WS-Mediator for Improving Dependability of Service Composition

Thesis by

Yuhui Chen

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

Newcastle University
Newcastle upon Tyne, UK

July, 2008

Abstract

Web Services and service-oriented architectures (SOAs) represent a new paradigm for building distributed computing applications. In recent years, they have started to play a critical role in numerous e-Science and e-Commerce applications. The advantages of Web Services, such as their loosely coupled architecture and standardized interoperability, make them a desirable platform, especially for developing large-scale applications such as those based on cross-organizational service composition.

However, the Web Service technology is now facing many serious issues that need to be addressed, one of the most important ones being the dependability of their composition. Web Service composition relies on individual component services and computer networks, particularly the Internet. As the component services are autonomous, prior to use their dependability is unknown. In addition to that, computer networks are inherently unreliable media: from the user's perspective, network failures may undermine the dependability of Web Services. Consequently, failures of individual component services and of the network can undermine the dependability of the entire application relying on service composition.

Our research is intended to contribute to achieving higher dependability of Web Service composition. We have developed a novel solution, called WS-Mediator system, implementing resilience-explicit computing and fault tolerance mechanisms to improve the dependability of Web Service composition. It consists of a number of subsystems, called Sub-Mediators, which are deployed at various geographical locations across the Internet to monitor Web Services and dynamically generate Web Service dependability metadata in order to make resilience-explicit decisions. In

addition to applying the fault tolerance mechanisms that deal with various kinds of faults during the service composition, the resilience-explicit reconfiguration mechanism dynamically selects the most dependable Web Services to achieve higher service composition dependability fault tolerance.

A specific instance of the WS-Mediator architecture has been developed in the Java Web Service technology. A series of experiments with real-world Web Services, in particular in the bioinformatics domain, have been carried out using the Java WS-Mediator. The results of the experiments have demonstrated the applicability of the WS-Mediator approach.

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, Prof. Alexander (Sascha) Romanovsky. 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 Dr. Aad van Moorsel and Dr. Neil Speirs, 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. Peter Li and Dr. Panayiotis Periorellis 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. Mila Romanovskaya. 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.

List of Figures

List of Tables

Contents

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 vary from e-Commerce [5] applications, for example, Internet search engines [6] or online auctions [7], to complex large-scale e-Science projects [8, 9]. 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. Users' demand for Web Services seems to be driving the technology further. However, all the opportunities that this paradigm has brought notwithstanding, the Web Service technology at present is still far from maturity. The overwhelming pace of technological progress has also, inevitably, caused problems which may undermine the future of Web Services. Among these, their dependability is one of the most critical issues to be addressed.

Web Services have addressed many issues existing in the conventional technologies, such as Enterprise Application Integration (EAI) [10] and Common Object Request Broker Architecture (CORBA) [11, 12], to name just two of the more popular ones, extensively applied in the past decades. In these conventional distributed applications, service integration commonly relies on centralized brokers, or coordinators, which implement *objects-based* or *message-based interoperability* [4] with the participating component services and interact with them to perform automated business processes. The limitation of such 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 of the integration of cross-organizational

autonomous and heterogeneous services, especially when development cost, security and confidentiality are concerned [4]. Web Services resolve these issues with their loosely-coupled interaction pattern, standardized interoperability, extended peer-to-peer integration fashion, etc. [4]. In Web Services, functionalities implemented by the internal business procedures are deployed and exposed as services that can be discovered and accessed through the Web. The interaction between the client and the services generally relies on the SOAP/HTTP message binding [13-15]. The client, a business logic application (e.g. e-Science or e-Commerce workflow), invokes Web Services by sending them a SOAP message [2, 15], with the service request attached. Web Services receive and parse the SOAP message, process the business logic according to the service request, and return the results to the client via SOAP messages. During the integration, the client does not necessarily know anything about the Web Services involved other than their WSDL interface [16]; 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. [17]

Nevertheless, even with the advantages described above, Web Services are not a magic solution to all problems of distributed applications. Similarly to other distributed technologies, Web Service middleware relies on the existing underlying middleware, such as network protocols, to implement the essential low-level services [4]. Naturally, they inherit many of the dependability issues the conventional infrastructure suffers from. For example, the interaction between the client and the Web Services relies on the Web or other networks, which are inherently unreliable media that may cause a loss, delay or damage of the message [3, 18-20]; Web

Services are deployed on application servers, which may become unreliable or out-of-service, due to system maintenance or other internal activities [20, 21]; the design or implementation of the Web Service business procedure may contain faults and result in their erratic behaviour [20-22]. 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. It is only logical that the dependability of Web Services as a research domain has attracted active interest in recent years.

1.2 Our Research

This dissertation reports our work in developing solutions to improving the dependability of Web Services. We started the research by investigating the dependability means in the context of Web Services, followed with an in-depth analysis of dependability issues in Web Services based on our experiments with several real-world Web Services. 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 to improving the dependability of Web Services. Conceptually, this solution is based on our understanding of the specific dependability characteristics of Web Services. It addresses some dependability 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 duplicating others' work. We have adopted several novel approaches and concepts in the solution proposed, developed certain unique mechanisms to ensure the novelty, feasibility and efficiency of our approach, and proved them in a series of experiments with real-world Web Services. This work has been reported at various academic events and

conferences, including the International Conference on Dependable Systems and Networks 2006 [23], UK e-Science All Hands Meeting 2006 [24], the 3rd International Service Availability Symposium [37], ReSIST Student Seminar 2007 [25], etc. A comprehensive description of the WS-Mediator approach is published by the IT Professional magazine [26] in this year's May/June issue.

1.3 Our Contributions

While the recent active research effort aiming at the dependability of Web Services has developed some effective solutions, including those focusing on service composition, we believe that there are still many issues remaining in this domain, particularly concerning the dependability of service composition that relies on autonomous Web Services. Our approach does not follow the methodology commonly applied in other related work. We have learnt from our experiments and studies of related work that in SOA the client's perspective on the services might be dramatically affected by the network consequences. This calls for solutions that would improve the dependability of Web Service composition from the client's perspective, ensuring the continuity of the service provided to it. In order to address the outstanding dependability issues in the existing Web Service applications, our solution is based, in addition to the classic fault tolerance techniques, on certain novel concepts, such as Resilience-explicit computing [27], path diversity, etc. The contributions of our work are as follows:

- We have developed a WS-Mediator solution to improving the dependability of Web Service applications. The approach offers an off-the-shelf mediator system to ensure the dependability of service composition based upon the existing legacy Web Services.

- We have devised a WS-Mediator architecture which employs the dependability monitoring of Web Services, resilience-explicit dynamic reconfiguration of service composition as well as fault tolerance mechanisms to accomplish a smart system that can explicitly select most appropriate components to improve the dependability of the entire service composition.
- We have implemented a prototype of the WS-Mediator using the Java Web Service technology. The Java WS-Mediator implements a Web Service dependability monitoring mechanism to achieve the dependability of the services from the client's perspective. Its novel Resilience-explicit dynamic reconfiguration mechanism allows an on-the-fly dynamic integration of component services to utilize the richness of service redundancy available in the Web Service infrastructure, and optimizes the conventional service diversity strategies. The off-the-shelf fault tolerance mechanisms allow the system to cope with various types of faults. Moreover, the Java WS-Mediator can be deployed on a personal computer and seamlessly integrated into the existing Java client applications. It can be especially beneficial for the development of new Java client applications by providing intuitive invocation APIs to utilize the functionalities provided by the WS-Mediator for improving their dependability.
- We have conducted a number of experiments with real-world Web Services to evaluate the WS-Mediator approach and the Java WS-Mediator. The results of the experiments demonstrate the applicability and effectiveness of this solution.

1.4 Thesis Outline

The dissertation is organised as follows:

- Chapter 2 explains the fundamental concepts and definitions of SOA and Web Services. We define dependability in the context of Web Services and analyse their dependability. Finally, we summarize some related work in the area.
- Chapter 3 presents our WS-Mediator approach. In this chapter we discuss our objectives and introduce the notion of the WS-Mediator as well as explaining the WS-Mediator architecture and its components in detail.
- Chapter 4 introduces a prototype of the WS-Mediator. In this chapter, we explain how to implement the WS-Mediator system using the Java Web Service technology.
- Chapter 5 reports on the experiments conducted to evaluate the WS-Mediator approach. The results of the experiments with real-world Web Services are analysed to demonstrate the applicability of the WS-Mediator approach.
- Chapter 6 concludes this dissertation, offering our vision of the possible further development of the WS-Mediator system.

2. Dependability of Service-Oriented Architecture

2.1 Introduction

In this chapter, we will analyse dependability issues in the context of SOA and Web Services. Even though Web Services are becoming, with all their promising potential, a fundamental technology and platform in many distributed computing applications [6-9], they are now facing a range of critical challenges, dependability being one of the most crucial. In this chapter, we will introduce the general concept of dependability and discuss dependability means in the context of Web Services. We will then provide a brief overview of the background and foundation that our work is built upon.

The chapter is organized as follows: section 2.2 defines the basic terms and introduces the problem domain. Section 2.3 presents our analysis of dependability issues in the context of Web Services. We will then describe our experiments involving several Web Services used in the bioinformatics domain. These experiments have helped us to understand the dependability behaviour of real-world Web Services. Finally, some classic theories and technologies for achieving dependability are discussed. Section 2.4 introduces our study of the existing work concerned with improving Web Service dependability. Section 2.5 specifically analyses dependability issues in Web Service composition. Section 2.6 concludes the chapter and summarizes the key points covered in it.

2.2 Preliminaries

Although often used, the terms *SOA* and *Web Services* are not always consistently defined. It is, however, essential here to clearly define these terms as fundamental for this dissertation.

2.2.1 Service-Oriented Architecture

In this dissertation, we follow the definitions of *SOA* and *Web Services* provided by the World Wide Web Consortium (W3C) [2]:

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

The above is a basic definition which describes what SOA is, and yet it is rather abstract: it does not make the underlying concepts and technologies it relies on explicit. It is the specification [1] that refines the definition, presenting SOA as a form of distributed systems architecture in which services implement abstracted interface for exchanging messages with clients. The machine-processable abstracted interface describes only those details of services that are important for using them. Their implementation details and internal structure are hidden from clients. The message exchange between services and clients relies on the underlying computer network, such as the Internet. The actual technologies for constructing a SOA are not made specific in these definitions and may vary in realistic applications.

2.2.2 Web Services

The definition of Web Services is given in [2] 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 clear that Web Services are a form of SOA. The definition specifically constrains the underlying technologies involved in constructing Web Services. Some of these technologies, such as the Web Service Description Language (WSDL) [16] and the Simple Object Access Protocol (SOAP) [15], have been purposefully developed for Web Services, while others have been adopted from the existing standards and protocols, such as the Hyper-Text Transport Protocol (HTTP) [14] and the Extensible Markup Language (XML) [28].

2.2.3 Dependability

In this dissertation, we start with the definition of dependability given in paper [21], a well known and widely accepted source which offers a comprehensive clarification of the basic concepts and means of *dependability* in computing systems:

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

The above definition is universally recognised in related research domains. It is, however, very abstract and brief. Paper [21] offers an alternative definition:

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 definition of *dependability*. Although it is still abstract, it precisely defines the criterion for deciding if a system is dependable. The paper specifies the attributes of dependability as *reliability, availability, safety, security, survivability and maintainability* [21]. Thus, researchers can identify and specify the means of dependability in their specific research domains according to the above taxonomy.

2.3 Dependability of SOA and Web Services

SOA and Web Service technologies have been developing very fast in recent years, becoming critical in many commercial and scientific distributed computing applications [6-9] and thus prompting a great deal of research interest in the issue of their dependability. The term *dependability* covers varied characteristics, while dependability means may vary from one context to another. It would not be feasible to cover all of its aspects in our research. In this section, we will describe the dependability means we are concerned with in our study. We will also offer a specific analysis of the dependability issues commonly manifested in the existing Web Service applications. Lastly, we will report on our studies of some relevant work conducted by other researchers working in related fields.

2.3.1 Overview of SOA and Web Services

SOA and Web Services implement standardized interoperability [13] between services and clients. These services are software components implementing capabilities and functionalities, and can be discovered and accessed via computer networks, especially the Internet. Their implementation details are invisible to clients. However, their interface needs to be defined, described and published in a machine-

processable format. The definition of Web Services specifically states that their interface should be described in the WSDL. Clients interact with them through SOAP messages relying on the underlying network protocols such as HTTP.

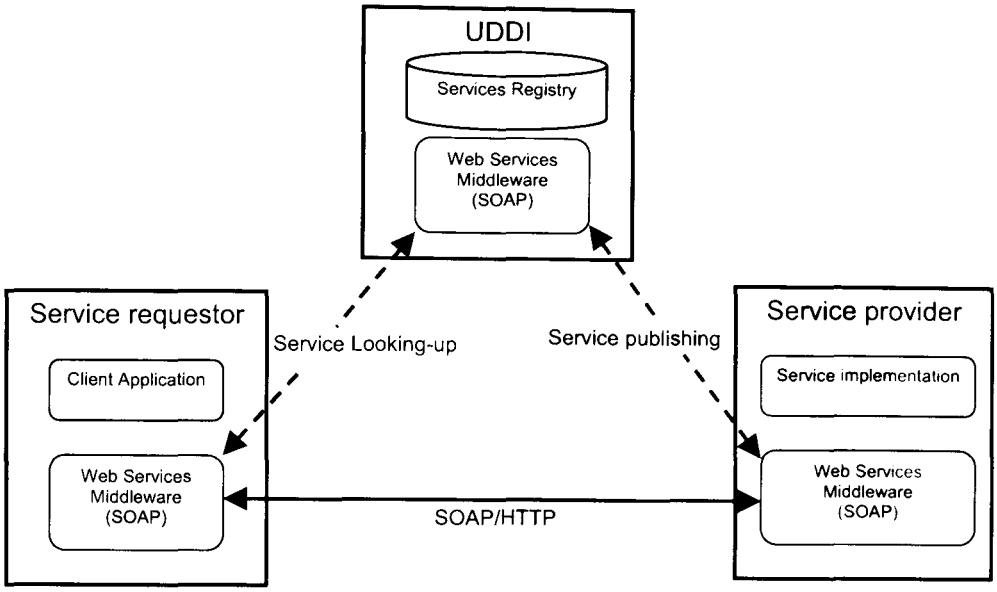

Figure 2-1: Typical interaction in Web Services

In Web Services, clients and services are assumed to be loosely-coupled, which means that they are stand-alone systems independent of each other [4]. The services are normally autonomous, and developed and deployed by different service providers. Because of the nature of Web Services, the services developed by the same service providers can also, to some extent, be regarded as autonomous of each other. Clients can discover services through various discovery services, such as the UDDI [29]. The discovered information is sufficient for implementing invocations to Web Services. The Web Service implementation details and internal structure are hidden from clients. Figure 2-1 shows the typical Web Service architecture.

In Web Services, the term *client* is often used to refer to the application software which invokes Web Services to perform business processing logic (e.g. an e-Science or e-commerce workflow), and Web Services act as clients when they invoke other Web Services to implement their internal business logic [4]. In this dissertation, the term *client* refers to the client application that invokes Web Services, unless stated otherwise. Web Service applications often rely on service composition, which integrates multiple Web Services to implement the entire business logic.

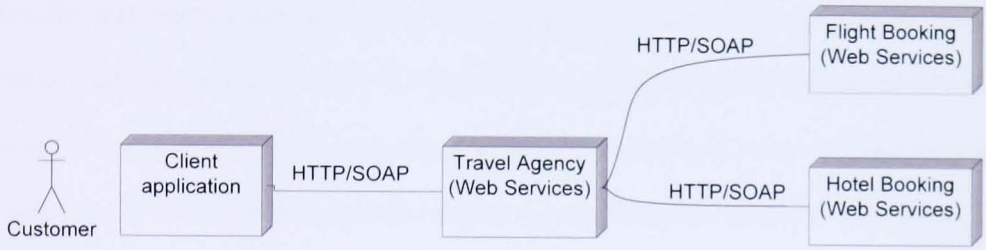

Figure 2-2: The automated travel booking process based on Web Service composition

We will use an automated *travel booking* use case (see Figure 2-2) to explain how the Web Service applications function. A travel booking procedure comprises a set of operations intended to meet a customer's request to book a journey via a travel agency for him/her. The procedure consists of the following steps: *a booking request, booking processing, booking quotation, and booking fulfilment.* To start the booking procedure, the customer sends a *booking request* to the travel agency for them to book a flight to his/her destination as well as hotel accommodation for his/her stay there. The travel agency starts *processing* the *booking* when it receives a *booking request. Processing* involves the analysis of the *booking request* placed by the customer and other internal business processing logic, including finding the appropriate flight and hotel, booking a flight with an airway company and booking a room with a hotel, registering the booking details in the database, and so on. Therefore, along with the Web Services

offered by the travel agency, the airway company and the hotel also need to provide Web Services for the relevant processes to be carried out. *Booking fulfilment* involves sending the booking reference, flight details, and hotel details to the customer. Note that in this abstract *travel booking* use case we only focus on the computing systems that are involved in the procedure, unconcerned with the details of the actual business activities.

In order to deal with the issue of possible conflicts within Web Service specifications [4], the Web Service Interoperability Organization (WS-I) [30] has instituted the Web Service interoperability profile [13] to promote and standardise the interoperability of Web Services by clarifying such specifications. It consists of some non-proprietary Web Service specifications, further refining the mechanisms defined in Web Service specifications, such as SOAP message binding, Web Service publishing, etc., to construct an interoperable Web Service infrastructure. The WS-I profile is well recognised and supported by the majority of the Web Service middleware [31-33], therefore it is safe to assume Web Services to be universally interoperable in scientific research unless there are specific circumstances to make this false. Thus, in the *travel booking* use case, the travel agency can freely invoke the flight booking and hotel booking Web Services without the service providers having to participate for the interaction to occur.

2.3.2 Dependability of Web Services

Because of the nature of their architecture, unreliability is an intrinsic characteristic in distributed systems. It is therefore essential to consider dependability issues as the architectural implication for distributed systems [1]. Many researchers are aware of this, reporting on and discussing their relevant experiences [18, 19, 34-36]. Our

experiments [37, 38], conducted upon the real-world bioinformatics Web Services (see section 2.3.3), have also revealed some important aspects of the dependability issues of real-world Web Services used in scientific applications.

Web Services implement capabilities and functionalities via computer networks, especially the World Wide Web (Web) [39]. They are typically autonomous and deployed by various companies or organizations to loosely couple with clients. The result of this manner of composition has been a wide range in the dependability characteristics of the Web Services being developed, especially those built upon legacy components. The hardware and software faults in Web Services or other internal activities can lead to failures of the client. Because Web Services are administrated by various independent providers, it is difficult to develop the corresponding handling mechanisms in the client application. For example, a Web Service can develop halt failures [21] when it is shut down without informing its clients. When the client invokes the service, an exception will arise indicating the unavailability of the service, yet without detailed information about the failure. Without collaboration from the service provider, it is difficult to implement further actions to handle the failure because of the lack of information about the state of the service. Some Web Services can return error messages to their clients, indicating an exceptional state of the service. However, these error messages are normally insufficient for implementing handling mechanisms at the client side.

The network which the Web Service infrastructure relies on is an unreliable medium [18, 19, 34]. There are many common network-related problems, such as latency of response, loss of messages, corrupted messages, traffic congestion, etc. The services can be inaccessible entirely because of network failures. For instance, paper [18]

points out that "local and network conditions are far more likely to impede service than server failures". This conclusion is further supported by paper [19]: "Network-related outages can potentially render more than 70% of the hosts inaccessible to the user. Host-related failures tend to be of a shorter duration than failures that might involve the network". The development of dependable Web Service applications thus calls for solutions capable of dealing with exceptional behaviours of individual component Web Services as well as network failures [40].

According to the classification and taxonomy proposed in papers [20, 21], the issues described above can be grouped into the following types of failures:

- Service failure: an event that occurs when the delivered service deviates from correct service.
- Network failure: An event that occurs during the exchange of messages between the client and the service, including delay, loss and change of the content of the message.

In turn, service failures can be classified as follows:

- Omission failures: The service omits to respond to an input. It can be the result of a system crash, poor system maintenance and hardware or software component failures.
- Erratic failures: Service responds to the inputs; however, the result is incorrect, or the response time is unreliable or abnormal.

Network failures can be further grouped in the following way:

- Omission failures: message lost during an exchange of messages.

- Timing failures: unusual network latency during an exchange of messages.
- Content failures: the content of the message changed during an exchange of messages.

2.3.3 Our experiments on the dependability of Web Services

To analyse the dependability of realistic Web Services, we have conducted some experiments with real-world Web Services, developing a Web Service dependability Assessment Tool (WSsDAT) in order to assess Web Service dependability [37]. The tool can continuously monitor a number of Web Services and generate metrics from the monitoring results to present the dependability characteristics of the services. More details about the WSsDAT tool can be found in Appendix A. Some of the experiments, in which the tool was used, are reported in papers [37, 38].

Here we briefly report the experiment with two BLAST Web Services, commonly used in Bioinformatics research [41], which provide similar functionalities. In the experiment, we used the WSsDAT to monitor the BLAST Web Services from three locations simultaneously to observe the differences in their behaviour and how the locations (networks) affect the dependability. Below are listed the two Web Services:

- EBI BLAST Web Service[1], deployed by the European Bioinformatics Institute (EBI), Cambridge, UK [41]
- DDBJ BLAST Web Service[2], hosted by the DNA Databank, Japan [42]

Two WSsDAT tools were located in Newcastle upon Tyne, UK: one was deployed from the campus network at Newcastle University, whilst the other one was hosted on

[1] http://www.ebi.ac.uk/collab/mygrid/service4/soap/services/alignment::blastn_ncbi?wsdl
[2] http://xml.nig.ac.jp/wsdl/Blast.wsdl

a computer connected to it with 1MB broadband via a domestic Internet Service Provider, Telewest Broadband (UK) [43]. The remaining WSsDAT was deployed in the China Education and Research Network (CERNET) in Tianjin [44].

In order to observe the variances of the dependability and performance metrics over different periods - during working days, the weekend, daytime and night time - the two BLAST services were monitored continuously for over a month. Here we report a set of data collected from Friday, March 18, 2005 until Sunday, March 20, 2005. The total duration was 72 hours and the interval between the successive service invocations was 30 minutes. All measurements were stored in a database for further analysis.

During the experiment, the EBI BLAST service behaved very erratically. Below is a report of the results collected concerning the service:

- Successively tested 132 times in 72 hours at each location
- Domestic Broadband (Telewest), Newcastle Upon Tyne, UK
    - Average response time: 842.1s (239s ~ 760s)
    - Failure rate: 58.3% (76 invalid results)
- Newcastle University Campus Network
    - Average response time: 764.6s (240s ~1000s)
    - Failure rate: 62.9% (82 invalid results)
- CERNIC, China
    - Average response time: 945.7s (261s ~1886s)

- Failure rate: 43.2% (56 invalid results)

All of the failures were caused by the EBI service returning the SOAP message, with the error message "Gateway failure" attached. The error message seemed to indicate the failure of an internal service component. However, without collaboration by the service provider we do not have information about the failure.

In contrast, the dependability of the DDBJ service was very good during the experiment, with no failures recorded. There were two delays registered at each of the three roots, indicating unknown states of the service or some part of the network.

- Successively tested 132 times in 72 hours at each location
  - 100% successful
- Domestic Broadband (Telewest), Newcastle Upon Tyne, UK
  - Average response time: 103.1s
  - Delays: 180s, 728s
- Newcastle University Campus Network
  - Average response time: 97.8s
  - Delays: 369s, 925s
- CERNIC, China
  - Average response time: 130.0s
  - Delays: 397s, 940s

Figure 2-3 shows the charts drawn from these results. Our experiment shows that the dependability of a BLAST service can vary dramatically. This empirical conclusion can be extended to the global Web Service infrastructure, where the dependability of services are all different from the user's perspective [18, 38].

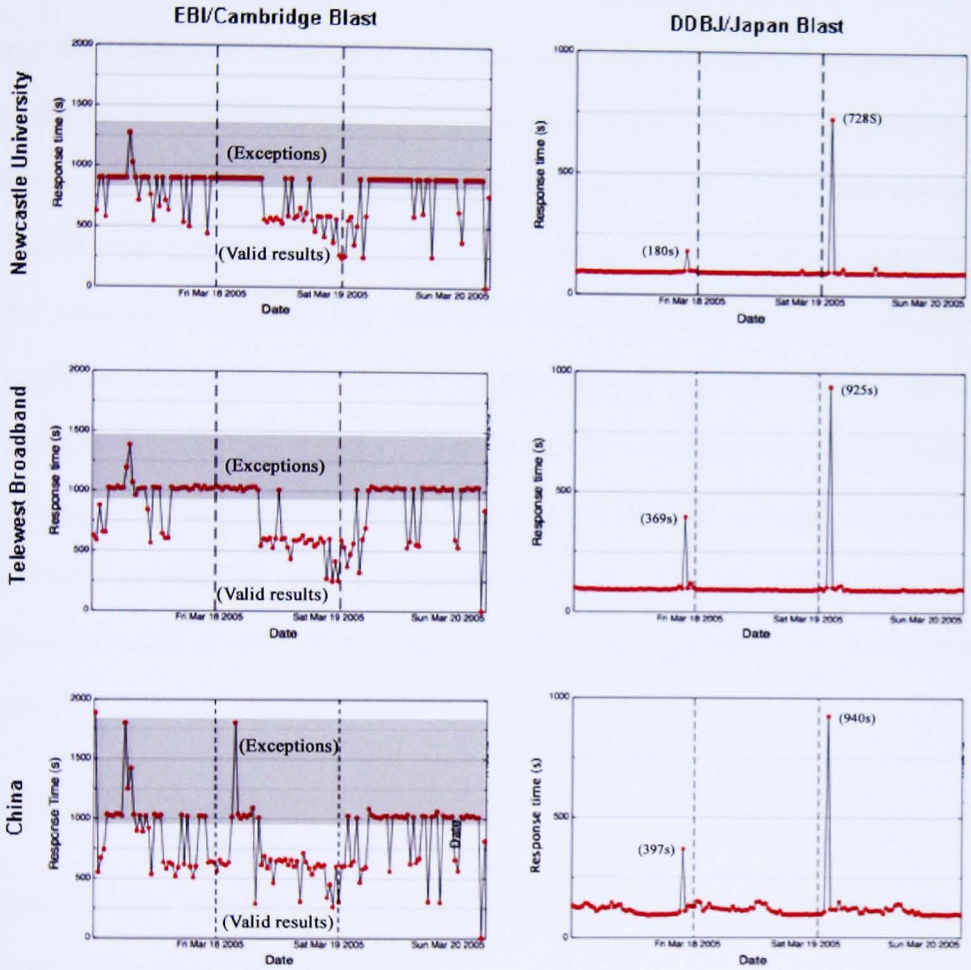

Figure 2-3: Performance metrics obtained using the WSsDAT from the BLAST services deployed at the EBI and DDBJ when invoked from the University of Newcastle campus network, a commercial broadband supplier (UK) and from China. Service failures have been shaded in grey.

With the superior richness of services offered by SOA, Web Service applications extensively use this diversity to improve the dependability of service composition (see, for example, the solutions proposed in [40, 45-48], to name a few). This strategy is based on the fact that, in SOA, different service providers may provide similar services which can be used as redundancy and alternatives to each other. We believe that the information collected in our experiments can be used to understand the behaviour of the BLAST Web Services and thereby allow scientists to select those that are the most reliable for use in their data analyses. This makes it possible to select Web Services from among similar services based upon their dependability behaviour. Our experiments indicate that, based on the comparison of its dependability characteristics with those of the EBI BLAST service, the DDBJ BLAST service should be the first choice for users. Furthermore, the fact that it is possible to deploy and use the WSsDAT in different physical locations can lead to insights on how the network can affect the dependability and performance of Web Services, pointing towards the idea of on-location monitoring of Web Service dependability at the client side.

2.3.4 Means for Achieving Dependability

There are many techniques used to achieve dependability. Paper [21] groups them in the following categories:

- Fault prevention
- Fault tolerance
- Fault removal
- Fault forecasting

Current research on the dependability of Web Services implements the above approaches - individually or in combination - to deal with different types of failures [21].

Fault prevention can eliminate a number of faults hidden in the design and implementation of the system. It has to be applied during the system design stage by employing quality control techniques such as modularization, structured programming, etc. [21].

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 remaining faults. It typically consists of two phases: error detection and system recovery [21]. Error detection is used to identify the presence of errors, whilst system recovery is aimed at, by applying error and fault handling, 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, whilst fault handling prevents faults from being activated again [21, 49].

Fault forecasting performs qualitative evaluation of component failures and quantitative evaluation of the probability of failures with respect to fault occurrence or activation. The dependability attributes of a system may change during the life cycle of the system because of system aging. 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 the implementation of fault tolerance mechanisms. More examples can be seen in papers

[50, 51], which report how to use the fault-injection technique to assess the dependability of Web Services.

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 [21].

2.3.5 Fault tolerance in SOA

With their complex architecture and complicated application scenarios, Web Service applications are doomed to a potentially high rate of failures. This calls for a variety of methods to be designed to minimize failures occurring in Web Services and in their interaction with clients. Nevertheless, faults can never be completely removed from real-world systems, nor can the occurrence of errors be ever entirely prevented [22]. In this respect, the application of appropriate fault tolerance (FT) techniques is critical for improving the dependability of Web Service applications. Generally speaking, in fault tolerance, system recovery consists in error handling and fault handling. Error handling may involve the following forms [21]:

- Rollback, which brings the system back to a correct state saved at checkpoints before the occurrence of errors.
- Roll forward, where the state without detected errors is a new state.
- Compensation, where the erroneous state contains enough redundancy to enable errors to be masked.

Fault handling prevents located faults from being activated again, by employing the following steps [21]:

- Fault diagnosis, which identifies and records the location and type of cause(s) of error(s).
- Fault isolation, which excludes the faulty components from service processing.
- Reconfiguration, which switches service processing from faulty to redundant components.
- Reinitialization, which sets the new system configuration.

The selection of the fault tolerance techniques strongly depends on the fault assumptions made, and mostly lead to two basic fault tolerance strategies: backward and forward recovery [21, 52]. Backward recovery typically implements the recovery block fault tolerance technique [52, 53] to maintain the continuity of the service in spite of faults. If errors occur during the transaction, the system rolls back to a previous correct state, and then applies a retry or service diversity to tolerate the faults.

In contrast to backward, forward recovery transforms the system into a correct state. It mainly relies on exception handling [20] techniques to tolerate errors occurring during transactions. Exception handling mechanisms can be found in many mainstream programming languages, for example Java, C++, and etc. They provide methods and tools to handle exceptional states and activities during the execution of software so as to achieve more reliable and robust software and systems.

N-version programming [54] is an important compensation technique, typically employed in dependability-critical applications. It is used for tolerating design and implementation faults. The approach requires multiple versions of software or components to be developed by independent developers to identical specifications. Although it is still impossible to avoid all of them, the approach can sufficiently

minimize the probability of common faults, thereby improving the reliability of system software [55]. In practice, however, the cost of applying the N-version programming approach is high and its effectiveness often overestimated, resulting in misjudgements of the reliability of the software or the system [55].

In the context of SOA, there has been some research focusing on applying the Recovery block [52, 53] and N-version programming [54-56] techniques, which employ the diversity approach to implement fault tolerance mechanisms. This normally includes service and messaging path diversity.

Diversity is a natural advantage of Web Services because of their loosely coupled architecture and standardised interoperability. Several Web Services implementing similar functionalities are likely to be found in the growing Internet world, and can be used for implementing service diversity. Furthermore, there is normally path diversity to be found on the Internet. A lot of applications [46, 47, 57] utilize similar services to implement the diversity approach. In Recovery blocks, diverse services can be used as alternatives replacing the faulty services to maintain continuous service. The approach can be especially beneficial for employing N-version programming in an application, with the development cost dramatically reduced by using the existing services as redundancy. This strategy may potentially be at risk from the problem of common faults, whereby the services may share the same faulty services as external component services. However, the probability of such problems can be minimized by applying appropriate techniques, such as the solution proposed in paper [58].

2.4 Overview of the Existing Work

As part of our research of Web Service dependability, we have studied the existing work, focusing on improving Web Service dependability and constructing dependable Web Service applications. Such solutions typically rely on the techniques outlined in section 2.3.4. There are too many different factors in the dependability of Web Services, and it is impossible to deal with all kinds of faults in one solution. Therefore, various approaches have been developed based upon particular fault assumptions.

In general, depending on their purposes, these can be classified into two categories: one aimed at developing dependable Web Services, and the other at dependable applications based on Web Service composition. Approaches of the first kind adopt various dependability-attaining techniques in service design and development to improve their dependability. According to their fault assumptions and the implementation of dependability-attaining techniques, many of them can be classified as application-level protocols, exception handling, system diagnosis and modelling, etc. Approaches of the second type often adopt service diversity and dynamic reconfiguration of service composition to improve the dependability of the entire application. These solutions are typically complex. Most of them implement the broker/proxy-type architecture and apply multiple dependability-attaining techniques in different combinations to deal with various types of faults. Below we will briefly introduce some typical work to summarize the current state of research in this domain.

2.4.1 Application-level Protocols

Current W3C Web Service specifications do not define standards and mechanisms to guarantee the Quality of Service (QoS) and dependability of Web Services. Additional protocols and standards have been developed to standardize the implementation of QoS and dependability mechanisms. Such protocols and standards particularly focus on application-level messaging dependability in addition to the lower-level network protocols, most commonly HTTP [14]. The Service Reliability (WS-Reliability) specification [59] is one of such solutions, which has been formally declared as an OASIS [59] standard.

The WS-Reliability defines a protocol that guarantees the reliability of SOAP message delivery. It can cope with failures of software components, the system and the network during message delivery between distributed applications. This application-level messaging protocol is designed to prevent duplicates and loss of messages, and to guarantee message ordering. It cannot, however, deal with service failures or unavailability of particular services. Therefore, it requires upper-level fault tolerance mechanisms to deal with other types of failures.

2.4.2 Exception Handling Approaches

Exception handling is a classic fault tolerance technique [20]. Solutions based on it implement exception handling mechanisms to cope with errors occurring in Web Services, therefore achieving a highly dependable individual Web Service. Some of these emphasise the tolerance of internal hardware and software faults, while others also deal with network failures.

AmberPoint Inc. [60] presents a solution for managing exceptions in a commercial Web Service environment. The solution implements an intermediary-based Exception Manager (EM) to detect run-time exceptions in a set of Web Services. The EM executes localized resolutions to deal with exceptions. The approach overcomes the shortcomings of the traditional programmatic exception handling mechanisms applied in the context of Web Services.

Salatge and Fabre [46] introduce a connector-based solution for ensuring the dependability of Web Services for clients. It proposes a special language for implementing fault tolerance connectors to couple services and clients. Clients, Web Service providers or dependability experts can implement the connectors in their applications. The connectors implement error handling mechanisms to deal with failures and exceptions during communication between clients and services. They can also collect error information during execution in order to monitor the health of Web Services. In addition to the above techniques, the service redundancy strategy is also employed in this solution, based upon the Ontology technology. The solution can improve the robustness of communication between clients and services. It is especially suitable in developing a Web Service application in which clients and service providers are correlative and can efficiently cooperate in implementing connectors.

Dobson [61] proposes a container-based approach to fault tolerance in SOA. This work is based on the assumption that, in SOA, services may fail for many reasons, including resource starvation, faults in implementation and network instability. The authors have developed a notion of fault-tolerant service container, an extensible architecture, to employ component diversity in a SOA application. The container is

configured with a fault tolerance policy. It allows the use of fault tolerance mechanisms to leverage the existing services at the application level. A software development kit (SDK) and a deployment tool are developed to implement the container. This container-based approach addresses the problem of the traditional hardware redundancy strategy commonly adopted by service providers. It achieves redundancy at the service level, allowing both software and hardware redundancy. The approach can employ service diversity by binding services available at a service marketplace. In this way, service redundancy can be achieved at low cost. The container acts as a proxy to the actual services. It intercepts messages transmitted between the client and the services and applies exception handling techniques to deal with failures of services. Such message interception is transparent to both the client and service provider, and controlled by the fault tolerance policy model. The fault tolerance procedures in the container implement the actions of fault tolerance policy models.

The solutions based upon exception handling techniques can improve Web Service dependability and/or the interaction between services and clients. They are often highly application-specific and especially suitable for those service providers which offer dedicated client-applications to their clients to improve the usability of their services. As exception handling mechanisms need to be developed in the design and implementation stages, such solutions can hardly benefit the existing legacy Web Services without modification. Users may be able to employ them for implementing their client applications; this, however, requires collaboration from providers.

2.4.3 System Diagnosis Approaches

In developing systems, some approaches apply diagnosis and assessment techniques to achieve highly dependable Web Services. These approaches commonly implement system diagnosis and assessment mechanisms to assess the dependability of internal and external system components, and act upon diagnosis results to avoid failures.

Ardissono, Furnari, Goy, Petrone and Segnan [62] present an approach relying on consistency-based diagnosis aimed to achieve intelligent exception management. This approach applies fault tolerance to compose Web Services by implementing exception handling which relies on smart failure identification and diagnostic information-aware exception handlers. In addition to the traditional model-based diagnosis approaches, this work allows local diagnosers to analyse exceptions that arise in each component Web Service and to extend the diagnostic-reasoning information in the business logic description of each component Web Service. A global diagnoser is then introduced to conduct global reasoning. It identifies the causes of exceptions by consulting the local diagnosers. The existing component Web Services need to be modified so that they can interact with the corresponding local diagnosers and achieve diagnostic information awareness.

Vieira, Laranjeiro, and Madeira [50] propose a fault injection technology for assessing Grid Web Service dependability. The authors have developed a fault injection toolkit, which allows network-level fault injection for real-time middleware message interception and fault injection. The toolkit can precisely inject specific rather than random faults into middleware messages, which makes it valuable for assessing Grid middleware for constructing dependable Grid applications. The toolkit can also be used as a tool to test individual Web Services.

The above summarises some typical approaches based upon system diagnosis and assessment. Such approaches can help developers to build highly dependable Web Services, such as dependability-critical applications where service dependability is vital. It is difficult to apply such solutions in the existing systems, and the development cost of such solutions is quite high.

2.4.4 Approaches to Dependable Service Composition

The solutions aimed at improving the dependability of Web Service composition typically implement the service broker architecture and fault tolerance mechanisms. They intercept communication between the client and Web Services and act upon exceptions and failures to maintain service continuity. As for those applications that integrate Web Services dynamically discovered from registries and invoke them according to their WSDL interface, it is difficult to implement specific fault tolerance mechanisms to ensure the dependability of service composition because of the lack of information. In such circumstances, functionally similar Web Services are often used to employ the service diversity strategy.

Alwagait and Ghandeharizadeh [45] propose a dependable Web Service framework (DeW) for solving problems caused by service migration. When a Web Service migrates to a different location or gets disconnected from the Internet, clients typically have to manually rediscover the service or its replicas from the UDDI and modify their application code to invoke them to the new location. The DeW implements Web Service registry proxies to automatically re-direct the client's invocation of a service to the old location to the new location of the service or its replicas. When a Web Service migrates, the service provider can register the new location of the service or its replica in the DeW. When the client invokes the service

using its old location, an exception will rise. The exception will be handled by the DeW proxy, which will find the new location of the service or its replicas, and redirect the client's invocation there.

Laranjeiro and Vieira [48] propose a mechanism for adopting service diversity into composite Web Service applications. It simplifies the implementation of service redundancy commonly applied in the context of Web Service architecture. The mechanism, called Fault tolerant Web Services (FTWS), allows programmers to specify alternative Web Services for each operation and offers a set of artefacts that simplify the software design and coding process. It is able to deal with all aspects related to the redundant Web Service invocation and responses voting, as well as evaluating and comparing the alternative services. The evaluation procedure generates data for resolving voting impasses. When developing a SOA application, programmers normally have to select component Web Services and redundant alternative Web Services when constructing composite ones. It is their job to code all the service redundancy and voting mechanisms. Such procedures are typically error-prone. With the FTWS deployed as a proxy Web Service, it can automatically deal with all aspects related to service redundancy and responses voting. In short, it is an off-the-shelf proxy Web Service that implements service redundancy and voting mechanisms to simplify the development of composite Web Services.

Tsai, Song, Paul, Cao, and Huang [47] propose a framework that extends the existing Web Services to achieve dynamic reconfiguration for Web Services. It can perform automatic reconfiguration of participating services at run-time to cope with service unavailability, network inability as well as software and hardware failures. This framework extends the current WSDL interface specification, specifying a service by

its interface, scenarios and constraints (ISC), i.e. representing its actors, conditions, data, actions, timing and events (ACDATE). The ISC specification specifies the static and dynamic structure of services.

The authors have developed a run-time distributed dynamic reconfiguration tool based on the ISC. The Dynamic Reconfiguration Service framework (DRS) uses the ISC specification for improving Web Service dependability, maintaining a service registry for monitoring and managing registered Web Services. It is implemented and deployed with redundancy to avoid a single point of failure. Multiple DRSs can be deployed in each system layer, communicating and synchronizing with each other to enhance the dependability of the framework. Every DRS has a Service Directory (SD) and a Standard Service Naming Directory (SSND) for managing Web Services and needs to interact with services providers to obtain information for them. The DRS can track the status of participating Web Services and rank them according to user feedback reports from participating agents. It generates a proxy agent for each abstract node in its SD. When the client invokes a participating Web Service, it is the proxy agent rather than the actual address of the service that is invoked. The DRS implements auditing agents to monitor the status of participating services at run-time and to generate a profile for each active service. With the DRS performing dynamic reconfiguration at run-time, if a participating service becomes unreliable, the client's invocation can be automatically switched to an alternative service.

Townend, Groth and Xu [58] propose a provenance-aware weighted fault tolerance scheme for developing dependable Web Service applications. This approach identifies common-mode failures in applications using multi-version design. It introduces a provenance system to record the flow of data from a service to identify shared

services. The recorded provenance information can be used to determine weighting of the results delivered by each service for result voting. The results from those services whose weightings are below the threshold are eliminated from the voting procedure. A Java-based Web Service implementation of the Provenance Recording Protocol, called Provenance Recording for Services, is implemented to support a provenance-aware SOA.

The service broker architecture was popular in the conventional distributed applications, such as the message broker in EAI and the object request broker in CORBA [4]. In these systems, the service broker was the key service component for performing service integration. The client's business logic depends on the service broker for interaction with participating component services in order to execute business processes. However, the service broker can at the same time cause problems in developing cross-organizational applications because of its lack of ability to integrate autonomous component services. Because of their standardized interoperability, these limitations do not apply to the service broker in Web Services. Therefore, the dependability-improving service brokers proposed in the above solutions are feasible in Web Service applications. In fact, the Web Service specification [1] describes a Web Service called Web Service intermediary which develops value-adding services between the client and Web Services, and which can be used to implement service brokers in the way fully compliant with Web Service specifications. Unfortunately, the potential of this architecture is not recognised in the above solutions, where the researchers develop their own architecture to implement service brokers. As a result, these solutions can hardly be seamlessly integrated into the existing applications, and they do not support on-the-fly dynamic service

integration that would allow new component services to be integrated in service composition without recompiling the client applications and the service broker.

2.5 Problems Involved in Web Service Composition

Among the many studies aimed at improving Web Service dependability, those developing dependable Web Service composition constitute a significant part, emphasising how important it is to ensure the dependability of applications based on service composition. However, although the existing work has addressed certain dependability issues effectively, there are still some problems remaining.

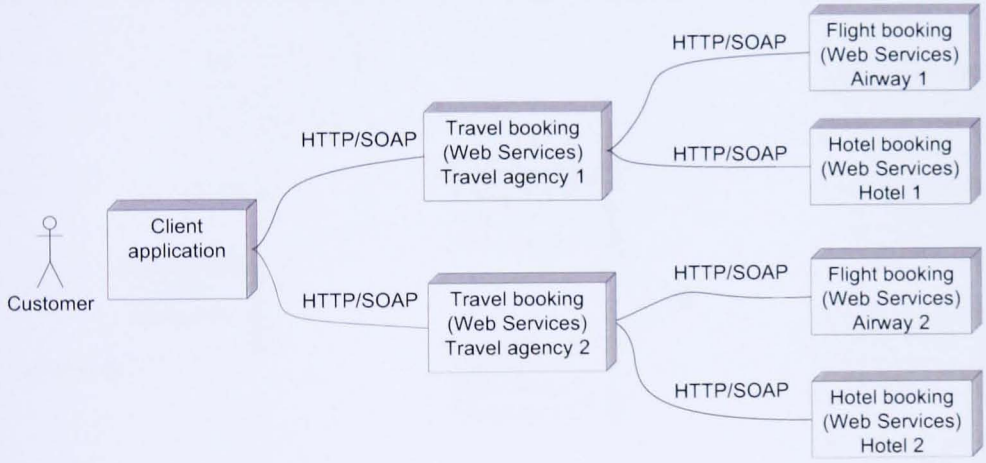

Figure 2-4: The automated travel booking process with multiple travel agencies

Web Service composition relies on multiple component services to implement entire business processes. These component services are developed and administrated by different service providers. In reality, there is no guarantee that all component services are highly dependable. For instance, in the *travel booking* use case, by employing appropriate dependability solutions the Web Services provided by the travel agency and the airway company can be developed in such a way as to meet a

high dependability standard because this is essential for these businesses. However, it might be seen as less important to the hotel business, with the development of highly dependable Web Services restricted by a limited budget. Therefore, the dependability of the entire travel booking process can be eventually undermined by undependable hotel booking Web Services.

In such circumstances, it is well worth employing service diversity strategy to develop a client application. As there are several travel agencies offering the same business, the client can send *quotation requests* to multiple agencies, booking the journey with one of them (see Figure 2-4). Thus, things become less problematic to the customer, as long as one of the travel agencies can eventually complete the booking process.

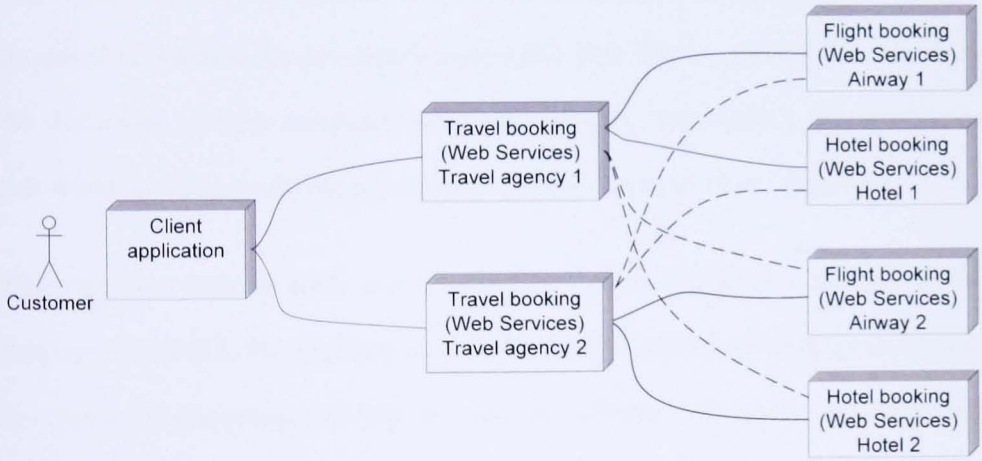

Figure 2-5: The automated travel booking process with multiple travel agencies implementing service diversity. The solid lines represent primary routes and the dashed lines alternative routes.

However, the situation is very different for the travel agencies from what it is for the customer. The travel agencies have to compete with each other, and the dependability of their services is their key to success (note that we are not concerned here with other

business factors, such as price, service quality, etc.) Therefore, the travel agencies also need to build service diversity into their *travel booking* services, to prevent their business from failing due to undependable external component services, such as the Web Services provided by the participating business partners, and the network needed to access them. In a scenario, the use case illustrated in Figure 2-4 may turn into that in Figure 2-5, in which both travel agencies (TA), TA1 and TA2 use the same Airways (AW), AW1 and AW2, and hotels (HT), HT1 and HT2, as external services. However, these Web Services have different dependability characteristics. The selection of the appropriate components during service composition is one of the most important elements in defining the dependability of the entire application.

The service diversity strategy and the proxy/broker architecture have been extensively employed in solutions for developing dependable Web Service applications. However the limitations of those solutions have restricted their applicability and efficacy in real-world applications. In the following, we discuss some of these limitations.

There are two ways to apply service diversity: service alternatives as used in the Recovery block [52, 53] fault tolerance technique and service redundancy as used in N-version programming [52-55]. In this dissertation, we draw the following distinction between them:

- Service alternative: component services are used as alternatives to the primary service, and the business logic processor only invokes them when the primary service fails to deliver valid results.
- Service redundancy: component services are used synchronously, the business logic processor invokes them at the same time and processes the results returned from them according to certain preference.

The above diversity strategies have been employed in some of the existing solutions. However, to the best of our knowledge, the existing work does not provide features for making justified selection of the diversity strategies and component services.

In practice, it is difficult to choose which diversity strategy to use, because their applicability largely depends on the environmental variables, such as network bandwidth, system capacity, etc. [36]. These variables are especially restrictive in the service redundancy approach. It may straightforward applying the approach to the simple business model illustrated in Figure 2-5, yet as the number of redundant component services grows, the approach becomes less applicable, possibly undermining the dependability of the application [36]. We believe the above issue was not sufficiently addressed in the existing work.

Many solutions employ the service alternative diversity strategy, because of its simplicity. However the strategy for selecting the component services is seldom discussed. Obviously, which primary component service is selected mostly defines how efficient and feasible the service alternative approach will be. A highly dependable primary service can benefit the performance of the entire service composition. Unfortunately, to the best of our knowledge, there is no satisfactory solution currently to help application developers to select component services. Although some solutions implement service ranking mechanisms, such as in [47], there is not enough information to reflect the changing behaviour of Web Service dependability. Moreover, computer networks play a very important role in Web Services, with the dependability of the computer network between the client and services crucial for service composition. The dependability of a Web Service may

change dramatically from one client's perspective to another's, because of the different networks between clients and the service provider.

Many solutions use similar services to implement service diversity. However even though the candidate services provide similar functionalities, their interfaces, required input parameters, etc. can be very different. Some solutions propose interface mapping mechanisms to deal with the issue; in addition to the difficulties of implementing and maintaining such mechanisms and mapping registries, these approaches often undermine the compatibility with some Web Service security mechanisms [13]. For example, it is unlikely that an encrypted SOAP message provided by the client can be decrypted by all candidate services, and that a security key issued by a service will be accepted by other services. For similar reasons, those approaches are often inapplicable for the stateful Web Services [3], whereas if a service fails in the middle of the business logic process, diverting the client's request to other candidate services will cause problems, because they do not contain the states or their internal business logic implementations can be very different.

We can now summarise several problems still existing in Web Service composition which have not yet been satisfactorily dealt with in the relevant work:

- Dynamically selecting appropriate fault tolerance mechanisms
- Dynamically selecting diverse component services in corresponding mechanisms
- Failures of component services undermining the dependability of service composition

[3] http://xml.coverpages.org/statefulWebServices.html

- Network failure can undermine the dependability of Web Services from the client's perspective
- Compatibility with Web Service security mechanisms
- Compatibility with stateful Web Services.

2.6 Conclusions

The dependability of Web Services is an active and important research domain. The loosely-coupled distributed architecture of Web Services has brought benefits for developing e-Science and e-commerce applications. However, such architecture is inherently undependable. Research on the dependability of Web Service applications needs to deal with both service failures and network failures. It is also very important that such solutions need to be compliant with the Web Service specifications [1] and the WS-I interoperability profile [30]. There have been many approaches developed to ensuring the dependability of Web Service and service composition. However, our analysis shows that the limitations of those solutions restricted their applicability and efficacy. There is a need for solutions to help develop dependable Web Service applications. We conclude that such solutions will need to improve the dependability of the existing legacy Web Services for clients without modifying them, thus benefit clients whose applications rely on the services dynamically discovered from the UDDI or other registries and employed in their applications. This can minimize the development cost whilst fully utilizing the richness of services in the Web Service world. New solutions are needed to improve the dependability of Web Service applications from the user's perspective to minimize the problems caused by service and network failures. New techniques are also required for improving the efficiency of such solutions by explicitly utilizing service diversity strategies and using the most

dependable components to ensure dependable service composition. Moreover, the solutions should have better compatibility with Web Service security mechanisms and stateful Web Services. The above considerations motivated our research on improving the dependability of Web Services.

3. The WS-Mediator System

3.1 Introduction

In this chapter, we present the WS-Mediator approach. Generally speaking, the WS-Mediator is a Web Service intermediary system which implement an overlay architecture [63-65], resilience-explicit computing [27] and fault tolerance mechanisms to improve the dependability of Web Service composition. It explicitly mediates clients' requests to Web Services in accordance with the dependability behaviour of these services and of the communication media (the Internet). The WS-Mediator is implemented as a distributed network of dedicated services (called Sub-Mediators) which allows monitoring of the dependability of the Web Services from different locations. Monitoring results are used to dynamically generate and update the dependability metadata of these Web Services, which makes it possible to achieve explicit dynamic adaptation of Web Service composition at run-time. The system can be seamlessly employed by applications, to provide off-the-shelf (ready-made) fault tolerance mechanisms for improving the dependability of service composition without modifying component services. This is especially beneficial for integrating autonomous Web Services.

The chapter is organised as follows. Section 3.2 defines the objectives of the solution, while section 3.3 overviews the architecture of the WS-Mediator system. Section 3.4 explains the structure and internal components of Sub-Mediator, and describes the design principle of the WS-Mediator system in detail, with a particular focus on the functional components. Section 3.5 demonstrates how to use the WS-Mediator system in applications. Finally, section 3.6 concludes this chapter and highlights its main contributions.

3.2 Research Objectives

In the previous chapter, we briefly overviewed relevant work on improving Web Service dependability, highlighting the problems that have not been sufficiently addressed in the existing solutions, which do not fully explore the impact of the Internet and the quality of the service received by clients. Some solutions allow clients to utilize service diversity in their applications. However, they neither support justified selection of the diversity strategies nor select the component services dynamically according to their changing dependability behaviours. Moreover, the client application and the service brokers implementing these solutions often need to be recompiled every time new component services are added to the composition schema. Besides, these solutions tend to require a degree of collaboration from service providers as additional information has to be obtained to implement relevant mechanisms [46, 58]. This is, however, rarely suitable in cross-organizational applications, thus eliminating the applicability of these solutions.

Yet ensuring the dependability of service composition with autonomous Web Services is an important issue. Motivated by the problems described in section 2.5, our work aims to tackle them, and accordingly we define the objectives for our approach in the following way:

- To propose a solution to improving the dependability of Web Service composition, which can maintain the continuity of services despite failures of component services and network.
- This solution should be compliant with the Web Service specifications and interoperability, and support on-the-fly dynamic integration of component services according to their dependability characteristics.

- To make it possible to carry out an easy dynamic integration of new component services to business logic to employ service diversity in service composition.
- To develop a dependability monitoring mechanism to assess the dependability of component services from the client's perspective and generate dependability metadata representing the dependability behaviour of component services.
- To provide off-the-shelf fault tolerance mechanisms and dynamic reconfiguration of these to deal with various fault assumptions.

As a result of our research, we have developed an architectural solution achieving the above objectives. Below we will present the approach in detail.

3.3 Overview of the WS-Mediator

Our solution, the WS-Mediator (Web Service Mediator) system, realizes an off-the-shelf mediator architecture [66] to ensure the dependability of Web Service applications. The WS-Mediator system implements the Web Service intermediary architecture [1]. Being autonomous of the client, it mediates between the client and Web Services to ensure the continuity of services by employing resilience-explicit computing and fault tolerance mechanisms.

The term *Resilience-Explicit Computing* refers to "the explicit use of information (metadata) on the resilience characteristics of system components, infrastructure and environment to guide decision-making at either design time or in the running system" [27, 63, 65]. Resilience-explicit computing is specifically addressing dependability issues in SOA to achieve highly dependable SOA applications.

In theory, resilience-explicit computing originally refers to the situation in which a client imposes a dependability requirement when attempting integration with services, whilst the services present dependability metadata at their interface [65]. In practice, the above service lookup and integration process can be carried out by introducing into the architecture a special service that can mediate between the client and the services to match the dependability requirement of the client and the dependability metadata of the services by employing explicit reasoning about service composition. In the current Web Service technology, there is no standard definition of how dependability metadata should be presented at the Web Service interface, nor is there a standard way to implement them so that they can be universally understood by the client. A special service should therefore be developed to resolve this issue. This could, for instance, behave as a service coordinator between the client and the services, and implement a conversion mechanism to convert the dependability metadata from different services to a standard format that can be understood by the client.

Our WS-Mediator approach followed the above route, extending it to adopt some concepts and mechanisms from adaptive fault tolerance technology [67, 68], which has already been applied in developing dependability-critical applications (e.g. [69]) for many years, to resolve the dependability issues in Web Service composition.

In SOA, from some perspectives the distinction between a service provider and a client is blurred. When it invokes other Web Services, a service provider acts as a client [4]. The WS-Mediator monitors the dependability of Web Services and generates dependability metadata from monitoring results. The system overlay architecture [63-65] allows the subsystem, i.e. Sub-Mediators, to be deployed at

various locations in the Internet. In practice, the Sub-Mediator can be deployed at the same root where the client application executes. Thus, Sub-Mediators can perform on-location monitoring of component services to consider the network impact. The notion of *on-location monitoring* implies that it is performed at the client side by distributed Sub-Mediators to realise the dependability behaviour of Web Services from the client's perspective (see Figure 3-1). Sub-Mediators can also utilize the overlay architecture to implement message-routing strategies to deal with network-related faults. The dependability Web Service metadata are used by the resilience-explicit dynamic reconfiguration mechanism to make decisions about which Web Service to select as the most appropriate for performing dynamic service composition during the business procedure. This novel approach improves the efficiency and feasibility of service diversity by applying it according to the dependability of component services. The system does not limit the selection of candidate component services, allowing new component services to be introduced into service composition without modification or recompiling of any of its service components. Clients can flexibly provide a number of candidate Web Services at run-time for implementing service diversity.

Unlike the existing solutions (e.g. [46-48]), our approach does not create additional difficulties for adapting systems to their applications. Furthermore, the system provides integrated off-the-shelf fault tolerance mechanisms corresponding to various fault assumptions and application scenarios, to be integrated into the client application at run-time, thereby reducing the development cost of a dependable service composition.

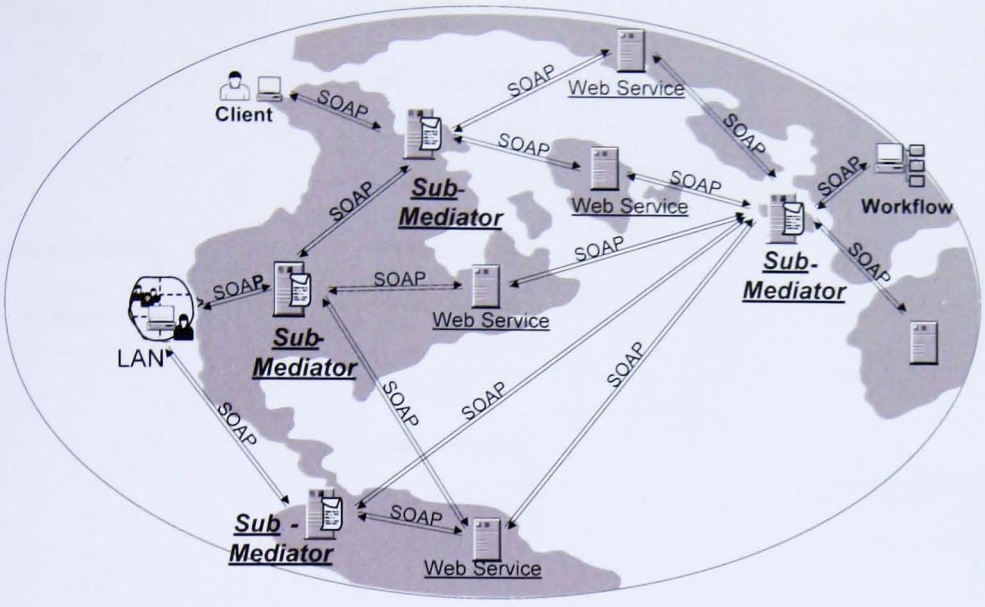

Figure 3-1: The overlay architecture of the WS-Mediator system allows monitoring the dependability of Web Services from different locations by a dedicated global network of Sub-Mediators. The system helps the clients to dynamically select the best Web Services for service composition, and apply fault tolerance mechanisms to ensure dependable applications.

The flexible and scalable architecture of the WS-Mediator allows it to be easily tailored for various specific applications. There are many ways to deploy Sub-Mediators - for example, they can be deployed on a local network, to be shared by local clients; or a virtual organization could deploy a Sub-Mediator on each node of the framework to construct the WS-Mediator system. A company could deploy a number of Sub-Mediators at different locations to utilize the WS-Mediator architecture so as to improve the dependability of their services for globally distributed users. Figure 3-1 illustrates the general architecture of the WS-Mediator system. Below we will explain its architecture and system components in detail.

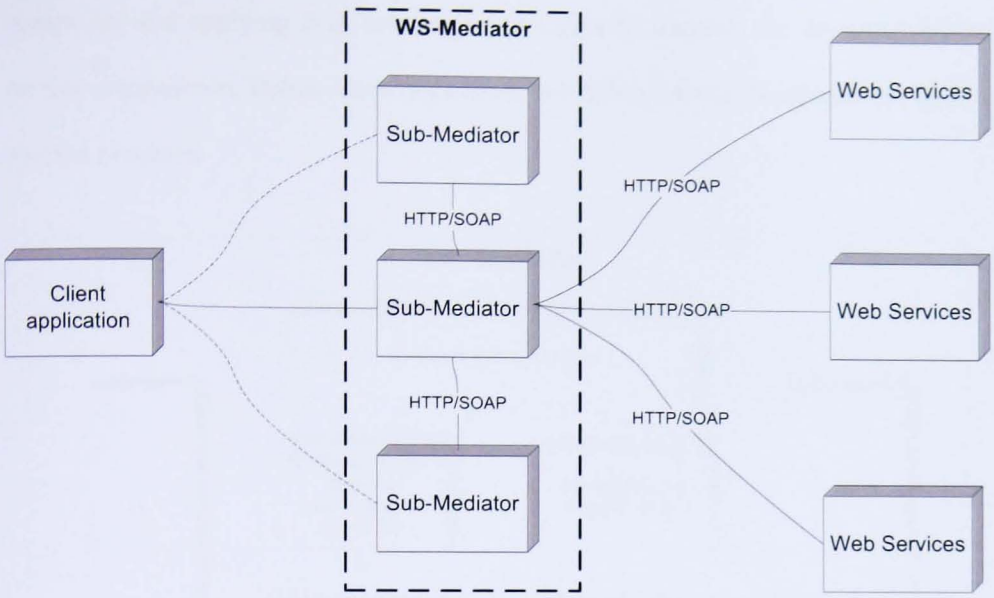

Figure 3-2: Deployment of the WS-Mediator system, which consists of a number of Sub-Mediators which implement an interface that accepts invocations from the client. They monitor Web Services and other Sub-Mediators and generate dependability metadata so that resilience-explicit computing can be performed. The system also applies fault tolerance techniques to deal with faults. The dashed lines represent optional message routes.

3.4 System Architecture

The WS-Mediator system consists of a set of interconnected Sub-Mediators, forming an overlay architecture [64] (see Figure 3-2). Sub-Mediators are globally distributed over the Internet to monitor the dependability of Web Services, and provide accurate dependability metadata, presenting Web Service dependability characteristics from the client's perspective. They are functionally identical; if implementation diversity is intended, however, their implementations can be different. The client invokes a Sub-Mediator as the portal of the WS-Mediator system. Sub-Mediators intercept the interaction between the client and component services, performing resilience-explicit

computing and applying fault tolerance techniques to improve the dependability of service composition. Below we will describe the Sub-Mediator functionalities and its internal structure.

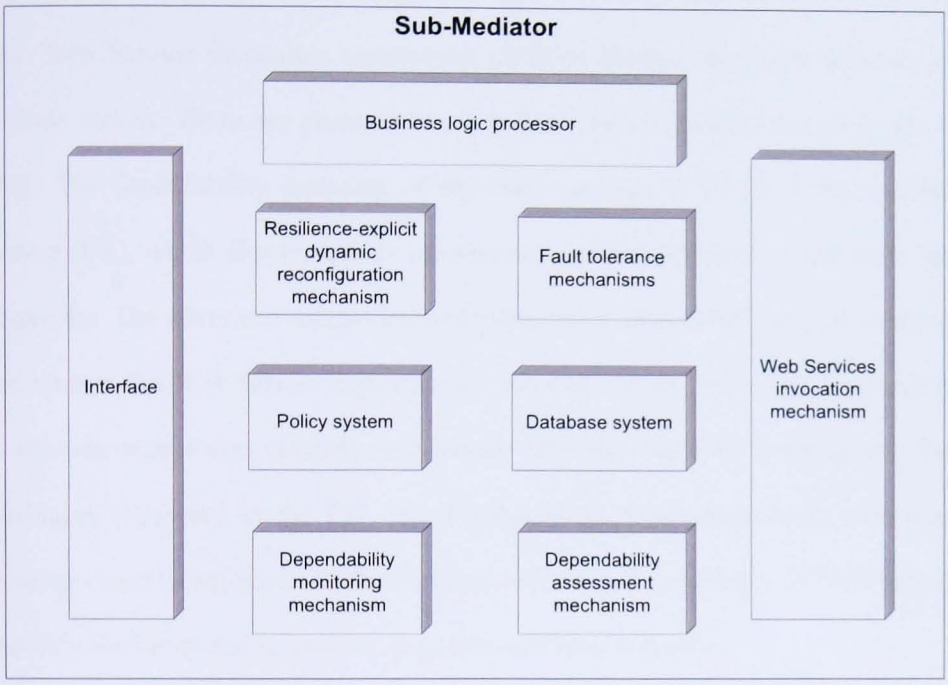

Figure 3-3: The internal structure of the Sub-Mediator

3.4.1 Sub-Mediator Structure

Figure 3-3 illustrates the internal structure of the Sub-Mediator. The Sub-Mediator implements an interface (SMI) to accept the client's invocation. The client's request is parsed and realized by the Business logic processor (BLP), which controls other internal components, performing business logic procedures to fulfil the client's request. The Resilience-explicit dynamic reconfiguration (REDRM) implements a resilience-explicit computing mechanism to dynamically select and integrate the best component services in service composition according to their dependability metadata. Preferences in this selection are constrained by policies defined by the client and

managed by the Policy system (PS) of the Sub-Mediator. The Fault-tolerance mechanisms (FTMs) implements different fault tolerance techniques to deal with different kind of faults. The client can define corresponding policies to select the appropriate fault tolerance mechanisms to improve service composition dependability. The Web Service invocation mechanism (WSIM) invokes the Web Services and collects results. These are processed by the BLP and returned to the client via the SMI. The dependability metadata of the Web Services is stored in the Database system (DS), which also comprises information about Web Services and other Sub-Mediators. The client can submit and edit information about Web Services to the DS and retrieve the Web Service dependability metadata via the WSI. The dependability monitoring mechanism (DMM) successively monitors the Web Services and Sub-Mediators registered in the DS. The Dependability Assessment (DA) mechanism processes monitoring results by the DMM to assess the dependability of Web Services and Sub-Mediators and to generate their dependability metadata.

3.4.2 Sub-Mediator Interface (SMI)

The Sub-Mediator interacts with the client via the SMI, which can be implemented in different forms, such as APIs and Web Services, according to the concrete implementation of the Sub-Mediator. Essentially, the SMI should have the following functionalities:

- Accepting a client's service request for dynamically mediated service composition with candidate Web Services
- Accepting service policies as defined by the client
- Accepting information submission by Web Services
- Accepting a client's request for Web Service dependability metadata

- Returning mediated results to the client
- Returning Web Service dependability metadata to the client for dependability analysis.

The mediating service is the main service provided by the WS-Mediator system. When the client (e.g. an e-Science workflow) requests the WS-Mediator to mediate service composition, it needs to provide one or several candidate Web Services, and an invocation message to be sent to each candidate Web Service. The number of the candidate services depends on the intended fault tolerance mechanisms. The invocation message carries the actual request to each corresponding Web Service. The Sub-Mediator generates a mediated result, based on the results collected from candidate Web Services, according to service policies. The mediated result needs to indicate the source of the initial results, i.e. the candidate Web Services which returned the results that it generated from. In case of no candidate returning a valid result, or other types of failures, the mediated results need to attach an error message indicating the type and details of the error.

The Sub-Mediator allows the client to submit and edit information about Web Services, e.g. the endpoint address, the required message binding methods, etc. via the SMI to help the WS-Mediator system to monitor Web Services. This information is then stored in the DS, and Web Services monitored by the Sub-Mediator. The client can also retrieve Web Service dependability metadata via the SMI for dependability analysis. For example, a Sub-Mediator can request the dependability metadata on particular Web Services to identify the best messaging routes.

3.4.3 Business Logic Processor (BLP)

The BLP controls the business logic process in order to fulfil the client's request. It parses the client's request and service policies, assembles the business process procedures and carries out a set of activities to perform the procedures. Figure 3-4 illustrates the assembly of BLP business procedures and execution activities. The actual process of each procedure node is carried out by the corresponding mechanisms.

3.4.4 Policy System (PS)

The PS manages two types of policies: service and system configuration policies. They define essential and optional configuration parameters to constrain the execution of service procedures as well as internal behaviours.

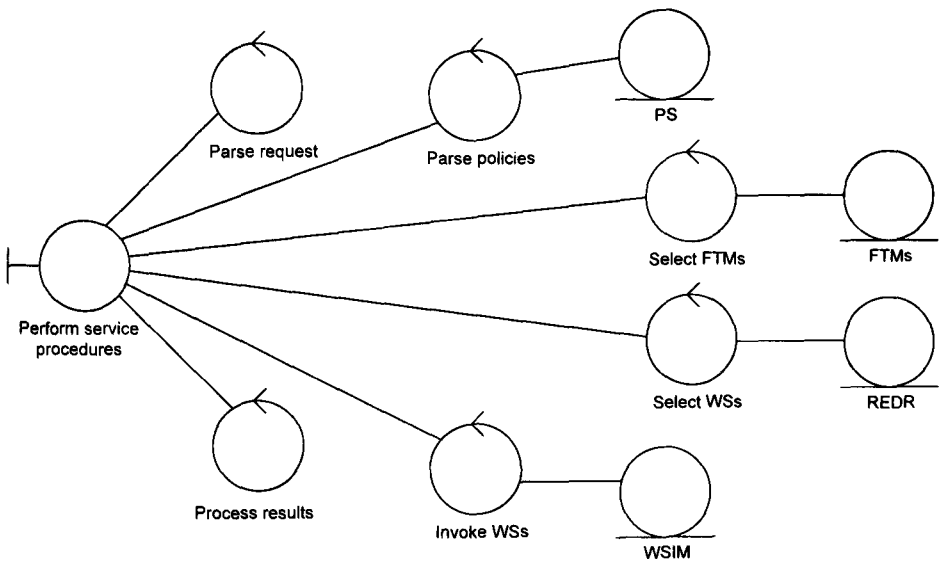

Figure 3-4: Assembly of BLP business procedures and internal activities

Service policies comprise a set of entities allowing the client to define service preference and other processing parameters, such as constraints on the invocation method used for invoking component services, selection of fault tolerance mechanisms, criteria for selecting candidate component services, etc.

System configuration policies contain entities representing system settings. They set parameters to define the corresponding behaviours of the system and its components. For example, they can set the maximum number of synchronous invocations the system allows at a time, the maximum number of entities that the DS can store, etc.

3.4.5 Database System (DS)

The DS comprises two databases: the Web Service database (WSD) and the Sub-Mediator database (SMD). The WSD stores information on the registered Web Services and their dependability metadata, whilst the SMD stores information on the registered Sub-Mediators and their dependability metadata. The information on Web Services needs to be sufficient for the Sub-Mediator to invoke and monitor them, including their endpoint address, operation name and so on. Different operations offered by the same Web Services are regarded as different services. The dependability metadata comprises entities representing the Web Service dependability characteristics, such as their dependability rank, average response time, major types of failures, etc. The structure and content of the SMD is similar to that of the WSD.

3.4.6 Dependability Monitoring Mechanism (DMM)

The DMM monitors the dependability of both Web Services and Sub-Mediators. It retrieves the information on Web Services and Sub-Mediators from the DS to compose test scripts to invoke the services and collect their dependability metrics,

such as the availability measurement (*m*), round-trip response time (*t*), type of failure (*f*), etc. The test scripts run continuously, with the interval defined by the system configuration policies, which also define the dependability metrics, e.g. *m, t, f*, that the test script needs to collect. For instance, when the DMM monitors a Web Service (WS), it invokes it using the test script and waits for a response. If it returns a valid result that does not contain any error message, then its availability measurement (*m*) increases. The round-trip response time of the invocation is recorded for calculating the average response time (*r*) of a WS. If it returns an invalid response, its *m* decreases, and the error message is logged in the database for the type of failures statistic (*f*). If it fails to respond, or an exception arises during the invocation, its *m* also decreases, and the type of the exception is also logged for the statistic *f*.

3.4.7 Dependability Assessment Mechanism (DAM)

The DAM assesses the dependability metrics of services and their dependability characteristics to generate dependability metadata. It can generate and update both permanent dependability metadata *(m, t, f)*, which represent the long-term dependability characteristics of services, and temporary dependability metadata *(m, t, f)* defining their short-term dependability characteristics. The system configuration policies determine the time frame for calculating the short-term dependability metadata *(m, t, f)*. Theoretically, the short-term dependability metadata more accurately represent the dependability of component services during run-time dynamic service composition, whilst the long-term dependability metadata can help to understand the changing behaviour of services.

3.4.8 Resilience-explicit Dynamic Reconfiguration mechanism (REDRM)

The REDRM component dynamically selects and integrates component services according to their dependability metadata *(m, t, f)*. Until now, solutions implementing service diversity have not emphasised strategy of selecting candidate services. The execution order of the alternative services has been decided randomly by the service diversity mechanism, without reasoning. However, as shown in our experiments [37, 38], the dependability characteristics of a Web Service may change from one moment to another. For instance, the availability (*m*) and the round-trip response time (*t*) of the service can vary dramatically, and the service suffers from different type of failures (*f*) at different times. Moreover, the above characteristics can also vary from different clients' viewpoints as well as becoming less predictable because of the variations in the network and other relevant environmental factors. In section 2.5, the use case illustrated in Figure 2-5 demonstrates that inappropriately selecting primary component services when applying service diversity may undermine the efficiency of service composition. Therefore, we introduce resilience-explicit computing for making decisions about selecting component services in dynamic service composition to improve the feasibility and efficiency of the service diversity approach. The Sub-Mediator uses the candidate Web Services provided by the client to implement service diversity. Before carrying out service composition, the REDRM uses the relevant service policies defined by the client to sort the candidate services by their dependability metadata (*m, t, f*) in the DS. The best Web Services are used primarily to perform service integration, whilst the others are used as alternatives. The following shows how to apply resilience-explicit dynamic reconfiguration in service composition:

Service composition: /* collect component services

Aggregation A = $\{s_1, s_2, \ldots s_n\}$

Dependability metadata: /* set the criterion for dynamic selection

Criterion C = *m*: availability /* the criterion set by the selection policy

Sort component services: /* sort services according to metadata

Order O = (A – sorted)

Adaptation: replace (Service S, O) /* switch to new component services

Below is an example which shows how to apply resilience-explicit computing in the design of an application implementing service alternatives:

Set

$\{s_n \mid$ services (n)$\}$: list of candidate component services

criterion = *m* (availability) : parsed from selection policy

threshold t : parsed from selection policy

Retrieve

$\{a_n \mid$ availability (n)$\} = m_n$: metadata (*m*) of s_n

Filter

$\{c_n \mid$ candidates (n)$\} = s_n$ where a_n is equal to or greater than t

Sort c_n : sort according to a_n

Composition

Try

service S = c_1

response r = invoke (c_1);

if (r is valid) then **Finish**

else replace S with next c_n

Try … /* try alternatives

Finish

return r /* return response to the upper level class

The benefits of this approach are clear. Integrating explicitly selected component services can maximize the dependability and performance of service composition as the less dependable component services are avoided to prevent them from undermining the dependability of the entire application.

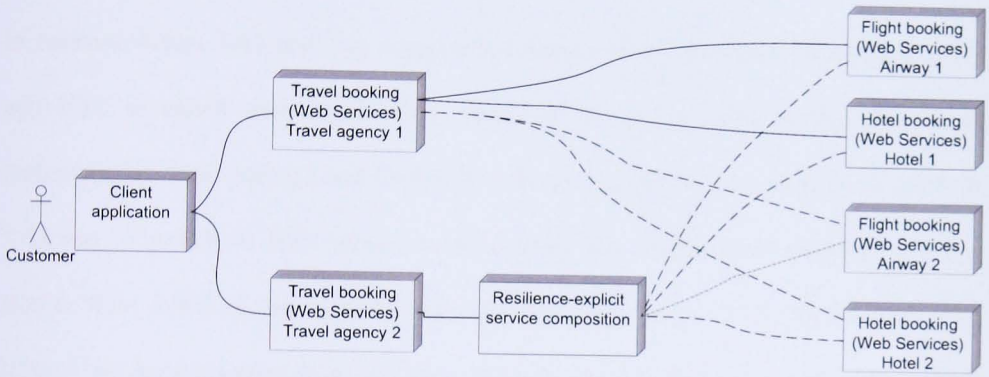

Figure 3-5: The resilience-explicit service composition in travel booking use case. The solid lines represent fixed message routes, and the dashed lines redundant/alternative message routes.

Here we use the *travel booking* use case to demonstrate the feasibility of resilience-explicit computing in service composition. The *travel booking* illustrated in Figure 3-5 extends the one illustrated in Figure 2-5, where both travel agencies (TA), TA1 and TA2 use the same Airways (AW), AW1 and AW2, and hotels (HT), HT1 and HT2, as external component services. Normally, TA1 uses AW1 and HT1 as primary component services for travel booking, with AW2 and HT2 used as alternatives if AW1 or HT1 fails. TA2 implements resilience-explicit service composition in its travel booking business procedure. AW1, AW2, HT1 and HT2 are equally used as redundant component services. When TA2 receives a *quotation request* from the

client, the resilience-explicit computing mechanism checks the dependability metadata (*m, r*) of AW1, AW2, HT1 and HT2, and selects the most dependable ones to perform service composition. Let us assume that the HT1 is an undependable Web Service, whilst HT2 is very dependable, and that TA2 uses HT2 primarily to check the hotel. At the same time, the performance of AW2 is better than of AW1, and TA2 uses AW2 to check the flight. In this scenario, TA2 achieves the best dependability and shortest response time for the client.

In contrast, when TA1 receives a *quotation request* from the client, it invokes AW1 and HT1 to check their availability. However, as we already know, HT1 is an undependable Web Service and therefore fails to respond to TA1 enquiry. Therefore, TA1 has to switch to HT2 to check the availability. Meanwhile, although AW1 is slower than AW2, it successfully delivers the response to TA1. Eventually, TA1 returns the *booking quotation*; however, it loses the competition against TA2, which delivers faster response because of the superior service implementation. Below we demonstrate how to apply resilience-explicit computing in designing TA2:

Services

{hotel | HT1, HT2 }

{airway | AW1, AW2 }

Metadata

{*m* (%) | HT1: 60%, HT2: 90%, AW1: 90%, AW2: 90% }

{*r* (ms) | HT1: 500ms, HT2: 400ms, AW1: 800ms, AW2: 600ms }

Selection policy

{primary_criterion : *m* (availability) | no threshold;

Second_criterion: *r* (response time) | no threshold }

Sort

{hotel | HT2, HT1} /* $m_{HT2} > m_{HT1}$

{airway | AW2, AW1} /* $m_{AW2} = m_{AW1}$ but $r_{AW2} < r_{AW1}$

Composition

Try check hotel

hotel h = HT2

response rh = invoke (h)

if (r is valid) then Finish hotel booking

h = HT1

Try ...

Finish hotel booking

Try check flight

airway a= AW2

response ra = invoke (h)

if (r is valid) then Finish airway booking

a = AW1

Try ...

Finish check flight

Finalize

quotation = rh + ra + service charge

return quotation

There are also other benefits gained through resilience computing. For example, the REDRM can appropriately set relevant parameters when integrating component services according to the information in the dependability metadata. The information may contain average or maximum response time of the component service, and the

REDRM can set the invocation time-out parameter according to the response times to improve the performance of service composition.

3.4.9 Fault-tolerance mechanisms (FTMs)

The Sub-Mediator implements fault tolerance techniques to tolerate temporary and permanent *service* and *network failures*. They are implemented as different fault tolerance execution modes aggregated in the FTMs. There are currently three types of fault tolerance execution modes included.

A. Service Alternative Execution Mode

The Service Alternative execution mode implements the Recovery block fault tolerance technique [52] to apply the service diversity strategy [20]. When the client selects the Service Alternative execution mode and provides a number of Web Services as candidates, the REDRM mechanism will first check the dependability metadata of the candidate Web Services, removing the Web Services that do not meet the acceptance thresholds from the candidate list. Then the REDRM sorts the Web Services according to prior criteria defined in the *service policies* comprised in the PS. The Web Service with the best dependability metadata will be selected as the primary one and the others used as alternatives. If the primary Web Service fails, the next best alternative Web Services will be invoked. Eventually, when a valid result is received from a Web Service, the execution will be terminated. The result will then be delivered to the BLP, which uses it to generate the mediated result to be sent to the client as the response to the service request. Figure 3-6 illustrates the use case of the Service Alternative execution mode.

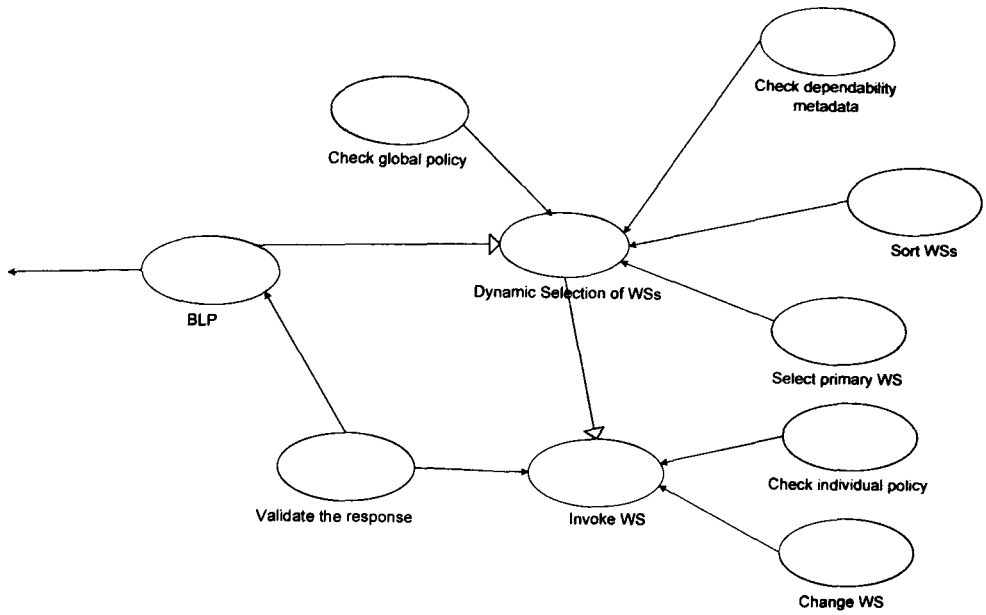

Figure 3-6: The use case of the Service Alternative execution mode

B. N-version Programming Execution Mode

The N-version Programming execution mode implements the N-version Programming technique [70]. The N-version Programming mode invokes a number of Web Services simultaneously, and the results received from Web Services will be processed according to the corresponding *service policies*. Note that the technique used in Web Services is sometimes different from the classical N-version programming technique applied in conventional software/system development, where the multiple versions are mostly developed from the same requirements and specifications, and their processing results can be voted for result validation. With Web Services, similar Services can be used for implementing service diversity; they are, however, very likely to be irrelative to each other, not meeting the same implementation specifications. Thus, the results can only be voted after transforming and matching processes, which mechanisms are not intended in the WS-Mediator system. Using the result voting mechanism in this

execution mode is subject to applicability. Figure 3-7 illustrates the use case of the N-version programming execution mode.

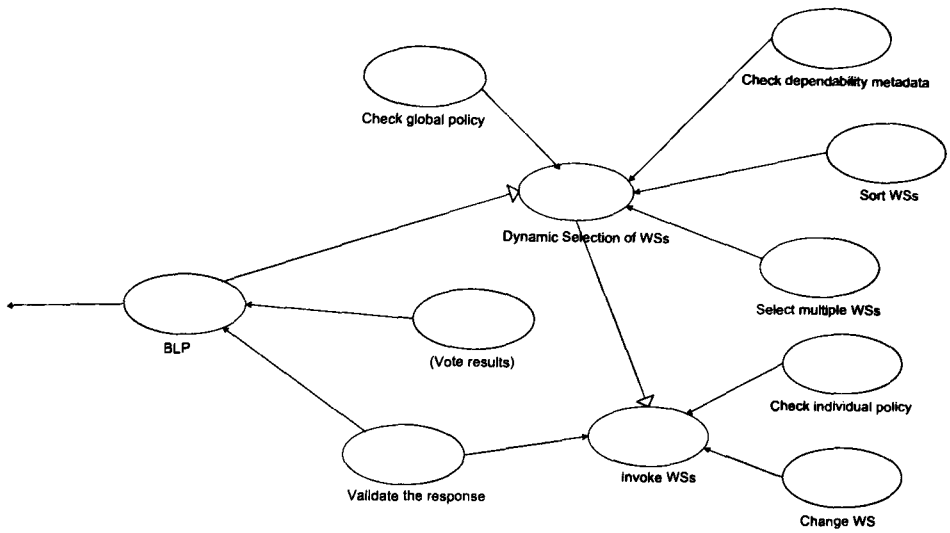

Figure 3-7: The use case of the N-version Programming execution mode

C. Message Routing Execution Mode

The Message Routing execution mode implements a unique fault tolerance mechanism which extends the conventional Message Routing diversity strategy to achieve explicit selection of message routing. When this execution mode is selected, the Sub-Mediator checks the dependability metadata of each candidate Web Service from the Sub-Mediators registered in its Sub-Mediator registry. If the dependability metadata of a Web Service in the participating Sub-Mediators meet the parameters defined in the *service policies*, the Sub-Mediator can be selected as a message routing intermediary. Once the required number of intermediaries is satisfied, the local Sub-Mediator passes the invocation details of the Web Service to the intermediary Sub-Mediators. The intermediary Sub-Mediators then invokes the Web Service from their

locations. The results will be retuned to the local Sub-Mediator. If more than one message route is selected, the results will be processed according to the *service policies*. Figure 3-8 illustrates the use case of the Message Routing execution mode.

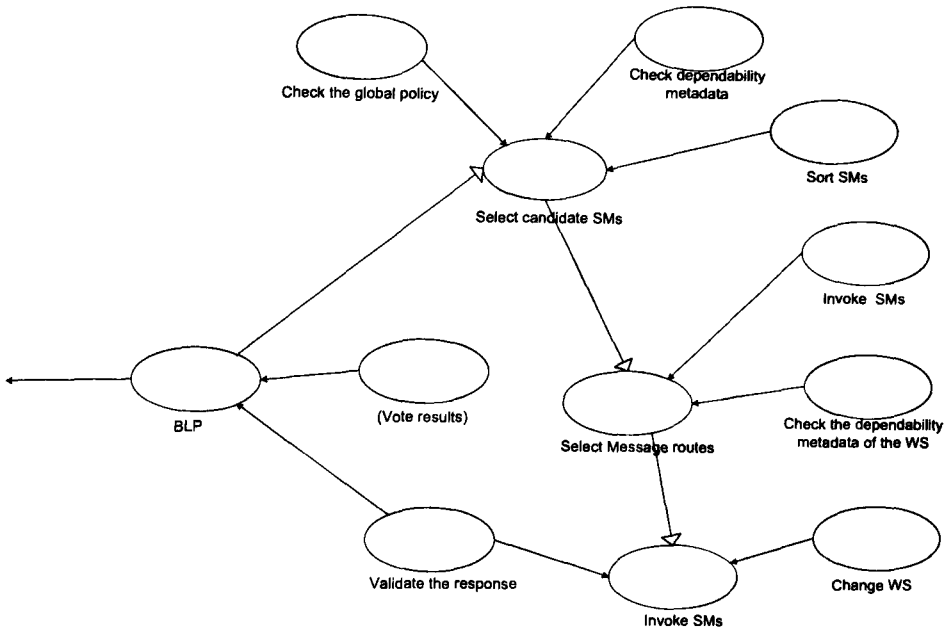

Figure 3-8: The use case of the Message Routing execution mode.

D. Dynamic Reconfiguration of Fault-tolerance Mechanisms

The fault tolerance mechanisms are designed to deal with various types of failures as well as different types of application scenarios. The efficiency of the WS-Mediator system greatly relies on the selection of fault tolerance mechanisms during service composition. Resilience-explicit computing can also be applied in making decisions about the selection of fault tolerance mechanisms. The novelty of our approach is that the resilience-explicit dynamic reconfiguration mechanism consults the statistic of type of failures (f) of Web Services to select the most appropriate fault tolerance mechanism for dealing with typical failures of Web Services. For instance, if a Web

Service often fails because of network-related failures, then it may be advisable to apply the *message routing execution mode* integrated with the service; if a Web Service only rarely fails due to temporary faults, such as an occasional time-out, system maintenance, and so on, it can be a good choice to make it the primary service and apply the *service alternative execution mode*, whilst using other, less dependable ones, as alternatives. Furthermore, it is also feasible to automatically select the *N-version programming execution mode* when the availability measurement (m) of all candidate Web Services is much lower than certain standards.

3.4.10 Web Service Invocation Mechanism (WSIM)

The development of Web Services relies on Web Service middleware provided by a variety of organizations and companies [31-33], which implements mechanisms defined in the Web Service specifications. As this middleware commonly supports different message binding methods, invocation methods, etc., the WSIM needs to aggregates different message binding and invocation methods to suit different Web Services. The message binding method and invocation type can be defined in the *service policies*.

3.5 Application of the WS-Mediator

Applying the WS-Mediator is easy. It can be seamlessly integrated in Web Service composition applications. It does not require component services to be modified, because of its compliance with the interoperability standards. The WS-Mediator simplifies the development of the client application by enhancing service composition procedures and fault tolerance mechanisms with the off-the-shelf functionalities implemented in the WS-Mediator. Therefore, the client application only needs to

provide candidate component services and define service policies for the WS-Mediator, avoiding the complexity of service composition. Moreover, the WS-Mediator can dramatically improve the dependability and performance of service composition without increasing the complexity and cost of application development, and these benefits become more prominent when the scale of service composition increases, involving more component services.

Moreover the WS-Mediator approach improves the applicability and efficacy of the service diversity strategy based on the functionally-similar autonomous services without undermining the compatibilities with Web Services security mechanisms and stateful Web Services. The approach allows the client to set specific requests (including encrypted messages) and service policies for each candidate services so that the system explicitly selects the best component services during dynamic composition. In the case of stateful Web Service composition, the system allows the client to decide how to continue the execution of a workflow when a failure occurs in the middle of the interactions with a stateful component service. For example, the client can provide replica services as alternatives so that these replica services can retrieve the processing state and continue the business logic process; or the client can decide to abandon the interrupted business logic process and use other similar services to process the business logic from the top.

While providing flexible transaction-oriented fault tolerance to improve the dependability of service composition, the WS-Mediator system does not interfere with the execution of the client application. We believe that the client will typically be in a better position to choose how to compose the business logic and decide how to control the workflow, while the WS-Mediator system can help the client application to use

the best services and improve the dependability of the transactions between the client and the services.

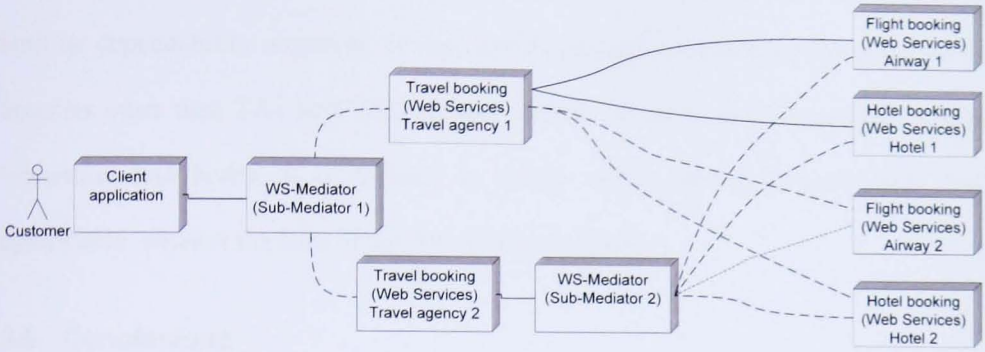

Figure 3-9: Travel booking use case with the WS-Mediator system. The solid lines represent fixed primary and the dashed lines redundant/alternative message routes.

Here we use the *travel booking* use case again to demonstrate the advantages of the WS-Mediator system. Figure 3-9 illustrates the *travel booking* use case that integrates the WS-Mediator system into service composition. The client application and TA2 both develop their business logic relying on the WS-Mediator system, whilst TA1 retains the conventional implementation. TA2 relies on Sub-Mediator2 to implement dynamic integration with AW1, AW2, HT1 and HT2, applying fault tolerance mechanisms in the interaction between TA2 and the external component services. Obviously, TA2 provides higher dependability and better performance than TA1 does. Sub-Mediator1 monitors the dependability of TA1 and TA2. When the client requests the WS-Mediator to perform service composition for *travel booking*, TA2 will be selected by Sub-Mediator2 to fulfil the booking request. While in reality TA2 may fail to deliver the service to the client during the process of the booking process because of failures of component services or the network beyond what the fault tolerance mechanisms can deal with, the dependability metadata provide quantitative evidence

suggesting TA2 is less likely to fail than TA1. Thus, the performance of the *travel booking* procedure is optimized because all participating component services are explicitly selected. Consequently, TA1 will lose business when competing with TA2, until its dependability improves. In real-world applications, there are far more travel agencies other than TA1 and TA2 offering similar services, as well as more airway companies and hotels. It is difficult to decide which service is trustworthy and dependable, without the help of the WS-Mediator system.

3.6 Conclusions

In section 3.2, we have outlined the objectives we set for our research. We believe these have been successfully achieved in the WS-Mediator approach:

A. The WS-Mediator is a generic solution reinforcing and extending the existing work on improving the dependability of Web Services via its overlay architecture to ensure the continuity of services.

B. The innovation of the WS-Mediator lies in its off-the-shelf mediating architecture and resilience-explicit computing, which allow dynamic integration of Web Services according to their dependability behaviour.

C. The WS-Mediator supports genuine on-the-fly integration with Web Services via its interoperable Web Service interface and invocation mechanism.

D. The Policy-driven dynamic reconfiguration of the fault tolerance mechanisms makes the WS-Mediator applicable to dealing with various types of faults and the changing behaviour of Web Services and the network.

E. The WS-Mediator is compliant with the Web Service interoperability standards.

F. The flexible and scalable design of the approach allows it to be extended or tailored to suit specific applications.

In this chapter, we have described the architecture of the WS-Mediator system and explained the functionalities of the system components. We have specifically focused on how to generate dependability metadata according to monitoring results, and how to utilize these metadata in resilience-explicit computing to achieve dynamic service composition with the most dependable Web Services. Moreover, the WS-Mediator improves the dependability of service composition by employing a variety of fault tolerance techniques.

4. Java WS-Mediator

4.1 Introduction

In this chapter we present the Java WS-Mediator, which is a prototype of the WS-Mediator system implemented using the Java Web Service technology [71]. The Java WS-Mediator has been developed with the aim of evaluating the WS-Mediator approach and demonstrating the applicability of the approach in a number of realistic Web Service applications. We chose Sun Microsystems Glassfish [33] as the Java Web Service platform for the development of the prototype. Our implementation supports two types of Sub-Mediator. The Sub-Mediator Elite is implemented as an additional layer on top of the Glassfish Java Web Service Middleware. It can be easily deployed on a personal computer to enable WS-Mediator Java APIs to be invoked by the client application. The Web Service intermediary Sub-Mediator implements Web Service interface and is developed to be deployed on the Glassfish application server. It uses the Sub-Mediator Elite as the underlying middleware to achieve the designed functionalities.

The chapter is organised as follows: section 4.2 briefly introduces the Java Web Service technology, section 4.3 presents the design of the Java WS-Mediator, and section 4.4 concludes this chapter.

4.2 Java Web Service middleware

Web Services is a paradigm of distributed systems that extends the conventional peer-to-peer middleware protocols to override some shortcomings of the conventional distributed systems. The implementation of Web Services relies on middleware infrastructure known as *Web Service middleware*. This middleware shares the

underlying infrastructure with the conventional middleware to provide fundamental underlying services such as transaction support, etc. See a representation of Web Service architecture in Figure 4-1.

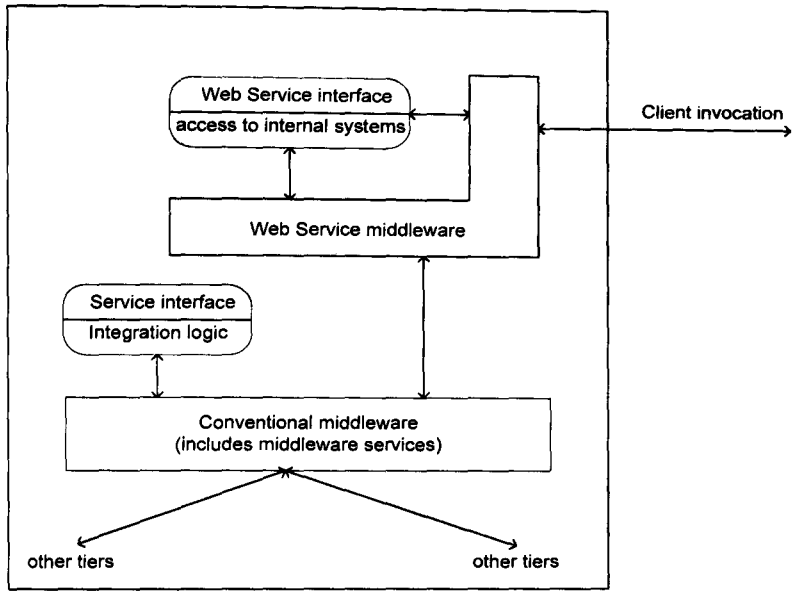

Figure 4-1: Basic architecture of Web Services. [1]

The client application also relies on *Web Service middleware* which implements underlying protocols atop conventional middleware. The architecture of the client application is illustrated in Figure 4-2.

Web Service middleware can be developed based upon different technologies. Today's middleware typically relies on the .NET [72] or J2EE [73]. While comparing these is beyond the scope of this dissertation, our choice of the Java Web Services based on the J2EE technology to develop the WS-Mediator was prompted by the platform-independent nature of the J2EE technology. Besides there are sufficient recourses and supports available for Java Web Services free of charge, which makes them a cost-efficient platform to conduct academic research and experiments.

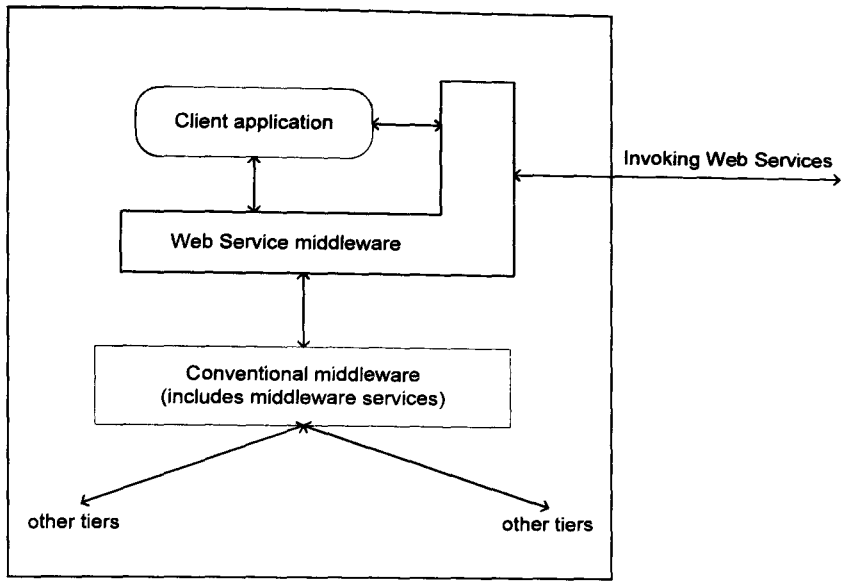

Figure 4-2: The architecture of Web Service client

There are several implementations of the *Java Web Service middleware* developed by different providers, such as Aparche Axis [32], JBoss [31], and Glassfish [33]. All of them are sufficient for developing complex Web Service applications. While each has its unique features and advantages over the others, we chose Glassfish for the following reasons:

- Its comprehensive development environment and tools integrated in the NetBeans IDE for developing Web Service applications [74].
- Sufficient support of dynamic Web Service invocation provided by the powerful *Dispatch<T>* interface.
- Compliancy with the current Web Service specifications and Web Service Interoperability standards.
- Open-source project with strong industrial support by both Sun Microsystems and Microsoft.

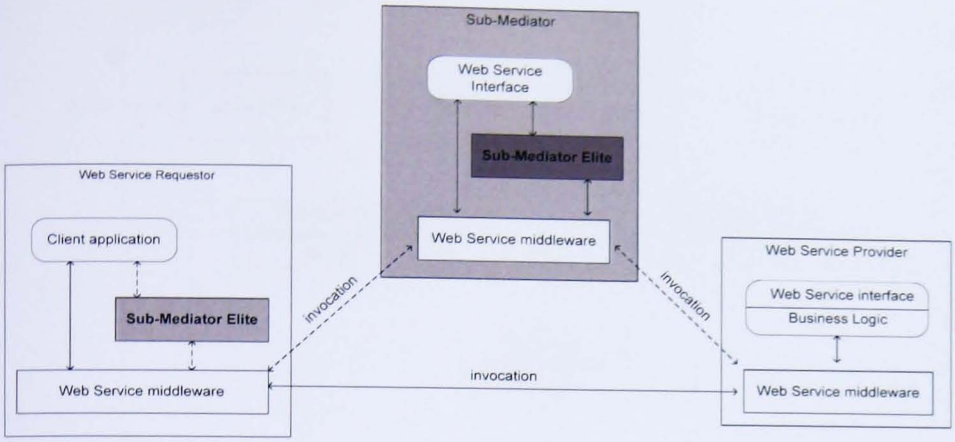

Figure 4-3: Web Service application with the Java WS-Mediator

Below is the development environment and packages for implementing the Java WS-Mediator:

- Development IDE: NetBeans v5.5.1 [74]
- Java SDK: J2EE v1.5
- Web Service platform: Glassfish V2
- Java Web Service API: JAX-WS 2.1 [75] and JAX-RPC 1.6

4.3 Structure of the Java WS-Mediator

The WS-Mediator system is structured of functionally identical Sub-Mediators. These can be implemented in different forms, as long as they agree with the fundamental principles and designed functionalities presented in chapter 3. We have developed a special Java WS-Mediator middleware called Mediator-Elite to accomplish the designated structure and functionalities of the Sub-Mediator.

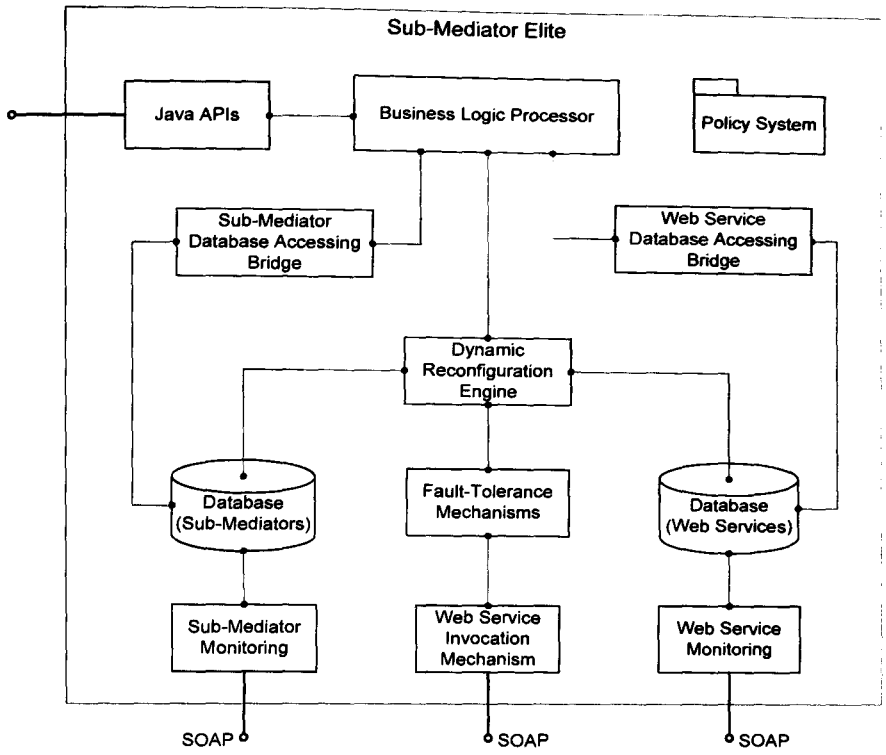

Figure 4-4: Internal structure of the Sub-Mediator Elite, which implements Java APIs as interface to accept invocations from the client application. It monitors Web Services and other Sub-Mediators registered in its database, and generates their resilience metadata to perform resilience-explicit dynamic reconfiguration.

4.3.1 Structure of the Sub-Mediator Elite

The Sub-Mediator Elite is implemented as an additional layer atop the *Glassfish Web Service middleware*. It can be deployed on personal computers. The Java client application can invoke the Java APIs of the Sub-Mediator Elite to use it as a locally deployed Sub-Mediator. The Sub-Mediator Elite can also be used for implementing the Web Service intermediary type Sub-Mediator by deploying it on the Glassfish Application Server, as well as realizing a Web Service interface corresponding to the Java APIs of the Sub-Mediator Elite (see Web Service architecture with the Java WS-Mediator shown in Figure 4-3).

Figure 4-4 illustrates the internal structure and components of the Sub-Mediator Elite. It implements *Java APIs (JAPIs)* to accept the invocation from the client application. The *BLP* parses the client's requests and service policies, and assigns tasks to the corresponding components to implement the business logic process procedures. The *Web Service Database (WSD)* stores the information about Web Services and keeps their dependability metadata. The *Web Service Database Accessing Bridge (WSDAB)* allows editing the information about Web Services and retrieving their dependability metadata. The *Sub-Mediator Database (SMD)* stores the information about other Sub-Mediators and keeps their dependability metadata. The *Sub-Mediator Database Accessing Bridge (SMDAB)* edits the information about the Sub-Mediators and retrieves their dependability metadata. The *Dynamic Reconfiguration Engine (DRE)* implements a resilience-explicit mechanism to integrate Web Services and apply fault-tolerance techniques. It selects the most desirable, according to the service policies, Web Services and then chooses fault tolerance execution modes to perform service composition. The *Fault-tolerance Mechanisms (FTMs)* implement different fault tolerance execution modes to deal with different fault assumptions. The *Web Service Monitoring (WSM)* and *Sub-Mediator Monitoring (SMM)* monitor Web Services and Sub-Mediators respectively and generate their dependability metadata. The *Web Service Invocation Mechanism (WSIM)* implements various message binding and invocation methods to improve the interoperability with real-world Web Services. In the following sections we will describe the functionalities of each component in detail.

4.3.2 Java APIs of the Sub-Mediator Elite

The Sub-Mediator accepts service requests via its JAPIs interface. There are three basic types of service requests classified by their purpose:

- Accessing the Web Service database
- Accessing the Sub-Mediator database
- Requesting mediating services

The above requests are dealt with by corresponding service components. Below is an explanation of each type of service requests.

A. Accessing Web Service Database

The Sub-Mediator Elite allows adding, editing, and removing the information about Web Services via the WSDAB. After the client adds a Web Service to the WSD, it is periodically monitored by the Sub-Mediator Elite for later use. The client needs to provide the following information associated with it:

- *Endpoint address of the Web Service*
- *Operation name*
- *Description of the Web Service*
- *Test SOAP message*
- *Test policy*

The *endpoint address* and *operation name* are used for identifying the Web Service and the client-intended service function provided by the Web Service. Different operations provided by the same Web Service are regarded as different entities. The

description gives a briefly memo about the Web Service. The WSM mechanism uses the *test SOAP message* to invoke the Web Service and the corresponding service operation. Figure 4-5 shows a simple example of the *test SOAP message*:

```
<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\">
        <soapenv:Body>
            <addNumbers xmlns=\"http://mediator.wsmediator.org\">
                <arg0>10</arg0>
                <arg1>20</arg1>
            </addNumbers>
        </soapenv:Body>
</soapenv:Envelope>
```

Figure 4-5: An example of the *test SOAP message*

The *test policy* is used for defining relevant parameters, such as the invocation method and expected timeout. Figure 4-6 illustrates an abstract model of the *test policy*:

```
<wsp:Policy  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsmip="http://schemas.wsmediator.org/testpolicy/policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <parameter1>{value}</parameter1>
                ...
            <parameterN>{value}</parameterN>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 4-6: An example of the *test policy*

The client can also edit and remove the existing Web Services from the WSD, as well as retrieve the information about Web Services by providing their *endpoint address* and *operation name*. The client can request the dependability metadata of a Web Service via the corresponding JAPIs. The dependability metadata will be capsulated in a SOAP message returned to the client.

B. Accessing the Sub-Mediator Database

The client can add and edit information about other Sub-Mediators in the SMD. In order to add a Sub-Mediator, the client needs to submit the following items:

- *Endpoint address of the Sub-Mediator*
- *Its Location and ISP*
- *Brief memo*

The *endpoint address* is used for identifying the Sub-Mediator. The test script for monitoring a Sub-Mediator is automatically generated by the SMM mechanism. The client may request the dependability metadata of Sub-Mediators by providing the *endpoint address*. The dependability metadata of a Sub-Mediator will be attached into the SOAP message sent to the client.

C. Requesting Mediating Services

The most important type of requests is for mediating services. It is the core service offered by the WS-Mediator system. The client invokes the corresponding API to submit a mediating service request. The following information needs to be attached to a service request message:

- *One or more candidate Web Services*

- *Endpoint addresses of the Web Services*
- *Operation names of the services being invoked*
- *SOAP messages to each candidate Web Service*
- *An individual execution policy associated with each Web Service*
- *A global execution policy*

The candidate Web Services are not limited to those existing in the WSD. However, only the Web Services that have already been monitored by the Sub-Mediator can be used explicitly since only their dependability metadata are available. The SOAP message associated with each candidate Web Service is identical to that used for invoking the Web Service directly from the client without using the Sub-Mediator. The *individual execution policy* constrains the instruction indicating how to process a candidate Web Service. The *global execution policy* indicates how to process the client's request. An abstract example of the service request SOAP message is illustrated in Figure 4-7.

```
<SOAP abstract>
    <ws>
        <endpointAddress>{EndpointAddress_ws1}</endpointAddress>
        <functionName>{FunctionName_ws1}</functionName>
        <SOAPMessage>{SOAP_to_ws1}</SOAPMessage>
        <individualPolicy>{InExPolicy_XML_ws1}</ individualPolicy>
    </ws>
    <ws>
        <endpointAddress>{EndpointAddress_ws2}</endpointAddress>
        <functionName>{FunctionName_ws2}</functionName>
        <SOAPMessage>{SOAP_to_ws2}</SOAPMessage>
```

```
            <individualPolicy>{InExPolicy_XML_ws2}</ individualPolicy>
        </ws>
        <ws>
            <endpointAddress>{EndpointAddress_ws3}</endpointAddress>
            <functionName>{FunctionName_ws3}</functionName>
            <SOAPMessage>{SOAP_to_ws3}</SOAPMessage>
            <individualPolicy>{InExPolicy_XML_ws3}</ individualPolicy>
        </ws>
        <globalExecutionPolicy>
            {GlobalExecutionPolicy_XML}
        </globalExecutionPolicy>
</SOAP abstract>
```

Figure 4-7: An abstract of the service request SOAP message

4.3.3 Business Logic Processor (BLP)

The BLP implements service operations corresponding to the Web Service Interface, diverting service requests to the corresponding service processing components. A service request for accessing the WSD will be diverted to the WSDAB, one for accessing the SMD to the SMDAB, and one for mediating services to the DRE.

When service components complete the execution of service requests, they pass the results back to the BLP, which assembles the processing result into a SOAP message and returns it to the client.

4.3.4 Database System

There are two databases comprised in the DS of the Sub-Mediator Elite. The WSD consists of the *Web Service Registry* and the *Web Service Dependability Metadata*

Database. The SMD consists of the *Sub-Mediator Registry* and the *Sub-Mediator Dependability Metadata Database.*

A. *Web Service Database (WSD)*

The *Web Service Registry* maintains the information about a number of Web Services added by the clients and the system administrators. It contains the information associated with each Web Services:

- *Endpoint address of the Web Service*
- *Operation name*
- *Description of the Web Service*
- *Test SOAP message*
- *Test policy*

The above information is used for monitoring Web Services. Figure 4-8 illustrates an abstract model of the *Web Service Registry* in the XML format.

```
<?xml version="1.0" encoding="UTF-8"?>
<webServicesRegistry>
        <ws>
            <endpointAddress>{Endpoint_ws1}</endpointAddress>
            <operationName>{Operation_ws1}</operationName>
            <description>{Memo_Text_ws1}</description>
            <testSOAPMessage>{TestSOAPMessage_ws1}</testSOAPMessage>
            <testPolicy>{TestPolicy_ws1}</testPolicy>
        </ws>
        <ws>
```

```
        <endpointAddress>{Endpoint_ws2}</endpointAddress>
        <operationName>{Operation_ws2}</operationName>
        <description>{Memo_Text_ws2}</description>
        <testSOAPMessage>{TestSOAPMessage_ws2}</testSOAPMessage>
        <testPolicy>{TestPolicy_ws2}</testPolicy>
    </ws>
    ...
</webServicesRegistry>
```

Figure 4-8: An example of the Web Service Registry

The *Web Service Dependability Metadata Database* stores the dependability metadata of the corresponding Web Services, i.e. attributes which represent their dependability characteristics. Figure 4-9 illustrates an abstract model of the dependability metadata of a Web Service in the XML format.

```
<?xml version="1.0" encoding="UTF-8"?>
<ws service={Name_of_ws1}>
        <endpointAddress>{Endpoint_ws1}</endpointAddress>
        <operationName>{Operation_ws1}</operationName>
        <dependabilityAttribute1>{value}</dependabilityAttribute1>
        <dependabilityAttribute2>{value}</dependabilityAttribute2>
                        ...
        <dependabilityAttributeN>{value}</dependabilityAttributeN>
</ws>
```

Figure 4-9: An abstract model of the dependability metadata of a Web Service

If a Web Service registered in the *Web Service Registry* is not used for a certain period of time, it will be removed from the database, along with its metadata.

B. Sub-Mediator Database (SMD)

The *Sub-Mediator Registry* contains the following information about a number of Sub-Mediators:

- *Endpoint address of the Sub-Mediator*
- *The Location and ISP of the Sub-Mediator*
- *Memo*

Sub-Mediators implement a universal test service for monitoring. The Sub-Mediator Monitoring Mechanism uses the *endpoint address* of the Sub-Mediator to automatically generate the test script. The *endpoint address* can be used to identify the Sub-Mediator in the *Sub-Mediators Registry*. The *location* and *ISP* of the Sub-Mediator help the client to locate it and can also be used for implementing message routing strategies. The *memo* briefly describes the Sub-Mediator. Figure 4-10 gives an abstract model of the *Sub-Mediator Registry*.

```
<?xml version="1.0" encoding="UTF-8"?>
<subMediatorRegistry>
        <ws>
            <endpointAddress>{Endpoint_sm1} </endpointAddress>
            <location>{city, country}</location>
            <isp>{NameofISP}</isp>
            <memo>{MemoText_sm1}</memo>
        </ws>
        <ws>
            <endpointAddress>{Endpoint_sm2} </endpointAddress>
            <location>{city, country}</location>
```

```
            <isp>{NameofISP}</isp>
            <memo>{MemoText_sm2}</memo>
        </ws>
        ......
    </ subMediatorRegistry >
```

Figure 4-10: An example of the Sub-Mediator Registry

The *Sub-Mediator Dependability Metadata Database* stores the dependability metadata of Sub-Mediators in the registry. Figure 4-11 shows an abstract model of the dependability metadata of a Sub-Mediator in the XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<sm service={Name_of_sm1}>
    <endpointAddress>{Endpoint_sm1}</endpointAddress>
    <operationName>{Operation_sm1}</operationName>
    <dependabilityAttribute1>{value}</dependabilityAttribute1>
    <dependabilityAttribute2>{value}</dependabilityAttribute2>
                                    ...
    <dependabilityAttributeN>{value}</dependabilityAttributeN>
</sm>
```

Figure 4-11: An example of the dependability metadata of a Sub-Mediator

4.3.5 Policy System

There are three types of policies implemented in the Sub-Mediator Elite, listed below:

- *Test Policy*
- *Individual execution policy*
- *Global execution policy*

As the *test policy* was introduced above, we will now focus on the *individual execution policy* and *global execution policy.*

A. Individual Execution Policy

As mentioned already, when the client invokes a Sub-Mediator requesting mediator services, it needs to define an *individual execution policy* for each candidate Web Service. The *individual execution policy* is an instruction for processing invocation for every Web Service, which may set, for example, the invocation method, the timeout parameter, etc. However, it can be omitted from the service request, with the Sub-Mediator using the system default settings to set parameters for invoking the Web Service. Figure 4-12 shows an abstract model of the *individual execution policy*:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp:Policy xmlns:wsp = http://schemas.xmlsoap.org/ws/2004/09/policy
    xmlns:wsmip = "http://schemas.wsmediator.org/individualPolicy/policy">
     <wsp:ExactlyOne>
         <wsp:All>
             <parameter1>{value}</parameter1>
             <parameter2>{value}</parameter2>
                    ...
             <parameterN>{value}</parameterN>
         </wsp:All>
     </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 4-12: An abstract model of the *individual execution policy*

To implement the *individual execution policy* described above, we have developed a *WS-Mediator Policy framework,* extending the WS-Policy framework in [76]. Below we show the *individual execution policy* specially developed in one of our experiments, followed by a brief explanation of each policy entity:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp:Policy xmlns:wsp = http://schemas.xmlsoap.org/ws/2004/09/policy
    xmlns:wsmip = "http://schemas.wsmediator.org/indevidualPolicy/policy">
     <wsp:ExactlyOne>
         <wsp:All>
                 <bindingMethod>SOAP11HTTP</bindingMethod>
                 <invocationMode>Sync</invocationMode>
                 <timeout>20000ms</timeout>
                 <autotimeout>maximum</autotimeout>
                 <retryAfterFailure>3</retryAfterFailure>
                 <retryInterval>3000ms</retryInterval>
         </wsp:All>
     </wsp:ExactlyOne>
</wsp:Policy>
```

- *<bindingMethod>*: this indicates the binding method of the SOAP message. Web Service invocation APIs should follow the binding method while invoking the Web Service. Default value: SOAP11HTTP
- *<invoactionMode>*: this entity indicates the invocation method of the Web Service. There are three types of invocation methods: synchronous, asynchronous invocation and the conventional RPC (Remote Procedure Call) invocation. Default value: Sync (Synchronous invocation)

- *<timeout>*: this sets the timeout parameter for an invocation. If the invocation does not complete in the timeout period, it will be terminated and a timeout exception will be raised. The value of the timeout parameter can be automatically set by the Sub-Mediator when the value is set as 0ms.
- *<autotimeout>*: the Sub-Mediator can automatically set the timeout parameter for invoking a particular Web Service according to dependability metadata. There are three options: average, minimum and maximum, representing *average, minimum* and *maximum response time.*
- *<retryAfterFailure>*: the Sub-Mediator implements the retry strategy to tolerate temporary *service* and *network failures*. This entity sets the number of retry invocations of a particular Web Service before giving up.
- *<retryInterval>*: this entity sets the interval between retries.

B. Global Execution Policy

When the client requests a mediating service from a Sub-Mediator, it needs to attach a *global execution policy* to the service request message. The *global execution policy* is an instruction which indicates how to process the entire service request. It sets important parameters for performing service procedures according to the service request. Figure 4-13 shows an abstract model of the *global execution policy*:

```
<?xml version="1.0" encoding="UTF-8"?>

<wsp:Policy xmlns:wsp=http://schemas.xmlsoap.org/ws/2004/09/policy
            xmlns:wsmgp="http://schemas.wsmediator.org/globalPolicy/policy">

    <wsp:ExactlyOne>

        <wsp:All>

            <wsmExecutionMode:executionMode1 execution="true">
```

```
                <exeMode1_parameter1>{value}</ exeMode1_parameter1>
                <exeMode1_parameter2>{value}</ exeMode1_parameter2>
                                ...
                <exeMode1_parameterN>{value}</ exeMode1_parameterN>
            </ wsmExecutionMode: executionMode1>
            <wsmExecutionMode: executionMode2 execution="false">
                <exeMode2_parameter1>{value}</ exeMode2_parameter1>
                <exeMode2_parameter2>{value}</ exeMode2_parameter2>
                                ...
                <exeMode2_parameterN>{value}</ exeMode2_parameterN>
            </ wsmExecutionMode: executionMode2>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 4-13: An example of the *global execution policy*

The above abstract model has also been also implemented upon the *WS-Mediator Policy framework*. Node <wsmExecutionMode> represents fault tolerance mechanisms. The *boolean* attribute "execution" indicates whether the execution mode is selected. The concrete implementation of the *global execution policy* can be found in section 4.3.8.

4.3.6 Dependability Monitoring Mechanism (DMM)

The Sub-Mediator Elite implements monitoring mechanisms to periodically monitor the registered Web Services and Sub-Mediators. The monitoring mechanisms generate dependability metadata according to monitoring results. These dependability metadata are used for resilience-explicit computing. Because the monitoring is

performed by each Sub-Mediator itself, the generated dependability metadata present the dependability of Web Services from the perspective of the Sub-Mediator. If the Sub-Mediator is deployed close enough to the client, the metadata can accurately present the dependability of the Web Services from the client's perspective.

A. Web Service Monitoring (WSM)

The WSM mechanism retrieves the information about Web Services from the *Web Service Registry*, using it to periodically invoke them. Having sent a *test SOAP message* to invoke a Web Service, the mechanism waits a certain period of time defined by the *test policy* for the result. If the latter is not returned until timeout, the test fails, and the *dependability rank* of this Web Service will be reduced. If the result is received before timeout, the monitoring mechanism checks the validity of the result. When the *test policy* specifies an expected result, the monitoring mechanism compares the received result with the *expected SOAP message*. If the messages match, the result is valid, and then the *dependability rate* of the Web Service will increase. If the expected *SOAP message* is not given, the monitoring mechanism will check the semantic validity of the result. Unless there is an error message attached to the SOAP message, the result will be regarded as valid. The monitoring mechanism also records the response time of the successful invocations, and calculates the *average*, *minimum* and *maximum response time* of Web Services.

B. Monitoring Sub-Mediators

A Sub-Mediator monitors other Sub-Mediators registered in its *Sub-Mediator Registry*. It invokes the other Sub-Mediators via a special test interface to check their

dependability, upon which the test results are processed for updating the dependability metadata of the Sub-Mediators.

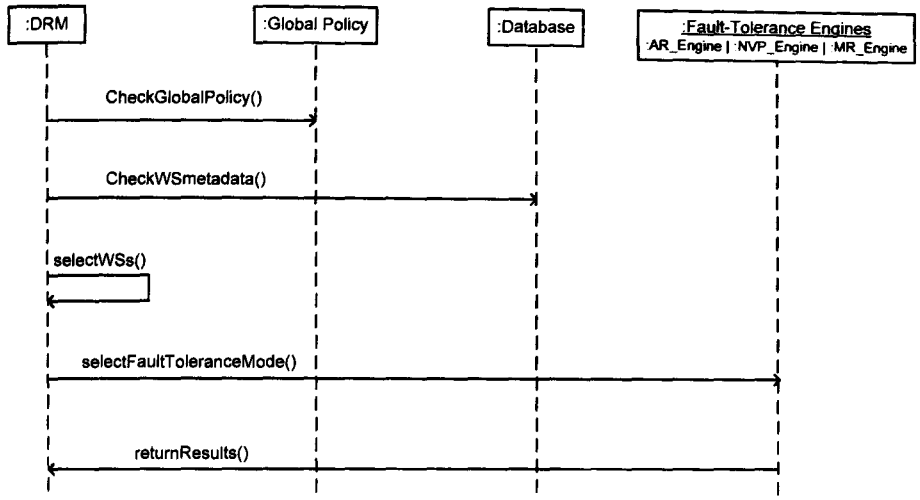

Figure 4-14: The execution sequence of the Dynamic Reconfiguration Engine

4.3.7 Dynamic Reconfiguration Mechanism (DRM)

The DRM is the core component of the Sub-Mediator Elite, which dynamically reconfigures service composition and fault tolerance mechanisms, implementing resilience-explicit computing algorithms to suit different fault tolerance mechanisms. The execution procedure of the DRM starts with checking the *global execution policy* to decide which fault tolerance mechanism to apply, and the user-defined criterion (e.g. *m, f, r*) to select component services. Then the DRM checks the metadata of component services and dynamically sorts them according to their dependability metadata. If the dependability metadata of a component service is lower than the user-defined threshold (e.g. $r_{ws} < r_{threshold}$), the component service will be removed from the candidate list. At the end, the sorted list of component services is passed to the

selected fault tolerance execution mode to perform service composition. Figure 4-14 illustrates the execution sequence of the DRM.

Below is the DRM execution procedure:

List of component services

services = {ws_1 ... ws_n}

Global execution policy

execution_mode = {*Service Alternatives* | *NVP* | *Multi-routing*}

primary_criterion = {metadata | *m, r, f* | threshold};

second_criterion = {metadata | *m, r, f*};

Metadata

{ws_1 | *m* (%), *r* (ms)}

{ws_n | *m* (%), *r* (ms)}

Sort

services_sort = services sorted by primary_criterion/second_criterion

Execute

execute(execution_mode)

End

4.3.8 Fault-tolerance Execution Modes

The DRM invokes the fault tolerance mechanisms to perform service composition. The execution procedures in the fault tolerance execution modes are different and component services are used differently, according to the particular fault tolerance techniques.

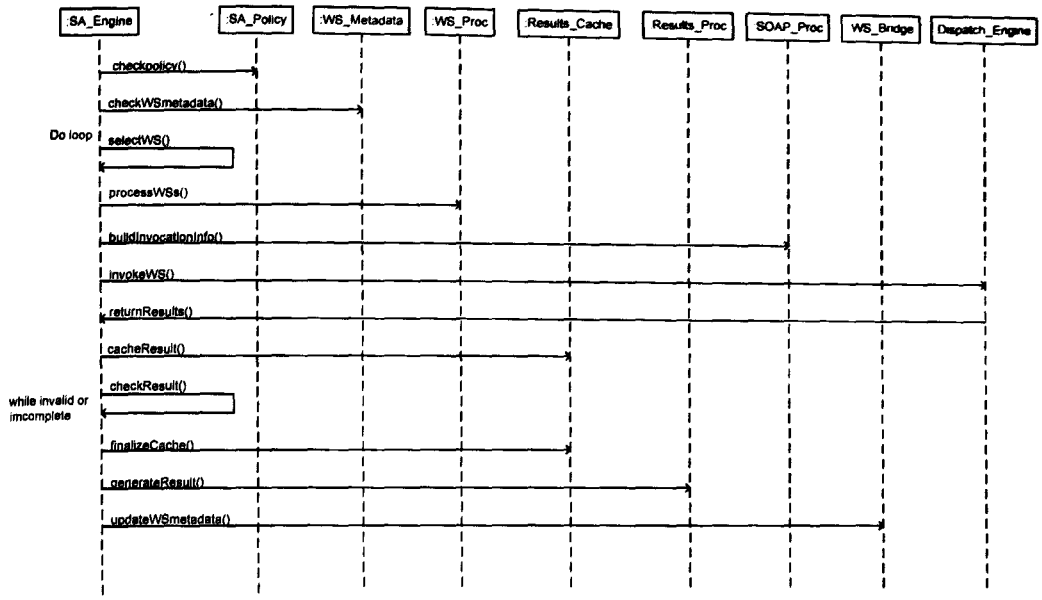

Figure 4-15: The execution sequence of the service alternative execution mode

A. Service Alternative Execution Mode.

Figure 4-15 illustrates the execution sequence of the Service Alternative execution mode. At beginning of the execution sequence, the execution engine checks the *global execution policy* to set the relative execution parameters. The *global execution policy* defined for the Service Alternative execution mode is illustrated below, followed by the explanation of the main entities.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsp:Policy xmlns:wsp=http://schemas.xmlsoap.org/ws/2004/09/policy
            xmlns:wsmgp="http://schemas.wsmediator.org/globalPolicy/policy">

    <wsp:ExactlyOne>

        <wsp:All>

            <wsmFTMode:ServiceAlternatives execution="true">
```

```
                <priority>{value}</priority>
                <dependabilityAcceptance>{value}</dependabilityAcceptance>
                <responseTimeAcceptance>{value}</responseTimeAcceptance >
                <timeout>{value}</timeout>
            </wsmFTMode:ServiceAlternatives>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

- *<wsmFTMode:ServiceAlternatives execution="true">*: this entity defines the fault tolerance execution mode. Here it indicates the *Service Redundancy* execution mode. The value *"true"* of the attribute *execution* indicates this fault tolerance execution mode is selected for processing the request. The nested entities are the parameters for this execution mode.

- *<priority>*: this sets the criterion for sorting candidate Web Services. Web Services can be sorted according to their *dependability rate* or *average response time,* as shown by their dependability metadata.
- *<dependabilityAcceptance>*: this entity sets the *minimum acceptance* of the *dependability rate.* The Web Services with a *dependability rate* lower than that will be removed from the list of candidate Web Services.
- *<responseTimeAcceptance>*: this entity sets the *maximum acceptance* of the *minimum response time.* If the *minimum response time* of any Web Service is greater than the *maximum acceptance*, the Web Service will be removed from the list of candidate Web Services.

- *<timeout>*: this sets the timeout parameter for the entire service request. If the Sub-Mediator cannot complete the request before timeout, it will return an error message to the client.

Once the execution parameters are set, the execution engine checks dependability metadata to set the parameters for invoking component services. For example, the maximum response time of a component service recorded in the dependability metadata can be used to set the timeout parameter of the invocation. Then the execution engine selects the first component service in the sorted list and invokes the service to perform service integration. Once the component service has returned the result, the execution engine checks its validity. If it is valid, the execution engine finalizes the execution procedure and returns it to the BLP. If the component service fails to deliver valid results, the next component service in the list will be invoked, and so on.

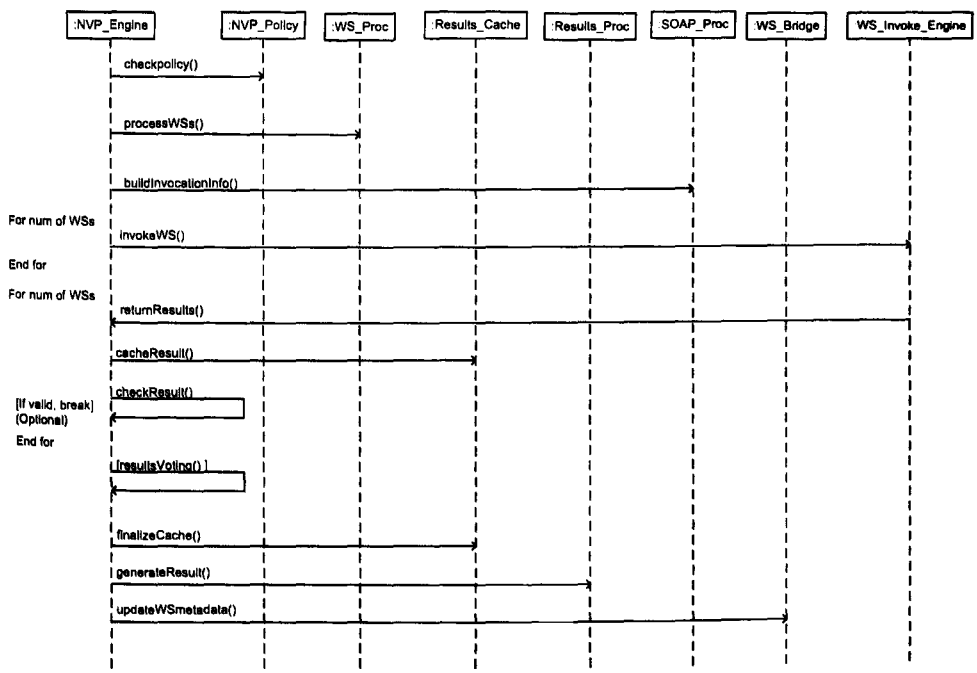

Figure 4-16: Execution sequence of the *N-version programming execution mode*

B. N-Version Programming Execution Mode

Figure 4-16 presents the execution sequence of the *N-Version Programming* execution mode. First of all, the execution engine checks the *global execution policy* to set the relative execution parameters, such as the number of synchronous invocations, the number of expected results, etc. The *global execution policy* defined for the N-Version Programming execution mode is illustrated below, followed by the explanation of the main entities.

```
<wsmFTMode:nVersionProgramming execution="true">
        <priority>{value}</priority>
        <dependabilityAcceptance>{value}</dependabilityAcceptance>
        <responseTimeAcceptance>{value}</responseTimeAcceptance >
        <resultsProcessing>{value}</resultsProcessing>
        <numberOfSyncInvocation>{value}</numberOfSyncInvocation>
        <numberOfExpectedResults>{value}</numberOfExpectedResults>
        <timeout>{value}</timeout>
</wsmFTMode: nVersionProgramming >
```

- *<resultsProcessing>*: this defines how to process the results returned from candidate Web Services. There are three options: *vote, quickest,* and *all.* In the *vote* option, the service request terminates when *result voting* is completed. In the *quickest* option, the entire service request terminates when a valid result is received. In the *all* option, the service request terminates until the invocations to the Web Services are all completed.

- *<numberOfSyncInvocation>*: in the N-Version Programming execution mode, a number of Web Services will be invoked simultaneously. This entity defines the maximum number of simultaneous invocations allowed at a time.
- *<numberOfExpectedResults>*: If the number of candidate Web Services is greater than the number of allowed simultaneous invocations, they will be divided into groups and invoked in a certain order. This entity defines the number of expected results. Once there are enough results received, the execution will be terminated.

Once the execution parameters are set, the execution engine selects the required number of component services from the candidate list, and invokes them synchronously. The results returned from component services are checked by the execution engine. If some of the invoked services fail to deliver valid results, the execution engine retrieves alternative component services from the list and invokes them until the expected number of valid results is fulfilled. Then the execution engine finalizes the execution procedure and processes the received results.

C. *Multi-Routing Execution Mode*

Figure 4-17 illustrates the execution sequence of the *Multi-Routing* execution mode. The execution engine interprets the *global execution policy* to define the execution procedure and set execution parameters. Then it checks the dependability of Sub-Mediators and selects the defined number of Sub-Mediators to implement the Multi-Routing Strategy. Similar to the *N-Version Programming* execution mode, the execution engine invokes the selected Sub-Mediators synchronously and validates the results returned by them. The execution procedure terminates when the expected number of valid results are received.

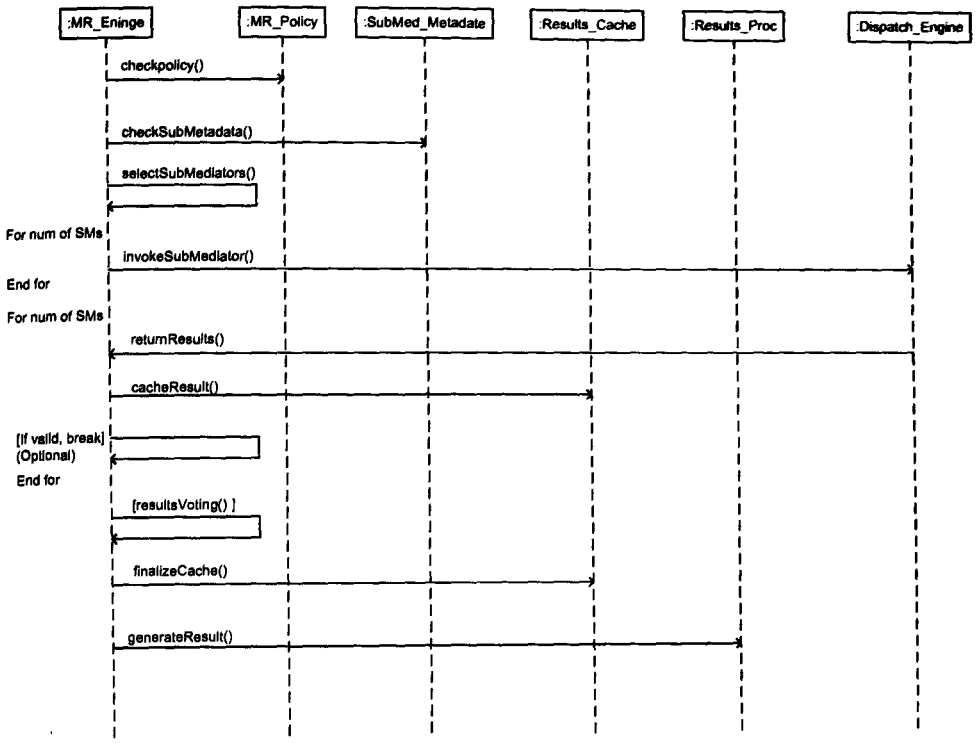

Figure 4-17: The execution sequence of the multi-routing execution mode

The *global execution policy* corresponding to the Message Routing execution mode is illustrated below, followed by the explanation of the main entities.

```
<wsmFTMode:MessageRouting execution="true">
        <dependabilityAcceptance>{value}</dependabilityAcceptance>
        <responseTimeAcceptance>{value}</responseTimeAcceptance >
        <resultsProcessing>{value}</resultsProcessing>
        <numberOfRoutes>{value}</ numberOfRoutes>
        <timeout>{value}</timeout>

</wsmFTMode: MessageRouting >
```

- *<dependabilityAcceptance>*: this entity sets the *minimum acceptance* of the *dependability rate*. If the *dependability rate* of a Web Service recorded on the participating Sub-Mediator is lower than that, the Sub-Mediator will not be selected as an intermediary for implementing the message routing.
- *<responseTimeAcceptance>*: this entity sets the *maximum acceptance* of the *minimum response time*. If the *minimum response time* of a Web Service registered on the participating Sub-Mediator is greater than the *maximum acceptance*, the Sub-Mediator will not be selected as an intermediary.
- *<numberOfRoutes>*: this entity defines the number of the messaging routes, i.e. the number of Sub-Mediators that will be selected as intermediaries.
- *<timeout>*: this sets the timeout parameter for the entire service request. If the Sub-Mediator cannot complete the request before timeout, it will return an error message to the client.

4.4 Conclusions

In this chapter, we presented the Java WS-Mediator, a prototype of the WS-Mediator system based on the Java Web Service technology. The Java WS-Mediator system is constructed of Java Sub-Mediators. The chapter also proposed an implementation of the Sub-Mediator Elite as a lightweight Sub-Mediator for local deployment, used to develop the Web Service type Sub-Mediators. In addition, we explained the structure and execution sequences of the components and mechanisms. Overall, the Java WS-Mediator proves the WS-Mediator approach can be realized on the basis of the current Web Service technologies.

5. Evaluation

5.1 Introduction

In this chapter, we describe our evaluation of the WS-Mediator approach. We have conducted a series of experiments with different application scenarios, carefully selected to represent typical Web Services applications occurring in the real world. In these experiments, we utilized the Java WS-Mediator to implement several composite applications based on real-world Web Services, developed and deployed by a variety of independent Web Service providers. The analysis of the results of the experiments will demonstrate the applicability and effectiveness of the WS-Mediator approach.

This chapter is organized as follows: section 5.2 introduces the objectives of the experiments and provides a brief outline of the evaluation of the approach. Section 5.3 reports the experiments that monitor the dependability of several real-world Web Services. We will use the results of the experiments to prove the feasibility of on-location monitoring of the dependability of generic Web Services. In section 5.4, we will focus on an experiment conducted with an e-Science application. This experiment was conducted upon three Web Services frequently used in Bioinformatics research. We have developed a realistic application based upon the Java WS-Mediator to demonstrate how to improve the dependability of e-Science workflows by adopting the WS-Mediator approach. Section 5.5 concludes this chapter.

5.2 Evaluation Objectives

The evaluation of the WS-Mediator approach is based on our experiments on the real-world Web Services. The approach was developed as a result of our studies of the latest Web Services technologies and other relevant work. The design of the solution

is compliant with the current Web Service specifications and standards. However, the applicability and the effectiveness of the approach can only be verified in real-world applications. The WS-Mediator is a generic solution that can be tailored to fit different application scenarios. We have conducted a series of experiments to verify its applicability by developing realistic applications using the prototype implementation of the approach, the Java WS-Mediator. The experiments were carefully planned to achieve the following objectives:

- To evaluate the applicability of monitoring Web Service dependability. Web Services can be autonomously deployed by independent Web Service providers or explicitly deployed by the participating providers within a virtual organization.
- To evaluate the effectiveness of the resilience-explicit dynamic reconfiguration of dynamic service composition. The resilience-explicit dynamic reconfiguration mechanism of the WS-Mediator calculates dependability metadata to make run-time decisions for selecting component Web Services. The experiments need to produce quantitative results to prove the effectiveness of the approach.
- To evaluate the applicability of fault-tolerance execution models. The fault-tolerance mechanisms that are designed to deal with the designated faults are selected by the client and dynamically applied at run-time. We need these experiments to prove that the dynamic reconfiguration of fault-tolerance mechanisms can provide flexible means of achieving Web Service dependability based on specific fault assumptions.
- To verify the ease of developing Web Service applications using the WS-Mediator system.

- To verify the message intercepting ability of the WS-Mediator system.

The above are the most important objectives of our experiments, which evaluate the core concepts and components of the WS-Mediator approach. There were also many other experiments conducted to evaluate various aspects of the approach and its prototype implementation, which are not as central for this dissertation.

5.3 Evaluation of Dependability Monitoring

Monitoring Web Service dependability is the fundamental part of the WS-Mediator approach. The dependability monitoring mechanism assesses the dependability of Web Services and generates their dependability metadata. Resilience-explicit computing adapted to the WS-Mediator approach relies on dependability metadata to make decisions. Our research emphasises the notion of Web Service dependability from the client's perspective. This requires on-location monitoring of Web Services at the same locations where clients run their applications. In chapter 4, we described how this approach was achieved in the Java WS-Mediator. The experiments reported in this section will emphasize the feasibility of the approach by demonstrating the dependability monitoring of real-world Web Services using the Java WS-Mediator.

As we have shown above, Web Services used in an application can either be deployed by autonomous providers or by cooperative providers to the client. These autonomous Web Services can be discovered from the UDDI or from another registry of Web Services. Commonly, providers only reveal limited information that is sufficient only for invoking their Web Services. No collaboration between the client and the service provider is expected in such application scenarios, and so such Web Services are typically regarded by clients as *black box* components. Since message-exchanging

between the client and Web Services is guaranteed by the Web Service Interoperability standards, the implementation of the client application and of Web Services both need to be compliant with the Web Services Interoperability. This is one of the fundamental principles in developing a generic Web Service, although this may not be a crucial criterion for the Web Services that are developed only to serve the correlative clients, because of the possibility of implementing corresponding mechanisms in the client application. However, unless this may bring additional benefits, it is always undesirable to undermine the interoperability of a Web Service. Most Web Services and client applications are developed upon the existing Web Services middleware (e.g. Aparche Axis [32], JBoss [31], and Glassfish [33]) which provides underlying infrastructure to support the interoperability of the Web Service applications by default, and so for a generic solution such as the WS-Mediator, it is safe to consider the Web Services as universally interoperable. Furthermore, specific mechanisms can always be implemented in addition to the standard invocation mechanisms to cope with the corresponding changes at the Web Service side. Below Web Services are assumed to be interoperable, enabling the invocation mechanisms of the Java WS-Mediator to invoke them without modification.

The evaluation of dependability monitoring was conducted on a number of autonomous Web Services in addition to those deployed by our colleagues for their research project. In the following text, we will report the experiments.

5.3.1 Dependability Monitoring of Public Web Services

In order to validate the ability of the Sub-Mediator Elite to monitor the dependability of real-world Web Services, we randomly discovered some publicly deployed Web Services from a popular Web Services publisher, *The XMethods* [1]. These Web

Services are deployed by different service providers and upon different platforms, as listed below:

- *WS1*: Get conversion rate from one currency to another currency

  Endpoint: http://www.webservicex.com/CurrencyConvertor.asmx?wsdl

- *WS2*: Lotto Number Generator

  Endpoint: http://reto.checkit.ch/Scripts/Lotto.dll/wsdl/IgetNumbers

- *WS3*: Returns the date of Easter for a given year

  Endpoint: http://www.stgregorioschurchdc.org/wsdl/Calendar.wsdl

- *WS4*: Translate English to Pig Latin

  Endpoint:

  htttp://www.aspxpressway.com/maincontent/webservices/piglatin.asmx?wsdl

- *WS5*: Find a ZIP Code given a U.S. City and State

  Endpoint: http://ws.strikeiron.com/InnerGears/ZipByCityState2?WSDL

We deployed the Sub-Mediator Elite on a computer connected to the Campus network of Newcastle University and registered the selected Web Services for dependability monitoring. These Web Services all provide very simple services, returning responses according to the client's inputs. A test script was written for each Web Service according to its WSDL interface, and a global test policy defined to set the parameters for monitoring them. During the experiments, 100 invocations were made on each

Web Service with the interval between each invocation being 60 minutes (see Figure 5-1).

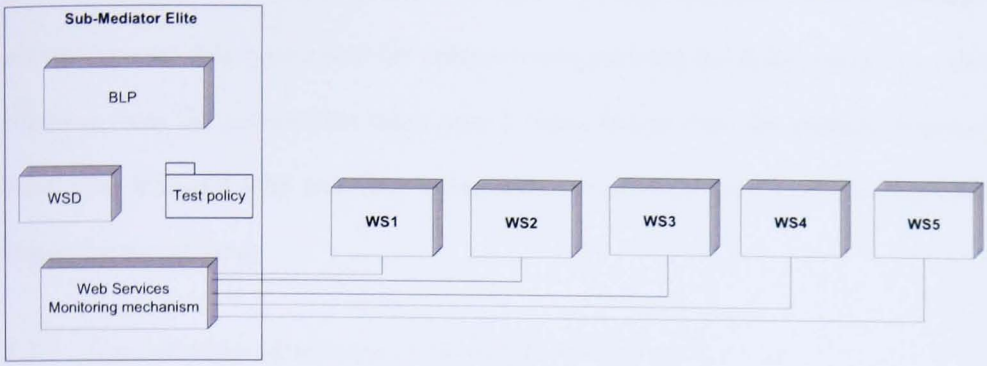

Figure 5-1: Dependability monitoring of autonomous Web Services

There were no technical problems in the interaction between the Sub-Mediator Elite and the Web Services. The Sub-Mediator Elite invoked the Web Services successfully and received expected results from the Web Services except for failures of some of the Web Services.

Web Services	Invocations	Average response time	Dependability rate	Unusual delays	Failures		
					Service failures	Omission failures	time out
WS1	100	152	100%	3	0	0	0
WS2	100	175	100%	7	0	0	0
WS3	100	132	93%	5	0	3	4
WS4	100	186	17%	0	83	0	0
WS5	100	119	95%	9	1	2	2

Table 5-1: Dependability monitoring results of the public Web Services

Table 5-1 shows the results of dependability monitoring. Four of the Web Services achieved a high rate of dependability during the monitoring. The *WS4*, which

translates English to Pig Latin, was successfully invoked 17 times but became inactive thereafter, providing only an error message indicating that unknown service failures occurred in the service. The *WS1* and *WS2* were the most reliable, although several unusual delays occurred for unknown reasons (*unusual delay* refers to a valid response from the service that takes over 2 times longer than the average response time). The *WS3* and *WS5* were less dependable with varied types of failures captured during the monitoring.

5.3.2 Dependability Monitoring of the GOLD Web Services

The results presented in the previous section demonstrate the ability of the Sub-Mediator Elite application to monitor the dependability of autonomous Web Services. The monitoring mechanism of the Sub-Mediator Elite successfully recorded the dependability behaviour of Web Services and generated their dependability metadata. However, we could not obtain confirmation from the service providers about the correctness of the monitoring results due to the autonomy of Web Services. In addition, the reasons behind some of the failures and delays of the Web Services were unknown to us. We have therefore conducted additional experiments to verify the validity of dependability monitoring using two Web Services kindly provided to us by colleagues working on the GOLD project [2]. These two Web Services were

- *GOLDPeople*: a Web Service returning the list of the people in the GOLD project.
- *GOLDPolicies*: a Web Service returning the aggregation of the policies developed for the GOLD project.

The two Web Services are formal Web Services deployed for research purposes. However, they are by no means expected to be reliable because they are also used for software testing and debugging. Therefore, these two Web Services may behave unreliably when software testing and debugging are taking place on servers.

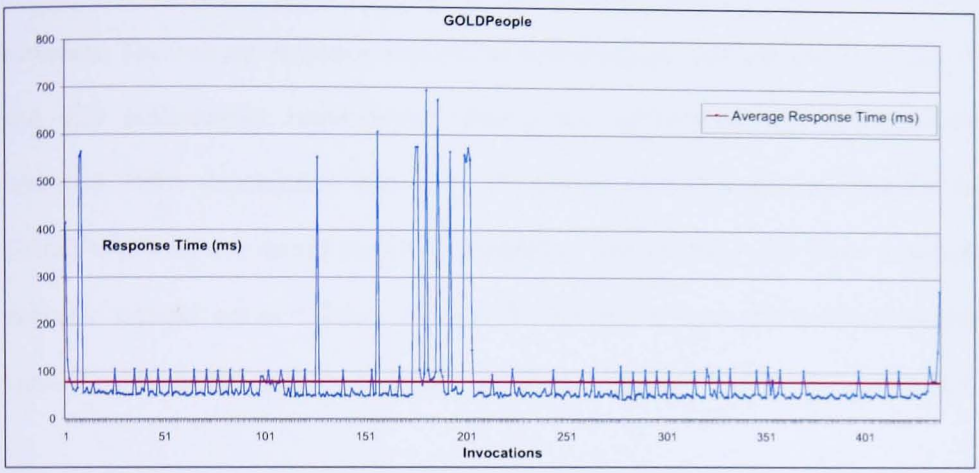

Figure 5-2: Dependability monitoring result of the *GOLDPeople*

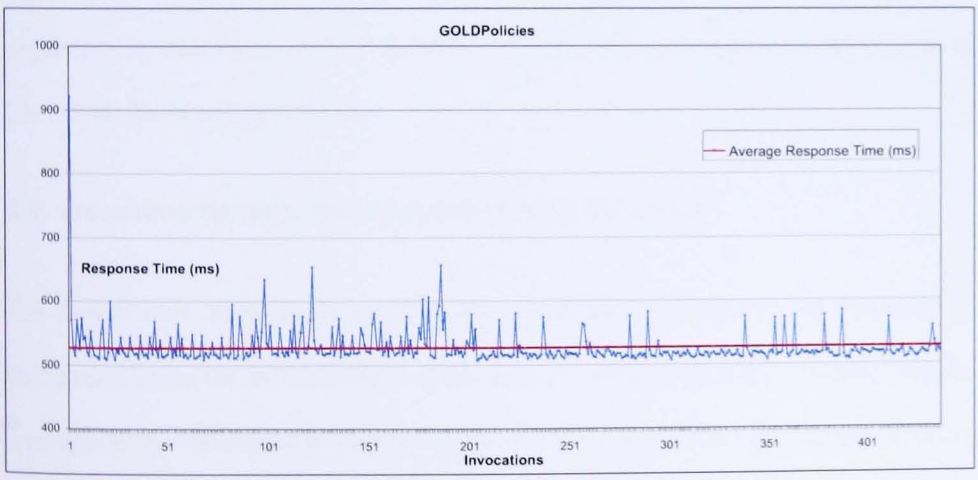

Figure 5-3: Dependability monitoring result of the *GOLDPolicies*

The two Web Services are deployed on the campus network of Newcastle University. We deployed the Sub-Mediator Elite on a computer connected to the same network. The WS-Mediator Elite performed dependability monitoring on the two Web Services and logged the returned results. Figure 5-2 and Figure 5-3 illustrate the results of the dependability monitoring of the two Web Services, as shown in their dependability metadata. The average response time of the *GOLDPeople* and *GOLDPolicies* are 77 and 526 milliseconds respectively. During the monitoring, the *GOLDPolicies* remained 100% dependable. However, 13 service failures were recorded for the *GOLDPeople* service based on its dependability rate of 96%. The error messages indicated internal server failures in the *GOLDPeople* services representing ongoing unusual activities taking place on the server which were confirmed by our colleagues.

The dependability monitoring of the GOLD services proves the applicability and feasibility of on-location dependability monitoring mechanism implemented in the WS-Mediator. The generated dependability metadata can accurately represent the dependability behaviour of Web Services. The above experiment was reported in the UK All Hands Meeting 2006 [3].

5.4 Experiments with Bioinformatics Web Services

The experiments reported above prove the capability and feasibility of dependability monitoring using the WS-Mediator. They provide effective and quantitative evidence concerning the dependability behaviour of Web Services. The dependability metadata generated serve as a sufficient precondition to achieve resilience-explicit computing. Thus we were able to carry out a complete evaluation of the entire WS-Mediator system. Below we report experiments on three Bioinformatics Web Services aimed at demonstrating the applicability and effectiveness of the WS-Mediator approach.

In chapter 2, we presented experimental work analyzing the dependability of two BLAST Web Services used in the bioinformatics domain [4]. BLAST is an algorithm which is commonly used in *in silico* experiments in bioinformatics to search for gene and protein sequences that are similar to a given input query sequence [5]. We discovered dramatically different dependability characteristics of the BLAST Web Services. Dependability characteristics of each BLAST Web Service also varied when monitored from different geographical locations. Our analysis shows that the existing BLAST services are likely to offer a reasonable degree of diversity despite the fact that they all execute the same basic matching algorithms. This is due to differences between the DBs, the specific BLAST searches they execute, the hardware they are deployed on and the software code they run. This adds to the diversity of their geographical locations.

In order to evaluate the WS-Mediator approach, we conducted experiments on three BLAST Web Services with the Java WS-Mediator deployed on a computer in the campus of *Newcastle University, UK.* The experiments demonstrate the applicability of the WS-Mediator approach by employing it to real Web Services used in e-Science environment. The three BLAST Web Services involved in this case study are:

- The BLAST Web Service deployed by the European Bioinformatics Institute (EBI), Cambridge, UK [6]
- The BLAST Web Service hosted by the DNA Databank, Japan (DDBJ) [7]
- The BLAST Web Services hosted by Virginia Bioinformatics Institution (VBI), USA [8]

Before the experiment started, test scripts were submitted for monitoring each Blast Web Service and generating their dependability metadata (see Appendix C for the pattern and explanation of dependability metadata). The three services were monitored synchronously at an interval of 5 minutes between invocations. Appendix D shows some of the dependability metadata. Thus, the Java WS-Mediator can use the dependability metadata to perform resilience-explicit computing and to select the appropriate Web Services for service composition.

In our experiments, we have developed a Java client application based upon the Java WS-Mediator. This application (see Appendix E) uses the three BLAST Web Services as candidates and searches the genetic databases of the three Blast Web Services for a match to an input query sequence. An example of the expected result is shown in Appendix F. The Java client application invokes the request every 30 minutes. If erroneous replies are returned from a service, the client application makes three tries before switching to the redundant services. The interval between retries is 30 seconds. The timeout periods of the three Web Services are set automatically by the Sub-Mediator according to their maximum response time recorded in the metadata. We used the Service alternatives, N-version programming and Multi-routing execution modes in the experiments and logged the execution results for analysis. The example of successful and unsuccessful execution results of the business process are shown in Appendix G and Appendix H respectively. The execution results list the execution procedures performed during the business logic processing, and show the result of each step carried out during the execution. The final result of service execution and the execution report are attached to the execution results.

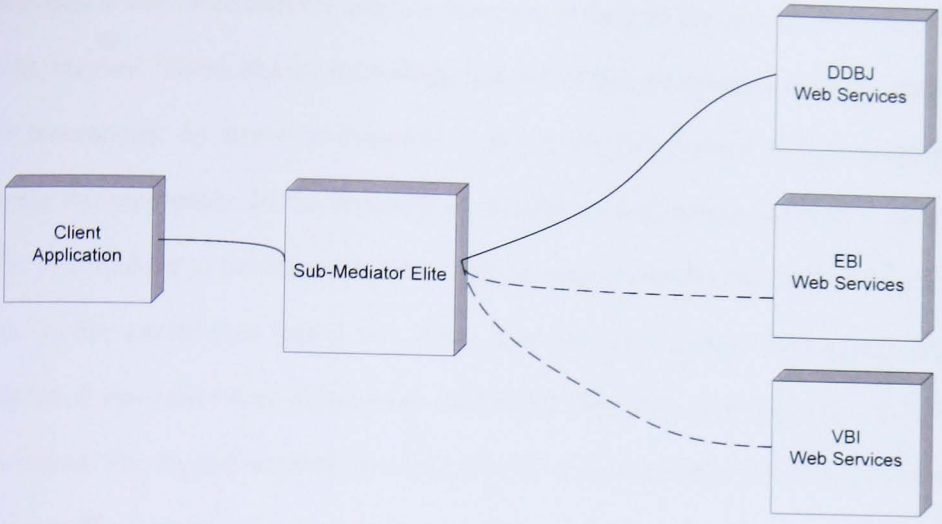

Figure 5-4: Evaluation of the Service alternative execution mode. The solid lines represent fixed or primary, and the dashed lines alternative message routes

5.4.1 Service Alternative Execution Mode

Figure 5-4 shows the application for evaluating the Service alternative execution mode. In the experiment, we set the dependability measurement (m) as the criterion for selecting the best component service. At the beginning of the run, the three BLAST Web Services were dynamically ordered by the WS-Mediator according to their dependability measurement (m) during the preceding execution. As the DDBJ was the most dependable Web Service, it was used as the primary BLAST Web Service. However, at some moment during the execution, the DDBJ became unreliable, repeating the message: "*The search and analysis service is very busy now. Please try again later.*" In these circumstances, the WS-Mediator switched to using the VBI after failed attempts with the DDBJ. The VBI returned valid results in most attempts. Because the DDBJ was not in a dependable state, its dependability measurement (m) dropped dramatically. Figure 5-5 shows the results of the experiment. From the moment shown in Figure 5-5 as point (A), the VBI became the

most dependable Web Service and was therefore chosen as the primary Web Service to be invoked. There was an interesting contrast of two switching sequences during the invocations. As shown in Figure 5-5, there were two entirely failed executions during the experiment. In the first one (see Figure 5-5, Point (B)), the DDBJ was the first Web Service to be called, the VBI was the second one and the EBI was the last one. In the second (see Figure 5-5, Point (C)), the VBI became the primary Web Service. It was called first, followed by the DDBJ. The EBI was still the last one to be attempted. The logged metadata generated by the monitoring mechanism ensured that the switching sequences were correct according to the dependability metadata at the time. In this execution mode, the average overhead of the Java WS-Mediator is only about 100 milliseconds. The average response times of the DDBI, VBI and EBI were about 24 seconds, 29 seconds and 63 seconds respectively.

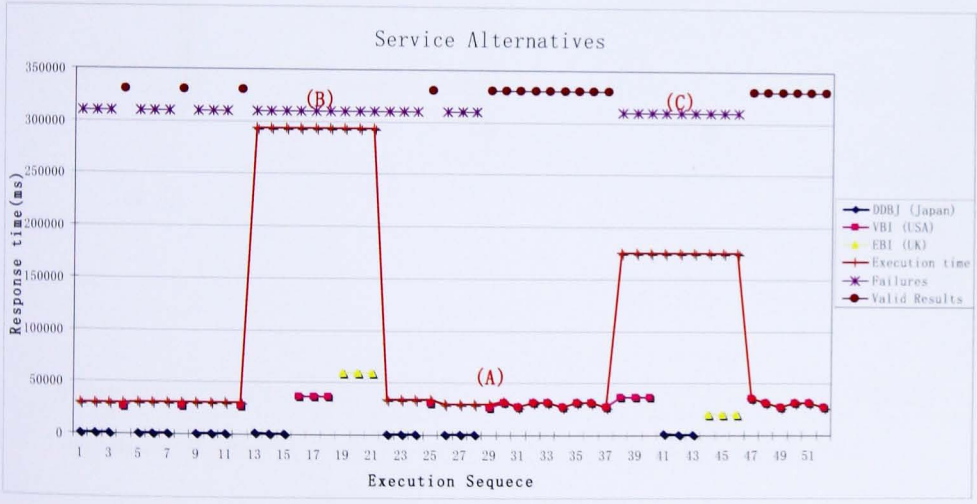

Figure 5-5: Results of the Service alternative execution mode

5.4.2 N-version Programming Execution Mode

Figure 5-6 shows the application for evaluating the N-version programming execution

mode. In this experiment, all of the three Web Services were invoked simultaneously. Once the quickest result is obtained from a Web Service, the execution terminates. This strategy is slightly different from the classic N-version programming technique, which commonly requires voting on results. However, in real-world Web Services applications, it is not always possible to vote on the results received from diverse services. The results can be semantically equivalent or similar when the SOAP messages are literally different. Therefore, in the WS-Mediator, result voting is optional. We believe the client should have better knowledge about how to process the results.

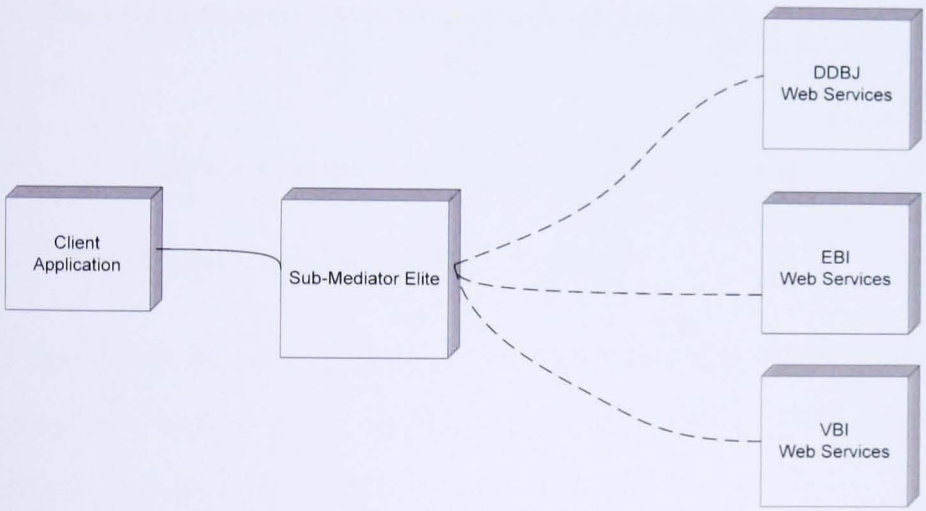

Figure 5-6: Evaluation of the N-version programming execution mode. The solid line represents a fixed message route, and the dashed lines redundant message routes

Figure 5-7 shows a proportion of the results collected in the N-version programming execution mode. Because the DDBJ and the EBI were, for unknown reasons, in very unstable states, they failed to provide valid results to the invocations. The final results of all executions were returned from the VBI. In this execution mode, the overhead of

the Java WS-Mediator was about 130 milliseconds. It was slightly higher than that in the Service alternative execution mode.

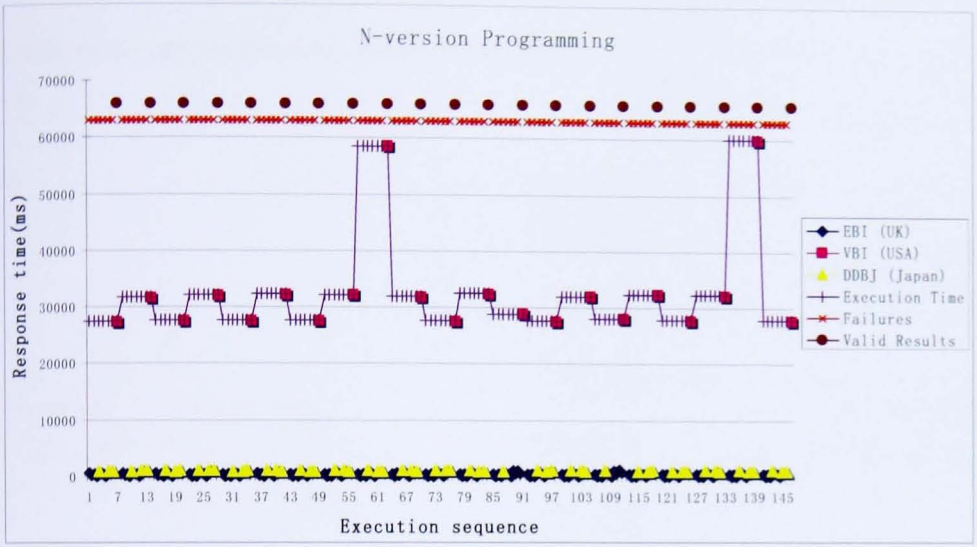

Figure 5-7: Results of the N-version programming execution mode

5.4.3 Multi-routing Execution Mode with the Planetlab

We deployed six Remote Sub-Mediators at six different sites on PlanetLab in the Multi-routing execution mode. PlanetLab is an open platform for developing, deploying, and accessing planetary-scale services [9],which provides a global research network for developing and experimenting with network services.

The six sites where we deployed the Sub-Mediators were located in China, UK and USA as illustrated in Figure 5-8. In each country, we deployed two Sub-Mediators in two different cities. The geographical locations of the Sub-Mediators were registered in the Mediator-Elite deployed on a computer in the Campus network of Newcastle University. This computer acted as the client's terminal. Such deployment was implemented with applying geographical diversity in mind. However, it is worth

mentioning that this experiment did not emphasize the selection of diverse network paths between the sites and the possible network overlap between the Sub-Mediators and the candidate Web Services. This experiment was designed only to validate the applicability and functionality of the WS-Mediator.

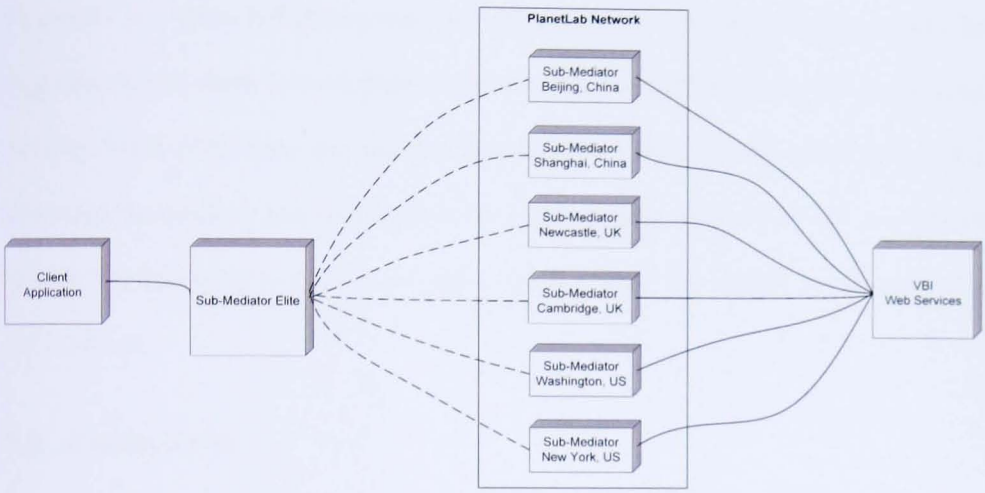

Figure 5-8: Evaluation of the multi-routing execution mode. The solid lines represent fixed or primary message routes, and the dashed lines alternative routes.

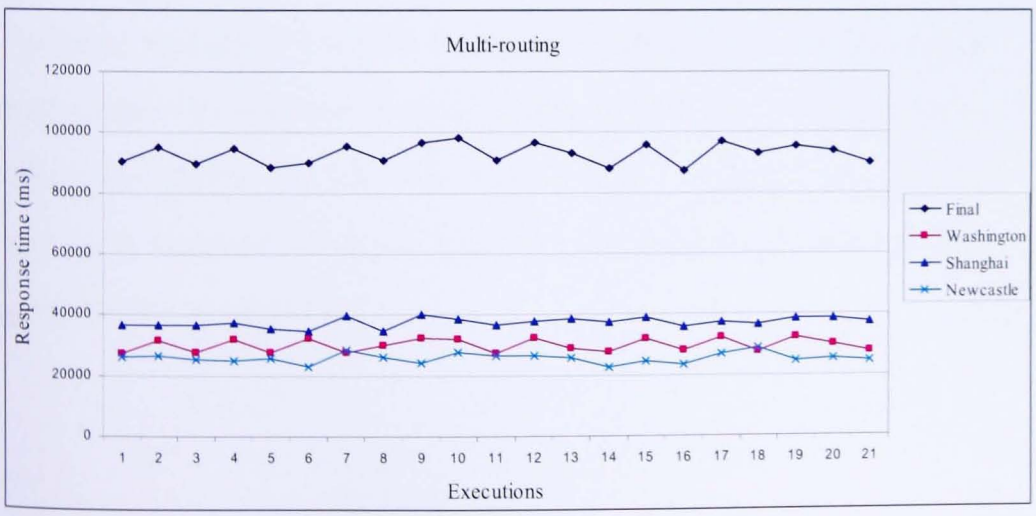

Figure 5-9: Results of the Multi-Routing execution mode

In this experiment we chose the VBI BLAST as the ultimate Web Service. Three routes with dependability acceptance of 70%were required. The level of routing diversity was set as "Country". During the execution, The Sub-Mediators located in Shanghai (China), Newcastle upon Tyne (UK), and Washington (USA), were selected as the routing intermediate nodes according to their dependability metadata (see Appendix I). Figure 5-9 shows some results obtained in this experiment. During the experiment, the three Sub-Mediators and the VBI BLAST Web Service performed reliably. Most of the time, the Sub-Mediator deployed in Newcastle upon Tyne (UK), delivered the quickest responses, while the one in Shanghai (China), was the slowest one. In this execution mode, the average overhead of the WS-Mediator was about 140 milliseconds.

5.5 Conclusions

The experiments reported in this chapter demonstrate the applicability of the WS-Mediator approach. The experiments were conducted with realistic Web Services deployed by diverse service providers in real-world environments. The results of the experiments have proved that the WS-Mediator is capable of providing the required functionalities. The quantitative evidence supports the evaluation of the approach as feasible and effective. The experiments conducted with the BLAST Web Services have clearly manifested the benefits of using the WS-Mediator approach with real-world Web Service applications.

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 studies reported in each chapter. In section 6.2, we outline certain possible extensions that could be made to our solutions. In addition, we discuss how the knowledge gained in this study can be applied in future work to improve the dependability of Web Service applications.

6.1 Summary

Web Service technology is developing very fast, and has started to play a critical role in more and more e-Commerce and e-Science applications. Due to the complexity of architecture and complicated application scenarios of Web Services, their dependability is a challenging research topic. While there have been many approaches developed to improving the dependability of individual Web Services and Web Service composition applications, there is still a need for solutions that would ensure the dependability of Web Service composition given the persistence of varied types of faults in the infrastructure. It is therefore essential to analyse concrete dependability characteristics of Web Services and involved components, such as individual component services, networks, etc. and develop solutions to cope with specific fault assumptions.

Web Service composition is an activity involving integration of several component services over computer networks. For instance, in the *travel booking* use case, the travel agent has to invoke both an airway company and a hotel to follow the business process logic. In practice, applications (e.g. [8, 9]) will be much more complicated

and service composition will involve far more component services for the business process logic to be implemented. The dependability of service composition relies on the dependability of individual component services and of the networks. Failures of a single node (e.g. a component service or a segment of the network) can undermine the dependability of the entire application. In our example, the travel booking process cannot be accomplished until the travel agent receives valid results from both the airway company and the hotel. However, in reality, it is impossible to ensure that Web Services do not fail during the integration; moreover, computer networks are inherently unreliable. Hence, solutions for improving the dependability of service composition need to deal with failures of individual component services and networks to ensure the continuity of services.

All this has prompted us to develop an approach focusing on the dependability of Web Service composition specifically from clients' point of view, with network failures considered to be part of the dependability characteristics of component Web Services. Compared to the existing solutions, the WS-Mediator approach innovatively adapts the resilience-explicit computing technology to improve the efficacy of fault tolerance techniques (including the service diversity strategy), commonly employed in other solutions. The WS-Mediator system utilises Sub-Mediators, deployed on the overlay architecture, to monitor the dependability of component services, generate dependability metadata reflecting clients' point of view and apply fault tolerance techniques to deal with faults. Dependability metadata consist of various attributes that represent the dependability characteristics of Web Services, such as response time, availability rate, types of failures, etc. The resilience-explicit dynamic reconfiguration mechanism of the WS-Mediator system makes run-time decisions according to these metadata to dynamically select the most dependable component

services for assembling the business process logic. In addition, the system implements a number of fault tolerance mechanisms (such as recovery blocks, N-version programming and path diversity) to deal with various types of faults in order to ensure the overall dependability of the service composition.

A prototype of the WS-Mediator system, called Java WS-Mediator, has been implemented using the Java Web Service technology. We have conducted a series of experiments with several real-world Web Services (e.g. the BLAST Web Services commonly used in the bioinformatics domain, and Web Services deployed by the GOLD project, etc) to evaluate our solution, and their results have demonstrated the applicability and efficacy of the WS-Mediator approach.

6.2 Suggestions for Future Work

The architecture of the WS-Mediator system is flexible and scalable, and there are many ways in which our system could be extended in future research. Below we outline several promising extensions:

1. The efficacy of the WS-Mediator approach relies on dependability metadata and the design and implementation of the dynamic reconfiguration mechanism. Currently, the WS-Mediator system generates dependability metadata comprising attributes such as response time (r), availability measurement (m) and types of failures (f). The dynamic reconfiguration mechanism utilises these attributes to select the most appropriate component services. In future development, this solution could be extended to a comprehensive metadata framework comprising more attributes to represent other dependability characteristics of Web Services, including their changing

dependability behaviour. For example, the response time (r) or availability measurement (m) of a service may be consistently different at different times of the day or on different days of the week because of the variations in the way the service is accessed. Therefore, metadata may comprise an attribute recording the average response time (r) or availability measurement (m) at a certain time of the day, on a certain day of the week, etc. Another example would be an attribute registering the average system down time [19, 34] after the occurrence of each type of failure, which would allow the service composition mechanism to decide when to retry the service after the occurrence of a certain type of failure. The dynamic reconfiguration mechanism could then be accordingly extended by more advanced algorithms corresponding to each particular attribute of metadata or their combinations. In particular, when the response time (r) or availability measurement (m) is chosen as a criterion for selecting component services, a new algorithm should be able to use a time slice of historic response time (r) or availability measurement (m) of a candidate service to forecast its changing dependability behaviour. Thus the algorithm can explicitly decide if it is reasonable to use the service at a certain time regardless of its overall response time (r) or availability measurement (m).

2. The WS-Mediator system implements a number of fault tolerance mechanisms as fault tolerance execution modes to deal with different types of faults. There are two major ways to select a fault tolerance mechanism during service composition: explicit selection by the client and automatic selection by the WS-Mediator system. The client can select a particular fault tolerance execution mode and set relevant parameters in the *global execution policy*. In

practice, however, because the dependability characteristics of autonomous component services are unknown, it may be difficult for the client to select the appropriate fault tolerance execution mode. The dynamic reconfiguration of the WS-Mediator system is designed to automatically select the most appropriate fault tolerance mechanisms according to the types of failures (*f*) captured in the dependability metadata related to particular component services. Currently, the efficacy of the approach is restricted by the simple form in which dependability metadata are recorded (for example, the types of failures are saved and analysed at a very coarse level). This could be improved in the future by developing a more efficient dynamic reconfiguration mechanism in conjunction with a more comprehensive metadata framework. In particular, specific algorithms could be developed to identify the common types of failures in component services at a much finer level (e.g. following the classification from [81]) and to select the suitable fault tolerance mechanisms to be applied in service composition.

3. The current development of the WS-Mediator system does not explicitly address security issues, and yet Web Service security is emerging as an active research topic today. There are several types of security techniques developed for Web Services, one of the most important being the OASIS Web Services Security (WSS) TC [82]. The WS-Mediator system implements the standard Web Service intermediary architecture, which is extensively employed in many applications implementing value-adding services between clients and Web Services. The special requirements of the Web Service architecture is realised in the research on security of Web Services. Paper [83] emphasises that the development of security models and mechanisms in Web Services

should be compatible with Web Service architecture, including such components as intermediaries. Therefore, in theory, the WS-Mediator should be compatible with those applications that employ security models and mechanisms described in [82]. This supposition needs, however, to be investigated in future work.

4. The Business Process Execution Language (BPEL) [84] has been extensively used in developing e-Commerce and e-Science applications in the past few years. Compared to the Java Web Service technology, BPEL simplifies service composition by specifically focusing on the description of the business process logic, with other jobs left to the underlying middleware. The WS-Mediator system offers the standard Web Service interface and can therefore be seamlessly integrated into applications developed in the BPEL. The executable process can directly invoke the WS-Mediator system to perform service composition. However, generally speaking, the BPEL is not as powerful as a general-purpose programming language like Java with regard to tasks such as message processing, etc. Therefore, it is well worth investing some effort in the future in improving the applicability of the WS-Mediator system to the development of applications in the BPEL.
5. The WS-Mediator approach addresses network-related issues in Web Service composition, using the message routing diversity mechanism to deal with some of them. Currently, message routing diversity is achieved by using several remote Sub-Mediators as intermediary nodes. However, some overlaps of message paths may still happen when we use this application-level message routing approach. In future, the message routing diversity mechanism could be implemented in a more elaborate way to discover low-level message paths by

tracing messages sent to services. This message routing information needs to contain specific network routes along which messages between the client and the service travel. By comparing message routing paths to a particular service from different Sub-Mediators, the WS-Mediator should be able to effectively select the less overlapping paths to implement path diversity to the service. Furthermore, by tracing messages, the WS-Mediator might be able to identify the dependability characteristics of particular networks and select message routing paths during service composition accordingly.

6. The WS-Mediator system monitors Web Services at different locations in the Internet and dynamically assesses their dependability. The dependability metadata generated by Sub-Mediators can help clients to select the most dependable services, taking into consideration the impact of the network. Currently, these dependability metadata can be retrieved via the Web Service interface of Sub-Mediators. In future, it would be possible to publish these dependability metadata on a special Web site. The system would automatically detect the IP address of the user who accessed it and dynamically publish dependability metadata generated by the Sub-Mediator closest to the user. This would help users to easily find out how dependable Web Services were and use them accordingly. At the same time, Web Service providers could use the Web site to obtain the dependability metadata about their services generated by Sub-Mediators distributed across the Internet.

Bibliography

1. W3C. (2004). 'Web Services Architecture'. [cited 30 Jan 2008]; Available from: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#service_oriented_architecture

2. W3C. (2004). 'Web Services Glossary -W3C Working Group Note 11 February 2004'. [cited 30 Jan 2008]; Available from: http://www.w3.org/TR/ws-gloss/

3. Attiya, H. and Welch, J., 2004. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. 2nd edition. Wiley series on parallel and distributed computing. New Jersey: John Wiley & Sons.

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

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

6. Google. 'Google SOAP Search API (Beta)'. [Retrieved: 03 March 2008]; Available from: http://code.google.com/apis/soapsearch/reference.html

7. Ebay. (2008). 'Ebay Developers Program'. [Retrieved: 03 March 2008]; Available from: http://ebaydeveloper.typepad.com/

8. Townend, P., Xu, J., Yang, E., Bennett, K., Charters, S., Holliman, N., Looker, N., and Munro, M., 2005. 'The e-Demand project: a summary'. in *Proceedings of the Fourth UK eScience All-Hands Meeting*. Nottingham, UK.

9. Hiden, H., Conlin, C., Perrioellis, P., Cook, N., Smith, R., and Wright, A.R. (2006). 'The GOLD Project: Architecture, Development and Deployment'. [Retrieved: 30 Jan 2008]; Available from: http://www.ncl.ac.uk/ceam/research/publication/46755

10. Gable, J. (2002). 'Enterprise application integration'. *Information Management Journal,* Issue: March/April 2002. [Retrieved: March/April 2002]; Available from: http://findarticles.com/p/articles/mi_qa3937/is_200203/ai_n9019202

11. Object Management Group. (2007). 'Catalog of Specialized CORBA Specifications'. [Retrieved: 30 Jan 2008]; Available from: http://www.omg.org/technology/documents/spec_catalog.htm

12. Orfali, R., harkey, D., and Edwards, J., 1997. *Instant CORBA*. USA: John Wiley & Sons, Inc.

13. WS-I. (2007). 'Basic Profile Version 1.2'. [Retrieved: 30 Jan 2008]; Available from: http://www.ws-i.org/Profiles/BasicProfile-1_2(WGAD).html

14. W3C. (2007). 'HTTP - Hypertext Transfer Protocol'. [Retrieved: 30 Jan 2008]; Available from: http://www.w3.org/Protocols/

15. W3C. (2007). 'SOAP Version 1.2'. [Retrieved: 30 Jan 2008]; Available from: http://www.w3.org/TR/soap/

16. W3C. (2001). 'Web Services Description Language (WSDL) 1.1'. [Retrieved: 30 Jan 2008]; Available from: http://www.w3.org/TR/wsdl

17. Ferguson, D.F., Storey, T., Lovering, B., and Shewchuk, J. (2003). 'Secure, Reliable, Transacted Web Services: Architecture and Composition'. [Retrieved: 25 Feb 2008]; Available from: http://msdn2.microsoft.com/en-us/library/ms996535.aspx

18. 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*. IEEE Computer Society Press. p. 473- 477

19. Kalyanakrishnan, M., Iyer, R.K., and Patel, J., 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*. IEEE Computer Society Press. p. 418-423

20. Cristian, F., 1991. 'Understanding fault--tolerant distributed systems', in *Communications of the ACM*. Vol. 34, Issue 4: p. 56-78.

21. Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C., 2004. 'Basic Concepts and Taxonomy of Dependable and Secure Computing', in *IEEE Transactions on Dependable and Secure Computing*. IEEE Computer Society Press. Vol. 1, No. 1: p. 11-33.

22. Lee, P.A. and Anderson, T., 1990. *Fault Tolerance: Principles and Practice*, 2nd edition. J.C. Laprie, A. Avizienis, and H. Kopetz (editors). Springer-Verlag New York, Inc.

23. Chen, Y. and Romanovsky, A., 2006. 'A Mediator System for Improving Dependability of Web Services', in *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2006.* Philadelphia, USA. Vol. Supplemental: p. 132-133.

24. Atkinson, M. and Trefethen, A. (2006). 'UK e-Science ALL HANDS MEETING'. [Retrieved: 30/12/2006]; Available from: http://www.allhands.org.uk/2006/

25. Chen, Y., 2006. 'On Improving Dependability of Web Services by employing the Mediator System', in *ReSIST Student Seminar.* San Miniato, Italy.

26. Chen, Y. and Romanovsky, A., 2008. 'WS-Mediator for Improving the Dependability of Web Services Integration', in *Journal of IT Professionals.* IEEE Computer Society Press. Vol.10, No. 3, Issue: May/June 2008: p. 29-35

27. Anderson, T., Andrews, Z., Fitzgerald, J., Randell, B., Glaser, H., and Millard, I., 2007. 'The ReSIST Resilience Knowledge Base', in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* Edinburgh, UK, Vol. Supplemental.

28. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., and Yergeau, F. (2006). 'Extensible Markup Language (XML) 1.0 (Fourth Edition)'. [Retrieved: 03 March 2008]; Available from: http://www.w3.org/TR/xml/

29. OASIS. (2004). 'UDDI Version 3.0.2'. [Retrieved: 30 Jan 2008]; Available from: http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm

30. Web Services Interoperability Organization. 'About WS-I'. [Retrieved: 30 Jan 2008]; Available from: http://www.ws-i.org/about/Default.aspx

31. JBoss Labs. 'JBoss Web Services'. [Retrieved: 16 Jan 2008]; Available from: http://labs.jboss.com/jbossws/

32. The Apache Software Foundation, 'Web Services - Axis'. [Retrieved: 16 Jan 2008]; Available from: http://ws.apache.org/axis/

33. Glassfish Community. 'GlassFish Project - Documentation Home Page'. [Retrieved: 27 March 2008]; Available from: https://glassfish.dev.java.net/javaee5/docs/DocsIndex.html.

34. Oppenheimer, D., Ganapathi, A., and Patterson, D., 2003. 'Why Do Internet Services Fail, and What Can Be Done About It?', in *Proceedings of USENIX Symposium on Internet Technologies and Systems.* Seattle, USA, Vol. 3: p.1-1.

35. Han, J. and Watson, D., 2006. 'An Experimental Study of Internet Path Diversity', in *IEEE Transactions on Dependable and Secure Computing.* IEEE Computer Society Press. Vol. 3, Issue 4: p. 273 - 288.

36. Mendonça, N.C. and Silva, J.A.F., 2005. 'An Empirical Evaluation of Client-side Server Selection Policies for Accessing Replicated Web Services', in *Proceedings of the 2005 ACM symposium on Applied computing.* Santa Fe, New Mexico: ACM. p. 1704-1708.

37. Chen, Y., Li, P., and Romanovsky, A., 2006. 'Web Services Dependability and Performance Monitoring', in *Proceedings of 21st Annual UK Performance Engineering Workshop, UKPEW 2005.* Newcastle Upon Tyne, UK.

38. Li, P., Chen, Y., and Romanovsky, A., 2006. 'Measuring the Dependability of Web Services for Use in e-Science Experiments', in *Service Availability*. Book series: *Lecture Note of Computing Science*. Springer: Berlin / Heidelberg. p. 193-205.

39. W3C. (2001). 'About the World Wide Web'. [Retrieved: 30 Jan 2008]; Available from: http://www.w3.org/WWW/

40. Tartanoglu, F., Issarny, V., and Romanovsky, A., 2003. 'Dependability in the Web Services Architecture in Architecting Dependable Systems'. In *Architecting Dependable Systems*. Book series: *Lecture Notes in Computer Science*. Springer: Berlin / Heidelberg. Vol. 2677: p. 90-109.

41. Stevens, R.D., Robinson, A.J., and Goble, C.A., 2003. 'myGrid: Personalised Bioinformatics on the Information Grid', in *Journal of Bioinformatics*. Vol. Supplement 1(19), No. 19: p. i302-i304.

42. Miyazaki, S. and Sugawara, H., 2000. 'Development of DDBJ-XML and its application to a database of cDNA', in *Journal of Genome Informatics*. Universal Academy Press, Inc (Tokyo). Issue 11: p. 380-381.

43. NTL Business Limited. (2008). 'ntl: Telewest business'. [Retrieved: 03 March 2008]; Available from: http://www.ntltelewestbusiness.co.uk/products__solutions/broadband__internet_services.aspx

44. CERNIC. (2008). 'China Education and Research Network (CERNET)'. [Retrieved: 03 March 2008]; Available from: http://www.edu.cn/HomePage/english/cernet/index.shtml

45. Alwagait, E. and Ghandeharizadeh, S., 2005. 'DeW: A Dependable Web Services Framework', in *Proceedings of the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)*. IEEE Computer Society Press, p. 111-118.

46. Salatge, N. and Fabre, J.-C., 2007. 'Fault Tolerance Connectors for Unreliable Web Services'. in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society Press. p. 51-60.

47. Tsai, W.T., Song, W., Paul, R., Cao, Z., and Huang, H., 2004. 'Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing'. in *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*. IEEE Computer Society Press, Vol. 01.

48. Laranjeiro, N. and Vieira, M., 2007. 'Towards fault tolerance in web services compositions', in *Proceedings of the 2007 workshop on Engineering fault tolerant systems*. Dubrovnik, Croatia: ACM.

49. Cristian, F., 1982. 'Exception Handling and Software Fault Tolerance', in *IEEE Transactions on Computers*. Vol. 31. Issue 6: p. 531-540.

50. Vieira, M., Laranjeiro, N., and Madeira, H., 2007. 'Assessing Robustness of Web-Services Infrastructures'. in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society Press.

51. Looker, N., Munro, M., and Xu, J., 2004. 'WS-FIT: A Tool for Dependability Analysis of Web Services'. in *Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - (COMPSAC'04)*. IEEE Computer Society Press, Vol. 02.

52. Randell, B., Romanovsky, A., Rubira, C.M.F., Stroud, R.J., Wu, Z., and Xu, J., 1995. 'From recovery blocks to concurrent atomic actions', in *Predictably Dependable Computing Systems*, H. Kopetz, J.C. Laprie, R. Brian, and B. Littlewood, (editors). Springer-Verlag New York, Inc. p. 87-101.

53. Randell, B. and Xu, J., 1994. 'The Evolution of the Recovery Block Concept', in *Software Fault Tolerance,* M. Lyu, (editor). J. Wiley. New York, p. 1-22.

54. Avizienis, A., 1985. 'The N-Version Approach to Fault-Tolerant Software', in *IEEE Transactions of Software Engineering.* IEEE Computer Society Press. Vol. 11, Issue 12: p. 1491-1501.

55. Knight, J.C. and Leveson, N.G., 1986. 'An experimental evaluation of the assumption of independence in multiversion programming', in *IEEE Transactions on Software Engineering.* IEEE Computer Society Press. Vol. 12, Issue 1: p. 96-109.

56. Eckhardt, D.E., Caglayan, A.K., Knight, J.C., Lee, L.D., McAllister, D.F., Vouk, M.A., and Kelly, J.J.P., 1991. 'An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability', in *IEEE Transactions on Software Engineering.* IEEE Computer Society Press. Vol.17, Issue 7: p. 692-702.

57. Salas, J., Perez-Sorrosal, F., Patiño-Martínez, M., and Jiménez-Peris, R., 2006. WS-replication: a framework for highly available web services, in *Proceedings of the 15th International Conference on World Wide Web* (Edinburgh, Scotland, May 23 - 26, 2006). WWW '06. ACM Press, New York, NY, 357-366.

58. Townend, P., Groth, P., and Xu, J., 2005. 'A Provenance-Aware Weighted Fault Tolerance Scheme for Service-Based Applications'. in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05).* IEEE Computer Society Press.

59. OASIS. (2008). 'OASIS Web Services Reliable Messaging (WSRM) TC'. [Retrieved: 30 Jan 2008]; Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm

60. AmberPoint Inc. (2003). Report: *Managing Exceptions in Web Services Environments.*

61. Dobson, G., 2005. 'A container-based mechanism for service fault tolerance'. [Retrieved: 30 March 2008]; Available from: http://www.dirc.org.uk/research/DIRC-Results/ServiceFaultTolerance.html

62. Ardissono, L., Furnari, R., Goy, A., Petrone, G., and Segnan, M., 2006. 'Fault Tolerant Web Service Orchestration by Means of Diagnosis', in *Proceedings of the third European Workshop on Software Architecture*. Series: Lecture Notes in Computing Science. Springer Berlin / Heidelberg. pp. 2-16.

63. Serugendo, G.D.M., Fitzgerald, J., Romanovsky, A., and Guelfi, N., 2007. 'A metadata-based architectural model for dynamically resilient systems'. in *Proceedings of the 2007 ACM symposium on Applied computing*. Seoul, Korea: ACM. p.566-572.

64. Goel, S., Talya, S.S., and Sobolewski, M., 2007. 'Service-based P2P overlay network for collaborative problem solving', in *Journal of Decision Support Systems*. Elsevier Science Publishers B. V. Vol. 42, Issue 2: p. 547-568.

65. Fitzgerald, J., Parastatidis, S., Romanovsky, A., and Watson, P., 2004. 'Dependability-explicit Computing in Service-oriented Architectures', in *Proceedings of the International Conference on Dependable Systems and Networks*. Florence, Italy. Vol. Supplement: p. 34-35.

66. Wiederhold, G., 1995. 'Mediation in information systems', in *Journal of ACM Computing Surveys*. ACM.Vol.27, Issue 7: p. 265-267.

67. Goldberg, J., Greenberg, I., Clark, R., Jensen, D., Kim, K., and Wells, D., (1994). 'Adaptive Fault-Resistant Systems'. in *SRI Technical Report*. SRI International. [cited 11 April 2008]; Available from: http://www.csl.sri.com/papers/sri-csl-95-02.

68. Fraga, J., Siqueira, F., and Favarim, F., 2003. 'An Adaptive Fault-Tolerant Component Model'. in *Proceedings of International Workshop on Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall.* 2003: p. 179-179.

69. Hecht, M., Hecht, H., and Shokri, E., 2000. 'Adaptive fault tolerance for spacecraft'. in *Proceedings of Aerospace Conference.* Big Sky, MT, USA: IEEE Computing Society press. Vol. 5: p. 521-533.

70. Avizienis, A. and Chen, L., 1977. 'On the Implementation of N-Version Programming for Software Fault Tolerance During Execution', in *Proceedings of IEEE Ann. Int'l Computer Software and Applications Conf. (COMPSAC 77).* Chicago, IL: IEEE Computer Society press. p. 149–155.

71. Sun Microsystems Inc. 'Web Services Overview'. [Retrieved: 30 Jan 2008]; Available from: http://java.sun.com/webservices/

72. Microsoft Corporation. '.NET Framework'. [Retrieved: 16 Jan 2008]; Available from: http://msdn2.microsoft.com/en-gb/netframework/default.aspx

73. Sun Microsystems Inc. 'Java EE at a Glance'. [Retrieved: 16 Jan 2008]; Available from: http://java.sun.com/javaee/index.jsp

74. NetBeans. 'Documentation, Training & Support'. [Retrieved: 16 Jan 2008]; Available from: http://www.netbeans.org/kb/

75. Glassfish Community. 'Project Description, Metro Project'. [Retrieved: 16 Jan 2008]; Available from: https://jax-ws.dev.java.net

76. W3C. (2006). 'Web Services Policy 1.2 - Framework (WS-Policy) '. [Retrieved: 30 Jan 2008]; Available from: http://www.w3.org/Submission/WS-Policy/

77. XMethods. 'Welcome to XMethods'. [Retrieved: 30 Jan 2008]; Available from: http://www.xmethods.net

78. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D.J., 1990. 'Basic local alignment search tool', *in Journal of Molecular Biology*. Issue 215: p. 403-410.

79. Virginia Bioinformatics Institute. (2007). 'Pathport, the pathogen portal project'. [Retrieved: 30 Jan 2008]; Available from: http://pathport.vbi.vt.edu/main/home.php

80. The Trustees of Princeton University. (2007) 'PLANETLAB'. [Retrieved: 30 Jan 2008]; Available from: https://www.planet-lab.org

81. Gorbenko, A., Mikhaylichenko, A., Kharchenko, V., Romanovsky, A. (2007). 'Experimenting With Exception Handling Mechanisms Of Web Services Implemented Using Different Development Kits', in CS-TR No 1010. School of Computing Science, Newcastle University,

82. OASIS. (2006). 'OASIS Web Services Security (WSS) TC'. [Retrieved: 19 April 2008]; Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss#announcements

83. IBM, Microsoft. (2002). 'Security in a Web Services World: A Proposed Architecture and Roadmap.' [Retrieved: 19 April 2008]; Available from: http://www.ibm.com/developerworks/library/specification/ws-secmap/

84. IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. (2007). 'Business Process Execution Language for Web Services version 1.1'. [Retrieved: 30 March 2008]; Available from: http://www.ibm.com/developerworks/library/specification/ws-bpel/

List of Abbreviations

AW:	Airway company
BLP:	Business logic processor
DA:	Dependability assessment mechanism
DMM:	Dependability monitoring mechanism
DS:	Database system
FTMS:	Fault-tolerance mechanisms
HT:	Hotel
PS:	Policy system
REDRM:	Resilience-explicit dynamic reconfiguration mechanism
SMD:	Sub-Mediator database
SMI:	Sub-Mediator Interface
SMM:	Sub-Mediator monitoring mechanism
SOA:	Service-oriented architecture
TA:	Travel agency
WS:	Web Service
WSD:	Web Services database
WSIM:	Web Service invocation mechanism
WSM:	Web Services monitoring mechanism

Appendix A – The WSsDAT tool

Our work on the tool started with formulating the essential requirements which a general Web Services dependability-monitoring tool needs to meet. The main requirement is that such a tool should be able to monitor a Web Service continuously for a preconfigured period of time and record various types of information in order for the dependability of a service to be measured. Firstly, the tool should provide an interface to accept user's inputs and map these user inputs into internal processing actions. Secondly, the tool has to be able to invoke the Web Service effectively and wait for results; internal and external exceptions should be monitored during this period. When the output of the service invocation is received, the response time for the service should be recorded and analyzed. Ideally, the output of the service needs to be assessed to determine whether the Web Service functioned properly and whether it passed or failed according to the users' demands. Moreover, when the test invocation failed then any fault messages generated by the service should also be documented. If available, these messages will provide insights behind the problems causing the service failure. Finally, the tool should be able to produce reports of the test and monitoring procedures.

Overview

The requirements of a general Web Services dependability-monitoring tool were realised by the development of a Java-based application called Web Services Dependability Assessment Tool (WSsDAT) which is aimed at evaluating the dependability of Web Services. The tool supports various methods of dependability testing by acting as a client invoking the Web Services under investigation. The tool

enables users to monitor Web Services by collecting the following reliability characteristics:

- ***Availability and Functionality:*** Calls are made to a Web Service at defined intervals to check if the Web Service is functioning. The tool is able to test the semantics of the response which are generated by the Web Service being monitored. It is possible to pre-configure the tool using a regular expression which represents the correct response expected by the scientist from a given Web Service and ensure the service is functioning according to that expected by its user. Results returned from a Web Service are recorded for further analysis which can be manually carried out by a user.
- ***Performance:*** The WSsDAT measures the round-trip response time of calls made to the Web Services. Average response time of successful calls is used as performance metric of a Web Service.
- ***Faults and exceptions:*** The tool records any faults generated by a failed invocation of a Web Service. Internal and external exceptions, for example, networking timeout exceptions are also recorded for further analysis.

Further to the above metadata recorded by WSsDAT, the tool can also be used to test and monitor the dependability of Web Services at geographically disparate locations through the deployment of the tool on different computers. It is important to understand the behaviour of a Web Service from the point of view of the clients, in order to comprehend the networking consequences between the clients and the Web Service.

General principles and architecture

One of the problems with using public scientific Web Services is that their interfaces differ from one resource to another. Therefore, testers would normally have to write a customized invocation script for each service because of the different interfaces and parameters required. The WSsDAT is an off-the-shelf tool offering general solutions for monitoring the dependability of Web Services. This tool is implemented using Apache Axis JAX-RPC style SOAP processing APIs.

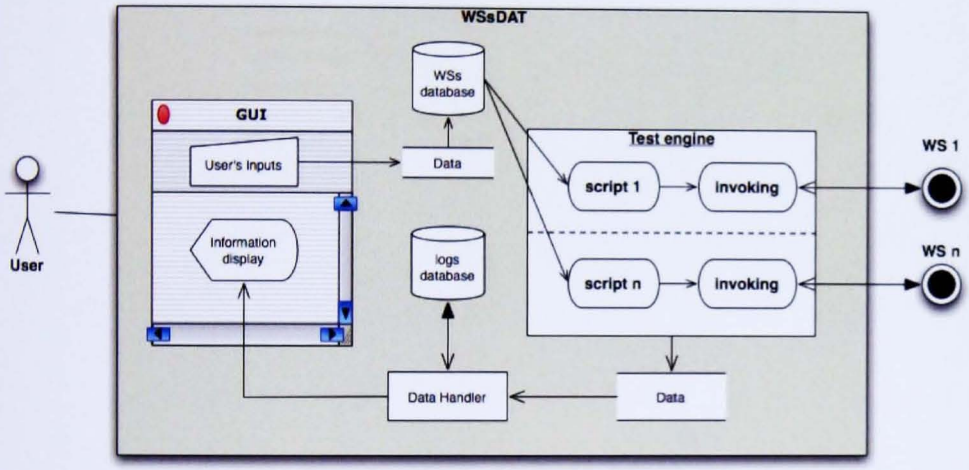

Figure A-1: The architecture of the WSsDAT

The architecture of WSsDAT is shown in Figure A-1. It consists of three main functional components, a graphical user interface (GUI), a Test Engine and a Data Handler. The GUI captures the user's request, and configures the test policy and system settings. These inputs are modeled, mapped and stored in a database for repeated use. The GUI is also a viewport which renders live dependability and performance metrics of the Web Services being monitored. The Test Engine is responsible for generating and executing invocation scripts using the modeled data stored in the Web Services database to invoke Web Services. The Test Engine is able

to run a batch of tests and measurements concurrently. The Data Handler processes and models all test and observation measurements data. After statistical analysis, these data are subsequently stored in a MySQL database or as plain text files; relevant information is passed and rendered in the viewport on the GUI.

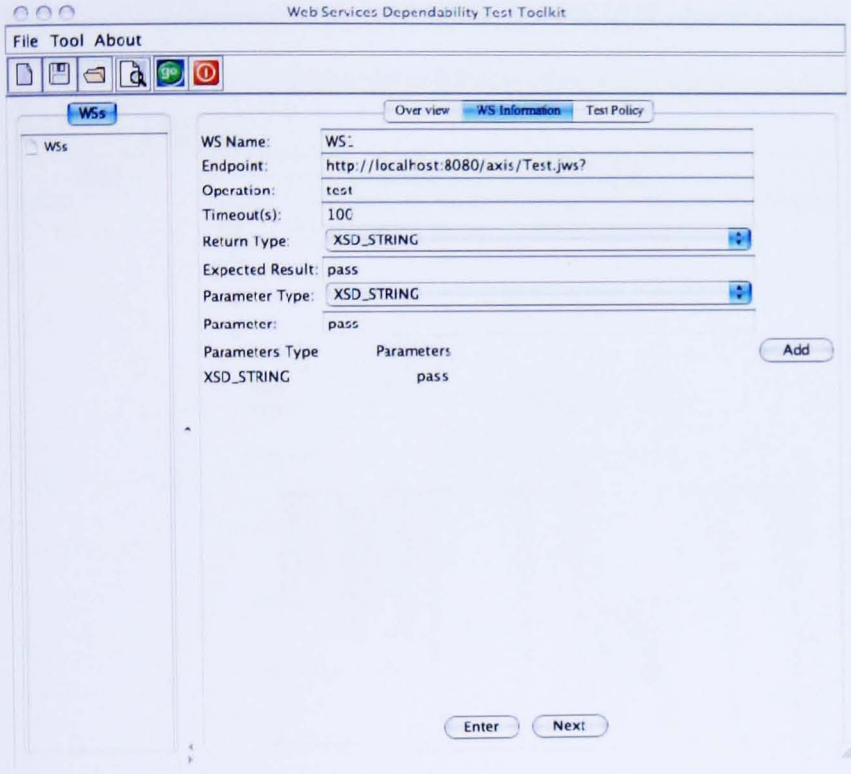

Figure A-2: GUI for Web Services information inputs

Graphical user interface (GUI)

We designed and implemented the GUI by which users can interact with the WSsDAT. Users can input information of Web Services on the GUI, set test parameters and configure test policies, as shown in Figure A-2. The WSsDAT is capable of testing multiple Web Services simultaneously. Each time the GUI accepts inputs for one Web Service. Once user's inputs are validated, these data are modeled and saved in a database, and the Web Service is entered into a test array. The Web

Services in the test array are listed on the GUI and can be selected individually for modification and information display. The viewport on the GUI renders information of Web Services, such as errors, average response time, and graphs of response times. The user can highlight a Web Service in the testing list for display. (See Figure A-3).

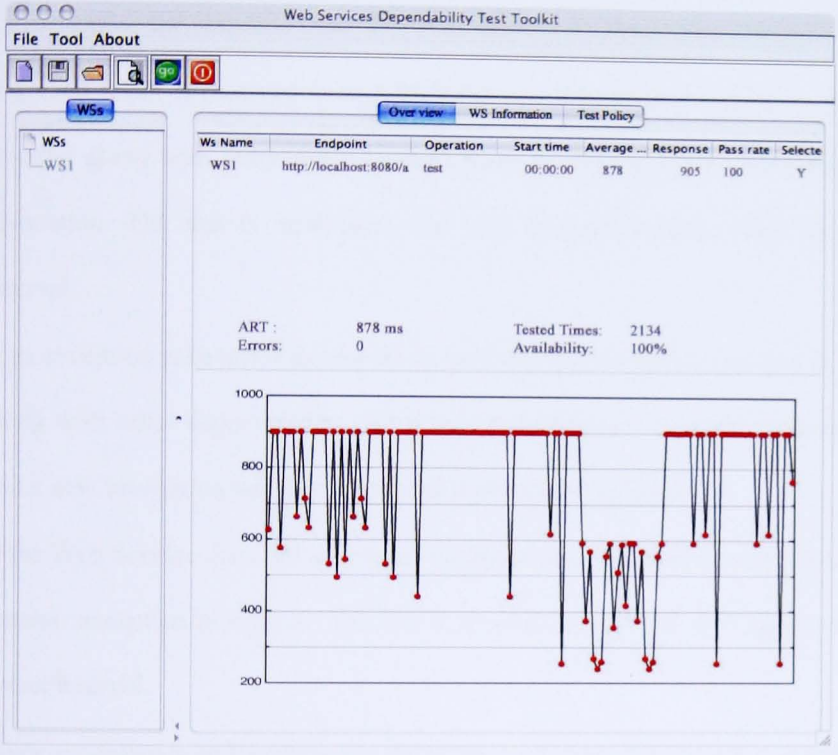

Figure A-3: GUI for test information display

Test engine

The Test Engine processes the user's inputs and implements service invocation scripts according to test policies. Tests on each Web Service are established as a single thread and all tests are carried out in parallel. The number of test threads is only restricted by the computer system's capability or restriction. Figure A-4 is an UML

diagram showing how the Test Engine cooperates with other components in the WSsDAT. The mechanism of a test procedure described briefly as following:

- The Test Engine assembles an invocation script for a Web Service to be monitored according to user's inputs.
- The Test Engine invokes the Web Service with the test script. A timer is started for measuring the response time. The start time of the invocation is logged.
- If a valid result is received from a Web Service, the result is passed to the Data Handler along with other measurements such as start time and end time of the invocation. The test is terminated and will be started again after the preset interval.
- If an exception is detected during the invocation, the exception message is logged along with other dependability and performance metrics. The test is terminated and a new invocation will be initiated after the preset interval.
- If the Web Service does not return any response after a preset timeout period, the timeout exception is logged. The test is terminated and will start again after the preset interval.
- Relevant statistics and analysis are processed and logged after each invocation.

The Test Engine implements the SOAP message processing mechanism. It is able to analyze the SOAP message received from the Web Services by reporting the error message attached in the SOAP message and thereby allowing users of the tool to understand what failures occurred during an unsuccessful invocation.

Data handler

The Data Handler processes all data generated during the test. After statistical analysis, these data are stored in a MySQL database, and passed to the GUI if appropriate. If a MySQL database is not installed on the computer, the WSsDAT has an option to save these data in formatted text files. The contents of these files are commented and split clearly and can be easily converted into Microsoft Excel or some other statistics software which can import data from formatted text files such as SPSS[4].

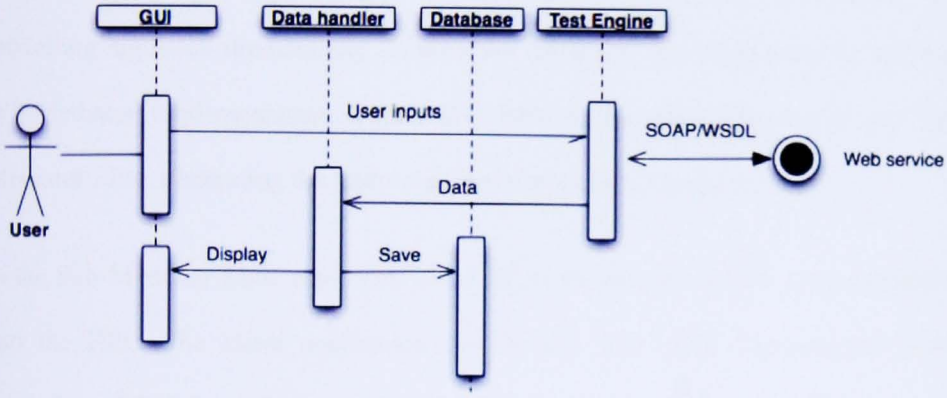

Figure A-4: Test procedure

[4] http://www.spss.com/SPSS/

Appendix B – Implementation of Java Sub-Mediator Elite

We started implementing the Java WS-Mediator by using the UML modelling tool [74] integrated in NetBeans to generate abstract classes of components. The modelling technique allowed us to construct an abstract prototype of the WS-Mediator and its components from scratch by defining attributes and operations to present the functionalities and behaviours of components. Moreover, we were able to validate the proposed system structure and components with Use Case and class diagrams along with the modelling-based system validation techniques. The modelling approach dramatically reduced the difficulty and complexity of the Java WS-Mediator implementation. Figure B-1 presents the class diagram of the Sub-Mediator Elite, illustrating the internal components of the implementation.

In the Sub-Mediator Elite, class *Med_Elite_SOAPPort()* acts as both service interface and the BPL. The client application can invoke Java APIs implemented in the *Med_Elite_SOAPPort()* class to request different services. This class interprets the client's requests and assigns jobs to the corresponding components. Figure B-2 illustrates the dependency of the *Med_Elite_SOAPPorts()* class. The *WS_Bridge()* and the *SubMed_Brisge()* classes are the components for accessing the *Web Service database* and the *Sub-Mediator database*. The *Dynamic_Reconf_Engine()* class implements the *Dynamic Reconfiguration Engine* of the Sub-Mediator to process the mediating service requests. The *Med_Elite_PolicyPort()* class interprets the *global execution policy*, while the *WS_ReqPolicy_Parser()* class extracts *individual execution policies*.

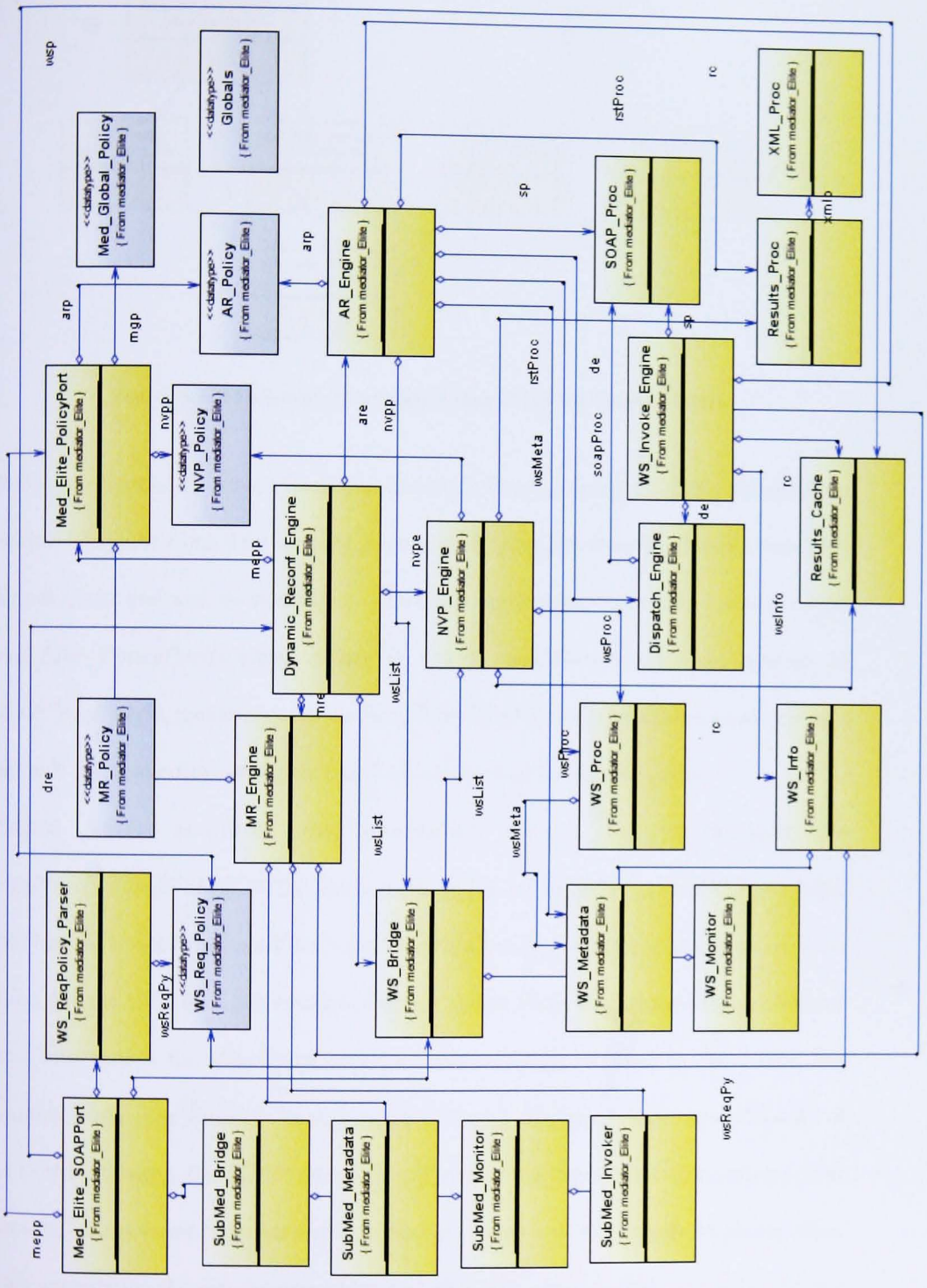

Figure B-1: Class diagram of the Sub-Mediator Elite

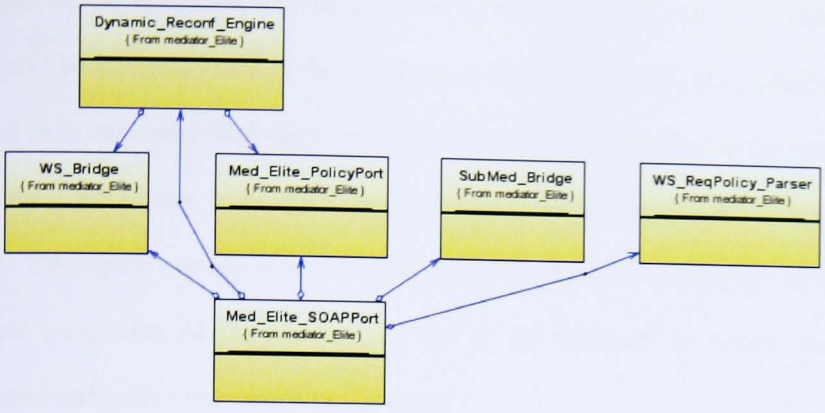

Figure B-2: The Service Processing Engine of the WS-Mediator Elite

Below we discuss a simple client application developed using the APIs provided by the Sub-Mediator Elite. The client requests a mediating service and provides two Web Services, *ws1* and *ws2,* as candidates. The client application creates an instance of the *Med_Elite_PolicyPort()* class, names it *mesp,* and then creates an instance of *SOAPProc()* class, and names it *soapProc*. The *SOAPProc()* class implements various methods for converting *String and XML document* into SOAP messages.

Method *ws1()* assembles the information about *ws1*. It invokes the *soapProc.bindingSOAP()* method to convert *String smRequest* into a SOAP message, and then uses *soapProc.readFileCreatDocument()* to generate an *individual execution policy* from a *XML* file. The variable *faults* is a Java *HashMap* containing customized error information for identifying specific error messages defined by the client. For instance, *faults.put("Result", "busy")* means if "busy" appeared in *Element* "Result" of the SOAP message, this SOAP message will be regarded as invalid and carrying error message. *mesp.insert ()* passes the information about *ws1* to the Sub-Mediator Elite. After capsulating the information about *ws1* and *ws2*, *mesp.setGlobalPolicy()* sets the *global execution policy* for this mediating service request. *mesp.execute()* starts the Sub-Mediator Elite to execute a service request.

The result of the execution will be returned as a *Java Vector*. The first element of *Vector* will be the final result in the response to a service request. If no valid result is obtained from candidate Web Services, an error message is returned as the result. The last element of *Vector* is an *XML processing report* explaining its structure and content. The *report* can be interpreted by a XML processing program to achieve automatic processing of the results. The rest of the elements in *Vector* stores the results returned from candidate Web Services.

```
import com.mediator.mediator_Elite.Med_Elite_SOAPPort;
import com.mediator.mediator_Elite.SOAP_Proc;
public class TestCase {
    private Med_Elite_SOAPPort mesp;
    private SOAP_Proc soapProc = new SOAP_Proc();
        ...
     public static void main(String[ args) {
       mesp = new Med_Elite_SOAPPort();
         ws1();
         ws2();
         globalPolicy= soapProc.readFileCreateDocument("C:\\ globalPolicy.xml");
         mesp.setGlobalPolicy(globalPolicy);
         Vector results = mesp.execute();
     }
     private void ws1(){
       QName serviceQName = new QName("http://xml.nig.ac.jp:80/xddbj/Blast", "Blast");
       QName portQName = new QName("http://tempuri.org/Blast", "Blast");
       SOAPMessage soapMessage = soapProc.bindingSOAP( (String) smRequest);
       xmlPolicy = soapProc.readFileCreateDocument("C:\\ws1_Policy.xml");
       HashMap faults = new HashMap();
       faults.put("Result", "busy");
       mesp.insert (serviceQName, portQName, soapMessage, xmlPolicy, faults);
    }
     private void ws2(){
        ...
     }
}
```

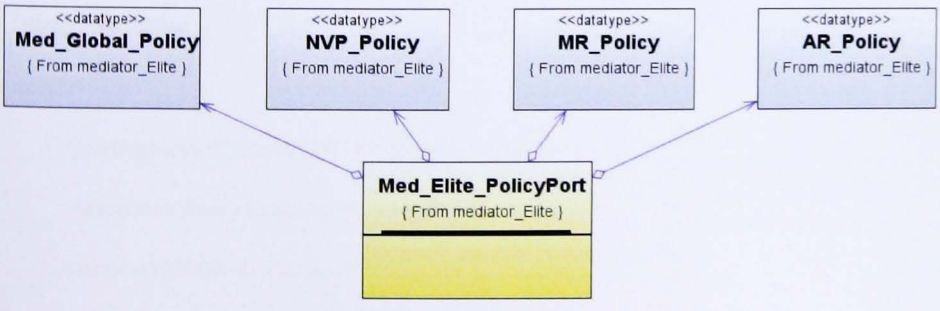

Figure B-3: Interpreting the *global execution policy*

Figure B-3 shows different types of execution policies extracted by the *Med_Elite_PolicyPort()* class. As explained in chapter 3, the *global execution policy* may change according to the execution mode. *NVP_Policy, MR_Policy* and *AR_Policy* present execution policies associated with the N-version programming, the Multi-Routing and the Service Alternative Redundancy execution modes respectively.

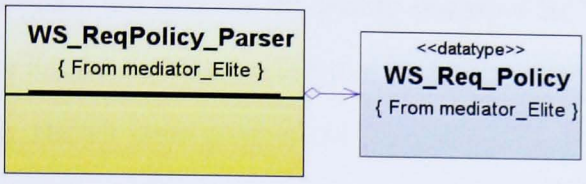

Figure B-4: The *individual execution policy*

As illustrated in Figure B-4, the *WS_ReqPolicy_Parser()* class extracts *individual execution policies* from the service request SOAP message. An individual execution policy is associated with each candidate Web Service. The Web Service Execution Engine uses individual policies to decide how to invoke each of them.

Below is an example of an *individual execution policy*, followed with the explanation of the entities.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsp:Policy xmlns:wsp = http://schemas.xmlsoap.org/ws/2004/09/policy
    xmlns:wsmip = "http://schemas.wsmediator.org/indevidualPolicy/policy">
     <wsp:ExactlyOne>
<wsp:All>
          <bindingMethod>SOAP11HTTP</bindingMethod>
          <invocationMode>Sync</invocationMode>
          <timeout>20000ms</timeout>
          <autotimeout>maximum</autotimeout>
```

```
        <retryAfterFailure>3</retryAfterFailure>
        <retryInterval>3000ms</retryInterval>
</wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

- *<bindingMethod>*: this indicates the binding method of the SOAP message. Web Service invocation APIs should follow the binding method to invoke the Web Service. Default value: SOAP11HTTP
- *<invoactionMode>*: this entity indicates the invocation method to the Web Service. There are three types of invocation methods: synchronous, asynchronous invocation and the conventional RPC (Remote Procedure Call) invocation. Default value: Sync (Synchronous invocation)
- *<timeout>*: this sets the timeout parameter for an invocation. If it does not complete in the timeout period, the invocation will be terminated and a timeout exception will be raised. The value of the timeout parameter can be automatically set by the Sub-Mediator if the value is set as 0ms.
- *<autotimeout>*: the Sub-Mediator can automatically set the timeout parameter for invoking a particular Web Service according to dependability metadata. There are three options: average, minimum and maximum, representing *average*, ***minimum*** and ***maximum response time***.
- *<retryAfterFailure>*: the Sub-Mediator implements the retry strategy to tolerate temporary *service* and *network failures*. This entity sets the number of retry invocations of a particular Web Service before giving up.
- *<retryInterval>*: this entity sets the interval between retries.

Class *Dynanic_Reconf_Engine()* implements the *Dynamic Reconfiguration Engine* of the Sub-Mediator Elite. Figure B-5 illustrates the dependent components of the *Dynamic Reconfiguration Engine*. The *WS_Bridge()* class implements methods to allow access to the Web Service database. Currently, there are three fault tolerance execution modes implemented in the Sub-Mediator Elite. *AR_Engine(),NVP_Engine()* and *MR_Engine()* implement the Service *Alternative Redundancy*, the *N-version Programming* (*Service Diversity*) and the *Multi-routing* execution mode.

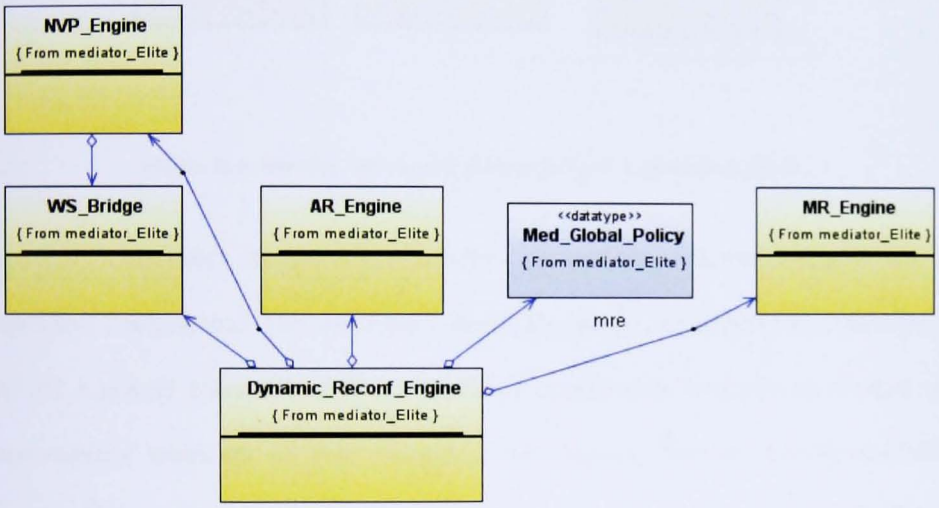

Figure B-5: The Dynamic Reconfiguration Engine of the Sub-Mediator Elite

The modelled system design and implementation of the Sub-Mediator Elite allow scalable and flexible adaptation of fault tolerance mechanisms by implementing them as individual fault tolerance execution models.

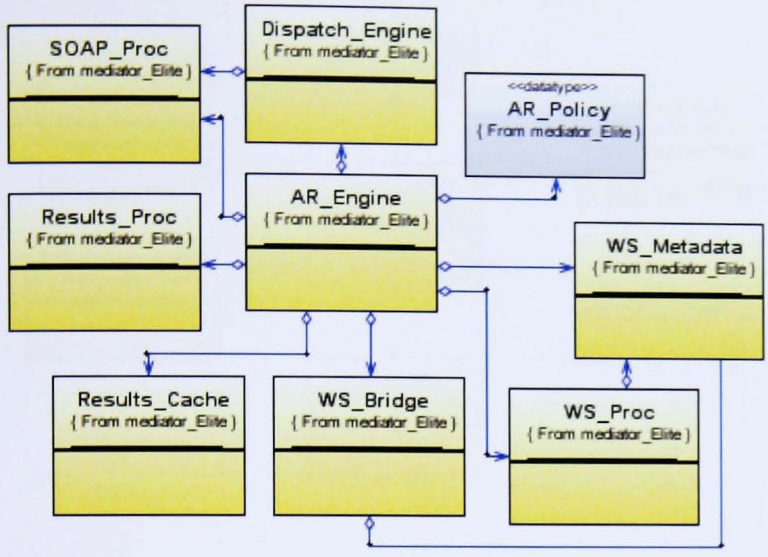

Figure B-6: Service Alternative Redundancy F-T execution mode

Figure B-6 illustrates the *Service Alternative Redundancy* execution engine and its dependent components. The *AR_Policy* execution policy constrains the execution of the *AR_Engine()* class. Class *WS_Metadata()* implements methods to retrieve the dependability metadata of Web Services. *AR_Engine()* checks the dependability metadata of candidate Web Services, and then sorts them according to *AR_Policy*. Class *WS_Proc()* implements methods for processing Web Services, such as sorting. The *SOAP_Proc()* class helps *AR_Engine()* to collect the necessary information for invoking Web Services. *Dispatch_Engine()* implements *Dispatch<T>* invocation API for invoking Web Services. When *AR_Engine()* receives a result via *Dispatch_Engine()*, it caches the result using the *Results_Cache()* class. If this result fails the validity check, the *AR_Engine()* class will retry the Web Service or switch to an alternative Web Service. If a valid result is received or all Web Services have been tried, the *AR_Engine()* finalizes *Result Cache* and generates final results using the *Results_Proc()* component.

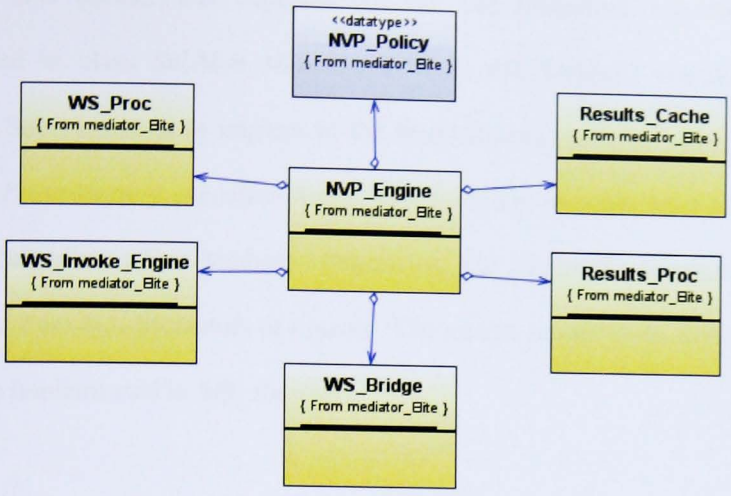

Figure B-7: N-Version Programming execution mode

Figure B-7 illustrates the *N-Version Programming execution engine* and its dependent components. It processes candidate Web Services according to *NVP_Policy*. Then it invokes the defined number of Web Services synchronously. All of the results returned from Web Services will be cached in *Results_Cache()*. The *NVP_Engine()* also performs the validity check. If a valid result is received, it is an option for the *NVP_Engine()* to terminate invocations and deliver the valid result as the first received result to the client. If a number of valid results are expected, the *NVP_Engine()* will wait until enough results have been received. If a Web Service fails an invocation before the expected number of valid results has been received, the *NVP_Engine()* will invoke alternative Web Services to continue execution. Valid results can be voted by the voting mechanism implemented in *NVP_Engine()*; however, it is an optional procedure.

Figure B-8 illustrates the *Multi-Routing* execution engine and its dependent components. The *MR_Engine()* interprets the *MR_Policy* to define the execution

procedure and checks the dependability of Sub-Mediators via the methods implemented in class *SubMed_Metadate()*. Then *MR_Engine()* selects a defined number of Sub-Mediators to implement the Multi-Routing Strategy. Similarly to the *N-Version Programming* execution mode, execution can be terminated when a valid result is received via a Sub-Mediator. Otherwise, *MR_Engine()* waits until all results are returned from Sub-Mediators or timeout. The results can be voted using the voting mechanism implemented in *MR_Engine()*.

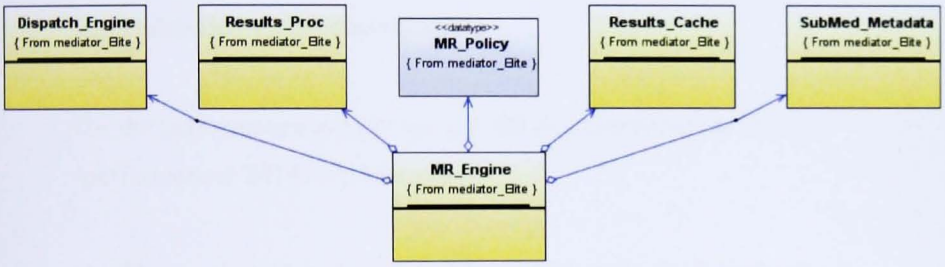

Figure B-8: The Multi-Routing Execution mode

The *Dispatch_Engine()* class implements dynamic Web Service invocation mechanisms. It utilizes the powerful *Dispatch<T>* dynamic Web Service invocation API provided by the JAX-WS 2.1 framework to achieve run-time dynamic integration of Web Services. The *Dispatch<T>* API supports synchronous, asynchronous and one-way invocation to suit different application scenarios. The Sub-Mediator Elite fully supports various invocation methods. An invocation method can be selected by an *individual execution policy*.

Appendix C – Dependability metadata

Below is given an example of dependability metadata implemented in the XML format. Element <ws> indicates the name of the Web Service using its endpoint. The nested elements represent various dependability attributes.

```
<?xml version="1.0"?>

<!-- Endpoint of the Web Service -->
<ws service="{http://xml.nig.ac.jp:80/xddbj/Blast}Blast">
        <!-- dependability rank of the Web Service -->
        <dependability>85%</dependability>

        <!-- the performance evaluation, e.g. the average response time -->
        <performance>24141</performance>

        <!-- The number of monitoring tests applied on the Web Services -->
        <numOfTests>340</numOfTests>

        <!-- The number of monitoring tests that returned valid results -->
        <succTests>290</succTests>

        <!-- the average response time of the valid invocations -->
        <aveResponseTime>24141ms</aveResponseTime>

        <!-- the minimum response time of the valid invocation -->
        <minimumResponseTime>1110ms</minimumResponseTime>

        <!-- the maximum response time of the invocations -->
        <maximumResponseTime>2750ms</maximumResponseTime>
</ws>
```

Appendix D – Dependability metadata database in XML

During dependability monitoring of Web Services, a time series of dependability metadata are kept in the dependability database. The changing dependability behaviour of Web Services can be understood by tracing their dependability metadata at different times, which helps the resilience-explicit decision-making mechanism to select the most desirable component services. Below is shown a fraction of the time-logged dependability metadata collected from one of our experiments.

```
<?xml version="1.0" encoding="UTF-8"?>
<report>
    <Execution startTime="Wed Mar 14 12:38:58 GMT 2007">
        <wslist>
            <ws
            service="{http://www.ebi.ac.uk/collab/mygrid service4 soap/se
            rvices/alignment::blastn_ncbi}AnalysisWSAppLabImplService
            ">
                <dependability>58</dependability>
                <performance>62500</performance>
                <numOfTests>340</numOfTests>
                <succTests>200</succTests>
                <aveResponseTime>62500</aveResponseTime>
                <minimumResponseTime>9999</minimumResponseTi
                me>
                <maximumResponseTime>61485</maximumResponse
                Time>
            </ws>
            <ws service="{http://xml.nig.ac.jp:80/xddbj/Blast}Blast">
                <dependability>85</dependability>
                <performance>24141</performance>
                <numOfTests>340</numOfTests>
```

```
                <succTests>290</succTests>
                <aveResponseTime>24141</aveResponseTime>
                <minimunResponseTime>1110</minimunResponseTi
                me>
                <maximumResponseTime>2750</maximumResponseT
                ime>
            </ws>
            <ws
            service="{http://pathport.bioinformatics.vt.edu:6565/axis/servi
            ces/blastbt}BlastbtService">
                <dependability>91</dependability>
                <performance>28990</performance>
                <numOfTests>340</numOfTests>
                <succTests>310</succTests>
                <aveResponseTime>28990</aveResponseTime>
                <minimunResponseTime>9999</minimunResponseTi
                me>
                <maximumResponseTime>36297</maximumResponse
                Time>
            </ws>
        </wslist>
</Execution>
<Execution startTime="Wed Mar 14 12:44:28 GMT 2007">
        <wslist>
            <ws
            service="{http://www.ebi.ac.uk/collab/mygrid/service4/soap/se
            rvices/alignment::blastn_ncbi}AnalysisWSAppLabImplService
            ">
                <dependability>58</dependability>
                <performance>62500</performance>
                <numOfTests>341</numOfTests>
                <succTests>200</succTests>
                <aveResponseTime>62500</aveResponseTime>
```

```
                <minimunResponseTime>9999</minimunResponseTi
                me>
                <maximumResponseTime>61485</maximumResponse
                Time>
            </ws>
            <ws service="{http://xml.nig.ac.jp:80/xddbj/Blast}Blast">
                <dependability>85</dependability>
                <performance>24141</performance>
                <numOfTests>341</numOfTests>
                <succTests>290</succTests>
                <aveResponseTime>24141</aveResponseTime>
                <minimunResponseTime>1110</minimunResponseTi
                me>
                <maximumResponseTime>2750</maximumResponseT
                ime>
            </ws>
            <ws
            service="{http://pathport.bioinformatics.vt.edu:6565/axis/servi
            ces/blastbt}BlastbtService">
                <dependability>91</dependability>
                <performance>28983</performance>
                <numOfTests>341</numOfTests>
                <succTests>311</succTests>
                <aveResponseTime>28983</aveResponseTime>
                <minimunResponseTime>9999</minimunResponseTi
                me>
                <maximumResponseTime>36297</maximumResponse
                Time>
            </ws>
        </wslist>
</Execution>
<Execution startTime="Wed Mar 14 12:49:58 GMT 2007">
        <wslist>
```

```
<ws
service="{http://www.ebi.ac.uk/collab/mygrid/service4/soap/se
rvices/alignment::blastn_ncbi}AnalysisWSAppLabImplService
">
        <dependability>58</dependability>
        <performance>62500</performance>
        <numOfTests>342</numOfTests>
        <succTests>200</succTests>
        <aveResponseTime>62500</aveResponseTime>
        <minimumResponseTime>9999</minimumResponseTi
        me>
        <maximumResponseTime>61485</maximumResponse
        Time>
</ws>
<ws service="{http://xml.nig.ac.jp:80/xddbj/Blast}Blast">
        <dependability>84</dependability>
        <performance>24141</performance>
        <numOfTests>342</numOfTests>
        <succTests>290</succTests>
        <aveResponseTime>24141</aveResponseTime>
        <minimumResponseTime>1110</minimumResponseTi
        me>
        <maximumResponseTime>2750</maximumResponseT
        ime>
</ws>
<ws
service="{http://pathport.bioinformatics.vt.edu:6565/axis/servi
ces/blastbt}BlastbtService">
        <dependability>91</dependability>
        <performance>28977</performance>
        <numOfTests>342</numOfTests>
        <succTests>312</succTests>
        <aveResponseTime>28977</aveResponseTime>
```

```
                    <minimumResponseTime>9999</minimumResponseTi
                    me>
                    <maximumResponseTime>36297</maximumResponse
                    Time>
                </ws>
            </wslist>
        </Execution>
</report>
```

Appendix E – Implementation of Java client application

The Java code shown below is an example of the Java client application based upon the Sub-Mediator Elite that uses three Blast Web Services as component services to implement service diversity strategy by using the N-version programming fault tolerance execution mode. We use comments in the code to explain how to implement a Java client application with the APIs provided by the Sub-Mediator Elite.

```
/*
 * TestCases.java
 *
 * Created on 21 February 2007, 17:43
 *
*/

package com.mediator.test;

/* The Java application needs to import the necessary classes. Med_Elite_SOAPPort
is the interface of the Sub-Mediator Elite. SOAP_Proc and XML_Proc provide
optional methods for processing SOAP messages and XML files. * /

import com.mediator.mediator_Elite.Med_Elite_SOAPPort;
import com.mediator.mediator_Elite.SOAP_Proc;
import com.mediator.mediator_Elite.XML_Proc;

import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.Date;
import java.util.Vector;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import org.omg.CORBA.DATA_CONVERSION;
```

```
import org.w3c.dom.Document;

/**
 * @Yuhui Chen
 */

public class TestCases {

   /* Creates the instance of the classes implemented in Sub-Mediator Elite. */
   private Med_Elite_SOAPPort mesp;
   private SOAP_Proc soapProc = new SOAP_Proc();
   private XML_Proc xmlp = new XML_Proc();

  /* Vector results is created for accepting the processing results returned from Sub-
     Mediator Elite.*/
   private Vector results;

   public TestCases() {
   }

   /* The main method that implements the business logic */
   public static void main(String[ args) {

     /* Creates a new instance of TestCases */
      TestCases tcs = new TestCases();

     /* Creates an instance of Log_Proc for logging the execution of the business
        procedures */
      Log_Proc logproc = new Log_Proc();

      /* Initiates the logging buffer */
      logproc.init();

     /* Executes the business process */
```

```
    tcs.execute(logproc);

  /* Prints the execution results returned from the Sub-Mediator Elite *
    tcs.printResult();
}

  /* Assembling invocation to the Sub-Mediator Elite */
  private long execute(Log_Proc logproc){

   /* logs start time */
   long t1 = System.currentTimeMillis();
    mesp = null;

   /* Initiates the interface of the Sub-Mediator Elite */
    mesp = new Med_Elite_SOAPPort();

   /* Initiates the vector accepting the execution results*/
    results = new Vector();

   /* Assembling invocations to the candidate Web Services */
    ws1();
    ws2();
    ws3();

   /* Imports the global execution policy*/
   Document globalPolicy = null;
   try {
      globalPolicy =
      xmlp.readFileCreateDocument("E:\\Projects\\Mediator\doc\Current\ globalP
      olicy.xml");
   } catch (Exception ex) {
      ex.printStackTrace();
   }
```

```
    /* Sets the global execution policy*/
    mesp.setGlobalPolicy(globalPolicy);

    Date startTime = new Date();

    /* Invokes the Sub-Mediator to execute the dynamic service composition */
    results = mesp.execute();

    /* Calculates the response time externally in the client application*/
   long t2 = System.currentTimeMillis();
    long responseTime = t2-t1;

   /* Logs the relevant results */
    logproc.append((Document)results.lastElement(), startTime,
       String.valueOf(responseTime));
    logproc.writeLog("E:\\Projects\\Mediator\\doc\\output\\log.xml");

    System.out.println("************************");
    System.out.println("* Response Time (ms) : " + responseTime);
    System.out.println("************************");
    return responseTime;
}

 /* Assembling the invocation to a candidate Web Service */
private void ws1(){

    /* The Japanese DDBJ Blast Web Service */
    QName serviceQName = new QName("http://xml.nig.ac.jp:80 xddbj Blast",
       "Blast");
    QName portQName = new QName("http://tempuri.org Blast", "Blast");

    /* String smRequest is the invocation SOAP message to DDBJ */
    String smRequest = "<soapenv:Envelope
       xmlns:soapenv=\"http://schemas.xmlsoap.org/soap envelope \"><soapenv:Bod
```

y><searchSimple
xmlns=\"http://xml.nig.ac.jp:80/xddbj/Blast\"><program>blastn</program><d
atabase>ddbjhum</database><arg2>ccccacatca ccactttgga taacgccaaa
tacaccttca acgggctagg atacttcctg ctggttcagg cccaggacag aaattcttcc ttcctgctgg
agggccgcac tgcccagact gattctgcca atgccacgaa cttcattgcc tttgcggccc aatacaacac
cagcagcctg aagtctccca tcacagttca gtggtttctt gagcccaatg acacaatccg agttgtacac
aataaccaaa cggtggcctt taacaccagc gacactgaag acttgcccgt attcaatgcc
actggtgtcc tactgatcca aaatggctcc caagtctcag ccaactttga tgggacagtg
accatctctg tgattgctct ctccaacatc cttcacgcct cctccagcct gtcagaggag
taccgcaacc acacaaaggg ccttctggga gtctggaatg acaatccaga agatgacttc
agaatgccca atggctccac catcccctcc aacacgtccg aggagactct tttccactat
ggaatgacat cggaaactaa cgggataggc ctccttgggg tgaggacaga ccctctgcct
tctgagttta ctcccatctt cttgtcccaa ctgtggaaca agagcggcgc cggtgaagac
ttgatctctg ggtgcaacga ggacgcacag tgcaagtttg acatcctggc cacaggaaac
agagacatcg gacaaagcac caactcaatc cttagaacat tccggcacgt gaatggcacg
ctcaaccagt acccaccccc tatccactac agcagcaaga ttcaagccta caaggggcga
gaacagtggc cattgagatc accagcaact ctaaggatgt cgtattcagc ctctccaaca
agtgcagtgg cctttgagct ctttgaaaac gggagtttgc acgtggacac caacatcccc
agaagaacgt acctggagat tctagcaagg gatgtcaaga ctaacttgtc atcggtactc
cagcctgaga cggtggcttg cttctgtagt aaggaggaac agtgtttgta caacgagacc
agcaaagagg gcaactcttc cactgaggtg accagctgca agtgcgatgg gaactccttc
ggccgcttgt gtgaacactc taaggacctc tgcactgagc catgcttccc taatgtggac
tgcattcctg ggaagggctg tcaggcctgc cctccaaaca tgactggaga tgggcgtcat
tgtgtagctg tggagatctc tgaattctgc cagaaccatt cctgtcctgt gaattactgc tataaccatg
gccattgcga catctctggg cctccagact gccagcccac ttgcacctgc gcccctgcct
tcactggtaa ccgctgcttc ctggccggga acaatttcac tcccatcatc tataaagagc ttcccttgag
gaccatcacg ctctctctca gggaggacga aaacgcctct aacgctgacg tcaatgcctc
ggtggcaaac gtactagaga acttggacat gcgggctttt ctctccaaca gcttagtgga
gctgatacga acctctcccg gagcaccagt ccttggcaag cccattcatc actggaaggt
cgtctcccac ttcaagtacc gtcccagggg acccctcatc cactatctga acaaccaact
gataagcgcc gtgatggagg ccttcctcct ccaggctcgg caggagaggc ggaagaggag
tggagaagcc aggaagaacg tccgcttctt ccccatctcg agggcagacg tccaggacgg
gatggccctg aacctaagta tgctggacga gtacttcacg tgcgatggct acaaaggcta
ccacttggtc tacagccccc aggatggcgt cacctgtgtg tcccccatgta gtgagggcta

```
ctgtcacaat ggaggccaat gcaagcacct gccagatggg  ccccagtgca cgtgcgcaac
cttcagcatc tacacatcct ggggcgaacg ctgtgagcat  ctaagcgtga aacttggggc
attcttcggg atcctctttg gagccctggg tgccctcttg  ctactggcca tcttagcatg tgtggtcttt
cacttctgcg gctgctccat gaacaagttc tcctaccctc tggactcaga
actgtga</arg2></searchSimple></soapenv:Body></soapenv:Envelope>";

/* String xmlPolicy contains the individual execution service policy */
String xmlPolicy= "<wsp:Policy
xmlns:wsp=\"http://schemas.xmlsoap.org/ws/2004/09/policy\"
xmlns:wsmip=\"http://schemas.wsmediator.org/indevidualPolicy/policy\"><w
sp:ExactlyOne><wsp:All><!-- Binging method --
><bindingMethod>SOAP11HTTP</bindingMethod><!-- Invocation mode:
RPC | Sync | Async --><invocationMode>Sync</invocationMode><!-- time
out parameter --><timeout>20000</timeout><!-- auto-set time out parameter:
average | max  --><autotimeout>average</autotimeout> <!-- How many time
to retry after failure--><retryAfterFailure>3</retryAfterFailure><!-- Interval
between retries --><retryInterval>30</retryInterval><!-- apply multi-routing,
and number of routes --><multirouting>0</multirouting><!-- start to monitor
this Web Service locally? no | locally | remotely--
><monitorThisWS>no</monitorThisWS><!-- find identical Web Services?
how many?--
><searchIdenticalWS>2</searchIdenticalWS></wsp:All></wsp:ExactlyOne>
</wsp:Policy>";

/* The endpoint address of DDBJ */
String endpointAddress = "http://xml.nig.ac.jp:80/xddbj/Blast";

/* Binding the invocation message to DDBJ in the invocation SOAP message
sending to the Sub-Mediator Elite */
SOAPMessage message = soapProc.bindingSOAP(smRequest);

/* Binding relevant information for invoking the Sub-Mediator Elite*/
mesp.insertWS(endpointAddress, serviceQName, portQName, message,
xmlPolicy);
```

```
}
/* Assembling the invocation to another candidate Web Service */
private void ws2(){
  String smRequest = "<soapenv:Envelope
    xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/ "><soapenv:Bod
    y><getFFEntry
    xmlns=\"http://www.themindelectric.com/wsdl/DDBJ/\"><accession>AB0000
    50</accession></getFFEntry></soapenv:Body></soapenv:Envelope>";
  QName serviceQName = new QName("http://xml.nig.ac.jp/xddbj/ DDBJ",
    "DDBJ");
  QName portQName = new QName("http://xml.nig.ac.jp/xddbj/ DDBJ",
    "DDBJ");
  String xmlPolicy= "<wsp:Policy
    xmlns:wsp=\"http://schemas.xmlsoap.org/ws/2004/09/policy/ "
    xmlns:wsmip=\"http://schemas.wsmediator.org/indevidualPolicy/ policy\"><w
    sp:ExactlyOne><wsp:All><!-- Binging method --
    ><bindingMethod>SOAP11HTTP</bindingMethod><!-- Invocation mode:
    RPC | Sync | Async --><invocationMode>Sync</invocationMode><!-- time
    out parameter --><timeout>30000</timeout><!-- auto-set time out parameter:
    average | max  --><autotimeout>average</ autotimeout> <!-- How many time
    to retry after failure--><retryAfterFailure>3</retryAfterFailure><!-- Interval
    between retries --><retryInterval>30</retryInterval><!-- apply multi-routing,
    and number of routes --><multirouting>0</multirouting><!-- start to monitor
    this Web Service locally? no | locally | remotely--
    ><monitorThisWS>no</monitorThisWS><!-- find identical Web Services?
    how many?--
    ><searchIdenticalWS>2</searchIdenticalWS></wsp:All></wsp:ExactlyOne>
    </wsp:Policy>";
  String endpointAddress = "http://xml.nig.ac.jp/xddbj/ DDBJ";
  SOAPMessage message = soapProc.bindingSOAP(smRequest);
  mesp.insertWS(endpointAddress,
    serviceQName,portQName,message,xmlPolicy);
}
```

```
/* Assembling the invocation to another Web Service */
private void ws3(){
  String smRequest = "<soapenv:Envelope
    xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\"><soapenv:Bod
    y><execute
    xmlns=\"http://www.themindelectric.com/wsdl/BlastDemo/\"><accession>AB
    000050</accession></execute></soapenv:Body></soapenv:Envelope>";
  QName serviceQName = new QName("http://xml.nig.ac.jp/xddbj/BlastDemo",
    "BlastDemo");
  QName portQName = new QName("http://xml.nig.ac.jp/xddbj/BlastDemo",
    "BlastDemo");
  String xmlPolicy= "<wsp:Policy
    xmlns:wsp=\"http://schemas.xmlsoap.org/ws/2004/09/policy\"
    xmlns:wsmip=\"http://schemas.wsmediator.org/indevidualPolicy/policy\"><w
    sp:ExactlyOne><wsp:All><!-- Binging method --
    ><bindingMethod>SOAP11HTTP</bindingMethod><!-- Invocation mode:
    RPC | Sync | Async --><invocationMode>Sync</invocationMode><!-- time
    out parameter --><timeout>60000</timeout><!-- auto-set time out parameter:
    average | max  --><autotimeout>average</autotimeout> <!-- How many time
    to retry after failure--><retryAfterFailure>3</retryAfterFailure><!-- Interval
    between retries --><retryInterval>30</retryInterval><!-- apply multi-routing,
    and number of routes --><multirouting>0</multirouting><!-- start to monitor
    this Web Service locally? no | locally | remotely--
    ><monitorThisWS>no</monitorThisWS><!-- find identical Web Services?
    how many?--
    ><searchIdenticalWS>2</searchIdenticalWS></wsp:All></wsp:ExactlyOne>
    </wsp:Policy>";
  String endpointAddress = "http://xml.nig.ac.jp/xddbj/BlastDemo";
  SOAPMessage message = soapProc.bindingSOAP(smRequest);
  mesp.insertWS(endpointAddress, serviceQName, portQName, message,
    xmlPolicy);
}

/* Method for printing execution results */
```

```
private void printResult(){
    System.out.println();
    System.out.println("============================");
    System.out.println("* Final result: *");

      System.out.println(soapProc.SOAPToXMLString((SOAPMessage)results.first
      Element()));

    System.out.println("============================");
    System.out.println("* Final report: *");

    try {
      xmlp.printNodeToConsole((Document)results.lastElement());
      System.out.println();
      //xmlp.printXML((Document)obj);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    //System.out.println();
    System.out.println("============================");

  }

  /* Logs execution results in a file */
  private void wrtFile(long rst){
    FileOutputStream out; // declare a file output object
    PrintStream p; // declare a print stream object
    try
    {
      // Create a new file output stream
      // connected to "myfile.txt"
      out = new FileOutputStream("E:\\Projects\\Current\\testCase.txt");

      // Connect print stream to the output stream
```

```
            p = new PrintStream( out );

            p.append(String.valueOf(rst));

            //p.close();
        }
        catch (Exception e)
        {
            System.err.println ("Error writing to file");
        }
    }
}
```

Appendix F – Example of the valid result from DDBJ

Here we show a valid result expected from the DDBJ Blast Web Service, which contains a gene sequence being used in bioinformatics research.

A. Invoking DDBJ Web Service

Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/DDBJ} DDBJ

Received response:

com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@422d0b

B. The result returned from DDBJ.

==============================

* Final result: *
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><soap:Body><n:g
etFFEntryResponse xmlns:n="http://tempuri.org/DDBJ"><Result
xsi:type="xsd:string">LOCUS AB000050 1755 bp DNA linear
VRL 05-FEB-1999
DEFINITION Feline panleukopenia virus DNA for capsid protein 2, complete cds.
ACCESSION AB000050
VERSION AB000050.1
KEYWORDS capsid protein 2.
SOURCE Feline panleukopenia virus
ORGANISM Feline parvovirus
Viruses; ssDNA viruses; Parvoviridae; Parvovirinae; Parvovirus.
REFERENCE 1 (bases 1 to 1755)
AUTHORS Horiuchi,M.
TITLE Direct Submission
JOURNAL Submitted (22-DEC-1996) to the DDBJ/EMBL/GenBank databases.
Motohiro Horiuchi, Obihiro University of Agriculture and
Veterinary Medicine, Veterinary Public Health; Inada cho, Obihiro,
Hokkaido 080, Japan (E-mail:horiuchi@obihiro.ac.jp,
Tel:0155-49-5392, Fax:0155-49-5402)
REFERENCE 2 (bases 1 to 1755)
AUTHORS Horiuchi,M.
TITLE Evolutionary pattern of feline panleukopenia virus differs from
that of canine parvovirus
JOURNAL Unpublished (1997)

```
COMMENT
FEATURES             Location/Qualifiers
     source          1..1755
                     /isolate="94-1"
                     /lab_host="Felis domesticus"
                     /mol_type="genomic DNA"
                     /organism="Feline panleukopenia virus"
     CDS             1..1755
                     /product="capsid protein 2"
                     /protein_id="BAA19011.1"

/translation="MSDGAVQPDGGQPAVRNERATGSGNGSGGGGGGGSGGVGIST
GT

FNNQTEFKFLENGWVEITANSSRLVHLNMPESENYKRVVVNNMDKTAVKGN
MALDDTH

VQIVTPWSLVDANAWGVWFNPGDWQLIVNTMSELHLVSFEQEIFNVVLKTV
SESATQP

PTKVYNNDLTASLMVALDSNNTMPFTPAAMRSETLGFYPWKPTIPTPWRYYF
QWDRTL

IPSHTGTSGTPTNVYHGTDPDDVQFYTIENSVPVHLLRTGDEFATGTFFFDCKP
CRLT

HTWQTNRALGLPPFLNSLPQSEGATNFGDIGVQQDKRRGVTQMGNTDYITEA
TIMRPA

EVGYSAPYYSFEASTQGPFKTPIAAGRGGAQTDENQAADGDPRYAFGRQHG
QKTTTTG

ETPERFTYIAHQDTGRYPEGDWIQNINFNLPVTNDNVLLPTDPIGGKTGINYTN
IFNT

YGPLTALNNVPPVYPNGQIWDKEFDTDLKPRLHVNAPFVCQNNCPGQLFVK
VAPNLTN

EYDPDASANMSRIVTYSDFWWKGKLVFKAKLRASHTWNPIQQMSINVDNQF
NYVPNNI
                GAMKIVYEKSQLAPRKLY"
BASE COUNT       618 a       271 c       346 g       520 t
ORIGIN
      1 atgagtgatg gagcagttca accagacggt ggtcaacctg ctgtcagaaa tgaaagagct
     61 acaggatctg ggaacgggtc tggaggcggg ggtggtggtg gttctggggg tgtggggatt
    121 tctacgggta ctttcaataa tcagacggaa tttaaatttt tggaaaacgg gtgggtggaa
    181 atcacagcaa actcaagcag acttgtacat ttaaatatgc cagaaagtga aaattataaa
    241 agagtagttg taaataatat ggataaaact gcagttaaag gaaatatggc tttagatgat
    301 actcatgtac aaattgtaac accttggtca ttggttgatg caaatgcttg gggagtttgg
    361 tttaatccag gagattggca actaattgtt aatactatga gtgagttgca tttagttagt
```

```
   421 tttgaacaag aaatttttaa tgttgtttta aagactgttt cagaatctgc tactcagcca
   481 ccaactaaag tttataataa tgatttaact gcatcattga tggttgcatt agatagtaat
   541 aatactatgc catttactcc agcagctatg agatctgaga cattgggttt ttatccatgg
   601 aaaccaacca taccaactcc atggagatat tattttcaat gggatagaac attaatacca
   661 tctcatactg gaactagtgg cacaccaaca aatgtatatc atggtacaga tccagatgat
   721 gttcaatttt atactattga aaattctgtg ccagtacact tactaagaac aggtgatgaa
   781 tttgctacag gaacattttt ttttgattgt aaaccatgta gactaacaca tacatggcaa
   841 acaaatagag cattgggctt accaccattt ttaaattctt tgcctcaatc tgaaggagct
   901 actaactttg gtgatatagg agttcaacaa gataaaagac gtggtgtaac tcaaatggga
   961 aatacagact atattactga agctactatt atgagaccag ctgaggttgg ttatagtgca
  1021 ccatactatt cttttgaagc gtctacacaa gggccattta aaacacctat tgcagcagga
  1081 cgggggggag cgcaaacaga tgaaaatcaa gcagcagatg gtgatccaag atatgcattt
  1141 ggtagacaac atggtcaaaa aactactaca acaggagaaa cacctgagag atttacatat
  1201 atagcacatc aagatacagg aagatatcca gaaggagatt ggattcaaaa tattaacttt
  1261 aaccttcctg taacaaatga taatgtattg ctaccaacag atccaattgg aggtaaaaca
  1321 ggaattaact atactaatat atttaatact tatggtcctt taactgcatt aaataatgta
  1381 ccaccagttt atccaaatgg tcaaatttgg gataaagaat ttgatactga cttaaaacca
  1441 agacttcatg taaatgcacc atttgtttgt cagaataatt gtcctggtca attatttgta
  1501 aaagttgcgc ctaatttaac gaatgaatat gatcctgatg catctgctaa tatgtcaaga
  1561 attgtaactt attcagattt ttggtggaaa ggtaaattag tatttaaagc taaactaaga
  1621 gcatctcata cttggaatcc aattcaacaa atgagtatta atgtagataa ccaatttaac
  1681 tatgtaccaa ataatattgg agctatgaaa attgtatatg aaaaatctca actagcacct
  1741 agaaaattat attaa
//
</Result></n:getFFEntryResponse></soap:Body></soap:Envelope>
==========
```

Appendix G - Execution sequence of unsuccessful process

Here we give an example of a logged execution sequence. The logged file is commented on during the execution and can be easily understood. In this example, no valid results were received from candidate Web Services, as reported in the final report section of the log.

```
init:
deps-jar:
compile-single:
run-single:

====== Parsing Web Service Request Policies ======

Binding Method:            SOAP11HTTP
Invocation mode:      Sync
timeout (ms):         60000
Auto timeout rule:    average
Retry times:          3
Retry interval:       30
Monitor this Web Service:    no
Search identical Web Services:      2

====== Parsing Web Service Request Policies ======

Binding Method:            SOAP11HTTP
Invocation mode:      Sync
timeout (ms):         60000
Auto timeout rule:    average
Retry times:          3
Retry interval:       30
Monitor this Web Service:    no
Search identical Web Services:      2

====== Parsing Web Service Request Policies ======

Binding Method:            SOAP11HTTP
Invocation mode:      Sync
timeout (ms):         60000
Auto timeout rule:    average
Retry times:          3
Retry interval:       30
Monitor this Web Service:    no
Search identical Web Services:      2
```

```
====== Parsing Global Policies ======

Number of Web Services:          3
Priority:                        dependability
Dependability Acceptance:        80
Performance Acceptance:          300
Timeout:                         1000
Web Service: {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
url: http://xml.nig.ac.jp:80/xddbj/Blast
dependability: 50
performance: 300

-------------------------------------------
Web Service: {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
url: http://xml.nig.ac.jp/xddbj/DDBJ
dependability: 80
performance: 400

-------------------------------------------
Web Service: {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
url: http://xml.nig.ac.jp/xddbj/BlastDemo
dependability: 80
performance: 500

-------------------------------------------
Sorting Web Services according Dependability metadata.
Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@dfd90f
------------------------------
Waiting for reply...
------------------------------
Invocation exception: HTTP transport error: java.net.UnknownHostException:
xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@dfd90f
------------------------------
Waiting for reply...
------------------------------
Invocation exception: HTTP transport error: java.net.UnknownHostException:
xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@dfd90f
------------------------------
Waiting for reply...
------------------------------
```

Invocation exception: HTTP transport error: java.net.UnknownHostException: xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/BlastDemo}BlastDemo
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@cefde4

Waiting for reply...

Invocation exception: HTTP transport error: java.net.UnknownHostException: xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/BlastDemo}BlastDemo
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@cefde4

Waiting for reply...

Invocation exception: HTTP transport error: java.net.UnknownHostException: xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/BlastDemo}BlastDemo
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@cefde4

Waiting for reply...

Invocation exception: HTTP transport error: java.net.UnknownHostException: xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@79b7b0

Waiting for reply...

Invocation exception: HTTP transport error: java.net.UnknownHostException: xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@79b7b0

Waiting for reply...

Invocation exception: HTTP transport error: java.net.UnknownHostException: xml.nig.ac.jp
Invoking Web Service (Sync): {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@79b7b0

Waiting for reply...

Invocation exception: HTTP transport error: java.net.UnknownHostException: xml.nig.ac.jp

==========
* Final result: *
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><soap:Fault>
<faultcode>soap:Mediator</faultcode><faultstring>No valid result
received!</faultstring><detail/></soap:Fault></soap:Body></soap:Envelope>
==========
==========
* Final report: *
<?xml version="1.0" encoding="UTF-8"?><report><ws
service="{http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp/xddbj/BlastDemo}BlastDemo"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp/xddbj/BlastDemo}BlastDemo"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp/xddbj/BlastDemo}BlastDemo"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp:80/xddbj/Blast}Blast"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp:80/xddbj/Blast}Blast"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws><ws
service="{http://xml.nig.ac.jp:80/xddbj/Blast}Blast"
validResult="false"><errorMessage>HTTP transport error:
java.net.UnknownHostException: xml.nig.ac.jp</errorMessage></ws></report>
==========
****
* Response Time (ms) : 2012
****
BUILD SUCCESSFUL (total time: 2 seconds)

Appendix H - Execution sequence of successful process

Here we give an example of a logged execution sequence of a successful business process. In this example, a valid result was received from the DDBJ Web Service, which terminated the entire execution, as the quickest response was expected. Details can be found in the final report section of the log.

```
init:
deps-jar:
compile-single:
run-single:

====== Parsing Web Service Request Policies ======

Binding Method:              SOAP11HTTP
Invocation mode:      Sync
timeout (ms):         60000
Auto timeout rule:    average
Retry times:          3
Retry interval:       30
Monitor this Web Service:    no
Search identical Web Services:       2

====== Parsing Web Service Request Policies ======

Binding Method:              SOAP11HTTP
Invocation mode:      Sync
timeout (ms):         60000
Auto timeout rule:    average
Retry times:          3
Retry interval:       30
Monitor this Web Service:    no
Search identical Web Services:       2

====== Parsing Web Service Request Policies ======

Binding Method:              SOAP11HTTP
Invocation mode:      Sync
timeout (ms):         60000
Auto timeout rule:    average
Retry times:          3
Retry interval:       30
Monitor this Web Service:    no
Search identical Web Services:       2
```

```
====== Parsing Global Policies ======

Number of Web Services:          3
Priority:                        dependability
Dependability Acceptance:        80
Performance Acceptanc:           300
Timeout:                         1000
Web Service: {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
url: http://xml.nig.ac.jp:80/xddbj/Blast
dependability: 50
performance: 300

-----------------------------------------
Web Service: {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
url: http://xml.nig.ac.jp/xddbj/DDBJ
dependability: 80
performance: 400

-----------------------------------------
Web Service: {http://xml.nig.ac.jp:80/xddbj/Blast}Blast
url: http://xml.nig.ac.jp/xddbj/BlastDemo
dependability: 80
performance: 500

-----------------------------------------
Sorting Web Services according Dependability metadata.
Invoking Web Service (Sync): {http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ
Outbound SOAP message:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@b48392
------------------------------
Waiting for reply...
------------------------------
Received response:
com.sun.xml.messaging.saaj.soap.ver1_1.Message1_1Impl@422d0b

============================
* Final result: *
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><soap:Body><n:g
etFFEntryResponse xmlns:n="http://tempuri.org/DDBJ"><Result
xsi:type="xsd:string">LOCUS       AB000050            1755 bp    DNA     linear
VRL 05-FEB-1999
DEFINITION  Feline panleukopenia virus DNA for capsid protein 2, complete cds.
ACCESSION   AB000050
VERSION     AB000050.1
KEYWORDS    capsid protein 2.
```

SOURCE Feline panleukopenia virus
ORGANISM Feline parvovirus
Viruses; ssDNA viruses; Parvoviridae; Parvovirinae; Parvovirus.
REFERENCE 1 (bases 1 to 1755)
AUTHORS Horiuchi,M.
TITLE Direct Submission
JOURNAL Submitted (22-DEC-1996) to the DDBJ/EMBL GenBank databases.
Motohiro Horiuchi, Obihiro University of Agriculture and
Veterinary Medicine, Veterinary Public Health; Inada cho, Obihiro,
Hokkaido 080, Japan (E-mail:horiuchi@obihiro.ac.jp,
Tel:0155-49-5392, Fax:0155-49-5402)
REFERENCE 2 (bases 1 to 1755)
AUTHORS Horiuchi,M.
TITLE Evolutionary pattern of feline panleukopenia virus differs from
that of canine parvovirus
JOURNAL Unpublished (1997)
COMMENT
FEATURES Location/Qualifiers
source 1..1755
/isolate="94-1"
/lab_host="Felis domesticus"
/mol_type="genomic DNA"
/organism="Feline panleukopenia virus"
CDS 1..1755
/product="capsid protein 2"
/protein_id="BAA19011.1"

/translation="MSDGAVQPDGGQPAVRNERATGSGNGSGGGGGGSGGVGIST
GT

FNNQTEFKFLENGWVEITANSSRLVHLNMPESENYKRVVVNNMDKTAVKGN
MALDDTH

VQIVTPWSLVDANAWGVWFNPGDWQLIVNTMSELHLVSFEQEIFNVVLKTV
SESATQP

PTKVYNNDLTASLMVALDSNNTMPFTPAAMRSETLGFYPWKPTIPTPWRYYF
QWDRTL

IPSHTGTSGTPTNVYHGTDPDDVQFYTIENSVPVHLLRTGDEFATGTFFFDCKP
CRLT

HTWQTNRALGLPPFLNSLPQSEGATNFGDIGVQQDKRRGVTQMGNTDYITEA
TIMRPA

EVGYSAPYYSFEASTQGPFKTPIAAGRGGAQTDENQAADGDPRYAFGRQHG
QKTTTTG

ETPERFTYIAHQDTGRYPEGDWIQNINFNLPVTNDNVLLPTDPIGGKTGINYTN
IFNT

YGPLTALNNVPPVYPNGQIWDKEFDTDLKPRLHVNAPFVCQNNCPGQLFVK
VAPNLTN

EYDPDASANMSRIVTYSDFWWKGKLVFKAKLRASHTWNPIQQMSINVDNQF
NYVPNNI
GAMKIVYEKSQLAPRKLY"
BASE COUNT 618 a 271 c 346 g 520 t
ORIGIN
1 atgagtgatg gagcagttca accagacggt ggtcaacctg ctgtcagaaa tgaaagagct
61 acaggatctg ggaacgggtc tggaggcggg ggtggtggtg gttctggggg tgtggggatt
121 tctacgggta ctttcaataa tcagacggaa tttaaatttt tggaaaacgg gtgggtggaa
181 atcacagcaa actcaagcag acttgtacat ttaaatatgc cagaaagtga aaattataaa
241 agagtagttg taaataatat ggataaaact gcagttaaag gaaatatggc tttagatgat
301 actcatgtac aaattgtaac accttggtca ttggttgatg caaatgcttg gggagtttgg
361 tttaatccag gagattggca actaattgtt aatactatga gtgagttgca tttagttagt
421 tttgaacaag aaatttttaa tgttgtttta aagactgttt cagaatctgc tactcagcca
481 ccaactaaag tttataataa tgatttaact gcatcattga tggttgcatt agatagtaat
541 aatactatgc catttactcc agcagctatg agatctgaga cattgggttt ttatccatgg
601 aaaccaacca taccaactcc atggagatat tattttcaat gggatagaac attaatacca
661 tctcatactg gaactagtgg cacaccaaca aatgtatatc atggtacaga tccagatgat
721 gttcaatttt atactattga aaattctgtg ccagtacact tactaagaac aggtgatgaa
781 tttgctacag gaacattttt ttttgattgt aaaccatgta gactaacaca tacatggcaa
841 acaaatagag cattgggctt accaccattt ttaaattctt tgcctcaatc tgaaggagct
901 actaactttg gtgatatagg agttcaacaa gataaaagac gtggtgtaac tcaaatggga
961 aatacagact atattactga agctactatt atgagaccag ctgaggttgg ttatagtgca
1021 ccatactatt cttttgaagc gtctacacaa gggccattta aaacacctat tgcagcagga
1081 cgggggggag cgcaaacaga tgaaaatcaa gcagcagatg gtgatccaag atatgcattt
1141 ggtagacaac atggtcaaaa aactactaca acaggagaaa cacctgagag atttacatat
1201 atagcacatc aagatacagg aagatatcca gaaggagatt ggattcaaaa tattaacttt
1261 aaccttcctg taacaaatga taatgtattg ctaccaacag atccaattgg aggtaaaaca
1321 ggaattaact atactaatat atttaatact tatggtcctt taactgcatt aaataatgta
1381 ccaccagttt atccaaatgg tcaaatttgg gataaagaat ttgatactga cttaaaacca
1441 agacttcatg taaatgcacc atttgtttgt cagaataatt gtcctggtca attatttgta
1501 aaagttgcgc ctaatttaac gaatgaatat gatcctgatg catctgctaa tatgtcaaga
1561 attgtaactt attcagattt ttggtggaaa ggtaaattag tatttaaagc taaactaaga
1621 gcatctcata cttggaatcc aattcaacaa atgagtatta atgtagataa ccaatttaac
1681 tatgtaccaa ataatattgg agctatgaaa attgtatatg aaaaatctca actagcacct
1741 agaaaattat attaa
//
</Result></n:getFFEntryResponse></soap:Body></soap:Envelope>
==========
==========
* Final report: *

==========XML Message
==========
<?xml version="1.0" encoding="UTF-8"?>
<report>
<ws service="{http://xml.nig.ac.jp/xddbj/DDBJ}DDBJ" validResult="true">

```
                    <responseTime>5264</responseTime>
                    <errorMessage>null</errorMessage>
        </ws>
</report>
=============================================================================
=========

**********************
* Response Time (ms) : 7814
**********************
BUILD SUCCESSFUL (total time: 9 seconds)
```

Appendix I – Dependability metadata of VBI

Below are shown the dependability metadata of VBI stored on six Sub-Mediators deployed on Planetlab:

- Sub-Mediator, Shanghai, China

```
<ws
service="{http://pathport.bioinformatics.vt.edu:6565/axis/services/blastbt}BlastbtService">
        <dependability>85</dependability>
        <aveResponseTime>54607</aveResponseTime>
        <maximumResponseTime>87267</maximumResponseTime>
</ws>
```

- Sub-Mediator, Beijing, China

```
<ws
service="{http://pathport.bioinformatics.vt.edu:6565/axis/services/blastbt}BlastbtService">
        <dependability>65</dependability>
        <aveResponseTime>59460</aveResponseTime>
        <maximumResponseTime>88506</maximumResponseTime>
</ws>
```

- Sub-Mediator, Newcastle upon Tyne, UK

```
<ws
service="{http://pathport.bioinformatics.vt.edu:6565/axis/services/blastbt}BlastbtService">
        <dependability>91</dependability>
        <aveResponseTime>28990</aveResponseTime>
        <maximumResponseTime>36297< maximumResponseTime>
</ws>
```

- Sub-Mediator, Cambridge, UK

```
<ws
service="{http://pathport.bioinformatics.vt.edu:6565/axis/services/blastbt}Bla
stbtService">
        <dependability>88</dependability>
        <aveResponseTime>26573</aveResponseTime>
        <maximumResponseTime>32675</maximumResponseTime>
</ws>
```

- Sub-Mediator, Washington, USA

```
<ws
service="{http://pathport.bioinformatics.vt.edu:6565/axis/services/blastbt}Bla
stbtService">
        <dependability>96</dependability>
        <aveResponseTime>23945</aveResponseTime>
        <maximumResponseTime>29267</maximumResponseTime>
</ws>
```

- Sub-Mediator, New York, USA

```
<ws
service="{http://pathport.bioinformatics.vt.edu:6565/axis/services/blastbt}Bla
stbtService">
        <dependability>96</dependability>
        <aveResponseTime>24901</aveResponseTime>
        <maximumResponseTime>31297</maximumResponseTime>
</ws>
```