Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 3545

Enabling Horizontal Scalability in an open source Enterprise Service Bus

Arun Kumar Hanumantharayappa

Course of Study:	Infotech
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Santiago Gómez Sáez
Begin date:	August 1, 2013
End date:	January 31, 2013
CR-Classification:	C.2.4, D.2.11, H.3.3, H.3.4

Abstract

Cloud computing is a recent paradigm which describes a new way of consuming and delivering IT Services. In the Platform as a Service (PaaS) model, an underlying infrastructure such as network, operative system or server is provided to the Cloud consumers for either deploying their own applications, or applications supplied by the Cloud provider. In effect, Cloud computing modifies how applications should be built, provided, and consumed, as they may provide or be totally exposed as services, or consume existing third party applications services. The main advantages in Cloud computing are related to dynamic scaling of resources which are able to adapt to changes based on demand of resources and the use of multi-tenancy techniques in order based on sharing of resources between different users towards achieving the economy of scale.

Enterprise Service Bus (ESB) is essential as an integration middleware between application and services within and between multiple Cloud infrastructures. Different communication protocols might be used by application services and it is therefore necessary to have a mediator between them. Several challenges might arise when using an ESB as communication mediator between applications in cloud when to scale in and scale out to optimize resource consumption. The number of ESB instances should vary depending on the load in the Cloud infrastructure. This can be achieved by dynamically scaling in and out multiple ESB instances which constitute the horizontal ESB cluster.

In this Master Thesis we focus on enabling horizontal scalability support for an open source multi-tenant aware Enterprise Service Bus (ESB). The investigation is based on two possible scenarios for enabling horizontal scalability: interconnected vs. non interconnected ESB instances. Therefore, in this work we investigate their advantages, disadvantages, and possible challenges and solutions. Based on previous investigations, a realization approach for enabling multi-instance management of a multi-tenant aware ESB is provided.

Contents

List of Figures

List of Listings

List of Tables

1. Introduction

Nowadays markets are changing faster and businesses are constantly required to adapt their processes in order to decrease their response to such changes. Cloud computing has been in the last years an emergent paradigm which delivers computation, storage, and application hosting services on a Service Level Agreement (SLA) based model for ensuring performance, uptime, etc. Such services also follow the "utility" pricing model where customers are charged based on their utilization of computational resources, storage, and transfer of data [BBG11].

Enterprise Service Bus (ESB) is used as an integration middleware to integrate applications benefiting from the principles of Service-oriented Architecture (SOA). Multi-tenancy is sharing of whole technological stack by different consumers at same time [ESBMt], to fulfill the characteristics of cloud computing the ESB must be multi-tenant aware and scalable. Depending on the load on application the number of ESB instances should be varied. Let's consider an online web shop hosting the services on Cloud infrastructure with ESB as an integration middleware. The website traffic can fluctuate a lot and during day times there will be a spike in traffic several times and less traffic during nights. The system should be able to grow and shrink number of resources depending on the load. Elastic load balancers can be used to distribute the traffic across the resources.

Furthermore, the utilization of virtualization and multi-tenancy techniques in the Cloud infrastructures has maximized the resource utilization and has increased even more the necessity of not only being able to handle the load at a given time, but also to optimize the scaling techniques towards maximizing the resource consumption. This can be achieved by enabling horizontal scalability support of the ESB which is the main goal of this thesis.

1.1. Scope of Work

This master's thesis has the goal to specify, design and implement to enable support for horizontal scalability of an open source ESB [ESBMt]. The system must simultaneously support multiple instances of ESB retaining clustering to ensure elasticity. Research includes analysis of existing approaches for enabling scalability from the previous works [Muh11].

Possible scenarios are clusters of equal Apache ServiceMix instances or clusters of different interconnected Apache ServiceMix instances and evaluating different service endpoint deployment configurations. Based on evaluation results a prototypical implementation for one of the scalable architecture is developed.

1.2. Outline

The remainder of this document is structured as follows:

Chapter 2 – Fundamentals: Provides the fundamental background knowledge necessary to understand the concepts and principles related to this thesis.

Chapter 3 – Related Works: Presents previous and current research approaches and available technologies in different computing domains to enable horizontal scalability.

Chapter 4 – Concept and Specification: We formalize the functional and non-functional requirements from the lessons learned in the previous chapters for scalability of the multi-tenant ESB.

Chapter 5 – Design: Gives a detailed overview of the architecture as well as specifics of the components which satisfy the requirements.

Chapter 6 – Implementation and Validation: Describes the implementation details as well as challenges encountered during coding and configuration.

Chapter 7 – Conclusion and Future Work: The last chapter summarizes the work done in this thesis and suggests further possible extensions as well as research topics related to horizontal scalability.

1.3. List of Abbreviations

The following list contains abbreviations used in this document.

API	Application Programming Interface
BC	Binding Component
BPEL	Web Services Business Process Execution Language 2.0
DSL	Domain Specific Language
EBS	Elastic Block Storage
EC2	Elastic Compute Cloud
ESB	Enterprise Service Bus
JBI	Java Business Integration
LB	Load Balancer
MOM	Message Oriented Middleware
NMR	Normalized Message Router

OSGi	Open Services Gateway initiative
PaaS	Platform-as-a-Service
POJO	Plain Old Java Object
RDS	Relational Database Service
SaaS	Software-as-a-Service
SE	Service Engine
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SQS	Simple Queue Service
SU	Service Unit
UUID	Universally Unique Identifier
URI	Uniform Resource Identifier
VIP	Virtual Internet Protocol
XSLT	Extensible Stylesheet Language Transformation

2. Fundamentals

This chapter provides the fundamentals this thesis is based upon. In particular the focus is on the Cloud computing and Service Oriented Architecture (SOA) and ESB. Later on some concepts related to Nginx proxy server and some technologies related to ESB and multi-instance management are introduced.

2.1. Cloud Computing

Cloud computing refers to the delivery of computing resources over the Internet. Instead of keeping data on your own hard drive or updating applications for your needs, you use a service over the Internet, at another location, to store your information or use its applications. The services themselves have been referred to as Software-as-a-Service (SaaS) [Gcca].

Cloud computing is the delivery of computing services over the Internet. Cloud services allow individuals and businesses to use software and hardware that are managed by third parties at remote locations [Gcca]. Examples of Cloud services include online file storage, social networking sites, webmail, and online business applications. The Cloud computing model allows access to information and computer resources from anywhere that a network connection is available. Cloud computing provides a shared pool of resources, including data storage space, networks, computer processing power, and specialized corporate and user applications.

The data center hardware and software is what we will call a Cloud. When a Cloud is made available in a pay-as-you-go manner to the general public, we call it a public Cloud. The service being delivered as metered service is utility computing. We use the term private Cloud to refer to internal data centers of a business or other organization, not made available to the general public, when they are large enough to benefit from the advantages of Cloud computing that we discuss here. Thus, Cloud computing is the sum of SaaS and utility computing [GCC], but does not include small or medium sized data centers, even if these rely on virtualization for management. People can be users or providers of SaaS, or users or providers of utility computing. We focus on SaaS providers (Cloud users) and Cloud providers, which have received less attention than SaaS users. In some cases, the same actor can play multiple roles. For instance, a Cloud provider might also host its own customer-facing services on Cloud infrastructure.

2.2. Service-oriented architecture

A Service-oriented architecture (SOA) is the underlying structure supporting communications between services. SOA defines how two computing entities, such as programs, interact in such a way as to enable one entity to perform a unit of work on behalf of another entity. Service interactions are defined using a description language. Each interaction is self-contained and loosely coupled, so that each interaction is independent of any other interaction.

Simple Object Access Protocol (SOAP)-based Web services are becoming the most common implementation of SOA. However, there are non-Web services implementations of SOA that provide similar benefits. The protocol independence of SOA means that different consumers can communicate with the service in different ways. Flexibility and interoperability communicational aspects must be ensured by incorporating an intermediate layer capable of supporting multiple communication protocols and message formats.

Nowadays multiple enterprises rely on SOA-based realizations. For example, an online shop. Let's use Land's End as an example [SSOA]. We look at their catalog and choose a number of items. You specify your order through one service, which communicates with an inventory service to find out if the requested items are available in the sizes and colors wanted. Your order and shipping details are submitted to another service which calculates your total, tells you when your order should arrive and furnishes a tracking number that, through another service, will allow you to keep track of your order's status and location en route to your door. The entire process, from the initial order to its delivery is supported by the interactions among multiple Web services, each one performing a specific task of the order process, all made possible by the underlying framework that SOA provides.

SOAs are often visually described as a three-legged triangle in which there are three participants: the *Service Producer*, the *Service Consumer*, and the *Service Registry.*

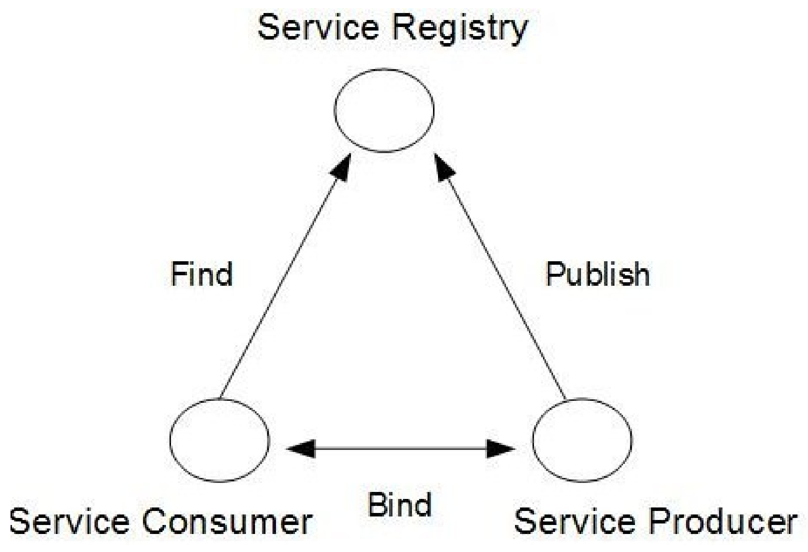


Figure 2.1: SOA Triangle [WSPA]

A Service is meant to be an independent entity that users can compose into business processes as they desire. In an SOA, it's the Service consumer, rather than the Service producer, that defines how an application behaves. Yet, loose coupling mandates that Service consumers and Service producers be independently created and controlled. In addition, we mentioned above that any given Service might have multiple contracts that define how a consumer can bind to it to gain required functionality. Finally, policies are increasingly controlling Services, where those policies define specific security, process, semantic, and governance constraints by which consumers can bind to specific Service producers [HTLC].

Given the highly variable nature of the Services being exposed, Service consumers search the registries in order to find and bind to the appropriate policy-controlled Service, and to insure that the composed application continues to operate in the face of continuous Service change. Dynamic discovery and the associated routing enable the association of Service requests with the appropriate Services at runtime regardless of any API-specific characteristics of the underlying components. As a result, it is clear to see that in a truly loosely-coupled SOA world, it is as impossible to operate without a registry as it is to operate in a global Internet without a DNS.

2.2.1. Enterprise Service Bus

Today there are many complex applications built on different platforms interacting with each other using point to point integration solutions. If we want to introduce a new application which wants to communicate with existing applications then we will have to develop new integration solutions. The complexity and maintenance also increases as we add new applications to the existing landscape of applications.

Businesses initially turned to manual integration and then enterprise application integration (EAI) but subsequently have focused on Service-oriented architectures. In a bus-like EAI architecture, disparate source applications send messages via a central "pipe" to receiving applications. The system contains software adapters and integration engines at all nodes, thereby distributing the intelligence. EAI enables automatic, machine-to-machine communications. However, as explained by Fulton, it works via point-to-point interfaces, which organizations must define and build one at a time, between applications [OJ07]. As companies use more applications to provide additional services, the amount of integration work required and managing the system becomes increasingly difficult.

To leverage SOAs' benefits effectively, companies are beginning to use an approach known as the enterprise service bus (ESB). ESB is the middleware glue that holds an SOA together and enables communication between Web-based enterprise applications. "Fundamentally," said Larry Fulton, senior analyst with Forrester Research, "ESB provides the connectivity between service requestors and service providers, physically conveying the requests and the responses." [OJ07].

Some of the Core functionalities provided by ESB are [TJ09]:

1) **Location transparency** The ESB decouples the service consumer from the service provider location. The ESB provides a central platform to communicate with any application necessary without coupling the message sender to the message receiver.

2) **Transport protocol conversion** An ESB should be able to seamlessly integrate applications with different transport protocols like HTTP(S) to JMS, FTP to a file batch, and SMTP to TCP.

3) **Message transformation** The ESB provides functionality to transform messages from one format to the other based on open standards like XSLT and Xpath.

4) **Message routing** Determining the ultimate destination of an incoming message is an important functionality of an ESB that is categorized as message routing.

5) **Message enhancement** An ESB should provide functionality to add missing information based on the data in the incoming message by using message enhancement.

6) **Security** Authentication, authorization, and encryption functionality should be provided by an ESB for securing incoming messages to prevent malicious use of the ESB as well as securing outgoing messages to satisfy the security requirements of the service provider.

7) **Monitoring and management** A monitoring and management environment is necessary to configure the ESB to be high-performing and reliable and also to monitor the runtime execution of the message flows in the ESB.

2.3. Technologies

In this section we present and introduce several technologies this thesis builds upon with respect to SOA and ESB.

2.3.1. Java Business Integration

The Java Business Integration (JBI) specification defines a standardized means for Assembled integration components are defined to create integration solutions [JBI01]. These components are plugged into a JBI environment and can provide or consume services through it in a loosely coupled way. The JBI environment then routes the exchanges between those components and offers a set of technical services.

JBI is built on top of state-of-the-art SOA standards. Service definitions are described in WSDL and components exchange XML messages in a document-oriented-model way. A JBI container provides facilities to plug in JBI-compliant components that interoperate through a central Normalized Message Router (NMR). A JBI component installed to a JBI container either as Binding Component (BC) or as Service Engine (SE). The former providing connectivity to external services, the latter implementing composition and transformation services. The NMR is a message-oriented mediator that ensures loose coupling between JBI components.

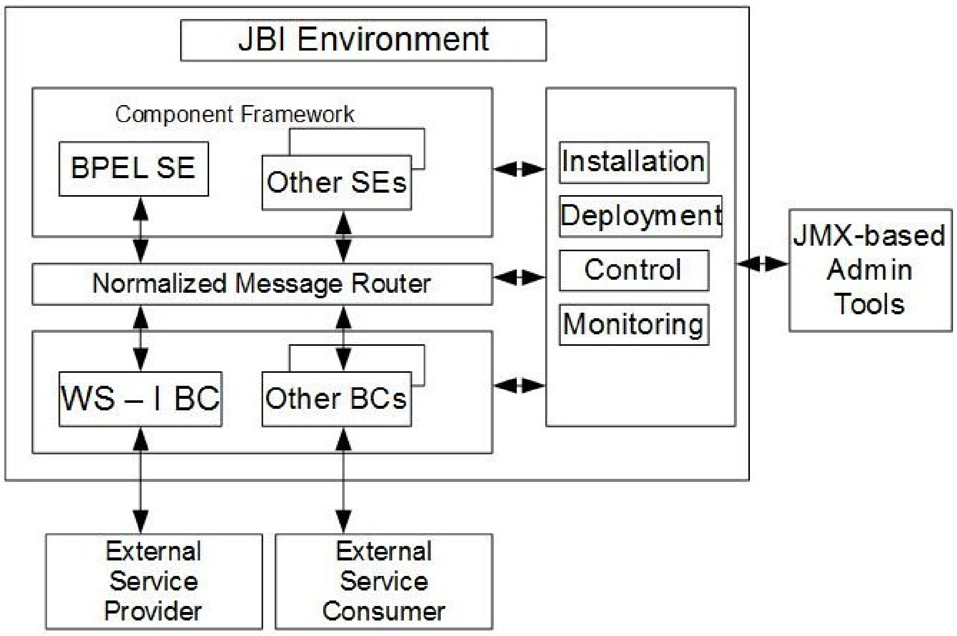


Figure 2.2: Top level view of JBI Architecture [JBI05]

The JBI specification defines four asynchronous message exchange patterns for communication between JBI components. They differ in being unidirectional or bidirectional and in being more or less reliable. Developers of JBI components use an API provided by the component framework to create and send message exchanges or to query service endpoints. Furthermore, the components life cycle is managed by the framework by providing management interfaces which must be implemented within the JBI component.

The Service Units (SUs) provide information about the services and endpoints to the binding components. The Service Assembly (SA) consist of a set of Service units packed in a zip file and the SA is deployed on top of BC. The SA contains a jbi.xml file that provides information about SUs and the target BCs. As we can see in Figure 2.3 there are 3 sets of SUs: XSLT style sheet, BPEL process and XML schema. The Extensible Stylesheet language Transformation (XSLT) style sheets are deployed on top of XSLT engine which provides transformation, the Business Process Execution Language (BPEL) is deployed on top of BPEL SE which provides orchestration for BPEL processes and the XML schema is deployed for validation of XML documents on top of XML SE.

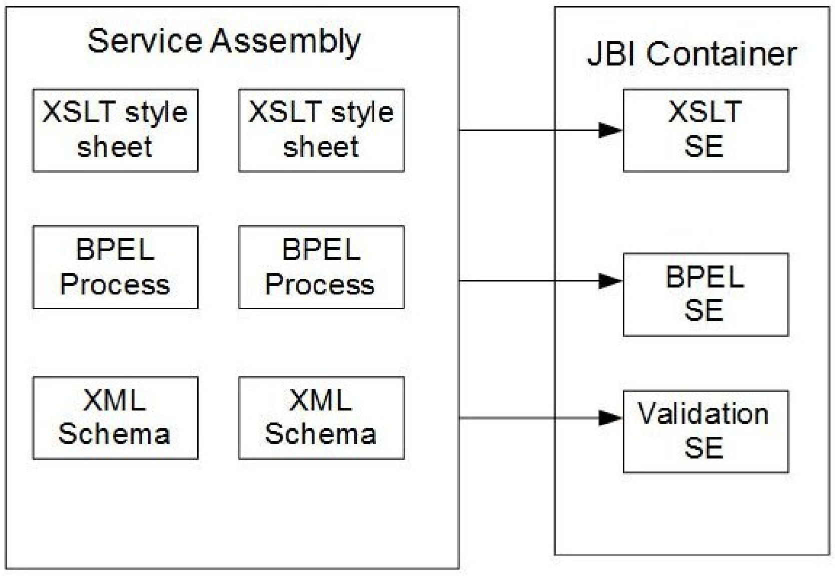


Figure 2.3 : Service Assemblies deployed on top of Service Engine inside JBI container [Cha04].

2.3.2. OSGi Framework

The Open Services Gateway Initiative (OSGi) defines an architecture for developing and deploying modular applications and libraries. The OSGi architecture allows applications to share a single Java VM. The OSGi platform is divided into following layers. Execution Environment provides a defined context for applications. The *Services layer* provides a collaboration model. The *Module layer* provides class loading and packaging specifications. The extensive Security layer is embedded in all layers [OSG09].

OSGi applications consist of executable and non-executable modules denoted as bundles. A bundle is packaged as JAR files and contains, in addition to Java class files, meta-data describing capabilities it provides and requirements it demands. An OSGi bundle is started by the container by executing the Bundle Activator interface. This interface must be realized by the OSGi bundle developer within the bundle package. The Bundle Activator gets a Bundle Context that provides access to the OSGi Framework functions The Framework provides the Start Level service to control the start/stop of groups of applications. The System bundle represents the OSGi framework and it provides an Application Program Interface (API) for managing the bundles. Once the OSGi bundle external dependencies are resolved, a bundle can be started by the framework. Therefore, executable bundles implement OSGi specific interfaces that allow the framework to start and stop the individual bundle. Furthermore, bundles can provide services to other bundles by dynamically registering service objects to the framework. Other bundles can then use an API to query the internal service registry for available service objects and finally bind to them [OSG11].

The Blueprint Container specification defines a dependency injection framework for OSGi. It is designed to deal with the dynamic nature of OSGi, where services can become available and unavailable at any time. The specification is also designed to work with plain old Java objects (POJOs) so that the same objects can be used within and outside the OSGi framework. The Blueprint XML files that define and describe the various components of an application are key to the Blueprint programming model.

The specification describes how the components get instantiated and wired together to form a running application.

The Blueprint Container specification uses an extender pattern, whereby an extender bundle monitors the state of bundles in the framework and performs actions on behalf of those bundles based on their state. The Blueprint extender bundle waits for the bundles to be activated and checks whether they are Blueprint bundles. A bundle is considered to be a Blueprint bundle when it contains one or more Blueprint XML files. These XML files are at a fixed location under the OSGI-INF/blueprint/ directory or are specified explicitly in the Bundle-BluePrint manifest header.

2.3.3. Apache ServiceMix

Apache ServiceMix is a flexible, open-source integration container that unifies the features and functionality of Apache ActiveMQ, Camel, CXF, ODE, Karaf into a powerful runtime platform that can be used to build your own integrations solutions. It provides a complete, enterprise ready ESB exclusively powered by OSGi [SMXa]. It is based on the OSGi Framework implementation Apache Karaf [APA11b] that builds the ServiceMix. Using OSGI framework brings a new important feature for SOA development i,e. modularity. That means that we can handle class loading and application lifecycle differently between the components.

The open source ESB we intend to enable horizontal scalability support is based on the multi-tenant aware ServiceMix ESB based on previous works[Muh11]. Apache ServiceMix 4.3.0 from the Apache Software Foundation, hereafter referred to as ServiceMix [SMXa]. On top of the kernel layer, OSGi bundles realize the technology layer of ServiceMix. The technology layer brings in the ESB functionality complying the JBI specification. Additionally, ServiceMix ships with various JBI components. BCs support diverse protocols, such as SOAP over HTTP, JMS, FTP, or SMTP. Each ServiceMix instance comes integrated with a message broker Apache Active MQ [AMQ] which is the foundation for NMR.

The admin console command allows the creation and management of instances of ServiceMix. However, the console does not only allow the management of OSGi bundles and services, but also allows the management of already installed JBI components and deployed service assemblies. Artifacts, such as OSGi bundles, JBI components, or service assemblies, can be installed by dropping them into a hot deployment directory. Although the JBI specification does not define a distributed deployment of JBI containers, ServiceMix implements a clustering engine. As a result, within a cluster each instance is aware of the service endpoints created on other instances. Furthermore, developers are provided with plug-ins for the software building tool Apache Maven [AMV], which simplifies the process of developing JBI components and service assemblies [FUS11].

2.4. Horizontal Scalability

For any network, system or process scalability is a desirable attribute. Poor scalability can result in degraded system performance with utilizing more resources than it is needed. They might be utilizing the resources repeatedly with poor scheduling algorithms either they cannot take complete advantage of parallelism or shared memory resources. In the below paragraphs we discuss about scalability approaches in multiple domains.

When there is an increasing and growing demand for a system scalability is crucial to achieve long term success. The ability to scale a system depends on the type of data structures used in the system also the underlying algorithms used to communicate with the components. The functions of the system are supported by the data structures and algorithms used to search these structures, co-ordinate and monitor the process between these data structures. Such kind of structures results in space or space-time scalability. Yet another kind of scalability we need to consider is the *Load Scalability* i.e., "ability to function gracefully, i.e., without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads while making good use of available resources" [Scal].

2.4.1. Measuring Scalability

There are different perceptions in how the scalability of a system can be measured. If we consider a transaction oriented system such as online banking one can measure scalability in terms of simultaneous number of users supported or maximum number of transactions supported per unit time.

In the Figure 2.4 load is plotted against X-axis and Throughput against Y-axis. The throughput initially increases as the load will be less. As we keep on increasing the load the throughput will remain constant for a certain amount of time and after that it starts to decrease gradually. This point is called as Threshold or knee point. If we further keep on increasing the load the throughput starts decreasing and we say at this point the system performance is degrading. This might be the result of unavailability or scarcity of resources.

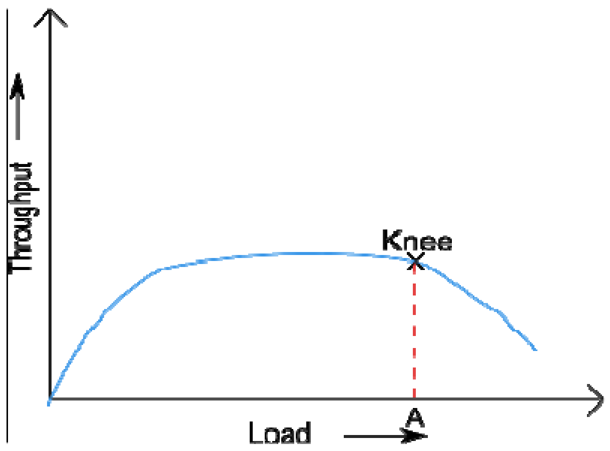


Figure 2.4 : Measuring Scalability [ScalDP]

Let us add some more hardware resources to the above system and in the below figure we can see that the new scalability has increased and so is the throughput. Also the threshold or knee point is drifted further away.

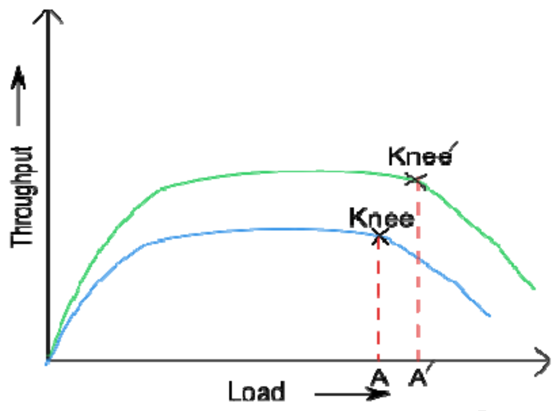


Figure 2.5 : Comparing Scalability [ScalDP]

However we cannot keep on adding the hardware indefinitely as there is a bottle neck for the system as well to handle maximum load. This is proved by Amdahl's law:

"If α is fraction of calculation essential and 1 - α is the fraction that cannot be parallelized then the maximum speedup that can be achieved by using P processors is given by [ScalDP] : "

$$\frac{1}{\alpha + (1 - \alpha)/P}$$

2.4.2. Scalability Patterns

There are many ways in which we can introduce scalability into a system and below we discuss some of the patterns used generally [ScalDP].

- Add Hardware – Here when the system reaches the Knee or Threshold point we identify additional and scarce resources and introduce them into the system.

- Introduce Parallelism – Here we identify the tasks which can be done parallel and split the task and assign it to different processes.

- Optimize Algorithm – Here we don't add hardware or introduce parallelism into the system. The key solution here is to identify areas for optimized performance with increased load. There are many algorithms available to do this which is described in section 2.5.1.

- Optimize Decentralization – Here we have introduced the parallelism but there may be resources in turn required by parallel processing paths which results in bottle neck. So we follow a decentralized approach where we don't concentrate on single resource but provide multiple resources to make the parallel paths independent.

- Control shared resources – Here we have introduced parallelism and shared resources but there might be some resources to be shared across parallel paths. So to overcome this problem we further categorize the shared resources into “Access only” and “Modifiable” resources.

- Intro-Process parallelism – Here a single process exploits parallelism and optimized usage of hardware resources to handle the increased load.

- Inter-Process parallelism – Here the system replicates its process across multiple instances. All these instances co-ordinate with each other to handle increased load in a distributed manner.

- Hybrid parallelism – Here the system has the capacity to replicate threads as well as processes.

2.5. Load Balancing

The distribution of the incoming traffic among different servers hosting the same application content then Load Balancing is a core networking solution. By balancing application requests across multiple servers, a load balancer prevents any application server from becoming a single point of failure, thus improving overall application availability and responsiveness. For example, when one application server becomes unavailable, the load balancer simply routes all new application requests to other available servers in the pool.

Load balancers also improve server utilization and maximize availability. Load balancing is the most straightforward method of scaling out an application server infrastructure. As application demand increases, new servers can be easily added to the resource pool and the load balancer will immediately begin sending traffic to the new server [LB]. There are several existing load balancing techniques to achieve scalability in an owned infrastructure. In the next section we will discuss how these techniques can be used for scalability in a Cloud infrastructure.

Figure 2.6 depicts two instances of Load Balancer LB1 and LB2. These share a virtual IP and divide the traffic between 1 to N Web servers. The purpose of this example is to create a high availability using load balancing.

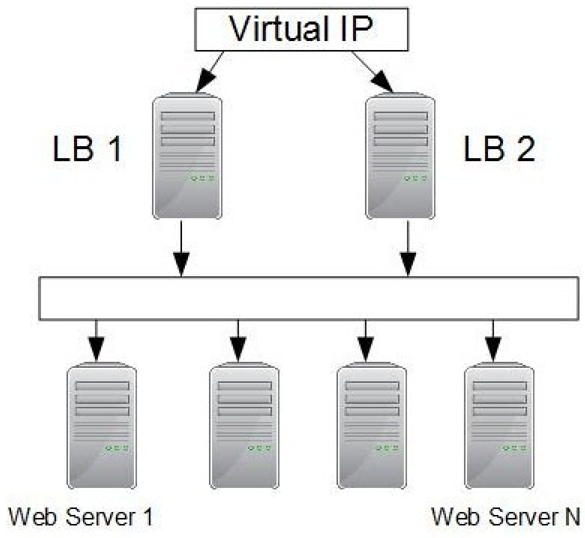


Figure 2.6 : Load Balancing with two Instances of LB [IBMSc]

2.5.1. Common Load Balancing Algorithms

Load balancing approaches can be realized at two different levels:

- Hardware load balancer
- Software load balancer

A regular way to scale web applications is by using hardware load balancer. The fundamental working rule is that network traffic is sent to a common IP in many cases called a virtual IP (VIP). This VIP is an address attached to the load balancer. Once the load balancer receives a request on this VIP it will need to make a decision on where to send it. This decision is usually made by a load balancing algorithm. The client request is then sent to the right server and the server will generate a response. Depending on the type of device, the response will be sent either back to the load balancer, in the case of a Layer 7 device, or more naturally with a layer 4 device straight back to the customer. The hardware load balancer is intended to handle high level of load, so it can simply scale. However, a hardware-based load balancer uses application specific hardware-based components, thus it is naturally expensive. Because of Cloud's commodity business model, a hardware load balancer is rarely offered by Cloud providers as a service. As an alternative, one has to use a software load balancer running on a generic server.

A software load balancer is not scalable solution. Since it is run on a generic server, the scalability is generally restricted by the CPU and network bandwidth capacity of the generic server. The generic server's capability is much smaller than that of hardware load balancer.

Some of the common load balancing algorithms are [ALB]:

- *Round Robin* - As the name suggests the servers are selected in a round robin fashion. This is a well-known and classic policy, which spreads the load evenly.

- *Random* - A random server is selected for each serving the request.
- *Sticky* - Sticky load balancing uses an expression to calculate a correlation key to perform the sticky load balancing.
- *Weighted Round* - The weighted Round-Robin load balancing policy allows us to specify a processing load distribution ratio for each server with respect to the others. In addition to the weight, endpoint selection is then further refined using round-robin distribution based on weight.
- *Least Connections* - The load balancer keeps count on each active connection and always routes to the server with the least.
- *Hashing* - Take a part of the incoming connection to create a Hash. This can be the target or source destination or the URL or parts of it.

2.6. Multi-tenancy

One of the main factors for utilizing the resources of a Cloud infrastructure for any organization is related to reducing operational costs. On the other side, Cloud providers aim to virtualize and share the same resources for many customers concurrently on their infrastructure towards maximizing the resource utilization and reducing their infrastructure expenses, as software vendors can utilize the resources by sharing a common code base and data.

Four maturity levels are defined by Chong and Carraro [CC06] for SaaS architecture. In the first level, each tenant is provided with specific application for exclusive use. In the second level, each tenant uses his own specific application but configuration tools are provided for application code. In the third level, all the tenants use the same instance of the application. In the fourth level, the tenants share a set of application instances.

Regarding the multi-tenant data architectures there three degrees of multi-tenancy exist [CC06]. In the first degree each tenant has its own database and hence the recovery of tenant data on failure can be achieved quite easily. However, this is not multi-tenant efficient as the number of databases per database server is limited. In the second degree, tenants have their own tables but all the data resides on the same database. Finally, in the last approach the data of different tenants are merged into same table and hence this architecture allows more tenants per database.

As part of this thesis for enabling scalability support multiple tenants share the same middleware resources, e.g. the LoadBalancer or ServiceMix cluster, Active MQ Broker network cluster and database registries.

2.7. Nginx Proxy Server

Nginx is a free open-source reverse proxy server which provides load balancing, caching, access and bandwidth control, and the ability to integrate efficiently with a variety of applications, have helped to make nginx a good choice for modern website architectures [Nginx]. There are also lot of third party modules, which come integrated with Nginx. Developers can also develop and integrate their own individual modules.

Also handling high concurrency along with high performance and efficiency is the main benefit of deploying the Nginx.

2.7.1. Nginx Architecture

Nginx is a specialized load balancing solution to achieve high performance and scalability. It follows a different approach than the traditional process or thread based model of handling concurrent connections. The architecture of Nginx is completely based on event based mechanisms. The tasks are assigned to processes based on multiplexing and event notifications. Connections are processed in a highly efficient run-loop in a limited number of single-threaded processes called workers. Within each worker Nginx can handle many thousands of concurrent connections and requests per second.

Nginx runs a single master process and several worker processes, cache loader and cache manager. All the processes use shared memory mechanism for inter-process communication. The master process is run as the root user. The cache loader, cache manager and workers run as an unprivileged user. The main operation of web server is reflected from the worker process as this is the main process responsible for accept, handle and process connections from the clients. The cache loader process loads the disk cache and in memory database. The cache manager is mostly responsible for cache expiration and invalidation.

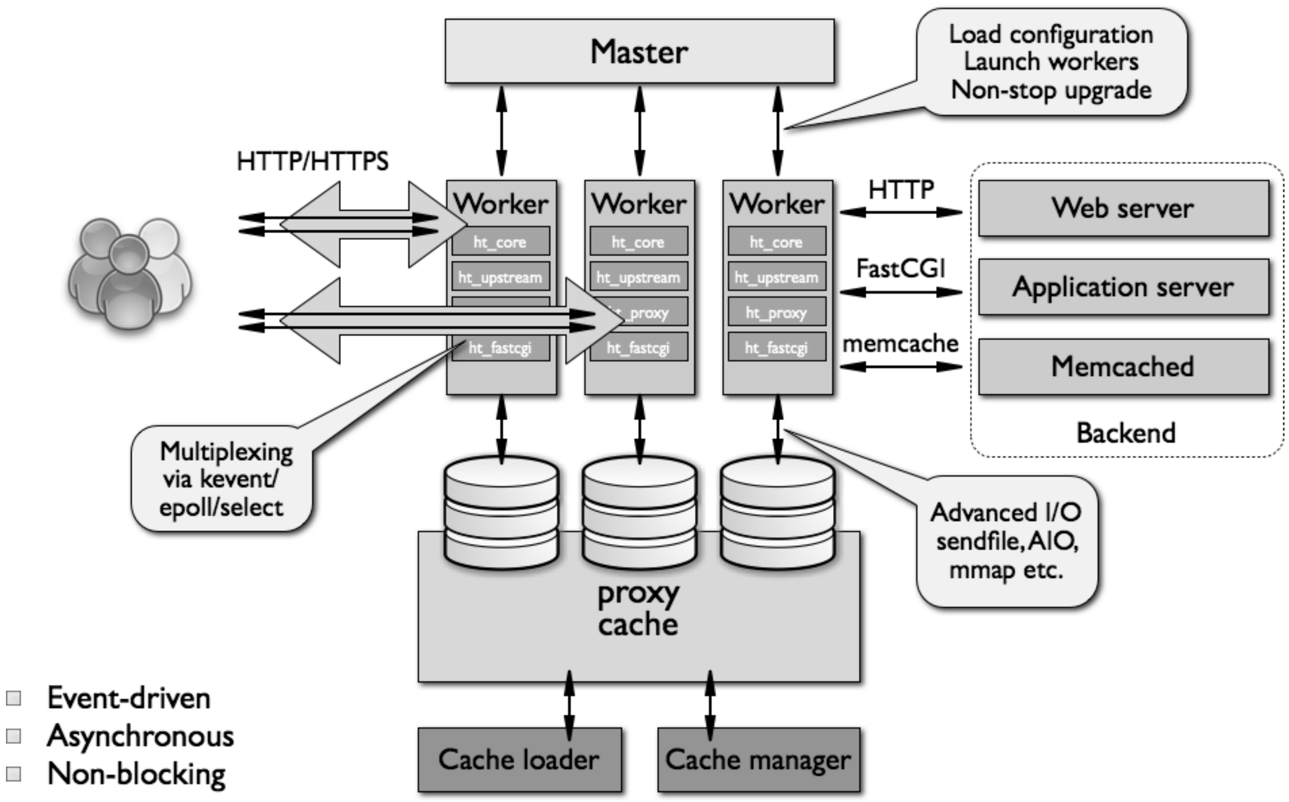


Figure 2.7: Nginx Architecture [Nginx]

The Nginx worker code contains the core and functional modules. The core is responsible for maintaining a tight run-loop and executing the modules code at each stage of processing the request. The presentation and application layer functionality is present in the functional code. Along with event notifications Nginx uses disk I/o performance enhancements in Linux, Solaris and BSD-based operating systems, like kqueue, epoll, and event ports [Nginx]. Also in terms of memory Nginx is very efficient as it does not fork a process or thread per connection. The scalability of Nginx is very high as it spawns several workers to handle connections. Generally, a separate worker per core allows full utilization of multicore architectures, and prevents thread thrashing and lock-ups. There's no resource starvation and the resource controlling mechanisms are isolated within single-threaded worker processes.

2.8. Apache Camel

Apache Camel is a powerful open source integration framework based on known Enterprise Integration Patterns with powerful Bean Integration. Apache Camel eases the realization of Enterprise Integration Patterns by means of specifying routing and mediation rules in either a Java based Domain Specific Language (or Fluent API), via Spring based Xml Configuration files or via the Scala DSL[APA11a]. The core feature of Camel is its routing and mediation engine. A routing engine will selectively move a message around, based on the route's configuration. In Camel's case, routes are configured with a combination of enterprise integration patterns and a domain-specific language (DSL).

A high level architecture of camel consists of *Routing Engine, Processors* and *Components*. The Routing Engine routes the messages under the hood. The message transformation and manipulation is done by the Processors during the message routing. Components are the extension points in Camel for adding connectivity to external systems i.e. the camel is exposed to outside environment through an endpoint interface called as Component. To expose these systems to the rest of Camel, components provide an endpoint interface [CiA11].

An endpoint is an abstraction that models the end of a message channel through which a system can send or receive messages. The endpoint representation in Apache Camel is done using Uniform Resource Identifiers (URIs). Hence Camel can work with any kind of Transport or messaging architecture such as HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF Bus API. Apache camel has minimal dependencies for embedding Java applications and also has a small set of libraries. Camel can work with same API regardless of which kind of Transport used.

The ServiceMix-camel JBI SE provides integration support between camel and JBI endpoints. Muhler extends this component and allows dynamic internal creation of tenant-aware endpoints in the ServiceMix-camel-mt JBI SE [Muh11]. The main goal of this extension is to provide an integrated environment between JBI and camel supported endpoints. However, multi-tenancy is only supported at the tenant level only between JBI endpoints. Therefore, Gomes has enabled the dynamic deployment of multi-tenant aware endpoints at the user level, by means of enabling the deployment of multiple user endpoints for a single tenant [Gom12].

2.9. ActiveMQ

ActiveMQ is an open source, Java Message Service (JMS) 1.1–compliant, message-oriented middleware (MOM) from the Apache Software Foundation that provides high availability, performance, scalability, reliability, and security for enterprise messaging [AMQinA]. The goal of ActiveMQ is to provide message-oriented integration across different languages and also across different platforms possible. ActiveMQ supports wide range of connectivity options and various kinds of protocols such as TCP, HTTP /S, UDP, SSL, STOMP and more. Since ActiveMQ is an implementation of JMS 1.1 Specification it provides synchronous and asynchronous message delivery capabilities. ActiveMQ also provides persistency and security, in terms of authentication and authorization depending on our security needs.

JMS is the main building block for the ActiveMQ. The JMS specification defines two kinds of clients: JMS clients and non-JMS clients. The JMS clients completely use JMS API's for communicating with JMS provider. If the JMS client uses any additional features then this client may not be portable with another JMS provider. The *MessageProducer* class is used by the JMS client for sending the JMS messages. When creating the Producer Session.createProducer() the default destination would be set however this can be overridden using MessageProducer.send() API. There are also lots of APIs for setting message headers. The JMS clients use *JMS MessageConsumer* class for consuming the produced messages. Messages can be consumed either Synchronously using receive() or asynchoronously using a *MessageListener* implementation. The MessageListener.onMessage() is invoked when the message arrives on the destination. Non-JMS clients use non JMS API's such as CORBA IIOP protocol or some other native protocol beyond Java RMI.

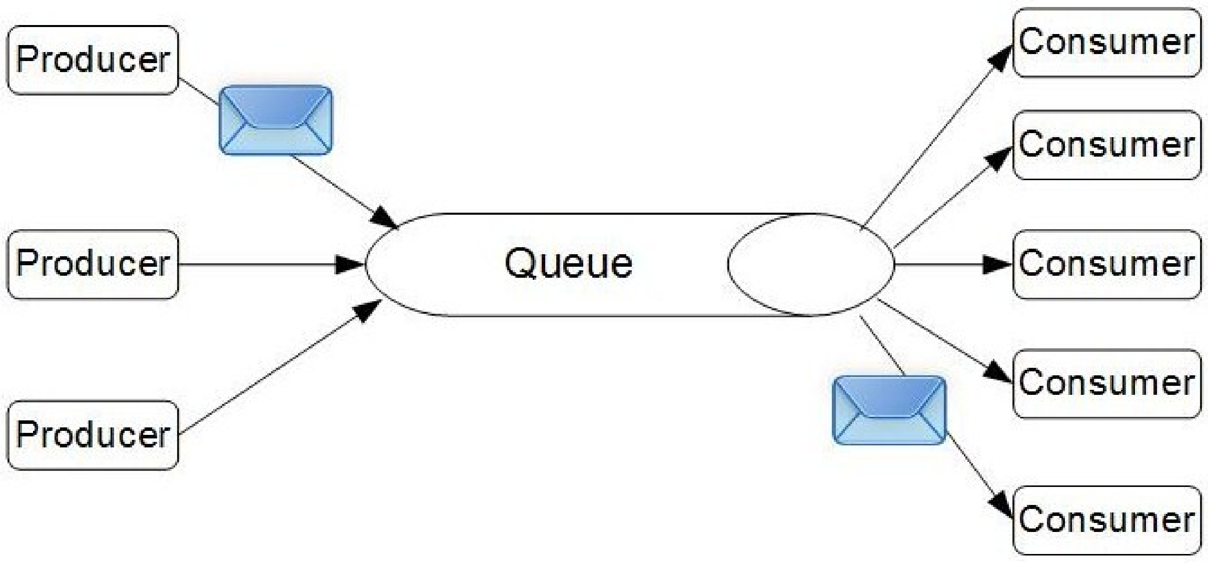


Figure 2.8 : Point-to-point messaging using one to one messaging paradigm. [AMQinA]

ServiceMix comes with an Active MQ instance and we use this ActiveMQ instance for interconnecting the ServiceMix instances forming a network of brokers which will be later used for JBI clustering to enable high scalability. These approaches are presented and described in detail in chapters 5 and 6.

3. Related Works

In the previous chapter, the fundamental background knowledge necessary to understand the concepts and principles discussed in this thesis were introduced. This chapter describes the existing approaches for enabling horizontal scaling of the ESB, and presents the work of other authors investigating load balancing and scalability solutions. Kumar and Kodukula [KK12] focus on proposing a scalable architecture in the Cloud, considering the Amazon Cloud and its available Web services. This thesis aims to extend previous works presented in [ESBMt] towards enabling administration and management of horizontally scaled multi-tenant aware ServiceMix instances.

3.1. Load Balancing Architectures in the Cloud

We introduce Scalability into a System so that the systems performance does not decrease even with the increased or varying load. There are two kinds of Scalability [IBMScl]:

- Vertical Scaling: This is increasing the size of the system by adding more processors and storage to enable symmetric multi-processing to extend processing capability. Generally this form of scaling employs only one instance of the operating system.

- Horizontal scaling: Here we will have multiple independent instances to provide more processing power. Independent instance means each and every instance will have its own operating system residing on separate server.

Horizontal Scalability is discussed in their work by means of adding more machines to a multi-instance container.

Kumar and Kodukula suggested a framework for building scalable architecture in the Cloud. It is complex to design an ideal scalable load balanced infrastructure. There exists a high dependency between the scalable infrastructure and application architecture. If the application architecture is not scalable, an enabled horizontal scalability cannot be guaranteed.

An ideal scalable architecture is defined by the following characteristics [KK12]:

- Increased resources result in proportional growth in performance.
- A scalable service must be able to handle heterogeneity
- A scalable service is operationally efficient
- A scalable service is durable
- A scalable service should become more cost effective when it grows.

The components used by Kumar and Kodukula are Amazon Elastic Compute Cloud (Amazon EC2), Amazon S3, Amazon Elastic Block Storage (EBS), Amazon Simple Queue Service (Amazon SQS), Amazon SimpleDB, Amazon Relational Database Service (RDS), Amazon CloudFront the underlying Cloud components. They have implemented this scalable architecture to achieve the above characteristics following the rules laid out in [Var11]. The main goal was to achieve automatic recovery on failover, decoupling of components, introducing elasticity and parallelization. Although Kumar and Kodukula investigations target load balancing and scalable architecture approaches, multi-tenancy awareness is not considered at the middleware level.

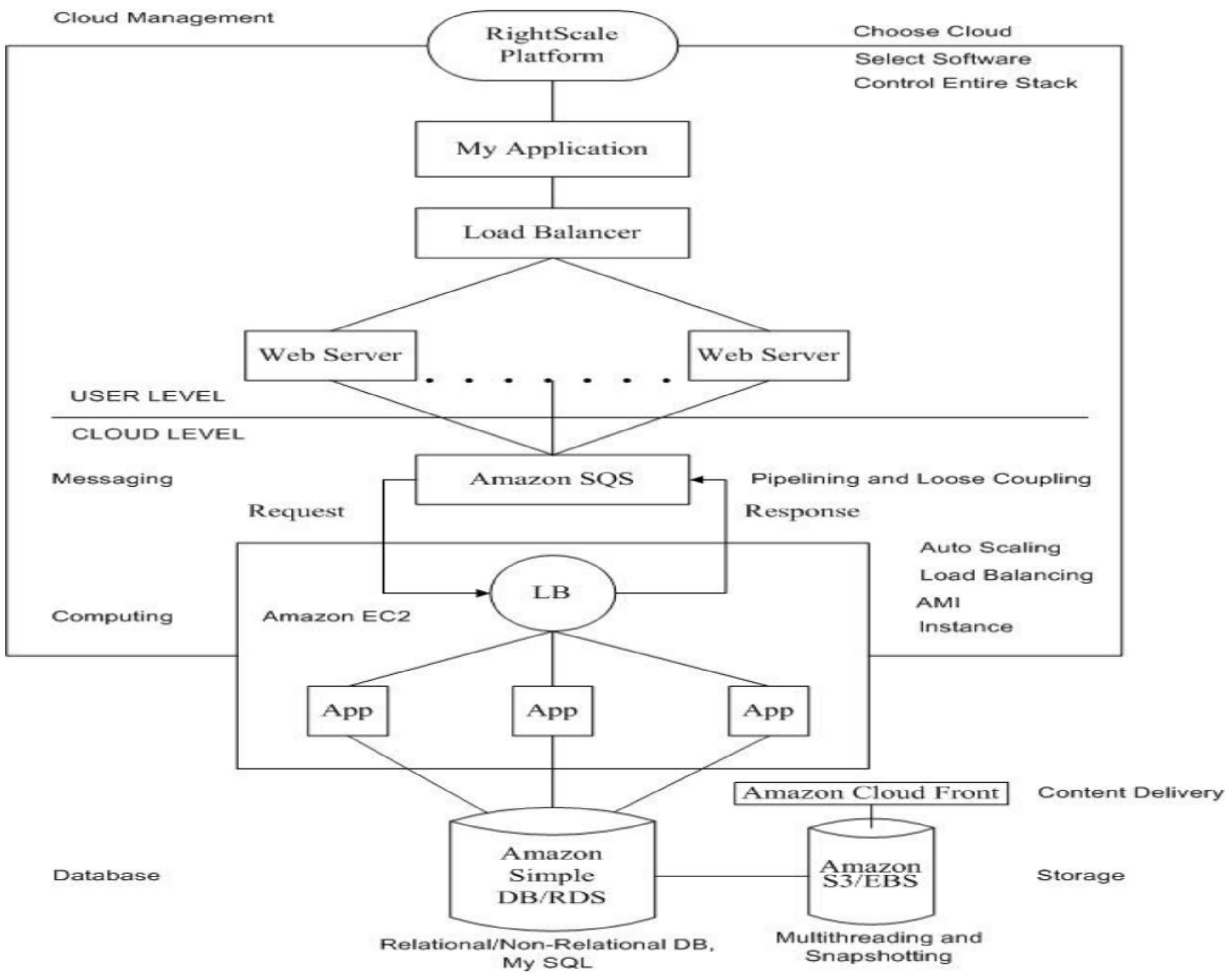


Figure 3.1 : Generalized Framework for scalable architecture [KK12]

In this thesis we also follow a similar architecture as in Figure 3.1 where we have a Nginx proxy server which receives the request and distributes the load to different instances. If we look at Muhler's thesis [Muh11] he has a set of ESB instances in a cluster. The JBIMulti2 application developed by Muhler puts the management message (containing BC or SA) to an external ActiveMQ topic. JMSManagementService OSGi bundle is developed and deployed on top of ApacheServiceMix. This service listens for the JMS topic and when a management message arrives on the JMS topic all the instances will deploy the same message. This approach can result in bottle neck when we need to support more number of tenants and users. So instead of replicating instances we interconnect the different instances and distribute the load across all the instances. In the next section we see how the clustering approach can be used to interconnect different ESB instances.

3.2. Multi-instance Management Approaches

Clusters constituted by one or more instances require administration and management operations in order to fulfill performance demands, e.g. discovery of available instances, dynamic deployment of instances to bear workload peaks, etc. Clustering is a technique for grouping similar instances or components to ease load to different components so that an individual request can be routed to a component that holds the specific data needed to process the request. This approach results in an increased performance as the load is redirected to specific instances in order to ameliorate the performance degradation. The probability of generating bottlenecks in the system is therefore reduced. If one of the instances is experiencing high load or becomes unavailable then another similar instance in some other cluster can process this request.

Clustering also provides the following benefits [ClusWSO2]:

- High availability – A server may be down due to many reasons like servicing or maintenance. Clustering for High availability results in less service interruptions.

- Simplified administration – It is easy to add or remove resources to meet the load requirements. Also Administration in cluster is simplified because it allows us to manage a group of systems as a single system.

- Ease of monitoring – It's easier to add and monitor new instances to a cluster.

- Increased Scalability – Since the load is distributed among the clusters this will certainly result in increased scalability able to handle more load.

- Low cost – Clustering increases the scalability and fault tolerance.

The above set of requirements can be reused for our approach. The above characteristics are essential especially if we are dealing with increased or varying load where performance and reliability is critical. Below we see how we can use these clustering techniques for grouping the ESB instances and how they relate to ActiveMQ.

Generally for enabling horizontal scalability there are 3 approaches:

- Load Balancing – Here all the ESB instances will be the same replicas hosting same binding components and service assemblies. So whenever a request arrives at LB we can route the request to any one of the instance which is not loaded with much traffic using one of the algorithms specified in section 2.5.1.

- Clustering – Here we establish a connection bridge between the ESB instances and distribute the load across all the instances so that there is no bottle neck to support more tenants or users.

- Hybrid – Here we use a combination of both of the above approaches by forming a network of brokers using ActiveMQ. We will see how this clustering of JBI endpoints is achieved in the coming chapters in 5 and 6.

3.3. Evaluating Clustering Scenarios

In this section two possible scenarios described by Muhler for enabling horizontal scalability in a multi-tenant aware ESB are analyzed.

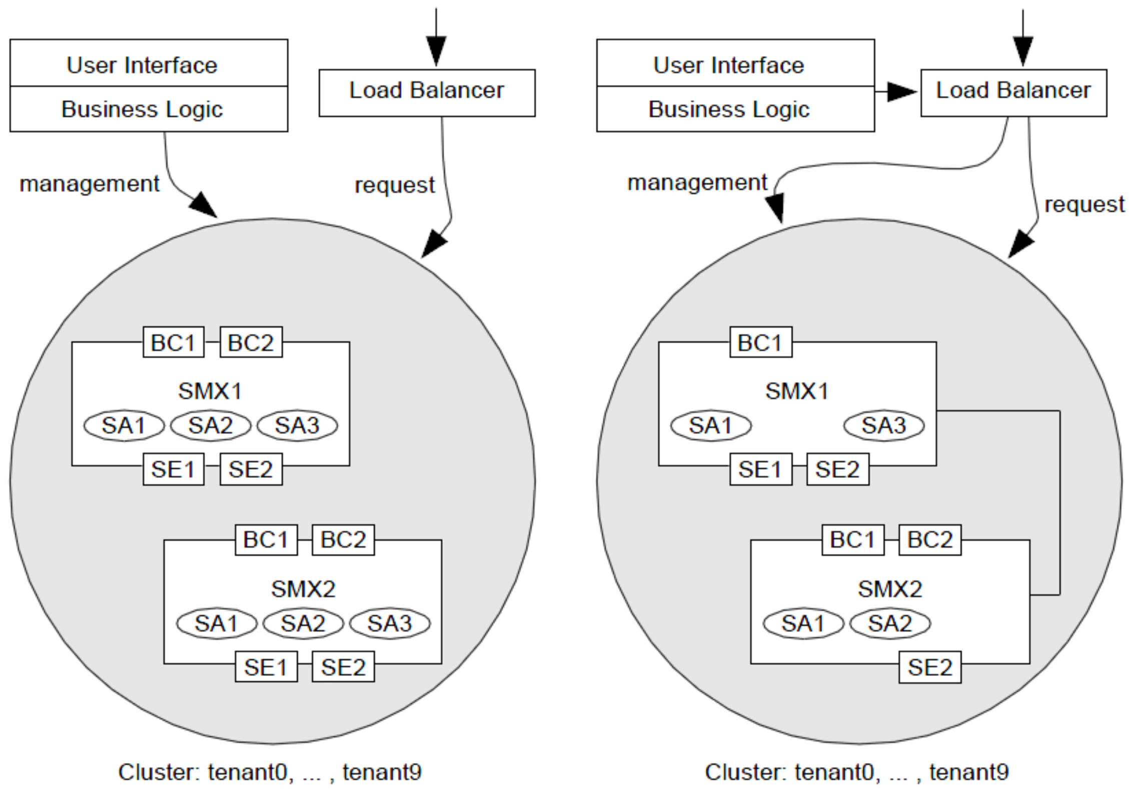


Figure 3.2 ; The two clustering scenarios being evaluated [Muh11]

In first scenario instance replication technique takes place, as all instances host same BC and service deployment endpoint configurations. Hence in this approach resources are not optimally utilized. Also each ESB can reach saturation due to the number of service endpoint configurations deployed. We might not be able to fulfill the request of tenant organization when the threshold limit is exceeded.

In second scenario the interconnected instances share BCs. The service endpoints are distributed across all the instances. Due to the endpoint distribution across multiple instances, e.g. based on assigning a concrete instance a task, the probability for a bottleneck to occur decreases when the number of tenants, users, and load increases. The second scenario can be implemented using JBI clustering technique and ActiveMQ network of brokers which offer lot of advantages as discussed below.

There are multiple advantages when forming a network of brokers [AMQMid]:

- Scalability – we can support increased load.
- High availability – Clients can attempt to connect to multiple brokers in a failover mode.
- Network Isolation and traffic limiting.
- Security – destination filtering can prevent certain users from sending requests to certain instances.

ServiceMix is shipped with a JBI cluster engine which can be used for clustering endpoints. Its main features are [SmxCl]:

- Transparent remoting.
- Rollback and redelivery when a JBI exchange fails.
- Load balancing among JBI containers able to handle a given exchange.
- Pause new exchanges processing when the number of concurrently processed messages reach a given threshold.

The second scenario offers more flexibility as there is no bottleneck to support more number of tenants or users so we will go-ahead with the implementation of this approach.

4. Concept and Specification

This chapter describes the System Overview of the components used for the administration and management of multi-tenant ESB. An overview of the key components used for managing the ESB cluster is also described. The functional and non-functional requirements including the detailed use cases are also given in the following sections.

4.1. System Overview

Muhler has developed a multi-tenant aware administration and management application for JBI environments [Muh11]. The specification described in below sections focus on the requirements for the load balancer, extensions done to the existing JBI Multi2 components including the JMS component.

4.1.1. Components

An highlevel overview of the system is given in the figure below.

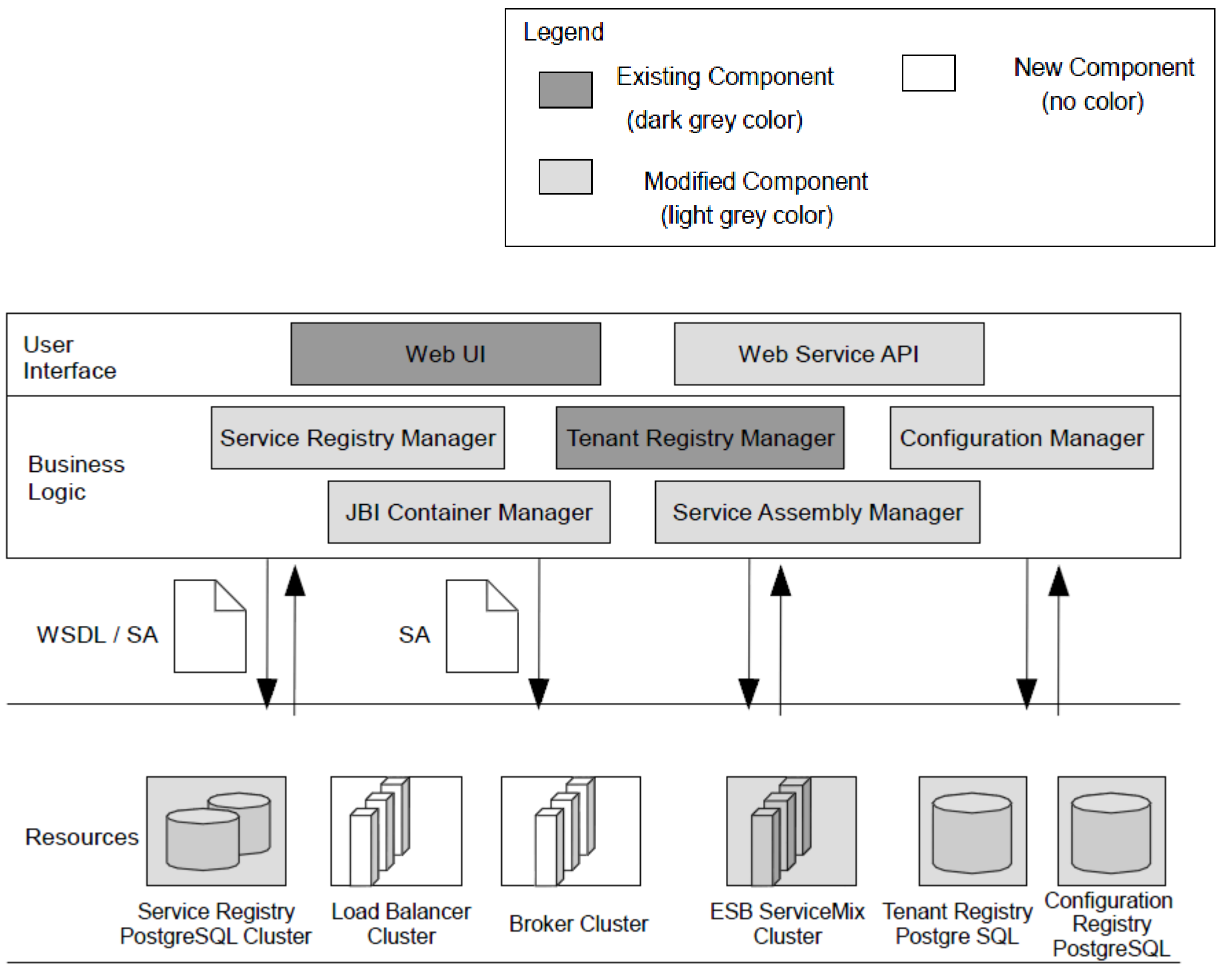


Figure 4.1 : Overview of JBIMulti2 component extended from [Muh11]

Muhler has designed and realized a multi-tenant aware administration and management application for JBI Environments [JBI05]. In his approach, ServiceMix has been used as the proof-of-concept in order to enable multi-tenancy in an ESB solution. This system is called JBI Multi-tenancy Multi-container support (JBIMulti2) as it distinguishes each tenant organization from the other by ensuring multi-tenancy. For enabling horizontal scalability on this system we extend some of the components which he has designed and introduce new components such as Load balancers and ActiveMQ clusters as shown in Figure 5.1.

JBIMulti2 resources layer is constituted by three main registries:: *Tenant Registry, Service Registry and Configuration Registry.* The tenant's information is obtained from the Tenant Registry and it also describes the roles and allowed operations. The Service Registry stores the deployed Service Assembly information for each tenant. This component is modified to store the endpoint information as well as for a single tenant we support distributing the endpoints across multiple instances. All the information not related to tenants is stored in Configuration Registry.

Load balancer cluster on the resource layer is shared by all the tenants. Each Load Balancer can manage many JBI clusters. The JBI clusters have to be registered with LB before it can control the cluster. The LB receives the request and queries the registries to find out the location of the endpoint and redirects the request to the specic instance.

The Broker cluster is formed by interconnecting all ESB instances with in one JBI cluster and this enables massive scalability. We also form a network of broker clusters to enable massive scalability. Broker cluster is formed by interconnecting all ESB instances with in one JBI cluster using internal ActiveMQ of each of the ESB instance.

4.2. Multi-tenancy and Multi-instance

The JBI Multi2 System designed by Dominik ensures isolation of the data not only between tenants but also between the users. The authentication and access control is done on two hierarchial levels of tenants and the tenant users. All the operations are done by System admin role and the tenant admin role as described below.

4.2.1. Role-based Access Control

The System Admin role has the maximum permissions and he is the one responsible for registering the Load Balancer, JBI Cluster and the ESB instances. The tenant role is further classified into Tenant administrator and Tenant operator. The System Admin creates the Tenant Admin and he can also assign the Tenant admin roles and the quotas of resources. The Tenant Admin has rights to create the Tenant operators who can deploy the Service Assemblies and consume the quotas of resources assigned to them by the Tenant Admin.

The Tenant user can have multiple tenant administrator roles or tenant operator roles [Muh11].
The System Admin when registering the new instance or the JBI Container has the option for specifying the type of protocols like Http, JMS or Email supported and also the kind of endpoints such as producer or consumer that can be hosted on the instance. Another main requirement while installing the Binding Component on the ESB instance is that it should use an unique port than all other instances running on the same server else we have a clash on the ports. For this reason the System Admin has the rights to assign the ports for each of the BC when installing

that particular JBI BC. Also each of the BC installed can be either multi-tenant or non-multi-tenant. So the System Admin also has the rights to specify the kind of tenancy supported while deploying the BC.

The tenant operator is the one responsible for deploying the Service Assemblies on top of these BCs. One of the optional requirement for the tenant operator is he could specify the instance ID where a particular SA has to be deployed. If the tenant operator does not specify the instance ID then the system in turn finds the list of ESB instances with in a particular JBI cluster which are compatible to deploy this SA on top of the BC. Then we narrow it down to the number of the endpoints deployed on each instance. The instance deployed with least number of endpoints is selected and this instance ID is associated with the SA. Also there are lot of web service api's available for the tenant operator in case if he wants to decide upon the instance where a particular SA should be deployed on.

4.2.2. Communication Requirements

Multi-tenant aware communication requirements have been previously identified [Gom12]. However, they are related to a single instance, rather than the administration and management of multiple instances within a cluster. Therefore, we provide the following communication requirements:

1. **Clustering:** We use JBI clustering for communication between the ESB instances supported by ActiveMQ. Clustering is done only for the provider endpoint.

2. **Tenant-aware messaging:** The messages exchanged between the tenant aware endpoints are enriched with tenant and user information.

3. **Tenant-aware endpoints:** Tenant aware endpoints are packed in their corresponding SUs and SUs in turn must be packed in individual Service Assemblies, as the system must support the endpoint deployment on a specific instance.

4. **Tenant-aware routing and context:** The deployment of multi-tenant aware endpoints must be followed by creation of tenant-aware context. The routing operations between the provider and consumer endpoints must identify the tenant and the user initiating the operation.

5. **Tenant Configuration isolation :** The tenant data between the two users belonging to different tenants should be isolated as it may contain sensitive information.

6. **Tenant-aware correlation :** The data exchange between the data sources and the tenants must not interfere with other tenants and hence the tenant requests must be correlated with its corresponding response.

.

4.3. Use Cases

Three main actors have been identified in previous works: System Administrator, Tenant Administrator and finally the Tenant operator [Muh11].

The System Administrator has access to all the resources in the system including the operations of Tenant Administrator and Tenant Operator. We identify in this thesis the necessary operations for administrating and managing multiple instances across multiple clusters. Therefore, the initial set of use cases defined in [Muh11] is extended in this work. The set of operations for the System Administrator and Tenant Operator are increased, as these incorporate the potential functionalities to enable multi-instance management. However there are no changes made on Tenant Administrator operations. The use case diagrams for both the System Administrator and Tenant Operator are shown in Figure 5.2 and 5.3 respectively.

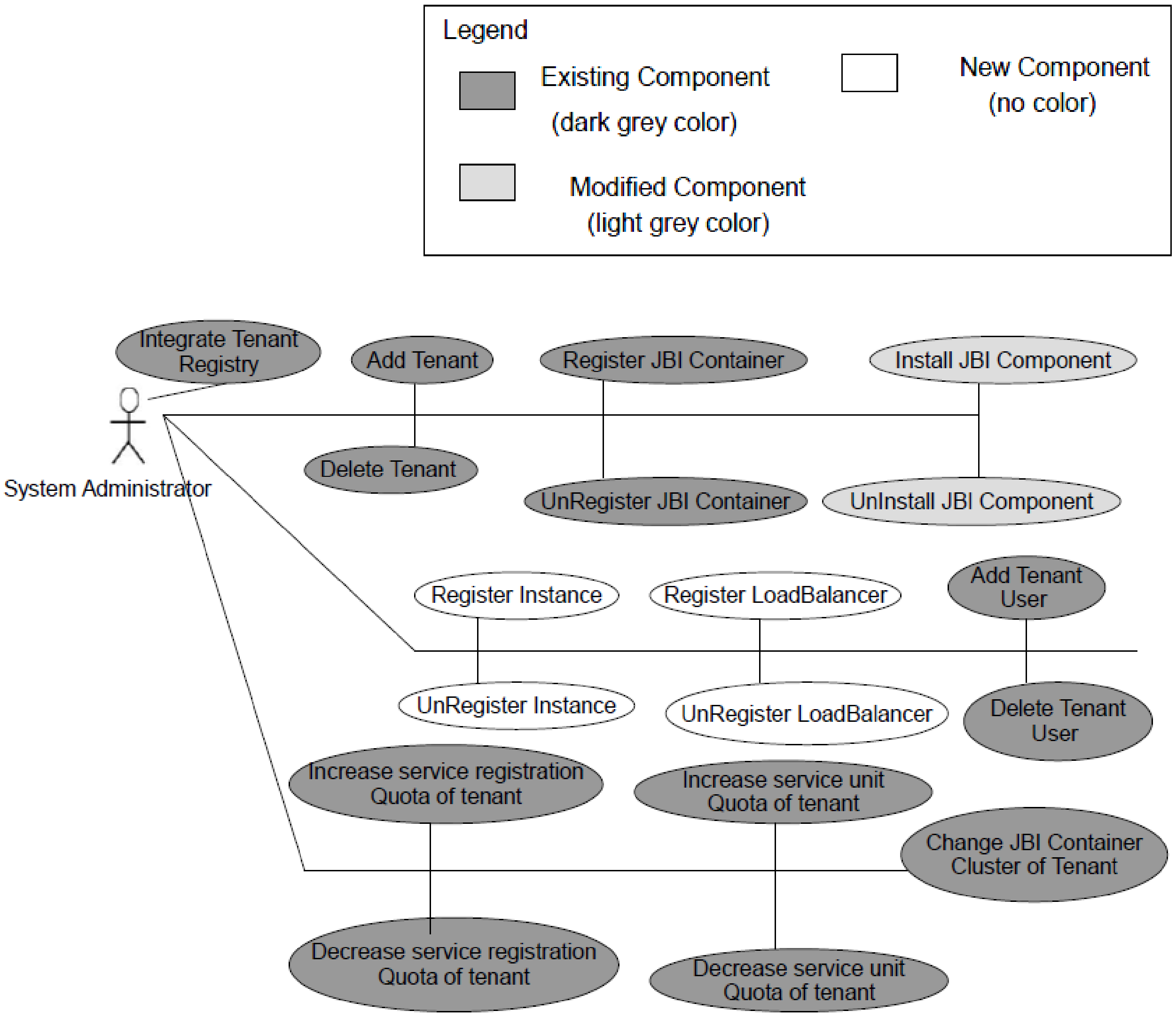

Legend
Existing Component (dark grey color)
New Component (no color)
Modified Component (light grey color)
Integrate Tenant Registry
Add Tenant
Register JBI Container
Install JBI Component
System Administrator
Delete Tenant
UnRegister JBI Container
UnInstall JBI Component
Register Instance
Register LoadBalancer
Add Tenant User
UnRegister Instance
UnRegister LoadBalancer
Delete Tenant User
Increase service registration Quota of tenant
Increase service unit Quota of tenant
Change JBI Container Cluster of Tenant
Decrease service registration Quota of tenant
Decrease service unit Quota of tenant

Figure 4.2 : System Administrator Use Case diagram extended from [Muh11]

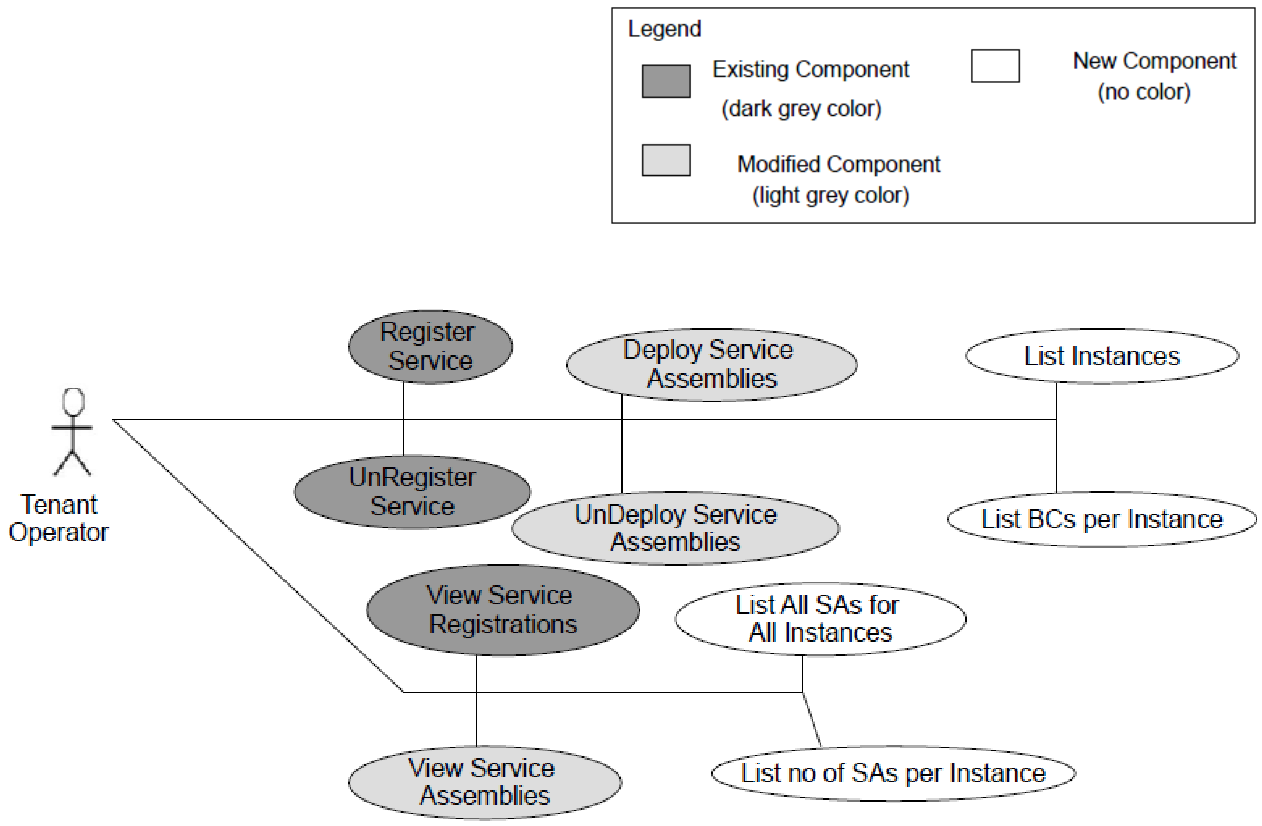

Legend
Existing Component (dark grey color)
New Component (no color)
Modified Component (light grey color)
Tenant Operator
Register Service
Deploy Service Assemblies
List Instances
UnRegister Service
UnDeploy Service Assemblies
List BCs per Instance
View Service Registrations
List All SAs for All Instances
View Service Assemblies
List no of SAs per Instance

Figure 4.3 : Tenant Operator Use Case diagram extended from [Muh11]

In the remainder of this section the detailed list of use cases are described.

Name	**Register new ESB instance**
Goal	The system administrator wants to register a new ESB Instance.
Actor	System Administrator
Pre-Condition	The JBI Cluster is already registered.
Post-Condition	The ESB instance is registered and added to the specified JBI cluster.
Post-Condition in Special Case	The ESB instance is not registered.
Normal Case	1. The System Administrator commands to register the new ESB instance specifying optional properties like protocol supported and end-point types. 2. The System starts a distributed atomic transaction. 3. The System registers the new ESB instance into the JBI Cluster. 4. The System finishes the distributed transaction.
Special Cases	2a. If invalid protocol is entered then proper error message is displayed and the system aborts the registration of the new ESB instance. 2b. If invalid endpoint type is entered then proper error message is displayed and the system aborts the registration of the new ESB instance. 2c. The specified cluster to register the new ESB instance does not exist or has been previously deleted, so the registration of the instance fails with proper error message. 3a. The system cannot commit the transaction to register a new instance in JBI Cluster. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.1 Description of Use Case *Register new ESB instance.*

Name	**Registering the Load Balancer**
Goal	The system administrator wants to register the Load balancer instance for the specified JBI clusters.
Actor	System Administrator
Pre-Condition	The JBI Cluster already exists and it contains registered JBI containers.
Post-Condition	The Load balancer is registered.
Post-Condition in special case	The Load balancer is not registered.
Normal case	1. The System Administrator commands to register the Load balancer by specifying the associated JBI clusters. 2. The System starts a distributed atomic transaction. 3. The System registers the Load balancer. 4. The System finishes the distributed transaction.
Special Cases	2a. If the given JBI cluster does not exist in the system proper error message is Displayed and transaction aborts. 2b. Concurrently if the JBI containers are unassigned from the system then system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the specified JBI clusters. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.2 Description of Use Case for *Register Load balancer.*

Name	**Install JBI BC**
Goal	Install a JBI BC on the ESB instance.
Actor	System Administrator
Pre-Condition	ESB instance is already registered with an instance ID.
Post-Condition	The JBI BC is installed successfully.
Post-Condition in Special case	The JBI BC installation failed.
Normal Case	1. The System Administrator commands to install the JBI BC on specified Instance ID along with the port for this BC. 2. The System starts a distributed atomic transaction. 3. The System installs the new JBI BC on the specified instance ID. 4. The System finishes the distributed transaction.
Special Cases	2a. The instance ID does not exist or the specified port is already occupied. Corresponding error message is displayed and the installation of JBI BC fails. If the port is already occupied then the available list of ports is displayed. 2b. Concurrently the registered instance might be unavailable or deleted. The system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the installation of JBI BC. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.3 Description of Use Case *Install JBI BC.*

Name	**Deploy Service Assembly without specifying the Instance ID**
Goal	Deploy two Service Assemblies in one request with one ServiceUnit each without specifying the instance ID.
Actor	Tenant Operator
Pre-Condition	The Tenant Operator has the permissions to install the SA.
Post-Condition	The SAs are deployed successfully.
Post-Condition in Special cases	The deployment of the SAs failed.
Normal Case	1. The tenant Operator commands to deploy the SAs specifying the endpoint types. 2. The System starts a distributed atomic transaction. 3. The System takes a look at the BC and gets the instanceId registered for These BCs. 4. The System figures out the instanceID with least number of endpoints already deployed. (If there are more than one instance matching for same BC with same number of endpoints then one of them will be selected randomly). 5. The System finishes the distributed transaction.
Special Cases	1a. If the endpoint types are invalid the transaction aborts and the deployment of the SAs fails with proper error message. 2a. Concurrently if the registered instances with the compatilbe BCs are deleted, the System shows a suitable message and aborts. 3a. If the System cannot find any instance ID compatible for installing the SAs (meaning no suitable BC found on the instances), then the transaction aborts with suitable message. 4a. If the SAs names already exist in the system then proper error message is displayed and the transaction is rolled back. 5a. The system cannot finish the transaction with the instance ID and the Configuration and Service Registry. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.4 Description of Use Case *Deploy Service Assembly without specifying the Instance ID.*

Name	**Deploy Service Assembly with specifying the Instance ID**
Goal	Deploy two Service Assemblies with one ServiceUnit each specifying the instanceID.
Actor	Tenant Operator
Pre-Condition	The Tenant Operator has the permissions to install the SA.
Post-Condition	The SAs are deployed successfully.
Post-Condition in Special cases	The deployment of the SAs failed.
Normal Case	1. The tenant Operator commands to deploy the SAs specifying the endpoint types and the instance ID for each of the SA to be deployed on. 2. The System starts a distributed atomic transaction. 3. The System takes a look at the BCs and gets the instance Id registered for these BCs. 4. The System deploys the SA on top of BCs on the specified instance Ids. 5. The System finishes the distributed transaction.
Special Cases	1a. If the endpoint types are invalid the transaction aborts and the deployment of the SAs fails with displaying proper error message. 2a. If there is no compatible BC installed on the specified instance ID then the deployment of the SAs fails with displaying proper error message. 2b. Concurrently if the registered instances are deleted, the System shows a suitable message and aborts. 3a. If the System cannot find any compatible BCs on the specified instances then the transaction aborts with suitable message. 4a. If the SAs names already exist in the system then proper error message is displayed and the transaction is rolled back. 5a. The system cannot finish the transaction with the instance ID and the Configuration and Service Registry. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.5 Description of Use Case *Deploy Service Assembly with specifying the Instance ID.*

Name	**Listing all the registered ESB instances**
Goal	The System Administrator wants to list all the registered ESB instances.
Actor	System Administrator
Pre-Condition	There are already ESB instances registered in a particular JBI cluster.
Post-Condition	The registred ESB instances are listed.
Post-Condition In Special Case	The ESB instances are not registered
Normal Case	1. The System Administrator commands to list all the instances specifying the JBI cluster. 2. The System starts a distributed atomic transaction. 3. The System lists all the registered ESB instances in the specified JBI cluster. 4. The System finishes the distributed transaction.
Special Cases	2a. If the JBI cluster itself does not exist then proper error message is displayed and transaction aborts. 2b. Concurrently if the JBI cluster is deleted the system shows a suitable message and aborts 3a. The system cannot finish the transaction with the JBI Cluster and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.6 Description of Use Case *Listing all the registered ESB instances.*

Name	**List available ports per instnace**
Goal	The tenant operator wants to list all the available ports (for deploying the BCs) per instance.
Actor	Tenant Operator
Pre-Condition	The ESB instance is already registered in a particular JBI cluster.
Post-condition	All the available ports for the specific instance are displayed.
Post-Condition in special case	The ports are not displayed.
Normal case	1. The Tenant Operator commands to list all the available ports specifying the instance ID. 2. The System starts a distributed atomic transaction. 3. The System lists all the available ports for the specified instance ID. 4. The System finishes the distributed transaction.
Special Cases	2a. If the instance ID itself is not registered in the system then proper error message is displayed and transaction aborts. 2b. Concurrently if the instanceID is deleted the system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the instance ID and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.7 Description of Use Case *List available ports per instnace.*

Name	**List available Binding Components per instance**
Goal	The tenant operator wants to list all the Binding Components per instance.
Actor	Tenant Operator
Pre-Condition	The ESB instance is already registered in a particular JBI cluster and there are Binding Components already installed on the instance.
Post-Condition	All the BCs for the specified instance ID are displayed.
Post-Condition in special case	The Binding Components are not displayed.
Normal case	1. The Tenant Operator commands to list all the Binding Components specifying the instance ID. 2. The System starts a distributed atomic transaction. 3. The System lists all the available BCs for the specified instance ID. 4. The System finishes the distributed transaction.
Special Cases	2a. If the instance ID itself is not registered in the system then proper error message is displayed and transaction aborts. 2b. Concurrently if the instance ID is deleted the system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the Instance ID and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message. 4a. If the instance ID exists but there are no BCs at all installed on the instance then proper message is displayed.

Table 4.8 Description of Use Case *List available Binding Components per instance.*

Name	**List all SAs for all instances in the specified JBI cluster**
Goal	The tenant operator wants to list all the deployed Service assemblies for all the instances in a particular JBI cluster.
Actor	Tenant operator
Pre-Condition	There are ESB instances already registered in a particular JBI cluster and there are Binding Components already installed and service assemblies deployed on these BCs.
Post-Condition	All the deployed SAs for all the instances with in the specified JBI cluster are displayed.
Post-Condition in special case	The Service assemblies are not displayed.
Normal case	1. The Tenant Operator commands to list all the Service Assemblies for all the instances in a particular JBI cluster specifying the JBI cluster name. 2. The System starts a distributed atomic transaction. 3. The System lists all the deployed SAs for all the instances within a particular JBI cluster. 4. The System finishes the distributed transaction.
Special Cases	2a. If the JBI cluster itself is not registered in the system then proper error message is displayed and transaction aborts. 2b. Concurrently if all the instances are deleted from the cluster then system shows a suitable message and aborts. 2c. Concurrently if all the hosted BCs are deleted from all the instances then the system shows a suitable message and aborts. 2d. Concurrently if all the deployed SAs are deleted from all the instances in the cluster then the system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the specified cluster name and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message. 4a. The cluster, containers and the JBI BCs exist but there are no SAs deployed on these instances. Then proper message is displayed.

Table 4.9 Description of Use Case *List all SAs for all instances in the specified JBI cluster.*

Name	**List all SAs for for the specified instance ID**
Goal	The tenant operator wants to list all the deployed Service assemblies for the specified instance ID.
Actor	Tenant operator
Pre-Condition	There are ESB instances already registered in a particular JBI cluster and there are BCs already installed and SAs deployed on these BCs.
Post-Condition	All the deployed SAs for the specified instance ID are displayed.
Post-Condition in special case	The SAs are not displayed.
Normal case	1. The Tenant Operator commands to list all the Service Assemblies for the specified instance ID. 2. The System starts a distributed atomic transaction. 3. The System lists all the deployed SAs for the specified instance ID. 4. The System finishes the distributed transaction.
Special Cases	2a. If the instance ID is not registered in the system then proper error message is displayed and transaction aborts. 2b. Concurrently if the specified instance ID is deleted from the cluster then system shows a suitable message and aborts. 2c. Concurrently if all the hosted BCs are deleted from the specified instance ID then the system shows a suitable message and aborts. 2d. Concurrently if all the deployed SAs are deleted from the specified instance ID in the cluster then the system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the specified instance ID and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message. 4a. The cluster, containers and the JBI BCs exist but there are no SAs deployed on these instances. Then proper message is displayed.

Table 4.10 Description of Use Case *List all SAs for for the specified instance ID.*

Name	**Unregister ESB instance**
Goal	The System Administrator wants to unregister the ESB instance ID from the ESB Cluster.
Actor	System Administrator
Pre-Condition	The ESB instance is already registered in a particular JBI cluster.
Post-Condition	The ESB instance is unregistered from the system.
Post-Condition in special case	The ESB instance is not unregsitered.
Normal case	1. The System Administrator commands to unregister the ESB instance by specifying the instance ID. 2. The System starts a distributed atomic transaction. 3. The System unregisters the ESB instance. 4. The System finishes the distributed transaction.
Special Cases	2a. If the instance ID is not registered in the system then proper error message is displayed and transaction aborts. 2b. Concurrently if the specified instance ID is deleted from the cluster then system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the specified instance ID and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.11 Description of Use Case for *Unregister ESB instance.*

Name	**Uninstall JBI BC from the specified instance ID**
Goal	The System Administrator wants to uninstall the JBI BC by specifying the component name and the Instance ID.
Actor	System Administrator
Pre-Condition	The ESB instance is already registered in a particular JBI cluster with the specified BC name already installed.
Post-Condition	The specified JBI BC is uninstalled from the specified instance ID.
Post-Condition in special case	The JBI BC is not uninstalled.
Normal case	1. The System Administrator commands to uninstall the JBI BC by specifying the BC name and the instance ID. 2. The System starts a distributed atomic transaction. 3. The System uninstalls the JBI BC from the specified instance ID. 4. The System finishes the distributed transaction.
Special Cases	2a. If the instance ID is not registered in the system then proper error message is displayed and transaction aborts. 2b. Concurrently if the specified instance ID is deleted from the cluster then system shows a suitable message and aborts. 2c. If the specified JBI BC is not installed on the system proper error message is displayed and transaction aborts. 3a. The system cannot finish the transaction with the specified instance ID, JBI BC and the Configuration Registry. a) The system rolls back the distributed atomic transaction and shows an error message.

Table 4.12 Description of Use Case for *Uninstall JBI BC from the specified instance ID.*

Name	**Undeploy Service Assembly**
Goal	The System Administrator wants to un deploy the Service Assembly for the specified tenant operator.
Actor	System Administrator
Pre-Condition	The tenant operator has a Service assembly already deployed.
Post-Condition	The Service assembly is un deployed.
Post-Condition in special case	The Service assemlby could not be un deployed.
Normal case	1. The System Administrator commands to un deploy the SA by specifying the SA name and tenant operator. 2. The System starts a distributed atomic transaction. 3. The System un deploys the SA. 4. The System finishes the distributed transaction.
Special Cases	2a. If the tenant operator does not exist in the system proper error message is displayed and transaction aborts. 2c. Concurrently if the specified tenant operator is deleted from the system then system shows a suitable message and aborts. 2d. If the specified SA is not deployed on the system proper error message is displayed and transaction aborts. 3a. The system cannot finish the transaction with the specified tenant operator, SA name and the Service Registry. a. The system rolls back the distributed atomic transaction and shows an error message.

Table 4.13 Description of Use Case for *Undeploy Service Assembly.*

Name	**Unregister Load balancer**
Goal	The System Administrator wants to un register the Load balancer instance.
Actor	System Administrator
Pre-Condition	The LB is already registered and JBI Cluster already exists and it contains registered JBI containers.
Post-Condition	The load balancer is un registered.
Post-Condition in special case	The load balancer is not un registered.
Normal case	1. The System Administrator commands to un register the Load balancer by specifying the Load balancer IP. 2. The system starts a distributed atomic transaction. 3. The system un registers the Load balancer. 4. The system finishes the distributed transaction.
Special Cases	2a. If the load balancer is not registered in the system then proper error message is displayed and transaction aborts. 2b. If the load balancer is not associated with any JBI clusters then proper error message is displayed and transaction aborts. 2c. Concurrently if the JBI containers are unassigned from the system then system shows a suitable message and aborts. 3a. The system cannot finish the transaction with the specified Load balancer IP. The system rolls back the distributed atomic transaction and shows an error message.

Table 4.14 Description of Use Case for *Unregister Load balancer.*

4.4. Non-functional Requirements

In addition to the functional aspects laid out in the previous parts of this chapter the following non-functional requirements have to be fulfilled for the multi-instance management of the ESB cluster.

- **Security** – One or more requirements about the protection of our system and its data. The tenant context contains not only tenant context information but also sensitive data such as access credentials, backend data source names. So this data should be available only to the system and should be protected from third party access or malicious attacks from outside.

- **Robustness** – Since the Load Balancer manages multiple clusters and acts as a gateway for all the containers we need to ensure that the system does not break under high loads.

- **Backward compatibility** – We need to ensure that the system must support both OSGi-based components and JBI-based components. Backward compatibility with multi-tenant JBI components developed in [Muh11] [Gom12] must be ensured.

- **Performance** – Extreme load on system is a possible bottle neck for the system. The Load Balancer must have a quick response time, high database transaction rates and throughput.

- **Extensibility** – The architecture, design and implementation of the system has been done is such a way to cater for future change. The components of the application are loosely coupled so that it is easier for further implementations or extensions.

- **Ease of Installation and maintainability** – The process to install and set up a running system must be well documented.

5. Design

This chapter describes the architectural design approaches for the concept and specifications identified in the previous chapter. This chapter is divided according to the two horizontal scalability approaches that must be taken into consideration in this thesis: non interconnected ESB instances and interconnected ESB instances. Furthermore, as JBIMulti2 is used as the basis to be extended in this thesis, the extensions in such application on the different levels are depicted.

5.1. Horizontal Scalability Support Architecture

Muhler describes in his work two possible scenarios for enabling horizontal scalability support for the multi-tenant aware ESB [Muh11]. In the first scenario all the ServiceMix instances host the same binding components and service endpoints deployment specifications. Therefore, when a request arrives at the load balancer it can be sent to any instance for serving it. In second scenario the ServiceMix instances are interconnected to each other and unlike first scenario they are not replicated. Each ServiceMix instance can contain different BCs and different endpoints. Messages can be routed between endpoints which are not hosted in the same instance.

5.1.1. Non Interconnected ESB Instances

In the first approach, all the instances host the same JBI BCs and the request can be sent to any one of the instances in the JBI cluster pool. This approach enables the categorization of cluster pools based on tenant organizations or quality of service levels.

The Load Balancer is responsible for managing the ESB cluster pools. When a request arrives to the LB it checks in Tenant Registry if tenant exists in system. The Configuration Registry is modified so that LB can keep track of the states of all the instances. The LB queries the Configuration Registry to get the instance which is least loaded and redirects the request to it.

With respect to the multi-tenancy support in JBIMutli2, in Muhler's work multi-tenancy was supported only at tenant level. However, as part of this thesis we introduce multi-tenancy at the user level as well. This means that when a request arrives to the LB it has the option to extract the tenant and user id meta-data information from the request. With this approach, the grouping of JBI clusters can be achieved depending on the tenant organizations and give preference for the tenants based on the class of service (for example, gold, silver, bronze etc.). However, this approach has two main drawbacks. On the one had since within one cluster all the instances host the same service endpoint deployment configurations there is wastage of resources. Secondly there may be saturation due to the fact that each of the instances within a cluster must host the same number of endpoints. By replicating ESB instances, the performance degradation can only be mitigated by adding replicas to the cluster. Therefore, the cluster might not be able to fulfill the request of the tenant organization when the number of user exceeds the ESB threshold limit.

With the previously described approach, we need to have a mechanism to monitor all the ESB instances within a JBI cluster and provide real-time feedback to Nginx server to be able to choose one of the most unloaded instances to serve the request. Calculating and estimating the load of replicated instances might increase the monitoring and workload distribution tasks complexity, as the ESB instances are horizontally replicated, and the number of endpoint does not vary across instances.

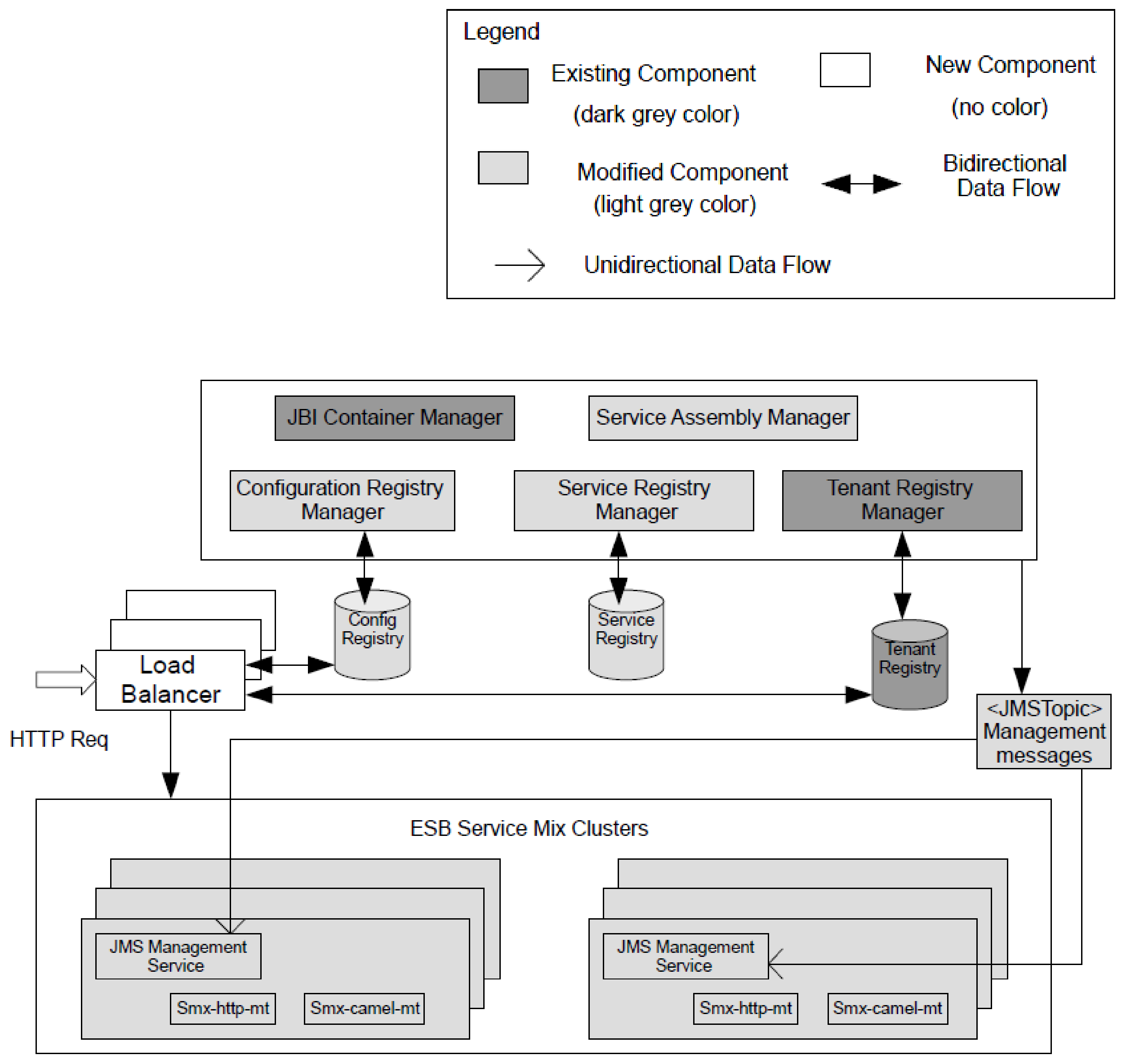


Figure 5.1: Architectural overview of the design approach to enable horizontal scalability with non Interconnected ESB instances.

5.1.2. Interconnected ESB Instances

ActiveMQ is a full-fledged Messaging Queuing Middleware. Each ServiceMix instance comes with built in ActiveMQ and we can use this ActiveMQ to connect all the instances to form a network of brokers. Network of brokers is usually used when we need large scalability and that is exactly the current need [AMQ]. Endpoints from one instance are reachable by another instance, as endpoint discovery is managed in conjunction by the JBI Clustering Engine and the Active MQ network of brokers. These capabilities can be exploited by assigning different roles to each ESB instance, e.g. one ESB instance is only responsible for message transformation, while another ESB instance provides support for message routing between HTTP endpoints.

We extend the JBI Multi2 component for supporting this approach so that the LB can figure out in which instance the service endpoint resides. It is also possible that the producer endpoint resides on one instance and the consumer endpoint on another instance. When a request arrives at the LB it extracts the tenantID and userID metadata from the request and by querying to external tenant and configuration registries it is discovered on which ESB instance this service endpoint is deployed and hence it can route the request to that specific instance. Unlike in approach 1 whenever a management message arrives on the JMSTopic it is not deployed on all the ESB instances instead filtering has been enabled at the JMSManagementService so that the BC or SA is installed only on the instance which it is targeted to.

The Active MQ network of brokers capabilities support three different configurations: *Static*, *Dynamic Discovery*, *Master Slave* and *Replicated Message Store*. In the static discovery support, the URIs must be explicitly defined. A connection using this discovery mechanism will attempt to connect to all URIs in the list until it is successful. In dynamic discovery, a discovery agent is used to locate the list of URIs to connect to. For this approach the brokers need to have the multicast discovery agent enabled on the broker. In master slave approach, messages are replicated on to slave machine so that even if the master goes down due to some technical issues we get immediate failover to slave with no message loss. In the replicated message store approach, the messages are stored on shared network drive so that if one broker network fails it can be taken over straight away by another network reducing the risk of message loss [Bnw].

Together with the ActiveMQ each ESB instance can use the JBI clustering engine which helps to implement the JBI applications in a clustered environment and ensure processing even when there is a hardware or software failure [ClusO]. A clustered instance belongs only to that particular cluster and there can be multiple clusters in a system’s domain.

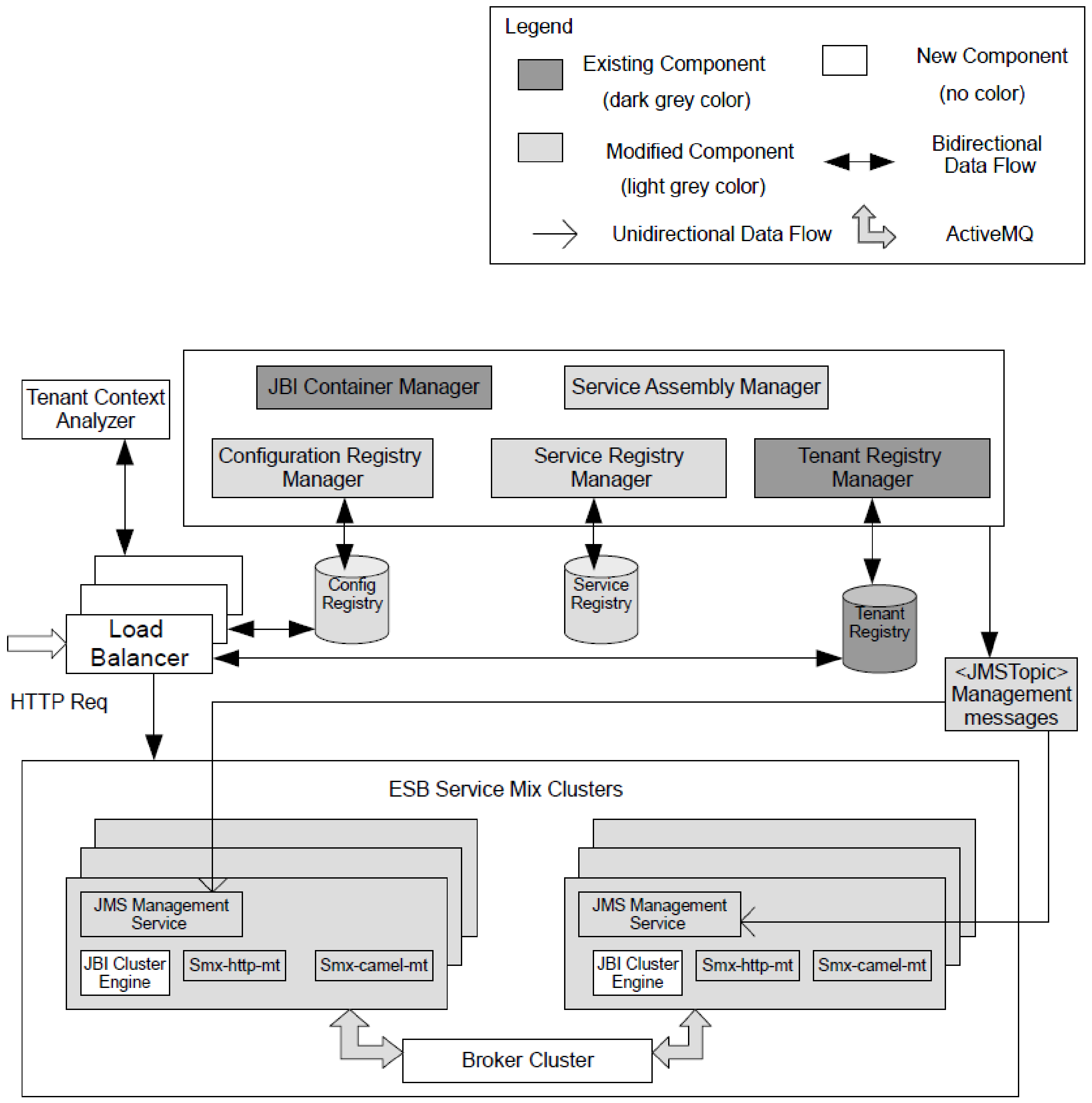


Figure 5.2 : Architectural overview of the design approach to enable horizontal scalability with Interconnected ESB instances.

Although the existence of a higher management overhead in the second scenario is evident, in long run the second approach will certainly yield better results. The number of endpoints supported in this second approach is higher as the endpoints are distributed across instances within the same cluster, instead of being replicated within a cluster. So we go-ahead with the implementation of the second approach.

5.2. Database Schemas

The JBIMulti2 resource layer is constituted by three main registries Tenant Registry, Service Registry and Configuration Registry.

5.2.1. Tenant Registry

The Database schema of the Tenant Registry is designed as a shared database because the same schema is used by all the ESB instances. The tenant user belongs to only one tenant group and he is represented by a primary key an UUID string. Tenant users and tenants can have arbitrary key-value pairs assigned to them, with each key-value pair only usable by subset of applications [Muh11]. When the LB receives a request it checks in Tenant Registry if the tenant exists in the system before processing the request. There are no modifications made on this component.

5.2.2. Configuration Registry

The Configuration Registry ensures data isolation between tenants by having a tenantID primary key on entity types as described in Chong et al [CCW06]. The *LoadBalancer* entity controls the JBI Cluster. It contains two attributes: loadBalancerIP and the maximum number of clusters that can be supported. A JBI Cluster contains many instances and hence for uniquely identifying each instance the use of an instance UUID is required. Since each instance hosts different BCs and SAs nodeIP is also associated with the container which is used for routing the request to a specific instance. The same BC can be deployed on any number of instances with in a cluster so we keep track of each binding component with each instance by associating the *JBI Component* entity with the instance UUID. A JBI Component consumes a specified number of ports which are unique for each instance. Therefore, a relationship between the *JBI component* and *Port* entities is mandatory, as the system must administer and manage the consumed and available ports in each node the ESB instance is deployed on. The *Port* entity also has a tenancy attribute which specifies whether the installed BC is multi-tenant or non-multi-tenant aware.

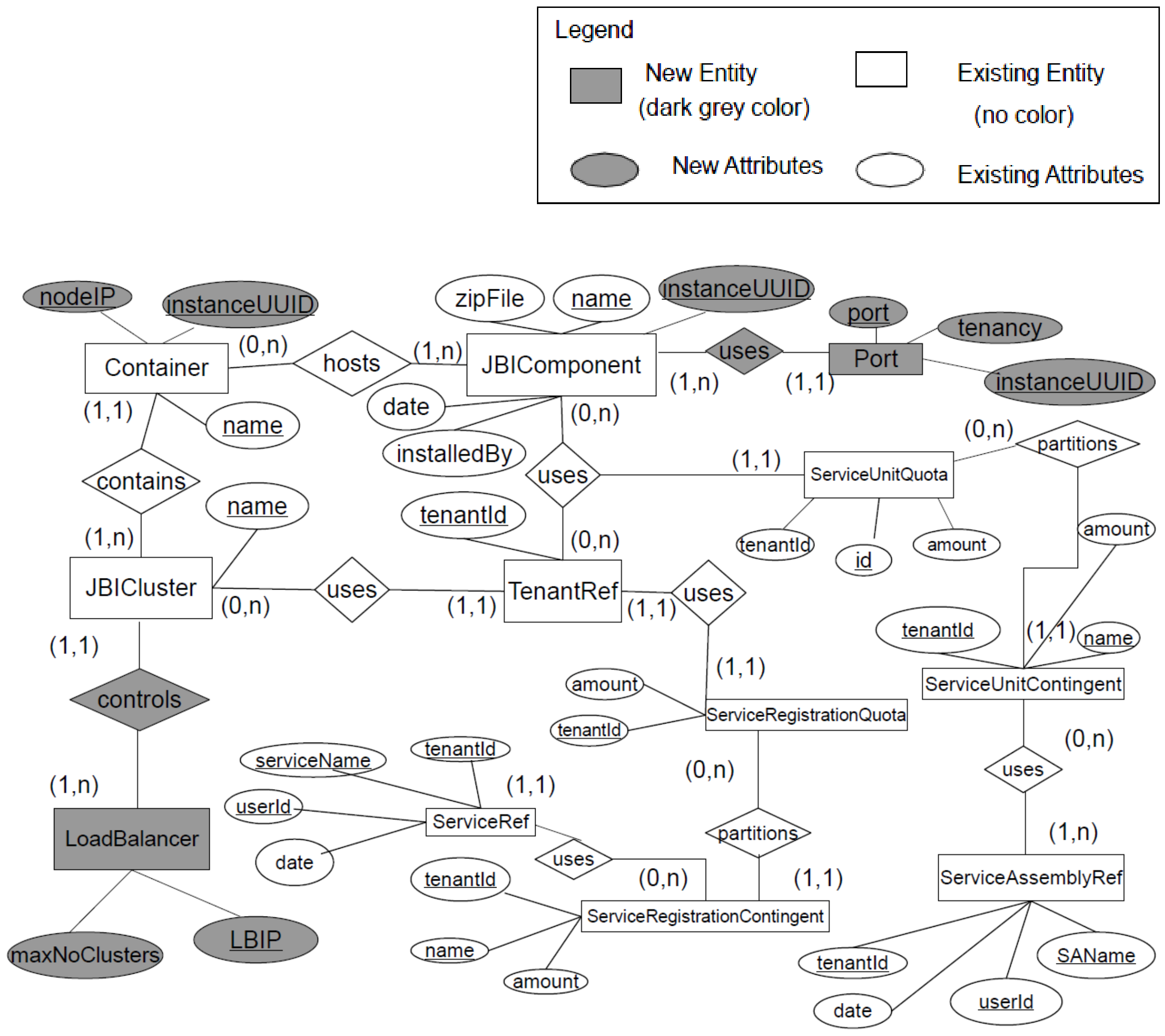


Figure 5.3: Entity-relationship diagram of the Configuration Registry extended from [Muh11]

5.2.3. Service Registry

In this thesis, the Service Registry developed in Muhler's work [Muh11] was extended to support the specification of the endpoint type associated with each deployed SA. A SA wraps the SUs where the end-point configuration is described. SUs contain the routing information configuration between endpoints. Hence, one service assembly identifies the tenant's user source data sources. The *Service* and *ServiceAssembly* entity contains tenantID and userID as the aggregated primary key also known as the tenant context. Service assembly ZIP files are stored as Binary Large Objects (BLOBs), whereas service WSDL files are stored as Character Large Objects (CLOBs) [Muh11]. The *EndPoint* entity has the endPointName as the primary key aggregated with tenant context information (combination of tenantID and userID). The endpointType refers to the kind of endpoint deployed on the SA such as either provider or consumer endpoint. The instanceUUID is a unique identifier which represents the ESB instanceID where this particular SA has been deployed. The *Endpoint* entity is introduced for keeping track of the information such as which instance hosts the endpoint for which user so that this information can be used by the LB to identify the ESB instance where a particular request should be sent to.

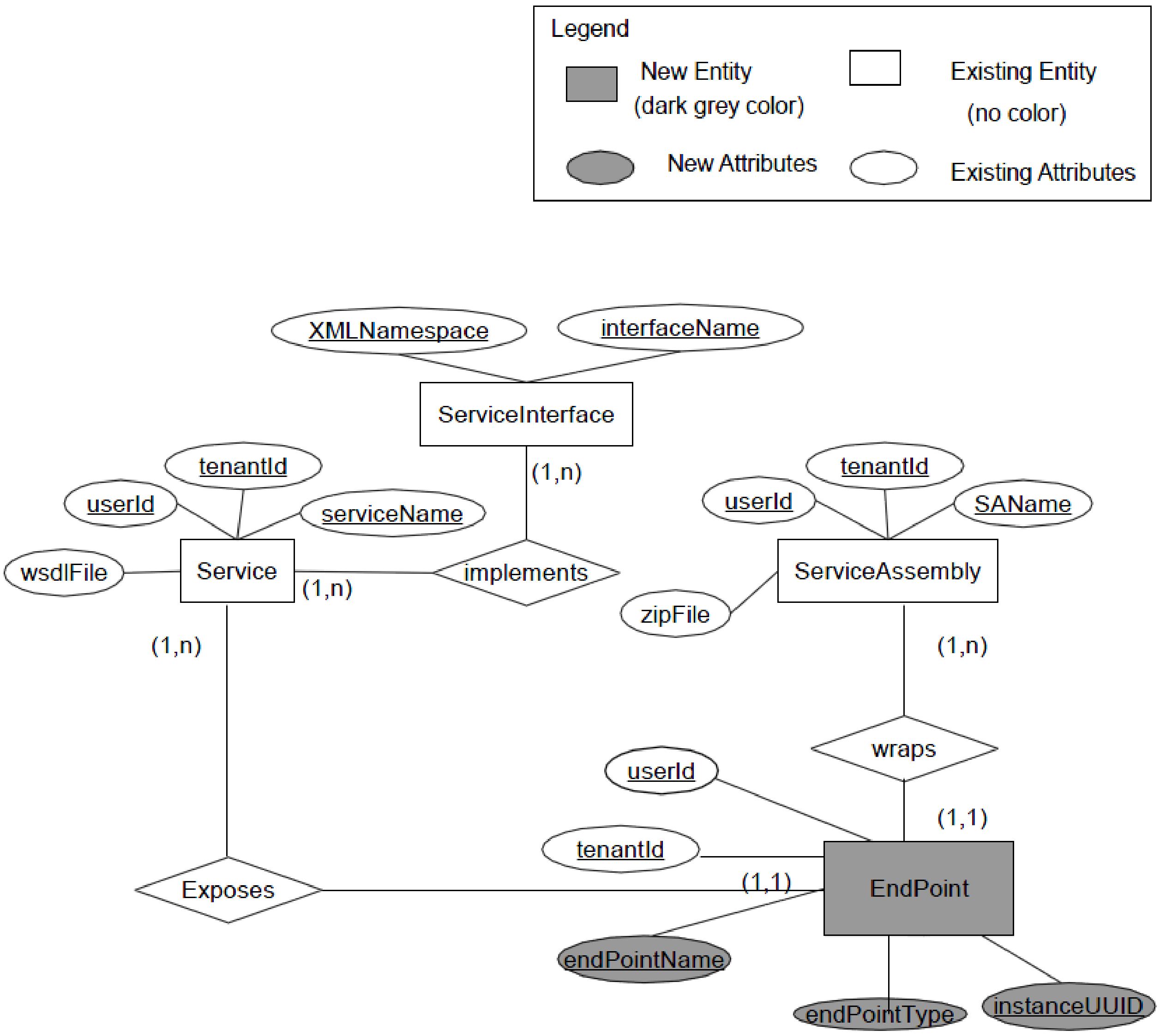


Figure 5.4 : Entity-relationship diagram of Service Registry extended from [Muh11]

5.3. ServiceMix Extensions

Muhler designed the JMSManagementService OSGi bundle for consuming the management messages coming from the web application since the original Apache ServiceMix does not come with a web service interface for managing JBI artifacts. Muhler also developed multi-tenant JBI components to introduce multi-tenant awareness by enabling message and data isolation. Here we discuss the extensions made to the JMSManagementService and BCs in this thesis.

5.3.1. Management Interface over Messaging

The JBIMulti2 application sends the management messages to a JMS topic. Management messages are used to deploy either BCs or SAs in a JBI Container. Muhler had implemented a JMSManagementService for ServiceMix which can receive and consume the management messages via a JMS topic. All the ServiceMix instances will listen to this topic. The management message contains target instance ID on which the message has to be installed or deployed. A configuration file is included on the ESB instance which contains the registered instanceID. Every ServiceMix after receiving the message checks with its configuration file if the instance ID matches then its proceeds with the message else the message is discarded. If the message is malformed or invalid it is sent to dead letter queue.

Messages sent to the JMSManagementService contain a combination of the following elements, each representing a management command [Muh11]:

- *JBI Component Install Command* instructs the ServiceMix instance to install all JBI components sent together as binary data if the instanceID in the message matches with the registered instanceID of a concrete ServiceMix instance.

- *JBI Component Uninstall Command* instructs the ServiceMix instance to uninstall all JBI components referenced by its name on the specified ServiceMix instanceIDs.

- *JBI Service Assembly Deploy* Command instructs the ServiceMix instance to deploy all SAs sent together as binary data if the instance ID in the message matches with the registered instanceID of a concrete ServiceMix instance. For each service assembly a tenant context element is mandatory.

- *JBI Service Assembly Undeploy Command* instructs the ServiceMix instance to undeploy all SAs referenced by the SA name on the specified instance IDs. For each service assembly a tenant context element is mandatory.

5.3.3. Multi-tenant aware JBI Binding Components

The original Apache ServiceMix BC [SMXa] for HTTP accepts locationURI where to listen to requests. This parameter is no longer valid in multi-tenant aware version of the component. A service endpoint comprises of service name and endpoint name. The two endpoint types *HttpConsumerEndpoint* and *HttpProviderEndpoint* implement the new interface *TenantEndpoint* that realizes method (*ensureMultiTenancy*()). This API is modified for applying the userID along with the tenantID to the endpoint configuration. The original *BaseXBeanDeployer* in the (servicemix-http BC) is extended to ensure multi-tenant awareness by the *XBeanDeployerMT*. This class checks the SU if there is a tenant context file available (generated by the JMSManagementService), and then modifies the endpoint to inject the tenant and user information into the endpoint dynamically before deployment.

```
/*
input: tenantId, tenantUri, userID, serviceLocalPart, endpointName,
configuredLocationUriPrefix
example:http://localhost:8193/tenantservices/54ed4755-5965-4b47-a121-d25907e29c04/
 ExampleService/98ed76-8954-321b4-7a121d6784e28b05/ep
*/
 locationUri ::= locationUriPrefix ( tenantId | tenantUri ) serviceEndpoint
 locationUriPrefix ::= "http://localhost:8193/tenantservices/" | configuredLocationUriPrefix
 serviceEndpoint ::= "/" serviceLocalPart "/" (userID) "/" endpointName

 /*
 input: tenantId, userID, serviceLocalPart, endpointName, configuredServiceNamespacePrefix
 example:{jbimulti2:tenantendpoints/54ed-47555-9654b-47a121d-25907e29c/98ed789-5432-
1b146-47a124e28b05}ExampleService:ep
 */
 serviceEndpoint ::= serviceName ":" endpointName
 serviceName ::= "{" serviceNamespacePrefix tenantId "/" userID "}" serviceLocalPart
 serviceNamespacePrefix ::= "jbimulti2:tenantendpoints/"| configuredServiceNamespacePrefix
```

Listing 5.1: Location URI (top) and service endpoint (bottom) replacing patterns for the multi-tenant HTTP BC servicemix-http-mt in Extended Backus–Naur Form (EBNF) extended from [Muh11].

6. Implementation and Validation

In this chapter we describe the challenges and problems encountered during the implementation phase to fulfill the requirements specified in Chapter 4 and the design approach presented in Chapter 5 of the system.

6.1. Nginx Server Configuration

Niginx is a reverse proxy server to load balance HTTP requests among backend servers. The main reason for choosing this server as Load Balancer is its extensibility features, as it easily supports development and integration of third party components. Furthermore, its embedded access support to back-end databases makes this reverse proxy suitable for supporting multi-tenant aware ESB clusters.

The LB receives the incoming http requests and can access the URI and it can access the HTTP headers and body and in our scenario it is suitable for accessing the SOAP message sent in HTTP body. The LB can control many ESB cluster and each of these clusters contain many ESB instances. For keeping track of each instance we have an unique identifier instance UUID, we can run many ESB instances on same server. Each of these ESB instances can host many BCs and to make sure that there are no port conflicts between the same BCs hosted on multiple instances we keep track of ports used by each instance.

The Nginx configuration file can be described as a list of directives organized in a logical structure. The directive *worker_process* defines the number of process. Nginx offers to separate the treatment of requests into multiple processes. The default value is 1, but it's recommended to increase this value if the CPU has more than one core. Besides, if a process gets blocked due to slow I/O operations, incoming requests can be delegated to the other worker processes [NginxH]. Within the Nginx we need to specify two blocks: one is *upstream* which defines the nodes with in the load balanced cluster and the second is http server config. The server directive we place within the upstream bock accepts several parameters such as *weight* and *max_fails* that influence the backend selection by nginx. The database upstream is used for defining the database credentials. Inside the *Server* block we define all the configurations and operations for the server. There is no clear separation between IP-based or name based (based on “Host” request header field) virtual server. The *listen* directive describes all addresses and ports that should accept connections for the server. We can have multiple servers inside a single configuration file. The *location* directive is used to set the configurations and execute the blocks depending on the URI path.

In the below configuration, the Nginx server accepts HTTP request and extracts the tenantID and userID from the URI. It then accesses the external backend data base in order to ensure that the tenant exists. After verifying that tenant is registered in the system, it retrieves the instanceID and the port to which this request has to be routed to. Finally it rewrites the HTTP request sending the request to destination.

```
 worker_processes  5;  ## Default: 1
 error_log  logs/error.log;
 pid        logs/nginx.pid;
 worker_rlimit_nofile 8192;
events {
    worker_connections  4096;  ## Default: 1024
  }

  http {
     include mime.types;
    default_type application/octet-stream;
     sendfile on;
    keepalive_timeout 65;
     ## Data base credentials
     upstream database {
       postgres_server 127.0.0.1 dbname=dbName user=postgres password=pass123;
     }

        server {
          listen 80;
          ## load balancer IP should be specified below
          server_name loadBalancerIP;
          set $instanceId defaultValue;
         set $port defaultPort;

          ## Extract tendID and userID from uri and send the request to
          ## backend database and get the instanceID for this user
          location ~ ^/tenant-service/(.*)$ {
               echo "\r";
               echo The request uri is: $request_uri;
           ## In the uri $2 represents tenantID and $4 represents userID
               echo_read_request_body;
               echo $request_body;

           postgres_pass database;
           rds_json on;
           postgres_query    HEAD GET  "SELECT * FROM EndPoint where tenantId = $1 and
           userId =$4";
           postgres_escape $instanceId $arg_instanceUUID;
           postgres_query    HEAD GET  "SELECT * FROM ports where instanceUUID =
           $instanceId";
           postgres_escape $port $arg_port;
     }  ## end of location directive

    rewrite ^ http://$instanceId:$port$request_uri?;
   }
 }
```

Listing 6.1: Nginx Http Configuration

6.2. JBIMulti2 extension

The JBIMulti2 application is extended to introduce new API in support of this thesis. To avoid giving clients direct access to business-tier components and to prevent tight coupling with the clients the Session Facade pattern with a superordinate AccessLayer is used [Muh11]. The business logic is implemented as the following stateful session beans: *SystemAdminFacadeBean*, *TenantAdminFacadeBean*, and *TenantOperatorFacadeBean* which provide a method for each use case described in section 4.3.

6.3. ESB ServiceMix Cluster

This section describes the implementation details for the clustering of the ServiceMix instances using JBI clustering technique and ActiveMQ. It also describes the implementation details for the modifications made to Multi-tenant Binding Component, Service Engine and JMSManagementService.

6.3.1. Multi-tenant Binding Component and Service Engine

The Servicemix-http-mt inserts the tenant UUID to the endpoint configuration to distinguish from tenants of other organizations [Muh11]. The *HttpConsumerEndpoint* and *HttpProviderEndpoint* implement the new interface *TenantEndpoint* that declares a method for applying the tenant UUID to the endpoint configuration. These two classes have been extended to insert userID as well into the endpoint URI. Since we might have the same BC installed on many instances running on the same node, it has to be ensure that each of the BCs use a separate port. Therefore, we have extended the *HTTPConfiguration* where we set the *tenantsLocationUri*. While installing the BC the user is prompted to enter one of the available ports on the instance and this port is set on the Servicemix-http-mt BC using the setTenantsLocationURI method.

6.3.2. JMSManagementService for Apache ServiceMix

The JMSManagementService is an OSGi bundle that is deployed in the Apache ServiceMix OSGi container and listens to a JMS topic for management messages from the JBIMulti2 web application [Muh11]. A configuration file is included in ServiceMix instance, which contains the registered instanceID for that instance. When a management message arrives we need to have a filtering technique to ensure that the message is intended to that particular instance only. To enable filtering on the JMSManagementService we set the instance id on the *InstallJBIComponents* and *DeployServiceAssemblies* class object so that the *InstallJBIComponentHandler* and *DeployServiceAssemblyHandler* on the JMSManagementServie can compare the instance id with the one included on ServiceMix instance and then decide either to install the component or discard it.

6.3.3. ActiveMQ Broker of Networks

For massive scalability the ESB instances inside the JBI cluster are interconnected using network connectors to form a broker network. The Broker network helps us to partition message destination to sub-domains of the network. The network connector can be considered as glue that defines the pathway between brokers and are responsible for controlling how a message propagates through the network [NwC]. An ActiveMQ consumer (within an ESB instance) is one who is connected to the broker network and this network keeps track of all the consumers and gets notified whenever a consumer gets connected or disconnected. Xml configuration file is used for specifying the connections and main parameters of this broker of networks. In XML, a network connector is defined using the *networkConnector* element, which is a child of the *networkConnectors* element.

The xml Configuration for network connectors is shown below.

```
<beans ...>
  <broker xmlns="http://activemq.apache.org/schema/core"
          brokerName="brokerA" brokerId="A" ... >
     ...
  <networkConnectors>
       <networkConnector name="linkToBrokerB"
          uri="static:(tcp://localhost:61002 ........)"
          networkTTL="3"
       />
    </networkConnectors>
     ...
    <transportConnectors>
         <transportConnector name="openwire" uri="tcp://0.0.0.0:61001"/>
      </transportConnectors>
  </broker>
</beans>
```

Listing 6.2: Network Connector configuration [NwC]

The *DiscoveryAgent* associated with the *NetworkConnector* will initiate setting up a bridge for each transport URI specified in the uri attribute of the *NetworkConnector*. For each of the network URI list if the bridge was not able to be successfully connected for some reasons then it will be attempted to recreate the bridge. So one broker establishes a network connector to another broker. As we expand this network of brokers we can build a complex network topology for more sophisticated message routing. We encountered some challenges before getting the broker network to work. We need to make sure that each of the ESB instance is using a different RMI registry port set in the *org.apache.management.karaf.cfg* configuration file. Furthermore, the Karaf instance name, the OSGi HTTP port should be unique for each of the ESB instances.

6.3.4. JBI Clustering

The JBI Clustering engine enables us to use the Apache ActiveMQ configured as a broker of networks to specify the endpoints of a clustered JBI environment. The clustering engine works in conjunction with the normalized message router (NMR), and uses Apache ActiveMQ in conjunction with specifically configured JBI endpoints to build clusters [JBICls]. Implementing clustering between JBI containers gives us access to features including load balancing and high availability.

Clustering enables JBI containers to form a network, and dynamically add and remove the containers from the network. It enables the JBI containers to handle the different tasks by spreading the workload across multiple containers.

The configuration for JBI cluster engine is show below. We need to change the default cluster and destination name for the Cluster eninge with the values of the properties set in the *org.apache.karaf.management.cfg* file.

```
<bean id="clusterEngine" class="org.apache.servicemix.jbi.cluster.engine.ClusterEngine">
  <property name="pool">
   <bean class="org.apache.servicemix.jbi.cluster.requestor.ActiveMQJmsRequestorPool">
      <property name="connectionFactory" ref="connectionFactory" />
      <property name="destinationName" value="${destinationName}" />
   </bean>
  </property>
  <property name="name" value="${clusterName}" />
 </bean>
  <osgi:list id="clusterRegistrations"
      interface="org.apache.servicemix.jbi.cluster.engine.ClusterRegistration"
      cardinality="0..N">
  <osgi:listener ref="clusterEngine" bind-method="register" unbind-method="unregister" />
  </osgi:list>
 <osgi:reference id="connectionFactory" interface="javax.jms.ConnectionFactory" />
      <osgi:service ref="clusterEngine">
          <osgi:interfaces>
               <value>org.apache.servicemix.nmr.api.Endpoint</value>
              <value>org.apache.servicemix.nmr.api.event.Listener</value>
             <value>org.apache.servicemix.nmr.api.event.EndpointListener</value>
             <value>org.apache.servicemix.nmr.api.event.ExchangeListener</value>
        </osgi:interfaces>
       <osgi:service-properties>
           <entry key="NAME" value="${clusterName}" />
       </osgi:service-properties>
     </osgi:service>
       <osgix:cm-properties id="clusterProps"
           persistent-id="org.apache.servicemix.jbi.cluster.config">
             <prop key="clusterName">${karaf.name}</prop>
            <prop key="destinationName">${servicemix.cluster.destination}</prop>
       </osgix:cm-properties>
  <ctx:property-placeholder properties-ref="clusterProps" />
</beans>
```

Listing 6.3: Cluster engine Configuration extended from [JBICls]

When using a JBI packaged SA, we must create a spring definition to register the endpoint as a clustered endpoint. The xbean-xml configuration file used in HTTP SOAP consumer endpoint service unit is shown.

```
<beans .....>
    <http:soap-consumer service="tx:httpSoapConsumer"
    endpoint="TaxiProviderHttpSoapConsumerEndpoint"
     locationURI="http://localhost:8163/httpSoapConsumer/
     TaxiProviderHttpSoapConsumerEndpoint"
     targetService="tx:httpSoapProvider"
     targetEndpoint="httpSoapProviderEndpoint"
     wsdl="classpath:service.wsdl"
    useJbiWrapper="false" />

    <bean class="org.apache.servicemix.jbi.cluster.engine.OsgiSimpleClusterRegistration">
       <property name="name" value="c1" />
       <property name="serviceName" value="tx:httpSoapConsumer" />
       <property name="endpointName" value="TaxiProviderHttpSoapConsumerEndpoint" />
    </bean>
</beans>
```

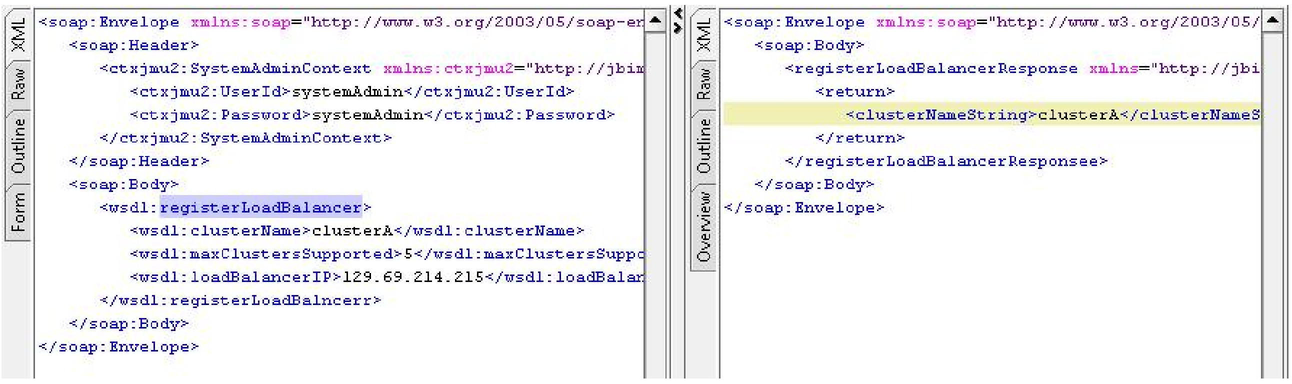
Listing 6.4: JBI packaged endpoint configuration

Some problems were encountered before we got the JBI clustering to work with ActiveMQ. We cannot use static and multicast network connectors together with in a JBI cluster. We have to use any one of them. Since we are using embedded broker network for clustering we should disable conduit subscriptions to ensure that use of message selectors is respected across different consumers listening on the cluster queue. Finally we need to ensure that all ServiceMix instances broker name is unique with in the cluster.

6.4. Validation

The validation is done on single virtual machine. We used an Echo web service as a backend shared service hosted on Apache Tomcat 7.0.23 [ApTom] for testing the SOAP HTTP multi-tenant configured clustered endpoints. JOnAS 5.2.2 [JOn] was used for hosting the JBIMulti2 web application. PostgreSQL 9.1.1 [PSQ] hosts the serviceRegistry database, tenantRegistry database and ConfigurationRegistry database. All these databases are accessed by the JBIMulti2 management application. Apache ActiveMQ 5.2.0 [AMQ] instance was used for creation of external topic *JbiMulti2.managementMessages* where the messages were delivered to from the JBIMulti2 web application. Apache ServiceMix 4.5.3 [SMXa] was used with along with internally run Active MQ 5.7.0 for forming a network of brokers. The JMSManagementService OSGi bundle was deployed on ServiceMix and it listens for management messages on the external ActiveMQ instance on the topic *JbiMulti2.managementMessages* to deploy the BCs and SAs. For clustering the JBI endpoints we used JBI Cluster engine 1.6.1 [JBIMvn]. For LB we used the Nginx 1.4.4 reverse proxy along with a third party postgres module which allows Nginx to directly communicate with PostgreSQL database.

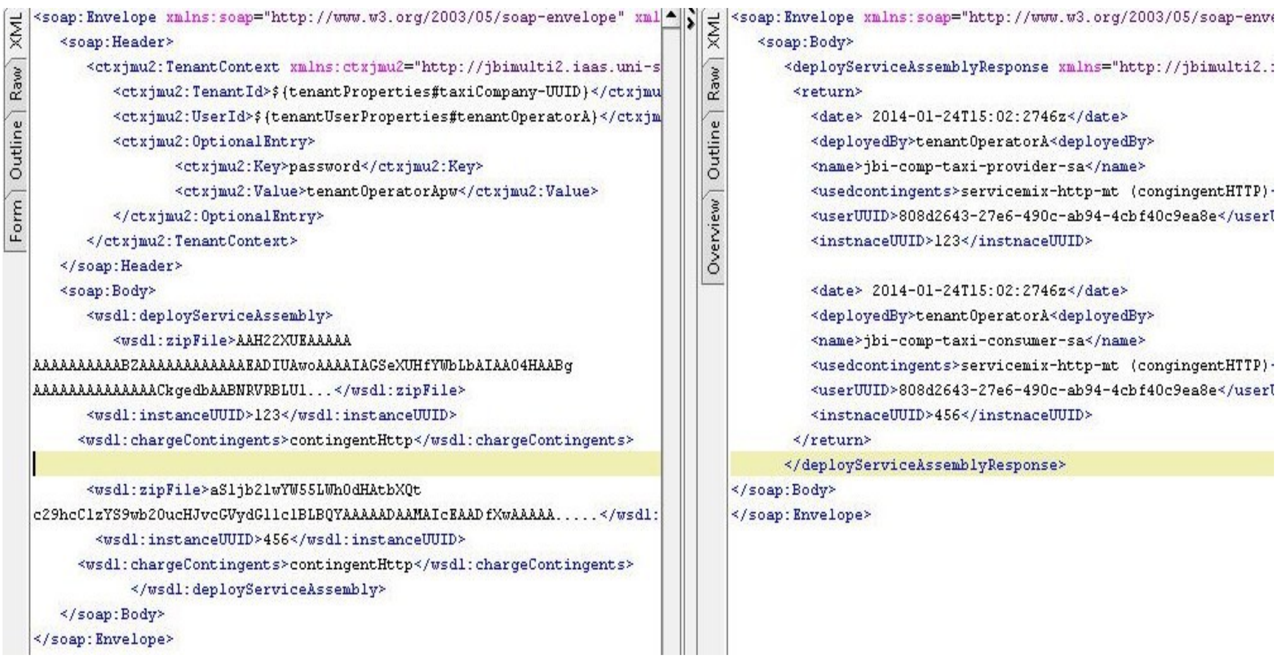
To interact with JBIMulti2 over Web Service API, we used the SoapUI 4.0.1 Web service testing tool. sample SOAP messages for the extended operations are depicted.



```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-en
   <soap:Header>
      <ctxjmu2:SystemAdminContext xmlns:ctxjmu2="http://jbim
         <ctxjmu2:UserId>systemAdmin</ctxjmu2:UserId>
         <ctxjmu2:Password>systemAdmin</ctxjmu2:Password>
      </ctxjmu2:SystemAdminContext>
   </soap:Header>
   <soap:Body>
      <wsdl:registerLoadBalancer>
         <wsdl:clusterName>clusterA</wsdl:clusterName>
         <wsdl:maxClustersSupported>5</wsdl:maxClustersSuppo
         <wsdl:loadBalancerIP>129.69.214.215</wsdl:loadBalan
      </wsdl:registerLoadBalncerr>
   </soap:Body>
</soap:Envelope>
```

```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/
   <soap:Body>
      <registerLoadBalancerResponse xmlns="http://jbi
         <return>
            <clusterNameString>clusterA</clusterNameS
         </return>
      </registerLoadBalancerResponsee>
   </soap:Body>
</soap:Envelope>
```

Figure 6.1 : Registering a load balancer with JBIMulti2 web application using SoapUI. On the left side we can see soap request and on right side the response message.



```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xml
   <soap:Header>
      <ctxjmu2:TenantContext xmlns:ctxjmu2="http://jbimulti2.iaas.uni-s
         <ctxjmu2:TenantId>${tenantProperties#taxiCompany-UUID}</ctxjmu
         <ctxjmu2:UserId>${tenantUserProperties#tenantOperatorA}</ctxjm
         <ctxjmu2:OptionalEntry>
               <ctxjmu2:Key>password</ctxjmu2:Key>
               <ctxjmu2:Value>tenantOperatorApw</ctxjmu2:Value>
         </ctxjmu2:OptionalEntry>
      </ctxjmu2:TenantContext>
   </soap:Header>
   <soap:Body>
      <wsdl:deployServiceAssembly>
         <wsdl:zipFile>AAH22XUEAAAAA
AAAAAAAAAABZAAAAAAAAAAAAAEADIUAwoAAAAIAGSeXUHfYWbLbAIAA04HAABg
AAAAAAAAAAAAAACkgedbAABNRVRBLU1...</wsdl:zipFile>
      <wsdl:instanceUUID>123</wsdl:instanceUUID>
     <wsdl:chargeContingents>contingentHttp</wsdl:chargeContingents>

      <wsdl:zipFile>aSljb2lwYW55LWh0dHAtbXQt
c29hcClzYS9wb20ucHJvcGVydGllclBLBQYAAAAADAAMAIcEAADfXwAAAAA.....</wsdl:
       <wsdl:instanceUUID>456</wsdl:instanceUUID>
     <wsdl:chargeContingents>contingentHttp</wsdl:chargeContingents>
           </wsdl:deployServiceAssembly>
   </soap:Body>
</soap:Envelope>
```

```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-enve
   <soap:Body>
      <deployServiceAssemblyResponse xmlns="http://jbimulti2.:
       <return>
         <date> 2014-01-24T15:02:2746z</date>
         <deployedBy>tenantOperatorA<deployedBy>
         <name>jbi-comp-taxi-provider-sa</name>
         <usedcontingents>servicemix-http-mt (congingentHTTP)-
         <userUUID>808d2643-27e6-490c-ab94-4cbf40c9ea8e</userU
         <instnaceUUID>123</instnaceUUID>

         <date> 2014-01-24T15:02:2746z</date>
         <deployedBy>tenantOperatorA<deployedBy>
         <name>jbi-comp-taxi-consumer-sa</name>
         <usedcontingents>servicemix-http-mt (congingentHTTP)-
         <userUUID>808d2643-27e6-490c-ab94-4cbf40c9ea8e</userU
         <instnaceUUID>456</instnaceUUID>
       </return>
      </deployServiceAssemblyResponse>
</soap:Body>
</soap:Envelope>
```

Figure 6.2: Deploying consumer SA on one instance and provider SA on another instance. With Clustering the provider SA end point configuration will be available on the consumer instance.

7. Outcome and Future Work

This master thesis contributes to another building block for a full-fledged PaaS Cloud Platform. In Chapter 2 we have presented relevant fundamentals like Cloud Computing, SOA, ESB, Load Balancing and Scalability patterns. After an investigation on the previous work on developing a scalable load balanced architecture in the Cloud we have introduced concepts for enabling horizontal scalability on a multi-tenant aware ESB in chapter 4. For scalability concepts and approaches we particularly focused on distributing the load across multiple interconnected ESB instances. For this purpose, the JBI clustering technique along with ActiveMQ broker of networks were used and adapted. A use-case analysis has brought out a specific management concept for multi-instance ESB administration and management. Together, these concepts target the fourth level of SaaS maturity model where the tenants share set of application instances [CC06].

These concepts have led to a system design in chapter 5 that describes a detailed architecture for managing multiple instances of ESB. Additionally extensions to JBIMulti2 component [Muh11] were done for installing JBI BCs and deploying SAs to individual ServiceMix instances to support this scalable architecture. Nginx proxy server is as a solution for the proposed load balancing cluster.

Due to time constraints extensive performance test against benchmarking could not be included in this thesis, and therefore are proposed as future work. Furthermore, automation and dynamic deployment of ESB instances using Cloud infrastructure framework such as Chef [Chf] can be also part of future work. To further enhance the scalability of the system vertical and diagonal scalability patterns can be introduced. In the future, the management application should get informed if the management tasks are successfully executed by Apache ServiceMix, currently there is a dead letter queue for unprocessed messages [Muh11]. There also should be an implementation for the web-based graphical user interface for managing the JBIMulti2 web application.

Bibliography

[AAG] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. *A view of cloud computing. Commun. ACM* 53, 4 (April 2010), 2010.

[ALB] The Apache Software Foundation. *Load Balancing Algorithms.* http://camel.apache.org/load-balancer.html

[AMQ] The Apache Software Foundation. *Apache ActiveMQ.* http://activemq.apache.org/.

[AMQinA] B.Snyder, D.Bosanac, R.Davies. *ActiveMQ in Action.* Manning, 2011.

[AMQMid] Puppet Labs. *ActiveMQ Middleware Configuration.* http://docs.puppetlabs.com/mcollective/deploy/middleware/activemq.html#settings-for-networks-of-brokers

[AMV] The Apache Software Foundation. *Apache Maven.* http://maven.apache.org/.

[APA11a] The Apache Software Foundation. *Apache Camel User Guide 2.7.0.* 2011. http://camel.apache.org/manual/camel-manual-2.7.0.pdf.

[APA11b] The Apache Software Foundation. *Apache Karaf Users' Guide 2.2.5.* 2011. http://repo1.maven.org/maven2/org/apache/karaf/manual/2.2.5/manual2.2.5.pdf.

[ApTom] The Apache Software Foundation. *Apache Tomcat.* http://tomcat.apache.org/

[BBG11] R. Buyya, J. Broberg, and A. Goscinski. *Could Computing Principles and Paradigms.* John Wiley & Sons, 2011.

[Bnw] The Apache Software Foundation. *Network of Brokers.* http://activemq.apache.org/clustering.html

[CC06] F. Chong and G. Carraro. *Architecture Strategies for Catching the Long Tail.* MSDN, 2006. MSDN. http://msdn.microsoft.com/en-us/library/aa479069.aspx

[CCW06] F. Chong, G. Carraro, and R. Wolter. *Multi-Tenant Data Architecture.* MSDN, 2006. http://msdn.microsoft.com/enus/library/aa479086.aspx.

[Cha04] D. A. Chappel. *Enterprise Service Bus: Theory in Practice.* O'Reilly Media, 2004.

[Chf] The Chef Community. *Chef Automation deployment.* http://docs.opscode.com/

[CiA11] Claus Ibsen and Jonathan Anstey. *Camel in Action.* Manning, 2011.

[ClusO] Oracle Corporation. *Configuring JBI Components for GlassFish clustering.* http://docs.oracle.com/cd/E19509-01/821-0826/jbi_clustering-about_c/index.html

[ClusWSO2] WSO2 Inc. *Clustering in WSO2.* http://docs.wso2.org/display/CLUSTER420/Overview

[ESBMt] S. Strauch, A. Andrikopoulos, S. Sáez, F. Leymann. *A Multi-tenant aware Enterprise Service Bus.* In: International Journal of Next-Generation Computing. Vol. 4(3), Perpetual Innovation Media Pvt. Ltd., 2013.

[FUS11] Red Hat Inc. Fuse ESB 4.4 – *Using Java Business Integration.* 2011. http://fusesource.com/docs/esb/4.4/jbi/.

[GCC] K. Stanoevska, T. Wozniak, S. Ristol. *Grid and Cloud Computing: A Business Perspective on Technology and Applications.* Springer, 2010.

[Gcca] Office of the Privacy Commissioner of Canada. *Cloud Computing.* http://www.priv.gc.ca/resource/fs-fi/02_05_d_51_cc_e.pdf

[Gom12] Gómez Sáez, Santiago. *Integration of Different Aspects of Multi-tenancy in an Open Source Enterprise Service Bus.* Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Studienarbeit Nr. 2394, 2013.

[HTLC] Ronald Schmelzer. *How to Think Loosely Coupled.* 2004. http://www.zapthink.com/2004/05/28/how-to-think-loosely-coupled/

[IBMSc] IBM Smart Cloud. *SCE high availability and load balancer workshop.*

[IBMScl] IBM Corp. *IBM Server Clusters Horizontal and Vertical Scaling.* http://pic.dhe.ibm.com/infocenter/brdotnet/v7r0m2/index.jsp?topic=%2Fcom.ibm.websphere.ilog.brdotnet.doc%2FContent%2FBusiness_Rules%2FDocumentation%2F_pubskel%2FRules_for_DotNET%2Fps_RFDN_Global478.html

[JBIMvn] The Apache Software Foundation. *JBI cluster engine.* http://mvnrepository.com/artifact/org.apache.servicemix.jbi.cluster/org.apache.servicemix.jbi.cluster.engine

[JBI01] Adrien Louis. *Use JBI Components for Integration.* 2006. http://www.javaworld.com/article/2071793/soa/use-jbi-components-for-integration.html

[JbI05] Sun Microsystems, Inc. *Java Business Integration 1.0.* August 2005.

[JBICls] Red Hat, Inc. *Clustering JBI endpoints.* http://access.redhat.com/site/documentation/en-US/Fuse_ESB_Enterprise/7.1/html/Using_Java_Business_Integration/files/ESBJBICluster.html

[JOn] OW2 Consortium. *OW2 JOnAS.* http://jonas.ow2.org/xwiki/bin/view/Main/.

[KK12] I. P. Kumar and S. Kodukula. *A Generalized Framework for Building Scalable Load Balancing Architectures in the Cloud.* International Journal of Computer Science and Information Technologies, Vol. 3 (1):3015 – 3021, 2012.

[LB] Citrix Systems, Inc. *Load Balancing.* http://www.citrix.com/glossary/load-balancing.html

[LVS] The Linux Virtual Server Project. *Job Scheduling Algorithms in Linux Virtual Server.* http://www.linuxvirtualserver.org/docs/scheduling.html.

[Muh11] D. Muhler. *Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management.* Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik. Diploma Thesis Nr. 3226, 2012.

[NwC] Red Hat Inc. *Network Connectors.* https://access.redhat.com/site/documentation/en-US/Fuse_ESB_Enterprise/7.1/html/Using_Networks_of_Brokers/files/FMQNetworksConnectors.html

[Nginx] Nginx Inc. *Nginx Architecture.* http://www.aosabook.org/en/nginx.html

[NginxH] Clement Nedelcu. *Nginx HTTP Server.* Packt, 2nd Edition 2013.

[OJ07] S. Ortiz Jr. *Getting on Board the Enterprise Service Bus.* Computer, 40:15–17, April 2007.

[OSG09] OSGi Alliance. *OSGi Service Platform: Service Compendium Version 4.2.* 2009. http://www.osgi.org/Download/Release4V42/.

[OSG11] OSGi^{TM} Alliance. *OSGi Service Platform: Core Specification Version 4.3.* 2011. http://www.osgi.org/Download/Release4V43/.

[PSQ] The PostgreSQL Global Development Group. *PostgreSQL.* http://www.postgresql.org/.

[Scal] Andre B. Bondi. *Characteristics of Scalability and their impact on performance.* In Proceedings of the 2nd International workshop on Software and Performance (WOSP'00). ACM.

[ScalDP] Kanwardeep Singh Ahluwalia. *Scalability Design patterns.* In Proceedings of the 14th Conference on Pattern Languages of Programs (PLOP'07). 2007. ACM.

[SMXa] The Apache Software Foundation. *Apache ServiceMix.* http://servicemix.apache.org/.

[SmxCl] The Apache Software Foundation. *Clustering in ServiceMix.* http://servicemix.apache.org/docs/5.0.x/user/what-is-smx4.html

[SSOA] TechTarget. *Service Oriented Architecture.* http://searchsoa.techtarget.com/

[TJ09] Tijs RadeMakers and Jos Dirksen. *Open-Source ESBs in Action.* Mannings, 2009

[Var11] J. Varia. *Architecting for the Cloud: Best practices.* 2011.

[WSPA] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more.* Prentice Hall, 2005.

Declaration

I hereby declare that the work presented in this thesis is entirely my own.

I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.

Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.

The electronic copy is consistent with all submitted copies.

Stuttgart, 31 January 2014

(Arun Kumar Hanumantharayappa)