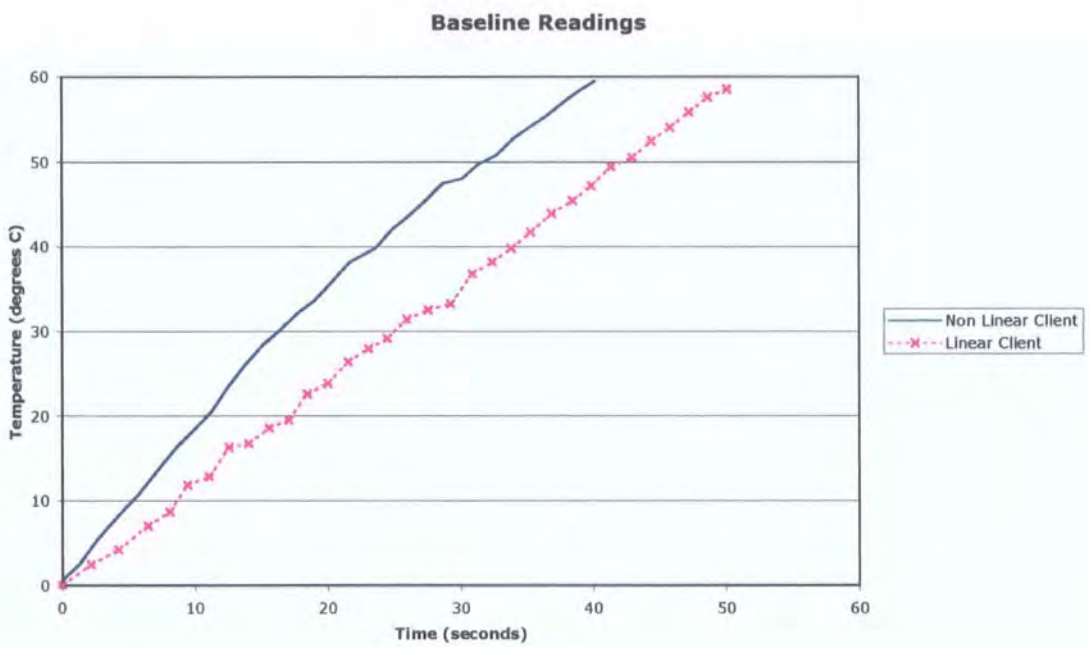


Figure 6-20: Baseline Readings

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 6-21), 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.

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*.

6.2.3.2 Configuration

Figure 6-21 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. WS-FIT 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.

6.2.3.3 Results

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 in spreadsheets.

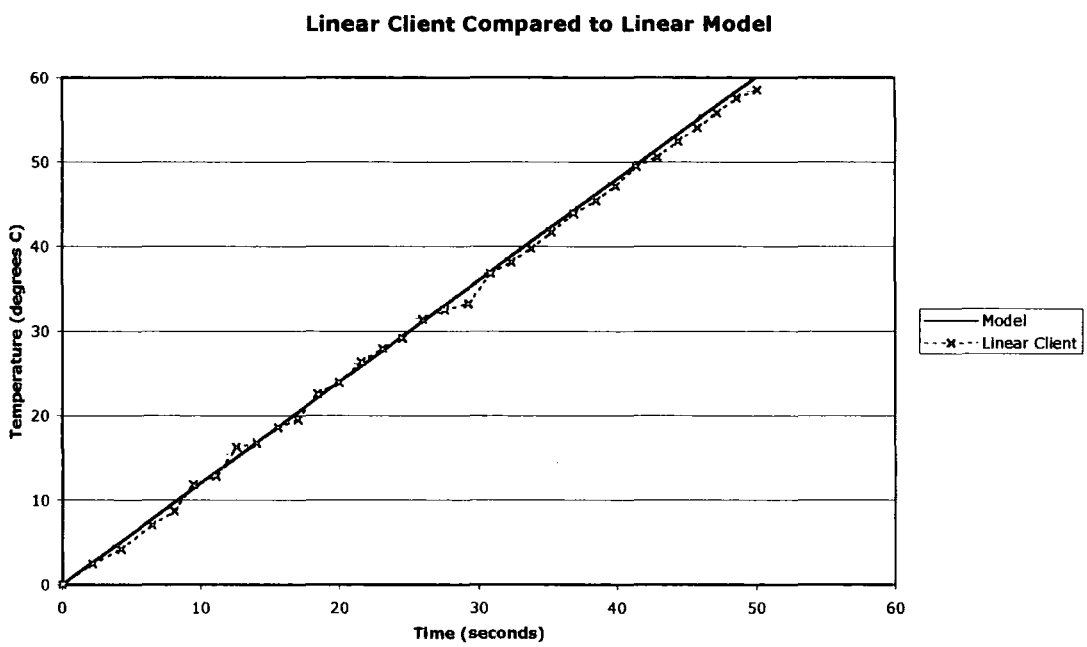


Figure 6-21: Model compared to Linear Client

The system can be assessed by monitoring the *getTemp* operation (See Figure 6-21). 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.

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.

6.2.3.4 Test Case Generation

The test case generation is based around the EFM detailed in Chapter 5 that can be used to provide automatic test case generation from a predefined set of alternatives. The test cases uses the scenario described in Section 6.2.3.1 but instead of devising a test script based on the Web Service design we will base our analysis purely on the Service specification which will be derived 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 an appropriate point. 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. This can be used to determine that a test script is producing the desired results.

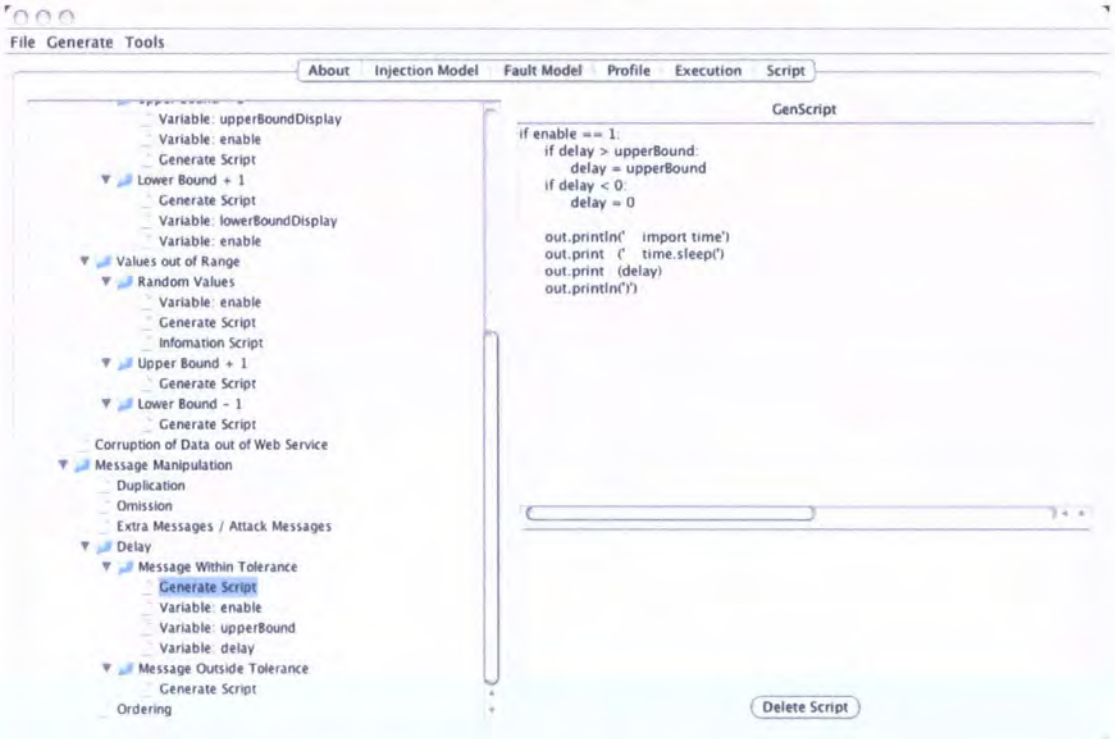


Figure 6-22: Constructing a Detailed Fault Model

Once it has been determined which RPC to target a detailed model must be selected from the Extended Fault Model. Since no detailed knowledge of the implementation of the system can be assumed, other than its specification, the detailed model takes general parameters and constructs a specific test case from these. If a suitable model was not present in the model a new detailed test could be added with the WS-FIT tool.

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 use 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.

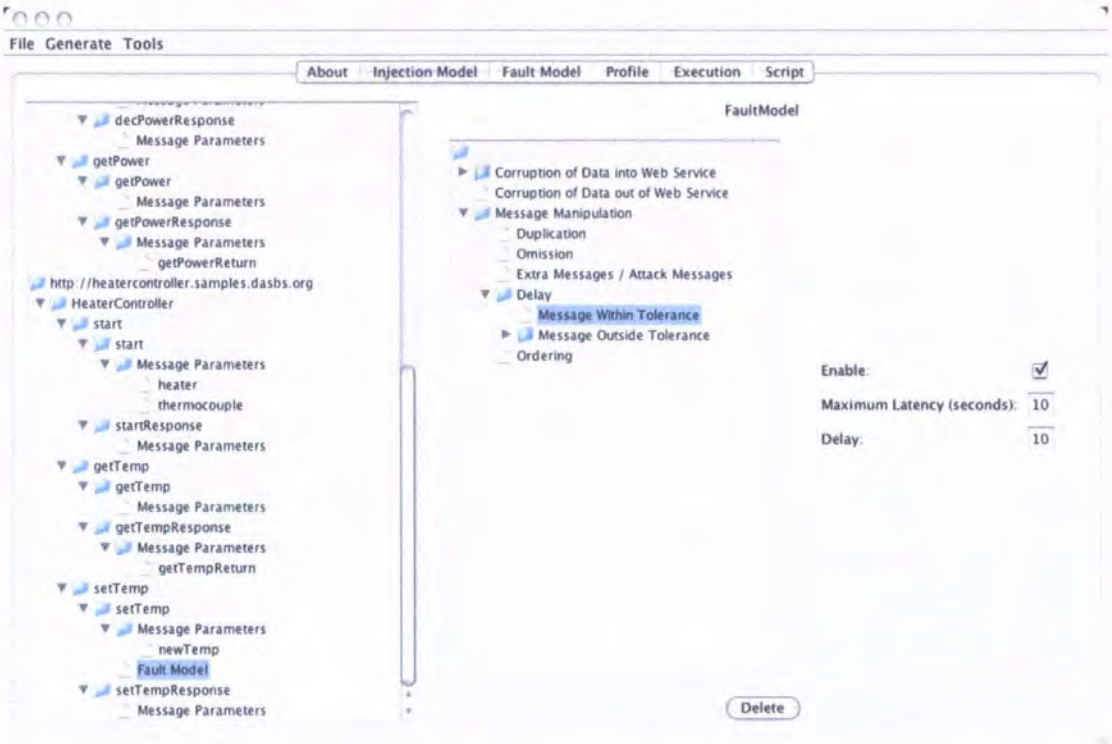


Figure 6-23: Applying a Fault Model to a Parameter

The fault model test does not actually run as part of the main test script, it merely 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.

6.2.3.5 Latency Injection

Since WS-FIT is attempting to affect the throughput of the system the 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.

The first test script was generated by adding a fault model to the *setTempRequest* message. From the Extended Fault Model the *Message Within Tolerance* model was selected. This model delays the return of a message from the FIE to the Hook Code which is within the tolerance of the SOAP protocol stack.

The 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. The visualization feature was used 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 affected. It is possible 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 verified by the information contained in the log files once they were analyzed.

The 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.

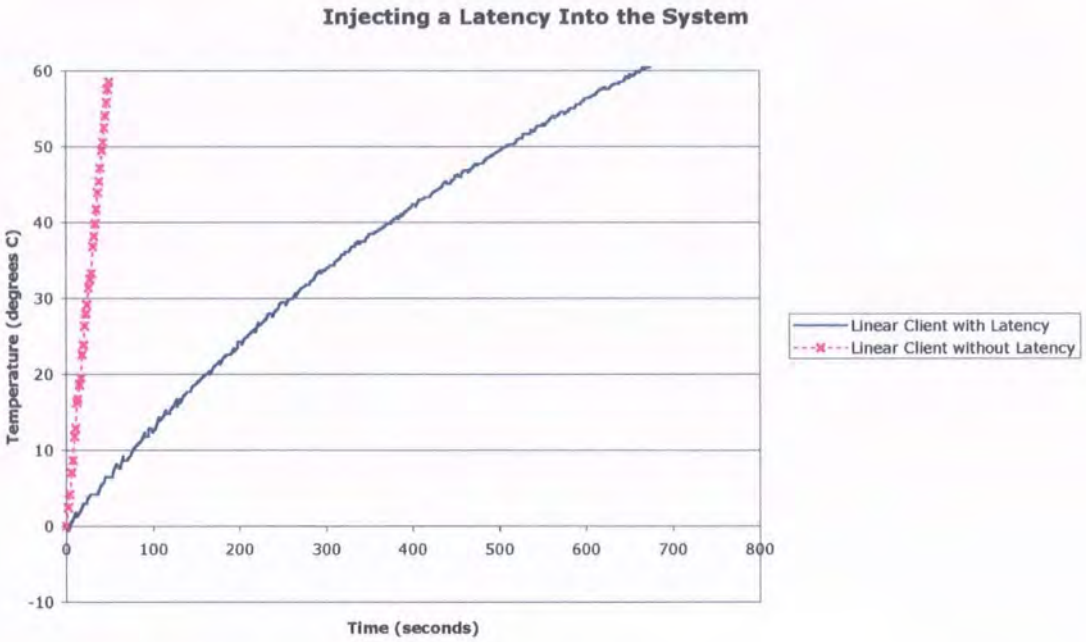


Figure 6-24: Injecting Latency into the System

The script introduced a 1 second delay into sending each *incPower* and *decPower* message to the *Heater*. In this way the throughput of the *HeaterController* to *Thermocouple* message exchange was reduced, and thus the performance of the system as a whole was altered.

The visualization facility was used to monitor the results of this experiment. The results are given in Figure 6-24 and clearly demonstrate that the SLA for this system has not 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 cannot be achieved.

6.2.3.6 Evaluation

This scenario has demonstrated how WS-FIT can be used as an aid to dependability assessment of QoS constraints. WS-FIT has been used 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.

It further demonstrates how the EFM 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 EFM through the WS-FIT tool.

6.3 Summary

This chapter has presented a number of case studies that demonstrate key features of the FIT method and its realisation in WS-FIT.

The first case study (Section 6.1.1) demonstrated that WS-FIT can inject faults into middleware messages, rather than network packets. Using this as a basis the triggering mechanism was demonstrated by allowing specific middleware messages to be triggered upon. Since individual messages could be targeted it was possible to perturb individual parameters within the targeted message.

The second case study (Section 6.1.2) demonstrates that WS-FIT can be used in an Internet based Web Service system in a non-invasive way, with the latency overhead being acceptable when used in this configuration. This case study further examined the types and amounts of data that can be exchanged in an SOA instrumented with WS-FIT.

The third case study (Section 6.1.3) demonstrated how the modified Network Level Fault Injection technique can be used as part of a specification based certification strategy and how it can be used to achieve the same effect as Code Insertion fault injection but with the added benefit that it is fare less invasive. Code Insertion also 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.

The fourth case study (Section 6.2.1) builds on the features demonstrated in the first three to show how the extended fault model and extended failure models can be applied to a SOA to perform a dependability assessment. This case study has demonstrated that the Extended Fault Model can be used to automatically generate fault injection test campaigns. The Extended Failure Model was applied to the test results to demonstrate it's potential for detecting failures in a SOA. This was accomplished by hand although at a future time it is envisaged that this functionality could be incorporated into the WS-FIT tool. Finally this case study demonstrated that it is possible to create parameter perturbation using WS-FIT and use it to aid in the dependability assessment of an SOA.

The fifth case study (Section 6.2.2) used WS-FIT to evaluate a fault tolerance mechanisms. Dependability attributes of reliability, integrity and performance were assessed by simulating threats in the SOA at known points thus verifying the means that were used to eliminate them. This case study not only allowed the demonstration of the dependability means and indicated that it provided an increase in dependability but was used as part of the debugging process to allow the discovery and repair of two undiscovered faults in FT-Grid. It was demonstrated that WS-FIT can be used in a non-invasive way when applied to a system with timing constraints and did not adversely affect the operation of the system. Timing constraints used are typical of a Web Service system operating over the Internet. Finally it was demonstrated that WS-FIT can also inject faults that do not involve parameter perturbation and thus function as a more traditional fault injector.

The final case study (Section 6.2.3) was a more detailed demonstration of how Communications faults can be applied to a SOA. It was further demonstrated how the EFM can be used to automatically generate test cases from WSDL definitions and specifications with comparable results to test cases generated manually. A demonstration of how the EFM can be enhanced using the tool was also given.

Chapter 7 - Conclusion

This thesis has presented research on devising a dependability assessment method for Web Services using network level fault injection. Although this work is based on standard Web Services the middleware combination used forms the core of current Grid middleware Globus 4 so the methods and techniques developed here should be applicable to this emerging field.

After reviewing current research it was found that little work had previously been undertaken on performing dependability assessment of distributed middleware and in particular Web Service and Grid middleware. The closest related work was undertaken on the assessment of CORBA middleware by using network level fault injection. This presented promising results but it was based on standard network level fault injection techniques and as such assessed only the middleware mechanism and protocol stacks rather than the CORBA components.

The FIT method, defined in Chapter 4, uses network level fault injection as a basis but extends it in two ways. Firstly it operates at a middleware message level rather than a network packet level, thus allowing complete middleware messages to be processed rather than having to rely on messages being contained within a single packet. Also by operating above the network interface level messages are can be intercepted before they are signed or encrypted thus allowing them to be decoded.

Secondly the FIT method allows, not only normal fault injection techniques to be applied to a middleware message, but targeted perturbations of RPC parameters. This can be accomplished by processing the message and combining this with information on the structure of the interface and messages exchanged. Using this information to

construct triggers it is possible to inject specific faults into RPC parameters in a way this is comparable to how Code Insertion fault injection injects faults at an API interface but in a far less invasive way.

The FIT method also incorporates a method for classifying and applying fault models to a Software-Oriented Architecture (SOA). This is the Extended Fault Model (EFM) and it is constructed as a taxonomy classifying faults into specific groupings to aid in their application to SOA, thus allowing a toolkit of common fault models to be constructed that can be easily applied.

This can be combined with the FIT concept of a System Model that is derived from Interface Definition Language (IDL) definitions of the Services and organizes the SOA into a taxonomy with Services at the top level and message parameters at the bottom level. Each parameter can be used as a trigger and then be linked to fault models contained in the EFM to generate a fault injection campaign.

Finally the FIT method includes a third taxonomy that allows the classification of failure modes in a similar way to the EFM. This is the Extended Failure Model (EFAM) and it can be applied in two ways: 1) Applied globally to an SOA to detect unexpected failures; 2) Linked to a model in the EFM to detect failures caused by injected faults.

A realization of the FIT method has been implemented called WS-FIT as detailed in Chapter 5. This applies the FIT method to Web Service middleware. WS-FIT implements the EFM and System Model whilst the EFAM must be applied by hand to the recorded log file at present although it could be incorporated into the tool in the future. The IDL definitions for the Service interfaces and messages are extracted from the WSDL definition for the Web Services making up an SOA.

The case studies conducted and documented here are intended to demonstrate this novel method and its applicability to providing dependability assessment of SOA. The case studies documented in Chapter 6 determine parameters for using WS-FIT in a non-invasive way and demonstrate the key features of the FIT method and its implementation WS-FIT. The first (Section 6.1.1) demonstrates the basic operation of WS-FIT; The second (Section 6.1.2) demonstrates that under typical Web Service environments WS-FIT introduces an acceptable latency overhead into an SOA under assessment; the third experiment (Section 6.1.3) compares WS-FIT to another fault injection technique and shows it can perform in a comparable way whilst being less invasive; the fourth (Section 6.2.1) demonstrates the application of the EFM and the EFAM; the fifth (Section 6.2.2) demonstrates the application of WS-FIT to assess a dependability means (Fault Tolerance); whilst the final (Section 6.2.3) demonstrates that WS-FIT can perform more traditional fault injection techniques.

7.1 Criteria For Success

This section will discuss how this thesis addresses the original criteria for success. The following five criteria will be addressed:

- Method
- Tool
- Test Case Construction Method
- Analysis Method
- Applicable to both development and testing phases

7.1.1 Method

Devise a method based on network level fault injection to perform dependability testing of Web Services. This method should be comparable to other code insertion methods but with the added benefit of minimal alteration to code.

This thesis has documented a method based on a modified version of network level fault injection (Section 4.2) which allows parameter perturbation to be achieved as well as a more traditional use of network level fault injection, for instance Corruption; Reordering; and Dropping of network packets. This method has been applied to Web Services through the WS-FIT tool (Section 5.2) and it has been demonstrated that parameter perturbation can be applied (Sections 6.1.1, 6.1.3, 6.2.1 and 6.2.2).

Further WS-FIT can be used in a non-invasive way, for instance with minimal introduced latency (Section 5.2.5 and 6.1.2) and also WS-FIT has been compared to Code Insertion fault injection (Section 6.1.3). It has been demonstrated that Code Insertion fault injection requires both access to the source code and many modifications whilst WS-FIT requires only the insertion of hook code in two places in a middleware stack. Since these modifications need not necessarily be on the machine hosting the service (Section 5.2) it is conceivable that WS-FIT could be used as part of a certification strategy since the server running the code need not be changed in any way from a production environment.

7.1.2 Tool

Construct a tool for use with the method. This tool will be tailored to injecting fault into SOAP packets and will handle the decoding of SOAP

packets so that lightweight scripts can be written by the user to implement test cases without the complexity of decoding SOAP packets.

The FIT method (Chapter 4) has been realized in a tool (Chapter 5) that applies the FIT method to Web Services using SOAP middleware. The tool intercepts and decodes middleware messages (Section 5.2) to allow the construction of lightweight test scripts (Section 5.3) that do not require SOAP messages to be decoded by the scripts.

The application of the WS-FIT tool to Web Services and the generation of scripts is demonstrated in Appendix A where a sample script is generated via the tool. A further demonstration of the use of scripting is given in Sections 6.1.3 and 6.2.

7.1.3 Test Case Construction Method

Devise a method to construct test cases for the method given above. This method should be devised to allow easy automation so that it can be incorporated into the tool.

The FIT method describes the concepts of a System Model and an Extended Fault Model (Section 4.3) that allows the construction of test scripts. The System Model represents all operations that can occur within the system so that triggers can be defined, whilst the Extended Fault Model contains predefined fault models that can be applied to these operations. The use of the Extended Fault Model and System Model is demonstrated in Sections 6.1.3 and 6.2 where it is applied to a SOA to construct a fault injection campaign. A further demonstration of the use of the Extended Fault Model is given in Section 6.2.2 where it is applied by a third party to a third party system to assess a dependability means.

7.1.4 Analysis Method

Devise an analysis method to assess dependability of result sets generated by the method. This analysis method should be applicable not only to the test phase of system development but also to the development stage.

The FIT method defines the concept of an Extended Failure Model and how to use it (Section 5.4). The Extended Failure Model contains a taxonomy that groups individual failure modes so they can be easily applied to a System Model. Whilst this mechanism has not been integrated into the WS-FIT implementation it has been possible to apply it by hand to demonstrate its potential. The application of the Extended Failure Model has been demonstrated in Section 6.2.

7.1.5 Applicable to both development and testing phases

The methods and tools should be applicable, no only to the testing phase of a project but also to the development phase. The method should also be able to test systems without access to source code.

The Extended Fault Model (Section 4.3) and Extended Failure Model (Section 4.4) are applicable to systems in a testing phase as demonstrated in Section 6.2.1 and to systems in a development phase as was demonstrated in Section 6.2.2 where previously unknown faults were uncovered in a third party system.

7.2 Future Work

Firstly we intend to apply WS-FIT to more complex systems to allow us to, not only improve and develop our method and tools, but to collect metrics for use by our Extended Fault Model and Extended Failure Model. This will allow us to refine our ontology and assess its reliability in detecting failure states in SOA.

In Section 6.1.2 WS-FIT was demonstrated to introduce a polynomial latency overhead when applied to RPC messages that contained large numbers of multiple elements within a SOAP message, for instance large arrays. Whilst this latency was demonstrated to be acceptable for small numbers of elements it quickly becomes invasive with increasing element size. Research is therefore required into reducing this latency overhead possibly by eliminating the use of a SAX parser and using a custom written state machine instead which may reduce the latency to a linear latency.

Currently the WS-FIT implementation does not integrate the Extended Failure Model into the tool and this part of the method must be applied by hand to the generated logs. Whilst this is acceptable for short test campaigns it would soon become time consuming for analysing large test campaigns that may run for log periods of time. We therefore propose that the WS-FIT tool should be extended to include this functionality, either as a post processing step or as part of the runtime visualization.

The Extended Fault Model and Extended Failure Model detailed in this thesis are intended only as proof of concepts and will need populating before they can be useful to real world dependability assessment case studies. Further enhancements should be made so that they will be capable of producing metric-based evaluations of fault injection on SOA. The aim is to improve dependability in services by providing criteria and metrics capable of evaluating dependability as well as uncovering faults for fault removal. This should produce comparison metrics capable of facilitating value judgments in the dependability of competing services as well as being of use in the debugging and testing phases of a product to assess dependability means.

One of the aims of this research has been to utilise FIT with current Grid technologies. Integration with the Globus 4 Toolkit would be advantageous since

Globus is the front running Grid technology and should be relatively straightforward since they share both Tomcat and Axis SOAP as a core but detailed experiments will have to be carried out in order to determine differences in message syntax.

From the case studies carried out here it was ascertained that the FIT method requires an experienced tester to utilize the method since expert knowledge of testing is required to select injection points within the system. Whilst FIT simplifies this process it has not been possible at this stage to totally automate the process. The FIT method currently defines an ontology that describes a system, the faults that can occur within it and the failures that can be observed. These are linked to allow specific failures to be deteched after a fault injection. Another ontology and engine is required that would utilise this information as a base but would construct fault injection campaigns utilising heuristic testing and rules captured in the ontology.

Finally applying FIT to a binary middleware, for instance CORBA, would demonstrate that FIT could be easily ported amongst dissimilar middleware products. CORBA is a good candidate since there is widespread use of CORBA and it also shares the concept of an IDL to define its interfaces. CORBA IDL does not define message formats for RPC exchanges but these can be inferred from the CORBA specification and since these message are binary in nature they would provide a good test of the portability of FIT since WS-FIT is based on an XML format message.

7.3 Summary

This thesis has documented the FIT method and the WS-FIT tool that realizes this method. The FIT method fulfils the criteria detailed in Chapter 1 and provides a basis for future research.

The modified version of network level fault injection, which works at a middleware message level rather than a packet level, used by the FIT method allows faults to be injected into Services in a manner that is comparable to Code Insertion fault injection with the added benefit that it requires far fewer modifications to a system than Code Insertion and can be implemented without the need for access to the source code of the service.

The System Model described here allows a service-based system to be described for the purpose of constructing triggers on messages without access to the Service source code. These triggers can then be used as a basis to undertake traditional network level fault injection or parameter perturbation.

The Extended Fault Model allows fault models to be categorized into descriptive classifications that can then be used to guide construction of fault injection campaigns. These fault models can then be linked to specific parameters/messages in the System Model and fault injection scripts can be automatically generated.

The Extended Failure Model allows failure modes to be categorised into descriptive groupings that again can aid a tester in selecting the correct failure mode. Individual failure modes can then be linked to individual fault models as expected outcomes and thus allow automation of failure detection to take place.

WS-FIT is the realization of the FIT method and it has been applied to a number of case studies and has successfully demonstrated these features. It has also been applied successfully to a third-party SOA to evaluate a dependability means (fault tolerance) and not only performed this dependability assessment but allowed debugging of the mechanism to be undertaken to uncovering two unknown faults in the SOA.

Appendix A - WS-FIT Tool

This appendix describes the operation of the WS-FIT tool. WS-FIT is implemented as a GUI application written in Java. Java was chosen because it allowed portability between platforms and there were also a number of packages available which simplified the parsing of SOAP messages. Python was chosen as the scripting language because it allowed object oriented scripts to be written which simplified the design and implementation of the support classes and also because it can be executed natively using the jython package from a Java JVM.

The WS-FIT GUI is divided into a number of tabbed panes (See Figure A-1). They are:

Injection Model – Allows a System Model to be constructed and used as detailed in Section 5.3.1

Fault Model – This allows the entry and manipulation of the Extended Fault Model described in Section 5.3.2

Execution – Allows a constructed test campaign to be executed and logged. This also allows real-time visualization of specified RPC parameters.

Other tabs are implemented for debugging purposes.

Figure A-1: Start-up Screen

A.1 Extended Fault Model

The Fault Model tab (See Figure A-2) allows the entry and manipulation of the Extended Fault Model (EFM). As shown in Table 5-5 a proof of concept EFM has been developed and this can be modified by using the Fault Model tab.

The Fault Model tab is split into three main parts:

1. A tree on the left hand side of the tab representing the Extended Fault Model
2. A dialog on the right hand side of the tab that allows data entry and display. This is context sensitive and its contents depend on which type of element is selected from the EFM tree on the left hand panel.

3. A set of context sensitive buttons on the lower right hand side of the tab. Again these are context sensitive and depend on which type of element is selected from the EFM tree in the left hand panel.

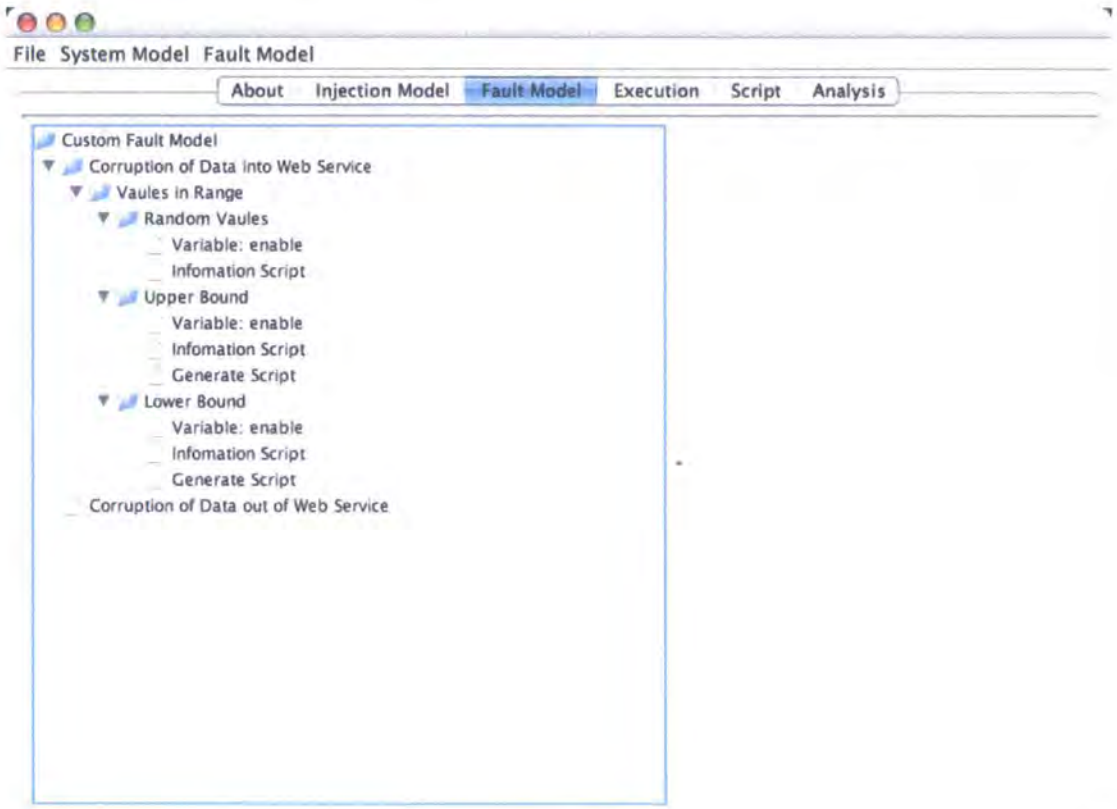


Figure A-2: An Extended Fault Model

The tool allows new high-level groupings of fault models to be added, such as Software Fault, Physical Fault, etc. and then sub-groupings to be added as decompositions until a fault model can be defined through scripting.

As a brief example a Communications Faults grouping can be added and grouped within this a sub-grouping of Server Loading (See Figure A-3). This can be done through the context sensitive buttons. At each stage the names for each of these categorizations can be entered.

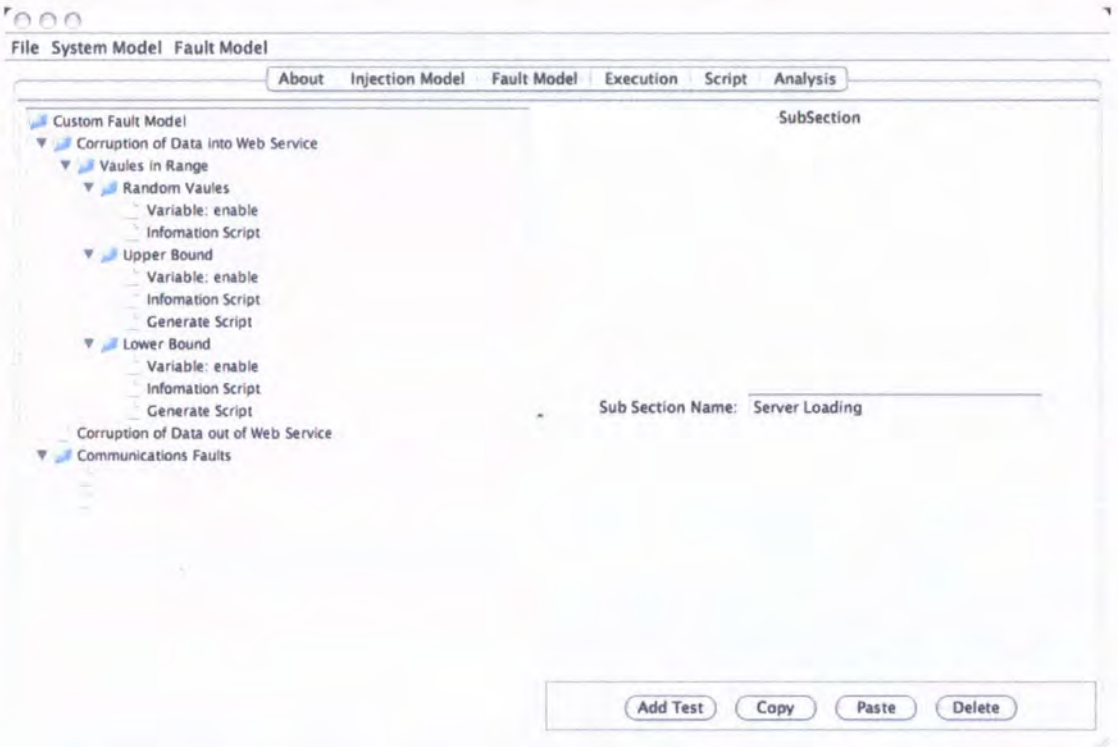


Figure A-3: Adding a new Sub-Section to a Extended Fault Model

Once suitable categorizations have been entered a fault model can be added to the appropriate sub-group (See Figure A-4). A Fault Model is made up of a python script that generates a fragment of script to run as part of the fault injection campaign (See Figure A-5). Details of how the script is implemented are given in Chapter 5. The script shown in Figure A-5 implements a simple constant delay in processing the SOAP message. This can be though of as equivalent to a server with a constant loading and therefore a constant latency in returning a SOAP request. A generate script is the minimum required to implement a fault model.

Some fault models will require data to be entered, for instance a variable to say if the fault model is enabled or a value to say how many times to iterate a sequence. These can be added to the Fault Model by using the Add Variable button (See Figure A-6). These state variables are then stored as part of the state information available in the

System Model (See section 0) that also contains other state information such as parameter upper and lower bounds. This state information can then be retrieved and used in the scripts.

Figure A-4: Adding a new Fault Model to the Extended Fault Model

Some scripts require complex setup and initialization. For this purpose it is possible to add an information script in the same manner as a generate script. The info script is executed when the fault model is selected and can be used to display an interface that can be used to enter complex information as well as processing the information. If the information script is not present any variables added to the fault model are automatically displayed as appropriate controls, for instance check boxes for Boolean variables, text boxes for string variables, etc.

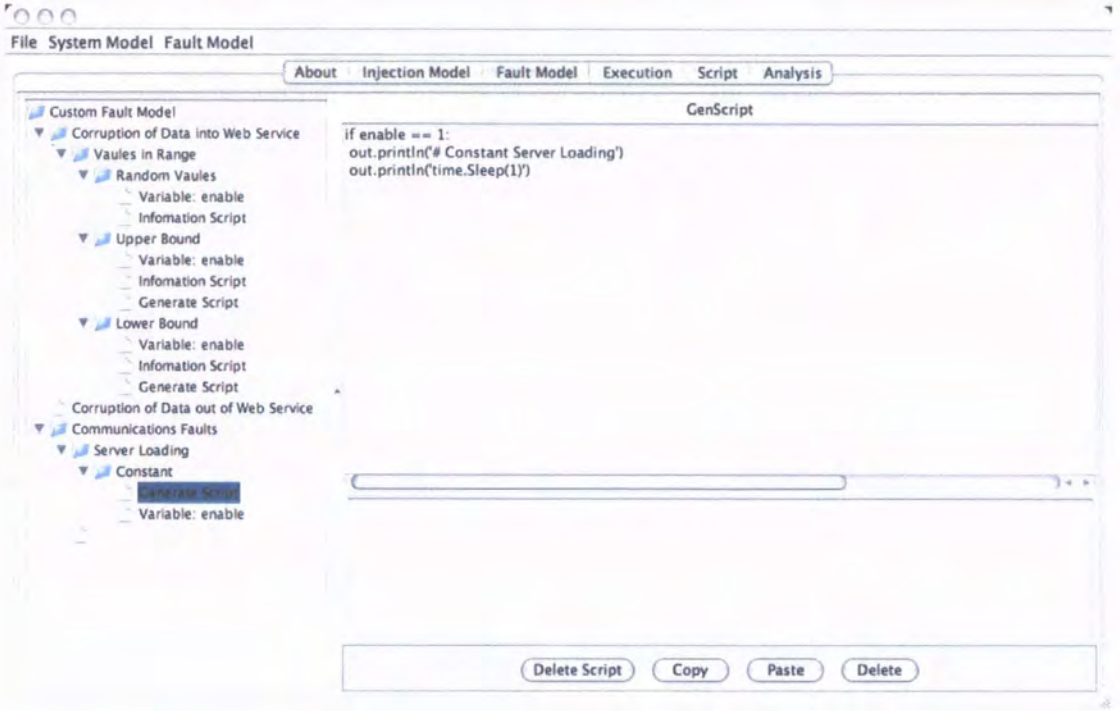


Figure A-5: Adding a Generation Script to a Fault Model

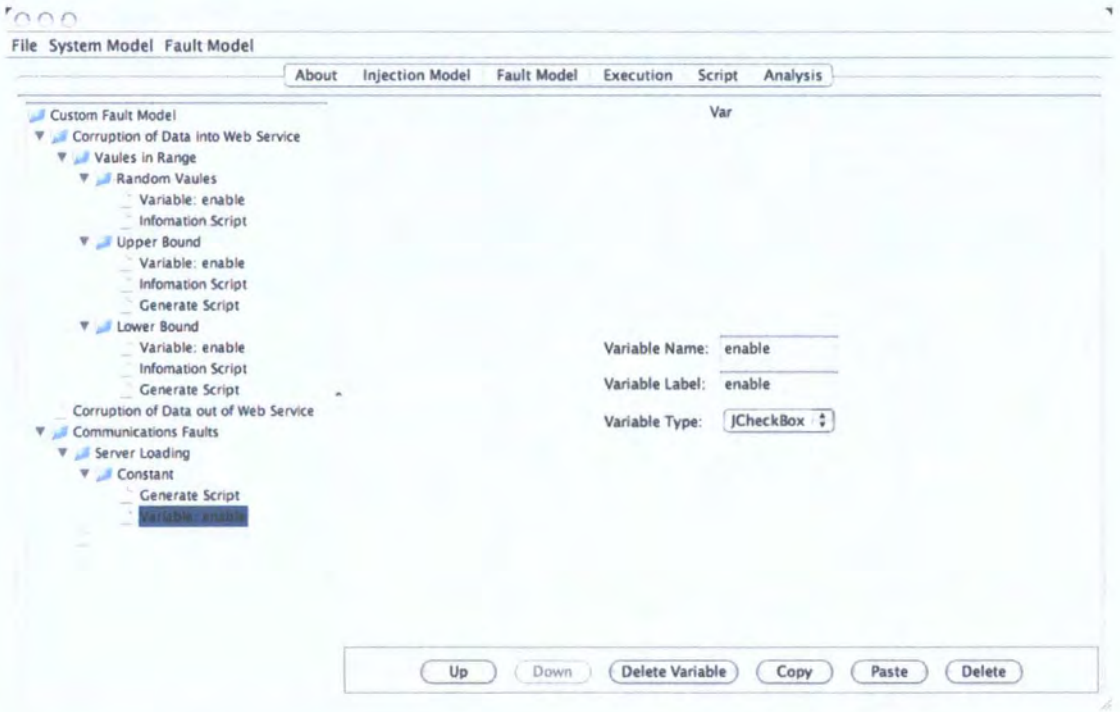


Figure A-6: Adding a variable to a script

A.2 System Model

Once an EFM has been constructed it can be applied to a system model. A system model can be constructed by using the Injection Model tab.

Since a System Model is made up of interface definitions a System Model can be populated by importing WSDL definitions for all services making up an SOA. This is done using the Import WSDL function (See Figure A-7).

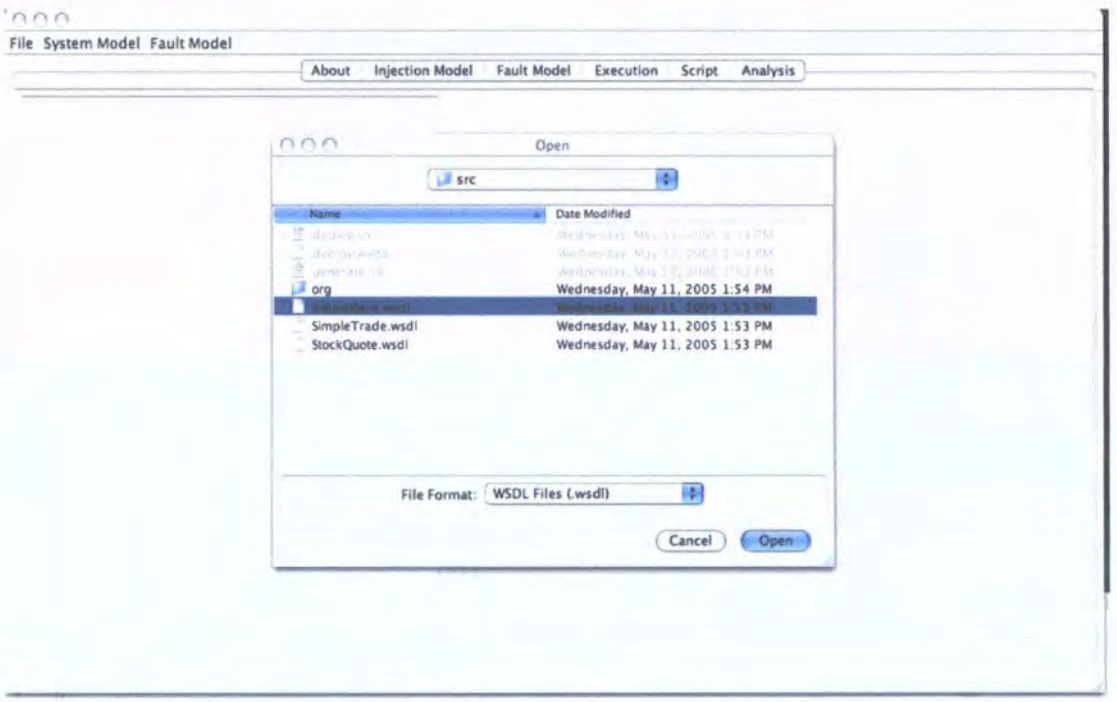


Figure A-7: Importing WSDL into a System Model

Imported WSDL is decomposed into a tree structure in the left hand tree pane. This shows a root entry for the Web Service, followed by the PortType, Operations, Messages and finally Parts.

Any number of Web Service can be imported in this way including duplicate Web Services to represent replicated services running on multiple servers. To differentiate these replicated Web Services the server address can be added to a specific Web Service

definition (See Figure A-8), thus making it a unique IP/Web Service combination that can be used as part of the triggering mechanism. In this way different IP/Web Service combinations can hand different fault models applied to them.

Figure A-8: Associating a Server with a Web Service

Since WSDL does not include any parameter bounds information other than general bounds associated with the type specification it is possible enter specific bounds information from a more detailed specification using the GUI (See Figure A-9). This information can then be utilized by the fault model scripts to generate appropriate fault campaign code.

Once the System Model has been constructed individual parameters can be associated with specific fault models contained within the EFM (See Figure A-10). In the example given in Figure A-10 the Constant Server Loading fault model is associated to a specific parameter in the System Model. This will then generate the appropriate code fragment complete with trigger conditions in the final fault injection campaign script.

More complex fault injection fault models can be implemented on a message basis by adding a manually written generation script to either the Message or Part on which they will operate. By adding a script to the Message the complete message body is available for manipulation whilst if the script is attached to the parameter only the parameter tags and body are available. In this way syntax errors can be introduced into the message to test the middleware protocol stack or to introduce fault models that the EFM cannot handle.

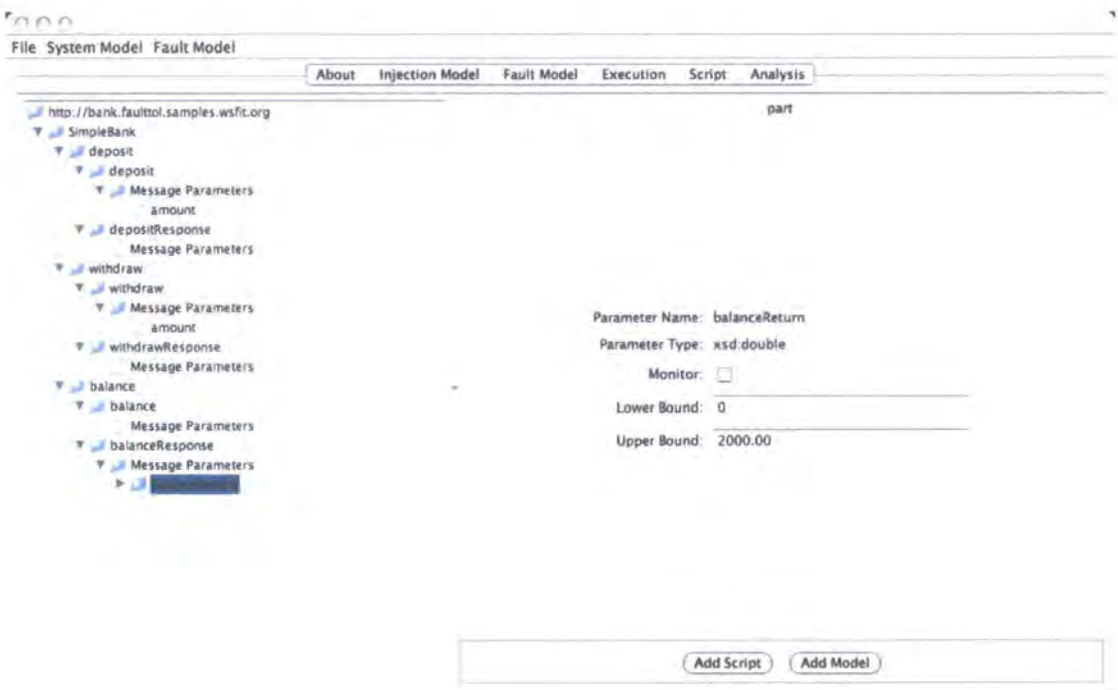


Figure A-9: Manually adding bounds information to an RPC parameter

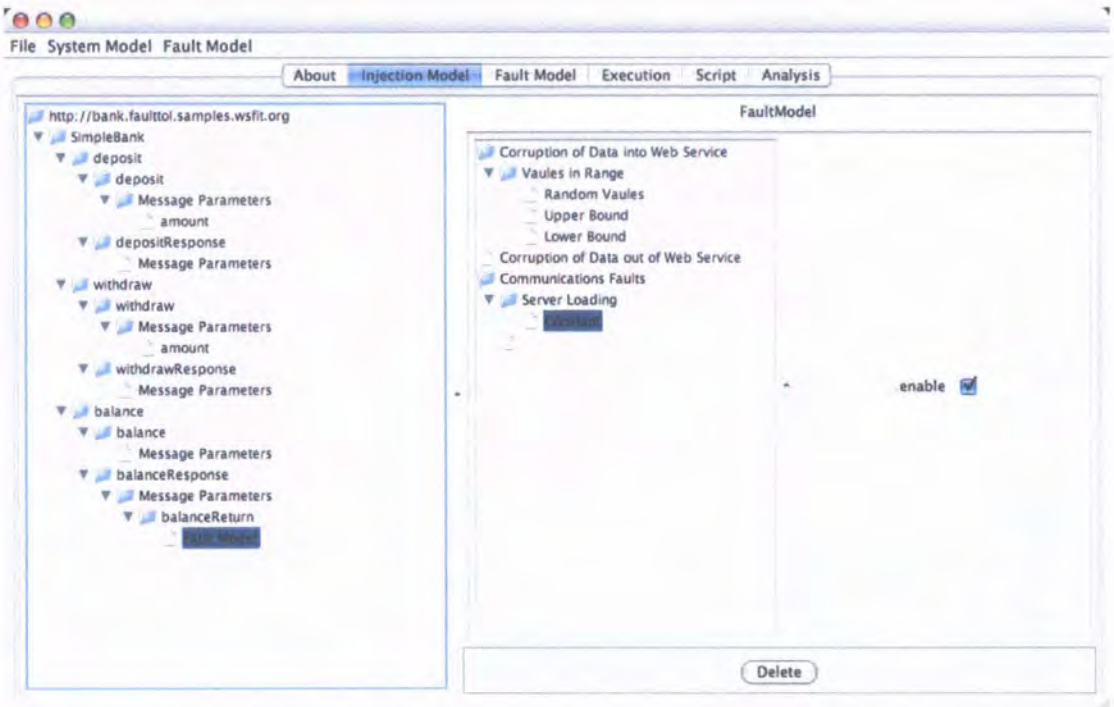


Figure A-10: Associating a Fault Model from the EFM to an RPC parameter

A.3 Execution

Once a System Model has been constructed a fault injection campaign script is automatically generated (See Figure A-11). The fault injection campaign script is generated by executing all the fault model generation scripts in sequence whilst generating the appropriate trigger code for each generated fragment. The script can be viewed on the Script tab that is included for debugging purposes.

The script fragment shown in Figure A-11 shows the trigger code to trigger on the balanceReturn parameter of the balance response message from the bank Web Service. The fault model applied generates a 1 second timed delay in returning the result but otherwise leaves the message unaltered.

Once the correct fault injection campaign script has been generated it can be executed using the Execute tab. The execute tab is structured as two panes:

- The top panel which displays visualization information
- The bottom panel that contains context sensitive buttons

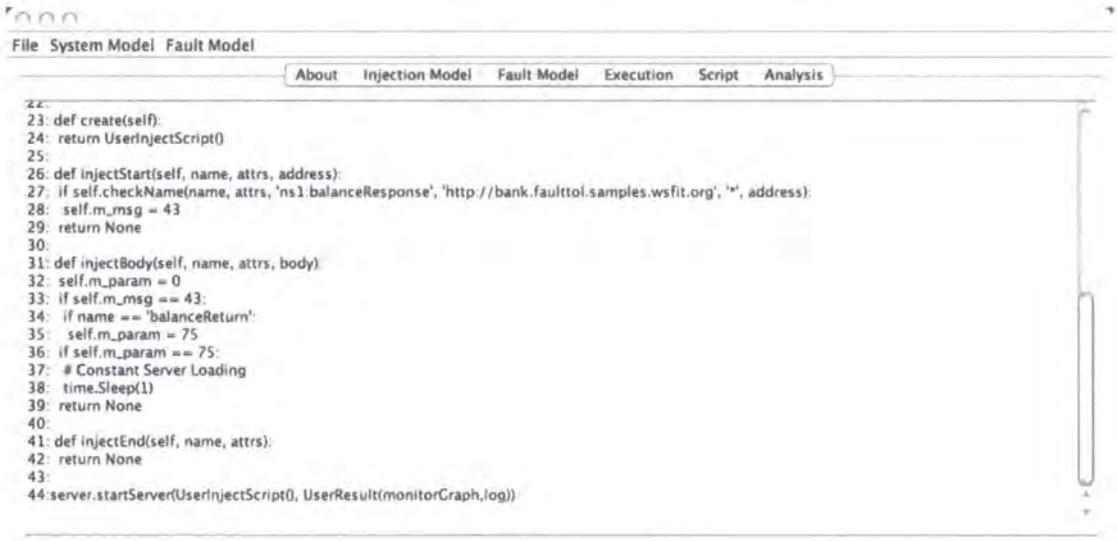


Figure A-11: Scripts generated from System Model using applied EFM

To start a fault injection campaign running the start button is pressed. This prompts the user to select a log file (See Figure A-12) that will be used to log the fault injection campaign as described in Section 5.2. Once the fault injection campaign is executing the top panel will display simple visualizations of RPC parameters (See Figure A-13). Simple numeric parameter in RPC can be configured to be monitored by selecting a 'Monitor' check box for the parameter in the system model (See Figure A-9).

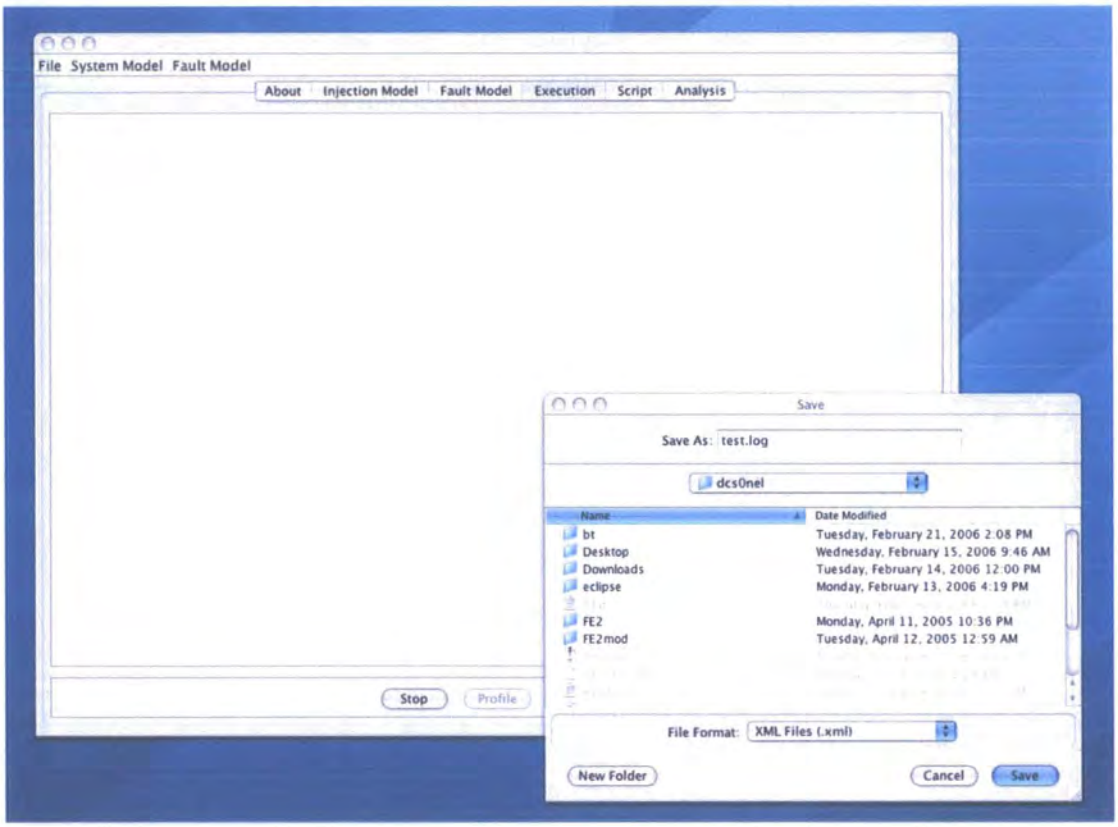


Figure A-12: Specifying a log file to record fault injection campaign execution

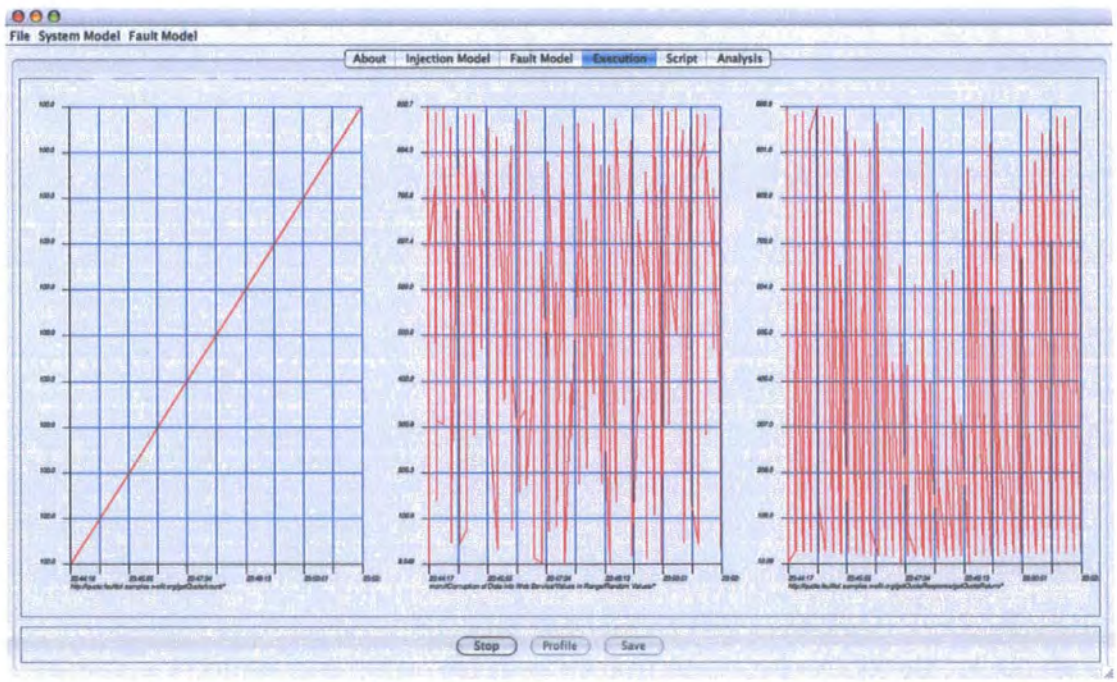


Figure A-13: Fault Injection Campaign execution

A.4 Summary

This appendix has documented how the WS-FIT tool realises the FIT method (Chapter 4) and implementation (Chapter 5) and has briefly shown how the EFM and System model can be constructed and used to assess a SOA.

This chapter has also examined under what circumstances WS-FIT can be considered non-intrusive as far as introduced latency is concerned. The demonstrations documented here indicate parameters for using WS-FIT.

Firstly, under normal conditions an RPC exchange over an Internet connection can take of the order of seconds to complete, including Web Service execution time with the default timeout for Axis SOAP being 10 seconds. This baseline demonstrated that there can be a large variation in the execution time of a Web Service RPC running over an Internet connection depending on the load that connection is under. WS-FIT can be considered acceptable if its induced latency falls within this variability.

Secondly, it was demonstrated that the number of triggers utilised by WS-FIT in a fault injection campaign gave a linear rise in latency which was within the standard deviation of the baseline non-instrumented data over the number of triggers tested and was thus and acceptable overhead.

Thirdly, it was demonstrated that for a Web Service RPC which utilised multiple element data structures such as arrays the latency introduced into the SOA rose linearly depending on the number of elements in the data structure. The same SOA instrumented for use with WS-FIT gives a polynomial rise dependent on message size. This is to be expected since the SOAP message is parsed and each element within the message must be processed.

By examining the data from the experiment an upper bound of 450 elements could be chosen since this falls below the default timeout of 10 seconds although under these circumstances WS-FIT could affect the running system. This upper value could also be adjusted if a larger timeout was set in the SOA to allow for long processing times by the Web Service etc.

Bibliography

[1] "Distributed Component Object Model (DCOM) Binary Protocol," Microsoft Corporation, 1997.

[2] "CORBA Component Model," Object Managment Group, 2002.

[3] "Overview of Jython Documentation," 2003.

[4] "Universal Description, Discovery, and Integration (UDDI)," 2005.

[5] Apache, "Client-Side Axis," vol. 2005: Apache Software Foundation, 2005.

[6] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of Physical and Software-Implemented Fault Injection Techniques," *IEEE Transactions on Computers*, vol. 52, pp. 1115-1133, 2003.

[7] A. Avizienis, "The N-version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1491-1501, 1985.

[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33, 2004.

[9] A. Avizienis, V. Magnus U, J. C. Laprie, and B. Randell, "Fundamental Concepts of Dependability," presented at ISW-2000, Cambridge, MA, 2000.

[10] K. H. Bennett, N. E. Gold, P. J. Layzell, F. Zhu, O. P. Brereton, D. Budgen, J. Keane, I. Kotsiopoulos, M. Turner, J. Xu, O. Almilaji, J. C. Chen, and A. Owrak, "A Broker Architecture for Integrating Data Using a Web Services Environment," presented at 1st International Conference on Service-Oriented Computing, Trento, Italy, 2003.

[11] K. H. Bennett and J. Xu, "Software services and software maintenance," presented at Software Maintenance and Reengineering, 2003.

[12] M. Berry, R. Jeffery, A. D. Birrel, and B. J. Nelson, "Implementing Remote Prodecure Calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39 - 59, 1984.

[13] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2(1), pp. 39-59, 1984.

[14] L. E. Boone and D. L. Kurtz, *Contemporary Business*, 5th ed: Dryden Press, 1988.

[15] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple Object Access Protocol (SOAP) 1.1," 1.1 ed: W3C, 2000.

[16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, "Extensible Markup Language (XML) 1.0," W3C, 2000.

[17] B. B. Brey, "The 8086/8088 microprocessor : architecture, programming, and interfacing." Columbus: Merrill Publishing Company, 1987, pp. 128.

[18] L. F. Cabrera, C. Kurt, and D. Box, "An Introduction to the Web Services Architecture and Its Specifications," in *MSDN White Paper*, vol. 2006, Microsoft, Ed.: Microsoft, 2004.

[19] J. Carreira and J. G. Silva, "Why Do some (weird) People Inject Faults?," *Software Engineering Notes*, vol. 23, pp. 42-43, 1998.

[20] J. V. Carreira, D. Costa, and S. J. G, "Fault Injection Spot-Checks Computer System Dependability," *IEEE Spectrum*, pp. 50-55, 1999.

[21] K. Channabasavaiah, K. Holley, and E. M. Tuggle, "Migrating to a Service-Oriented Architecture, Part 1," IBM Whitepaper, 2003.

[22] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL)," W3C, 2001.

[23] 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.

[24] M. Daran and P. Thevenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *Software Engineering Notes*, vol. 21, pp. 158-171, 1996.

[25] 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, USA, 1996.

[26] 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.

[27] T. Erl, *Service-Oriented Architecuture: A Field Guide to Integrating XML and Web Services*: Prentice Hall, 2004.

[28] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Argonne National Laboratory 2002.

[29] A. Fox and D. Patterson, "Approaches to Recovery-Oriented Computing," *IEEE Internet Computing*, vol. 9, pp. 14-16, 2005.

[30] S. Ghosh, "Fault Injection Testing for Distributed Object Systems," *Tools*, pp. 276-285, 2001.

[31] A. Gorbenko, V. Kharchenko, P. Popov, A. Romanovsky, and A. Boyarchuk, "Development of Dependable Web Services out of Undependable Web Components," School of Computing Science, University of Newcastle upon Tyne 2004.

[32] S. Graham, D. Davis, S. Simeonov, G. Daniels, P. Brittenham, Y. Nakamura, P. Fremantle, D. Koenig, and C. Zentner, *Building Web Services with Java: Making Sence of XML, SOAP, WSDL, and UDDI*: Sams, 2004.

[33] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, "SOAP Version 1.2 Part 2: Adjuncts," vol. 2005: W3C, 2003.

[34] H. Haas and A. Brown, "Web Services Glossary," W3C, 2004.

[35] 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.

[36] E. R. Harold and W. S. Means, *XML in a nutshell : A Desktop Quick Reference*: O'Reilly, 2001.

[37] H. Hecht and M. Hecht, "Qualitative Interpretation of Software Test Data," presented at Computer-aided design, test, and evaluation for dependability, Beijing, 1996.

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

[39] IFIP, "IFIP WG10.4 on Dependable Computing and Fault Tolerance," International Federation For Information Processing, 1988.

[40] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," presented at Network and distributed system security, San Diego, CA, 2000.

[41] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure Data Analysis of a LAN of Windows NT based Computers," presented at Reliable distributed systems, Lausanne, Switzerland, 1999.

[42] 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.

[43] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Transactions on Computers*, vol. 44, pp. 248, 1995.

[44] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering*, vol. 12, pp. 96-109, 1986.

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

[46] C. Lee, "Grid RPC, Events and Messaging," Global Grid Forum Advanced Programming Models Research Group Whitepaper, September 2001.

[47] B. Littlewood, P. Popov, and L. Strigini, "Choosing between Fault-Tolerance and Increased V&V for Improving Reliability," presented at DSN, New York, USA, 2000.

[48] N. Looker and M. Munro, "WS-FTM: A Fault Tolerance Mechanism for Web Services," University of Durham, 21st March 2005.

[49] M. R. Lyu, *Software Fault Tolerance*. Chichester ; New York: John Wiley, 1995.

[50] 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.

[51] A. Mani and A. Nagarajan, "Understanding Quality of Service for Web Services," vol. 2005: IBM, 2002.

[52] 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.

[53] E. Marsden and J. C. Fabre, "Failure Mode Analysis of CORBA Service Implementations," *Lecture Notes in Computer Science*, pp. 216-231, 2001.

[54] D. Mennie and B. Pagurek, "On Dynamic Service Composition and Its Applicability to E-Business Software Systems," presented at Workshop on Object-Oriented Business Solutions at ECOOP 2001, Budapest, Hungary, 2001.

[55] H. D. Mills, "On the Statistical Validation of Computer Programs," IBM Federal Systems Division 1972.

[56] L. J. Morell and J. M. Voas, "Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability," College of William and Mary in Virginia, Department of Computer Science September 1998 1988.

[57] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the Association of Computing Machinery*, vol. 27, pp. 228-234, 1980.

[58] S. Potts and M. Kopack, *Teach Yourself Web Services in 24 Hours*: SAMS, 2003.

[59] R. S. Pressman, "Software Engineering: A Practitioner's Approach," McGraw-Hill, 1992, pp. 549-594.

[60] I. Sommerville, *Software Engineering*: Addison-Wesley, 2004.

[61] I. Sommerville, "Verification and Validation," in *Software Engineering*: Addison-Wesley, 2004, pp. 513-588.

[62] W. Stallings, "Computer Organization and Architecture: Principles of Structure and Function," Macmillan Publishing Company, 1990, pp. 481-482.

[63] R. Steinmetz and K. Nahrstedt, *Multimedia Computing, Communications & Applications*: Prentice Hall, 1995.

[64] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," presented at ACM SIGCOMM Conference, Stockholm, Sweden, 2000.

[65] A. S. Tanenbaum, *Distributed Operating Systems*. London: Prentice-Hall International, 1995.

[66] A. S. Tanenbaum, *Distributed operating systems*. London: Prentice-Hall International, 1995.

[67] A. S. Tanenbaum and M. v. Steen, "Distributed Systems: Principles and Paradigms," Prentice Hall, 2002, pp. 393-400.

[68] P. Townend and J. Xu, "Replication-based Fault Tolerance in a Grid Environment," presented at U.K. e-Science 3rd All-Hands Meeting, Nottingham, UK, 2004.

[69] P. Townend and J. Xu, "Dependability in Grids," in *IEEE Distributed Systems ONLINE*, vol. 6, 2005.

[70] P. Townend, J. Xu, and M. Munro, "Building Dependable Software for Critical Applications: Multi-version Software versus One Good Version," presented at Workshop on Object-Oriented Real-Time Dependable Systems, Rome, Italy, 2001.

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

[72] T. K. Tsai, R. K. Iyer, and D. Jewitt, "An Approach towards Benchmarking of Fault-Tolerant Commercial Systems," presented at Fault-tolerant computing, Sendai; Japan, 1996.

[73] S. Vinoski, "Where is Middleware?," *IEEE Internet Computing*, vol. 6, pp. 83-85, April 2002.

[74] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*: John Wiley & Sons, 1998.

[75] A. Vogel, B. Gray, and K. Duddy, "Understanding any IDL - Lesson one: DCE and CORBA," presented at 3rd Workshop on Services in Distributed and Networked Environments (SDNE '96), 1996.

[76] J. A. Whittaker, "Software's Invisible Users," *IEEE Software*, vol. 18, pp. 84-88, 2001.

[77] J. A. Whittaker and H. H. Thompson, *How to Break Software Security*: Addison-Wesley, 2003.

[78] J. Zhang, "An Approach to Facilitate Reliability Testing of Web Service Components," presented at International Symposium on Software Reliability Engineering and Technology, Saint-Malo, Bretagne, France, 2004.