Fault Injection Testing Method of Software Implemented Fault Tolerance Mechanisms of Web Service Systems

Thesis by

Khaled Farj

In Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

Newcastle University
Newcastle upon Tyne, UK

September, 2011

Newcastle University

Abstract

Testing Web Services applications and their Fault Tolerance Mechanisms (FTMs) is crucial for the development of today's applications. The performance and FTMs of composed service systems are hard to measure at design time because service instability is often caused by the nature of the network. Testing in a real internet environment is difficult to set up and control. However, the adequacy of FTMs and the performance of Web Service applications can be tested efficiently by injecting faults and observing how the target system performs under faulty conditions.

This thesis investigates what is involved in testing the software-implemented fault tolerance mechanisms of Web Service systems through fault injection. We have developed a fault injection toolkit that emulates a WAN within a LAN environment between composed service components and offers full control over the emulated environments, in addition to the ability to inject communication and specific software faults. The tool also generates background workloads on the tested system for producing more realistic results.

The testing method requires that the target system be constructed as a collection of Web Services applications interacting via messages. This enables the insertion of faults into the target system to emulate the incorrect behaviour of faulty conditions by injecting communication faults and manipulating messages. This approach allows the injection of faults while not requiring any significant changes to the target system.

This testing method injects two classes of faults, manly communication and interface faults due to their big impact on Web service system dependability. The method differs from the previous work not only by injecting communication faults based on a Wide Area Network emulator, but also in its ability to inject a combination of communication and interface faults, which could cause what are called Byzantine faults (Arbitrary faults) at the application level.

The proposed fault injection method has been applied to test a Web Service system deploying what is called a WS-Mediator for improving the system reliability. The WS-Mediator claims to offer comprehensive off-the-shelf fault tolerance mechanisms to cope with various kinds of typical Web Service application scenarios. We chose to use the N-version programming mechanism offered by the WS-Mediator, which has been tested through out tool. The testing demonstrated the usefulness of the method and its capacity to test the target system under different circumstances and faulty conditions.

Newcastle University

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Newcastle 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:

1: first paper:

Farj. K, Speirs. N. A. (2009). ‘A Tool for Assessing Fault Tolerance Mechanisms applied to Web Service applications’ in *12th European Workshop on Dependable Computing (EWDC 2009).* Toulouse, France, 2.

2: second paper:

Farj. K, Speirs. N. A. (2010). ‘A Tool for Testing Fault Tolerance of Web Service Systems’. *2010 International Conference on Dependable Systems and Networks: Supplemental*. Chicago, Illinois, IEEE.

3: third paper:

Farj K, C.Y., Speirs. N. A. (2012), A Fault Injection Method for Testing Dependable Web Service Systems. *9th European Dependable Computing Conference* - EDCC 2012, 2012: p. 10.

Newcastle University

Acknowledgements

It is a pleasure to thank the people, who contributed in various ways to this thesis, making it possible.

First, I would like to thank my PhD supervisor, Dr. Neil Speirs. With his enthusiasm, inspiration and encouragement, he helped me to carry out the research and complete the work in various ways, providing explanation when necessary, advising me on my reading as well as the relevant work in close research domains, and many more. I would have been lost without his huge support.

I would also like to thank the members of my thesis committee board. They provided invaluable comments and suggestions, helping to keep the work on the right track.

There are many other people who assisted me at different stages of the research. I would like to express my gratitude to them. I am especially grateful to Dr. Nick Cook and Prof. Santosh Shrivastava for their kind assistance with the experimental work on the Web Services used in their research projects. They provided us with the information on the Web Services, helping us to set up the experiments. I wish to express my warm and sincere thanks to Mrs. Elizabeth Brooks. She greatly helped me to proof read and edit the thesis.

This list would not be complete without my family, on whose constant encouragement and love I have relied throughout my time at University. Without their unflinching support and understanding, it would have been impossible for me to finish this study.

It is to them that I dedicate this research.

Newcastle University

List of Figures

Figure 2-1 typical Web Service architecture. 11
Figure 2-2 client-side fault injection architecture 32
Figure 2-3 suggested fault injection architecture 33
Figure 3-1 A Simple composed service system 41
Figure 3-2 A general view of the initial proposed approach 46
Figure 3-3 how a WAN emulator runS over a LAN 49
Figure 3-4 Sample Wide Area Network 52
Figure 3-5 A high level view of the initial proposed Network Emulation 53
Figure 3-6. Initial high level view of WAN emulation connected to FIM 55
Figure 3-7 Fault classification hierarchy 59
Figure 3-8 Overview of the Network Fault Injection location 65
Figure 3-9 Software value faults 66
Figure 3-10 Software value injector 70
Figure 4-1 Web Service Middleware System 79
Figure 4-2 the chosen location for injecting faults into the system. 80
Figure 4-3 a general view of the Interception Mediator placed into a composed service system 82
Figure 4-4 Network Fault Injection Service (Netfis) 85
Figure 4-5 Request message through NetFIS 86
Figure 4-6 Response message through NetFIS 88
Figure 4-7 Message processed by FIE to inject Network Faults 96
Figure 4-8 Message processed by FIE to inject softeware Faults 98
Figure 4-9 a general view of the Network Node Emulators placed into a composed service system 102
Figure 5-1 Simple Network topology 122
Figure 5-2 A system being tested by the tool 124
Figure 5-3 Client invocation RTT 126
Figure 5-4 RTT of the test cases (Web Service 1) 129
Figure 5-5 RTT of the test cases (Client) 132

Newcastle University

List of Tables

Table 2-1 basic structure of an XML document 13
Table 2-2 Typical SOAP request Message 15
Table 2-3 typical SOAP response message 16
Table 2-4 typical SOAP Fault Message 17
Table 2-5 WSDL Document Structure Example 18
Table 3-1 How to inject network faults using our method 63
Table 3-2 Parameter values mutation algorithms. 69
Table 3-3 possible combination of injected faults 72
Table 4-1 a sample network topology with snippets of the topology file describing it 106
Table 4-2 a simple trace file 109
Table 4-3 A snippit of simple network topolgy file 111
Table 4-4 Value Fault Model File 114
Table 4-5 A Simple Example of A logging file 117
Table 5-1 response-time overhead 127
Table 5-2 drop and random error injected 130
Table 5-3 The affect of injected Drop and Error test cases (Web Service 1) 133

Contents

1. Chapter 1 - Introduction ... 1
   1.1 Motivation ... 1
   1.2 Our Research ... 3
   1.3 Our contributions ... 4
   1.4 Thesis Outline ... 5
2 Chapter 2 - Service-Oriented Architecture and Dependability ... 7
   2.1 Preliminaries ... 7
      2.1.1 Service-Oriented Architecture ... 7
      2.1.2 Web Services ... 8
   2.2 SOA, Web Services and their Dependability ... 8
      2.2.1 Overview of SOA and Web Services ... 9
      2.2.2 Web Service architecture ... 10
   2.3 Dependability assessment ... 19
      2.3.1 Means for Achieving Dependability ... 21
      2.3.2 Assessing dependability ... 22
      2.3.3 Fault injection ... 22
      2.3.4 The communication and interface faults impact on Web service systems ... 27
      2.3.5 Existing Fault Injection Tools ... 29
   2.4 Conclusions ... 33
3 Chapter 3 - Fault Injection Method for testing Web Services ... 35
   3.1 Problems Involved in Testing Web Service Systems ... 35
      3.1.1 Problem of modifying the system under test ... 36
      3.1.2 Problems of Setting up a distributed testing environment ... 36
      3.1.3 Problem of testing composed services for performance and fault tolerance ... 38
      3.1.4 Limitation of previous work ... 39
   3.2 Overview of the Proposed Approach ... 40
   3.3 Design Approach ... 42
      3.3.1 Fault injection location mechanism ... 43
      3.3.2 A General proposed architecture of fault injection location ... 45
      3.3.3 Run-time Environment ... 47
      3.3.4 Faults affecting web service systems ... 56
      3.3.5 Timing Faults ... 59
      3.3.6 Software faults ... 65

3.3.7 Value and Network faults injected together 71
3.3.8 System monitoring (failure detection) 72
3.4 Conclusions 76
4 Chapter 4 - NetFIS applied to SOAP based Web Service 78
4.1 Web Service middleware system 78
4.2 Fault injection mechanism 80
4.2.1 Fault Injection Mediator Service 84
4.2.2 Fault injection scenario cases 95
4.3 Network Emulation Mechanism 101
4.3.1 Network Controller Service (NCS) 105
4.3.2 Network Node Emulator Service (NNES) 106
4.4 Injecting Network Faults Mechanism 110
4.5 Injecting Software Faults Mechanism 112
4.6 Failure Detection Mechanism 116
4.7 Conclusions 118
5 Chapter 5 - Case studies 120
5.1 Testing a real web service system 120
5.1.1 Setting up the tool 120
5.1.2 Experiment setup 121
5.1.3 Network configuration: 122
5.1.4 Client configuration: 123
5.1.5 Experimental Results 125
5.2 Conclusions 136
6 Chapter 6 - Conclusions and Suggestions for Future Work 137
6.1 Contributions 137
6.2 Limitations and Future Work 140
7 Bibliography 142
8 Appendix – The Performance of NetFIS tool 151

1. Chapter 1 - Introduction

1.1 Motivation

Web Services [1] and Service-Oriented Architectures (SOAs) [2] represent a new paradigm for building distributed computing applications [3][4]. Their applications include such applications as e-commerce [5], for example, online auctions [6]. The advantages of Web Services, such as their loosely-coupled architecture and standardized interoperability, are attracting more and more users, along with growing body of work in the relevant research and development domains. Given the importance of this technology, it is essential that methods are developed to ensure that dependable and reliable software services are deployed.

Web Services have addressed many existing issues in the conventional technologies, such as Enterprise Application Integration (EAI) [7] and Common Object Request Broker Architecture (CORBA) [8], to name two of the more popular ones that have been extensively applied in the past decades. In these conventional distributed applications, service integration commonly relies on centralized brokers, or coordinators, which implement *object-based* or *message-based interoperability* [4] with the participating component services and interact with them to perform automated business processes. The limitation of such a paradigm lies in the fact that the middleware has to be centralized and trusted by all participating component service providers. Consequently, this becomes an issue for the integration of cross-organizations and heterogeneous services [4].

Web Services resolve these issues with their loosely-coupled interaction model, standardized interoperability, extended peer-to-peer integration fashion and so forth [4]. In Web Services, functionalities implemented by internal business procedures are deployed and exposed as services that can be discovered and connected through the Web. The interaction between the Service client and the Service provider generally relies on the SOAP/HTTP message binding [9][10][11]. The client invokes Web services by sending a SOAP request message [2][11]. Web services receive the SOAP request message and process it and return the results to the client via SOAP response messages. During the integration, the client does not necessarily know anything about the Web services involved, other than their WSDL interface [12]; the

communication between them is guaranteed by the standardized interoperability, and no third party service broker or coordinator is required. Therefore, compared with the conventional technologies, the integration of autonomous and independent services is achieved in Web services at a low cost.

Nevertheless, even with the advantages described above, Web services do not resolve all problems of distributed applications. Web service middleware, similarly to other distributed technologies, relies on existing underlying middleware, such as network protocols, to implement the essential low-level services [4]. Thus Web Services inherit many of the dependability issues the conventional middleware infrastructure suffers from. For example, the interaction between the client and Web services relies on the Web or other networks, which are inherently unreliable media that may cause communication problems such as a loss, delay or damage of the message [3][13][14]; also Web services are deployed on application servers, which may become unreliable or out-of service, due to system maintenance or other internal activities [15]; the design or implementation of the Web services business procedure may contain faults and result in their erratic behaviour [15]. Thus, their dependability is a vital issue in dependability-critical applications, even more so in those based on a service composition in which a service, as an undependable component, can undermine the dependability of the entire application. Thus, it is only to be expected that testing the dependability of Web services has attracted active interest as a research domain in recent years.

Testing dependability is a realistic approach for 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. Testing dependability is therefore important to aid in increasing the reliability of a system, not only to uncover existing problems with services.

1.2 Our Research

This dissertation reports our work in assessing Web Service systems by using fault injection techniques. We started the research by investigating the assessment of dependability in the context of Web services, followed by an in-depth analysis of the most common faults affecting Web Service based systems. At the same time, we studied related work conducted by other researchers working in similar research areas. As a result, we have developed a novel solution for injecting a combination of faults in order to test and improve the dependability of Web services.

Fault injection is a well-known method of testing software systems through exercising systems by injecting faults. Although much work has been done in the area of fault injection and Web Service systems in general, it appears to lack some important characteristics in terms of the kinds of faults injected, the workload during testing, and testing all parties (components) of the system, independent of the hosting testing environment.

Conceptually, this solution is based on our understanding of the specific testing characteristics of Web services. It addresses some testing issues that have not yet been covered by the existing work. In particular, our research focuses on the problem domain from certain original perspectives, avoiding a duplication of others' work. We have overcome some of the problems mentioned above through: adopting several novel approaches and concepts in the solution proposed; developing certain unique mechanisms to ensure the novelty and efficiency of our approach; and proving them in a series of experiments with real world Web services.

In this work we proposed a method for testing Web Service applications in order to overcome the above problems [16][17]. The testing method is for testing the performance and fault tolerance of either a single Web Service or a composed service, without any modification to the system being tested. No recompiling or patching is necessary. In addition, two classes of faults should be injected: communication faults and interface faults. Furthermore, the tool should generate background workload to more accurately emulate real

networks in order to produce more realistic results. Finally, the tool should also be independent of the hosting environment for portability.

1.3 Our contributions

While the recent active research efforts aimed at testing Web services have developed some effective fault injection testing methods, including those focusing on service composition, we believe that there are still many issues remaining in this domain, particularly concerning the testing of service composition that relies on autonomous Web services. We have learnt from many related works [18][19] that in Web Service systems the two sides (Service requester and Service provider) might be affected dramatically by communication faults, which can be a consequence of network faults and software faults.

In the internet world, the communication channels are very unreliable. It is hard to know when the next error situation will occur, such as connection break, messages failing to be delivered or being delayed more than expected, or messages transmitted in the wrong sequence, and even worse the application crashing during message transmission. From the literatures, there are many evidences indicating that communication and interface faults have a big affect of Web service system dependability, we elaborate on this in section 2.3.4.
The effect of each of these failure modes will depend on the fault tolerance capability of the system, in detecting them and preventing the system from deviating from its specified behaviour. This therefore calls for testing method solutions that would assess these fault tolerance mechanisms of Web service systems and their dependability from two sides (client and service sides). In order to address this, our testing method in assessing Web service applications is based on three main issues: injecting communication faults, injecting software faults and providing a network environment emulator. The main contributions of our work are as follows:

- We have developed a NetFIS tool to test the dependability and the fault tolerances of Web Service applications. The approach offers an off-the-shelf mediator to test

service composition applications without any modification to the system under test. The tool is independent of any hosting environment for portability purposes.

- By using NetFIS Communication faults are injected at application message level rather than at network message level. This allows faults to be injected on an entire application message and affect a specific middleware service, rather than test error recovery in a network protocol stack. Communication faults are injected based on a WAN emulator which generates background workload to more accurately emulate real networks in order to produce more realistic results. Emulating additional workload during the testing of a system could give rise to different results. Different workloads could lead to different testing results, due to causing different system activation patterns [20].

- By using NetFIS Software faults are injected through decoding middleware messages. Faults are injected into middleware messages by using information on message formats derived from interface definition language definitions of a service to allow targeted triggering on specific messages corresponding to specific operations. Using such information-specific parameters can be perturbed in an RPC exchange, allowing software faults to be injected alongside communication faults in order to emulate what are called Byzantine faults.

1.4 Thesis Outline

The dissertation is organised as follows:

- Chapter 2 explains the fundamental concepts and definitions of SOA and Web services. We explain how dependability is assessed in the context of Web services and analyse fault injection techniques and their roles in testing Web Service systems. Finally, we summarize some related work in the area.

- Chapter 3 presents some existing problems in testing Web Service systems and suggests some solutions by presenting our NetFIS approach. In this chapter we discuss some problems in the existing related work that we are trying to resolve and introduce our testing method as well as explaining the NetFIS architecture and its components in detail.

- Chapter 4 introduces a prototype of the NetFIS tool. In this chapter, we explain how to implement the NetFIS system using the Java Web service technology.

- Chapter 5 reports on the experiments conducted to evaluate the NetFIS approach. The results of the experiments with real-world Web services are analysed to demonstrate the applicability of the NetFIS approach.

- Chapter 6 concludes this dissertation, offering our vision for the possible further development of the NetFIS testing tool system.

2 Chapter 2 - Service-Oriented Architecture and Dependability

This chapter will define the concepts and technology of Service-Oriented Architecture (SOA) and Web services, which is particularly relevant to this thesis. We will explore a general definition of SOA and then explore how SOA is implemented via Web services.

Furthermore, having introduced the general concept of dependability, we will discuss what dependability means in the Web services context that our work is concerned with. Thereafter, we will explain how dependability can be assessed in existing Web service applications. Lastly, we will review studies conducted by other researchers working in related fields of some relevant assessment tools for testing the dependability of web service applications and we will provide a brief overview of the background and foundation that our work is built upon.

2.1 Preliminaries

Although often used, the terms Service-Oriented Architecture (*SOA)* and *Web Services* are not always consistently defined. It is, however, important to clearly define these terms here as fundamental concepts for this thesis.

2.1.1 Service-Oriented Architecture

The definition of *SOA* as provided by the World Wide Web Consortium (WC3) [2] is as follows:

Service-Oriented Architecture: A set of components which can be invoked, and whose interface descriptions can be published and discovered.

The above definition is a basic definition that describes what SOA is, and yet it is rather abstract: it does not explicitly define the underlying concepts and technologies it relies upon. It is the specification [1] that refines the definition, presenting SOA as a form of distributed systems architecture in which services implement an abstracted interface for exchanging messages with clients and other services. The machine-processable abstracted interface

describes only those details of services that are important for use by clients. The service implementation details and internal structure are hidden from clients.

The messages exchanged between services and clients rely on an underlying computer network, such as the Internet. The actual technologies used for constructing a SOA are not made specific in these definitions and may vary in real world applications.

2.1.2 Web Services

The definition of Web Services is also given in [1] as follows:

Web Service: a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Comparing the above definition with that of SOA, it becomes obvious that Web Services are a form of SOA. The definition explicitly specifies the underlying technologies involved in constructing Web Services. Some of these technologies, such as the Web Service Description Language (WSDL) [12] and the Simple Object Access Protocol (SOAP) [11], have been purposefully developed for Web Services, while others have been adapted from the existing standards and protocols, such as the Hyper-Text Transport Protocol (HTTP) [10] and the Extensible Mark-up Language (XML) [21].

2.2 SOA, Web Services and their Dependability

SOA and Web Service technologies have been developing extraordinarily fast in the recent years. They are becoming significant in many businesses and scientific distributed computing applications [22] and thus prompting a great deal of research interest in the field of dependability. In [15], the term *dependability* covers varied characteristics; however the meaning of dependability may vary from one context to another. In this work we will

describe the meaning of dependability that our work is concerned with in terms of measurements. We will also explain a specific analysis of the dependability measurement issues commonly manifested in existing Web Service applications. Lastly, we will report on our studies of some relevant testing tools conducted by other researchers working in related fields for testing the dependability of web service applications.

2.2.1 Overview of SOA and Web Services

Service-Oriented-Architecture (SOA) has received much attention over the past years. SOA is an architecture design that relies on loosely coupled software components the called services, which can be orchestrated to improve business agility and can be shared among different domains . Therefore SOA and service orientation are based on the idea that application software supporting a business process should be composed of a collection of smaller, related pieces, that is, services.

The main building blocks of SOA are: the Service provider, the Service registry, and the Service requester. Service providers are software applications that provide a service to the Service requester (client). The provider publishes a description of the services they provide via a service registry. Service requesters consume the service. When considering service composition, a software application can act as both a Service requester and a Service provider at the same time. Clients must be able to find the description of the services they require and must be able to bind to them.

Web Services are one of the technologies implementing SOA principles. A web service is a platform-independent, loosely coupled, self contained, programmable web-enabled piece of application that can be described, published, discovered, coordinated and configured using XML [21] artefacts for the sake of developing distributed interoperable applications. Web Services allow applications to work together over standard internet protocols (HTTP - Hypertext Transfer Protocol) [10] to automate business operations without any intervention by humans [23]. In Web Services, clients and services are assumed to be loosely-coupled, which means that they are stand-alone systems independent of each other. The services are normally autonomous, and developed and deployed by different service providers.

Regarding service composition, Web Services also implement the composed service of the SOA, thus the term 'composed Web Service'. This is a system that is composed of many Web Services applications and client applications to form a distributed system. Composed Web Service refers to two or more Web Services applications interacting with one or more client applications. Web Service applications can be both the Service requester and the Service provider at the same time.

2.2.2 Web Service architecture

A Web Service is a software service defined by many standards that can be used to provide interoperable data exchange and processing between dissimilar machines and architectures. For the purposes of this work we are concerned with Web Services defined by the W3C, that are described by WSDL [12] and implemented using SOAP [11] and the Remote Procedure Call (RPC) model [24].

The organization providing the Web Service owns the Web Service and implements the business logic that underlies the Web Service. The Web Service is, then, published in a service registry which is hosted by a service discovery agency. The description of the Web Service contains information about the business, service, and technical information concerning the Web Service. The Web Service publisher has to describe this information in a registry, in a predefined format specified by the discovery agency. Figure 2-1 shows a typical Web Service architecture.

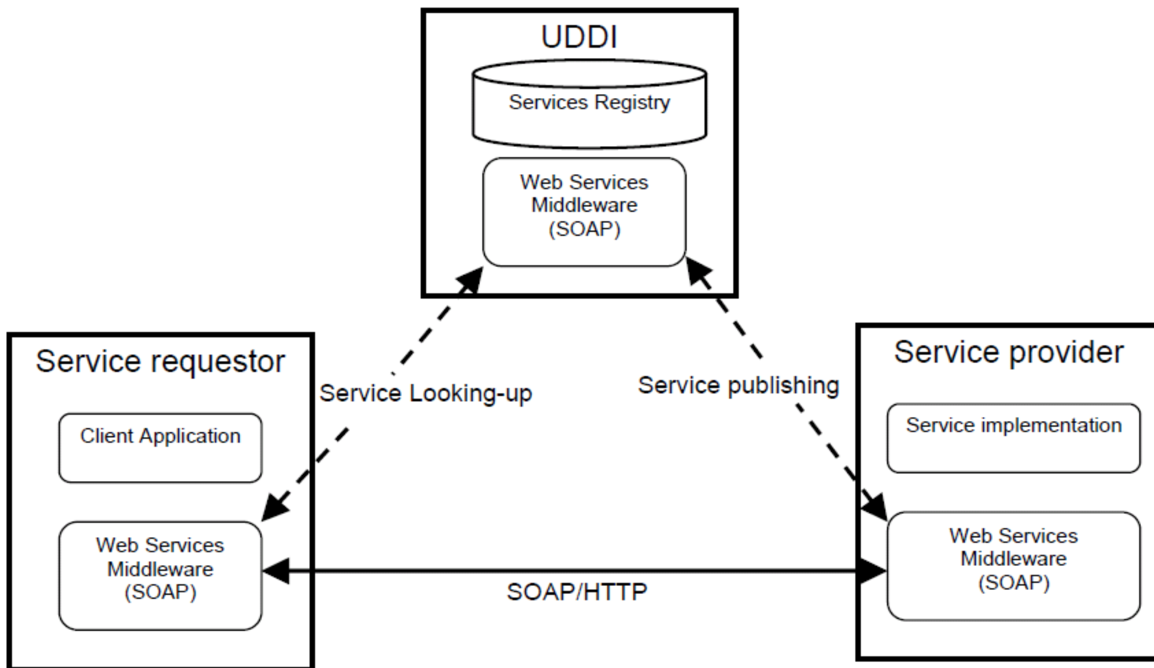


FIGURE 2-1 TYPICAL WEB SERVICE ARCHITECTURE.

The definition of Web Services (in Section 2.1.2) specifically states that a Web Service interface should be described in the WSDL. Clients interact with Web Services through SOAP messages relying on underlying network protocols such as HTTP. Clients can discover services through various discovery services, such as UDDI. The discovered information is sufficient for the client to make invocations on the Web Service.

2.2.2.1 Transport

Web Services are built up like an interoperable messaging architecture, and the transport technology is the foundation of this architecture. Web Services are inherently transport neutral, so one could transport Web Services messages though ubiquitous Web protocols such as HyperText Transport Protocol (HTTP) [10], Simple Mail Transfer Protocol (SMTP), Transmission Control Protocol (TCP) or Securer HTTP (HTTPS). Transport protocols are fundamental to Web Services because they are a defining factor in the scope of interoperability, but most of the time transport protocols details are hidden during the design

and development of Web Services applications, due to flexibility and platform interoperability.

Although as shown above, transportation is not restricted to a specific protocol or method, HTTP became the most popular way to exchange messages based on XML format between Web Services. Web Service systems typically utilize the HTTP protocol to transmit data between services using the XML format.

2.2.2.2 Remote Procedure Calls (RPC)

According to the concept of Remote Procedure Calls (RPC) [23], services communicate to each other by sending messages through the networks. RPCs have traditionally been procedures called in a programme on one machine (client) that go over the network to another programme (server) that actually implements the called procedure. The called procedure implementation details are hidden from the client programme in such a way as to make the invocation look like a normal routine call. Then the called programme bundles up the results of the procedure call and send those results back to the caller. The calling programme then continues executing. Hence RPC is a synchronous message passing paradigm.

Web Service RPC based applications use SOAP messages for formatting RPC calls over HTTP protocol according to the WSDL interface of the called Web Service [25].

2.2.2.3 Extensible Mark-up Language (XML)

Both WSDL and SOAP employ Extensible Mark-up Language (XML) [21] to define and implement Web Service message exchange. XML notation and terminology is used in this study and its basic terms are explained below.

XML is an abbreviation for eXtensible Mark-up Language [21]. It is designed to describe data and improve the functionality of the Web by providing more flexible and adaptable ways of information representation. It is called extensible and its format is not fixed like

HTML. Instead, XML is a meta-language that lets one design one's own customized markup languages. A markup is a mechanism to specify structures within a document, and the way to add a markup to a document is defined by the XML specification.

Unlike HTML, XML does not specify semantics or a set of tags. There is no prescribed method for rendering XML documents, so semantics will be defined by the application using it or by style sheets. The following
 is a simple example showing the basic structure of an XML document and how data is represented:

```
<?xml version="1.0" encoding="utf-8"?>
   <note noteID="1">
     <to>Bob</to>
     <from>Al</from>
     <heading>Meeting</heading>
     <body>university, computing school today</body>
   </note>
```

TABLE 2-1 BASIC STRUCTURE OF AN XML DOCUMENT

The above simple XML document starts with the XML declaration in the first line. It defines the XML version and the character encoding used. In this case the document uses version 1.0 of the XML standard and characters are encoded in UTF-8 (8-bit Unicode Transformation Format).

The next line describes the root element of the document. Elements are one way to store data in an XML document. The following four lines describe the child elements of the root (*to*, *from*, *heading* and *body*). By looking at the elements it is easy to see that the XML document represents a message. The last line describes the end of the root element, completing the *note from* Al *to* Bob. Along with the root element in the second line an attribute called *noteID* is specified. Attributes are additional ways to store data which are used to provide additional information about elements, also called meta-data.

A list of legal elements that defines the document structure is the Document Type Definition (DTD). A document with correct XML syntax is called "Well Formed" while a "Valid" XML document also conforms to a DTD.

More and more applications make use of XML to store information because of its benefits. Some of the benefits of using XML are:

- The structure is well-defined and can be passed between different computer systems which would otherwise be unable to communicate
- Data payload is encapsulated in XML tags and therefore readable by human viewers
- Due to their textual nature, XML-Files are platform-independent.

These advantages make XML a perfect format to communicate between Web Service systems. To ensure a platform- and language- independent use for every Web Service, SOAP was developed. It is an XML application with defined elements and a predefined structure. The following section will elaborate on SOAP in more detail.

2.2.2.4 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) [11] is one of the significant protocols of Web Services and provides a simple and relatively lightweight mechanism for exchanging structured and typed information between services.

> SOAP is designed to reduce the cost and complexity of integrating applications that are built on different platforms. SOAP has undergone revisions since its introduction, and the W3C has standardized the most recent version, SOAP 1.2. [26].

SOAP is an XML-based communication protocol for exchanging messages between services. Web Services rely on SOAP for exchanging messages between applications, regardless of their operating systems and programming environment.

Our research in this thesis is mainly concerned with the RPC mechanism over SOAP. This is defined by the W3C in [25] which describes a general purpose RPC mechanism. The message structure types that are involved in an RPC exchange and the relevant features used by our method are briefly detailed here.

A SOAP XML document instance is called a SOAP message or SOAP envelope. It is carried as the payload of some other network protocol such as HTTP [27]. A common SOAP structure has been documented in XML form with three main parts: an envelope, a header, and a body. Usually, the root element of the SOAP message is the envelope: it contains zero or more headers and at least one body. The header is an optional part of a SOAP message and it is a generic mechanism for adding extensible features to SOAP. The child element of the header is called the head block. SOAP defines several well-known attributes that one can use to indicate who should deal with a header block and whether processing it is optional or mandatory. The mandatory body element is always the last child element of the envelope: it includes the actual payload message content. The body element is populated with elements that make up the payload of a request, response or fault message.

An example of SOAP request messages is shown in Table 2-2. The body element contains one method call in the request. The called method's name is *GetStockPrice* whereas the method's parameter has the name *StockName* and the value *IBM*.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
     <soap:Body xmlns:m="http://www.example.org/stock">
          <m:GetStockPrice>
              <m:StockName>IBM</m:StockName>
         </m:GetStockPrice>
     </soap:Body>
</soap:Envelope>
```

TABLE 2-2 TYPICAL SOAP REQUEST MESSAGE

The *GetStockPrice* method is defined in the wsdl:operation (see next section) which contains parameter elements that represent the RPC Parameters. In the example above the *GetStockPrice* method takes one string parameter called *StockName* which contains the string data for that parameter which is *IBM*. Parameters are defined in WSDL by wsdl:part elements in wsdl:message elements (see next Section).

An example of a typical response message is shown in Table 2-3. The structure of the response message is similar to the request message, but the method response name is suffixed with the word Response, as defined in the wsdl:operation element in the WSDL.

```
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>
</soap:Envelope>
```

TABLE 2-3 TYPICAL SOAP RESPONSE MESSAGE

In the above example the response element name is *GetStockPriceResponse*. The response element contains elements that represent any method return value. Method return results follow the naming convention of the method name suffixed by the word Return and are represented in the WSDL by wsdl:part elements. The return element name is *Price* which contains the return data for that parameter, which is 34.5.

Table 2-4 shows a typical SOAP Fault Message. Fault Messages are used to return error information from a server to a client. The SOAP fault message contains a root element called `Fault` element which must appear within the Body element. The Fault element contains three elements: `faultcode`, `faultstring` and detail.

The `faultcode` element defines a small set of SOAP fault codes covering basic SOAP faults, and this set can be extended by applications. The `faultstring` element provides information about the fault in a form intended for a human reader. The `detail` element carries application-specific error information. For instance as the following table shows, when 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 the exception and the detail element is not used.

```
<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/XMLSchemainstance">
   <soapenv:Body>
     <soapenv:Fault>
         <faultcode>soapenv:Server.userException</faultcode>
         <faultstring>java.rmi.RemoteException: can&apos;t get a stock
price for this                    symbol
         </faultstring>
         <detail/>
     </soapenv:Fault>
   </soapenv:Body>
</soapenv:Envelope>
```

TABLE 2-4 TYPICAL SOAP FAULT MESSAGE

2.2.2.5 Services Description

The description of deployed services is a key aspect of Web Services technology. Such descriptions define metadata that fully describe the characteristics of services which are fundamental to achieving the loose coupling. It also accords with SOA architecture and provides the abstract information to deploy or interact with services.

The Web Services Description Language (WSDL) [12] is the leading standard for the description of Web Services and provides an XML-based syntax to specify the exposed interface and the location of a Web Service, as well as how to access it. It allows developers to describe the functions that a service performs. It tells the client what actions a service performs and how the messages are received and sent. In the Web Services world, SOAP is a message format that people use to understand the communication 'language', and WSDL is what people use to tell others what they can do.

"WSDL is an XML format for describing services as a set of endpoints that operate on message containing either document-oriented or procedure-oriented information." [26] (Weerawarana *et al.* 2005, p.40). As our research is mainly concerned with RPC message exchanges, the WSDL provides clear explicit information on the structure of message exchanges between Web Services and their clients.

The overall structure of WSDL documents is illustrated in Table 2-5 and can be split up into two parts. The abstract part, which consists of the *types*, *message*, and *portType* elements, describes the Web Service interface, that is, the exposed operations and used data types. The concrete part comprising the *service* and *binding* elements, describes a concrete implementation of the service's interface at a network endpoint, that is, where the Web Service exists and how it can be accessed.

```
<wsdl:definitions name=”..” targetNamespace=”uri”>
   <import namespace=”uri” location=”uri”/>
   <wsdl:types>
   …
   </wsdl:types>
   <wsdl message name=”..”>
   …
   </wsdl:message>
   <wsdl:portType name=”..”>
   …
   </wsdl:portType>
   <wsdl:binding name=”..” type=”..”>
   …
   </wsdl:binding>
   <wsdl:service name=”..”>
     <wsdl:port name=”..” binding=”..”>
     …
     </wsdl:port>
     …
   </wsdl:service>
</wsdl:definitions>
```

TABLE 2-5 WSDL DOCUMENT STRUCTURE EXAMPLE

The major elements in the above WSDL example are as follows:

- definitions – defines a bag of definitions for a single namespace;
- types – provides data type definitions used to describe the message exchanged;
- message – represents the description of messages exchanged in the Web Service;
- portType – is a set of abstract operations;
- binding – specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType;
- port – specifies an address for a binding, contains the endpoint address itself and refers to a binding;
- Service – is used to aggregate a set of related ports.

2.3 Dependability assessment

This section gives a review of the dependability terms used in this work. As our work is concerned with testing the reliability of Web Services, a measurement of the overall quality of a service is needed. The assessment of systems utilises some of the same attributes present in Dependability; however Dependability is concerned not only with measuring the Dependability of a system but also with the means to improve the Dependability of a System.

The definition of dependability is given in paper [15], a well known and widely used source which offers a comprehensive explanation of the basic concepts and meanings of *dependability* in computing systems:

Dependability: the ability to deliver service that can justifiably be trusted.

The above definition is universally recognised in the domain of dependability research. It is, however, very abstract and brief. Paper [15] offers an alternative definition of dependability:

Dependability: the ability to avoid service failures that are more frequent and more severe than is acceptable to the user(s).

The above further refines the *dependability* definition. In spite of the fact that it is still quite abstract, it precisely defines the principle for regarding if a system is dependable. The paper also specifies that dependability can be broken down into some specific attributes as follows:

- *Reliability* which characterizes the ability of a system to perform its service correctly when needed.
- *Availability* which means that the system is available to perform this service when required.
- *Safety* is a characteristic that quantifies the ability to avoid catastrophic consequences for the user(s) and the environment.
- *Confidentiality,* meaning the absence of unauthorized disclosure of information.

- *Integrity*, meaning the absence of improper system alterations.
- *Maintainability*, which is ability to undergo modifications and repairs.

Thus, researchers can identify and specify the means of dependability in their specific research domains according to the above taxonomy. For the purposes of this work we are interested in factors that can be used to test dependability by using fault injection [28]. However, we need to explain briefly some of the concepts that can affect the Dependability of a system and go through some ways that the Dependability of a system can be increased.

As detailed in paper [15], there are three main kinds of threats to dependability which may cause a drop in the dependability level: failure, error and fault. As our research is about assessing Web Service systems by injecting faults, it is necessary to explain the difference between these threats to the system.

1. A *failure* occurs when a running system deviates from its specified behaviour. The cause of a failure is called an *error.*
2. *An error* is the part of the system state that is liable to lead to a failure. The error itself is the result of a defect in the system or fault.
3. A *fault* is the cause of an error. In other words, a *fault* is the root cause of a failure. That means an error is the indication of a fault. A fault may not necessarily cause an error, however the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures.

For example, an incorrectly written instruction in a programme may decrement a variable instead of incrementing it. Clearly, if this instruction is executed, it will result in the incorrect value of that variable being written. If other programme statements use this variable afterwards, the system will deviate from its specified behaviour. In this case, the erroneous statement is the fault, the invalid value is the error, and the failure is the behaviour that results from the error. However if this variable is never read after being written, no failure will occur. Or, if the invalid statement is never executed, the fault will not lead to an error. Thus, the mere presence of errors or faults does not necessarily imply system failure.

2.3.1 *Means for Achieving Dependability*

Since faults may cause undesired deviation of the system, how to deal with faults is the main problem in building a reliable system. There are many techniques used to increase dependability. Paper [29] groups these techniques into the following four categories:

- **Fault prevention** can deal with and eliminate a number of faults hidden in the design and implementation of the system. It has to be deployed during the system design stage by employing quality control techniques such as modularization, structured programming, and so on [15].
- **Fault-tolerance** mechanisms act upon errors to maintain the continuity of services. The aim of fault tolerance is to avoid system failures in spite of the presence of faults. It mainly consists of two phases: error detection and system recovery [15]. Error detection is used to identify the presence of errors, while system recovery, by applying error and fault handling, is aimed at transforming a system state that contains one or more errors and (possibly) faults into a state without detected errors or faults that could be activated again. Error handling eliminates errors from the system state, while fault handling prevents faults from being activated again [15].
- **Fault removal** is generally applied in the development phase or during system maintenance. It focuses on discovering potential faults in a system and removing them to avoid failures [15].
- **Fault forecasting** is to predict possible faults by concurrently evaluating system performance. The dependability attributes of a system may change during the life cycle of the system because of system ageing. By employing modelling and testing techniques, dependability attributes can be evaluated, and the probabilistic estimates of dependability measures can help to make changes to the system to avoid system failures. Thus, in fault-tolerant systems, fault forecasting can evaluate the effectiveness of fault tolerance mechanisms and lead to improvements in their implementation. There are many examples, as presented in papers [30] and [31], which report how to use the fault-injection technique to assess the dependability of Web Services, as proposed in our study [16].

2.3.2 Assessing dependability

The dependability of computer systems in general can be examined by either model-based or measurement-based techniques [20]. Model-based techniques can be used at the design stage of the system to obtain potential predictions of its errors and faults. Measurement-based techniques can be used for existing systems and may not require access to source codes or design documentation.

There are two main measurement-based techniques:

- Observation measurement can be applied by observing errors and failures in a large set of operated systems such as [32]. This technique relies on error information obtained by logs from system administrators or by logs from automatic logging mechanisms. By analyzing the data obtained, information on the frequency of faults and information on the type of activity that led to the failure of the system can be obtained. The drawback of this technique is that failures are rare, which means data needs to be collected from the system over a long time span.
- Fault Injection is the deliberate insertion of faults into the target system in order to observe its behaviour in the presence of faults [28]. We will elaborate on this technique in the next section.

2.3.3 Fault injection

Fault injection consists of the deliberate insertion of faults or errors into a system and the observation its consequent behaviour [28]. This method is usually used to assess error recovery and fault tolerance mechanisms. It can also be used to exercise seldom-used control pathways within the system, which might go unused for a long period of time [33].

For a long time, fault injection techniques have been used to assess the dependability of a system by analyzing its behaviour when a fault occurs. Many attempts have been made to develop techniques for systematically injecting faults into a system. Most of the developed techniques fall into two main categories [28]:

- **Hardware-Implemented fault injection:** This is accomplished at the physical level. This type of fault injection technique is called Hardware Implemented Fault Injection (HWIFI) and simulates hardware failures within a system. It uses additional hardware to inject faults into the hardware of the system. Hardware-based fault injection methods fall into two categories:

 • **HWIFI** with contact. This is where the injection has direct physical contact with the system, such as injecting voltage or current changes externally to the target chip.

 • **HWIFI** without contact. This is where the injection has no direct physical contact with the system, such as heavy ion radiation and electromagnetic interference, causing spurious currents inside the target chip.

- **Software-Implemented fault injection:** The objective of this technique consists of reproducing errors at the software level that would have been produced by faults occurring in the hardware and software.

In recent years, software-implemented fault injection techniques have become more popular amongst researchers because they do not require expensive hardware. Furthermore, they can be used to target both applications and operating systems, which are difficult to target by using the hardware-implemented fault injection technique [28].

SWIFI techniques are categorized into two classes, based on when the faults are injected: during compile-time or during run-time.

1. **Compile-Time Injection:**

Compile-Time Injection is a fault injection technique which modifies the source code in order to inject simulated faults into a system. This technique injects errors into the source code of the target programme to emulate the effect of a hardware and software faults. The modified source code alters the execution of the programme instructions, causing injection. The injected fault produces an erroneous state and when the system is executed the fault is activated [28].

This technique requires the source code of the programme to be modified, but it does not require additional software or hardware during run-time. Further, because the faults are hard-coded into the system and require no communication with a fault injector, it has a considerably smaller impact on the execution timing of the system than other techniques. Finally this system is very simple to implement, but it does not allow the injection of faults as the tested programme runs.

However this technique has its own drawbacks. The main drawback is that it requires the modification of the source code of the system under test. That requires the availability of the actual source code of the system, which will not be possible in some cases, such as Common Off-The-Shelf (COTS) systems. In addition the altered source code being executed is not the same source code of the system under test.

2. **Run-Time Injection:**

A Runtime Injection technique is a mechanism to inject faults into a system during run-time. Faults can be injected by using a number of triggering mechanisms [28]. Commonly used triggering mechanisms include:

• **Time-out.** This technique is based on using a timer. When the time is expired, a fault is triggered and injected into the system. Specifically, the time-out event generates an interrupt to cause fault injection. The timer can be based on either hardware or software. This technique does not require any modification to the system under test. Since this method injects faults based on time points, rather than specific events or system state operations, it produces unpredictable effects in a system. It is suitable for emulating transient faults and intermittent hardware faults within a system.

• **Exception/trap**. In these mechanisms a hardware exception and a software trap are used to generate an event control for injecting faults. Using an exception/trap requires injecting a fault based on certain events or conditions occurring. For example, a hardware exception calls the Fault Injection Mechanism when a hardware observed event occurs, such as when a particular memory location is accessed. The trap mechanism is used by inserting a code into

a particular point of a target programme in order to execute the fault injection code. When the hardware exception or trap executes, it causes a transfer of control to an interrupt handler.

This technique has the advantage that it requires no modification to the code and can be triggered based on the occurrence of a specific event.

• **Code insertion**. This technique involves inserting instructions into the target programme that causes a fault injection code to execute just before an event is to occur; it is similar to the code-modification method. This technique differs from compile-time injection in that it injects its faults at run-time rather than at compile time and, rather than corrupt existing code, it adds code to perform the fault injection. As Code insertion performs fault injection during programme run time, it inserts instructions, instead of changing original instructions as code modification does.

It has the disadvantage that it requires the system source code to be modified. However it has the advantage that the fault injector can be used as a library and run as part of the system in user mode on systems that support this process protection model [34].

As detailed above, SWIFI techniques can be useful and economical. We now need to determine which SWIFI techniques are most appropriate for assessing the dependability of Web Services. A number of criteria have been identified as necessary to fulfil our stated goals.

One of the main features of Web Services is that they run in a heterogeneous environment, in terms of machine architecture and programming language. The fact that Web Services are run in a virtual machine can, however, prevent testers from injecting faults directly onto the hardware of the hosting machine.

One of our main stated aims of this research is that the developed fault injection method should be non-invasive. Consequently, any SWIFI technique selected should not make any modifications to the source code and the environment.

Therefore, when comparing Run-time and Compile-time injection methods, it is obvious the former would be the more appropriate technique. Run time injection methods do not modify the source code of the system. For example, techniques such as Code-insertion provide a more appropriate fault injection method than Compile-time injection, which corrupts existing algorithms. Code-insertion can be used to manipulate inputs and outputs with a high degree of control since it inserts extra code; for instance, method parameters can be corrupted when a method call is made.

Runtime injection techniques can use various techniques for injecting faults into a system via the triggering mechanisms described above. These fault injection techniques can be used to assess the impact of different fault classes on a system [20]. Some of these techniques are listed below, although the list is not exhaustive:

- Corruption of the memory space: this technique injects faults affecting the memory subsystem, such as corrupting RAM, processor registers, and I/O map.
- Syscall interposition techniques: this technique is concerned with fault propagation to the middleware from operating system kernel interfaces. Faults can be injected by intercepting operating system calls to the middleware and injecting faults into them, such as returning an error code from a system call and signalling an exception.
- Network Level fault injection: this technique is for injecting faults by corrupting, losing or reordering network packets at the network interface. Faults can be injected by instrumenting the operating system protocol stack as in [35]; however this runs the risk of being detected and rejected by the other end of the receiving systems protocol stack. It is therefore preferable to inject the fault at the application level [19].

These Run-time fault insertion techniques have to be compared in terms of their applicability for testing Web Service systems, which are the target of our research. The heterogeneous nature of Web Service environments contributes a major concern about whether it is appropriate to use any technique that depends on a specific hardware environment. Moreover, the technique capability can also be scaled by the number of faults that can be injected into the system. The technique must be able to inject faults into the system reliably and precisely in order to assess the system with different faults and circumstances.

Network level fault injection can be an appropriate fault insertion technique to use for testing Web Service systems. Web Service systems usually use networks to communicate by transferring messages; therefore injecting faults into these messages would satisfy the two criteria, namely the number of faults injected and the requirement not to tightly tie the fault injection to a specific hardware environment. However, the other two techniques (Corruption of the memory space and Syscall interposition techniques) are tightly coupled with specific hardware platforms.

In the case of Web Services the insertion techniques can be used to insert faults at both interface level and communication level. Interface faults affect operations input or output parameters and other SOAP message fields by corrupting data or assigning invalid parameter values. On the other hand, communication faults affect the messages transferred across the network by delaying the forwarding of messages or simulating a connection loss, for example.

2.3.4 The communication and interface faults impact on Web service systems

These two types of faults (communication and interface faults) are very important for testing Web Service systems. Web service middleware, similarly to other distributed technologies, relies on existing underlying middleware, such as network protocols, to implement the essential low-level services [4]. Thus Web Services inherit many of the dependability issues the conventional middleware infrastructure suffers from. For example, the interaction between the client and Web services relies on the Web or other networks, which are inherently unreliable media that may cause communication problems such as a loss, delay or damage of the message [3][13][14].

It is critical to guarantee the reliability of service-oriented applications. This is because they may employ remote Web Services as components, which may easily become unavailable in the unpredictable Internet environment [36], as a consequences, communication faults can be occurred. In [36] paper presents a practical distributed mechanism named WS-DREAM for assessing reliability of Web services in real-world applications. The benchmarking Web

Services include six identical Amazon book displaying and selling Web Services distributed in six countries. Users who request these Web services are distributed in five locations all over the world by executing 1,000,000 test cases. The results indicate that unstable Internet environments and server connections leads to unreliability of Web services [36]. In [37] presents a practical experience in benchmarking a number of existing Web Services, and investigating the instability of their performance and the delays induced by the communication medium. The experiment is conducted with two Web Services, which provide similar functionalities. One deployed by the European Bioinformatics Institute, Cambridge, UK, the other Web Service hosted by the DNA Databank, Japan. In the experiment, three consumers (clients) of those Web services are deployed from three locations so as to simultaneously send requests in order to observe the differences in their behaviour and how the locations (networks) affect the dependability of the two Web Service. Two of these clients were located in Newcastle upon Tyne, UK, whilst the other one was deployed in the China Education and Research Network, china. The results indicate that network instability significantly depends on the quality of service of a local Internet Service Provider. Besides, occasional transient and long-term Internet congestions, packet losses and network route changes that are difficult-to-predict also affect on stability of operation of Web Service systems.

In [14] presents the results of a 40 day reliability study on a set of 97 popular services done from the user's perspective. The results show that the majority of Connection Timeout failures caused by network problems, whereas the Connection Refusals failures caused by hosted Servers. That is, Network related failures marginally outnumber host related failures, In addition, results show that Network related failures last longer than host related failures. Moreover, as Web services are deployed on application servers, which may become unreliable or out-of service, due to system maintenance or other internal activities [15]; the design or implementation of the Web services business procedure may contain faults and result in their erratic behaviour [15].

Interface faults (related to problems in the interaction among software components [38]), are particularly relevant in an environment based in Web services. Web services must provide a robust interface to the client applications in spite of any presence of invalid inputs, which

may be occurred due to program defects or bugs in the client applications, corruptions caused by network faults, or even security attacks. There are many testing tools which assessing the Web services based on injecting interface faults. Injecting Interface faults has become an attractive method to assess the behaviour of a system in presence of erroneous inputs [39]. Such testing methods stimulate the system in a way that triggers internal errors, therefore systems can be assessed according to the number and type of errors uncovered. A number of such testing tools are detailed in the next section.

Based on all the discussion above, therefore, the success of using insertion techniques of fault injection for combining between interface and communication faults could generate more complex errors and permit deeper injection tests. We will therefore provide further elaboration of these in the next chapter.

2.3.5 Existing Fault Injection Tools

There are many Software Fault Injection tools for testing distributed systems in general and other tools for testing Web Service systems in particular. Fault injection tools for testing Web Service systems can be classified into two main categories, as outlined in this sub-section.

There are network level fault injecting tools which were not originally developed for testing Web Services but which might be very useful for injecting communication faults. The most commonly used tools, such as DOCTOR [40], and Orchestra [41], both support network level fault injection and could potentially be used to inject faults into Web Services.

DOCTOR [40] (integrateD sOftware fault injeCTiOn enviRonment) is a tool for injecting faults into CPU, memory, and network communication faults through time-outs, traps, and code modification.

Orchestra [41] is a fault injection tool which is based around Network Level Fault Injection. The fault injector layer is inserted between the protocol layer and the lower layers. It allows messages exchanged between the participant protocols to be manipulated. Messages can be

delayed, lost, reordered, duplicated, and modified. The message modifications are specified using a user-defined script. Orchestra was initially developed for the Mach Operating System; therefore the user has to implement his faults injection layer for other Operating Systems.

Ferrari [42] (Fault and Error Automatic Real-Time Injection) uses software traps to inject errors into the system. Software traps are triggered by either calls to the desired memory locations or a timer. When the traps are triggered, the trap handler injects faults into the system by changing the content of a register or memory locations to be corrupted. Experiments showed that error detection is dependent on the fault type.

All of these methods are invasive, in the sense that they operate on the same system as the hardware they test. Due to the heterogeneous environment operated in by Web Services, a fault injection testing tool must not rely on a particular hardware platform. Any tool used has to be able to run equally well under any hosting environment that Web Services operate in.

Moreover, using some of these tools for injecting communication faults is not enough to test Web Services. As some of these tools have been designed to inject faults at network protocol level, they can not decode complete middleware message sequences. In this level only packets can be captured and modified, and these usually do not correspond to application messages. Application level messages may span more than one network packet, therefore targeting a particular part inside a message (function call parameter) is very hard in this level. More addition, Injecting faults at the network level is very likely to produce side effects detectable by applications. These effects may not be what the user intended and wanted to inject. When a packet is dropped it may not lead to application level messages being dropped, but could lead to delays due to retransmissions and error correction. Therefore, other researchers have focused on providing other fault injection tools which have the capability to decode complete middleware SOAP messages, in order to inject significant interface faults.

There are many other fault injection tools for testing Web Service systems. In [43] a tool is developed for generating and validating test cases. Tools start from the WSDL schema types

and introduce an operator to generate a request with random data and a test script that manipulates the request parameters.

In [44] a technique for testing Web Services using mutation analysis is proposed. A mutant WSDL document is generated by applying mutant operators to the original WSDL document. A test tool called WSDLTest tool [45] generates Web Service requests from the WSDL schemas and tunes them in accordance with pre-conditions written by the tester and verifies the response against the post-conditions offline. In [46] a testing tool is proposed based on some rules defined in XML schema or DTD. The tool modifies the value of the parameters in requests by using boundary value testing, and of interaction perturbation, using mutation analysis.

The work described in [47] helps service requesters create test cases so as to select suitable and correct Web Services from public registries. It proposes a method where faults are injected into SOAP messages to test the boundaries of the parameters, as specified in the WSDL document.

This kind of tool suffers from two main disadvantages. First, its main focus is on injecting faults into SOAP messages by corrupting data and procedure parameters inside them. As they do not inject communication faults, problems such as message delaying, for example, cannot be assessed.

The second problem is about the type of tests that can be performed. As such tools perform by simulating Service requesters in order to test Service providers as shown in Figure 2.2, the fault injector needs to invoke the Service provider to be tested. In composed Web Services,

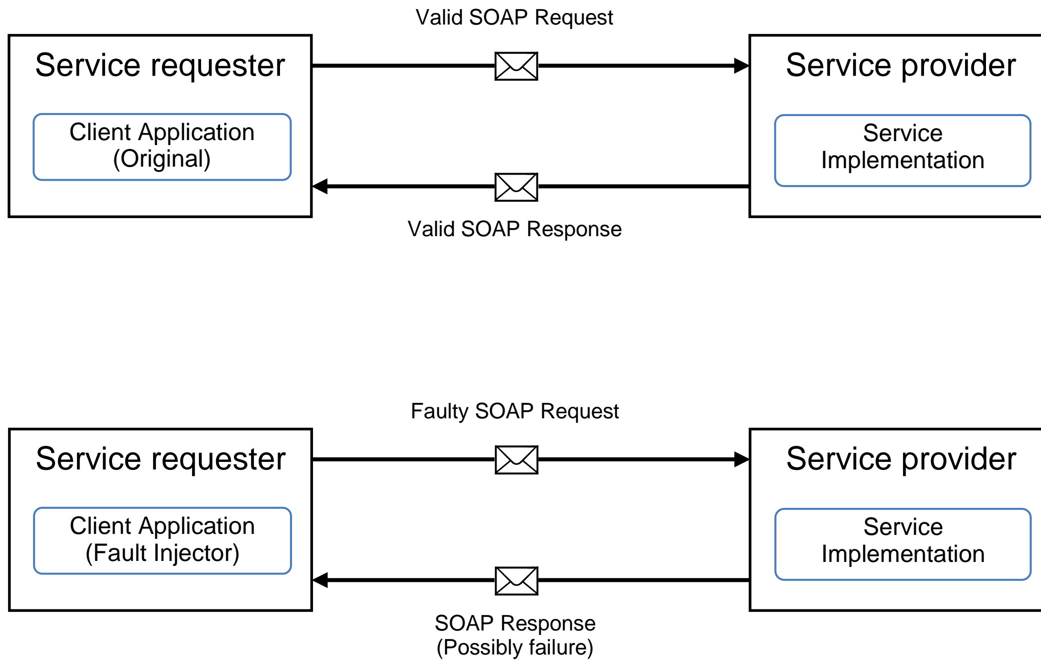


FIGURE 2-2 CLIENT-SIDE FAULT INJECTION ARCHITECTURE

where the Service provider needs to be a Service requester to another Service provider in order to serve a request, it is essential to test the Service requester and its reliability, in order to prevent the whole system from failing to provide the required service. Moreover, it is also impossible to use such tools to test communications between service partners in composed Web Services because testing communications between a service and its original client application will be substituted by the injector itself.

To overcome the above disadvantages, there is a need for a tool to address both communication and interface faults. A fault injector tool should be able to intercept communication messages exchanged between Service partners or between a Service provider and its Service requesters, as shown in the schema presented in Figure 2-3.

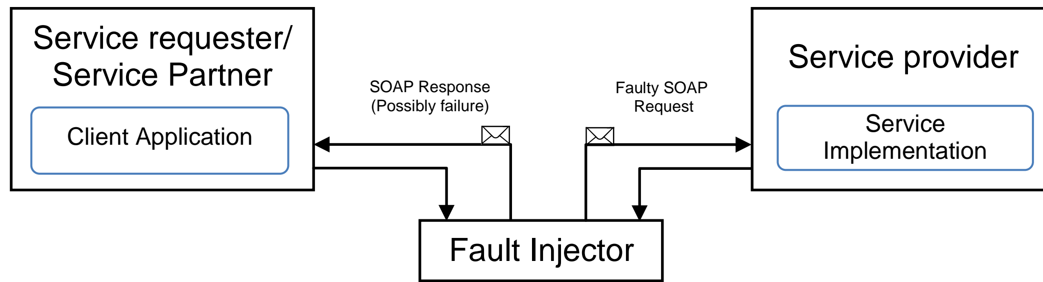


FIGURE 2-3 SUGGESTED FAULT INJECTION ARCHITECTURE

Another tool [19] introduces a framework intercepting and perturbing SOAP messages by injecting faults to corrupt the encoding schema addressed and inserting random text in the SOAP Body. To the best of our knowledge, WS-FIT [31] is currently the best Web Service fault injector that can fit this architecture. WS-FIT can inject both interface and communication faults by using scripts. The SOAP messages are intercepted between the Service requester and Service provider. The function parameters in the SOAP message are modified by using the value boundaries specified by the tester. Also the intercepted messages can be delayed and dropped between the two sides (Service requester and Service provider). However, WS-FIT needs to implement a set of *hooks* at the SOAP protocol layers of every machine hosting any Service requester or Service provider being tested. This introduces a strong intrusiveness and could disrupt the communication. Moreover, WS-FIT can only be used in a completely controlled testing environment so as to modify the SOAP protocol layers. In addition, it does not emulate additional workload in the system, which could give rise to different results. Different workloads could lead to different testing results, due to their creation of different system activation patterns [20].

2.4 Conclusions

In this chapter the concepts and the problems of Web Service technology for implementing SOA are reviewed. The loosely-coupled distributed architecture of Web Services has brought many benefits for distributed applications such as e-science and e-commerce systems. However, such architecture is inherently undependable. Research on the dependability of Web Service applications has to deal with both Service failures and network failures. It is important that solutions to these failures should be properly assessed and examined, especially when the system is constructed of a combination of services from different service

providers. Using this combination of services in a composed service which may not have been assessed and examined implies that the level of trust will be reduced, since interaction faults may be raised that have not been detected.

Although many approaches have been developed to assessing the fault tolerance mechanisms and the dependability of Web Services and service composition, our analysis shows that the limitations of those solutions restrict their applicability and efficacy. These limitations are varied: for example some tools are designed for general-purpose dependability assessment of network protocol stacks and not for assessing middleware products. Others concentrate only on limited classes of faults. Also further assessment tools are needed to test the underlying environment such as middleware layer or the network stack layer but the application of such tools could introduce a strong intrusiveness and disrupt the communication.

3 Chapter 3 - Fault Injection Method for testing Web Services

Using traditional testing techniques raises many problems with distributed systems. Performance and fault tolerance assessment provides a useful method for obtaining a level of reliability in a system. Although much work has been done in testing Web Services, there still appears to be a requirement for more research on Web Services in general and composed Web Services in particular.

It is particularly hard to test web service applications which are developed to run over the Internet. Testing Web Service applications which involve running software in different environments and platforms is not an easy task without modifying the sources or the networking libraries of the hosting operating systems. Testing such systems requires a run-time environment, which means using the Internet to test the performance of each component of the system and also to assess the impact of the fault tolerance mechanisms applied to the system. Such approaches are not always attractive or achievable. The cost of setting up a WAN or using the Internet for the sake of testing is very high. Moreover, controlling the dynamics of the network such as inputting more stress, load, or errors is almost impossible.

Some fault injection testing methods have been conducted using fault injection to test the performance and fault tolerance mechanisms of CORBA applications, with fault injection taking place at a network level with successful results [48]. Therefore our research used this as a starting point for our new fault injection testing method.

3.1 Problems Involved in Testing Web Service Systems

There are many problems for testing distributed software in the development cycle, and Web Service systems are no exception. Modifying the system for the sake of testing is not desirable, since the modified system under test is not the same as the system to be deployed. Furthermore, testing the performance and fault tolerance of a system using a real WAN

environment is affected by difficulties in setting up the environment itself and controlling it for evaluation purposes. Previous work has not resolved all problems; the tested systems lack the realization of a Wide Area Network that the system runs over. Most of these are low level tools, sometimes requiring special hardware or modifications to the middleware or networking stack.

3.1.1 Problem of modifying the system under test

To test Web Service systems properly requires putting them into certain states in order to ensure that particular paths are executed. The tester needs to have the ability to execute certain chains of events in order to produce hard-to-reach cases in the distributed computation. This can usually be done by injecting faults or hard-coding them within the system. Moreover, testing a web service system and its fault-tolerance capabilities requires that the particular chains of events (which would lead to hard-to-reach cases) are applied to a single participating service in the system, or to a small set of such services. This may require the modification of these services. However, it is not always possible or advisable to modify the code of the system under test. The source code may simply not be available in the case of published services or COT services. Furthermore, it is not reasonable to modify systems with hard-coded faults for the purpose of testing, for the very fact that the hard-coded faults in the system will mean that it has become different from the system to be tested.

3.1.2 Problems of Setting up a distributed testing environment

In Web Service systems, service location is typically dynamic, so that services are discovered, selected, and composed, possibly at run-time. With such services, it is hard to assess behaviour and performance in the presence of faults. As Web Services are usually distributed and run over the Internet, there is no guarantee that all parts of the service will have high reliability. In [20] it is reported that communication faults such as message loss, duplication reordering, or corruption have unpleasing effect on traditional distributed systems and their reliability such as CORBA applications. Moreover it has been found that unstable Internet environments and server connections can lead to the unreliability of Web Service applications [36].

Since Web Services usually run over the Internet, the performance and fault tolerance of a service are very difficult to measure at design time. Testing such systems requires a distributed testing environment, such as a Wide Area Network (WAN) or the Internet. Therefore testing the performance and fault tolerance of the system could be achieved by deploying the system and running it over a WAN or the Internet.

However using the Internet or WAN for the sake of testing is usually impracticable. It is costly in terms of time consumed and setting up a WAN or using the Internet for the sake of testing. Furthermore, it is almost impossible to control dynamic environments such as networks such as by increasing stress, load, or errors. Moreover, errors and faults may take a long time to occur. Some errors may not occur without applying a certain specific chain of events.

A more practical approach is to run the system in one machine or over a LAN using a WAN emulation system, which can provide the sense that the system is running over a WAN and attempts to provide all the properties of a dynamic WAN such as the Internet [16]. This allows the system testers to examine performance and fault tolerance by running the system under varying scenarios, such as different network traffic load, delays, loss rate, and so on. By using network emulation, not only can the performance of the whole system be measured under different circumstances, but the contribution of each service to the overall composed service system can be measured and a services bottleneck can be discovered. That is, firstly running the whole system components (services) on a different network emulation scenarios, such as identical network traffic load, bandwidth, packet drop and error rate. Secondly, running the same system on different emulated network which emulates a various network links that have different traffic load, bandwidth, packet drop and error rate, and so on. That will simulate different performance for each component (service) running on different emulated network link so as to measure the performance of each component in such runtime environment and can be compared with first scenario. Such a run-time environment should also be able to inject faults into the system under test. The aim of our research is to provide such an environment.

3.1.3 Problem of testing composed services for performance and fault tolerance

Web Services can be adopted to develop information systems through integration of services to obtain complex composed services. Web Service technology is being used to allow the creation of complex systems, composed of simple Web Services, which exchange messages to form complex conversation schemas [49]. These services are usually developed and administrated by different service providers, running on different platforms and also distributed over the Internet in different locations. In reality, there is no guarantee that all services will be highly reliable.

Execution time measurements of composed service systems are very hard to estimate at design time. When analysing the time of a composed service, it needs to be taken into account that the execution time of completing a composed service increases when slow services are present. The contribution of each component service within a composed service to the time required depends on the processing time of the process, and also to the network properties such as delay time. That is, the position of a slow processing service and the network quality influences the overall execution time. For example, if the execution flow of a composed service forks into three parallel branches with varying execution times, the most time consuming branch (say branch1) determines the whole execution time of system, since the other two branches terminate earlier, but the composed service system still has to wait for branch1 to complete. Similarly, if a service is invoked through the network, because the invocation takes a long time due to some network-related problem, its long response time affects the composed service execution time more than any other contributing service which has a good network connection.

In composed services, time-out mechanisms are often present and these too may cause some problems. If a response is not received within a specified time interval, an exception will be thrown by the sender which concludes that either the request or response message was lost somewhere in the network. The problem of this approach is how to choose a reasonable time-out interval. If the time-out interval is made too short, then there is a risk of duplicating messages and also, in some cases, reordering messages. If the interval is made too long, then

the performance of the system is badly affected. We will elaborate on this issue in section 3.3.5.

3.1.4 Limitation of previous work

As discussed in the previous chapter, although some of the testing tools developed earlier are stable and accurate, they have their limitations as follows:

I. Tools for injecting communication faults: testing tools such as [40] and [41], for injecting faults, are useful for injecting communication faults into Web Service systems, However they are invasive, in the sense that they operate on the same system as the hardware they test. Due to the heterogeneous environment where Web Services operate, the fault injection testing tool must not rely on a particular hardware platform. Any tool used has to be able to run equally well under any hosting environment that Web Services operate in. Because these tools have been designed to inject faults at network protocol level, they cannot decode complete middleware message sequences.

II. Tools for injecting interface faults: other researchers focused on providing other fault injection tools which have the capability to decode complete middleware SOAP messages in order to inject significant interface faults. In [43], [44], [45], [46] , and in [47], this kind of tool is seen to suffer from a limited class of faults that can be injected. Their focus is on injecting faults into SOAP messages by corrupting data and procedure parameters inside SOAP messages. Whereas they do not inject communication faults so that message delaying, for example, cannot be performed.

III. Simulating Service requesters as Fault injectors: most Web Service testing tools work by simulating Service requesters in order to test Service providers as shown in Figure 2.2. The fault injector needs to consume the Service provider to be tested. In composed Web Services where the Service provider needs to be a Service requester to another Service provider in order to serve a request, it is essential to test the Service requester (Service partner) and its reliability, in order to prevent the whole system from failing to provide the required service. Moreover, it is also impossible to use such tools to test communications

between Service partners in composed Web Services, because testing communications between a service and its original client application will be substituted by the injector itself.

IV. Tools for injecting both faults (communication and interface faults): although this is the only tool [19] we are aware of that can inject both classes of faults (communication and interface faults), it introduces a strong intrusiveness. WS-FIT [19] needs to implement a set of *hooks* at the SOAP protocol layers of every machine hosting any Service requester or Service provider being tested.

V. Emulating additional workload: to the best of our knowledge none of the Web Service testing tools emulate additional workload in the system being tested. Emulating additional workload during the testing of a system can give rise to different results. Different workloads could lead to different testing results, due to cause different system activation patterns [20].

In this work we develop a method for testing Web Service applications in order to overcome the above problems. The testing method is appropriate for testing the performance and fault tolerance of either a single Web Service or a composed service, without any modification to the system being tested. No recompiling or patching is necessary. In addition, two classes of faults may be injected, communication faults and interface faults. Furthermore, the tool generates background workload to more accurately emulate real networks in order to produce more realistic results. Finally the tool is independent of the hosting environment for portability.

3.2 Overview of the Proposed Approach

To address the need for testing the performance and the impact of fault tolerance mechanisms applied to Web Service applications, a new method has been introduced. The proposed testing method will try to overcome the problems mentioned above. The main goal of the proposed method is to set a distributed run-time environment using a WAN emulator which gives the sense that the system is running over the Internet, and injecting two classes

of faults, communication faults and interface faults, without any modification whatsoever to the system under test.

Our testing tool is called the Network Fault Injector Service (NetFIS) approach. The NetFIS method is basically intended for testing composed services for performance and fault tolerance mechanisms. For the purpose of the NetFIS method we will define a composed service 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 may further assume that a heterogeneous middleware layer is being used by each service, and also that services may run on heterogeneous platforms for portability reasons.

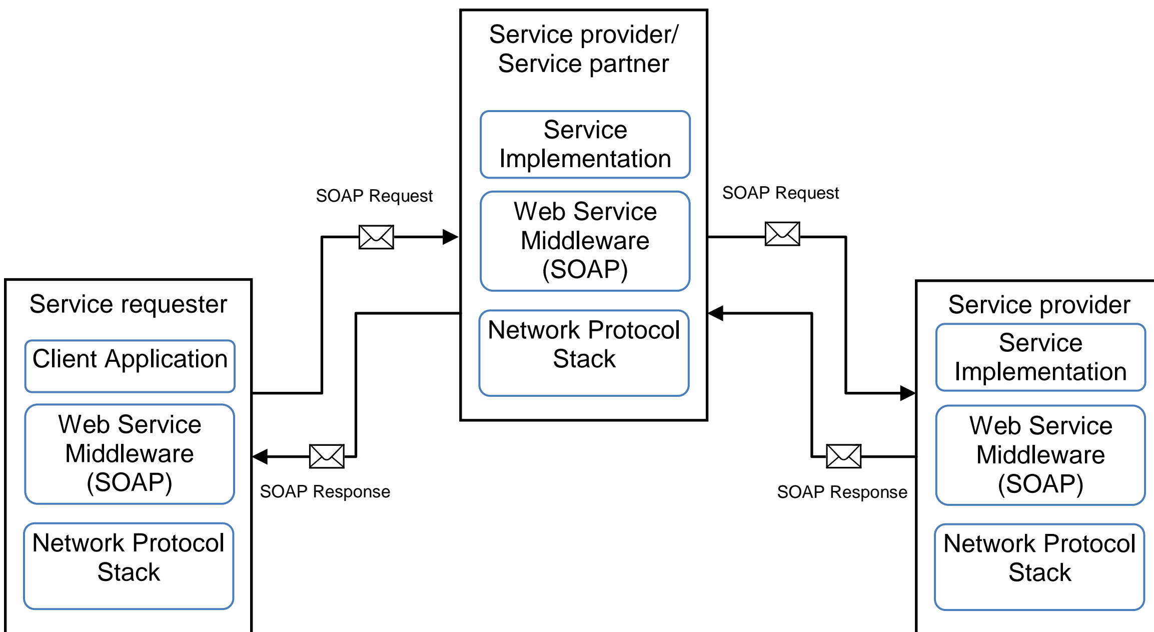


FIGURE 3-1 A SIMPLE COMPOSED SERVICE SYSTEM

This thesis details the NetFIS performance and fault tolerance testing method and shows how it can be applied to composed service-based systems. The only requirement for the application of the method is that the target system be distributed in a modular fashion of services interacting via SOAP formatted messages, so that messages can be intercepted and

manipulated to emulate incorrect behaviour of faulty services. NetFIS is an intermediary composed service, which implements our fault injection method in order to test the fault tolerance mechanisms applied to Web Service applications.

The major contribution of this work is the novel fault injection method that uses a Wide Area Network Emulator to inject network faults at application level and uses Mediators to inject software-specific faults without the need for any modifications to the system being tested. This novel injection method is achieved by intercepting messages going through proxies, giving the composed service systems the sense of running over a Wide Area Network and injecting appropriate faults without any modification to the system under test. It requires no modifications to the underlying operating system, networking libraries or the web service applications under test. By intercepting messages at this level it is possible to perform parameter perturbation. This is detailed in later sections.

3.3 Design Approach

The proposed fault injection method is not only for testing the Service provider but also for testing a Service requester. In composed service systems the Service provider could also be a Service requester to other service provider/s. In some composed systems the requested service in order to serve a request needs to call other services and afterwards sends the response to its requester. In such systems it is very important to test both sides of the system (Service requester and Service provider), because if either side failed to be sufficiently reliable it could bring down the whole system.

This method is for injecting communication faults and software faults at application level. The communication faults include dropping, delaying and randomly corrupting bytes of the exchanged messages. In addition, software faults are also injected into individual Remote Procedure Call (RPC) parameters, based on obtaining the relevant Web Services parameter definitions (including data types) from the Web Service interface. As the method can inject many different kinds of faults, it offers the user the flexibility to inject appropriate faults as

required and also gives the applications a configurable emulated network, creating in the system the sense it is running over a network without any modification [16].

Before describing further details of the proposed system architecture, it is worthwhile to go over some of the major design choices as follows:

I. The choice of injecting faults at the application or the network level is discussed along with why it differs from other testing methods and how the method will fit into the system under test.
II. What operational profile or simulated system activity should be applied during the test and whether to simulate or emulate a network and why LANs alone are not sufficient testing environments is discussed.
III. Which classes of faults are to be injected, why are they chosen, and where will they be inserted? The injection of a fault may be triggered by the occurrence of an event, occur after a predetermined time period, or randomly.
IV. How the tested system behaviour is monitored and the failure modes are classified. It is important that all significant events be observed, which may not be easy in a distributed system.

The above issues have been addressed precisely to target the exchanged messages between services forming a Web Service system distributed over a WAN.

3.3.1 Fault injection location mechanism

As detailed in previous chapter, there are many Fault Injection Mechanism techniques available. We have decided to base our fault injection method on network level fault injection techniques for the following reasons:

I. Since the test environment involves data transferred across a network interface, it is a similar architecture to the design of Web Service systems.

II. It can be simple to implement under the heterogeneous middleware used by services forming composed service systems, because it can be implemented using proxies, which can intercept messages exchanged between services.
III. Exchanged messages can be intercepted and tampered with to simulate a large number of fault classes, for instance network faults, API parameter faults, and so on.

Our fault injection method uses a modified network level fault injection technique to inject faults into Web Service systems. The standard network fault injection technique is usually administered at network level by performing network related faults such as corrupting, reordering, and dropping on network packets. Such a fault injection method is traditionally used for testing the reliability and performance of networking protocol stacks. However, our testing method is for testing the impact of communication faults and interface faults on middleware and at the application level. In analyzing the network level fault injection technique used for injecting network faults at the network protocol stack level, the following problems have been faced:

- At the network level, only packets can be captured and modified, and these usually do not correspond to application messages. Application level messages may span more than one network packet, therefore targeting a particular part inside a message (function call parameter) is very hard.
- Tampering with packets at network level involves high risk of being detected by the actual underlying network protocol stack. Modifying or dropping packets at the network level could be detected by the receiver's error detection mechanisms, such as checksums. In consequence a correction mechanism could be automatically applied to the packets such as retransmission. As a result, the application and middleware intended to be tested will most likely not experience the occurrence of such error; therefore their dependability will not be tested and examined in such circumstances.
- When the messages are intercepted after they are signed or encrypted, it makes it impossible for them be tampered with in order to inject faults into

them without them being detected as being tampered with and then rejected by the destination.

- Injecting faults at the network level is very likely to produce side effects detectable by applications. These effects may not be what the user intended and wanted to inject. When a packet is dropped it may not lead to application level messages being dropped, but could lead to delays due to retransmissions and error correction.

With all the potential drawbacks given above, a better choice may be to move the fault injection location away from the network interface and position it in the middleware layer [20]. Messages at middleware level are complete entities, so they can be intercepted as a complete message and faults can be injected into them, rather than into just part of a network packet, which may be discarded before it reaches the middleware layer at the other end. However intercepting messages in the middleware layer requires some customization to that layer, which will decrease the generality of our fault injection method. Our fault injection method aims to provide a general purpose testing methodology that facilitates testing all kinds of web service middleware systems. There are many web service middleware platforms such as AXIS, JBoss, Glassfish, and so on. Therefore customizing any particular middleware platform for the sake of intercepting messages prevents the tester from using such a method to test systems using different web service middleware.

Based on the discussion above, while the fault injection method can be based on injecting the mentioned network fault injection technique to inject network faults and application faults, it should move the fault injection location away from the network interface and middleware level and position it at the application level.

3.3.2 A General proposed architecture of fault injection location

Some major requirements of the fault injection system need to be further developed. The faults injected should attach themselves to the communication subsystem or medium without alerting or modifying the applications being tested. Nor should they alter or modify the

underlying networking libraries or host operating systems. Additionally, the fault injection should be transparent to the application. That is, it should not alter the application's behaviour, nor what it expects from the underlying network, nor break the semantics of Wide Area Networks. For example, if there is any error in the fault injection application, it should not cause the application being tested to break.

Based on the above requirements it would be a good design choice if the fault injection system were developed as a stand-alone Web Service mediator, as shown Figure 3-2. Each web service Mediator can capture messages between any Client requester and Service provider and then faults can be injected and also WAN emulation can be provided to the system under test. These Web Services are stand-alone services which can work on their own and even run on a different application space which can be a different physical machine, OS, or web service middleware.

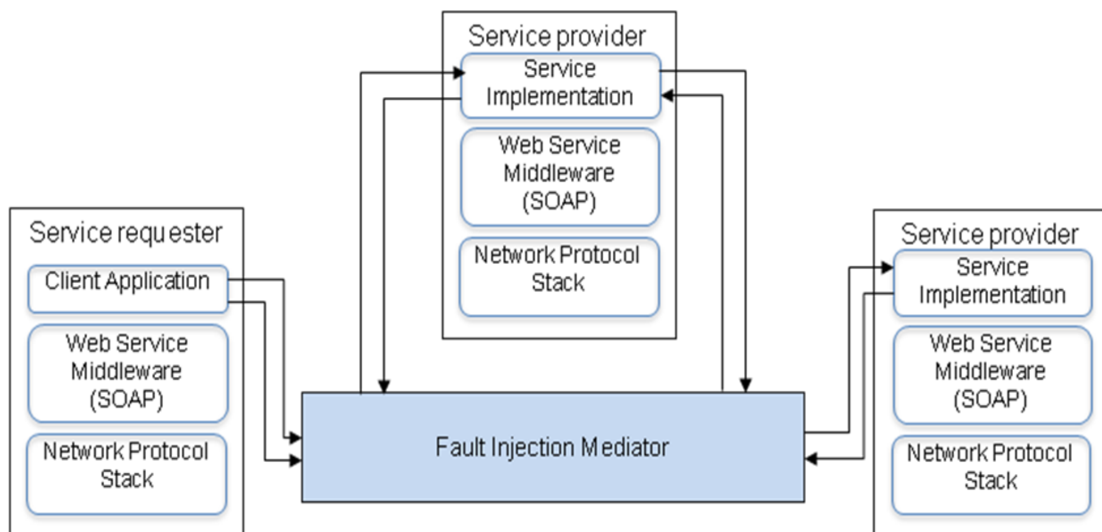


FIGURE 3-2 A GENERAL VIEW OF THE INITIAL PROPOSED APPROACH

By using this way of intercepting messages, not only is the generality is enhanced, but the non-intrusiveness is improved, in the sense that it does not require any modification to the middleware or to the hosting operating system and of course to the Web Service system under test. However, there is another side to intrusiveness which is the overhead of intercepting the messages and processing them. It is true that this overhead could be neglected if it is kept to a minimum. This is due to the fact that the actual systems will

operate over a Wide Area Network or the Internet. So, the delays introduced by the Fault Injection tool and the processing could be neglected if they fall within the delays usually incurred by transmission delays in such networks [17]. When designing this fault injection method, therefore, care must be taken to minimize this overhead and keep it to a minimum.

However, this way of intercepting system communications could introduce other problems, regarding how to make the system under test forward the exchanged message to the fault injection service. There are many web service technologies that could be used for this task, such as WS-Addressing [50], WS-Routing, and of course Web Service proxies. We will elaborate on these options in the next chapter (the implementation chapter).

3.3.3 Run-time Environment

As discussed earlier, the proposed method is for testing the impact of fault tolerance and the performance of Web Service systems. Therefore there is a need to test the system in a run-time environment such as the Internet or WAN. However (as discussed in Section 3.1.2) using the Internet or WAN for the sake of testing is usually impracticable. It is time consuming and costly in terms of setting up a WAN. In addition, using the Internet for the sake of testing is impossible in terms of controlling the variety of network conditions like latency, data loss/error/reordering and bandwidth and also putting more stress or load on the target network. Moreover, errors and faults may take a long time to occur. Some errors may not occur without applying a certain chain of events.

One of the alternatives is to test Web Service systems over local area networks (LANs). On the plus side, LANs are very cheap and very fast compared to WANs. LANs are also much more controllable. They are also very reliable. These characteristics are very useful for testing. However, these very characteristics are also drawbacks: LANs are not WANs (which are the target environment for Web Service systems). The performance of LANs is different from that of WANs and they have different faults.

Therefore, there is a need to emulate or simulate a virtual Wide Area Network environment and its behaviour, which would enable the tester to control the virtual network environment

and its parameters to test the system under different circumstances, such as different dropping rate, error rate, delay, workload traffic, and so on.

3.3.3.1 Wide Area Network Emulation

One of the more realistic approaches is to run the system in one machine or over a LAN using a WAN simulator or emulator, which can provide the sense that the system is running over a WAN. Such a method would enable the tester to control the virtual network environment and its parameters in order to test the system under different circumstances such as different dropping rate, error rate, delay and so on. However there are differences between network simulation [51] and network emulation [52]. While both create virtual networks or execution environments, emulators differ from simulators in that they do not create virtual applications. Instead, they work with real applications, which can be attached to the emulator and will behave as if they are attached to a real network.

Simulation requires the programmes to be recompiled or even modified for the simulator. This means that the programmes do not operate exactly as they are supposed to. Instead, they are built especially to work for the simulator. Simulation can be used in many situations, because of its ability to give a more accurate and controlled run-time environment. It also helps to analyse systems or models without the need to build (or implement) them.

However, emulation involves a more seamless method of running programmes in a virtual run-time environment. It is often the case that programmes are already developed and cannot be modified for the sake of a simulator. Emulation also gives more accurate results, since the programmes are not modified and are executed as they are running in the actual target run-time environment.

Based on the discussion above, and as part of the suggested fault injection method in this study, we propose emulating customizable and controllable WANs over LANs, as shown in Figure 3-3. This way, the web service systems are tested on virtual WANs that are very similar and comparable to the actual target WAN environments. Testing over these virtual WANs will avoid encountering the problems of real WANs, as discussed above. The

assumption made in this thesis is that the actual LANs used in hosting the virtual WANs are very reliable and fast, and thus uncontrolled faults and delays will be negligible.

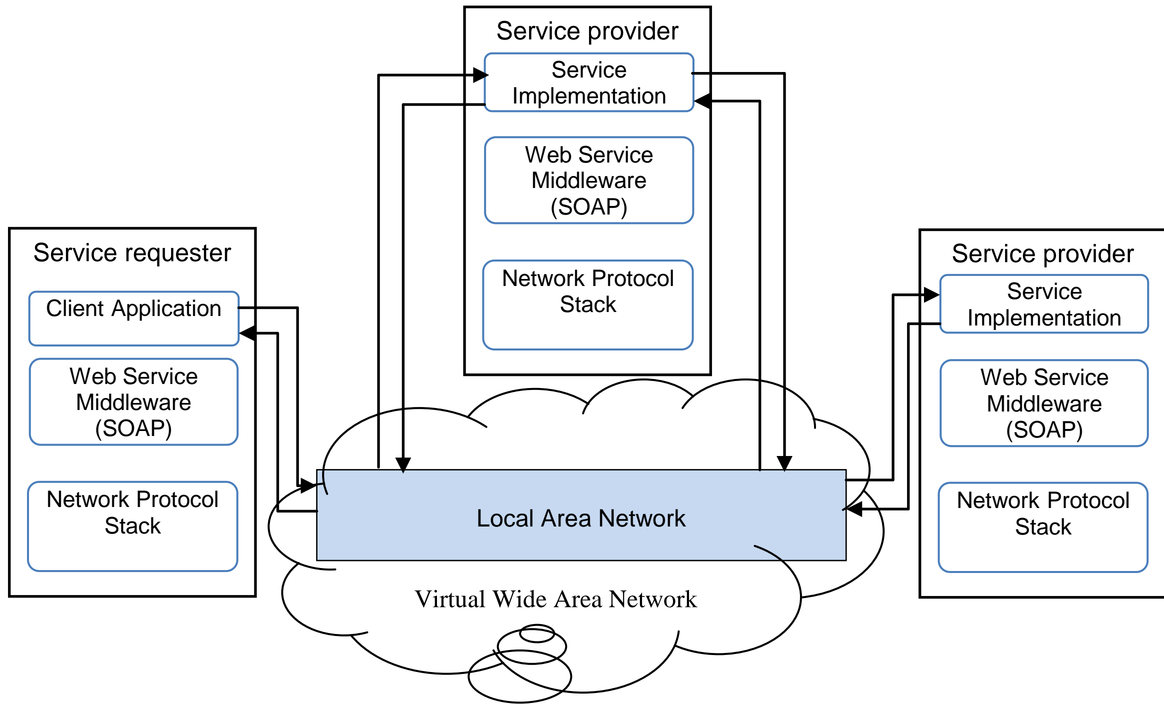


FIGURE 3-3 HOW A WAN EMULATOR RUNS OVER A LAN

However in order to provide a virtual Wide Area Network environment to run Web Service systems, the existence of other network traffic occupying the emulated network is also significant. Therefore it is better to emulate the Wide Area Network behavior over LANs and at the same time simulate the presence of other sources of network traffic as discussed in the next section.

3.3.3.2 Traffic simulation

A major feature of Wide Area Networks is the existence of other network traffic occupying the network. Apart from the increased delays, errors and routing, what makes a WAN different from a LAN is that applications performance can frequently be affected due to other

traffic occupying the network resources. What took a second to transfer minutes ago could now take 10 seconds, due to a surge in traffic.

To provide a virtual Wide Area Network environment on which to run Web Service systems, it is better to emulate the Wide Area Network behaviour and at the same time simulate the presence of other sources of traffic. Testing systems against network faults requires additional network traffic to be generated in order to test the reliability and the performance of the system under different traffic stress scenarios. That will help the testers to examine systems performance under different traffic loads that consume networking resources together with the distributed application being tested [53].

Although simulating Synthesizing network traffic is always the best approach to testing systems for reliability and performance, it is difficult to generate network packets with different aspects of network properties. Yet the results reported in [54] demonstrate that actual network traffic (including Ethernet LANs, WANs) is much more varied and assumes various forms and volumes. Therefore, generating effective testing traffics using the methods available for the network software system is highly important.

Nowadays, most software tools have the ability to measure a system’s performance through network testing. However, a large amount of studies shows that these testing systems only adopt traditional testing methods driven by specific test data or test scripts [53]. However simulating network traffic is difficult, in terms of reflecting realistic network traffic when measured over time scales ranging from milliseconds to minutes and hours. More recent studies [53] show that the self-similar model is much more accurate in its statistical aspects than the traditional network traffic models based on Poisson processes or other such relevant processes.

Therefore when designing this part of the system it needs to be taken into account that the traffic in a network can assume various forms and volumes, for example, actual traffic traces captured at routers, peer-to-peer traffic, and so on. The WAN emulator should support the simulation of various traffic models, including self-similar, random, and constant and even

replaying previously captured traffic traces. Since studies of network traffic show that it is self-similar in nature [53], it is more realistic that we opt for emulating the presence of continuous self-similar traffic in our emulated Wide Area Networks. Moreover, packet sizes in Wide Area Networks are not random, but follow a special distribution. Therefore generating artificial traffic with random sizes may not be accurate. Also designing the network emulation to generate and handle artificial traffic would increase the overhead of the system. The large volumes of traffic common in Wide Area Networks would make the emulation useless, unless simulating the network traffic were achieved in some way that did not task the emulation system. Artificial traffic generation is an integral part of the network emulation system. Thus, it is only reasonable to build the traffic generator inside the emulation engine.

Our method should make use of an existing WAN emulation [55]. This network emulator is used with a fault injection tool to test CORBA applications. However, in this emulator some of the concerns were already taken care of such as the traffic simulation, whereas others would need to be dealt with in our proposed network emulation implementation such as the way faults are injected (We will elaborate on other concerns in section 4.2).

3.3.3.3 Proposed architecture of the network emulation

Based on the network emulation of the CORBA fault injection tool and on the discussion above about building a Wide Area Network emulator for our fault injection method, we can use the sample network topology in Figure 3-4 to analyze the system to be emulated.

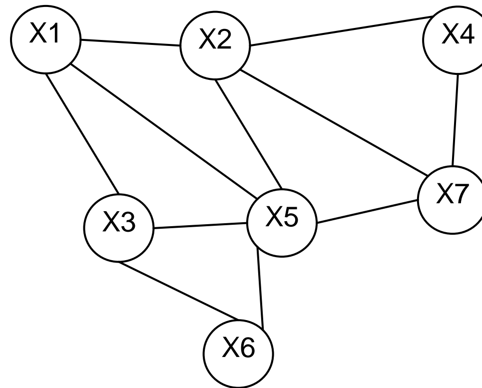


FIGURE 3-4 SAMPLE WIDE AREA NETWORK

The above sample network could be decomposed into two main components: Hops - representing nodes in the target network, and Links - representing communication mediums linking the nodes.

The Hops are the nodes in a Wide Area Network. These should be addressable machines that are capable of running actual application code. In the proposed emulation system, Hops should be represented virtually for the sake of emulating real network Hops and machines that execute the user code to be tested.

Network Hops (or nodes) have the following properties: names, address and Routing tables. The Hops are capable of the following actions: handling/forwarding network traffic and application communication.

Links should also have a virtual representation in the emulation system. Each virtual Link represents a physical network Link. Links should be represented to enforce the properties of actual Links in the network to be tested. Network Links have the following physical properties: bandwidth, delay and error rate. We can also assume networking buffers to be part of the virtual Links even though they are physically part of the networking interfaces in the Hops. They are also capable of a single action: carrying traffic.

Figure 3-5 shows the initial stage of the WAN emulation system design. The WAN emulator intercepts all system communication and emulates a WAN before forwarding messages to their destinations.

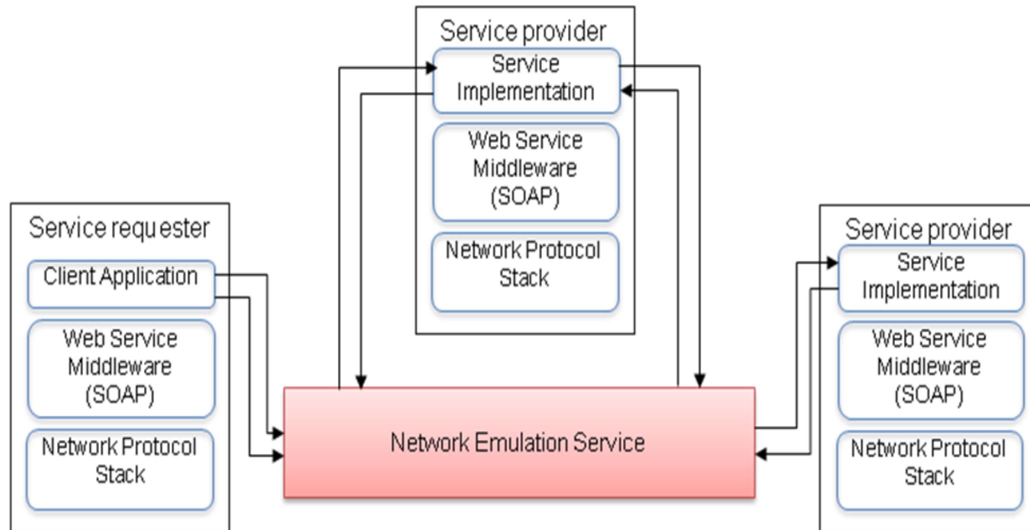


FIGURE 3-5 A HIGH LEVEL VIEW OF THE INITIAL PROPOSED NETWORK EMULATION

Looking into the design of the WAN emulator raises many design choices to be reviewed. These system design options are discussed below.

I. **Network Emulation Engine**

One of the design choices for the network emulation engine would be to have only one programme as a network emulation engine. It would be an immense, single application that would perform the emulation of all the links and nodes of the emulated network in one place. Such a design minimizes the overhead of communication and management. However, such a large application will not only be hard to develop but would require a single machine to host the entire emulation application. For large networks, a single machine may not be able to provide enough processing power to carry out a realistic emulation.

Another design consideration would be a multi-threaded emulation programme. It would distribute the processing between multiple threads. Each thread will represent a node or a link (or a combination of the two) that will be responsible for emulating that node or link in

the emulated network. The emulation includes both network functions and application management. Such a design is more flexible and is easier to develop. However, it might introduce the issue of scalability: could a single machine be capable of running multi-threads that emulate tens or even hundreds of nodes?

Using standalone Web Services like the multi-threaded design, each node or link will be represented by a Web Service. Each Web Service will be responsible for emulating that node or link on the network. However, these Web Services are stand-alone services that can work on their own and be running on a different application space, that might be a different physical machine, OS or Web Service middleware. This could overcome the scalability problem and introduce more benefits such as making the network emulation portable so as to be used by different distributed applications.

II. Providing network emulation to the system being tested:

A major requirement of the network emulation system is the ability to provide network emulation through the intercepted communication between the components of the system being tested. However as discussed earlier (section 3.3.2), communication between the system components being tested would be intercepted by using Fault Injection Mediators. One of the tasks so far for the Fault Injection Mediators is to provide network emulation to the captured messages of the system under test. To gain such a service, the Fault Injection Mediators have to communicate with the network emulation engine.

One of the design options for providing such communication between Fault Injection Mediators and the Network Emulation Engine is to integrate them within a single Web Service. The network emulation engine could therefore be built inside Web Service Mediators as a sub-component. Pursuing this design option would overcome any problem of communication between the Fault Injection Mediators and the Network Emulation. That is, when the Fault Injection Mediator captured a message, it would communicate internally with its Network Emulation Engine sub-component and the service of network emulation provided. However integrating the emulation engine inside the Fault Injection Mediators is likely to increase its complexity and overhead, negatively affecting its performance.

The other possible option for providing the network emulation would be where the Fault Injection Mediator and Network Emulation Engine are each built as a standalone service. Figure 3-6 below gives a more detailed view of this network emulation system architecture.

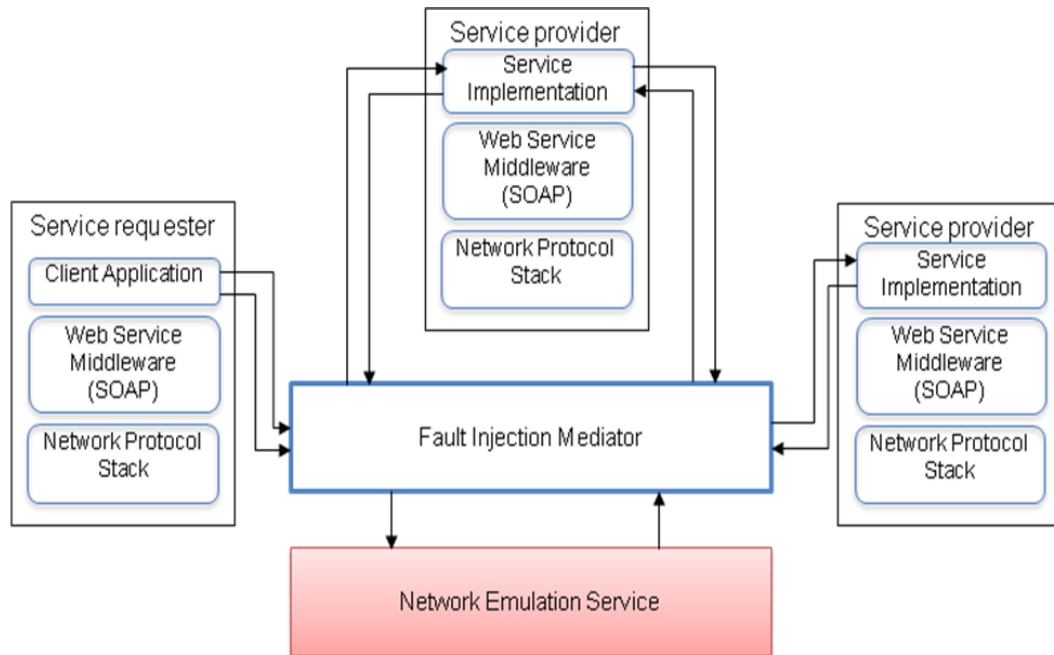


FIGURE 3-6. INITIAL HIGH LEVEL VIEW OF WAN EMULATION CONNECTED TO FIM

Designing the network emulation engine subsystem as a standalone service and also the Fault Injection Mediator also as a standalone service will reduce the emulation engine complexity and produce a clearer, more modular, design. In addition communication between the two services can be made on demand as required. Moreover if stand-alone services are applied, each service can work on its own and can run on a different application space which can be a different physical machine, OS, web service middleware or location. This could enhance the generality of the tool by separating the network Emulation Service from the Fault Injection Service so the Network Emulation can be used to provide Network Emulation to different distributed applications. However this is likely to introduce extra communication overhead (delays), therefore care should be taken when designing the system to address this issue and keep it to a minimum.

3.3.4 Faults affecting web service systems

From a fault-tolerance perspective, distributed systems have a major advantage, which is that they can easily be made redundant, a core attribute of all fault-tolerance techniques. However, distribution also means that the imperfect and fault-prone physical world cannot be ignored, so that as much as they help in supporting fault-tolerance, distributed systems may also be the source of many faults. In this section we review some of the faults and problems of distributed systems and in particular Web Service systems.

In Web Service systems distributed over the Internet where services are communicated by message passing, the failure of a service can be exhibited by its external behaviour, which is entirely represented by the messages the service sends (or fails to send). Thus, the failure of a service can be emulated by having the service send manipulated messages, failing to send, or delaying message sending. There is no need to be concerned about the internal conditions of the failed service.

A web service can fail in various ways. Classifying a system's faults helps in the design of fault injection campaigns that faithfully represent the actual faults that can exist in the system. There are many approaches to the process by which a system can fail. Some of these approaches are revisited below, in an attempt to cover the largest possible set of web service system faults.

A fault model is a type of fault that could occur in a system as it is running. There are many types of fault that can affect distributed systems, as classified in [20][56][57][58]. From these, we present a summary list of the classes of faults which might affect active Web Service applications. We will go into particular depth over the classifications most relevant to the faults that are selected to be injected by our fault injection method.

One of the classifications [20] has classified the faults affecting distributed systems in general into five types, as follows:

Newcastle University

1) ***Physical Faults***: this class covers all faults in the hosting hardware. For example faults effecting RAM or processor registers such as a hardware fault may cause a bit-flip in memory address.

2) ***Software Faults***: this class covers all faults in the distributed application's software. For example, design or programming errors such as where an application may pass an invalid pointer to the middleware.

3) ***Communication faults***: this class covers all faults in the communication system. For example, message loss, duplication, reordering or corruption, while this class of faults does not affect middleware that is built over a reliable LAN. However, Web Services run over WANs may be unreliable, especially in message delivery times.

4) ***Resource-management faults***: this class covers all faults related to resource management. For example, memory leakage and exhaustion of resource such as file descriptors.

5) ***Lifecycle faults***: this class covers all *process-aging* faults, for example, referencing a destroyed object.

The last two classes of faults (resource-management and lifecycle faults) are sometimes grouped together under the category of environmental faults. This list only covers classes of faults commonly found within most distributed systems.

Another fault classification approach [58] classifies faults in terms of their type in the distributed system instead of considering their location, cause or environment. This classification is based on the fact that message exchange is the attribute that distinguishes distributed from standalone systems. According to this way of classifying faults in distributed systems, faults are classified as omission faults, value faults, timing faults and arbitrary faults as follows:

1) ***Omission faults***: this class of faults prevents the expected response message from being received by the requester at all. An example of a component suffering from an omission fault is a communication link which occasionally looses messages.

2) ***Value faults***: this class of faults causes the response message to be received within the specified time frame, but with corrupted or erroneous contents. For example, a communication link which delivers corrupted messages on time suffers from a value fault.

3) ***Timing Faults***: this class of faults occurs where the response with correct contents is received outside the specified time frame, either early or late. An overloaded processor which produces correct values but with an excessive delay suffers from a timing failure. Timing failures can only occur in systems which impose timing constraints on computations.

4) ***Arbitrary or Byzantine faults***: subsumes all the previous three classes of faults. An arbitrary fault causes any deviation from a specified behaviour in terms of timing and/or value. It is possible for a component to fail in both domains in a manner which is not covered by one of the previous classes. A failed component which produces such an output will be said to be exhibiting an *arbitrary failure* (*Byzantine failure*).

We have chosen a fault model based on the above model of fault classification [58] as a general fault model for distributed systems, shown in Figure 3-7. The relationships among these fault classes can be expressed through the recognition that that an omission fault can be treated as either a (infinitely) late timing fault or a value fault causing no value to be produced. In addition, a Byzantine fault causes any violation from the specified behaviour in terms of timing and/or value.

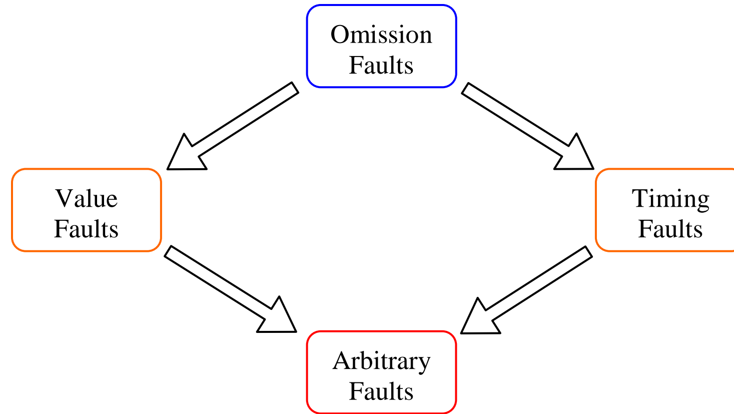


FIGURE 3-7 FAULT CLASSIFICATION HIERARCHY

As denoted by the colour-coding in Figure 3-7, we have slightly adapted this classification and consider faults in three classes: timing faults, value faults, and arbitrary faults.

Therefore our fault injection technique should be able to support two main fault classes which could lead to timing and vault faults being injected to the system being tested. The first class of faults is Network Faults, whereas the second class of faults is Software Value faults. We will review how these faults can be mapped into our fault injection system in the following sections.

3.3.5 Timing Faults

Timing faults can be the cause of what are called communication faults. Communication faults can be a result of either Network faults or Web Service applications crashing or overflowing buffered messages. Our research will pay more attention to Network faults as the main cause of timing faults and able to be used to simulate Web Service application faults that cause Timing faults.

Timing faults caused by Communication faults in general and network faults in particular can affect Web Service systems. A source of Web Service failure common in most distributed systems derives from network faults. The most obvious failure is a permanent hard failure of the entire communication between system components, such as network

partitioning [59]. Such failure makes communication between services in the system impossible. This type of failure can lead to partitioning of the system into multiple parts that are completely isolated from each other. The danger here is that the different parts of the system might perform activities that conflict with each other.

A different type of network failure is an intermittent failure. These are failures where messages exchanged through a network are lost, reordered, or duplicated. These faults are not always due to hardware failures, for example, a message may be lost because the system may have temporarily overflowed for buffering it [59].

Messages taking different paths through the network may cause message reordering. If the delays incurred on the network paths are different, they may overtake each other. Duplication can also occur in different ways, for example, it may occur as the result of a retransmission, due to the wrong conclusion that the original message was lost in transit.

One of the main problems with unreliable networks is that it is not always possible to make sure that a message that was sent has definitely been received by the intended remote service. A common technique for dealing with this problem is to use some type of positive acknowledgement protocol. In such protocols, the receiver notifies the sender when a message is received. However, there is also a possibility that the acknowledgement message itself will be lost, so that such protocols are for optimizing the problem but not solving it. The most common technique that is used for detecting lost messages is based on time-outs. If a positive acknowledgement is not received within some reasonable time interval, we conclude that it was lost somewhere in the network. The problem of this approach is how to distinguish between a message (or its acknowledgement) that is simply experiencing a delay in the network, from the situation in which a message has actually been lost. If the time-out interval is made too short, then there is a risk of duplicating messages and also reordering in some cases. If the interval is made too long, then the system becomes unresponsive [59].

While time delays are not necessarily failures, they can certainly lead to failures in some cases. As noted in the previous section, a delay can be misconstrued as a lost message. There are two different types of faults caused by message delays [59]. One type results from time

delays (jitter), when the time it takes for a message to reach its destination can vary significantly. As composed services usually run over unstable Internet environments, the time delays can be due to a number of factors, such as the route taken through the network, congestion in the network, congestion at the server sites (e.g., a busy receiver), intermittent hardware failures, and so on. If the transmission delay is constant, then it can be much more easy to assess when a message has been lost. Therefore, some communication networks are designed as synchronous networks, so that delay values are fixed and known in advance. However, even if the transmission delay is constant, other problems such as out-of-date information may still arise, although this is outside the scope of our work.

Transmission delays also lead to a complex situation about Message ordering. This is as a consequence of the fact that transmission delays between different services in a composed system may be different and also due to the different routes taken by the individual messages and the different delays along those routes. As a result, different services may see the same set of messages, but in a different order.

As shown above, most time faults are caused by network faults or the system. Therefore we have decided to use network faults to simulate faulty behaviour in the system.

3.3.5.1 Types of Network Faults

Networking faults can be classified based on three attributes: fault location, fault duration and fault type. A network fault can happen in one of the following networking component locations:

- Node: a node on the network can fail in many ways. Failure of a networking node affects its entire links. It might also affect the entire network if it results in partitioning it or breaking the routing.
- Link: a single link on the network can also fail in many ways. Failure of a network link affects only the traffic going through it. Similar to a failure of a node, it might also affect the entire network if it results in partitioning it or breaking the routing.

The other attribute used for classifying network faults is the fault duration. A network fault could either be a permanent hard failure or an intermittent failure, lasting for a temporary period, as explained in the last section.

The last attribute of network fault classification is the type of network faults. These are classified into the following categories:

1. Delays: delays can occur because of networking resources overloading. Network delays can cause a failure of applications. They could also be the source for other types of networking faults such as data loss or further delays.
2. Data loss: packets or entire messages could be lost due to buffer overruns, faulty links and nodes, and so on. Data loss can also be the cause of the failure of poorly designed applications and could also lead to other networking faults like delays.
3. Transmission errors: transmission errors occur due to faulty links and/or nodes. Unlike data loss, the data arrives at its destination but corrupted. Like the other types of networking faults, transmission errors may be the cause of the failure of applications and may also lead to other networking faults.
4. Data reordering: packet or message reordering occurs because of network traffic congestion and routing table reconfiguration. Distributed applications must ensure proper handling of data reordering. Otherwise, they will break and may lead to further networking faults.
5. Partitioning: network partitioning can occur due to node and/or link failures. It may also occur due to faulty routing.

The above classification should be considered and taken into account when producing our fault injection testing system. All possible network faults should be injected explicitly into the system under test and the reaction of the system should be observed so as to evaluate it. Table 3-1 shows how the three classifying attributes could be combined and injected explicitly to cover all the cases of networking faults together with some examples.

	Fault Type				
	Delays	Loss	Errors	Reordering	Partitioning
Permanent hard failure	Node generates extra artificial traffic	Node fails	Randomly corrupting SOAP body	Node changes routing causing dual routes	Node fails
	Link parameters change causing delays	Link fails causing data loss	Link fails causing errors	Indirectly: link causes delays causing retransmission leading to reordering	Link fails
Intermittent failure	Node generates a burst of extra artificial traffic	Node fails temporarily	Randomly corrupting SOAP body	Node changes routing table without breaking routing	Node fails
	Link buffers fill causing delays	Link fails temporarily	Link fails causing errors (e.g. due to temporary overheating)	Indirectly: link causes temporary delays causing retransmission leading to reordering	Link fails

TABLE 3-1 HOW TO INJECT NETWORK FAULTS USING OUR METHOD

3.3.5.2 Injecting WAN Faults

Although the fault injection tool has to provide network emulation as a run-time environment, to run the system to be tested, it also needs to provide the ability to inject all possible network faults and, of course, application-specific faults; and then to observe the tested system's reaction to such faults. Our initial architecture designed in Figure 3-6 should be extended to emulate the faults common in Wide Area Networks. Some design options and considerations about where to inject network faults and how should be taken into account as follows:

- **Network Fault Injection Location:** as suggested in Section 3.3.3.3 (Proposed Architecture of the Network Emulation), the communication between the system components (Service requesters and Service providers) should be intercepted by the

sub-component Interception Mediator, and the network emulated by the Network Emulation Engine sub-component. Therefore the network fault injection subsystem could be built inside the emulation engine or could also be built inside the Interception Mediator.

Integrating the fault injection as a subsystem inside the emulation engine is likely to increase its complexity, reduce its probability of extension and introduce more overhead, negatively affecting its performance. On the other hand, designing the network fault injection subsystem inside the Interception Mediator will reduce the emulation engine's complexity and produce a clearer, more modular, design. This design option will also enhance the implementation of the Interception Mediator and its extendibility to support injecting other faults into the system under test. Moreover only the Interception Mediator needs to be modified in the case of supporting additional platforms (e.g. CORBA, RMI, raw sockets, etc.).

- **Network Fault Sources:** As network faults should be injected as a result of the Network Emulation subsystem, the design of the network fault injection subsystem should take into account that the networking faults described earlier should be injected based on the decisions coming from the emulation engine. If the fault injection subcomponent is placed in the Interception Mediator, the Interception Mediator needs to contact the Network Emulation Engine when any communication message is intercepted in order to provide the network emulation.

Figure 3-8 below shows the updated overview system architecture that supports Network Fault Injection.

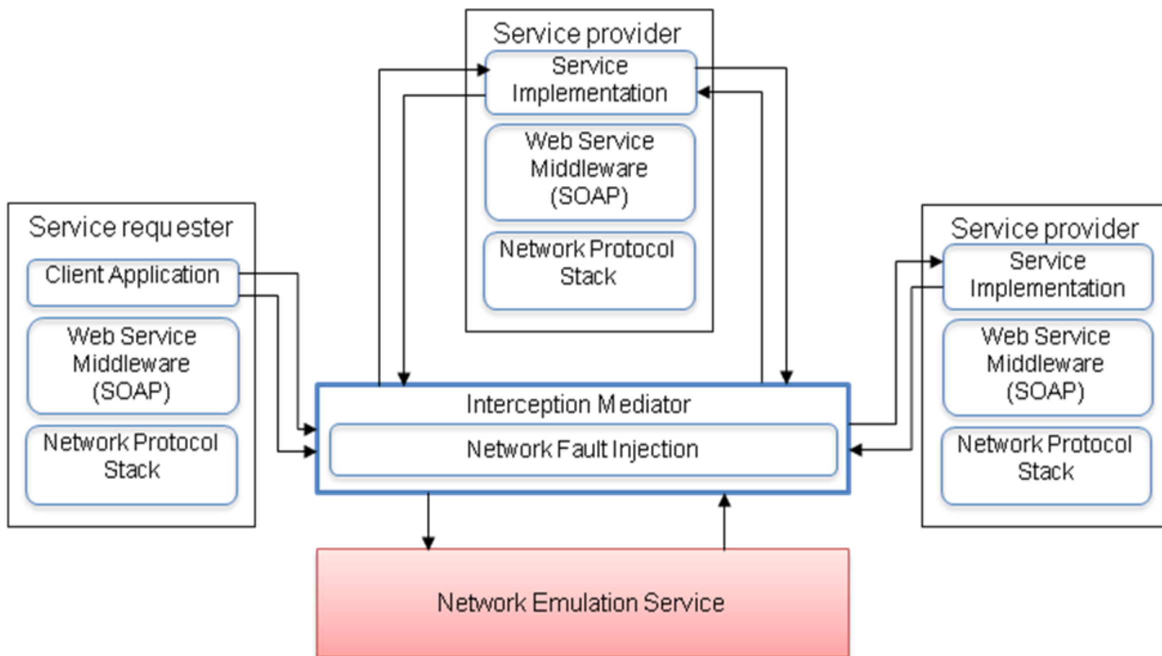


FIGURE 3-8 OVERVIEW OF THE NETWORK FAULT INJECTION LOCATION

3.3.6 Software faults

Simulating software faults to assess the impact of remaining bugs or validating fault tolerance mechanisms is extremely important. Many studies show a clear predominance of software faults (i.e. programming defects or design problems) [32][60] as the root cause of system failures. As the enormous complexity of today's software increases, the volume of software faults will tend to increase as well.

Web Services are not an exception, as they normally use intricate software components that implement a compound service, in some cases comprising compositions of a number of web services, which make them even more complex.

Web Services provide a simple interface between a Service requester and a Service provider and are a convenient means of exchanging data. Interface faults, related to problems in the interaction between software components, are particularly relevant in an environment that is based in Web Services.

Based on the literature of classifying software value faults, we generated Figure 3-9, which shows all possible causes of value faults.

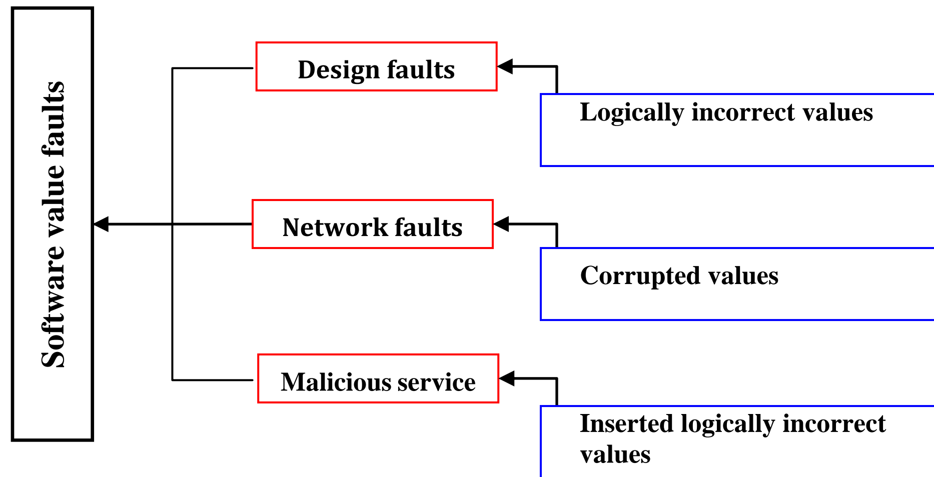


FIGURE 3-9 SOFTWARE VALUE FAULTS

In fact, web service applications should provide a reliable interface to client applications, even in the case of invalid inputs, which may be caused due to bugs in the code applications (design faults), corruptions caused by network failures (transmission faults), or even security attacks (malicious service). They can be tolerated by some fault tolerance mechanisms applied to the system, for example, being caught by exceptions or discarded in some voting process [55, 61].

Our proposed approach will concentrate on injecting a set of tests (i.e. invalid Web Services call parameters) that are applied during execution in order to reveal both programming and design problems. Injecting invalid values into the system helps in assessing the system's robustness.

The robustness of a system can be defined as its capacity to present acceptable behaviour in the presence of faults or stressful environmental conditions [62]. Testing for robustness is becoming an attractive evaluation method to characterize the behaviour of a system in the presence of invalid values [63][64]. This kind of testing can be achieved by modifying data at a component's interface level and observing its behaviour. It can also be achieved by running the system under a heavy load and observing its behaviour. Robustness testing is particularly useful in assessing how a system uses fault tolerance mechanisms in order to

prevent invalid input, such as input that may be received by a system when it is under malicious attack. It also can be used to assess the system under different load so as to give an indication of how a scalable a system will be when in use.

3.3.6.1 Injecting software faults (value faults)

Value faults can be injected into the system through many different SWIFI approaches. For example, a code mutation approach could be very useful for injecting value faults. A code mutating SWIFI tool utilizes the source code of the application under test to inject value faults directly into the code. This approach violates the requirement of not utilizing the source code of the application. However in the robustness testing approach, the explicit interface of the system is utilized to inject faulty data at the system interface while it is under test. This approach is limited by the requirement that the system has clear and explicit interfaces. In the Network level fault injection approach, faults are injected into the system through the messages being exchanged between the system components. This approach suffers the least limitations and has been shown to be effective for injecting software faults [20].

Web Service applications interact through invoking operations on the Web Service through message exchange. In order to inject faults into Web Service systems, messages must be captured and manipulated. As web service applications communicate with one another through message exchanges, messages provide a natural and convenient way of injecting faults into the system. Exchanged messages could be intercepted and manipulated for injecting not only software faults but also communication faults, as our work proposes.

As discussed in Section 3.3.2, the location of capturing the exchanged messages should be moved up to avoid the limitations of Network level fault injection, by emulating this technique at the application level. Messages captured at application level would be taken as complete entities, so they can be manipulated, modified and faults can be injected. In addition messages at this level are captured as XML documents; this makes it easy to target any part of the message and manipulate it, such as targeting individual Remote Procedure Call (RPC) parameters in the message and modifying it to simulate a large number of software faults.

In Web Service systems, the rules used to define the interface of a Web Service are typically encapsulated in a WSDL document. The WSDL document explicitly defines the messages to be exchanged with the Service requester. Therefore, it is possible to use the published WSDL document to decompose the service interface into method calls with their associated messages and within the messages to identify specific parameters.

Thus, by using a WSDL document, information about each message's required structure can be obtained. As a WSDL contains information about the operations, associated messages, parameters and types, it would be possible to arrange this information into groups of information. Each group of information could present all the information about one particular operation included in the WSDL interface required to construct a fault injection trigger. For example, each group contains nodes representing the operation name, message request/response, parameter names and types of each parameter. Although trigging faults can be constructed to any node in the group, our fault injection system is primarily concerned only with manipulating parameters in RPC messages.

There are many testing tools for injecting value faults through function call parameters, such as [64] and [30]. Ballista [64] is a well known tool, with successful results, that uses fault injection to test software components for robustness, focusing especially on operating systems. Tests are made using combinations of exceptional and acceptable input values of parameters of kernel system calls, based on the data types of the parameter list.

As Ballista testing can be performed on almost any API that employs calls with parameter lists [64], our proposed value testing approach could use Ballista as a starting point for injecting value faults through the function call parameters. The suggested approach consists of a set of valid and invalid Web Services call parameters that can be injected during the run-time. The value fault model is based on combinations of exceptional and acceptable input values of function call parameters based on the data types of each parameter as shown in Table 3-1.

Before starting the testing, a number of test values must be generated for each data type used in the system under test. For example, if an operation to be tested requires an integer data type as an input parameter, test values must be generated for testing integers. Values to test integers might include 0, 1, upper or lower integer bounds.

As shown in

Table 3-2, triggering faults is based on the type of the parameters and the return results. For each type of parameter, a list is presented of all possible mutations that can be performed by using a set of mutation algorithms.

Type	Test case	Parameter Mutation
String	Insert ASCII	Randomly insert ASCII character
	Skip character	Randomly remove character
	double character	Randomly double character
	Null	Replace by null value
	Missing value	Replace by empty string
	Predefined	Replace by predefined string
	Non Printable	Replace by string with nonprintable characters
	Add None Printable	Add nonprintable character to the string
	Add Number	Replace by alphanumeric string
	Overflow	Duplicate characters until maximum sized overflows is caused
Number	Null	Replace by null value
	Empty	Replace by empty value
	Absolute Minus One	Replace by -1
	Absolute One	Replace by 1
	Absolute Zero	Replace by 0
	Upper Bound	Upper Bound value
	Lower Bound	Lower Bound value
	Upper Plus One	Replace by Upper Bound + 1
	Lower Minus One	Replace by lower Bound - 1
	Between upper and lower	Random values between upper and lower bounds
Date	Null	Replace by null value
	Empty	Replace by empty date
	Max	Replace by maximum valid date
	Min	Replace by minimum valid date
	Max Plus One	Replace by maximum valid date + 1
	Min Minus One	Replace by minimum valid date – 1
	Different formats	Switching the month and day fields

TABLE 3-2 PARAMETER VALUES MUTATION ALGORITHMS.

By using the above table and creating triggers at specific parameters of the message, our fault injection method must target individual elements of a message, rather than inject random faults into exchanged messages as in standard network level fault injection techniques (and as used in some other fault injection tools, such as the CORBA fault injection-testing tool [65]). An important aspect of our proposed testing method is that the source code of the Web Services is not required. This is true for both the Service requester and the Service provider.

Value fault injection should be designed as a subcomponent of the designed fault injection system, as shown in Figure 3-10.

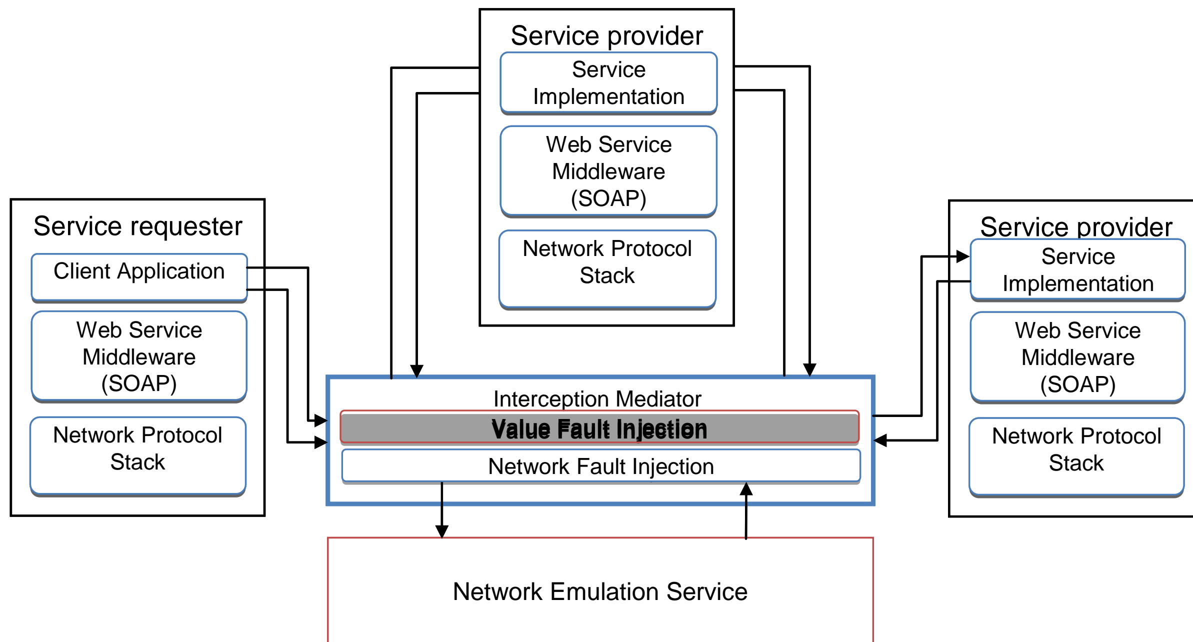


FIGURE 3-10 SOFTWARE VALUE INJECTOR

The same design as discussed in Section 3.3.5.2 (Injecting WAN faults), about extending the emulator architecture to support Network fault injection, also applies for the addition of the value fault injection subcomponent. To maintain modularity of the design and minimize the system complexity and overhead, the value fault injection subcomponent should be excluded from the emulation engine. It should be attached to the Interception Mediator as a separate

fault injection subcomponent. Figure 3-10 shows extended system architecture that supports a value fault injection subcomponent inside the Interception Mediator alone, with the Network Fault Injection subcomponent.

Although such fault injection methods already exist, our proposed value fault injection method could be performed with a combination of other faults. One of our aims in this work is to study the effect of injecting value faults through the function call parameters with simultaneous injection of timing faults. By injecting value and time faults together, we could simulate what are known as Byzantine faults.

3.3.7 Value and Network faults injected together

Although much work has been done in the area of testing value faults in Web Service applications, our work is differentiated from others through the way that the value fault is injected into the system alongside the injection of delays into the system under test. Injecting delays into the system is based on the Network Emulation Engine, whose general task is to emulate a WAN, in order to give the impression that the system under test is running over a WAN.

Injecting value and time faults together into the system is in order to simulate what is called an *arbitrary failure* (*Byzantine failure*). *Byzantine failure/arbitrary failure* cause a deviation from the specified behaviour of the system in terms of combined timing and value faults [58]. It is possible for a service to fail in both the domains of faults (value and time faults) in a manner which is not covered by either of the two domains alone. A failed component which produces such an output will be said to be exhibiting an *arbitrary failure* (*Byzantine failure*). As detailed earlier, our fault injection model is a combination of two main fault models: communication faults (Network faults) and interface faults (software faults/value faults). Choosing how the two fault models are combined to form testing scenarios should be left to the tester, in order to make the method more flexible by using both Network configuration files and the generated value test cases. However the method should be

designed with a default setup. The method should be able to be configured through many built-in setup cases. Table 3-3 below shows how the two fault models can be combined together and how they override each other in any constructed test cases.

Possible Test cases	Network faults			Generated Script
	Drop	Delay	Random Corrupt	Manipulating value
Case 1	Applied	None	None	None
Case 2	None	Applied	None	Applied
Case 3	None	None	Applied	None
Case 4	None	None	None	Applied

TABLE 3-3 POSSIBLE COMBINATION OF INJECTED FAULTS

As shown in Table 3-3, faults have a priority for injection into the system based on their occurrence. When the message is sent through the network, the first possibility for faults to occur is through the network, while other faults may take place thereafter. Therefore the table shows injecting network faults have more priority than injecting value faults. However value faults can be injected into the same message where a network delay was injected. Moreover value faults are also injected when no network faults have been injected into the message. The next chapter will show how a user can control and choose the desired scenario test case by using well defined configuration files provided by the tool.

3.3.8 System monitoring (failure detection)

Here we will discuss the many ways in which distributed systems, and in particular web service systems, can fail and the effects of each failure on the system. In [20] failure modes for CORBA applications have been classified, while [34] has classified failure modes for web service systems. In [64] there is a classification of the target operating system according to the CRASH scale failure modes.

Based on these failure mode classifications, we have summarized the failure modes of Web Service systems as follows:

1) Crash of a service instance/hosting environments: the Web Service crashes or the application server used to run the web-service under testing may become corrupted or the machine crashes or reboots.
2) Hang of a service: hung services are present in the system in a way that can be seen by other services, but do not process requests or return results.
3) Corruption of data coming into the system: this class covers all failure modes resulting from bad or corrupted input.
4) Corruption of data coming out of the system: this covers failure modes resulting from bad or corrupt replies.
5) Duplication of messages: this class covers failure modes resulting from duplicated messages.
6) Omission of messages: this class covers failure modes resulting from lost request and response messages.
7) Delay of messages: delay of messages may cause failure due to message time-outs.

This list is very general. Every Web Service system has its own specific failure modes. However, the majority, if not all, of these failure modes can be classified into one of the above major classes. Listing failure modes can help in both the design of the fault injection campaign and the monitoring of the outcomes of our fault injection testing method.

The effect of each of these failure modes will depend on the capacity of the system's fault tolerance to detect them and prevent the system from deviating from its specified behaviour. Corrupted data coming into the system should be detected by the middleware (or the web service application) and rejected, then raise the appropriate error exception as a response. Corruption of data coming out of the system should be handled by the middleware at the service client. However this is the most severe failure mode when it is not signalled by the system and propagated from the middleware to the application level, where a mechanism must be deployed to deal with this.

The duplication and omission of messages should also be handled by the middleware layer of the service and should raise the appropriate exceptions. However, the omission of messages

from client to service must be detected by the client's middleware, since the service would have no mechanism for knowing the message had been sent, so it could not generate an exception.

If the application server has crashed, it will not be able to accept the invocation and the client will receive an exception from the transport layer. If the application server hangs, it may either accept the invocation but not respond so the client will not know what is happening, or the application server may not be able to accept the invocation at all, producing an effect that is similar to a crash.

Delayed messages are not a failure but may cause timing faults. These should be detected by the middleware at the service side when a response message is not received. However at the service client, there is a problem of distinguishing between a lost request message and the message experiencing a long delay in the network. To minimise this issue, a reasonable time span should be deployed before raising a time-out exception at the service client.

Because of all the problems noted above, some of the failure modes are very difficult to detect. For example, as discussed above, it is difficult to distinguish between crash and hung failure modes, in some cases where the testing run by the service client does not have access to the application server logs where the Web Service is running. In addition some other failure modes are also difficult to detect when the tester has no access to the service client logs, for example the omission of requests when a client request is lost before reaching the service provider. As a result of this, the mechanism deployed to detect omission of request messages at the service client cannot be tested.

To face some of these problems we rely on the logging mechanism of the proposed method. We propose a simplified failure detection that is based on two observable detected outcomes from our fault injection method, as follows: 1) Detecting exception, 2) No effect.

1) Detecting exception: when corrupted data are propagated to the middleware or to the application level, the normal reaction is an exception sent back to the client. Middleware need to have the capacity to detect corruption in the SOAP messages,

while Service implementations need to have the capacity to detect the corruption of faults in the SOAP function parameters by raising exceptions. When the exception of the corrupted data is expected but not detected, the testing method will flag it to the user as a discrepancy. When the expected exception is received it will be classed as the correct behaviour outcome. In addition, unexpected exceptions can be also raised by the service providers. These unexpected exceptions can result from perturbed input parameters in allowed parameter bounds and should therefore be flagged up to the user. The same mechanism applies to exceptions resulting from duplication, crashes and the omission of messages (responses) from service to client which should sent by the service provider.

2) The 'No effect' would be expected mainly from normal message flows continued with no data corruption or exceptions, as, for example, in the case of omission of messages from client to server, since the server would have no mechanism for knowing the message had been sent, so it could not raise an exception. With regard to injecting delay before sending the message, if the time span of the time-out is not reached, this would result in performance degradation rather than time-out exceptions.

It is important to emphasize that all the failure modes mentioned before can be easily observed by not only analysing the tool's logging mechanism (exception, corruption of data), but also by using application server logs if available (failure modes crash and hang of the service) and at service client logs (omission of message from the client to service).

The logging mechanism should be placed in the tool where it can monitor and log all the communications of the system under test and the fault injection activity. The Interception Mediator would be able to capture all the system communication. Moreover in the Interception Mediator all the injected faults (both network faults and value faults) will be performed, it would be the best place to deploy the logging mechanism subcomponent.

However, care must be taken when implementing it. For example the logging mechanism should minimize the overhead latency introduced by the logging mechanism by using simple

logging files. Also data should not be stored into the logging files each time a message is captured.

3.4 Conclusions

This chapter has detailed the Network Fault Injection Service (NetFIS) method. The NetFIS method is composed of a novel Wide Area Network Emulator, combined with a Fault Injection Mechanism for injecting two classes of faults: communication faults and interface faults.
The novel fault injection technique used allows communication faults to be injected based on a WAN emulator and injecting software faults (interface faults) into the intercepted message allowing perturbations of specific RPC parameters at middleware message level.

Previous work which called upon the WAN emulator for testing CORBA applications has been adapted for testing Web Service applications. This was done by adopting the architecture of a Wide Area Network emulator used for testing CORBA systems, and extending it to test composed service systems so as to emulate customizable and controllable WANs, over LANs. This way the Web Service systems are tested on virtual WANs that are very similar and comparable to the actual target WAN environments. Testing over these virtual WANs will not encounter the problems of using real WANs. Using the real network is time consuming and costly in terms of setting up a WAN; it also presents difficulties in controlling a variety of network conditions and network parameters.

Injecting communication faults is achieved by the use of the WAN emulator. That is done by controlling network conditions such as latency, data loss/error/reordering and bandwidth and also putting more stress or load on the target network. Errors and faults may take a long time to occur. Some errors may not occur without applying a certain chain of events. Our method would enable the tester to control the virtual network environment and its parameters to test the system under different circumstances such as different dropping rate, error rate, delay, and so on. The method also generates additional traffic workload on the tested system in order to produce more realistic results.

Interface faults are injected through intercepting the messages exchanged between the services. In order to inject interface faults into Web Service applications, messages are captured and manipulated. As Web Service applications communicate with one another through message exchanges, messages provide a natural and convenient way of injecting faults into the system. The location of capturing the exchanged messages should be moved up to avoid the limitations of Network level fault injection and emulating this technique at the application level. Messages are captured at application level as complete entities so they can be manipulated and modified and faults can be injected. In addition messages at this level are captured as XML documents; therefore it is easy to target any part of the message and manipulate it, for example, targeting individual Remote Procedure Call (RPC) parameters in the message and modifying them to simulate a large number of software faults.

4 Chapter 4 - NetFIS applied to SOAP based Web Service

The Network Fault Injection Service (NetFIS) detailed in Chapter 3 describes a generic method that can be applied to a number of RPC based middleware [17], such as CORBA, Web Services and so on. This chapter applies the FIM to Web Service middleware in order to demonstrate the method. This implementation is termed Network Fault Injection Service or NetFIS.

NetFIS is conceived in four distinct phases. The first phase is an implementation of a fairly conventional fault injector proxy, but with the enhancement that it processes middleware messages as opposed to network packets. The second phase is the network emulator which is implemented to give the sense that the system is running over a WAN. The third phase is built onto the first phase, allowing network faults to be injected into the system based on the network emulator in the second phase. The fourth phase builds on the first, allowing meaningful faults to be injected into specific parts of a middleware message.

4.1 Web Service middleware system

The NetFIS implementation for Web Services uses the method described in Chapter 3 and provides a performance and fault tolerance assessment method that can be applied to SOA based on any Web Service middleware SOAP stack (see Figure 4-1). Because our fault injection has been made applicable to any web service middleware system, the implementation of our method is independent of any Web Service middleware and runs as a standalone application. This implementation has, however, been applied to a range of Web Service middleware such as Apache Axis 2 and JBOSS Web Service SOAP 2.3 for the SOAP implementation.

This NetFIS implementation can be deployed with any heterogeneous distributed system comprising many different machine architectures, with the middleware layer allowing

interoperability between them (See Figure 4-1). The NetFIS must be generic enough to overcome any heterogeneous middleware layer of Web Service systems.

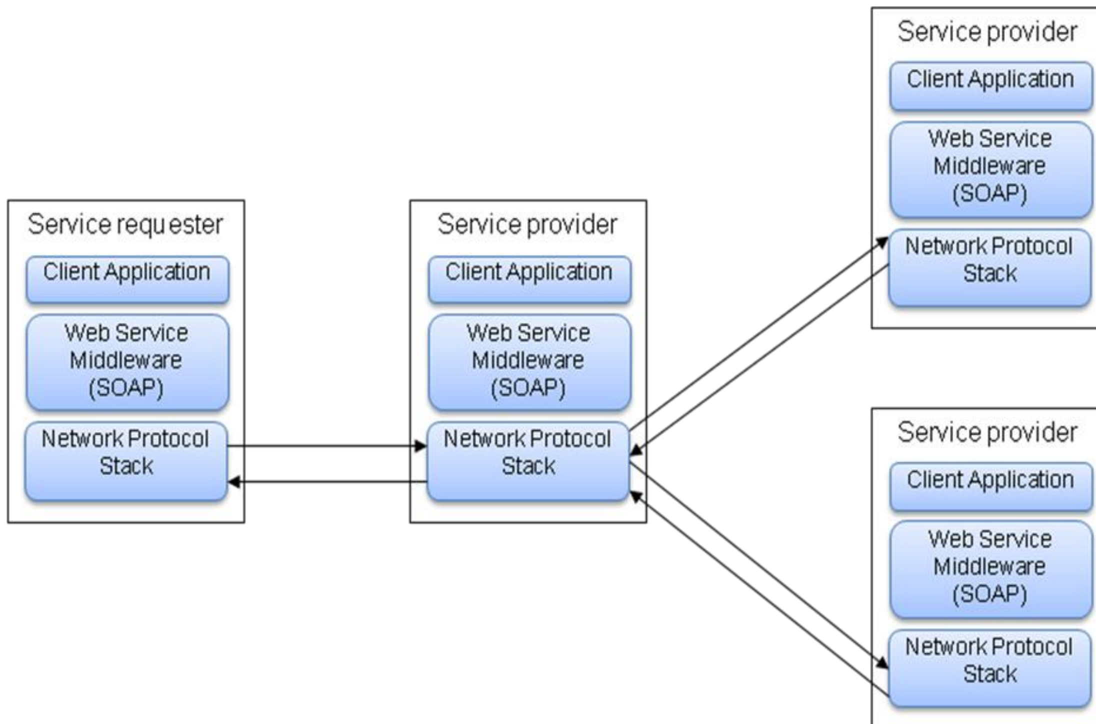


FIGURE 4-1 WEB SERVICE MIDDLEWARE SYSTEM

The NetFIS implementation addresses the following areas which were defined in Chapter 3:

1. **Fault injection mechanism**
2. **Network emulation mechanism**
3. **Injecting network faults**
4. **Injecting software faults**
5. **Failure detection (System monitoring)**

This chapter shows the details of how the NetFIS method has been implemented and applied to Web Services and describes the concepts behind network emulator implementation. In addition it demonstrates how network and software faults were injected into the system being tested. Finally it demonstrates how the system under test can be monitored and how failures can be detected.

4.2 Fault injection mechanism

As discussed in Section 3.3.1 (Fault injection location) our fault injection method is based on injecting Network faults and also software faults at application level. We have moved the fault injection location away from the Network interface and positioned it at the application level by using proxies between the Service requester/s and the Service provider/s participating in a Web Service system.

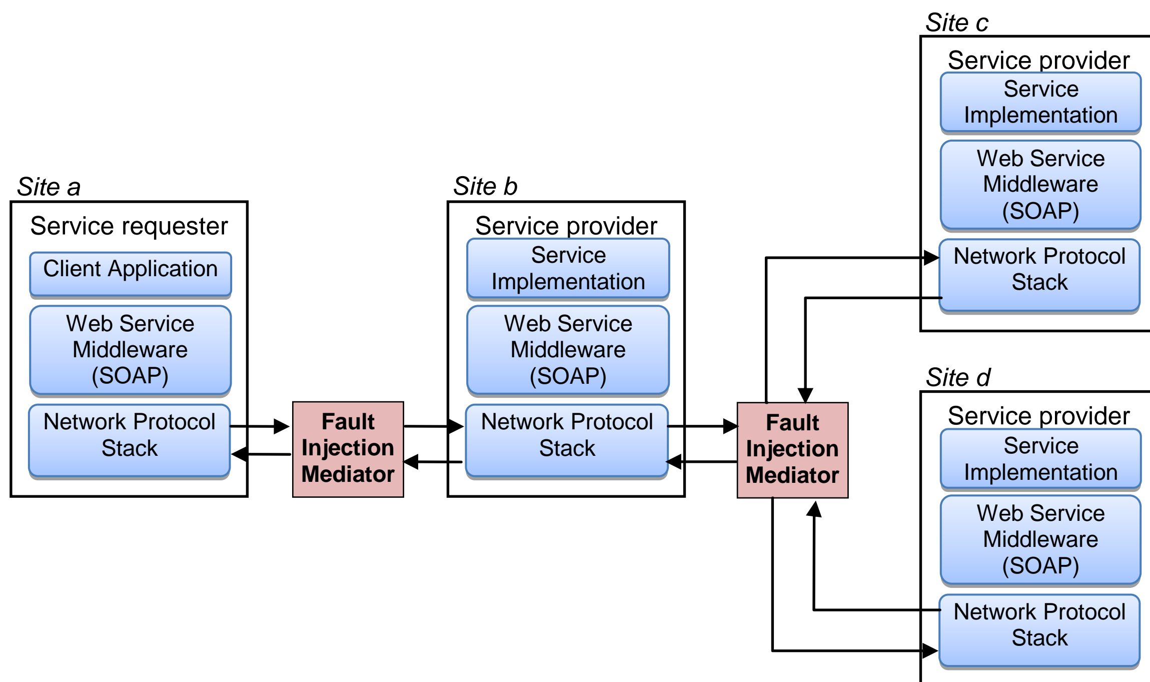


FIGURE 4-2 THE CHOSEN LOCATION FOR INJECTING FAULTS INTO THE SYSTEM.

As shown in Figure 4-2 the Fault Injection Mediator (Interceptor) instances can be installed into the system under test depending on the total number of Service providers and Service clients. For example in Figure 4-2 the Fault Injection Mediator (proxy) instances are installed three times. Every connection between a Service requester (client) and a Service provider needs one fault injection Interceptor instance. According to the system in Figure 4-2, for every Service provider, a proxy is generated by the fault injection tool. The Service requester

can therefore invoke the intended Service provider through its proxy. In case a Service provider needs to connect other Service providers in order to serve a request, this Service provider (as in site b in Figure 4-2) will become a Service requester to connect to the intended Service provider (Service provider in site c in Figure 4-2). Therefore the Fault Injection Mediator instance generates a proxy Web Service for the Service provider to be called through. In this way all the communication between the system components (clients and Web Services exchanging messages) under test can be intercepted by the generated proxies. Request messages coming from Service requester at site a) are intercepted by the Service provider proxy between them; and response messages coming from Service provider at site b) are also intercepted by the same Service provider proxy. As messages go through proxies, faults are injected into the messages by the fault injection engine tool.

The proxy code is further sub-divided into two sub-components: one component is for intercepting incoming messages (requests) whereas the second component is for intercepting outgoing messages (responses). 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 to allow differentiation between the two message types. Although it would be possible to utilize only one pathway for both messages (incoming and outgoing messages), subdividing it into two components has certain advantages in terms of the flexibility of the method and injecting faults.

The extra flexibility offered by our fault injection tool is to separate the interception location of the requests from the interception location of the responses by deploying two Fault Injection Mediator (FIM) instances between any Service requester and Service provider as shown in Figure 4-3. The tasks of each FIM are based on where it is deployed (Service requester side or Service provider side).

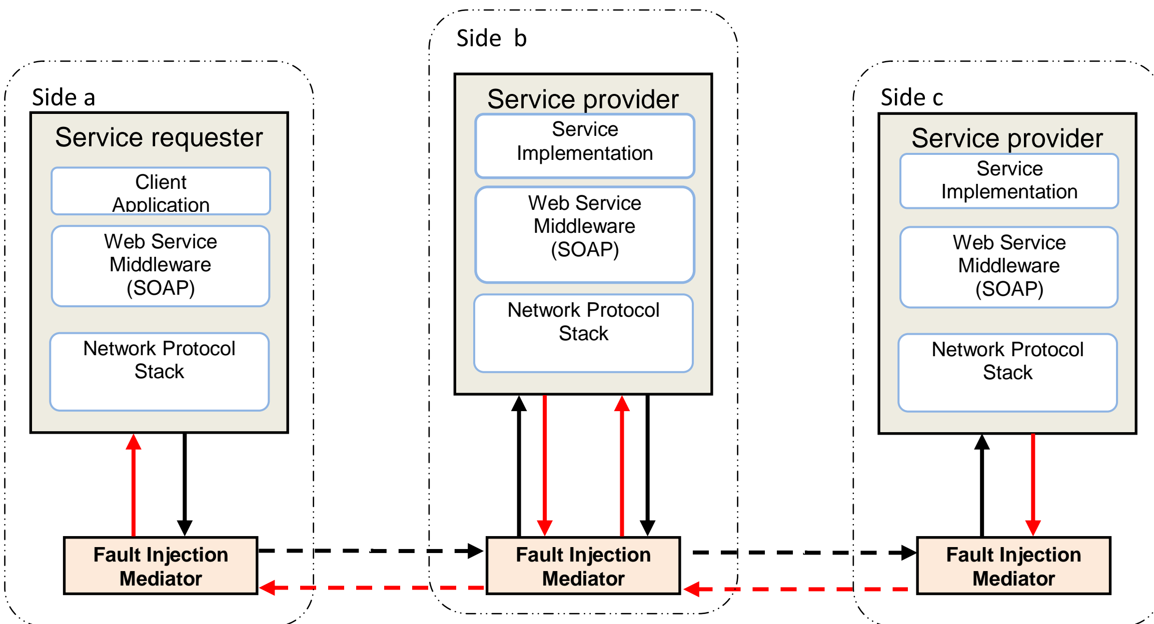


FIGURE 4-3 A GENERAL VIEW OF THE INTERCEPTION MEDIATOR PLACED INTO A COMPOSED SERVICE SYSTEM

- *The FIM at the Service requester tasks are:*

1) Generating any number of Web Service proxies that this Service requester might invoke;

2) Intercepting incoming messages from Service requester;

3) Injecting the appropriate faults (Network and Software faults);

4) Forwarding the requests to the other FIM at the actual invoked Web Service;

5) Forwarding the responses to the Service requester.

- *Whereas the FIM at service provider tasks are:*

1) Intercepting outgoing messages from Service provider;

2) Injecting appropriate faults (Network and Software faults);

3) In the case of Web Service systems where the Service provider needs to invoke another Service provider in order to serve a request, the FIM at Service provider can generate any number of Web Service proxies required. In other words, any Service requester invoking any number of Web Services has its own FIM. In addition every Service provider has its own FIM;

4) Forwarding the responses to the other FIM at the service client.

For instance as shown in Figure 4-3 the Service requester on site a) can send a request to the Service provider on site b) through the proxy generated by FIM at site a). Then FIM at site a) injects faults (if any) through its subcomponents (detailed later); thereafter it forwards the request to FIM at side b) whose task here is only to forward the request to the actual invoked Web Service at site b). In case the Web Service at site b) needs to call another Web Service (Web Service at site c) in order to serve the request, it sends the request to the FIM at site b) through the proxy generated for calling the Web Service at site c). Then the request at FIM at site b) forwards it to the FIM at site c) which will forward it to the actual called Web Service at site c).

This way of deploying two instances of FIM (which both separates the interception of the requests and is separated from the interception of the responses) enhances the architecture of the Network emulation, increases the performance and simplifies the monitoring of the system under test. Every Fault Injection Mediator (FIM) will be able to serve the emulation of only one network node of the target emulated network and also could provide other services such as injecting faults, as discussed in Chapter 3. The proposed architecture of the network emulation is maintained by providing one Interception Mediator for each node of the target emulated network (this is detailed later in the section network emulation). The performance is also increased by the possibility of running each FIM instance in a separate machine. Moreover different fault injection configurations can be used by feeding each FIM with any desired different fault model configuration file. Thus, as a consequence, monitoring the system will be also more easily realised. We will elaborate on this in the coming sections of the implementation.

The implementation of FIM is divided into two main components. The first component is the implementation of how the messages are intercepted whereas the second component details how the faults are injected into the system under test. In the next section, the implementation details of how the FIM is implemented are discussed.

4.2.1 Fault Injection Mediator Service

The Fault Injection Mediator Service (FIMS) is the implementation of the FIM. The FIMS is implemented as a SOAP-based Web Service and is to be deployed between the system components (Service requester/s and Service provider/s). In addition to its main task of injecting faults, the FIMS acts as a Proxy Web Service that the Service requester will need to invoke in order to be connected to the actual called Web Service. In this way all the messages exchanged between the system components are captured and faults can be injected.

Based on the architecture of the Fault Injection Method discussed in Chapter 3, the location of our fault injection method is implemented as a standalone Web Service application between the system components under test. The FIMS is the core of our fault injection tool, whose main tasks are to intercept the messages exchanged between the components of the system under test, injecting the proper faults and mentoring the system during the test.

The implementation of the FIMS is divided into three sub components (Proxy Generator (PG), Message Interceptor (MI), and Fault Injection Engine (FIE), in order to fulfil the proposed architecture as shown in Figure 4-4.

Before going into further detail about how the FIMS and its sub-components are implemented, it will be worthwhile to explain a general overview of how messages are processed and faults injected into a system using our tool. This will help in understanding how the messages are intercepted and how the sub-components work.

– *The message's journey through NetFIS:*

The scenario shown in Figure 4-4 is a simple system consisting of two applications (Service requester and Service provider), communicating with each other by exchanging SOAP messages over a Wide Area Network Emulator.

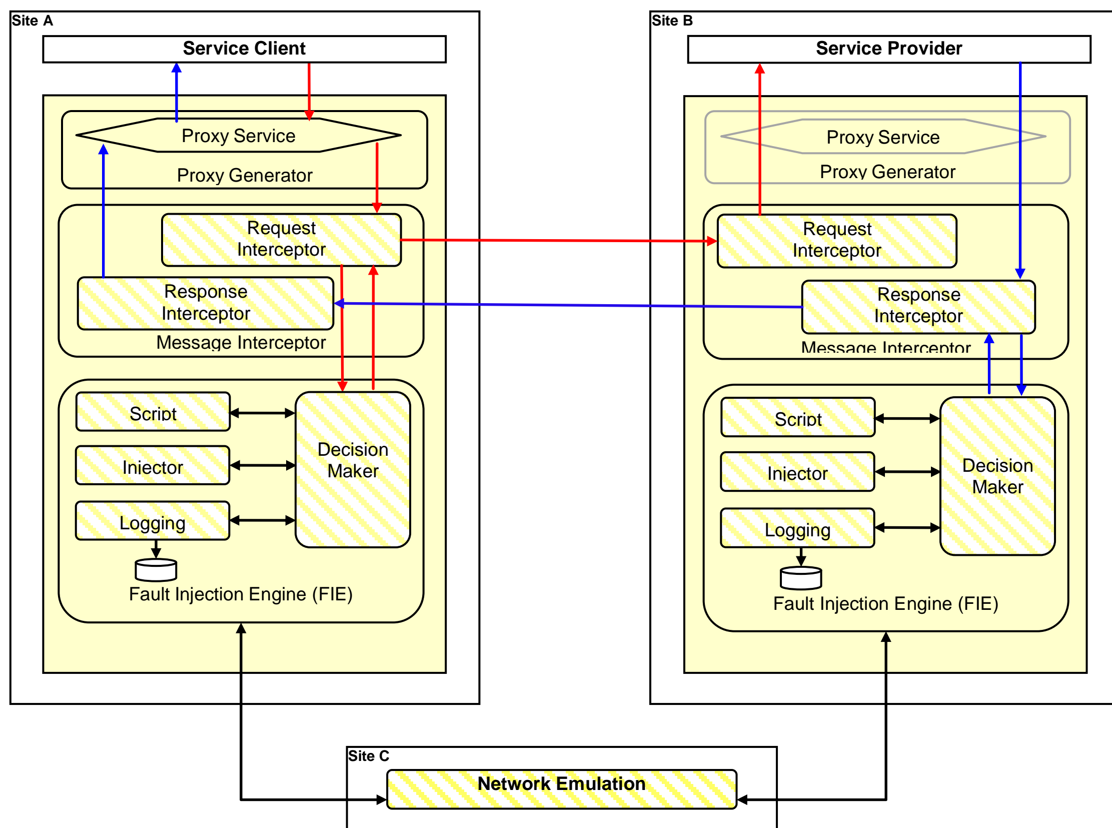


FIGURE 4-4 NETWORK FAULT INJECTION SERVICE (NETFIS)

The request messages from the Service requester to the Service provider are processed by the *NetFIS* in the following steps, as shown in Figure 4-5:

1- The Proxy Generator (PG) sub-component of the FIMS deploys a Proxy Web Service (PWS) to be called by the Service Client/requester (SC). The PG uses the WSDL file of the actual intended web service to generate the PWS.
2- The SC transmits a SOAP request message by using the WSDL of the published PWS.
3- When the request is received by the PWS, the Message Interceptor (MI) sub-component of the FIMS intercepts the request and forwards it to the third sub-component Fault Injection Engine (FIE).
4- The FIE decides the fate of the request by using the Network Emulation and the user script fault model.

5- If the decision is to drop the message, the message is dropped and no further processing takes place.

6- If the decision is otherwise (do nothing with the message, delay, randomly flip bytes of the SOAP body, manipulate the function parameters), the fault is injected (if any) and the message is transmitted back to the MI.

7- The MI transmits the request message to the other corresponding FIS which runs in front of the actual invoked Web Service.

8- The FIS in this situation only needs to forward the received request message to the actual called web service.

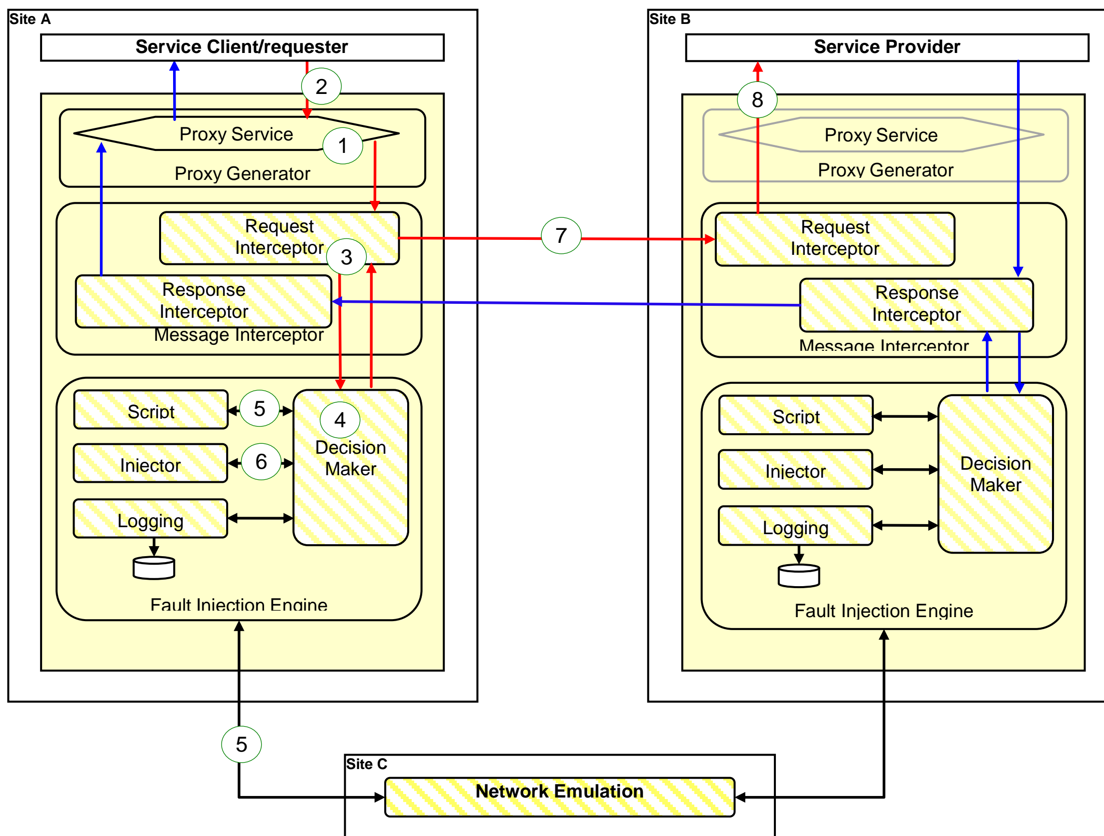


FIGURE 4-5 REQUEST MESSAGE THROUGH NETFIS

The above scenario gives a general view about how NetFIS injects both Communication and interface faults (Network and Software faults). As discussed in Section 3.1.4, although the only tool [31] we are aware of that can inject both classes of faults (communication and

interface faults), it introduces some kind of intrusiveness in terms of modifying the SOAP stack library. WS-FIT needs to implement a set of hooks at the SOAP protocol layers of every machine hosting any Service requester or Service provider being tested. That made it a less general purpose testing tool. Our fault injection tool has been designed in such a way that no modifications to the system under test have been needed. It requires no modifications to the underlying operating system, networking libraries, middleware, or the web service applications under test.

By intercepting messages in this way it is possible to perform parameter perturbation because of complete message entities obtained at this level; it also gives the system under test the sense of running over a Wide Area Network and injecting network faults. Moreover it gives the system under test the sense that there are other − synthetic − applications running at the same time and sharing the networking resources without a perceivable emulation overhead (as will be discussed further in the coming sections).

The response messages from the Service provider to Service requester are processed in a somewhat similar way, with the exception that the faults injected into the messages can be different from the faults injected to the requests by NetFIS in the following way (see Figure 4-6):

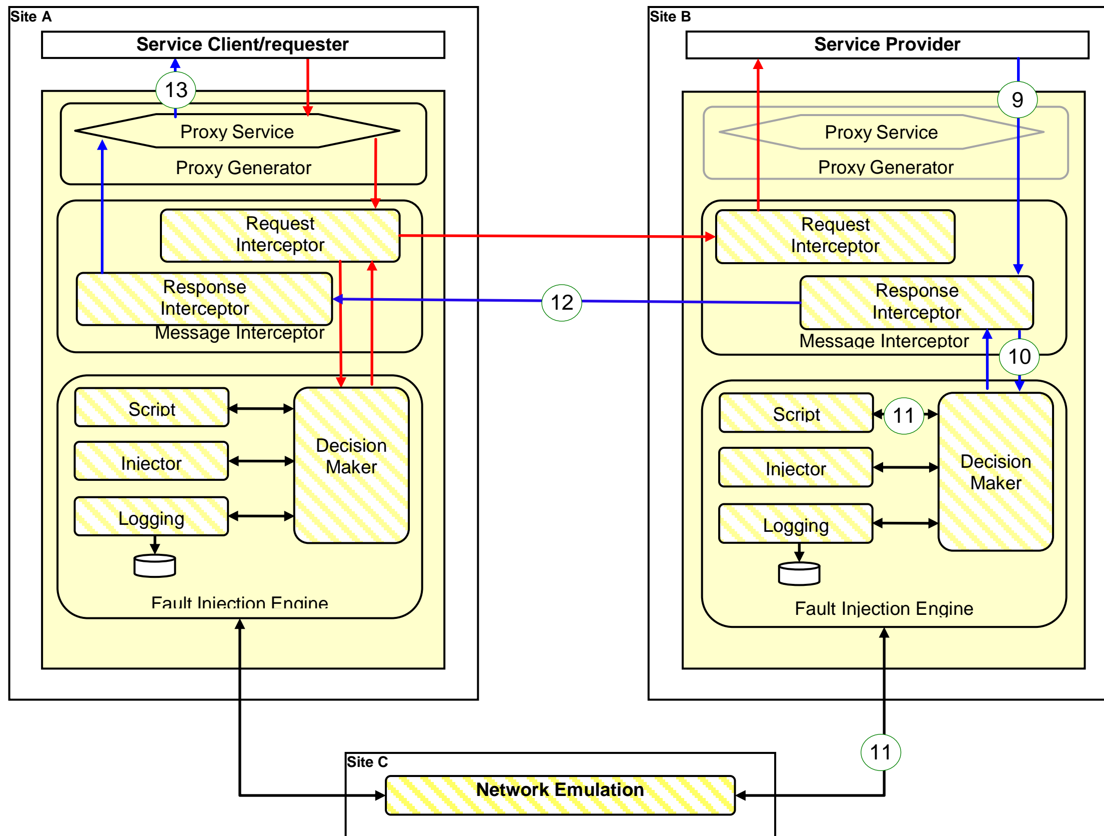


FIGURE 4-6 RESPONSE MESSAGE THROUGH NETFIS

9- Because the request message received by the Web Service comes from the FIMS running in front of the Web Service, by default the response message is transmitted to the same FIMS came from.

10- At FIMS, the received response message is intercepted by the sub-component MI which forwards it to the other sub-component FIE.

11- The FIE provides the network emulation for the received response and injects the appropriate fault (if any), in the same way as was done for the request message at the other FIMS running in front of the Service requester.

12- Afterwards, the response message (if any) is sent back to the FIMS running in front of the Service requester.

13- When the response is received by the FIMS running in front of the Service requester the sub-component MI forwards it to the Service requester.

Our fault injection method has extended the standard Network Level Fault injection in the way it injects Network faults based on the Network emulation provided with the tool. The fault injection tool gives the system under test the sense of running over a Wide Area Network and injecting Network faults such as loss, delay, and corrupt exchanged message. Moreover it gives the system under test the sense that there are other − synthetic − applications running at the same time and sharing networking resources without a perceivable emulation overhead, which will be discussed in the coming sections. In this way the tool provides a run-time environment to test not only systems reliability but also to measure the performance of systems under test.

Although there are other architectures which could provide similar implementation to achieve the required goal, our choice of intercepting the exchanged messages in two FIMs has some advantages in terms of flexibility and performance. We could have opted for a simpler architecture, using only one FIMS to intercept all the exchanged messages between all the system components (Service clients and Service providers). However, intercepting all the system messages through only one FIMS would increase the overheads of the tool and decrease its performance, in particular when a tested system consists of many components exchanging a large number of messages. Also it would be more complex and difficult both to monitor and to provide a clear picture of the online system monitoring. The most important issue is to have two FIMS for each Client and service running in different locations, means adapting the same architecture which has been chosen for emulating a WAN. That is, every Network Emulator Service instance (Node emulator) is responsible for emulating only one node and all the links from that node to other nodes (as discussed in section 4.3 on Network emulation mechanisms).

More details of the three sub-component roles of the FIS implementation are each described in some detail below.

i. ***Proxy Generator (PG)***: its main task is to generate any number of Proxy Web Services (PWSs) needed by a Service requester (Client).

Any Web Service involved in the system under test needs a PG sub-component of the FIMS to generate a proxy Web Service in order to be called by a Client. As mentioned earlier, each Service requester and each Service provider has its FIMS running in front of it. Therefore every FIMS running in front of a Service requester has to generate a Web Service proxy of the actual Web Service that the Service client needs to invoke. In this way all the requests from this Service client to a particular Web Service are received by its corresponding Web Service proxy as shown in Figure 4-5.

In the case that the Service client needs to connect to more than one Web Service, the PG has the ability to generate any number of web service proxies needed. When the Web Service proxies are generated and deployed, all the requests coming from this particular Service client for any particular web service proxy are directed to a separate instance of Message interceptor (MI) thread controlled by the FIMS. In this way all requests going to a particular Web Service can be treated differently such as injecting different faults, providing different configuration for the network emulation, or logging in a specific file.

The only requirement that GP needs to generate a web service proxy is the URL of where the WSDL document published. Every Service client involved in the tested system needs to deploy a FIMS in an application server, and then needs to call this FIMS so as to send the URL of the WSDL of the actual Web Service that needed to be called by the Service client. Then the FIMS passes the WSDL URL to its GP sub-component, which will use it to generate a new Web Service proxy by publishing the same actual WSDL but with a different endpoint URL. The new URL will point to this FIMS. Furthermore, the FIMS will also generate two different interceptors, one for dealing with the request messages, whereas the other deals with response messages. In this way, every Web Service proxy has its own interceptor's instances (for requests and responses). Although − in the case of a number of proxies generated − it would be possible to intercept all the requests by using only one interceptor instance and also intercept all the responses by using only one interceptor, capturing all the messages for every proxy separately has certain advantages in terms of the flexibility of injecting

different faults, and emulating different network links. Above all it enhances the flexibility and clarity of the method.

We have used this design of intercepting the system messages, 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 intercept the outgoing messages and the incoming messages by using a plug-in module in the processing chain in the SOAP Stack. While this would provide a solution for a particular implementation of SOAP, it would not be possible to implement a plug-in module under all implementations, since this facility is not available on some SOAP implementations.

As our system would use WS-Addressing [50] or WS-MessageDelivery [66], it would be possible to redirect the SOAP messages to our fault injector via the protocol stack, effectively removing the requirement for generating Web Service proxies on systems under test, but this would restrict the tools to working on SOAP based SOA that supported these standards and would make it harder to use the fault injector on other middleware systems that are not supporting these standards.

One of our main objectives is not to do any modification to the system under test, which will affect the generality of the tool in a way it would not be used to test other middleware distributed systems. Our Fault Injection Mediator Service is implemented as a standalone fault injection system which can be used by any other distributed systems for not only injecting faults but also providing the impression that they are running over a WAN.

ii. ***Message Interceptor (MI)***: its task is to intercept all messages (incoming and outgoing messages) exchanged between service client/s and service provider/s. The MI acts as a message interceptor to the messages exchanged between the client/s and Web Service/s of the system under test and then forwards them to their final destinations. The ***MI*** is divided into two sub-components (Request interceptor and Response interceptor). All the request messages coming from the Service client for a particular

Web Service proxy are intercepted by the sub component MI. When the first request message for a particular Web Service proxy is received by the MI, the MI generates two instances of interceptors (one instance for intercepting requests and the other one for intercepting response messages). That is, from this point on, all the requests and responses coming in or going out are intercepted by these two interceptors.

Although it would be possible to have only two interceptors for all the web service proxies generated in each FIMS, generating two interceptors for each Web Service proxy has some advantages. These advantages are − in terms of distinguishing between messages going to different Web Services − that it helps in providing a different fault injection service and providing different network emulation, especially for different messages going to different Web Services in different locations of the tested system. We will elaborate on this in the next section.

MI implantation is developed in a way that the task of its interceptors is different, depending on where the messages are coming from and going to as set out in what follows.

Request interceptor tasks as shown in Figure 4-5 ***are:***

1) When the request is received from the Service client the Request interceptor sub-component task is to pass the request message to the FIE sub-component for injecting the appropriate fault (if any).
2) The FIE then passes back the request message (if the fault is not to drop the message) to the Request interceptor, which in turn forwards the message to the other FIMS running in front of the actual called Web Service.
3) Afterwards the Request interceptor of the other FIMS only has the task of forwarding the request message to the actual invoked Web Service.

Whereas Response interceptor tasks as shown in Figure* 4-6 *are:

1) When the response message is received from the actual Web Service by the FIMS running in front of the Web Service, the response interceptor task is to pass on the message to the FIE sub-component for injecting the appropriate fault.
2) Then the FIE passes back the response message (if any) to the response interceptor, which in turn forwards the message to the other FIMS which the request came from.
3) The message is received by the response interceptor sub-component in the latterly-mentioned FIMS, whose task is only to forward the response message to the service client.

In order to implement the proposed scenario above, when the GP generates a Web Service proxy, two interceptors are generated (Request and Response interceptors) and assigned to deal with all the messages coming from or going to from this particular Service client. Whereas, when a request is received for the first time coming from another FIMS, two interceptors are generated and assigned to deal with all the messages coming out from or going in to this particular FIMS. In this way the messages coming out from and going in to any particular Service client and Web Service are separated from other messages.

Therefore this way enhances the simplicity of the architecture and also simplifies how to distinguish between the messages going through the FIMS. Moreover, as mentioned before, different scenarios for each message going in to or coming out from can be treated differently in terms of the types of injected faults and also Network emulation.

iii. ***Fault Injection Engine (FIE)***: this sub-component is the core component of our NetFIS (Network Fault Injection Service) tool; its main task is to inject the proper faults into the intercepted messages based on the Network emulation and on the tester fault model script. It also has to log all the required events so that they can be analysed for evaluating the system under test offline.

Once a SOAP message has been received from either the Request or Response interceptor by the FIE, it must be processed. The FIE is split up into a number of sun-components to aid processing these messages, the details of these components are as follows.

Decision Maker: all the messages received by the FIE are controlled by the sub-component Decision Maker. Decision Maker is a programme that is responsible for taking decisions about whether or not to inject faults, and which faults to inject, and connects to a WAN emulator Service which is responsible for emulating a node and the link of the emulated network.
Decision Maker also connects to the sub-component Message Manipulator in the case of injecting software-specific faults (function parameter value faults). In order to minimize the overhead latency of the fault injection process, the Decision Maker relies on a priority algorithm (as explained in Table 3.3) for injecting the proper fault. That is, whenever the fault that needs to be injected is a drop message, Decision Maker does not have to contact the sub-component Message Manipulator (MM) which is responsible for injecting software-specific faults. Instead it contacts only the sub-component Fault Injector (FI).

Message Manipulator: this sub-component task is to inject software-specific faults (function parameter value faults) into the intercepted messages as appropriate by using the user fault injection configuration file (this will be detailed in the later section on Injecting Software faults). Message Manipulator uses the user fault injection configuration file to determine whether or not software faults are to be injected into the message, which part of the message needs to be manipulated, and what fault should be injected. It then passes all the information to the Decision Maker, which in turn passes commands to a Fault Injector sub-component to inject the fault into the message in the proper chosen location.

Fault Injector: its task is to inject any faults decided on by the Decision Maker. The Decision Maker sends all the information about what fault/s are to be injected, and also

where they are to be injected into the message, in the case of manipulating the function parameters or the return result of the SOAP message.

Logging: this sub-component is responsible for logging all the required events going through the tool so that it can be used to monitor the tool and to collect all the information needed to for the offline analysis. All the original messages (requests and responses) going through the tool are to be logged. Moreover, logging takes place of all messages after faults injected are into them, also recording what kind of fault has been injected.

In order to speed up the logging mechanism XML [21] files are used to store all the required information. That is, the logged information is stored in the memory for only a short period of time, then it is frequently flushed into an XML file. More details are given in the section on fault detection or monitoring.

Whilst the above sub-components are the functional steps followed by the FIMS design, in the actual implementation some sub-components will be involved to perform the injection of each specific fault for efficiency's sake. For example, when the drop message fault needs to be injected only the Decision Maker and the Fault Injector will be involved. Therefore the Message Manipulator will not be involved in this specific fault injection process and needs not to be called by the Decision Maker. This will speed up the processing and reduce the overhead introduced by the tool to the tested system.

4.2.2 Fault injection scenario cases

As presented in Table 3.3 the software faults and network faults that can be injected are based on some rules introduced by the tool. Corresponding to the design, there are four fault injection cases performed by the tool. Therefore the implementation is optimized depending on what faults are to be injected, from the decisions taken by Decision Maker sub-component which is based on the tester configuration.

These paths are performed as mentioned earlier, depending on the fault injection case campaign. The first case campaign is when only Network Faults have to be injected, where the second case campaign is followed when both Network and Software-specific faults have

to be injected. We assume that the two mentioned path campaign cases are the default configuration and NetFIS have been optimized accordingly when both the Network Emulation Service is enabled and the User Fault Model Document is provided. We have chosen this way of choosing which fault to inject because we have implemented the FIE according to how faults occur in the real world, as discussed earlier.

– **First case of injecting faults**

The First case specifies the steps and the path through the code when a message is received and only Network Faults are injected into it. This is shown in Figure 4-7.

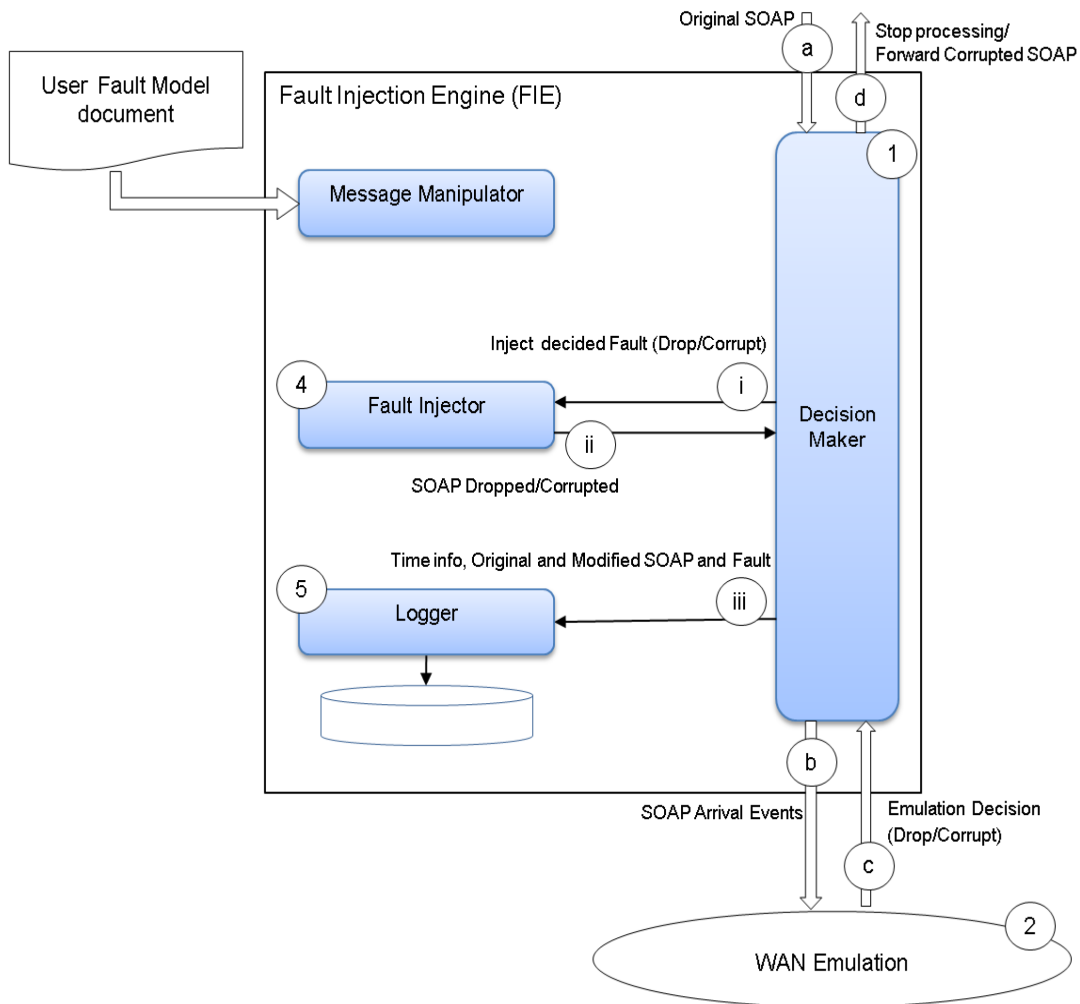


FIGURE 4-7 MESSAGE PROCESSED BY FIE TO INJECT NETWORK FAULTS

1. The extracted SOAP Message (a) is received by Decision Maker (1).
2. The Decision Maker sends a request to the Network Emulation (2) responsible for emulating a WAN of this node (location). The request is for a SOAP message (b) containing the URL destination of the SOAP Message (a).
3. The Network Emulation therefore provides a WAN Emulation for the system running in this node (location) by deciding the fate of SOAP Message (a).
4. Then a SOAP response message (c) is received by the Decision Maker containing the fate of the SOAP message (a) which is in this case can be either DROPPING or RANDOM_DAMAGE.
5. The Decision Maker, in this case, has no reason to contact Message Manipulator, because the message has to be dropped or randomly corrupted based on the Network Emulation Decision (b). Therefore the Decision Maker's decision is to call (i) the sub-component, Fault Injector (4) to drop or damage the SOAP Message (a).
6. When response (ii) is received by the Decision Maker (1) which contains the corrupted SOAP message or in the case of dropping a message, contains a confirmation tells that the message is dropped, the Decision Maker sends the response (d) to the Request Interceptor to stop any further processing of the message (a); or in the case of damaging the message, the damaged SOAP Message (a) is sent back to the Request Interceptor to forward the message to its destination.

Although the above steps are performed when the Network faults are to be injected, namely to drop and corrupt messages, delaying intercepted messages can also be injected, when only a Network Emulation Service is enabled and User Fault Model is disabled. However Delays can be also injected when injecting Software faults, as discussed in Table 3.3. Implementing the tool in such a way provides the test user with flexibility in combining what faults to inject based on Network faults and/or Software faults.

The first case fault injection allows traditional fault injection operations to be implemented, for instance corruption of bytes within a SOAP message, or dropping messages. However our tool provides a Network Emulation Service that has a network background workload

running, which gives the system the sense that it is running over a WAN. This can be used for SOAP protocol stack assessment and to test the application's fault tolerance mechanisms.

- **Second case of injecting faults**

The second case of injecting faults is also implemented by default to reflect the choices detailed in Table 3.3 as shown in Figure 4-8.

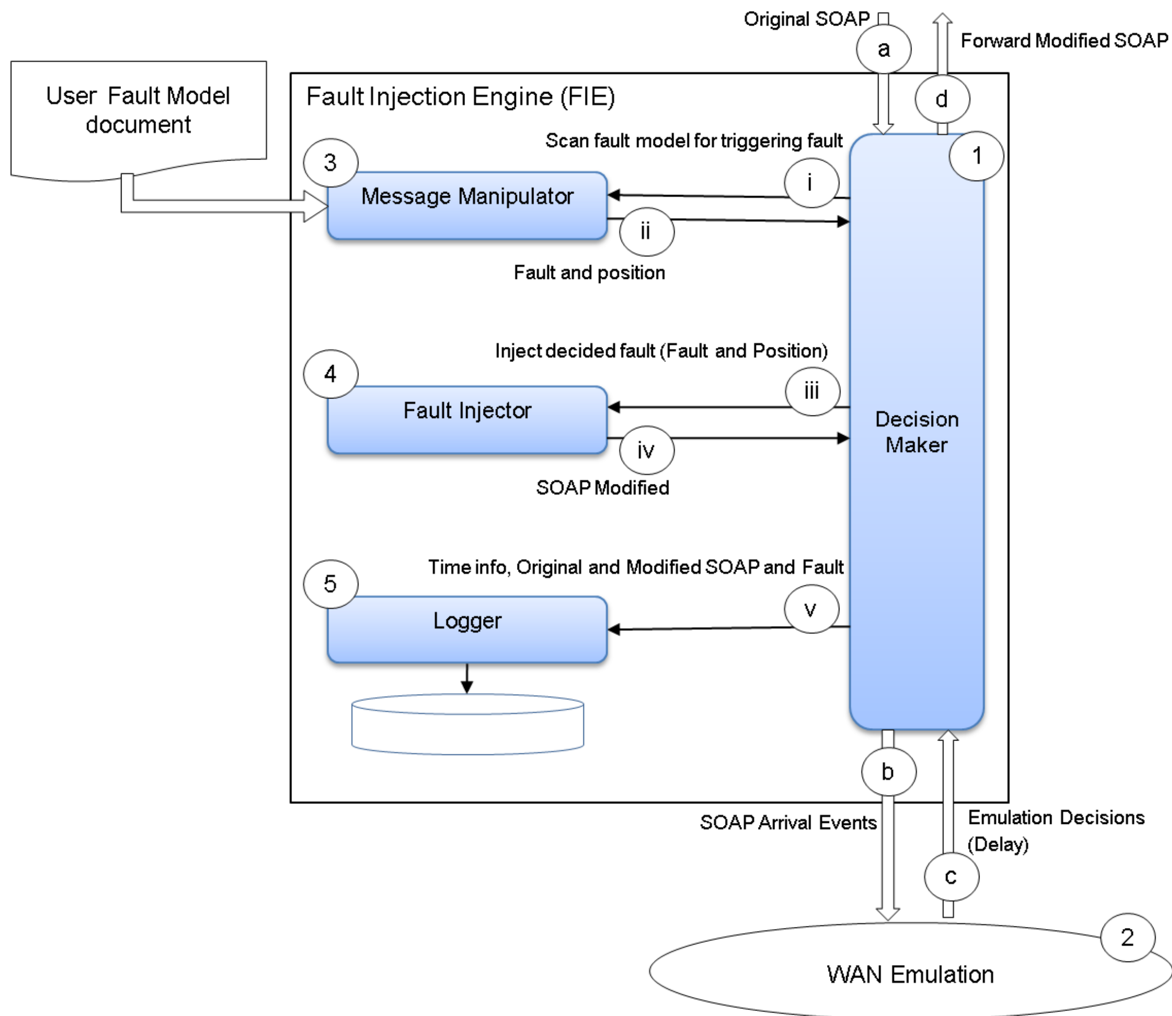


FIGURE 4-8 MESSAGE PROCESSED BY FIE TO INJECT SOFTEWARE FAULTS

1. The extracted SOAP Message (a) is received by the Decision Maker (1).

2. The Decision Maker sends a SOAP request to the Network Emulator Service responsible for emulating a WAN of this node (location), The request is a SOAP message (b) that contains the URL destination of the SOAP Message (a).
3. The Network Emulation Service (2) receives the requested SOAP Message (b). Therefore it provides a WAN Emulation for the system running in this node (location) by deciding the fate of SOAP Message (a) based on the emulated WAN.
4. The response SOAP Message (c) contains in this case the decision DELAY and the Period of the Delay.
5. When Decision Maker receives the response (c) containing the Network Emulation Service Decision, it contacts (i) the sub-component Message Manipulator (3) by sending the SOAP message (a), which is then used by the Message Manipulator (3) 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 a request/response message and message number, and/or if it contains specific RPC parameters. This stage uses a SAX parser implementation and the decision was made to implement it using pattern matching to reduce latency overheads.
6. If the SOAP message does not match the trigger criteria, the data are passed back (ii) to the Decision Maker (1) with a NONE value, which relays that no faults were injected (15).
7. If the SOAP message matches the trigger criteria the data is returned (ii) to the Decision Maker (1). The data is the precise location in the SOAP message where the fault associated with a trigger should be inserted.
8. Once the location in the message has been found and the fault is generated in (3), the Decision Maker (1) sends this information (iii) to the Fault Injector (4) that injects the faults into the SOAP message (a). In addition to modifying the message the required delay period decision is also performed by the Fault Injector (4).
9. After the Fault Injection process has been performed, the manipulated SOAP message is sent back (iv) to the Decision Maker which in turn passes the manipulated SOAP message (d) to the Message Interceptor to forward to its destination.

The other two testing cases which can be performed by the tool are injecting only Network faults or injecting only Software faults. Network faults such as dropping, corrupting and delaying messages, can be injected into the system under test without injecting any other Software faults. That is done by configuring the tool to enable only Network Emulation. The other Fault Injection campaign case is to only configure the tool to inject Software faults by feeding the tool with User Fault Model configuration files only, and disabling the WAN Emulation. How to configure the tools to inject Network faults and Software faults is discussed in the coming sections.

– **Logging**

All the cases write data to a log at two main stages during the testing (v) shown in the above two Figures. The Logger (5) stores the data in a structured XML document that in each transaction is enclosed in an XML element to aid in the analysis of the data.

The first stage is in the initial processing of the SOAP message, when Decision Maker (1) receives the SOAP message (a) from the Message Interceptor. Decision Maker sends the data to the Logger (5), for example timestamp which indicates the time when the message has been received by the Message Interceptor, and the original SOAP message (a). This provides a record of all messages and the timestamps going through the tool for later analysis. This also allows the tester to obtain all the data from any system that is running, when there is no fault to be injected. This allows comparisons to be made with later fault injection campaigns.
The second stage data is written to the log at the end of the Fault Injection process (v) when faults are injected. The Logger (5) logs the modified SOAP message along with the injected fault and a timestamp, however in the case of dropping the message a NONE is logged instead of the message. This logging stage allows the logging of the faulty message and the original message in order to be compared when the experiment result is analyzed offline. The timestamp is used to assess the latency introduced by the FIMS and also is used to calculate the time difference between the timestamp of the request and the timestamp of the response in order to calculate the Round Trip Time (RTT) of messages.

4.3 Network Emulation Mechanism

As discussed in the previous chapter (section WAN Emulation), there are many design choices to implement in the Network emulation. We have decided to implement the Network emulation as standalone Web Service system, as shown in Figure 4-9. A Web Service application will represent each node or link, which will be responsible for emulating that node or link on the target-emulated network. We have chosen this design for the following reasons.

1. These Web Services will be stand-alone services that can work on their own and be running on a different application space, which can be a different physical machine, OS or Web Service middleware.
2. The design will reduce the emulation engine's complexity, as every Web Service Network Emulator will be responsible for emulating only one node of the target Network.
3. This design would produce a clearer more modular design as the nodes number of the target-emulated network is represented in the same fashion by the same number of Network Emulator Web Services.
4. Moreover, this could overcome the scalability problem that could be introduced in the case of designing only one complex network emulation as mentioned in the previous chapter.
5. It will introduce more benefits such as making the Network emulation portable to different distributed applications such as CORBA, GRID, and so on.

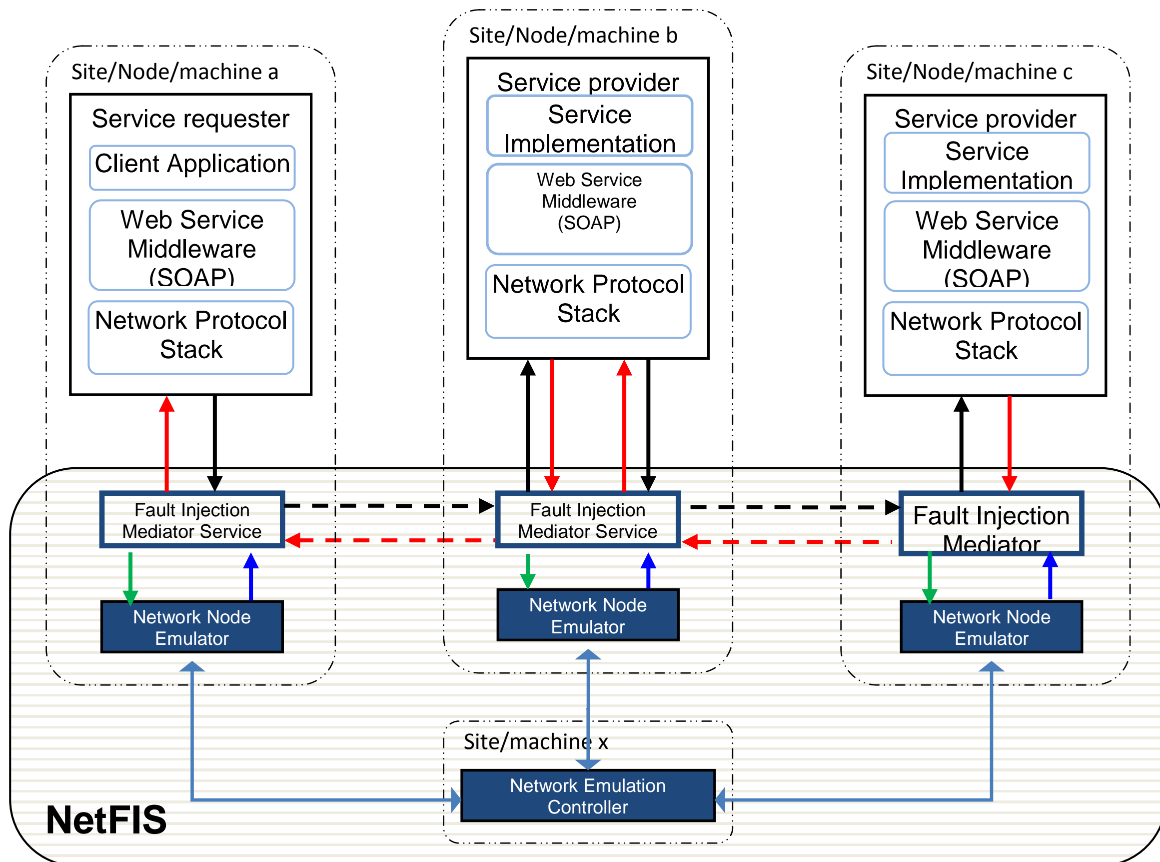


FIGURE 4-9 A GENERAL VIEW OF THE NETWORK NODE EMULATORS PLACED INTO A COMPOSED SERVICE SYSTEM

Figure 4-9 shows a composed Web Service system consisting of three components, namely a Client Service, Web Service 1 and Web Service 2. These system components are distributed over a virtual network consisting of three nodes (a, b and c). In the shaded area at every site/node in the virtual network (Emulated Network), there is a Network Node Emulator Service. There are also Fault Injection Mediator Services for capturing the exchanged messages between the system components and consulting the Network Node Emulator Services to provide WAN emulation for the captured traffic. In addition to the Fault Injection Mediator Services and the Network Node Emulators, there is a single service component, which is the Network Controller Service (at site x). It facilitates the interaction between the neighbouring Network Node Emulators.

By implementing the network emulation in this way, Network emulation will be transparent to the system under test and only the Fault Injection Mediator Service need know of its existence. As each Interception Mediator will be capturing the communication of the

system under test, it also needs to contact the Network Node Emulator to provide the network emulation for a particular node of the emulated network. Therefore, the major feature of the proposed Network Emulation is that it does not require any modification and access to the system under test. That means any Web Service system can obtain the network emulation service, even with the absence of its source code.

The internal design of the network emulation is based on the internal design of other network emulation done for CORBA applications [55] with successful results for testing such applications [48]. The original testing method is for emulating the behaviour of WAN and injecting network faults at Object Request Broker (ORB) level. The messages exchanged between CORBA components are intercepted (using CORBA Interceptors), network emulation is provided to the system, and then network faults are injected.

Our research uses [55] as a starting point for our fault injection testing method for testing Web Service systems for the reasons set out as follows.

1. Web Service applications and CORBA applications have a lot in common. They both exist to add the capability to applications to communicate over a network. The applications can be integrated remotely to form more complex systems through the use of a technology called middleware.
2. In [20] it is reported that communication faults such as message loss, duplication reordering, or corruption have an effect on traditional distributed systems such as CORBA applications. Moreover it has been found that unstable Internet environments and server connections can lead to the unreliability of Web Service applications [36]. Therefore by using this technique, various network faults can be simulated in the system such as dropping, delaying, reordering, and corrupting messages.
3. A fault injection method is needed in order to develop a network run-time environment emulator to estimate the contribution of each contributed service to the overall system and to measure the performance and fault tolerance mechanisms of the system under different circumstances, such as different network traffic load, delays, loss rate, and so on.

4. Extending and modifying the architecture of the previous network emulator will not only provide a network emulation but also a convenient way of injecting network faults and also messages can be manipulated such as modifying the function call parameters and return values inside exchanged SOAP messages.
5. This work can be modified and extended to inject more classes of faults affecting Web Service systems. For example, complete exchanged messages between Web Services can be intercepted and a WAN emulation can be provided, and also the intercepted messages can be manipulated and then forwarded to their destination again without altering the system under test.

However, there are some shortcomings of the CORBA fault injection approach [55]. At the ORB level, communication is typically carried over TCP. This means there should be no lost messages, nor should there be any errors. If such problems arise, ORBs throw COMM CORBA exceptions. Therefore this method does not have the ability to inject faults such as dropping or modifying messages at this level. As a result it does not target any particular elements in the message to inject faults such as function parameters in the case of RPC. The only things this method does are delaying messages, or throwing all kinds of exceptions at this level. This means that the CORBA fault injection method assumption is to inject the mentioned faults to test only the system's ability to deal with such exceptions; whereas it is more logical to inject explicitly the fault and observe its effects on the system. Injecting faults such as dropping and delaying messages can help developers to assign a reasonable time period before the system times out. The problem of the CORBA approach cannot help in how to distinguish between cases in which a message (or its acknowledgement) is simply experiencing a delay in the network from those in which a message has actually been lost. If the time-out interval is made too short, then there is a risk of duplicating messages and reordering in some cases. If the interval is made too long, then the system becomes unresponsive.

All the issues discussed above have been taken into account in order to produce a WAN Emulation for our fault injection method. As discussed in the previous section, the messages are intercepted at application level by using proxies, so at this level, the complete message

entities are captured and any particular part of the messages could be manipulated. In addition, network faults could be injected explicitly (dropped or corrupted messages). And also the proposed solution for choosing a reasonable time-out period should be tackled by testing the system under different real delay rate and drop rate scenarios, then monitoring the system in order to assign the best time-out period that can minimize the risk of confusing normal network delays with message losses.

The internal network emulation subsystem is implemented as two main components, Network Node Emulator Service and Network Controller Service as described in the next two subsections.

4.3.1 Network Controller Service (NCS)

The *NCS* is a Web Service controlling the emulated network which could consist of a set of Network Node Emulators Services (*NNES) that are responsible for* emulating the nodes of the targeted network. Its main task is to start emulating the network by firstly loading the target topology network configuration file and then waiting for the *NNESs* to register themselves. It then acts as a controller of the emulation by issuing Start and Stop commands to the registered NNESs. It can also reconfigure the topology and the parameters for each node. The Controller provides a directory which is like a repository for the NNESs. It provides a directory of Web Services locations. It also provides the locations of the NNESs between themselves. As NCS is implemented as a SOAP-based Web Service, it is communicated with SOAP messages.

All the information required for the emulated network is written in the network topology file as shown in Table 4-1. The topology file is an XML file in a simple format recognized by Java's StAX parser class [67]. It is meant to wrap all the information about the target emulated network in nested tags. The topology file must describe the target network topology accurately. It must state the number of nodes in the network, along with their virtual names. For each node, the topology file must state the arrivals trace file name to be used. If the unit time (bin size) used to create the trace file is not provided in the topology file, then the default value of 1000 ms is used. The default values of 12000, 1.0 and 'true' are used for the node's default SOAP packet size, SOAP packet size modifier and the node's active status respectively. A topology file must also include the routing table for each node.

The topology file must also state each node's outbound links. This is done by stating the number of outbound links, the names of the nodes on the other end of these links and for each link, the following parameters: active status (active or down), bandwidth, buffer size, propagation delay, loss rate and drop rate.

```
<topology>
  <hostsNumber>3</hostsNumber>
  <hostsNames>A,B,C</hostsNames >
  <SOAPPacketSize>1.0< SOAPPacketSize >
  <host>
    <Name>A</Name>
    <traceFile>traceA.atf</traceFile>
    <active>true</active>
    <routes defult=B>
      <route to=B>B </route>
      <route to=C>C </route>
    </routes>
    <links>
          <linksNum>2</linksNum>
     <link>
 <Name>B</name>
 <active>true</active>
 <bufferSize>256000</bufferSize>
 <propagation>2<propagation>
 <bandwidth>4000000</bandwidth>
 <dropRate>0.0001</dropRate>
 <errorRate>0.0001</errorRate>
     </link>
     <link>
                <Name>C</name>
                <active>true</active>
                <bufferSize>256000</bufferSize>
                <propagation>50<propagation>
                <bandwidth>4000000</bandwidth>
                <dropRate>0.0000</dropRate>
                <errorRate>0.0000</errorRate>
     </link>
    <links>
         </host>
        <host>
    <Name>B</Name>
         …
        <host>
        <host>
    <Name>C</Name>
         …
        <host>
< topology >
```

TABLE 4-1 A SAMPLE NETWORK TOPOLOGY WITH SNIPPETS OF THE TOPOLOGY FILE DESCRIBING IT

4.3.2 Network Node Emulator Service (NNES)

NNES has been implemented as a Web Service component responsible for emulating one node of the target emulated network. A NNES is basically a Web Service that provides emulation decisions to the Fault Injection Mediator Service and to other neighbouring NNESs that emulate other nodes of the target network. Based on the configuration parameters for the node it is emulating, NNES calculates the load on the network node resources and gives an estimation of how long it will take to transfer a request between two ends (nodes). It handles synthetic arrival events from its traffic generator sub-component and the messages arrivals from the Interception Mediator and its neighbouring NNESs. The NNES decisions are affected by some major parameters like the buffer size, drop and error rates, bandwidth and delays.

The NNES should have the following "reconfigurable" parameters provided by the NCS as shown in Table 4-1:

- Network trace file: any trace file that shows arrival counts per unit time. It could be self-similar, real trace captured over time, bursty trace or Poisson trace.
- Unit time: the time that an entry in the trace file corresponds to. Smaller unit times require powerful machines and fast languages. A reasonable unit time for Java ranges from 1 second to 0.1 seconds.
- Packet error rate: for example, 1 packet in every 1 million contains an error.
- Bandwidth: WAN link bandwidth. For a WAN cloud, it is reasonable to select the BW of the slowest link in the path. Our WAN is symmetric (up and down rates are equal).
- WAN Propagation Delay: one-way propagation delay between two ends of the WAN cloud (or link). This equals the sum of link delays along a path. Could simply be taken out of Ping traces.
- Buffer size: number of packets we can have waiting in the system before we stop accepting new packets.
- SOAP packet size: translation rate between a SOAP request (or reply) and a real network packet. The default is 1 SOAP transmission equals 1 network packet. This is needed to calculate transmission delays.

The implementation of NNES consists of three sub-modules: the trace file reader, the traffic generator and the core emulation engine.

1. **The core emulation engine:** uses the network trace file reader to supply per unit time network arrival counts. The trace file reader can use any type of traffic trace. It could be a real traffic trace captured from a network. It could be also a synthetic trace generated using Poisson, self-similar or any other arrival process model. The traffic takes arrival counts from the trace file reader and generates synthetic network arrivals during every time bin. The arrival events sent to the core emulator engine also include the packet sizes. The implementation of the traffic generator uses even packet distribution policy within each time bin. It also uses packet size distribution from standard packet traces over the Internet. The job of the core emulation engine is also to handle real arrival events sent by the Interception Mediator. In case the client and server are running on different nodes, the NNES − with help from the Network Controller Service − contacts other NNESs in order to reach an accurate emulation of the network topology given by the Network Controller Service.
2. **Arrivals Trace File Reader:** is another internal component of an emulator. Its sole job is to return the next arrivals count from a trace file. It also supports continuous operation by its ability to loop back whenever the trace file ends. That is if it is configured to do so. It also reports packet target distributions if any are encountered in the trace file.

   Arrivals trace files are only text files containing arrival counts per unit time. Each line in the file contains a number that represents how many packets will arrive during the unit time. Such trace files can be used to gain the power to control the behaviour of the NNES and to decide what WAN effects Web Service applications are subjected to. Each NNES has its own arrivals trace file. This way, different nodes in a topology can have their own traffic traces, making the network emulation more accurate. The trace files also contain packet target distribution. This makes it possible to control the target of the traffic generated at each node. The only action needed to change the current packet target distribution of a node is adding a line in the trace file that shows the new distribution in the form of ‘percentage target’ pairs separated by commas.

```
#### Arrivals Trace File For Node A
####
#### FORMAT: Each line must be one of the following
####    - int showing packet count for this time slot
####    - comment starting with #
####    - destination distribution of the following format
####        destinations = prob dest,prob dest,prob dest,...
####
####        Where
####            prob = probability of this being a destination
####            dest = the destination
####
####
Destinations = 0.34 B, 0.33 C
35
36
34
29
29
29
38
36
35
34
32
28
```

TABLE 4-2 A SIMPLE TRACE FILE

Favourite models can be used to generate traffic traces. Traces captured from the main network used can also be used. Traces can even be created by hand to express independent arrivals. A simple trace file as above can be generated through a simple wizard used with the WAN Emulator for testing CORBA applications [55]. This simple wizard can be used in creating arrival trace files. We have used it to support Poisson and Self-Similar models. You can create self-similar traces using two different algorithms: the Fast Fourier Transform approximation [68] and the Random Mid-point Displacement algorithms [69]. Although these two algorithms model self-similar processes, it is not always guaranteed that the traces created will correspond to what is needed. This is another open research area. The wizard assumes that the user has basic knowledge about the models and their parameters. A discussion of these models and their properties is outside the scope of this document.

3. **Synthetic Traffic Generator:** this is an internal component of an emulator. It generates network arrival events based on the number of arrivals during a unit time. The generator included generates arrival events that are evenly distributed within the unit time (the bin). Other than arrival events, the generator is also responsible for the sizes of the generated packets. The included generator generates packets of sizes that follow the distribution of packet lengths on the Internet. It is based on traces from NASA's Amex Internet Exchange [70] and Sprint IP Backbone [71]. The generator is also responsible for the targets of the generated packets. If a packet target distribution (from the trace file) is set in the generator, it is used to assign targets for the packets.

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

4.4 Injecting Network Faults Mechanism

Injecting network faults is done through our WAN emulator. Although the network faults are decided by the WAN emulator, the injection task is performed by the Fault Injection Mediator Service. Separating the injection mechanism from the WAN emulator makes the tool flexible and portable. The flexibility means it reduces the complexity of the tool, which can make it easier for the tool to be improved and upgraded in the future. Moreover the separation helps each of the two services to be deployed separately; for example the WAN emulator service can be deported and used for a different purpose, such as providing a WAN emulator for other systems in different environments.

The WAN emulator's decisions to inject network faults are based on the emulated network parameters, which are included in the network topology configuration files as shown in Table 4-3. As the tester has to provide this configuration file to the WAN emulator, the tester needs to provide all the network parameters of the emulated network. These network parameters are used by the WAN emulator Service in order to decide the fate of the

simulated network load traffic and the fate of the system's exchanged messages, as detailed in Section 4.2.2.

Table 4-3 shows an example about how the intended network faults can be provided to the WAN emulator as follows:

```
<topology>
  <hostsNumber>3</hostsNumber>
  <hostsNames>A,B,C</hostsNames >
  <SOAPPacketSize>1.0</SOAPPacketSize>
  <host>
           <Name>A</Name>
           <traceFile>traceA.atf</traceFile>
           <active>true</active>
           <routes defult=B>
             <route to=B>B </route>
             <route to=C>C </route>
           </routes>
           <links>
                   <linksNum>2</linksNum>
                   <link>
                      <Name>B</name>
                      <active>true</active>
                      <bufferSize>256000</bufferSize>
                      <propagation>2<propagation>
                      <bandwidth>4000000</bandwidth>
                      <dropRate>0.0001</dropRate>
                      <errorRate>0.0001</errorRate>
                   </link>
                   <link>
                      <Name>C</name>
                      <active>true</active>
                      <bufferSize>256000</bufferSize>
                      <propagation>50<propagation>
                      <bandwidth>4000000</bandwidth>
                      <dropRate>0.0000</dropRate>
                      <errorRate>0.0000</errorRate>
                   </link>
           <links>
         </host>
        <host>
           <Name>C</Name>
             .
             .
        </host>
        <host>
           <Name>C</Name>
             .
             .
        </host>
</topology>
```

TABLE 4-3 A SNIPPIT OF SIMPLE NETWORK TOPOLGY FILE

The table shows that the emulated network consist of three nodes (A, B and C). Each node has its own network parameters, such as how many links it has and a description of the links. For example, node/host A has two links: one links to Node B whereas the other links to Node C. Each link can be described differently, which makes the emulation more accurate as the network links are in the real world.

By using these link parameters, the tester can control how the network faults should be injected. For example, injecting delays into the exchanged system messages is controlled by using the element `<propagation>` which specifies the propagation delays of the emulated link. That is, as shown in Table 4-3, all the messages going from Node A to Node B are delayed by 2 milliseconds. Our design of the WAN emulator does not target a specific message to be delayed, because our objective is based on emulating the behaviour of a WAN. Moreover, injecting loss and corruption into the exchanged messages is done by using the other two elements, `<dropRate> and <errorRate>` respectively.

However, injecting loss messages can be done by using different configurations. For example the link-generated virtual network traffic load can be configured in a way which may lead to the dropping of packets and system messages, caused by an overflow of the receiver buffer (`<bufferSize>`). We will not develop this further at this point, because our main concern here is to show how faults can be injected directly, whereas the network configuration is elaborated in the next chapter.

4.5 Injecting Software Faults Mechanism

As noted in the discussion of the location of injecting faults discussed in Section 3.3.1 (Fault injection location), our fault injection method is able to inject software faults at application level. We have moved the fault injection location away from the Network interface and positioned it at the application level by using our Fault Injection Mediator Service between the Service requester/s and the Service provider/s participating in a Web Service system. Therefore all exchanged messages (requests and responses) will be captured. Messages captured by the Fault Injection Mediator Service are complete entities, so they can be manipulated, modified and injected with faults. In addition messages at this level are

captured as XML documents; therefore it can be easy to target any part of the message and manipulate it, such as targeting individual Remote Procedure Call (RPC) parameters in the message and modifying them to simulate a large number of software faults.

Before being able to inject value faults, we need to obtain some definitions of the Web Service operations, parameters, data types and domains. As mentioned earlier, a Web Service interface is described as a WSDL file. The WSDL file explicitly defines the messages to be exchanged with clients. Therefore, we use the published WSDL document to decompose the service interface into method calls with their associated messages and within the messages, to identify specific parameters.

Thus by using the WSDL document, the information about the required structure for each message is obtained. As the WSDL contains information about the operations, associated messages, parameters (including return values) and the associated data types, this information will be composed into groups of information. Each group of information could present all the information about one particular operation included in the WSDL interface required to construct a fault injection trigger. For example, each group contains nodes representing the operation name, message request/response, parameter names and types of each parameter. Although triggering faults can be constructed for any node in the group, our fault injection system is primarily concerned only with manipulating parameters in RPC messages.

Based on the information obtained from the groups, a set of valid and invalid Web Services call parameters and return values can be injected during the run-time. The value fault model is based on combinations of exceptional and acceptable input values of function call parameters, based on the data types of each parameter, as shown in Table 3-1.

Our tool uses what is called a Value Fault Model (VFM) document which can be produced manually or by our simple attached application. The information required for producing VFM is obtained from the WSDL document. The VFM produced is based on the discussion

above about grouping the operation name, message request/response, parameter names and types of each parameter, as shown in
Table 4-4.

```
<faultModel>
  <operation>
       <input>
         <message>
              <parameter>
                     <name>    </name>
                     <type>    </type>
                     <injectionRate> <injectionRate>
                     <domain> </domain>
              </parameter>
              <parameter>
                     <name>    </name>
                     <injectionRate> <injectionRate>
                     <type>    </type>
                     <domain> </domain>
              </parameter>
              .
              .
         </message>
       </input>
       <output>
       .
       .
       </output>
  </operation>
  .
  .
<faultModel>
```

TABLE 4-4 VALUE FAULT MODEL FILE PRODUCED, BASED ON WSDL AND PARAMETER VALUES MUTATION ALGORITHMS

As shown in the simple form of a Value Fault Model (VFM) file, as in
Table 4-4, the information included is based on the WSDL description and the required injected faults provided.

This information is composed into <operation> tags (groups) representing the wsdl:operation element in WSDL, which also equivalent to a method in a system.

Furthermore, each <operation> contains both <input> and <output> tags which in turn represent both a wsdl:input element and a wsdl:output element for referring to the request

and response messages in WSDL description. Lastly <message> tags representing wsdl:message elements in the WSDL <message> tag contain all the information needed to inject the required faults. It contains <parameter> tags for every function call parameter or return result involved in an operation (method) call. The <parameter> tag consists of five tags required for generating value mutation. The first tag is <name> which is equivalent to the name attribute of the wsdl:part element in WSDL, which represents the name of the parameter or return result, so as to distinguish between different parameters of the same <operation>. The second tag is <type> which is equivalent to the type attribute of the wsdl:part element in WSDL, which will be used by the algorithms to determine the value of the injected fault. The third tag is <injectionRate> which indicates the overall percentage for the specified fault to be injected into the exchanged messages. The final tag is <domain> which is used to specify the allowed domain range for a parameter or return numbers.

However, the valid domain of the parameter or return result cannot be extracted from the WSDL description. Information on the valid domains for each parameter (including for parameters based on complex data types, which are decomposed in a set of individual parameters) should be provided by the user when producing the VFM file.

Based on the information obtained from VFM and on

Table 3-2, the sub-component Message Manipulator generates faults of the parameters and the return results as appropriate. The message Manipulator relies on a set of logarithms for each type of parameter so as to generate a list of all possible mutations that can be performed. A number of test values will be generated for each data type used in the captured exchanged SOAP messages. The generated mutations consist of a set of invalid and valid values based on [64]. However the generated value mutations are neither intended to be exhaustive nor complete, but are rather to demonstrate our tool implementation.

4.6 Failure Detection Mechanism

As discussed in Section 3.3.8 (System monitoring) the failure modes of Web Service systems should be monitored in order to detect any problems in the system under test caused by the injected faults. As in Section 3.3.8, the failure modes described here are intended to demonstrate NetFIS rather than be an exhaustive model that can be applied to any Web Service system.

Our failure detection mechanism uses a set of high-level failure modes [34] as a starting point for monitoring the target system . These classify how a Web Service can fail. The classifications are outlined below.

Our failure detection mechanism is built upon a set of classified failure modes. These classifications indicate how a Web Service can fail:

1) Crash of a service instance/hosting environment;
2) Hang of a service;
3) Corruption of data coming into the system;
4) Corruption of data coming out of the system;
5) Duplication of messages;
6) Omission of messages;
7) Delay of messages.

Listing failure modes can help in both the design of the fault injection campaign and the monitoring of the outcomes of our fault injection testing method.

The effect of each of these failure modes will depend on the capacity of the fault tolerance mechanisms of the system to detect them and prevent the system from deviating from its specified behaviour.

Due to the many problems detailed in Section 3.3.2 (System monitoring) in detecting these failure modes, we rely on a logging mechanism implemented in the NetFIS tool. We have implemented a simplified logging mechanism to monitor and detect failures, based on two observable detected outcomes from our fault injection method, as follows: 1) Detecting

exception, 2) No effect.

Based on the decision explained in Section 3.3.2, the logging sub-component is placed in the FIMS. Placing the logging sub-component in the FIMS will enable the tool to monitor all the exchanged messages going through the tool (request, response, fault messages).

The logging sub-component is written in Java language. All the logged information is written into an XML file. The logged information in the XML file will be extracted and analyzed by using SAX parser Java class, so as to speed up the analyzing process.


```
▼<!--
  all elements here are explicitly in the HTML namespace
 -->
▼<logging>
 ▼<message>
  ▼<Request>
     <Date>Fri Jul 09 17:21:03 BST 2010</Date>
     <TimeInMill>1,278,692,463,235.699</TimeInMill>
     <SOAPLength>82</SOAPLength>
   ▼<SOAPText>
       <ns2:getAvailableSimMethods xmlns:ns2="http://webservices.calibayes.ncl.ac.uk/" />
     </SOAPText>
     <InjectedFault>None</InjectedFault>
   </Request>
  ▼<Response>
     <Date>Fri Jul 09 17:21:03 BST 2010</Date>
     <TimeInMill>1,278,692,463,547.776</TimeInMill>
     <SOAPLength>307</SOAPLength>
   ▼<SOAPText>
       <ns2:getAvailableSimMethodsResponse xmlns:ns2="http://webservices.calibayes.ncl.ac.uk/"><methods>fern-stochastic</methods><methods>copasi-
       stochastic</methods><methods>copasi-deterministic</methods><methods>copasi-hybrid_rk</methods><methods>copasi-hybrid_lsoda</methods>
       </ns2:getAvailableSimMethodsResponse>
     </SOAPText>
     <InjectedFault>None</InjectedFault>
     <DurationInMill>291.093</DurationInMill>
     <ResponseReceived>1</ResponseReceived>
   </Response>
  </message>
 ▼<message>
  ▼<Request>
     <Date>Fri Jul 09 17:21:03 BST 2010</Date>
     <TimeInMill>1,278,692,463,702.295</TimeInMill>
     <SOAPLength>82</SOAPLength>
   ▼<SOAPText>
       <ns2:getAvailableSimMethods xmlns:ns2="http://webservices.calibayes.ncl.ac.uk/" />
     </SOAPText>
     <InjectedFault>None</InjectedFault>
   </Request>
  ▼<Response>
     <Date>Fri Jul 09 17:21:03 BST 2010</Date>
     <TimeInMill>1,278,692,463,764.164</TimeInMill>
     <SOAPLength>307</SOAPLength>
   ▼<SOAPText>
       <ns2:getAvailableSimMethodsResponse xmlns:ns2="http://webservices.calibayes.ncl.ac.uk/"><methods>fern-stochastic</methods><methods>copasi-
       stochastic</methods><methods>copasi-deterministic</methods><methods>copasi-hybrid_rk</methods><methods>copasi-hybrid_lsoda</methods>
       </ns2:getAvailableSimMethodsResponse>
     </SOAPText>
     <InjectedFault>None</InjectedFault>
     <DurationInMill>62.293</DurationInMill>
     <ResponseReceived>1</ResponseReceived>
   </Response>
  </message>
   :
   <closingFileDate>Fri Jul 09 17:37:46 BST 2010</closingFileDate>
   <AverageResponseTime>46,734.585</AverageResponseTime>
   <TotalRequest>1000</TotalRequest>
   <TotalResponse>1000</TotalResponse>
   <MaxResponseTime>291.093</MaxResponseTime>
   <MinResponseTime>1.000</MinResponseTime>
 </logging>
```


TABLE 4-5 A SIMPLE EXAMPLE OF A LOGGING FILE

Table 4-5 shows a snippet of the logged data during an experiment. The data logged into the XML format file contains all the messages going through the system, so that they can be analyzed offline to detect defects and failures in the system under test. The logging sub-component is able to log all the information about these messages such as the time of arrival, the length of the SOAP message, the original message, the injected fault, the faulty message, and the duration taken to receive the response message, as shown in Table 4-5. This information can be easily used to analyze the results of the testing experiments.

4.7 Conclusions

This chapter has detailed the implementation of the NetFIS method proposed in Chapter 3 which has been applied to Web Service applications, in particular applications using SOAP as a middleware protocol.

This chapter elaborates a description of how the message-based injection mechanisms are performed. The location chosen for injecting faults is detailed. As noted in the discussion of the location of injecting faults discussed in Section 3.3.1 (Fault injection location) our fault injection method is based on injecting Network faults and also software faults at application level. We have moved the fault injection location away from the Network interface and positioned it at the application level by using Web Service proxies between the Service requester/s and the Service provider/s participating in a Web Service system. By using this way of intercepting messages, not only is the generality enhanced but also the non-intrusiveness is improved in the sense that it does not require any modifications to the middleware or to the hosting operating system and of course to the Web Service system under test.

The WAN emulation is achieved by implementing a WAN Emulator Web Service system. The WAN Emulator service is implemented with a collection of Web Service applications (composed service) in such a way that each Web Service application is responsible for emulating one node of the target emulated network. Whereas only one Web Service application is implemented as a coordinator whose job is to control the WAN emulator service. The WAN emulator enables the tester to control the virtual network environment

and its parameters by using network configuration files so as to test the system under different circumstances, such as different dropping rate, error rate, delay, and so on.

The injection of communication faults is based on our WAN emulator Service. The communication faults are decided by the WAN emulator, whereas the injection task is performed by the Fault Injection Mediator Service (whose main responsibility is intercepting the system messages). Separating the injection mechanism from the WAN emulator makes the tool flexible and portable.

Injecting interface faults is effected through the Fault Injection Mediator Service and based on the Web Service interface, document WSDL. A system fault model can be created by importing WSDL for the Web Services making up the system and mapping certain WSDL elements to certain elements within the fault model, based on their type. Following this, a fault model can be generated and used to create triggers for injecting faults into the system.

5 Chapter 5 - Case studies

This chapter describes main case studies which demonstrate the use of the NetFIS tool. The case studies is conducted to explain the use of the NetFIS operations and the performance of the tool. It demonstrates the operation of the Fault Injection Mechanism in terms of how the messages are intercepted and how the faults are injected, so as to measure the tool overhead introduced to the system under test. It also demonstrates how the tool is configured. The case studies also demonstrate how NetFIS is used to assess a fault tolerance mechanism applied to a running Web Service application. The Web Service application has been tested under different scenarios using the NetFIS tool.

5.1 Testing a real web service system

In this section we describe an experiment [72] that injects a number of network-related faults (delaying, dropping and randomly corrupting SOAP messages) into a Bioinformatics Web Service [73]. We deployed the WS-Mediator [74] at the client site to invoke three identical Bioinformatics Web Services [73] simultaneously via the NetFIS. The performance and the fault tolerance protocols of the system under test have been examined. Moreover the overheads introduced to the system by using our tool are also measured. The results obtained from logging files are analyzed and discussed. The setup of the experiment is explained in the forthcoming sections.

5.1.1 Setting up the tool

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

The Topology file is a simple configuration xml file that describes the target network topology, listing the nodes in the network together with their configuration. In addition, a trace file also must be provided for each node that describes its traffic load. This shows packet counts per unit time and can either be created by hand, captured from real traffic traces or produced using network traffic modelling algorithms. Following this, the *NCS*, which is itself a Web Service, is started up. The *NCS* is used by *NESs* to provide node configuration parameters and locations of neighbouring *NESs*. Each node of the emulated network is represented by one *FIS* and one *NES*.

Each *FIS* at the client side needs to be provided with an xml file containing the URL(s) of the Web Service(s) under test. The client needs to call this in order to generate a Web Service proxy, which will be invoked by the client instead of the actual Web Service under test. The xml file also contains the URL of the *NES* emulating the same node.

The tool does not require any modifications to the system under test, unless the only job for the client is to start calling the proxy service generated by the *FIS* instead of calling the actual Web Service.

5.1.2 Experiment setup

The topology of the target network that we emulated is a four-node network setup, as shown in Figure 5-1. In the experiments various types of faults were injected into the emulated network. The Network Emulation Service is enabled to generate synthetic traffic through the network. For a real application deployed on a WAN, there is a significant variation in performance due to other traffic occupying the network resources. NetFIS supports various simulated traffic models including, but not limited to, self-similar, random, constant and even replayed, previously-captured, traffic traces. Since studies of network traffic suggest that it is self-similar in nature [75], we chose to emulate continuous self-similar traffic in our network. The mean packet rate is 30 packets/second on each link, and the self-similarity value is 0.8. The packet size distribution follows measurements taken from Internet backbones [71]. The link utilization varies based on the generated packet size and the link configuration.

5.1.3 Network configuration:

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

i). LAN Configuration: the LAN was used without deploying NetFIS to test the actual performance of the system. The client issuing the requests was loaded onto machine A in Figure 5-2. The three services participating in the test are run on machines B, C and D respectively.

ii). Fast WAN Configuration: The propagation delays are fixed at 2ms, which is typical of inter-city links within the UK. The bandwidth of each link is 4mb/s. The average utilization of each link given this bandwidth, and the simulated traffic described in previous section, ranges between 10% and 20%.

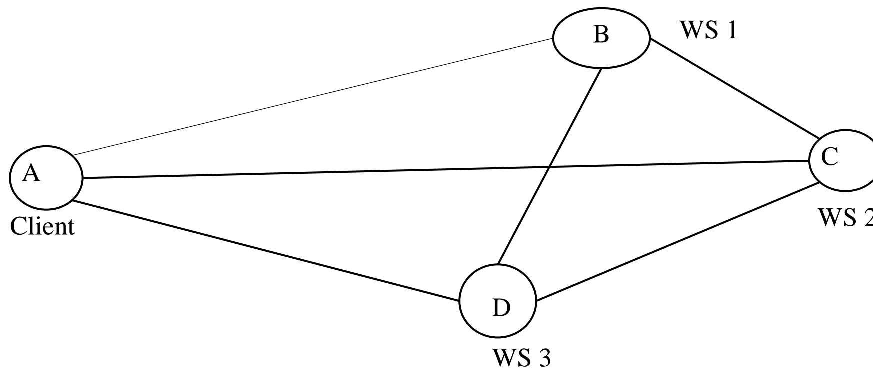


FIGURE 5-1 SIMPLE NETWORK TOPOLOGY

iii). Slow WAN configuration: This configuration represents the other extreme. All services are located in distant geographical locations and connected by slow links. The propagation delays are fixed at 50ms which is typical of distant locations and international links (e.g. between Newcastle, England and Tripoli, Libya). The bandwidth of each link is 512Kb/s. The average utilization of each link, given this bandwidth and the traffic, ranges between 20% and 40%.

iv). Heterogeneous WAN configuration: this configuration represents a case somewhere between the two extremes. One of the services was placed in a far away

location (connected by slow WAN links) while the other servers and the client were closer to each other (connected by fast WAN links). The links and loads used here are similar to those used for the slow and fast WAN configurations.

5.1.4 Client configuration:

We have developed a special client application implementing several test cases corresponding to the fault injection configurations applied during the experiment. The client application is implemented on the WS-Mediator framework [74] as shown in Figure 5-2, and utilizes the built-in fault tolerance and logging mechanisms of the framework. The WS-Mediator claims to offer comprehensive off-the-shelf fault tolerance mechanisms to cope with various kinds of typical Web Service application scenario. It also includes a monitoring mechanism to benchmark a collection of candidate Web Services that would be used during service composition and generate their dependability metadata for dynamic composition reasoning. The framework allows the client to submit a number of candidate Web Services for service composition and define a reconfiguration policy to specify how to make use of the candidate Web Services, and thus to reduce the development cost of a dependable client application.

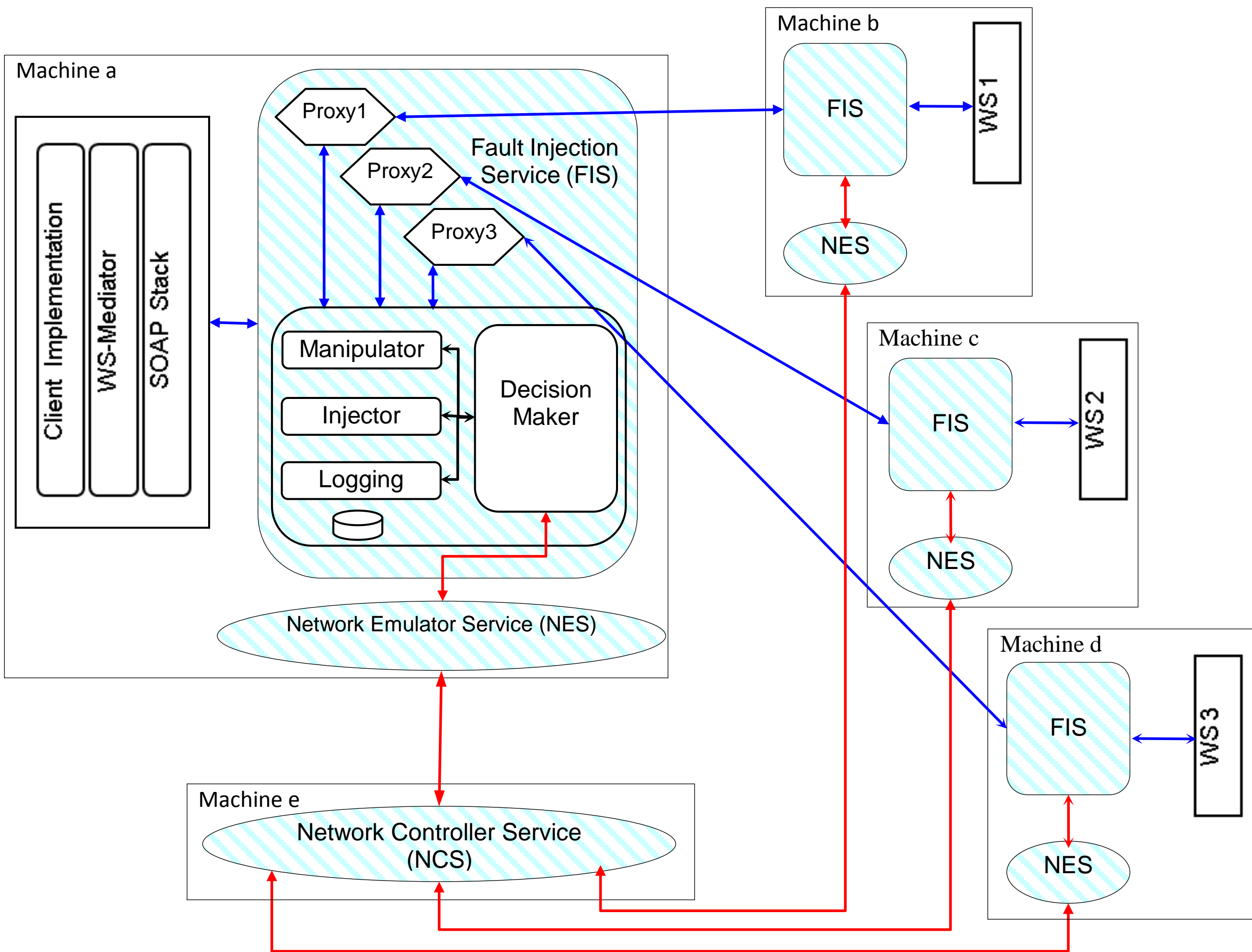


FIGURE 5-2 A SYSTEM BEING TESTED BY THE TOOL

In our client application, we chose to use the N-version programming mechanism offered by the WS-Mediator to invoke the NetFIS proxies simultaneously and choose the first valid response to service a client's request. During the invocations, all request and response messages are logged using the built-in monitoring mechanism of the WS-Mediator. The complexity and processing overheads of the WS-Mediator have been minimized with these settings. It is worth noting that a classic N-version programming approach normally requires voting for result validation. However in Web Service applications, although similar Web Services may return semantically identical responses, they are not usually exact matches. The WS-Mediator framework allows policy-based response mapping; however since the NetFIS tool injects random faults into the SOAP messages especially with random timing,

the response mapping and voting mechanisms are of little value in our test cases. Nevertheless, the late responses are also logged for further analysis.

Besides the fault tolerance mechanisms deployed in the client application, the functionality of the client is fairly simple. It invokes the three replicated Web Services repeatedly, with or without the NetFIS. The number of invocations and the delay interval between invocations can be configured dynamically.

5.1.5 Experimental Results

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

Section 1: the NetFIS emulates different types of network simulating varied traffic load. The detailed settings are shown in

Table 5-1. A preliminary test was carried out to check the network conditions and the Web Service prior to the other test cases. The client invoked the three Web Services directly 1,000 times (interval: 1,000ms) without the NetFIS. The overall maximum, minimum, and average round trip response time (RTT) received by the client application are 102ms, 8ms and 57ms respectively.

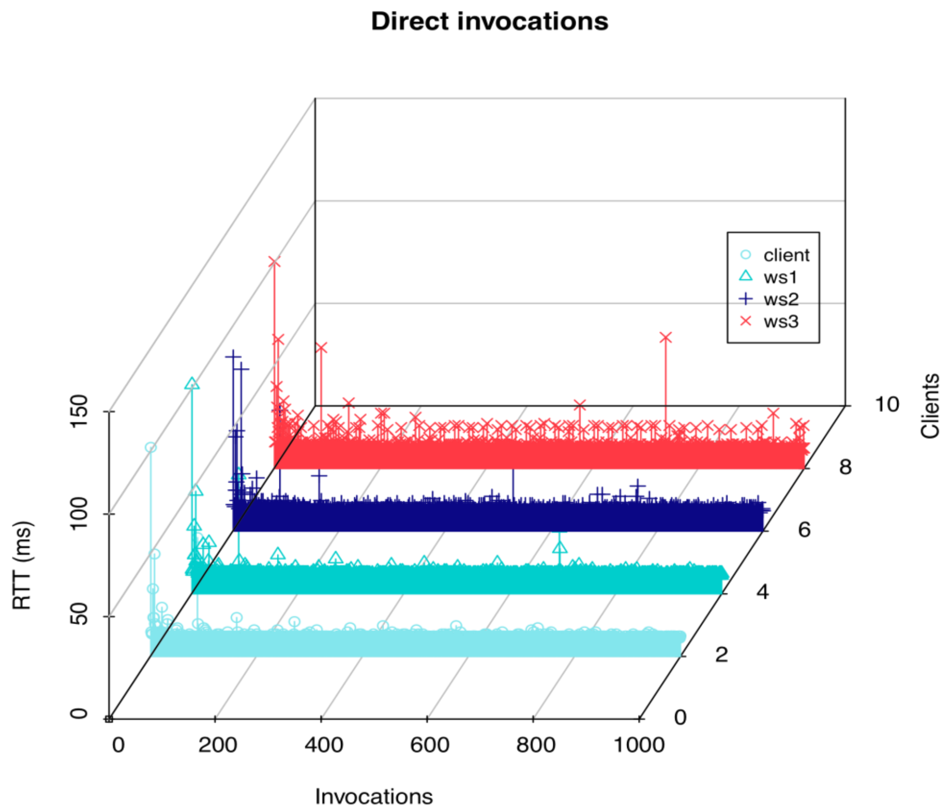


FIGURE 5-3 CLIENT INVOCATION RTT

Figure 5-3 shows the RTT for the three Web Services logged by the WS-Mediator at the client application. It is very interesting to see the three Web Services had much longer RTT at the very beginning of the test, suggesting the RTT could have been optimized by some kind of caching mechanism employed in the Web Services. It is also worth noting that although the three replicated Web Services have identical hardware, operating systems, middleware, and so on, WS3 constantly had longer delays than WS1 and WS2. However, the RTT variations of a Web Service and between different Web Services are clearly insignificant compared with the delays to be injected by the NetFIS, and therefore can be safely ignored. The average RTT of WS1, WS2 and WS3 are 10ms, 11ms and 12ms.

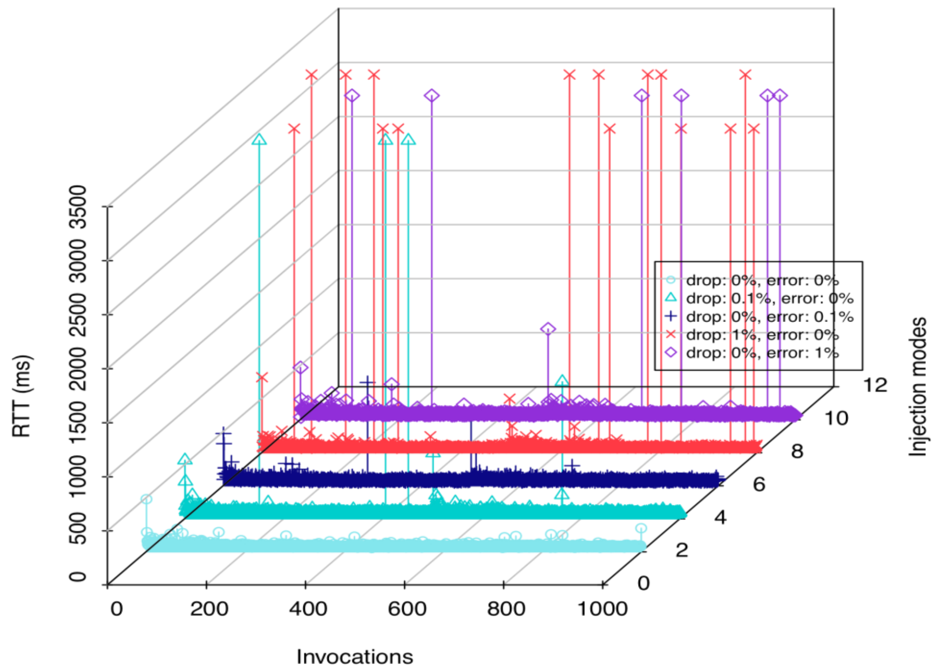
The average client RTT was slightly smaller than 10ms, because it always uses the quickest response from the three Web Services.

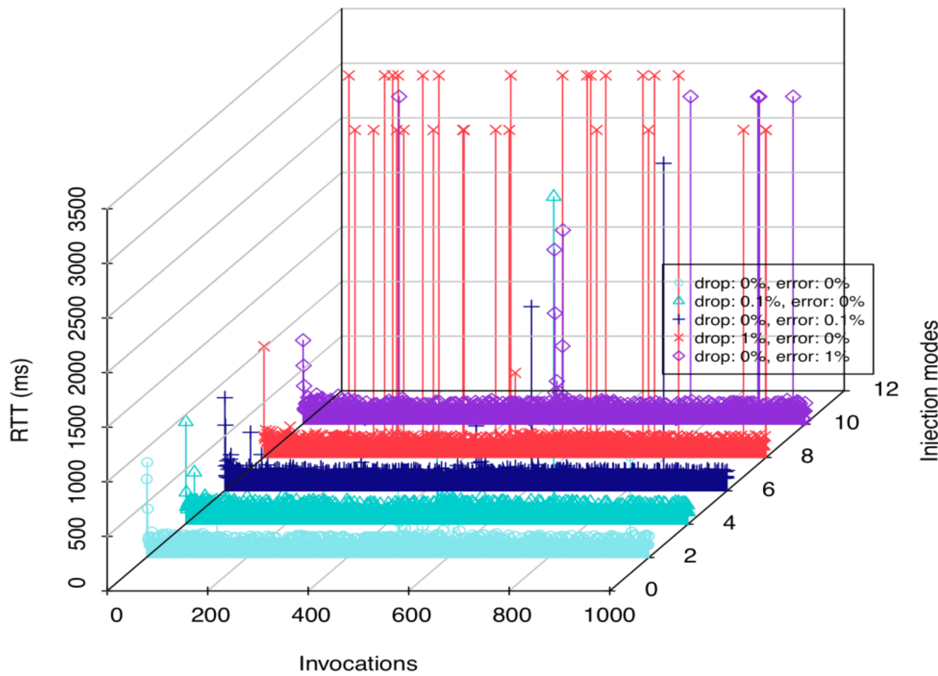
The preliminary test results provided the benchmarking information on the physical LAN and Web Services involved in the experiment. Then the NetFIS were added between the client and the Web Services, and the client made 1,000 invocations in each setting.

Network	Bandwidth (Mb/s)	Response time		
		Max, ms	***Min, ms***	***Average, Ms***
LAN	N/A	102	8	57
Fast WAN	4000	488	35	59
Slow WAN	512	698	110	190
Hetero.WAN	Fast and Slow	870	99	104

TABLE 5-1 RESPONSE-TIME OVERHEAD

Table 5-1 clearly indicate the effectiveness of emulating the three different networks between the system components. The average RTT − without injecting drop and error faults − shows when emulating the Fast WAN is 59ms, where the average RTT without using our tool in LAN is 57ms. That means the overhead delay introduced by the NetFIS to the system under test is clearly insignificant. However the differences in the average response time between the Fast WAN and the Slow WAN are indeed considerable. This is related to the configurations of the two emulated networks, specifically the propagation delays and the bandwidths which in the Fast WAN are 2ms, 4mb/s and in the Slow WAN are 50ms, 512Kb/s respectively. When considering Heterogeneous WAN, the average response time is almost between the average response times of the Fast and Slow WANs. That is due to the fact that the Heterogeneous WAN is configured of a combination of the other two WANs (Fast and Slow).


WS1: fast network
RTT (ms)
0
500
1000
1500
2000
2500
3000
3500
0
200
400
600
800
1000
Invocations
0
2
4
6
8
10
12
Injection modes
drop: 0%, error: 0%
drop: 0.1%, error: 0%
drop: 0%, error: 0.1%
drop: 1%, error: 0%
drop: 0%, error: 1%


WS1: heterogeneous network
RTT (ms)
0
500
1000
1500
2000
2500
3000
3500
0
200
400
600
800
1000
Invocations
0
2
4
6
8
10
12
Injection modes
drop: 0%, error: 0%
drop: 0.1%, error: 0%
drop: 0%, error: 0.1%
drop: 1%, error: 0%
drop: 0%, error: 1%

WS1: slow network

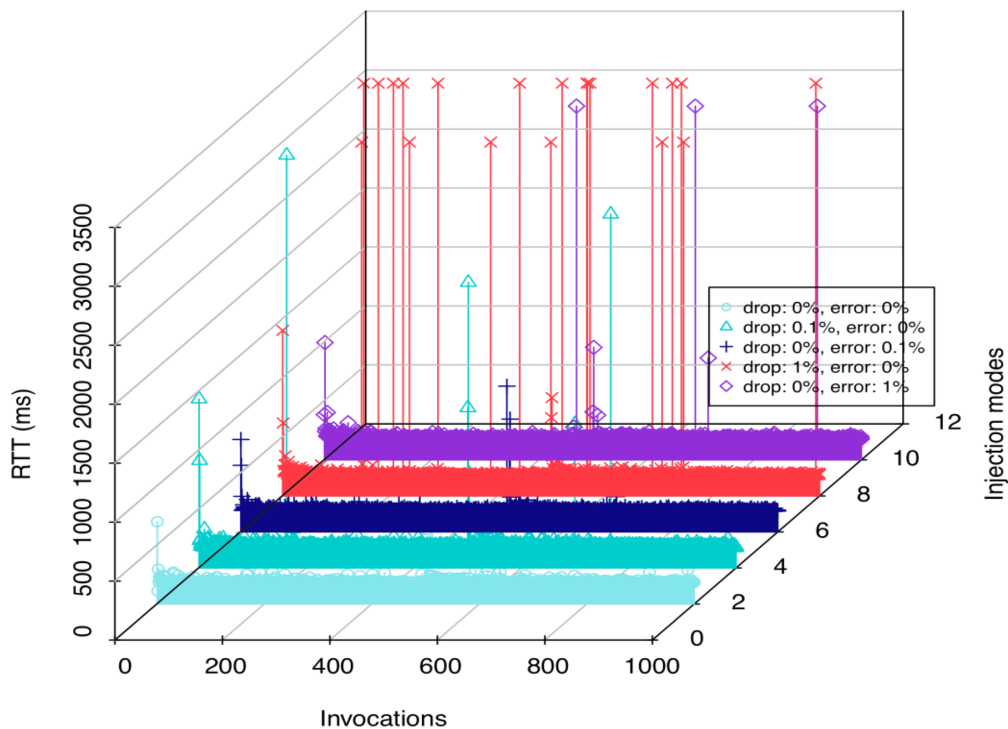


FIGURE 5-4 RTT OF THE TEST CASES (WEB SERVICE 1)

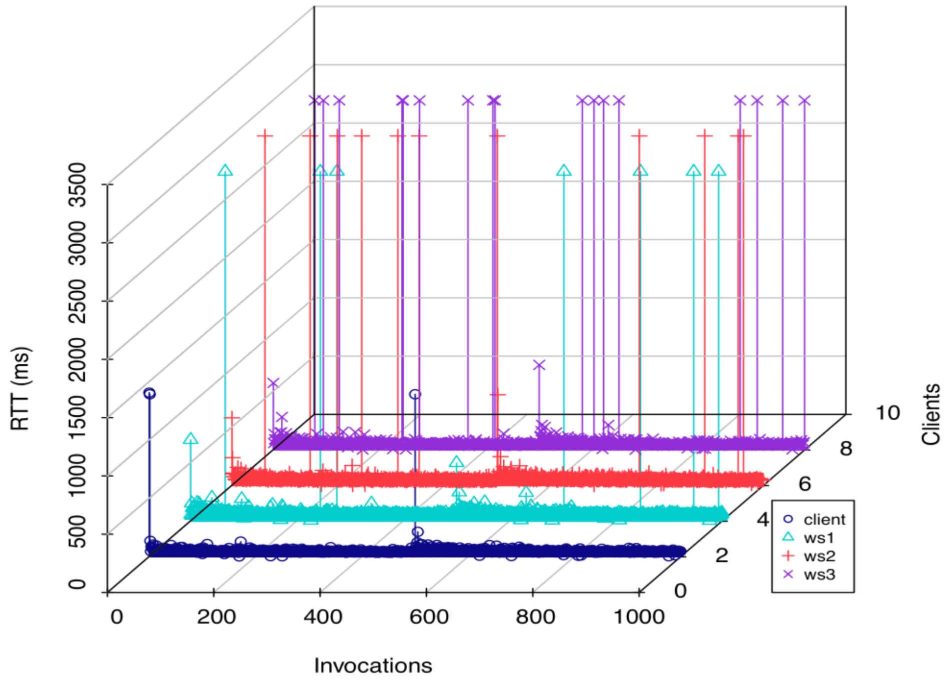
The Figure 5-4 shows the RTT of WS1 (monitored at the WS-Mediator client) at different drop and error rates and network conditions. The ‘injection modes’ axis represents the invocation RTT of each injection mode shown in the Figure legend. The overall average RTT of the fast network is much smaller than that of the other two network conditions. The Figure clearly shows greater RTT variations of the heterogeneous network than the slow network. The time-out value has been regulated to 3,000ms in the Figure to make the plots more readable.

Network Emulated	Injected Drop rate		Injected Error rate	
	Target %	***Achieved (total messages)***	***Target %***	***Achieved (total messages)***
Fast WAN	0.1	1	0.1	1
	1	9	1	10
Slow WAN	0.1	1	0.1	1
	1	10	1	10
Hetero. WAN	0.1	1	0.1	1
	1	9	1	10

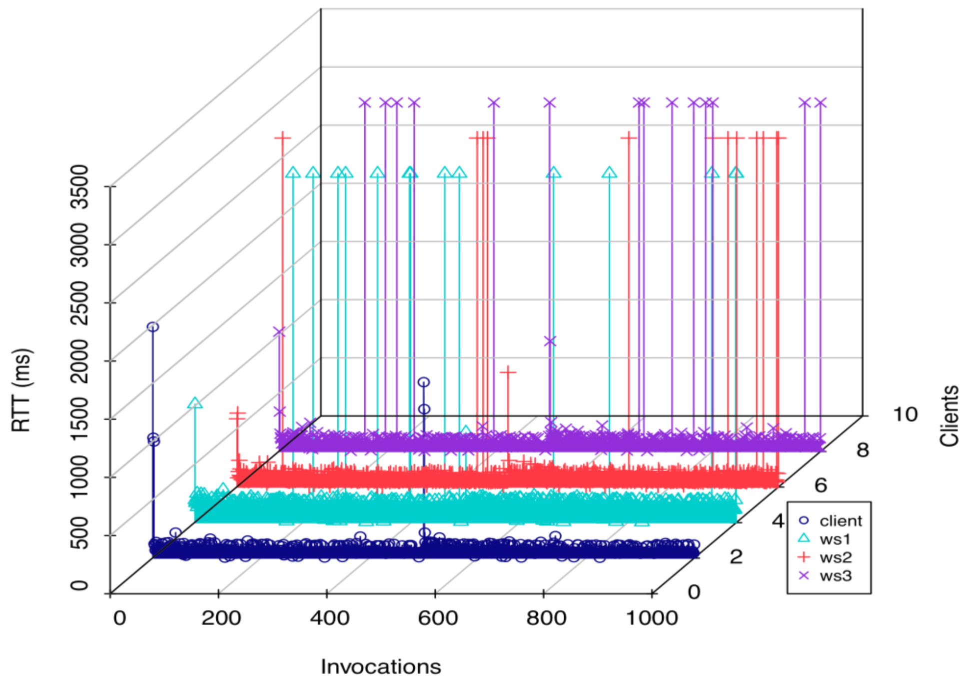
TABLE 5-2 DROP AND RANDOM ERROR INJECTED

- ***Section 2:*** The NetFIS injects various types of faults between the client and the Web Services in different emulated network conditions. The combinations of the injected faults are shown in Table 5-2. The client invoked the Web Services via the NetFIS 1,000 times in each setting. Table 5-2 shows the statistics of the results in each test case as logged by WS-Mediator logging mechanism. The results indicate that the tool coped well with the settings and injected the expected faults correctly. When "drop" was injected, the client threw a "time-out" exception after a 10 second wait, indicating that the response was lost. When errors were injected, except messages reading "*Cannot find dispatch methodfor{http:% /webservices. calibayes.ncl.ac.uk/} getAvailableSimMethods*" were thrown by the client, indicating corrupted SOAP messages, but the JAX-WS framework was unable to deal correctly with the responses. Figure 5 and Figure 6 show the plots of the results for some test cases, which clearly demonstrate the effectiveness of the tool. The tool simulates real work network conditions and faults to help with robust client application development (in this case, by applying the WS-Mediator).

Network: fast, drop: 1%, error: 0%



Network: heterogeneous, drop: 1%, error: 0%



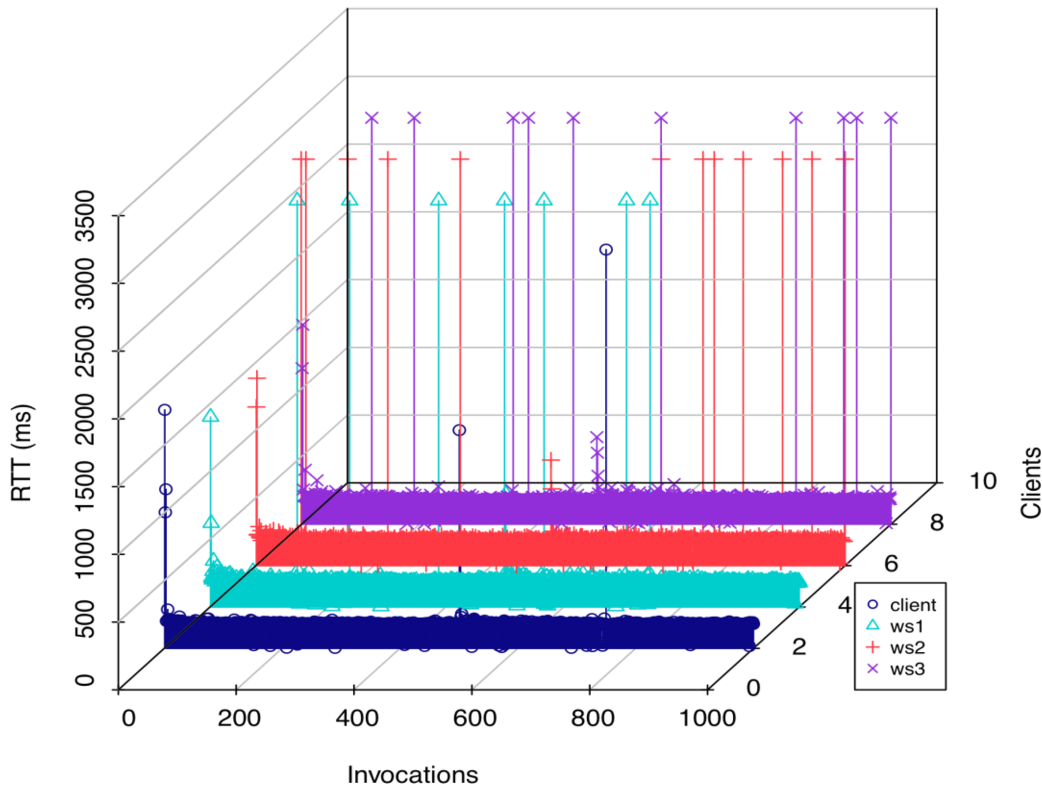


FIGURE 5-5 RTT OF THE TEST CASES (CLIENT)

Figure 5-5 shows a comparison of the RTT of the three Web Services and the final responses delivered to the client by the WS-Mediator. The ‘clients’ axis represents the invocation RTT monitored at each client thread (which respectively deals with WS1, WS2, WS3) and the client application that employs the WS-Mediator to deal with the results received by the client threads. We chose the 1% drop rate injection scenario to show the comparison, since this test case affects the RTT most. As the faults were injected arbitrarily into the three Web Services, the N-version programming fault tolerance mechanism in the WS-Mediator successfully dealt with the faults in most test cases and masked the reliability problems to the client. The client application only threw exceptions when the three Web Services failed simultaneously.

NETWORK EMULATED	MESSAGE TYPE	INJECTED DROP RATE IN 1000 REQUESTS ***FOR EACH LINK***				INJECTED ERROR RATE IN 1000 REQUESTS ***FOR EACH LINK***			
		Target %	***Achieved %***	***Exp. Except***	***Generated Exceptions***	***Target %***	***Achieved %***	***Exp. Except***	***Generated Exceptions***
Fast WAN	Requests	**1%**	**0.9%**	**1%**	No exceptions by WS but timed out by client and retrans-mitted	**1%**	**1.1%**	**1%**	**0.6%** fault response by WS
	Responses	**1%**	**0.9%**	**1%**	No action by WS but timed-out by client with no retrans-mission.	**1%**	**1%**	**1%**	**0.4%** Detected by client by throwing excep-tions

TABLE 5-3 THE AFFECT OF INJECTED DROP AND ERROR TEST CASES (WEB SERVICE 1)

- ***Section 3:*** Table 5-3 shows the affect of injecting Drop and error into the exchanged messages between the client and WS1 by using NetFIS logging mechanism. We have chose only WS1 to demonstrate the affectance of the injected faults as the three replicated Web Services have identical hardware, operating systems, middleware, and so on. The test was performed using a loop iteration of 1000 and was repeated 3 times for each emulated WAN. There was no variation in results for each set of results produced for a particular emulated WAN in terms of drop rate and error rate which was a very small variation in percentage totals between each set of loop iterations, which can be safely neglected. As Table 5-3 shows, the NetFIS logging mechanism recorded the following classes of faults for WS1 when Fast WAN emulated. We chose the 1% drop rate and 1% error rate injection scenarios to show failures recorded by NetFIS, since these two fault injected cases affects the system most:

1. Expected exception and generated exception.

2. Expected exception but not generated exception.

From the table above it can be noted that when dropping requests and responses in the rate 1% , no failures or exceptions detected by the NetFIS. Web Service has no way of knowing that a request has been sent by the client, that's due requests dropped before reaching the Web service so NetFIS can not detect the affect of dropping requests. However NetFIS has recorded that all the discarded requests have been retransmitted again by the client. That indicates a level of reliability within the client side which uses some kind of time-out mechanism which times out when the specified time out elapses and retransmits the lost requests. However discarded response messages a response message from server to client generates no network fault messages since it is discarded in the middleware level. However this is handled by a time out at the client side. However the requests of the lost responses have not retransmitted again by the client whereas it was an exception propagated from the middleware to the client application.
This dropping messages test appeared to indicate that the system under test has a high degree of resilience to SOAP message loss over an unreliable network during the duration of the test, since the system continued to operate correctly under the presence of these faults. No unexpected figures were faced indicating a level of dependability under this test case. Dropping SOAP messages seemed to cause no adverse effects on either the client or the server such as server crashes/hangs, that may was because of the N-version fault tolerance mechanism applied at client side which uses a number of replicated Web services and the high probability that at least one of the responses could be received of the other two lost or delayed.

The table above shows also recorded the percentage random error fault injected into requests and responses going from client to server, and from server to client, respectively. The table shows 1% of request messages have been damaged by inserting extra characters into SOAP messages. it can be seen from the table that only 0.6% of the damaged messages have caused exceptions to be thrown by the server. Which means almost 50% of the corrupted request messages went undetected by the server. That's due to the inserted errors has been inserted into the message randomly, that is, in this case the errors inserted into SOAP-ENV:Body element by chance. An exception was expected because the fault would break the SOAP stack syntax or would be detected by the client application in the case the error would cause a corruption to other elements of the message. In the case of

errors inserted into SOAP-ENV:Body element, It seems it was syntactically correct according to XML but no exceptions were thrown. However it did not appear to affect the dependability of the system during the test, which continued to function correctly in all other respects.

In the case of errors inserted into other parts of the SOAP message which affected the syntax of the message according to XML, the server threw exceptions for 0.6% of the 1% of the error message rate injected into the request messages, and then fault messages sent back to the client. When response messages were corrupted randomly by the tool, the client reacts in the same way as the server did. Overall the system appeared to have a high degree of dependability since it continued to operate even when injecting errors into SOAP envelope. Furthermore, the client was reliable to handle exceptions thrown by the server in the case of corrupting the syntax of the message, and the client middleware propagated the faults to the client application.

The overall assessment to tested system shows that NetFIS can be used to conduct assessment of the performance and fault tolerance mechanisms of Web services. As the faults were injected into the three Web Services, the N-version programming fault tolerance mechanism in the WS-Mediator successfully dealt with the faults in most test cases and masked the reliability problems to the client. However the system failed to mask problems when the three Web Services failed simultaneously. The failures was due to faults were injected by NetFIS in the same time when the three requests sent out to the three Web Services. In spite of the fact that the WS-Mediator could not mask the failure of the system when the three Web Services failed in the same time, the failure average rate was only 0.09% of the total requests of the three Web Services when the drop/error rate is high (1%).

In both normal and faulty environment, the system provides a good RTT performance. This is reasonable since the first properly returned response will be taken as the final outcome. This strategy (N-Version) can be used to tolerate faults and achieve better respond time performance, however in the faulty environment; the whole system reliability is 99.9% due to the three Web services failed stimulatingly. Moreover, the dependability of Web service applications in the faulty environment is in average of 99%, which are due to the corrupted request messages went undetected by the Web service applications and also the dropped responses did not get retransmitted again by the Web service applications which is due to the faults are happened in the middleware level. We

suggest that at the Web service application a fault tolerance mechanism such as WS-ReliableMessaging [76] need to be deployed so as to detect dropped responses, which defines protocols that enable Web services to ensure reliable messages exchange.

5.2 Conclusions

The NetFIS tool has been tested by using real Web Service system case studies. The case studies demonstrated that the tool that can inject faults into Web Service applications without touching the code of the system under test. In addition, there is great flexibility in the number and type of fault that can be injected. The overhead introduced into the system by the NetFIS is not significant. This is a negligible overhead that conforms to the design assumption that the target WAN to be emulated exhibits much greater delays than this small overhead.

Furthermore, the tool demonstrated its capacity to emulate a wide range of networks by using different network configurations. The NetFIS also demonstrated how background traffic can be generated into the experiment.

The network emulation may not exactly mirror the real world environment. However, it is a significant advance on testing using a single machine or a LAN. In particular, sample traffic from a real network can be used in the emulation as well as self-similar traffic patterns.

Our experiments have clearly demonstrated the network simulation and fault injection capacities of the NetFIS and an example of how to use the functionalities of the tool for testing the fault-tolerance mechanisms of the client application. In this case, the WS-Mediator has demonstrated its fault tolerance capacity with service diversity and dynamic service composition reconfiguration. However as the tool is capable of injecting a large number of faults, it was impossible to represent the extent of the tool’s ability. Thus, the tool provides a flexible configuration for allowing the tester to configure the tool to suit different scenarios and cases.

6 Chapter 6 - Conclusions and Suggestions for Future Work

In this chapter, we summarize our work and make suggestions for further work. In Section 6.1, we summarize our research and the studies reviewed in each chapter. In Section 6.2, we outline a number of possible extensions that could be made to our testing method. In addition, we discuss how the knowledge gained in this study can be applied in future work to improve the testing of Web Service applications.

6.1 Contributions

Web service technology is developing very fast, and has been regarded as playing a critical role in many e-Commerce and e-Science applications. Due to the complexity of architecture and complicated application scenarios of Web Services, assessing their reliability is a challenging research topic. While there have been many approaches developed for improving the assessment of the dependability of individual Web Services and Web Service composition applications, there is still a need for testing solutions that can assess the dependability of Web Service composition, given the persistence of varied types of faults in the infrastructure. It is therefore essential to analyse the concrete dependability characteristics of Web Services and other involved components, such as individual component services, networks, and so on, and to develop testing solutions to assess Web Service systems with specific fault assumptions.

As Web Service composition is an activity involving the integration of several component services over computer networks, the dependability of service composition therefore relies on the dependability of individual component services and of the networks. The failure of a single node (e.g. a component service or a segment of the network) can undermine the dependability of the entire application. In reality, it is very hard to ensure that Web services do not fail during integration; moreover, computer networks are inherently unreliable. Hence, dependability solutions such as fault tolerance mechanisms applied to service composition need to be assessed in order to discover how they deal with failures of individual component services and networks to ensure the continuity of services.

Using traditional testing techniques raises many problems when testing service composition systems. It is hard to test Web Service applications which are developed to run over the Internet. Testing Web Service applications which involve running software on different environments and platforms is not an easy task without modifying the sources or the networking libraries of the hosting operating systems. Testing such systems requires a run-time environment − which is logically the Internet − to test the performance of each component of the system and also to assess the impact of the fault tolerance mechanisms applied to the system. However, such approaches are not always attractive or achievable. The cost of setting up a WAN or using the Internet for the sake of testing is very high. Moreover, it is almost impossible to control the dynamics of such networks, such as through increasing stress, load, or errors.

Compared with the existing testing solutions, the most closely-related work was undertaken on the assessment of CORBA applications, by using a Wide Area Network emulator at application level. This presented promising results [48]. However, some shortcomings of this CORBA fault injection approach were identified. In the CORBA interceptor level, the messages are already coded in binary code; therefore this method does not target any particular elements in the message to inject faults into, such as function parameters in the case of RPC. Furthermore, corruption and dropping messages are only injected by throwing exceptions. That means the CORBA fault injection method assumption is for injecting the mentioned faults solely to test the system's ability to deal with such exceptions; whereas it is more logical to inject explicitly the fault and observe its effects on the system.

All this has prompted us to develop a fault injection testing method focused on assessing the performance and the fault tolerance mechanisms of composed services, specifically service component failures and communication failures.

The testing method is for testing the performance and fault tolerance of either a single Web Service or a composed service, without any modification to the system being tested. No recompiling or patching is necessary. The method has three main features: first, injecting communication faults (loss, delay and corrupting system exchanged messages) based on a WAN emulator. The second feature is injecting specific faults into RPC parameters of the exchanged messages. And thirdly, the tool generates background

workload, to more accurately emulate real networks and thus produce more realistic results.

A prototype of our fault injection testing method, called Network Fault Injection Service (NetFIS), has been implemented using only the Java Web Service technology. The three main features of the method have been implanted as a composed service deployed to overlay the network architecture.

The first feature is the WAN emulator. The main task of the WAN emulator is to provide a network running environment which generates a virtual workload in the system. We have implemented the Network emulation as a standalone composed service system, which consists of a number of Web Service applications. Each Web Service application represents a node or link, and is responsible for emulating that node or link on the target-emulated network. We have chosen this design to improve the flexibility of the tool, in terms of its capacity to be run on a different application space, which can be a different physical machine, OS, or Web Service middleware. In addition, the WAN emulator enables the tester to control the virtual network environment and its parameters by using the network configuration files so as to test the system under different circumstances such as different dropping rate, error rate, delay, and so on.

The second main feature of the tool is injecting communication faults which are based on the WAN emulator service. The communication faults are decided by the WAN emulator, whereas the injection task is performed by the Fault Injection Mediator Service (whose main responsibility is intercepting the system messages). Separating the injection mechanism from the WAN emulator makes the tool more flexible and portable to different distributed systems.

The third feature is injecting interface faults, which is done through the Fault Injection Mediator Service and based on the Web Service interface document, WSDL. A system fault model can be created by importing WSDL for the Web Services making up the system and mapping certain WSDL elements to certain elements within the fault model based on their type. Then a fault model can be generated and used to create triggers for injecting faults into the system by injecting specific faults into RPC parameters of SOAP exchanged messages.

We have conducted a series of experiments with several scenarios relating to real-world Web Services (a Bioinformatics Web service [73], which deployed the WS-Mediator [74] to improve its dependability) to evaluate our solution, and their results have demonstrated the applicability and efficacy of the NetFIS approach.

6.2 Limitations and Future Work

Software testing involves two separate issues: the ability to test and the selection of test scenarios. The ability to test has been tackled in this thesis by providing all the necessary functionalities in the tool to inject communication faults and software-specific faults into the system. However as each fault tolerance mechanism applied to systems must tolerate and cope with some particular class of failures, a testing method should be able to generate testing scenarios based on this fault tolerance mechanism and its claimed ability to mask such failures. Because of the considerable number of possible testing scenarios which could be built based on the software itself, this research has left the choice of scenarios to the tester. The experiments reported in the thesis were mainly intended to demonstrate the functionality and the usefulness of the NetFIS tool. Research is therefore required into improving the tool's capacity to help the tester to generate testing scenarios based on the fault tolerant mechanisms claimed to be applied to the system (i.e. the N-version).

Using the NetFIS method needs to be inserted as proxies into the target system so as to intercept and manipulate exchanged messages. This obviously causes delays to the intercepted messages. The amount of delay added is implementation-dependent and its impact on the experiments also depends on the nature of the target system. The important issue is whether the added delay is within acceptable bounds. Although the design of the tool assumes that the Wide Area Networks to be emulated are large enough so that the overhead delay introduced by the emulation is negligible compared to the actual network delays, this delay may be unacceptable for some real-time systems. An example of one such case is when a value fault is to be injected and the added delay would make the fault resemble a value and timing fault (arbitrary fault). Therefore a solution is required for reducing this overhead delay: one possibility is running each Network Node Emulator Service emulating a network node and its corresponding Fault Injection Mediator Service

in the same machine as the service component of the target system. This would reduce the time consumed in connecting to each other.
In Chapter 3, we discussed the possibility of adapting the NetFIS tool and applying it to other existing message-based systems such as Grid and CORBA systems. Further detailed investigations are needed in this area and the approach should be evaluated with practical examples in order to determine differences in message syntax.

Currently the NetFIS implementation does not generate the configuration files of the NetFIS (i.e. Value Fault Model, network topology, etc.) which are instead written manually. Whilst this is acceptable for short test campaigns it would soon become time consuming for large test campaigns. We therefore propose that the NetFIS tool should be extended to include this functionality, either as a post-processing step or as part of the run-time visualization.

From the test cases carried out here, our experiments have clearly demonstrated the network simulation and fault injection capacities of the NetFIS and an example of how to use the functionalities of the tool for testing Web Service systems. The tool provides a large number of flexible configurations for allowing the tester to configure the tool to suit different scenarios and cases. Therefore further experiments can be carried out in the future when different Web Service systems need to be assessed.

7 Bibliography

1. W3C (2004) Web Services Architecture. W3C Working Group Note, 11th February, 2004. Available from: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#service_oriented_architecture. (Accessed on 30 June 2011).

2. W3C (2004). Web Services Glossary -W3C Working Group Note 11 February 2004. Available from: http://www.w3.org/TR/ws-gloss/. (Accessed on 30 June 2011).

3. Attiya, H. and Welch, J. (2004) *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Hoboken, NJ: John Wiley & Sons.

4. Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004) *Web Services: Concepts, Architectures and Applications*. Berlin: Springer.

5. Laudon, K. and Traver C. (2002) *E-Commerce*. Boston: Addison Wesley.

6. Ebay (2008) ‘Ebay Developers Program’. Available from: http://ebaydeveloper.typepad.com/ (Accessed on 05/08/2011).

7. Gable, J. (2002) ‘Enterprise application integration’. Available from: http://findarticles.com/p/articles/mi_qa3937/is_200203/ai_n9019202. (Accessed 16th August, 2011).

8. O.M.G. (2007) ‘Catalog of Specialized CORBA Specifications’. Available from: http://www.omg.org/technology/documents/spec_catalog.htm. (Accessed on 16th August, 2011).

9. Web Services-Interoperability Program (2007) ‘Basic Profile Version 1.2’. Available from: http://www.ws-i.org/Profiles/BasicProfile-1_2(WGAD).html. (Accessed on 22nd August 2011).

10. W3C. (2007) 'HTTP - Hypertext Transfer Protocol'. Available from: http://www.w3.org/Protocols/. (Accessed on 30th June 2011).

11. W3C. (2007) 'SOAP Version 1.2. Available from: http://www.w3.org/TR/soap/. (Accessed on 30th June, 2011).

12. W3C. (2001) 'Web Services Description Language (WSDL) 1.1'. Available from: http://www.w3.org/TR/wsdl. (Accessed on 30th June, 2011).

13. Merzbacher, M. and Patterson, D. (2002) 'Measuring end-user availability on the Web: practical experience', in *Proceedings of the International Conference on Dependable Systems and Networks*. Los Alamitos, CA: IEEE Computer Society Press.

14. Kalyanakrishnan, M., R.K. Iyer, and J. Patel (1997) 'Reliability of Internet Hosts - A Case Study from the End User's Perspective', in *Proceedings of the 6th International Conference on Computer Communications and Networks*. Los Alamitos, CA: IEEE Computer Society. p. 418.

15. Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr C. (2004) 'Basic concepts and taxonomy of dependable and secure computing', *IEEE Transactions on Dependable and Secure Computing*, 1(1): p. 11-33.

16. Farj, K, and Speirs, N.A. (2009) 'A Tool for Assessing Fault Tolerance Mechanisms applied to Web Service applications' in *12th European Workshop on Dependable Computing (EWDC)*.Toulouse, France, 2009: p. 2.

17. Farj, K, and Speirs, N.A. (2010) 'A Tool for Testing Fault Tolerance of Web Service Systems', in *2010 International Conference on Dependable Systems and Networks: Supplemental*. Chicago, Illinois: IEEE.

18. Yuhui, C., Romanovsky, A., Gorbenko, A., Kharchenko, V., Mamutov, S. and Tarasyuk, O. (2009) 'Benchmarking Dependability of a System Biology

Application', *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*. 2-4 June 2009.

19. Looker, N. and X. Jie (2003) 'Assessing the dependability of SOAP RPC-based Web services by fault injection', in *Proceedings. Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*.Capri, Italy, October 1st-3rd.

20. Marsden, E., Fabre J.C., and Arlat J. (2002) 'Dependability of CORBA systems: service characterization by fault injection', in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*. 2002.

21. W3C. (2008) 'Extensible Markup Language (XML) 1.0 (Fifth Edition)'. Available from: http://www.w3.org/TR/xml/. (Accessed on 30th June, 2011).

22. Google (2007) 'Google SOAP Search API'. Available from: http://code.google.com/apis/soapsearch/reference.html. (Accessed on 30th June, 2011).

23. Papazoglou, M. (2008) 'Web services: Principles and technology', *Harlow*: Pearson Prentice Hall.

24. Birrell, A.D. and B.J. Nelson (1984) 'Implementing remote procedure calls', *ACM Trans. Comput. Syst.*, 2(1): p. 39-59.

25. W3C. (2007) 'SOAP Version 1.2 Part 2'. Available from: http://www.w3.org/TR/soap12-part2/#soapforrpc. (Accessed on 30th June, 2011).

26. Weerawarana, . Curbera, F. Leymann F. and Ferguson D. (2005) 'Web Services Platform Architecture', *Upper Saddle River*, NJ: Prentice Hall.

27. IBM (2001) 'The structure of a SOAP message'. Available from: http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp?topic=/com.ibm.cic

s.ts31.doc/dfhws/concepts/soap/dfhws_message.htm. (Accessed on 30th June, 2011).

28. Mei-Chen, H., T.K. Tsai, and R.K. Iyer (2004) 'Fault injection techniques and tools', *Computer*, 30(4): p. 75-82.

29. A Avizienis, J C Laprie, B Randell. (2000) 'Fundamental Concepts of Dependability'. Available from: http://www.mendeley.com/research/fundamental-concepts-dependability/. (Accessed on 12th August, 2011).

30. Vieira, M., N. Laranjeiro, and H. Madeira (2007). 'Assessing Robustness of Web-Services Infrastructures', in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Edinburgh, Scotland, June 25st-28rd .2007.

31. Looker, N., M. Munro, and J. Xu (2004) 'WS-FIT: A tool for dependability analysis of web services', in *Proceedings - International Computer Software and Applications Conference*. Hong Kong, China, September 28st-30rd .2004.

32. Kalyanakrishnam, M., Z. Kalbarczyk, and R. Iyer (1999) 'Failure data analysis of a LAN of Windows NT based computers', in *Proceedings of the 18th IEEE Symposium on.Reliable Distributed Systems*, Lausanne , Switzerland, October 19st-22rd. .1999.

33. Carreira, J.V., Costa G. and Silva J.G. (1999) 'Fault computer injection spot-checks system dependability', *IEEE Spectrum*, 38 (8): 50-55.

34. Looker, N., Burd L, Drummond S., Xu J. and Munroe M. (2005) 'Pedagogic data as a basis for Web service fault models'. in IEEE Workshop onService-Oriented System Engineering. 20-21 Oct. 2005.

35. Dawson, S., Jahanian, F., Mitton, T., Teck-Lee Tung. (1996). 'Testing of fault-tolerant and real-time distributed systems via protocol fault injection', in *Proceedings of Annual Symposium on Fault Tolerant Computing*. Sendai , Japan, 25-27 Jun. 1996 .

36. Zibin, Z. and M.R. Lyu. (2008). WS-DREAM: A Distributed Reliability Assessment Mechanism for Web Services. *Dependable Systems and Networks With FTCS and DCC*, 24-27 June. 2008.

37. A. Gorbenko, V. Kharchenko, O. Tarasyuk, Y. Chen, A. Romanovsky. 'The Threat of Uncertainty in Service-Oriented Architecture'. Proc. RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems (SERENE'2008), pp. 49-54, 2008.

38. Chillarege, R., Michael R. Lyu Ed. (1995). 'Orthogonal Defect Classification'. Chapter 9 of Handbook of Software Reliability Engineering", IEEE Computer Society Press, McGraw-Hill, 1995.

39. Vieira, M., N. Laranjeiro, and H. Madeira. (2007). 'Benchmarking the Robustness of Web Services'. *in Dependable Computing,PRDC 2007. 13th Pacific Rim International Symposium on*. 2007.

40. Seungjae, H., K.G. Shin, and H.A. Rosenberg (1995) 'DOCTOR: an integrated software fault injection environment for distributed real-time systems', *in Proceedings of International Computer Performance and Dependability Symposium*. 24-26 Apr. 1995.

41. Dawson, S., Jahanian F., and Mitton T. (1996) 'ORCHESTRA: a probing and fault injection environment for testing protocol implementations', in *Proceedings of IEEE Computer Performance and Dependability Symposium*. 4-6 Sep 1996.

42. Kanawati, G.A., N.A. Kanawati, and J.A. Abraham. (1992). FERRARI: a tool for the validation of system dependability properties. in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*. 8-10 Jul. 1992.

43. de Almeida, L.F. and Vergilio S.R. (2006) 'Exploring Perturbation Based Testing for Web Services' in *ICWS '06. International Conference on Web Services*. 18-22 Sept. 2006.

44. Siblini, R. and Mansour M. (2005) 'Testing Web services'. in *Computer Systems and Applications', in The 3rd ACS/IEEE International Conference on*. 2005.

45. Sneed, H.M. and H. Shihong. (2006). 'WSDLTest - A Tool for Testing Web Services'. in *Web Site Evolution, 2006. WSE '06. Eighth IEEE International Symposium on*. 2006.

46. Jeff, O. and X. Wuzhi (2004) 'Generating test cases for web services using data perturbation', *SIGSOFT Softw. Eng. Notes*, 29(5): 1-10.

47. Jia, Z. and R.G. Qiu (2006) 'Fault Injection-based Test Case Generation for SOA-oriented Software', in *IEEE International Conference on Service Operations and Logistics, and Informatics*. 2006, pp. 1070-1078.

48. Inayat Q., M. Alsaeed. , Speirs N.A. and Ezhilchelvan P.D. (2007) 'Examining the Performance Benefits of Fail–Signal Approach using a Network Emulator' in *Proceedings of the International Conference on Computer, Control and Communication*. Karachi, Pakistan.

49. Michael, P.P., Willem, and H. Jan Van Den (2006) 'Service oriented design and development methodology', *Int. J. Web Eng. Technol.*, 2(4): 412-442.

50. W3C (2004). 'Web Services Addressing (WS-Addressing)'. Available from: http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/. (Accessed on 30th July 2011).

51. Wikipedia (2011) 'Network simulation'. Available from: http://en.wikipedia.org/wiki/Network_simulation#cite_ref-AG07_0-0. (Accessed on 30th July, 2011).

52. Wikipedia (2011) 'Network emulation'. Available from: http://en.wikipedia.org/wiki/Network_emulation. (Accessed on 30th July, 2011).

53. Tian, J., J. Xu, and C. Zhang Hua (2008) 'A Parallel Self-Similar Network Traffic Simulation Method on a Large Time Scale', in *ISISE '08. International Symposium on Information Science and Engineering*. 20-22 Dec. 2008.

54. Shibin, S., J.K.Y. Ng, and T. Bihai (2004) 'Some results on the self-similarity property in communication networks', *IEEE Transactions on Communications*, 52(10): 1636-1642.

55. Alsaeed, M. and Speirs, N.A. (2007) 'A Wide Area Network Emulator for CORBA Applications', in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. 2007.

56. Looker, N., Munro M., and Xu J. (2005) 'A comparison of network level fault injection with code insertion', in *Proceedings - International Computer Software and Applications Conference*. Edinburgh, Scotland.

57. Ghosh, S. (2001) 'Fault injection testing for distributed object systems', in *TOOLS 39. 39th International Conference and Exhibition on.Technology of Object-Oriented Languages and Systems*. Santa Barbara, CA, USA.

58. Sha Tao, P.D. Ezhilchelvan, S.K. Shrivastava. (1995) 'Focused fault injection testing of software implemented fault tolerance mechanisms of Voltan TMR nodes'. *Distributed Systems Engineering*, 1995. 2: p. 39-49.

59. Selic, S.B., (2004) 'Fault tolerance techniques for distributed systems'. Available from: http://www.ibm.com/developerworks/rational/library/114.html. (Accessed on 30th July, 2011).

60. Inhwan, L. and R.K. Iyer (1995) 'Software dependability in the Tandem GUARDIAN system', *IEEE Transactions on Software Engineering*, 21(5): 455-467.

61. Xu, J., Townend, P., Looker, N. and Groth, P. (2008) 'FT-Grid: a system for achieving fault tolerance in grids', *Concurrency and Computation: Practice and Experience*, 20, (3), pp. 297-309.

62. Marsden, E., J.-C. Fabre (2001) '*Failure Mode Analysis of CORBA Service Implementations Middleware'*, in Guerraoui, R.(ed). Vol. 2218 Springer Berlin / Heidelberg, pp. 216-231.

63. Mukherjee, A. and D.P. Siewiorek (1997) 'Measuring software dependability by robustness benchmarking', *IEEE Transactions on Software Engineering*, 23(6): 366-378.

64. Koopman, P. and DeVale J. (1999) 'Comparing the robustness of POSIX operating systems', in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. Digest of Papers. 1999.

65. Alsaeed, M. and Speirs N.A. (2008) 'Examining the Performance Benefits of Fail-Signal Approach using a Network Emulator': p. 12.

66 W3C (2004) 'WS-MessageDelivery'. Available from: http://www.w3.org/Submission/2004/SUBM-ws-messagedelivery-20040426/. (Accessed on 30th July, 2011).

67. Sun, J. (2008) 'Java API for XML Processing (JAXP) Tutorial'. Available from: http://java.sun.com/webservices/reference/tutorials/jaxp/html/stax.html. (Accessed on 30th June, 2011).

68. Paxson, V. (1997) 'Fast, approximate synthesis of fractional Gaussian noise for generating self-similar network traffic*', SIGCOMM Comput. Commun. Rev.,* 1997. **27**(5): p. 5-18.

69. Wing-Cheong, L., Erramilli, A., Wang, J. L. and Willinger, W. (1995) 'Self-similar traffic generation: the random midpoint displacement algorithm and its properties', in *Communications, ICC '95 Seattle, 'Gateway to Globalization', IEEE International Conference on*. 18-22 Jun 1995.

70. (AIX) (2000) 'N.A.I.E. Packet Length Distributions'. Available from: http://www.caida.org/analysis/AIX/plen_hist/. (Accessed on 18th August, 2011).

71. Fraleigh, C., Moon, S., Lyles, B., Cotton, C., Khan, M., Moll, D., Rockell, R., Seely, T. and Diot, S. C. (2003) 'Packet-level traffic measurements from the Sprint IP backbone', *Network, IEEE*, 17, (6), pp. 6-16.

72. Farj K, C.Y., Speirs. N.A, (2012) '*A Fault Injection Method for Testing Dependable Web Service Systems', 9th European Dependable Computing Conference - EDCC 2012*, 2012: p. 10. (submitted).

73. Chen, Y., Lawless, C., Gillespie, C. S., Wu, J., Boys, R. J. and Wilkinson, D. J. (2010) 'CaliBayes and BASIS: integrated tools for the calibration, simulation and storage of biological simulation models', *Briefings in Bioinformatics*, 11, (3), pp. 278-289.

74. Chen, Y. and A. Romanovsky (2008) 'Improving the Dependability of Web Services Integration', *IT Professional*', 10(3): 29-35.

75. Mark, E.C. and B. Azer (1997) 'Self-similarity in World Wide Web traffic: evidence and possible causes', *IEEE/ACM Trans. Netw.*, 5(6): 835-846.

76. Sanjiva Weerawarana, F.C., Frank Leymann, *Web Services Platform Architecture*, in *Prentice Hall PTR* 2005.

8 Appendix – The Performance of NetFIS tool

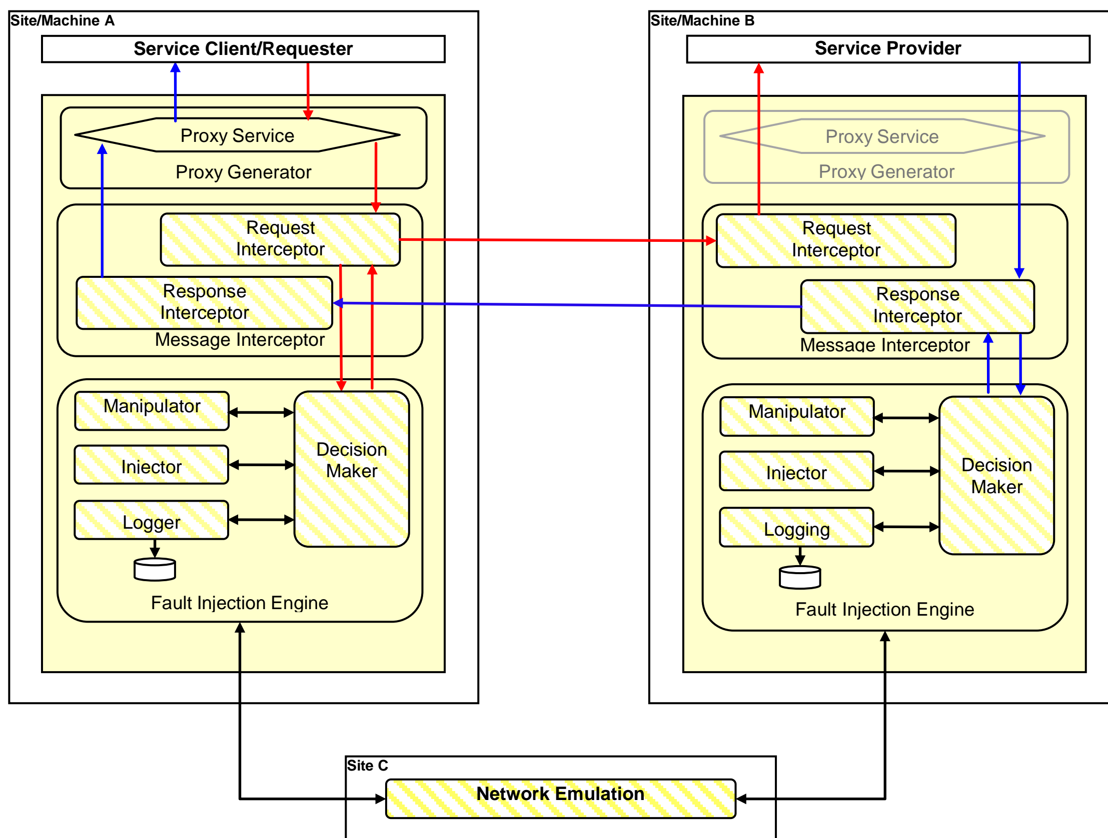
Tool Performance:

This section presents the basic operations of the NetFIS implementation and measures the overhead introduced by the tool to the system under test. The features to be examined are:

1. Intercepting messages at application level by using proxies;
2. Providing Network Emulation;
3. Injecting Timing fault;
4. Injecting Random Value fault.

These features describe the Network Fault Injection and the Software Fault Injection at the application level on which the NetFIS method is based.

Scenario:
The scenario is based on a simple Client and Server application. As detailed in [17], the Server is running a SOAP based Web Service application, which is invoked by the Client as shown in the below.



A SIMPLE CLIENT AND SERVER APPLICATION

The SOAP Web Service was implemented and deployed on a machine in the Newcastle University LAN (Machine B), providing a set of operations that return a simple sequence of characters. A SOAP client was implemented which invokes these operations. The client is also executed on a separate machine in the Newcastle University LAN (Machine A).

The client application is to first invoke the Web Service directly, then via the NetFIS. The the basic operations of the NetFIS and its performance have been examined. Moreover the overheads introduced to the system by using our tool are also measured. The result obtained from the logging files has been analyzed and discussed. The configuration of the experiment is explained in the next section.

Configuration:

Two configuration setups were used to perform this experiment, as follows:

1. The first configuration was for running the client and the Web Service by using a middleware package of Apache Axis2. Both the server and the client were hosted on two machines in the Newcastle University LAN using Redhat Linux 8.0, running Java version 1.6.14.

   The client was implemented to measure the response-time delay of the requests in three different runs: 1,000, 5,000 and 10,000 requests. The client logged the response-time delays for each execution, after which the logging was analyzed offline.

2. The second configuration was accomplished by using the same configuration as the first, but deploying the NetFIS tool. The client had to invoke the server through the NetFIS. In turn the NetFIS had to emulate a WAN and to inject network-related faults for evaluating the overhead and the functionality of the NetFIS prototype. The target WAN topology is a very simple two-node network setup, which makes it easier to monitor its behaviour. Various types of faults were examined; however the synthetic traffic of the *NESs* was disabled. There were no sources of traffic other than the normal traffic in the Newcastle University LAN, so as to simplify evaluating the tool.

In the first configuration, the response-time delay of the system under test was measured without using the tool. In the second configuration, the performance and the functionality

of the tool were measured in two emulated network configurations: a fast WAN and a slow WAN. The fast WAN configuration was as follows: the propagation delays are fixed to 2ms which is typical of inter-city links within the UK, and the bandwidth of each link is 4mb/s. The slow WAN parameters were as follows: the propagation delays are fixed to 20ms which is typical of distant locations and links that span the oceans (e.g. between London, UK and Tripoli, Libya), and the bandwidth of each link is 1,000Kb/s. The system being tested was a Web Service client-server pair. The server simply offers a service that returns an array of characters of fixed length (100 bytes). The server is running on one node in the network while the clients are running on the other node.

In order to emulate our simple two-node network, two *NES* services and two *Fault Injection Mediator Services (FIMSs)* were required (each node needs one FIMS for injecting faults − if any − and one *NES* for emulating the network of the node); furthermore, one *NCS* was needed for controlling the WAN.

All the systems (Client and Web Service) and the tools were run on three machines from a cluster of 24 Linux machines (Fedora Core 5) running on Apache Axis2, connected by a 100Mb Ethernet LAN. The system application pair was tested using a loop iteration of 1,000, 5,000 and 10,000 requests. The WAN emulation was provided and faults (dropping, and randomly corrupting the SOAP envelope) were injected as configured in the WAN emulator and all the events were logged.

Results and analysis:

The test has been undertaken with two different setup configurations. The first setup was for testing the delay overhead introduced by the tool. The second setup was for testing the functionality of injecting faults, such as delaying, dropping and corrupting the SOAP messages exchanged between the client and server.

The table below shows the data collected from the logging files for the first setup experiment. The response-time delay in milliseconds was collected by running the system without using our fault injection tool (NetFIS). It also shows how the response-time delays were measured by running the same experiment under the control of the tool with two different emulated network configurations (fast and slow), without generating any additional traffic and of course without injecting any dropping and corruption into the

exchanged messages. However a delay injected into the system resulted from the configuration of the propagation delay parameters for each emulated WAN, namely: 2 millisecond propagation delays for Fast WAN and 20 millisecond delays for Slow WAN.

Emulated Network	Injected Delay, (ms)	Total requests	Response time Delay		
			Max (ms)	**Min (ms)**	**Avg (ms)**
LAN	None	1000	44.87	7.15	8.085
		5000	56.93	6.67	7.465
		10000	112.93	6.13	7.09
Fast WAN	2	1000	291.51	32.87	47.87
		5000	559.77	31.42	41.30
		10000	281.80	29.23	38.56
Slow WAN	20	1000	198.89	80.45	91.55
		5000	199.31	79.27	88.96
		10000	217.96	78.59	87.79

RESPONSE-TIME OVERHEAD WITH DELAY

The average response-time delays of running the system (Client and Server) without the tool being deployed (when the client invokes the web service) were 8.085, 7.465 and 7.09 milliseconds for each loop iteration, namely 1,000, 5,000 and 10,000 respectively. There was no big variation in results for each result produced for any particular loop iteration value but there was a very small variation in percentage totals between loop iterations. However the averages of running the system under the control of the NetFIS show that the overhead of running the tool ranges between 36 and 45 milliseconds (when subtracting the injected delay) when emulating the fast WAN. Also, it ranges between 85 and 89 milliseconds (when subtracting the injected delay) when emulating the slow WAN. This is a negligible overhead that conforms to the design assumption that the target WAN to be emulated exhibits much greater delays than this small overhead.

Table above shows the results of the second setup configuration. The data in Table above were collected when running the system by using our NetFIS tool. The same configurations of the two emulated WANs (fast and slow) were used, except a drop and corruption of the exchanged messages needed to be injected.

No. of requests	Emulated Network	Injected drop		Injected error	
		Target (%)	**Achieved (total messages)**	**Target (%)**	**Achieved (total messages)**
5000	Fast	0.001	5	0.001	5
10000		0.001	9	0.001	10
5000	Slow	0.001	5	0.001	5
10000		0.001	11	0.001	12

DROP AND RANDOM ERROR INJECTED

As the table above indicates, the tool accurately injects the target network drop and error faults by achieving (or almost achieving) the exact rates targeted. For example, we tried to emulate fast and a slow network links with a drop rate of 1 message per 1,000 messages, which were achieved for 5,000 and 10,000 requests. The target injected random error into messages was also achieved in both network cases (fast and slow).

It is important to point out that in real WAN the underlying layers (middleware and networking stack layers) will be most likely capable of masking and recovering from errors and dropping of messages. However, this method is for propagating such faults to the application, so as to enable the application developers to test the performance of their applications under such conditions.