Institut für Architektur von Anwendungssystemen (IAAS)

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3361

Enabling the Compatible Evolution of Services based on a Cloud-enabled ESB Solution

Sumadi Lie

Course of Study:	Informatik
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dr. Vasilios Andrikopoulos
Commenced:	16.07.2012
Completed:	15.01.2013
CR-Klassifikation:	D.2.7, D.2.12, H.4.0

Abstract

Software services are susceptible to changes because of the rapid growth and challenges in business environment. Business operations offered by service providers have to able to cope with various and countless service demands from service consumers. Such a case could also be experienced in cloud environment. As cloud platform, the Platform-as-a-Service (PaaS) platform allows application developers as tenants to deploy and configure their service artifacts in cloud infrastructure. A multi-tenant aware Enterprise Service Bus (ESB) as applications integrator is introduced to serve tenants in terms of management and administration. The purpose is to ensure data isolation between tenants.

The goal of this diploma thesis is to extend an open source multi-tenant aware ESB with service version control management framework so that the ESB can facilitate the version management of service providers and consumers in a transparent manner, and ensure service compatibility among tenants. The extension can be further decomposed in terms of management and administration, as well as message flows in versions inside the multi-tenant ESB.

Table of Contents

List of Figures ... 7
List of Tables ... 9
List of Listings ... 11
1. Introduction ... 13
1.1. Motivation Scenario ... 13
1.2. Problem Definition and Scope of Work ... 15
1.3. Research Design ... 16
1.4. Outline ... 16
2. Background and Related Work ... 19
2.1. Fundamental ... 19
2.1.1. Java Business Integration ... 19
2.1.2. Enterprise Service Bus ... 20
2.1.3. Service Registry ... 21
2.1.4. Service Versioning and Compatibility ... 22
2.2. State of Art ... 23
2.2.1. Service Versioning ... 23
2.2.2. Service Compatibility ... 24
2.2.3. Identification Model of Service Change ... 25
2.3. The Compatible Evolution Framework ... 26
2.3.1. Abstract Service Description Model ... 26
2.3.2. T-Shaped Change ... 28
3. Requirements and Specification ... 29
3.1. Service Version Control Management ... 29
3.2. Service Version Control Management Requirements for Multi-tenant aware ESB ... 31
3.2.1. Management Application ... 31
3.2.2. Multi-tenant aware ESB ... 35
4. System Design and Architecture ... 39
4.1. High-level System Overview ... 39
4.2. System Design ... 40
4.2.1. Use Cases ... 40
4.2.2. Database Schema ... 43

4.3. System Architecture 45
4.3.1. Functionalities of System Architecture 45
4.3.2. Communication between Components 47
4.3.2.1. Version Control Management 48
4.3.2.2. Message Processing Logic 51
5. Implementation 57
5.1. Foundational Technologies 57
5.2. Implemented Components 58
5.2.1. Version Registry 58
5.2.2. Java Method Definitions Extensions 61
5.3. Deployment Guide 63
6. Conclusion and Future Work 65
Bibliography 67
Declaration 71

List of Figures

Figure 1.1: Taxi Company and Taxi Service Provider integration by multi-tenant ESB [10] .. 14
Figure 2.1: Abstract Service Description Model [34] 27
Figure 4.1: High-level view of multi-tenant aware ESB with version control management ... 39
Figure 4.2: Use Case Diagram extending the uses cases for JBIMulti2 [6] 41
Figure 4.3: ER-Diagram of Version Registry 44
Figure 4.4: Building blocks of system architecture 45
Figure 4.5: Table representation of Version Registry 46
Figure 4.6: Activity diagram of replace service use case 49
Figure 4.7: Activity diagram of deploy service in parallel use case 50
Figure 4.8: Activity Diagram of deploy service assembly use case 51
Figure 4.9: Message flow for incoming message request 52
Figure 4.10: Sequence Diagram of processing compatible message 53
Figure 4.11: Sequence Diagram of processing incompatible message 54
Figure 4.12: Sequence Diagram of processing identical message 54
Figure 4.13: Sequence Diagram of processing undetermined message 55
Figure 5.1: Java package files describing the object-relation mapping 60

List of Tables

Table 2.1: Guideline for compatibility assessment [27] 25
Table 3.1: Tenant roles and corresponding actions for service registration and employment 30
Table 3.2: Mapping of use cases, roles, and databases for service-related information...........32
Table 3.3: Extension for enabling version control management based on tenant roles...........34
Table 4.1: Description of use case replace service 42
Table 4.2: Description of use case deploy service in parallel 43

List of Listings

Listing 2.1: Service versioning using XML Schema Definition 23
Listing 2.2: Compatibility Checking Function (CCF) algorithm [12] 28
Listing 3.1: Service endpoint replacing pattern for service version-aware HTTP BC in Extended Backus-Naur Form (EBNF)...37
Listing 4.1: Algorithm for message processing in Apache Camel..55
Listing 5.1: DDL of Version Registry 59
Listing 5.2: Example of query cache that holds information from the result query 60
Listing 5.3: Method definitions of replace and deploy service in parallel use cases 61
Listing 5.4: Method implementations of replace and deploy service in parallel use cases 61
Listing 5.5: Excerpt from Java Interfaces definition for required permission to perform replace and deploy service in parallel..62
Listing 5.6: Extension to CamelProviderEndpoint.java to support service version-aware......62
Listing 5.7: Extension to CamelConsumerEndpoint.java to support service version-aware...63

1. Introduction

A software service tends to change over time because of rapid growth and challenges in the business environment [1]. Service signatures, for example, might change their service data types, messages and operations in order to adapt to the new functionality that the service is offering. Such service changes could break the interaction between service consumer and service provider, and might have unexpected effects; thus it is important to be able to control and handle it properly.

Cloud computing, on the other hand, is a new computing paradigm that changes the way how computing resources such as software services and virtual servers are offered by service providers and used by service consumers [2]. The project 4CaaSt [3] is an EU-funded project which target a Platform-as-a-Service (PaaS) cloud platform. This project allows application developers as tenants to deploy and configure their own artifacts, for example, software services or libraries to the cloud infrastructure.

Enterprise Service Bus (ESB) solutions are at the heart of Service-oriented Architecture (SOA) [4]. An ESB integrates applications by taking benefits of the standardized technologies, loose coupling, and distributed deployment [5]. To realize the cloud capabilities, the ESB has to support multi-tenancy, and the works in [6] have identified and presented a multi-tenant ESB in management and administration aspects. This implies that each tenant who deploys the software and configuration artifacts should not be aware of other tenant presences, and moreover, the data produced by the corresponding tenant is completely isolated from other tenants [7].

In addition to the management and administration features supported by the multi-tenant ESB, the ESB has to also be equipped with a framework which enables the evolution of services as described previously. This allows the tenants to focus on new software services introduction and deployment while relieving them from the service version management itself. The multi-tenant ESB has to facilitate the version management of service providers and consumers in a transparent manner, and ensure service compatibility among tenants.

1.1. Motivation Scenario

The Taxi Scenario, which is a use case in the 4CaaSt project [8], will be adopted as a running example throughout this work. The Taxi Scenario describes a taxi booking service and it involves Taxi Company GUI, Taxi Transmitter GUI, Taxi Service Provider process, Google Maps Web Services Adapter, and Context Casting Context-Management Framework (C-CAST CMF) Adapter [9] as system components. All components of the taxi booking service are Web applications. The first two components can be deployed in a separate servlet container and referred as Taxi Company, while the rest of the components can also be deployed in another servlet container and will be referred as Taxi Service Provider (see Figure 1.1). The multi-tenant ESB will integrate both Web applications by allowing each application to communicate with an individual connectivity service provided by the ESB. Herein, the Taxi Company and Taxi Service Provider will be assumed as tenants in multi-tenant ESB because each of them might be a tenant of another.

The Taxi Scenario is illustrated as follows: Taxi Companies use taxi management software to offer taxi booking service to Taxi Customers. The customers can request taxi transportation by providing the pick-up location and the desired destination. The Taxi Company will then forward this information to the Taxi Service Provider which in turn requests information about nearby taxi cab location and taxi driver contact to service offered by C-CAST CMF, and distance calculation between the cab location and pick-up location to Google Map Web Services. By using this information the Taxi Service Provider will send requests to the Taxi Transmitters which are carried by taxi drivers. The taxi drivers that are near the pick-up location can confirm the taxi request by committing it to the Taxi Service Provider. Confirmation is then sent to the corresponding Taxi Company, and finally it reaches the Taxi Customer.

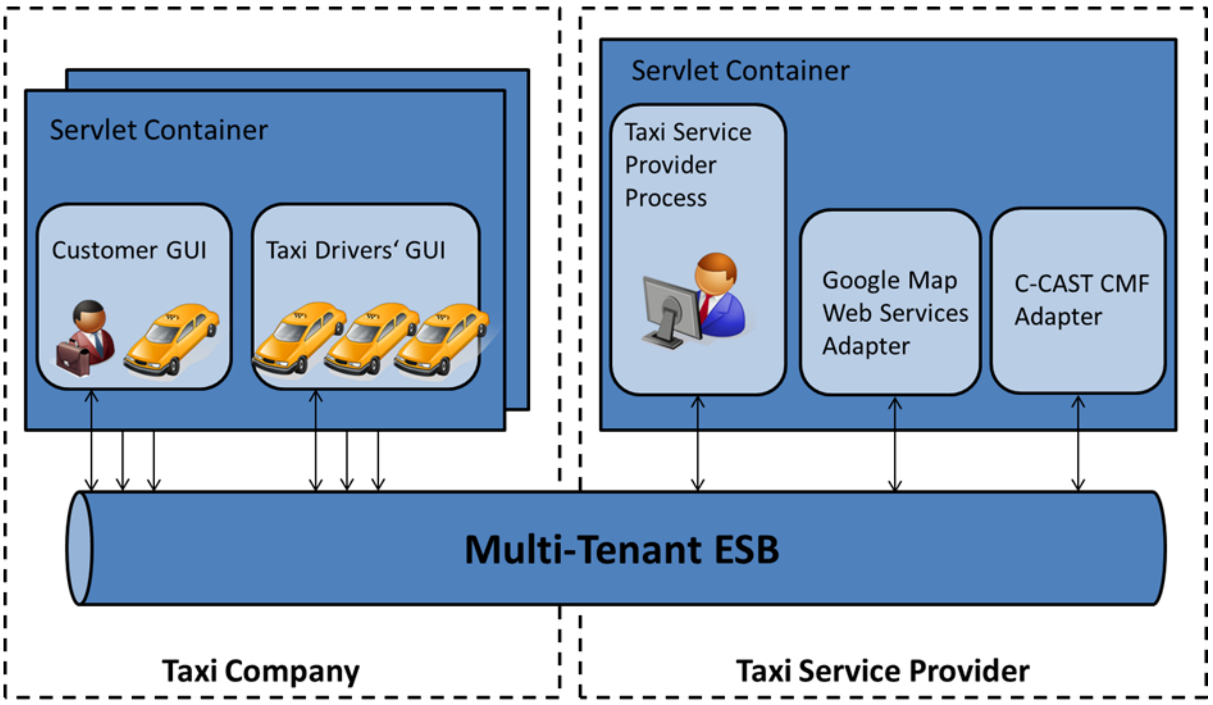


Figure 1.1: Taxi Company and Taxi Service Provider integration by multi-tenant ESB [10]

As described above, a tenant can be either a service consumer or producer. In case of the Taxi Scenario, Taxi Company and Taxi Service Provider might participate in the role of either service consumer or provider or even both using the multi-tenant ESB. The roles are explained in the following:

- Taxi Company acts as service provider: it registers its services with the multi-tenant ESB so that the services can be exposed to Taxi Customer.
- Taxi Company acts as service consumer: it uses services offered by the Taxi Service Provider i.e. to get any available and nearby taxis for Taxi Customers.
- Taxi Service Provider acts as service provider: it registers its services so that the services can be exposed to the Taxi Company.
- Taxi Service Provider acts as consumer: it consumes services provided by the Taxi Transmitter for taxi driver confirmation purpose.

In either case, whenever the Taxi Company or Taxi Service Provider wants to expose or consume services i.e. as service providers or consumers, they need to register the services with the system, or choose which services they want to use, respectively. In the following example, there are several possible service evolution paths that can be identified (for ease of readability, the service versions are given in format Vx.x):

1. Taxi Service Provider deploys service version $V_{1.0}$, Taxi Company uses the service of Taxi Service Provider $V_{1.0}$.
2. Taxi Service Provider might replace service version $V_{1.0}$ to $V_{1.1}$, this should be still compatible with $V_{1.0}$ so that the Taxi Company can still use $V_{1.0}$ or update to newer version $V_{1.1}$. If the Taxi Company wants to keep using service version $V_{1.0}$, the incoming requests will be adapted to service version $V_{1.1}$ as necessarily.
3. Assuming now the Taxi Company uses $V_{1.1}$ of Taxi Service Provider. In order to cover foreseen updates in the Taxi Service Provider, the Taxi Company upgrades unilaterally to its own version of service version $V_{1.2}$. As long as the service version $V_{1.2}$ of the Taxi Company conforms to service version $V_{1.1}$ of Taxi Service Provider then the communication between both parties can be still performed. Otherwise an error will be generated.
4. Taxi Service Provider might replace its service to version $V_{2.0}$. Assuming that service version $V_{2.0}$ breaks its consumers, the Taxi Company is required to update its service to $V_{2.0}$.
5. Taxi Service Provider might offer service version $V_{2.0}$ in parallel with service version $V_{1.1}$. Assuming that version $V_{2.0}$ breaks existing consumers, and the previous service version $V_{1.1}$ is still active. All requests from existing consumers i.e. from version $V_{1.0}$ and $V_{1.1}$ will be routed to the service version $V_{1.1}$ while service version $V_{2.0}$ targets new service consumers.

1.2. Problem Definition and Scope of Work

Based on the given roles and also the possible service evolution paths as described in the previous section, the main objective of this diploma thesis is to extend the open source ESB Apache ServiceMix [11] which has already been enhanced with multi-tenancy capabilities [6] so that the ESB can enable a service version control management framework based on the Compatible Evolution Framework defined in [12]. This extension is decomposed in terms of management and administration, as well as message flows in versions inside the multi-tenant ESB.

The service version control management framework concentrates on the service descriptions changes dimension as defined in [12]. This means that the evolution of service descriptions does not address directly changes to the actual service implementation. Similarly, the framework is aimed towards the Web services WS-* technological stack. In this case, Web Service Description Language (WSDL) documents in the service description level are the main focus on service versioning and service compatibility purposes. RESTful service is not in the scope of this work. Moreover, the communication and interaction among the service providers and consumers take place in the multi-tenant ESB. The ESB has to ensure that the requests and the corresponding responses are wired correctly to related service providers and consumers.

1.3. Research Design

The research design in this diploma thesis is to introduce a service version control management framework based on a theoretical framework defined in [12]. The theoretical framework provides a mechanism which guarantees a correct service versioning transition and service compatibility checking so that the service changes can evolve consistently and transparently. A multi-tenant ESB as applications integrator has to provide the capabilities for managing service evolution by implementing the framework discussed in [12].

To realize this goal, the requirements for service version control management framework in general should be investigated in the first case. This might include new use cases of how a tenant registers, replaces, or deploys in parallel her service versions in the management application. Moreover, a new data source, version registry, which maintains and holds the service version-related information, should also be introduced. Together with these fundamental specifications, the requirements for multi-tenant ESB can be examined. Any necessary APIs that need to interconnect the existing multi-tenant ESB and service version control management framework are proposed.

1.4. Outline

There are overall six chapters in this diploma thesis and each of them is structured and shortly described as follow:

- **Fundamentals, Chapter 2**—this chapter provides concepts and background related to service evolution, the corresponding infrastructure, its current state of art, and also the available theoretical methodologies.

- **Requirements and Specification, Chapter 3**—fundamental requirements for service version control management are inspected to solve common issues in service versioning. In addition, requirements for multi-tenant ESB are also examined. The specifications to satisfy those requirements in conjunction are then proposed.

- **System Design and Architecture, Chapter 4**—in the first section of this chapter, a high-level system overview is presented. Then it is followed by system design which defines how the existing management system is extended and the database schema of version-related registry is represented. The last section shows the entire system architecture, and it can be observed from its functionalities and communication between system components.

- **Implementation, Chapter 5**—any related technologies to implement the service version control framework are shortly discussed. The implementation for the system in terms of extending an existing multi-tenant aware ESB is performed.

- **Conclusion and Outlook, Chapter 6**—this chapter concludes the overall work and it gives outlook how this work could be further extended.

2. Background and Related Work

This chapter provides an overview of fundamental concepts and background related to this work. This information should give a better understanding and insight about which infrastructures or components are involved in the context of service evolution, what is the current state of art in service evolution, and which theoretical concepts and methodologies can be adopted to support this work.

The first section of this chapter examines the JBI, ESB and service registry as important foundations and infrastructures in service-oriented architecture world. The basic idea and definition of service versioning and compatibility are also covered in this section. The second section deals with the work contributions to service versioning and service compatibility. The last section covers the compatible evolution framework that act as a technology- and language-independent formal framework.

2.1. Fundamental

In this section background information on infrastructure and notions associated with evolution of services shall be presented. The understanding of these principles will serve as the basis for this work.

2.1.1. Java Business Integration

Java Business Integration (JBI) defines a standard-based architecture that enables the service containers interoperability, services connectivity, and services integration [13]. The JBI architecture allows integration artifacts to be constructed based on the components that are pluggable into the JBI environment. These components are referred to JBI components and can be classified into service engines (SEs) and binding components (BCs).

SEs can provide or consume services to/from other components in the JBI environment. For example SEs might have the capability of messages routing and transformation as services. In the other hand, BCs allow external system to communicate with the JBI environment by providing various communication protocols such as HTTP and JMS. BCs and SEs do not communicate directly. Instead, they communicate through a Normalized Message Router (NMR). All message exchanges that flow inside the JBI environment are normalized messages i.e. a XML payload and meta-data.

The JBI specifications also specify a management framework that allows the system administrator for an instance to install JBI components in the JBI environment, and deploy service configuration artifacts on the components through Java Management Extensions (JMX). The artifact that can be deployed into running BCs or SEs is referred to a service unit. A group of service units can be packaged together and can create a service assembly. Once a service assembly is deployed into JBI container, each of the service units will be deployed to its specific SE or BC.

2.1.2. Enterprise Service Bus

According to Rademakers et al. [14], there are several reasons why applications integration using an enterprise service bus (ESB) should be taken into account. The first reason is that in a large system landscape it is nearly impossible to have a homogeneous system because each existing application has its own way to perform the business processes by implementing different technologies and protocols. In order to communicate between any two applications a custom interface has to be built on each side to translate a data format that both of them can process. Such point-to-point integration would not make any sense in a large system environment since the number of connections of various applications will grow rapidly as the system grows. This only increases the overall system complexity and also makes the system difficult to scale. Another reason is to minimize the total cost of system maintenance. Applications' maintenance in a traditional point-to-point architecture can become very cumbersome and expensive. Therefore, the introduction of ESB as a central integration solution could help to alleviate the system management task and cost.

An ESB is an open standard-based message backbone designed to enable the implementation, deployment, and management of distributed service-oriented architecture (SOA) [Papazoglou08]. In its essence, the ESB has to ensure that it provides interoperability and connectivity among applications since it has to deal with different protocols, for example HTTP, JMS, and SOAP [5]. The ESB provides a layer abstraction of system integration which enables system developers to fully focus on service design and development.

As a SOA centerpiece, ESB must implement or support some core functionalities. In the following the core functionalities that are closely related to the diploma thesis will be discussed briefly [14], [15].

Message Transformation

It is very common that a data format of source application differs from the data format of target application. For the communication to take place properly a message from the source application must be transformed into a message format that is understood by the target application before it can be forwarded. The message transformation is performed based on open standards such as *Extensible Stylesheet Language Transformation* (XSLT) and XPath.

Message Enhancement
The message transformation described above can perform its jobs by cooperating with message enhancement. After a source message is transformed to target data format, one might need to add some additional information before sending the message out to destination. This is desirable because the target destination might require information which is not available from the source message. An external source e.g. a database could provide such additional data.

Intelligent Routing
Message routing is another important functionality that an ESB must support. Its task is to deliver requests from service consumers to correct service providers, and then forward responses back to the service consumers. There are some criteria how messages are routed. Content-based routing is a routing capability that sends messages based on their contents. Another example is message routing that routes a specific message to several destinations. In this case the service consumers subscribe to a certain topic and receive responses based on the topic.

Service Management
In a rapid changing SOA environment it is necessary to somehow govern and handle the environment in a proper way. Services that reflect business tasks need to be managed in order to ease service discovery and reuse. *Service registry* as a fundamental part of ESB gives a great deal of service management. This topic will be reviewed again in next section in greater details.

2.1.3. Service Registry

Service registry is a place where all information about services at runtime is stored. Its historical background comes from one of the main roles of the well-known SOA triangle, service broker. In SOA triangle, a service provider registers its services to a service broker. A service consumer that wants to use a particular service can find the related information in the registry. By using this information the service consumer can now locate and call the service [15].

According to Daigneau [16], a service registry might contain data types and messages definitions which are defined by XML Schema Definition Language. Moreover, WSDL files that identify the input and output message types, binding protocols, and addresses of services are also stored in the registry.

A mature service registry has several important responsibilities in SOA architecture. It has to be able to support the reuse of Web services and ease the communication between service providers and service consumers. More importantly it should have the capability to deal with the evolving SOA, that is, it maintains the evolution of services [17].

2.1.4. Service Versioning and Compatibility

In a SOA environment, services grow and evolve over time to reflect new business requirements which could drive existing services to be changed or modified. This might result in new variant of services, that is, the original services with upgraded or customized functionality. The creation and management of existing and new releases of services is called service versioning [18].

An upgraded service could be either compatible or incompatible with its older version of services. It means whether or not the new service version could be understood and processed by existing service consumers. There might be a possibility that the new version introduces e.g. different data types than the previous one causing the existing service consumers to break. Moreover, service compatibility can be divided into backward compatibility and forward compatibility [19]. Backward compatibility means a new version of service provider can be introduced without breaking the existing service consumers, while forward compatibility means a new version of service consumer can be carried out without breaking the existing service producer. If both conditions are fulfilled then it is called full compatibility [1].

Service versioning can appear either in service interface or service implementation [1]. Service interface versioning deals with service description, that is, metadata which defines the characteristic and interaction of services [4]. Meanwhile, service implementation versioning is versioning related to software code and its documentation, and can be managed by Software Configuration Management (SCM) techniques [20]. This diploma thesis will solely focus on interface versioning i.e. on WSDL files [21].

Based on the notion of service versioning and compatibility, old and new service interfaces should be managed in a way to avoid having too many active versions for the same service interfaces. On [12] it is proposed that the number of active versions should be kept as minimum as possible. When a compatible new service version is brought into production, the older version can be marked as deprecated and then withdrawn in a certain period of time as long as no service consumers are using it anymore. This could help reducing the high cost of service provision and management [15], [22].

2.2. State of Art

After a brief discussion of the basic notions of service versioning and service compatibility on Section 2.1.3, several existing approaches for them will be introduced. These approaches are classified into:

- How service versions are specified and which kind of methods can be adopted to implement them.
- How compatibility assessment is performed.
- How service consumers are aware of new service releases.

2.2.1. Service Versioning

Service versioning on service interfaces should support old and new version of services. One common way to realize this is by naming the service version with major and minor release sequence. For an example a service with version "2.3" indicates that the service has a major version of 2 and a minor version of 3. In addition, the version number is arranged in increasing order, and it is assumed that the higher the version number the newer the service version implies. Major releases are introduced when significant changes have occurred on the service interfaces that make the service consumers to break. Normally minor releases will not cause service consumers to break since the modifications are still compatible with the previous version [23].

Another way to name a service version is by using date. The following listing shows how to deploy versioning on the data structure defined by XML Schema Definition.

```
<types>
    <schema targetNamespace="http://example.com/2012/09/01/Schema.xsd"
             xmlns="…">
        <element name="GetWeatherRequest">
            …
        </element>
        <element name="GetWeatherResponse">
            …
        </element>
    </schema>
</types>
```

Listing 2.1: Service versioning using XML Schema Definition

This approach does not provide any information whether the release is compatible or incompatible with the previous version, but it can be used to validate the data types before messages exchange occurs [16].

There are several service versioning methods and each of them will be summarized as follows.

- A common method for implementing service versioning is to impose a new XML namespace. Unfortunately every introduction of a new namespace will break existing service consumers. This indicates that something new about the service or data types has been released, and the existing consumers that bind the previous namespace need to update their references. Applying a new XML namespace is always assumed to create a major version of services. If a minor release or compatible version wants to be achieved, version identifiers can be employed together with the namespace. In this case new XML namespaces will describe a major or incompatible service version, while version identifiers will indicate a minor or compatible change [24].
- As discussed on Section 2.1.2 an advanced service registry should be able to maintain the historical information of service interfaces. For an instance, how Universal Description, Discovery, and Integration (UDDI) as a public registry to support service evolution will be discussed. In UDDI, service versioning is realized through the extension of tModel [25]. tModel is a generic description of registered services in UDDI registry. The extension is carried out by providing a service version (major minor sequence) on instanceDetails data structure contained in tModel. Another approach is to have multiple tModels, that is, each tModel represents one service version [25].
- The service version graph approach proposed on [26] is another method to represent service versioning. One service interface is represented by exactly one version graph. The service version graph is a directed graph whose nodes contain versioning information and edges shows the relationship between service versions. The relationship creates a *successor* and *predecessor* role, that is, successor represents a changed service description of its predecessor. In the version graph, all successors refer to one original service. Service version graphs could also be stored in service registry.

In the work on service versioning by Yamashita et al. [27], versioning is applied only on a portion of WSDL/XML Schema such as operations and data types. These fragments are referred as *features*. The goal of this versioning approach is to create a version only to the features that have been changed either directly or indirectly influenced, not the entire service description for a better control and comprehension of interface changes impact. Indirect feature influenced mean the feature itself does not change but the other feature it depends on has been changed. For an example data type change from *int* to *double* might affect message element that is using the data type.

2.2.2. Service Compatibility

Service compatibility is an important notion in service evolution. It allows the introduction of a new service version while continuing supporting the old version of services. Maintaining backward compatibility defined on Section 2.1.3 always serves as a goal for service providers, and the term compatible should be by default regarded as backward compatible. On the other hand forward compatibility is harder to realize because it requires the ability to deal with a major change.

The works on service compatibility have been proposed and contributed to let service consumers of either the previous or the new version to successfully process the new message or old message format, respectively. References to the current works of service compatibility are summarized below.

1. Beside proposed service versioning method, Yamashita et al also define compatibility assessment that based on current guidelines or best practices as pointed in [28], [26], [29]. Table 2.1 shows the summary of these guidelines. Moreover the algorithm presented in [27] is an adaptation from [30] that takes two service features and evaluates them recursively based on guideline bellow for compatibility checking.

No.	Feature Type Change	Description	Backward Compatible
1	Add operation	Add new operation to service interface	Yes
2	Add type	Add new type to a new operation or a new type	Yes
3	Add type	Add new type to existing operation or type	No
4	Update type	Modification in description e.g. cardinality or order	No
5	Remove Type	Remove type dependency	No
6	Remove Operation	Remove operation dependency	No

Table 2.1: Guideline for compatibility assessment [27]

2. In order to avoid breaking changes introduced by new release of services, either service consumers or producers should be able to somehow ignore new contents in terms of new messages and data types that they do not understand. This is what the service design pattern *Tolerant Reader* should realize and it can be implemented on consumers or producers side. The *Tolerant Reader* must have the capability to differentiate and extract which information it can process and ignore the remaining one as long as the semantic of the service is not violated. In this way *Tolerant Reader* can preserve the backward or forward compatibility [16].

2.2.3. Identification Model of Service Change

There are several possibilities for service consumers to identify a new release of a service. The most straightforward way is that a service consumer itself has to recognize the new service version. This can be done by regularly checking the service registry whether the services being used are already deprecated. If that is the case then it implies that a new service is available regardless of whether it is a major or minor version [23], [31].

In the other way round, a service consumer will get a notification once a new version is ready to deploy. In [24] consumer using a certain service will get a first notification when a new service is released. After the old service is decommissioned the consumer will obtain another notification, e.g., a fault message indicating that the old version has already been taken out from production. Another example is to extend UDDI with service versioning functionality and allow service consumers to subscribe to an event related to the versioning information [32].

The approaches like client and notification models are deployed without any concerns whether or not new services break service consumers. However, transparent model ensures that as long as the new service is compatible with its previous version, the consumers will not be aware of the service modification and can keep using the existing service without any impact [33].

2.3. The Compatible Evolution Framework

The state of the art in service versioning and compatibility presented above is insufficient to build a solid framework supporting services evolution. For an example, the method that incorporates XML namespace and version identifier, when used intensively will create a maintenance problem. In addition this approach also requires service consumers to validate the version compatibility on their own. The approaches also depend heavily on particular standards e.g. WSDL and XML Schema Definition.

The compatible evolution framework proposed in [34] tries to overcome such drawbacks by providing a robust technology- and language-independent theoretical framework which guarantees the evolution of services can take place correctly and uniformly. The main purposes of this framework are to preserve the service compatibility and ensure a transparent version identification model so that every compatible change will have a seamless and consistent version transition.

2.3.1. Abstract Service Description Model

Service interfaces should not be specified restricted to a particular technology or language implementation because they have to be able to be deployed in any infrastructures or frameworks. The *Abstract Service Description* (ASD) meta-model allows a general representation of service descriptions as defined in [1]. The meta-model comprises three layers, namely structural, behavioral, and non-functional layer as shown in Figure **2.1** below. The structural layer describes service signatures of a service such as service data types, messages and operations. A behavioral layer shows how services behave when communication between service consumer and provider takes place. The non-functional layer is concerned with *Quality of Service* (QoS) represented by a set of policy constraints or assertions.

From here on, only the structural layer of ASD will be discussed since the behavioral and non-functional layers are out scope of this diploma thesis. As depicted in Figure 2.1, the structural layer contains *operation*, *message*, and *information type* constructs. These constructs are associated and can be mapped into a WSDL operation, message, and data type, respectively. Each construct is called an *element* and each element relates to each other based on their syntactical and semantic dependencies. Elements and their relationships are referred as *records*. Each element might have *properties* that describe its role in ASD, and *attributes* that contain a particular value. The property of an element can be assigned with a number of allowable predefined values e.g. *Datatype* of *valueType* property can be assigned with one of simple data types in XML Schema such as *int*, *double*, and *string*. *Document* data type refers to a complex type of XML.

As service interface representations, ASDs are also inevitable to being changed or modified. Every change results in a new version of ASD. So this reflects to the structural layer, that is, every record on this layer will be maintained and versioned accordingly. A *versioned ASD* contains a collection of the *versioned records* $\mathcal{R}$, and each record r is identified by a unique version identifier.

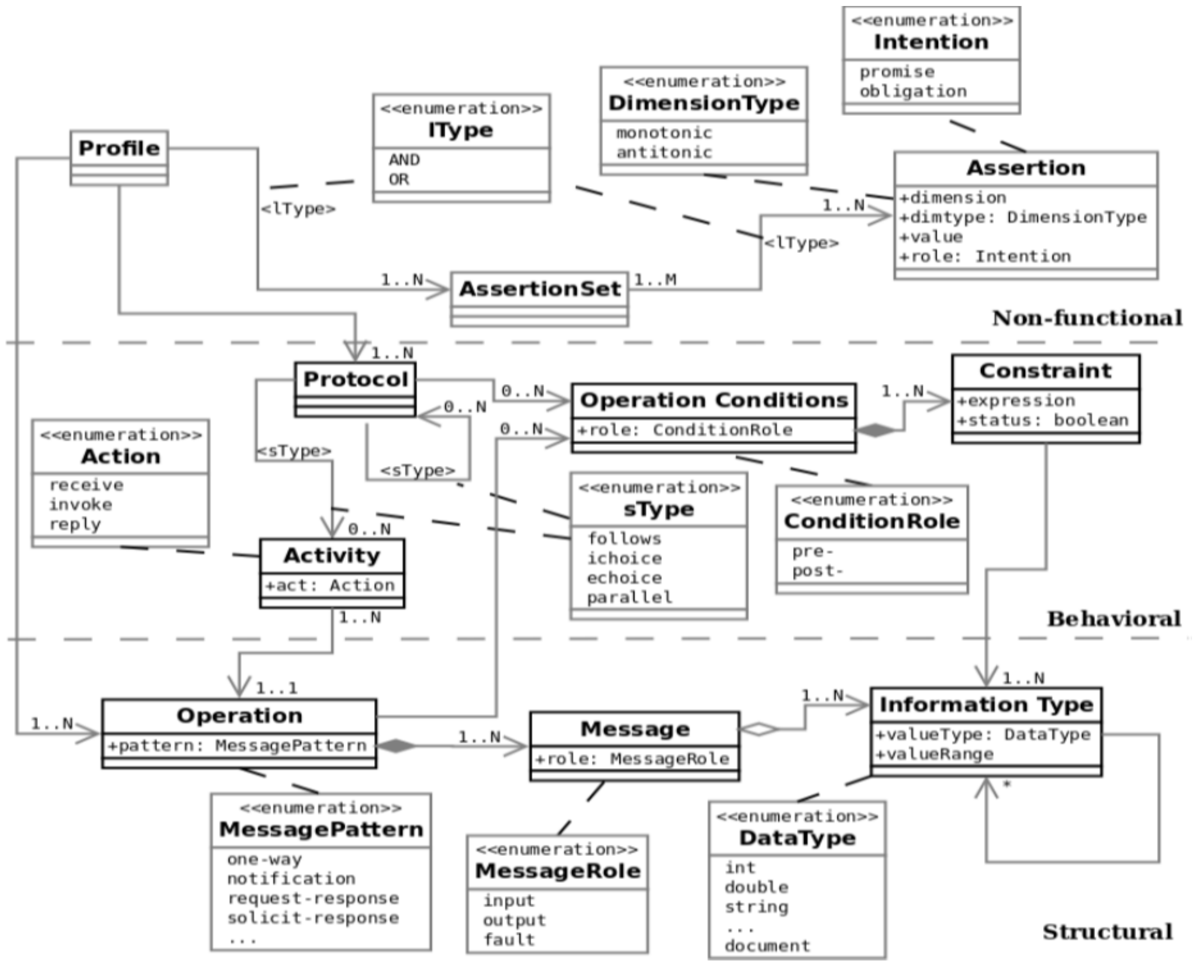


Figure 2.1: Abstract Service Description Model [34]

The compatibility among records can be defined using the help of *subtyping* relation. Subtyping evaluates whether one record is a subtype of another record, and this can be denoted as $r \leq r'$. Compatibility between service versions defined in [ABP12] allocates versioned records $\mathcal{R}$ into two proper subsets $\mathcal{R}_{pro}$ and $\mathcal{R}_{con}$ which represent a set of records of message producer and message consumer, respectively. Recall the service compatibility definitions described on Section 2.1.4service compatibility definitions between $\mathcal{R}$ and $\mathcal{R}'$ can be extended in terms of subtyping as follows:

- Backward compatibility: $\mathcal{R} <_b \mathcal{R}' \Leftrightarrow \forall r' \in \mathcal{R}'_{con}, \exists r \in \mathcal{R}_{con}, r \leq r'$.
- Forward compatibility: $\mathcal{R} <_f \mathcal{R}' \Leftrightarrow \forall r \in \mathcal{R}_{pro}, \exists r' \in \mathcal{R}'_{pro}, r' \leq r$.
- Full compatibility: $\mathcal{R} <_c \mathcal{R}' \Leftrightarrow \mathcal{R} <_b \mathcal{R}' \wedge \mathcal{R} <_f \mathcal{R}'$.

2.3.2. T-Shaped Change

The term compatibility has been further extended and classified by Andrikopoulos [34] into two dimensional scopes: *horizontal* and *vertical compatibility*. The horizontal compatibility or interoperability between services means that service versions can interoperate successfully with each other, either as a service producer or as a service consumer. On the other hand, the vertical compatibility or substitutability (provider's point of view) or replaceability (consumer's point of view) between services means that one service version can replace another version. The combination of the two definitions above results in so-called *T-shaped changes*.

In correlation to service compatibility in terms of subtyping, a change set $\Delta\mathcal{R}$ resulting into an ASD $\mathcal{R}'$ is considered a T-shaped change if and only if it results in a full compatibility of service description. In this case, the *Compatibility Checking Function* (CCF) algorithm defined below can be applied on $\mathcal{R}_{pro}$ and $\mathcal{R}_{con}$. The first *for* iteration (line 1 – line 5) and the second *for* iteration (line 6 - line 11) evaluate whether the changes on $\mathcal{R}'_{con}$ and $\mathcal{R}'_{pro}$ comply with the backward compatibility and forward compatibility, respectively. If the result returns the value *true*, it means the new service version is fully compatible with its previous one.

```
 1:    for all $r' \in \mathcal{R}'_{con}$ do
 2:        if $\nexists r \in \mathcal{R}_{con}, r \leq r'$ then
 3:            return false;
 4:        end if
 5:    end for
 6:    for all $r \in \mathcal{R}_{pro}$ do
 7:        if $\nexists r' \in \mathcal{R}'_{pro}, r' \leq r$ then
 8:            return false;
 9:        end if
10:    end for
11:    return true;
```

Listing 2.2: Compatibility Checking Function (CCF) algorithm [12]

3. Requirements and Specification

This chapter will discuss about requirements which are needed to build a service version control management system. In the first section, version control management requirements in general are introduced, and are used to address several common issues related to service versioning. In the second section, version control management in conjunction with multi-tenant ESB is defined. The required system specifications for the version control management are based on an existing multi-tenant ESB architecture [10].

3.1. Service Version Control Management

One of the main tasks of service version control management is to govern and maintain all service interface descriptions which were registered by service providers. This implies that the version control system will not only keep track of each service description registration with the ESB, but also of the corresponding service provider. In order to completely define the service version control management functionalities, there are several important considerations to take into account when dealing with service versioning:

1. Which service interface descriptions – the WSDL documents – are going to be managed and versioned.
2. When service compatibility assessment between service versions is carried out.
3. Where the service interface versioning information and history are stored.
4. How to uniquely distinguish the service versions.
5. What kind of possibilities that are offered by the underlying system so that a tenant (as a service consumer or service provider) can decide which service version that she wants to consume or expose, respectively.

In general, all service descriptions should be managed and versioned, that is, the ones which are registered with a system and exposed as service endpoints. From the motivation scenario (Section 1.1) this will include the service descriptions of Taxi Service Provider, Taxi Company, C-CAST CMF, and Google Maps Web Services as they are registered with the ESB. Although the service versioning is closely related to service compatibility assessment, both tasks are independent to each other. However, the compatibility assessment will be typically performed after a new service is registered and there are at least two services available for the compatibility evaluation. All service interface versioning and compatibility assessment information should be stored in a separate database that is managed by a service version control system, leaving the main system to fully concentrate only on its core functionalities.

Service versioning should not be a concern for service providers. It is not necessary to explicitly define for example a particular service version name or number (see Section 2.2.1) on the service descriptions to implement the service interface versioning. The version control management system will decouple this task from the service provider by adding service versioning information internally to differentiate between the services.

To describe the last point of the considerations above, it is necessary to recall the roles of tenant presented in the motivation scenario. A tenant can act either as a service provider, a service consumer, or both. The association between two tenants can be classified into the following relationship:

- *One-way relationship*: this is a typical case where a service consumer consumes services exposed by a service provider. However, it is not valid in the other way round. For an example Taxi Customer will use services from the Taxi Company to make a taxi order.
- *Reciprocal relationship*: a service consumer consumes services from a service provider, and for some reasons the service provider will also use services provided by the service consumer to accomplish the whole communication task. In this case each tenant has both roles as service provider and service consumer. From the Taxi Scenario, Taxi Company will use services from the Taxi Service Provider. To fulfill the taxi request from the Taxi Company, the Taxi Service Provider will utilize services from Taxi Transmitter which are parts of the Taxi Company services in order to communicate with the taxi driver.

Depending on in which role the tenant participates, she might have several possibilities with respect to how she registers and chooses the service versions. This will be summarized into the following table.

Role of Tenant	**Action**
Service Provider	When a tenant registers her services, she can define whether she wants to add a new service, replace an existing service, or deploy a new service in parallel to an existing one.
Service Consumer	1. When a tenant is added, she will be able to choose which service provider she wants to use. 2. A tenant can also decide for groups of her users to use a particular version of services from the service provider. 3. A tenant has an option to unilaterally move to another version of the service, even one that is not registered by the service provider. This allows tenant as the service consumer to evolve independently from the service providers.
Service Provider and Service Consumer	1. One-way relationship: a tenant implements actions as described for the service provider and consumer. 2. Reciprocal relationship: the same actions as in one-way relationship but with an additional task. After a tenant has registered her services and chose a service provider, she needs to explicitly notify the service provider to employ her services so that the communication between both parties can be established.

Table 3.1: Tenant roles and corresponding actions for service registration and employment

3.2. Service Version Control Management Requirements for Multi-tenant aware ESB

The work in [6] has identified the requirements for multi-tenancy on ESB (multi-tenant aware ESB) [10] in terms of administration and management level. The requirements for multi-tenant aware ESB involve a management application that connects to the ESB, and also the enhancement of the ESB itself. The requirements from [6] will be shortly introduced, and together with the fundamental functionalities specified in the previous section, the requirements for multi-tenant aware ESB which support service version control management shall be investigated.

3.2.1. Management Application

To enhance the ESB with multi-tenancy functionality, a management application is required. This application needs to maintain a group of tenants that want to access the ESB. Moreover there should be data sources that hold all important tenant-related information as well.

JBIMulti2 is management prototype developed in [6] to support multi-tenancy on ESB level. All multi-tenant functionalities required for management application are encapsulated into the JBIMulti2 system. The system implements *role-based access model* [35] that allows tenants to interact with the system in restrictive manner, and JBIMulti2 also assumes that tenants might be both service producers and consumers. JBIMulti2 is built on top of Apache ServiceMix [36] and connected to a collection of data sources.

In JBIMulti2 the actors play an important part in the system because the functionalities in the underlying system are performed by different roles. In addition, a set of databases in JBIMulti2 store different kind of information including tenant-, system configuration-, and service-related information. By carefully investigating the roles and the databases involved, it is possible to point out which system functionalities and information can be carried out in relation to service versioning.

Role-based Access

In JBIMulti2, all administration and management operations are restricted based on role access control. Each role has its own privileges to access the underlying system and data sources. The role-based access control can be classified into *system role* and *tenant role* which are represented by three actors of the system, namely system administrator, and tenant administrator and tenant operator, respectively. In the following the responsibilities of the JBIMulti2 system actors will be explained briefly:

- System administrator: she is responsible for the JBIMulti2 system and inherits all functionalities from tenant administrator and tenant operator. Her main tasks are to add tenants and provide them with resource usage.

- Tenant administrator: she can create other tenant administrator and tenant operator roles, and assign a set of permitted operations to them. She also uses the resource partitions provided by system administration to create service unit and service registration contingent for the tenant operators.
- Tenant operator: the service registration and service unit contingent created will be consumed by tenant operators to register services and deploy service assemblies to the system, respectively.

Databases in Management Application

A management application needs access to databases. In the case of JBIMulti2, there are three databases that the system currently relies on to store the system information, namely *Tenant Registry*, *Configuration Registry*, and *Service Registry*. Tenant Registry stores all information about the tenants and also the tenant users. Configuration Registry maintains configurations produced by the system administrator and tenant administrator. The third database, Service Registry keeps all service-related information. This includes service description documents and also the service assembly files. The JBIMulti2 accesses the databases using *Tenant Registry-*, *Configuration Registry-*, and *Service Registry Manager.*

The responsibilities of the JBIMulti2 actors described previously are translated into a list of use cases [37]. These use cases describe the interaction between the actors, the underlying system and the databases in specified permissions. Based on the use cases and the given system, the investigation about the operations supported or involved in the service-related information can be explored. This information is crucial because it gives indication which parts of the system could be extended for service version-aware purposes. The result is presented in the following table.

Use case	Role	Database
Integrate tenant registry	System administrator	Configuration-, Service-, Tenant registry
Delete tenant	System administrator	Configuration-, Service-, Tenant registry
Unregister JBI Container	System administrator	Configuration-, Service registry
Uninstall JBI component	System administrator	Configuration-, Service registry
Change JBI container cluster of tenant	System administrator	Configuration-, Service registry
Delete service unit contingent	Tenant administrator	Configuration-, Service registry
Delete service registration contingent	Tenant administrator	Configuration-, Service registry
View service assemblies	Tenant operator	Configuration-, Service registry
View service registrations	Tenant operator	Configuration-, Service registry
Register and unregister service	Tenant operator	Configuration-, Service registry
Deploy and undeploy service assembly	Tenant operator	Configuration-, Service registry

Table 3.2: Mapping of use cases, roles, and databases for service-related information

Based on the use cases shown in Table 2.1, the related key operations are listed below:

- Register service: the tenant operator can register services that she wants to expose by submitting the WSDL documents to the Service Registry.
- Unregister service: the WSDL files can be removed if they will not be used anymore.
- Deploy service assembly: when the configuration files in the service assembly are deployed to JBI components on the ServiceMix, a new service will be created on the components and it can be invoked through its service endpoints. The target service endpoint is also important to be specified for message exchange purpose.
- Undeploy service assembly: the created services are removed from JBI components.

All operations are implemented by the Service Registry Manager. The system administrator has also permissions to conduct the use cases since she inherits all functionalities from tenant operator. But from here on only tenant operator will be referred when dealing with these operations. For more comprehensive information about the access control and JBIMulti2 in general, please refer to the JBIMulti2 documentation [6].

Extensions to Management Application

This section will describe the extension for the management application to enable service version awareness based on the requirements in Section 3.1, and given the roles and use cases described above. The requirements are revisited again and the solutions are proposed after each requirement.

1. *Where the service interface versioning information and history are stored.*
   The current Service Registry database aims to keep the actual service descriptions only (services in runtime – see Section 2.1.2). For service interface versioning purposes, a new database – *Version Registry* – shall be introduced to maintain all versioning information and history.
2. *How to uniquely distinguish the service versions.*
   After service descriptions are registered by the tenant operator, a unique identifier will be assigned to each of them to distinguish them in the Version Registry.
3. *What kind of possibilities that are offered by the underlying system so that a tenant (as a service consumer or service provider) can decide on which service version that she wants to consume or expose.*
   The proposed approaches depend on which role the tenants play. Therefore, Table 2.1 is revisited again with extension or additional action, and is summarized into Table 3.3.

Role of Tenant	Action	Extension/Additional Actions
Service Provider	When a tenant registers her services, she can define whether she wants to add a new service, replace an existing service, or deploy a new service in parallel to an existing one.	There are two use cases that need to be added beside the existing ones, namely *replace* and *deploy service in parallel* use cases.
Service Consumer	1. When a tenant is added, she will be able to choose which service provider she wants to use. 2. A tenant can also decide for groups of her users to use a particular version of services from the service provider. 3. A tenant has an option to unilaterally move to another version of the service, even one that is not registered by the service provider. This allows tenant as the service consumer to evolve independently from the service providers.	There is no extension needed for the system, but some additional tasks should be implemented: 1. The tenant operator has to configure the service unit file by specifying the target service endpoint of the preferred service provider before deploying the service assembly. 2. In order to decide on which service versions to be adopted, one particular service has to be available at least in two versions i.e. they must be deployed in parallel in advance. Then based on policies the tenant users can be assigned to a certain service version. 3. The tenant operator has to register her service and apply the compatibility checking against the version(s) offered by the service provider. As long as both services are compatible, message requests can be created from the service description.
Service Provider and Service Consumer	1. One-way relationship: a tenant implements actions as described for the service provider and consumer. 2. Reciprocal relationship: the same actions as in one-way relationship but with an additional task. After a tenant has registered her services and chose a service provider, she needs to explicitly notify the service provider to employ her services so that the communication between both parties can be established.	1. The tenant operator employs the extensions and additional operations as defined for the service provider and consumer. 2. In this case, it is assumed that the tenant operators of the service provider should have knowledge who will consume her service and whose service she will consume. For instance, after the tenant operator has registered her service and deploy the service assembly, she should inform the chosen service provider at which location the WSDL document can be found. The service provider can then use the document to create message requests for communication purposes.

Table 3.3: Extension for enabling version control management based on tenant roles.

4. *Which service descriptions – the WSDL documents – are going to be managed and versioned.*
   All service descriptions that are registered and exposed by the ESB should be managed and versioned. However, the challenges are to separate the actual service descriptions (in Service Registry) from the service versioning information (in Version Registry) while preserving the existing service registration, unregister service operation, and also the new use cases (replace and deploy parallel) proposed in Table 3.3. In other words the Service Registry Manager should act as a front-end interface that deals directly with the tenant operators, and the version control management will sit behind the interface in order to monitor any service interface descriptions-related actions and add any service versioning information as necessarily.
5. *When service compatibility assessment between service versions is carried out.*
   The compatibility assessment might be performed in the background after the tenant operator has invoked the register or replace service operation.

3.2.2. Multi-tenant aware ESB

In addition to role-based access control and the registries described in the previous section, the work in [6] has also identified requirements for the Apache ServiceMix to support multi-tenancy on ESB level. The requirements should also work for ESBs that implement the JBI specifications. In the case of ServiceMix, JBI components (binding components and service engines) must be multi-tenant aware to ensure data isolation between tenants, thus the motivation will be on the service assembly/service units processing. When tenant operators execute the deploy service assembly operation, a management message will be sent from the management application to ServiceMix instance. Before the service assembly is deployed, there are two steps that need to be performed:

- The tenant context must be fetched from the management message. Information such as the tenant identifier, user identifier or tenant URI will be injected into the service assembly name, service unit names, and JBI deployment descriptor.
- The service units targeting different JBI components will create new services for each tenant by reading the tenant context and replacing the service name of the service endpoint with tenant identifier, user identifier, or tenant URI to generate multi-tenant aware JBI service endpoints. Therefore, the data isolation between tenants can be preserved.

Extensions to Multi-tenant aware ESB

Based on the information provide by Table 3.3, several use cases extensions or additional actions should be provided to the overall system to support version control management. The extensions for the use cases for management application have been covered in the previous section. In this section, the extension to the actual ESB will be explored.

Binding components (BCs) and service engines (SEs) are JBI components in a JBI environment. The BCs provide connectivity to existing external applications and can expose internal JBI services to external JBI environments. The SEs offer functionality to other components and can also consume services. Therefore, JBI components are the target extension in this case. To implement a message flow of the communication between service consumers and service providers, tenant operators have to define the service unit configuration files and deploy them as a service assembly.

There are two requirements needed to support version control management on multi-tenant aware ESB:

1. The most important of the ESB capabilities is message routing and this functionality is provided by a service engine. The routing engine of the multi-tenant aware ESB has to be aware of the versions of each incoming messages to route the messages to correct service endpoints. Hence, it should be provided with additional information such as tenant identifiers, user identifiers, target service endpoints, compatibility status between service versions, and version of expected and actual incoming message. The routing engine has to implement a *message interceptor*. Before any routing operations are executed, the message interceptor has to verify every incoming message with the information specified earlier to make a proper routing decision whether the message can be wired to an active compatible service version, or an error message should be generated. The compatibility information is important because it can be used as an indicator by the message interceptor whether the incoming messages are necessary to be transformed and routed to the target service endpoints despite of different service versions.
2. Besides the capability to internally transform endpoint URIs and JBI service endpoints to the ones that are multi-tenant aware, the multi-tenant aware ESB needs to be extended in order to create endpoints that have the functionality to interpret services in version (essential when services are deployed in parallel). However, the endpoint URIs that expose services to external JBI environment should not be enhanced with service version information. The rationale behind this is to release tenants from service version intricacies, and allow the transparency of service evolution. Nevertheless, service version awareness has to be enabled inside the JBI environment targeting the JBI service endpoints for outgoing message exchanges. This is because the routing capability needs to properly wire the messages to the correct service endpoints. This is important when service providers deploy services in parallel. To fulfill this requirement the multi-tenant aware ESB could be enhanced by adopting the same approach like the one for multi-tenancy i.e. by adding a unique service version identifier in management messages sent by JBIMulti2 when deploying service assemblies. Then the endpoint name of the outgoing JBI service endpoint can be extended by adding a service version identifier (see Listing 3.1).

```
/*
Input: tenantId, serviceLocalPart, endpointName, configuredLocationUriPrefix,
serviceId
Example:{jbimulti2:tenant-endpoints/tenantId}ExampleService:ep/service_version_id
*/
serviceEndpoint           ::=serviceName ":" endpointName (serviceId)
serviceName               ::="{"serviceNamespacePrefix tenantId"}"serviceLocalPart
serviceNamespacePrefix    ::="jbimulti2:tenant-endpoints/" |
                                configuredServiceNamespacePrefix
serviceId                 ::="/service_version_id"
```

Listing 3.1: Service endpoint replacing pattern for service version-aware HTTP BC in Extended Backus-Naur Form (EBNF)

The Listing 3.1 above explains how a service endpoint in multi-tenant HTTP BC *servicemix-http-mt* [6] can be extended to enable service version-aware. A Service endpoint is composed of a service name and endpoint name. In case of multi-tenancy, the service name of the service endpoint has been extended to include tenant context, thus, each tenant can have separate service endpoint. Similarly, in order to support service version-aware the new endpoint name has to be concatenated with a unique service identifier indicating a specific version of a service.

4. System Design and Architecture

This chapter proposes a system design and corresponding architecture to realize the system requirements defined in the previous chapter. First, a system overview of the service version control management based on multi-tenant aware ESB will be introduced. In the second section, a system design that reflects the system requirements is presented. The system architecture that represents the system functionalities and interactions will conclude this chapter.

4.1. High-level System Overview

In Chapter 3, the service version control requirements in conjunction with multi-tenant aware ESB have been covered. Based on the specified requirements there are three main components that build up the overall system. The system consists of the *Version Control Manager* (as an extension of JBIMulti2), the *Version Registry* database, and the *Multi-tenant aware ESB*. Figure 4.1 depicts a high-level view of the system and how each component is connected to each other.

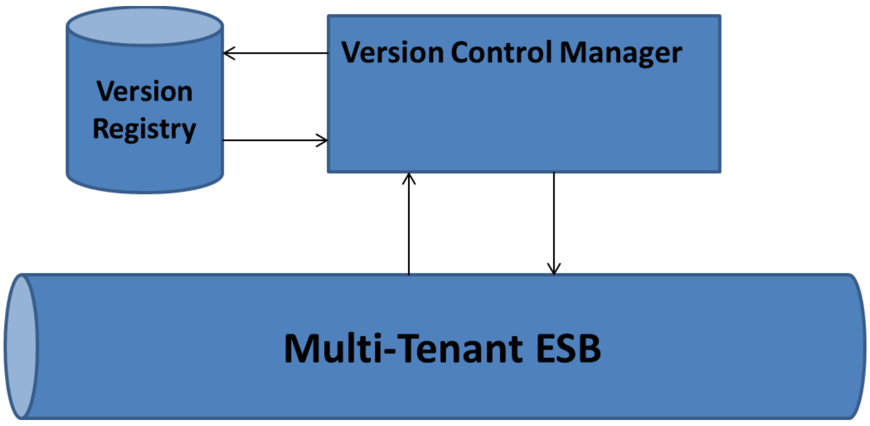


Figure 4.1: High-level view of multi-tenant aware ESB with version control management

Version Registry

As a data source the Version Registry should be able to provide the Version Control Manager and the ESB the service versioning information. The information contained in the Version Registry includes service version-related information, compatibility status, and tenant context. The database schema and the functionalities of Version Registry can be viewed in Section 4.2.2 and Section 4.3.1, respectively.

Version Control Manager
At its very basic functionalities the Version Control Manager should be able to retrieve data from the Version Registry, and update the multi-tenant aware ESB with the most current state of the service versioning information. The complete functionalities of Version Control Manager can be found in Section 4.3.1.

Multi-tenant aware ESB
Multi-tenant aware ESB must be able to forward a request message from a particular tenant to a service provider that employs a certain service version, and vice versa by using the information updated by the Version Control Manager.

Communication between components
The version control manager might retrieve service versioning information from Version Registry and then forward it to the ESB. This communication way is the most straightforward approach that can be applied. In the following several patterns are introduced to realize the communication between the components.

1. Communication between Version Registry and Version Control Manager.
    - A timer pattern can be implemented to poll the database from the Version Control Manager in interval basis.
    - Event listener in the Version Control Manager reacts whenever there is an update in the Version Registry.
2. Communication between Version Control Manager and multi-tenant aware ESB.
    - Any of the patterns introduced above can serve as a complement to publish-subscribe pattern [38]. The information retrieved from the Version Registry will be published to the multi-tenant aware ESB because it listens on the topic subscription from the registry.

4.2. System Design

The proposed system design is based on the design for JBIMulti2 management system. In the first section, the use cases as required for the version control purposes are given and represented as a use case diagram. The first section is followed by database schema design for Version Registry in Section 4.2.2.

4.2.1. Use Cases

In this section the related use cases of a tenant operator that were examined in Table 3.2 are put together and presented as one use case diagram. In addition to the existing operations there are also new use cases, namely the *replace service* operation and *deploy service in parallel* operation as stated in Table 3.3. The replace service use case has an include relationship with the existing use cases: the register and unregister service; while the deploy service in parallel use case also has an include relationship with register service operation. The use case diagram is shown in Figure 4.2, and the description of the replace and deploy service in parallel operation are given in Table 4.1 and Table 4.2, respectively.

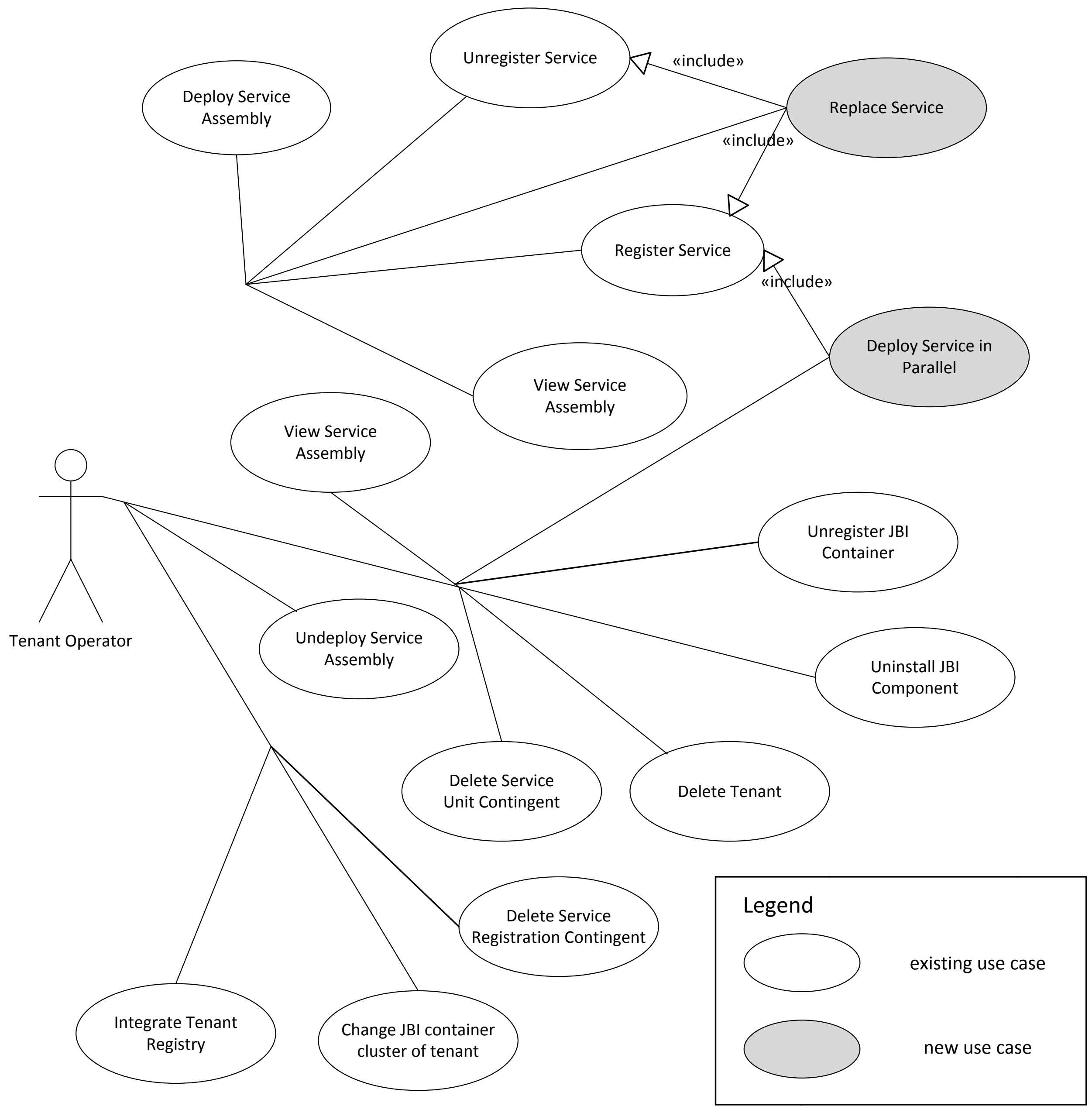


Figure 4.2: Use Case Diagram extending the uses cases for JBIMulti2 [6]

Name	**Replace Service**
Goal	The tenant operator wants to replace an old service version with a new service version using an existing service registration contingent.
Actor	Tenant Operator
Pre-Condition	The tenant operator has the permission to use the service registration contingent.
Post-Condition	The old service version is unregistered and the new service version is registered
Post-Condition in Special Case	The old service version is not unregistered and the new service version is not registered.
Normal Case	1. The tenant operator will instruct the system to unregister the old service version by removing it from Service Registry and then update the service state in Version Registry into decommissioned. 2. Tenant operator registers the new service version with the system by adding the WSDL document into Version Registry and Service Registry. Additional service versioning information including setting the service state to active is stored in the Version Registry.
Special Cases	1. The tenant operator has lost the permission to use the service registration contingent. a) The system shows an error message and aborts. 2. The system cannot finish the transaction with the Service Registry or Version Registry. a) The system shows an error message and aborts. 3. The file for service registration purpose does not exist. a) The system shows an error message and aborts. 4. The service description document is not a valid WSDL file. a) The system shows an error message and aborts.

Table 4.1: Description of use case replace service

Name	Deploy Service in Parallel
Goal	The tenant operator wants to deploy a new service version in parallel with an existing service version using an existing service registration contingent.
Actor	Tenant Operator
Pre-Condition	The tenant operator has the permission to use the service registration contingent.
Post-Condition	The new service version is registered and is available in parallel with the old service versions. The number of available service registrations is decreased by one for the used service registration contingent.
Post-Condition in Special Case	The new service version is not registered and cannot be deployed in parallel with the old service version.
Normal Case	1. The tenant operator will instruct the system to register the new service version by adding WSDL document into Service Registry and Version Registry. Additional service versioning information including setting the service state to active is stored in the Version Registry.
Special Cases	1. The tenant operator has lost the permission to use the service registration contingent. b) The system shows an error message and aborts. 2. The system cannot finish the transaction with the Service Registry or Version Registry. b) The system shows an error message and aborts. 3. The file for service registration purpose does not exist. b) The system shows an error message and aborts. 4. The service description document is not a valid WSDL file. b) The system shows an error message and aborts.

Table 4.2: Description of use case deploy service in parallel

4.2.2. Database Schema

Figure 4.3 shows an Entity Relationship Diagram (ER-Diagram) of the Version Registry. The ER-Diagram is composed of three entities and they will be defined as follows:

1. Service version: it is an entity that stores the versioning information for a service description. This entity contains several attributes.
    - Service version identifier: an identifier that is uniquely assigned to each service description. Universally Unique Identifiers (UUIDs) are used for this purpose.
    - Service name: a name of the service description. It is preferably related `WSDL:targetNamespace`.
    - Service status: a service might have one of the two states, namely active, and decommission.
    - Start time: time the service is being activated or in active state.
    - End time: time the service was decommissioned.

- Service interface current: WSDL file of actual service description.
- Service interface previous: WSDL file of previous service description.

2. Service Compatibility: it is represented as a recursive relationship between service versions and it describes the compatibility of the services. It also has an attribute of compatibility status: compatible, incompatible, identical and undetermined.
3. Tenant details: this entity contains the tenant and user identifier.

The entities of the Version Registry also build relationships between each other. A particular tenant user can only use at most one service version a certain point of time, but one service version can be adopted by many tenant users. One service version can be related to many other service versions, and vice versa. The attribute `serviceEndpoint` is an attribute resulting from the relationship between `service_version` and `tenant_details`. This attribute holds information about the enhanced outgoing endpoint name with service version identifier of the JBI components.

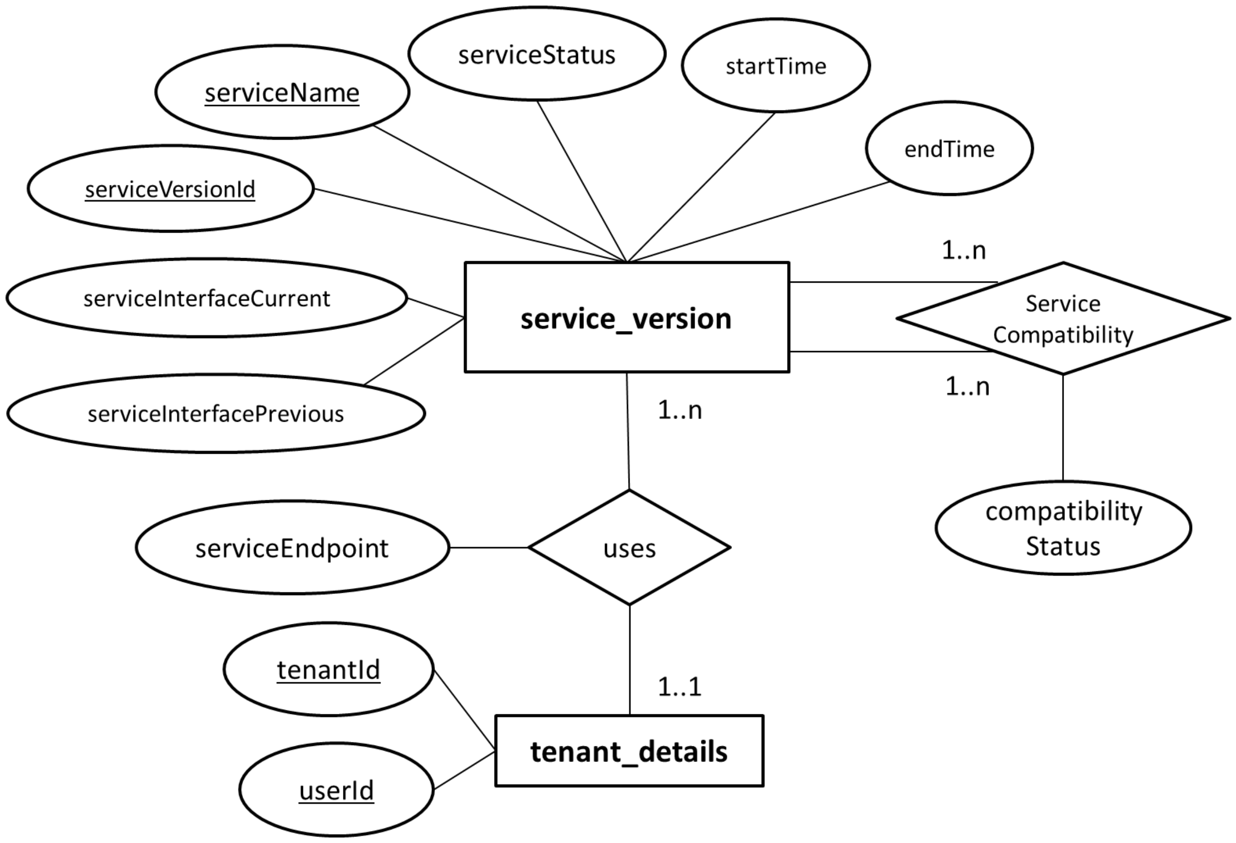


Figure 4.3: ER-Diagram of Version Registry

4.3. System Architecture

This section serves as a result to the system requirements and system design discussed in the preceding sections. Herein the system architecture can be observed in two facets, namely from its functionalities, and communication or the wiring between the components.

4.3.1. Functionalities of System Architecture

From Figure 4.1 the overall system architecture can be refined into a composition of existing core components and data sources of JBIMulti2 system, also in addition to the new Version Control Manager and Version Registry as an extension of the existing system as depicted by Figure 4.4.

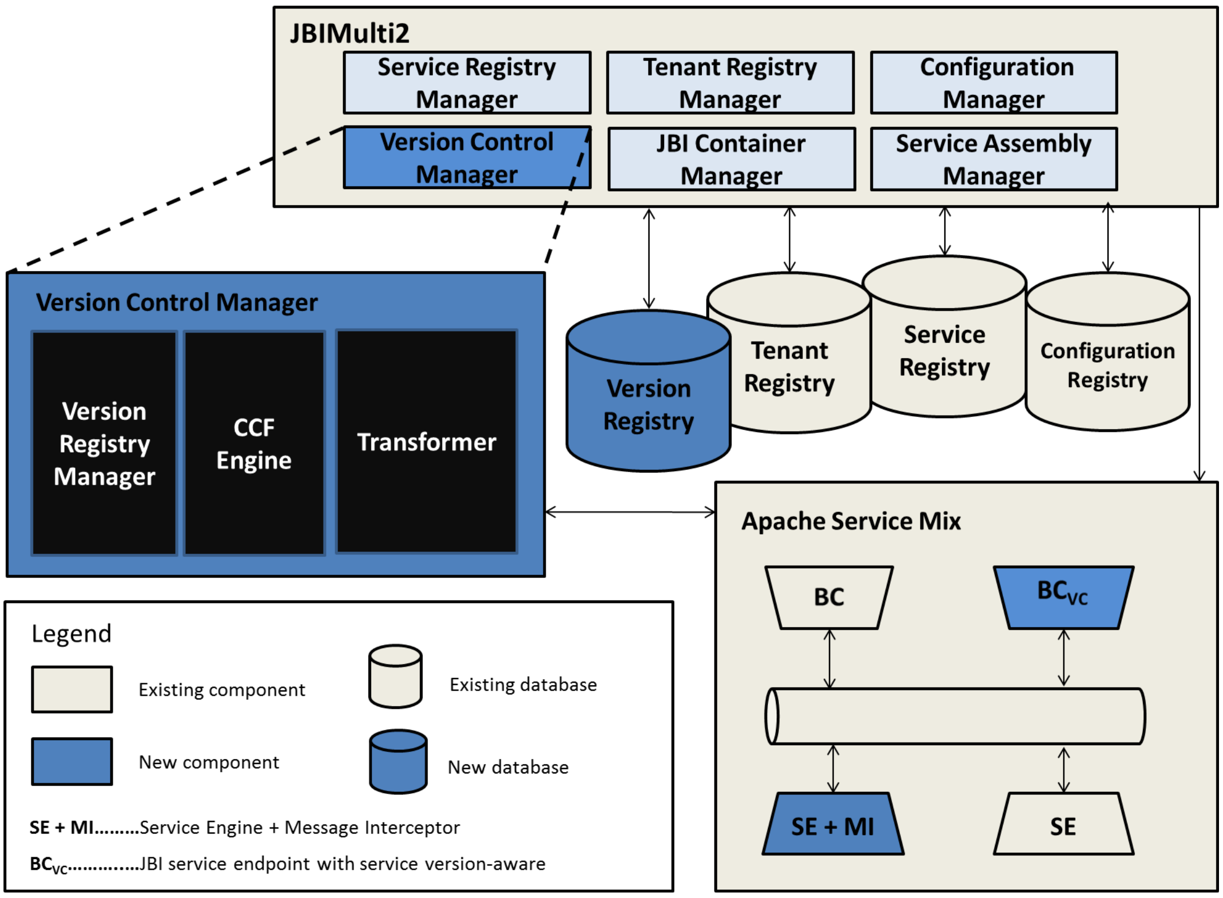


Figure 4.4: Building blocks of system architecture

Extending the JBIMulti2 System

- Service Registry Manager: it registers service assemblies and services to the Service Registry. In addition to those operations, replace and deploy service in parallel operations will be added as new functionalities of the Service Registry Manager.

- Tenant Registry Manager: it stores and retrieves tenant-related information to/from the Tenant Registry.
- Configuration Manager: it stores configuration data created by system administrators or tenants to the Configuration Registry.
- JBI Container Manager: it is designated to communicate with underlying multi-tenant aware ESB implementation.
- Service Assembly Manager: it adds the tenant context to each service unit contained in a service assembly, so that once deployed in multi-tenant aware ESB they do not interfere with service units of other tenants.
- Service Registry: it is a database that stores service assemblies and the active service interface.
- Tenant Registry: it is a database that stores tenant information.
- Configuration Registry: it keeps the configuration data.

Version Registry

The existing database Service Registry will not be used to maintain service versioning. Service Registry is designated to act as a front-end database for service registration purposes for the tenant operator, and it maintains only services in runtime. For service interface versioning purpose a new data resource, *Version Registry*, is proposed. The Version Registry is a special purpose database dedicated to holds all service versioning-related information and history. This database serves as a back-end data source for the system i.e. it will not have a direct connection with the JBIMulti2 system. The Version Control Manager is responsible to abstract the communication from the JBIMulti2 by interconnecting the Version Registry and the system, and it will act upon based on the new service registration, unregister service, replace and deploy service in parallel operations.

The entities defined in ER-Diagram (see Section 4.2.2) are transformed into a set of tables (see Figure 4.5). Each table represents an entity with its corresponding rows representing the attributes of the entity. The relationship between tables is also shown. The entity `tenant_details` and `service_version` produces a separate table called `service_tenant_assignment` which is represented by their respective primary keys.

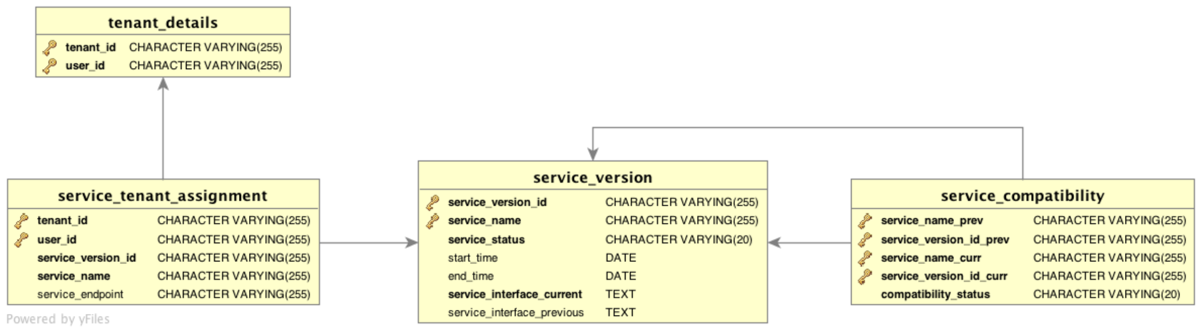

tenant_details
tenant_id CHARACTER VARYING(255)
user_id CHARACTER VARYING(255)
service_tenant_assignment
tenant_id CHARACTER VARYING(255)
user_id CHARACTER VARYING(255)
service_version_id CHARACTER VARYING(255)
service_name CHARACTER VARYING(255)
service_endpoint CHARACTER VARYING(255)
service_version
service_version_id CHARACTER VARYING(255)
service_name CHARACTER VARYING(255)
service_status CHARACTER VARYING(20)
start_time DATE
end_time DATE
service_interface_current TEXT
service_interface_previous TEXT
service_compatibility
service_name_prev CHARACTER VARYING(255)
service_version_id_prev CHARACTER VARYING(255)
service_name_curr CHARACTER VARYING(255)
service_version_id_curr CHARACTER VARYING(255)
compatibility_status CHARACTER VARYING(20)
Powered by yFiles

Figure 4.5: Table representation of Version Registry

Version Control Manager

Version Control Manager consists of three main components and will be explained as follows.

1. Version registry manager: it is an API that listens to events of new service registration, unregister service, replace or deploy service in parallel operations occurred in the Service Registry. It is also responsible to call CCF engine to assess the compatibility of service versions, and update Version Registry and message interceptor with the result from CCF engine. The version registry manager might implement a registry cache that will retain a set of query results for period of time from the Version Registry to avoid excessive access from the message interceptor.
2. CCF engine: it checks the compatibility of service descriptions. This component is invoked by the version registry manager after the manager received an event of service registration, or replace services. The CCF engine will perform compatibility assessment and return the result in form of service compatibility status.
3. Transformer: it is invoked by the service engine from multi-tenant aware ESB side to adapt the outgoing message when the incoming message request is compatible with the service version implemented by the service endpoint, and vice versa.

Binding Component with service version-awareness

The outgoing JBI service endpoints have to support version control management. This can be realized by enriching the endpoint names with unique service version identifiers for each endpoint representing one service version offered by service producer. Incoming messages targeting these endpoints will be routed correctly by routing engine of multi-tenant aware ESB.

Service Engine with Message Interceptor

In case of service engine which support routing capability, Apache Camel SE is used. *Message interceptor* listens to the communication channel for messages in versions that are compatible and triggers the Transformer in order to transform them appropriately, before pushing them back to the channel.

4.3.2. Communication between Components

This section refines the high-level system components introduced in Section 4.1 in terms of the communication between components. The first part of this section explains how the system behaves in the management and administration aspects of the version control management. In the second part, the message processing logic of the multi-tenant aware ESB will be discussed.

4.3.2.1. Version Control Management

The new use cases described in Section 4.2.1 allow tenant operators to replace and deploy services in parallel, thus, enabling the management application i.e. the JBIMulti2 system to provide version control capability. The activity diagrams shown in this section will describe the procedural executions and flows of related system components when tenant operators apply the new use cases and deploy service assembly.

Figure 4.6 shows an activity diagram for service replacement. The system components involve here are the Service Registry and Version Registry. In this case, the tenant operator who acts as a service provider will replace an existing service version with another service. The service replacement use case is just a chain of unregister and register service operation. When the tenant operator execute the replace service action, the JBIMulti2 system will first of all unregister the service by removing it from Service Registry and then setting the service status in Version Registry to decommissioned. After the two operations have been carried out, the system will continue with registering the new service to the Service Registry, and in turn it will be stored into the Version Registry together with all service version-related information. To indicate that the new service is the most actual version, the service status is set to active. The replace service operation ends when the JBIMulti2 receives an acknowledgement.

Another use case is the deploy service in parallel. The tenant operator will simply register a new service version and let it be available at the same time with the existing service. In this case the new service information will be added into Service Registry as well as into Version Registry. The status of new service is set to active. The operation commits after the service has successfully been stored in the registries.

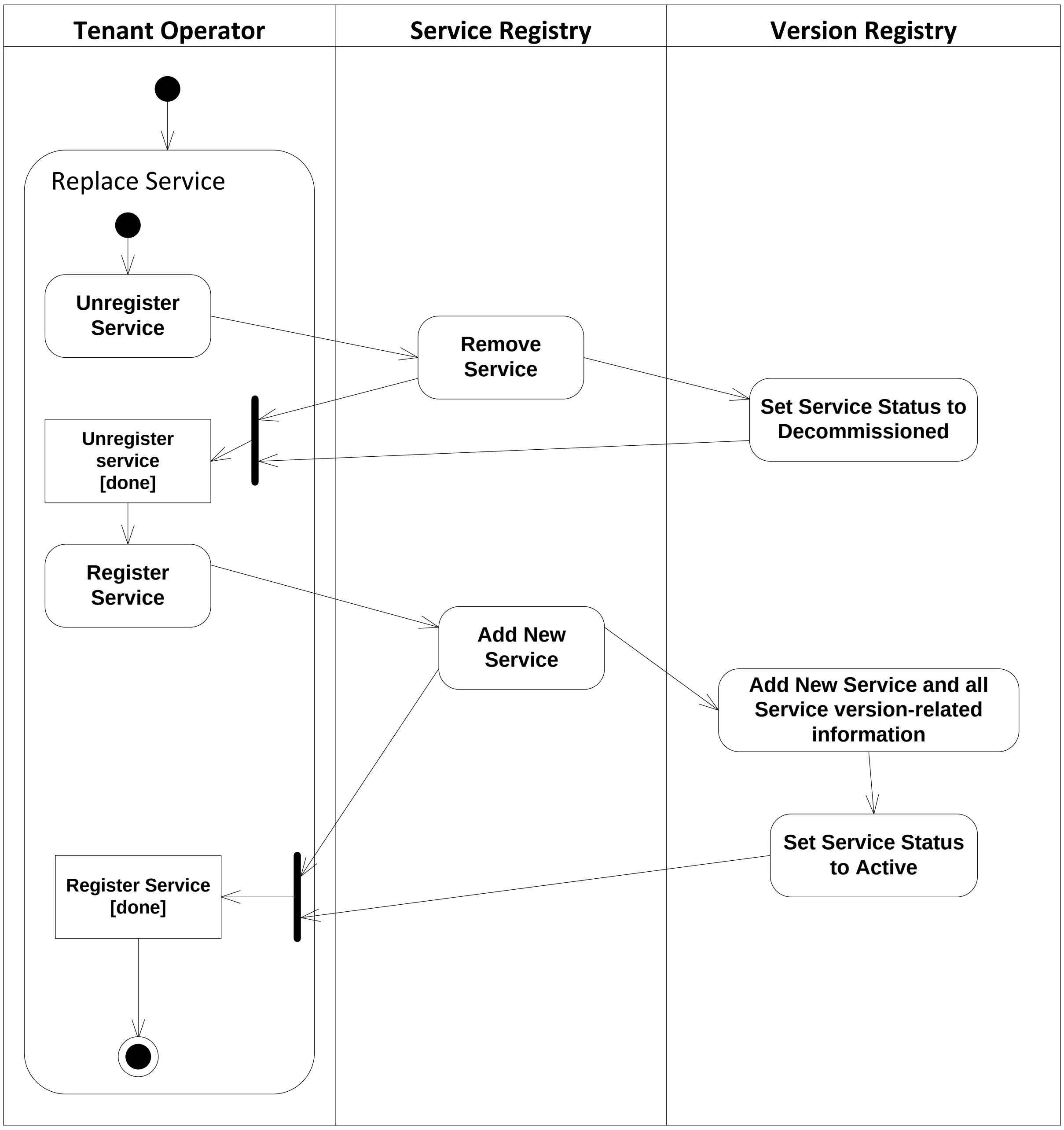

Tenant Operator
Service Registry
Version Registry
Replace Service
Unregister Service
Remove Service
Set Service Status to Decommissioned
Unregister service [done]
Register Service
Add New Service
Add New Service and all Service version-related information
Set Service Status to Active
Register Service [done]

Figure 4.6: Activity diagram of replace service use case

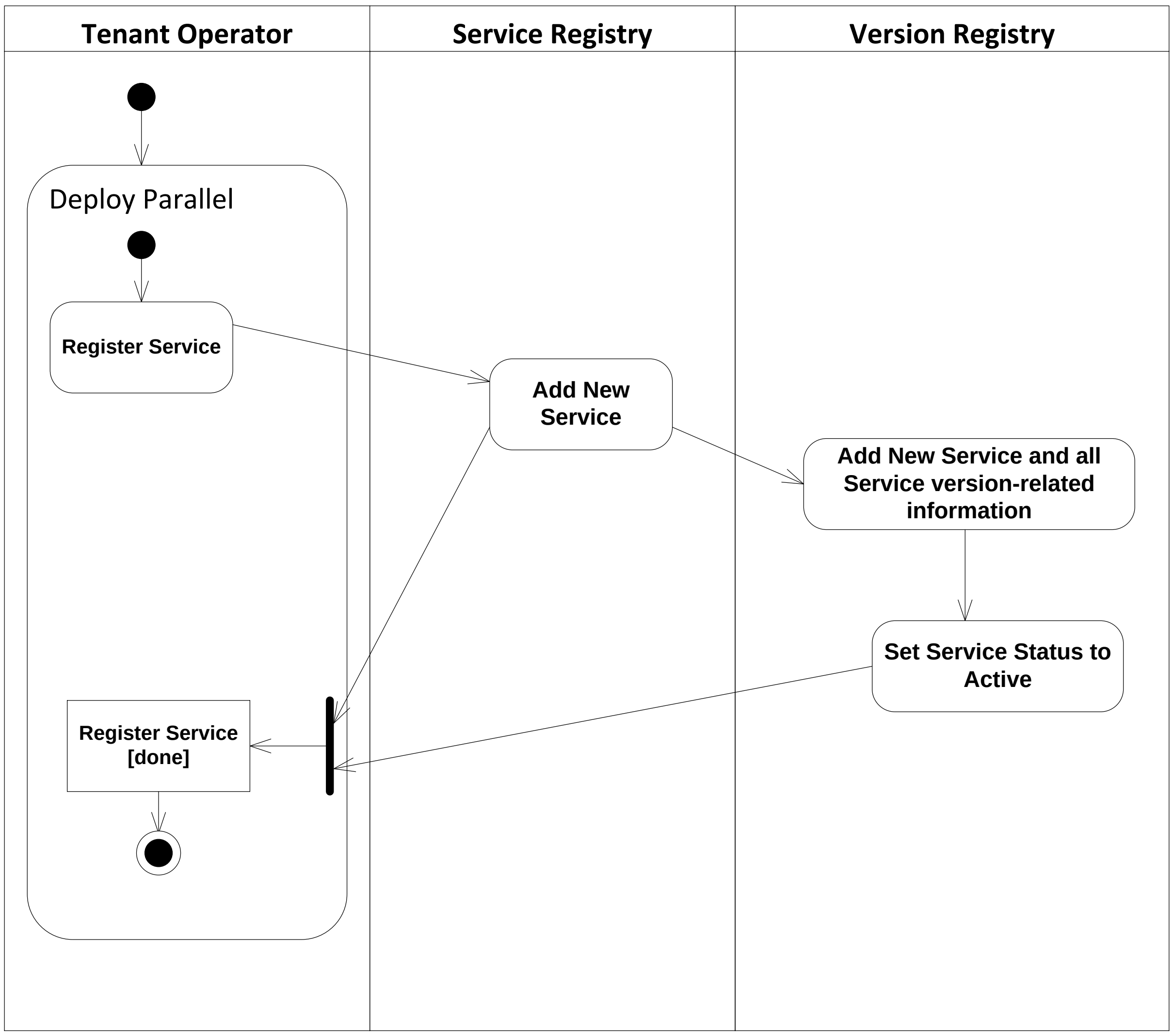


Figure 4.7: Activity diagram of deploy service in parallel use case

The last two activity diagrams showed a tenant operator as a service provider that replaced or deployed service in parallel. In Figure 4.8, the tenant as service consumer or provider needs to set up the configuration files to define which service she wants to consume or expose, respectively. The configuration files in form of service units will be packaged into a service assembly. When the deploy service assembly operation is performed by tenant operator, the service assembly artifact will be stored in the Service Registry. Then the file will be sent together with the management message from JBIMulti2 system to multi-tenant aware ESB. Before the service assembly is deployed, tenant context and service version identifier will be injected into the service assembly file. By using those information, multi-tenant and service version-aware service endpoints are generated. In the end, the JBIMulti2 will receive an acknowledgment from multi-tenant aware ESB indicating that the service assembly has been successfully deployed.

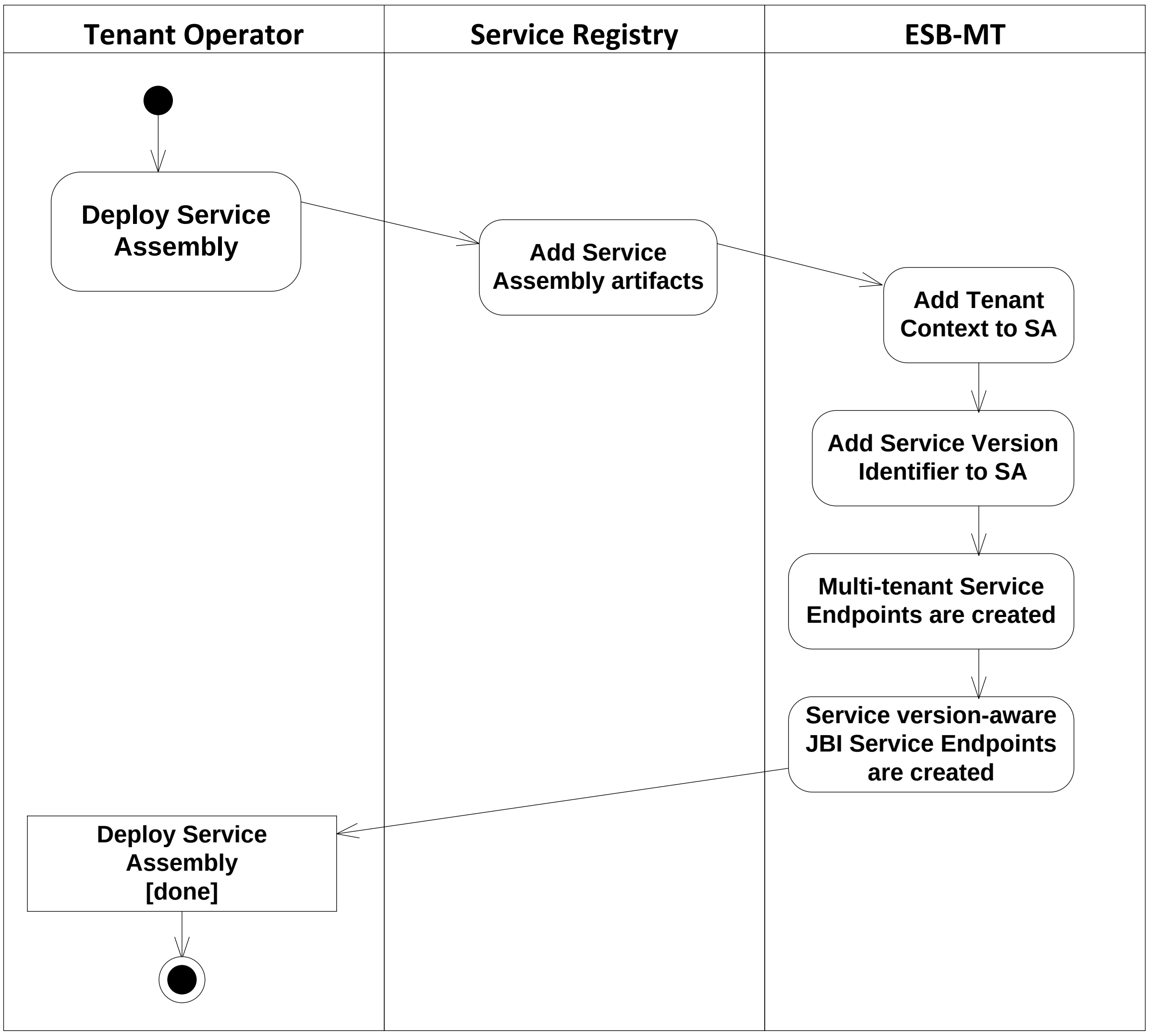

Tenant Operator
Service Registry
ESB-MT
Deploy Service Assembly
Add Service Assembly artifacts
Add Tenant Context to SA
Add Service Version Identifier to SA
Multi-tenant Service Endpoints are created
Service version-aware JBI Service Endpoints are created
Deploy Service Assembly [done]

Figure 4.8: Activity Diagram of deploy service assembly use case

4.3.2.2. Message Processing Logic

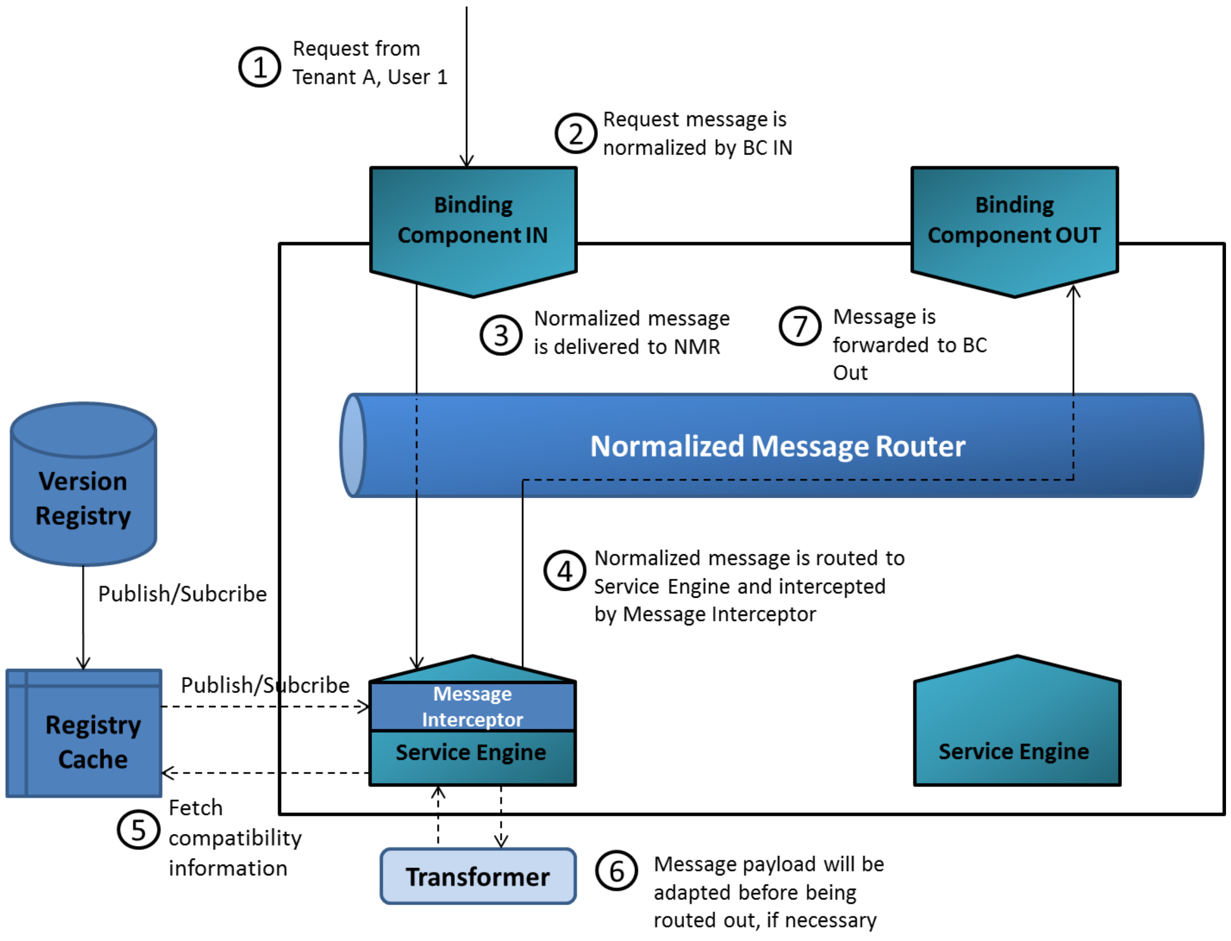
After the system interactions in management and administration perspective have been illustrated in previous section, it is also important to see how incoming messages targeting a particular service endpoint are being processed. shows the message flow of an incoming request from a tenant. It also illustrates how the message interceptor in the routing engine intercepts the incoming message and route it to the correct outgoing service endpoint. The message flow for incoming requests is explained in the following:

- A request message comes from a particular tenant user targeting an endpoint URL of BC_{in}.
- The message is normalized by the BC_{in} and wired to a SE for the corresponding BC_{in}.
- The normalized message is intercepted by the message interceptor on SE.

- Based on the service versioning information from the registry cache, the message interceptor can decide whether the message payload needs to be transformed before being forwarded to BC_{out}.
- The normalized message is routed by SE to BC_{out}.

Likewise, the response message for the tenant user can be described as follows:

- A reply message comes from BC_{out} for the tenant user.
- The message is normalized by the BC_{out} and wired to a SE for the corresponding BC_{out}.
- The normalized message is intercepted by the message interceptor on SE.
- Based on the service versioning information from the registry cache, the message interceptor can decide whether the message payload needs to be transformed before being forwarded to BC_{in}.
- The normalized message is routed by SE to BC_{in} where the request message came from.



① Request from Tenant A, User 1
② Request message is normalized by BC IN
Binding Component IN
Binding Component OUT
③ Normalized message is delivered to NMR
⑦ Message is forwarded to BC Out
Normalized Message Router
Version Registry
Publish/Subcribe
④ Normalized message is routed to Service Engine and intercepted by Message Interceptor
Registry Cache
Publish/Subcribe
Message Interceptor
Service Engine
Service Engine
⑤ Fetch compatibility information
Transformer
⑥ Message payload will be adapted before being routed out, if necessary

Figure 4.9: Message flow for incoming message request

Depending on the compatibility status information from the registry cache, the actions performed by the message interceptor might vary. The compatibility status between two services can be compatible, incompatible, identical, or even undetermined. To better illustrate how the ESB reacts, the sequence diagrams for each compatibility status will be presented.

Figure 4.10 shows a sequence diagram of how the system components behave when incoming message and the target service have the compatibility status of compatible. When the incoming message comes for BC_{in}, the message is normalized and wired to the SE of corresponding BC_{in}. The message interceptor on SE will fetch the compatibility information from the registry cache. If the result returns a compatible status, the message interceptor will invoke the transformer to adapt the incoming message payload as necessarily. The transformed normalized message will be finally wired to BC_{out}.

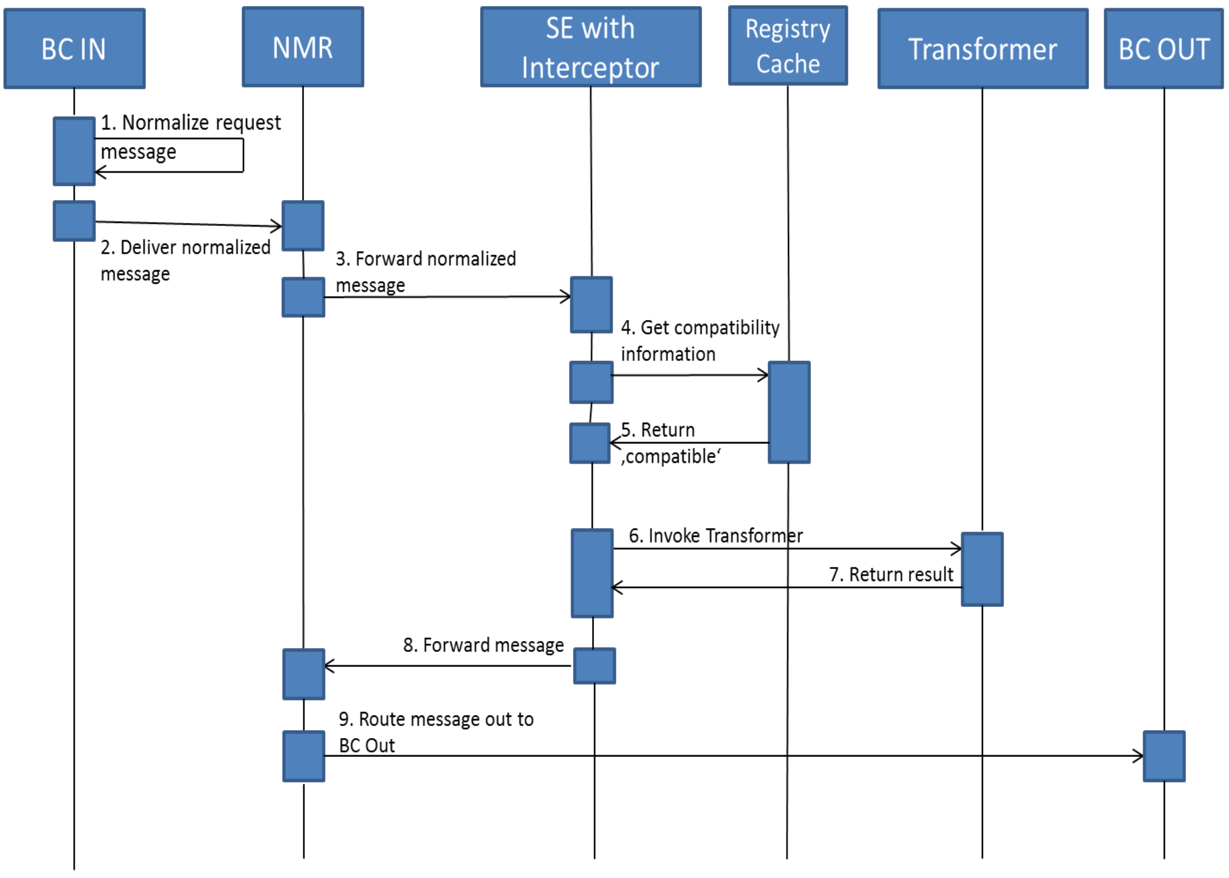

BC IN
NMR
SE with Interceptor
Registry Cache
Transformer
BC OUT
1. Normalize request message
2. Deliver normalized message
3. Forward normalized message
4. Get compatibility information
5. Return ‚compatible'
6. Invoke Transformer
7. Return result
8. Forward message
9. Route message out to BC Out

Figure 4.10: Sequence Diagram of processing compatible message

Figure 4.11 shows a sequence diagram of how the system components process the incompatible incoming message. When the incoming message comes for BC_{in}, the message is normalized and wired to the SE of corresponding BC_{in}. The message interceptor on SE will retrieve the compatibility information from the registry cache. When the compatibility status is incompatible, it means that the transformer will not be invoked. In this case an error message will be returned to the BC_{in}.

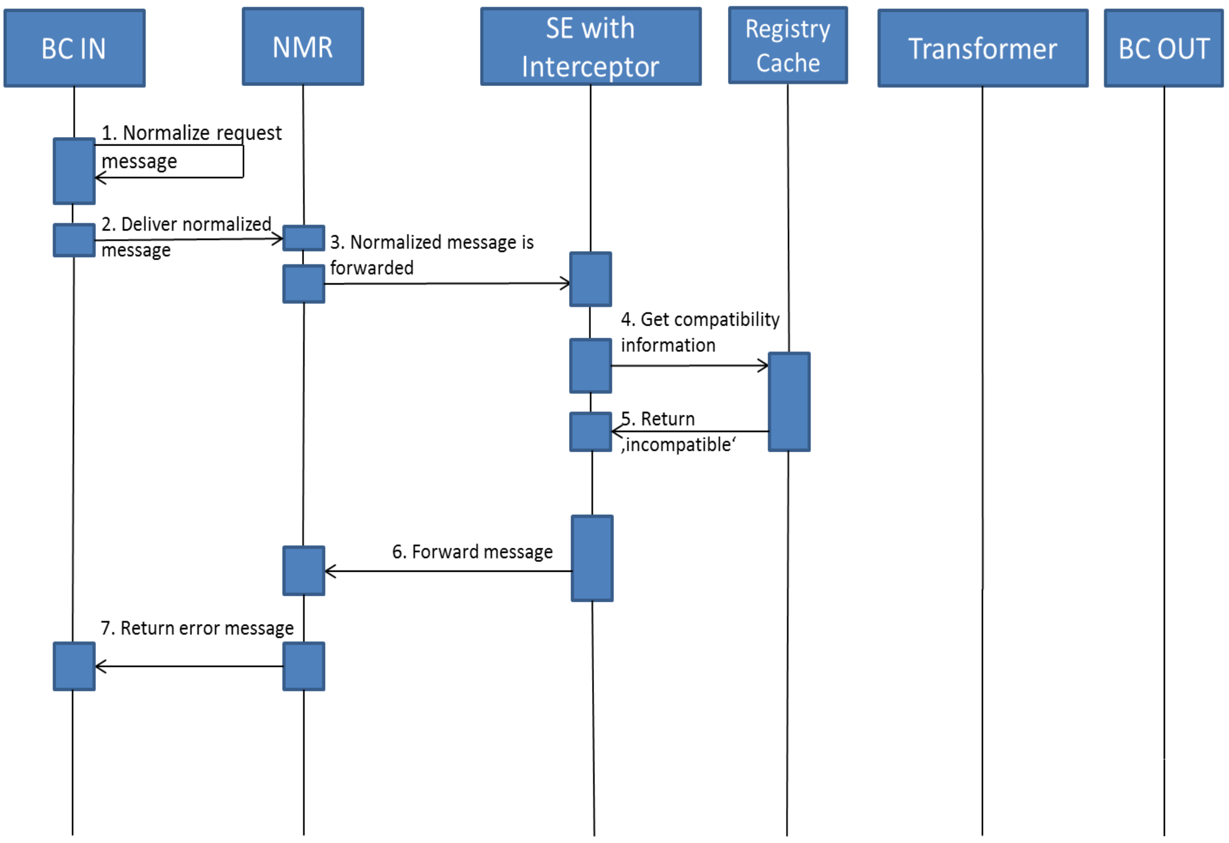

BC IN
NMR
SE with Interceptor
Registry Cache
Transformer
BC OUT
1. Normalize request message
2. Deliver normalized message
3. Normalized message is forwarded
4. Get compatibility information
5. Return ‚incompatible'
6. Forward message
7. Return error message

Figure 4.11: Sequence Diagram of processing incompatible message

In the case of identical message, it simply means that the incoming message has the same service version with the one adopted by the service provider. When the message interceptor gets a result of identical service version, the SE will directly route the message to the BC$_{out}$. This process is depicted in Figure 4.12.

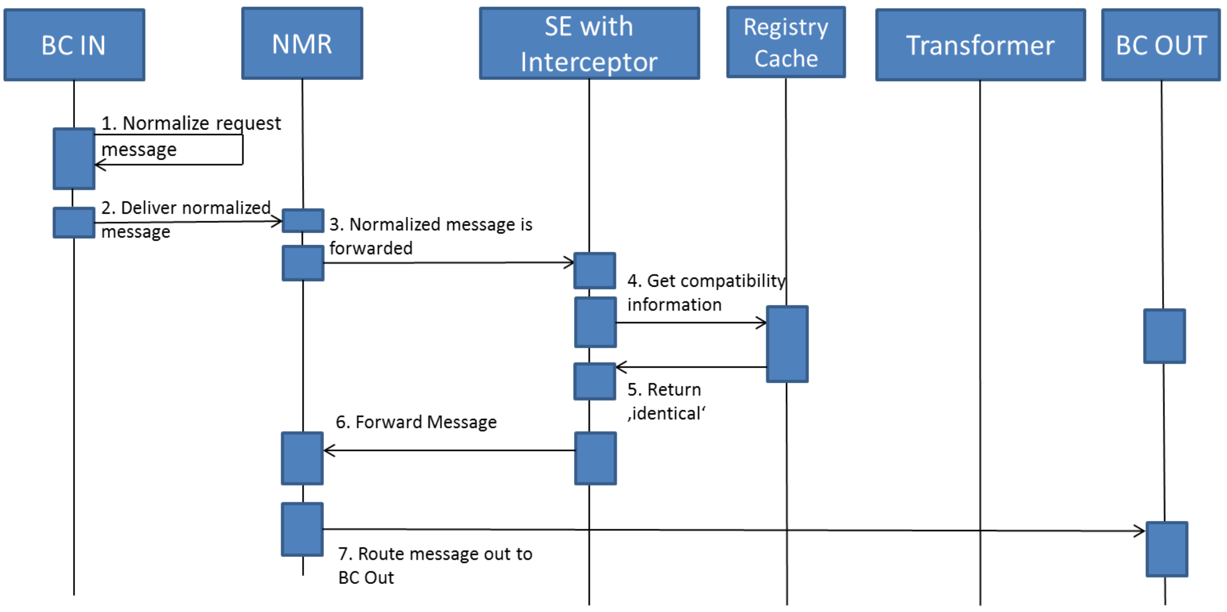

BC IN
NMR
SE with Interceptor
Registry Cache
Transformer
BC OUT
1. Normalize request message
2. Deliver normalized message
3. Normalized message is forwarded
4. Get compatibility information
5. Return ‚identical'
6. Forward Message
7. Route message out to BC Out

Figure 4.12: Sequence Diagram of processing identical message

The sequence diagram for processing the undetermined message in Figure 4.13 is similar to the sequence diagram of processing incompatible message. What makes difference is only the information returned by the registry cache i.e. undetermined compatibility status. No transformer will be called and an error message will be returned as a result to the BC_{in}.

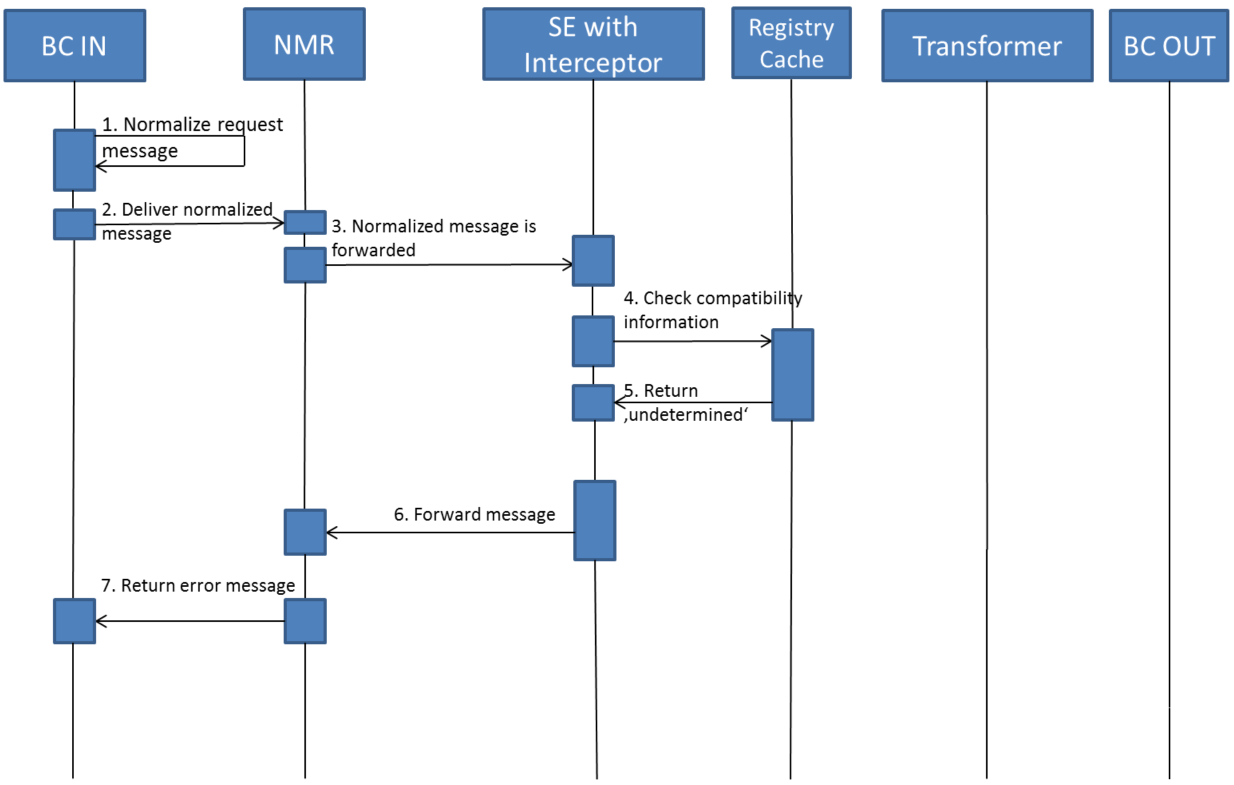

BC IN
NMR
SE with Interceptor
Registry Cache
Transformer
BC OUT
1. Normalize request message
2. Deliver normalized message
3. Normalized message is forwarded
4. Check compatibility information
5. Return ‚undetermined'
6. Forward message
7. Return error message

Figure 4.13: Sequence Diagram of processing undetermined message

The sequence diagrams depicted above can be represented as an algorithm that shows how a message interceptor implements the message processing logic. The algorithm is given as pseudo code and is listed in Listing 4.1.

```
IF incoming message = "identical" THEN
     Forward message to corresponding target endpoint

ELSE IF incoming message = "compatible" THEN
     Invoke Transformer AND
     Forward transformed message to corresponding target endpoint

ELSE IF incoming message = "incompatible" THEN
     Generate error message OR
     Generate a message for new service version update

ELSE
     Generate error message

ENDIF
```

Listing 4.1: Algorithm for message processing in Apache Camel

5. Implementation

This chapter gives an overview of the foundational technologies being used in the first section. The second section presents the system components that have been implemented. This includes the Version Registry creation, and also the JBIMulti2 and Apache Camel extensions. The last section gives a short deployment guide how the entire system can be set up.

5.1. Foundational Technologies

This section describes the adopted technological solutions to implement the system design and architecture specified in Chapter 4. A short overview of the related technologies will be presented below.

PostgresSQL 9.1.1
PostgresSQL is an open-source object-relational database system [39] that can be run on all major operating system such as Linux, UNIX, or Windows, and it also supports native programming interfaces for many programming languages i.e. Java, C/C++, or .NET. The PostgresSQL implementations follow ANSI-SQL:2008 standard.

Java Platform, Enterprise Edition v.5 (Java EE 5)
Java EE 5 platform [40] provides APIs and runtime environment for developing enterprise applications. The enterprise applications can be deployed to any application server that conforms to Java EE 5 or later specifications. In the following a set of APIs that implement the business logic, and Web services shall be given:

- **Enterprise Java Bean (EJB) 3.0**: EJB 3.0 is server-side components that can be used to build parts of the enterprise application such as the business logic or persistence layer. These components comprise of session beans, message-driven beans, and entity beans. The components live in a container which manages and provides services to them [41].
- **Java Message Service (JMS) 1.1**: the JMS specification defines a messaging standard that allows applications integration in loosely-coupled manner by connecting to a JMS provider interface [42].
- **Java API for XML-based Web Services (JAX-WS) 2.0**: JAX-WS is used to map Java Interfaces to WSDL documents, and vice versa. It also allows the development of Web service providers and clients [43].
- **Java Architecture for XML Binding (JAXB) 2.0**: this API can be used to marshal Java objects into XML file and unmarshal XML file back into Java objects [44].

Apache ServiceMix 4.3.0
Apache ServiceMix [11] is an open source ESB from Apache Software Foundation that adopts OSGi framework for its kernel layer. ServiceMix that implements JBI specification is shipped with JBI components which expose the ESB core functionalities. BCs that provide service connectivity offer various communication protocols like HTTP, JMS, FTP, and Mail. Meanwhile SEs can provide services such as the implementation of Enterprise Integration Patterns (EIP) defined in [38] and messaging foundation. The former is bundled into Apache Camel SE [45], and the latter into Apache ActiveMQ [46]. In addition, ServiceMix provides a console management that allows its users to perform any management tasks such as JBI components installation and service assembly deployment.

Apache Camel 2.7.0
Apache Camel [45] is an open source integration framework based heavily on the EIP. The core functionalities of Camel are routing and mediation engine. The routing rules in Camel can be configured by using either Java based Domain Specific Language (DSL) or Spring DSL. The RouteBuilder class describes how a route in Camel can be created. By using this class, it gives developers capability to define the routing logic and message filtering rule for examples. The endpoints representation in Camel is based on URI. This allows Camel to integrate with many messaging transport protocol supported e.g. HTTP, JMS, etc. Camel is embedded into ServiceMix as a routing engine to wire messages.

Apache ActiveMQ 5.3.1
Apache ActiveMQ [46] is a message broker for system communication using JMS in an asynchronous and loosely coupled manner, and it can be integrated with any Java compliant application servers. ActiveMQ supports point-to-point and publish-subscribe messaging model. The point-to-point message model describes a single message produced by message producer is consumed by a single message consumer, while publish-subscribe model means a single message is consumed by a group of message consumers. In ServiceMix, ActiveMQ is used to receive a management messages in form of JMS messages.

5.2. Implemented Components

JBIMulti2 adopts three-tier-architecture that consists of presentation layer, business logic layer, and data sources. The service version control management components such as Version Registry and version control manager will become additional parts of the existing data source and business logic components, respectively.

5.2.1. Version Registry

The database schema of Version Registry (Section 4.2.2) will be realized and implemented by using PostgreSQL 9.1.1. The PostgreSQL will generate a database for Version Registry which comprises of four tables. The Data Definition Language (DDL) for the Version Registry is shown is Listing 5.1.

```
CREATE DATABASE versionregistry;

CREATE TABLE service_version (
      service_version_id           varchar(255),
      service_name                 varchar(255),
      service_status               varchar(20) NOT NULL,
      start_time                   date,
      end_time                     date,
      service_interface_current    text NOT NULL,
      service_interface_previous   text,
      PRIMARY KEY (service_version_id, service_name)
);

CREATE TABLE service_compatibility (
      service_name_prev           varchar(255),
      service_version_id_prev     varchar(255),
      service_name_curr           varchar(255),
      service_version_id_curr     varchar(255),
      compatibility_status        varchar(20) NOT NULL,
      PRIMARY KEY (service_name_prev, service_version_id_prev,
                  service_name_curr, service_version_id_curr),
      FOREIGN KEY (service_name_prev, service_version_id_prev) REFERENCES
                  service_version (service_name, service_version_id),
      FOREIGN KEY (service_name_curr, service_version_id_curr) REFERENCES
                  service_version(service_name, service_version_id)
);

CREATE TABLE tenant_details (
      tenant_id              varchar(255),
      user_id                varchar(255),
      PRIMARY KEY (tenant_id, user_id)
);

CREATE TABLE service_tenant_assignment (
      tenant_id              varchar(255),
      user_id                varchar(255),
      service_version_id     varchar(255),
      service_name           varchar(255),
      service_endpoint       varchar(255),
      PRIMARY KEY (tenant_id, user_id),
      FOREIGN KEY (tenant_id, user_id) REFERENCES tenant_details
            (tenant_id, user_id),
      FOREIGN KEY (service_version_id, service_name) REFERENCES
                  service_version(service_version_id, service_name)
);
```

Listing 5.1: DDL of Version Registry

Object and Database Mapping

One of the EJB types is entity beans which are Java objects with @Entity annotation. The object-relational mapping (ORM) defines how data in the entity beans are mapped into database tables. In Java world, JPA is the Java solution to persist entity beans into database. Entity beans are managed by persistent provider through JPA EntityManager. The EntityManager is an API for performing persistence operations for entities.

The package `versioncontrol.domain` in Figure 5.1 contains the Java files that map to the database tables of Version Registry. The `ServiceCompatibility`, `ServiceVersion`, TenantDetails classes are mapped to service_compatibility, service_version, and tenant_details tables of Version Registry respectively. The table service_tenant_assignment is defined and mapped by TenantDetails class because it is the ManyToOne side of the relationship with service_version. The ServiceCompatibility-, ServiceVersion-, TenantDetailsPK define the composite primary key for the corresponding ServiceCompatibility, ServiceVersion, TenantDetails classes. The composite primary key classes will be embedded into the entity classes by using @EmbeddedId annotation.

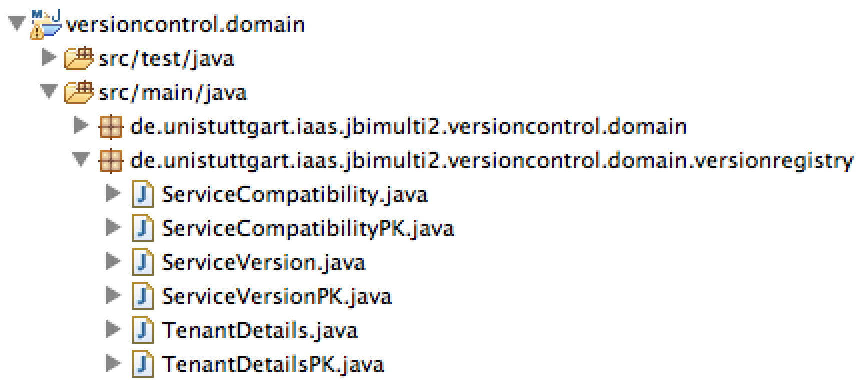

```
versioncontrol.domain
  src/test/java
  src/main/java
    de.unistuttgart.iaas.jbimulti2.versioncontrol.domain
    de.unistuttgart.iaas.jbimulti2.versioncontrol.domain.versionregistry
      ServiceCompatibility.java
      ServiceCompatibilityPK.java
      ServiceVersion.java
      ServiceVersionPK.java
      TenantDetails.java
      TenantDetailsPK.java
```

Figure 5.1: Java package files describing the object-relation mapping

Registry Cache

Caching will increase database performance by reducing the database round-trips. When the data in the database does not change so often caching might be the right approach to be considered. The registry cache that resides in version control manager is required to query information from Version Registry database. Since the query is used to retrieve tenant-, and service-related information from the database, same parameters will always be used and executed frequently. In this case, *query cache* can be adopted to hold the query result sets.

Listing 5.2 shows a query cache that retrieves information from the table service_compatibility and service_tenant_assignment. The setHint method is responsible for storing the query result in a certain time interval. So the same query retrieval will not result into database access.

```
Query query = entityManager.createQuery
      ("SELECT sc, td FROM ServiceCompatibility sc, TenantDetails td" +
                  " WHERE sc.compositePrimaryKey.serviceVersionIdCurr” +
                              ‟ = td.serviceEndpoint");

query.setHint("org.hibernate.cacheable", true);
```

Listing 5.2: Example of query cache that holds information from the result query

5.2.2. Java Method Definitions Extensions

This section describes how the extensions to corresponding system components can be implemented. First, the JBIMulti2 extensions with respects to the new use cases will be presented. Then it is followed by the Apache Camel enhancement for service version-aware.

Extensions to JBIMulti2

The new use cases, namely replace and deploy service in parallel will be added as Java methods in the service registry manager of business logic layer. The method definitions and the implementations will be presented in Listing 5.3 and Listing 5.4, respectively.

```
public interface ServiceRegistry {
...
     public void replaceService(String tenantId, String userId,
      Service service, String serviceName, String wsdlFile);


     public void deployParallel(String tenantId, String userId,
      String serviceName, String wsdlFile);
...
}
```

Listing 5.3: Method definitions of replace and deploy service in parallel use cases

The replace service use case is just a chain of existing unregister and register methods. The `replaceService` method will reuse the existing `unregisterService` and `registerService` methods to remove a particular service and to register a new one. Meanwhile, the `deployParallel` method uses `registerService` method to register another service version.

```
public class ServiceRegistryBean implements ServiceRegistry {

...
     public void replaceService(String tenantId, String userId,
Service service, String serviceName, String wsdlFile){

          unregisterService(service);
          registerService(tenantId, userId, serviceName, wsdlFile);
     }

     public void deployParallel(String tenantId, String userId,
String serviceName, String wsdlFile){

          registerService(tenantId, userId, serviceName, wsdlFile);
...
}
```

Listing 5.4: Method implementations of replace and deploy service in parallel use cases

```
@PermissionType(type = PermissionTypeEnum.USE_SERVICE_REGISTRATION_CONTINGENT)
public ServiceRegistrationEntry replaceService(String userUUID, String wsdlFile,
 String chargeContingent, String serviceName)
           throws AuthorizationException, ExecutionException;


@PermissionType(type = PermissionTypeEnum.USE_SERVICE_REGISTRATION_CONTINGENT)
public ServiceRegistrationEntry deployParallel(String wsdlFile,
 String chargeContingent)
           throws AuthorizationException, ExecutionException;
```

Listing 5.5: Excerpt from Java Interfaces definition for required permission to perform replace and deploy service in parallel

Listing 5.5 shows an excerpt from `TenantOperatorFacade.java` that defines the required permissions to execute the `replaceService` and `deployParallel` methods specified above. Based on the role-based access described in Section 3.2.1, whenever the system actor, in this case, the tenant operator wants to execute one of these methods, the system will check the corresponding Tenant Registry and Configuration Registry to verify whether the tenant operator has a permission to perform this method call. The Java annotation `@PermissionType` define which permission type has been assigned to the corresponding actors.

Extensions to Apache Camel

In order to enable service version-aware on Apache Camel, the multi-tenant Camel SE *servicemix-camel-mt* [6] has to be modified. In class `CamelProviderEndpoint` there is a method called `ensureMultiTenancy` that modified the original service namespace URI to support multi-tenancy. This method can be overridden to also include service version identifier (see Listing 5.6). In the case of class `CamelConsumerEndpoint`, it contains a method `configureTenancyAwareExchange` that transforms Camel URIs dynamically to JBI service endpoint names and includes tenant context to the service name for outgoing message exchanges.

```
public void ensureMultiTenancy(String tenantId, String versionId) {
        QName configuredService = this.getService();

        String endpointPrefix = getTenantsEndpointURI();
        if (!endpointPrefix.endsWith("/")) {
            endpointPrefix = endpointPrefix + "/";
        }

        // rename service namespace URI
        QName modifiedService = new QName(endpointPrefix + tenantId,
                configuredService.getLocalPart() + "/" + serviceId);
        this.setService(modifiedService);
    }
```

Listing 5.6: Extension to CamelProviderEndpoint.java to support service version-aware

Likewise, a service version identifier can be appended (see Listing 5.7). The message interceptor can process incoming messages based on the URIs information provided by the `configureTenancyAwareExchange` and information retrieved from the registry cache. By matching outgoing Camel URIs and target service endpoint, the routing engine can wire the incoming messages to the correct service endpoints.

```
private void configureTenancyAwareExchange(MessageExchange exchange,
ComponentContext context, String destinationUri, String endpointUri)
{
…
else if (destinationUri.startsWith("endpoint:")) {
            String uri2 = destinationUri.substring(9);
            String[] parts = URIResolver.split3(uri2);
            if (isTargetServiceGlobal(destinationUri, endpointUri)
                    && !parts[0].startsWith(tenantsEndpointURI))
           {
                ServiceEndpoint se = context.getEndpoint(new
                QName(parts[0], parts[1]), parts[2]);
                exchange.setEndpoint(se);
           }else {
                QName modifiedService = new QName(tenantsEndpointURI
                + tenantId, parts[1]);
                ServiceEndpoint se =
     context.getEndpoint(modifiedService, parts[2]+ "/" + serviceId
);
                exchange.setEndpoint(se);
            }
…
}
```

Listing 5.7: Extension to CamelConsumerEndpoint.java to support service version-aware

5.3. Deployment Guide

In order to put the entire system into the motion, there are several steps that need to be carried out in each architectural layer. On the data source layer, the new database Version Registry has to be created besides the existing databases. The Version Registry can be generated by using an SQL script that can be executed by the PostgreSQL commands.

The JBIMulti2 Web application consists of different Java modules and libraries. Because the Java modules share common libraries, they are packaged into a single enterprise application (EAR file). The extension of JBIMulti2, that is, the service control manager can be added into the existing EAR file. The new EAR file is then deployed on Java Open Application Server (JOnAS) 5.2.2 [47]. To connect the application server with the Version Registry, the jonas.properties file in JOnAS application server has to be configured to support the new datasource Version Registry.

On the side of ESB, Apache ServiceMix 4.3.0 has to be installed. To manage to ServiceMix instance from JBIMulti2, a graphical SOAP-based Web services testing tool called soapUI [48] is used. All use cases described in [6] can be executed from soapUI. In the case of this work, soapUI is used to test the wiring between components to see whether the incoming messages in version are correctly routed to the target service endpoint.

6. Conclusion and Future Work

The main focus of this diploma thesis is to introduce a service version control management framework in line with the multi-tenant aware ESB, thus, resulting in system design and architecture of the entire participating system components. In Chapter 2, the fundamental concepts and background related to this work are discussed. This includes infrastructures or components involved in the context of service evolution, current state of art, and available theoretical concepts and methodologies. In Chapter 3, the requirements for service version control management in general are investigated. There are several key considerations when dealing with service versioning. These issues lay a foundation of what functionalities have to be provided by the version control framework in order to address them. Based on this fundamental specification, the service version-aware requirements for multi-tenant aware ESB should be pointed out. In order to add the service version-aware functionality properly, the management application and multi-tenant aware ESB itself should be considered as target extensions. The management application maintains a group of tenants and the tenants access the system in restrictive manner based on their access role. For service version-aware purposes, the use cases of the role tenant operator have to be extended. The tenant operator as service provider has to be able to replace an old service version with a new service version. Moreover, she can also deploy services in parallel with the existing ones. In another case, when a tenant acts as service consumer, she needs to deploy service configuration file to indicate which services she wants to use. The requirements inside the multi-tenant ESB have to be defined as well. The most important functionality for ESB is the routing capability that must be able to wire messages in version to correct service endpoint despite of different service version available.

In Chapter 4, a system design and architecture illustrates the solutions of specifications from the previous chapter. Following the management application requirements that need to add the new use cases to reflect to service version-aware capabilities, a use case diagram is given. The new replace and deploy service in parallel use cases are reuse the existing register and unregister service operation. To keep service versioning-related information, Version Registry as a new data source is also introduced. The database schema is explained using ER-Diagram. The system architecture that represents the entire system components is given in terms of the functionalities and wiring between components. Besides the existing components of the JBIMulti2 management application, the version control manager is added to retrieve data from the Version Registry, and update the multi-tenant aware ESB with the most current state of the service versioning information. From the ESB side, the JBI components should also be extended. The main target extension is the routing engine of the multi-tenant aware ESB. In the routing engine, a message interceptor has to be applied because it listens to the communication channel for messages in versions that are compatible and triggers the Transformer in order to transform them appropriately, before pushing them back to the communication channel. In the message processing logic section, an algorithm of how the message interceptor works is given in Listing 4.1, and the illustrations are depicted in form of sequence diagrams.

Chapter 5 presents the components implementation. In the persistence layer, the Version Registry is generated and accessed through Java Persistence API. Then the Java methods definitions for JBIMulti2 and Apache Camel extensions are covered. The former shows how the new use cases presented in Chapter 4 can be implemented and accessed by a proper system role. The latter shows which method of the `CamelProviderEndpoint` and `CamelConsumerEndpoint` can be enhanced with unique service version identifier.

For the future work, the multi-tenant aware ESB should be extended to support behavioral and policy-induced changes as defined in [12]. The behavioral changes means changes in way of interaction between service producers and consumers, while policy-induced changes describe changes in policy assertions and constraints. The multi-tenant aware ESB should also be provided with Web GUI to ease the interaction between the underlying system and the tenants, especially when tenants perform the management and administration tasks such as service registration and service assembly deployment.

Bibliography

[1] M. P. Papazoglou, V. Andrikopoulos and S. Benbernou, "Managing Evolving Services," *IEEE Software's SWSI: Component Software beyond Software Programming,* vol. 28, pp. 49-55, May/June 2011.

[2] F. Leymann and R. Mietzner, "Applications in the Cloud," in *ITPC Cloud Day*, 2009.

[3] "The 4CaaSt Project," [Online]. Available: http://www.4caast.eu.

[4] 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, NJ, USA: Prentice Hall PTR Upper Saddle River, 2005.

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

[6] D. Muhler, Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management. Diploma Thesis 3226, Institute of Architecture of Application Systems, University of Stuttgart, 2012.

[7] F. Chong and G. Carraro, "Architecture Strategies for Catching the Long Tail," Microsoft, 2006. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa479069.aspx#docume_topic5.

[8] S. G. Gomez, "Use Case Applications eMarketplace for SMEs: Scenario Definition," in *Deliverable D8.1.1, the 4CaaSt Consortium*, August 2011.

[9] "The C-CAST project:," [Online]. Available: http://www.ict-ccast.eu.

[10] S. Strauch, V. Andrikopoulos, F. Leymann and D. Muhler, "ESB^MT: Enabling Multi-Tenancy in Enterprise Service Buses," in *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'12)*, 2012.

[11] "Apache ServiceMix," [Online]. Available: http://servicemix.apache.org.

[12] V. Andrikopoulos, S. Benbernou and M. P. Papazoglou, "On the Evolution of Services," *In: IEEE Transactions on Software Engineering (TSE),* vol. 38 (3), pp. pp. 609-628, May-June, 2012.

[13] "Java Business Integration (JBI) 1.0, Final Release, 2005. JSR-208," [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr208/.

[14] T. Rademakers and J. Dirksen, Open Source ESBs in Action, Manning Publications Co, 2009.

[15] N. M. Josuttis, SOA in Practice: The Art of Distributed System Design, O'Reilly Media Inc, 2007.

[16] R. Daigneau, Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Pearson Education, Inc, 2012.

[17] M. P. Papazoglou, Web Services: Principles and Technology, Pearson Education Limited, 2008.

[18] M. Rouse, "Versioning Definition," SearchSofwareQuality.Techtarget, 2007. [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/versioning.

[19] D. Orchard, "Extending and versioning languages: XML languages [Editorial Draft]," World Wide Web Consortium (W3C), July 2007. [Online]. Available: http://www.w3.org/2001/tag/doc/versioning-xm.

[20] K. Bennett and V. T. Rajlich, "Software maintenance and evolution: a Roadmap.," in *Proceedings of the Conference on The Future of Software Engineering*, Limerick, Ireland, 2000.

[21] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," March 2001. [Online]. Available: http://www.w3.org/TR/wsdl.

[22] M. Endrei, M. Gaon, J. Graham, K. Hogg and N. Mulholland, "Moving forward with web services backward compatibility," May 2006. [Online]. Available: http://www.ibm.com/developerworks/java/library/ws-soa-backcomp/index.html?ca=drs-.

[23] K. Jerijærvi and J. Dubray, "Contract versioning, compatibility and composability," December 2008. [Online]. Available: http://www.infoq.com/articles/contract-versioning-comp2.

[24] C. Peltz and A. Anagol-Subbarao, "Design strategies for web services versioning," 2004. [Online]. Available: http://soa.sys-con.com/node/44356.

[25] D. Parachuri and S. Mallick, "Service versioning in SOA, December," 2008. [Online]. Available: http://www.infosys.com/offerings/IT-services/soa-services/white-papers/pages/index.aspx.

[26] P. Leitner, A. Michlmayr, F. Rosenberg and S. Dustdar, "End-to-end versioning support for web services," *In Services Computing 2008. SCC'08. IEEE International Conference,* vol. 1, pp. 59-66, 2008.

[27] M. Yamashita, B. Vollino, K. Becker and R. Galante, "Measuring Change Impact based on Usage Profiles," in *IEEE 19th International Conference on Web Services*, 2012.

[28] K. Brown and M. Ellis, "Best practices for web services versioning," Jan 2004. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-version/.

[29] D. Frank, L. Lam, L. Fong, R. Fang and M. Khangaonkar, "Using an interface proxy to host versioned web services.," in *Services Computing, 2008. SCC'08. IEEE International Conference on, vol. 2. IEEE, 2008, pp. 325-332*.

[30] K. Becker, A. Lopes, D. Milojicic, J. Pruyne and S. Singhal, "Automatically determining compatibility of evolving services," in *Web Services, 2008. ICWS'08. IEEE International Conference on. IEEE, 2008, pp. 161–168.*.

[31] J. Kenyon, "Web Service Versioning and Deprecation," January 2003. [Online]. Available:

http://soa.sys-con.com/node/39678.

[32] R. Fang, L. Lam, L. Fong, D. Frank, C. Vignola, Y. Chen and N. Du, "A version-aware approach for web service directory," in *ICWS 2007, July, 2007, pp. 406-413.*.

[33] A. Narayan and I. Singh, "Designing and versioning compatible web services," March 2007. [Online]. Available: http://www.ibm.com/developerworks/websphere/library/techarticles/0705 narayan/0705 narayan.html.

[34] V. Andrikopoulos, A Theory and Model for the Evolution of Software Services, Ph.D. Dissertation, Tilburg University Press, 2010.

[35] R. S. Sandhu, E. Coyne, H. Feinstein and C. E. Youman, Role-based Access Control Models. Computer, 29:38–47, February 1996.

[36] "Apache ServiceMix," The Apache Software Foundation., [Online]. Available: http://servicemix.apache.org.

[37] D. Muhler, "JBI Multi-tenancy Multi-container Support (JBIMulti2) – Requirements Specification," 2012.

[38] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Professional, 2003.

[39] "PostgreSQL," [Online]. Available: http://www.postgresql.org/.

[40] "Java Platform, Enterprise Edition 5 (Java EE), Final Release, JSR-244," 2006. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr244/.

[41] "Enterprise JavaBeans (EJB) 3.0, Final Release, 2006. JSR-220," [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr220/.

[42] "Java Message Service (JMS) 1.1, Final Release, 2002. JSR-914," [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr914/.

[43] "The Java API for XML-Based Web Services (JAX-WS) 2.0, Final Release, 2006, JSR-224," [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr224/.

[44] "The Java Architecture for XML Binding (JAXB) 2.0, Final Release, 2006. JSR-222," [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr222/.

[45] "Apache Camel User Guide 2.7.0," The Apache Software Foundation, 2011. [Online]. Available: http://camel.apache.org/manual/camelmanual2.7.0.pdf.

[46] "Apache ActiveMQ," The Apache Software Foundation, [Online]. Available: http://activemq.apache.org.

[47] "OW2 Consortium. JOnAS: Java Open Application Server," [Online]. Available: http://wiki.jonas.ow2.org.

[48] "SmartBear Software. soapUI," [Online]. Available: http://www.soapui.org.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Stuttgart, January 15, 2013 ----------------------------

(Sumadi Lie)