2011

POLICY BASED THIRD PARTY WEB SERVICE MANAGEMENT

Md Sakibul Hasan

POLICY BASED THIRD PARTY WEB SERVICE MANAGEMENT

(Thesis format: Monograph)

by

Md Sakibul Hasan

Graduate Program in Computer Science

A thesis report submitted in partial fulfillment

of the requirements for the degree of

Master of Science

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO

School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Examiners:

Supervisor:

....
Michael James Katchabaw

....
Dr. Hanan Lutfiyya

Supervisory Committee:

....
Sylvia L. Osborn

....
Stuart A. Rankin

The thesis by

Md Sakibul Hasan

entitled:

Policy based third party web service management

is accepted in partial fulfillment of the

requirements for the degree of

Master of Science

....
Date

....
Chair of the Thesis Examination Board

Abstract

Service-oriented computing potentially can help businesses respond more quickly and more cost-effectively to changing market-conditions. Web services are the basic building elements of service-oriented architecture. There are often expectations expected from services that are related to non-functional aspects (i.e. response time, availability) of the web service. The non-functional requirements are referred to as Quality of Service (QoS) requirements. Service Level Agreements (SLAs) are contracts between service providers and service consumers by which the service providers are bound to maintain a certain level of the Quality of Service. SLAs specify conditions on metrics, that represent some aspect of run-time behaviour, that are to be satisfied at run-time. Monitoring of services is needed to determine when SLAs are violated. Adaptive recovery actions are taken to maintain the quality of the service promised on the SLAs. Policies are used to guide the decision making process to determine the appropriate action.

In this work a new system architecture which uses policies to manage web services is proposed and a prototype is implemented to validate the architecture. In this system policies could be added, modified or deleted at system run time. The management task is totally handled by the third party and so, management tasks on the client end are reduced.

The results of the conducted experiments validates the functionality of our proposed architecture and proves that the overhead of using the architecture is less.

Keywords: Web Service, Web Service Management, Web Service Policy

Acknowledgements

First of all I would like to thank my supervisor, Prof. Hanan Lutfiyya, for her guidance and assistance. She was very helpful and encouraging in choosing the topic of this thesis and pursuing the research. Thanks to her for the help and inspiration she extended. I would like to give a nod to my fellow students and DiGS members for being an excellent sounding board for both ideas and the venting of frustrations. I would also like to thank faculty members and Staff of the Department of Computer Science for making my studying here highly enjoyable. And many thanks go to my beloved family and friends for their understanding and endless love through the duration of my Masters program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Service-oriented computing [23] introduces the concept of assembling application components (services) into an application where the services communicate with each other. This application can span multiple organizations and computing platforms. The applications are often aligned with a business process. Service-oriented computing potentially can help businesses respond more quickly and more cost-effectively to changing market conditions. Web services are the basic building elements of service-oriented architecture. There are often expectations of services that are related to non-functional aspects of the web service. For example, there may be expectations on the response time, availability etc. The non-functional requirements are referred to as Quality of Service (QoS) requirements. Service Level Agreements (SLA)s [26] are contracts between service providers and service consumers by which the service providers are bound to maintain a certain level of the Quality of Service. SLAs specify conditions on metrics, that represent some aspect of run-time behaviour, that are to be satisfied at run-time. Monitoring of services is needed to determine when SLAs are violated. Adaptive recovery actions are taken to maintain the quality of the service promised on the SLAs. Policies are used to guide the decision making process to determine the appropriate action.

1.1 Problem statement

Web services are loosely-coupled, self-contained, self-describing software modules that perform a predetermined task. Monitoring is required to ensure that SLAs are satisfied. The ability to respond to SLA violations is also important. Responses are application specific and thus the use of policies is useful. Determining if an SLA has been violated and responses to SLA violations are examples of management tasks. Current work usually has the client implement the management tasks. The disadvantage is that a client typically has limited knowledge of the behaviour of the various services that the application is composed of. For example, assume an application being used by the client consists of two web services: WS1 and WS2. WS2 may be slow causing WS1 to be slow. The client communicates with the application using WS1. It is relatively easy for the client to detect that WS1 is slow but the client may not have sufficient information to realize that WS2 may be the problem. Identifying the source of a problem requires the client's enterprise to continuously monitor and analyze behaviour to determine the source of a problem and the response to the problem. An infrastructure is needed to support this. Service selection is also something that should be supported. For many clients it may be desirable to outsource the infrastructure to a third party. Different clients use different criteria and hence this infrastructure should be able to accommodate this. This can be done using policies. However, little work shows how policies can be incorporated into the infrastructure to support multiple client needs.

1.2 Thesis focus

This thesis focuses on the design of a policy-based third party architecture for web service management, the implementation of a prototype of that architecture, and the validation of the architecture by evaluating the prototype. We propose a new system architecture which uses policies. Policies could be added, modified or deleted at system run time. Our proposed system is a third party management architecture where the client and the provider have to register to use the services provided by the third party system. The client defines three types of policy. The first type of policy provides hints to the third party system on criteria to be used in selecting a service. The second type of policy is used to specify what constitutes a SLA violation (from the client's view) and the third type of policy is used to specify what action is to be taken in response of SLA violations. Based on the policy defined by the clients, the third party system monitors the quality of the provided service and takes recovery action if any violation occurs. The advantage of using a third party for management is that this can be integrated with service discovery. The complexity of management and service discovery can be hidden from clients and providers by outsourcing this functionality. The uniqueness of our architecture is that we introduced policies in a third party system to manage web services which makes the system automated and responsive on SLA violations.

1.3 Thesis organization

The remainder of this thesis is organized as follows: Chapter 2 covers *Background and Related Work*, including key definitions, concepts and a review of the current research relevant to web service monitoring, adaptivity and policies. Chapter 3, *Architecture*, describes the architecture of our proposed policy based third party management system for web services. Chapter 4, *Implementation*, describes the implemented prototype of our proposed architecture. Chapter 5, *Validation*, details the testing of our implemented prototype and evaluate the proposed architecture. Finally, Chapter 6, *Conclusion*, provides some final conclusions and presents ideas and thoughts for future research in the area.

Chapter 2

Background and Related work

This chapter presents key definitions and concepts, and reviews the current research relevant to web service monitoring, adaptivity and policies. Section 2.1 introduces basic concepts useful for understanding this work. Section 2.2 presents different aspects of web service monitoring and some monitoring mechanisms proposed in different papers. Section 2.3 describes failures in web service scenario and reasons for these failures; and actions to recover from these failures. Section 2.4 describes the application of policies. Finally, Section 2.5 presents gaps in current research in web service management.

2.1 Basic concepts

This section describes the basic concepts related to web services. The discussion includes Service Oriented Architecture, platform elements and type of web services and web service composition.

2.1.1 Service Oriented Architecture (SOA)

The World Wide Web Consortium (W3C) refers to Service Oriented Architecture as " A set of components which can be invoked, and whose interface descriptions can be published and dis-

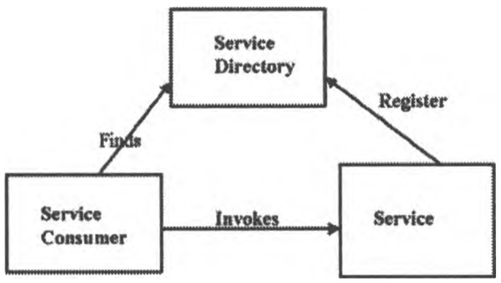

Figure 2.1: SOA Architecture [22]

covered" [16]. According to Component Based Development and Integration [16] SOA refers to "The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface". A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.

2.1.2 Web service

The basic building block of service-oriented architecture is the web service. A web service [31] is a self-contained, self-describing software module that performs a predetermined task. An example task is the following: "verify a customer's credit history". Web services are application components that provide an API that is accessible via Hypertext Transfer Protocol (HTTP) and are executed on a remote system host where the requested services reside. Web services communicate using open protocols to complete tasks, solve problems, or conduct transactions on behalf of a user or application. Web services communicate over private or public network to virtually form a single logical system.

The main platform elements of web services are SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery and Integration). [31]

SOAP (Simple Object Access Protocol)

SOAP [33] is an XML-based protocol used to access web services and to allow independent services to exchange information over HTTP. It is a format for sending messages used by services to communicate over the Internet. SOAP is platform and language independent and allows an application to get around firewalls. This format is standardized by the World Wide Web Consortium (W3C) [6]. As an example of how SOAP can be used, a SOAP message could be sent to a web-service-enabled web site, e.g., a real-estate price database, with the parameters needed for a search. The site would then return an XML-formatted document with the resulting data, e.g., prices, location, features. Since the data is returned in a standardized machine-parseable format, it could then be integrated directly into a third-party web site or application.

Figure 2.2 is an example of a SOAP message. A SOAP XML document instance is called a SOAP message or SOAP envelope. It is carried as the payload of other network protocols like HTTP. The SOAP message consists of an Envelope (line 7) element containing an optional Header element (line 11) and a mandatory Body element (line 17). The optional SOAP Header element contains application-specific information (e.g., authentication, payment, etc.) about the SOAP message. The mandatory SOAP Body element contains the actual SOAP message intended for the ultimate endpoint of the message.

WSDL (Web Services Description Language)

WSDL [35] is an XML-based language which is used to locate and describe web services. It is a specification schema that describes web services by specifying the API of a web service that

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
  <m:Trans xmlns:m="http://www.w3schools.com/transaction/"
  soap:actor="http://www.w3schools.com/appml/">234
  </m:Trans>
</soap:Header>

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Figure 2.2: SOAP Message [32]

can be used by external entities. WSDL describes the contract for application communication. Web services can be made accessible by using WSDL definitions to generate code that knows precisely how to interact with the web service described, and hides details in sending and receiving SOAP messages over different protocols.

A WSDL document is a XML document which contains a set of definitions to describe a web service. Figure 2.3 contains an example of a WSDL message. The PortType element (line 37) describes a web service, the operations that can be performed, and the messages that are involved. It can be compared to a function library in a traditional programming language. The Message element (line 29,33) defines the data elements of an operation. The Types element (line 9) defines the data types that are used by the web service. For maximum platform neutrality, WSDL uses XML Schema syntax to define data types. The Binding element (line 44) defines the message format and protocol details for each port. A WSDL document can also contain other elements, like extension elements, and a service element (line 57) that makes it possible to group together the definitions of several web services in one single WSDL document.

UDDI (Universal Description, Discovery and Integration)

UDDI [34] is a directory service where companies can register and search for Web services. The description of web service interfaces which are written in WSDL are stored in UDDI. UDDI is an open industry initiative enabling businesses to publish service listings and discover each other and define how the services or software applications interact over the Internet. Communication to the UDDI occurs through the SOAP protocol.

2.1.3 Types of web services

Web services can be categorized based on two different criteria. These are described in this section.

Protocol category

Web services can be divided into two types based on the protocol used: SOAP based "Big" web services and "RESTful" web services [1].

- **SOAP based "Big" web services:** Big web services use XML messages that use the SOAP standard. In such systems, there is often a machine-readable description of the operations offered by the service written in WSDL. The architecture of such web services indicate complex non-functional requirements such as transactions, security, trust etc.
- **"RESTful" web services** RESTful web services require minimal resources for building. WSDL service-API definitions are not required for RESTful web services. This is why RESTful web services are inexpensive and their adoption rate is currently high [1]. RESTful web services are stateless and a caching infrastructure can be leveraged for performance boosting. In existing web sites web service delivery and aggregation can be easily enabled using RESTful style.

```xml
<?xml version="1.0"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote.wsdl"
             xmlns:tns="http://example.com/stockquote.wsdl"
             xmlns:xsd1="http://example.com/stockquote.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
         <complexType>
           <all>
             <element name="price" type="float"/>
           </all>
         </complexType>
      </element>
    </schema>
  </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>

  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>

  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>

</definitions>
```

Figure 2.3: WSDL Message [18]

Level of simplicity category

Based on the simplicity and usages of web services, web services can be categorized in two types: Simple web services and complex web services [11].

- **Simple web services:** The web services which provide request/response type functionality and do not support transactions are *simple web services*. These kind of web services are informational. These services provide access to content through interaction with an end user by means of simple request/response sequences, or alternatively may expose back-end business applications to other applications.

- **Complex web services:** The web services which support transactions and provide a framework for business-to-business collaborations and business process management are *complex web services*. Complex web services typically involve the assembly and invocation of many pre-existing services found in diverse enterprises to complete a multi-step business interaction.

2.1.4 Web service composition

Web service composition provides an open, standards-based approach for connecting web services together to create higher-level business processes. Services provided by different providers can be merged to create a new service which is referred to as a *Composite web service*. Web service composition is defined in [2] [39] [5]. Anis Charfi [2] states that web service composition provides a means to create value-added web service by combining existing web services. Farhana H. Zulkernine [39] states that Web services can be composed to create complex business processes that span multiple organizations. Boualem Benatallah [5] states that a composite Web service is an umbrella structure that aggregates multiple other elementary and composite Web services, which interact according to a given process model.

Today businesses need to quickly adapt to customer needs and market conditions. This is why businesses need to be flexible internally and externally. Web services offer the greatest potential of weaving together multiple services dynamically into a composite service system representing a business process [13] . This allows businesses to be adaptable.

Service orchestration [30] [37] and service choreography [38] are two different approaches for web service composition. In service orchestration there is a central controller process which controls and co-ordinates all the web services involved in the application. This central process can be a web service as well. The constituent web services do not know that they are participating in a higher-level business process. How the constituent web services will be called and what will be the control flow are known only to the central controller process. The other web services simply serve the requests whenever called. On the other hand in service choreography, there is no central controller process. All the constituent web services know when to call, whom to interact, when to execute operations etc. Service choreography can be viewed as a collaborative effort of many participating web services and as there is no controller hence all the web services need to know the actual business process and things involved in it like message exchanges, time of call, etc. The Web Services Business Process Execution Language (WS-BPEL) [36] specification defines a language to specify service orchestration and Web Services Choreography Description Language (WSCDL) [7] is a language to specify service choreography.

2.1.5 QoS attributes of web services

Web services have functional and non-functional requirements. Non-functional requirements of web services are also known as Quality of Service (QoS) requirements. QoS attributes of a web service refer to the quality aspect of a web service. A QoS attribute represents some aspct of run-time behaviour. Anton Michlmayr [15] defines a QoS model where QoS attributes of web services are grouped into four categories: performance, dependability, security/trust and cost/payment. Each category consists of related metrics (attributes) that represent some aspect of run-time behaviour. These are briefly described in this section.

Performance

Performance metrics include service time, latency, response time, throughput and scalability. *Service time* represents the time it takes for a service to execute a request. *Latency* is the time that is needed for the client request to reach the service. The *response time* measures the overall time needed for the request at the service consumer. *Throughput* represents the number of service requests that can be processed within a given time period. *Scalability* defines performance behavior of a service when the throughput increases.

Dependability

Availability and Accuracy are dependability related attributes that address the ability of services to avoid frequent and severe failures. These attributes are measured for service. *Availability* represents the probability that a service is up and running. *Accuracy* defines the ratio of successful service executions in relation to the total number of requests.

Security/Trust

Security for web services means providing authentication, authorization, confidentiality, traceability and data encryption. *Authentication* refers to the identification of users who can access service and data. *Authorization* means users should be authorized so that only authorized users can access the protected services. *Confidentiality* refers data that only authorized users can access or modify the data. *Traceability* means that it should be possible to trace the history of a service when a request was serviced. *Data encryption* means that data should be encrypted.

Cost/Payment

Price and Penalty fall in this category. *Price* refers to the money that a client has to pay to use the service. *Penalty* represents the compensated amount the service provider will pay a customer if the contract is being violated by them.

2.2 Web service monitoring

It is critical that web services be monitored to ensure that Quality of Service (QoS) requirements are satisfied. This section describes the necessity of monitoring, monitoring of QoS attributes, basic monitoring techniques, monitoring of composite web services and some monitoring mechanism proposed on different articles.

2.2.1 Quality of Service (QoS) monitoring

Quality of service plays an important role in service oriented system. Quality of services attributes could be classified as deterministic and non-deterministic. Deterministic QoS attributes are known before a service is invoked such as price. Non-deterministic QoS attributes are unknown at service invocation time such as service time, availability etc. Monitoring is needed for measuring non-deterministic attributes.

Quality of services is guaranteed by Service level agreement (SLA). Service Level Agreements (SLA)s [26] are signed contracts between two parties for satisfying clients, managing expectations, regulating resources and controlling costs. An SLA has a set of Service level Objectives (SLOs) [26]. A Service-Level Objective (SLO) is a condition on a QoS attribute. For example, “the response time should be less than 5000 milliseconds”. An SLA consists of SLOs. To ensure that SLAs are satisfied, efficient monitoring of the SLAs is needed.

2.2.2 Basic monitoring techniques

The SOAP messaging protocol is typically used for web service communication. Information is carried through the SOAP messages. The typical mechanism [39] used to monitor web services is through message interception. There are several ways to monitor web services through message interception. These are briefly described in this section.

Internal agents in messaging framework

One approach to message interception is to have internal agents in the messaging framework at the servers that host the web services [39]. The task of these agents is to collect data by intercepting messages and send this data to an external agent that is responsible for maintaining this data. These internal agents are considered as a standard part of the messaging framework and provide monitoring data as a service. The advantage of this approach is that since the agent is an internal part of the messaging framework there is less overhead on communication and there is no bottleneck or point of failure [39]. On the other hand the management of this agent is complicated as there is a need to modify the messaging framework to update the agent.

External intermediaries

Another approach is to have external intermediaries [29] which reside between the service provider and the service consumer that intercept messages which are transferred between the service provider and the service consumer. The intermediaries are separate entities from the messaging framework and thus are easier to manage. On the other hand, it requires an additional level of message redirection which causes overhead on communication, possible bottlenecks and a point of failure.

Code level instrumentation

Another monitoring technique is code level instrumentation [39] that provides various monitoring and reporting functions. There is a well defined application programming interface (API) for these monitoring and reporting functions. The advantage of this technique is that it can report extensive and accurate monitoring data whereas the cost of maintaining the code can be considerable. It would be an efficient solution to the monitoring problem by publishing management web services for querying performance data or getting automated event notifications from the service providers. However, it requires that the service provider implements customized management frameworks.

2.2.3 Proposed monitoring infrastructures

Extensive research has been done on monitoring QoS attributes of web services and several monitoring infrastructures are proposed. This section describes some of these monitoring infrastructures.

Grand Slam

Josef Spillner [27] proposed a monitoring module called Grand Slam. Grand Slam consists of several parts. A core monitor module manages and controls the other modules. It installs a trigger in the database which notifies of additions or removals of SLAs. The measurement module interacts with sensors that monitor QoS attributes. The measurement module assumes that there are existing sensors. The sensors are not part of the SLAM infrastructure. The measured values of the QoS attributes are compared to threshold values. For each QoS attribute, there is a lower threshold value and an upper threshold value. The values between the lower and upper threshold values are referred to as the critical area. If the measured value of the QoS attribute is not within the critical area then a violation occurs. The measurement module returns measured values to the core monitor module. The core monitor module stores the measurements

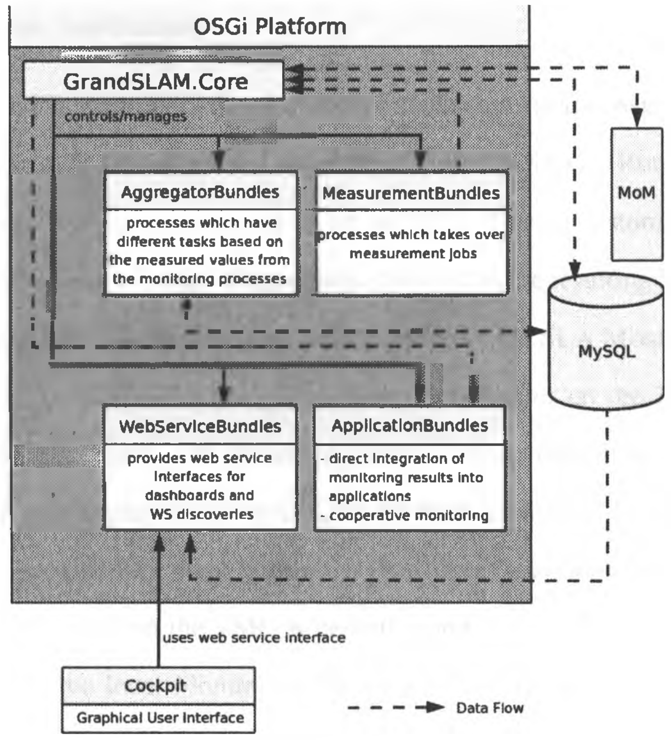
OSGi Platform
GrandSLAM.Core
controls/manages
AggregatorBundles
processes which have different tasks based on the measured values from the monitoring processes
MeasurementBundles
processes which takes over measurement jobs
MoM
MySQL
WebServiceBundles
provides web service interfaces for dashboards and WS discoveries
ApplicationBundles
- direct integration of monitoring results into applications
- cooperative monitoring
uses web service interface
Cockpit
Graphical User Interface
Data Flow

Figure 2.4: Web service monitoring component proposed by Josef Spillner [27]

and results of the comparisons of the measurements to critical areas into the database. Grand SLAM's aggregators are started by the monitor core but work independently from it. In Grand SLAM there are three aggregators: One generates aggregated values of a QoS attribute (e.g. service time, availability) like average, minimum and maximum. A second aggregator creates scalar vector graphics which contain pie and line charts and there is an aggregator that creates XML files with a ranking of the monitored services based on the aggregated values generated by the first aggregator. The fourth part is an Axis 2 server. On this server, a web service is deployed which allows an invocation from a service discovery. The task of this web service is to register and unregister service level agreements which have to be observed. This system is graphically illustrated in Figure 2.4.

Distributed Monitor Architecture

Josef Spillner also [28] proposed a distributed monitor architecture consisting of one Service Management Platform (SMP) and several distributed Tradeable Service Runtime (TSR) servers for hosting services. Web services are deployed on TSRs. Once a customer has negotiated a contract via the SLA Manager's SLA Negotiation component, the resulting SLA is stored in the SLA Repository and the SLA Manager sends a message to the SLA Monitoring Coordinator that a new SLA is available. The SLA Monitoring Coordinator on the TSR then starts the appropriate monitoring sensors. The monitoring sensors collect data from various sources and compares the data with expected values of QoS attribute as defined by SLOs. The sensors are assumed to be provided by other parties. It then stores the monitoring data to the local Monitoring Database found on the TSR. A central monitoring backend of SMP collects the monitoring data from the local Monitoring Database of a TSR and merges it into a central database found at the SMP. Consumers can access the monitored data via Monitoring as a Service (MaaS) and do calculations that allow for detection of SLA violations. In case of an SLA violation, the SLA Monitoring component triggers the adaptation coordinator to start one of the adaptation mechanisms. This system is graphically illustrated in Figure 2.5.

SLM Engine

Akhil Shahai [26] proposed a SLM Engine for monitoring SLAs. The management components of the SLM engine are coordinated by the SLM Process Controller. Whenever the SLM Engine receives an SLA as input the SLM process is initiated. The SLA is sent to the SLA customizer component that in turn creates the SLOs and stores those SLOs in the SLA repository. Once configured, the SLA evaluator is activated to start evaluation of the SLOs. The SLA evaluator compares the data collected from various sources against thresholds defined on QoS parameters. If the result of the comparison indicates a violation then it is maintained as a violation record in the violation engine component. The SLA evaluator is installed at both the client side and the server side and so it is able to be evaluated at any side based on the

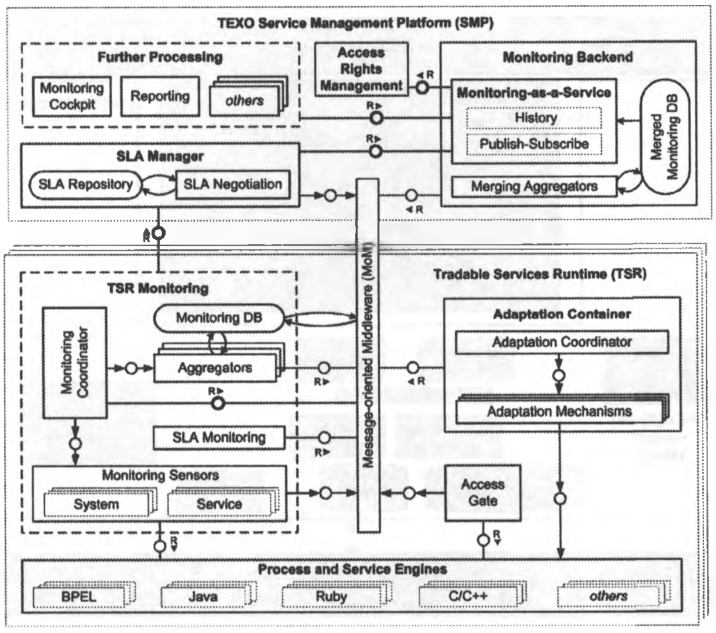

Figure 2.5: Web service monitoring component proposed by Sandro Reichert [28]

availability of measurement items. If however, a QoS attribute is measured at the client side (e.g. availability), and some others are at the server side (e.g. service time), then the evaluation takes place at the server side. In this case the client side monitoring data is transferred to the server side. The Measurement exchange protocol is used to transfer these measurements. This system is graphically illustrated in Figure 2.6.

Performance Monitor

Farhana H. Zulkernine [39] proposed the Performance Monitor (PM) for monitoring SLAs. The PM takes a set of negotiated SLAs and a workflow description as input and monitors the performance of the component services to verify that the SLAs are satisfied. The PM is comprised of two subsystems: a Primary Subsystem (PS) and multiple Secondary Subsystems (SS). The SSs monitor service performance at the service providers' locations using code level instrumentation [39] monitoring techniques and send the reports to the PS. The PS accepts monitoring requests, receives monitoring reports, analyzes the reports to verify SLAs, and ac-

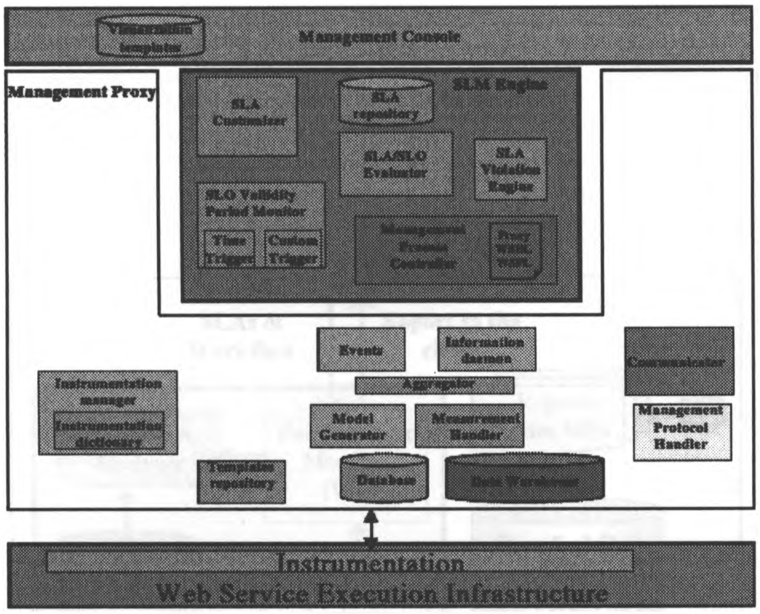

Figure 2.6: Web service monitoring component proposed by Akhil Shahai [26]

cordingly generates notifications for the respective service consumers.The PS has Performance Monitor Web Service, Workflow Analyzer, Performance Monitor Database and Report Analyzer components to accomplish these tasks. This system is graphically illustrated in Figure 2.7.

2.3 Adaptivity

QoS violations occur because of different system failures and these failures are caused by faults. This section describes different failures, reasons for these failure and actions to recover from these failures.

2.3.1 Web service failure and fault

The IEEE Standard Glossary of Software Engineering Terminology defines failure and fault [12]. According to IEEE, *failure* is the inability of a system or component to perform its

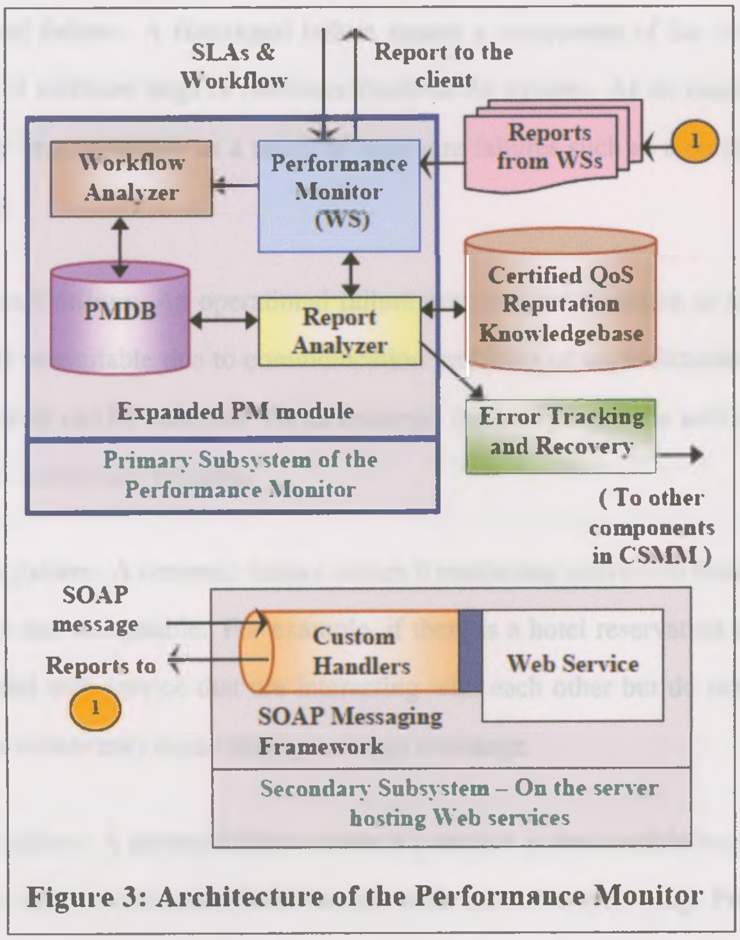

Figure 2.7: Web service monitoring component proposed by Farhana H. Zulkernine [39]

required functions within specified performance requirements. IEEE defined that a *fault* is (1) A defect in a hardware device or component; (2) An incorrect step, process, or data definition in a computer program. A fault is the cause of failure and failure is the result of fault. W. He [10] defined several kinds of failures which caused QoS violations.

- *Functional failure:* A functional failure means a component of the system has failed because of software bugs or hardware faults in the system. As an example, a web service may be unavailable as a result of hardware failures such as a computer hard drive crashing.

- *Operational failure:* An operational failure occurs when a system or some participant service is unavailable due to communication problems or unpredictable load. No more new requests can be accepted. As an example, the heavy load of a web service makes it unable to accept new requests.

- *Semantic failure:* A semantic failure occurs if interacting operations between two participants are not compatible. For example, if there is a hotel reservation web service and a car rental web service that are interacting with each other but do not use same time format, a failure may occur during message exchange.

- *Privacy failure:* A privacy failure occurs if a service is inaccessible because the service is privacy sensitive and would not disclose information to everybody. For instance, a car rental web service may require customers' age or their detailed travel schedules, while other web services would not expose this information because of their privacy policies.

- *Security failure:* A security failure arises when data are accessed without enough credential or authority, or without special secure link. As an example, a web service only accepts SOAP messages over the secure HTTP, while the SOAP messages are sent over standard HTTP. In that case a security failure will occur.

2.3.2 Recovery Actions

Service oriented recovery actions are dependant on the internal structure or orchestration of the service. The recovery actions [8] that could be applied on the web services as a remedy for QoS violations include the following:

- *Retry web service invocation:* If the service is unavailable temporarily, possible action is to suspend the execution of the process and retry the invocation of the unavailable services.

- *Substitute web services:* If a service is considered to be definitely unavailable, then it is required to substitute the failed service with new similar service.

- *Reallocate resources of web services:* If the fault that causes the QoS violation is a lack of hardware of software resources then a possible action is to reallocate resources to the web service.

- *Change of process structure:* Modifying the process structure takes place if service substitution or reallocation can not solve the problem. The new process is defined by studying the log of old process and applied on the system. For example, if the whole task of a composite web services has some processes and shuffling the order of processes does not impact on the final result, then changing the process order can solve the problem.

2.4 Policy

Policies can be used to guide decision making in management systems. This section defines policy and describes different policy languages and application of policies on web service scenario.

2.4.1 What is policy

Emil C. Lupu [14] defined that a policy is information that can be used to modify the behavior of a system without the need for re-compiling or re-deployment of the system. In the context of web service management, a policy can be defined as “a high level statement as to how business requirements should be processed in the management system” [25]. Policies are used to guide the management system’s decisions and actions. Policies can be associated with or attached to a service or interactions. Interactions that policies can be applied to include authentication, authorization, auditing, privacy protection, routing, performance etc.

Policies can take several forms. The most common form of a policy associates an event with one or more rules of conditions and actions. An event represents a change of state that is of interest. Notifications of events are through messages. The rules of a policy are evaluated by a management component when a notification of the event is received by that management component. The rules are used to determine the actions to be taken in response to the event. Another form of policy is an assertion which defines a condition that must be satisfied by the system that the policy applies to.

There are several languages for specifying policies including IETF, Ponder, KAoS, Rei and WS-Policy [24]. Policy languages are used to standardize the policies and to organize the business logic of a system properly.

2.4.2 Policies for self-healing web services

To automatically maintain the desired conditions on QoS attributes is referred to as self-healing. Self-healing requires monitoring of system behaviour to determine possible degradation, diagnosis to determine the root cause of the degradation and a repair process which may require the execution of one or more actions. In this section we describe how policies can be used to determine actions[4].

Reactive policies

The following are reactive policies [9] which can be used to determine possible actions to recover from a degradation of system behaviour:

- *Retry policy:* This policy is activated by an event indicating a degradation of service behaviour. The retry policy will invoke the same faulty service hoping that the failure is transient. The condition used to determine if the action can be taken is if the number of retries has not exceeded some threshold value and that the service is idempotent. The service is idempotent if the response of each request produce the same value every time.
- *Substitute policy:* Substitute policy is activated by an event which indicates a degradation of service quality. This policy will substitute the faulty service and dynamically bind to a replacement service that offers equivalent functional and QoS properties. One condition that is used to determine if the action can be taken is if the number of retries exceeded some threshold value. The replacement service should leave the process in an equivalent state that was expected from the substituted service. In some case a faulty service could be replaced by a service composition that has equivalent effects as the faulty service.
- *Parallel execution policy:* This policy is activated by an event indicating a degradation of service behaviour. The policy will find out other services that offers equivalent functional and QoS properties. It will then invoke all of these services and wait for the first responding service. This strategy is more suitable for data lookup services and freely available services such as web search.
- *Dynamic binding policy:* This policy is activated with indicating of service quality degradation. This policy will perform a structure change which will impact an indication the flow of execution of the whole process. The policy can either change the direction of flow of the process or use some control command (i.e. skip, wait, start, terminate, suspend and resume a process activity) to manipulate the execution result.

Predictive policy

In a predictive self-healing policy, the monitoring services cooperate with the prognosis services to predict service degradation and to act appropriately by reconfiguration plans. In this scenario, policies are defined which can predict future possibilities of degrading QoS properties by analysing historical data of web services and take proactive actions to get rid of possible future degradation of QoS properties. These proactive actions may include adapting service composition by switching locally or remotely between different web service instances.

2.5 Research gap

Web services are a loosely-coupled, self-contained, self-describing software module that performs a predetermined task. QoS monitoring is required to ensure the integrity and quality of the service. In this chapter we discussed QoS monitoring of web service based on QoS requirements and policies. An overview of different monitoring mechanisms is given. In these mechanisms, management tasks are often found on the client side and there is a lack of the use of policies to determine actions to respond to QoS violations. It may be feasible to have a third party management to carry out the management tasks of client. The third party management system would manage on behalf of multiple clients. The management decisions for different clients are often based on different criteria. Policies can be used to influence decision making.

Chapter 3

Architecture

This chapter describes the architecture of our proposed policy based third party management system for web services. Section 3.1 describes policies that we have used in our architecture. Section 3.2 provides an overview of the system. The remaining sections describe the functionality of each component of the system and the interactions among the various system components.

3.1 Policies

In our proposed architecture we have defined three types of policies: service selection policy, violation policy and recovery policy,

3.1.1 Service selection policy

For a type of service there may be more than one service instance. Composition requires that a service instance be chosen. This service instance selection should result in a service that satisfies the QoS requirements of the composition. We assume that the QoS requirements can be characterized by desired values of attributes where an attribute represents some aspect of run-time behaviour. Service selection can be guided by service selection policies defined by the clients. The general form of a service selection policy is the following:

```
POLICY policy name {
        ServiceType (service type name),
        QoSParameters {
                (attributeName, lowerbound, upperbound, rateofChange,priority),
        }
}
```

An example of a policy that can be used for service selection is presented in Example 1.

Example 1:

```
POLICY selection-xyz {
        ServiceType (xyz),
        QoSParameters {
                (ServiceTime, 2000, 4000, 100, 1),
                (Availability, 0.7, 0.9, 0.05, 3),
                (Cost, 20, 90, 10, 2),
        }
}
```

3.1.2 Violation policy

Another type of policy is used to define what constitutes a violation of an SLA. The general form is the following:

```
POLICY policy name {
        serviceType (service type name),
        QoSParameters {
                (attributeName, maxNumViolations),
        }
}
```

An example of a policy that can be used for detecting violations is presented in Example 2.

Example 2:

```
POLICY violation-xyz {
        ServiceType (xyz),
        QoSParameters {
                (ServiceTime, 5),
                (Availability, 5),
        }
}
```

3.1.3 Recovery policy

When the system detects a SLA violation, it will take recovery actions. These actions are guided by recovery policies that are defined by the client. The general form of a recovery policy is the following:

```
POLICY policy name {
        serviceType (service type name),
        QoSParameters {
                (violationType, action),
        }
}
```

An example of a policy that can be used for taking recovery action is presented in Example 3.

Example 3:

```
POLICY recovery-xyz {
        ServiceType (xyz),
        QoSParameters {
                (ServiceTimeViolation, changeProvider),
                (AvailabilityViolation, doNothing),
        }
}
```

3.2 System Overview

Our proposed policy based third party management system does the monitoring and management of SLAs on behalf of the client. The basic system architecture is presented in Figure 3.1. There are main three components: (1) The Client Agent which processes and manages data on the client side; (2) The Provider Agent which processes data related to the server; and (3) The Third Party Agent which is responsible for negotiation, SLA management, diagnosis of SLA violations and determining recovery actions. To use the system, clients and providers register with the third party. We assume that the registration process includes the negotiation of a SLA. When a client uses the service of the provider the client invokes a request which is intercepted by the Client Agent. The Client Agent executes pre-defined processing task requests and sends requests to the Provider. The Provider Agent intercepts requests, executes pre-defined processing tasks and forwards requests to the provider. The provider's response is intercepted by the Provider Agent. The Provider Agent adds information to the response and forwards the modified response to the Client. The Client Agent intercepts the response, processes the response and stores information about the service invocation in a local database. The Client Agent then sends the response to the client. The local database is continuously synchronized with the cen-

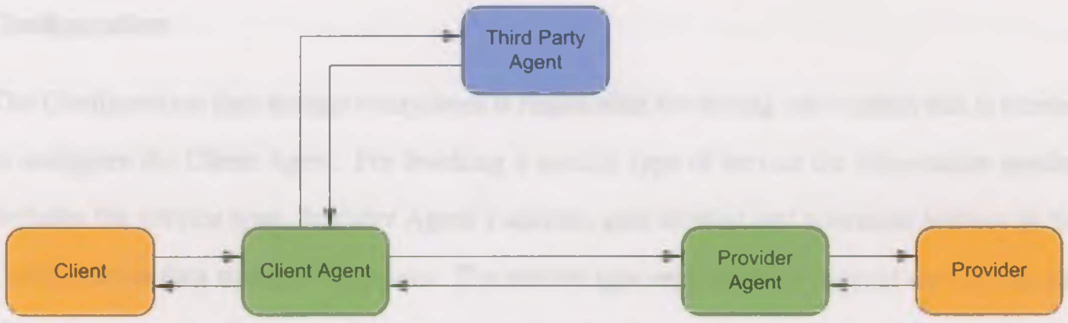

Figure 3.1: System Architecture

tral database of the Third Party Agent. The Third Party Agent analyses the data to determine SLA violations and actions in response to those violations.

This approach essentially reflects message interception by a process that is not part of the messaging framework. The advantage of this approach is that if the messaging framework changes it is easier to make changes to the Client Agent and the Provider Agents.

3.3 System Components

The system consists of three major components. This section describe the details of each of these components.

3.3.1 Client Agent (CA)

The Client Agent component is a proxy for a client which is provided by the Third Party when the client registers with the Third Party system. The Client Agent is installed on the client side and every service request is intercepted by the Client Agent. The Client Agent has two data storage components: Configuration and Logs. The Client Agent also has two processing components: Collector and Synchronizer.

Configuration

The Configuration data storage component is responsible for storing information that is needed to configure the Client Agent. For invoking a specific type of service the information needed includes the service type, Provider Agent's address, port number and operation address in the Configuration data storage component. The service type represents the type of service that the client is requesting. Every service type is associated with a group of policies (Service selection policy, violation policy and recovery policy) which is defined by the client. A policy identifier uniquely identifies this group of policies. The Provider Agent's address is an IP address where the Provider Agent is installed. The port number is the port number of the machine where the Provider Agent is listening for service requests. The operation address is the address of a specific operation that the client wants to invoke and it is in the form of a URI (Uniform Resource Identifier).

Collector

The task of the Collector is to forward the service request to the Provider Agent associated with the service request, return the response from the service to the client and maintain a log of service requests. The Client invokes a service request mentioning the desired type of service. This service request is intercepted by the Collector. The Collector uses the service type to find the information needed for invoking the requested service from the Configuration data storage. The Collector then forwards the service request to the Provider Agent. When a response for a requested service is received from the Provider Agent, the Collector parses the response and stores QoS data (service time, availability) for that service invocation in the Logs data storage component. The Collector then forwards the response to the Client.

Logs

Logs is a data storage component of the Client Agent which stores information of the service request. For each service request the Policy Identifier, SLA Identifier, Provider Agent's address, port number, operation address, starting time of the request, end time of the request, processing time of the request, service time of the request (time taken by the provider to process the service) and availability of the provider is stored. This stored data is uploaded to the central data storage of the Third Party Agent by the Synchronizer processing component at regular intervals.

Synchronizer

The task of the Synchronizer is to synchronize the service invocation data found in the Logs data storage of the Third Party Agent with the service invocation data of the Logs data storage of the Client Agent. The Synchronizer runs at regular time intervals to determine if there is any new data in the Logs data storage of the Client Agent. Every time the Synchronizer runs, it re-initializes the Logs data storage of the Client Agent. Thus for each check, the data in the Logs represents data that is not in the central repository. When the Synchronizer finds new data in the Logs data storage of the Client Agent it uploads the new data to the Logs data storage of the Third Party Agent.

3.3.2 Provider Agent (PA)

The Provider Agent functions as a proxy for the service from the Provider. It is provided by the third party and the Provider installs it on its machine as a contract term of it being registered with the third party. The task of the Provider Agent is to calculate the service time which is the time taken by the server to process the requested service. When the Provider Agent receives the service request from the Client Agent it starts a timer, forwards the request to the Provider and waits for the response from the Provider. When it receives the response from the Provider it stops the timer. The elapsed time between the start and end of the timer is used to represent the service time. The Provider Agent then sends the calculated service time to the Client Agent by appending the information with the response to the service invocation. This data is used by the Third Party Agent to detect SLA violations.

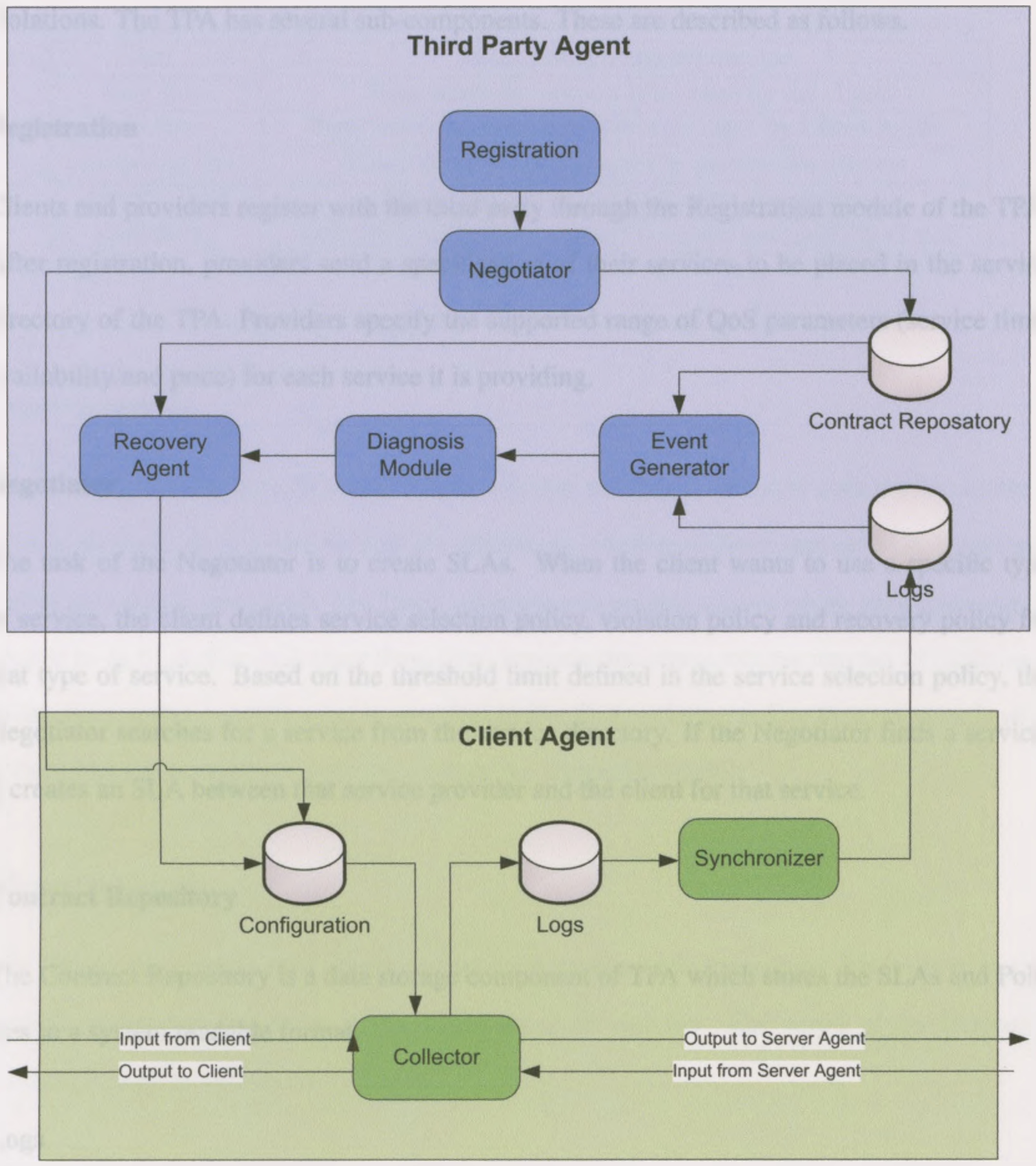

Figure 3.2: Client Agent and Third Party Agent components and their interaction

3.3.3 Third Party Agent (TPA)

The Third Party Agent is responsible for management and decision making to respond to SLA violations. The TPA has several sub-components. These are described as follows.

Registration

Clients and providers register with the third party through the Registration module of the TPA. After registration, providers send a specification of their services to be placed in the service directory of the TPA. Providers specify the supported range of QoS parameters (service time, availability and price) for each service it is providing.

Negotiator

The task of the Negotiator is to create SLAs. When the client wants to use a specific type of service, the client defines service selection policy, violation policy and recovery policy for that type of service. Based on the threshold limit defined in the service selection policy, the Negotiator searches for a service from the service directory. If the Negotiator finds a service, it creates an SLA between that service provider and the client for that service.

Contract Repository

The Contract Repository is a data storage component of TPA which stores the SLAs and Policies in a system readable format.

Logs

Logs is the other data storage component of the TPA. It is synchronized with the Logs data storage of the Client Agent. Logs data storage component contains a table which stores service request information. Each entry in the Logs represents information about a service invocation. The information is presented in Table 3.1

Column Name	Description
id	Primary key for the table
contract_id	Id of the SLA
host	Address of the Provider Agent of the contracted provider
port	Port number of the server on which Provider Agent is running
uri	URI of the contracted service
start_time	Time when the request is invoked by the client
end_time	Time when the response is sent to client by Client Agent
processing_time	Time taken by Client Agent to process the request
service_time	Time taken by server to process the request
availability	Indicates whether the request is served successfully or not

Table 3.1: logs table of Logs data storage of TPA

Event Generator

The Event generator uses SLAs and SLA violation policies to generate events that represent SLA violations. SLA violation policies specify the number of times that an SLA is violated before an event is generated. The Event generation maintains a table where each entry corresponds to an SLA. For each entry there is an attribute that represents the number of violations. When the number exceeds what is specified in the SLA violation policy then an event is generated. The generated events are the input of the Diagnosis Module.

Diagnosis Module

The Diagnosis Module uses events as input for its diagnosis algorithm. The output of this module is the root cause that is causing SLAs to be violated. The output of the Diagnosis module is used by the Recovery Agent.

Recovery Agent

The Recovery Agent takes input from the Diagnosis module, analyses the input and executes reactive actions based on the recovery policy defined by the client to prevent the SLA violations.

3.4 Work flow of the system

In the proposed system there are three parties: client, provider and third party. This section describes the interactions between these parties.

3.4.1 Registration

Both the client and the provider need to be registered with the third party. This is done through the use of the Registration module.

Client registration

The Client provides its information (i.e. Name, Address, Contact number) in the format expected by the third party. The Client also provides information for payment. This information is used to pay the service charge for the services provided by the third party. The Client also has to pay the third party for finding and composing the services. The Third party provides the client Client Agent software. The Client installs this software.

Provider registration

The Service Provider provides its basic information (i.e. Name, Address, Contact number) when it registers with the TPA. The Service Provider also has to pay the third party service manager for TPA facilities such as the service directory. This requires that the Service Provider provides information needed for payment. The Service provider provides a specification of all of its services with supported range of QoS parameters (service time, availability and price) to the service directory of the third party service manager so that those services can be found by the Negotiator of the TPA. The Third party provides the service provider with the Provider Agent software. The service provider installs this software.

3.4.2 Define policies

When a client wants to use a specific type of service, it defines policies for the use of that service. These policies were described in Section 3.1.

3.4.3 Find services and create SLA

The Negotiator finds a service provider that meets client needs based on the service selection policy defined by the client. The SLA is then made between the client and the provider to to allow the client to use the service of the provider. The steps are described as follows.

Finding service provider

The Negotiator of the TPA uses the client defined selection policies to find an appropriate service in the service directory of TPA. Algorithm 1 describes the algorithm for finding an appropriate service. First, all active services of the desired service type are retrieved from the service directory (line 1). The service time threshold is set to the lower bound limit of service time defined in the service selection policy (line 3), the availability threshold is set to the upper bound limit of availability defined in the service selection policy (line 4) and the price threshold is set to the lower bound limit of price defined in service selection policy (line 5). A search of the list of service instances is carried out to find a service where the service time of the service is less than the service time threshold (line 11), the availability of the service is greater than the availability threshold (line 12) and the price of the service is less than the price threshold (line 13). If an appropriate service is found then that service is selected (line 19). If there found several appropriate service, then one service will be selected among them based on the priority defined by the client (line 17). If no service is found then each of the thresholds are increased by the change rate defined in the policy (line 21, 22, 23) and the list is searched again. This search will continue until the service time threshold reaches the upper bound limit, the availability threshold reaches the lower bound limit and the price threshold reaches the upper bound limit defined in the policy (line 8). If there is no appropriate service then no service will be selected.

Algorithm 1 Find Appropriate Service

```
1: $ServiceList \leftarrow$ Get all services of the service type defined in the policy
2:
3: for all services of ServiceList do
4:    $ServiceTimeThreshold \leftarrow PolicyDefinedServicetimeLowerbound$
5:    $AvailabilityThreshold \leftarrow PolicyDefinedAvailabilityUpperbound$
6:    $PriceThreshold \leftarrow PolicyDefinedPriceLowerbound$
7:
8:    while ServiceTimeThreshold != PolicyDefinedServicetimeUpperbound AND
      AvailabilityThreshold != PolicyDefinedAvailabilityLowerbound AND
      priceThreshold != PolicyDefinedPriceUpperbound do
9:
10:      Find a service from ServiceList where
11:      $ServiceTimeThreshold \geq ServiceTime$ AND
12:      $AvailabilityThreshold \leq Availability$ AND
13:      $PriceThreshold \geq Price$
14:
15:      if found then
16:         if found several services then
17:            select one service among them based on the priority defined by the client
18:         end if
19:         return the service as proper service
20:      else
21:         $ServiceTimeThreshold \leftarrow ServiceTimeThreshold + ServiceTimeChangeRate$
22:         $AvailabilityThreshold \leftarrow AvailabilityThreshold + AvailabilityChangeRate$
23:         $PriceThreshold \leftarrow PriceThreshold + PriceChangeRate$
24:      end if
25:   end while
26: end for
```

Create SLA

After service provider selection, the Negotiator creates an SLA between the service provider and the client. This SLA should include the conditions of the service usage. Service usage can be defined in one of the following ways:

- **Time basis:** The SLA specifies that the client uses the service for a fixed length of time (i.e. 15 days, 30 days). The SLA specifies the amount of payment to be paid for that time period.

- **Quota basis:** The SLA specifies that the client uses the service for a fixed number of requests (i.e. 1000 service requests). The SLA specifies the one time pre-paid amount to be paid for that number of service invocations. The contract should be renewed when the quota is reached.

- **Every service invocation basis:** The SLA specifies that the client uses the service and the client has to pay a fixed amount for each service request. A bill is issued to the client periodically (e.g. by-weekly/monthly) for using the service during that period.

3.4.4 Recording service request information

Each service request and response of the client is intercepted by the Client Agent. The service invocations are stored in the Logs data storage component by the Collector module of the Client Agent. The following occurs after a request is invoked by a client.

- When a client invokes a service request it is intercepted by the Client Agent.

- After the Client Agent intercepts the request, it retrieves the required information from the Configuration Data Storage component. This is used to call the Provider Agent of the service being requested. The Client Agent then sends the service request to the Provider Agent.

- The Provider Agent processes the incoming service request to determine the service to be invoked. It then forwards the service request to that server which is serving the requested service.

- The provider generates a response for the request. It then sends back that response to the client which invoked the request. The Provider Agent intercepts the response message.

- After intercepting the response, the Provider Agent appends the calculated service time (time that is taken by the server to process the service request) to the response and sends the modified response to proper client.

- The Client Agent intercepts the response message modified by the Provider Agent, parses the response message to get the QoS data (service time, availability) of that invocation. It then stores this information along with other information (contract identifier, port number, operation address, processing time etc.) in the Logs data storage component. It then forwards the response to the client.

3.4.5 Synchronize logs of service invocations

The Client Agent uses the Synchronizer component for updating the Logs Data Storage component of the TPA with the data from the Client Agent's Logs Data Storage component. The Synchronizer runs at fixed time intervals and checks the Logs data storage component of the Client Agent to keep track of appearance of new data. If it finds any new data, it uploads the new data to Logs data storage component of TPA.

3.4.6 Monitoring

The Event Generator component of TPA runs in fixed time intervals and monitors the data in the Logs data storage component of TPA to generate events representing SLA violations.

Algorithm 2 describes the generation of service time violation events. At first, all the logs which have not yet been processed for monitoring are retrieved (line 1). For each log entry the service time is evaluated to determine if there is a SLA violation (line 4). If there is a violation then the count of evaluations is compared to a threshold value specified in a SLA violation policy (line 5). If the count exceeds the threshold then an event indicating an SLA violation is generated (line 6). Otherwise the counter is increased by one (line 8).

Algorithm 2 Service time Violation event generation

```
1: $LogList \leftarrow$ Get all logs for which monitoring is not done
2:
3: for Every log of LogList do
4:    if $ReceivedServiceTime > ExpectedServiceTime$ then
5:       if $ServiceTimeViolationLimit > PolicyDefinedServiceTimeViolationLimit$ then
6:          Generate Service time Violation event
7:       else
8:          $ServiceTimeViolationLimit \leftarrow ServiceTimeViolationLimit + 1$
9:       end if
10:    end if
11: end for
```

Algorithm 3 describes the generation of availability violation events. At first, all the logs which have not yet processed for monitoring are retrieved (line 1). For each log entry if it is found that the service is unavailable (line 4), then current availability rate is evaluated to determine if there is a SLA violation (line 6). If there is a violation then the count of evaluations is compared to a threshold value specified in a SLA violation policy (line 7). If the count exceeds the threshold then an event indicating an SLA violation is generated (line 8). Otherwise the counter is increased by one (line 10).

Algorithm 3 Availability Violation event generation

```
1: $LogList \leftarrow$ Get all logs for which monitoring is not done
2:
3: for Every log of LogList do
4:    if Service unavailable for current invocation then
5:       Calculate $CurrentAvailabilityRate$
6:       if $CurrentAvailabilityRate < ExpectedAvailabilityRate$ then
7:          if $PolicyDefinedAvailabilityViolationLimit > AvailabilityViolationLimit$ then
8:             Generate Availability Violation event
9:          else
10:             $AvailabilityViolationLimit \leftarrow AvailabilityViolationLimit + 1$
11:          end if
12:       end if
13:    end if
14: end for
```

3.4.7 Recovery

The Diagnosis module analyses events to determine root cause. The result of Diagnosis modules is the input of the Recovery Agent. The Recovery Agent makes a decision on an action in reaction to the SLA violation. Changing the current provider of the client to a new one is one possible action. Another recovery action is to do nothing.

Chapter 4

Implementation

This chapter describes the implemented prototype of our proposed architecture.

4.1 Implementation of system components

We have implemented the Client Agent, the Provider Agent and the Third Party Agent. The implementation details of these components are described in this section.

4.1.1 Client Agent

The Client Agent has two processing components (Collector and Synchronizer) and two data storage components (Configuration and Logs). Implementation details of these components are described in this section.

Configuration

A text file is used to store the configuration information for the Client Agent. Each line of the Configuration text file represents information needed to invoke a specific service instance. Each line consists of the service type, its corresponding policy identifier of the group of policies (service selection policy, violation policy and recovery policy), current SLA identifier, host address of the Provider Agent, port number on which the Provider Agent is listening to requests, and the operation address of the service of the provider.

Collector

The Collector is a proxy and so we needed to build a client-server mechanism to implement the Collector. The Collector is a client of a Provider Agent and it is a server for a client. Communication is done through the Apache HttpClient API [3]. After the Collector receives the response from Provider Agent, it writes information related to the service request into Logs by using a file handler of Java [20].

Logs

Logs is a text file where information for each service request is stored by the Collector. Each line of the Logs text file contains information about a single service request.

Synchronizer

The Synchronizer is implemented using Java Thread Programming [21]. The Client Agent starts the Synchronizer which is a thread. The thread checks the Logs on a certain interval for new data. If there is any new data in Logs the data is uploaded to the Logs component of the Third Party Agent. JDBC API [19] is used to upload new data at Logs of Third Party Agent.

4.1.2 Provider Agent

The Provider Agent is a proxy and so we built a client-server mechanism to implement the Provider Agent. The Provider Agent is a client of a provider and it is a server for the Client Agent. Communication is done through the Apache HttpClient API [3]. After the Provider Agent receives the response from the provider it appends the calculated service time to the response and forwards it to the client.

4.1.3 Third Party Agent

The implementation details of the third party agent are described in this section.

Registration and Negotiator

We have implemented a web interface by which clients and providers register with the system. The client defines policies through this web interface. The provider also use this web interface to provide information about its services.

Contract Repository and Logs

We used MySQL database [17] to implement the Contract Repository and the Logs data storage components. We created a table named *slas* to store the contracts and a table named *logs* where each entry represents information about a service request.

Event Generator

The Event Generator is implemented using Java Thread Programming [21]. The TPA starts the Event Generator which is a thread. The thread evaluates new data from Logs based on the SLAs of Contract Repository in a fixed time interval and generates events if any violation occurs. To store the events we created a table named *events*.

Diagnosis

We did not implement this module. This is future work.

Recovery Agents

The Recovery Agent is implemented using Java Thread Programming [21]. The TPA starts the Recovery Agent which is a thread. The thread analyses the events and takes necessary recovery actions to recover the SLA violation. The actions can be changing the current provider of the client to a new one or to do nothing.

4.2 Database

A MySQL database [17] is used by the Third party to store different information (i.e. contracts, logs, events, client and provider informations etc.). For our prototype, we have ten tables. The *clients* and *providers* tables are used to store information of clients and providers. The services of providers are stored in a *services* table. Clients define policies that are saved in the *policies* table. The *service types* table represents different types of services supported by the third party. The *violation types* table stores different violation types (e.g. service time violation, availability violation) which could be evaluated by the system. When a SLA is violated, the provider is blacklisted for that client. This information is stored in the *blacklisted services* table. The *slas* table represents the contracts made between the clients and the providers. The *logs* table is synchronized with the Logs data storage component of the Client Agent continuously. Events which represent SLA violations are saved in the *events* table. The relationship among these tables is shown in Figure 4.1.

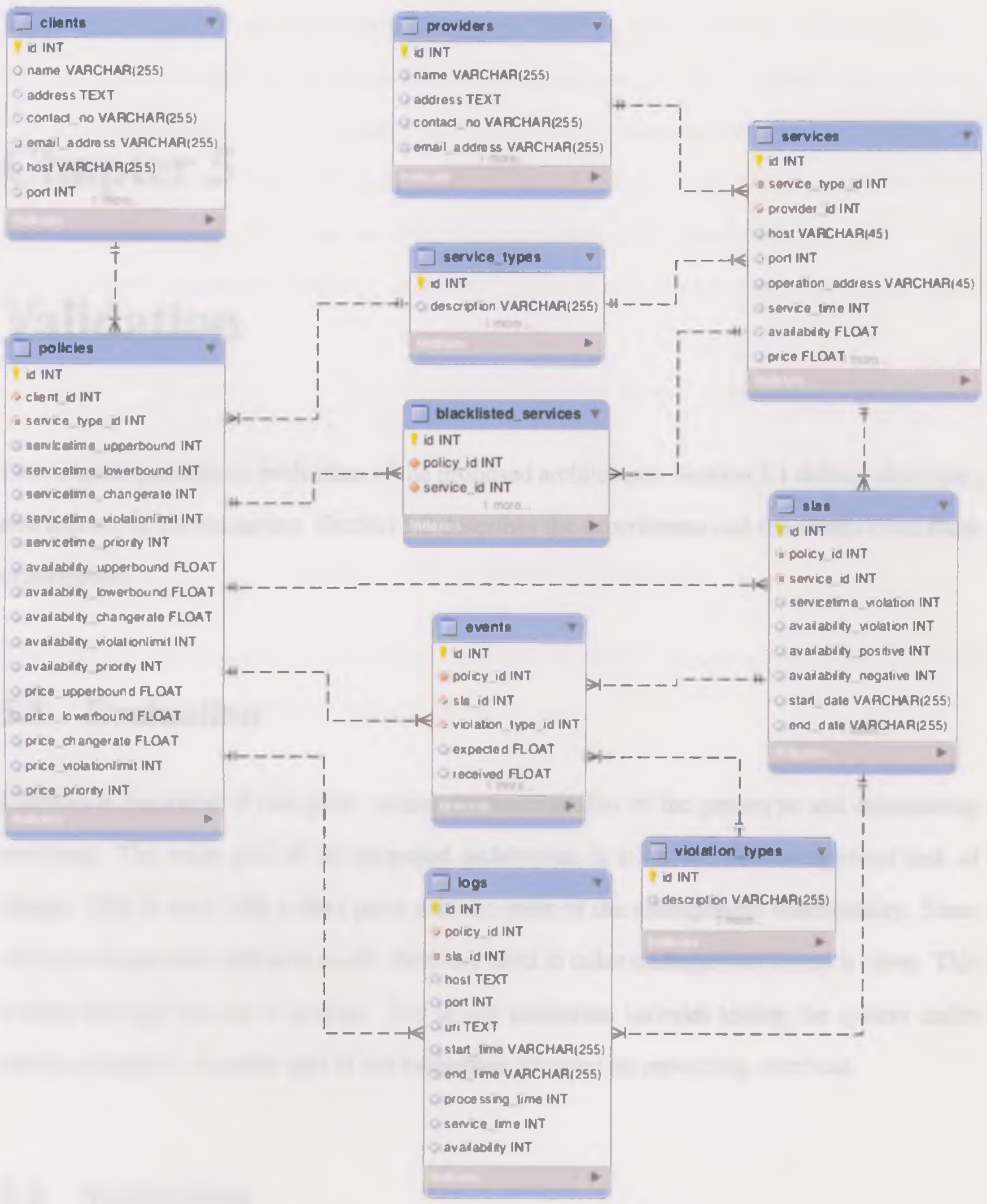

Figure 4.1: Database of prototype

Chapter 5

Validation

This chapter presents an evaluation of the proposed architecture. Section 5.1 defines objectives and metrics of the evaluation. Section 5.2 describes the experiments and the results from these experiments.

5.1 Evaluation

Evaluation consisted of two parts: testing the functionality of the prototype and determining overhead. The main goal of the proposed architecture is to reduce the management task of clients. This is done with a third party that has some of the management functionality. Since different clients have different needs, there is a need to tailor management for each client. This is done through the use of polices. Part of our evaluation includes testing the system under various scenarios. Another part of our evaluation focusses on measuring overhead.

5.2 Validation

The experiments, made to validate our proposed architecture are described in detail in this section.

5.2.1 Functionality testing

For testing we used six machines with centOS 5.3 operating system running on those machines. We used three machines as providers (alfa10.syslab.csd.uwo.ca, alfa11.syslab.csd.uwo.ca and alfa12.syslab.csd.uwo.ca), two machines as clients (alfa05.syslab.csd.uwo.ca, alfa06.syslab.csd.uwo.ca) and one machine as a third party (alfa01.syslab.csd.uwo.ca). Clients and providers are registered with the system. For a specific type of service (subtraction) each of the providers placed their services in the service directory of the third party. The parameters defined by the providers are as follows:

Provider	Service Type	Service Time	Availability	Price
alfa10.syslab.csd.uwo.ca	subtraction	2200	0.75	25
alfa11.syslab.csd.uwo.ca	subtraction	3100	0.8	20
alfa12.syslab.csd.uwo.ca	subtraction	2750	0.7	40

Table 5.1: Service directory of third party

After that one client (alfa05.syslab.csd.uwo.ca) defined policies for using "subtraction" type of service. The policies defined by the client were as follows:

```
POLICY selection-subtraction {
        ServiceType (subtraction),
        QoSParameters {
                (ServiceTime, 2000, 3100, 100, 1),
                (Availability, 0.7, 0.9, 0.05, 3),
                (Cost, 20, 90, 10, 2),
        }
}
```

```
POLICY violation-subtraction {
        ServiceType (subtraction),
        QoSParameters {
                (ServiceTime, 5),
                (Availability, 5),
        }
}

POLICY recovery-subtraction {
        ServiceType (subtraction),
        QoSParameters {
                (ServiceTimeViolation, changeProvider),
                (AvailabilityViolation, changeProvider),
        }
}
```

The other client (alfa06.syslab.csd.uwo.ca) also defined policies for using the "subtraction" type of service. The policies defined by that client were as follows:

```
POLICY selection-subtraction {
        ServiceType (subtraction),
        QoSParameters {
                (ServiceTime, 2100, 3200, 100, 2 ),
                (Availability, 0.65, 0.85, 0.05, 1),
                (Cost, 15, 50, 10, 3),
        }
}
```

```
POLICY violation-subtraction {
          ServiceType (subtraction),
          QoSParameters {
                        (ServiceTime, 4),
                        (Availability, 3),
          }
}


POLICY recovery-subtraction {
          ServiceType (subtraction),
          QoSParameters {
                        (ServiceTimeViolation, changeProvider),
                        (AvailabilityViolation, changeProvider),
          }
}
```

Within this environment all aspects of the system were tested to provide confidence in the correctness of the implementation.

Testing involved using policies with very low threshold values. This allowed for a simulation of violations. Figures 5.1 and 5.2 illustrate for one test what happens when service time and availability violations occur respectively. The client was never informed about the provider change. This experiment validated that our implemented prototype reduces the management task for the client. Also the prototype took all the decisions based on the policies defined by the client which validates our proposed architecture.

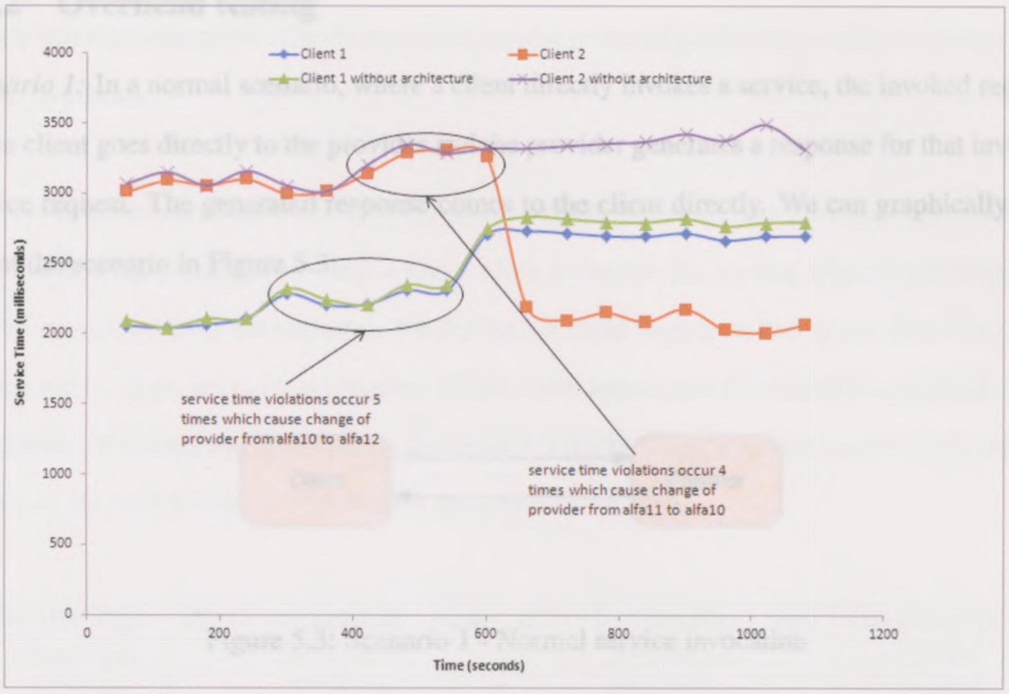

Figure 5.1: Service time violations

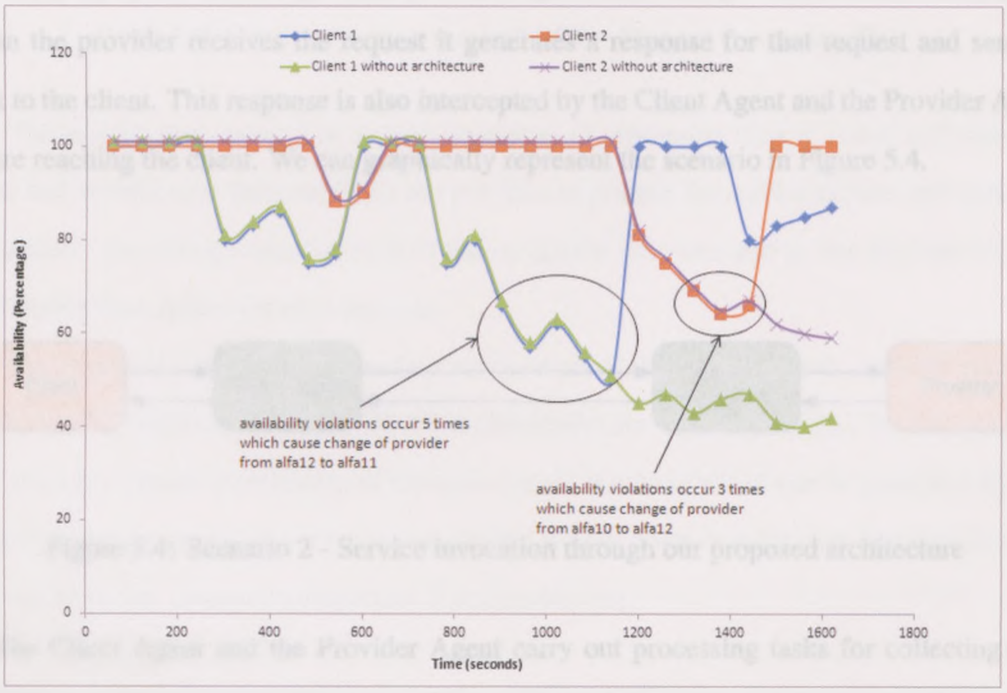

Figure 5.2: Availability violations

5.2.2 Overhead testing

Scenario 1: In a normal scenario, where a client directly invokes a service, the invoked request of the client goes directly to the provider and the provider generates a response for that invoked service request. The generated response comes to the client directly. We can graphically represent the scenario in Figure 5.3.

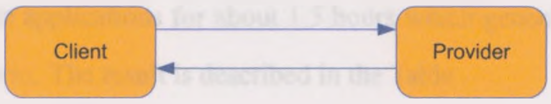

Figure 5.3: Scenario 1 - Normal service invocation

Scenario 2: In our architecture, when the client invokes a service request the service request is intercepted by our Client Agent and Provider Agent before the provider receives the request. When the provider receives the request it generates a response for that request and sends it back to the client. This response is also intercepted by the Client Agent and the Provider Agent before reaching the client. We can graphically represent the scenario in Figure 5.4.

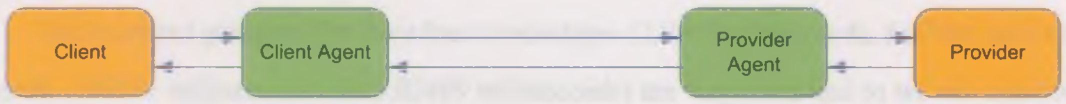

Figure 5.4: Scenario 2 - Service invocation through our proposed architecture

The Client Agent and the Provider Agent carry out processing tasks for collecting data which is used for management purpose. The metric used for measuring the overhead is request processing time. This is the time between the the client invoking a service request and the time that it receives a response. The difference in request processing time between the two scenarios is the overhead.

We developed a client application (OverheadTester) which invokes a service request on every second. The application starts a timer when it invokes the service request and stops the timer when it receives the response. We did the overhead testing on four types of services. On every service type, we ran two instances of this client application (OverheadTester) for the two scenarios. We ran those applications for about 1.5 hours which generates about 5000 service requests for each scenario. The result is described in the Table .

Service Type	**RPT* in Scenario 1 (ms)**	**RPT* in Scenario 2 (ms)**	**Overhead (ms)**
subtraction	203.61124	206.54614	2.9349
addition	323.6219	326.62256	3.00066
multiplication	483.53406	486.56967	3.03561
getPolicies	934.81173	937.83672	3.02499

Table 5.2: Overhead testing result (* RPT = Request Processing Time)

The request processing time is the summation of processing time at client end, network time and service time (time taken by the provider to process the service request and generate response). The service time for each of the services is not same and so the resultant request processing time differs for each services.

The overhead generated for these four service types (*2.9349* milliseconds, *3.00066* milliseconds, *3.03561* milliseconds and *3.02499* milliseconds) are very close and so we can consider the average of these experiments as the approximate overhead which can be generated by our implemented prototype. From these experiments we can come to a decision that the overhead generated by our proposed architecture is reasonably less.

Chapter 6

Conclusions and Future work

This thesis relates to the area of web service management and the focus is on design and implementation of a policy based third party architecture for managing web services. Section 6.1 presents the conclusions. Section 6.3 presents possible future work.

6.1 Conclusions

Our proposed policy based third party management system is used to dynamically manage the use of web services. Policies are used to guide decision making by the third party system. Having the clients provide the policies allows for the third party to base its decisions that is client specific.

The architecture is flexible. Management tasks such as selecting a service and determining the action in response to SLA violations are guided by the clients by using policies from the clients. Currently we use three kind of policies (service selection policy, violation policy and recovery policy) but the architecture is flexible enough that it is possible to introduce a new type of policy. The architecture supports the change of policies at run-time which allows the clients to change their requirements in run-time. This feature is validated with functionality testing using the environment presented in chapter 5.

Our architecture is modular. Each module gets input from a specific module and the generated output of the module goes as input to another specific module. This kind of behaviour makes each module loosely coupled but highly cohesive. It makes the architecture easy to enhance. Each module can be outsourced or separately built and can be used as a plug-in of the architecture

We successfully reduced the management task to be carried out by clients. Most of the management tasks are carried out by the third party. The client tells the third party which type of service it is looking for. It is the third party that selects a service for the client based on the requirements of the client. The third party monitors for SLA violations and changes services as a recovery action to recover SLA violations. By using the third party the client does not need to concern itself with managing the service quality.

We validated the functionality of our architecture by implementing a prototype for the architecture and carrying out experiments for several scenarios. We also showed that our approach generates minimal overhead.

6.2 Contributions

By using our proposed architecture the complexity of management and service discovery can be hidden from clients and providers by outsourcing this functionality to third party. We have used policies in a third party management system so that decision making tasks of the third party can be automated and this automation is customized for each clients. It helps the third party to respond promptly. Client gets uninterrupted service usage while SLA gets violated as new SLA is built instantly based on previously defined policies of client.

6.3 Future Work

There is a good deal of room for improvement in several modules. The algorithms for selecting a service and checking for violations are very simple with support for a few conditions. These algorithms can be made more complex and support more conditions. Several more parameters (throughput, accuracy, authentication) can be introduced to increase the performance of these algorithms.

With the increased number of clients and providers in the system the log generation rate will increase too. It will also increase the workload of the event generator. In that case to increase the performance of the event generator we can implement a distributed event generator. A distributed event generator will process multiple logs in parallel and thus increase the performance of event generator

In our implemented prototype, we did not implement the diagnosis module. The task of the diagnosis module is to analyse events and find out the root cause of a SLA violation. We leave this for future work.

Bibliography

[1] Oracle and/or its affiliates. The java ee 6 tutorial - types of web services. [Online; accessed 24-August-2010].

[2] Mira Mezini Anis Charfi, Rainer Berbner and Ralf Steinmetz. On the management requirements of web service compositions. In Monique Calisti, Marius Walliser, Stefan Brantschen, Marc Herbstritt, Thomas Gschwind, and Cesare Pautasso, editors, *Emerging Web Services Technology, Volume II*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 97–109. Birkhuser Basel, 2008.

[3] Apache. Httpclient overview. [Online; accessed 24-February-2011].

[4] R. Ben Halima, K. Drira, and M. Jmaiel. A qos-oriented reconfigurable middleware for self- healing web services. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 104 –111, 2008.

[5] Boualem Benatallah, Marlon Dumas, and Zakaria Maamar. Definition and execution of composite web services: The self-serv project. *Data Engineering Bulletin*, 25:2002, 2002.

[6] World Wide Web Consortium. Soap. [Online; accessed 27-October-2010].

[7] World Wide Web Consortium. Web services choreography description language version 1.0. [Online; accessed 2-November-2010].

[8] M. Fugini E. Mussi B. Pernici P. Plebani D. Ardagna, C. Cappiello. Faults and recovery actions for self-healing web services. *15th Int. World Wide Web Conf.*, 2006.

[9] A. Erradi, P. Maheshwari, and V. Tosic. Recovery policies for enhancing web services reliability. In *Web Services, 2006. ICWS '06. International Conference on*, pages 189 –196, 2006.

[10] W. He. Recovery in web service applications. In *e-Technology, e-Commerce and e-Service, 2004. EEE '04. 2004 IEEE International Conference on*, pages 25 – 28, 2004.

[11] IBM. Types of web services. [Online; accessed 28-October-2010].

[12] IEEE. Ieee standard glossary of software engineering terminology. [Online; accessed 9-December-2010].

[13] Paul Lipton. Composition and management of web services. [Online; accessed 29-November-2010].

[14] E.C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *Software Engineering, IEEE Transactions on*, 25(6):852 –869, 1999.

[15] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive qos monitoring of web services and event-based sla violation detection. In *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, MWSOC '09, pages 1–6, New York, NY, USA, 2009. ACM.

[16] Microsoft Corporation MSDN. Understanding service-oriented architecture. [Online; accessed 27-October-2010].

[17] MySQL. Mysql:the world's most popular open source database. [Online; accessed 24-February-2011].

[18] Anders Mller and Michael Schwartzbach. Wsdl example. [Online; accessed 29-November-2010].

[19] Oracle. Jdbc overview. [Online; accessed 24-February-2011].

[20] Oracle. Lesson: Basic i/o. [Online; accessed 24-February-2011].

[21] Oracle. Thread. [Online; accessed 24-February-2011].

[22] Inc. O'Reilly Media. An introduction to service-oriented architecture from a java developer perspective. [Online; accessed 22-November-2010].

[23] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40:38–45, 2007.

[24] Tan Phan, Jun Han, J.-G. Schneider, T. Ebringer, and T. Rogers. A survey of policy-based management approaches for service oriented systems. pages 392 –401, mar. 2008.

[25] Stephan Reiff-Marganiec and Kenneth J. Turner. Use of logic to describe enhanced communications services. In *Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, FORTE '02, pages 130–145, London, UK, 2002. Springer-Verlag.

[26] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad P. A. van Moorsel, and Fabio Casati. Automated sla monitoring for web services. In *DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 28–41, London, UK, 2002. Springer-Verlag.

[27] Josef Spillner and Jan Hoyer. Sla-driven service marketplace monitoring with grand slam. In Boris Shishkov, Jos Cordeiro, and Alpesh Ranchordas, editors, *ICSOFT (2)*, pages 71–74. INSTICC Press, 2009.

[28] Josef Spillner, Matthias Winkler, Sandro Reichert, Jorge Cardoso, and Alexander Schill. Distributed contracting and monitoring in the internet of services. In *DAIS '09: Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, pages 129–142, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] Peter Trger, Harald Meyer, Ingo Melzer, and Markus Flehmig. Dynamic provisioning and monitoring of stateful services. In *Proceedings of the 3rd International Conference on Web Information Systems and Technology (WEBIST 2007)*, pages 434–438, Setbal, Portugal, 2007. INSTICC.

[30] Dr. Do van Than. Web service orchestration: An open and standardised approach for creating advanced services. [Online; accessed 29-November-2010].

[31] W3Schools. Introduction to web services. [Online; accessed 24-August-2010].

[32] W3Schools. Soap example. [Online; accessed 29-November-2010].

[33] W3Schools. Soap tutorial. [Online; accessed 29-November-2010].

[34] W3Schools. Wsdl and uddi. [Online; accessed 22-November-2010].

[35] W3Schools. Wsdl tutorial. [Online; accessed 29-November-2010].

[36] Wikipedia. Business process execution language. [Online; accessed 2-November-2010].

[37] Wikipedia. Orchestration (computing). [Online; accessed 29-November-2010].

[38] Wikipedia. Web service choreography. [Online; accessed 29-November-2010].

[39] Farhana H. Zulkernine, Patrick Martin, and Kirk Wilson. A middleware solution to monitoring composite web services-based processes. In *SERVICES-2 '08: Proceedings of the 2008 IEEE Congress on Services Part II*, pages 149–156, Washington, DC, USA, 2008. IEEE Computer Society.