Performance Measurement and Modeling of BPEL Orchestrations

by

Jun Li

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

In

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

© 2013
Jun Li
ABSTRACT

Service-oriented architecture (SOA) is a design approach for the development, integration, and maintenance of complex enterprise information systems composed of loosely-coupled services. Runtime services composition introduces performance overheads. Software developers may have difficulties in assessing the performance costs of the final composite product. One objective of this thesis is to design and implement a unified logging facility, which allows us to instrument different layers of a SOA system under study, to automatically collect measurement data and to correlate the separate logs from sub-systems and components into a unified system view. Next, the logging facility is used to collect performance measurement data for a case study system. The measurements are collected separately for the application code and for the underlying service engine. The measured performance data is used for building a Layered Queuing Network model of the case study system and to validate it against measurements.
Table of Contents

ABSTRACT .................................................................................................................... II
TABLE OF CONTENTS ..................................................................................................... III
ACRONYMS .................................................................................................................. V
LIST OF FIGURES ......................................................................................................... VI
LIST OF TABLES ............................................................................................................ VIII

1 INTRODUCTION ..................................................................................................... 1
  1.1 MOTIVATION, PROBLEM STATEMENT AND OBJECTIVES .................................. 1
  1.1.1 Motivation ....................................................................................................... 1
  1.1.2 Problem Statement ....................................................................................... 4
  1.1.3 Thesis Objectives .......................................................................................... 4
  1.2 PROPOSED APPROACH ................................................................................... 6
  1.3 THESIS CONTRIBUTIONS ................................................................................ 8
  1.4 THESIS CONTENTS ........................................................................................... 9

2 BACKGROUND ..................................................................................................... 10
  2.1 SOA AND WEB SERVICES ............................................................................. 10
  2.1.1 Service Orchestration ............................................................................... 11
  2.1.2 Web Service ................................................................................................ 11
  2.1.3 WSDL .......................................................................................................... 12
  2.1.4 SOAP ............................................................................................................ 14
  2.1.5 BPEL ............................................................................................................ 15
  2.1.6 BPEL Engine ............................................................................................... 19
  2.1.7 BizTalk Server ............................................................................................ 20
  2.1.8 Difference between BizTalk and BPEL standard ....................................... 27
  2.2 SOFTWARE PERFORMANCE ........................................................................ 28
  2.2.1 Software Performance Engineering (SPE) .................................................. 29
  2.3 PERFORMANCE MODELS .......................................................................... 32
  2.4 TOOLS FOR BPEL ORCHESTRATION DEVELOPMENT AND TESTING ....... 34
  2.4.1 Microsoft BizTalk server 2006 .................................................................. 34
  2.4.2 Orchestration Design Tool .......................................................................... 34
  2.4.3 SOAP_UI ..................................................................................................... 35
  2.4.4 LOAD_UI ..................................................................................................... 36

3 DESIGN AND IMPLEMENTATION OF CASE-STUDY SYSTEM .................. 37
  3.1 BUSINESS LOGIC .......................................................................................... 37
  3.2 INDIVIDUAL ELEMENTS OF THE APPLICATION ............................................ 39
  3.3 APPLICATION AND PLATFORM OVERHEADS ............................................. 41
  3.4 PERFORMANCE TESTING FACILITIES FOR A SOA SYSTEM .................. 43

4 PERFORMANCE MEASUREMENTS ................................................................. 46
  4.1 INTRODUCTION ............................................................................................. 46
  4.2 TEST STRATEGY FOR THESIS PROJECT ..................................................... 48
  4.2.1 Custom logging facilities footprint .............................................................. 48
  4.2.2 Performance factors ................................................................................... 48
  4.2.3 Logging parameters ................................................................................... 49
<table>
<thead>
<tr>
<th>ACRONYMS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed Component Object Model</td>
</tr>
<tr>
<td>EAI</td>
<td>Enterprise Application Integration</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Bean</td>
</tr>
<tr>
<td>ESB</td>
<td>Enterprise Service Bus</td>
</tr>
<tr>
<td>HAT</td>
<td>Health and Activity Tracking</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IIS</td>
<td>Internet Information Server</td>
</tr>
<tr>
<td>LQN</td>
<td>Layered Queuing Network</td>
</tr>
<tr>
<td>OASIS</td>
<td>Organization for the Advancement of Structured Information Standards</td>
</tr>
<tr>
<td>QN</td>
<td>Queuing Network</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SPE</td>
<td>Software Performance Engineering</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator URL</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Service Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1-1 BPEL Orchestration ................................................................. 2
Figure 1-2 BPEL Choreography ................................................................ 3
Figure 2-1 Web Services Architecture ....................................................... 12
Figure 2-2 WSDL concepts ...................................................................... 13
Figure 2-3 SOAP Protocol ......................................................................... 14
Figure 2-4 SOAP message structure ......................................................... 15
Figure 2-5 Architecture of BizTalk Server ................................................ 21
Figure 2-6 Receive Port contains Receive Locations .................................... 23
Figure 2-7 Receive Location Configuration ............................................... 24
Figure 2-8 Distinguished Fields ............................................................... 26
Figure 2-9 Promote Distinguished Fields and Properties Fields ............... 26
Figure 2-10 The SPE Workflow (Smith, 2001) .......................................... 31
Figure 2-11 Visual Studio Orchestration Tool ........................................... 35
Figure 2-12 Orchestration test example in SOAP-UI ................................. 36
Figure 3-1 Use Case Diagram of Insurance Broker Service System ......... 38
Figure 3-2 Activity Diagram of Business Logic ........................................ 38
Figure 3-3 Insurance Quotations BPEL Orchestration .............................. 40
Figure 3-4 Activity Diagram of Composed Service with consideration of Overheads .... 42
Figure 4-1 Insurance Broker Orchestration Service Proxy ....................... 46
Figure 4-2 BizTalk Web Services Publishing Wizard ................................ 47
Figure 4-3 Multi-Tier of Insurance Broker SOA service ......................... 52
Figure 4-4 Monitoring spots labeled (0) to (28) .......................................... 53
Figure 4-5 SoapUI test cases ................................................................. 54
Figure 4-6 Logging information from BizTalk HAT tool ............................ 55
Figure 4-7 Logging information from SQL Profiler tool ........................... 55
Figure 4-8 Use LoadUI to do auto and concurrent testing ....................... 57
Figure 4-9 Combine four log files into one log file .................................. 57
Figure 4-10 Import data from log file to T3200 table ................................. 58
Figure 4-11 Data structure for collecting the sample data .......................... 60
Figure 4-12 BPEL Receive Request sample mean distribution .................. 66
Figure 4-13 Manu Life WS Database Execution Distribution .................. 68
Figure 4-14 ManuLife WS Database Disk Access Distribution ............... 69
Figure 4-15 Insurance Quotations BPEL Orchestration ............................ 70
Figure 4-16 SunLife WS Database Disk Access Distribution .................... 71
Figure 4-17 Manu Life BPEL Build WS Request XML Distribution ........... 72
Figure 4-18 Quantifying the time method accuracy .................................... 74
Figure 5-1 MARTE annotated Activity Diagram ....................................... 80
Figure 5-2 LQN model ........................................................................... 82
Figure 5-3 Setting Frequency for Concurrent Testing .............................. 85
Figure 5-4 Comparison of LQN results and Actual measurement for concurrent testing 86
Figure 5-5 Comparison of LQN model prediction and Performance Measurement for one user and different message sizes ................................................. 87
Figure 5-6 LQN Response time in function of the number of users for Case A ........ 88
Figure 5-7 Task and process utilization for Case A .................................... 89
Figure 5-8 LQN Response Time for Case B................................................................. 90
Figure 5-9 Task Utilization for Case B....................................................................... 90
Figure 5-10 Processor utilization for Case B............................................................... 91
Figure 5-11. Response time for Case C compared to Case B........................................ 92
Figure 5-12 Processor utilization for Case C............................................................... 92
List of Tables

Table 2-1 BPEL Invoke element ................................................................. 17
Table 2-2 BPEL receive element ............................................................... 17
Table 2-3 BPEL reply element ................................................................. 17
Table 2-4 BPEL assign element ............................................................... 18
Table 2-5 SOA Suites from different vendors ........................................... 19
Table 2-6 BPEL Partner Links in Orchestration Designer ............................ 22
Table 4-1 Definition of custom performance logging parameters .............. 49
Table 4-2 Sample performance sample data in a log file ......................... 55
Table 4-3 Import raw sample test data into T3200 table ......................... 59
Table 4-4 Import data from T3200 to a strong name Experiment1 table .......... 59
Table 4-5 Response Time Calculation from each execution step ............... 60
Table 4-6 Confidence Interval Statistics for Orchestration Total Response Time 63
Table 4-7 Performance measurements for Web Server ............................ 65
Table 4-8 Statistics of BPEL Receive Activity ......................................... 66
Table 4-9 Distribution for BPEL Receive Activity .................................... 66
Table 4-10 Performance Measurement for Manu Life Web Service .......... 67
Table 4-11 Statistics for Manu Life WS Database Execution Activity ........ 67
Table 4-12 Statistics for Manu Life WS Database Execution Activity ........ 68
Table 4-13 Statistics for Manu Life WS Database Disk Access ............... 68
Table 4-14 Distribution for Manu Life WS Database Disk Access ............. 69
Table 4-15 Performance Measurement for Sun Life Web service ............. 69
Table 4-16 Statistics of Sun Life WS Database Execution .................... 70
Table 4-17 Distribution of Sun Life WS Database Execution ................. 70
Table 4-18 Statistics of Sun Life WS Database Execution .................... 71
Table 4-19 Distribution of Sun Life WS Database Disk Access ............... 71
Table 4-20 Statistics for Manu Life BPEL Build Activity ...................... 72
Table 4-21 Distribution of Manu Life BPEL Build Activity .................... 72
Table 4-22 BizTalk BPEL Engine Platform related Overheads versus message size ..... 75
Table 5-1 Service Time for each step in BPEL Orchestration .................. 83
Table 5-2 Comparison of LQN model prediction and Performance Measurement for one user and different message sizes ........................................... 86
1 INTRODUCTION

1.1 Motivation, Problem Statement and Objectives

1.1.1 Motivation

According to (Kruchten, 2004), software has four key differentiating characteristics as opposed to hardware:

• absence of a fundamental theory;
• ease of change;
• rapid evolution of technologies;
• low manufacturing costs.

After generations of software evolution, modern enterprise information systems are heterogeneous, containing a range of different systems, applications, technologies, and architectures, so their integration is crucial. In the absence of a fundamental theory for software, different methodologies have emerged trying to solve these issues. Service-oriented architecture (SOA) is an architectural design approach concerned with the development, integration, and maintenance of complex enterprise information systems built by service composition. SOA is based on loosely coupled services as the fundamental units for building enterprise solutions that enable business flexibility in an interoperable manner. SOA applications realize business processes or workflows by discovering, selecting and invoking a set of services at runtime, using interface-based service description. (Borges et al., 2004). A business process is a series of coordinated service invocations and related activities that produce a business result, either within a single organization or across several organizations. SOA is an architectural style, based on services as the fundamental unit for constructing enterprise solutions from services.
There are two main approaches for composing services: orchestration and choreography. In the case of orchestration, a central or master element controls all aspects of the process, while in choreography each element is autonomous and controls its own agenda. The two terms come from ballet performance metaphors. The first compares the interworking of services with an orchestra, where the conductor provides master control, sets the pace and signals when different instruments have to join in. The second compares services with dance: each dancer knows their individual role and interacts with the other dancers at the tempo set by the music. The ballet emerges from the overall action of all the dancers, rather than from some central control (Rosen, 2008).

As shown in Figure 1-1, orchestration either receives requests from web service and sends back replies or directly invokes web services where it takes control and coordinates the execution of different web services operations involved (Matjaz, 2006). In this thesis, we study the performance of service orchestrations.

![Figure 1-1 BPEL Orchestration](image)

On the other hand, choreography does not rely on a central coordinator. Rather, each web service involved in the choreography knows exactly when to execute its operations and how to interact with others (Matjaz, 2006). As shown in Figure 1-2, each participant
web service knows when and which web service to invoke based on exchange of messages.

![Figure 1-2 BPEL Choreography](image)

For general adoption, business process automation solutions required a standard foundation and language for composing services into business processes. Such a language is WS-BPEL, short for Web Services Business Process Execution Language, an OASIS standard (OASIS TC, 2007). BPEL is an Orchestration language and its main goal is to standardize the process of automation between web services. However composing services introduces overheads, which affects the performance of the entire solution. The software developers may not have a clear understanding of those overheads or may have difficulties assessing the performance costs of the final composite product. Understanding the performance overheads is important in the early design phases, when important architectural decisions are made; since the system is not fully implemented and cannot be measured yet, performance evaluation in early phases should be based on the analyst’s experience with previous systems. This thesis will focus on studying the performance of a BPEL Orchestration.
1.1.2 Problem Statement

Develop and implement a method for assessing the overall performance of SOA systems during the early software development phases, by using measurement and modeling techniques.

1.1.3 Thesis Objectives

The first objective of this thesis is to measure the performance of SOA applications. Here are the challenges:

- A SOA system is usually multi-tiered system, and each tier has different measurement tools;
- Different measurement tools provide results in different formats;
- Lack of correlation of measurement fragments in each tier in the same execution session.

Performance cannot be retrofitted, but must be designed into software from the beginning. Software architecture studies have shown the importance of architecture in achieving software quality objectives, including performance. While decisions made at every phase of the development process are important, architectural decisions have the greatest impact on quality attributes such as modifiability, reusability, reliability, and performance, according to (Clements and Northrup, 1996):

“Whether or not a system will be able to exhibit its desired (or required) quality attributes is largely determined by the time the architecture is chosen.”

For this reason, the Software Performance Engineering (SPE) technique first proposed in the late 80’s (Smith, 1990) and further developed (Smith and Williams, 2002) became increasingly accepted and practiced (Cortellessa et al, 2011). SPE
recommends using quantitative methods based on performance models, such as Queuing Networks (Lazowska et al., 1984), Layered Queuing Networks (Woodside et al, 1995) or Petri Nets (Ajmone Marsan et al., 1995) to evaluate the performance effects of different design and implementation choices throughout the development of a system. SPE supports the idea that in order to build systems that meet their performance requirements, we need to integrate performance analysis into the software development process, from the earliest to the last phases. This would eliminate the “late-fixing” of performance problems, a frequently practiced approach that postpones any performance concerns until the system is completely implemented. Late fixes tend to be very expensive and inefficient, and the software may never reach its original performance requirements.

In order to build a quantitative performance model for BPEL Orchestration, we need to know what resources are required, how much they are used by the application itself and how much for operations provided by the underlying platforms (Alhaj and Petriu, 2012).

Therefore, the second objective of the thesis is to differentiate between the measurements of resource consumption by the application code itself and by the underlying platforms (such as the BPEL engine and the message communication). This second objective of the thesis stems from the fact that the performance characteristics of a system depend on all of its elements: software, operating system, middleware, and hardware. The performance contributions of these elements are quite difficult to identify and separate, mainly due to the limitations of the measurement tools that are available. Moreover, performance characteristics are highly non-linear in systems with contention for different resources, which happens in most software systems. The thesis follows the SPE instrumentation and measurement guideline. We design and implement a logging
system, which allow us to instrument the SOA software being developed, in order to automatically collect, correlate and evaluate performance test data. We also need to verify the credibility of the collected samples. Our measurements are focused on resource consumption, such as the execution demands for different parts of the application software and underlying BPEL engine, as presented in Chapter 4. This is different from other work in the literature, as for example (El Saddik, 2006), where the focus is on end-to-end performance measurements for testing the scalability and performance of a given web service application.

The third objective of the thesis is concerned with LQN modeling of SOA systems and the validation of the LQN model versus measurements. We build an LQN model corresponding to our case study by hand, following an existing methodology (Petriu, 2010). We compare the LQN results with measurements in order to validate the model, and then illustrate how to use the model for further analyzing the system performance for different deployments and configurations. The measurements for platform overheads obtained in this thesis can be reused in the future for building LQN models of other SOA applications that will be running on similar platforms. In general, the measurements can help SOA designers and researchers to gain a better understanding of the performance impacts of BPEL Orchestrations on similar platforms.

1.2 Proposed Approach

There are various tools and platforms with similar capabilities to implement BPEL Orchestration workflow solutions. Typically, a BPEL Orchestration development tool provides an easy-to-use visual interface for the designer to drag and drop different
activity shapes to “orchestrate” different web service calls into a composed workflow, and to deploy the solution to a platform-dependent BPEL engine.

In our research we developed a fictional Life Insurance Brokerage Service as a case study, which is a BPEL Orchestration that queries two different insurance companies, makes decisions on behalf of the client and returns the final deal back to the client. Since Microsoft BizTalk Server has a big market share in the BPEL engine market, it will be helpful if some platform dependent overheads can be provided for early performance estimation. This is the main reason why we utilized Microsoft Visual Studio as the development tool and Microsoft BizTalk server (Chappell et al., 2005) as the running platform to design and host BPEL Orchestrations.

Since a BPEL Orchestration is a SOA distributed system composed of multiple software components and services, it is difficult to collect data from the individual components in order to construct the whole picture for performance. To address this issue, we used C# language to write a logging facility that allowed us to instrument the case study software into different layers, in order to collect data and correlate it into a central database table (running on a Microsoft SQL 2008 Server). Finally we used Microsoft Excel as a statistical tool to calculate the sample mean and sample sizes based on Confidence Intervals and draw distribution charts for analyzing the data.

For individual case testing, we used SOAP-UI (Kankanamge, 2012), a popular Java based web service testing tool allowing us to create test cases that invoke web services by generating a web service proxy on the client side. We picked this tool because of its easy visual configuration and powerful cross platform heterogeneity compatibility. For load
testing, we used LOAD-UI (SmartBear, 2013) to emulate concurrent execution of test cases created in SOAP-UI because of its small footprint and scalable testing.

We conducted sample tests and measured service time on each BPEL Orchestration component; the number of samples was determined based on confidence intervals. Then we used each sample mean as input parameters into a manually constructed LQN model, which was in turn used to compute the performance of the final composed BPEL Orchestration. We used the same testing strategy to measure the end-to-end performance of the integrated SOA system in order to compare it with the LQN results to verify the accuracy of our model.

1.3 Thesis Contributions

This research will allow getting better insight into performance overheads in various stages of service-based BPEL orchestration. The main contributions of this thesis are as follows:

- Design and implement a unified logging facility for SOA systems that allows us to instrument different layers of the system under study, in order to automatically collect measurement data and to correlate the separate logs from sub-systems and components into a unified system view.

- Use the logging facility to collect performance measurement data for the case study system designed and implemented for this research. The measurements are collected separately for the performance effects of the application code and for the underlying BPEL engine. Confidence intervals techniques are used to insure the accuracy of the measurements. The dependency of communication overheads in function of the message size is also taken into account.
• Build a LQN model of the case study system using measured data as parameters, and validate the model results against end-to-end measurements. Use the LQN model for further performance analysis for different deployments and configurations. The measured platform overheads can be reused for building predictive LQN models for other SOA systems.

1.4 Thesis Contents

Chapter 2 provides background concepts on the thesis study, including SOA, Service Composition, BPEL Orchestration, Software Performance Engineering, Performance modeling, as well as tools for BPEL Orchestration development and testing, e.g. Microsoft Visual Studio, Microsoft BizTalk Server, BizTalk Orchestration Profiler Soap UI and Load UI.

Chapter 3 uses Insurance Broker Service as a case study to demonstrate how a BPEL orchestration is designed and how Microsoft BizTalk Server is used to implement it.

Chapter 4 present a strategy and procedures for automatic performance testing, using confidence intervals to calculate the sample size, measure performance parameters for each subsystem and correlate the logs into a unified view. The logging facility was implemented using C# and used for measurements of the case study system.

Chapter 5 is concerned with building a LQN model for the Insurance Brokerage BPEL orchestration study case and validating the model by comparing its results with actual performance measurement of the implemented system.

Chapter 6 presents the conclusions of the thesis and future work.

The appendices show the logging facilities source code, sample test results, and LQN files.


2 BACKGROUND

This chapter describes background knowledge about concepts, architecture, frameworks and tools utilized in our research process. These topics include:

- SOA, Web Services, Service Orchestration, BPEL and WSDL;
- Tools and platform for orchestration development and testing;
- Software Performance;
- Software Performance Engineering;
- Performance Models.

2.1 SOA and Web Services

Service-Oriented Architectures (SOA) is a distributed computing paradigm for developing large complex applications by composing software components running on heterogeneous platforms, which offer services to one another through well-defined interfaces (Earl, 2005). These services implement well-defined business functionalities offered by software components which can be reused for different applications. There are different supporting technologies (provided by service platforms) that help developers to implement systems, each providing different features for publishing, discovering and invoking services. SOA is not directly related to any specific technology, although it is most often implemented with Web Services standardized by OASIS. The set of standards for Web Services includes a specification for service registries known as Universal Description Discovery and Integration (UDDI), a service description language named Web Services Description Language (WSDL), a language for service orchestration called
Business Process Execution Language (BPEL) and a distributed object communication protocol called Simple Object Access Protocol (SOAP).

Web Services realize a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It is an approach to have software resources in an enterprise available and discoverable on network as well defined services. Each service would achieve a predefined business objective and perform discrete units of work. Well designed services are independent and do not depend on the context or state of the other services. They work within distributed systems architecture. (OASIS, 2012).

2.1.1 Service Orchestration

Service Orchestration is an organization workflow which composes multiple services based on the enterprise business process and exposes the result as a new service; it is the combination of service interactions to create higher-level business services.

Service orchestration works through the exchange of messages. This is usually accomplished through Enterprise Application Integration (EAI), which enables data integration, or the use of a central messaging engine such as an Enterprise Service Bus (ESB), which routes, transforms and enriches messages.

2.1.2 Web Service

A web service is a piece of software that makes itself available over the internet and users a standardized XML messaging to communicate with the consumer.

Figure 2-1 shows the OASIS Web Service architecture, which rely on standards like XML, WSDL and SOAP.
According to the high-level definition of SOA, business functionality must be distributable and accessible in some way. This is done with the Web Service Description Language (WSDL) described below, an XML-based language that is used for describing the functionality offered by a Web Service.

2.1.3 WSDL

WSDL describes network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. (W3C, 2001) The operations and messages are described abstractly, and then bound to a concrete network protocol and message format. It is platform and language independent and is used primarily (but not exclusively) to describe SOAP enabled services as follows:

- **Address** – Where the service resides, i.e., details of the protocol specific address, e.g., a URL;
- **Binding** – How to invoke it, i.e., details of the data formats and protocols necessary to access the service’s operations.
- **Contract** – What a service does, i.e., the operations the service provides.

Figure 2-1 Web Services Architecture
This so-called A-B-C schema uses the following elements, as shown in Figure 2-2:

- Port is defined by associating a network address with a reusable binding
- Collection of ports defines a service
- Messages are abstract descriptions of the data being exchanged
- Port types are abstract collections of supported operations
- The concrete protocol and data format specifications for a particular port type (Operation) constitutes reusable bindings, where operations and messages are then bound to concrete network protocols and message formats (such as SOAP).

Figure 2-2 WSDL concepts
2.1.4 SOAP

Simple Object Access Protocol (SOAP) is one of the protocol specifications for exchanging structure information in the implementation of Web Services as shown in Figure 2-3. It relies on Extensible Markup Language (XML) for its message format and other Application Layer protocols like HTTP, SMTP, TCP or Message Queue, etc. for message negotiation and transmission.

![Figure 2-3 SOAP Protocol](image)

As shown in Figure 2-4, the SOAP message structure contains:

- **SOAP Envelope** – the root container of the SOAP message, which contains Namespaces, Encoding rules, Header and Body elements
- **SOAP Headers** – used for:
  - Security: WS-Security and SAML place additional security information (like digital signatures and public keys) in the header.
  - Quality of Service: negotiate particular qualities of service such as reliable message delivery and transactions.
Session State Support: Many services require several steps, so it requires maintenance of session state, equivalent to cookies in HTTP. The Session identifier is in the header.

- SOAP body – contains the actual XML message defined in the WSDL schema.

![SOAP message structure](image)

**Figure 2-4 SOAP message structure**

WSDL Service description and SOAP infrastructure isolate all technical details, e.g., hardware platform and implementation language specific elements away from service consumer with service provider.

### 2.1.5 BPEL

To facilitate Service Orchestration, the Business Process Execution Language (BPEL) was created to standardize integration logic and process automation between web
services. It is an OASIS standard jointly developed by IBM, BEA, SAP, Siebel and Microsoft etc. The design goals of BPEL are as follows (Andrews et al, 2003):

- Define business processes that interact with external entities through web service operations defined using WSDL 1.1 and that manifest themselves as Web services defined using WSDL 1.1.
- Define business processes using an XML-based language.
- Define a set of Web service orchestration concepts that are meant to be used by both the external (abstract) and internal (executable) views of a business process.
- Provide both hierarchical and graph-like control regimes, and allow their use to be blended as seamlessly as possible.
- Provide data manipulation functions for the simple manipulation of data needed to define process data and control flow.
- Support an identification mechanism for process instances that allows the definition of instance identifiers at the application message level. Instance identifiers should be defined by partners and may change.
- Support the implicit creation and termination of process instances as the basic lifecycle mechanism.
- Define a long-running transaction model that is based on proven techniques like compensation actions and scoping to support failure recovery for parts of long-running business processes.
- Use Web Services as the model for process decomposition and assembly.
- Build on Web services standards (approved and proposed) as much as possible in a composable and modular manner.
A BPEL process is a XML document consisting of steps, each one a XML element called an activity. BPEL supports basic and structured activities. Basic activities represent basic constructs and are used for common tasks, such as:

- **<invoke>** – Invoking other web services, when invoking a synchronous request/response operation, it requires both input Variable and output Variable, as shown in Table 2-1

  Table 2-1 BPEL Invoke element

  ```xml
  <bpel:invoke partnerLink="SunLifeWSPort" portType="q2:ServiceSoap" operation="Quote" inputVariable="SunLifeRequestMsg" outputVariable="SunLifeResponseMsg" />
  ```

- **<receive>** – Waiting for an incoming message (operation invocation), either for the initial to start the BPEL process, or for a callback; shown in Table 2-2:

  Table 2-2 BPEL receive element

  ```xml
  <bpel:receive partnerLink="Client" portType="q1:ClientPortType" operation="ClientQuote" variable="ClientRequestMsg" createInstance="yes" />
  ```

- **<reply>** – Generating a response for synchronous operations, <reply> is always related to the initial <receive> through which the BPEL process started, shown in Table 2-3:

  Table 2-3 BPEL reply element

  ```xml
  <bpel:reply partnerLink="Client" portType="q1:ClientPortType" operation="ClientQuote" variable="ClientResponseMsg" />
  ```
• **<assign>** – Holding and manipulating data variables, it can perform one or more
  **<copy>** commands within. For each **<copy>** we have to specify the source (**<from>**) and the destination (**<to>**). The syntax of an assignment is presented Table 2-4:

  **Table 2-4 BPEL assign element**

  ```xml
  <bpel:assign>
    <bpel:copy>
      <bpel:from>
        <PremiumAmount xmlns="" xmlns="">0</PremiumAmount>
        <Insurer xmlns="" xmlns="" xmlns="">0</PremiumAmount>
        <Insurer/>
        </InsuranceResponse>
      </bpel:from>
      <bpel:to variable="varClientResponseXmlDOM" />
    </bpel:copy>
  </bpel:assign>
  ```

• **<throw>** – Indicating faults and exceptions

• **<wait>** – Waiting for some time

• **<terminate>** – Terminating the entire process.

Structured activities are used to combine basic activities to construct BPEL process.

The most import structured activities are:

• Sequence (**<sequence>**) for defining a set of activities that will be invoked in an ordered sequence

• Flow (**<flow>**) for defining a set of activities that will be invoked in parallel

• Case-switch construct (**<switch>**) for implementing branches

• While (**<while>**) for defining loops

• Use **<pick>** to select one of a number of alternative paths
2.1.6 **BPEL Engine**

SOA vendors offer comprehensive suites of SOA products that in theory should be capable of addressing all aspects of SOA, including governance, integration, business process management and others.

Formation of SOA suites is having a big impact on how SOA products are selected, as many organizations are being tempted to settle for "one stop shop" approach as opposed to doing proper product evaluation within each SOA product category.

Table 2-5 (MYARCH Journal, 2007) shows what a typical SOA suite looks like and how offerings from different vendors support different aspects of SOA. SOA products within the same category can differ substantially in terms of their feature sets. Definitions of ESB, registry and other SOA product categories are not standardized and so vendors are free to categorize their products as they wish. Detailed analysis and evaluation is still a requirement when selecting SOA products.

**Table 2-5 SOA Suites from different vendors**

<table>
<thead>
<tr>
<th>SOA Suites</th>
<th>IBM</th>
<th>BEA</th>
<th>Oracle</th>
<th>Microsoft</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Services Container (application server)</td>
<td>WebSphere Application Server</td>
<td>WebLogic Application Server</td>
<td>Oracle Application Server</td>
<td>.NET/Windows</td>
</tr>
<tr>
<td>ESB</td>
<td>WebSphere ESB Message Broker</td>
<td>AquaLogic Service Bus</td>
<td>Enterprise Service Bus</td>
<td>BizTalk Server(some ESB capabilities)</td>
</tr>
<tr>
<td>Registry/Repository</td>
<td>WebSphere Registry and Repository</td>
<td>AquaLogic Service Registry</td>
<td>Oracle Service Registry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AquaLogic Enterprise Registry</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BizTalk Talk Server is one of the most popular and widely used BPEL platforms in the market; it was created by Microsoft (Chappell, 2005). It is an integration server product used to connect different applications, systems, or business processes within an organization or between organizations. It is used to aggregate services to build a Service-Oriented Architecture.

BizTalk is a message-oriented middleware, which means that all the communication between BizTalk server and external systems happens by exchanging messages. In BizTalk, all messages are internally converted to XML, which adheres to the specified XML Schema Definition (XSD).
The architecture of BizTalk server 2006 can be broadly divided into the Messaging System and Business Processes (called Orchestration). Messaging involves receiving, storing, and forwarding of messages and interacting with various external systems. Orchestration is where the business logic is executed. Figure 2-5 describes the BizTalk server architecture graphically.

![Figure 2-5 Architecture of BizTalk Server](image)

This thesis uses the following BizTalk components:

### 2.1.7.1 Ports

Ports exchange messages with the outside world. The logical ports defined in Orchestration are mapped to the physical ports at the time of deployment by a process known as binding.

The following physical ports are available to BizTalk:

- **Receive Ports**: Used to receive messages into BizTalk.
- **One-Way Ports**: Only receive messages.
- **Request-Response Ports**: Receive as well as send back a response.
- **Send Ports**: Used to send messages out of BizTalk.
- Static One-Way Ports: Only send messages, and has a fixed send adapter and destination address.
- Static Solicit-Response Port: Send as well as get an acknowledgment back, and has a fixed send adapter and destination address.
- Dynamic One-Way Port: Only send messages; the actual destination of the message and the protocol/adapter to be used are determined at run time.
- Dynamic Solicit-Response Port: Send as well as get back an acknowledgment; the actual destination of the message and the protocol/adapter to be used are determined at run time.

Physical ports can be created during development using BizTalk Explorer inside Visual Studio and during deployment using the BizTalk Deployment Wizard, as shown in Table 2-6.

**Table 2-6 BPEL Partner Links in Orchestration Designer**

<table>
<thead>
<tr>
<th>For Service Consumer</th>
<th>For Service Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- means sending Request out first
- means receiving Response back later
- means receiving Request first
- means sending Response back later
2.1.7.2 Receive Locations

Messages are received into BizTalk at a receive location of a receive port. A receive port can have multiple receive locations, each with its own receive adapter and receive pipeline, so that it can receive incoming messages by various means and in various formats.

It is useful to have multiple receive locations on a single receive port, so that each receive location can deal with a specific interface type as shown in Figure 2-6 and Figure 2-7, a receive port contains one of many receive locations, each has its own definition of protocol type and pipeline etc.

![Figure 2-6 Receive Port contains Receive Locations](image-url)
2.1.7.3 Adapters

Adapters are used by BizTalk to interface with the outside world, for example to connect to external applications or systems. BizTalk provides built-in adapters to communicate with external systems using protocols like SOAP, HTTP, SMTP, and FTP. It can also communicate with SQL Server and the Windows file system using the SQL and FILE adapters respectively. However, if you need to connect to systems that have a proprietary interface/protocol, then you either need to build your own custom adapters using the Adapter Framework provided by BizTalk or purchase the adapters from third party vendors. Adapter type selection is shown in Figure 2-7 where SOAP is picked in this case.

2.1.7.4 Message Contexts

Each message has a message context associated with it. Message contexts are constructed by the Messaging Engine and consist of information such as message type,
message ID, and receive port name among other things. They can also contain user data from the message itself that are explicitly promoted by the developer.

2.1.7.5 Promoted Properties

The promoted properties are mainly used for Content Based Routing of messages and also to correlate the request and response messages in an asynchronous communication.

For example, Insurance Broker Orchestration is interested in processing only those insurance amounts whose total value is above $100,000. In such cases we promote the Insured Amount field of the message so that it can be accessed by the messaging engine without loading the whole message into memory to evaluate the subscriptions.

The properties to be promoted are specified in the schema files using BizTalk Editor inside Visual Studio. Doing so will actually create a separate Property Schema that will contain only the promoted properties. The values are promoted at runtime by the receive pipeline. In this thesis we don’t use Promoted Properties.

2.1.7.6 Distinguished Fields

Distinguished Fields are quite similar to Promoted Properties and are used to access the content of the message easily without having to load the whole message into memory but only from within an Orchestration. They allow the developer to access the content in an object oriented fashion as given in Figure 2-8.
Distinguished fields are lightweight and do not need a separate Property Schema to be created. They can be created easily using the Orchestration Schema Editor as shown in Figure 2-9.
2.1.7.7 Message Box

Message Box is central to the messaging system. Once an incoming message passes through the receive pipeline, it is stored in the Message Box and then routed to its respective destinations based on publish-subscribe mechanism. Message Box is an SQL Server database that stores messages and their contexts. Messages from the Message Box can either be routed to the Orchestration or to a Send Port for delivery to an external system or both, based on the subscriptions.

When Orchestrations and Send Ports are enlisted, they create a subscription indicating what kind of messages they are interested in. These may include a set of criteria such as “The total amount of an insurance amount must be above $100000 etc”. When a message arrives into the Message Box (known as publishing the message) the messaging engine compares the context properties of the message with the available subscriptions and if they are found to match, the message is routed to the appropriate Orchestration or Send Port.

2.1.8 Difference between BizTalk and BPEL standard

BPEL focuses on business process automation while BizTalk is a SOA platform and framework; BPEL focuses on standardizing presentation of workflow orchestration and choreography while BizTalk is trying to create a one shop SOA platform for enterprise. The BizTalk server extends the BPEL standard with some platform-dependent features to make the platform more powerful. For example:

- ability to use XLang and .NET code inside orchestrations;
- easy-to-use User Interface to present the BPEL XML document as a graph;
• built-in adapters and pipelines support for all the major protocols such as HTTP, MSMQ and SMTP etc.

2.2 Software Performance

Software Performance is the degree to which a software system or component behavior fulfils its objectives for timeliness (Smith, 2000). The performance is described in term of requests made by the users and system responses, as well as usage of system resources (software and hardware) to serve the respective requests. Frequently used performance measures are response time and throughput. Response time is typically described from a user perspective and is defined as the time required by the system to serve a user's request. The throughput is the number of requests that can be processed per unit of time. The maximum throughput indicates the capacity of the system. The performance parameters are workload dependent. Workload intensities specify the level of usage of the system, and can be open or closed. An open workload is usually specified by the arrival rate of the requests, and a closed workload by the number of concurrent users.

Performance is an essential attribute of every software system. The performance failure of a software system can cause various consequences including damaged customer relations, income loss, need for additional project resources, market loss, business failure, and project failure (Smith, 1998).

Functional requirements of a software system are usually clearly stated and understood, but non-functional requirements such as performance are often omitted at the early stage of the software development until it is too late to meet them. Many projects have failed not because they could not provide the required functionality, but because of
their poor quality, which in many cases meant that they could not meet their performance requirements. According to (Smith, 1990), many software systems could not be used as initially implemented due to performance problems.

To understand and define performance, we need to view a software system in three related ways (Woodside, 2008):

- Scenarios: a scenario describes system behavior for a high-level operation by a sequence of sub-operations.
- Resource demands: we need to know how an operation loads the system resources.
- Architecture: the architecture describes the organization of objects and modules with interaction relationships.

2.2.1 Software Performance Engineering (SPE)

SPE is a systematic, quantitative approach to construct software systems to meet their performance objectives. It uses quantitative techniques to identify architectural, design, and implementation alternatives that will meet performance objectives (Smith, 1990; Smith and Williams, 2001, Cortellessa et al. 2011). SPE is a model-based and software-oriented approach. Modeling is central to both SPE and object-oriented development. By building and analyzing models of the proposed software, we can discover its characteristics to decide if it will meet its requirements before we actually commit to build it. In SPE, performance study begins as early as possible in the development process, when a preliminary design exists.

Integration of software performance engineering into the software development process will greatly improve the software quality and lower the risk of software failure.
The type of performance models used depends on the purpose of analysis, the level of detail available, and the precision of the input data. Early in the software development process, the knowledge of the software’s design and implementation details is insufficient to model the system’s performance precisely. Simple models with estimated parameters are used to get rapid feedback on whether the proposed software is likely to meet the performance objectives. Rough performance estimate at an early stage can prevent costly performance mistakes that may make the systems miss its performance requirements by orders of magnitude. As software development progresses, more and more details of the software design and implementation become available, so it’s possible to replace the rough estimates of resource usage with more detailed estimates or with measurements of actual usage. Consequently, the accuracy of the performance predictions can improve as the project progresses (Petriu, 2010).

Software architectures need to be decided at the earliest stage in a software development project. Architecture is a specification of the components of a system and the interaction between them. Architecture guarantees certain behavioral properties of a conforming system and can be a powerful tool to aid the process of predicting the behavior of a system with that architecture, of managing the construction of a system, and of maintaining it.

The architectural decisions have the greatest impact on software quality. Although a good architecture alone could not guarantee that the product will reach the quality goals, a poor architecture surely can prevent their realizations. Inappropriate decisions or errors at early stages will cost excessively or cause the failure of the project. Most performance failures are because of a lack of consideration of performance issues early in the
development process and in the architectural design stage. The “fix-it-later” approach is dangerous; it could not save the project if its major architecture is not suitable to handle the required tasks, no matter how the software is implemented.

The SPE process includes the following nine steps (as shown in Figure 2-10):

1. Assess performance risk;

![Figure 2-10 The SPE Workflow (Smith, 2001)](image-url)
2. Identify critical use cases;
3. Select key performance scenarios;
4. Establish performance objectives;
5. Construct performance models;
6. Determine software resource requirements;
7. Add computer resource requirements;
8. Evaluate the models;
9. Verify and validate the models.

To be effective, SPE should be an integral part of the software development process. SPE should start as early as possible in the development process.

2.3 Performance Models

A performance model captures the performance characteristics of the entire system: software, operating system, middleware, and hardware, as it captures all the system resources and the competing demands made on resources when executing different scenarios under different workloads. The results of a performance model are system-level measures such as end-to-end delay, response time, throughput, utilization, etc. There are many performance-modeling methods and tools available, based on different formalisms such as queuing networks, stochastic Petri nets, stochastic process algebra, and stochastic automata. All formalisms can be used for modeling software systems (Corttelessa et al., 2011). In this thesis we will be using Layered Queuing Networks (LQN), which is a kind of extended queuing networks; the difference between QN and LQN is explained below.

*Queuing Network (QN) Models* have been widely and successfully used as a model for predicting the performance of traditional computer systems for more than three
decades (Kleinrock, 1975). The QN model assumes that the features that affect performance most are the queuing for devices. It uses mean value analysis as a performance evaluation method to estimate the system performance. The problem with the QN model is that it is difficult to represent nested services, so it often fails to capture important details of logical resources and process communication, or complex interactions among various software and hardware components in distributed and networked system such as client/server systems.

Layered Queuing Network (LQN) is developed as an extension of Queuing Network model for handling the complex interactions in distributed and concurrent software system (Rolia, 1995; Woodside et al., 1995; Woodside, 1998). LQN is a performance model as well as a new kind of architectural model to capture the queuing based contention and simultaneous resource possession aspects of a system. LQN models predict the performance of distributed and concurrent software system based on the structure of the software and its use of logical and physical resources (Petriu, 2003). A layered model illustrates the system architecture as software and hardware modules with resources embedded in them in a layered fashion, and with resource demands as parameters (Petriu, 2010). The advantage of LQN over QN is that LQN can easily represent nested services. A server that serves clients’ requests could be in turn a client to other servers, and the returning results from these servers could be used to serve its own clients. The nested services are common for concurrent and distributed applications, which have complex interactions among software components, network, and hardware devices.
2.4 Tools for BPEL Orchestration Development and Testing

2.4.1 Microsoft BizTalk server 2006

Microsoft BizTalk Server (Chappell et al. 2005) provides a development and execution environment to design and host workflow system. It has the following features:

- Ability to define XML schema based business document specifications and how these documents have to be transformed when passing between applications.
- Provides a standard gateway for sending and receiving documents across the network as well as providing a range of services to ensure data integrity, delivery, and security.
- Ability to monitor and log server activities.
- BizTalk Messaging Services provides the ability to send business documents securely and reliably.
- BizTalk Orchestration or workflow services enable to define the business processes that are used to create the messages and then implement them using an integrated graphical environment.

2.4.2 Orchestration Design Tool

Orchestration designer tool is a plug-in to Visual Studio Integrated Development Environment provides an easy-to-use graphical interface to design workflow and map it to the XML based BPEL Orchestration documents. Screenshot of the tool is shown in Figure 2-11.
2.4.3 SOAP_UI

SOAP-UI is an open source cross platform testing tool (Kankanamge, 2012). It provides easy-to-use graphical interface to create functional, regression, compliance and load tests. SOAP-UI allows importing the WSDL definition of a web service, and then it generates a web service proxy and client, and calls the web service. Beside many other functions, it provides traces of SOAP messages exchanged by client and web service, as well as end-to-end execution time measurement. It allows for the creation of test cases with assertions, like any other unit testing tool or framework. Figure 2-12 shows a test case of Insurance Broker BPEL Orchestration Service in SOAP-UI.
2.4.4 LOAD_UI

LOAD-UI is an open source load testing tool; it can easily set the concurrent throughput by graphical user interfaces and put the demands on test cases created in SOAP-UI and it has various data logging tools with capabilities to generate graphs and summary reports.

Examples of LOAD-UI test cases and graphs generated are shown in Chapter 3.
3 Design and Implementation of Case-Study System

3.1 Business Logic

We will use insurance broker composite service as an example to explain the elements of a BPEL Orchestration and how they work together. Later we will measure the performance of each individual element of the composed service and use them as input parameters to a LQN model to predict the performance of the final composed service.

The use case diagram of the Insurance Broker System is shown in Figure 3-1, from the use case diagram, we can see:

- Insurance Broker Service Subsystem is a service-based application connecting client and different services
- Insurance Broker Service Subsystem provides a composite interface to clients to dynamically send requests to different insurance services and pick the most affordable one, which mitigates the need for the client to construct different request formats to different providers and parse different responses. It provides a loose coupling between the client and the backend services (and services on the cloud).
- Inside the Insurance Broker Service System is a central workflow process – BPEL orchestration, it takes control over the involved web services and coordinates the execution of different operations on the web services involved in the operation.
Figure 3-1 Use Case Diagram of Insurance Broker Service System

Figure 3-2 shows the Activity Diagram of the Insurance Broker Service.

Figure 3-2 Activity Diagram of Business Logic
3.2 Individual Elements of the Application

Based on the use case diagram and activity diagram derived from the analysis, end user will either call in or directly use browser to get access to the quoting service, in either case, a service call will be launched to the composed service. We use the following steps to implement the application:

- We designed a database to hold different insurance quotation rates based on certain risk factors like birthday, sex and health condition etc. for two fictional insurance companies, called here Manu Life and Sun Life; then we insert large data feeds into the tables in order to prolong the query time and deliberately leave the table of Sun Life quote un-indexed in order to mimic a situation that one service provider is slower than the other. We created SQL stored procedures to accept quoting criteria as parameters to get the insurance quote.

- We used ASP.NET and Internet Information Server to create two web services, one for each insurance provider to accept quoting criteria, then call the back-end database to return the quote to the client.

- We used BizTalk Server as the BPEL engine to compose Manu Life and Sun Life web services into a new broker service based on Activity Diagram of the Business Logic.

- In order to get the performance measurement data, we injected custom logging code into each web service and BizTalk Orchestration as well, which later we are going to compare with logs provided by the BizTalk Server platform as well as the operating system.

The implemented BizTalk Orchestration is shown in Figure 3-3.
Figure 3-3 Insurance Quotations BPEL Orchestration
### 3.3 Application and Platform overheads

The composed BPEL service in our use case study includes two individual web services, each running on its own Microsoft Internet Information Server and SQL database; there is also a BizTalk BPEL Orchestration engine that integrates these 2 services together, each composing element will introduce the following performance overheads:

- Construct request message using XML DOM
- Send request and receive response through SOAP protocol stack
- Parse SOAP Message and get parameters and results
- Business logic execution in code
- SQL database execution
- Disk Access

We put all these aforementioned application and platform overheads as blue shapes in the Activity Diagram shown in, while the white shapes are purely business logic related BPEL logic.
Figure 3-4 Activity Diagram of Composed Service with consideration of Overheads
Another consideration factor is the granularity of capturing the performance overheads. For instance, the WSDL protocol goes through the abstract service interface definition to the concrete service as shown in. There is no easy way to use tools to log the QoS parameters in each step; tools like IBM Rational Quantify (Rational Quantify, 2004) are good for analyzing stand alone applications that do not communicate with other systems. However, for SOA software, Quantify would generate too many complicated low level measurements to analyze. Since a SOA system consists of services that communicate with different applications and platforms, when we evaluate the performance of SOA system, we need to consider the performance overhead of each composing element at a higher level granularity. Using the SOAP stack as an example, we only focus on the cost of sending and receiving the SOAP requests and parsing the request XML, while ignoring the cost of finding the service interface, etc.

3.4 Performance Testing Facilities for a SOA system

In order to evaluate the quality of multi-tier SOA systems from both the consumer and the service provider point of view, we need to use special logging mechanisms to retrieve the QoS measurement data. Logging facilities include custom logging and system logging provided by the software. There are pros and cons about them.

The advantages of using system logging tools are:

- Logging tools are created by platform designers, who are familiar with how the system works and are able to minimize the performance impact on the product introduced by the tool.
The measurement data of logging tools is available right away; there is no need to spend time and effort to design logging facilities which will lower the cost and budget of evaluating the system performance.

The disadvantages of using system logging tools are:

- Different logging tools provide different logging formats. E.g., BizTalk uses HAT to view technical details of a particular orchestration, pipeline or message instance, as well as the message flow of a particular message that enters the system. On the other hand, Microsoft SQL Database Server uses SQL Server Profiler to log and track database execution details. They are in different formats. One cannot use a single tool to read both logs without special adaptors for the translation.

The advantages of custom logging facilities are:

- Focus only on the performance factors we want to measure.
- Log data in unified formats and be able to correlate them across different software platforms.
- By unifying the format of data, we can use automatic procedures to test, extract and analyze, which makes large volume test and analyze feasible and efficient.

The disadvantages of injecting custom logging facilities are:

- Source code in which to inject the logging code has to be available.
- Performance testing implementation and source code modification are needed, which might introduce bugs and extra performance overheads and require extra time.
This thesis suggests a hybrid measurement system which uses both custom and built-in logging tools. The built-in system logging tools coming with BizTalk are used to verify that the custom measurement code is small and accurate enough to be used as SOA QoS evaluation tool, whereas custom logging is used for automatic gathering, extracting and analyzing sample data. The logging facility developed in this thesis is described in the next chapter.
4 PERFORMANCE MEASUREMENTS

4.1 Introduction

SOA system is a multi-tier system consisting of different layers of applications and services. The SOA components used in this project are as shown below:

- Client service – this is a new service exposed to users from the orchestration of different services. It provides a public interface and a proxy to be consumed externally. In our thesis project, this is an ASP.NET web application running on Microsoft IIS Web Application Server communicating with the BizTalk Orchestration Engine using SOAP protocol as shown in Figure 4-1.

![Insurance Broker Orchestration Service Proxy](image)

Figure 4-1 Insurance Broker Orchestration Service Proxy
This service proxy can be generated using BizTalk Web Services Publishing Wizard as shown in Figure 4-2.

![BizTalk Web Services Publishing Wizard](image)

**Figure 4-2 BizTalk Web Services Publishing Wizard**

- BizTalk Orchestration Middleware – this is the BPEL execution engine running in Microsoft BizTalk 2006 server. In our thesis project, this is the Insurance Broker Quoting Service Workflow that queries two other web services, compare the quotes and gets the lowest return to the client as shown in Figure 3-1.

- Backend database – Microsoft SQL Databases stores insurance quotation formula data used by different insurance companies.

- Two other distributed web services, fictional Sun Life and Manu Life Insurance web services, they are also multi-tier systems
4.2 Test strategy for thesis project

The goal of this thesis is to evaluate the integrated SOA system based on measurements of each participating service and BPEL Orchestration platform. We used confidence intervals to determine the sample test size and measured the performance of Manu Life, Sun Life web service and BizTalk BPEL orchestration engine overhead separately; later used these measurements as input parameters to a LQN model to analyze the performance of the composed SOA service; and finally we did the measurement of the implemented SOA system to validate the accuracy of the prediction.

4.2.1 Custom logging facilities footprint

Due to the complexity of testing SOA integrated system, we need to inject custom measurement code to retrieve logging information. We verify the footprint of the custom logging with system logging tools provided by the software platforms like BizTalk Health, Activity Tracking (HAT) and SQL Server Profiler, etc. We need to know if the footprint of the custom logging facility is small enough in order to further use the custom logging facility to automatically gather, extract and analyze sample data. Section 4.3 describes how this was done and why we concluded that the footprint is small enough to be used for our measurements.

4.2.2 Performance factors

There are many factors that can impact the performance of a system, e.g. CPU and memory of the machine can directly make a difference of the final performance. In this thesis, we will focus on the following two factors which are directly related to the business:
• Message size being transmitted – in order to see the performance trend for the system, in our study case, we picked 1KB, 3KB, 30KB, 300KB and 3MB as per typical message size
• Number of concurrent users – we picked 1 to 50 concurrent users, as per different scenarios.

4.2.3 Logging parameters

To design our custom logging facility, first we defined the unified performance measurement logging format as shown in Table 4-1, and then injected the measurement code into different SOA tiers.

**Table 4-1 Definition of custom performance logging parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Sample Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogID (GUID)</td>
<td>An unique identifier for a logging entry</td>
<td>8bf905c9-63a4-4914-b64a-2456cecf4813</td>
</tr>
<tr>
<td>Sequence</td>
<td>A number represents the order of a logging entry along the execution path</td>
<td>1</td>
</tr>
<tr>
<td>LogTimeStamp</td>
<td>The time stamp of a logging entry</td>
<td>01/24/2013 10:52:25.465 PM</td>
</tr>
<tr>
<td>ProcessTotalTimeSpan (accuracy: 100 nano sec)</td>
<td>The total time process executed from the previous step to the current step</td>
<td>710'900.375 nano second</td>
</tr>
<tr>
<td>ProcessUserTimeSpan (accuracy: 100 nano sec)</td>
<td>The time process executed the application code from the previous step to the current step</td>
<td>557’800.125 nano second</td>
</tr>
<tr>
<td>ProcessKernelTimeSpan (accuracy: 100 nano sec)</td>
<td>The time process executed code of operating system core (system libraries) from the previous step to the current step</td>
<td>153’100.25 nano second</td>
</tr>
<tr>
<td>ThreadTotalTimeSpan (accuracy: 100 nano sec)</td>
<td>The total time thread executed from the previous step to the current step</td>
<td>29’600.875 nano second</td>
</tr>
<tr>
<td>ThreadUserTimeSpan (accuracy: 100 nano sec)</td>
<td>The time thread executed the application code from the previous step to the current step</td>
<td>25’000 nano second</td>
</tr>
<tr>
<td>ThreadKernelTimeSpan (accuracy: 100 nano sec)</td>
<td>The time thread executed code of operating system core (system libraries) from the previous step to the current step</td>
<td>4’600.875 nano second</td>
</tr>
</tbody>
</table>
4.2.4 Considerations for choosing the logging parameters:

- **LoggingID** – Global Unique Identifier (GUID) to correlate performance data from different log files across different systems. For example, if LogID is “8bf905c9-63a4-4914-b64a-2456ccef4813”, no matter whether the data was logged in BizTalk Orchestration or Manu Life Web Service, they are all related to the same Insurance Quote instance. Each quote instance contains 26 monitoring spots as shown in Figure 4-4. Each monitoring spot will log current time stamp, TotalProcessorTime, UserProcessorTime and PrivilegedProcessorTime of the current process and the current thread, respectively.

- **Current time stamp** – we use .NET framework’s System.DateTime.Now, which retrieves the current local date and time and allows a local time to be expressed unambiguously as a single point in time, and in turn makes that time value portable across systems. In Windows platform, the accuracy of DateTime is 100 nano seconds. For example, we log the time using format “MM/dd/yyyy hh:mm:ss.fff tt” like “01/24/2013 10:52:25.465 PM” the accuracy is up to 0.001 milliseconds. Later we compare this parameter with the time stamp logged in BizTalk HAT and Microsoft SQL Server Profiler to see whether they are close enough to confirm whether the custom logging facilities are accurate.

- **TotalProcessorTime**, gets the total processor time for a process or thread.

- **UserProcessorTime**, gets the amount of time that the associated process or thread has spent running code inside the application….

- **PrivilegedProcessorTime**, gets the amount of time that the thread has spent running code inside the operating system core.
• TotalProcessorTime = UserProcessorTime + PrivilegedProcessorTime

• When measuring the performance of BPEL Orchestration, we need to use the TotalProcessorTime, because we need to count the time spent on the User Code as well as the time spent on calling the system libraries.

• Use ProcessThread to obtain information about a thread that is currently running on the system. Doing so allows us, for example, to monitor the thread's performance characteristics.

• A thread is a path of execution through a program. It is the smallest unit of execution that Win32 schedules. It consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler.

• A process consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores, and dynamically allocated memory. Each resource of a process is shared by all that process's threads.

• A program executes when the system scheduler gives execution control to one of the program's threads. The scheduler determines which threads should run and when. A lower-priority thread might be forced to wait while higher-priority threads complete their tasks. On multiprocessor computers, the scheduler can move individual threads to different processors, thus balancing the CPU load.

• Each process starts with a single thread, which is known as the primary thread. Any thread can create additional threads. All the threads within a process share the address space of that process.
• When we inject custom testing facilities, in order not to introduce overheads to the system’s performance, we put logging information into variables in memory, whose time is fast and can be ignored, and write this logging information to log files after all the important tasks are finished and the system exits.

4.2.5 Logging facilities

Since our thesis project contains multiple tiers and they are connected by messaging mechanisms, each tier has its own logging file as show in Figure 4-3.

![Figure 4-3 Multi-Tier of Insurance Broker SOA service](image)

Totally there are 4 log files and later we need to merge them into one and correlate each test using the unique LogID:

- **BPELOrchestrationLog.log** – logs execution time inside the BPEL orchestration
- **BPELOrchestration QoS.log** – logs the response time from the entry point and exit point in the BPEL orchestration
- **ManuLifeWSQoS.log** – logs the execution time inside Manu Life Quoting Service – one of the web services being called by BPEL orchestration
- **SunLifeWSQoS.log** – logs the execution time inside Sun Life Quoting Service – one of the web services being called by BPEL orchestration
After we correlate 4 log files into one log file we will have a performance picture of the whole composed service as shown in Figure 4-4.

![Diagram of the composed service](image)

**Figure 4-4 Monitoring spots labeled (0) to (28)**

### 4.3 Confirming code injection test accuracy

In order to check the footprint of our custom logging facility, first we used SoapUI tool to retrieve sample data:
- Create a test project in SoapUI to where the WSDL of the BPEL Orchestration web service is assigned. SoapUI generate an empty XML request based on the WSDL.

- Fill in web service parameters into the empty XML request and send it to the endpoint, a XML response returned successfully as shown in Figure 4-5.

![Figure 4-5 SoapUI test cases](image_url)

- We measured the timestamp of each execution step. The difference between the current step and the previous one is the service time spent from the last step to the current one. The logging code is shown in Appendix. In the signature of WriteLog(string logId, int order), logId is the unique identifier that identifies a single test, and order is the sequence of the execution steps.

- The test generates 4 log files in a Tab delimited format with carriage return at each row as shown in Table 4-2.
We compared the time stamp logged in the log files with corresponding items in BizTalk HAT (shown in Figure 4-6) and Microsoft SQL Server Profiler (shown in Figure 4-7 and found out they are exactly matched, so the footprint of custom logging facility is very minor and can be ignored. We will use it as the unique source for sample collection.

**Table 4-2 Sample performance sample data in a log file**

<table>
<thead>
<tr>
<th>Action</th>
<th>Action Type</th>
<th>Time</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Orchestration</td>
<td>9:26:28.250 PM</td>
<td>2/14/2013</td>
</tr>
<tr>
<td>Receive_ClientRequest</td>
<td>Receive</td>
<td>9:26:28.257 PM</td>
<td>2/14/2013</td>
</tr>
<tr>
<td>Receive_ClientRequest</td>
<td>Receive</td>
<td>9:26:28.260 PM</td>
<td>2/14/2013</td>
</tr>
<tr>
<td>Log1</td>
<td>VariableAssign...</td>
<td>9:26:28.260 PM</td>
<td>2/14/2013</td>
</tr>
<tr>
<td>Log1</td>
<td>VariableAssign...</td>
<td>9:26:28.260 PM</td>
<td>2/14/2013</td>
</tr>
<tr>
<td>ParallelActions_Quote</td>
<td>Parallel</td>
<td>9:26:28.263 PM</td>
<td>2/14/2013</td>
</tr>
<tr>
<td>Log_2</td>
<td>VariableAssign...</td>
<td>9:26:28.267 PM</td>
<td>2/14/2013</td>
</tr>
<tr>
<td>Log_2</td>
<td>VariableAssign...</td>
<td>9:26:28.270 PM</td>
<td>2/14/2013</td>
</tr>
</tbody>
</table>

**Figure 4-6 Logging information from BizTalk HAT tool**

**Figure 4-7 Logging information from SQL Profiler tool**
4.4 Auto testing

Since it is impossible and unnecessary to test the whole population, we will only test part of the population, called the sample, to calculate the sample mean and its accuracy to the population mean using confidence intervals. This means we need to collect large amounts of measurement data which can be thousands of items. It is infeasible to retrieve this amount of data by a manual process; so we used a load test tool called LoadUI to emulate the repeating of testing. We created a Load test project in LoadUI and referenced the SOAP UI just created, so that we can set up the number of repeats for the auto testing.

We needed to deliberately design the frequency of repeating the testing based on some initial SoapUI tests. We found out that the average response time for a normal size (1KB) of the BPEL Insurance Quotation web service is about 3 seconds. In order not to introduce competition for resources and queuing, we decided to launch a request per 10 seconds, so the rate was set at 6/Min. As shown in Figure 4-8, there are no concurrent calls to BPEL web service, so the burst size is 1 (the default value). We set the Request Limit to 3201 times, later the very first request will be removed since it will do the Just-In-Time compilation for the .NET Intermediate Language to binary code and will take longer time and cause bias to the sample data.
Figure 4-8 Use LoadUI to do auto and concurrent testing

After collecting the samples in memory, log files from different software components were generated. A tool was developed using C# language to combine all log files into one in order to automate the process of extracting the measurement data for analysis, as shown in Figure 4-9. The source code to combine all the dispersed log files is shown in Appendix.

Figure 4-9 Combine four log files into one log file
Later the combined test sample log file T3200.log was imported to a table called T3200 in MasterThesis SQL using SQL Server Import and Export Wizard as shown in Figure 4-10. The table structure is shown in Table 4-3. We import the raw sample data into a table with strong column name called Experiment1, which has the same number of columns as T3200 but having clarified the definition of what a column stands for, as shown in Table 4-4. Finally we used the SQL script shown in Table 4-5 to extract and transform sample test data from table Experiment1 to analysis table called LQNInputs,
where each field contains the execution time corresponding to the performance logging stages. The whole transformation process is shown in Figure 4-11.

**Table 4-3 Import raw sample test data into T3200 table**

```sql
CREATE TABLE [dbo].[T3200](
[Column 0] [varchar](50) NULL,
[Column 1] [varchar](50) NULL,
[Column 2] [varchar](50) NULL,
[Column 3] [varchar](50) NULL,
[Column 4] [varchar](50) NULL,
[Column 5] [varchar](50) NULL,
[Column 6] [varchar](50) NULL,
[Column 7] [varchar](50) NULL,
[Column 8] [varchar](50) NULL
) ON [PRIMARY]
GO
```

**Table 4-4 Import data from T3200 to a strong name Experiment1 table**

```sql
delete from dbo.Experiment1
--insert data from import table
insert into dbo.Experiment1(
logid,
sequence,
logtimestamp,
ProcessTotalTimeSpan,
ProcessUserTimeSpan,
ProcessKernelTimeSpan,
ThreadTotalTimeSpan,
ThreadUserTimeSpan,
ThreadKernelTimeSpan
)
select * from dbo.T3200
```

The grouping of logging entries for the same test is identified by the same LogID. We can group them by using SQL statements, as shown below.
Table 4-5 shows how to calculate the response time out of the samples collected in 26 monitoring spots; the results are stored into table LQNInputs.

**Table 4-5 Response Time Calculation from each execution step**

- BPELReceiveRequest: op1 - op0
- SunLifeBPELBUILDWSSendRequestXML: (op3 - op2) for Sun Life branch
- ManuLifeBPELBUILDWSSendRequestXML: (op13 - op12) for Manu Life branch
- SunLifeBPELSendWSSendRequest: (op10 - op3) for Sun Life branch
• ManuLifeBPELSendWSRequest: (op20 - op13) for Manu Life branch
• SunLifeWSReceiveMessage: (op4 - op3) for Sun Life branch
• ManuLifeWSReceiveMessage: (op14 - op13) for Manu Life branch
• SunLifeWSParseSOAP: (op5 - op4) for Sun Life branch
• ManuLifeWSParseSOAP: (op15 - op14) for Manu Life branch
• SunLifeWSEXecuteLogic: (op6 - op5) for Sun Life branch
• ManuLifeWSEXecuteLogic: (op16 - op15) for Manu Life branch
• SunLifeWSDatabaseExecution: (op7 - op6) for Sun Life branch
• ManuLifeWSDatabaseExecution: (op17 - op16) for Manu Life branch
• SunLifeWSDatabaseDiskAccess: (op8 - op7) for Sun Life branch
• ManuLifeWSDatabaseDiskAccess: (op8 - op7) for Sun Life branch
• SunLifeWSSendMessage: (op9 - op8) for Sun Life branch
• ManuLifeWSSendMessage: (op19 - op18) for Manu Life branch
• SunLifeBPELReceiveWSResponse: (op11 - op9) for Sun Life branch
• ManuLifeBPELReceiveWSResponse: (op21 - op19) for Manu Life branch
• BPELDecisionFolk: (op22 - op11) for Sun Life branch
• BPELConstructResponseXML: (op23 - op22) for Sun Life branch OR (op25 - op24) for Manu Life branch, they won't execute at the same time
• BPELSendReply: (op26 - op23) for Sun Life branch OR (op27 - op25) for Manu Life branch, they won't execute at the same time
• TotalResponseTime: (op26 - op1)

After the sample test data was populated to table “LQNInputs”, it is copied to an Excel spreadsheet, as we use Excel Data Analysis tool for analyzing the sample data.
4.5 Using Confidence Intervals for Data Analysis

Confidence intervals for the sample mean are an interval estimate of the real mean. Interval estimates are usually desirable, since the estimate of the mean varies from sample to sample. Instead of a single estimate of the mean, a confidence interval produces a lower and upper limit for the mean. The interval estimate provides a sign of how much uncertainty there is in the estimate of the real mean. The narrower the interval, the more accurate is our estimate. Confidence intervals are built at a confidence level selected by the user, for example 95%. It means that if an identical population is sampled on many occasions and interval estimates are made each time, the resulting intervals would bracket the real population parameter in approximately 95% of the cases. A confidence stated at a 1-\(\alpha\) level can be thought of as the inverse of the significance level, \(\alpha\) (Altman et al. 2000).

Confidence limits or intervals are defined as follows (Law and Kelton, 2000):

“If one constructs a very large number of independent percent confidence intervals each based on \(N\) observations, where \(N\) is sufficient large, the proportion of these confidence intervals that contain \(\theta\) should be 1-\(\alpha\). We call this proportion the coverage for the confidence interval. – Law and Kelton (2000)”

Approximate confidence intervals are given by the following equation:

\[
\bar{M} \pm t_{(\alpha/2, N-1)} \frac{\sigma}{\sqrt{N}} \tag{4-1}
\]

where \(\bar{M}\) is the sample mean, \(\sigma\) is the sample standard deviation, \(N\) is the sample size, \(\alpha\) is the desired significance level and \(t_{(\alpha/2, N-1)}\) is the upper critical value of the \(t\) distribution with \(N-1\) degrees of freedom. Note that 1-\(\alpha\) is the confidence coefficient.
As shown in Equation (4-1), it is clear that the width of the interval is controlled by two factors:

- As $N$ increases, the interval gets narrower from the $\sqrt{N}$ term. One way to obtain more precise estimates for the mean is to increase the sample size.
- The larger the sample standard deviation, the larger the confidence interval. This means that data with large standard deviation generates wider intervals than data with a smaller standard deviation.

The standard deviation is given by the following equation:

$$\sigma = \sqrt{\frac{\sum (M - \bar{M})^2}{N}}$$  \hspace{1cm} (4-2)

where $\bar{M}$ is the mean of all the samples of experiments, $M$ is a mean of one experiment in the data set and $N$ is the number of experiments. (Law and Kelton, 2000)

Here is the result based on 3200 sample test size and 1K request message length.

**Table 4-6 Confidence Interval Statistics for Orchestration Total Response Time**

<table>
<thead>
<tr>
<th>(X) [TotalResponseTime]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
<td>2791.732</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>338.9118</td>
</tr>
<tr>
<td>$t_{0.025}$ (unlimited 2.5% one tail probability):</td>
<td>1.96</td>
</tr>
<tr>
<td>$e(\pm 10)$</td>
<td>10</td>
</tr>
<tr>
<td>$N$:</td>
<td>4412.508</td>
</tr>
</tbody>
</table>

As shown in Table 4-6, for 1K request message length, the average response time (the time spent between logging point 1 and 26) is 2791.732 ms, while the standard deviation is 338.9118 ms. By repeating 4413 sample tests, we are confident that 95% of the chance the real population mean is within the interval of $\pm 10$ms of the sample mean 2791.732.
Since \( ME = t_{(a/2, N-1)} \frac{\sigma}{\sqrt{N}} \) is the Margin Error, in this case equal to 10 (ms), we can deduct:

\[
N = \left( t_{(a/2, N-1)} \frac{\sigma}{ME} \right)^2 = (1.96 \times \frac{338.9118}{10})^2 = 4413
\]

So the sample size should be 4413. We already have 3198 samples, we need 4413 – 3198 = 1215 more samples. But at the same time if we only have 3198 samples, we can deduct the Margin Error is 11.75ms, which is 0.8% of the sample mean and acceptable. So we don’t need to do further sample test.

After analysis on the entire sample test data for all performance parameters, as shown in Figure 4-12 and Figure 4-14, we noticed that:

- All the Receive Shapes need big sample size. We can see the distribution is flat which means the standard deviation is big and the response time is unpredictable, that match the nature of non-deterministic characteristics of the network communication.

- The standard deviation of execution time of BPEL activity shape is very small, which means the execution time of each sample test is very consistent.

In this thesis, we used different request message sizes to measure the platform overhead of Microsoft BizTalk Server and listed the measurement as reference for future developers and researchers. The following sections show the results for 1KB message size, other results are shown in APPENDIX.
4.6 Measurement results for the Web Server

Since our study case is a composed service, the web server plays an important role because it receives the service requests and sends the responses back. Here we measured 6 individual performance measures:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BPELReceiveRequest]</td>
<td>Time spent on receiving request to the insurance broker composed service</td>
</tr>
<tr>
<td>[BPELSendReply]</td>
<td>Time spent on sending response back from the insurance broker composed service</td>
</tr>
<tr>
<td>[ManuLifeWSReceiveMessage]</td>
<td>Time spent on receiving response from Manu Life service from the orchestration</td>
</tr>
<tr>
<td>[ManuLifeBPELSendWSRequest]</td>
<td>Time spent on sending request to Manu Life service from the orchestration</td>
</tr>
<tr>
<td>[SunLifeBPELReceiveRequest]</td>
<td>Time spent on receiving response from Sun Life service from the orchestration</td>
</tr>
<tr>
<td>[ManuLifeBPELSendWSRequest]</td>
<td>Time spent on sending request to Sun Life service from the orchestration</td>
</tr>
</tbody>
</table>

All the other parameters are listed in APPENDIX; here we use BPELReceiveRequest parameter as an example:
Table 4-8 Statistics of BPEL Receive Activity

<table>
<thead>
<tr>
<th>[BPELReceiveRequest]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
<td>331.4618512</td>
<td></td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>153.6102955</td>
<td></td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
<td></td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>N:</td>
<td>908.3195364</td>
<td></td>
</tr>
</tbody>
</table>

Table on the left shows the confidence interval calculation of [BPELReceiveRequest] parameter, you can see the standard deviation is quite big and it requires 908 samples to get the confidence of 95% with the stand error ±10ms.

Figure 4-12 shows the distribution chart of BPELReceiveRequest, it looks more bimodal than unimodal and the shape is flat, which indicates that the data is largely spread and not so many values falls in the 1 standard deviation. This matches the unpredictable characteristics of network traffic. The bimodal distribution is a cluster effect explained in section 4.10.

Table 4-9 Distribution for BPEL Receive Activity

<table>
<thead>
<tr>
<th>Bin</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.58544</td>
<td>0</td>
</tr>
<tr>
<td>109.4303</td>
<td>280</td>
</tr>
<tr>
<td>220.4461</td>
<td>642</td>
</tr>
<tr>
<td>331.4619</td>
<td>644</td>
</tr>
<tr>
<td>442.4776</td>
<td>724</td>
</tr>
<tr>
<td>553.4934</td>
<td>654</td>
</tr>
<tr>
<td>664.5091</td>
<td>249</td>
</tr>
<tr>
<td>775.5249</td>
<td>4</td>
</tr>
<tr>
<td>886.5407</td>
<td>0</td>
</tr>
<tr>
<td>997.5564</td>
<td>0</td>
</tr>
<tr>
<td>1108.572</td>
<td>1</td>
</tr>
<tr>
<td>1262.182</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4-12 BPEL Receive Request sample mean distribution
4.7 Measurement results for Manu Life Web Service

Manu Life web service is one of the services to be composed as part of the BPEL orchestration. In this thesis, we will measure the performance overhead for each building block of the final system to predict and confirm the final integration.

Table 4-10 Performance Measurement for Manu Life Web Service

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ManuLifeWSParseSOAP]</td>
<td>Time spent on parsing SOAP message in the web service</td>
</tr>
<tr>
<td>[ManuLifeWSExecuteLogic]</td>
<td>Time spent on executing the business logic for quoting</td>
</tr>
<tr>
<td>[ManuLifeWSDatabaseExecution]</td>
<td>Time spent on querying SQL database</td>
</tr>
<tr>
<td>[ManuLifeWSDatabaseDiskAccess]</td>
<td>Time spent on disk accessing</td>
</tr>
</tbody>
</table>

Here we use ManuLifeWSDatabaseExecution and ManuLifeWSDatabaseDiskAccess as an example.

Table 4-11 Statistics for Manu Life WS Database Execution Activity

<table>
<thead>
<tr>
<th></th>
<th>Table 4-11 Statistics for Manu Life WS Database Execution Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
<td>940.7667</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>287.0009</td>
</tr>
<tr>
<td>$t_{0.025}$ (unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>$e(\pm10)$</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>3170.769</td>
</tr>
</tbody>
</table>

shows the confidence interval calculation of the parameter [ManuLifeWSDatabaseExecution]. Its standard deviation is very big so it needs 3171 samples to gain the confidence of 95% with margin error $\pm10$ms. And from its distribution chart shown in Figure 4-13, it has bimodal shapes, where it doesn’t apply a normal distribution at all. The cluster effect
will be explained in 4.10.

Table 4-12 Statistics for Manu Life WS Database Execution Activity

<table>
<thead>
<tr>
<th>Bin</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>711</td>
<td>1</td>
</tr>
<tr>
<td>738.6429</td>
<td>60</td>
</tr>
<tr>
<td>766.2857</td>
<td>624</td>
</tr>
<tr>
<td>793.9286</td>
<td>768</td>
</tr>
<tr>
<td>821.5714</td>
<td>646</td>
</tr>
<tr>
<td>849.2143</td>
<td>237</td>
</tr>
<tr>
<td>876.8571</td>
<td>83</td>
</tr>
<tr>
<td>904.5</td>
<td>40</td>
</tr>
<tr>
<td>932.1429</td>
<td>21</td>
</tr>
<tr>
<td>959.7857</td>
<td>14</td>
</tr>
<tr>
<td>987.4286</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 4-13 Manu Life WS Database Execution Distribution

Table 4-13 Statistics for Manu Life WS Database Disk Access

<table>
<thead>
<tr>
<th>[ManuLife WS Database Disk Access]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>81.40806754</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>24.95656829</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>23.97550365</td>
</tr>
</tbody>
</table>

Table 4-13 shows the confidence interval calculation for parameter [ManuLifeWSDatabaseDisk], the standard deviation is quite small and only needs 24 samples in order to gain 95% confidence with margin error ±10ms. In its distribution chart shown in Figure 4-14 we can see that the shape is unimodal and it’s a normal distribution shape, even though it is skewed to the right tail a little bit.
By comparing the distribution of parameter [ManLifeWSDatabaseExecution] and [ManuLifeWSDatabaseDisk], we can see Database Disk Access is more consistent and easy to predict than [ManLifeWSDatabaseExecution], which will introduce more variance to the final composed system. By doing this pre-implementation analysis, it gives us insight to understand why part of the system is more vulnerable as the bottleneck of the final integration part.

4.8 Measurement results for Sun Life Web Service

Table 4-15 Performance Measurement for Sun Life Web service

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[SunLifeWSParseSOAP]</td>
<td>Time spent on parsing SOAP message in the web service</td>
</tr>
</tbody>
</table>
**Table 4-16 Statistics of Sun Life WS Database Execution**

| [SunLife WS Database Execution] |  
|---------------------------------|----------------------|
| Average                         | 1872.652             |
| Standard Deviation              | 277.0787             |
| t0.025(unlimited 2.5% one tail probability) | 1.962               |
| e(±10)                          | 10                   |
| N:                              | 2955.318             |

Consistent with Manu Life database execution, parameter [SunLifeWSDatabaseExecution] is quite variable that it needs 2955 samples to gain 95% confidence with margin error ±10ms. Its distribution chart also shows cluster bimodal shapes.

**Table 4-17 Distribution of Sun Life WS Database Execution**

<table>
<thead>
<tr>
<th>Bin</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1600</td>
<td>1</td>
</tr>
<tr>
<td>1626.339</td>
<td>0</td>
</tr>
<tr>
<td>1652.679</td>
<td>10</td>
</tr>
<tr>
<td>1679.018</td>
<td>171</td>
</tr>
<tr>
<td>1705.357</td>
<td>555</td>
</tr>
<tr>
<td>1731.696</td>
<td>779</td>
</tr>
<tr>
<td>1758.036</td>
<td>513</td>
</tr>
<tr>
<td>1784.375</td>
<td>222</td>
</tr>
<tr>
<td>1810.714</td>
<td>97</td>
</tr>
<tr>
<td>1837.054</td>
<td>75</td>
</tr>
<tr>
<td>1863.393</td>
<td>48</td>
</tr>
<tr>
<td>1889.732</td>
<td>21</td>
</tr>
<tr>
<td>1916.071</td>
<td>16</td>
</tr>
<tr>
<td>1942.411</td>
<td>3</td>
</tr>
<tr>
<td>1968.75</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 4-15 SunLife WS Database Execution Distribution**
Table 4-18 Statistics of Sun Life WS Database Execution

<table>
<thead>
<tr>
<th>[SunLifeWSDatabaseDiskAccess]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average:</strong></td>
<td>162.4534084</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>24.10300437</td>
</tr>
<tr>
<td><strong>t0.025 (unlimited 2.5% one tail probability):</strong></td>
<td>1.962</td>
</tr>
<tr>
<td><strong>e(±10)</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>N:</strong></td>
<td>22.36353044</td>
</tr>
</tbody>
</table>

Consistent with Manu Life service parameter [SunLifeWSDatabaseDiskAccess] is very predictable and only requires 22 samples to gain 95% confidence with margin error ±10ms and its distribution chart is unimodal with most of the sample values fall into 1 standard deviation region.

Table 4-19 Distribution of Sun Life WS Database Disk Access

<table>
<thead>
<tr>
<th>Bin</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>139</td>
<td>1</td>
</tr>
<tr>
<td>141.2857</td>
<td>5</td>
</tr>
<tr>
<td>143.5714</td>
<td>5</td>
</tr>
<tr>
<td>145.8571</td>
<td>128</td>
</tr>
<tr>
<td>148.1429</td>
<td>809</td>
</tr>
<tr>
<td>150.4286</td>
<td>712</td>
</tr>
<tr>
<td>152.7143</td>
<td>369</td>
</tr>
<tr>
<td>155</td>
<td>245</td>
</tr>
<tr>
<td>157.2857</td>
<td>99</td>
</tr>
<tr>
<td>159.5714</td>
<td>50</td>
</tr>
<tr>
<td>161.8571</td>
<td>43</td>
</tr>
<tr>
<td>164.1429</td>
<td>30</td>
</tr>
<tr>
<td>166.4286</td>
<td>13</td>
</tr>
<tr>
<td>168.7143</td>
<td>2</td>
</tr>
<tr>
<td>171</td>
<td>1</td>
</tr>
</tbody>
</table>

Parameter [SunLifeWSDatabaseExecution] is more variable and shows a bimodal distribution. This is because the database query is software related and has different execution plans based on the SQL server pre-compile process, it’s easier to introduce cluster effects (discussed in section 4.10); while [SunLifeWSDatabaseDiskAccess] is more hardware related and more predictable.
4.9 Measurement results of BPEL Orchestration

BPEL Orchestration shapes usually execute very fast, either several milliseconds or too fast that it cannot be measured by the aforementioned testing facilities in Table 4-1.

Use [ManuLifeBPELBuildWSRequestXML] as an example:

Table 4-20 Statistics for Manu Life BPEL Build Activity

<table>
<thead>
<tr>
<th>[ManuLifeBPELBuildWSRequestXML]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
<td>2.390243902</td>
<td></td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>2.551746328</td>
<td></td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
<td></td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>0.250653056</td>
<td></td>
</tr>
</tbody>
</table>

This BPEL shape has very small standard deviation and very consistent. From the distribution chart, we can see it has a very significant cluster effect.

Table 4-21 Distribution of Manu Life BPEL Build Activity

<table>
<thead>
<tr>
<th>Bin</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1070</td>
</tr>
<tr>
<td>1.125</td>
<td>0</td>
</tr>
<tr>
<td>2.25</td>
<td>0</td>
</tr>
<tr>
<td>3.375</td>
<td>1891</td>
</tr>
<tr>
<td>4.5</td>
<td>0</td>
</tr>
<tr>
<td>5.625</td>
<td>0</td>
</tr>
<tr>
<td>6.75</td>
<td>148</td>
</tr>
<tr>
<td>7.875</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10.125</td>
<td>66</td>
</tr>
<tr>
<td>11.25</td>
<td>0</td>
</tr>
<tr>
<td>12.375</td>
<td>0</td>
</tr>
</tbody>
</table>

... ...

Figure 4-17 Manu Life BPEL Build WS Request XML Distribution
4.10 Cluster effect

When analyzing the distribution charts shown above, we noticed that some performance service time distribution has cluster effects. Data from the whole study sample can be classified into a number of different groups, referred to as clusters (Galbraith et al., 2010). E.g. Sun Life WS Database Execution Distribution has a bimodal shape shown in Figure 4-15 and Manu Life BPEL build WS Request XML Distribution has multi-modal like ripple shapes as shown in Figure 4-17. This is probably caused by the accuracy of the Timer method. As (Kuperberg et al., 2009) mentioned, each timer method has its accuracy limitation. When we measured different service times, we used the Windows operating system’s Timer Methods like ProcessTotalTimeSpan etc., as mentioned in. A timer method is a software method that accesses a hardware timer, and its accuracy is described by (Lilja, 2000) as the “smallest incremental change that can be detected and displayed”. E.g. the unit of java.lang.System.nanoTime() is 1 ns, while in practice, its resolution and resulting accuracy of measurements) are often hundreds of ns (D. Holmes, 2006) and (M. Kuperberg et al., 2009).
Figure 4-18 Quantifying the time method accuracy

When the service time of the work to measure is smaller than the Timer method accuracy, the measurement will show 0ms as shown for Manu Life BPEL Build WS Request XML service time in Figure 4-17. In fact, many samples are recorded around 0ms. This doesn’t mean that building an XML request doesn’t take any execution time; on the contrary, it is just the accuracy of the time method is not fast enough so the counter has not been updated yet.

So the service time of a task either belongs to the previous time where the counter of the time method was updated or the next time it was updated, nowhere in between, which creates the effect of clustering.
4.11 BizTalk Server Platform Overheads

Table 4-22 shows our measurement results based on the following hardware configurations:

- **Processor**: AMD Athlon™ 64x2 Dual Core 4600+ 2.40 GHz
- **Memory (RAM)**: 8.00 GB
- **Operating System**: Windows 2008 Enterprise 32-bit

### Table 4-22 BizTalk BPEL Engine Platform related Overheads versus message size

<table>
<thead>
<tr>
<th>Activity</th>
<th>1KB (ms)</th>
<th>3KB (ms)</th>
<th>30K (ms)</th>
<th>300KB (ms)</th>
<th>3MB (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPEL Receive Request</td>
<td>331</td>
<td>348</td>
<td>430</td>
<td>497</td>
<td>1721</td>
</tr>
<tr>
<td>BPEL Construct Message</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>BPEL Send external message</td>
<td>49</td>
<td>52</td>
<td>64</td>
<td>74</td>
<td>255</td>
</tr>
<tr>
<td>BPEL Receive external message</td>
<td>433</td>
<td>455</td>
<td>565</td>
<td>651</td>
<td>2252</td>
</tr>
<tr>
<td>BPEL Parse SOAP message</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>Average Web Service Execution</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Average Database Execution</td>
<td>1873</td>
<td>1968</td>
<td>2440</td>
<td>2812</td>
<td>9742</td>
</tr>
<tr>
<td>Average Database Disk Access</td>
<td>162</td>
<td>170</td>
<td>212</td>
<td>243</td>
<td>849</td>
</tr>
<tr>
<td>Average Web Service Reply</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>BPEL Decision Fork</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>BPEL Send Response</td>
<td>23</td>
<td>24</td>
<td>30</td>
<td>35</td>
<td>123</td>
</tr>
<tr>
<td>(X) [TotalResponseTime]</td>
<td>2792</td>
<td>2931</td>
<td>3609</td>
<td>4076</td>
<td>14447</td>
</tr>
</tbody>
</table>

After gathering all the performance overheads for each building block of the SOA system, we can build a performance prediction model by using different modeling...
techniques like Queuing Networks, Layered Queuing Networks or Petri Nets etc. (Margaria et al., 2008).

The advantage of LQN over QN is that LQN can easily represent nested services. A server that serves clients’ requests could be in turn a client to other servers, and the returning results from these servers could be used to serve its own clients. The nested services are common for concurrent and distributed applications, especially in a SOA system like our study case in this thesis, which have complex interactions among software components, network, and hardware devices. So we selected Layer Queuing Networks as the modeling tool for performance prediction.
5 LQN Model

Measurements are important in a performance study (Smith, 2001). But measurements alone are not enough and have many limitations:

- Measurements can only be done in a limited set of experiments. Internal measures need suitable and powerful tools that can measure the demands of CPU, I/O, and logical resources. External measures need a suitable hardware set-up, so that we can measure the use of hardware resources by software.

- Concurrency also adds complexity to the performance measurements. Concurrency generates contentions for both physical and software resources. Waiting to access the resources could take a significant portion of the total response time under heavy workloads. As the concurrency level increases, contention for the system’s resources would dominate the response time of a process.

- Performance for various workloads may be very time consuming, so the measurements can take a long time. In today’s highly competitive market, there is no time for such thorough tests.

- In some cases, end-to-end performance measurement is not possible. In these cases, performance models are the only options for studying the performance.

Performance models are very useful to evaluate the overall performance of software systems during software analysis and design stages. Modeling could extend to explore those scenarios that are difficult or impossible to measure directly. Moreover, modeling could provide some guidelines on how to achieve higher performance, even if there is no real system implemented yet.
In this chapter, we will build a LQN model for Insurance Broker Service BPEL Orchestration and predict the performance of the system at various workloads. The LQN results will be compared with the measurements. After validation, the LQN model is used for exploring the system performance for deployments and configurations that could not be used in measurements.

5.1 Building the LQN model

Performance of a software system is determined by how the system behavior uses system resources (Petriu, 2012). Scenarios define execution paths with externally visible end points, to which we can attach performance requirements, such as response time, throughput. The resources used by the software can be active or passive, logical or physical software or hardware. Some of these resources belong to the software itself (e.g., critical section, software server, lock, buffer), others to the underlying platforms (e.g., process, thread, processor, disk, communication network).

Each scenario is composed from scenario steps joined by predecessor-successor relationships, which may include fork/join, branch/merge and loops. A step may represent an elementary operation or a whole sub-scenario. Quantitative resource demands for each step must be given in the performance annotations. Each scenario is executed by a workload, which may be open (i.e., requests arriving in some predetermined pattern) or closed (a given number of users or jobs).

Performance information regarding the resources and their use can be added to UML model with the help of specialized profiles. OMG has defined and adopted two standard profiles which provide, among other features, the ability to define performance annotations: the UML Profile for Schedulability, Performance and Time (SPT) defined
for UML 1.X versions (OMG, 2005) and the *UML Profile for Modeling and Analysis of Real-Time and Embedded systems* (MARTE) defined for UML2.X versions (OMG, 2009).

An LQN model of a SOA system can be obtained from a UML+MARTE software model of the same system which contains the following views: a structural view representing the software architecture at the granularity level of concurrent runtime components and their allocation to hardware resources, as well as a behavioral view showing representative scenarios with their respective resource usage and workloads.

We consider the scenario represented by the Activity Diagram from Figure 5-1, which shows the behavior of the Insurance Broker system that we intend to model in LQN. The Activity diagram is divided into partitions. Each partition stereotyped with «PaRunTInstance» contains the activities executed by a concurrent component. The activity edges that cross from a partition to another represent requests that are made from a component to another. Each activity represents a scenario Step and is stereotyped with «PaStep». Examples of execution step attributes are *hostDemand* giving the value and unit for the required execution time and *prob* giving the probability for optional steps.
Figure 5-1 MARTE annotated Activity Diagram
The performance of a scenario is analyzed under a given workload, represented by a stream of triggering events \textit{WorkloadEvent}. In our example, the scenario is driven by a closed workload with a given number of concurrent users $N_{users}$.

As shown in the Activity Diagram with platform overhead shown in Figure 3-4, the annotated diagram from Figure 5-1 (which was derived from Figure 3-4), contains both SOA application execution times as well as and BizTalk platform overheads.

According to (Petriu, 2010), an annotated UML+MARTE model can be transformed into an LQN model by doing the following mappings:

- Map concurrent component instances to LQN tasks;
- Map deployment processors nodes to LQN processors;
- Map a service offered by a component to an LQN entry
- Map UML activities either to LQN activities or phases.

Figure 5-2 shows the LQN model for the Insurance Broker Quotation BPEL Orchestration, which is composed of services and exposes the composition as a new service. It is a nested service system, so it is a good candidate for LQN modeling constructed manually based on this MARTE annotated model, by following the approach presented by (Petriu, 2010).
Figure 5-2 LQN model
The execution demands of different activities stereotyped with «PaStep» in Figure 5-1 were obtained from the measurements obtained in Chapter 4. These values are used to compute the service time parameters of the LQN model. The following sample averages shown in Table 5-1 were used as the input parameters for the LQN input file shown in APPENDIX.

**Table 5-1 Service Time for each step in BPEL Orchestration**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Service Time (ms) for 1KB request size</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B) [BPELReceiveRequest]</td>
<td>331.4618512</td>
</tr>
<tr>
<td>(C) [SunLifeBPELBuildWSRequestXML]</td>
<td>3.174769622</td>
</tr>
<tr>
<td>(D) [SunLifeBPELSendWSRequest]</td>
<td>49.29049406</td>
</tr>
<tr>
<td>(E) [SunLifeWSReceiveMessage]</td>
<td>432.5059412</td>
</tr>
<tr>
<td>(F) [SunLifeWSParseSOAP]</td>
<td>2.823639775</td>
</tr>
<tr>
<td>(G) [SunLifeWSEexecuteLogic]</td>
<td>1.675422139</td>
</tr>
<tr>
<td>(H) [SunLifeWSDatabaseExecution]</td>
<td>1872.651657</td>
</tr>
<tr>
<td>(I) [SunLifeWSDatabaseDiskAccess]</td>
<td>162.4534084</td>
</tr>
<tr>
<td>(J) [SunLifeWSSendMessage]</td>
<td>1.469981238</td>
</tr>
<tr>
<td>(K) [SunLifeBPELReceiveWSResponse]</td>
<td>281.41995</td>
</tr>
<tr>
<td>(L) [ManuLifeBPELBuildWSRequestXML]</td>
<td>2.390243902</td>
</tr>
<tr>
<td>(M) [ManuLifeBPELSendWSRequest]</td>
<td>38.01250782</td>
</tr>
<tr>
<td>(N) [ManuLifeWSReceiveMessage]</td>
<td>408.362414</td>
</tr>
<tr>
<td>(O) [ManuLifeWSParseSOAP]</td>
<td>2.563789869</td>
</tr>
<tr>
<td>(P) [ManuLifeWSEexecuteLogic]</td>
<td>1.867729831</td>
</tr>
<tr>
<td>(Q) [ManuLifeWSDatabaseExecution]</td>
<td>940.7667292</td>
</tr>
<tr>
<td>(R) [ManuLifeWSDatabaseDiskAccess]</td>
<td>81.40806754</td>
</tr>
<tr>
<td>(S) [ManuLifeWSSendMessage]</td>
<td>1.561913696</td>
</tr>
<tr>
<td>(T) [ManuLifeBPELReceiveWSResponse]</td>
<td>350.4693558</td>
</tr>
<tr>
<td>(U) [BPELDecisionFolk]</td>
<td>4.360225141</td>
</tr>
<tr>
<td>(V) [BPELConstructResponseXML]</td>
<td>1.986866792</td>
</tr>
<tr>
<td>(W) [BPELSendReply]</td>
<td>22.93370857</td>
</tr>
<tr>
<td>(X) [TotalResponseTime]</td>
<td>2791.73202</td>
</tr>
</tbody>
</table>
5.2 LQN Model Validation

After building the LQN model, we validated it by comparing its response time results with measured values for the same response time. We considered two configurations:

- Vary the number of concurrent users from 1 to 35, for a configuration where all tasks are single-threaded and are running on a single processor with two cores. We selected up to 35 users because the system gets very saturated for 35 concurrent users. Since the operating point for the system is chosen below saturation, we decided not to continue for more than 35 users.

- Vary the message size being transmitted between the BPEL orchestrations and keep a single user; all tasks are single-threaded and running on a 2-core processor. The message size being transmitted was: 1KB, 3KB, 30KB, 300KB and 3MBas per message.

We run the BizTalk composed service with SOAP UI and then automated the testing with LOAD-UI. For concurrent testing, we used LoadUI (running on a different machine than the SOA system being measured) to generate the workload. Since we are trying to emulate the end user’s random pattern of sending requests, we picked the “Random Generator” following the Exponential distribution, within 1 minute there will be multiple requests being sent out randomly and it will repeat the same process again in a per minute basis as shown in Figure 5-3.
For the first series of tests with 1 to 35 concurrent users, the comparison between LQN results and measurements is given in Table 5-3 and Figure 5-4. We can see that the model and measured values are quite close, but they begin to diverge for more than 30 users, when the system is very saturated.

**Table 5-3. Comparison of LQN results and Performance Measurement for concurrent users**

<table>
<thead>
<tr>
<th>N</th>
<th>Measured RT</th>
<th>LQN Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2750</td>
<td>3112.61</td>
</tr>
<tr>
<td>5</td>
<td>12390</td>
<td>15260.94</td>
</tr>
<tr>
<td>10</td>
<td>30156.2</td>
<td>30804.74</td>
</tr>
<tr>
<td>15</td>
<td>45012.6</td>
<td>46448.14</td>
</tr>
<tr>
<td>20</td>
<td>57852.3</td>
<td>62109.54</td>
</tr>
<tr>
<td>25</td>
<td>72636.8</td>
<td>77580.74</td>
</tr>
<tr>
<td>30</td>
<td>91583.6</td>
<td>92987.44</td>
</tr>
<tr>
<td>35</td>
<td>125683.6</td>
<td>108604.54</td>
</tr>
</tbody>
</table>
Figure 5-4 Comparison of LQN results and Actual measurement for concurrent testing

For the second set of tests, we compared the LQN results for a single user, while varying the size of the messages exchanged when invoking services. The system has the same configuration as before: all software tasks are single threaded, and they are running on a single processor with two cores. The model and measured results given in Table 5-4 and Figure 5.5 are pretty close.

Table 5-2 Comparison of LQN model prediction and Performance Measurement for one user and different message sizes

<table>
<thead>
<tr>
<th>Message Size (KB)</th>
<th>Predicted (ms)</th>
<th>Measured (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2800</td>
<td>2791</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
<td>2931</td>
</tr>
<tr>
<td>30</td>
<td>3600</td>
<td>3609</td>
</tr>
<tr>
<td>300</td>
<td>4000</td>
<td>4076</td>
</tr>
<tr>
<td>3000</td>
<td>15000</td>
<td>14447</td>
</tr>
</tbody>
</table>
5.3 Using the LQN model for performance analysis

This section illustrates the use of the LQN model for further performance analysis of the system, for different deployments and configurations. We start from the Case A, which represents the system in the configuration used for the measurements reported in the previous section: single-threaded tasks, all running on a 2-core processor.

Figure 5-6 represents the response time in function of the number of users, and Figure 5-7 the utilizations of the tasks and the processor they are running on, the orange line shows the task utilization of BPEL Orchestration, which is slightly higher than 100% around 5 concurrent users, that is caused by the LQN solver which did not converge and caused model error. The processor has two cores; therefore its utilization can be maximum 2, while all the tasks are single-threaded, and so their utilization can be
maximum 1. An inspection of the utilization graph shows that the system bottleneck is task BPEL1 (which corresponds to the main BPEL orchestration. This is a case of software bottleneck, and the solution is task multithreading.

![Figure 5-6 LQN Response Time versus Number of users for single threaded tasks running on 1 (2-core) processor](image)

**Figure 5-6** LQN Response time in function of the number of users for Case A
The next case, B attempts to mitigate the software bottleneck from Case A by multi-threading the tasks that are saturated or highly utilized. In order to simplify our experiments, we decided to multi-thread all tasks up to 30 threads. At the same time, we distributed the deployment, by using three processors with 2 cores each: BPEL1 is running on Proc1, BPEL2 (the ManuLife service) and its database ML-DB on Proc2, and BPEL3 (the SunLife service) and its database SL-DB on Proc3. Also, we added a network between the processors, with an average message latency of 100 ms for each message (i.e., each service request and each reply). This is a rather simple way of treating the network delay, which works pretty well when for short messages and non-saturated network, as is the case here. The LQN results for Case B are shown in Figure 5-8 (response time), Figure 5-9 (task utilization) and Figure 5-10 (processor utilization).
Response Time Vs Number of users (multithreaded tasks running on 3 (4-core) processors)

![Graph showing response time vs number of users for Case B with different thread counts.](image)

**Figure 5-8 LQN Response Time for Case B**

Utilization of tasks (each 30 threads, running on 3 (2-core) processors)

![Graph showing task utilization for Case B with different task types.](image)

**Figure 5-9 Task Utilization for Case B**
The first observation is that multi-threading the tasks helps, as the response time improves for more threads. The effect has diminishing returns, in the sense that adding the second thread helps a lot, but after 10 threads the improvement due to every additional thread is very small. By inspecting the utilization graphs, we realize that the software tasks BPEL1 and BPEL2, and the processor Proc2 saturate first. As the lowest saturated task among them is the bottleneck, we conclude that the system bottleneck is Proc2, which affects in turn the utilization of BPEL2, which affects BPEL1. This is a case of hardware bottleneck, and a solution is to add more computing power.

Case C attempts to mitigate the hardware bottleneck by replacing the 2-core processors with 4-core ones. The effect on response time for 30 threads is shown in Figure 5-11, as which compared case C and Case B. Figure 5-12 shows again that we are in a hardware bottleneck case, where Proc2 is the bottleneck.
Figure 5-11. Response time for Case C compared to Case B

Response time (30 threads for each task, running on 3 (4-core) Vs. 3 (2-core) processors)

Figure 5-12 Processor utilization for Case C

Processor Utilizations (30 threads for each task, running on 3 (4-core) processors)
A solution to this problem should attempt either to lower the load on Proc2 or to increase its processing power by replacing the processor with a faster one or by adding more cores. From the description of the case study, we can see that the database for the Sun Life service was made slower on purpose. A solution would be to optimize the database by indexing the tables. Further solutions can be tried, finding the bottleneck at every step and then taking measures to relieve it, until the performance requirements would be met.

This section illustrated how a LQN model can be useful to designers for assessing the performance effects of different solution alternatives in the early development phases, before they commit to decisions that would be detrimental to performance and would be difficult to undo later.
6 Conclusions

The objective of the thesis is to measure the performance of a typical BPEL Orchestration solution in order to facilitate Software Performance Engineering in early development stages. BPEL Orchestration composes services, which means it combines more resources and requires longer execution times and introduces further system performance deterioration. Usually software architects and developers do not have a clear understanding of the performance overheads of service composition and will ignore them in the early software design phases. This thesis addresses this issue by measuring the following BPEL Orchestration overheads:

- Message performance cost for send and receive adapters for different message sizes and workloads;
- Performance cost for different BPEL elements like Condition, Branching and Parallel Fork, etc;
- Performance cost for each child service which is part of the composed SOA service by using BPEL Orchestration;
- Performance cost for database engine execution and disk access;
- The number of samples for the measured data was determined by using confidence intervals. Sample means of different BPEL Orchestration elements where used as parameters to construct an LQN model for the case study system and to validate it against measurements taken under different workloads and message sizes.

We designed and implemented a hybrid measuring techniques by using system tools to validate the footprint of custom logging facilities designed for SOA systems. Later we
used this custom logging to automate the testing process and data gathering for different SOA components.

We built LQN model using sample means as input parameters and showed how the LQN model can be used for further performance analysis of the system, for different deployments and configurations. This can be useful to designers for assessing the performance effects of different solution alternatives in the early development phases, before the system is entirely built and can be measured.

6.1 Limitations of the work

The case study in this thesis is relatively simple compared to real world SOA systems. We only consider factors like concurrent users and message sizes, but there are other important factors like network conditions, load balancing, server memory and CPU speed. An important limitation of our work was that we have not focused on the effect of the communication network on the system performance, but used a very simple solution. This is an item for future research.

We only focused on certain BPEL activities like receiving requests, invoking external web services and sending reply, but there are other important activities like message mapping (transfer from one message format to another) and security encryption and decryption impact we have not tested. We observed bi-modal cluster phenomenon in overheads like SQL database query and total response time, and proposed an explanation that it was caused by the accuracy of System Timer Method, but we did not dig into more detail to validate our hypothesis using convincing measurement data. Future work is necessary to overcome these limitations.
6.2 Future work

This work only represents a first step for performance measurements and evaluations of BPEL Orchestration. There are various dimensions for testing and analysis in the following foreseeable directions:

- Use a variety of SOA systems to model them in LQN and perform a more thorough validation.
- Performance measurements and modeling can be applied to other BPEL elements not used in this thesis.
- Look for tools and logging facilities to measure resource demands and performances results like the BPEL engine, where no source code is available.
- Study in more depth the effect of communication network on SOA systems performance.
- Use the testing and modeling methodologies in this thesis to see if a practical performance modeling tool can be developed for the performance prediction of BPEL Orchestrations.
## Appendix

1. **Input parameters for LQN Model for 1KB request message size:**

<table>
<thead>
<tr>
<th>(X) [TotalResponseTime]</th>
<th>(B) [BPEL.ReceiveRequest]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average:</strong></td>
<td>2791.732</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>338.9118</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.96</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>4412.508</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>331.4618512</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>153.6102955</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>908.3195364</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(C) [SunLifeBPEL.BuildWSRequestXML]</th>
<th>(D) [SunLifeBPEL.SendWSRequest]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average:</strong></td>
<td>3.180113</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>2.942056</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>0.333196</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>49.29049406</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>23.03314669</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>20.42229536</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(E) [SunLifeWS.ReceiveMessage]</th>
<th>(F) [SunLifeWSParseSOAP]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average:</strong></td>
<td>432.5059</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>146.1256</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>821.9597</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>2.823639775</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>3.83734562</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>0.566839152</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(G) [SunLifeWS.ExecuteLogic]</th>
<th>(H) [SunLifeWSDatabaseExecution]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average:</strong></td>
<td>1.675422139</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>1.014565</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>0.039623949</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>1872.652</td>
</tr>
<tr>
<td><strong>Standard Deviation:</strong></td>
<td>277.0787</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>2955.318</td>
</tr>
<tr>
<td>(I) [SunLifeWSDatabaseDiskAccess]</td>
<td>(J) [SunLifeWSSendMessage]</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Average:</td>
<td>162.4534</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>24.103</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>22.36353</td>
</tr>
<tr>
<td>Average:</td>
<td>1.469981238</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>2.027232351</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>0.158199484</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(K) [SunLifeBPELReceiveWSResponse]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
</tr>
<tr>
<td>Standard Deviation:</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
</tr>
<tr>
<td>e(±10)</td>
</tr>
<tr>
<td>N:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(L) [ManuLifeBPELBuildWSRequestXML]</th>
<th>(M) [ManuLifeBPELSendWSRequest]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
<td>2.390244</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>2.551746</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>0.250653</td>
</tr>
<tr>
<td>Average:</td>
<td>38.01250782</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>17.79773316</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>12.19347208</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(N) [ManuLifeWSReceiveMessage]</th>
<th>(O) [ManuLifeWSParseSOAP]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
<td>408.3624</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>167.5773</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>1081.007</td>
</tr>
<tr>
<td>Average:</td>
<td>2.563789869</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>3.713482897</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>0.530836604</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(P) [ManuLifeWSEncryptWSMessage]</th>
<th>(Q) [ManuLifeWSDatabaseExecution]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average:</td>
<td>1.867729831</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>1.011252</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>0.039365593</td>
</tr>
<tr>
<td>Average:</td>
<td>940.7667</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>287.0009</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability):</td>
<td>1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>10</td>
</tr>
<tr>
<td>N:</td>
<td>3170.769</td>
</tr>
<tr>
<td>(R) [ManuLifeWSDatabaseDiskAccess]</td>
<td>(S) [ManuLifeWSSendMessage]</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Average: 81.40807</td>
<td>Average: 1.561913696</td>
</tr>
<tr>
<td>Standard Deviation: 24.95657</td>
<td>Standard Deviation: 2.255167568</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability): 1.962</td>
<td>t0.025(unlimited 2.5% one tail probability): 1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>e(±10)</td>
</tr>
<tr>
<td>N: 23.9755</td>
<td>N: 0.195774282</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(T) [ManuLifeBPELReceiveWSResponse]</th>
<th>(U) [BPELDecisionFolk]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average: 408.5743</td>
<td>Average: 2791.168025</td>
</tr>
<tr>
<td>Standard Deviation: 167.5822</td>
<td>Standard Deviation: 338.6829424</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability): 1.962</td>
<td>t0.025(unlimited 2.5% one tail probability): 1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>e(±10)</td>
</tr>
<tr>
<td>N: 1081.069</td>
<td>N: 4415.548449</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(V) [BPELConstructResponseXML]</th>
<th>(W) [BPELSendReply]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average: 1.986866792</td>
<td>Average: 22.93370857</td>
</tr>
<tr>
<td>Standard Deviation: 1.967792627</td>
<td>Standard Deviation: 22.1267282</td>
</tr>
<tr>
<td>t0.025(unlimited 2.5% one tail probability): 1.962</td>
<td>t0.025(unlimited 2.5% one tail probability): 1.962</td>
</tr>
<tr>
<td>e(±10)</td>
<td>e(±10)</td>
</tr>
<tr>
<td>N: 0.149058472</td>
<td>N: 18.84657375</td>
</tr>
</tbody>
</table>

2. LQN Model Input File

```
#BPEL-1 : calls Insurance Broker BPEL Orchestration
#Most phases and act. are det: f=1; c=0. Lqns 5.7.9 does not work if all have c=0.
$N=[1,5,10,15,20,25, 30, 35, 40, 45, 50]

G "3 activity-described BPELs; all tasks single-threaded -> 1 CPU (2-core) & two disks"
.00001 400 1 0.9 -1
P 0
p UserP f i
p CPU f m 2 %u $up
p DiskP2 f %u $ud2
p DiskP3 f %u $ud3
-1

# Task Information: (the zero is necessary; it may be replaced by the number of tasks)
# SYNTAX: t TaskName RefFlag EntryList -1 ProcessorName [multiplicity]
# TaskName is any string, globally unique among tasks
# RefFlag = r (reference or user task)
# | n (other)
# multiplicity = m value (multithreaded)
# | i (infinite)
T 0
t User r user -1 UserP m $N %f $thru # $N reference tasks; collect thruput
t Bpel1 n bpe11 -1 CPU %u $uBpe11
t SLbpel n slbpel -1 CPU %u $uBpe2
SLDB n slquery -1 CPU %u $uSLdb
t SLDisk n sldiskaccess -1 DiskP2
t MLbpel n mlbpel -1 CPU %u $uBpe3
t MLDB n mlquery -1 CPU %u $uMLdb
```
t MLDisk n mldiskaccess -1 DiskP3
-1

#Entry Information: (the zero is necessary; it may also give the total number of entries)
E 0
# SYNTAX-FORM-A: Token EntryName Value1 [Value2] [Value3] -1
# EntryName is a string, globally unique over all entries
# Values are for phase 1, 2 and 3 (phase 1 is before the reply)
# Token indicate the significance of the Value:
# s HostServiceDemand for EntryName
# c HostServiceCoefficientofVariation
# f PhaseTypeFlag
# SYNTAX-FORM-B: Token FromEntry ToEntry Value1 [Value2] [Value3] -1
# Token indicate the Value Definitions:
# y SynchronousCalls (no. of rendezvous)
# F ProbForwarding (forward to ToEntry rather than replying)
# z AsynchronousCalls (no. of send-no-reply messages)
# o Fanout (for replicated servers)(ignore this)
# i FanIn (for replicated servers)(ignore this)
# ----- User -----
s user 331.46 -1 %s1 $rt
y user bpel1 1 -1 #one request to the server per cycle
# ----- Bpel1 Workflow -----
A bpel1 RecRequest # for entry containing activities: give the first activity
A slbpel slreceive
A mlbpel mlreceive
# ----- SunLifeDB -----
s slquery 1872.651657 -1
f slquery 1 -1
c slquery 0 -1
y slquery sldiskaccess 1 -1
# ----- SunLifeDisk -----
s sldiskaccess 162.4534084 -1
f sldiskaccess 1 -1
c sldiskaccess 0 -1
# ----- ManuLifeDB -----
s mlquery 940.7667292 -1
f mlquery 1 -1
c mlquery 0 -1
y mlquery mldiskaccess 1 -1
# ----- ManuLifeDisk -----
s mldiskaccess 81.40806754 -1
f mldiskaccess 1 -1
c mldiskaccess 0 -1
-1
# Optional sections for definition of activities
# One section for each task that has activities, beginning A TaskName
# list of activity parameters, using the syntax for entry parameters, but
# with just one value and no terminator -1
# : (separator), then a section for precedence among activities
# Syntax for precedence:
# # a1 -> a2 for sequence
# # a1 -> a2 & a3 ... AND-fork (any number)
# # a1 & a2 ... -> a3 AND join
# # a1 & a2 ... -> a3 & a4 ... AND join followed by AND fork
# # a1 -> (prob2)a2 + (prob3)a3 ... OR fork (any number, with probabilities)
# # a1 -> meanCount*a2,a3,...for a repeated activity a2, followed by a3
# # (notice that activities that follow a2 are inside the loop)
# # a6[entryName] indicates that after a6, a reply will be sent to entryName
A Bpel1
s RecRequest 331.46 #every activity must have a host demand

100
f RecRequest 1    # deterministic
  c RecRequest 0
  s BuildXML 6
  f BuildXML 1
  c BuildXML 1
  s CallSL 0.0001
  f CallSL 1    # deterministic activity
  c CallSL 0
  y CallSL slbpel 1 # call SunLife service exactly once
  s CallML 0.0001
  f CallML 1    # deterministic activity
  c CallML 0
  y CallML mlbpel 1 # call ManuLife service exactly once (deterministic)
  s ConstructResponseXml 4.360225141
  f ConstructResponseXml 1
  c ConstructResponseXml 0
  s SendReply 22.93370857
  f SendReply 1
  c SendReply 0
:
  RecRequest -> BuildXML;
  BuildXML -> CallSL & CallML;
  CallML & CallSL -> ConstructResponseXml;
  ConstructResponseXml -> SendReply;
  SendReply[bpel1] # reply of the respective [entry] sent back to client task
-1 #
  A SLbpel
  s slreceive 432.5059412
  f slreceive 1
  c slreceive 0
  s slparse 2.823639775 # parse soap activity
  f slparse 1
  c slparse 0
  s slwscode 1.675422139 # web service business logic
  f slwscode 1
  c slwscode 0
  y slwscode slquery 1 # Call DB with SQL query
  s slwsresponse 281.41995 # send back SL service response
  f slwsresponse 1
  c slwsresponse 0
:
  slreceive -> slparse;
  slparse -> slwscode;
  slwscode -> slwsresponse;
  slwsresponse[slbpel]
-1
  A MLbpel
  s mlreceive 408.362414
  f mlreceive 1
  c mlreceive 0
  s mlparse 2.563789869 # parse soap
  f mlparse 1
  c mlparse 0
  s mlwscode 1.867729831 # web service business logic
  f mlwscode 1
  c mlwscode 0
  y mlwscode mlquery 1 # call ManuLife DB
  s mlwsresponse 350.4693558
  f mlwsresponse 1
  c mlwsresponse 0
:

101
mlreceive -> mlparse;
mlparse -> mlwscode;
mlwscode -> mlwsresponse;
mlwsresponse[mlbpel]
-1
#
$0=$N
$thru $rt $uBpel1 $uBpel2 $uSLdb $uBpel3 $uMLdb $up $ud2 $ud3
-1

3. Custom logging facility source code

```csharp
public void WriteLog(string logId, int order)
{
    TextWriterTraceListener traceListener = new TextWriterTraceListener(@"c:\temp\BPELOrchestrationLog.txt");
    ProcessThread currentProcessThread = GetCurrentProcessThread();
    #region log request start QoS
    StringBuilder sb = new StringBuilder();
    sb.Append(logId.ToString());
    sb.Append("t");
    sb.Append(order.ToString());
    sb.Append("t");
    sb.Append(DateTime.Now.ToString("MM/dd/yyyy hh:mm:ss.fff tt"));
    sb.Append("t");
    sb.Append(currentProcess.TotalProcessorTime.TotalMilliseconds);
    sb.Append("t");
    sb.Append(currentProcess.UserProcessorTime.TotalMilliseconds);
    sb.Append("t");
    sb.Append(currentProcess.PrivilegedProcessorTime.TotalMilliseconds);
    sb.Append("t");
    sb.Append(currentProcessThread.TotalProcessorTime.TotalMilliseconds);
    sb.Append("t");
    sb.Append(currentProcessThread.UserProcessorTime.TotalMilliseconds);
    sb.Append("t");
    sb.Append(currentProcessThread.PrivilegedProcessorTime.TotalMilliseconds);

    traceListener.WriteLine(sb.ToString());
    traceListener.Flush();
    traceListener.Close();

    #endregion
}
```

4. Combine log files source code

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;

namespace CombineLogFilesIntoOne
{
```
class Program
{
    static void Main(string[] args)
    {
        string[] filePaths = Directory.GetFiles(@"C:\temp\");

        int counter = 0;
        string line;

        using (System.IO.StreamWriter wfile = new System.IO.StreamWriter(@"C:\temp\CU10-Int20s.txt"))
        {
            foreach (string filePath in filePaths)
            {
                // Read the file and display it line by line.
                System.IO.StreamReader rfile =
                    new System.IO.StreamReader(filePath);
                while ((line = rfile.ReadLine()) != null)
                {
                    wfile.WriteLine(line);
                    counter++;
                }

                Console.WriteLine(filePath + " read.");
                rfile.Close();
            }
            wfile.Close();
            Console.ReadLine();
        }
    }
}
## 5. Concurrent User Testing Results

<table>
<thead>
<tr>
<th>Concurrent User Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPEL.ReceiveRequest</td>
<td>136</td>
<td>570</td>
<td>300</td>
<td>1116</td>
<td>3670</td>
<td>1026</td>
<td>6523</td>
<td>5676</td>
<td>1126</td>
</tr>
<tr>
<td>SunLifeBPEL.BuildWSRequestXML</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>SunLifeBPEL.SendWSRequest</td>
<td>90</td>
<td>143</td>
<td>153</td>
<td>123</td>
<td>133</td>
<td>166</td>
<td>513</td>
<td>206</td>
<td>126</td>
</tr>
<tr>
<td>SunLifeWS.ReceiveMessage</td>
<td>136</td>
<td>773</td>
<td>2196</td>
<td>653</td>
<td>2020</td>
<td>656</td>
<td>2490</td>
<td>966</td>
<td>323</td>
</tr>
<tr>
<td>SunLifeWS.ParseSOAP</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>130</td>
</tr>
<tr>
<td>SunLifeWS.ExecuteLogic</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SunLifeWS.DatabaseExecution</td>
<td>1916</td>
<td>3768</td>
<td>2382</td>
<td>1849</td>
<td>1753</td>
<td>1748</td>
<td>1677</td>
<td>1695</td>
<td>1762</td>
</tr>
<tr>
<td>SunLifeWS.DatabaseDiskAccess</td>
<td>166</td>
<td>327</td>
<td>207</td>
<td>160</td>
<td>152</td>
<td>145</td>
<td>147</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>SunLifeWS.SendMessage</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>SunLifeBPEL.ReceiveWSResponse</td>
<td>473</td>
<td>156</td>
<td>406</td>
<td>4316</td>
<td>6096</td>
<td>12523</td>
<td>15886</td>
<td>19743</td>
<td>22136</td>
</tr>
<tr>
<td>ManuLifeBPEL.BuildWSRequestXML</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ManuLifeBPEL.SendWSRequest</td>
<td>53</td>
<td>1136</td>
<td>3413</td>
<td>6856</td>
<td>9893</td>
<td>15046</td>
<td>19673</td>
<td>22336</td>
<td>24380</td>
</tr>
<tr>
<td>ManuLifeWS.ReceiveMessage</td>
<td>300</td>
<td>2360</td>
<td>3543</td>
<td>7316</td>
<td>10553</td>
<td>15260</td>
<td>20030</td>
<td>22626</td>
<td>24993</td>
</tr>
<tr>
<td>ManuLifeWS.ParseSOAP</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>120</td>
</tr>
<tr>
<td>ManuLifeWS.ExecuteLogic</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ManuLifeWS.DatabaseExecution</td>
<td>824</td>
<td>851</td>
<td>1637</td>
<td>2247</td>
<td>803</td>
<td>2293</td>
<td>800</td>
<td>791</td>
<td>2241</td>
</tr>
<tr>
<td>ManuLifeWS.DatabaseDiskAccess</td>
<td>71</td>
<td>74</td>
<td>142</td>
<td>195</td>
<td>69</td>
<td>199</td>
<td>69</td>
<td>68</td>
<td>194</td>
</tr>
<tr>
<td>ManuLifeBPEL.ReceiveWSResponse</td>
<td>540</td>
<td>330</td>
<td>480</td>
<td>143</td>
<td>710</td>
<td>513</td>
<td>130</td>
<td>426</td>
<td>500</td>
</tr>
<tr>
<td>BPEL.DecisionFolk</td>
<td>0</td>
<td>3</td>
<td>766</td>
<td>3066</td>
<td>2243</td>
<td>3356</td>
<td>1353</td>
<td>1570</td>
<td>3673</td>
</tr>
<tr>
<td>BPEL.ConstructResponseXML</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>BPEL.SendReply</td>
<td>23</td>
<td>143</td>
<td>426</td>
<td>263</td>
<td>110</td>
<td>90</td>
<td>63</td>
<td>100</td>
<td>46</td>
</tr>
<tr>
<td>TotalResponseTime</td>
<td>2750</td>
<td>5183</td>
<td>6393</td>
<td>10316</td>
<td>12390</td>
<td>18673</td>
<td>21630</td>
<td>24233</td>
<td>28240</td>
</tr>
</tbody>
</table>
REFERENCES


Robert, A. Sabetta, Eds.), ISTE Ltd and John Wiley & Sons Inc.


MYARCH Automation++ (2007), Comparison of SOA Suites, myarch.com


