

UNIVERSIDAD AUTÓNOMA
METROPOLITANA

MASTER IN SCIENCE AND TECNOLOGIES OF INFORMATION

**Quality of Service Management for
Enterprise Service Bus (ESB)**

Author:

Mariano VARGAS SANTIAGO

Supervisor:

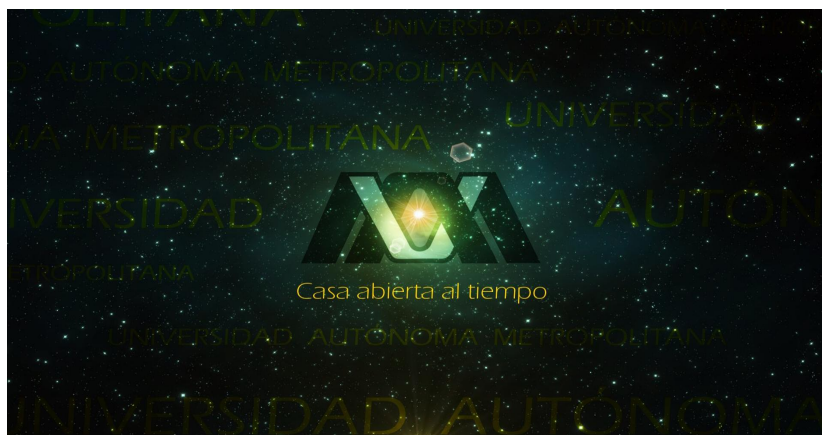
Dr. Luis Martín ROJAS
CÁRDENAS

*A thesis submitted in fulfilment of the requirements
for the degree of Master in Science*

in the

Maestría en Ciencias y Tecnologías de la Información
PCyTI

December 2013



“The important thing is not to stop questioning. Curiosity has its own reason for existing. One cannot help but be in awe when he contemplates the mysteries of eternity, of life, of the marvelous structure of reality. It is enough if one tries merely to comprehend a little of this mystery every day. Never lose a holy curiosity.”

“Everything should be made as simple as possible, but no simpler.”

Albert Einstein

“The truth of things is the chief nutriment of superior intellects.”

Leonardo da Vinci

“All truths are easy to understand once they are discovered; the point is to discover them.”

Galileo Galilei

“Experience: that most brutal of teachers. But you learn, my God do you learn.”

C. S. Lewis

UNIVERSIDAD AUTÓNOMA METROPOLITANA

Abstract

UAM

PCyTI

Master in Science

Quality of Service Management for Enterprise Service Bus (ESB)

by Mariano VARGAS SANTIAGO

Nowadays with the accelerated evolution of Internet and the advent of the cloud computing, networked applications and distributed systems in general are more and more designed as a composition of distributed services following the Service Oriented Architecture (SOA) paradigm. The standardization efforts around SOA give open and standard interfaces facilitating the integration and the interoperability of services. And the Enterprise Service Bus (ESB) is mostly used as the infrastructure of services integration as it follows and applies defined standards. Given its role and importance, an ESB brings up many challenges like knowing its limits and constraints, but also including strategies to guarantee and improve its performance and reliability. The main goal of this paper is to present an experimental platform that allows evaluating and modeling ESB performance. The platform can be used to make resource planning before deploying a business around an ESB but also to propose, develop, test and validate solutions aimed at managing ESB scalability and quality of service...

Acknowledgements

Thanks to God Almighty for the completion of this master's thesis.

To my father, my mother, my brothers, and all my family, who have always believed in me and who have been there when I needed them most.

To Assaely my partner in life who has never stopped believing in me, always been very supportive, confidence and encouraged me to continue on and on.

To Dr. Luis Rojas Cárdenas , for his teaching, advice and especially for the direction of this work, as the primary part of this.

To Dr. Ernesto Expósito, co-counsel for this work and for his comments to improve it.

To Codé Diop, for taking the time to review my work and guide me with his comments for improvement.

To my synod: Dr. Alfonso Castro for taking his time to review my work and orientation feedback for improvement of this thesis.

To my degree teachers for my bearings, with good tips and lessons to improve as a person and as a student.

To my friends, for supporting me when I need it and distract me when necessary.

To the National Council for Science and Technology (CONACyT) for the support given, which was essential for my studies.

To LAAS CNRS for the opportunity and to the FP7-ICT IMAGINE research and development project, co-funded by the European Commission under the “Virtual Factories and Enterprises” (FoF-ICT- 2011.7.3, Grant Agreement No: 285132).

Contents

Abstract	ii
Acknowledgements	iii
List of Figures	vi
List of Tables	viii
Abbreviations	ix
0.1 Abstract	1
0.2 Introducción	1
0.3 Estado del Arte	3
0.4 Metodología	6
0.5 Resultados	7
0.6 Conclusiones	10
1 Introduction	11
1.1 Welcome and Thank You	11
1.1.1 Objectives	11
1.2 Motivation	12
1.3 Basic Concepts	13
1.3.1 Service Oriented-Architecture	13
1.3.2 System Integration	14
1.4 Enterprise Service Bus	16
1.4.1 Functionalities of an ESB	17
1.4.2 Use case Example of an ESB	17
1.4.3 Dissertation Structure	18
2 State of the Art	19
2.1 Related Work	19
3 Emulator	27
3.1 Problem Statement	27
3.2 Motivation	27
3.2.1 Different ways to Evaluate Distributed Systems	28

3.3	ESB's General Environment	30
3.3.1	General Use Case	31
3.4	Emulators Architecture	32
3.4.1	Emulator's Components General View	33
3.4.2	Emulator's Component Diagram	34
3.5	Current Environment	35
4	Results and Analysis	37
4.1	Test with SoapUI	37
4.2	Avoiding Bottlenecks	39
4.2.1	Performance issues from the provider side and impact	40
4.2.2	Performance issues from the consumer side and impact	47
4.3	Identified Problem while Evaluating the ESB's Performance	47
4.3.1	What is the cost of monitoring the ESB?	48
4.4	OUT OF HEAP MEMORY	50
5	Emulators Outputs	54
5.1	Obtained Metrics	54
5.1.1	Obtained Graphs	55
6	Structure Learning	59
6.1	Motivation	59
6.2	Data sampling	60
6.3	PC Algorithm	62
6.4	Graphical Model	63
6.5	Analysis	64
6.6	Conclusions	68
7	Conclusions and Perspectives	69
7.1	Conclusions	69
	Bibliography	71

List of Figures

1	<i>Emulators architecture</i>	7
2	<i>Bayesian Network</i>	9
1.1	System Integration Point to Point	14
1.2	System Integration EAI	15
1.3	System Integration ESB	16
2.1	Typical Components Around an ESB [UT06]	20
2.2	Direct Proxy [AP11]	21
2.3	Content Based Routing [AP11]	22
2.4	Transformation Proxy [AP11]	22
2.5	Direct Service Orchestration and BPEL Orchestration [SJA]	23
2.6	Direct Service Orchestration and BPEL a) First experiment. b) Second experiment [GJMCGS10]	24
3.1	ESB General Environment	30
3.2	Emulator General Use Case	31
3.3	Emulator Architecture	33
3.4	Emulator Black Box	34
3.5	Emulator Component Diagram	35
3.6	Current Environment	36
4.1	Results Using SoapUI	38
4.2	Scenario Topology and Metrics	39
4.3	Phase One Application Server Resources	40
4.4	Phase One Response Time	41
4.5	Phase Two Application Server Resources	41
4.6	Phase Two Response Time	42
4.7	Phase Three Application Server Resources	42
4.8	Phase Three Response Time	43
4.9	Phase Four Application Server Resources	44
4.10	Phase Four Response Time	44
4.11	Phase Five Application Server Resources	45
4.12	High Response Time and losses detection issue	46
4.13	Resume of Three Phases	46
4.14	Out of Heap Memory at the Consumer Side	47
4.15	ESB Times	48
4.16	ESB Mediation Comparison	50
4.17	Consumer Concurrency	51

4.18	ESB Concurrency	52
4.19	Provider Concurrency	52
4.20	Consumer vs ESB vs Provider Concurrency	53
5.1	Computed Response Time	56
5.2	ESB Concurrency view by the Consumer	56
5.3	ESB Heap Memory view by the Consumer	57
5.4	ESB CPU view by the Consumer	58
6.1	Bayesian Network	64

List of Tables

2	ESB Evaluation Survey (Resumen)	5
3	Tested Scenarios (Resumen)	8
4	Variables and States (Resumen).	9
2.1	ESB Evaluation Survey	25
3.1	Characteristics of VMs	36
4.1	First Scenario, No Mediation	49
4.2	Second Scenario, With Mediation	49
5.1	Stored Information	54
6.1	Tested Scenarios	60
6.2	CPU	61
6.3	Memory Used	61
6.4	Concurrency	61
6.5	Number of Requests	62
6.6	Response Time	62
6.7	Marginal Probabilities	64
6.8	High Concurrency (C=H)	65
6.9	Recommended RT	65
6.10	Very High NRs	66
6.11	New Marginal Probabilities	66
6.12	New High Concurrency (C=H)	67
6.13	New Recommended RT	67
6.14	New Very High NRs	67

Abbreviations

ESB	E nterprise S ervice B us
XML	e X tensible M arkup L anguage
QoS	Q uality of S ervice
SOA	S ervice O riented A rchitecture
SOAP	S imple O bject A ccess P rotocol
TPS	T ransactions P er S econd
VM	V irtual M achine

Dedicated to my parents and Diana Assaely the love of my life...

Resumen

0.1 Abstract

Hoy en día, la mayoría de las organizaciones siguen un paradigma SOA (Service Oriented Architecture – arquitectura orientada al servicio) basada en un ESB (Enterprise Service Bus – Bus de Servicios Empresariales). Un ESB es una solución middleware que permite la interconexión de aplicaciones y servicios heterogéneos, típicamente entregá de mensajes de los consumidores de servicios a los proveedores de los mismos. El uso de un ESB es la tendencia que la mayoría de las organizaciones siguen actualmente, él esta encargado de la integración de sistemas, la solicitud de enrutamiento, la transformación y adaptación de datos. Sin embargo, un ESB trae consigo muchos retos, como saber sus límites y problemas, debido a que tiene como meta reducir los costos y mejorar la eficiencia operativa del negocio. En general la mayoría de los B2B (Business-to-Business - negocio-a-negocio) tienen sistemas complejos de integraciónón en torno a un ESB, de hecho la integración de cualquier sistema utilizando un ESB como columna vertebral de integración, tiene dificultades para descubrir problemas potenciales antes de su despliegue. Por lo tanto, contar con una herramienta que pueda implementar y evaluar complejos B2B a bajo costo es indispensable. En este trabajo se propone un emulador genérico para evaluar el desempeño de cualquiera que sea el ESB. Proponemos un entorno basado en la nube, ejecutando el ESB, consumidores y proveedores de servicios en diferentes máquinas virtuales. Estamos haciendo más fácil la evaluación del desempeño de un ESB, en condiciones de alta presión como en un entorno real...

0.2 Introducción

En el contexto de la rápida evolución del Internet, aplicaciones distribuidas y más ampliamente los sistemas distribuidos se basan cada vez más en los servicios que se integran para implementar procesos de negocio complejos. La diversidad y heterogeneidad de

estos servicios plantean muchas necesidades durante la integración. Para satisfacer estas, el ESB se ha propuesto para el desarrollo de estrategias, basado en estándares de integración de servicios y aplicaciones. Sin embargo, la competencia entre los servicios y el contexto dinámico de SOA (movilidad, aumento de consumidores y proveedores de servicios, etc. . .) puede causar acontecimientos imprevisibles, como la falta de disponibilidad de servicios, tiempos de respuesta más altos, menor fiabilidad, y problemas por cuestiones de seguridad. En este contexto, debe ser una prioridad garantizar la calidad de servicio (QoS), mientras se acceda a servicios distribuidos desplegados en el ESB.

SOA es una arquitectura que modulariza servicios. En una SOA con éxito, puede recombinar estos servicios en diversas formas para la ejecución de procesos empresariales nuevos o mejorados [VAMD09]. SOA permite a las empresas reutilizar los servicios que crean a través de la integración de múltiples aplicaciones en más de un proceso de negocio sin necesidad de programación adicional. En esencia, SOA establece un repositorio de estos servicios, que los desarrolladores de procesos de negocio pueden descubrir a través de protocolos servicios Web [Ort07].

Juntos SOA y ESB proporcionan una infraestructura que permite la comunicación entre diferentes aplicaciones que se ejecutan en diferentes plataformas y escritos en diversos idiomas. Un ESB es definido en [Cha09] como un sistema de integración middleware que permite a los sistemas heterogéneos distribuidos tratar y gestionar los datos. Los esfuerzos de normalización en torno SOA crean como enfoque principal de integración al ESB, ya que un ESB es el enfoque basado en estándares. Además un ESB permite una integración más rápida y más barata de los sistemas y permite tener una mejor solución de escalabilidad para despliegues empresariales distribuidos.

En general la mayoría de los B2B (negocio-a-negocio) usualmente cuenta con sistemas complejos de integración en torno a un ESB. Es difícil y costoso de implementar o aplicar las topologías complejas alrededor de un ESB. Debido a la falta de experiencia de los desarrolladores de procesos de negocio y el uso de los recursos dedicados a una aplicación o servicio en proceso que no puede fallar o sufrir interrupciones. La tendencia del B2B para integrar cualquier sistema (aplicaciones o servicios), construido sobre un ESB como una columna vertebral de integración, encuentra dificultad en descubrir problemas potenciales antes de la implantación de nuevos negocios. Ya que, la mayoría de los problemas se producen en el ESB bajo alta carga de trabajo.

A lo largo de los años, muchos estudios se han realizado con el fin de evaluar el desempeño del ESB, tratando de averiguar si tiene problemas de seguridad, fiabilidad y de escalabilidad. La evaluación del performance ESB se puede hacer midiendo el rendimiento y el tiempo medio de respuesta.

Ser capaz de evaluar los ESB realmente es una necesidad de las empresas y muchos enfoques y herramientas están propuestas en el trabajo relacionado. Por ejemplo, el objetivo de capacidad de planificación en [UT06], para determinar qué hardware comprar para cumplir con los requisitos de coste, el rendimiento y la escalabilidad. Tomando otro ejemplo, para evaluar el desempeño de la ESB en relación con la orquestación de servicios [SJA], que las empresas utilizan para crear servicios compuestos de más alto nivel y más útiles.

Como consecuencia de ello, desarrollamos una herramienta para evaluar el comportamiento de los ESB. Utilizando como KPIs (Key Performance Indicators – indicadores clave de rendimiento) el tiempo de respuesta, la fiabilidad, y el número de solicitudes simultáneas. En concreto la contribución de este trabajo es presentar una herramienta para evaluación comparativa del rendimiento ESB. Además de contar con la posibilidad de implementar diferentes escenarios en un entorno controlado.

0.3 Estado del Arte

Muchos estudios se han realizado con el fin de evaluar el desempeño de un ESB, en [UT06] Ken Ueno et al. propuso una técnica de prueba de capacidad, realizada muy temprano en la vida de un proyecto (fase de capacidad de planificación). Desplegó un consumidor y proveedor de servicios Web ligero para probar el rendimiento del ESB. Los resultados de sus pruebas de capacidad pueden revelar la capacidad máxima actual del servidor ESB en una plataforma específica. Se hicieron pruebas de la ESB sin mediación y con mediación. Se midió el rendimiento como transacciones por segundo (TPS), y el tiempo medio de respuesta.

En [AP11], podemos encontrar otros enfoques estudiados acerca de la evaluación de tres diferentes ESBs: Mule, ServiceMix y WSO2. Los autores tomaron como parámetros de evaluación: el tiempo de respuesta promedio y las TPS. Para generar peticiones concurrentes, se utilizó Grinder. En este artículo podemos encontrar tres escenarios de prueba diferentes, la primera con el ESB simplemente obteniendo la solicitud del cliente y pasándola al proveedor de servicios, esto se conoce como virtualización o proxy. Los tres componentes implicados (clientes, ESB y Proveedores de servicios Web) se desplegaron en diferentes máquinas. Otra prueba fue hecha cuando el ESB realiza algún tipo de tratamiento de datos, identificando una parte de entrada del mensaje que viene del lado del cliente para enrutar correctamente al proveedor de servicios. Esto se llama enrutamiento de contenido base, y es una característica desplegada en cada uno de los ESB estudiados. Los autores midieron el rendimiento como las transacciones por segundo, y el tiempo medio de respuesta.

En algunas otras obras de los autores centraron su estudio en una de las muchas características que un ESB debe lograr, por ejemplo, el rendimiento de la mediación [SJA]. Ellos evaluaron la función de mediación de los siguientes ESB: ServiceMix, Mule y JBoss. La aplicación implementada en todos los escenarios fue la aplicación corredor de préstamos; esta simula más o menos el comportamiento de una aplicación real. Para conocer la capacidad máxima de un ESB, debe sobrecargarse; con este fin los autores utilizaron Apache JMeter para enviar muchas solicitudes concurrentes por parte del cliente.

El objetivo en [GJMCGS10] fue evaluar las características de tres ESB de código abierto: Fuse, Mule y Petals ESB. En particular, los autores realizaron un análisis del rendimiento del tiempo de respuesta con respecto a la invocación de servicios externos y BPEL (Business Process Execution Language) de procesos basados en un sistema existente.

Para concluir esta sección se presenta la caracterización principal de QoS de los ESBs estudiados en los trabajos relacionados. Algunos de los parámetros QoS utilizados para caracterizar el ESB son: el tiempo de respuesta, la escalabilidad, la saturación del sistema, transacciones por segundo (TPS). Los cuales pueden ser obtenidos utilizando nuestro enfoque propuesto.

La Tabla 2 muestra un resumen del estudio hecho. Esta Tabla ilustra: la evaluación del o de los ESB evaluado(s), el entorno de evaluación del ESB, las herramientas prueba de estrés utilizadas para evaluar al ESB a su máxima capacidad, la aplicación utilizada, una descripción de si es emulación o metrología y, finalmente, el objetivo que muestra por qué se evaluó el ESB.

TABLE 2: ESB Evaluation Survey.

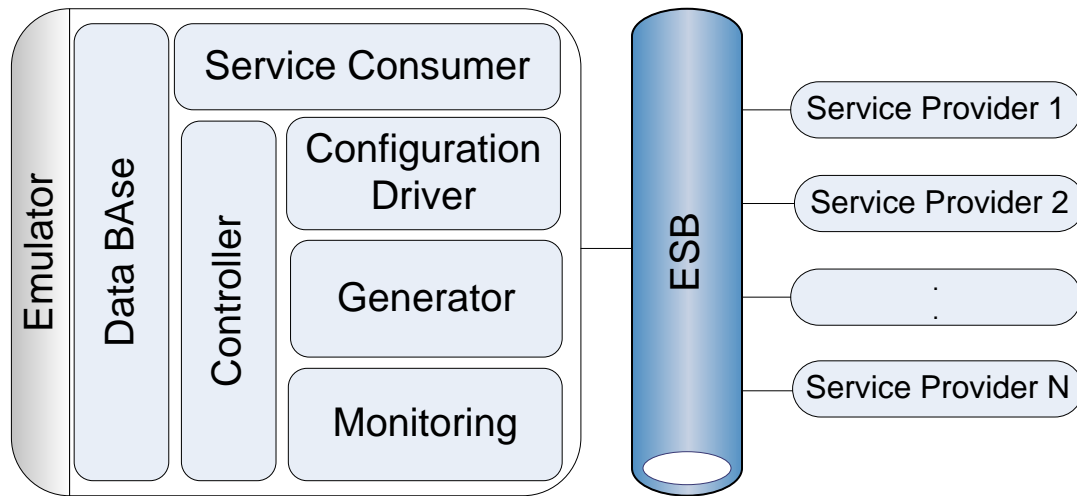
Parameter	[UT06]	[AP11]	[SJA]	[Bre09]	[Bru]	[GJMCGS10]
Evaluation	Response Time Throughput	Response Time Throughput (Proxy, Routing, Mediation)	Response Time Throughput (Service Orchestration)	Response Time Throughput	Security Issues Integration Issues	Response Time Throughput
ESB	-----	Mule ESB ServiceMix WSO2 ESB	Mule ESB ServiceMix JBoss ESB	Mule ESB	ServiceMix	Mule ESB Fuse ESB Petals ESB
Environment	1. No ESB 2. ESB (without Mediation) 3. ESB (with Mediation)	1. Direct Proxy 2. Routing Proxy 3. Transformation Proxy	1. Direct Orchestration 2. BPEL Orchestration	1. Inside one Server 2. Everything != Server	Observatory (Real Application)	1. No JMS 2. With JMS
Parameters	At Provider Multi @IP HTTP Keep Alive # of Threads Payload	At Consumer # of Clients Payload	At Consumer # of Clients Payload	At Consumer Payload until saturation	No Parameter	At Consumer # of Invocations
Application E/M	HTTP Load Generator E	Grinder E	Apache JMeter Loan Broker E	Apache JMeter Loan Broker E	Real Application M	Real Application M
Goal	Capacity Planning	ESB Core Features	Orchestration Performance	Performance Issues Scalability Issues	ESB Issues	System Integration Evaluation

0.4 Metodología

Proponemos un emulador agnóstico, que no es producto ESB dependiente. Sin embargo, algunas medidas de adaptación tienen que ser efectuadas dentro del ESB correspondiente para calcular las marcas de tiempo correspondientes. Una ventaja en la que se centro nuestro enfoque es que esta basado en la nube, lo que nos permite controlar el uso actual de los recursos de cada máquina virtual que alberga a: consumidores, ESB y proveedores. Además de la capacidad de obtener métricas KPI eficazmente, es decir, de manera directa.

Hemos presentado herramientas estrés como son: SoapUI, Grinder y Apache JMeter, en el estado del arte. Algunas de estas herramientas son de código abierto, pero otras como SoapUI tienen versiones comerciales. Estas herramientas permiten tener diferentes métricas de calidad de servicio (QoS), tales como: el tiempo medio de respuesta y TPS, pero su fallo se mantiene en no ser capaz del monitoreo de la CPU y memoria heap del sistema ESB. Como consecuencia se propone una arquitectura para ser capaz de controlar los defectos antes mencionados.

La Figura 1 muestra los componentes principales que intervienen en la arquitectura del emulador. El *Consumidor de Servicio(s)* que es la instancia a cargo de la comunicación con el ESB. La *Base de datos* donde se almacenará toda la información de cada solicitud–respuesta. El *Controlador* es responsable de la visualización de gráficos y la información recuperada de la ejecución de un escenario, los resultados. Este componente tiene la responsabilidad de invocar a cada LD (Load Driver –*controladorde carga*). Un tercer componente, el *Driver de Configuración*, almacena el número de invocaciones que cada LD puede lograr, y contiene el tiempo de sueño emulado en el lado proveedor y el tamaño de los datos de respuesta en bytes que el/los proveedor(es) deben generar. El *Generador*, su nombre lo dice, genera cada invocación hecha de un LD a un servicio Web (proveedor). Por último, el *Monitoreo* es una instancia independiente encargada del monitoreo de la memoria Heap de la ESB y el uso de la CPU.

FIGURE 1: *Emulators architecture*

0.5 Resultados

Este trabajo tiene como meta principal agilizar el análisis de rendimiento de un ESB. Para lograr dicha agilización se afianzo que el emulador debería regresar de manera directa métricas KPI. Estas pueden ser propuestas por expertos en el tema; como son los desarrolladores de software. Nuestro emulador obtiene de manera directa, de un escenario previamente ejecutado, gráficas como son: el tiempo de respuesta, número de peticiones concurrentes, la memoria Heap y el uso de la CPU. Con las cuales se puede llevar a cabo un análisis del comportamiento y rendimiento del ESB en cuestión. Teniendo en cuenta que para determinar el máximo rendimiento del sistema ESB se deben evitar cuellos de botella. A continuación se muestra un análisis de diagnóstico probabilístico basado en una red Bayesiana.

Para lograr este objetivo algunas herramientas conocidas establecen relaciones de dependencia entre las variables que se encuentran en una muestra de datos y construyen un modelo probabilístico, esto se llama aprendizaje estructural. El Algoritmo PC (PC Algorithm) es una de las técnicas de aprendizaje para encontrar las relaciones de dependencia entre variables en un modelo, y se construye un modelo gráfico en el que se puede inferir el conocimiento. Esta herramienta necesita un conjunto de datos discretos, que se

genera fácilmente después de ejecutar varios escenarios con nuestro enfoque propuesto. La discretización de indicadores clave de rendimiento (KPIs) y los datos monitorizados es una primera clasificación necesaria, después se genera una BN (Bayesian Network - red bayesiana), que es el resultado de las pruebas de dependencia, antepuesta posteriormente al cálculo del DAG (Direct Acyclic Graph – gráfica acrílica directa). DAG representa las relaciones causales entre las variables. Finalmente, se calculan a partir de los datos discretos, las tablas de probabilidad condicional (CPT), para conseguir una BN.

Hemos implementado una serie de diferentes escenarios como se muestra en la Tabla 3.

TABLE 3: Tested Scenarios (Resumen)

Scenario	# of Load Drivers	# of Request	Response Size	Processing Time (ms)
1	1	4000	1000 B	100
2	2	8000	1000 B	100
3	3	12000	1000 B	100
4	4	16000	1000 B	100
5	6	24000	1000 B	100
6	8	32000	1000 B	100

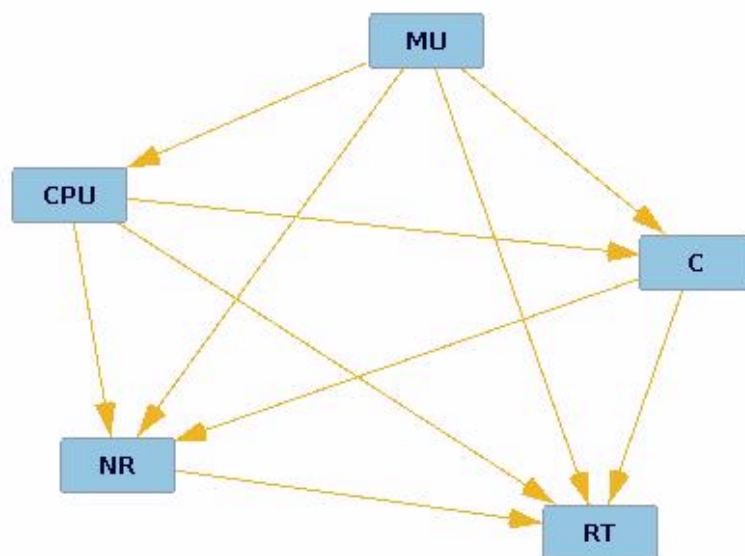
El conjunto de datos consta de cinco variables: el porcentaje utilizado de CPU y de memoria, el tiempo de respuesta, el nivel de concurrencia y el número de peticiones. La Tabla 4 muestra los valores discretos y sus estados.

Después de ejecutar el algoritmo PC, usando Tetrad v4.3, se obtuvo la siguiente red bayesiana en su representación gráfica, Figura 6.1. Las relaciones de dependencia entre las variables se establece mediante pruebas de hipótesis estadísticas, de pruebas específicas, independencia condicional.

La figura 2 muestra la fuerte dependencia de cada uno de los indicadores clave de rendimiento monitorizados y medidos. Lo que esto significa es que todas las variables dependen una de la otra.

TABLE 4: Variables and States (Resumen).

Variable	States
CPU	Low = [0,25), Med = [25,50), High = [50, 75), Very High = [75,100]
MU	
C	Low = [0, 339), Med = [339, 967), High = [967, 4219), Very High = [4219, 9919]
RT (Measured in seconds according to the UTI G.1010 Recommendation)	Preferred = [0; 2), Acceptable = [2; 4), Unacceptable = [4; +∞)
NR	Low = [0; 10000), Med = [10000; 20000), High = [20000; 30000), Very High = [30000; +∞)

FIGURE 2: *Bayesian Network*

0.6 Conclusiones

En este trabajo, hemos propuesto una herramienta que facilita la evaluación de un ESB con el objetivo principal centrado en circunstancias de alta carga de trabajo como en una producción real. Utilizando el emulador podríamos implementar complejas y diferentes topologías de redes de negocios. Con nuestro enfoque fácilmente se pueden encontrar cuestiones de fiabilidad, transacciones por segundo y el tiempo de respuesta entre otras características de calidad de servicio de un ESB.

Hemos encontrado un problema de fiabilidad cuando enviamos 40,000 solicitudes concurrentes (10 LD) para este ESB, ya que este asigna una cantidad dinámica de almacenamiento de memoria por cada solicitud que recibe, y por lo tanto se desborda la memoria Heap.

Chapter 1

Introduction

1.1 Welcome and Thank You

In the context of the rapid evolution of the Internet, distributed applications and more broadly distributed systems are increasingly based on services that are integrated to implement complex business processes. Diversity and heterogeneity of these services raise many needs during integration. To meet these needs, the ESB (Enterprise Service Bus) has been proposed for the development of strategies, standards-based integration services and applications. However, the competition between the services and the dynamic context of service-oriented architectures (mobility, increase of service consumer and providers, etc. . .) may cause unforeseeable events, such as service unavailability, higher response times, reduced reliability, safety, etc. . . In this context, ensuring quality of service while accessing distributed services deployed on the ESB must be a priority. The main contribution of this work is to study the quality of service problems that may occur with the use of an ESB.

1.1.1 Objectives

- State of the art of ESB and service quality problems due to the use of an ESB.

Develop a tool to assess the behavior of ESBs. This tool should allow to:

Deploy providers and consumers of services configurable on the ESB.

Evaluate the ESB using as KPI (Key Performance Indicator) response time, reliability, etc. . .

Configure the processing time of a request by the suppliers, the frequency of consumer requests/second, the data exchanged size.

Extract and analyze the simulation results.

- Design, develop and evaluate mechanisms to control the quality of service.

The expected results are:

- State of the art and a final Emulator.

1.2 Motivation

Nowadays, most of the organizations follow a Service Oriented Architecture model (SOA). SOA provides business functions to be reusable in practically throughout the whole organization. SOA grants a collection of services to communicate with each other, because an application needs to share and communicate data with other applications. In the past, the most common architecture used was the Enterprise Application Integration (EAI), EAI was used to integrate services or applications in an easier way, since all the communications passed only through a message broker. The EAI handles routing and data transformation, but the EAI has scalability problems, because if the message broker fails the rest of the network applications will not be able to communicate with each other, this is known as a single point of failure in the network. Plus EAI is not standards-based. Nor SOA or EAI handle all the features of a new solution proposed called Enterprise Service Bus (ESB). With SOA paradigm an ESB performs as a mediator facilitating the provision and consumption of services. An ESB is a middleware solution, it permits heterogeneous environments using a service oriented architecture to interoperate. The invention of the ESB was not an accident. The ESB is a result of vendors working with forward-thinking customers who were trying to build a standards-based integration network using a foundation of SOA, messaging, and XML [Cha09]. Therefore, the ESB is the new component burden with the role of connecting heterogeneous applications and services. In other words an ESB is a bus in charge of delivering messages from service requesters to service providers.

Many approaches have been done in order to assess the ESB performance. The evaluation of the ESB performance can be done by assessing the throughput and the mean response time, it is important to remark that the ESB must be under high load to get its maximum capacity performance. There is much ongoing research on performance evaluation of ESBs, this kind of studies are important, since evaluating an ESB is no easy task but needed in some important aspects for an enterprise. For example, to determine what hardware to buy to meet the cost, performance and scalability requirements which is the objective of capacity planning in [UT06]. Another important fact is that if the communication between each service or application is done in a point to point way, after

time the interaction between them becomes difficult and this rises the system costs and maintenance. The above mention problems of system integration are the main reasons why we need a method to evaluate the capacity of an ESB.

The aim of this study is to evaluate the performance of an ESB, deploying an emulator. This emulator will allow us to retrieve the throughput and mean response time using Linux shell scripts. This emulator will assess firstly, under high load the ESB will act as a pass through network device, just forwarding the incoming messages from the client to the provider. Secondly, the same evaluation will be made but this time with some sort of the ESB processing, for example routing or transformation inside the ESB, both of these examples are features of an ESB called mediation.

1.3 Basic Concepts

1.3.1 Service Oriented-Architecture

As mentioned before a SOA (Service Oriented-Architecture) is a paradigm that most of the organizations, nowadays, follow. An SOA allows to reuse the code of an application facilitating its integration. In this context, SOA follows standards like XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol) to integrate the applications in a standards-based approach instead of a vendor-based approach. So, SOA separates a monolithic approach of integrating applications or services. The goal in SOA is to make each subsystem of a company presents their capabilities through adequate services. SOA is an architecture that can build business applications from a set of loosely coupled black box components. SOA links together business process having an orchestrated well-defined service level. The reuse of existing business applications is done adding a simple adapter to the black box components regardless of how they were built. SOA takes the best software assets used and packages them so that it lets you use them and reuse them, so you don't have to discard software.

Any SOA system is built using different units, services for a specific functionality, to have complex systems. These systems usually involve various physical resources, i.e., network resources, processing components, and of course the logical organization.

SOA enables flexible connectivity and communication among applications by representing each as a service with an interface that lets them communicate readily with one another. SOA lets companies reuse the services they create—via the integration of multiple applications— in more than one business process without additional programming.

In essence, SOA sets up a repository of these services, which business-process developers can discover via Web services protocols [Ort07].

1.3.2 System Integration

Enterprises' software components or modules (usually deployed as services) running on two or more enterprise's networks, are usually known as distributed enterprise applications. Most of the time, the enterprise network is heterogeneous and is composed of diverse computers, devices, and operating systems. Since the network is heterogeneous, systems consist in different protocols for data exchange, technologies, and devices distributed across a network. In recent years the industry environment has become increasingly distributed and heterogeneous across multiple organizational and geographical boundaries, there's a strong demand to integrate various distributed applications in order to enhance or increase enterprises' competitiveness.

There are many ways for system integration, some of them are: point to point, EAI and ESB.

In a point to point integration, consider Figure 1.1, all the services and applications need to be programmed in order to have an interface to interact with other applications and services. This kind of integration is feasible when in an organization there is no need or a small need of applications to communicate and share data between them.

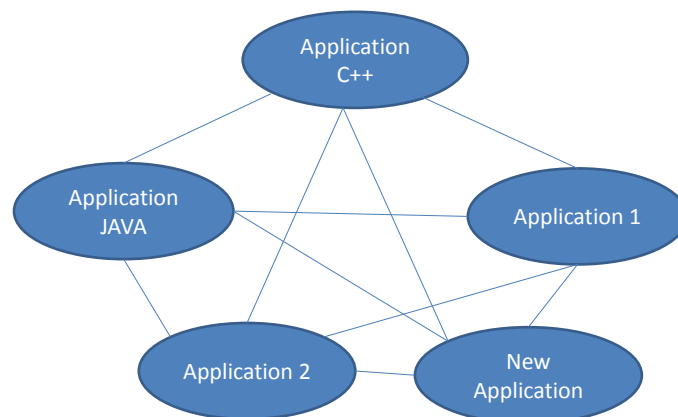


FIGURE 1.1: System Integration Point to Point.

The Figure 1.1 show a mesh of applications, communicating in a point to point SOA approach. This approach brings with it high maintenance costs and it is difficult to reuse applications. The figure also shows the difficulty of integrating a new application, for this new application we need to program all the interfaces in order to communicate with other applications. As example if we need to add another application called new application2, we need to program all its interfaces to communicate with all other applications. We could conclude, that this approach has scalability problems, as a consequence another integration approach was created called EAI (Enterprise Application Integration).

In an EAI approach, consider Figure 1.2, its main component is a message broker that handles all the communication and data sharing of the applications and services. In this approach the integration of a new application or service is easier than the point to point integration. Due to the fact that a new application only needs to report itself to the message broker, after the message broker handles all the communication between the new application and whichever other application it wants to communicate with. The problem with EAI is that it introduces the well-known point of failure to the network, for example lets think that the message broker is no longer available in the network, how does application in C++ communicate with the application in JAVA?

Another, flaw of EAI is that is not standards-based, this introduces another problem, EAI's are not able to communicate between them. To solve this kind of problem, and the problems mentioned for point to point integration another solution called ESB (Enterprise Service Bus) was proposed [AP11].

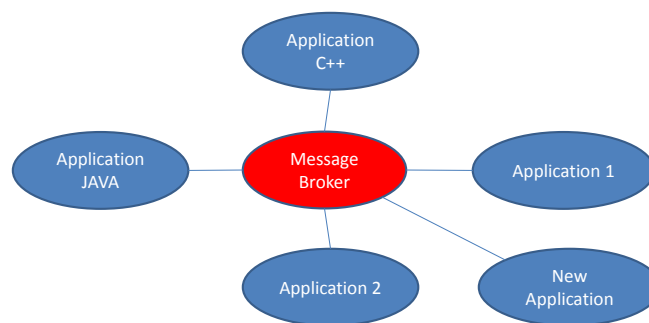


FIGURE 1.2: System Integration EAI.

To have a better understanding of an ESB, we present the Figure 1.3. An ESB is a middleware solution for system integration, as mentioned before an ESB solves the flaws that other approaches like point to point or EAI have.

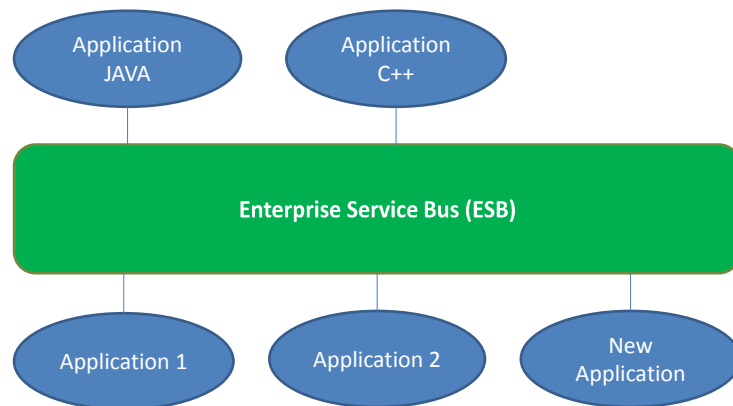


FIGURE 1.3: System Integration ESB.

The ESB provides a kind of mechanism that makes it easier to add new applications as shown in the Figure 1.3. As noticed from it, all applications communicate with each other through the ESB. Therefore, the complexity of integration logic is dealt by the ESB; and it is done only once.

1.4 Enterprise Service Bus

An ESB is a middleware layer that allows the integration of heterogeneous applications, using a standards-based approach. An ESB is a software architecture usually based on Web Services (WS) standards. It provides foundational services for more complex Service-Oriented Architectures (SOA) via an XML-based messaging engine (the bus), and thus provides an abstraction layer on top of an enterprise messaging system that allows integration architects to exploit the messaging without writing code [UT06].

Together SOA and ESBs provide an infrastructure enabling communications between different applications, running on different platforms and written in different languages. The efforts for standardization around SOA make an ESB the main integration approach,

additionally an ESB allows faster and cheaper integration of systems and it allows to have a better scalability solution for enterprise distributed deployments. An ESB provides flexibility in changing systems as the requirements change.

The ESB as mentioned before, is an improvement of point to point and EAI architectures, an ESB now plays the role of connecting heterogeneous applications and services in an SOA. An ESB handles transformation, routing, and adaptation of data. The ESB has the responsibility to ensure that the formats of the messages match between the service consumer and the service provider.

1.4.1 Functionalities of an ESB

The main functionalities that an ESB must integrate are: Virtualization, Intelligent Routing, and Mediation, these functionalities are known as the core features of an ESB [AP11].

Virtualization: Virtualization, or proxying, is a role that an ESB can play. In this role, the ESB acts as a proxy for the service provider and handles the requests of the service consumer. The ESB can handle authentication, authorization, and auditing, so the service provider can focus solely on the business logic.

Routing: The ESB has the capability to route the incoming requests on a single endpoint to the appropriate service. The ESB can look at a wide array of things like the message content or the message header to determine where the request should be routed.

Mediation: Mediation, or message transformation, is another core feature of an ESB. The ESB has the capability to take an incoming request and transform the message payload before sending it to the end Web Service.

1.4.2 Use case Example of an ESB

The typical topology of a distributed system having the ESB as integration backbone, consists of five tiers.

1. The first tier is usually represents the way on how consumers of a service interact or request a service; via a Web Browser.
2. The second tier is in charge of data management for example: Servlets, Web Services.

3. In the third tier we usually find the ESB.
4. The fourth is where the providers of a service are usually deployed.
5. The last tier consist of data servers where the providers take the data they were requested for.

1.4.3 Dissertation Structure

The Dissertation Structure is as follows:

- Chapter 1 has presented the motivation of this work as well as some basic concepts around it.
- Chapter 2 presents a state of the art of previous works made till today, we are presenting a survey on how to evaluate the performance of different ESBs. Here we present, the most common architectures around a testbed or a real system involving the use of an ESB. We also present the assessment of the core functionalities of an ESB as mediation, routing and transformation.
- Chapter 3 presents our proposal, in concrete, the emulator; which is born out of the need presented in the related work. We present a general Use Case, and the components diagram around our emulator. In this chapter we could find the general approach functions of our emulator.
- Chapter 4 introduces some results and analysis, a comparison between two different scenarios was made. This Chapter 4 gives the consequences of totally monitoring the ESB.
- Chapter 5 describes the emulators outputs, this chapter illustrates the information collected using the emulator and also illustrates the graphs important for the analysis.
- Chapter 6 describes one of possible cases where the emulator can be used; in this chapter our emulator has the function of data mining.
- Chapter 7 presents conclusions for this work and also the future work in the domain.

Chapter 2

State of the Art

2.1 Related Work

The use of SOA in organizations continues growing, this means more industries are adopting SOA which is one of many branches of new engineering challenges for research. How to assess the performance of an ESB is one of the new challenges. To predict the capacity of an ESB we must measure its performance which differs from predictions for traditional application servers.

Many studies have been done in order to evaluate performance of an ESB, in [UT06] the authors propose a capacity testing technique, it is conducted very early in a project's lifetime (capacity planning phase). What the authors deployed to test the performance of an ESB was a lightweight Web Service provider and consumer, the results of their capacity testing can reveal the actual maximum capacity of the ESB server on the specific platform.

The Figure 2.1 show the architecture deployed in [UT06]. This Figure 2.1 could be seen as a three tier environment (consumers of a service, the ESB and the providers).

1. The first tier is the Web Service Clients (consumers), since the goal of this article was to evaluate the ESB under high workload to its maximum capacity. They used a HTTP Workload Simulator that generates concurrent requests in order to stress test the ESB.
2. The second tier is the ESB. Which they test with and without mediation.
3. The third tier, the Web Service Provider; the authors used a Lightweight service provider that emulates a delay to each request, trying to emulate a real world application.

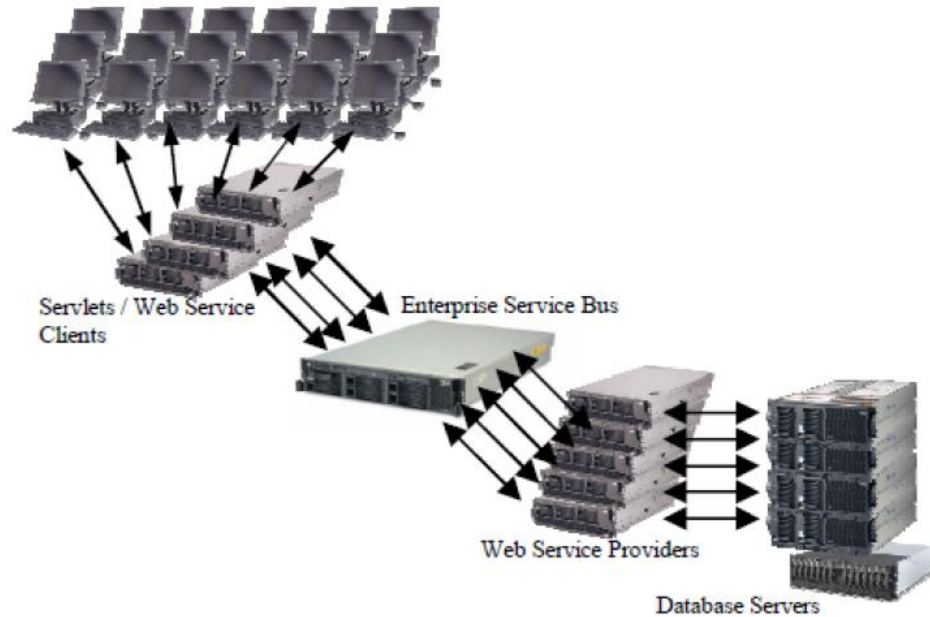


FIGURE 2.1: Typical Components Around an ESB [UT06].

The authors do not mention what ESB was taken for their tests, they simply refer to this as the ESB system. They measured the throughput as the transactions per second (TPS), and the mean response time.

Although, the results of capacity planning reveal the maximum capacity of an ESB; the resource reallocation is needed to evaluate the ESB with different system characteristics.

In [AP11], we can find other studied performance assessing approaches for three different ESBs: Mule, ServiceMix and WSO2 which have similar features and are open source. In this article we could find three different test scenarios (Direct Proxy, Content Based Routing and Transformation Proxy).

1. In the first test scenario we found the ESB configured as *Virtualization or Direct Proxy*. Where the ESB simply gets the request from the client and passes it to the service provider as depicted in Figure 2.2. All three involved components (Client, ESBs and Web Services) were deployed on different machines.

2. The second scenario consist on the ESB configured as *Content Based Routing*. Where the ESB makes some sort of processing in this case it identifies a part of an incoming message from the client side to correctly route to its adequate end side web service; this feature is deployed in each of the studied ESBs. As depicted in the Figure 2.3.
3. In the third scenario, the Figure 2.4, the ESB was configured as *Transformation Proxy*. Where the ESB shows the the ability to transform the incoming messages to a correct form that the server side will interpret. Data transformation is done through the use of XSLT (eXtensible Stylesheet Language Transformations) enabling XML based message transformation

To evaluate the ESB under high workload to its maximum capacity. In order to stress test the ESB, the authors used Grinder to generate multiple requests. They also measured the throughput as the transactions per second, and the mean response time.

Although this article evaluates the core functionalities of an ESB; they have fixed configurations of their systems lacking of resource reallocation to evaluate the ESB with different system characteristics.

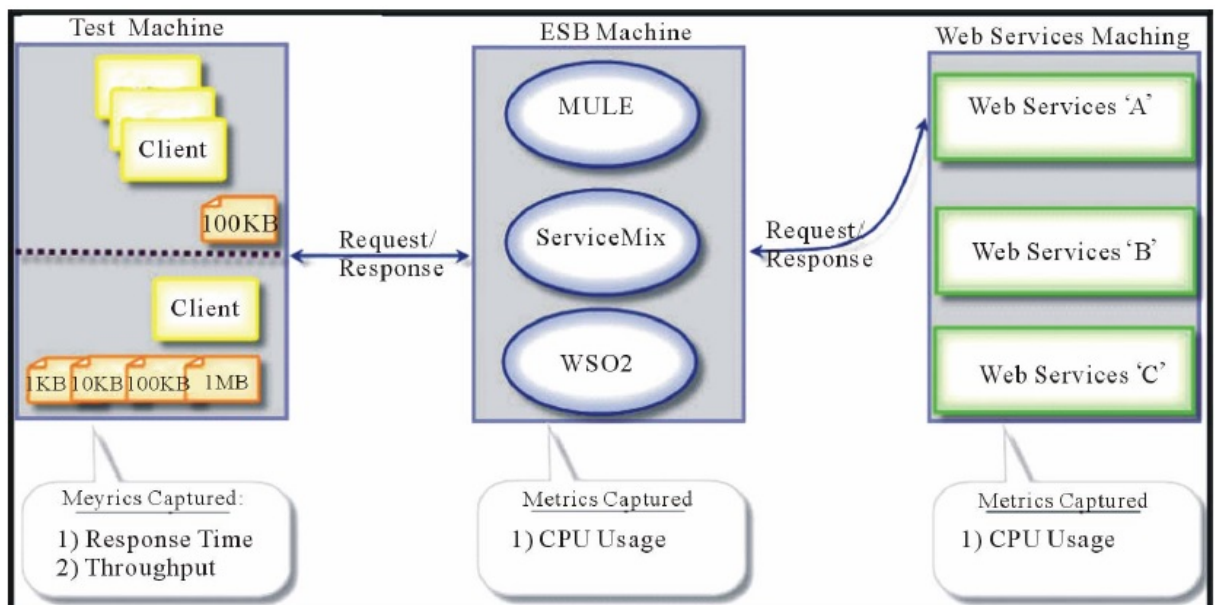


FIGURE 2.2: Direct Proxy [AP11].

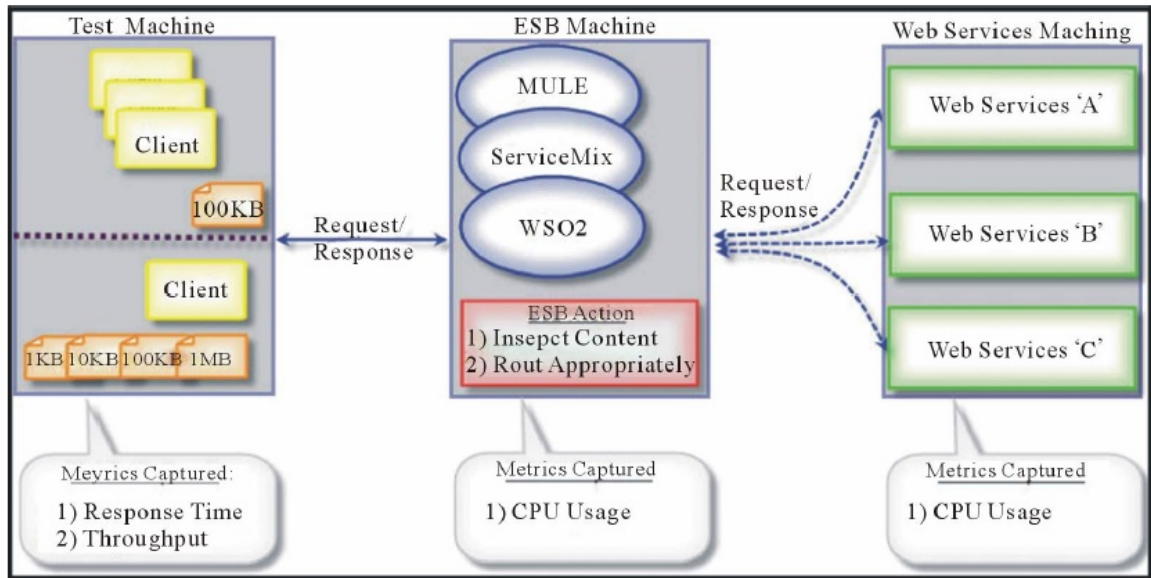


FIGURE 2.3: Content Based Routing [AP11].

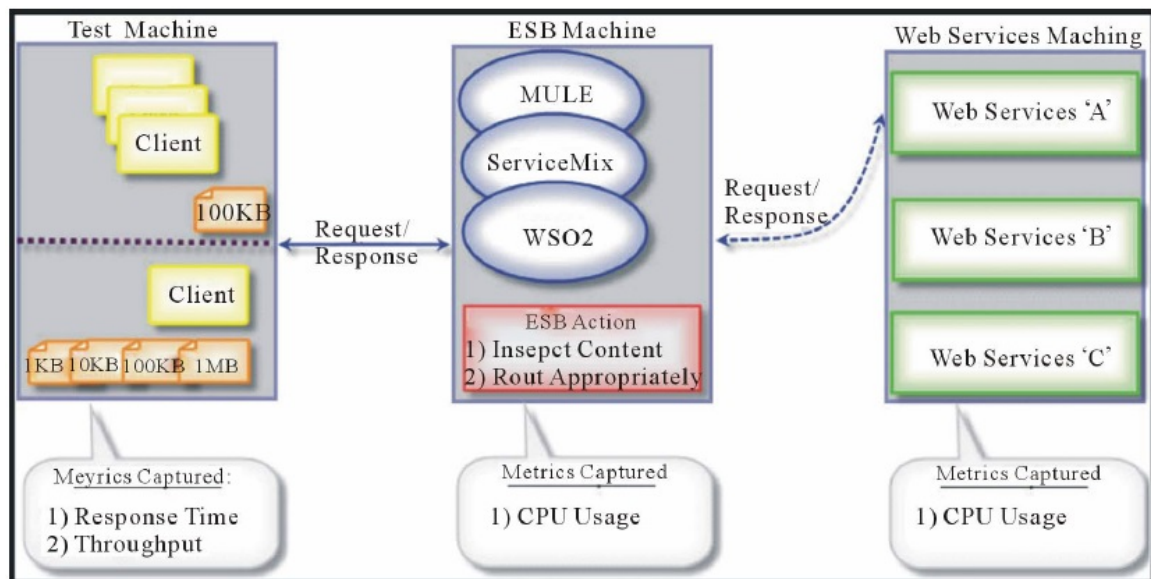


FIGURE 2.4: Transformation Proxy [AP11].

In other studies the authors focused in one of the features of an ESB, for example the mediation module performance of the ESB. In [SJA] the authors evaluated the mediation feature of the following ESBs: ServiceMix, Mule, and JBoss. Figure 2.5 only shows ServiceMix, but the authors tried to make equivalent scenarios for the other ESBs, this is one of the reasons why in Figure 2.5 Apache ODE was deployed outside ServiceMix

and not inside, giving no advantage over the other two ESBs. As we could also see from Figure 2.5, the application deployed in all scenarios was the loan broker, the loan broker application simulates more or less the behavior of a real world application.

Regarding to the mediation feature for the above mentioned ESBs; the maximum capacity of an ESB is reached under high workload circumstances; for achieving such goal the authors used Apache JMeter which generates many concurrent requests from the client side.

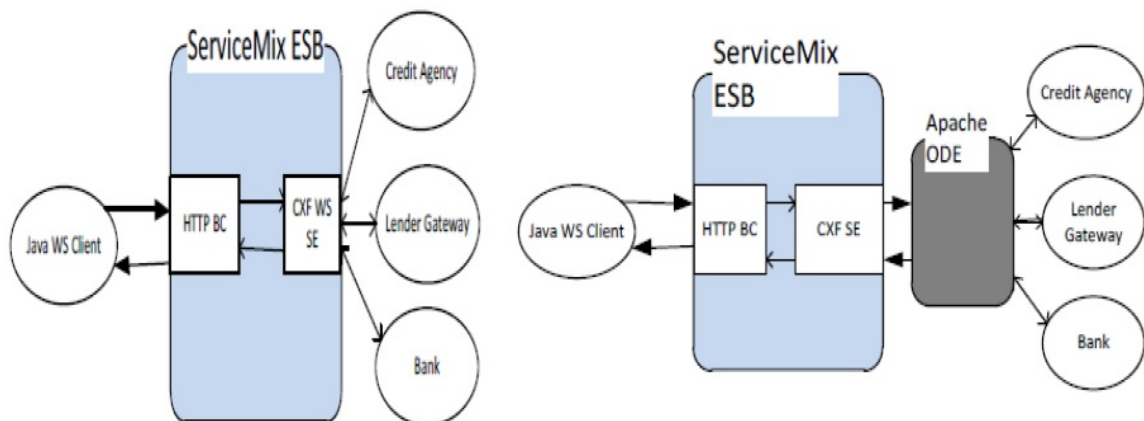


FIGURE 2.5: Direct Service Orchestration and BPEL Orchestration [SJA].

The aim of [GJMCGS10] García et al. was to evaluate the features of three ESBs open source: Fuse, Mule and Petals. In particular the authors made a performance analysis of the response time regarding to invoking external services and BPEL (Business Process Execution Language) process based on an existing system. The Figure 2.6 shows the scenarios configured for such performance analysis.

In [Bre09] the authors show the application of their method called Service Oriented Performance Modeling; a method and tool support for performance modeling of large-scale heterogeneous SOAs. Implementing their approach SOPM to the MULE ESB Loan Broker application in a laboratory context. They use a test-bed, deploying everything inside the same server and comparing it to another deployment where all components (Client, ESB and Providers) were in different machines.

In other work we could find the possible issues of using an ESB as an integration platform [Bru]. In this article the authors use the ServiceMix ESB to integrate their applications, this is a real world integration for an Ocean Observatory.

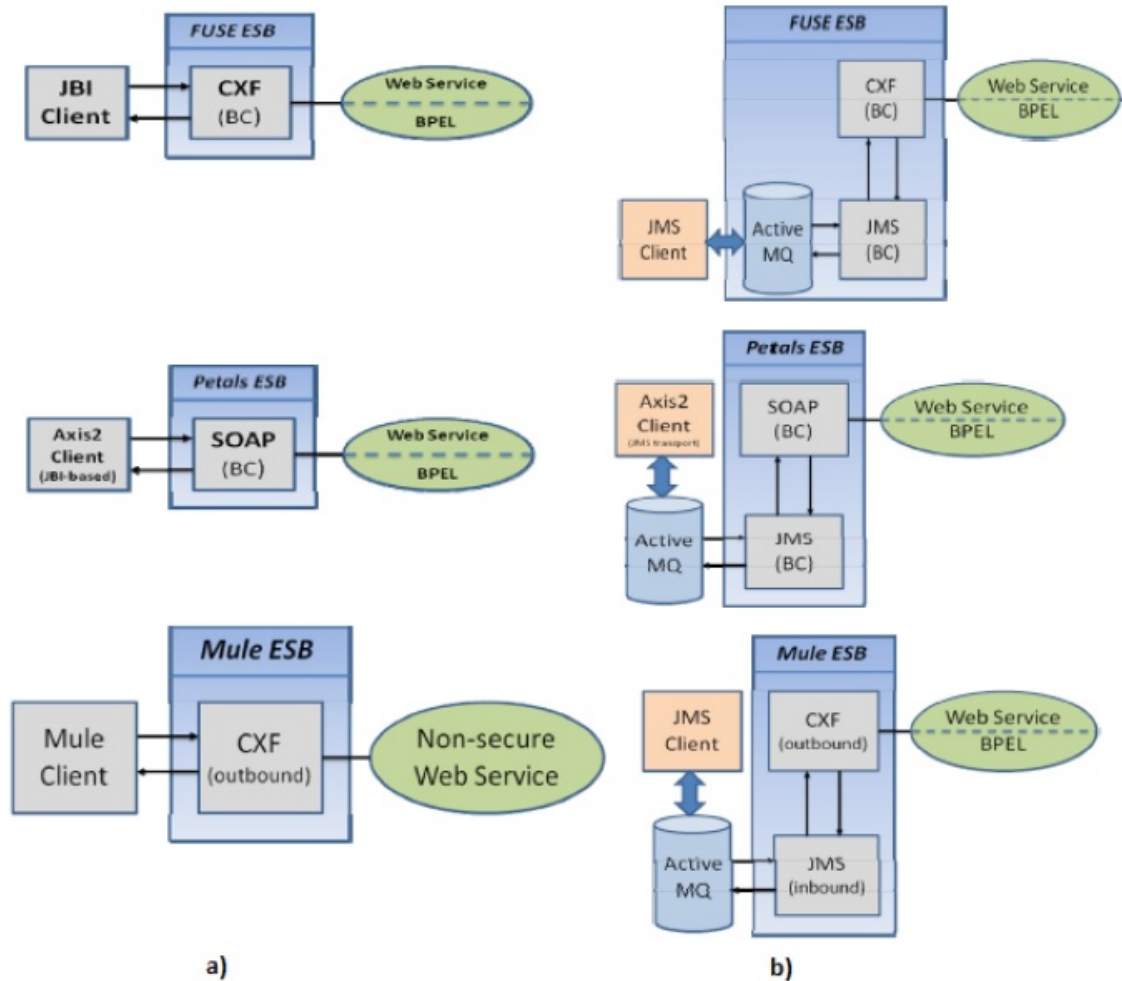


FIGURE 2.6: Direct Service Orchestration and BPEL a) First experiment. b) Second experiment [GJMCGS10].

Table 2.1 shows a survey made, showing the evaluation, the evaluated ESB(s), the environment under which the authors evaluated the ESB, the stress test tools used to evaluate the maximum capacity of the ESB, the application tested, a description of whether is Emulation or Metrology and finally the goal which shows why they evaluated the ESB.

TABLE 2.1: ESB Evaluation Survey.

Parameter	[UT06]	[API1]	[SJA]	[GJMCGS10]	[Bru]	[Bre09]
Evaluation	Response Time Throughput	Response Time Throughput (Proxy, Routing, Mediation)	Response Time Throughput (Service Orchestration)	Response Time Throughput	Security Issues Integration Issues	Response Time Throughput
ESB	—————	Mule ESB ServiceMix WSO2 ESB	Mule ESB ServiceMix JBoss ESB	Mule ESB	ServiceMix	Mule ESB Fuse ESB Petals ESB
Environment	1. No ESB 2. ESB (without Mediation) 3. ESB (with Mediation)	1. Direct Proxy 2. Routing Proxy 3. Transformation Proxy	1. Direct Orchestration 2. BPEL Orchestration	1. Inside one Server 2. Everything != Server	Observatory (Real Application)	1. No JMS 2. With JMS
Parameters	At Provider Multi @IP HTTP Keep Alive # of Threads Payload	At Consumer # of Clients Payload	At Consumer # of Clients Payload	At Consumer Payload until saturation	No Parameter	At Consumer # of Invocations
Application	HTTP Load Generator	Grinder	Apache JMeter Loan Broker	Apache JMeter Loan Broker	Real Application	Real Application
E/M	E	E	E	E	M	M
Goal	Capacity Planning	ESB Core Features	Orchestration Performance	Performance Issues Scalability Issues	ESB Issues	System Integration Evaluation

None of the aforementioned studies perform a study of all the ESB functionalities, this is beyond scope to this work. To conclude this chapter we present the main KPIs (Key Performance Indicator) of Quality of Service (QoS) ESB found in the studied related work. Some of these KPIs found are: response time, scalability, system saturation, transactions per second etc. . .

Hence there is not much works based on implementing a tool for the ESB's performance evaluation. Is a cloud based approach able to let us know the maximum performance of an ESB? Can we find out problems related to the QoS: scalability, high response times ? . . .

Therefore, we propose a cloud based Emulator; to discover potential problems of using an ESB as an integration backbone before the deployment into an organization's actual business.

Chapter 3

Emulator

3.1 Problem Statement

This work is situated in the context distributed integration systems, more specifically in the context of ESB as system integration middleware, it is interesting to model and characterize the QoS (Quality of Service). We want to be able to identify the limits of ESBs, for this we will have particular loaded scenarios. In this case we are interested on having a tool that allows:

- Comparing the execution of several scenarios for the same ESB model/characterize the QoS observed.
- Comparing several scenarios with diverse ESBs solutions.

3.2 Motivation

Let us start our motivation with a natural question.

Why there is a need for an Emulator?

There is a lot of ongoing research on SOA, middleware integration systems ESBs, and high performance systems. The concept of an emulator can be seen as an experimental platform to measure performance of different ESBs. An emulator gives the option to built complex network topologies, making it a tool for benchmarking the ESB's performance behavior in a readily manner.

We can't make predictions of the performance of an ESB. So we could say that predicting the performance of an ESB is different from predictions for traditional application

servers, this is because an ESB plays not just a server role but also plays a client role for multiple service providers. Meaning that the methods used for a J2EE server's performance evaluation [Bre09] aren't suitable for an ESB.

With an Emulator we can evaluate the ESB system capacity with a small hardware environment. The mediation functionalities also cause some differences in how ESB performance is evaluated, compared to evaluating simple intermediaries like TCP/IP network routers. Nevertheless, performance estimation of an ESB in the capacity planning phase, which happens at a very early stage in the project life-cycle, is critical for a successful project [UT06].

With an Emulator we could select an ESB that best fits the needs of an enterprise, at low price. Given the particular characteristics of SOA application and its environment, many organizations would like to evaluate ESBs, because this evaluation is less costly and time consuming than actually implementing SOA application or even purchasing an ESB. In other words, the evaluation can help organizations to select an ESB. Or it could be an open source ESB, but an enterprise wants place it as part as their organization. Hence, to improve some aspects of the ESB, first we need to know the issues.

The evaluation of some capabilities (routing, transformation, service orchestration) of an ESB are also another reason. It is important to know if an ESB has scalability problems, caused by the use of the previous features.

Previous works show different evaluations and provide interesting results. However:

First They often compare commercial ESBs only.

Second They often set out too general evaluation criteria (i.e Orchestration assesment) that are useful for categorizing ESBs or judging providers only.

Third They lack some important evaluation criteria e.g. high availability, reliability etc ...

Fourth They don't report scalability problems.

Concluding this section we could say a framework allowing to deploy configurable scenarios, to test it and to adapt it is needed. Our proposal (emulator) can be built on a small environment compared to a real life application, but giving the same characteristics.

3.2.1 Different ways to Evaluate Distributed Systems

In this section, a brief definition of distributed system is given, then we justify why emulation is used.

A distributed system should make resources easily accessible, it should reasonably hide the fact that resources are distributed across a network, it should be open and it should be scalable [TVS02]. In system integration the component that glues the services and applications must consider the former definition.

There are diverse forms of evaluating distributing systems using: mathematical functions, simulations, an experimental approach, and emulation.

1. **Mathematical functions** studied for traditional application servers, for which many studies have been done; aren't suitable for predicting the performance of an ESB [UT06], because of their wide features.
2. **Simulation** is the imitation of the operation of a real-world process or system over time, for example: NS3, NS2, and OPNET. Currently there's no simulation platform that deploys an ESB.
3. **Experimentation or metrology** the science of measurement it includes all theoretical and practical aspects of measurement. For organizations it is way too expensive interrupting an already deployed service used for testing the ESB. Also, organizations using an ESB as their integration backbone usually have complex topologies, and business process.
4. **Emulation** is an alternative way to improve software quality for distributed applications, allowing analysis of the application behavior in a given environment before deployment. To evaluate distributed systems, virtualization can be applied, as it allows creation of a large-scale controlled environment with few physical resources [SCDR].

For these reasons our approach adopted emulation as an alternative, which fulfills the evaluation of an ESB. To conclude, the concept of an emulator can be seen as an experimental platform to measure performance of different ESBs. An emulator gives the option to deploy many scenarios with different topologies; implemented with few resources environment compared to a real life application, but offering the same characteristics. Indeed, allowing us to specify the exact environment conditions for the scenario.

Our approach deploys a Virtual Machine (VM) for the client side, it also deploys a VM for the system holding the ESB software and it deploys another VM for the Web Service Provider.

3.3 ESB's General Environment

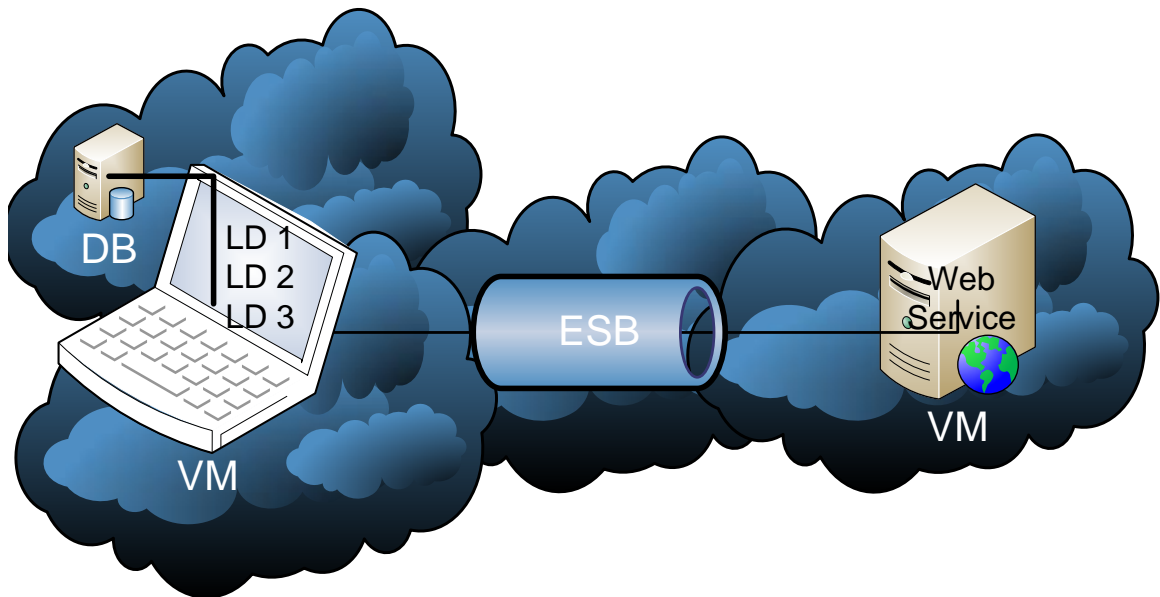


FIGURE 3.1: Most general environment.

Figure 3.1 intends to show the most general view of the system we are proposing. Here we have three basic components that will hold the system deployed. The figure shows a Load Driver held inside the cloud in a VM (Virtual Machine), the Load Driver will be in charge of sending concurrent requests to the ESB. The ESB is deployed in the cloud, being inside another VM. The Server is also a VM, it will be holding the web services applications for the provider side.

For the VM containing the end Web service, we need to be careful about its resources due to the fact that we want to measure the capability of an ESB, so it's important that the provider side does not become a bottleneck. For these reasons, we need to allocate enough resources to the machine holding the Server. The same considerations must be taken into account for the machine holding the client side, here the system has to be able to send many requests concurrently, so it needs to be able to create many threads and without becoming a bottleneck. More information may be found in Chapter 4 in section "Avoiding Bottlenecks".

Figure 3.1 shows that we are following an architecture like the one deployed in [[UT06], [AP11], [SJA], [GJMCGS10]], but in this articles they deployed their work in different machines, we are also going to deploying ours in different machines but with the option of dynamically changing the resources, so we could say we are not limit to the capacity of the systems holding the VMs.

3.3.1 General Use Case

With our emulator the user will be able to Create Scenario, Launch Scenario and to View Results.

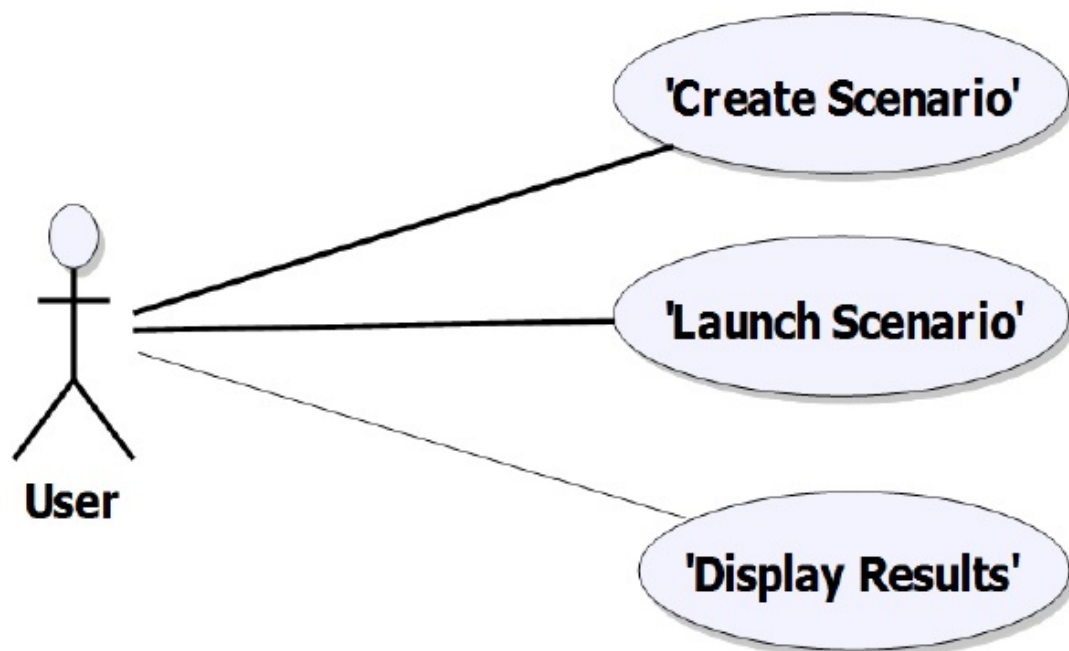


FIGURE 3.2: General Use Case.

Figure 3.2 shows the functionality of the system, this use case diagram has a high level abstraction.

To Create Scenario the user has two options, the first one is to choose the already deployed services (client and producer) and test the performance of the an ESB. The second option is to upload a web service provider, so that the user can test the performance of the new deployed service.

To Launch Scenario, the user can choose from existing scenarios to directly run one of them, or he could Create an Scenario using the other use case established.

To Display Results, the user can display results after the emulation of the whole system finished running. The user will be able to monitor the CPU usage of the system holding the ESB, other parameters that a user will be able to view are the TPS, and the mean response time.

3.4 Emulators Architecture

We propose an agnostic emulator, this means any ESB shall be able to be plugged in straightforwardly. However, some adaptive actions have to be made inside the corresponding ESB to compute relevant timestamps.

Stress tools like: Grinder and Apache JMeter are open source but others like SoapUI have commercial versions. These tools permit having different QoS metrics, like: average response time and TPS, but their flaw remains in not being able to monitor the CPU and Heap Memory of the ESB system. As consequence we propose an architecture to be able to heal the aforementioned flaws.

Figure 3.3 shows the principal components involved in the emulator's architecture. As main components we have the *Service Consumer* which is in charge of communicating with the ESB. The *Controller* is responsible for displaying graphs and information retrieved from the execution of a scenario (the results). This component has the responsibility to invoke each Load Driver (LD). A third component, the *Configuration Driver*, stores the number of invocations that each LD can achieve, and it contains the sleep time emulated at the provider side and the response data size in bytes the providers must generate. The *Generator*, its name says it, generates each invocation made from a LD to a service web (provider). Finally, the *Monitoring* it is a standalone instance in charge of monitoring the ESB's Heap Memory and CPU usage. Nevertheless, we want to stress the ESB to its maximum capacity, as consequence the LDs have to generate multiple concurrent request alike the aforementioned tools. To determine the maximum performance of the ESB system, some considerations must be taken into account; section 4.2 "Avoiding Bottlenecks" describes such. In subsection 3.4.1 we describe the component diagram, which represents the dependency between software components and how they are divided.

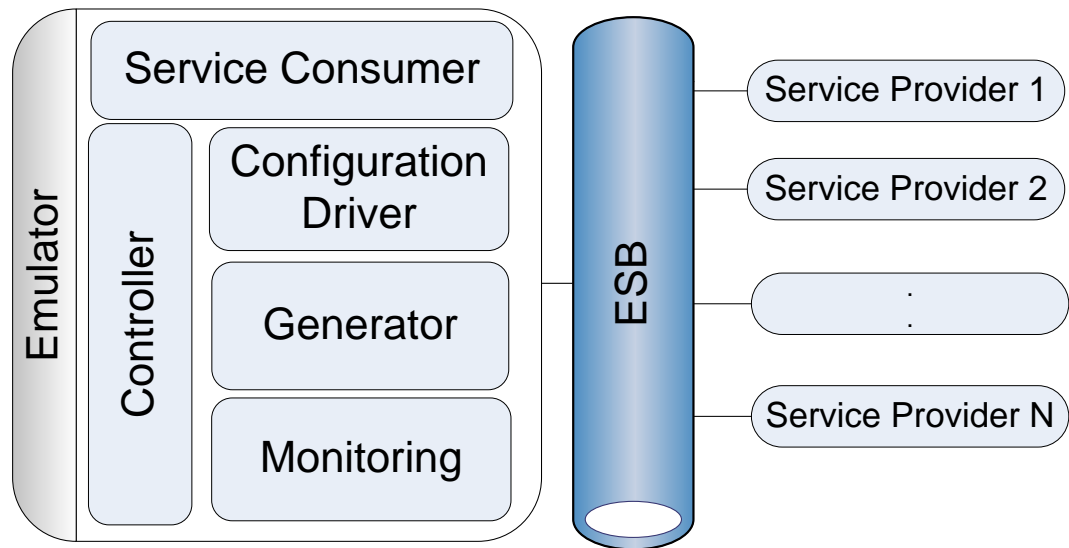


FIGURE 3.3: Emulators Architecture.

3.4.1 Emulator's Components General View

Figure 3.4 illustrates a general view or black box of the approach for the Emulator, as we can notice from the depicted Figure 3.4. The ESB is outside of the emulator, meaning that our Emulator is independent of the ESB used. Figure 3.4 intends to show the interaction between the different components involved, Emulator as a whole and the ESB as the system evaluated. The user will have the option to parametrize the scenario to be emulated, and to retrieve the results from it, with *UsertoEmulator* and *EmulatortoUser* respectively. Also, the Emulator is able to configure the ESB and retrieve data from it, with the *EmulatortoESB* and *ESBtoEmulator*.

This component diagram has a high abstraction level, the next step is to open the black box, shown in the next section, and wire all the components and interfaces needed to form the software system. Where the component is required to execute a function.

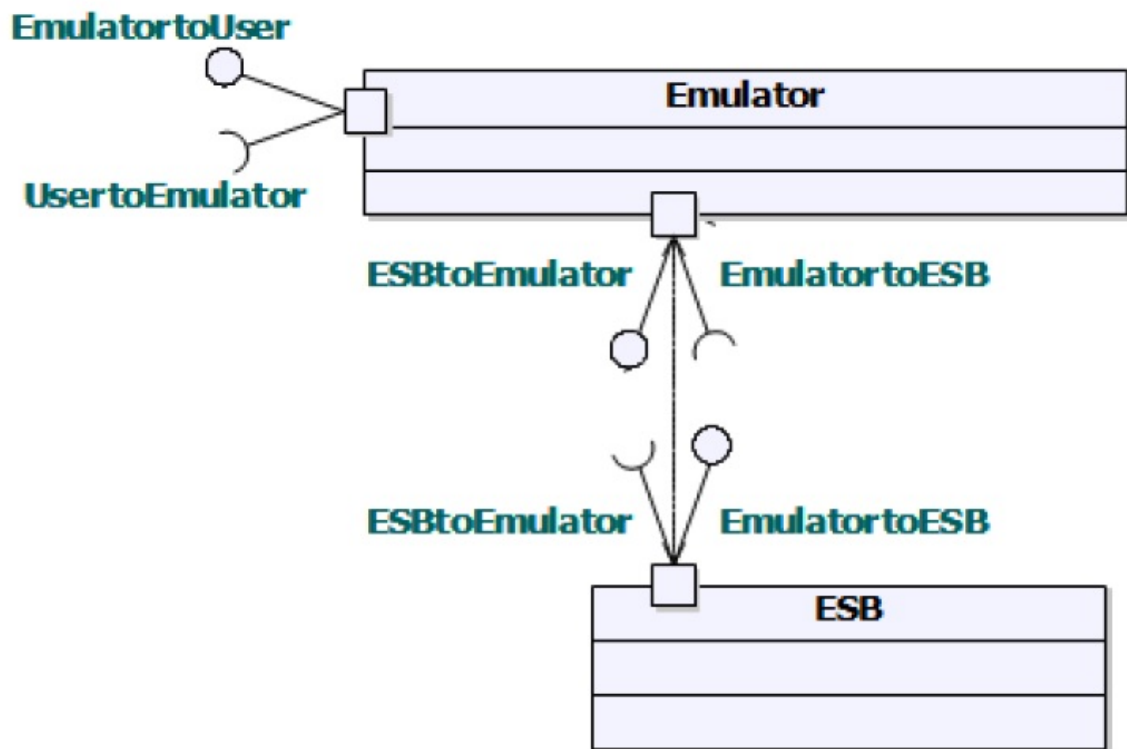


FIGURE 3.4: Black Box.

3.4.2 Emulator's Component Diagram

We intend to separate each of the components involved like the Coffee Machine example of [Exp13]. Figure 3.5 depicts a deeper look into the components that the emulator must have, as we could see the main component is called **Controller**. This Controller will be in charge of distributing parameters and knowing what task is executed by the user. The **Generator** is in charge as its name says to generate concurrent users to stress test the ESB. The **Config** goes in action when the user wants to add a new web service or the ESB must be configured to perform a mediation action. Finally, the **Monitor** will be in charge of monitoring the ESB, mean heap memory usage and CPU usage.

The components diagram is useful since it allows the reuse of previously constructed components and it also takes into account that a component could be replaced by another if needed, for example if the requirements change or due to an update the system.

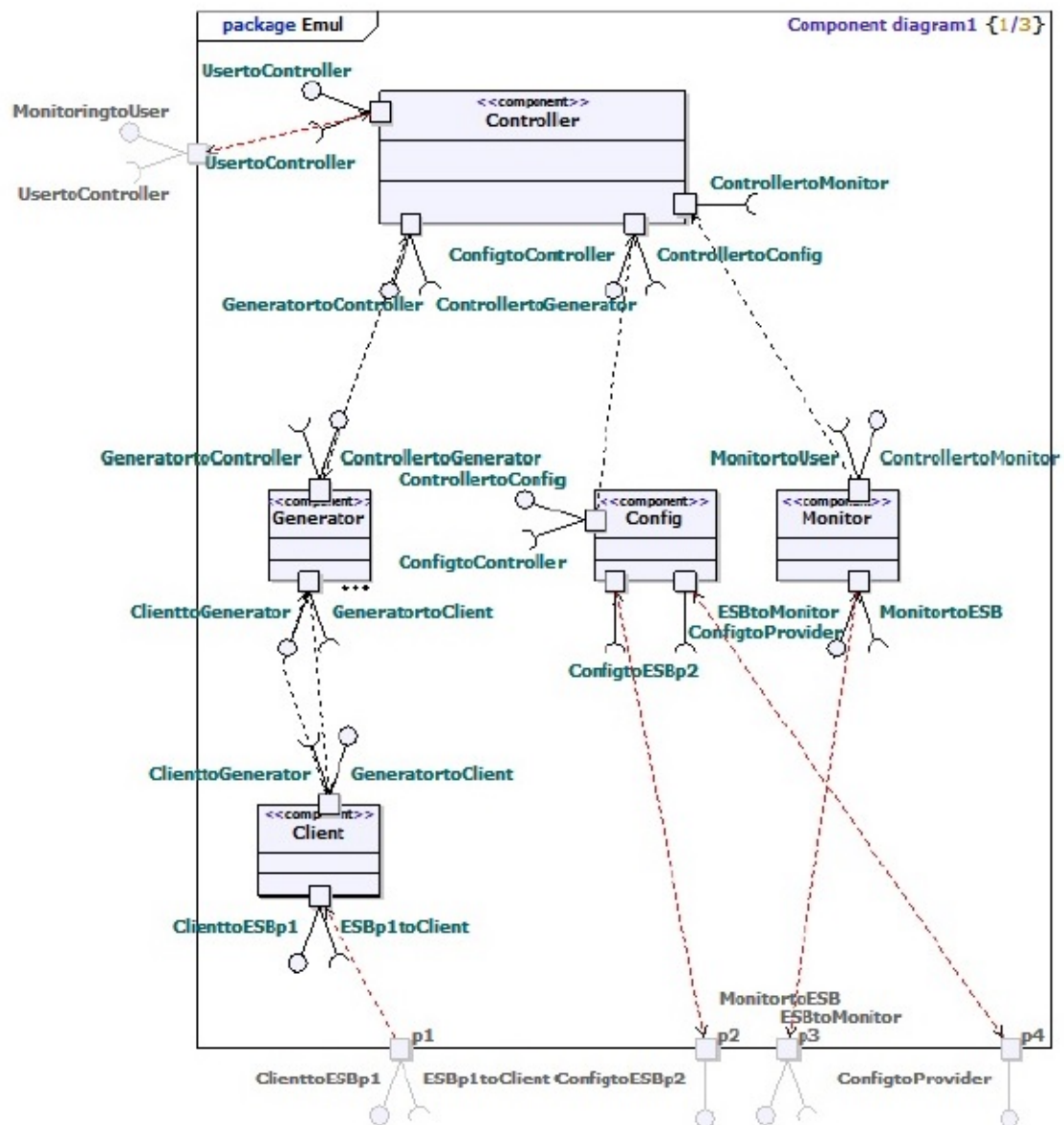


FIGURE 3.5: Component Diagram.

3.5 Current Environment

First we describe the cloud based environment at which the emulator is currently deployed. The environment consists of a PROXMOX Virtual Environment¹ where the required VMs for the tests are currently deployed. Each component has a Linux Ubuntu 12.04 LTS x86 as operating system, with a QEMU Virtual CPU version 1.050. To be clear about the environment we present the Figure 3.6. In the Figure 3.6 we deployed many instances of LD (Load Drivers), these are deployed inside a VM depending of its characteristics. For a VM with 4 GB of RAM, it could create at most three instances; due to the fact that each LD is assigned with a finite amount of resources for its Java

¹<http://www.proxmox.com/>

Virtual Machine (JVM). The ESB server system is hosted inside a VM with 2 GB of RAM, specially dedicated to its JVM, currently running WSO2 ESB version 4.6.0. Finally, each service provider hosting the end Web service is hosted inside a different VM, with 4GB of RAM running WSO2 Application Server (AS) version 5.0.1.

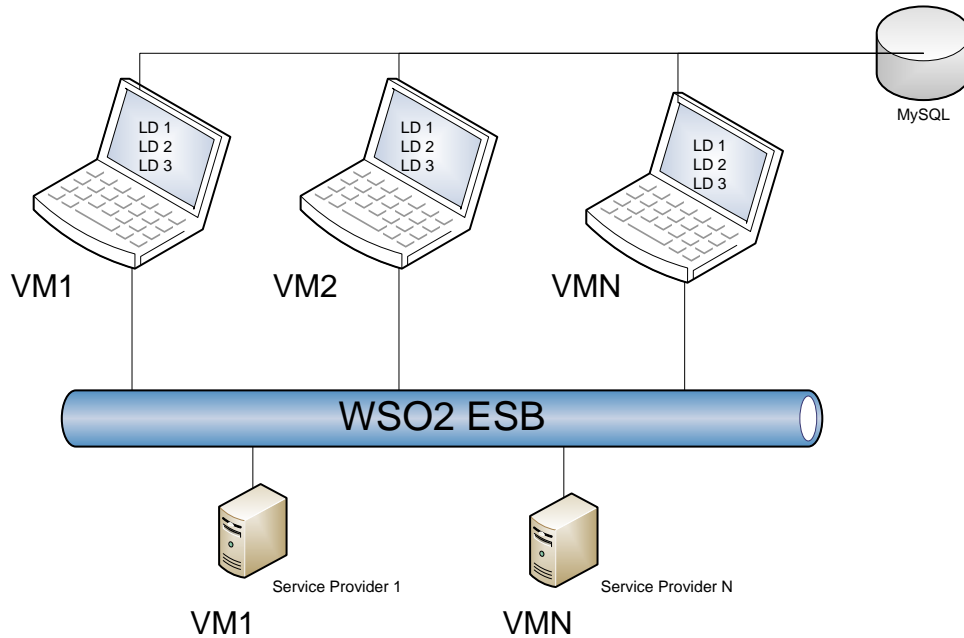


FIGURE 3.6: Current Environment.

Table 3.1, summarizes the characteristics each of the VMs.

TABLE 3.1: Characteristics of VMs

Component	Number of CPUs	Memory
Load Driver (LD)	2	1024 MB
ESB	1	2048 MB
Application Server (Providers)	2	4096 MB

Chapter 4

Results and Analysis

For the results we consider using stress test tools like SoapUI and Apache JMeter, with them we could *overwhelm* the ESB to its maximum capacity. SoapUI: is an open source web service testing application for service-oriented architectures (SOA). Its functionality covers web service inspection, invoking, development, simulation and mocking, functional testing, load and compliance testing. Apache JMeter: is an Apache project that can be used as a load testing tool for analyzing and measuring the performance of a variety of services, with a focus on web applications.

The next section shows some of the test made with SoapUI; this tool gave us some ideas of how to *attack* the ESB, unfortunately it does not monitor the CPU nor the Heap Memory of an ESB. We also executed some test with Apache JMeter, but again it lacks on how to monitor the ESB. As consequence we decided to deploy our own tool; it is merely written in JAVA.

4.1 Test with SoapUI

We use the SoapUI tool to stress test the ESB. With SoapUI we could send a concurrent number of request to an endpoint, we took the same environment deployed in the Figure 3.6. But this time instead of having a Java Client (LD) running multiple threads, we used SoapUI that generates multiple threads to invoke the service provided by the WSO2 AS.

We run this scenario for 2 minutes with three different configurations of the machine holding the ESB having the next characteristics:

- 1 GB of RAM, 1 Processor
- 2 GB of RAM, 2 Processors
- 3 GB of RAM, 3 Processors

We wanted to know the average response time, when we have a fixed payload of 1 Kb, but varied the number of concurrent clients invoking a Web service passing through the ESB. From Figure 4.1, we could see that the average response time is quite similar for all the configurations of the machine holding the ESB. This is maybe, because the ESB is not working hard enough, what we mean by this is that the ESB is not performing or using any of its mediation features. We also monitored the behavior of the CPU usage holding the ESB, and we can conclude that the ESB is able to manage simple messages passing through it with no difficulty, since the CPU usage never overpassed 40% of its usage, not even for the first test ran with the machine holding the ESB configured with 1GB of RAM and 1 Processor. Further, other tests must be made but this time to evaluate one of the core features of an ESB (mediation, transformation and routing).

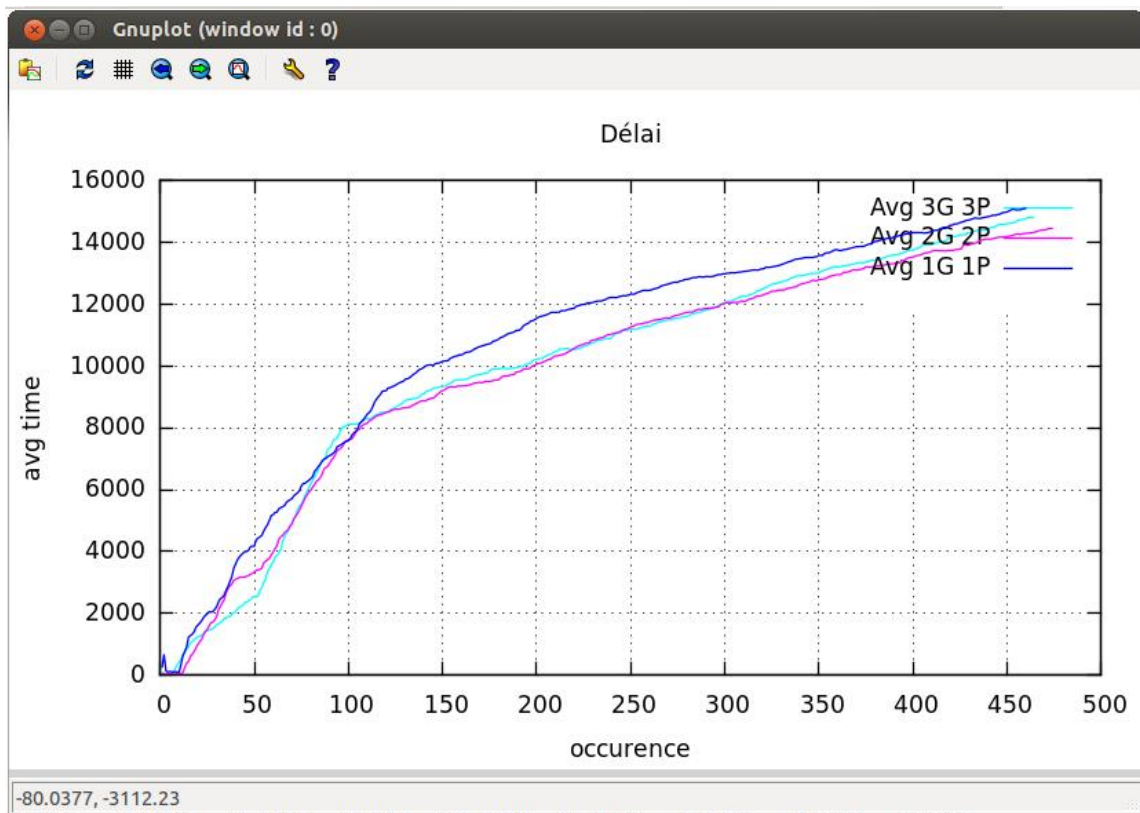


FIGURE 4.1: Results using SoapUI.

Plus we have to be sure the consumers and providers of a service passing through an ESB do not become a bottleneck, which can be a misconfiguration while evaluating the ESB.

4.2 Avoiding Bottlenecks

Using our emulator, we emulate a set of scenarios to identify the impact a failure at the provider or consumer side, will have against its counterpart. For example what happens to the consumers of a service when the provider becomes a bottleneck. The answer is simple, the consumers will receive a higher response time, than when the provider is under normal conditions.

The Figure 4.2 illustrates, a set of service consumers interacts with a provider. The number of service consumers can be parameterized. We can also configure for each service consumer the provider that it invokes, the number of sending requests, the size of these requests. At the provider side, the processing time can be emulated and the size of the response can be parameterized. The resources allocated (physical or virtual) to the consumer, the ESB or the provider can be configured. After running the scenario, a set of metrics can be taken with the emulator.

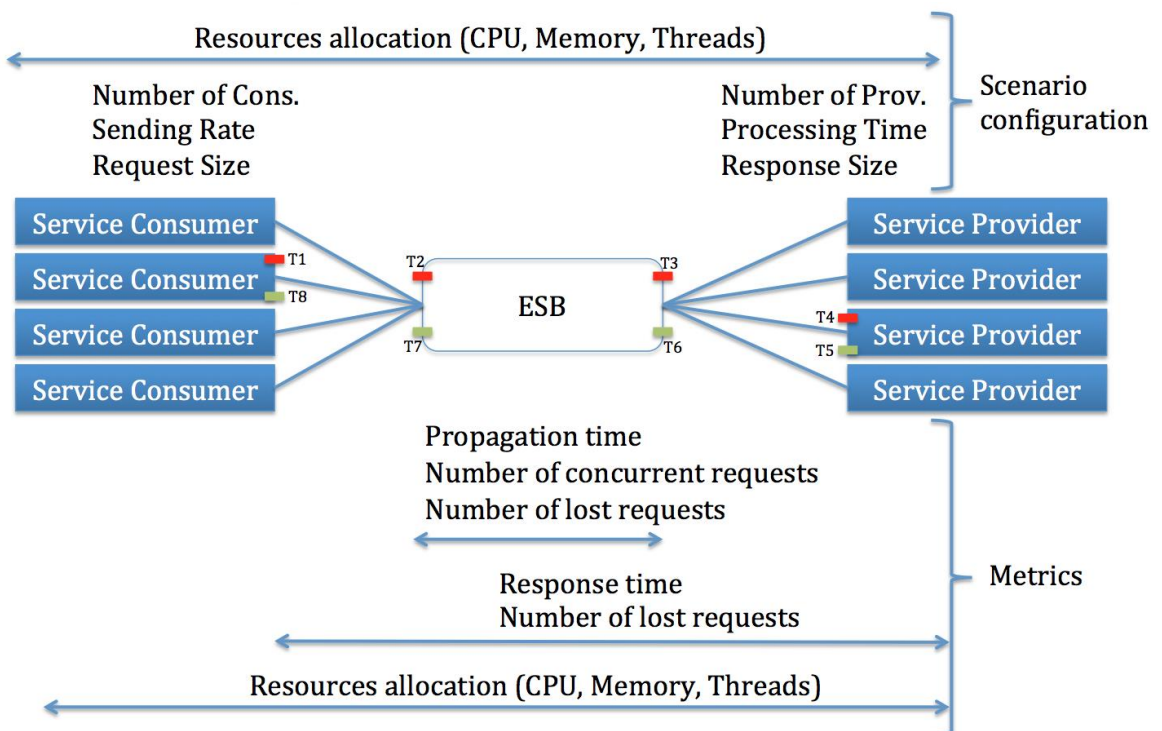


FIGURE 4.2: Scenario Topology and Metrics.

4.2.1 Performance issues from the provider side and impact

The first three phases (1,2,3) intent to illustrate the AS' CPU usage, and the response time computed for the consumers, phases 1,2 and 3 represent a scenario. What we want to find out is what occurs when high processing time or high CPU usage is present at the AS, and how it will affect the consumer and ESB side.

The next configurations were used to test the AS.

- Phase 1 is configured as follows: 4000 concurrent requests at the client side and the service provider emulating 100 ms sleep time and generating 1 KB as reply answer.
- Phase 2 is configured as follows: 8000 concurrent requests at the client side and the service provider emulating 100 ms sleep time and generating 1 KB as reply answer.
- Phase 3 is configured as follows: 12000 concurrent requests at the client side and the service provider emulating 100 ms sleep time and generating 1 KB as reply answer.

The Figure 4.3 depicts the WSO2 AS resources monitored with JConsole, it shows that the AS is running on high CPU load even though it has Heap Memory. Causing a high RT shown in the Figure 4.4. The Figure shows that the LD only sent 4000 requests, and that the delay indeed is due to the AS.

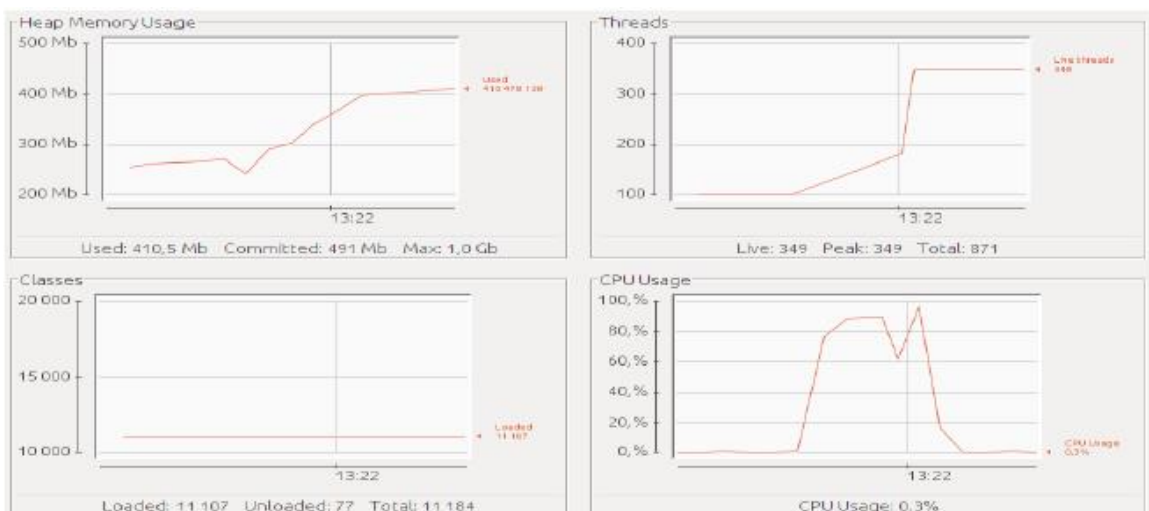


FIGURE 4.3: Monitoring AS using Jconsole (Phase 1).

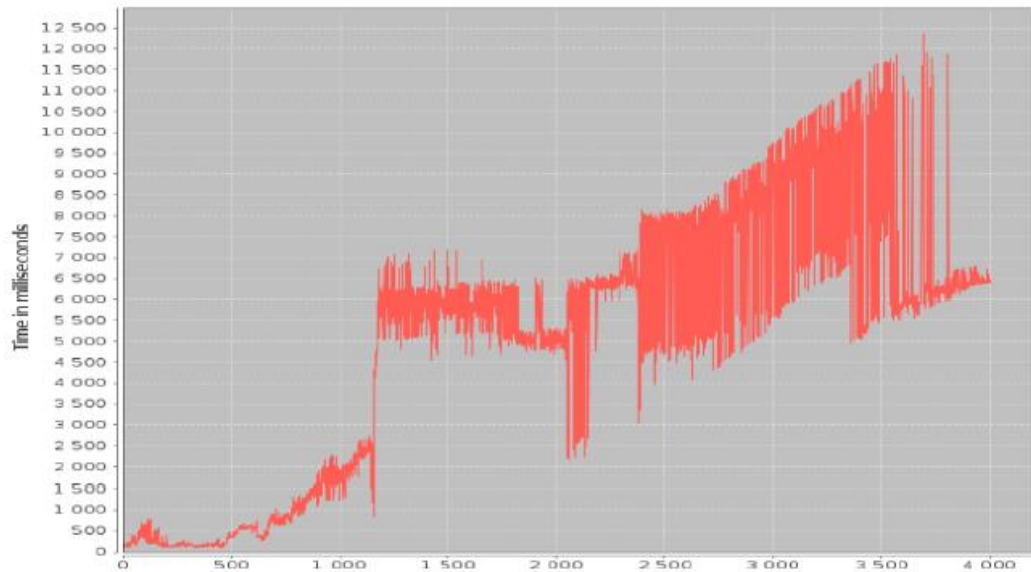


FIGURE 4.4: Computed Response Time (Phase 1).

Figure 4.5 depicts the AS resources, it shows how the CPU processing time is slightly increased due to the fact that we doubled the number of request and therefore we used two LDs. Figure 4.6 shows an increased number of requests, and that the average RT triggered to higher values.

Figure 4.6 shows an increased RT having as average 15s. We could infer that something is wrong having this kind of values. Therefore, the correlation between the consumer and the AS is indispensable and notable.

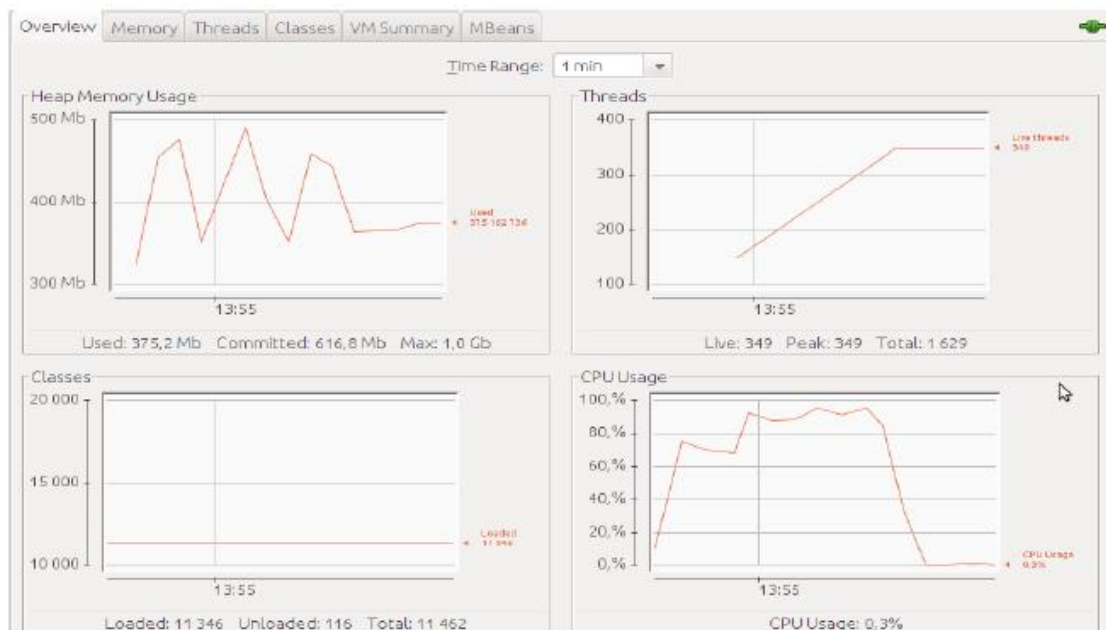


FIGURE 4.5: Monitoring AS using Jconsole (Phase 2).

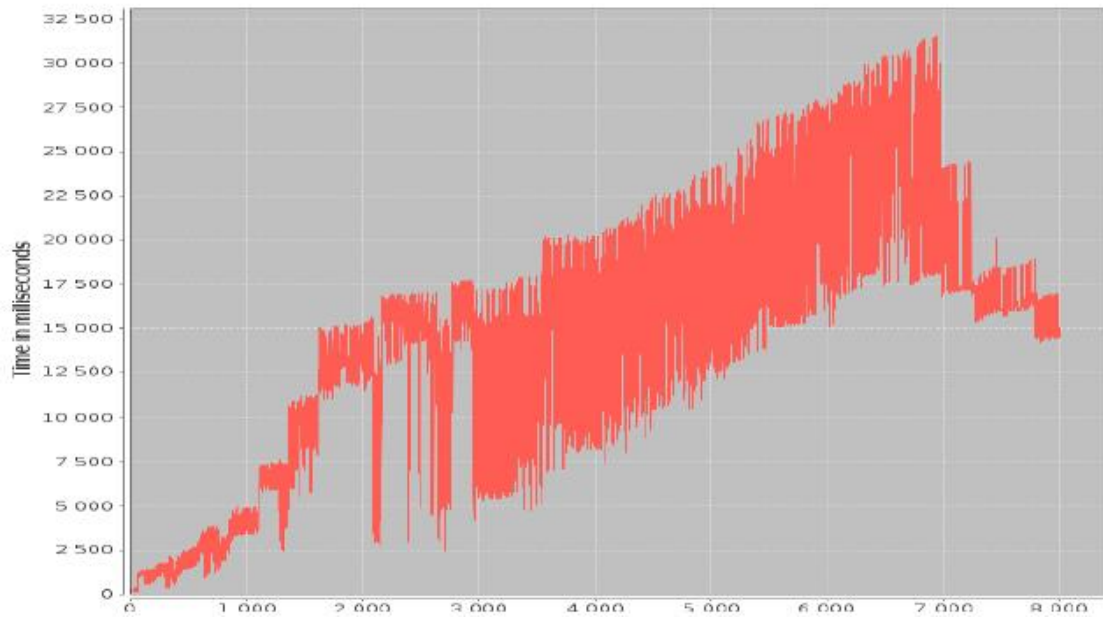


FIGURE 4.6: Computed Response Time (Phase 2).

Figure 4.7 presents an AS with quite lower CPU usage, but the thing here is that the processing time was from 1 min (as shown in figure 4.5) to 5 minutes.

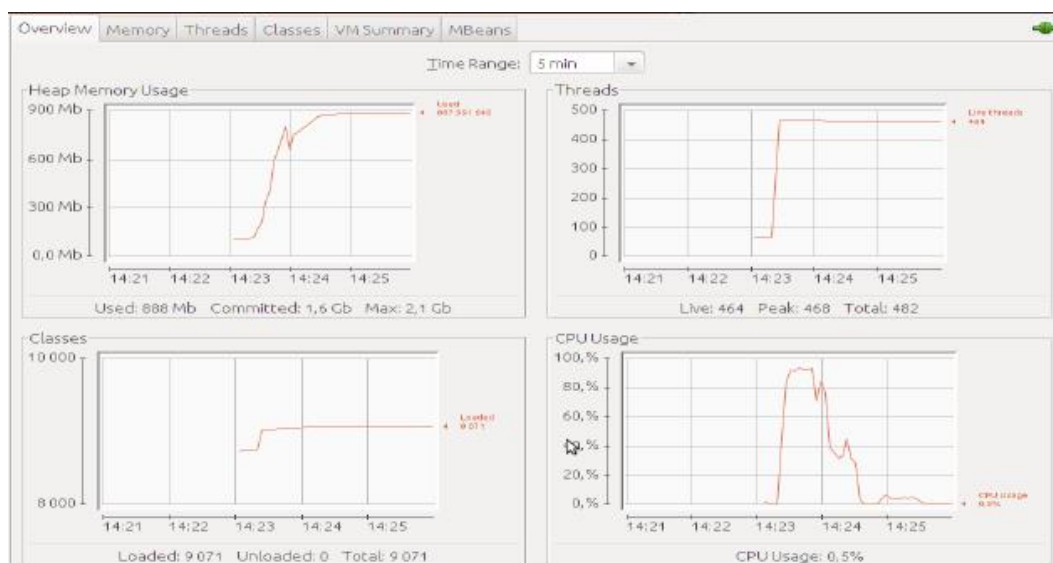


FIGURE 4.7: Monitoring AS using Jconsole (Phase 3).

The Figure 4.8 illustrates an average RT of 25s. Even though it was quite an amount of requests (12,000) sent to the AS, we didn't expect this kind of RT. The AS queues the requests till it has enough CPU to deal with them.

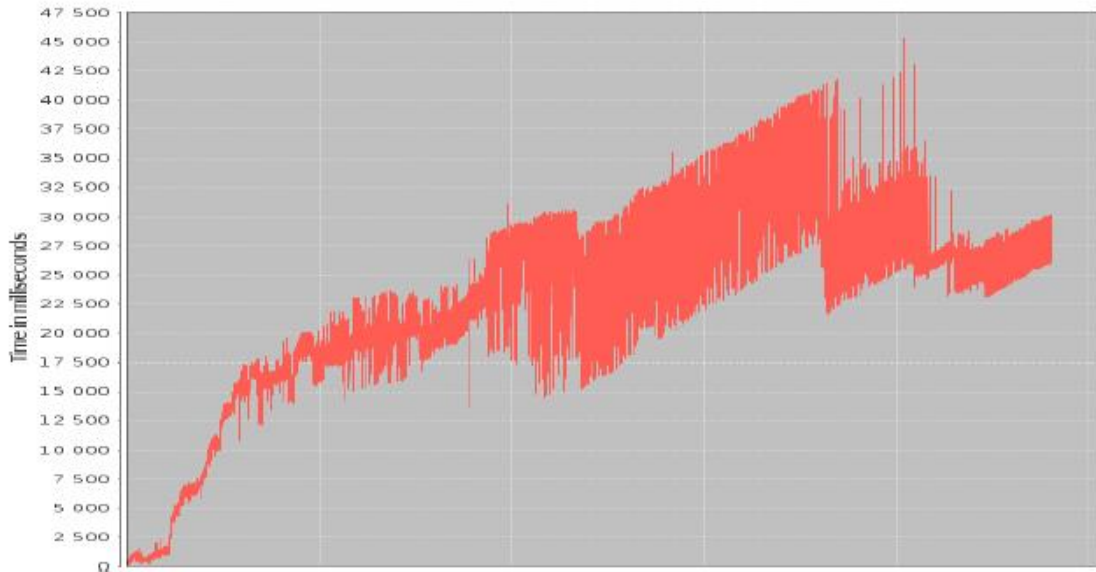


FIGURE 4.8: Computed Response Time (Phase 3).

Finally, in the next phases we created a reliability problem or issue, the AS becomes a bottleneck. Replaying the scenario of Phase 1 but this time increasing the payload the AS must generate, as follows:

- Phase 4 is configured as follows: 4000 concurrent requests at the client side and the service provider emulating 100 ms sleep time and generating 6 KB as reply answer.
- Phase 5 is configured as follows: 8000 concurrent requests at the client side and the service provider emulating 100 ms sleep time and generating 6 KB as reply answer.

The average processing time per request is high 1890 ms and the average response time is higher than the one calculated in Phase 1.

The Figure 4.9 shows an AS CPU completely overloaded, but still attending each of the 4000 requests received.

Figure 4.10 shows the RT of the 4000 requests sent. The RT is always increasing, because the AS queues the requests and the consumer is pending till it receives an answer.

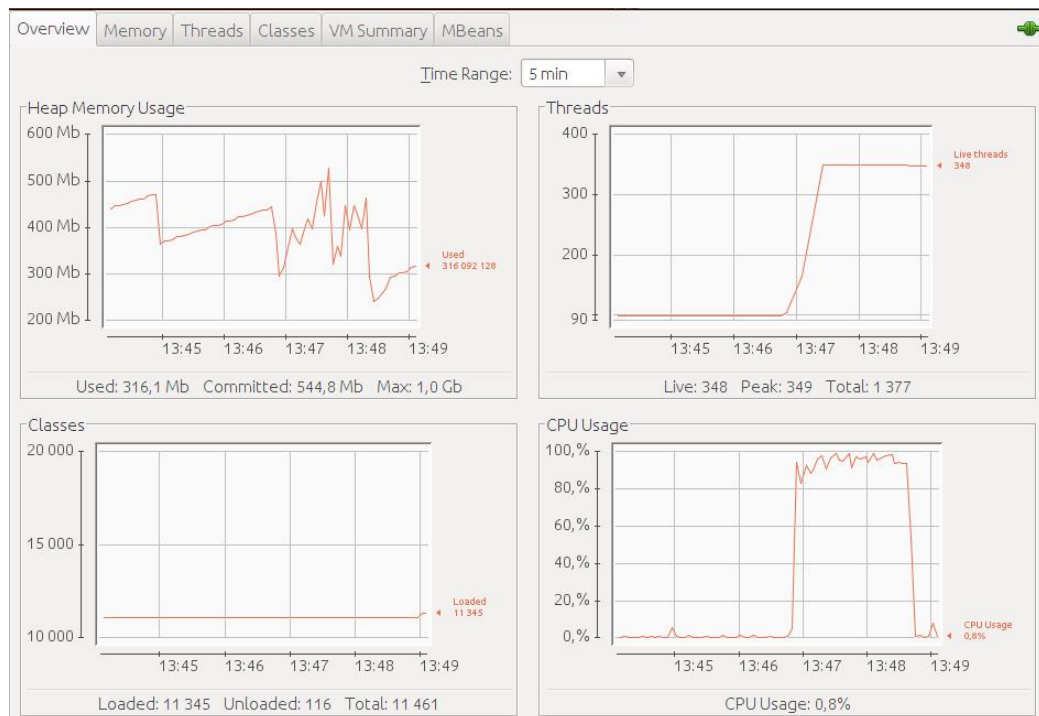


FIGURE 4.9: Monitoring AS using Jconsole (Phase 4).

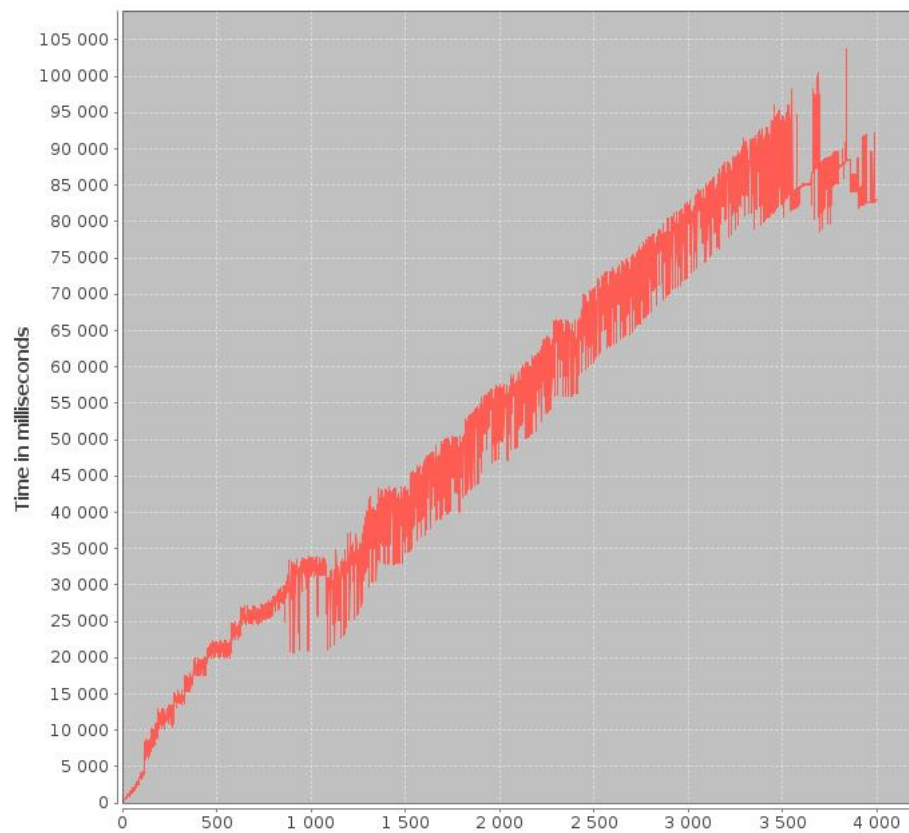


FIGURE 4.10: Computed Response Time (Phase 4).

Using the same configuration of the Phase 4 and putting the number of concurrent requests to 8000, we create the reliability issue at the AS due to a lack of resources. The relation between provider, ESB and consumer makes we detect losses at the consumer or user side.

Figure 4.11 shows the AS curve of the CPU usage. This figure illustrates when the AS becomes a bottleneck. This curve has high and low peaks, meaning that the AS cannot manage the burst of requests. Hence, the AS is in charge of generating the response size data, in this case of 6 KB, it queues requests till it has the ability to perform the aforementioned task.

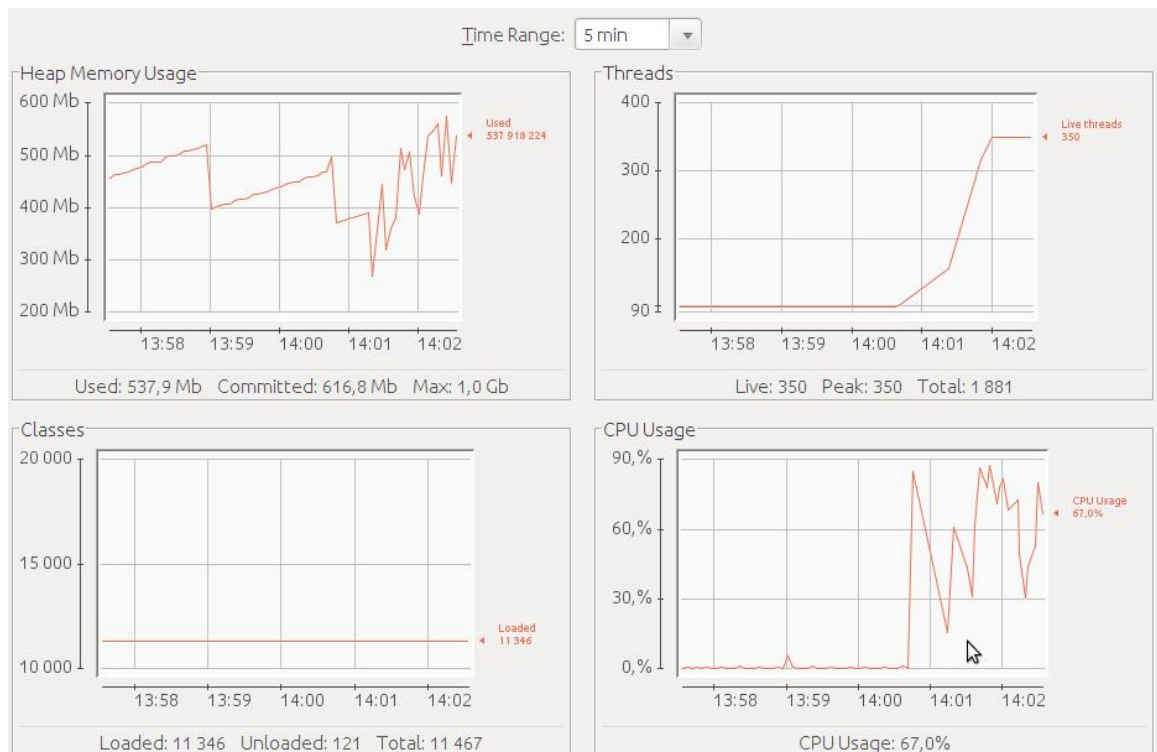


FIGURE 4.11: Monitoring AS using Jconsole (Phase 5).

Finally, the Figure 4.12 depicts that from the 8000 initial requests made from the consumer (Phase 5) only 1750 were attended correctly by the three actors of this scenario: consumer, ESB and AS (end Web service). This Figure intends to depict the response time calculated for Phase 5, as shown in previous Phases 4, 3, 2 and 1, but with the difference that this Phase didn't get all the reply back.

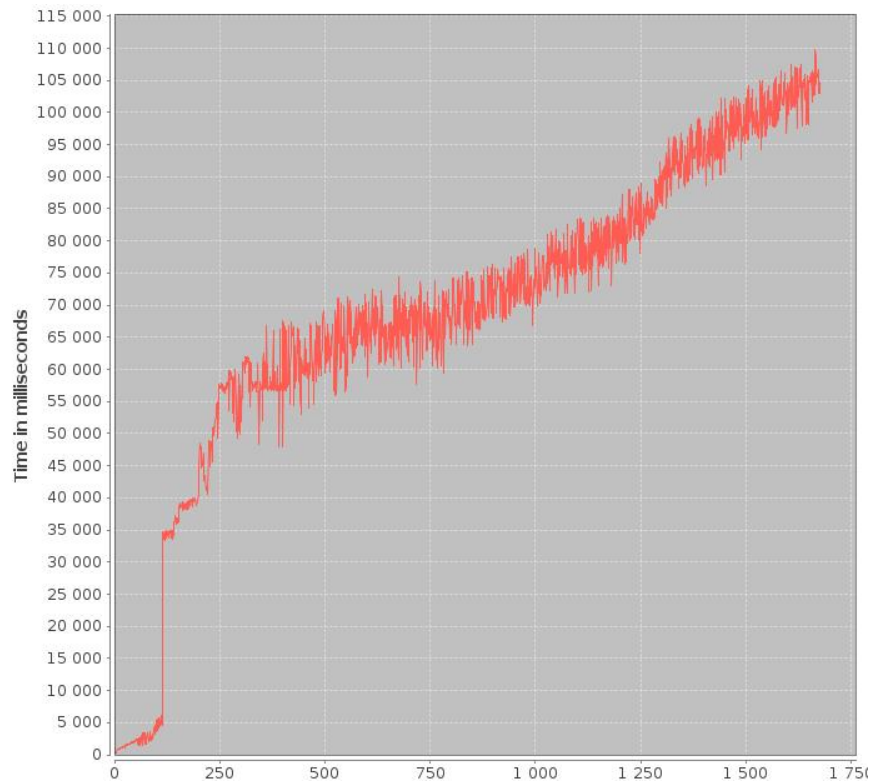


FIGURE 4.12: High Response Time and losses detection issue.

Figure 4.13 shows that it is not feasible having a provider with small amount of resources. As shown in this figure the processing time is way above the expected emulated time of 100 ms. Hence, the provider is consider as a bottleneck.

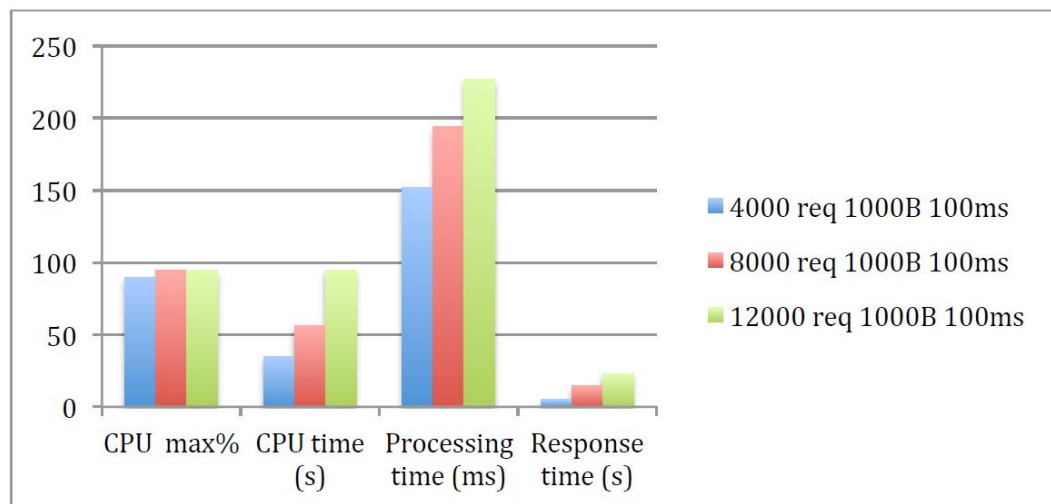
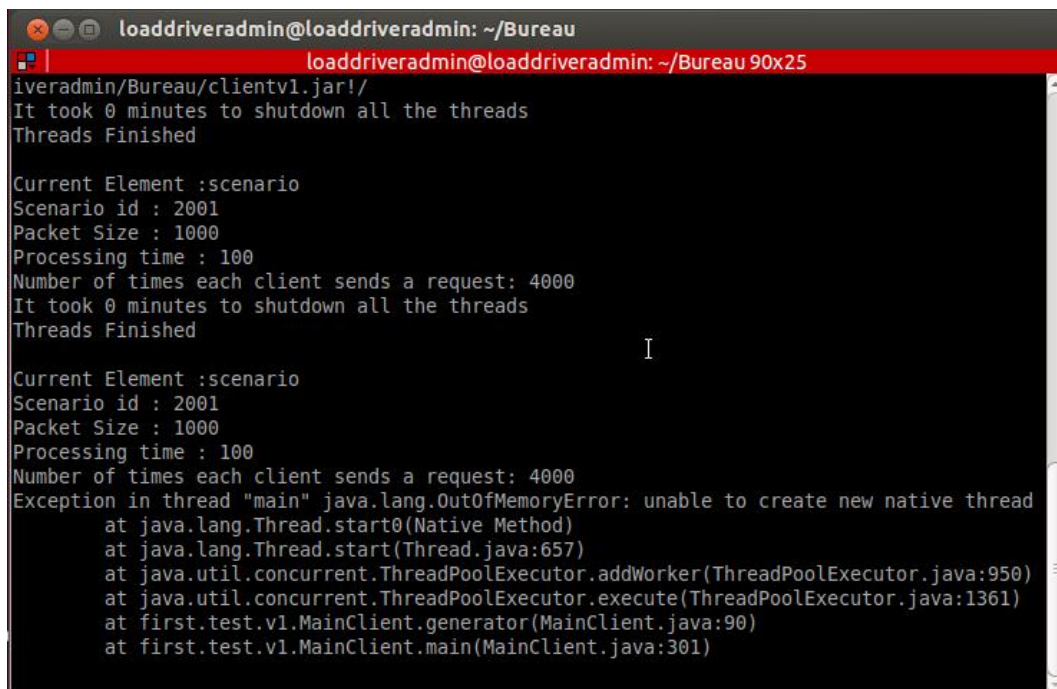


FIGURE 4.13: Resume of three phases.

4.2.2 Performance issues from the consumer side and impact

We create a client congestion out of heap memory by reducing the allocated resources and sending 8000 requests.

Figure 4.14 depicts a problem when the consumer does not have enough resources to create a thread or a consumer of the service. The Figure shows one of the two LD finished its tasks, but the other didn't. Since, the LDs are executed concurrently it could of been either one which failed. And let's remember that a LD has specially dedicated resources to its JVM as mentioned in section 3.5 "Current Environment".



```
loaddriveradmin@loaddriveradmin: ~/Bureau
loaddriveradmin@loaddriveradmin: ~/Bureau 90x25
iveradmin/Bureau/clientv1.jar!/
It took 0 minutes to shutdown all the threads
Threads Finished

Current Element :scenario
Scenario id : 2001
Packet Size : 1000
Processing time : 100
Number of times each client sends a request: 4000
It took 0 minutes to shutdown all the threads
Threads Finished

Current Element :scenario
Scenario id : 2001
Packet Size : 1000
Processing time : 100
Number of times each client sends a request: 4000
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:657)
    at java.util.concurrent.ThreadPoolExecutor.addWorker(ThreadPoolExecutor.java:950)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1361)
    at first.test.v1.MainClient.generator(MainClient.java:90)
    at first.test.v1.MainClient.main(MainClient.java:301)
```

FIGURE 4.14: Out of Heap Memory at the consumer side.

4.3 Identified Problem while Evaluating the ESB's Performance

Figure 4.15 shows a need to compute the real times that a message (request or invocation) passing through the ESB must take into account.

To solve the problem of not knowing the times shown in the Figure 4.15, we simply wrote to a LOG file inside the ESB, for each time a request is processed in the ESB. To

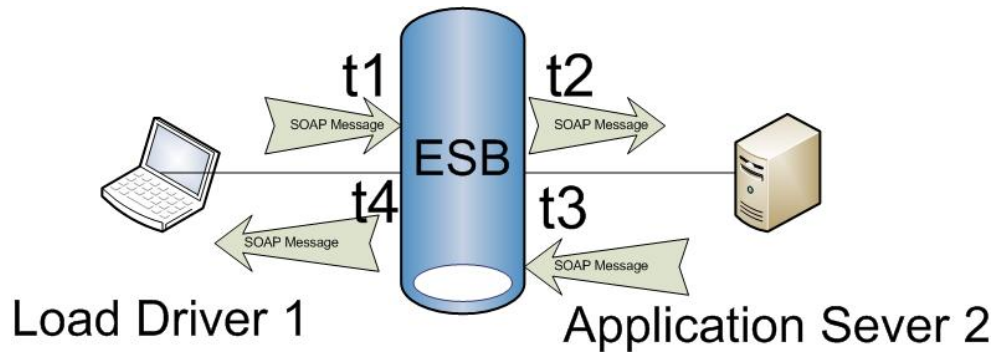


FIGURE 4.15: ESB Times.

be clear about what the ESB writes to a log file, next we give an example of a processed message:

- [2013-06-22 18:59:30,325] INFO - LogMediator testing = Request arrives from Client, Time t1 = 1371920370325
- [2013-06-22 18:59:30,326] INFO - LogMediator testing = Request sent to Provider, Time t2 = 1371920370326
- [2013-06-22 18:59:30,331] INFO - LogMediator testing = Request received from Provider, Time t3 = 1371920370331
- [2013-06-22 18:59:30,333] INFO - LogMediator testing = Request sent by the ESB, Time t4 = 1371920370333

But what is the cost of retrieving all these timestamps?

4.3.1 What is the cost of monitoring the ESB?

Two different scenarios were deployed in order to analyze the cost of monitoring the ESB completely. In the first scenario the ESB acts as a pass through proxy. In the second scenario the ESB also acts as a pass through proxy but it also writes some timestamps into a log file.

The first and second scenarios were tested with the following configurations:

- Fixed number of concurrent requests (4000) the provider side emulating 100 ms sleep time and generating .5, 1, 1.5, 3, 6, 7, 8, 9 and 10 KB as reply answer.
- Calculating the Response Time (RT) and Propagation Time (PT) in ms.

Next we give a table, Table 4.1, to show the results for the first scenario:

TABLE 4.1: First Scenario, No Mediation

# of Load Drivers	# of invocations	Receiving Size	Sleep Time (ms)	Response Time (ms)	Propagation Time (ms)	Sent Requests	Answered Requests
1	4000	500	100	549	447	4000	4000
1	4000	1000	100	550	448	4000	4000
1	4000	1500	100	562	458	4000	4000
1	4000	3000	100	1258	1053	4000	4000
1	4000	6000	100	4182	3309	4000	4000
1	4000	7000	100	5536	4197	4000	4000
1	4000	8000	100	7131	5131	4000	4000
1	4000	9000	100	9030	6298	4000	4000
1	4000	10000	100	10813	7271	4000	4000

Next, Table 4.2, depicts the results for the second scenario (when the ESB writes to a log file).

TABLE 4.2: Second Scenario, With Mediation

# of Load Drivers	# of invocations	Receiving Size	Sleep Time (ms)	Response Time (ms)	Propagation Time (ms)	Sent Requests	Answered Requests
1	4000	500	100	13187	13078	4000	4000
1	4000	1000	100	14570	14464	4000	4000
1	4000	1500	100	17130	17018	4000	4000
1	4000	3000	100	20272	20128	4000	4000
1	4000	6000	100	28472	27896	4000	4000
1	4000	7000	100	32077	31172	4000	4000
1	4000	8000	100	46772	45403	4000	4000
1	4000	9000	100	52093	50038	4000	4000
1	4000	10000	100	71376	68300	4000	4000

Then the Figure 4.16 shows the difference between two different ways of direct Proxies, a comparison was made. The parameters configured for both of the scenarios were the size of the data generated at the provider side, and processing time was fixed to 100 ms.

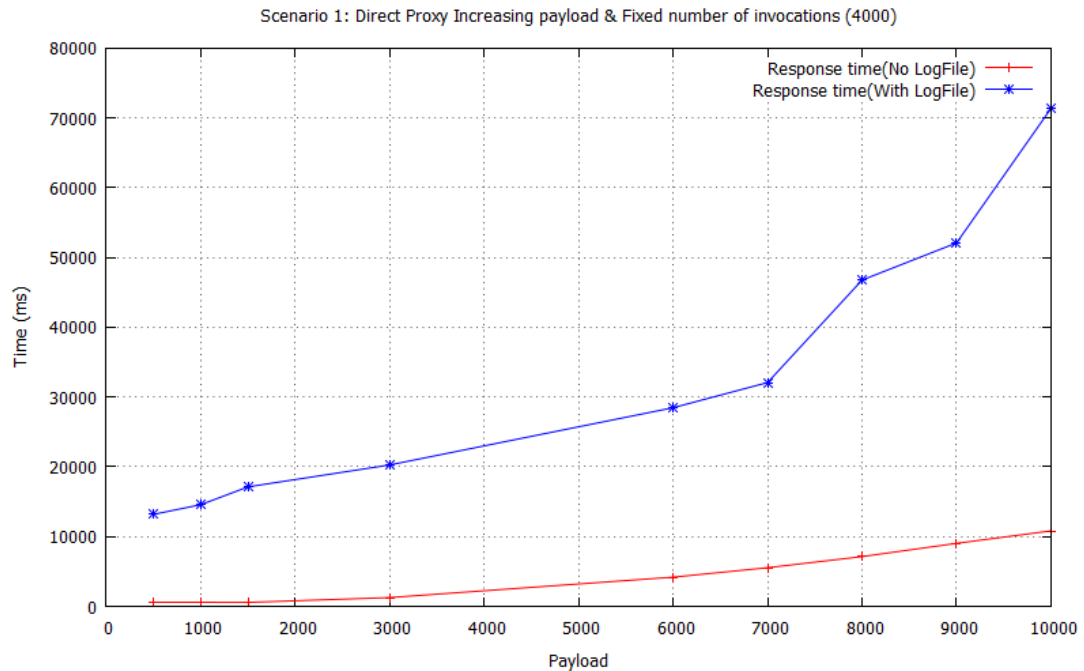


FIGURE 4.16: ESB with Mediation vs ESB without Mediation.

The Figure 4.16 shows the RT from both scenarios described before. Hence, it is unacceptable for real time and Web applications (UTI G.1010 Recommendation), having higher values of $RT > 4s$, adding a log file (some sort of mediation, but not totally mediation inside the ESB) to monitor the ESB; we could conclude it is unacceptable to add this kind of monitoring.

Now that we have taken into consideration the recommendations made by the section 4.2 "Avoiding Bottlenecks", and we know which values are optimal to overwhelm the ESB, we present the next section.

4.4 OUT OF HEAP MEMORY

In this section we address a problem found when many concurrent invocations (request) are sent to the ESB, the ESB tries to attend all incoming request but unfortunately it

runs out of heap memory. So the first thing we need to know is the behavior of the outgoing request (client side), the processing of the same request and how many can be treated before running out of memory at the ESB side. Finally we need to know how the requests are treated by the provider side.

The next figures illustrate the number of concurrent requests, of each involved actor of the system and how they perform.

The Figure 4.17 shows the number of concurrent requests from the point of view of the client. The aforementioned figure is obtained as the increment on a counter each time a request was sent from the client. Opposite to, when a request is replied by the provider the counter decreases. In the x axis we find the number of concurrent requests during the time-line. What this means is for example that at the instant 5000 we have computed 1000 concurrent requests.

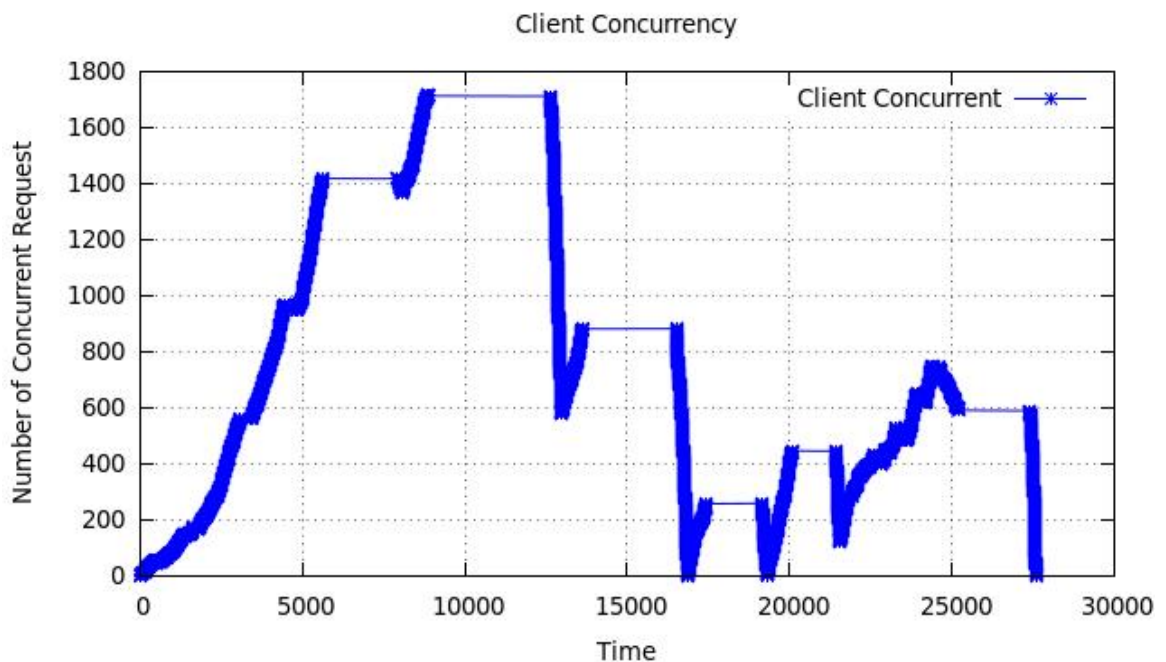


FIGURE 4.17: Consumer Concurrency.

The Figure 4.18 shows the number of concurrent request from the point of view of the ESB. For the ESB, we have to increase a counter each time it receives an incoming request or reply from the client or provider respectively. Again the x axis represents an instant during the scenarios time-line.

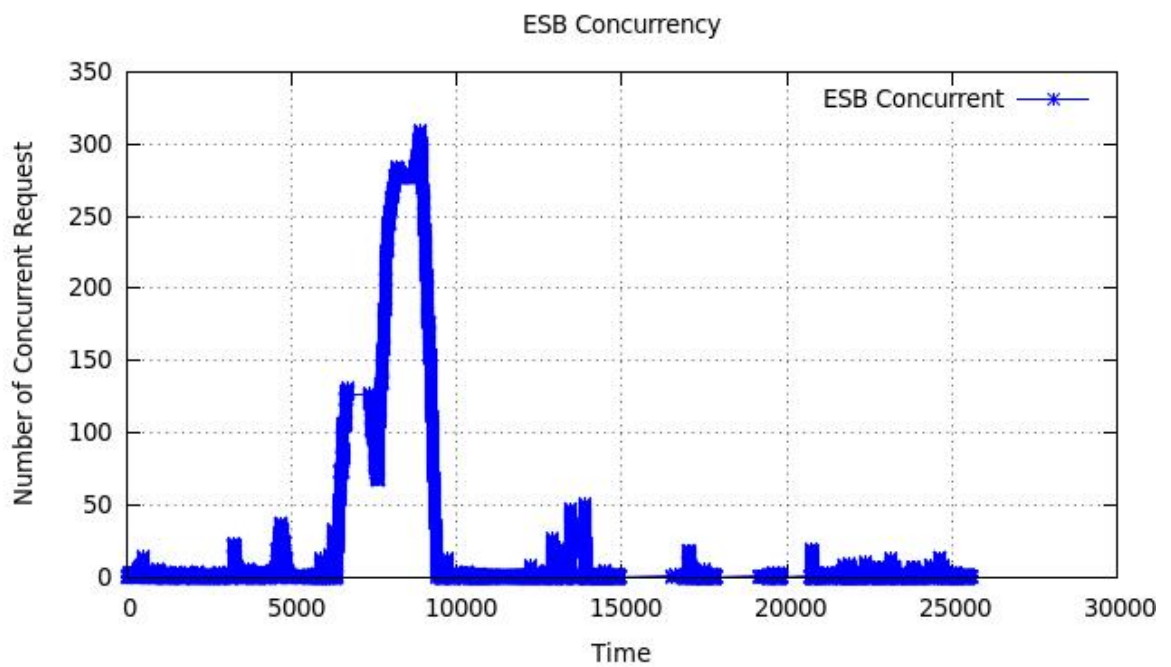


FIGURE 4.18: ESB Concurrency.

The Figure 4.19 shows the concurrency from the point of view of the provider.

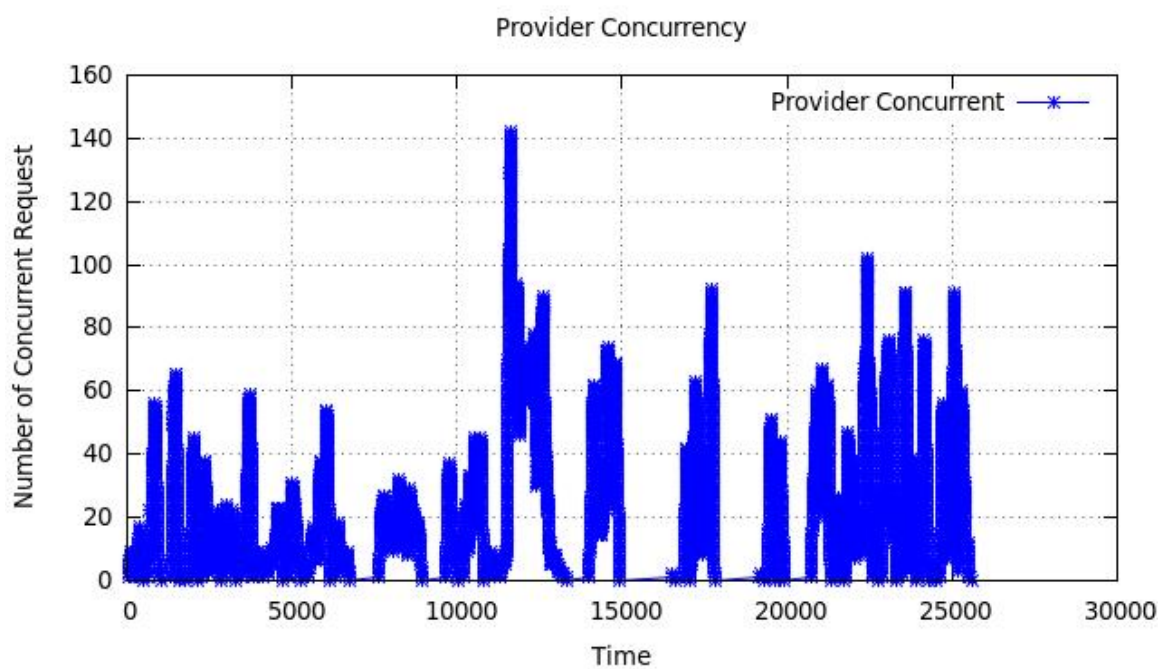


FIGURE 4.19: Provider Concurrency.

Finally the Figure 4.20 shows the concurrency from the point of view of the three actors involved.

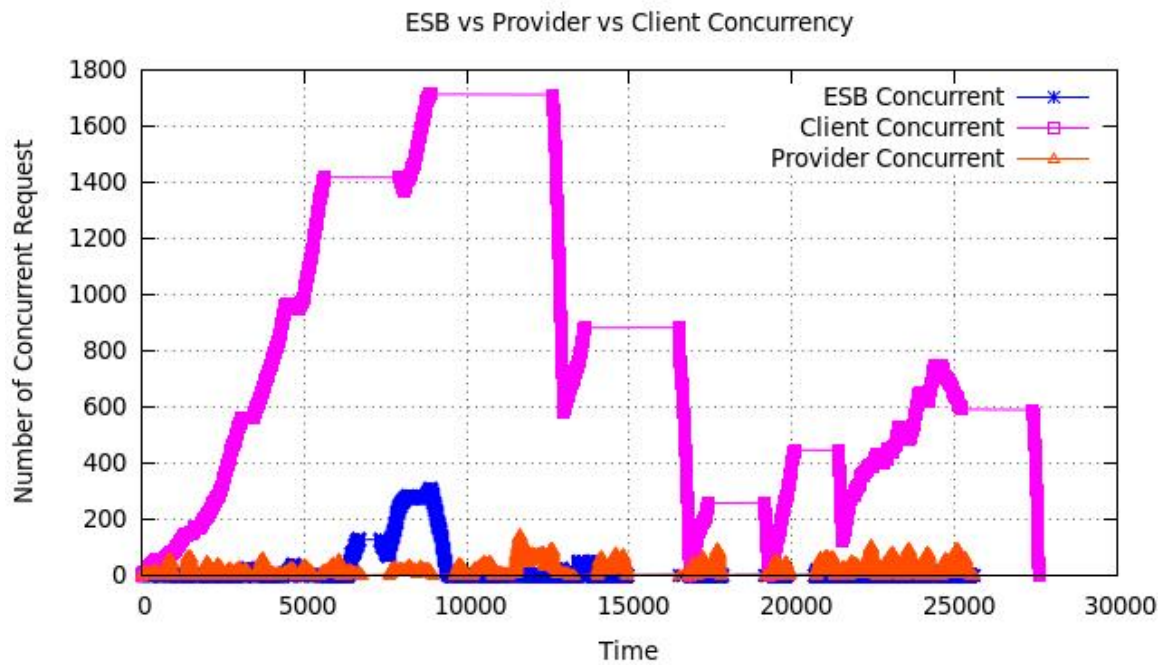


FIGURE 4.20: Consumer vs ESB vs Provider Concurrency.

It seems that the number of concurrent request has a strong correlation with the Heap Memory of the ESB, since we monitor the Heap Memory using our Emulator. We have run many test trying to detect an issue for the ESB. Running these scenarios we realized that the Heap Memory increases each time it handles a request. So if we have a great number of concurrent requests coming from a client the ESB is most likely to run out of heap memory, since it has to allocate heap memory for each one of the requests. The former graphs were helpful since we noticed that the ESB never allocates more than 400 threads to treat the incoming requests from consumers and providers. But what it does allocate is Heap Memory.

Chapter 5

Emulators Outputs

5.1 Obtained Metrics

In this Chapter 5, we present the Quality of Service (QoS) metrics obtained using our emulator, helping to evaluate the performance and efficiency of an ESB. Since, our system has a common database to which all LDs write to, we offer a table describing the information received with respect to each request a LD has properly processed. The monitoring instance is also writing to the database. Table 5.1 shows in detail the data gathered by the emulator, for each request sent and received answer. It shows the idLD (Load driver identifier), the idR (Request id), the timestamps and the CPU and HM (Heap Memory).

TABLE 5.1: Stored Information

idLD	idR	t1	t2	t3	t4	t5	t6	t7	t8	CPU	HM
------	-----	----	----	----	----	----	----	----	----	-----	----

As described in Figure 4.2 for each invocation the consumer takes the following timestamps and stores them in a common database:

- t1 request is sent by the requester to the ESB.
- t2 request is received by the ESB.
- t3 request is sent by the ESB to the provider.
- t4 request is received by the provider.
- t5 response is sent from the provider to the ESB.

- t6 response is received by the ESB.
- t7 response is sent from the ESB to the consumer.
- t8 response is received in the requester.

It is interesting to have this kind of tables, in specific the database, because for example if we want to plot a graph to have a better idea of the the response behavior (RT), given a certain time, all we need to compute is $RT = t8 - t1$. RT was taken as the amount of time passed since the moment the request was sent till the time a reply was received. Table 5.1 represents each time a LD sends a request and it gets its response back these info is stored inside the database. Another, example: for knowing the concurrence level (viewed at the client side) we applied the following rule, to the MySQL database:

$$A.t1 < B.t1 \ \& \ A.t8 > B.t8 \ \& \ A.t1 < B.t8$$

To compute the concurrence we created a temporary table called B, then we compare each request against another table A, checking for A containing B. Meaning the request were concurrent. All the information analysis is done in off-line mode.

5.1.1 Obtained Graphs

Using our proposed approach, is easy to obtain the QoS of a deployed scenario. For example we configured the following scenario:

- One LD 4000 concurrent request.
- One Web service provider: emulating 100 ms delay and generating 1 KB as reply answer.
- The ESB was configured as a Direct Proxy

The above scenario was created because or methodology, the previous chapters, give us the optimal values under which the ESB will work under high pressure circumstances; and the consumers and the providers will not be under high work demands.

The following graph, Figure 5.1 shows the RT computed for the 4000 concurrent invocations sent to the end service, and the ESB system acting as a direct proxy. Figure 5.1 also shows the average response time = 3775 ms. The number of request and the size of such affect in the behavior of the ESB.

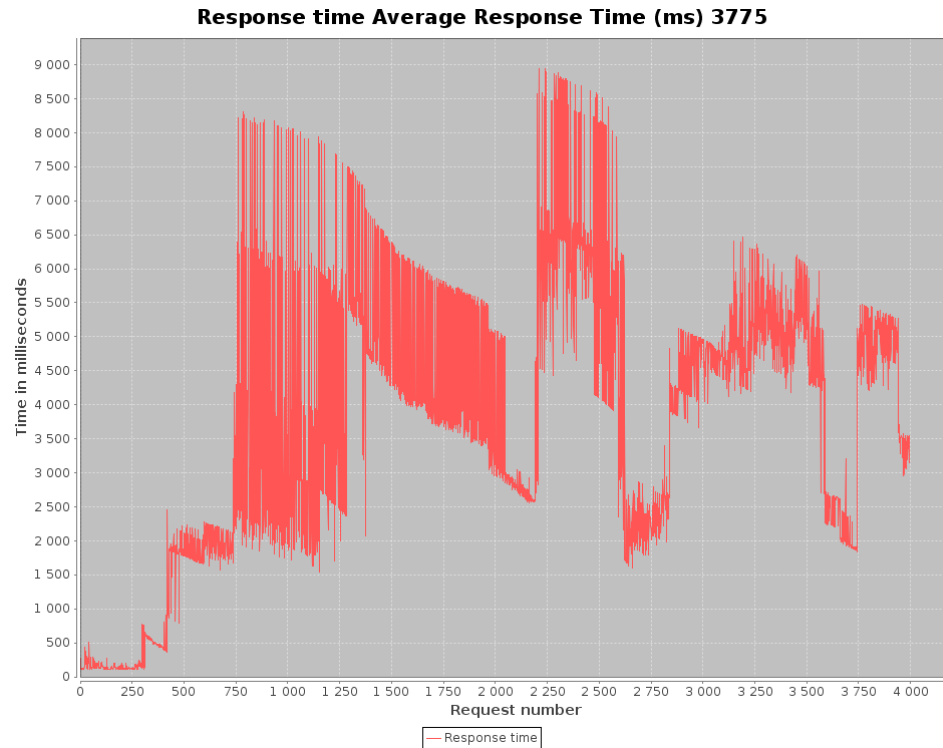


FIGURE 5.1: Computed Response Time.

A common database gives important advantages like being able to compute the concurrency level. Figure 5.2 shows the concurrency computed at the client side. The average concurrency = 760 requests; meaning that the ESB system queues request till it has space for a new task.

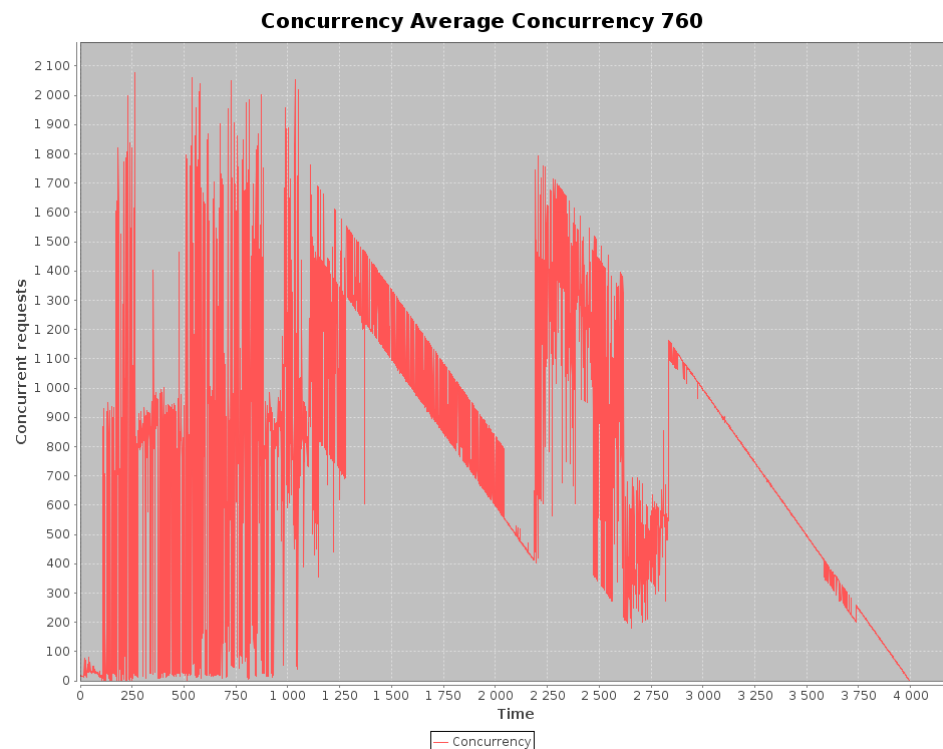


FIGURE 5.2: ESB Concurrency view by the Consumer.

Figure 5.3 illustrates the monitoring done remotely to the ESB, this graph presents the heap memory of the ESB. The average Heap Memory = 20% ; meaning that the ESB is working under normal circumstances for this scenario. As presented in figure 5.3 the Heap Memory is always increasing, after a drop at the beginning. This means that the Garbage Collector (GC), played its role at the beginning but not afterwards. It is important to mention that the GC depends on the configuration given to the Java Virtual Machine. During the execution of this scenario the ESB was configured as follows : $-Xms512M -Xmx2048M -XX:MaxPermSize = 1024M$. These values mean that the ESB's JVM has as minimum Heap Memory of 0.5 GB and that it could use at most 2 GB and the JVM could create an object which uses less than 1 GB. These values also mean that the GC doesn't interfere very often.

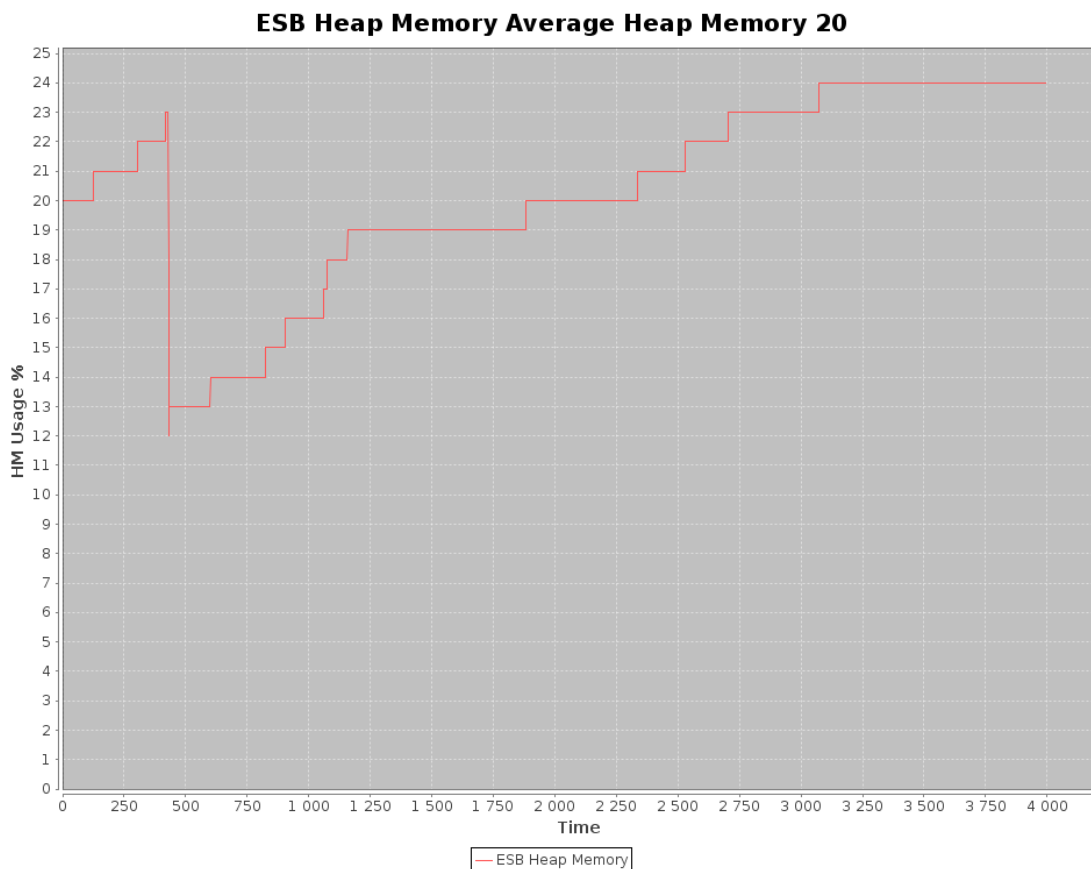


FIGURE 5.3: ESB Heap Memory view by the Consumer.

The concurrency level, the RT and the Heap Memory are tightly dependent on the ESB's CPU % value, it is indispensable to monitor it. Figure 5.4 presents a graph of the CPU % usage during the execution of the aforementioned scenario. The average CPU = 67%, meaning that the CPU is working at a medium percentage. Reason why the average RT is acceptable less than 4s recommended value by UTI G.1010 recommendation.

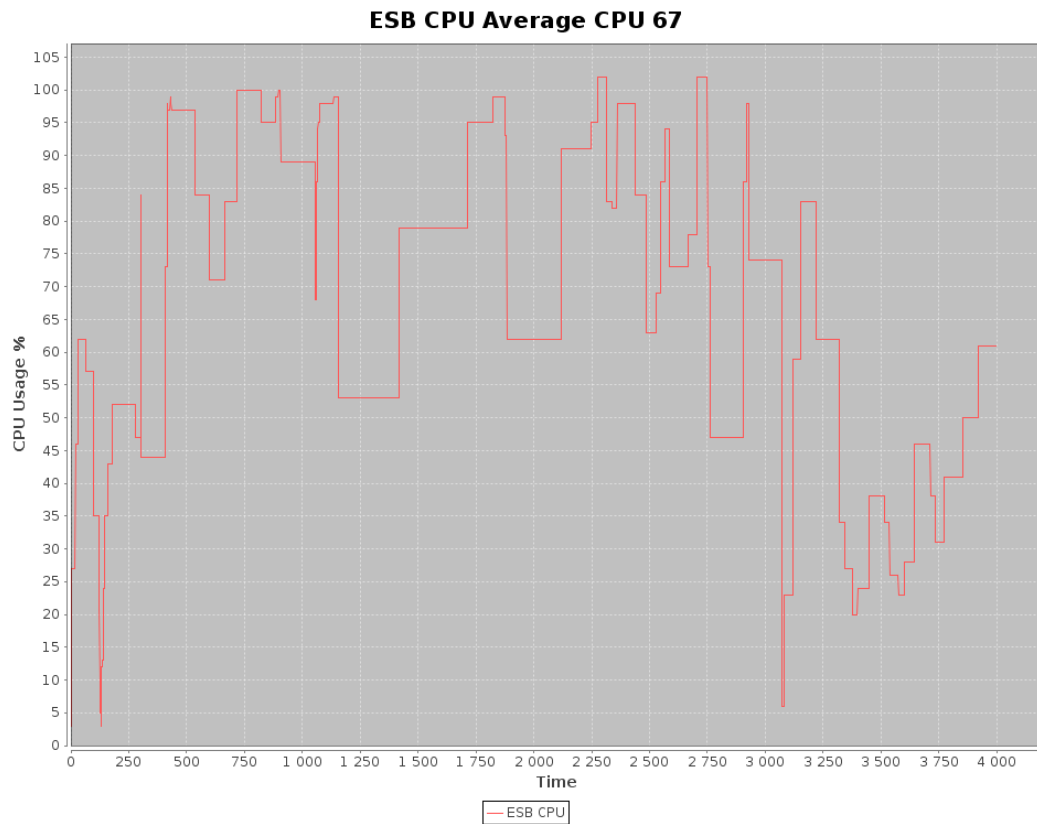


FIGURE 5.4: ESB CPU view by the Consumer.

To conclude this Chapter 5 we discuss the aforementioned scenario was for illustrating purpose. Summarizing, in order to have a high workload test scenario, we created more Load Drivers increasing the number of requests and therefore the workload in the ESB.

Chapter 6

Structure Learning

6.1 Motivation

From observations made learning techniques can predict effects applying the principles of probability. Some know tools establish dependency relationships between variables found in a data sample and build a probabilistic model, this is called structure learning. The dependency relationships between variables is established by statistical hypothesis testing, in specific, conditional independence tests.

The PC algorithm is one of the learning techniques to find the dependency relationships between variables in a model building a graphical model on which we can infer knowledge. Being able to represent the dependencies using a graphical model allows probabilistic application tools for Bayes networks, Markov networks or graphs Factor. The following section describes the PC algorithm.

We applied the PC algorithm in order to calculate a Bayesian network from a data sample obtained from a dataset of stress tests performed over a ESB platform. The sample contains 134,824 cases.

The resulting Bayesian Network (BN) is useful to make inferences about the current states in performance parameters of the platform when we only have some evidences, for example, to diagnose a problem and to determine its cause and other effects. From another perspective the BN could help to define setting parameters in order to get a good performance level.

In this particular section we want to evaluate the KPI (Key Performance Indicators) of an ESB, namely CPU load, Memory load, Concurrency level, and the Response Time (RT) for each request. The Concurrency level measures the amount of request that can

be simultaneously mediated by the ESB, while the RT is the time elapsed between a request sending and response receiving.

In the table 6.1 we defined six scenarios in which we change different parameters in order to consider most liked real conditions. The response message size corresponds to the size in bytes of the response messages. The processing time is the time the provider requires answering to one invocation.

Each LD can create up to 4000 simultaneous requests. For each scenario we have different number of LD, but the same response message size (100 B) and processing time (100 ms).

The table 6.1 resumes in detail each one of the tested scenarios.

TABLE 6.1: Tested Scenarios

Scenario	# of Load Drivers	# of Request	Response Size (ms)	Processing Time (ms)
1	1	4000	1000 B	100
2	2	8000	1000 B	100
3	3	12000	1000 B	100
4	4	16000	1000 B	100
5	6	24000	1000 B	100
6	8	32000	1000 B	100

6.2 Data sampling

The data set consists of five variables, related to interesting aspects in the ESB platform, namely, the percentage used of CPU and Memory, the Response time, and the concurrence level observed in each request mediated by the ESB; as part of each test, the number of request is also included.

These aspects and the measures, used are defined in the next tables. We also establish intervals to define states for each variable in each case.

1. CPU

Measure unit: percentage used

TABLE 6.2: CPU

Name	Tag	Interval	Description
Low	L	$[0, 25)$	Correct level of function
Medium	M	$[25, 50)$	Correct level of function
High	H	$[50, 75)$	High CPU load
Very High	VH	$[75, 100]$	Critical CPU load

2. Memory Used (M)

TABLE 6.3: Memory Used

Name	Tag	Interval	Description
Low	L	$[0, 25)$	Correct level of function
Medium	M	$[25, 50)$	Correct level of function
High	H	$[50, 75)$	Memory Loaded
Very High	VH	$[75, 99]$	Critical Memory
ERROR	ER	100	System in nonfunctional condition (Heap Memory overloaded)

3. Concurrency

TABLE 6.4: Concurrency

Name	Tag	Interval	Description
Low	L	$[0, 339)$	Low number of consumers
Medium	M	$[339, 967)$	Medium number of consumers
High	H	$[967, 4219)$	High number of consumers
Very High	VH	$[4219, 9919]$	VH number of consumers

4. Number of Requests

TABLE 6.5: Number of Requests

Name	Tag	Interval	Description
Low	L	$[0, 10000)$	Low number of Requests
Medium	M	$[10000, 20000)$	Expected number of requests
High	H	$[20000, 30000)$	High number of requests
Very High	VH	$[30000, 40000]$	VH number of requests

5. Response Time

Following the Recommendation G.1010 of ITU, we define the states for the response time (RT) parameter.

Measure unit: Time

Dimension: seconds

TABLE 6.6: Response Time

Name	Tag	Interval
Recommendable	R	$[0, 2)$
Acceptable	Acc	$[2, 4)$
Unacceptable	Unacc	$[4, +INF)$
ERROR	ER	$[-INF, 0]$

6.3 PC Algorithm

The PC algorithm consist on building a causal graphical model based, where each node represents a variable, and the edges represent the causal relations. The first step is create an undirected complete graph, relating each variable with all other variables.

From the complete graph, independence tests are performed for determining whether the variables are conditionally independent given another subset of variables, in this way, edges that are not necessary to represent the conditions of independence are removed from the model. After, the remaining edges are oriented, forming a Directed Acyclic Graph (DAG) that represents the causal relations among the variables. From the data sample, the Conditional Probability tables (CPT) are calculated, getting a Bayesian Network.

Before we describe the algorithm it is necessary to define three terms, the first one $ADJ(A)$ this refers to the set of nodes adjacent to node A in the graph C . The second term is $I(X, Y \mid Z)$ concerning conditional independence between the variables X and Y given Z . And the third term is $SepSet(X, Y)$ denotes the set variables separates the variables X, Y .

1. Start with a complete undirected graph G'
2. $i = 0$
3. **Repeat**
 4. **For each** $A \in V$
 5. **For each** $B \in ADJ_A$
 6. Test wheter $\exists S \subseteq ADJ_A - \{B\}$ with $|S| = i$ and $I(A, B \mid S)$
 7. **If** this set exists
 8. Make $S_{AB} = \mathbf{S}$
 9. Remove $A - B$ link from G'
 10. $i = i + 1$
11. **Until** $|ADJ_A| \leq i, \forall A$

6.4 Graphical Model

Here we applied structured learning to a set of data known as data set. This data set consists of data monitored from the ESB and Java consumer. After running the PC algorithm we obtained the next Bayesian network in its graphical representation. The tool used to obtain the Bayesian network was Tetrad-4.3.10-6.

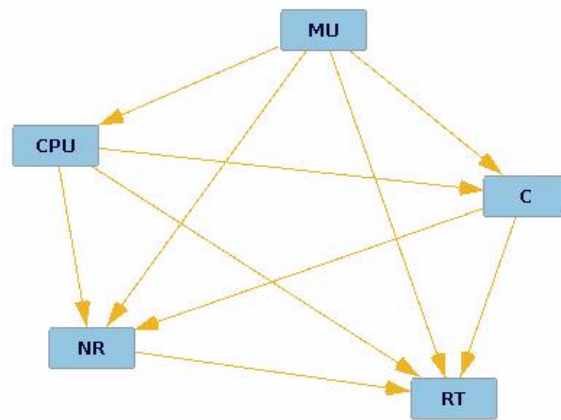


FIGURE 6.1: Bayesian Network.

Figure 6.1 shows the strong dependency of each of the variables monitored and measured. What this means is that all variables depend one from another.

The next table 6.7 shows the marginal probabilities obtained for each of the values we are interested on knowing their dependency.

TABLE 6.7: Marginal Probabilities

Memory Used (M)	Concurrency (C)	# of Request (NR)	CPU	Response Time (RT)
$P(\text{MU} = \text{Er}) = 0.6366$	$P(\text{C} = \text{H}) = 0.1763$	$P(\text{NR} = \text{H}) = 0.1141$	$P(\text{CPU} = \text{H}) = 0.4005$	$P(\text{RT} = \text{Acc}) = 0.0467$
$\text{H} = 0.2373$	$P(\text{C} = \text{L}) = 0.2997$	$P(\text{NR} = \text{L}) = 0.0809$	$P(\text{CPU} = \text{M}) = 0.1780$	$P(\text{RT} = \text{Er}) = 0.0364$
$\text{M} = 0.1261$	$P(\text{C} = \text{M}) = 0.1475$	$P(\text{NR} = \text{M}) = 0.1155$	$P(\text{CPU} = \text{VH}) = 0.4215$	$P(\text{RT} = \text{R}) = 0.0580$
	$P(\text{C} = \text{VH}) = 0.1306$	$P(\text{NR} = \text{VH}) = 0.2434$		$P(\text{RT} = \text{Unacc}) = 0.0922$

6.5 Analysis

Now, since one of the main goal of this study is to know the dependency among aspects in an ESB, we have to be able to tell when an ESB may fail or which are the optimal conditions under which an ESB may have its best performance. A way to find how each variable is influenced by another is applying the conditional probability.

So, for example we have a high number of concurrent request passing through an ESB ($P(C = H) = 1.0$), this event leads us have the next table of posterior conditional probabilities, where the most probable states of other variables is in bold type, shown in table 6.8:

TABLE 6.8: High Concurrency ($C=H$)

MU	NR	CPU	RT
$P(\mathbf{MU} = \mathbf{Er} \mid \mathbf{C} = \mathbf{H}) = \mathbf{0.7392}$	$P(\text{NR} = \text{H} \mid \text{C} = \text{H}) = 0.1653$	$P(\text{CPU} = \text{H} \mid \text{C} = \text{H}) = 0.3671$	$P(\text{RT} = \text{Acc} \mid \text{C} = \text{H}) = 0.0524$
$P(\text{MU} = \text{H} \mid \text{C} = \text{H}) = 0.1364$	$P(\text{NR} = \text{L} \mid \text{C} = \text{H}) = 0.1105$	$P(\text{CPU} = \text{M} \mid \text{C} = \text{H}) = 0.2633$	$P(\text{RT} = \text{Er} \mid \text{C} = \text{H}) = 0.000$
$P(\text{MU} = \text{M} \mid \text{C} = \text{H}) = 0.1245$	$P(\text{NR} = \text{M} \mid \text{C} = \text{H}) = 0.2633$	$\mathbf{P(\text{CPU} = \mathbf{VH} \mid \text{C} = \text{H}) = \mathbf{0.3696}}$	$P(\text{RT} = \text{R} \mid \text{C} = \text{H}) = 0.000$
	$\mathbf{P(\text{NR} = \mathbf{VH} \mid \text{C} = \text{H}) = \mathbf{0.2829}}$		$\mathbf{P(\text{RT} = \text{Unacc} \mid \text{C} = \text{H}) = \mathbf{0.4638}}$

Now lets say we have for example an application that needs or requires a recommended response time ($P(\text{RT} = \text{R}) = 1.0$). The next table 6.9 shows how this condition affects other variables:

TABLE 6.9: Recommended RT

RT	MU	NR	CPU	C
$\text{Acc} = 1.0$	$\text{Er} = 0.6454$ $\text{H} = 0.1242$ $\text{M} = 0.2304$	$\text{H} = 0.0459$ $\text{L} = 0.2502$ $\text{M} = 0.2355$ $\text{VH} = 0.1236$	$\text{H} = 0.3671$ $\text{M} = 0.2633$ $\text{VH} = 0.3696$	$\text{H} = 0.000$ $\text{L} = 0.5690$ $\text{M} = 0.1843$ $\text{VH} = 0.000$

Table 6.9 tells us that there is a need to add resources to the current system holding the ESB, why? Because, in order to have recommended Response Time (RT) we have both probabilities Very High (VH) and High (H) of 36% of having a CPU usage, which tell us that the CPU might be overloaded most of the time.

Now lets say we want to know the behavior of the ESB when there is a whole bunch of requests to a server. So, we set the event $\text{NR} = \text{VH} = \text{true}$ as shown in table :

TABLE 6.10: Very High NRs

NR	MU	C	CPU	RT
VH = 1.0	Er = 0.6047 H = 0.2823 M = 0.1129	H = 0.10 L = 0.3382 M = 0.0329 VH = 0.1953	H = 0.4235 M = 0.1129 VH = 0.4635	Acc = 0.0377 Er = 0.1808 R = 0.0170 Unacc = 0.1723

We have shown some examples of many possible combinations from table 6.7 to table 6.10, so each one of the tables presented here represents a possible combination the aim of this study is not to present all possible combinations but to try to land our work on a possible real case. For example we are interested in the case when many request are made at the same time, we are also interested on knowing what is the average concurrent request an ESB may attend.

In order to have a more realistic study we divided the previous data set into three parts and then randomly choose a new data sample. Table 6.11 shows the new marginal probabilities.

TABLE 6.11: New Marginal Probabilities

Memory Used (M)	Concurrency (C)	# of Request (NR)	CPU	Response Time (RT)
P(MU = Er) = 0.6392 H = 0.2354 M = 0.1254	P(C = H) = 0.1746 P(C = L) = 0.2997 P(C = M) = 0.1495 P(C = VH) = 0.1316	P(NR = H) = 0.1149 P(NR = L) = 0.0819 P(NR = M) = 0.1165 P(NR = VH) = 0.2424	P(CPU = H) = 0.3992 P(CPU = M) = 0.1772 P(CPU = VH) = 0.4236	P(RT = Acc) = 0.0464 P(RT = Er) = 0.0362 P(RT = R) = 0.0588 P(RT = Unacc) = 0.0926

Now the same example as the one from table 6.8, we are interested to know what happens when we have a high number of concurrent request passing through an ESB, this event leads us have the next table 6.12:

TABLE 6.12: New High Concurrency (C=H)

C	MU	NR	CPU	RT
H = 1.0	Er = 0.7361 H = 0.1393 M = 0.1246	H = 0.1668 L = 0.1082 M = 0.2626 VH = 0.2824	H = 0.3708 M = 0.2592 VH = 0.3700	Acc = 0.0533 Er = 0.000 R = 0.000 Unacc = 0.4609

The next table 6.13 shows for example an application that needs a recommended response and shows how this condition affects other variables:

TABLE 6.13: New Recommended RT

NR	MU	C	CPU	RT
Acc = 1.0	Er = 0.6487 H = 0.1264 M = 0.2248	H = 0.1335 L = 0.1599 M = 0.2287 VH = 0.0487	H = 0.2894 M = 0.3149 VH = 0.3957	H = 0.000 L = 0.5684 M = 0.1865 VH = 0.000

Now lets say we want to know the behavior of the ESB when there is a whole bunch of requests to a server. So, we set the event NR = VH = true, as shown in table 6.14:

TABLE 6.14: New Very High NRs

NR	MU	C	CPU	RT
VH = 1.0	Er = 0.6066 H = 0.2804 M = 0.1129	H = 0.1030 L = 0.3374 M = 0.0304 VH = 0.1982	H = 0.4211 M = 0.1129 VH = 0.4659	Acc = 0.0378 Er = 0.1799 R = 0.0164 Unacc = 0.1742

6.6 Conclusions

First thing we could say is that our model is correct due to the fact that we made an analysis of a complete data set, meaning that we didn't chunk no data. After we divided our original data set into three subsets of data and randomly analyzed and got the same results, almost all probabilities are the same. These kind of analysis are really interesting, because they allow us to predict, given a specific parameter to specify the conditions a system will probably have.

Our analysis shows logical facts, but allows us to have a greater point of view of what happens when some conditions become true, or given a certain condition.

Chapter 7

Conclusions and Perspectives

7.1 Conclusions

In the context of distributed systems and applications, an important role must be played by the component that "*glues*" applications and services together; enterprises based on these services could implement complex business process. Hence an ESB plays the aforementioned role, there is indeed a lot of studies around it, trying to figure out if it has scalability problems. The ESB is the trend mostly used, in charge of system integration, request routing, transformation and adaptation of data, it brings with it many challenges; like knowing its limits and constraints. In general business-to-business usually have complex integration systems around an ESB, therefore it is indispensable to have a tool that can deploy and evaluate an ESB at low cost, before deploying it in their actual business. In this work we focused our effort on a proposed tool that eases the assessment of an ESB with the main goal centered on high load circumstances alike in a real world production. With our approach we could deploy different business to business complex network topologies using small hardware resources, compared to a real world application, but with the same characteristics as the former. We have deployed scenarios with and without mediation (adaptation of data). We have shown useful figures obtain straightforwardly, with them is easy to characterize the QoS of a given scenario. Therefore, using our approach it is easy to expose the maximum capacity of an ESB.

Furthermore we found a reliability problem when we sent 40,000 concurrent requests (using 10 LDs), as mentioned before this ESB (WSO2) allocates an amount of Heap Memory to each request it receives, having an out of Heap Memory issue.

We have also shown, an analysis based on a probabilistic approach obtaining a Bayesian Network, which gives the dependency relationship between variables. Furthermore we made inferences to a variable, and analyzed effects among other variables.

Future work must be done, in order to modify the way to compute the ESB's concurrency level from the client point of view, since it takes a whole lot of time to be computed at MySQL database. A Graphical User Interface (GUI) is also another perspective for the near future.

Bibliography

- [AP11] Sanjay P Ahuja and Amit Patel. Enterprise service bus: A performance evaluation. *Communications and Network*, 3(3):133–140, 2011.
- [Bre09] Paul Brebner. Service-oriented performance modeling the mule enterprise service bus (esb) loan broker application. In *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, pages 404–411. IEEE, 2009.
- [Bru] Durga Pavani Brundavanam. Enterprise service bus evaluation as integration platform for ocean observatories.
- [Cha09] David A Chappell. *Enterprise service bus*. O'Reilly Media, Inc., 2009.
- [Exp13] Ernesto Exposito. *Advanced Transport Protocols: Designing the Next Generation*. John Wiley & Sons, 2013.
- [GJMCGS10] FJ García-Jiménez, MA Martínez-Carreras, and AF Gómez-Skarmeta. Evaluating open source enterprise service bus. In *e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on*, pages 284–291. IEEE, 2010.
- [Ort07] Sixto Ortiz. Getting on board the enterprise service bus. *Computer*, 40(4):15–17, 2007.
- [SCDR] Mauro Storch, Rodrigo N Calheiros, and César AF De Rose. Virtual machines networking for distributed systems emulation.
- [SJA] Themba Shezi, Edgar Jembere, and Mathew Adigun. Performance evaluation of enterprise service buses towards support of service orchestration.
- [TVS02] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*, volume 2. Prentice Hall, 2002.
- [UT06] Ken Ueno and Michiaki Tatsubori. Early capacity testing of an enterprise service bus. In *Web Services, 2006. ICWS'06. International Conference on*, pages 709–716. IEEE, 2006.

- [VAMD09] M Hadi Valipour, Bavar Amirzafari, Kh Niki Maleki, and Negin Daneshpour. A brief survey of software architecture concepts and service oriented architecture. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 34–38. IEEE, 2009.

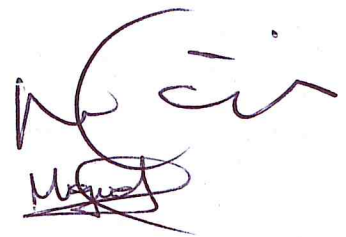
**Quality of Service Management for Enterprise
Service Bus (ESB)**

Tesis que presenta
Mariano Vargas Santiago
Para obtener el grado de
**Maestro en Ciencias
y Tecnologías de la Información**

Asesor: Dr. Luis Martín Rojas Cárdenas, UAM-I.

Jurado Calificador:

Presidente: Dr. Luis Martín Rojas Cárdenas
Secretario: Miguel Alfonso Castro García
Vocal: Ernesto Expósito

Handwritten signatures in purple ink. The top signature is large and stylized, likely belonging to the President, Dr. Luis Martín Rojas Cárdenas. Below it is a smaller signature, likely belonging to the Secretary, Miguel Alfonso Castro García. A third signature is partially visible at the bottom, likely belonging to the Vocal, Ernesto Expósito.